

Beginning XML, 2nd Edition: XML Schemas, SOAP,  
XSLT, DOM, and SAX 2.0  
by David Hunter, Kurt Cagle, Chris Dix et al.  
ISBN:0764543946  
Wrox Press © 2003 (784 pages)

This book teaches you all you need to know about XML--what it is, how it works, what technologies surround it, and how it can best be used in a variety of situations, from simple data transfer to using XML in your web pages.

## Table of Contents

### [Beginning XML, 2nd Edition—XML Schemas, SOAP,XSLT,DOM, and SAX 2.0](#)

#### Introduction

Ch

apt - What is XML?  
er

1

Ch

apt - Well-Formed XML  
er

2

Ch

apt - XML Namespaces  
er

3

Ch

apt - XSLT  
er

4

Ch

apt - Document Type Definitions  
er

5

Ch

apt - XML Schemas  
er

6

Ch

apt - Advanced XML Schemas  
er

7

Ch

- apt - The Document Object Model (DOM)

er

8

Ch

- apt - The Simple API for XML (SAX)

er

9

Ch

- apt - SOAP

er

10

Ch

- apt - Displaying XML

er

11

Ch

- apt - XML and Databases

er

12

Ch

- apt - Linking and Querying XML

er

13

Ca

se

- Stu - Using XSLT to Build Interactive Web Applications

dy

1

Ca

se

- Stu - XML Web Services

dy

2

Ap

pe

- ndi - The XML Document Object Model

x

A

Ap

pe

- ndi - XPath Reference

x

B

Ap

pe

- ndi - XSLT Reference

x

C

[Ap](#)

[pe](#)

- [ndi](#) - Schema Element and Attribute Reference

[x](#)

[D](#)

[Ap](#)

[pe](#)

- [ndi](#) - Schema Datatypes Reference

[xE](#)

[Ap](#)

[pe](#)

- [ndi](#) - SAX 2.0: The Simple API for XML

[xF](#)

[Ap](#)

[pe](#)

- [ndi](#) - Useful Web Resources

[x](#)

[G](#)

[Index](#)

---

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

## Back Cover

Extensible Markup Language (XML) is a rapidly maturing technology with powerful real-world applications, particularly for the management, display, and transport of data. Together with its many related technologies, it has become the standard for data and document delivery on the Web.

This book teaches you all you need to know about XML—what it is, how it works, what technologies surround it, and how it can best be used in a variety of situations, from simple data transfer to using XML in your web pages. It builds on the strengths of the first edition, and provides new material to reflect the changes in the XML landscape—notably SOAP and Web Services, and the publication of the XML Schemas Recommendation by the W3C.

## Who is this book for?

*Beginning XML, 2nd Edition* is for any developer who is interested in learning to use XML in web, e-commerce, or data storage applications. Some knowledge of mark up, scripting, and/or object oriented programming languages is advantageous, but no essential, as the basis of these techniques is explained as required.

## What does this book cover?

- XML syntax and writing well-formed XML
- Using XML Namespaces
- Transforming XML into other formats with XSLT
- XPath and XPointer for locating specific XML data
- XML validation using DTDs and XML Schemas
- Manipulating XML documents with the DOM and SAX 2.0
- SOAP and Web Services
- Displaying XML using CSS and XSL
- Incorporating XML into traditional databases and n-tier architectures
- XLink for linking XML and non-XML resources.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Beginning XML, 2nd Edition-XML Schemas, SOAP,XSLT,DOM, and SAX 2.0

**Kurt Cagle**

**Chris Dix**

**David Hunter**

**Roger Kovack**

**Jon Pinnock**

**Jeff Rafter**

Published by

**Wiley Publishing, Inc.**

10475 Crosspoint Boulevard

Indianapolis, IN 46256

[www.wiley.com](http://www.wiley.com)

Copyright © 2003 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

Library of Congress Card Number: 2003107073

ISBN: 0-7645-4394-6

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

1B/RQ/QW/QT/IN

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8700. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4447, E-Mail: [permcoordinator@wiley.com](mailto:permcoordinator@wiley.com).

**LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY:** WHILE THE PUBLISHER AND AUTHOR HAVE USED THEIR BEST EFFORTS IN PREPARING THIS BOOK, THEY MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK AND SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES OR WRITTEN SALES MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR YOUR SITUATION. YOU SHOULD CONSULT WITH A PROFESSIONAL WHERE APPROPRIATE. NEITHER THE

PUBLISHER NOR AUTHOR SHALLBE LIABLE FOR ANYLOSS OF PROFIT OR ANYOTHER COMMERCIALDAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (800) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

**Trademarks:** Wiley, the Wiley Publishing logo, Wrox, the Wrox logo, the Wrox Programmer to Programmer logo and related trade dress are trademarks or registered trademarks of Wiley in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

### **Trademark Acknowledgements**

Wrox has endeavored to provide trademark information about all the companies and products mentioned in this book by the appropriate use of capitals. However, Wrox cannot guarantee the accuracy of this information.

### **Credits**

#### **Authors**

Kurt Cagle  
Chris Dix  
David Hunter  
Roger Kovack  
Jonathon Pinnock  
Jeff Rafter

#### **Technical Reviewers**

Steve Baker  
David Beauchemin  
Martin Beaulieu  
Natalia Bortniker  
Oli Gauti Gudmundsson  
Paul Houle  
Graham Innocent  
Sachin Kanna  
Sing Li  
Steven Livingstone  
Nikola Ozu  
Jeff Rafter  
Gareth Reakes  
Eddie Robertsson  
David Schultz  
Ian Stokes-Rees

#### **Category Managers**

Simon Cox  
Dave Galloway

#### **Technical Architect**

Peter Morgan

**Technical Editors**

Sarah Larder

Simon Mackie

**Indexers'**

Michael Brinkman

Fiona Murray

**Production Manager**

Liz Toy

**Production Project Co-ordinator**

Mark Burdett

**Author Agent**

Marsha Collins

**Production Assistant**

Abbie Forletta

**Project Manager**

Vicky Idiens

**Cover**

Dawn Chellingworth

**Proof Reader**

Keith Westmoreland

**About the Authors**

**Kurt Cagle**

Kurt Cagle is a writer and developer specializing in XML and Internet related issues. He has written eight books and more than one hundred articles on topics ranging from Visual Basic programming to the impact of the Internet on society, and has consulted for such companies as Microsoft, Nordstrom, AT&T and others. He also helped launch Fawcette's XML Magazine and has been the DevX DHTML and XML Pro for nearly two years.

**Kurt Cagle contributed [Chapter 11](#) to this book.**

**Chris Dix**

Chris Dix has been developing software for fun since he was 10 years old, and for a living for the past 8 years. He is one of the authors of Professional XML Web Services, and he frequently writes and speaks on the topic of XML and Web Services. Chris is Lead Developer for NavTraK, Inc., a leader in automatic vehicle location systems located in Salisbury, Maryland, where he develops Web Services and system architecture. He can be reached at [cdux@navtrak.net](mailto:cdux@navtrak.net).

*I would like to thank my wife Jennifer and my wonderful sons Alexander and Calvin for their love and support. I would also like to thank the people at Wrox for this opportunity, and for their technical expertise in helping make this possible.*

**Chris Dix contributed Case Study 2 to this book.**

## David Hunter

David Hunter is a Senior Architect for MobileQ, a leading mobile software solutions developer, and the first company to ship an XML-based mobility server. David has extensive experience building scalable applications, and provides training on XML. He also works closely with the team that develops MobileQ's flagship product, XMLEDge, which delivers the ideal mobile user experience on a diverse number of mobile devices.

*First of all, I would like to thank God for the incredible opportunities he has given me to do something I love, and even write books about it. I pray that the glory will go to him. I would also like to thank Wrox's editors; if this book is helpful, easy to read, and easy to understand, it's because the editors made it that way.*

*And finally, I'd like to thank the person who gave me the most support, but probably doesn't even realize it. Thank you, Andrea, for helping me through this."*

**David Hunter contributed [Chapters 1,2,3,4,8,10, 12, and 13](#) to this book.**

## Roger Kovack

Roger Kovack has more than 25 years of software development experience, started by programming medical research applications in Fortran on DEC machines at the University of California. More recently he has consulted to Wells Fargo and Bank of America, developing departmental information systems on desktop and client/server platforms. Bitten by Java and the web bug in the mid '90s he developed web applications for Commerce One, a major B2B software vendor; and for LookSmart.com, one of the best known and still operating web portals. He was instrumental in bringing Java into those organizations to replace ASP and C++. Roger can be contacted on <http://www.xslroot.com>.

*"My deep thanks to my wife, Julie, for the encouragement and support for writing this chapter. I'm also endlessly grateful for the help and attention the editorial team at Wrox Press provided. Their concern for quality content can't be overstated.*

*Words can't express my sorrow and compassion for the innocent victims and their families whose lives were shattered by the terrorist attacks on New York and Washington DC on September 11, 2001. The personal, permanent wound that has caused me plead for world peace."*

**Roger Kovack contributed Case Study 1 to this book.**

## Jon Pinnock

Jonathan Pinnock started programming in Pal III assembler on his school's PDP 8/e, with a massive 4K of memory, back in the days before Moore's Law reached the statute books. These days he spends most of his time developing and extending the increasingly successful PlatformOne product set that his company, JPA, markets to the financial services community. JPA's home page is at: [www.jpassoc.co.uk](http://www.jpassoc.co.uk)

*"My heartfelt thanks go to Gail, who first suggested getting into writing, and now suffers the consequences on a fairly regular basis, and to Mark and Rachel, who just suffer the consequences."*

**Jon Pinnock contributed [Chapter 9](#) to this book.**

## Jeff Rafter

Jeff Rafter currently resides in Iowa City, where he is studying Creative Writing at the University of Iowa. For the past two years, he has worked with Standfacts Credit Services, a Los Angeles based company, developing XML interfaces for use in the mortgage industry. He also leads the XML development for Defined Systems, a web hosting company founded with his long time friend Dan English. In his free time, Jeff composes sonnets, plays chess in parks, skateboards, and reminisces about the Commodore64 video game industry of the late 1980s.

*"I thank God for his love and grace in all things. I would also like to thank my beautiful wife Ali, who is the embodiment of that love in countless ways. She graciously encouraged me to pursue my dreams at any cost. Thanks also to Mike McKay who was first a servant and then a friend as I worked through the writing process.*

*Finally, I would like to thank Vicky, Peter, Sarah, Simon, Victoria, Marsha and everyone at Wrox for the opportunity and support. I would also like to express my gratitude to the invaluable reviewers."*

**Jeff Rafter contributed [Chapters 5, 6, and 7](#) to this book**

---

[!\[\]\(71ceb62b681518c82e95d615e7265d66\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(9c4f697052545ae4fab36076e03db94f\_img.jpg\) NEXT !\[\]\(9d674a9457e5768b1d3049faa21b2696\_img.jpg\)](#)

# Introduction

Welcome to Beginning XML, 2nd Edition, the book I wish I'd had when I was first learning the language!

When we wrote the 1st Edition of this book, XML was a relatively new language, but was already gaining ground fast, and becoming more and more widely used in a vast range of applications. By the time we started the 2nd Edition, XML had already proven itself to be more than a passing fad, and was in fact being used throughout the industry for an incredibly wide range of uses. There are also quite a number of specifications surrounding XML, which either use XML or provide functionality in addition to the XML core specification, which aim to allow developers to do some pretty powerful things.

So what is XML? It's a markup language, used to describe the structure of data in meaningful ways. Anywhere that data is input/output, stored, or transmitted from one place to another, is a potential fit for XML's capabilities. Perhaps the most well known applications are web related (especially with the latest developments in handheld web access?for which some of the technology is XML-based). But there are many other non-web based applications where XML is useful?for example as a replacement for (or to complement) traditional databases, or for the transfer of financial information between businesses.

This book aims to teach you all you need to know about XML?what it is, how it works, what technologies surround it, and how it can best be used in a variety of situations, from simple data transfer to using XML in your web pages. It will answer the fundamental questions:

- What is XML?
- 
- How do I use XML?
- 
- How does it work?
- 
- What can I use it *for*, anyway?

## Who is this Book For?

This book is for people who know that it would be a pretty good idea to learn the language, but aren't 100% sure why. You've heard the hype, but haven't seen enough substance to figure out what XML is, and what it can do. You may already be somehow involved in web development, and probably even know the basics of HTML, although neither of these qualifications is absolutely necessary for this book.

What you don't need is knowledge of SGML (XML's predecessor), or even markup languages in general. This book assumes that you're new to the concept of markup languages, and we have tried to structure it in a way that will make sense to the beginner, and yet quickly bring you to XML expert status.

The word "Beginning" in the title refers to the style of the book, rather than the reader's experience level. There are two types of beginner for whom this book will be ideal:

-

Programmers who are already familiar with some web programming or data exchange techniques. You will already be used to some of the concepts discussed here, but will learn how you can incorporate XML technologies to enhance those solutions you currently develop.

Those working in a programming environment but with no substantial knowledge or experience of web development or data exchange applications. As well as learning how XML technologies can be applied to such applications, you will be introduced to some new concepts to help you understand how such systems work.

[!\[\]\(0cc5c4c18dd72a91e21b90220aef9c5d\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(9ea682cef02bbbdc0191f78cdae1d433\_img.jpg\) NEXT >](#)

# What's Covered in this Book?

I've tried to arrange the subjects covered in this book to take you from no knowledge to expert, in as logical a manner as I could. We'll be using the following format:

- First, we'll be looking at what exactly XML is, and why the industry felt that a language like this was needed.
- After covering the *why*, the next logical step is the *how*, so we'll be seeing how to create well-formed XML.
- Once we understand the whys and hows of XML, we'll unleash the programmer within, and look at an XML-based programming language that we can use to transform XML documents from one format to another.
- Now that you're comfortable with XML, and have seen it in action, we'll go on to some more advanced things you can do when creating your XML documents, to make them not only well-formed, but valid. (And we'll talk about what "valid" really means).
- XML wouldn't really be useful unless we could write programs to read the data in XML documents, and create new XML documents, so we'll get back to programming, and look at a couple of ways that we can do that. We'll also take a look at a technology that allows us to send messages across the Internet, which uses XML.
- Since we have all of this data in XML format, it would be great if we could easily display it to people, and it turns out we can. We'll look at a technology you may already have been using in conjunction with HTML documents that also works great with XML. We'll also look at how this data fits in with traditional databases, and even how we can link XML documents to one another.
- And finally, we'll finish off with some case studies, which should help to give you ideas on how XML can be used in real life situation, and which could be used in your own applications.

This book builds on the strengths of the first edition, and provides new material to reflect the changes in the XML landscape-notably SOAP and Web Services, and the publication of the XML Schemas Recommendation by the W3C-since we first published it in June 2000.

The chapters are broken down as follows:

## Chapter 1: What is XML?

Here we'll cover some basic concepts, introducing the fact that XML is a markup language (a bit like HTML) where you can define your own elements, tags and attributes (known as a vocabulary). We'll see that tags have no presentation meaning-they're just a way of describing the structure of data.

## Chapter 2: Well-Formed XML

As well as explaining what well-formed XML is, we'll take a look at the rules that exist (the XML 1.0 Recommendation) for naming and structuring elements-you need to comply with these rules if your XML is to be well-formed.

## Chapter 3: Namespaces

Because tags can be made up, we need to avoid name conflicts when sharing documents. Namespaces provide a way to uniquely identify a group of tags, using a URI. This chapter explains how to use namespaces.

## Chapter 4: XSLT

XML can be transformed into other XML, HTML, and other formats using XSLT stylesheets, which are introduced here. The XPath language, used to locate sections in the XML document, is also covered. This chapter has been completely overhauled from the first edition.

## Chapter 5: Document Type Definitions

We can specify how an XML document should be structured, and even give default values, using Document Type Definitions (DTDs). If XML conforms to the associated DTD it is known as valid XML. This chapter covers the basics of using DTDs. Though we have shifted our emphasis on validation technologies in this edition (by giving more coverage to XML Schemas), we still recognise the importance of DTDs in XML programming, and have provided a completely new and refocused chapter on the subject.

## Chapter 6: XML Schemas

XML Schemas recently became a Recommendation by the W3C. They are a more powerful alternative to DTDs and are explained here. This chapter and the *Advanced Schemas* chapter that follows are both new to this edition.

## Chapter 7: Advanced Schemas

Some more advanced concepts of using XML Schemas are covered in this chapter.

## Chapter 8: The Document Object Model (DOM)

Programmers can use a variety of programming languages to manipulate XML, using the Document Object Model's objects, interfaces, methods, and properties, which are described here.

## Chapter 9: The Simple API for XML (SAX)

An alternative to the DOM for programmatically manipulating XML data is to use the Simple API for XML (SAX) as an interface. This chapter shows how to use SAX, and has been updated from the first edition to focus on SAX 2.0.

## Chapter 10: SOAP

The Simple Object Access Protocol (SOAP) is a specification for allowing cross-computer communications, and is fundamental to XML Web Services. We can package up XML documents, and send them across the Internet to be processed. This chapter explains SOAP and XML Web Services, and is new to this edition.

## Chapter 11: Displaying XML

Web site designers have long been using Cascading Style Sheets (CSS) with their HTML to easily make changes to a web site's presentation without having to touch the underlying HTML documents. This power is also available for XML, allowing you to display XML documents right in the browser. Or, if you need a bit more flexibility with your presentation, you can use XSLT to transform your XML to HTML or XHTML.

## Chapter 12: XML and Databases

XML is perfect for structuring data, and some traditional databases are beginning to offer support for XML. These are discussed, as well as a more general overview of how XML can be used in an n-tier architecture.

## Chapter 13: Linking XML

We can locate specific parts of the XML document using XPath and XPointer. We can also link sections of documents and other resources using XLink. Both XPointer and XLink are described in this chapter.

## **Case Studies 1 and 2**

Throughout the book you'll gain an understanding of how XML is used in web, business to business (B2B), data storage, and many other applications. These case studies cover some example applications and show how the theory can be put into practice in real life situations. Both are new to this edition.

## **Appendices**

These provide reference material that you may find useful as you begin to apply the knowledge gained throughout the book in your own applications.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# What You Need to Use this Book

Because XML is a text-based technology, all you really need to create XML documents is Notepad, or your equivalent text editor. However, to really see some of these samples in action, you might want to have Internet Explorer 5 or later, since this browser can natively read XML documents, and even provide error messages if something is wrong. For readers without IE, there will be some screenshots throughout the book, so that you can see what things would look like.

If you do have IE, you also have an implementation of the DOM, which you may find useful in the chapter on that subject.

Some of the examples, and the case studies, require access to a web server, such as Microsoft's IIS (or PWS).

Throughout the book, other (freely available) XML tools will be used, and we'll give instructions for obtaining these at the appropriate place.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Conventions

To help you understand what's going on, and in order to maintain consistency, we've used a number of conventions throughout the book:

When we introduce new terms, we **highlight** them.

Important

These boxes hold important information.

*Advice, hints, and background information comes in an indented, italicized font like this.*

## Try It Out

After learning something new, we'll have a Try It Out section, which will demonstrate the concepts learned, and get you working with the technology.

## How It Works

After a Try It Out section, there will sometimes be a further explanation, to help you relate what you've done to what you've just learned.

Words that appear on the screen in menus like the File or Window menu are in a similar font to what you see on screen. URLs are also displayed in this font.

On the occasions when we'll be running the examples from the command line, we'll show the command and the results like this:

```
>msxsl blah.xml blah.xsl
<root>
  <results/>
</root>
```

Keys that you press on the keyboard, like **Ctrl** and **Enter**, are in italics.

We use two font styles for code. If it's a word that we're talking about in the text, for example, when discussing `functionNames()`, `<Elements>`, and `Attributes`, it will be in a fixed pitch font. If it's a block of code that you can type in and run, or part of such a block, then it's also in a gray box:

```
<html>
  <head>
    <title>Simple Example</title>
  </head>
  <body>
    <p>Very simple HTML.</p>
  </body>
</html>
```

Sometimes you'll see code in a mixture of styles, like this:

```
<html>
  <head>
    <title>Simple Example</title>
  </head>
```

```
<body>
  <p>Very simple HTML.</p>
</body>
</html>
```

In this case, we want you to consider the code with the gray background, for example to modify it. The code with a white background is code we've already looked at, and that we don't wish to examine further.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Customer Support

We always value hearing from our readers, and we want to know what you think about this book: what you liked, what you didn't like, and what you think we can do better next time. You can send us your comments, either by returning the reply card in the back of the book, or by e-mail to [feedback@wrox.com](mailto:feedback@wrox.com). Please be sure to mention the book title in your message.

## How to Download the Sample Code for the Book

When you visit the Wrox site, <http://www.wrox.com/>, simply locate the title through our Search facility or by using one of the title lists. Click on Download in the Code column, or on Download Code on the book's detail page.

The files that are available for download from our site have been archived using WinZip. When you have saved the file to a folder on your hard-drive, you need to extract the files using a de-compression program such as WinZip or PKUnzip. When you extract the files, the code is usually extracted into chapter folders. When you start the extraction process, ensure your software is set to use folder names.

## Errata

We've made every effort to make sure that there are no errors in the text or in the code in this book. However, no one is perfect and mistakes do occur. If you find an error in one of our books, like a spelling mistake or a faulty piece of code, we would be very grateful for feedback. By sending in errata you may save another reader hours of frustration, and of course, you will be helping us provide even higher quality information. Simply e-mail the information to [support@wrox.com](mailto:support@wrox.com), your information will be checked and if correct, posted to the errata page for that title and used in subsequent editions of the book.

To see if there are any errata for this book on the web site, go to <http://www.wrox.com/>, and simply locate the title through our Search facility or title list. Click on the Book Errata link, which is below the cover graphic on the book's detail page.

## E-mail Support

If you wish to directly query a problem in the book with an expert who knows the book in detail then e mail [support@wrox.com](mailto:support@wrox.com) with the title of the book and the last four numbers of the ISBN in the subject field of the e-mail. A typical e-mail should include the following things:

- The **title of the book, last four digits of the ISBN, and page number** of the problem in the Subject field.
- Your **name, contact information**, and the **problem** in the body of the message.

We **won't** send you junk mail. We need the details to save your time and ours. When you send an e-mail message, it will go through the following chain of support:

- Customer Support?Your message is delivered to our customer support staff, who are the first people to read it. They have files on most frequently asked questions and will answer anything general about the book or the web site immediately.

- Editorial?Deeper queries are forwarded to the technical editor responsible for that book. They have experience with the programming language or particular product, and are able to answer detailed technical questions on the subject.
- The Authors?Finally, in the unlikely event that the editor cannot answer your problem, he or she will forward the request to the author. We do try to protect the author from any distractions to their writing; however, we are quite happy to forward specific requests to them. All Wrox authors help with the support on their books. They will e-mail the customer and the editor with their response, and again all readers should benefit.

The Wrox Support process can only offer support to issues that are directly pertinent to the content of our published title. Support for questions that fall outside the scope of normal book support is provided via the community lists of our <http://p2p.wrox.com/> forum.

## p2p.wrox.com

For author and peer discussion join the P2P mailing lists. Our unique system provides **programmer to programmer?** contact on mailing lists, forums, and newsgroups, all in addition to our one-to-one e-mail support system. If you post a query to P2P, you can be confident that it is being examined by the many Wrox authors and other industry experts who are present on our mailing lists. At p2p.wrox.com you will find a number of different lists that will help you, not only while you read this book, but also as you develop your own applications. Particularly appropriate to this book are the XML and XSLT lists.

To subscribe to a mailing list just follow these steps:

1. Go to <http://p2p.wrox.com/>.
2. Choose the appropriate category from the left menu bar (in this case, XML).
3. Click on the mailing list you wish to join.
4. Follow the instructions to subscribe and fill in your e-mail address and password.
5. Reply to the confirmation e-mail you receive.
6. Use the subscription manager to join more lists and set your e-mail preferences.

## Why this System Offers the Best Support

You can choose to join the mailing lists or you can receive them as a weekly digest. If you don't have the time, or facility, to receive the mailing list, then you can search our online archives. Junk and spam mails are deleted, and your own e-mail address is protected by the unique Lyris system. Queries about joining or leaving lists, and any other general queries about lists, should be sent to [listsupport@p2p.wrox.com](mailto:listsupport@p2p.wrox.com).

# Chapter 1: What is XML?

## Overview

**Extensible Markup Language (XML)** is a buzzword you will see everywhere on the Internet, but it's also a rapidly maturing technology with powerful real-world applications, particularly for the management, display and organization of data. Together with its many related technologies, which will be covered in later chapters, it is an essential technology for anyone using markup languages on the Web or internally. This chapter will introduce you to some of the basics of **XML**, and begin to show you why it is so important to learn about it.

We will cover:

- The two major categories of computer file types?binary files and text files?and the advantages and disadvantages of each
- The history behind XML, including other markup languages?**SGML and HTML**
- How XML documents are structured as **hierarchies** of information
- A brief introduction to some of the other technologies surrounding XML, which you will be working with throughout the book
- A quick look at some areas where XML is proving to be useful

*While there are some short examples of XML in this chapter, you aren't expected to understand what's going on just yet. The idea is simply to introduce the important concepts behind the language, so that throughout the book you can see not only how to use XML, but also why it works the way that it does.*

# Of Data, Files, and Text

XML is a technology concerned with the description and structuring of *data*, so before we can really delve into the concepts behind XML, we need to understand how data is stored and accessed by computers. For our purposes, there are two kinds of data files that are understood by computers: **text files** and **binary files**.

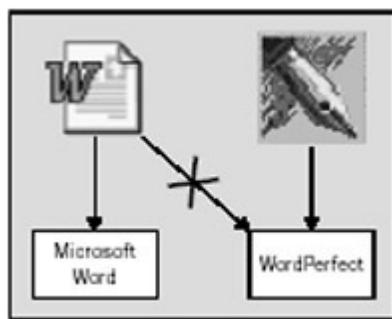
## Binary Files

A **binary file**, at its simplest, is just a stream of **bits** (1's and 0's). It's up to the application that created a binary file to understand what all of the bits mean. That's why binary files can only be read and produced by certain computer programs, which have been specifically written to understand them.

For example, when a document is created with a word processor, the program creates a binary file in its own proprietary format. The programmers who wrote the word processor decided to insert certain binary codes into the document to denote bold text, other codes to denote page breaks, and many other codes for all of the information that needs to go into these documents. When you open a document in the word processor it interprets those codes, and displays the properly formatted text on the screen, or prints it to the printer.

The codes inserted into the document are **meta data**, or *information about information*. Examples could be "this word should be in bold", "that sentence should be centered", etc. This meta data is really what differentiates one file type from another; the different types of files use different kinds of meta data.

For example, a word processing document will have different meta data from a spreadsheet document, since they are describing different things. Not so obviously, word processing documents from different word processing applications will also have different metadata because the applications were written differently:



As the above diagram shows, a document created with one word processor cannot be assumed to be readable in or used by another, because the companies who write word processors all have their own proprietary formats for their data files. So Word documents open in Microsoft Word, and WordPerfect documents open in WordPerfect.

Luckily for us most word processors come with translators, which can translate documents from other word processors into formats that can be understood natively. Of course, many of us have seen the garbage that sometimes occurs as a result of this translation; sometimes applications are not as good as we'd like them to be at converting the information.

The advantage of binary file formats is that it is easy for computers to understand these binary codes, meaning that they can be processed much faster, and they are very efficient for storing this meta data. There is also a disadvantage, as we've seen, in that binary files are "proprietary". You might not be able to open binary files created by one application in another application, or even in the same application running on another platform.

## Text Files

Like binary files, **text files** are also streams of bits. However, in a text file these bits are grouped together in standardized ways, so that they always form numbers. These numbers are then further mapped to characters. For example, a text file might contain the bits:

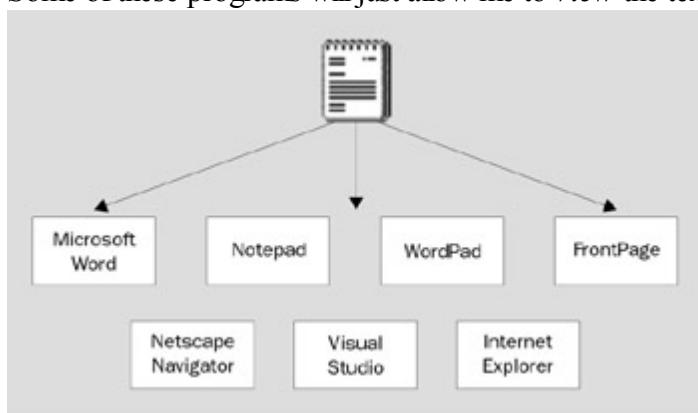
1100001

This group of bits could be translated as the number "97", which would then be further translated into the letter "a".

*This example makes a number of assumptions. A better description of how numbers are represented in text files is given in the section on "Encoding" in [Chapter 2](#).*

Because of these standards, text files can be read by many applications, and can even be read by humans, using a simple text editor. If I create a text document, anyone in the world can read it (as long as they understand English, of course), in any text editor they wish. There are still some issues, like the fact that different operating systems treat line ending characters differently, but it is much easier to share information with others than with binary formats.

The following diagram shows just some of the applications on my machine that are capable of opening text files. Some of these programs will just allow me to *view* the text, while others will let me *edit* it as well.



In its beginning, the Internet was almost completely text-based, which allowed people to communicate with relative ease. This contributed to the explosive rate at which the Internet was adopted, and to the ubiquity of applications like e-mail, the World Wide Web, newsgroups, etc.

The disadvantage of text files is that it's more difficult and bulky to add other information?our meta data in other words. For example, most word processors allow you to save documents in text form, but if you do then you can't mark a section of text as bold, or insert a binary picture file. You will simply get the words, with none of the formatting.

## A Brief History of Markup

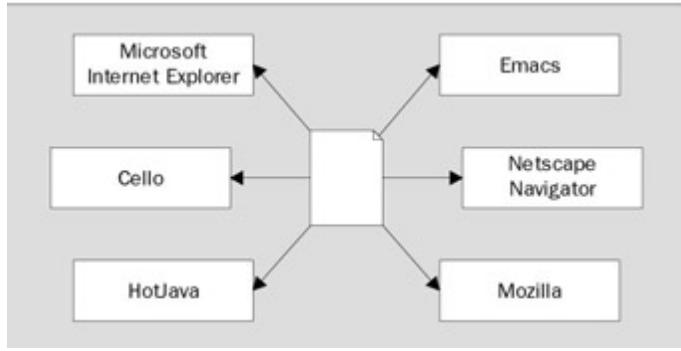
We can see that there are advantages to binary file formats (easy to understand by a computer, compact), as well as advantages to text files (universally interchangeable). Wouldn't it be ideal if there were a format that combined the universality of text files with the efficiency and rich information storage capabilities of binary files?

This idea of a universal data format is not new. In fact, for as long as computers have been around, programmers have been trying to find ways to exchange information between different computer programs. An early attempt to combine a universally interchangeable data format with rich information storage capabilities was **SGML (Standard Generalized Markup Language)**. This is a text-based language that can be used to **mark up** data?that is, add meta data?in a way which is **self-describing**. We'll see in a moment what self-describing means.

SGML was designed to be a standard way of marking up data for any purpose, and took off mostly in large document management systems. It turns out that when it comes to huge amounts of complex data there are a lot of

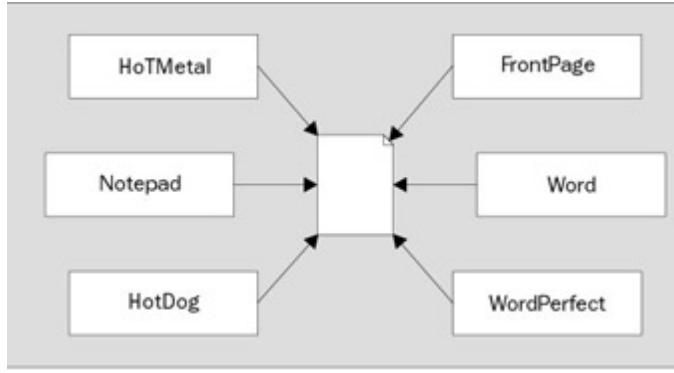
considerations to take into account and, as a result, SGML is a very complicated language. With that complexity comes power though.

A very well-known language, based on the SGML work, is the **HyperText Markup Language**, or **HTML**. HTML uses many of SGML's concepts to provide a universal markup language for the display of information, and the linking of different pieces of information. The idea was that any HTML document (or **web page**) would be presentable in any application that was capable of understanding HTML (termed a **web browser**).



Not only would that browser be able to display the document, but also if the page contained links (termed **hyperlinks**) to other documents, the browser would be able to seamlessly retrieve them as well.

Furthermore, because HTML is text-based, anyone can create an HTML page using a simple text editor, or any number of web page editors, some of which are shown below:



Even many word processors, such as WordPerfect and Word, allow you to save documents as HTML. Think about the ramifications of these two diagrams: any HTML editor, including a simple text editor, can create an HTML file, and that HTML file can then be viewed in any web browser on the Internet!

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# So What is XML?

Unfortunately, SGML is such a complicated language that it's not well suited for data interchange over the web. And, although HTML has been incredibly successful, it's also limited in its scope: it is only intended for displaying documents in a browser. The tags it makes available do not provide any information about the content they encompass, only instructions on how to display that content. This means that I could create an HTML document which displays information about a person, but that's about all I could do with the document. I couldn't write a program to figure out from that document which piece of information relates to the person's first name, for example, because HTML doesn't have any facilities to describe this kind of specialized information. In fact, that program wouldn't even know that the document was about a person at all.

Extensible Markup Language (XML) was created to address these issues.

*Note that it's spelled "Extensible", not "eXtensible". Mixing these up is a common mistake.*

XML is a **subset** of SGML, with the same goals (mark up of any type of data), but with as much of the complexity eliminated as possible. XML was designed to be fully compatible with SGML, which means that any document which follows XML's syntax rules is by definition also following SGML's syntax rules, and can therefore be read by existing SGML tools. It doesn't go both ways though, so an SGML document is not necessarily an XML document.

It is important to realize, however, that XML is not really a "language" at all, but a standard for creating languages that meet the XML criteria (we'll go into these rules for creating XML documents in [Chapter 2](#)). In other words, XML describes a syntax that you use to create your own languages. For example, suppose I have data about a name, and I want to be able to share that information with others and I also want to be able to use that information in a computer program. Instead of just creating a text file like this:

John Doe

or an HTML file like this:

```
<html>
<head><title>Name</title></head>
<body>
<p>John Doe</p>
</body>
</html>
```

I might create an XML file like this:

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

Even from this simple example you can see why markup languages like SGML and XML are called "self-describing". Looking at the data, you can easily tell that this is information about a `<name>`, and you can see that there is data called `<first>` and more data called `<last>`. I could have given the tags any names I liked, however, if you're going to use XML, you might as well use it right, and give things *meaningful* names.

You can also see that the XML version of this information is much larger than the plain-text version. Using XML to mark up data will add to its size, sometimes enormously, but achieving small file sizes isn't one of the goals of XML; it's only about making it easier to write software that accesses the information, by giving structure to data. However,

this larger file size should not deter you from using XML. The advantages of easier-to-write code far outweigh the disadvantages of larger bandwidth issues. Also, if bandwidth is a critical issue for your applications, you can always compress your XML documents before sending them across the network-compressing text files yields very good results.

## Try It Out-Opening an XML File in Internet Explorer

If you're running IE 5 or later, our XML from above can be viewed in your browser.

1.

Open up Notepad and type in the following XML:

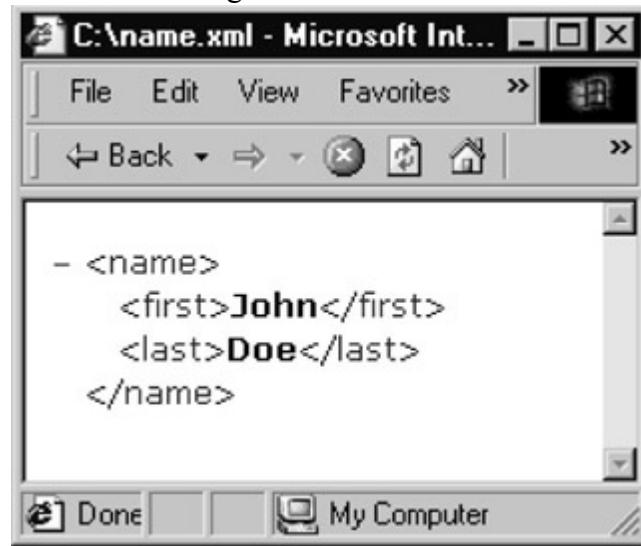
```
<name>
<first>John</first>
<last>Doe</last>
</name>
```

2.

Save the document to your hard drive as name.xml.

3.

You can then open it up in IE 5 (for example by double-clicking on the file in Windows Explorer), where it will look something like this:



## How It Works

Although our XML file has no information concerning display, IE 5 formats it nicely for us, with our information in bold, and our markup displayed in different colors. Also, <name> is collapsible, like your file folders in Windows Explorer; try clicking on the - sign next to <name> in the browser window. For large XML documents, where you only need to concentrate on a smaller subset of the data, this can be quite handy.

This is one reason why IE 5 can be so helpful when authoring XML: it has a default **stylesheet** built in, which applies this default formatting to any XML document.

*XML styling is accomplished through another document dedicated to the task, called a stylesheet. In a stylesheet the designer specifies rules that determine the presentation of the data. The same stylesheet can then be used with multiple documents to create a similar appearance among them. There are a variety of languages that can be used to create stylesheets. In Chapter 4 we'll learn about a transformation stylesheet language called Extensible Stylesheet Language Transformations (XSLT) and in Chapter 11 we'll be looking at*

a stylesheet language called Cascading Style Sheets (CSS).

As we'll see in later chapters, you can also create your own stylesheets for displaying XML documents. This way, the same data that your applications use can also be viewed in a browser. In effect, by combining XML data with stylesheets you can separate your data from your presentation. That makes it easier to use the data for multiple purposes (as opposed to HTML, which doesn't provide any separation of data from presentation-in HTML, *everything* is presentation).

## What Does XML Buy Us?

But why would we go to all of the bother of creating an XML document? Wouldn't it be easier to just make up some rules for a file about names, such as "The first name starts at the beginning of the file, and the last name comes after the first space"? That way, our application could still read the data, but the file size would be much smaller.

## How Else Would We Describe Our Data?

As a partial answer, let's suppose that we want to add a middle name to our example:

John Fitzgerald Doe

Okay, no problem. We'll just modify our rules to say that everything after the first space and up to the second space is the middle name, and the rest after the second space is the last name. Oh, unless there is no second space, in which case we'll have to assume that there is no middle name, and the first rule still applies. So we're still fine. Unless a person happens to have a name like:

John Fitzgerald Johansen Doe

Whoops. There are two middle names in there. The rules get more complex. While a human might be able to tell immediately that the two middle words compose the middle name, it is more difficult to program this logic into a computer program. We won't even discuss "John Fitzgerald Johansen Doe the 3rd"!

Unfortunately, when it comes to problems like this many software developers just throw in the towel and define more restrictive rules, instead of dealing with the complexities of the data. In this example, the software developers might decide that a person can only have *one* middle name, and that the application won't accept anything more than that.

*This is pretty realistic, I might add. My full name is David John Bartlett Hunter, but because of the way many computer systems are set up, a lot of the bills I receive are simply addressed to David John Hunter, or David J. Hunter. Maybe I can find some legal ground to stop paying my bills, but in the meantime my vanity takes a blow every time I open my mail.*

This example is probably not all that hard to solve, but it points out one of the major focuses behind XML. Programmers have been structuring their data in an infinite variety of ways, and with every new way of structuring data comes a new methodology for pulling out the information we need. With those new methodologies comes much experimentation and testing to get it just right. If the data changes, the methodologies also have to change, and the testing and tweaking has to begin again. With XML there is a standardized way to get the information we need, no matter how we structure it.

In addition, remember how trivial this example is. The more complex the data you have to work with, the more complex the logic you'll need to do that work. It is in these larger applications where you'll appreciate XML the most.

## XML Parsers

If we just follow the rules specified by XML, we can be sure that it will be easy to get at our information. This is

because there are programs written, called **parsers**, which are able to read XML syntax and get the information out for us. We can use these parsers within our own programs, meaning our applications never have to even look at the XML directly; a large part of the workload will have been done for us.

*There are also parsers available for parsing SGML documents, but they are much more complex than XML parsers. Since XML is a subset of SGML, it's much easier to write an XML parser than an SGML parser.*

In the past, before these parsers were around, a lot of work would have gone into the many rules we were looking at (like the rule that the middle name starts after the first space, etc.). But with our data in XML format, we can just give an XML parser a file like this:

```
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

and it can tell us that there is a piece of data called `<middle>`, and that the information stored there is Fitzgerald Johansen. The parser writer didn't have to know any rules about where the first name ends and where the middle name begins. It didn't have to know anything about my application or `<name>`s at all. The same parser could be used in my application, or in a completely different application. The language my XML is written in doesn't even matter to the parser; XML written in English, Chinese, Hebrew, or any other language could all be read by the same parser, even if the person who wrote it didn't understand any of these languages.

#### Important

Just like any HTML document can be displayed by any web browser, any XML document can be read by any XML parser, regardless of what application was used to create it, or even what platform it was created on. This goes a long way to making your data universally accessible.

There's also another added benefit here: if I had previously written a program to deal with the first XML format, which had only a first and last name, that application could also accept the new XML format, without me having to change the code. So, because the parser takes care of the work of getting data out of the document for us, we can add to our XML format without breaking existing code, and new applications can take advantage of the new information if they wish.

*Note that if we subtracted elements from our `<name>` example, or changed the names of elements, we would still have to modify our applications to deal with the changes.*

On the other hand, if we were just using our previous text-only format, any time we changed the data at all, every application using that data would have to be modified, retested, and redeployed.

Because it's so flexible, XML is targeted to be the basis for defining data exchange languages, especially for communication over the Internet. The language makes it very easy to work with data within applications, such as one that needs to access the `<name>` information above, but it also makes it easy to share information with others; we can pass our `<name>` information around the Internet, and even without our particular program the data can still be read. People can even pull the file up in a regular text editor and look at the raw XML, if they like, or open it in a viewer such as IE 5.

## Why "Extensible"?

Since we have full control over the creation of our XML document, we can shape the data in any way we wish, so that it makes sense to our particular application. If we don't need the flexibility of our `<name>` example, and decide to describe a name in XML like this:

```
<designation>John Fitzgerald Johansen Doe</designation>
```

we are free to do so. If we want to create data in a way that only one particular computer program will ever use, we can do so. And if we feel that we want to share our data with other programs, or even other companies across the Internet, XML gives us the flexibility to do that as well. We are free to structure the same data in different ways that suit the requirements of an application or category of applications.

### Important

This is where the *extensible* in Extensible Markup Language comes from: anyone is free to mark up data in any way using the language, even if others are doing it in completely different ways.

HTML, on the other hand, is not extensible, because you can't add to the language; you have to use the tags which are part of the HTML specification. Web browsers can understand

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

because the `<p>` tag is a pre-defined HTML tag, but can't understand

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

because the `<paragraph>` tag is not a pre-defined HTML tag.

Of course, the benefits of XML become even more apparent when people use the same format to do common things, because this allows us to interchange information much more easily. There have already been numerous projects to produce industry-standard **vocabularies** to describe various types of data. For example, **Scalable Vector Graphics (SVG)** is an XML vocabulary for describing two-dimensional graphics; **MathML** is an XML vocabulary for describing mathematics as a basis for machine to machine communication; **Chemical Markup Language (CML)** is an XML vocabulary for the management of chemical information. The list goes on and on. Of course, you could write your own XML vocabularies to describe this type of information if you so wished, but if you use other, more common, formats, there is a better chance that you will be able to produce software that is immediately compatible with other software.

Since XML is so easy to read and write in your programs, it is also easy to convert between different vocabularies when you need to. For example, if you want to represent mathematic equations in your particular application in a certain way, but MathML doesn't quite suit your needs, you can create your own vocabulary. If you wanted to export your data for use by other applications you might convert the data in your vocabulary to MathML, for the other applications to read. In fact, in [Chapter 4](#) we'll be covering a technology called **XSLT**, which was created for transforming XML documents from one format to another and that could potentially make these kinds of transformations very simple.

## HTML and XML: Apples and Red Delicious Apples

What HTML does for display, XML is designed to do for data exchange. Sometimes XML won't be up to a certain task, just like HTML is sometimes not up to the task of displaying certain information. How many of us have Adobe Acrobat readers installed on our machines for those documents on the Web that HTML just can't do properly? When it comes to display, HTML does a good job most of the time, and those who work with XML believe that, most of the time, XML will do a good job of communicating information. Just like HTML authors sometimes give up precise layout and presentation for the sake of making their information accessible to all web browsers, XML developers will give up the small file sizes of proprietary formats for the flexibility of universal data access.

There is, of course, a fundamental difference between HTML and XML:

- HTML is designed for a *specific* application; to convey information to humans (usually visually, through a web browser)
- XML has no specific application; it is designed for whatever use you need it for

This is an important concept. Because HTML has its specific application, it also has a finite set of specific markup constructs (<P>, <UL>, <H2>, etc.), which are used to create a correct HTML document. In theory, we can be confident that any web browser will understand an HTML document because all it has to do is understand this finite set of tags. In practice, of course, I'm sure you've come across web pages which displayed properly in one web browser and not another, but this is usually a result of non-standard HTML tags, which were created by browser vendors, instead of being part of the HTML specification itself.

On the other hand, if we create an XML document, we can be sure that any XML parser will be able to retrieve information from that document, even though we can't guarantee that any application will be able to understand *what that information means*. That is, just because a parser can tell us that there is a piece of data called <middle>, and that the information contained therein is "Fitzgerald Johansen", it doesn't mean that there is any software in the world that knows what a <middle> is, or what it is used for, or what it means.

So we can create XML documents to describe any information we want, but before XML can be considered useful, there must be applications written which understand it. Furthermore, in addition to the capabilities provided by the base XML specification, there are a number of related technologies, some of which we'll be covering in later chapters. These technologies provide more capabilities for us, making XML even more powerful than we've seen so far. Unfortunately, some of these technologies exist only in draft form, meaning that exactly how powerful these tools will be, or in what ways they'll be powerful, is yet to be seen.

## Hierarchies of Information

We'll discuss the syntactical constructs that make up XML in the [next chapter](#), but before we do, it might be useful to examine how data is structured in an XML document.

When it comes to large, or even moderate, amounts of information, it's usually better to group it into related sub-topics, rather than to have all of the information presented in one large blob. For example, this chapter is broken down into sub-topics, and further broken down into paragraphs; a tax form is broken down into sub-sections, across multiple pages. This makes the information easier to comprehend, as well as making it more accessible.

Software developers have been using this paradigm for years, using a structure called an **object model**. In an object model, all of the information that's being modeled is broken up into various objects, and the objects themselves are then grouped into a **hierarchy**. We'll be looking in more detail at object models in later chapters.

## Hierarchies in HTML

For example, when working with **Dynamic HTML (DHTML)** there is an object model available for working with HTML documents, called the **Document Object Model** or **DOM**. This allows us to write code in an HTML document, like the following JavaScript:

```
alert(document.title);
```

Here we are using the alert() function to pop up a message box telling us the title of an HTML document. That's done by accessing an object called document, which contains all of the information needed about the HTML document. The document object includes a property called title, which returns the title of the current HTML document.

The information that the object provides comes to us in the form of **properties**, and the functionality available comes to us in the form of **methods**. Again, this is a subject we'll come back to later on.

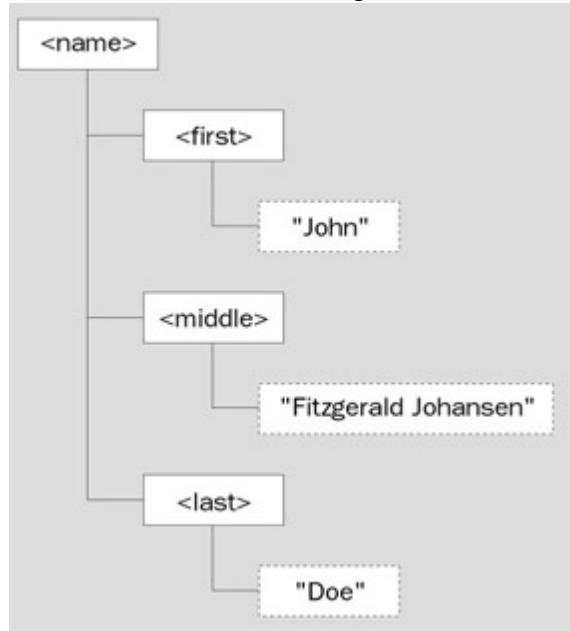
## Hierarchies in XML

XML also groups information in hierarchies. The items in our documents relate to each other in **parent/child** and **sibling/sibling** relationships.

### Important

These "items" are called [elements](#). We'll see a more precise definition of what exactly an element is in [Chapter 2](#). For now just think of them as the individual pieces of information in the data.

Consider our <name> example, shown hierarchically:



<name> is a parent of <first>. <first>, <middle>, and <last> are all siblings to each other (they are all children of <name>). Note also that the text is a child of the element. For example the text John is a child of <first>.

This structure is also called a **tree**; any parts of the tree that contain children are called **branches**, while parts that have no children are called **leaves**.

*These are fairly loose terms, rather than formal definitions, which simply make it easier to discuss the tree-like structure of XML documents. I have also seen the term "twig" in use, although it is much less common than "branch" or "leaf".*

Because the <name> element has only other elements for children, and not text, it is said to have **element content**. Conversely, since <first>, <middle>, and <last> have only text as children, they are said to have **simple content**.

Elements can contain both text and other elements. They are then said to have **mixed content**.

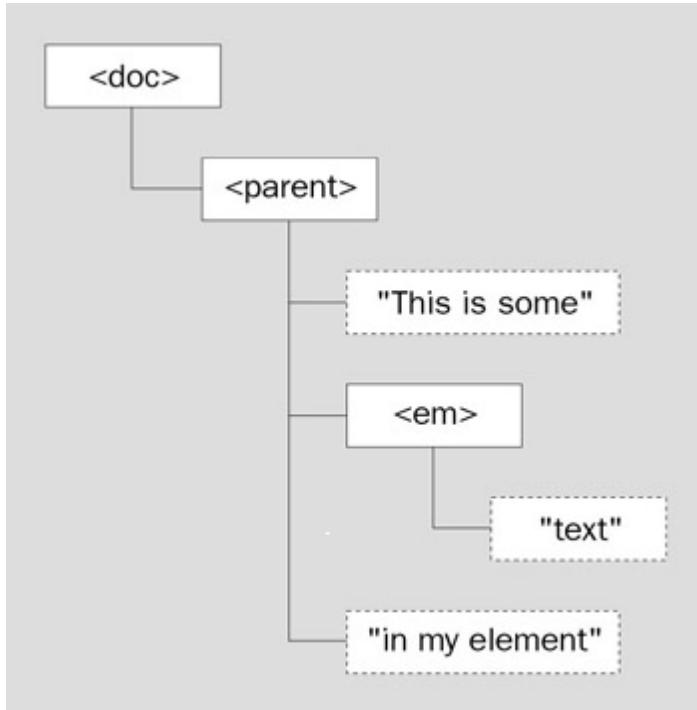
For example:

```
<doc>
  <parent>this is some <em>text</em> in my element</parent>
</doc>
```

Here, <parent> has three children:

- A text child containing the text "this is some"
- An <em> child
- Another text child containing the text "in my element"

It is structured like this:



Relationships can also be defined by making the family tree analogy work a little bit harder: <doc> is an **ancestor** of <em>; <em> is a **descendant** of <doc>.

Once you understand the hierarchical relationships between your items (and the text they contain), you'll have a better understanding of the nature of XML. You'll also be better prepared to work with some of the other technologies surrounding XML, which make extensive use of this paradigm.

In [Chapter 8](#) you'll get a chance to work with the Document Object Model that we mentioned earlier, which allows you to programmatically access the information in an XML document using this tree structure.

## What's a Document Type?

XML's beauty comes from its ability to create a document to describe any information we want. It's completely flexible as to how we structure our data, but eventually we're going to want to settle on a particular design for our information, and say "to adhere to our XML format, structure the data like this".

For example, when we created our <name> XML above, we created some **structured data**. We not only included all of the information about a name, but our hierarchy also contains implicit information about how some pieces of data relate to other pieces (our <name> contains a <first>, for example).

But it's more than that; we also created a specific set of elements, which is called a **vocabulary**. That is, we defined a number of XML elements which all work together to form a name: <name>, <first>, <middle>, and <last>.

But, it's even more than that! The most important thing we created was a **document type**. We created a specific type of document, which must be structured in a specific way, to describe a specific type of information. Although we haven't explicitly defined them yet, there are certain rules that the elements in our vocabulary must adhere to, in order for our <name> document to conform to our document type. For example:

- The top-most element must be the <name> element
- The <first>, <middle>, and <last> elements must be children of that element
- The <first>, <middle>, and <last> elements must be in that order
- There must be information in the <first> element and in the <last> element, but there doesn't have to be any information in the <middle> element

And so on.

In later chapters, you'll see that there are different syntaxes we can use to formally define an XML document type. Some XML parsers know how to read these syntaxes, and can use them to determine if your XML document really adheres to the rules in the document type or not.

However, all of the syntaxes used to define document types so far are lacking; they can provide some type checking, but not enough for many applications. Furthermore, they can't express the human meaning of terms in a vocabulary. For this reason, when creating XML document types, human-readable documentation should also be provided. For our <name> example, if we want others to be able to use the same format to describe names in their XML, we should provide them with documentation to describe how it works.

In real life, this human-readable documentation is often used in conjunction with one or more of the syntaxes available. Ironically, the self-describing nature of XML can sometimes make this human-readable documentation even more important! Often, because the data is already labeled within the document structure, it is assumed that people working with the data will be able to infer its meaning, which can be dangerous if the inferences made are incorrect, or even just different from the original author's intent.

## No, Really-What's a Document Type?

Well, okay, maybe I was a little bit hasty when labeling our <name> example a document type. The truth is that others who work with XML may call it something different.

One of the problems people encounter when they communicate is that they sometimes use different terms to describe the same thing or, even worse, use the same term to describe different things! For example, I might call the thing that I drive a car, whereas someone else might call it an auto, and someone else again might call it a G-Class Vehicle. Furthermore, when I say car I *usually* mean a vehicle that has four wheels, is made for transporting passengers, and is smaller than a truck. (Notice how fuzzy this definition is, and that it depends further on what the definition of a truck is.) When someone else uses the word car, or if I use the word car in certain circumstances, it may instead just mean a land-based motorized vehicle, as opposed to a boat or a plane.

The same thing happens in XML. It turns out that when you're using XML to create document types, you don't really have to think (or care) about the fact that you're creating document types; you just design your XML in a way that makes sense for your application, and then use it. If you ever did think about exactly what you were creating, you might have called it something other than a document type.

Important

The terms document type and vocabulary are ones we picked for this book because they do a good job of describing what we need to describe, but they are not universal terms used throughout the XML community. Regardless of the terms you use, the concepts are very important.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# What Is the World Wide Web Consortium?

One of the reasons that HTML and XML are so successful is that they're **standards**. That means that anyone can follow these standards, and the solutions they develop will be able to interoperate. So who creates these standards?

The **World Wide Web Consortium (W3C)** was started in 1994, according to their web site (<http://www.w3.org>), "to lead the World Wide Web to its full potential by developing common protocols that promote its evolution and ensure its interoperability". Recognizing this need for standards, the W3C produces **Recommendations** that describe the basic building blocks of the Web. They call them recommendations, instead of standards, because it is up to others to follow the recommendations to provide the interoperability.

Their most famous contribution to the Web is, of course, the HTML Recommendation; when a web browser producer claims that their product follows version 3.2 or 4.01 of the HTML Recommendation, they're talking about the Recommendation developed under the authority of the W3C.

The reason specifications from the W3C are so widely implemented is that the creation of these standards is a somewhat open process: any company or individual can join the W3C's membership, and membership allows these companies or individuals to take part in the standards process. This means that web browsers like Netscape Navigator and Microsoft Internet Explorer are more likely to implement the same version of the HTML Recommendation, because both Microsoft and Netscape were involved in the evolution of that Recommendation.

Because of the interoperability goals of XML, the W3C is a good place to develop standards around the technology. Most of the technologies covered in this book are based on standards from the W3C; the XML 1.0 Specification, the XSLT Specification, the XPath Specification, etc.

# What are the Pieces that Make Up XML?

"Structuring information" is a pretty broad topic, and as such it would be futile to try and define a specification to cover it fully. For this reason there are a number of inter-related specifications that all work together to form the XML family of technologies, with each specification covering different aspects of communicating information. Here are some of the more important ones:

- **XML 1.0** is the base specification upon which the XML family is built. It describes the syntax that XML documents have to follow, the rules that XML parsers have to follow, and anything else you need to know to read or write an XML document. It also defines DTDs, although they sometimes get treated as a separate technology. See the next bullet.
  - 
  - Because we can make up our own structures and element names for our documents, **DTDs and Schemas** provide ways to define our document types. We can check to make sure other documents adhere to these templates, and other developers can produce compatible documents. DTDs and Schemas are discussed in [Chapters 5, 6](#) and [7](#).
  - 
  - **Namespaces** provide a means to distinguish one XML vocabulary from another, which allows us to create richer documents by combining multiple vocabularies into one document type. We'll look at namespaces in detail in [Chapter 3](#).

[XPath](#) describes a querying language for addressing parts of an XML document. This allows applications to ask for a specific piece of an XML document, instead of having to always deal with one large "chunk" of information. For example, XPath could be used to get "all the last names" from a document. We'll discuss XPath in [Chapter 4](#).

- As we discussed earlier, in some cases we may want to display our XML documents. For simpler cases, we can use **Cascading Style Sheets (CSS)** to define the presentation of our documents. And, for more complex cases, we can use **Extensible Stylesheet Language (XSL)**, that consists of [XSLT](#), which can transform our documents from one type to another, and **Formatting Objects**, which deal with display. These technologies are covered in [Chapters 4](#) and [11](#).

- [XLink](#) and [XPointer](#) are languages that are used to link XML documents to each other, in a similar manner to HTML hyperlinks. They are described in [Chapter 13](#).

To provide a means for more traditional applications to interface with XML documents, there is a document object model—the **DOM**, which we'll discuss in [Chapter 8](#). An alternative way for programmers to interface with XML documents from their code is to use the Simple API for XML (**SAX**), which is the subject of [Chapter 9](#).

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Where Is XML Used, and Where Can it Be Used?

Well, that's quite a question. XML is platform and language independent, which means it doesn't matter that one computer may be using, for example, Visual Basic on a Microsoft operating system, and the other is a UNIX machine running Java code. Really, any time one computer program needs to communicate with another program, XML is a potential fit for the exchange format. The following are just a few examples, and we'll be discussing such applications in more detail throughout the book.

## Reducing Server Load

Web-based applications can use XML to reduce the load on the web servers. This can be done by keeping all information on the client for as long as possible, and then sending the information to those servers in one big XML document.

## Web Site Content

The W3C uses XML to write its specifications. These XML documents can then be transformed to HTML for display (by XSLT), or transformed to a number of other presentation formats.

Some web sites also use XML entirely for their content, where traditionally HTML would have been used. This XML can then be transformed to HTML via XSLT, or displayed directly in browsers via CSS. In fact, the web servers can even determine dynamically what kind of browser is retrieving the information, and then decide what to do. For example, transform the XML to HTML for older browsers, and just send the XML straight to the client for newer browsers, reducing the load on the server.

In fact, this could be generalized to *any* content, not just web site content. If your data is in XML, you can use it for any purpose, with presentation on the Web being just one possibility.

## Remote Procedure Calls

XML is also used as a protocol for **Remote Procedure Calls (RPC)**. RPC is a protocol that allows objects on one computer to call objects on another computer to do work, allowing distributed computing. As we'll see in [Chapter 10](#), using XML and HTTP for these RPC calls, using a technology called the **Simple Object Access Protocol (SOAP)**, allows this to occur even through a firewall, which would normally block such calls, providing greater opportunities for distributed computing.

## e-Commerce

**e-commerce** is one of those buzzwords that you hear all over the place. Companies are discovering that by communicating via the Internet, instead of by more traditional methods (such as faxing, human-to-human communication, etc.), they can streamline their processes, decreasing costs and increasing response times. Whenever one company needs to send data to another, XML is the perfect fit for the exchange format.

When the companies involved have some kind of on-going relationship this is known as business-to-business (**B2B**) e-commerce. There are also business to consumer (B2C) transactions?you may even have used this to buy this book if you bought it on the Internet. Both types have their potential uses for XML.

And there are many, many other applications where XML makes a good fit. Hopefully, after you've finished this book, you'll be able to intelligently decide when XML works, and when it doesn't.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

In this chapter, we've had an overview of what XML is and why it's so useful. We've seen the advantages of text and binary files, and the way that XML combines the advantages of both while eliminating most of the disadvantages. We have also seen the flexibility we have in creating data in any format we wish.

Because XML is a subset of a proven technology, SGML, there are many years of experience behind the standard. Also, because there are other technologies built around XML, we can create applications that are as complex or simple as our situation warrants.

Much of the power that we get from XML comes from the rigid way in which documents must be written. In the [next chapter](#), we'll take a closer look at the rules for creating well-formed XML.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 2: Well-Formed XML

## Overview

We've discussed some of the reasons why XML makes sense for communicating data, so now let's get our hands dirty and learn how to create our own XML documents. This chapter will cover all you need to know to create **well-formed XML**.

### Important

Well-formed XML is XML that meets certain syntactical rules outlined in the XML 1.0 specification.

You will learn:

- How to create XML **elements** using **start-** and **end-tags**
- How to further describe elements with **attributes**
- How to declare your document as being XML
- How to send instructions to applications that are processing the XML document
- Which characters aren't allowed in XML

Because XML and HTML appear so similar, and because you may already be familiar with HTML, we'll be making comparisons between the two languages in this chapter. However, if you don't have any knowledge of HTML, you shouldn't find it too hard to follow along.

If you have Internet Explorer 5 or later, you may find it useful to save some of the examples in this chapter on your hard drive, and view the results in the browser. If you don't have IE 5 or later, some of the examples will have screenshots to show what the end results look like. One nice result of doing this is that the browser will tell you if you make a syntax mistake. I do this quite often; just to sanity-check myself, and make sure I haven't mistyped anything.

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Tags and Text and Elements, Oh My!

It's time to stop calling things just "items" and "text"; we need some names for the pieces that make up an XML document. To get cracking, let's break down the simple name.xml document we created in [Chapter 1](#):

```
<name>
<first>John</first>
<middle>Fitzgerald Johansen</middle>
<last>Doe</last>
</name>
```

The text starting with a "<" character, and ending with a ">" character, is an XML tag. The information in our document (our data) is contained within the various tags **that constitute** the markup of the document. This makes it easy to distinguish the *information* in the document from the *markup*.

As you can see, the tags are paired together, so that any opening tag also has a closing tag. In XML parlance, these are called **start-tags** and **end-tags**. The end-tags are the same as the start-tags, except that they have a "/" right after the opening < character.

In this regard, XML tags work the same as start-tags and end-tags do in HTML. For example, you would mark a section of HTML bold like this:

```
<B>This is bold.</B>
```

As you can see, there is a <B> start-tag, and a </B> end-tag, just like we use for XML.

All of the information from the start of a start-tag to the end of an end-tag, and including everything in between, is called an **element**. So:

- <first> is a start-tag
- </first> is an end-tag
- <first>John</first> is an element

The text between the start-tag and end-tag of an element is called the **element content**. The content between our tags will often just be data (as opposed to other elements). In this case, the element content is referred to as **Parsed Character DATA**, which is almost always referred to using its acronym, **PCDATA**, or with a more general term such as "text content" or even "text node".

*Whenever you come across a strange-looking term like PCDATA, it's usually a good bet the term is inherited from SGML. Because XML is a subset of SGML, there are a lot of these inherited terms.*

The whole document, starting at <name> and ending at </name>, is also an element, which happens to include other elements. (And, in this case, since it contains the entire XML document, the element is called the **root element**, which we'll be talking about later.)

To put this newfound knowledge into action, let's create an example that contains more information than just a name.

## Try It Out?Describing Weirdness

We're going to build an XML document to describe one of the greatest CDs ever produced, *Dare to be Stupid*, by Weird Al Yankovic. But before we break out Notepad and start typing, we need to know what information we're capturing.

In [Chapter 1](#), we learned that XML is hierarchical in nature; information is structured like a tree, with parent/child relationships. This means that we'll have to arrange our CD information in a tree structure as well.

1.

Since this is a CD, we'll need to capture information like the artist, title, and date released, as well as the genre of music. We'll also need information about each song on the CD, such as the title and length. And, since Weird Al is famous for his parodies, we'll include information about what song (if any) this one is a parody of.

Here's the hierarchy we'll be creating:



Some of these elements, like `<date-released>`, will appear only once; others, like `<song>`, will appear multiple times in the document. Also, some will have PCDATA only, while some will include their information as child elements instead. For example, the `<artist>` element will contain PCDATA (no child elements) only: the name of the artist. On the other hand, the `<song>` element won't contain any PCDATA of its own, but will contain child elements that further break down the information.

2.

With this in mind, we're now ready to start entering XML. If you have Internet Explorer 5 or later installed on your machine, type the following into Notepad, and save it to your hard drive as `cd.xml`:

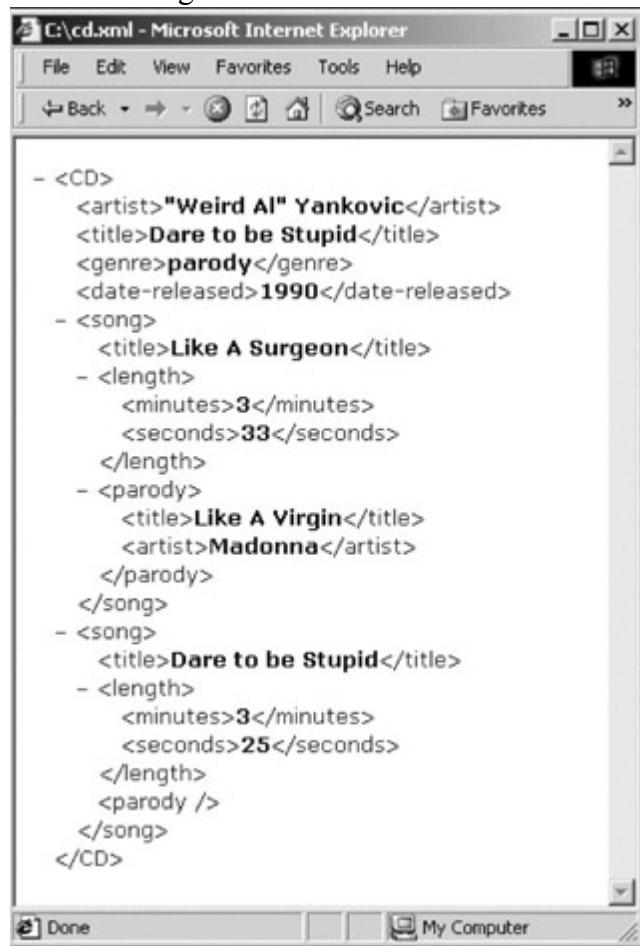
```
<CD>
<artist>"Weird Al" Yankovic</artist>
<title>Dare to be Stupid</title>
<genre>parody</genre>
<date-released>1990</date-released>
<song>
  <title>Like A Surgeon</title>
  <length>
    <minutes>3</minutes>
    <seconds>33</seconds>
  </length>
  <parody>
    <title>Like A Virgin</title>
    <artist>Madonna</artist>
  </parody>
</song>
<song>
  <title>Dare to be Stupid</title>
  <length>
    <minutes>3</minutes>
```

```
<seconds>25</seconds>
</length>
<parody></parody>
</song>
</CD>
```

*For the sake of brevity, we'll only enter two of the songs on the CD, but the idea is there nonetheless.*

3.

Now, open the file in IE 5 or later. (Navigate to the file in Explorer and double-click on it, or open up the browser and type the path in the URL bar.) If you have typed in the tags exactly as shown, the cd.xml file will look something like this:



## How It Works

Here we've created a hierarchy of information about a CD, so we've named the root element accordingly.

The <CD> element has children for the artist, title, genre, and date, as well as one child for each song on the disc. The <song> element has children for the title, length, and what song (if any) this is a parody of. Again, for the sake of this example, the <length> element was broken down still further, to have children for minutes and seconds, and the <parody> element broken down to have the title and artist of the parodied song.

You may have noticed that the browser changed <parody></parody> into <parody />. We'll talk about this shorthand syntax a little bit later, but don't worry: this is called a **self-closing tag** and it's perfectly legal.

If we were to write a CD Player application, we could make use of this information to create a play-list for our CD. It could read the information under our <song> element to get the name and length of each song to display to the user, display the genre of the CD in the title bar, etc.

## Rules for Elements

Obviously, if we could just create elements in any old way we wanted, we wouldn't be any further along than our text file examples from the [previous chapter](#). There must be some rules for elements, which are fundamental to the understanding of XML.

### Important

XML documents must adhere to these rules to be well-formed.

We'll list them, briefly, before getting down to details:

- Every start-tag must have a matching end-tag, or be a self-closing tag
- Tags can't overlap
- XML documents can have only one root element
- Element names must obey XML naming conventions
- XML is case-sensitive
- XML will keep whitespace in your text

It is these rules that make XML such a universal format for interchanging data. As long as your XML documents follow all of the rules in the XML Recommendation, any available XML parser will be able to read the information it contains.

### Every Start-tag Must Have an End-tag

One of the problems with parsing HTML documents is that not every element requires a start-tag and an end-tag. Take the following example:

```
<HTML>
<BODY>
<P>Here is some text in an HTML paragraph.
<BR>
Here is some more text in the same paragraph.
<P>And here is some text in another HTML paragraph.</p>
</BODY>
</HTML>
```

Notice that the first `<P>` tag has no closing `</P>` tag. This is allowed?and sometimes even encouraged?in HTML, because most web browsers can figure out where the end of the paragraph should be. In this case, when the browser comes across the second `<P>` tag, it knows to end the first paragraph. Then there's the `<BR>` tag (line break), which by definition has no closing tag.

Also, notice that the second `<P>` start-tag is matched by a `</p>` end-tag, in lowercase. This is not a problem for HTML browsers, because HTML is not case-sensitive, but as we'll see soon, this would cause a problem for an XML parser.

The problem is that this makes HTML parsers much harder to write. Code has to be included to take into account all of these factors, which often makes the parsers much larger, and much harder to debug. What's more, the way that files are parsed is not standardized?different browsers do it differently, leading to incompatibilities.

For now, just remember that in XML the end-tag is required, and its name has to exactly match the start-tag's name.

## Tags Cannot Overlap

Because XML is strictly hierarchical, you have to be careful to close child elements before you close their parents. (This is called **properly nesting** your tags.) Let's look at another HTML example to demonstrate this:

```
<P>Some <STRONG>formatted <EM>text</STRONG>, but</EM> no grammar no good!</P>
```

This would produce the following output on a web browser:

Some formatted text, but no grammar no good!

As you can see, the `<STRONG>` tags cover the text "formatted text", while the `<EM>` tags cover the text "text, but".

But is `<em>` a child of `<strong>`, or is `<strong>` a child of `<em>`? Or are they both *siblings*, and children of `<p>`? According to our stricter XML rules, the answer is none of the above. The HTML code, as written, can't be arranged as a proper hierarchy, and could therefore not be well-formed XML.

*Actually, the HTML above isn't really proper HTML either; according to the HTML 4 Recommendation, tags should not overlap like this, but web browsers will do their best to render the content anyway.*

If ever you're in doubt as to whether your XML tags are overlapping, try to rearrange them visually to be hierarchical. If the tree makes sense, then you're okay. Otherwise, you'll have to rework your markup.

For example, we could get the same effect as above by doing the following:

```
<P>Some <STRONG>formatted <EM>text</EM></STRONG><EM>, but</EM> no grammar no good!</P>
```

Which can be properly formatted in a tree, like this:

```
<P>
  Some
  <STRONG>
    formatted
    <EM>
      text
    </EM>
  </STRONG>
  <EM>
    ,
  </EM>
  no grammar no good!
</P>
```

## An XML Document Can Have Only One Root Element

In our `<name>` document, the `<name>` element is called the **root element**. This is the top-level element in the document, and all the other elements are its children or descendants. An XML document must have one and only one root element: in fact, it must have a root element even if it has no content.

For example, the following XML is not well-formed, because it has two root elements:

```
<name>John</name>
<name>Jane</name>
```

To make this well-formed, we'd need to add a top-level element, like this:

```
<names>
  <name>John</name>
  <name>Jane</name>
</names>
```

So while it may seem a bit of an inconvenience, it turns out that it's incredibly easy to follow this rule. If you have a document structure with multiple root-like elements, simply create a higher-level element to contain them.

## Element Names

If we're going to be creating elements we're going to have to give them names, and XML is very generous in the names we're allowed to use. For example, there aren't any reserved words to avoid in XML, as there are in most programming languages, so we have a lot flexibility in this regard.

However, there are some rules that we must follow:

- Names can start with letters (including non-Latin characters) or the "\_" character, but not numbers or other punctuation characters.
- After the first character, numbers are allowed, as are the characters "-" and ".".
- Names can't contain spaces.
- Names can't contain the ":" character. Strictly speaking, this character is allowed, but the XML specification says that it's "reserved". You should avoid using it in your documents, unless you are working with namespaces (which we'll be looking at in the [next chapter](#)).
- Names can't start with the letters "xml", in uppercase, lowercase, or mixed?you can't start a name with "xml", "XML", "XmL", or any other combination.
- Unfortunately, the XML parser shipped with Internet Explorer doesn't enforce this rule. However, even if you are using IE's XML parser, you should never name elements starting with the characters "xml", because your documents would not be considered well-formed by other parsers.*
- There can't be a space after the opening "<" character; the name of the element must come immediately after it. However, there can be space before the closing ">" character, if desired.

Here are some examples of valid names:

```
<first.name>
```

<rsumé>

And here are some examples of invalid names:

<xml-tag>

which starts with xml,

<123>

which starts with a number,

<fun=xml>

because the "=" sign is illegal, and:

<my tag>

which contains a space.

#### Important

Remember these rules for element names?they also apply to naming other things in XML.

## Case-Sensitivity

Another important point to keep in mind is that the tags in XML are **case-sensitive**. (This is a big difference from HTML, which is case-insensitive.) This means that <first> is different from <FIRST>, which is different from <First>.

*This sometimes seems odd to English-speaking users of XML, since English words can easily be converted to uppercase or lowercase with no loss of meaning. But in almost every other language in the world, the concept of case is either not applicable (in other words, what's the uppercase of ? Or the lowercase, for that matter?), or is extremely important (what's the uppercase of é? The answer may be different, depending on the context?sometimes it will be "É", but other times it will just be "é"). To put intelligent rules into the XML specification for converting between uppercase and lowercase (sometimes called **case-folding**) would probably have doubled or trebled its size, and still only benefited the English-speaking section of the population. Luckily, it doesn't take long to get used to having case-sensitive names.*

For this reason our previous <P></p> HTML example would not work in XML; since the tags are case-sensitive, an XML parser would not be able to match the </p> end-tag with any start-tags, and neither would it be able to match the <P> start-tag with any end-tags.

#### Important

Warning! Because XML is case-sensitive, you could legally create an XML document which has both <first> and <First> elements, which have different meanings. This is a bad idea, and will cause nothing but confusion! You should always try to give your elements distinct names, for your sanity, and for the sanity of those to come after you.

To help combat these kinds of problem, it's a good idea to pick a naming style and stick to it. Some examples of common styles are:

-

- <first\_name>
- 
- <firstName>
- 
- <first-name>
- 
- <FirstName>

Which style you choose isn't important; what is important is that you stick to it. A naming convention only helps when it's used consistently. For this book, I'll usually use the <FirstName> convention, because that's what I've grown used to.

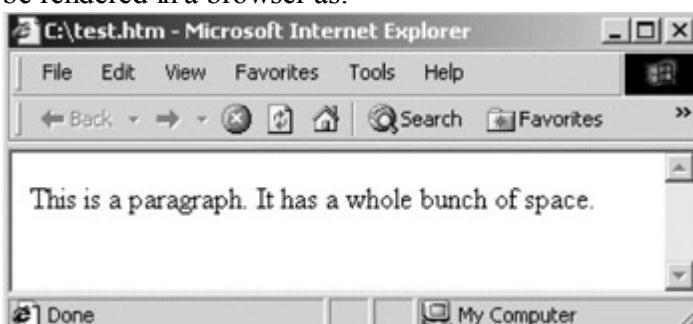
## Whitespace in PCDATA

There is a special category of characters, called **whitespace**. This includes things like the space character, new lines (what you get when you hit the *Enter* key), and tabs. Whitespace is used to separate words, as well as to make text more readable.

Those familiar with HTML are probably quite aware of the practice of whitespace stripping. In HTML, any whitespace considered insignificant is stripped out of the document when it is processed. For example, take the following HTML:

```
<p>This is a paragraph.           It has a whole bunch  
of space.</p>
```

As far as HTML is concerned, anything more than a single space between the words in a <p> is insignificant. So all of the spaces between the first period and the word It would be stripped, except for one. Also, the line feed after the word bunch and the spaces before of would be stripped down to one space. As a result, the previous HTML would be rendered in a browser as:



In order to get the results as they appear in the HTML above, we'd have to add special HTML markup to the source, like the following:

```
<p>This is a paragraph. &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;It has a whole bunch<br>&nbsp;&nbsp;&nbsp;of space.</p>
```

Here, &nbsp; specifies that we should insert a space (nbsp stands for **Non-Breaking SSpace**), and the <br> tag specifies that there should be a line feed. This would format the output as:



Alternatively, if we wanted to have the text displayed exactly as it is in the source file, we could use the `<pre>` tag. This specifically tells the HTML parser not to strip the white space, but to display the text exactly as it appears in the HTML document, so we could write the following and also get the desired results:

```
<pre>This is a paragraph.           It has a whole bunch  
of space.</pre>
```

However, in most web browsers, the `<pre>` tag also has the added effect that the text is rendered in a fixed-width font, like the courier font we use for code in this book.

Whitespace stripping is very advantageous for a language like HTML, which has become primarily a means for displaying information. It allows the source for an HTML document to be formatted in a readable way for the person writing the HTML, while displaying it formatted in a readable, and possibly quite different, way for the user.

In XML, however, no whitespace stripping takes place for PCDATA. This means that for the following XML tag:

```
<tag>This is a paragraph.           It has a whole bunch  
of space.</tag>
```

the PCDATA is:

```
This is a paragraph.           It has a whole bunch  
of space.
```

Just like our second HTML example, none of the whitespace has been stripped out. As far as white space stripping goes, all XML elements are treated just as for the HTML `<pre>` tag. This makes the rules much easier to understand for XML than they are for HTML:

Important

In XML, the whitespace stays.

*Unfortunately, if you view the above XML in IE 5 or later the whitespace will be stripped out?or will seem to be. This is because IE is not actually showing you the XML directly; it uses a technology called XSL to transform the XML to HTML, and it displays the HTML. Then, because IE is an HTML browser, it strips out the whitespace.*

## End-of-Line Whitespace

However, there is one form of whitespace stripping that XML performs on PCDATA, which is the handling of **new line** characters. The problem is that there are two characters that are used for new lines ? the **line feed** character and the **carriage return** ? and computers running Windows, computers running UNIX, and Macintosh computers all use these characters differently.

For example, to get a new line in Windows, an application would use both the line feed and the carriage return character together, whereas on UNIX only the line feed would be used. This could prove to be very troublesome when creating XML documents, because UNIX machines would treat the new lines in a document differently from the

Windows boxes, which would treat them differently from the Macintosh boxes, and our XML interoperability would be lost.

For this reason, it was decided that XML parsers would change all new lines to a single line feed character before processing. This means that any XML application will know, no matter which operating system it's running under, that a new line will be represented by a single line feed character. This makes data exchange between multiple computers running different operating systems that much easier, since programmers don't have to deal with the (sometimes annoying) end-of-line logic.

## Whitespace in Markup

As well as the whitespace in our data, there could also be white space in an XML document that's not actually part of the document. For example:

```
<tag>
  <anotherTag>This is some XML</anotherTag>
</tag>
```

While any white space contained within `<anotherTag>`'s PCDATA is part of the data, there is also a new line after `<tag>`, and some spaces before `<anotherTag>`. These spaces could be there just to make the document easier to read, while not actually being part of its data. This "readability" whitespace is called **extraneous white space**.

While an XML parser must pass all whitespace through to the application, it can also inform the application which whitespace is not actually part of an element's PCDATA, but is just extraneous white space.

So how does the parser decide whether this is extraneous whitespace or not? That depends on what kind of data we specify `<tag>` should contain. If `<tag>` can only contain other elements (and no PCDATA) then the whitespace will be considered extraneous. However, if `<tag>` is allowed to contain PCDATA, then the whitespace will be considered to be part of that PCDATA, so it will be retained.

Unfortunately, from this document alone an XML parser would have no way to tell whether `<tag>` is supposed to contain PCDATA or not, which means that it has to assume none of the whitespace is extraneous. We'll see how we can get the parser to recognize this as extraneous whitespace in [Chapter 5](#) when we discuss content models.

◀ PREVIOUS

[< Free Open Study >](#)

NEXT ▶

# Attributes

In addition to tags and elements, XML documents can also include **attributes**.

## Important

Attributes are simple name/value pairs associated with an element.

They are attached to the start-tag, as shown below, but not to the end-tag:

```
<name nickname="Shiny John">
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

Attributes must have values?even if that value is just an empty string (like "")?and those values must be in quotes. So the following, which is part of a common HTML tag, is not legal in XML:

```
<INPUT checked>
```

and neither is this:

```
<INPUT checked=true>
```

Either single quotes or double quotes are fine, but they have to match. For example, to make this well-formed XML, you can use either of these:

```
< INPUT checked='true'>
< INPUT checked="true">
```

but you can't use:

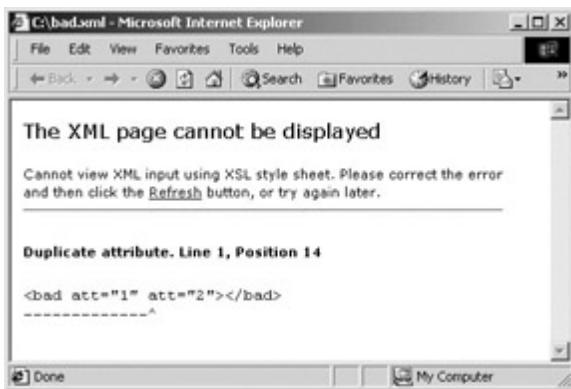
```
< INPUT checked="true'>
```

*Because either single or double quotes are allowed, it's easy to include quote characters in your attribute values, like "John's nickname" or 'I said "hi" to him'. You just have to be careful not to accidentally close your attribute, like 'John's nickname'; if an XML parser sees an attribute value like this, it will think you're closing the value at the second single quote, and will raise an error when it sees the "s" which comes right after it.*

The same rules apply to naming attributes as apply to naming elements: names are case sensitive, can't start with "xml", and so on. Also, you can't have more than one attribute with the same name on an element. So if we create an XML document like this:

```
<bad att="1" att="2"></bad>
```

we will get the following error in IE 5:



Finally, the order in which attributes are included on an element is not considered relevant. In other words, if an XML parser encounters an element like this

```
<name first="John" middle="Fitzgerald Johansen" last="Doe"></name>
```

it doesn't necessarily have to give us the attributes in that order, but can do so in any order it wishes.

Therefore, if there is information in an XML document that must come in a certain order, you should put that information into elements, rather than attributes.

## Try It Out?Adding Attributes to Al's CD

With all of the information we recorded about our CD in our earlier Try It Out, we forgot to include the CD's serial number, or the length of the disc. Let's add some attributes, so that our hypothetical CD Player application can easily find this information.

1.

Open your cd.xml file created earlier, and resave it to your hard drive as cd2.xml.

2.

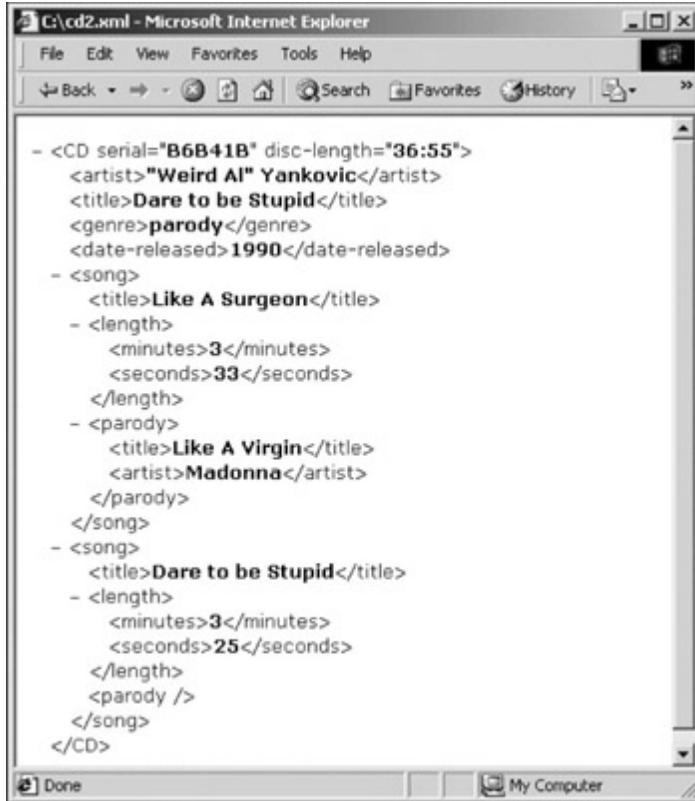
With our new-found attributes knowledge, add two attributes to the <CD> element, like this:

```
<CD serial="B6B41B"
disc-length="36:55">
<artist>Weird Al Yankovic</artist>
<title>Dare to be Stupid</title>
<genre>parody</genre>
<date-released>1990</date-released>
<song>
  <title>Like A Surgeon</title>
  <length>
    <minutes>3</minutes>
    <seconds>33</seconds>
  </length>
  <parody>
    <title>Like A Virgin</title>
    <artist>Madonna</artist>
  </parody>
</song>
<song>
  <title>Dare to be Stupid</title>
  <length>
    <minutes>3</minutes>
    <seconds>25</seconds>
  </length>
  <parody></parody>
</song>
```

</CD>

3.

Save the file, and view it in IE 5 or later. It will look something like this:



## How It Works

Using attributes, we added some information about the CD's serial number and length to our document:

```
<CD serial=B6B41B  
disc-length='36:55'>
```

When the XML parser got to the "=" character after the serial attribute, it expected an opening quotation mark, but instead it found a B. This is an error, and it caused the parser to stop and raise the error to the user.

So we changed our serial attribute declaration:

```
<CD serial='B6B41B'
```

and this time the browser displayed our XML correctly.

The information we added might be useful, for example, in the CD Player application we considered earlier. We could write our CD Player to use the serial number of a CD to load any previous settings the user may have previously saved (such as a custom play list).

## Why Use Attributes?

There have been many debates in the XML community about whether attributes are really necessary, and if so, where they should be used. Here are some of the main points in that debate:

## Attributes Can Provide Meta Data that May Not be Relevant to Most Applications Dealing with Our XML

For example, if we know that some applications may care about a CD's serial number, but most won't, it may make sense to make it an attribute. This logically separates the data most applications will need from the data that most

applications won't need.

In reality, there is no such thing as "pure meta data"?all information is "data" to *some* application. Think about HTML; you could break the information in HTML into two types of data: the data to be shown to a human, and the data to be used by the web browser to format the human-readable data. From one standpoint, the data used to format the data would be meta data, but to the browser or the person writing the HTML, the meta data *is* the data. Therefore, attributes can make sense when we're separating one type of information from another.

## What Do Attributes Buy Me that Elements Don't?

Can't elements do anything attributes can do?

In other words, on the face of it there's really no difference between:

```
<name nickname='Shiny John'></name>
```

and:

```
<name>
  <nickname>Shiny John</nickname>
</name>
```

So why bother to pollute the language with two ways of doing the same thing?

The main reason that XML was invented was that SGML could do some great things, but it was too massively difficult to use without a fully-fledged SGML expert on hand. So one concept behind XML is a simpler, kinder, gentler SGML. For this reason, many people don't like attributes, because they add a complexity to the language that they feel isn't needed.

On the other hand, some people find attributes easier to use?for example, they don't require nesting and you don't have to worry about crossed tags.

## Why Use Elements, If Attributes Take Up So Much Less Space?

Wouldn't it save bandwidth to use attributes instead?

For example, if we were to rewrite our <name> document to use only attributes, it might look like this:

```
<name nickname='Shiny John' first='John' middle='Fitzgerald' last='Doe'></name>
```

Which takes up much less space than our earlier code using elements.

However, in systems where size is really an issue, it turns out that simple compression techniques would work much better than trying to optimize the XML. And because of the way compression works, you end up with almost the same file sizes regardless of whether attributes or elements are used.

Besides, when you try to optimize XML this way, you lose many of the benefits XML offers, such as readability and descriptive tag names.

## Elements Can Be More Complex Than Attributes

When you use attributes, you are limited to simple text as a value. However, when you use elements, your content can be as simple or as complex as you need. That is, when your data is in an element, you have room for expansion, by adding other child elements to further break down the information.

## Sometimes Elements Can Get In The Way

Imagine a case where you have a <note> element, which contains annotations about the text in your XML document. Sometimes the note will be informational, and sometimes a warning. You could include the type of note using an element, such as:

```
<note>
  <type>Information</type>
  This is a note.
</note>
```

or:

```
<note><Information>This is a note.</Information></note>
```

However, it would probably be much less intrusive to include the information in an attribute, such as

```
<note type="Information">This is a note.</note>
```

## Attributes are Un-Ordered

The order of attributes is considered irrelevant. Hence, sometimes you may need to use elements, rather than attributes, for information that must come in the document in a certain order.

## Why Use Attributes When Elements Look So Much Better? I Mean, Why Use Elements When Attributes Look So Much Better?

Many people have different opinions as to whether attributes or child elements "look better". In this case, it comes down to a matter of personal preference and style.

In fact, *much* of the attributes versus elements debate comes from personal preference. Many, but not all, of the arguments boil down to "I like the one better than the other". But since XML has both elements and attributes, and neither one is going to go away, you're free to use both. Choose whichever works best for your application, whichever looks better to you, or whichever you're most comfortable with.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Comments

**Comments** provide a way to insert into an XML document text that isn't really part of the document, but rather is intended for people who are reading the XML source itself.

Anyone who has used a programming language will be familiar with the idea of comments: you want to be able to annotate your code (or your XML), so that those coming after you will be able to figure out what you were doing. (And remember: the one who comes after you may be you! Code you wrote six months ago might be as foreign to you as code someone else wrote.)

Of course, comments may not be as relevant to XML as they are to programming languages; after all, this is just data, and it's self-describing to boot. But you never know when they're going to come in handy, and there are cases where comments can be very useful, even in data.

Comments start with the string `<!--` and end with the string `-->`, as shown here:

```
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle></middle>
  <last>Doe</last>
</name>
```

There are a couple of points that we need to note about comments. First, you can't have a comment inside a tag, so the following is illegal:

```
<middle></middle <!--John lost his middle name in a fire--> >
```

Second, you can't use the string `--` inside a comment, so the following is also illegal:

```
<!--John lost his middle name -- in a fire-->
```

The XML specification states that an XML parser doesn't need to pass these comments on to the application, meaning that you should never count on being able to use the information inside a comment from your application. Comments are only there for the benefit of someone reading your XML markup.

## Important

HTML programmers have often used the trick of inserting scripting code in comments, to protect users with older browsers that didn't support the `<script>` tag. That kind of trick can't be done in XML, since comments won't necessarily be available to the application. Therefore, if you have data that you need to get at later from your applications, *put it in an element or an attribute!*

## Try It Out?Some Comments On Al's CD

Since we've only included a couple of the songs from this fine album in our document, perhaps we should inform others that this is the case. That way some kind soul may finish the job for us!

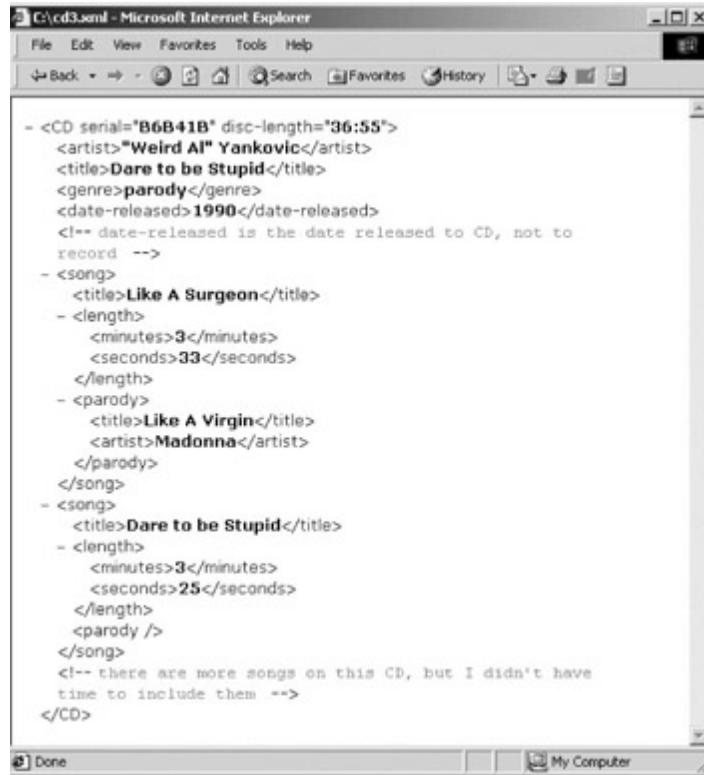
1.

Open up your cd2.xml file, make the following changes, and save the modified XML file as cd3.xml:

```
<CD serial='B6B41B'  
disc-length='36:55'>  
<artist>"Weird Al" Yankovic</artist>  
<title>Dare to be Stupid</title>  
<genre>parody</genre>  
<date-released>1990</date-released>  
<!--date-released is the date released to CD, not to record-->  
<song>  
    <title>Like A Surgeon</title>  
    <length>  
        <minutes>3</minutes>  
        <seconds>33</seconds>  
    </length>  
    <parody>  
        <title>Like A Virgin</title>  
        <artist>Madonna</artist>  
    </parody>  
</song>  
<song>  
    <title>Dare to be Stupid</title>  
    <length>  
        <minutes>3</minutes>  
        <seconds>25</seconds>  
    </length>  
    <parody></parody>  
</song>  
<!--there are more songs on this CD, but I didn't have time to include  
them-->  
</CD>
```

2.

View this in IE 5 or later:



## How It Works

With the new comments, anyone who reads the source for our XML document will be able to see that there are actually more than two songs on "Dare To Be Stupid". Furthermore, they can see some information regarding the

<date-released> element, which may help them in writing applications that work with this information.

In this example, the XML parser included with IE 5 *does* pass comments up to the application, so IE 5 has displayed our comments. But remember that a lot of the time, for all intents and purposes this information is only available to people reading the source file. The information in comments *may or may not* be passed up to our application, depending on which parser we're using. We can't count on it, unless we specifically choose a parser that does pass them through. This means that the application has no way to know whether or not the list of songs included here is comprehensive.

## Try It Out?Making Sure The Data Gets Seen

If we really need the information contained in the comments in the previous example, we should add in some real markup to indicate it.

1.

Modify cd3.xml like this, and save it as cd4.xml:

```
<CD><!--our attributes used to be here-->
<songs>11</songs>
<!--the rest of our XML...-->
<artist>"Weird Al" Yankovic</artist>
<title>Dare to be Stupid</title>
<genre>parody</genre>
<date-released>1990</date-released>
<song>
    <title>Like A Surgeon</title>
    <length>
        <minutes>3</minutes>
        <seconds>33</seconds>
    </length>
    <parody>
        <title>Like A Virgin</title>
        <artist>Madonna</artist>
    </parody>
</song>
<song>
    <title>Dare to be Stupid</title>
    <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
    </length>
    <parody></parody>
</song>
</CD>
```

2.

This XML is formatted like this in IE 5 or later:



The screenshot shows a Microsoft Internet Explorer window displaying an XML file named 'cd4.xml'. The XML code is as follows:

```
<?xml version="1.0"?>
<CD>
  <!-- our attributes used to be here -->
  <songs>11</songs>
  <!-- the rest of our XML... -->
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  - <song>
    <title>Like A Surgeon</title>
    - <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    - <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
  </song>
  - <song>
    <title>Dare to be Stupid</title>
    - <length>
      <minutes>3</minutes>
      <seconds>25</seconds>
    </length>
    <parody />
  </song>
</CD>
```

This way, the application could be coded such that if it only finds two `<song>` elements, but it finds a `<songs>` element which contains the text "11", it can deduce that there are nine songs missing.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Empty Elements

Sometimes an element has no PCDATA. Recall our earlier example, where the `<middle>` element contained no name:

```
<name nickname='Shiny John'>
<first>John</first>
<!--John lost his middle name in a fire-->
<middle></middle>
<last>Doe</last>
</name>
```

In this case, you also have the option of writing this element using the special **empty element syntax**:

```
<middle/>
```

This is the one case where a start-tag doesn't need a separate end-tag, because they are both combined together into this one tag. In all other cases, they do.

Recall from our discussion of element names that the only place we can have a space within the tag is before the closing "`>`". This rule is slightly different when it comes to empty elements. The "`/`" and "`>`" characters always have to be together, so you can create an empty element like this:

```
<middle />
```

but not like these:

```
<middle/ >
<middle / >
```

Empty elements really don't buy you anything-except that they take less typing-so you can use them, or not, at your discretion. Keep in mind, however, that as far as XML is concerned `<middle></middle>` is *exactly* the same as `<middle/>`; for this reason, XML parsers will sometimes change your XML from one form to the other. You should never count on your empty elements being in one form or the other, but since they're syntactically exactly the same, it doesn't matter. (This is the reason that Internet Explorer felt free to change our earlier `<parody></parody>` syntax to just `<parody/>`.)

*Interestingly, nobody in the XML community seems to mind the empty element syntax, even though it doesn't add anything to the language. This is especially interesting considering the passionate debates that have taken place on whether attributes are really necessary.*

One place where empty elements are very often used is for elements that have no (or optional) PCDATA, but instead have all of their information stored in attributes. So if we rewrote our `<name>` example without child elements, instead of a start-tag and end-tag we would probably use an empty element, like this:

```
<name first="John" middle="Fitzgerald Johansen" last="Doe"/>
```

Another common example is the case where just the element name is enough; for example, the HTML `<BR>` tag would be converted to an XML empty element, such as the XHTML `<br/>` tag. (XHTML is the latest "XML-compliant" version of HTML.)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# XML Declaration

It is often very handy to be able to identify a document as being of a certain type. XML provides the **XML declaration** for us to label documents as being XML, along with giving the parsers a few other pieces of information. You don't need to have an XML declaration, but it's considered good practice to include it.

A typical XML declaration looks like this:

```
<?xml version='1.0' encoding='UTF-16' standalone='yes'?>
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle/>
  <last>Doe</last>
</name>
```

Some things to note about the XML declaration:

- The XML declaration starts with the characters <?xml, and ends with the characters ?>.
- If you include it, you must include the version, but the encoding and standalone attributes are optional.
- The version, encoding, and standalone attributes must be in that order.
- Currently, the version should be 1.0. If you use a number other than 1.0, XML parsers that were written for the version 1.0 specification can reject the document. (If there is ever a new version of the XML Recommendation, the version number in the XML declaration will be used to signal which version of the specification your document claims to support.)
- The XML declaration must be right at the beginning of the file. That is, the first character in the file should be that "<"; no line breaks or spaces. Some parsers are more forgiving about this than others.

So an XML declaration can be as full as the previous one, or as simple as:

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

The next two sections will describe more fully the encoding and standalone attributes of the XML declaration.

## Encoding

It should come as no surprise to us that text is stored in computers using numbers, since numbers are all that computers really understand.

## Important

A **character code** is a one-to-one mapping between a set of characters and the corresponding numbers to represent those characters. A **character encoding** is the method used to represent the numbers in a character code digitally (in other words how many bytes should be used for each number, etc.)

One character code that you might have come across is the **American Standard Code for Information Interchange (ASCII)**. For example, in ASCII the character "a" is represented by the number 97, and the character "A" is represented by the number 65.

There are seven-bit and eight-bit ASCII encoding schemes. 7-bit ASCII uses 7 bits for each character, which limits it to 128 different values, while 8-bit ASCII uses one byte (8 bits) for each character, which limits it to 256 different values. 7-bit ASCII is a much more universal standard for text, while there are a number of 8-bit ASCII character codes, which were created to add additional characters not covered by ASCII, such as ISO-8859-1. Each 8-bit ASCII encoding scheme might have slightly different sets of characters represented, and those characters might map to a different number. However, the first 128 characters are always the same as the 7-bit ASCII character code

ASCII can easily handle all of the characters needed for English, which is why it was the predominant character encoding used on personal computers in the English-speaking world for many years. But there are way more than 256 characters in all of the world's languages, so obviously ASCII (or any other 8-bit encoding limited to 256 characters) can only handle a small subset of these. This is why **Unicode** was invented.

## Unicode

Unicode is a character code designed from the ground up with internationalization in mind, aiming to have enough possible characters to cover all of the characters in any human language. There are two major character encodings for Unicode: **UTF-16** and **UTF-8**. UTF-16 takes the easy way, and simply uses two bytes for every character (two bytes = 16 bits = 65,356 possible values).

UTF-8 is more clever: it uses one byte for the characters covered by 7-bit ASCII, and then uses some tricks so that any other characters may be represented by two or more bytes. This means that 7-bit ASCII text can actually be considered a subset of UTF-8, and processed as such. For text written in English, where most of the characters would fit into the ASCII 7-bit character encoding, UTF-8 can result in smaller file sizes, but for text in other languages, UTF-16 can be smaller.

Because of the work done with Unicode to make it international, the XML specification states that all XML processors must use Unicode internally. Unfortunately, very few of the documents in the world are encoded in Unicode. Most are encoded in **ISO-8859-1**, or **windows-1252**, or **EBCDIC**, or one of a large number of other character codes. (Many of these character codes, such as ISO-8859-1 and windows-1252, are actually 8-bit ASCII character codes. They are not, however, subsets of UTF-8 in the same way that "pure" 7-bit ASCII is.)

## Specifying A Character Encoding for XML

This is where the encoding attribute in our XML declaration comes in. It allows us to specify, to the XML parser, what character encoding our text is in. The XML parser can then read the document in the proper encoding, and translate it into Unicode internally. If no encoding is specified, UTF-8 or UTF-16 is assumed (parsers must support at least UTF-8 and UTF-16). If no encoding is specified, and the document is not UTF-8 or UTF-16, it results in an error.

Sometimes an XML processor is allowed to ignore the encoding specified in the XML declaration. If the document is being sent via a network protocol such as HTTP, there may be protocol-specific headers that specify a different encoding than the one specified in the document. In such a case, the HTTP header would take precedence over the

encoding specified in the XML declaration. However, if there are no external sources for the encoding, and the encoding specified is different from the actual encoding of the document, it results in an error.

If you're creating XML documents in Notepad on a machine running a Microsoft Windows operating system, in the English speaking world, the character encoding you are probably using by default is windows-1252. So the XML declarations in your documents should look like this:

```
<?xml version="1.0" encoding="windows-1252"?>
```

However, not all XML parsers understand the windows-1252 character set, meaning that a document that claims to use it may cause the parser to raise an error, indicating that it can't be processed. If that's the case, try substituting ISO-8859-1, which happens to be very similar. Or, if your document doesn't contain any special characters (like accented characters, for example), you could use ASCII instead, or leave the encoding attribute out, and let the XML parser treat the document as UTF-8.

If you're running Windows NT or Windows 2000, Notepad also gives you the option of saving your text files in Unicode, in which case you can leave out the encoding attribute in your XML declarations.

## Standalone

If the standalone attribute is included in the XML declaration, it must be set to either yes or no.

- - yes specifies that this document exists entirely on its own, without depending on any other files
  - no indicates that the document may depend on an external DTD (DTDs will be covered in [Chapter 5](#))

This little attribute actually has its own name: the **Standalone Document Declaration**, or **SDD**. The XML specification doesn't actually require a parser to do anything with the SDD. It is considered more of a hint to the parser than anything else.

It's time to take a look at how the XML declaration works in practice.

## Try It Out-Declaring Al's CD to the World

Let's declare our XML document, so that any parsers will be able to tell right away what it is. And, while we're at it, let's take care of that second <parody> element, which doesn't have any content.

1.

Open up the file cd3.xml, and make the following changes:

```
<?xml version='1.0' encoding='windows-1252' standalone='yes'?>
<CD serial='B6B41B'
    disc-length='36:55'>
    <artist>"Weird Al" Yankovic</artist>
    <title>Dare to be Stupid</title>
    <genre>parody</genre>
    <date-released>1990</date-released>
    <!--date-released is the date released to CD, not to record-->
    <song>
        <title>Like A Surgeon</title>
        <length>
            <minutes>3</minutes>
            <seconds>33</seconds>
        </length>
```

```
<parody>
  <title>Like A Virgin</title>
  <artist>Madonna</artist>
</parody>
</song>
<song>
  <title>Dare to be Stupid</title>
  <length>
    <minutes>3</minutes>
    <seconds>25</seconds>
  </length>
  <parody/>
</song>
<!--There are more songs on this CD, but I didn't have time
   to include them!--&gt;
&lt;/CD&gt;</pre>
```

## 2.

Save the file as cd5.xml, and view it in IE 5 or later:



## How It Works

With our new XML declaration, any XML parser can tell right away that it is indeed dealing with an XML document, and that document is claiming to conform to version 1.0 of the XML specification.

Furthermore, the document indicates that it is encoded using the windows-1252 character encoding. Again many XML parsers don't understand windows-1252, so you may have to play around with the encoding. Luckily, the parser used by Internet Explorer 5 does understand windows-1252, so if you're viewing the examples in IE 5 you can leave the XML declaration as it is here.

In addition, because the Standalone Document Declaration declares that this is a standalone document, the parser knows that this one file is all that it needs to fully process the information.

And finally, because "Dare to be Stupid" is not a parody of any particular song, the `<parody>` element has been changed to an empty element. That way we can visually emphasize the fact that there is no information there. Remember, though, that to the parser `<parody/>` is exactly the same as `<parody></parody>`, which is why this part of our document looks the same as it did in our earlier screenshots.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Processing Instructions

Although it isn't all that common, sometimes you need to embed application-specific instructions into your information, to affect how it will be processed. XML provides a mechanism to allow this, called **processing instructions** or, more commonly, **PIs**. These allow you to enter instructions into your XML which are not part of the actual document, but which are passed up to the application.

```
<?xml version='1.0' encoding='UTF-16' standalone='yes'?>
<name nickname='Shiny John'>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle/>
  <?nameprocessor SELECT * FROM blah?>
  <last>Doe</last>
</name>
```

There aren't really a lot of rules on PIs. They're basically just a "<?", the name of the application that is supposed to receive the PI (the **PITarget**), and the rest up until the ending "?>" is whatever you want the instruction to be. The PITarget is bound by the same naming rules as elements and attributes. So, in this example, the PITarget is nameprocessor, and the actual text of the PI is SELECT \* FROM blah.

PIs are pretty rare, and are often frowned upon in the XML community, especially when used frivolously. But if you have a valid reason to use them, go for it. For example, PIs can be an excellent place for putting the kind of information (such as scripting code) that gets put in comments in HTML. While you can't assume that comments will be passed on to the application, PIs always are.

## Is the XML Declaration a Processing Instruction?

At first glance, you might think that the XML declaration is a PI that starts with xml. It uses the same "<? ?>" notation, and provides instructions to the parser (but not the application). So is it a PI?

Actually, no: the XML declaration isn't a PI, but in most cases it really doesn't make any difference whether it is or not. The only places where you'll get into trouble are the following:

- Trying to get the text of the XML declaration from an XML parser. Some parsers erroneously treat the XML declaration as a PI, and will pass it on as if it were, but most will not. The truth is, in most cases your application will never need the information in the XML declaration; that information is only for the parser. One notable exception might be an application that wants to display an XML document to a user, in the way that we're using IE 5 to display the documents in this book.
- Including an XML declaration somewhere other than at the beginning of an XML document. Although you can put a PI anywhere you want, an XML declaration must come at the beginning of a file.

## Try It Out?Dare to be Processed

Just to see what it looks like, let's add a processing instruction to our Weird AI XML:

1.

Make the following changes to cd5.xml and save the file as cd6.xml:

```
<?xml version='1.0' encoding='windows-1252' standalone='yes'?>
```

```
<CD serial='B6B41B'  
disc-length='36:55'>  
<artist>"Weird Al" Yankovic</artist>  
<title>Dare to be Stupid</title>  
<genre>parody</genre>  
<date-released>1990</date-released>  
<!--date-released is the date released to CD, not to record-->  
<song>  
    <title>Like A Surgeon</title>  
    <length>  
        <minutes>3</minutes>  
        <seconds>33</seconds>  
    </length>  
    <parody>  
        <title>Like A Virgin</title>  
        <artist>Madonna</artist>  
    </parody>  
</song>  
<song>  
    <title>Dare to be Stupid</title>  
    <length>  
        <minutes>3</minutes>  
        <seconds>25</seconds>  
    </length>  
    <parody/>  
</song>  
<?CDParser MessageBox("There are songs missing!")?>  
</CD>
```

2.

In IE 5 or later, it looks like this:



## How It Works

For our example, we are targeting a *fictional* application called CDPParser, and giving it the instruction `MessageBox("There are songs missing!")`. The instruction we gave it has no meaning in the context of XML itself, but only to our CDPParser application, so it's up to CDPParser to do something meaningful with it.

[PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

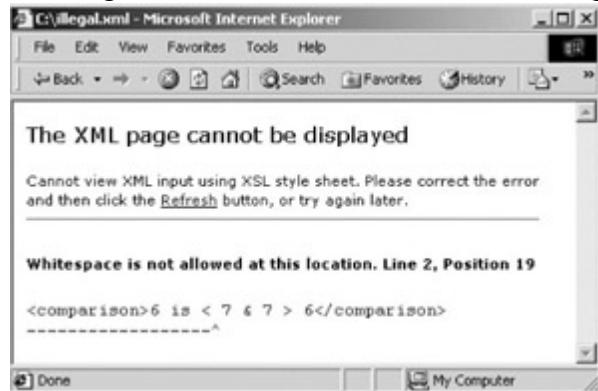
NEXT &gt;

# Illegal PCDATA Characters

There are some reserved characters that you can't include in your PCDATA because they are used in XML syntax, the "<" and "&" characters:

```
<!--This is not well-formed XML!-->
<comparison>6 is < 7 & 7 > 6</comparison>
```

Viewing the above XML in IE 5 or later would give the following error:



This means that the XML parser comes across the "<" character, and expects a tag name, instead of a space. (Even if it had got past this, the same error would have occurred at the "&" character.)

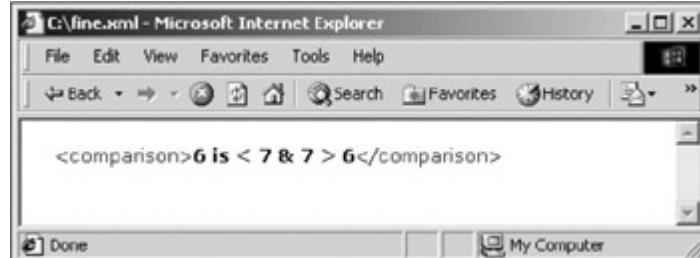
There are two ways you can get around this: **escaping characters**, or enclosing text in a **CDATA section**.

## Escaping Characters

To escape these two characters, you simply replace any "<" characters with "&lt;" and any "&" characters with "&amp;". (In addition, you can also escape the ">" character with "&gt;". It isn't necessary, but it does make things more consistent, since you need to escape all of the "<" characters.) The above XML could be made well-formed by doing the following:

```
<comparison>6 is &lt; 7 &amp; 7 &gt; 6 </comparison>
```

Which displays properly in the browser:



Notice that IE 5 automatically un-escapes the characters for you when it displays the document, in other words it replaces the "&lt;", "&amp;" and "&gt;" strings with "<", "&" and ">" characters.

&lt; and &amp; are known as **entity references**. The following entities are defined in XML:

-

- &#x26;-the & character
- &lt;-the < character
- &gt;-the > character
- &apos;-the ' character
- &quot;-the " character

Other characters can also be escaped by using **character references**. These are strings such as &#nnn;, where "nnn" would be replaced by the Unicode number of the character you want to insert. (Or &#xnnn; with an "x" preceding the number, where "nnn" is a hexadecimal representation of the Unicode character you want to insert. All of the characters in the Unicode specification are specified using hexadecimal, so allowing the hexadecimal numbers in XML means that XML authors don't have to convert back and forth between hexadecimal and decimal.)

Escaping characters in this way can be quite handy if you are authoring documents in XML that use characters your XML editor doesn't understand, or can't output, because the characters escaped are *always* Unicode characters, regardless of the encoding being used for the document. As an example, you could include the copyright symbol () in an XML document by inserting &#169; or &#xA9;;

## CDATA Sections

If you have a lot of "<" and "&" characters that need escaping, you may find that your document quickly becomes very ugly and unreadable. Luckily, there are also **CDATA sections**.

### Important

CDATA is another inherited term from SGML. It stands for Character DATA.

Using CDATA sections, we can tell the XML parser not to parse the text, but to let it all go by until it gets to the end of the section. CDATA sections look like this:

```
<comparison><! [CDATA[6 is < 7 & 7 > 6]]></comparison>
```

Everything starting after the <!<![CDATA[ and ending at the ]]&gt; is ignored by the parser, and passed through to the application as is. The only character sequence that can't occur within a CDATA section is "]]&gt;", since the XML parser would think that you were closing the CDATA section. If you needed to include this sequence, you would be better off keeping it out of the CDATA section, like this:</p>

```
<! [CDATA[This text contains the sequence '']]>]]><! [CDATA[' in it.]]>
```

In these trivial cases, CDATA sections may look more confusing than the escaping did, but in other cases it can turn out to be more readable. For example, consider the following example, which uses a CDATA section to keep an XML parser from parsing a section of JavaScript:

```
<script language='JavaScript'><! [CDATA[function myFunc()
{
    if(0 < 1 && 1 < 2)
        alert("Hello");
}
```

```
}]></script>
```

This displays in the IE 5 or later browser as:



Notice the vertical line at the left-hand side of the CDATA section. This is indicating that although the CDATA section is indented for readability, the actual data itself starts at that vertical line. This is so we can visually see what whitespace is included in the CDATA section.

If you're familiar with JavaScript, you'll probably find the if statement much easier to read than:

```
if(0 < 1 && 1 < 2)
```

## Try It Out-Talking About HTML in XML

Suppose that we want to create XML documentation to describe some of the various HTML tags in existence.

1.

We might develop a simple document type such as the following:

```
<HTML-Doc>
<tag>
  <tag-name></tag-name>
  <description></description>
  <example></example>
</tag>
</HTML-Doc>
```

In this case, we know for sure that our <example> element is going to need to include HTML syntax, meaning that there are going to be a lot of "<" characters included. This makes <example> the perfect place to use a CDATA section, meaning that we don't have to search through all of our HTML code looking for illegal characters. To demonstrate, let's document a couple of HTML tags.

2.

Create a new file and type this code:

```
<HTML-Doc>
<tag>
  <tag-name>P</tag-name>
  <description>Paragraph</description>
  <example><![CDATA[
<P>Paragraphs can contain <EM>other</EM> tags.</P>
]]></example>
```

```
</tag>
<tag>
  <tag-name>HTML</tag-name>
  <description>HTML root element</description>
  <example><![CDATA[
<HTML>
<HEAD><TITLE>Sample HTML</TITLE></HEAD>
<BODY>
<P>Stuff goes here</P>
</BODY></HTML>
]]></example>
</tag>
<!--more tags to follow...-->
</HTML-Doc>
```

### 3.

Save this document as html-doc.xml and view it in IE 5 or later:



## How It Works

Because of our CDATA sections, we can put whatever we want into the `<example>` elements, and don't have to worry about the text being mixed up with the actual XML markup of the document. This means that even though there are typos in the second `<example>` element (the `</P>` is missing the `>` and `/HTML>` is missing a `<`), our XML is not affected.

[PREVIOUS](#)

[< Free Open Study >](#)

[NEXT](#)

# Parsing XML

The main reason for creating all of these rules about writing well-formed XML documents is so that we can create a computer program to read in the data, and easily tell markup from information.

*According to the XML specification (<http://www.w3.org/TR/1998/REC-xml-19980210#sec-intro>): "A software module called an **XML processor** is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module, called the application."*

An XML processor is more commonly called a **parser**, since it simply parses XML and provides the application with any information it needs. That is, it reads through the characters in the document, determines which characters are part of the document's markup and which are part of the document's data, and does all of the other processing of an XML document that happens before an application can make use of it. There are quite a number of XML parsers available, many of which are free. Some of the better-known ones are listed below.

## Microsoft Internet Explorer Parser

Microsoft's XML parser, MSXML, first shipped with Internet Explorer 4, and implemented an early draft of the XML specification. With the release of IE 5, the XML implementation was upgraded to reflect the XML version 1 specification. The latest version of the parser is available for download from <http://msdn.microsoft.com/downloads/webtechnology/xml/msxml.asp>.

## Apache Xerces

The Apache Software Foundation's Xerces sub-project of the Apache XML Project (<http://xml.apache.org/>) has resulted in XML parsers in Java and C++, plus a Perl wrapper for the C++ parser. These tools are free, and the distribution of the code is controlled by the GNU Public License.

## James Clark's Expat

Expat is an XML 1.0 parser toolkit written in C. More information can be found at <http://www.jclark.com/xml/expat.html> and Expat can be downloaded from <ftp://ftp.jclark.com/pub/xml/expat.zip>. It is free for both private and commercial use.

## Vivid Creations ActiveDOM

Vivid Creations (<http://www.vivid-creations.com>) offers several XML tools, including ActiveDOM. ActiveDOM contains a parser similar to the Microsoft parser and, although it is a commercial product, a demonstration version may be downloaded from the Vivid Creations web site.

## Xml4j

IBM's AlphaWorks site (<http://www.alphaworks.ibm.com>) offers a number of XML tools and applications, including the xml4j parser. This is another parser written in Java, available for free, though there are some licensing restrictions regarding its use.

## Errors in XML

As well as specifying how a parser should get the information out of an XML document, it is also specified how a parser should deal with errors in XML. There are two types of error in the XML specification: **errors** and **fatal errors**.

- - An error is simply a violation of the rules in the specification, where the results are undefined; the XML processor is allowed to recover from the error and continue processing.
  - 
  - Fatal errors are more serious: according to the specification a parser is *not allowed to continue as normal* when it encounters a fatal error. (It may, however, keep processing the XML document to search for further errors.) This is called **draconian error handling**. Any error which causes an XML document to cease being well-formed is a fatal error.

The reason for this drastic handling of non-well-formed XML is simple: it would be extremely hard for parser writers to try and handle "well-formedness" errors, and it is extremely simple to make XML well-formed. (Web browsers don't force documents to be as strict as XML does, but this is one of the reasons why web browsers are so incompatible; they must deal with *all* of the errors they may encounter, and try to figure out what the person who wrote the document was really trying to code.)

But draconian error handling doesn't just benefit the parser writers; it also benefits us when we're creating XML documents. If I write an XML document that doesn't properly follow XML's syntax, I can find out right away and fix my mistake. On the other hand, if the XML parser tries to recover from these errors, it may misinterpret what I was trying to do, but I wouldn't know about it because no error would be raised. In this case, bugs in my software would be much harder to track down, instead of being caught right at the beginning when I was creating my data. Even worse, if I sent my XML document to someone else, their parser might interpret the mistake differently.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

This chapter has provided you with the basic syntax for writing well-formed XML documents.

We've seen:

- Elements and empty elements
- How to deal with whitespace in XML
- Attributes
- How to include comments
- XML declarations and encodings
- Processing instructions
- Entity references, character references, and CDATA sections

We've also learned why the strict rules of XML grammar actually benefit us, in the long run, since they force us to catch our errors sooner, rather than later, and how some of the rules for authoring HTML are different from the rules for authoring well-formed XML.

Unfortunately—or perhaps fortunately—you probably won't spend much of your time just authoring XML documents. You have to do something useful with the data! In the chapters that follow we'll learn about a very important part of XML, namespaces.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 3: XML Namespaces

## Overview

We can now create well-formed XML documents, and have seen why XML provides some benefits over binary formats. But the time is going to come when our applications get more complex, and we'll need to combine elements from various document types into one XML document.

Unfortunately, there will very often be cases where two document types have elements with the same name, but with different meanings and semantics. This chapter will introduce **XML Namespaces**, the means by which we can differentiate elements and attributes of different XML document types from each other when combining them together into other documents, or even when processing multiple documents simultaneously.

In this chapter you will learn:

- Why we need namespaces
- What namespaces are, conceptually, and how they solve the problem
- The syntax for using namespaces in XML documents
- What a URI is, what a URL is, and what a URN is

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Why Do We Need Namespaces?

Because of the nature of XML, it is possible for any company or individual to create XML document types which describe the world in their own terms. If my company feels that an `<order>` should contain a certain set of information, and another company feels that it should contain a different set of information, we can both go ahead and create different document types to describe that information. We can even both use the name `<order>` for entirely different uses, if we wish.

However, if everyone is creating personalized XML vocabularies, we'll soon run into a problem: there are only so many words available in human languages, and a lot of them are going to get snapped up by people defining document types. How can I define a `<title>` element, to be used to denote the title in a person's name, when XHTML already has a `<title>` element, which is used to describe the title of an HTML document? How can I then further distinguish those two `<title>` elements from the title of a book?

If all of these documents were to be kept separate, this still would not be a problem. If we saw a `<title>` element in an XHTML document, we'd know what kind of title we were talking about, and if we saw one in our own proprietary XML document type, we'd know what that meant too. Unfortunately, life isn't always that simple, and eventually we're going to need to combine various XML elements from different document types into one XML document. For example, we might create an XML document type containing information about a person, including that person's title, but also containing the person's résumé, in XHTML form. Such a document may look similar to this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<person>
  <name>
    <title>Sir</title>
    <first>John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
  <position>Vice President of Marketing</position>
  <r sum >
    <html>
      <head><title>Resume of John Doe</title></head>
      <body>
        <h1>John Doe</h1>
        <p>John's a great guy, you know?</p>
      </body>
    </html>
  </r sum >
</person>
```

To an XML parser, there isn't any difference between the two `<title>` elements in this document. If we do a simple search of the document to find John Doe's title, we might accidentally get "Resume of John Doe", instead of "Sir". Even in our application, we can't know which elements are XHTML elements and which aren't without knowing in advance the structure of the document. That is, we'd have to know that there is a `<r sum >` element, which is a direct child of `<person>`, and that all of the descendants of `<r sum >` are a separate type of element from the others in our document. If our structure ever changed, all of our assumptions would be lost. In the document above it looks like anything inside the `<r sum >` element is XHTML, but in other documents it might not be so obvious, and to an XML parser it isn't obvious at all.

## Using Prefixes

The best way to solve this problem is for every element in a document to have a completely distinct name. For example, we might come up with a naming convention whereby every element for my proprietary XML document type gets my own prefix, and every XHTML element gets another prefix.

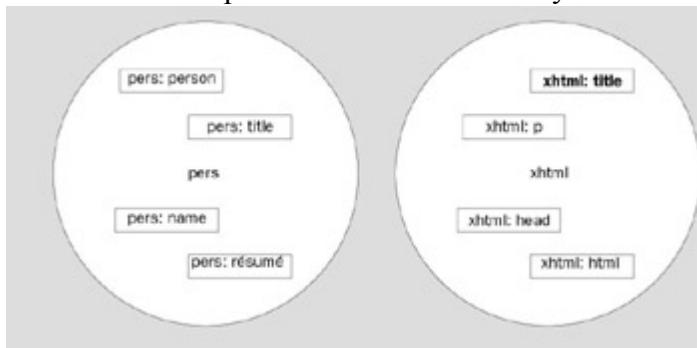
We could rewrite our XML document from above something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pers:person>
  <pers:name>
    <pers:title>Sir</pers:title>
    <pers:first>John</pers:first>
    <pers:middle>Fitzgerald Johansen</pers:middle>
    <pers:last>Doe</pers:last>
  </pers:name>
  <pers:position>Vice President of Marketing</pers:position>
  <pers:résumé>
    <xhtml:html>
      <xhtml:head><xhtml:title>Resume of John Doe</xhtml:title></xhtml:head>
      <xhtml:body>
        <xhtml:h1>John Doe</xhtml:h1>
        <xhtml:p>John's a great guy, you know?</xhtml:p>
      </xhtml:body>
    </xhtml:html>
  </pers:résumé>
</pers:person>
```

This is a bit uglier, but at least we (and our XML parser) can immediately tell what kind of title we're talking about: a <pers:title> or an <xhtml:title>. Doing a search for <pers:title> will always return "Sir". We can always immediately tell which elements are XHTML elements, without having to know in advance the structure of our document.

By separating these elements using a prefix, we have effectively created two kinds of elements in our document: pers types of elements, and xhtml types of elements. So any elements with the pers prefix belong in the same "category" as each other, just as any elements with the xhtml prefix belong in another "category". These "categories" are called **namespaces**.

These two namespaces could be illustrated by the following diagram:



Note that namespaces are concerned with a *vocabulary*, not a *document type*. That is, the namespace distinguishes which names are in the namespace, but not what they mean or how they fit together. It is simply a "bag of names".

#### Important

A namespace is a purely abstract entity; it's nothing more than a group of names that belong with each other conceptually.

*The concept of namespaces also exists in certain programming languages, such as Java, where the same problem exists. How can I name my Java variables whatever I want, and not have those names conflict with names already defined by others, or even by the Java library itself? The answer is that Java code is broken up into packages, where the names within a package must be unique, but the same name can be used in any package.*

For example, there is a class defined in Java named java.applet.Applet. The actual name of the class is just Applet; java.applet is the package which contains that class. This means that I can create my own package, and in that package I could define a class of my own named Applet. I can even use java.applet.Applet from within my package, as long as I specify the package in which it resides, so that Java always knows which Applet I'm referring to.

## So Why Doesn't XML Just Use These Prefixes?

Unfortunately, there is a drawback to the prefix approach to namespaces used in the previous XML: who will monitor the prefixes? The whole reason for using them is to distinguish names from different document types, but if it is going to work, the prefixes themselves also have to be unique. If one company chose the prefix pers and another company also chose that same prefix, the original problem would still exist.

In fact, this prefix administration would have to work a lot like it works now for domain names on the Internet: a company or individual would go to the "prefix administrators" with the prefix they would like to use; if that prefix wasn't already being used, they could use it, otherwise they would have to pick another one.

In order to solve this problem, we could take advantage of the already unambiguous Internet domain names in existence, and specify that **URIs** must be used for the prefix names.

### Important

A URI (Uniform Resource Identifier) is a string of characters that identifies a resource. It can come in one of two flavors: URL (Uniform Resource Locator), or URN (Universal Resource Name). We'll look at the differences between URLs and URNs later in this chapter.

For example, if I work for a company called **Serna Ferna Inc.**, which owns the domain name sernaferna.com, I could incorporate that into my prefix. Perhaps the document might end up looking like this:

```
<?xml version="1.0" encoding="windows-1252ISO-8859-1"?>
<{http://sernaferna.com/pers}person>
  <{http://sernaferna.com/pers}name>
  <{http://sernaferna.com/pers}title>
    Sir
  </{http://sernaferna.com/pers}title>
<!--etc...-->
```

Voila! We have solved our problem of uniqueness. Since our company owns the sernaferna.com domain name, we know that nobody else will be using that http://sernaferna.com/pers prefix in their XML documents, and if we want to create any additional document types, we can just keep using our domain name, and add the new namespace name to the end, such as http://sernaferna.com/other-namespace.

It's important to note that we need more than just the sernaferna.com part of the URI; we need the whole thing. Otherwise there would be a further problem: different people could have control of different sections on that domain, and they might all want to create namespaces. For example, the company's HR department could be in charge of http://sernaferna.com/hr, and might need to create a namespace for names (of employees), and the sales department could be in charge of http://sernaferna.com/sales, and also need to create a namespace for names (of customers). As long as we're using the whole URI, we're fine?we can both create our namespaces (in this case http://sernaferna.com/hr/names and http://sernaferna.com/sales/names respectively). We also need the protocol (http) in there because there could be yet another department which is in charge of, for example, ftp://sernaferna.com/hr and ftp://sernaferna.com/sales.

The only drawback to this solution is that our XML is no longer well-formed. Our names can now include a myriad

of characters that are allowed in URIs but not in XML names: / characters, for example, and for the sake of this example we used {} characters to separate the URL from the name, neither of which is allowed in XML element or attribute name.

What we really need, to solve all of our namespace-related problems, is a way to create two-part names in XML: one part would be the name we are giving this element, and the other part would be an arbitrarily chosen prefix that refers to a URI, which specifies which namespace this element belongs to. And, in fact, this is what **XML Namespaces** provide.

*The XML Namespaces specification is located at: <http://www.w3.org/TR/REC-xml-names/>*

---

[!\[\]\(ab902d5cae40e0c1113d17a0b183f101\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(aa74e793ac544b887eed0d5d51f43a8b\_img.jpg\) NEXT >](#)

# How XML Namespaces Work

To use XML Namespaces in your documents, elements are given **qualified names**. (In most W3C specifications "qualified name" is abbreviated to **QName**.) These qualified names consist of two parts: the **local part**, which is the same as the names we have been giving elements all along, and the **namespace prefix**, which specifies to which namespace this name belongs.

For example, to declare a namespace called `http://sernaferna.com/pers`, and associate a `<person>` element with that namespace, we would do something like the following:

```
<pers:person xmlns:pers="http://sernaferna.com/pers"/>
```

The key is the `xmlns:pers` attribute ("xmlns" stands for XML NameSpace). Here we are declaring the `pers` namespace prefix, and the URI of the namespace which it represents (`http://sernaferna.com/pers`). We can then use the namespace prefix with our elements, as in `pers:person`. As opposed to our previous prefixed version, the prefix itself (`pers`) doesn't have any meaning—its only purpose is to point to the namespace name. For this reason, we could replace our prefix (`pers`) with any other prefix, and this document would have exactly the same meaning.

This prefix can be used for any descendants of the `<pers:person>` element, to denote that they also belong to the `http://sernaferna.com/pers` namespace. For example:

```
<pers:person xmlns:pers="http://sernaferna.com/pers">
  <pers:name>
    <pers:title>Sir</pers:title>
  </pers:name>
</pers:person>
```

Notice that the prefix is needed on both the start and end tags of the elements. They are no longer simply being identified by their names, but by their QNames.

*By now you have probably realized why colons in element names are so strongly discouraged in the XML 1.0 specification (and in this book). If you were to use a name that happened to have a colon in it with a namespace-aware XML parser, the parser would get confused, thinking that you were specifying a namespace prefix.*

Internally, when this document is parsed, the parser simply replaces any namespace prefixes with the namespace itself, creating a name much like the names we used earlier in the chapter. That is, internally a parser might consider `<pers:person>` to be similar to `<{http://sernaferna.com/pers}person>`. For this reason, the `{http://sernaferna.com/pers}person` notation is often used in namespace discussions to talk about **fully-qualified names**. Just remember that this is only for the benefit of easily discussing namespace issues, and is not valid XML syntax.

## Try It Out—Adding XML Namespaces to Our Document

Let's see what our document would look like with proper XML Namespaces. Luckily for us, there is already a namespace defined for XHTML, which is <http://www.w3.org/1999/xhtml>. We can use this namespace for the HTML we're embedding in our document.

1.

Open Notepad, and type in the following XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pers:person xmlns:pers="http://sernaferna.com/pers"
               xmlns:html="http://www.w3.org/1999/xhtml">

  <pers:name>
    <pers:title>Sir</pers:title>
    <pers:first>John</pers:first>
    <pers:middle>Fitzgerald Johansen</pers:middle>
    <pers:last>Doe</pers:last>
  </pers:name>
  <pers:position>Vice President of Marketing</pers:position>
  <pers:résumé>
    <html:html>
      <html:head><html:title>Resume of John Doe</html:title></html:head>
      <html:body>
        <html:h1>John Doe</html:h1>
        <html:p>John's a great guy, you know?</html:p>
      </html:body>
    </html:html>
  </pers:résumé>
</pers:person>
```

2.

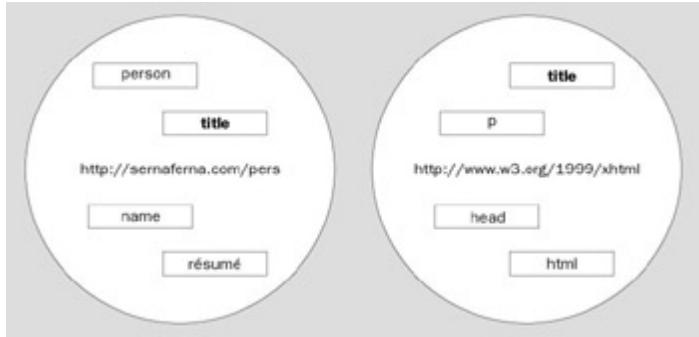
Save this document to your hard drive as namespace.xml.

3.

Open namespace.xml in IE 5 or later. You should get the normal color-coded view of your XML document, the same as you do for any other XML document in IE 5. (If you don't, go back and make sure you haven't made any mistakes!)

## How It Works

We now have a document with elements from two separate namespaces, which we defined in the highlighted code, and any namespace-aware XML parser will be able to tell them apart. (The fact that the file opens up fine in Internet Explorer indicates that the parser bundled with this browser understands namespaces properly; if it didn't, the document might raise errors instead.) The two namespaces now look more like this:



The `xmlns` attributes specify the namespace prefixes we are using to point to our two namespaces:

```
<pers:person xmlns:pers="http://sernaferna.com/pers"
               xmlns:html="http://www.w3.org/1999/xhtml">
```

That is, we declare the `pers` prefix, which will be used to specify elements that belong to the `pers` namespace, and the `html` prefix, which will be used to specify elements that belong to the XHTML namespace. However, remember that the prefixes themselves mean nothing to the XML parser, they get replaced with the URI internally. We could have used `pers` or `myprefix` or `blah` or any other legal string of characters for the prefix; it's only the URI to which they point that the parser cares about, although using descriptive prefixes is good practice.

Because we have a way of identifying which namespace each element belongs to, we don't have to give them special, unique names. We have two vocabularies, each containing a `<title>` element, and we can mix both of these `<title>`

elements in the same document. If I ever need a person's title, I can easily find any {<http://sernaferna.com/pers>} title elements I need, and ignore the {<http://www.w3.org/1999/xhtml>} title elements.

However, even though my <title> element is prefixed with a namespace prefix, the name of the element is still <title>. It's just that we have now declared what namespace that <title> belongs to, so that it won't be confused with other <title> elements which belong to other namespaces.

## Default Namespaces

Although the previous document solves all of our namespace-related problems, it's just a little bit ugly. We have to give every element in the document a prefix to specify which namespace this element belongs to, which makes the document look very similar to our first prefixed version. Luckily, we have the option of creating **default namespaces**.

### Important

A default namespace is exactly like a regular namespace, except that you don't have to specify a prefix for all of the elements that use it.

It looks like this:

```
<person xmlns="http://sernaferna.com/pers">
  <name>
    <title>Sir</title>
  </name>
</person>
```

Notice that the `xmlns` attribute no longer specifies a prefix name to use for this namespace. As this is a default namespace, this element, and any elements descended from it, belongs to this namespace, unless they explicitly specify another namespace. So the <name> and <title> elements both belong to this namespace. Note that these elements, since they don't use a prefix, are no longer called QNames, even though they are still universally unique. Many people use the generic term **universal name**, or **UName**, to describe a name in a namespace, whether it is a prefixed QName or a name in a default namespace.

You can declare more than one namespace for an element, but only one can be the default. This allows us to write XML like this:

```
<person xmlns="http://sernaferna.com/pers"
         xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <name/>
  <xhtml:p>This is XHTML</xhtml:p>
</person>
```

In this case all of the elements belong to the `http://sernaferna.com/pers` namespace, except for the <p> element, which is part of the `xhtml` namespace. (We've declared the namespaces and their prefixes, if applicable, in the root element so that all elements in the document can use these prefixes.) However, we can't write XML like this:

```
<person xmlns="http://sernaferna.com/pers"
         xmlns="http://www.w3.org/1999/xhtml">
```

This tries to declare two default namespaces. In this case, the XML parser wouldn't be able to figure out to what namespace the <person> element belongs. (Not to mention that this is a duplicate attribute, which, as we saw in [Chapter 2](#), is not allowed in XML.)

## Try It Out-Default Namespaces in Action

Let's rewrite our previous document, but use a default namespace to make it cleaner.

1.

Make the following changes to `namespace.xml` and save it as `namespace2.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<person xmlns="http://sernaferna.com/pers"
         xmlns:html="http://www.w3.org/1999/xhtml">
  <name>
    <title>Sir</title>
    <first>John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
  <position>Vice President of Marketing</position>
  <r sum >
    <html:html>
      <html:head><html:title>Resume of John Doe</html:title></html:head>
      <html:body>
        <html:h1>John Doe</html:h1>
        <html:p>John's a great guy, you know?</html:p>
      </html:body>
    </html:html>
  </r sum >
</person>
```

## How It Works

In the `<person>` start-tag, the first `xmlns` attribute doesn't specify a prefix to associate with this namespace, so this becomes the default namespace for the element, along with any of its descendants, which is why we don't need any namespace prefixes in many of the elements, such as `<name>`, `<title>`, etc.

But, since the XHTML elements are in a different namespace, we do need to specify the prefix for them, for example:

```
<html:head><html:title>Resume of John Doe</html:title></html:head>
```

## Declaring Namespaces on Descendants

So far, when we have had multiple namespaces in a document, we've been declaring them all in the root element, so that the prefixes are available throughout the document. So, in our previous Try It Out, we declared a default namespace, as well as a namespace prefix for our HTML elements, all on the `<person>` element.

This means that when we have a default namespace mixed with other namespaces, we would create a document like this:

```
<person xmlns="http://sernaferna.com/pers"
         xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <name/>
  <xhtml:p>This is XHTML</xhtml:p>
</person>
```

However, we don't have to declare all of our namespace prefixes on the root element; in fact, a namespace prefix can be declared on any element in the document. We could also have written the above like this:

```
<person xmlns="http://sernaferna.com/pers">
  <name/>
  <xhtml:p xmlns:xhtml="http://www.w3.org/1999/xhtml">
    This is XHTML</xhtml:p>
</person>
```

In some cases this might make our documents more readable, because we're declaring the namespaces closer to where they'll actually be used. The downside to writing our documents like this is that the `xhtml` prefix is only available on the `<p>` element and its descendants; we couldn't use it on our `<name>` element, for example, or any other element that wasn't a descendant of `<p>`.

But we can take things even further, and declare the XHTML namespace to be the *default* namespace for our `<p>` element and its descendants, like this:

```
<person xmlns="http://sernaferna.com/pers">
  <name/>
  <p xmlns="http://www.w3.org/1999/xhtml">This is XHTML</p>
</person>
```

Although `http://sernaferna.com/pers` is the default namespace for the document as a whole, <http://www.w3.org/1999/xhtml> is the default namespace for the `<p>` element, and any of its descendants. Again, in some cases this can make our documents more readable, since we are declaring the namespaces closer to where they are used.

## Try It Out-Default Namespaces for Children

In the interest of readability, let's write the XML from our previous Try It Out again, to declare the default namespace for the `<html>` tag and its descendants.

Here are the changes to be made to `namespace2.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<person xmlns="http://sernaferna.com/pers">
  <name>
    <title>Sir</title>
    <first>John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
  <position>Vice President of Marketing</position>
  <r sum >
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head><title>Resume of John Doe</title></head>
      <body>
        <h1>John Doe</h1>
        <p>John's a great guy, you know?</p>
      </body>
    </html>
  </r sum >
</person>
```

Save this as `namespace3.xml`. This looks a lot tidier than the previous version, and represents the same thing.

## Cancelling Default Namespaces

Sometimes you might be working with XML documents in which not all of the elements belong to a namespace. For example, you might be creating XML documents to describe employees in your organization, and those documents might include occasional XHTML comments about the employees, such as in the following short fragment:

```
<employee>
  <name>Jane Doe</name>
  <notes>
    <p xmlns="http://www.w3.org/1999/xhtml">I've worked
       with <name>Jane Doe</name> for over a <em>year</em>
       now.</p>
  </notes>
</employee>
```

In this case, we have decided that anywhere the employee's name is included in the document it should be in a `<name>` element, in case the employee changes his/her name in the future, such as if Jane Doe ever gets married. (In this case changing the document would then be a simple matter of looking for all `<name>` elements that aren't in a namespace, and changing the values.) Also, since these XML documents will only ever be used by our own application, we don't have to create a namespace for it.

However, as you see above, one of the `<name>` elements occurs under the `<p>` element, which declares a default namespace, meaning that the `<name>` element also falls under that namespace. So if we searched for `<name>` elements that had no associated namespace, we wouldn't pick this one up. The way to get around this is to use the `xmlns` attribute to *cancel* the default namespace, by setting the value to an empty string. For example:

```
<employee>
<name>Jane Doe</name>
<notes>
  <p xmlns="http://www.w3.org/1999/xhtml">I've worked
  with <name xmlns="">Jane Doe</name> for over a <em>year</em>
  now.</p>
</notes>
</employee>
```

Now the second `<name>` element is not in any namespace. Of course, if we had a namespace specifically for our `<employee>` document, this would become a non-issue, because we could just use the ways we've already learned to declare that an element is part of that namespace (using a namespace prefix, or a default namespace). But, in this case, we're not declaring that the element is part of a namespace—we're trying to declare that it's *not* part of any namespace, which is the opposite of what we've been doing so far.

## Do Different Notations Make Any Difference?

We've now seen three different ways to combine elements from different namespaces. We can fully qualify every name, like this:

```
<pers:person xmlns:pers="http://sernaferna.com/pers"
  xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <pers:name/>
  <xhtml:p>This is XHTML</xhtml:p>
</pers:person>
```

Or, we can use one namespace as the default, and just qualify any names from other namespaces, like this:

```
<person xmlns="http://sernaferna.com/pers"
  xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <name/>
  <xhtml:p>This is XHTML</xhtml:p>
</person>
```

Or, we can just use defaults everywhere, like this:

```
<person xmlns="http://sernaferna.com/pers">
  <name/>
  <p xmlns="http://www.w3.org/1999/xhtml">This is XHTML</p>
</person>
```

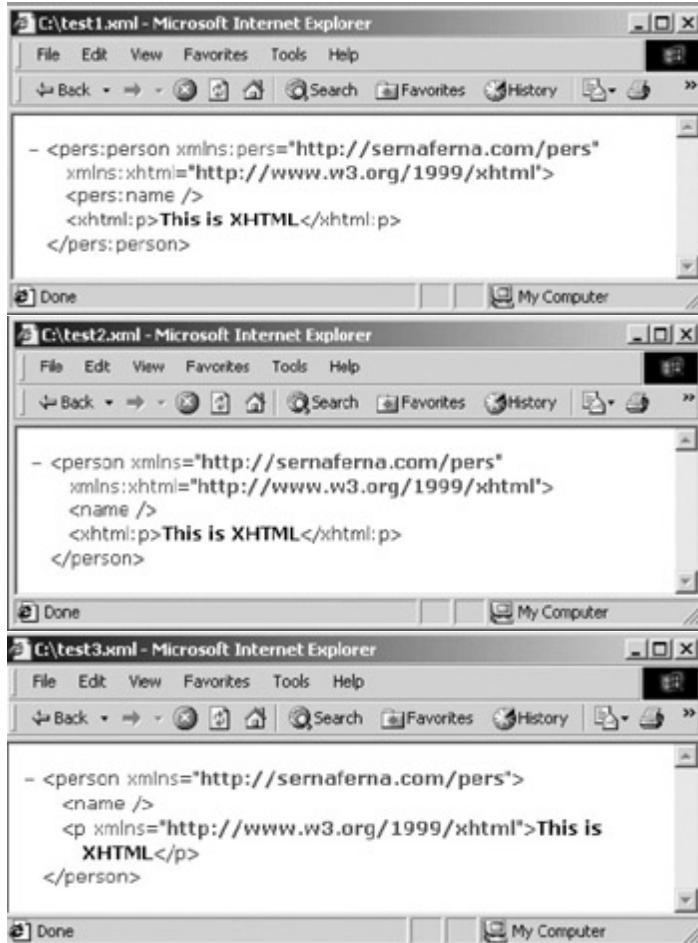
This begs the question: do these three fragments of XML really mean exactly the same thing?

From the pure namespaces point of view, yes, these documents mean exactly the same thing. All three documents

have the same three elements, and in each instance, each element still belongs to the same namespace as it does in the other two instances.

From the point of view of most applications, these fragments also mean the same thing. When you're doing work with an XML document, you usually only care what elements you're dealing with; you don't care whether the element's namespace was declared using a default declaration or an explicit prefix, any more than you care if an element with no data was written as a start-tag and an end-tag pair or as an empty element.

There are, however, some applications that actually do differentiate between the three examples above, such as an application that reads in XML and displays the sourcecode to a user. As you may have noticed if you used IE 5 or later to view the XML from the previous Try It Outs, it does display each one differently. Let's look at the three examples above:



As you can see, the browser displays the documents exactly as they were written; so if we declare our namespaces using defaults; the browser displays them using defaults; if we declare them with prefixes, the browser displays them with prefixes.

The two dominant technologies to programmatically get information out of XML documents, the **Document Object Model (DOM)** and **Simple API for XML (SAX)**, which are covered in later chapters, provide methods which allow you to get not only the namespace URI for a QName but also the prefix for those applications which need the prefix. This means that you can not only find the fully-qualified namespace names for these elements, but you can go so far as to see *how* the XML author wrote those names. In real life, however, you will hardly ever need the namespace prefix, unless you are writing applications to display the XML as entered to a user. Internet Explorer's default XSL style sheet can differentiate between the above cases because it pulls this information from the DOM implementation shipped with the browser.

## Namespaces and Attributes

So far all of our discussions have been centered on elements, and we've been pretty much ignoring attributes. Do

namespaces work the same for attributes as they do for elements?

The answer is no, they don't. In fact, attributes usually don't have namespaces, the way that elements do. They are just "associated" with the elements to which they belong. Consider the following fragment:

```
<person xmlns="http://sernaferna.com/pers">
  <name id="25">
    <title>Sir</title>
  </name>
</person>
```

We know that the `<person>`, `<name>`, and `<title>` elements all belong to the same namespace, which is declared in the `<person>` start-tag. The `id` attribute, on the other hand, is not part of this namespace; it's simply associated with the `<name>` element, which itself is part of that default namespace. We could use a notation like this to identify it, for discussion:

```
"{http://sernaferna.com/pers}name:id"
```

However, if we used prefixes, we could specify that `id` is in the namespace like so:

```
<a:person xmlns:a="http://sernaferna.com/pers">
  <a:name a:id="25">
    <a:title>Sir</a:title>
  </a:name>
</a:person>
```

Unfortunately, there is a bit of a gray area left by the Namespaces specification concerning attributes. For example, consider the following two fragments:

```
<a:name id="25">
<a:name a:id="25">
```

Are these two fragments identical? Or are they different? Well, actually, programmers can make up their own minds whether these two cases are the same or different. (In XSLT, for example, the two cases would be considered to be different.) For this reason, if you need to be sure that the XML processor will realize that your attributes are part of the same namespace as your element, you should include the prefix. On the other hand, most applications will treat the two situations identically.

Consider the case where you want to perform some processing on every attribute in the `http://sernaferna.com/pers` namespace. If an application considers both of the above cases to be the same, then in both cases the `id` attribute will get processed. On the other hand, if the application doesn't consider both of the above to be the same, you'll only get the second `id` attribute, because it is specifically declared to be in the namespace we're looking for, whereas the first one isn't.

Is this purely theoretical? Well, yes, it is. In most cases, applications don't look for attributes on their own; they look for particular elements, and then process the attributes on those elements.

However, attributes from a particular namespace can also be attached to elements from a different namespace. Attributes that are specifically declared to be in a namespace are called **global attributes**. A common example of a global attribute is the XHTML `class` attribute, which might be used on any XML element, XHTML or not. This would make things easier when using CSS to display an XML document.

## Try It Out-Adding Attributes

To see this in action, let's add an `id` attribute to our `<name>` element, as well as adding a `style` attribute to the HTML paragraph portion of our résumé.

Change namespace2.xml to this, namespace4.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<person xmlns="http://sernaferna.com/pers">
  <name id="1">
    <title>Sir</title>
    <first>John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
  <position>Vice President of Marketing</position>
  <r sum >
    <html:html xmlns:html="http://www.w3.org/1999/xhtml">
      <html:head><html:title>Resume of John Doe</html:title></html:head>
      <html:body>
        <html:h1>John Doe</html:h1>
        <html:p html:style="FONT-FAMILY: Arial">
          John's a great guy, you know?
        </html:p>
      </html:body>
    </html:html>
  </r sum >
</person>
```

Because we want the style attribute to be specifically in the XHTML namespace, we have gone back to using prefixes on our XHTML elements, instead of a default namespace. Another alternative would have been to declare the XHTML namespace twice: once as the default, for `<html>` and all of its descendants, and once with a prefix, which could be attached to the attribute.

## How It Works

The `id` attribute that we added is associated with the `<name>` element, but it doesn't actually have a namespace.

Similarly, the `style` attribute is associated with the `<p>` element, but in this case the attribute is specifically in the XHTML namespace.

Again, applications may or may not treat both of these the same, and consider them to be in the same namespace as the elements to which they are attached. All applications will treat the `style` attribute as being in the XHTML namespace, because we have specifically said so, but some will think `id` is in the same namespace as `<name>`, and some won't.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

◀ PREVIOUS

< Free Open Study >

NEXT ▶

# What Exactly are URIs?

We have mentioned that namespaces are specified using URIs, and most of the examples shown so far have been URLs. To really understand namespaces, we'll have to look at this concept a little further.

Because so much of the work done on the Internet somehow involves finding and retrieving **resources**, much thought has been put into this process. So what is a resource? Well, simply put, a **resource** is anything that has identity. It could be a tangible item, such as a .gif file or a book, or it could be a conceptual item, like the current state of the traffic in Toronto. It could be an item which is retrievable over the Internet, such as an HTML document, or an item which is not retrievable over the Internet, such as the actual person who wrote that HTML document.

Recall our earlier definition of a URI:

Important

A URI (Uniform Resource Identifier) is a string of characters that identifies a resource. It can come in one of two flavors: URL (Uniform Resource Locator), or URN (Universal Resource Name).

*There is a document which formally describes the syntax for URIs at the IETF (Internet Engineering Task Force) web site, located at <http://www.ietf.org/rfc/rfc2396.txt>; one which describes the syntax for URNs, located at <http://www.ietf.org/rfc/rfc2141.txt>; and one which describes the syntax for URLs, located at <http://www.ietf.org/rfc/rfc1738.txt>.*

## URLs

If you have been on the Internet for any length of time, you are probably already familiar with URLs?and most Internet-savvy people understand how URLs work. The first part of the URL specifies the **protocol**; http being the most common, with mailto and ftp also being used frequently, and others (such as gopher, news, telnet, file, etc.) being used on occasion. (Officially, the protocol part of the URL is called a **scheme**.)

The protocol is followed by a colon, and after the colon comes a path to the resource being identified.

For example, here's a URL to a web page on the Internet:

`http://sernaferna.com/default/home.htm`

This URL contains information that can be used to retrieve a file named home.htm from a server on the Internet named sernaferna.com. It specifies that the file is in the default directory (or virtual directory), and that the file should be retrieved via the HTTP protocol.

We can also create a URL to an e-mail account, like so:

`mailto:someone@somewhere.com`

Of course, there is a limitation on the resources that can be retrieved via URLs: obviously they must be resources of a type that is retrievable from a computer!

## URNs

URNs are not as commonly seen as URLs. In fact, most people, even those who have been on the Internet for years, have never seen a URN. They exist to provide a persistent, location-independent name for a resource.

For example, a person's name is similar to a URN, because the person has the same name, no matter where they are. Even after a person dies, the name still refers to the person who used to have it when they were alive. A name is different from a URN, though, because more than one person can have the same name, whereas URNs are designed to be unique across time and space.

A URN looks something like this:

```
urn:foo:a123,456
```

First comes the string `urn`, upper-or lowercase, and a colon. After the first colon comes the **Namespace Identifier**, or **NID**, (foo in this case) followed by another colon. And finally comes the **Namespace Specific String**, or **NSS** (a123,456 for example). As you can see from the terminology, URNs were designed with namespaces already in mind.

The NID portion of the URN declares what type of URN this is. For example, to create URNs for Canadian citizens, we might declare an NID of `Canadian-Citizen`.

The NSS portion of the URN is the part that must be unique, and persistent. In Canada, all citizens are assigned unique Social Insurance Numbers. So, a URN for a Canadian citizen with a Social Insurance Number of 000-000-000 might look like this:

```
urn:Canadian-Citizen:000-000-000
```

## Why Use URLs for Namespaces, Not URNs?

The XML Namespace specification states that namespaces are identified with URIs, which leaves us the possibility of using either URLs or URNs. It seems that URNs are better suited for naming namespaces than URLs?after all, a namespace is a *conceptual* resource, not one that can be retrieved via the Internet; so why are most namespaces named using URLs instead?

Some people find it easier to create unique namespace names using URLs, since they are already guaranteed to be unique. If I own the `sernaferna.com` domain name, I can incorporate it into my namespace names, and know that they will be unique.

Of course, this is still by convention; nothing stops someone at another company, say `Ferna Serna Inc.`, from stealing `Serna Ferna Inc.`'s domain name and maliciously using it as the name for a namespace. But if everyone follows the convention then we can be sure that there won't be *accidental* collisions, which is good enough for our purposes. You could still construct a URN like `urn:SernaFernaHR.name`, but many people feel that things are just simpler if you use URLs.

And there can also be side benefits to using URLs as namespace names. If we wanted to, we could put a document at the end of the URL that describes the elements in that namespace. For example, we have been using `http://sernaferna.com/pers` as a fictional namespace. If `Serna Ferna Inc.` wanted to make the `pers` namespace public, for use in public document types, it might put a document at that location which describes the various XML elements and attributes in that namespace.

But regardless of what people are doing, the possibility of using a URN as a namespace identifier still exists, so if you have a system of URNs that you feel is unique, it is perfectly legal. URNs provide no benefits over URLs, except for the conceptual idea that they're a closer fit to what namespace names are trying to do. (That is, *name* something, not

point to something.)

## What Do Namespace URIs Really Mean?

Now that we know how to use namespaces to keep our element names unique, what exactly do those namespace URIs mean: in other words what does <http://sernaferna.com/pers> really represent?

The answer, according to the XML Namespaces specification, is that it doesn't mean anything. The URI is simply used to give the namespace a name, but doesn't mean anything on its own. In the same way the words "John Doe" don't mean anything on their own?they are just used to identify a particular person.

Many people feel that this isn't enough for XML. In addition to keeping element names distinct, they would also like to give those elements meaning?that is, not just distinguish <my:element> from <your:element>, but also define what <my:element> means. What is it used for? What are the legal values for it? If we could create some kind of "schema" which would define our document type, the namespace URI might be the logical place to declare this document as adhering to that schema.

The XML Namespaces specification (<http://www.w3.org/TR/REC-xml-names/>) states that "it is not a goal that [the namespace URI] be directly useable for retrieval of a schema (if any exists)." (A **schema** is a document that formally describes an XML document type. There are a number of languages available for creating schemas, such as **DTDs** and the **XML Schema** language from the W3C, which will be covered in later chapters.) In other words, as we've been saying, the URI is just a name or identifier: it doesn't have any kind of meaning. However, it is not strictly forbidden for it to have a meaning. For this reason, someone creating an application could legally decide that the URI used in a namespace actually does indicate some type of documentation, whether that is a prose document describing this particular document type, or a technical schema document of some sort. But, in this case, the URI still wouldn't mean anything to the XML parser; it would be up to the higher-level application to read the URI and do something with it.

As an example of where this might be useful, consider a corporate information processing system where users are entering information to be stored in XML format. If different namespaces are defined for different types of documents, and those namespaces are named with URLs, then you could put a help file at the end of each URL. If users are viewing a particular type of XML document in the special application you have written for them, all they have to do is hit **F1** to get help, and find out about this particular type of document. All your application has to do is open a web browser, and point it to the URL that defines the namespace.

## RDDL

So, in addition to providing human-readable documentation for your namespace, there also exist the options of providing schemas. However, there are a number of these languages available (a few of which are covered in this book)?how do we decide what to put at the end of a URL we use for a namespace name? Do we put human-readable documentation, which describes the namespace? Or do we put a document there in one of these machine-readable formats? One answer is to use the **Resource Directory Description Language**, or **RDDL** (the RDDL specification can be found at <http://www.openhealth.org/RDDL/>).

RDDL was created to combine the benefits of human-readable documentation with the benefits of providing machine-readable documentation for an XML Namespace. An RDDL document is actually an XHTML document, which makes it human-readable. However, since XHTML is XML, other machine-readable resources can be included in the document, using a technology called [XLink](#) to link the various documents together. (XLink will be covered in [Chapter 13, Linking XML](#).) In this way, human-readable documentation can be provided on an XML Namespace, while at the same time providing links to as many other resources as needed, such as machine-readable documents on the namespace, executable code, etc.

# When Should I Use Namespaces?

By this point in the chapter, we've covered everything that we need to know about namespaces from a technical standpoint. We know what they mean, how to use them, and how to combine them. So let's sit back for a while, put our feet up, and talk philosophy. When should we create a new namespace, and when should we add new elements to an existing one?

In the course of this chapter, we have created three namespaces:

- <http://sernaferna.com/pers>
- <http://sernaferna.com/locator>
- <http://sernaferna.com/name>

All of these namespaces were for use by our fictional company called Serna Ferna, Inc. We could instead have created one namespace, which the company would use for all of its XML document types. Or, conversely, we could have broken down our namespaces further, perhaps splitting the `<person>` document type into multiple namespaces. Why did we choose three?

Remember that a namespace is just a "bag of names": that is, it's a group of element names that belong together, and that are distinct from element names in other namespaces. This is shown in the following diagram:



The key is the phrase "belong together". You might think of the elements in a namespace as being the vocabulary for a language, the same way that English words are in the English vocabulary. Any words that belong to that language would go in that namespace, and words from other languages would go into other namespaces. It's up to you to decide which elements belong in the same vocabulary, and which ones should go in different vocabularies.

*The W3C went through this process when creating XHTML, the HTML language "redone" in XML. The problem is that XHTML is based on HTML4, which has three flavors: **Frameset** (which includes support for HTML frames), **Strict** (which is designed for clean structural markup, free from all layout tags), and **Transitional** (which allows formatting markup for older browsers, such as a bgcolor attribute on the `<body>` tag). Some HTML elements, such as `<p>`, appear in all three flavors, while others, such as `<frameset>`, may only appear in certain flavors.*

*This led the W3C, in the initial specifications for XHTML, to indicate that there would be three different namespaces used, one for each flavor. However, the web community strongly disagreed with this approach. Most people consider HTML (or XHTML) to be one language—even though there may be more than one "flavor" or "dialect" of that language—so they argued that XHTML should only have one namespace associated with it. In the end, the W3C decided to go with the one-namespace approach (the namespace they chose is <http://www.w3.org/1999/xhtml>), which is why we've been using it for our XHTML examples).*

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

This chapter introduced the concept of namespaces, along with their implementation in XML. We've seen:

- What benefit namespaces can potentially give us in our documents
- How to declare and use namespaces
- How to effectively use a URI as the name of a namespace

The idea behind namespaces may not seem all that relevant, unless you're combining elements from various vocabularies into one XML document. You may be thinking, "If I'm just going to create XML documents to describe my data, why mess around with all of this namespace stuff?" However, when you remember that you will be using other XML vocabularies, such as XSLT to transform your documents or XHTML to display your data, namespaces become much more relevant. Learning the concepts behind namespaces will help you combine your documents with these other document types, in addition to any further document types you may create yourself.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 4: XSLT

## Overview

So far in the book we've learned how to create well-formed XML data with a hierarchical structure. We've chosen that structure for our data, but what if we want to change that structure to suit an application or tailor our data for its users? Enter **Extensible Stylesheet Language Transformations**, **XSLT**: a language which can transform XML documents into any text-based format, XML or otherwise.

As we'll see later, XSLT is a sub-component of a larger language called XSL. XSLT is a powerful tool for e-commerce, as well as any other place where XML might be used.

As an example, suppose we're going to be running a news-oriented web site. In addition to our site itself, we'll also want to provide a way for our readers to get individual news stories from us in XML. In other words, people can use the news stories we provide in their own applications, for whatever purposes they wish; some might want to display the stories on their own web sites, others might want to put the data into their own back-end databases for some other application.

However, when we're deciding on what structure to give the XML for our news stories, we probably won't be thinking about those other uses people have for our XML; in fact, it would be impossible to guess all of the different things that people will want to do with our data. Instead, we'll be concentrating on inventing element names and a structure that properly describe a news story.

This is where XSLT comes in. Once others have received our XML documents, they can use this transformation language to transform these documents into whatever other format they wish—HTML for display on their web sites, a different XML-based structure for other applications, or even just regular text files.

XSL relies on finding parts of an XML document that match a series of predefined templates, and then applying transformation and formatting rules to each matched part. Another language, [XPath](#), is used to help in finding those matching parts.

In this chapter we'll learn:

- What XSL, XSLT, and XPath are
- How XSLT, a **declarative** language, differs from **imperative** programming languages, like JavaScript
- What templates are, and how they are used
- How to address the innards of an XML document using XPath
- The main XSLT elements, and how to use them

◀ PREVIOUS

[< Free Open Study >](#)

NEXT ▶

# Running the Examples

In order to perform an XSLT transformation, you need at least three things: an XML document to transform, an **XSLT stylesheet**, and an **XSLT engine**.

- - The stylesheet contains the instructions for the transformation you want to perform; the "sourcecode". This is the part we'll write, and it's what this chapter is concerned with.
  - - The engine is the software that will carry out the instructions in your stylesheet. There are a number of engines available, many of which are free, such as:
      - Saxon, an opensource XSLT engine written in Java, available at <http://saxon.sourceforge.net>.
      - Xalan, which is also an opensource XSLT engine, with both C++ and Java versions offered, available at <http://xml.apache.org/xalan-c/index.html> and <http://xml.apache.org/xalan-j/index.html>, respectively.
      - MSXML, the XML parser that ships with Microsoft Internet Explorer, and is also available separately at <http://msdn.microsoft.com/xml>. This is the engine we'll be using for the Try It Outs in this chapter.

## MSXML

**MSXML**, the XML parser that ships with Internet Explorer 5, includes an XSLT engine. However, IE 5 shipped before the XSLT Recommendation was finished, so the preview version of XSLT that MSXML understands is different from the Recommended XSLT. (To avoid confusion, many people call this preview version of the language "**WD-xsl**", instead of "XSLT". "WD" is short for "Working Draft", since this language is based on an earlier working draft, instead of the full XSLT Recommendation.) After the release of IE 5, Microsoft kept releasing newer versions of MSXML, which included increasing support for the XSLT recommendation, as well as some of the other W3C XML-related Recommendations.

The examples in this chapter were tested with **MSXML 3 SP1**, available at <http://msdn.microsoft.com/xml>. To run the examples in this chapter, you should make sure that you have this version of MSXML, or a later one.

MSXML 3 will not replace your existing MSXML parser unless you explicitly tell it to; instead, it will run "side-by-side" with the old version of MSXML, allowing you to use either one. This would be handy if you have already done development using the WD-xsl, and still want your old applications to work, however, that's probably not the case for most people. If you would like to use the new version exclusively, Microsoft provides a tool, `xmllist.exe`, to let you run MSXML 3 in **Replace Mode**, meaning that it replaces the previous version of MSXML shipped with IE 5. It is also available from the MSDN URL above. For the examples in this book, and indeed your day-to-day work, you should install MSXML in Replace Mode, and ignore the older WD-xsl syntax. (Indeed, if you are using MSXML 4 or later, you don't have the option of using the older syntax, since support for it is removed from the parser.)

*Note that IE 6 ships with MSXML 3 already installed in Replace Mode.*

In addition, Microsoft also provides a tool to run MSXML's XSLT engine from the command line, called **MSXSL**.

Again, it can be downloaded from the same MSDN link given above. MSXSL is run like this:

```
msxsl source stylesheet [-o output]
```

Where *source* is the XML document you want to transform, *stylesheet* is the XSLT stylesheet you are using, and *output* is the optional file you want to create, to contain the results of the transformation. If no output file is specified, the results are simply printed on the screen.

MSXSL is just a wrapper for the MSXML parser, which uses its XSLT processing capabilities. Any time that this chapter refers to MSXSL, remember that it's actually the parser/XSLT processor, MSXML, which is doing the work.

You might want to download MSXSL to a common directory, where you will be storing your XSLT stylesheets, such as C:\XSLT. You should also consider adding this directory to your PATH, so that you can easily run the utility from anywhere on your machine.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# What is XSL?

**Extensible Stylesheet Language**, as the name implies, is an XML-based language used to create **stylesheets**. An XSL engine uses these stylesheets to transform XML documents into other document types, and to format the output. In these stylesheets you define the layout of the output document, and where to get the data from within the input document. That is, "retrieve data from this place in the input document; make it look like this in the output". In XSL parlance, the input document is called the **source tree**, and the output document the **result tree**.

It is important to remember that, because XSL is an XML-based language, every XSL stylesheet must be a well-formed XML document. This is a common source of confusion for novice XSLT programmers. For example, if you wanted to create a simple "hello world" stylesheet, which created an HTML document, you might do it something like this:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
  <p>Hello World!<br></p>
</html>
</xsl:template>
</xsl:stylesheet>
```

We'll look at what this XSLT syntax means throughout the chapter, but it simply outputs an HTML document, that says "Hello World!". However, even though the HTML syntax used is perfectly acceptable in a browser, this is not a valid XSLT stylesheet, since it isn't well-formed XML—the `<br>` tag is not closed. (To fix this problem, you would probably change the `<br>` tag to an XML empty element, such as `<br/>`.)

There are actually two complete languages under the XSL umbrella:

- 

- a transformation language, which is named **XSL Transformations**, or **XSLT**
- a language used to format XML documents for display, **XSL Formatting Objects**, or **XSL-FO**

Of course, the two languages can be used together, so that XSLT transforms the data, and XSL Formatting Objects then further modifies the data for display, much like Cascading Style Sheets, which will be covered in [Chapter 11, Displaying XML](#).

*XSLT became a W3C Recommendation in November of 1999; the specification can be found at <http://www.w3.org/TR/xslt>. XSL Formatting Objects was still being worked on at the time of writing, but the latest version of the specification can be found at <http://www.w3.org/TR/xsl/>.*

This chapter will focus on XSLT. Most XSL processors only implement the XSLT half of XSL anyway; XSLT is designed to be self-standing, meaning that developers can write XSLT engines, even if they don't want to implement the XSL Formatting Objects functionality.

Note that the output from XSLT doesn't have to be well-formed XML—it can also output HTML, or even plain text. In fact, one of the most useful aspects of XSLT is for transforming XML documents into HTML documents, for display in a browser.

## Why is XSLT So Important for e-Commerce?

To get an idea of the power of XSLT, let's create a fictional example to demonstrate. Imagine that there are two companies working together, sharing information over the Internet. *Mom And Pop* is a store, which sends purchase orders to *Frank's Distribution*, which fulfills those orders. Let's assume that we have already decided that XML is the best way to send that information.

Unfortunately, the chances are that *Mom And Pop* will need a different set of information for this transaction from *Frank's Distribution*. Perhaps *Mom And Pop* needs information on the salesperson that made the order for commissioning purposes, but *Frank's Distribution* doesn't need it. On the other hand, *Frank's Distribution* needs to know the part numbers, which *Mom And Pop* doesn't really care about.

*Mom And Pop* uses XML such as the following:

```
<?xml version="1.0"?>
<order>
  <salesperson>John Doe</salesperson>
  <item>Production-Class Widget</item>
  <quantity>16</quantity>
  <date>
    <month>1</month>
    <day>13</day>
    <year>2000</year>
  </date>
  <customer>Sally Finkelstein</customer>
</order>
```

Whereas *Frank's Distribution* requires XML that looks more like this:

```
<?xml version="1.0"?>
<order>
  <date>2000/1/13</date>
  <customer>Company A</customer>
  <item>
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

In this scenario, we have three choices:

- 

*Mom And Pop* can use the same structure for its XML that *Frank's Distribution* uses. The disadvantage is that it now needs a separate XML document to accompany the first one, for its own additional information. However, the business-to-business (B2B) communication is much easier, since both companies are using the same format.

- 

*Frank's Distribution* can use the same format for its XML that *Mom And Pop* uses. This would have the same results as the previous choice.

- 

Both companies can use whatever XML format they wish internally, but transform their data to a common format whenever they need to transmit the information outside. Obviously this option provides the most flexibility for both companies, and still allows cross-company communication.

For the last option, both companies would probably agree on this common format in advance. Then, whenever *Mom and Pop* wanted to send information to *Frank's Distribution*, it would first transform it XML to this common format, and then send it. Similarly, whenever *Frank's Distribution* wanted to send information to *Mom and Pop*, it would first transform its internal XML documents to this common format, and then send it. In this way, any time either company receives information from the other, it will be in a common format.

With XSLT, this kind of transformation is quite easy: it's probably one of the most important uses of XSLT, and one of the more exciting areas where XML is already making its presence felt.

## Try It Out-A Simple Business-to-Business Example

I claimed the transformation would be easy, so I guess I'd better put my money where my mouth is, and create a stylesheet that can do it!

1.

Open Notepad, and type in the XML for *Mom and Pop's* data:

```
<?xml version="1.0"?>
<order>
  <salesperson>John Doe</salesperson>
  <item>Production-Class Widget</item>
  <quantity>16</quantity>
  <date>
    <month>1</month>
    <day>13</day>
    <year>2000</year>
  </date>
  <customer>Sally Finkelstein</customer>
</order>
```

Save this as MomAndPop.xml.

2.

Now open a new file and type in the following XSLT, which contains the instructions for the transformation:

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

<xsl:template match="/">
<order>
  <date>
    <xsl:value-of select="order/date/year"/>/<xsl:value-of
      select="order/date/month"/>/<xsl:value-of select="order/date/day"/>
  </date>
  <customer>Company A</customer>
  <item>
    <xsl:apply-templates select="order/item"/>
    <quantity><xsl:value-of select="order/quantity"/></quantity>
  </item>
</order>
</xsl:template>

<xsl:template match="item">
<part-number>
  <xsl:choose>
    <xsl:when test=". = 'Production-Class Widget'">E16-25A</xsl:when>
    <xsl:when test=". = 'Economy-Class Widget'">E16-25B</xsl:when>
```

```
<!--other part-numbers would go here-->
<xsl:otherwise>00</xsl:otherwise>
</xsl:choose>
</part-number>
<description><xsl:value-of select=". "/></description>
</xsl:template>
</xsl:stylesheet>
```

Save this as order.xsl.

Looks like a big jumble of nonsense, right? Don't worry: by the end of this chapter this stylesheet will look pretty simple. For the time being, just note that this stylesheet is written in XML.

3.

In order to perform the transformation, we'll get our first use of MSXSL. Open a command prompt, and find the directory where you downloaded this utility.

4.

Type in msxsl MomAndPop.xml order.xsl, specifying the full path to the XML and XSLT files if they are in a different directory from MSXSL. For example,

```
C:\XSLT\msxsl C:\input\MomAndPop.xml C:\input\order.xsl
```

Your output should look like this:

```
>msxsl MomAndPop.xml order.xsl
<?xml version="1.0" encoding="UTF-8"?>
<order>
<date>2000/1/13</date>
<customer>Company A</customer>
<item>
<part-number>E16-25A</part-number>
<description>Production-Class Widget</description>
<quantity>16</quantity>
</item>
</order>
```

which, you may notice, is the same as the *Frank's Distribution* XML document we saw earlier, with the slight addition of an encoding attribute in the XML declaration. *Mom And Pop* now has an XML document type that suits its business needs, but also has the ability to send the same information to *Frank's Distribution* in a format that is understandable to it.

5.

Now type in msxsl MomAndPop.xml order.xsl -o Franks.xml, again specifying the full path to the XML and XSLT files, if need be. This time, because we have used the -o switch (for "output"), MSXSL won't send any output to the screen; instead, it will create a file called Franks.xml, which will contain the results of the transformation. Open it up in Notepad or in IE 5 or later to verify that the results are the same.

## How XSLT Stylesheets Work-Templates

As mentioned earlier, XSLT stylesheets are XML documents. XSLT processors act on any XML elements that are in the XSLT namespace (which is <http://www.w3.org/1999/XSL/Transform>), and any other elements in the document, regardless of their namespace, are added to the result-tree as-is.

By convention, most people use the prefix "xsl" for elements in the XSLT namespace. However, as we learned in the [previous chapter](#), you can use whatever prefix you want in your own stylesheets.

*In the interest of brevity, the XSLT namespace declaration will be omitted for many of the examples in this chapter. Any time the "xsl" prefix is used, the XSLT namespace will be assumed.*

XSLT stylesheets are built on structures called **templates**. A template specifies what to look for in the source tree, and what to put into the result tree. For example:

```
<xsl:template match="quantity">quantity tag found!</xsl:template>
```

Templates are defined using the XSLT `<xsl:template>` element. There are two important pieces to this template: the `match` attribute, and the contents of the template.

The `match` attribute specifies a pattern in the source tree; this template will be applied for any nodes in the source tree that match that pattern. In this case, the template will be applied for any elements named "quantity".

Everything inside the template, between the start- and end-tags, is what will be output to the result tree. Any elements in the XSLT namespace are special XSLT elements, which indicate to the processor that it should do some work, while any other elements or text which appear inside the `<xsl:template>` element will end up in the output document as-is.

The contents of a template can be-and almost always are-much more complex than simply outputting text. There are numerous XSLT elements you can insert into the template to perform various actions. For example, there's an `<xsl:value-of>` element, which can take information from the source tree, and add it to the result tree. The following will work the same as our previous template, but instead of outputting "quantity tag found!" this template will add the contents of any elements named "quantity" to the result tree:

```
<xsl:template match="quantity"><xsl:value-of select=". . ."/></xsl:template>
```

In other words, if the XSLT engine finds:

```
<first>John</first>
```

it will output John, and if it finds:

```
<first>Andrea</first>
```

it will output Andrea.

*We'll take a closer look at the `<xsl:template>` and `<xsl:value-of>` elements later on.*

## Associating Stylesheets with XML Documents Using Processing Instructions

In many, if not most, cases XSLT transformations will be performed by passing an XML document and an XSLT stylesheet to an XSLT processor. However, an XSLT stylesheet can also be associated with an XML document using a processing instruction like this:

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

In this case, when a browser that understands XML and XSLT loads the XML document, it will automatically do the XSLT transformation for us, and display the results.

However, since IE 5 or greater is currently the only major web browser that understands XML and XSL, this will only work for IE 5. Your stylesheets would have to use the WD-xsl syntax for XSLT-newer stylesheets won't be understood by IE 5's XSLT engine. Or, better yet, you can run MSXML in Replace Mode, as we have done above. (However, this means that any users of your application would also have to be running MSXML 3 in Replace Mode.)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Imperative Versus Declarative Programming

Before we go on to study XSLT in depth, we should stop here and discuss the type of programming you'll be doing with XSLT.

There are two classifications of programming languages in the computing world: **imperative** and **declarative**.

## Imperative Programming

Imperative programming languages are ones like JavaScript, Java, or C++, where the programmer tells the computer exactly what to do, and how to do it. For example, if we had three strings, and wanted to create some HTML using those strings for paragraphs, we might write a function like the following in JavaScript:

```
function createHTML()
{
    //our three strings
    var aParagraphs = new Array("Para 1", "Para 2", "Para 3");
    var sOutput, sTemp, i;

    //create the opening HTML and BODY tags
    sOutput = new String("<html>\n<body>\n");

    for(i = 0; i < aParagraphs.length; i++)
    {
        //add a new paragraph with this string
        sTemp = new String("<p>" + aParagraphs[i] + "</p>\n");
        sOutput = sOutput + sTemp;
    }

    //add the closing HTML and BODY tags
    sOutput = sOutput + "</body>\n</html>";
    return sOutput;
}
```

This returns the following HTML:

```
<html>
<body>
<p>Para 1</p>
<p>Para 2</p>
<p>Para 3</p>
</body>
</html>
```

which is what we wanted. But in order to create the HTML we wanted with JavaScript, we had to instruct the computer to do all of the following:

- Create the `<html>` and `<body>` start-tags.
- Loop through all of the strings. For each string, create a `<p>` start-tag, add the string, and append a `</p>` end-tag. Then add this string to the main string.
-

Create the </body> and </html> end-tags.

In between these steps we also had to tell the computer to add whatever new lines we needed ("n" is a new line in JavaScript), and we had to make sure we did everything in the proper order. (We need to have the <html> start-tag before the <p> elements, for example.)

We had to do all of this work because JavaScript doesn't understand the concept of "HTML"; it only knows about strings. The fact that the string we created is HTML is over JavaScript's head.

## Declarative Programming

XSLT is not an imperative programming language like JavaScript, it's **declarative**, and declarative languages don't require quite so much work from the developer. With XSLT, we don't specify to the computer *how* we want anything done, just *what* we want it to do. We do this using **templates**, which specify the conditions in which the process takes place and the output that's produced. How to do the work is entirely up to the processor.

For example, assuming that our three strings were in an XML file like so (paras.xml):

```
<?xml version="1.0"?>
<strings>
  <s>Para 1</s>
  <s>Para 2</s>
  <s>Para 3</s>
</strings>
```

we could get the same HTML as above using an XSLT stylesheet like this (paras.xsl):

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes" encoding="UTF-8"/>
<xsl:template match="/">
  <html>
    <body>
      <xsl:for-each select="strings/s">
        <p><xsl:value-of select="."/></p>
      </xsl:for-each>
    </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

If you try this out in MSXSL, you'll see that we get the same results:

```
>msxsl paras.xml paras.xsl
<html>
<body>
<p>Para 1</p>
<p>Para 2</p>
<p>Para 3</p>
</body>
</html>
```

Notice that this stylesheet looks very much like the output, with some XSLT elements mixed in to specify where content should go?these are the elements in the <http://www.w3.org/1999/XSL/Transform> namespace. The XSLT engine will:



Find the document root of the source tree. (That is, the virtual root of the document's hierarchy?more on the document root coming up.)

- - Match the document root against the single template in our stylesheet.
  - - Output the HTML elements in the template.
    - - Process the XSLT elements in the template.
        - - The <xsl:for-each> works like a template-in-a-template, and applies to any <s> elements which are children of a <strings> root element. The contents of this template are a <p> element, which will be output to the result tree, and the <xsl:value-of> element.
          - - The <xsl:value-of> element will output the contents of those <s> elements to the result tree, as the contents of the <p> element.

Phew. That's a lot of work the XSLT engine does for us!

If we wanted to, we could even specify the order in which we want our paragraphs sorted, by simply changing the <xsl:for-each> element from this:

```
<xsl:for-each select="strings/s">
  <!--other XSLT here-->
</xsl:for-each>
```

to this:

```
<xsl:for-each select="/strings/s">
  <xsl:sort select="."/>
  <!--other XSLT here-->
</xsl:for-each>
```

which tells the XSLT engine to sort the results, and tells it what data to sort by. (In this case, the results are sorted alphabetically by the contents of the <s> element. We'll learn more about the <xsl:sort> element later in the chapter.)

We don't have to program any logic in our stylesheet on how to sort the strings, we simply tell it to do it, and let the XSL processor take care of the sorting for us. (Programming this logic into our JavaScript function would have required us to know complex rules about sorting algorithms, which aren't required for sorting in XSLT.)

XSLT can make things simpler for the programmer like this because it is a very specialized language?it is designed for transforming XML documents into other formats, and that's it. XSLT already knows what XML is, and it already knows what HTML is, which allows the XSLT engine to perform this work for us.

## XSLT Has No Side Effects

Whenever people start talking about the benefits of XSLT, they usually mention the fact that it has **no side effects**. In order to understand why this might be considered a benefit, let's look at what they mean by "side effect".

Suppose you're writing a program in Java, or C++, or some other imperative language, which will take an XML document and print it out from a printer. Also suppose that we have a global variable in that program, which is keeping track of the current page number. Since a global variable is available to any code in the program, we can call different functions and they can update that variable, to increment it as more pages are printed.

When we call a function and it updates our global variable, this is a side effect. If any other function accesses that variable after it has been changed, that function will see the new value. This is very important, because it means we have to call the functions in a particular order. For example, if we have one function that prints out the page number, and another function that updates the page number, calling them in the wrong order would mean that the wrong page number is printed in the output.

This means two things:

- As programmers, we have to be careful to do everything in exactly the right order. If we do all the right things, but do them in the wrong order, it's just as bad as doing the wrong things!
- When we compile our code, the compiler has to run it exactly as we tell it to. That is, even if our code would run more efficiently in a different order, it has to run the way we wrote it, or else the program won't work correctly.

XSLT doesn't have these side effects. We can't modify global variables at run-time the way we can with other programming languages. This means:

- As programmers, we aren't as concerned with doing things in the proper order, just what we want the end result to be. This can greatly reduce the number of bugs in our code, since it's one less thing that we, as programmers, need to worry about.
- An XSLT engine can run the code in your stylesheet in any order it wishes. It can even run multiple pieces of code simultaneously. (Realistically, these optimizations may not exist in current XSLT engines, or may exist only to a very limited degree. However, declarative languages are inherently easier to optimize than imperative ones, so it's only a matter of time.)

There's another consequence to this: programmers who are familiar with imperative languages will have to learn a completely new style of programming. Working with, and changing, variables has been one of the most fundamental aspects of imperative programming, and it can take some getting used to working declaratively. If you're more used to declarative programming languages, XSLT as a declarative language makes a lot of sense, and allows great power and flexibility. However, if you're used to imperative languages like JavaScript, it might take a while to train yourself to stop thinking in imperative terms. Nevertheless, you can be sure that once you get used to XSLT's declarative nature, you'll begin to appreciate the power that's available to you.

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# XPath

Even in the short examples we've seen so far in this chapter, it seems like we're spending a lot of time looking at pieces of the source tree; we must need a pretty flexible way of pointing to different pieces of an XML document. In fact, there's a whole separate specification from the W3C for a language called **XPath**, which does exactly that: address sections of an XML document, to allow us to get the exact pieces of information we need. XSL uses XPath extensively.

*XPath version 1.0 became a W3C recommendation on 16 Nov 1999, and can be found at <http://www.w3.org/TR/xpath>. XPath is not only used in XSLT, it is also used in other W3C technologies, such as XPointer.*

You use **XPath expressions** to address your documents by specifying a **location path**, which conveys to XPath, step-by-step, where you are going in the document. Of course, you can't know where you're going unless you know where you are, so XPath also needs to know the **context node**; that is, the section of the XML document from which you're starting.

## Important

Think of an XPath expression as giving directions through the XML document. ("Okay, you're here, so you need to go down here, turn right at the next street, and it's the first house on the left.")

A **node** is simply a "piece" of an XML document; it could be an element, an attribute, a piece of text, or any other part of an XML document. XPath adds the concept of a node because it's often useful to work with the parent/child relationships of an XML document without having to worry about whether the branches and leaves are elements, attributes, pieces of text, or anything else. XPath also introduces the concept of a **node-set**, which is simply a collection (or "set") of nodes.

In order to illustrate the XPath examples below we'll use a sample XML document, which is a modified version of our order XML from the earlier Try It Out (Franks.xml), so you might want to save the following to your hard drive as order.xml, and keep it handy:

```
<?xml version="1.0"?>
<order OrderNumber="312597">
  <date>2000/1/13</date>
  <customer id="A216">Company A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

We've just added some attributes, for the sake of demonstrating how to address them with XPath.

## How Do Templates Affect the Context Node?

An important point to remember in XSLT is that whatever the template uses for its match attribute becomes the context node for that template. This means that all XPath expressions within the template are **relative** to that node. Take the following example:

```
<xsl:template match="/order/item">
    <xsl:value-of select="part-number"/>
</xsl:template>
```

This template will be instantiated for any `<item>` element that is a child of an `<order>` element that is a child of the document root. That `<item>` element is now the context node for this template, meaning that the XPath expression in the `<xsl:value-of>` attribute is actually only selecting the `<part-number>` element which is a child of the `<item>` element already selected for this template.

## XPath Basics

As mentioned earlier, XPath expressions give directions through an XML document, from one point in the document to another. These directions are given step-by-step, starting at the context node, where each step is separated by a `"//"` character. Consider the following:

```
order/item
```

This expression gives the following directions:

- Start at the context node (this direction is implicit?it is not actually stated in the expression)
- Go to the first child element which is named "order"
- Finally, select the first child element of the `<order>` element which is named "item"

Each step in the XPath expression specifies an XML element, in this case an `<order>` element and an `<item>` element. Alternatively, attributes can be specified in XPath expressions, by putting a `"@"` character in front of the name of the attribute. For example,

```
order/@OrderNumber
```

is the same as the previous expression, except that instead of selecting an `<item>` child element of the `<order>` element, it is selecting an `OrderNumber` attribute of the `<order>` element.

XPath expressions are used for XSLT elements such as `<xsl:value-of>`, which is used to pull information out of the source tree, such as the following:

```
<xsl:value-of select="order/item"/>
```

which would insert the value of the `<item>` element into the result tree.

In addition to XPath expressions, there are also **XPath patterns**. In an XPath pattern, instead of giving directions to a particular node, you are specifying a *type* of node against which to match.

Patterns are used in templates, to indicate to the processor when the template should be instantiated. Consider the following:

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

This template will match any `<item>` element in the source tree which is a child of an `<order>` element.

Note that I find it a lot easier to read XPath patterns backwards (from right to left). In the `<xsl:template>` above, I would read the XPath pattern as "match against any `<item>` element which is a child of an `<order>` element", whereas in the `<xsl:value-of>` earlier I would instead read the expression similar to "go to the `<order>` child of the context node, then to the `<item>` child of that `<order>` child, and select that".

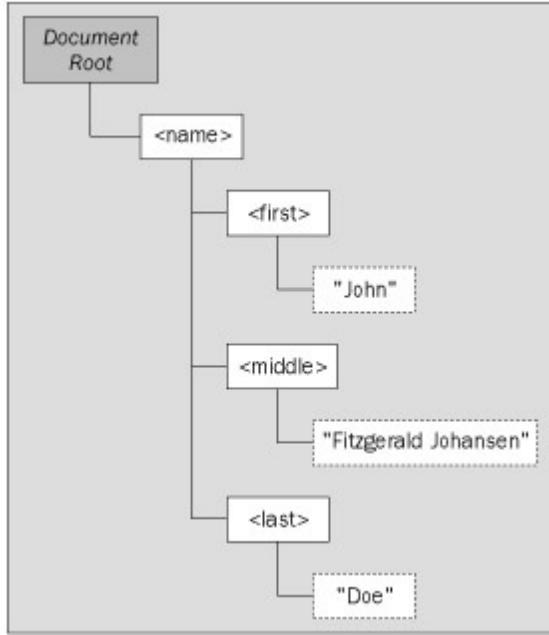
## The Document Root

An important thing to note about XPath is the concept of the **document root**, which is **not** the root element we learned about in [Chapter 2](#). Since there can be other things at the beginning or end of an XML document before or after the root element, XPath needs a virtual document root to act as the root of the document's hierarchy. In fact, the root element is a child of the document root.

Recall our `<name>` example from [Chapter 2](#), which had this XML:

```
<?xml version="1.0"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

the hierarchy according to XPath would be more like this:



The document root doesn't point at any element in the document; it's just the conceptual root of the document. In XPath, the document root is specified by a single `"+"`. This is why the template in our first XSLT example, which matched against `"+"`, applied to the entire document.

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

The `"+"` character also refers to the document root when it is at the beginning of an XPath expression or pattern. For example,

```
<xsl:value-of select="/name/first"/>
```

tells the XSLT processor to go to the document root, then to the `<name>` root element, and finally to the first child of that `<name>` element named "first", and get the value of it. This will always refer to the same element, regardless of what the context node might be, because it starts right from the document root and goes from there.

### Important

Remember: XPath expressions and patterns can be **relative**, meaning that they start at the context node, or **absolute**, meaning that they start at the document root.

The document root is especially important in XSLT stylesheets, since this is where all processing starts. The XSLT processor looks through the stylesheet for a template that matches the document root, and from this point on, all processing is controlled by the contents of this template.

### Important

If no template is supplied which matches against the document root, a default template is provided. More on default templates coming up.

## Filtering XPath Expressions and Patterns

So now that we can address pieces of our XML document, can we get more specific? Of course! Not only can we locate nodes which are in specific locations in our document, but we can match against nodes with specific values, or that meet other conditions. So, for example, instead of matching against `<first>` elements from the source tree, we could match only `<first>` elements whose text is "John".

This is done using "`[]`" brackets, which act like a **filter** (also called a **predicate**) on the node being selected. We can test for elements with specific child nodes by simply putting the name of the child node in the "`[]`" brackets. So:

- "order[customer]" matches any `<order>` element which is a child of the context node and which has a `<customer>` child element.
- "order[@number]" matches any `<order>` element which is a child of the context node and which has a number attribute.
- And to take it a step further, "order[customer = 'Company A']" matches any `<order>` element which is a child of the context node, and which has a `<customer>` child with a value of "Company A".

Notice that in all of the cases previous we're selecting the `<order>` element, *not* the number attribute or the `<customer>` element. The "`[]`" brackets just act as a filter, so that only certain `<order>` elements will be selected.

To check for a node with a specific value, you would use `". = 'Company A'"` for the comparison. (In XPath, `".` refers to the context node.) For example, "customer[. = 'Company A']" matches any `<customer>` element with a value of "Company A".

These filters can be as complex as we want. For example, "order[customer/@id = '216A']" says select any `<order>` elements which have a child element named `<customer>`, which in turn has an id attribute with a value of "216A". But again, it's still the `<order>` element we're selecting, not the `<customer>` element or the id attribute.

## Dealing with Complex Expressions

For more complex XPath expressions, like the following:

```
/order[@number = '312597']/item/part-number[@warehouse = 'Warehouse 11']
```

you might find it easier to break things down into steps. For example, you can specify the XPath expression to the node you want to select, and then add in any filters after. In this case, we can first write the XPath expression as:

```
/order/item/part-number
```

We can then narrow this down to only select the <part-number> elements whose warehouse attribute has a value of "Warehouse 11":

```
/order/item/part-number[@warehouse = 'Warehouse 11']
```

And finally, we select these <part-number> elements only if they are descended from an <order> element whose number attribute had the value "312597":

```
/order[@number = '312597']/item/part-number[@warehouse = 'Warehouse 11']
```

## XPath Functions

In order to make XPath even more powerful and useful, there are a number of [functions](#) that can be used in XPath expressions. Some of these can be used to return nodes and node-sets that can't be matched with regular parent/child and element/attribute relationships. There are also functions that can be used to work with strings and numbers, which can be used both to retrieve information from the original document, and to format it for output.

You can recognize functions because they end with "()" parentheses. Some functions need information to work; this information is passed in the form of **parameters**, which are inserted between the "()" parentheses.

For example, there is a function called string-length(), which returns the number of characters in a string. It might be used something like this:

```
string-length('this is a string')
```

The parameter is the string "this is a string". The function string-length() will use this information to calculate its result (which, in this example, is 16, since there are 16 characters in that string).

Some functions will take parameters, but will also have **default parameters**. In these cases, you can use the function without passing it any parameters, but the function will operate as if you had passed it the default parameters. For example, the string-length() function defaults to the context node if no parameters are passed. So simply saying:

```
string-length()
```

is the same as saying

```
string-length(.)
```

where the period, ".", represents the context node.

We'll be introducing some of these XPath functions as needed throughout the chapter. For information on all of the functions available, see the XPath Recommendation, at <http://www.w3.org/TR/xpath>, for the core XPath functions, as well as the XSLT Recommendation, at <http://www.w3.org/TR/xslt>, for some extension functions added to XPath by XSLT.

*In addition, some XSLT engines allow you to define your own extension functions, to include in your stylesheets. However, the way in which you do so varies from engine to engine, meaning that any stylesheets that include such extension functions will, to some extent, be non-portable, and only work for that particular XSLT engine.*

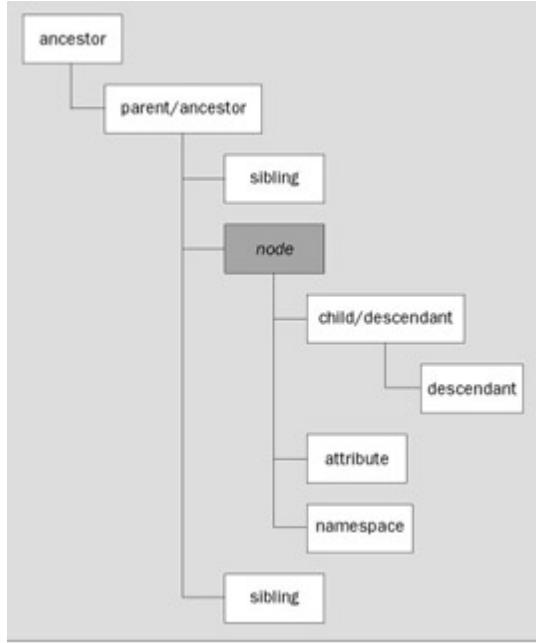
## XPath Axis Names

So far we've always been dealing with children and attributes. However, there are other sections of the tree structure

in XML that we could be looking at. Or, to put it another way, there are numerous directions we can travel, in addition to just going down the tree from parents to children.

The way that we move in these other directions is through different [axes](#). There are 13 axes defined in XPath, which you use in XPath expressions by specifying the axis name, followed by ::, followed by the node name. For example, "child::order" would specify the <order> element in the child axis, and "attribute::OrderNumber" would specify the OrderNumber attribute in the attributeaxis.

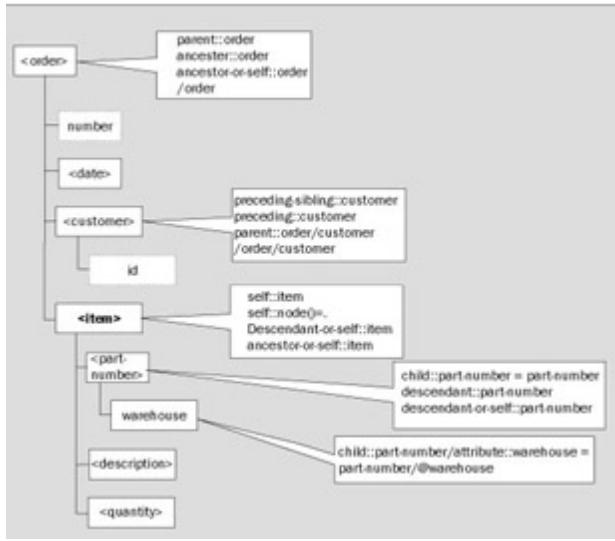
Consider the following diagram:



From our context node, we have access to parents and ancestors, children and descendants, and siblings. If the node is an element, we also have access to any attributes that are attached to that node. And the [last section](#) of the tree we have access to is the namespace. For example, consider the following XML document:

```
<?xml version="1.0"?>
<order number="312597">
  <date>2000/1/1</date>
  <customer id="216A">Customer A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

If the <item> element is the context node, then this diagram represents some of the ways the various nodes could be referenced, using XPath's axes:



Note that not all of the possible XPath expressions have been included; this diagram is just intended to give you a taste of what you can do with axes. Also, notice that in some cases, the axis names have shortcuts. So, for example, the "@" notation we've been using is a shortcut for the attribute:: axis, and the child:: axis name can always be omitted.

## Try It Out?Using XPath Axes

To try out some of these examples, we'll create an XSLT stylesheet that selects <customer> as the context node, and then prints out values according to the XPath expression we give it.

1.

Save the following XML as xpath.xml.(Note that this is the same as our previous Franks.xml document, except that a couple of attributes have been added, for the sake of addressing them in XPath):

```
<?xml version="1.0"?>
<order number="312597">
  <date>2000/1/13</date>
  <customer id="216A">Customer A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

2.

Type in the following, and save it as xpath.xsl:

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

<xsl:template match="item">
  <xsl:for-each select="child::part-number">
    <xsl:value-of select="."/>
  </xsl:for-each>
</xsl:template>

<xsl:template match="text()"><!--do nothing--></xsl:template>
</xsl:stylesheet>
```

Note the select attribute of the <xsl:for-each> element. This is where you will be entering your XPath

expressions. Simply replace the existing XPath expression ("child::part-number") with the XPath expression you wish to test.

3.

Run this stylesheet through MSXSL. The results will be the value of any nodes which match the XPath expression you enter. For example, the stylesheet shown above would produce:

```
>msxsl xpath.xml xpath.xsl  
E16-25A
```

This is enough to get us started using XPath in our XSLT stylesheets. As we go through the rest of this chapter, we will be learning more and more about XPath expressions and patterns, as needed.

---

[!\[\]\(b31da84c9865902068bebba50bc02d6e\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(6cc607de914fea67a62039667a4a2d36\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# XSLT Fundamentals

So far we've covered some of the reasons for using XSLT, and some of the general methodologies used, including XPath, which is used extensively in XSLT. You've also been teased with some sample stylesheets, with promises that all would be made clear later. Now it's time to roll up our sleeves and take a look at some of the XSLT elements we can use in our stylesheets, which will do the actual work of our XSL transformations.

*This chapter won't list all of the elements available with XSLT 1.0, but will introduce the more common ones that you're likely to encounter. Furthermore, not all of the attributes are listed for some of the elements, but again, only the more common ones. For in-depth coverage of all of the XSLT elements, and their use, see the Wrox Press book, *XSLT Programmer's Reference 2nd Edition*, by Michael Kay, ISBN 1-861005-06-7.*

## <xsl:stylesheet>

The <xsl:stylesheet> element is the root element of nearly all XSLT stylesheets, and is used like this:

```
<xsl:stylesheet version="version number"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  exclude-result-prefixes="space-separated list of NS prefixes">
```

If you are following the current (at the time of writing) XSLT specification from the W3C, you should use "1.0" for the version number. Remember also that all of the XSLT elements are in the XSLT namespace, which is almost always declared on the <xsl:stylesheet> element, since it's the root element for the document. As mentioned before, we'll use the convention of using "xsl" as our namespace prefix, although you can use whatever prefix you wish. (Some people simply use "x" for their XSLT namespace prefix, to save on typing.)

Instead of the <xsl:stylesheet> element, you could use the <xsl:transform> element. It has exactly the same meaning and syntax as <xsl:stylesheet>, and is available for those who wish to differentiate their XSLT stylesheets from their XSL Formatting Objects stylesheets. Most people don't use it, however, and stick to <xsl:stylesheet> exclusively, which is the practice we'll follow.

For stylesheets which will only define one template, which matches against the document root, the <xsl:stylesheet> element is optional. For example, instead of this stylesheet from earlier:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="strings/s">
          <p><xsl:value-of select="."/></p>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

we could have used this instead:

```
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<body>
  <xsl:for-each select="strings/s">
    <p><xsl:value-of select="."/></p>
```

```
</xsl:for-each>
</body>
</html>
```

Both of these stylesheets mean exactly the same thing to an XSL processor. All you need to specify are the xsl:version global attribute and the XSLT namespace prefix, and include the rest of the XSLT elements you need as usual. (Notice that the xsl:version attribute needs to be explicitly associated with the XSLT namespace. It is now a global attribute, as discussed in the [previous chapter](#).) This shorthand syntax comes in handy most often when transforming XML documents to HTML, but can be useful in other transformations as well.

*In fact, this shorthand was developed specifically for people who already know HTML, and who want to learn XSL. It was believed that it would be easier for them to get their feet wet before jumping right into fully-fledged XSLT.*

Finally, notice that there is an exclude-result-prefixes attribute for this element. This is used when you are using namespace prefixes in your stylesheet, that you don't want included in the result tree. For example, consider the following XML document:

```
<name xmlns="http://www.sernaferna.com/name">John</name>
```

We might want to process this document using a stylesheet such as the following:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:in="http://www.sernaferna.com/name"
    exclude-result-prefixes="in">
<xsl:template match="/">
    <employee-name><xsl:value-of select="$in:name"/></employee-name>
</xsl:template>
</xsl:stylesheet>
```

In order to get the value from the `<name>` element in our source tree, we have to specify the namespace to which it belongs—remember, we're not just dealing with a `<name>` element, but with a `<{http://www.sernaferna.com/name}name>` element. So, in our stylesheet we have declared a prefix called "in", which refers to that namespace, and used it in our `<xsl:value-of>`. However, by default any namespace prefixes that are declared in a stylesheet will also be declared in the result tree, which is not what we want in this case, since we don't actually use that namespace in our output. The `exclude-result-prefixes` attribute helps with this situation. It instructs the XSLT processor that, if this prefix is not used in the output, it doesn't need to be declared.

As an aside, notice that the `<name>` element in our source document used a default namespace, whereas the stylesheet used a namespace prefix to provide the same information. The XSLT stylesheet doesn't care—all it needs to know is what namespace prefix is being sought, regardless of how that namespace is declared. (By default, any namespace prefixes that are declared in your stylesheet will be included in the output, even if they aren't actually used, as in the example above.)

## **<xsl:template>**

As we have seen already, `<xsl:template>` is the element used to define the templates which make up our XSLT stylesheets. Although this is one of the more important elements you'll be using with XSLT, it's also one of the simplest:

```
<xsl:template match="XPath pattern"
    name="template name"
    priority="number"
    mode="mode name">
```

The match attribute is used to specify the XPath pattern which is matched against the source tree.

The name attribute can be used to create a **named template**; that is, a template which you can explicitly call in your stylesheet. Named templates will be discussed in a section of their own, later in the chapter.

The mode attribute can be used when the same section of the source tree must be processed more than once. Modes will also be discussed in a section of their own, later in the chapter.

## Template Priority

In some cases, there will be more than one template in a document which matches a particular node. In these cases, XSLT has some rules, which it uses to determine which template to call. The most important one is that a more specific template has a higher priority than a less specific one. Consider the following two templates:

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

An `<item>` element with any parent other than `<order>` will match the first template, but not the second, so the first template will be used. On the other hand, an `<item>` element with a parent of `<order>` will match both templates, and in this case the XSLT engine will use the second one instead, because it is more specific.

If there is more than one template which could be instantiated for a node, and XSLT's rules give them the same priority, the priority attribute can be used to force one to be called over the other. (The XSLT engine will instantiate the template with the highest priority.) The attribute takes a numeric value, such as "0", "0.5", or "-0.25". (Higher numbers have a higher priority, and lower numbers have a lower priority.) If two templates have exactly the same priority it is an error-however, the XSLT engine *may* still continue its processing, by choosing the template which occurs last in the stylesheet. Otherwise, it must signal an error.

## <xsl:apply-templates>

The `<xsl:apply-templates>` element is used from within a template, to call other templates:

```
<xsl:apply-templates select="XPath expression"
 mode="mode name">
```

For example, consider the following:

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

This template matches `<order>` elements, so any time the XSLT engine comes across one in the source tree, this template will be instantiated. When it is, a `<requisition>` element will be inserted into the result tree. However, we also have an `<xsl:apply-templates>` element in here, meaning that the XSLT engine will then look for any children of the `<order>` element, in the source tree, which can be matched against other templates. If there are any other such templates, the content that they add to the result tree will be inserted within the `<requisition>` element's start and end tags.

If the select attribute is specified, then the [XPath](#) expression specified will be evaluated, and the result will be used as the context node which other templates are then checked against. It is optional, though, so if it's not specified then the current context node will be used instead. For example, consider the following modification of our previous template:

```
<xsl:template match="order">
<requisition>
  <xsl:apply-templates select="item">
    <xsl:apply-templates select="item"/>
  </requisition>
</xsl:template>
```

In this case, the XSLT engine will only look for templates that match the `<item>` child of the `<order>` element. Even if `<order>` has other children, the XSLT engine will not try to match them against templates, so they won't get processed.

The mode attribute works along with the mode attribute of the `<xsl:template>` element, so will be discussed in the section on [Modes](#) later on.

## Try It Out-Our First Stylesheet

We now have enough XSLT elements to create a rudimentary stylesheet. For this example, we will create an XML document containing a list of names. Our stylesheet will create a simple HTML document, and for each `<name>` encountered, it will output a paragraph with the text "name encountered".

*This isn't really the type of processing you would probably do in the real world, but it will prepare us for some more complex transformations later in the chapter, and give us a chance to use the XSLT elements we've come across so far.*

1.

Here is our XML document, which you should save as `simple.xml`:

```
<?xml version="1.0"?>
<simple>
  <name>John</name>
  <name>David</name>
  <name>Andrea</name>
  <name>Ify</name>
  <name>Chaulene</name>
  <name>Cheryl</name>
  <name>Shurnette</name>
  <name>Mark</name>
  <name>Carolyn</name>
  <name>Agatha</name>
</simple>
```

2.

Now we'll create a stylesheet for this XML. First, we'll need a template to create the main HTML elements. This template will match against the document root, and then call `<xsl:apply-templates>` to let other templates take care of processing the `<name>` elements:

```
<xsl:template match="/">
<html>
  <head><title>Sample XSLT Stylesheet</title></head>
  <body>
    <xsl:apply-templates/>
  </body>
</html>
</xsl:template>
```

3.

Then all we need is a template to process the `<name>` elements:

```
<xsl:template match="name">
<p>Name encountered</p>
</xsl:template>
```

4.

Combining these templates together, along with an `<xsl:stylesheet>` element, produces our XSLT stylesheet in all its glory. Save this as simple.xsl:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<head><title>Sample XSLT Stylesheet</title></head>
<body>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>

<xsl:template match="name">
<p>Name encountered</p>
</xsl:template>
</xsl:stylesheet>
```

Okay, so it's not very exciting, and neither is the output. Running this through MSXSL produces the following:

```
>msxsl simple.xml simple.xsl
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-16">
<title>Sample xslt stylesheet</title></head>
<body>
<p>Name encountered</p>
</body>
</html>
```

5.

Actually, on your screen the output may look more like this:

```
< h t m l > < h e a d > < M E T A h t t p - e q u i v = e t c .
```

This is because the output is in the UTF-16 encoding; the Windows command prompt is treating this two-byte encoding as if it were regular ASCII text, and is outputting a space for the first byte in each two-byte character. Later we'll see how we can tell the XSLT processor to use a different encoding, such as UTF-8, which will be more readable from the command prompt.

Notice that MSXSL has added a <META> tag, that wasn't in the stylesheet, which gives some information about this HTML document. This is because the XSLT processor has determined that the document being produced is an HTML document. We'll see a bit more information on this later, when we discuss the <xsl:output> element.

6.

Remember that you can add a third parameter to MSXSL to specify the output file. So running this again like so:

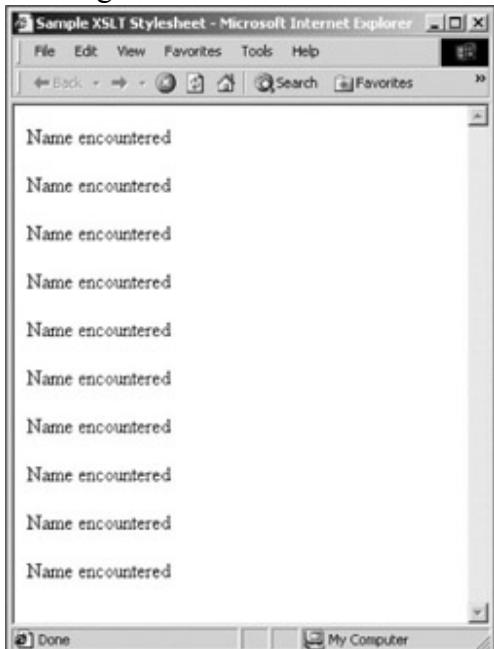
```
>msxsl simple.xml simple.xsl -o simple.html
```

would produce a simple.html file, which would contain the result of the XSLT transformation.

*MSXSL doesn't care what you name the file; it doesn't affect the XSLT processing, so the file doesn't have to have an html extension. We could just as easily name the file simple.xyz, although using the .html extension makes it easier to view the file in a web browser.*

7.

Viewing this HTML in a browser will look something like this:



## How It Works

After the XSL processor has loaded the XML document and the XSLT stylesheet, it instantiates the template which matches the document root. This template outputs the <html>, <head>, <body>, and <title> elements, and then comes across the <xsl:apply-templates>.

It then searches the source tree for further nodes that can be matched against templates, and for each <name> element instantiates the second template. The second template simply outputs "<p>Name encountered</p>".

[PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Getting Information from the Source Tree with `<xsl:value-of>`

Okay, so the last stylesheet was kind of boring. Now that we've got the basics for creating XSLT stylesheets, it would be great if we could actually get them to do something useful with the information in our source tree. The `<xsl:value-of>` element is the way to accomplish this:

```
<xsl:value-of select="XPath expression"
    disable-output-escaping="yes or no"/>
```

The element is simple. It searches the context node for the value specified in the select attribute's XPath expression, and inserts it into the result tree. For example:

```
<xsl:value-of select="customer/@id">
```

inserts the text from the id attribute of the `<customer>` element, and

```
<xsl:value-of select=".">
```

inserts the PCDATA from the context node into the output. (Remember, in XPath `".` is a shorthand for the context node.)

The disable-output-escaping attribute causes the XSLT processor to output "&" and "<" characters, instead of "&#" and "&lt;" escape sequences. Normally, the XSLT processor automatically escapes these characters for you if they make it into the output, but if you specify disable-output-escaping as "yes", then they won't. (The default is "no".) For example, if we have the following XML element in the input file:

```
<name>& ;</name>
```

we have two options. This:

```
<xsl:value-of select="name" disable-output-escaping="no"/>
```

which produces:

& ;

and this:

```
<xsl:value-of select="name" disable-output-escaping="yes"/>
```

which produces:

&

Of course, you need to keep in mind that your output will not be well-formed XML if you do this. Therefore, you

should only use disable-output-escaping if your output is in some format other than XML (such as HTML or plain text), or if you just have no other alternative. (For example, if you need your output to contain "&";, you might put "&&" into your stylesheet, with disable-output-escaping set to "yes". Then the XSLT engine will process the first "&" and insert "&" into the stylesheet, which will then be followed by the second "amp;". However, this is a rare situation, so you won't often need to resort to tricks like this.)

## Try It Out?<xsl:value-of> in Action

Now that we can access the information in our source tree, let's make better use of our list of names. We'll modify the stylesheet from the last Try It Out, so that in the second template each paragraph will contain the text from the <name> elements, instead of a simple string.

1.

Open up simple.xsl, and change the second template to this:

```
<xsl:template match="name">
<P><xsl:value-of select=". "/></P>
</xsl:template>
```

2.

Running the modified stylesheet through MSXSL produces the following:

```
>msxsl simple.xml simple.xsl

<html>

<head>

<META http-equiv="Content-Type" content="text/html; charset=UTF-16">

<title>Sample XSLT Stylesheet</title>

</head>

<body>

<p>John</p>

<p>David</p>

<p>Andrea</p>

<p>Ify</p>

<p>Chaulene</p>

<p>Cheryl</p>

<p>Shurnette</p>

<p>Mark</p>

<p>Carolyn</p>

<p>Agatha</p>
```

</body>

</html>

Again, there will be additional spaces interspersed in the text, as we haven't learned how to fix that yet.

3.

Viewing this HTML in a browser will look something like this:



---

[< Free Open Study >](#)

[PREVIOUS](#)

[NEXT ▶](#)

# Creating the Output

So far we've seen how we can create our result tree by inserting elements and other markup into the stylesheet; anything that isn't in the XSLT namespace will be inserted as-is. However, sometimes we might need more flexibility than inserting hard-coded markup into the stylesheet. We will sometimes also need a way to tell the XSLT processor what kind of output to create-XML, HTML, or plain text.

This section will include some XSLT elements we can use to do these things.

## <xsl:output>

We've mentioned that XSLT can output XML, or HTML, or even plain text. The `<xsl:output>` element allows us to specify the method we'll be using, and also gives us much more control over the way our output is created. If it is included in a stylesheet, it must be a direct child of the `<xsl:stylesheet>` element. In XSLT, elements which must be direct children of `<xsl:stylesheet>` are called **top-level** elements. The syntax of `<xsl:output>`, with some of its most commonly used attributes, is as follows:

```
<xsl:output method="xml or html or text"
    version="version"
    encoding="encoding"
    omit-xml-declaration="yes or no"
    standalone="yes or no"
    cdata-section-elements="CDATA sections"
    indent="yes or no"/>
```

The method attribute specifies which type of output is being produced:

- "xml"-the output will be well-formed XML.
- "html"-the output will be HTML, and will follow the rules outlined in the W3C's HTML Recommendation
- "text"-the output will be plain text
- Some other method. The XSLT specification says that XSLT processors are free to define other types for the method attribute, which they can use for other output methods. If so, the value of the method attribute must be a QName, so that the XSLT processor can use its namespace prefix to identify this method.

If the `<xsl:output>` element is not included, and the root element in the result tree is `<html>` (in uppercase or lowercase), then the default output method is "html"; otherwise, the default output method is "xml". When the method attribute is specified as "html", the XSL processor can do things like changing "`<br/>`" in the stylesheet to "`<br>`" in the result tree, since HTML browsers expect the `<br>` tag to be written like that. (Of course, if you wished to produce XHTML, you would probably set the method attribute to "xml", to ensure that the XSLT engine would produce well-formed XML. There isn't an "xhtml" value for the method attribute.) This is also why the results of our previous stylesheets, which produced HTML, included that extra `<META>` tag-the XSLT processor determined that the output type was "html", since the root element was `<html>`, and treated the document accordingly.

The encoding attribute specifies the preferred encoding to use for the result tree's text. The XSLT processor is free to ignore this attribute, if it doesn't understand the specified encoding. If the output method is "xml", this encoding will be included in the XML declaration. (MSXSL also includes the encoding in the <META> tag, if the output method is "html". Other XSLT processors may not include this <META> tag.)

The version and standalone attributes can also be used when the output method is "xml". Values specified for these attributes will be used to create the XML declaration for the result tree. (Note that if the version attribute is not included, "1.0" will be used for the XML declaration.)

For this book, we'll be specifying UTF-8 for the encoding in all of our stylesheets, because otherwise MSXSL will default the encoding to UTF-16. This will fix the problem we've been having when viewing the results of our transformations from the command line.

The omit-xml-declaration attribute, as its name implies, can be used when outputting XML, to specify if the XML declaration should be included. (For example, if you knew that the output of your transformation was going to be inserted as part of a larger XML document, you would want to make sure you didn't output the XML declaration, so that you wouldn't end up with two XML declarations in the final document.) The default is "no" when the output method is "xml", meaning that the XML declaration *will* appear in the result tree.

The indent attribute can be used to specify some formatting to be used for the result tree. If it is set to a value of "yes", the XSL processor is allowed to add extra whitespace to the result tree, for "pretty printing". For example, earlier we had the following XSLT stylesheet, where we specified the <xsl:output> element with indent set to "yes":

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

<xsl:template match="/">
  <order>
    <date>
      <xsl:value-of select="/order/date/year"/>/<xsl:value-of
        select="/order/date/month"/>/<xsl:value-of select="/order/date/day"/>
    </date>
    <customer>Company A</customer>
    <item>
      <xsl:apply-templates select="/order/item"/>
      <quantity><xsl:value-of select="/order/quantity"/></quantity>
    </item>
  </order>
</xsl:template>

<xsl:template match="item">
  <part-number>
    <xsl:choose>
      <xsl:when test=". = 'Production-Class Widget'">E16-25A</xsl:when>
      <xsl:when test=". = 'Economy-Class Widget'">E16-25B</xsl:when>
      <!--other part-numbers would go here-->
      <xsl:otherwise>00</xsl:otherwise>
    </xsl:choose>
  </part-number>
  <description><xsl:value-of select=". /></description>
</xsl:template>
</xsl:stylesheet>
```

The result of that transformation was:

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
<date>2000/1/13</date>
<customer>Company A</customer>
<item>
<part-number>E16-25A</part-number>
```

```
<description>Production-Class Widget</description>
<quantity>16</quantity>
</item>
</order>
```

If we had left out the `<xsl:output>` element, or set indent to "no", our output would have looked like this instead:

```
>msxsl MomAndPop.xml order.xsl
<?xml version="1.0" encoding="UTF-8"?>
<order><date>2000/1/13</date><customer>Company A</customer><item><part-number>E16
-25A</part-number><description>Production-Class Widget</description><quantity>16
</quantity></item></order>
```

As far as most applications are concerned, these documents are the same. However, if a human will be reading the output, then the extra new lines are helpful. The XSLT specification gives a lot of leeway when it comes to the indent attribute, so different XSL processors may output the result tree differently when it is set to "yes", but in most cases, if not all, it will still be more easily to read by a human.

The `cdata-section-elements` attribute specifies any elements in the result tree which should be output using CDATA sections. The value of this attribute is a space-separated list of QNames of the elements that you wish to be output with CDATA sections. For example, if `cdata-section-elements` has a value of "customer", and the stylesheet contains an element like this:

```
<customer>A& P</customer>
```

or like this:

```
<customer><! [CDATA[A&P] ]></customer>
```

then the output will always look like this:

```
<customer><! [CDATA[A&P] ]></customer>
```

Since CDATA sections are really provided as a convenience to the XML author, and to make XML documents with a lot of escaped characters more readable, you would probably only use the `cdata-section-elements` attribute for elements that you think will have a lot of escaped characters. For example, if you were authoring XHTML documents, you might do this for the `<script>` element, which is to contain scripting code.

## **<xsl:element>**

We've already seen how to insert elements directly into the result tree, but what if we don't know ahead of time what elements we're creating? That is, what if the names of the elements that go into the result tree depend on the contents of the source tree?

The `<xsl:element>` element allows us to dynamically create elements:

```
<xsl:element name="element name"
  use-attribute-sets="attribute set names"
  namespace="namespace URI">
```

The `name` attribute specifies the name of the element. So the following:

```
<xsl:element name="blah">My text</xsl:element>
```

will produce an element like this in the result tree:

```
<blah>My text</blah>
```

Of course, this isn't very exciting, or dynamic. We could have done the same thing by simply writing that <blah> element into the stylesheet manually, and with less typing. The exciting and dynamic part comes into play when we change the name attribute like so:

```
<xsl:element name="{.}">My text</xsl:element>
```

Anything inserted into those curly braces gets evaluated as an XPath expression! In this case, we will create an element, and give it the name from the value from the context node. That is, if we have a template like this:

```
<xsl:template match="name">
  <xsl:element name="{.}">My text</xsl:element>
</xsl:template>
```

then running it against:

```
<name>Andrea</name>
```

will produce:

```
<Andrea>My text</Andrea>
```

and running it against:

```
<name>Ify</name>
```

will produce:

```
<Ify>My text</Ify>
```

The namespace attribute specifies what namespace, if any, this element belongs to. Like the name attribute, you can include curly braces, and have the namespace calculated at run-time from an XPath expression. For example, if your source tree included an element like this:

```
<some-element>http://www.sernaferna.com/namespace</some-element>
```

you could create an element like so:

```
<xsl:element name="ns-elem" namespace="{.}">...</xsl:element>
```

which would produce an element like this:

```
<ns-elem xmlns="http://www.sernaferna.com/namespace">...</ns-elem>
```

Or, if you wish to use a namespace prefix, you can use a QName for the name attribute, and whatever prefix you specify will be used for the namespace prefix in the output. For example,

```
<xsl:element name="x:ns-elem" namespace="{ . }">...</xsl:element>
```

would produce this:

```
<x:ns-elem xmlns:x="http://www.sernaferna.com/namespace">...</x:ns-elem>
```

We'll revisit the use-attribute-sets attribute in the [next section](#). First let's have a look at another example.

## Try It Out-<xsl:element> in Action

To demonstrate <xsl:element>, let's take an XML document which uses attributes exclusively for its information, and transform it to one which uses elements instead.

1.

Save the following XML as PeopleAttrs.xml:

```
<?xml version="1.0"?>
<people>
  <name first="John" middle="Fitzgerald Johansen" last="Doe"/>
  <name first="Franklin" middle="D." last="Roosevelt"/>
  <name first="Alfred" middle="E." last="Neuman"/>
  <name first="John" middle="Q." last="Public"/>
  <name first="Jane" middle="" last="Doe"/>
</people>
```

Now let's create an XSLT stylesheet which can convert all of those attributes to elements.

2.

The first step is our template to match against the document root. This template can output the root <people> element, and then call <xsl:apply-templates> to match any further elements:

```
<xsl:template match="/">
<people>
  <xsl:apply-templates select="people/name"/>
</people>
</xsl:template>
```

3.

We then need to take care of those <name> elements. To do this, we can create a template to output the new <name> element, and then call <xsl:apply-templates> to take care of all of the attributes:

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

Notice the "@\*" notation. The "\*" character is an **XPath wildcard**. An XPath pattern like "\*" means match *any element*, and a pattern like "@\*" means match *any attribute*.

4.

And finally, we need a template to process those attributes. For each attribute encountered, an element with the same name and same value should be output. This template will do the trick:

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

*Notice that for the element name, we're using an XPath function called local-name(). This function will return the local part of an element's QName, without the namespace prefix. For example, for <xns-elem>, local-name() would return "ns-elem".*

*There is also a name() function, which will return the full QName. In this case, it would return "x:ns-elem".*

5.

Combining these templates together produces the following stylesheet, which you should save as AttrsToElems.xsl:

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

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

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

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

6.

Running it through MSXSL produces the following:

```
>msxsl PeopleAttrs.xml AttrsToElems.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name>
<first>John</first>
<middle>Fitzgerald Johansen</middle>
<last>Doe</last>
</name>
<name>
<first>Franklin</first>
<middle>D.</middle>
<last>Roosevelt</last>
</name>
<name>
<first>Alfred</first>
<middle>E.</middle>
<last>Neuman</last>
</name>
<name>
<first>John</first>
<middle>Q.</middle>
```

```
<last>Public</last>
</name>
<name>
<first>Jane</first>
<middle/>
<last>Doe</last>
</name>
</people>
```

## **<xsl:attribute> and <xsl:attribute-set>**

Similarly to `<xsl:element>`, `<xsl:attribute>` can be used to dynamically add an attribute to an element in the result tree:

```
<xsl:attribute name="attribute name"
               namespace="namespace URI">
```

It works in exactly the same way, with the `name` attribute specifying the name of the attribute, the `namespace` attribute specifying the namespace URI (if any), and the text inside the `<xsl:attribute>` element specifying its value. For example:

```
<name><xsl:attribute name="id">213</xsl:attribute>Chaulene</name>
```

will produce the following in the result tree:

```
<name id="213">Chaulene</name>
```

and:

```
<name><xsl:attribute name="{ . }">213</xsl:attribute>Chaulene</name>
```

will do the same, but the name of the attribute will be the text of the context node. Be careful, though! The name of the attribute still has to be a valid XML attribute name! So if the value of the context node above were 'Fitzgerald Johansen', you would be attempting to create an attribute name with a space in it, which the XSLT Processor would reject.

Note that `<xsl:attribute>` must come before any PCDATA of the element to which it is being appended. So if we rewrote the above XSLT like this:

```
<name>Chaulene<xsl:attribute name="{ . }">213</xsl:attribute></name>
```

the attribute wouldn't get appended to the `<name>` element. (MSXSL simply ignores the attribute in this case. Other processors may treat the error differently.)

## **Related Groups of Attributes**

An `<xsl:attribute-set>` is a handy shortcut for a related group of attributes that always go together:

```
<xsl:attribute-set name="name of att set"
                   use-attribute-sets="att set names">
```

For example, if we have an `id` attribute and a `size` attribute, we can define an attribute set as follows:

```
<xsl:attribute-set name="IdSize">
```

```
<xsl:attribute name="id">A-213</xsl:attribute>
<xsl:attribute name="size">123</xsl:attribute>
</xsl:attribute-set>
```

which will append an id with a value of "A-213" and a size with a value of "123" to any element which uses this attribute set. For example,

```
<xsl:element name="order" use-attribute-sets="IdSize"/>
```

would produce this output

```
<order id="A-213" size="123"/>
```

Or we can define the attribute set like this:

```
<xsl:attribute-set name="IdSize">
  <xsl:attribute name=".{}">A-123</xsl:attribute>
  <xsl:attribute name="size"><xsl:value-of select=".{}"/></xsl:attribute>
</xsl:attribute-set>
```

which will append one attribute with the same name as the text of the context node, and a value of "123", and another value with the name size, and the value of the text of the context node.

This attribute set can then be appended to elements in the result tree by using the use-attribute-sets attribute of `<xsl:element>`. Notice also that attribute sets can use other attribute sets, since `<xsl:attribute-set>` also has a use-attribute-sets attribute!

## Try It Out-Attributes and Attribute Sets in Action

Just for fun, let's take an XML document which uses no attributes, and transform it back to a format which uses attributes exclusively for its information.

1.

Save the following document as PeopleElems.xml:

```
<?xml version="1.0"?>
<people>
  <name>
    <first>John</first>
    <middle>Fitzgerald</middle>
    <last>Doe</last>
  </name>
  <name>
    <first>Franklin</first>
    <middle>D.</middle>
    <last>Roosevelt</last>
  </name>
  <name>
    <first>Alfred</first>
    <middle>E.</middle>
    <last>Neuman</last>
  </name>
  <name>
    <first>John</first>
    <middle>Q.</middle>
    <last>Public</last>
  </name>
  <name>
    <first>Jane</first>
```

```
<middle></middle>
<last>Doe</last>
</name>
</people>
```

## 2.

For our stylesheet, the first template will simply match against the document root and output the <people> tag, like our previous templates. But our second template, to take care of the <name> elements, is different:

```
<xsl:template match="//name">
  <xsl:element name="name" use-attribute-sets="NameAttributes"/>
</xsl:template>
```

It creates a <name> element, and then uses an attribute set for the rest of the information.

The "://" notation used here is a special XPath construct; it means "match any <name> element, anywhere in the document".

### Important

This "://" syntax is called the **recursive descent operator**. There are cases, in XPath expressions, where it is useful. However, it can also greatly decrease the performance of your stylesheets. (This is because the XSLT engine must search through the entire XML document looking for matches.) As a general rule, you shouldn't use the "://" syntax unless you really need to. This template doesn't need the recursive descent operator-we could have just used "name", instead of "//name"- but I wanted to introduce it. In fact, you would never use the recursive descent operator in an XPath pattern-only in an expression.

## 3.

The last piece we need is the attribute set:

```
<xsl:attribute-set name="NameAttributes">
  <xsl:attribute name="first"><xsl:value-of select="first"/></xsl:attribute>
  <xsl:attribute name="middle"><xsl:value-of select="middle"/></xsl:attribute>
  <xsl:attribute name="last"><xsl:value-of select="last"/></xsl:attribute>
</xsl:attribute-set>
```

## 4.

And here is the whole XSLT stylesheet that will do the transformation for us, which you should save as ElemsToAttrs.xsl:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" encoding="UTF-8" indent="yes"/>
<xsl:template match="/">
  <people>
    <xsl:apply-templates/>
  </people>
</xsl:template>

<xsl:template match="//name">
  <xsl:element name="name" use-attribute-sets="NameAttributes"/>
</xsl:template>

<xsl:attribute-set name="NameAttributes">
```

```
<xsl:attribute name="first"><xsl:value-of select="first"/></xsl:attribute>
<xsl:attribute name="middle"><xsl:value-of select="middle"/></xsl:attribute>
<xsl:attribute name="last"><xsl:value-of select="last"/></xsl:attribute>
</xsl:attribute-set>
</xsl:stylesheet>
```

## 5.

Running that through MSXSL produces the following:

```
>msxsl PeopleElems.xml ElemsToAttrs.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name first="John" middle="Fitzgerald Johansen" last="Doe" />
<name first="Franklin" middle="D." last="Roosevelt" />
<name first="Alfred" middle="E." last="Neuman" />
<name first="John" middle="Q." last="Public" />
<name first="Jane" middle="" last="Doe" />
</people>
```

## <xsl:text>

The <xsl:text> element, as its name implies, simply inserts some text (PCDATA) into the result tree.

```
<xsl:text disable-output-escaping="yes or no">
```

Much of the time the <xsl:text> element isn't needed, since you can just insert text directly into the stylesheet, but there are two reasons you may sometimes want to use it: it preserves all whitespace, and you can disable output escaping.

We've already seen the disable-output-escaping attribute, in the <xsl:value-of> element. We can do the same when inserting normal text into the document by using <xsl:text> like this:

```
<xsl:text disable-output-escaping="yes">6 is &lt; 7 &amp;gt; 6</xsl:text>
```

The text that is inserted into the result tree will look like this:

```
6 is < 7 &gt; 6
```

(Notice that we still had to escape the characters in the XSLT stylesheet. Remember, no matter what you want the output to look like, your stylesheet is XML, and must be well-formed.)

The other advantage is that all whitespace in the <xsl:text> element is preserved. To see why this is important, consider the following:

```
<xsl:value-of select="'John'/'> <xsl:value-of select="'Fitzgerald Johansen'/'>
<xsl:value-of select="'Doe'/'>
```

You might expect that to produce:

```
John Fitzgerald Johansen Doe
```

However, it will actually produce:

```
JohnFitzgerald JohansenDoe
```

because the spaces between the <xsl:value-of> elements are stripped out by the XSLT processor. (The space in the

text "Fitzgerald Johansen" is preserved, however.) If we were to insert something like this in between each one:

```
<xsl:text> </xsl:text>
```

then a space would be inserted, since the space in `<xsl:text>` is always preserved.

`<xsl:text>` can only have PCDATA for its content. It cannot contain other XSLT elements.

---

[◀ PREVIOUS](#)

[<Free Open Study>](#)

[NEXT ▶](#)

# Default Templates

I mentioned earlier that there are some built-in XSLT templates. Let's take a look at those templates, so that you won't be surprised when the XSLT processor starts using them.

If you don't supply a template in your document which matches against the document root, XSLT provides a default one, which simply then applies any other templates which exist. This default template is defined as follows:

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

This matches against any elements in the document, or the document root, and calls `<xsl:apply-templates>`, to process any children. There is also a built-in template for text and attribute nodes, which is:

```
<xsl:template match="text() | @* ">
  <xsl:value-of select=". />
</xsl:template>
```

This template simply adds the value of the text node or attribute to the result tree. The net result of combining these two default templates is that you could create a stylesheet with no templates of your own, and the defaults would go through every element and attribute in the source tree, and print out the values.

Finally, there is a default template for PIs (Processing Instructions) and comments, which does nothing. It is defined as:

```
<xsl:template match="processing-instruction() | comment() "/>
```

Similarly, the XSLT Recommendation states that there is a built-in template for namespace nodes, which also does nothing. However, there is no pattern that can match against a namespace node, so you can't override this template with one of your own. If you need information about a namespace prefix or URI, you will need to get it out of the source tree manually, using `<xsl:value-of>` or a similar construct.

An important thing to remember about the default templates is that they are always lower priority than the templates you define in your stylesheets. This means that the defaults will only be used if there are no other templates defined for a particular node.

## Try It Out? Making Use of the Built-In Templates

Let's create a quick example, to see how default templates work.

1.

We'll use our familiar `<name>` example, which you probably already have saved from previous chapters as `name.xml`:

```
<?xml version="1.0"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

2.

Our first crack at this stylesheet will be simple; all we want it to do is print out the value of the <first> element. Save the following as default.xsl:

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

<xsl:template match="first">
    First name: <xsl:value-of select=". "/>
</xsl:template>
</xsl:stylesheet>
```

3.

Run this through MSXSL. You will get the following results:

```
>msxsl name.xml default.xsl
```

```
First name: John
Fitzgerald Johansen
Doe
```

We didn't supply a template for the document root, so the default was applied, which is what we wanted. However, the default was also applied for text nodes in our document, which is *not* what we wanted. We can fix that by adding one more template to our stylesheet. Modify default.xsl as follows:

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

<xsl:template match="first">
    First name: <xsl:value-of select=". "/>
</xsl:template>

<xsl:template match="text() | @* ">
    <!--do nothing-->
</xsl:template>
</xsl:stylesheet>
```

4.

Run this through MSXSL again. This time you should get the proper results:

```
>msxsl name.xml default.xsl
```

```
First name: John
```

## How It Works

When the XSLT processor begins processing this XML document, the first thing it does is look for a template that matches the document root. However, in this stylesheet, there is no such template, so the XSLT processor uses the default template instead. All this template does is call <xsl:apply-templates>, so the XSLT processor will look at each child of the document root, and look for templates to instantiate.

In this case, the only child of the document root is the <name> element. Once again, however, there is no template which matches a <name> element, so the default template is called, which simply calls <xsl:apply-templates> again.

This time, when <xsl:apply-templates> is called, there are a number of children: the <first>, <middle>, and <last> elements, and the whitespace text nodes in between all of these elements. In our first crack at the stylesheet, the XSLT processor was applying the default template for text nodes to the whitespace in our <name> element, while in the second iteration we overrode the default template, to print out nothing for text and attribute nodes. I included an XML comment in this template, to make it explicit that I didn't want to process these nodes. (Of course, our source document doesn't have any attributes, but for a generic template like this it's often better to be safe than sorry.)

Next, the XSLT processor comes to the <first> element, and the results are obvious: it instantiates our template, which prints out some text and the value of the element.

And finally, the XSLT processor will process our <middle> and <last> elements. Because we have defined no templates for these elements, the default will again be applied, which will call <xsl:apply-templates>. Each of these elements contains only a text node. In the first iteration of our stylesheet, this text node was being handled by the default, and therefore being printed into the result tree, whereas our second iteration of the stylesheet overrode the default template, and kept this text from being added to the result.

## Try It Out?Default Template Gotchas

Although the default templates can be very helpful in your stylesheets, they can also cause unexpected results, when you're not careful. For example, in step 2 of the Try It Out above we created a stylesheet to print out the value from the <first> element, but because of the default templates, we ended up with the values from *all* of the elements being printed out. Creating templates for specific elements that you want to ignore will be a common thing to do in XSLT, so that you override the defaults.

The defaults can also cause you problems when you're debugging your stylesheets, as this Try It Out will demonstrate.

1.

For this Try It Out, we'll create a stylesheet to process the order.xml document, and produce a text report of the information contained therein.

2.

Create the following stylesheet, and save it to your hard drive as defaulttemplates.xsl:

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

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

<xsl:template match="date">
  Date:  <xsl:value-of select="."/>/>
</xsl:template>

<xsl:template match="customers">
  Customer:  <xsl:value-of select="."/>/>
</xsl:template>

<xsl:template match="item">
  Part number:  <xsl:value-of select="part-number"/>
  Description:  <xsl:value-of select="description"/>
  Quantity:  <xsl:value-of select="quantity"/>
</xsl:template>
</xsl:stylesheet>
```

3.

Run the file through MSXSL. You will get results similar to the following:

**>msxsl order.xml defaulttemplates.xsl**

Date: 2000/1/13

Customer A

Part number: E16-25A

Description: Production-Class Widget

Quantity: 16

Notice that, instead of "Customer: Customer A" in our output, we just get "Customer A". Why is this? Unfortunately, we have misspelled the name "customer" in our stylesheet, and called it "customers" instead. This means that this template doesn't match any elements in the source tree, and will never get called. However, since XSLT provides a default template, it is processing the <customer> element, and appending its contents to the result tree for us!

This is potentially very confusing, and has caused me personally untold hours of grief in my own stylesheets when I have made similar mistakes! Because of the way our output is structured, it looks like our template is getting called, even though it's not. If that template were more complicated, I would probably assume that the problem was in the template itself, and start trying to simplify it to find the problem, before I would even realize that the template wasn't getting called in the first place.

4.

The solution to this problem is simple, of course: change the match attribute of the third template from "customers" to "customer". You will then get the proper output:

**>msxsl order.xml defaulttemplates.xsl**

Date: 2000/1/13

Customer: Customer A

Part number: E16-25A

Description: Production-Class Widget

Quantity: 16

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Controlling the Flow of Stylesheets

As you are creating your stylesheets, you may sometimes need to decide at run-time how the results should be created. This section will introduce some XSLT elements that you can use to control the flow of processing within your stylesheets.

## Conditional Processing with `<xsl:if>` and `<xsl:choose>`

Every programming language in the world has to let you make choices in your code, otherwise it wouldn't be very useful. XSLT is no exception, and it allows us a couple of ways to make choices: `<xsl:if>` and `<xsl:choose>`:

```
<xsl:if test="Boolean expression">

<xsl:choose>
  <xsl:when test="Boolean expression">
  <xsl:when test="Boolean expression">
  <xsl:otherwise>
</xsl:choose>
```

For both `<xsl:if>` and `<xsl:choose>`, the Boolean expression is simply an XPath expression which is converted to a Boolean value.

### Important

Boolean values can only have one of two choices:  
True or False, yes or no, on or off, 1 or 0, etc.

The expression is converted to a Boolean according to the following rules:

- If the value is numeric, it's considered false if it is 0. If the number is any other value, positive or negative, it is true.
- If the value is a string, it is true if its length is longer than 0 characters.
- If the value is a node-set, it is true if it's not empty, otherwise it's false.
- Any other type of object is converted to a Boolean in a way that is dependent on the type of object.

For example, this expression would be considered true if there were a `<name>` element that is a child of the context node, otherwise it would be false:

```
<xsl:if test="name">
```

`<xsl:if>` is the simpler of the two conditional processing constructs. It evaluates the expression in the `test` attribute, and if it is True the contents of the `<xsl:if>` element are evaluated. If the test expression is not True, the contents of `<xsl:if>` are not evaluated. For example, consider the following:

```
<xsl:if test="name">Name encountered.</xsl:if>
```

If there is a <name> element that is a child of the context node, then the text "Name encountered." will be inserted into the result tree. If there is no <name> child of the context node, then nothing will happen.

Important

Note that <xsl:if> does not change the context node, the way a template match does. That is, even if the test expression evaluated as true, and we entered the <xsl:if> element, <name> would not be the context node; the context node would still be whatever it was before we evaluated the test expression.

<xsl:choose> provides a bit more flexibility than <xsl:if>. It allows us to make any one of a number of choices, and even allows for a "default" choice, if desired. For example:

```
<xsl:choose>
  <xsl:when test="salary[number(.) > 2000]">A big number</xsl:when>
  <xsl:when test="salary[number(.) > 1000]">A medium number</xsl:when>
  <xsl:otherwise>A small number</xsl:otherwise>
</xsl:choose>
```

If the <number> element contains a numeric value which is greater than 2000, then the string "A big number" will be inserted into the result tree. If the number is greater than 1000, the string "A medium number" will be inserted, and in any other case the string "A small number" will be inserted. (Notice that we use an XPath function, number(), to convert the value in the <number> element to a numeric value. Remember, all of the text in an XML document is just that-text. XSLT won't treat the data as anything else until we tell it to. If we hadn't used the number() function, and let XSLT treat the text as text, then any value would evaluate to True, as long as it was greater than 0 characters.)

*In this code fragment, we escaped the ">" signs using "&gt;". This isn't strictly necessary, since ">" characters are allowed in XML as-is; however, "<" characters are not, and this often trips people up when writing stylesheets. Many people consider it a good practice to always escape both ">" and "<" characters, just to be consistent.*

Note that for the <xsl:otherwise> to be evaluated, any of the following could be true:

- <number> could contain a numeric value, which is less than 1000
- There could be no <number> node where we specified it in our test expression
- <number> could contain text, instead of a numeric value. If the number() function is passed text which isn't a number, such as "John", it returns the special "NaN" value, which stands for "Not a Number". NaN evaluates to false, when treated as a Boolean.

That is, none of the other tests in the stylesheet can pass the test attribute, for any reason.

In an <xsl:choose>, as soon as one of the test expression evaluates to true, and the XSLT processor is done executing the <xsl:when>, control leaves the <xsl:choose>. So even if two of the <xsl:when> test expressions would evaluate to true, only the first one will be processed, and control will leave the <xsl:choose> without evaluating the second <xsl:when>. If we had written our example like this instead:

```
<xsl:choose>
  <xsl:when test="number[number(.) > 1000]">A medium number</xsl:when>
  <xsl:when test="number[number(.) > 2000]">A big number</xsl:when>
```

```
<xsl:otherwise>A small number</xsl:otherwise>
</xsl:choose>
```

then the second `<xsl:when>` would *never* get called, since any numbers which are greater than 1000 would cause the first `<xsl:when>` to be true.

The `<xsl:otherwise>` is not mandatory in an `<xsl:choose>`. If it is not included, and none of the `<xsl:when>` test expressions evaluates to true, then control will leave the `<xsl:choose>` without inserting anything into the result tree.

## Try It Out-Conditional Processing in Action

Let's imagine that we have a database of a company's employee information. The database can return the information in XML form, like so:

```
<?xml version="1.0"?>
<employee FullSecurity="1">
  <name>John Doe</name>
  <department>Widget Sales</department>
  <phone>(555) 555-5555<extension>2974</extension></phone>
  <salary>62,000</salary>
  <area>3</area>
</employee>
```

The `FullSecurity` attribute is determined by the ID in the database of the user who requested the information: "1" indicates a member of H.R., who has access to the salary information, and "0" indicates someone who has no such access. (A truly poor use of security, but helpful for demonstrating conditional processing.)

The company is broken down into various physical locations, so the `<area>` element tells us which physical location this employee is stationed at.

1.

We don't have an actual database, so we'll just save the above XML as `employee.xml`, and pretend that it came from a database. (We'll be looking at this for real in [Chapter 13](#).)

2.

Now let's create an HTML document which can display this employee's information. Our first template will do most of the work in creating this document; it will create the HTML layout, and insert the values for all of our information except for the phone number and the area. It will also determine if it is allowed to insert the salary, depending on the security.

```
<xsl:template match="/">
<html>
<head>
<title><xsl:value-of select="employee/name"/></title>
</head>
<body>
<h1>Employee: <xsl:value-of select="employee/name"/></h1>
<p>Department: <xsl:value-of select="employee/department"/></p>
<p><xsl:apply-templates select="employee/phone"/></p>
<xsl:if test="number(employee/@FullSecurity)">
  <p>salary: <xsl:value-of select="employee/salary"/></p>
</xsl:if>
<p>Location: <xsl:apply-templates select="employee/area"/></p>
</body>
</html>
</xsl:template>
```

Most of this is pretty easy to us by now, so the only portion of the code we've highlighted is the section that

determines whether we can output the salary. The value of the FullSecurity attribute is just text, so we have to first convert it to a number, using the number() XPath function, as we discussed earlier. Then, since the test attribute expects a Boolean value, it converts that number to true or false. In our case, we have decided to set FullSecurity to either 1 (which evaluates to true in XPath), or 0 (which evaluates to false).

3.

Because the phone number is a little more complex, we'll create a simple template to process just that:

```
<xsl:template match="phone">  
Phone: <xsl:value-of select="text()" /> <xsl:value-of select="extension"/>  
</xsl:template>
```

Notice that we used an XPath function, text(), for the XPath expression in our `<xsl:value-of>`. This function returns the text of the context node. So why didn't we just use `"."`?

The answer is that when we say `<xsl:value-of select=". . ."/>`, XSLT inserts the text of not only the context node, but also the text of any descendant elements. In this case, that would be "(555)555-55552974", which is not what we want. Luckily, `text()` only returns the text from the context node, and not the descendant nodes, so we can specify exactly what text we want.

4.

And finally, we need a template to take care of that `<area>` element. It needs to decide where this employee is located, based on the value in the XML document. We can use `<xsl:choose>` to do that, like this:

```
<xsl:template match="area">  
<xsl:choose>  
  <xsl:when test=". = '1'">Toronto</xsl:when>  
  <xsl:when test=". = '2'">London</xsl:when>  
  <xsl:when test=". = '3'">New York</xsl:when>  
  <xsl:when test=". = '4'">Tokyo</xsl:when>  
  <xsl:otherwise>Unknown Location</xsl:otherwise>  
</xsl:choose>  
</xsl:template>
```

5.

Save the final stylesheet as EmployeeToHTML.xsl, which will look like this:

```
<?xml version="1.0"?>  
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">  
<xsl:output method="html" encoding="UTF-8"/>  
  
<xsl:template match="/">  
  <html>  
    <head>  
      <title><xsl:value-of select="employee/name"/></title>  
    </head>  
    <body>  
      <h1>Employee: <xsl:value-of select="employee/name"/></h1>  
      <p>Department: <xsl:value-of select="employee/department"/></p>  
      <p><xsl:apply-templates select="employee/phone"/></p>  
      <xsl:if test="number(employee/@FullSecurity) ">  
        <p>Salary: <xsl:value-of select="employee/salary"/></p>  
      </xsl:if>  
      <p>Location: <xsl:apply-templates select="employee/area"/></p>  
    </body>  
  </html>  
</xsl:template>  
  
<xsl:template match="phone">
```

```
Phone: <xsl:value-of select="text()"/> x<xsl:value-of select="extension"/>
</xsl:template>

<xsl:template match="area">
  <xsl:choose>
    <xsl:when test="number(.) = 1">Toronto</xsl:when>
    <xsl:when test="number(.) = 2">London</xsl:when>
    <xsl:when test="number(.) = 3">New York</xsl:when>
    <xsl:when test="number(.) = 4">Tokyo</xsl:when>
    <xsl:otherwise>Unknown Location</xsl:otherwise>
  </xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

6.

When MSXSL is run against the XML above, this stylesheet produces the following HTML:

>**msxsl employee.xml EmployeeToHTML.xsl**

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>john doe</title>
</head>
<body>
<h1>Employee: John Doe</h1>
<p>Department: Widget Sales</p>
<p>
Phone: (555)555-5555 x2974</p>
<p>Salary: 62,000</p>
<p>Location: New York</p>
</body>
</html>
```

which looks like this in a web browser:

7.

Now change the FullSecurity attribute to "0", and the <area> element to "1", and re-run the transformation. The HTML will be changed to this:



>**msxsl employee.xml EmployeeToHTML.xsl**

```
<html>
<head>
```

```
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>John Doe</title>
</head>
<body>
<h1>Employee: John Doe</h1>
<p>Department: Widget Sales</p>
<p>Phone: (555)555-5555 x2974</p>
<p>Location: Toronto</p>
</body>
</html>
```

which looks like this in a browser:



## <xsl:for-each>

In many cases there is some specific processing that we want to do for a number of nodes in the source tree. For example, in the Try It Out section that demonstrated <xselement>, we had processing that we needed to do for every <name> element.

So far we've been handling this by creating a new template to do that processing, and then inserting an <xslapply-templates> element from where we wanted that template instantiated. But there is an alternative way to do this kind of processing, using <xsl:for-each>:

```
<xsl:for-each select="XPath expression">
```

The contents of <xsl:for-each> form a template-within-a-template; it is instantiated for any node matching the XPath expression in the select attribute. For example, consider the following:

```
<xsl:for-each select="name">
  This is a name element.
</xsl:for-each>
```

This "template" will be instantiated for every <name> element that is a child of the context node. In this case, all the template does is output the text "This is a name element."

Because <xsl:for-each> is a template, it also changes the context node, as does a regular template. For example, consider the following XML:

```
<names>
  <name>
    <first>John</first>
```

```
<last>Doe</last>
</name>
<name>
  <first>Jane</first>
  <last>Smith</last>
</name>
</names>
```

We could create HTML paragraphs for each `<first>` element, using a template like this:

```
<xsl:template match="names">
  <xsl:for-each select="name">
    <P><xsl:value-of select="first"/></P>
  </xsl:for-each>
</xsl:template>
```

When the template is instantiated, the context node is changed to the `<names>` element. So our `<xsl:for-each>` element need only specify `<name>` in the `select` attribute, as it's a child of this context node. Then, inside the `<xsl:for-each>`, the context node is changed again, to `<name>`. So the `select` attribute of the `<xsl:value-of>` only needs to specify the `<first>` element, as it's a child of this new context node.

## Try It Out- `xsl:for-each` in Action

To demonstrate `<xsl:for-each>`, let's rewrite an earlier stylesheet, which transformed an attributes-only document to an elements-only document. The original stylesheet was this (which we saved earlier as `AttrsToElems.xsl`):

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

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

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

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

1.

And here it is again, with the changes highlighted. Save this as `AttrsToElems2.xsl`:

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

<xsl:template match="/">
  <people>
    <xsl:for-each select="people/name">
      <name>
        <xsl:apply-templates select="@*"/>
      </name>
    </xsl:for-each>
  </people>
```

```
</xsl:template>

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

2.

Running this stylesheet through MSXSL produces the same results as the previous version:

```
>msxsl PeopleAttrs.xml AttrsToElements2.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name>
<first>John</first>
<middle>Fitzgerald Johansen</middle>
<last>Doe</last>
</name>
<name>
<first>Franklin</first>
<middle>D.</middle>
<last>Roosevelt</last>
</name>
<name>
<first>Alfred</first>
<middle>E.</middle>
<last>Neuman</last>
</name>
<name>
<first>John</first>
<middle>Q.</middle>
<last>Public</last>
</name>
<name>
<first>Jane</first>
<middle/>
<last>Doe</last>
</name>
</people>
```

## How It Works

This stylesheet seems a bit simpler than the first one; it is much more straightforward and maintainable. We now only have two templates, instead of three, since the template that was used for `<name>` elements has now been moved up into the first template.

If we look closely at the `<xsl:for-each>`, which is highlighted in the code, we might notice that the contents are exactly the same as the template we replaced.

Remember, for all intents and purposes `<xsl:for-each>` is a template. The only difference between using `<xsl:for-each>` and `<xsl:template>` is that `<xsl:for-each>` can be inserted into other templates, whereas `<xsl:template>` must stand on its own.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Copying Sections of the Source Tree to the Result Tree

In many cases the result tree from our XSLT stylesheets will be very similar to the source tree. Perhaps there will even be large sections that are exactly the same. XSLT provides a couple of elements that you can use to copy sections of the source tree directly to the result tree, for these occasions.

## **<xsl:copy-of>**

The `<xsl:copy-of>` element allows us to take sections of the source tree and copy them to the result tree. This is much easier than having to create all of the elements/attributes manually, and then copying the values using `<xsl:value-of>`, especially if we don't know ahead of time what the source tree will look like. The syntax is as shown:

```
<xsl:copy-of select="XPath expression"/>
```

The element is quite easy to use. The `select` attribute simply specifies an XPath expression pointing to the node or node-set required, and that node or node-set is inserted directly into the result tree, along with any attributes or child elements.

## Try It Out-<xsl:copy-of> in Action

As a simple example of `<xsl:copy-of>`, let's re-examine our `employee.xml` document. We've already seen how to use the `FullSecurity` attribute to specify when to include the information from our `<salary>` element when creating HTML output. But what if we have another application that needs to use the raw XML, and we can't trust it to do the right thing with our security? It would be better to create an XSLT stylesheet which will remove `<salary>`, under the right conditions, before the other application even gets to touch it. So let's create one.

1.

If you don't still have it, recreate the original employee XML document, and save it as `employee.xml` (if you already have it, simply change the `FullSecurity` attribute's value back to "1", and the `<area>` element's text node back to "3"):

```
<?xml version="1.0"?>
<employee FullSecurity="1">
  <name>John Doe</name>
  <department>Widget Sales</department>
  <phone>(555) 555-5555<extension>2974</extension></phone>
  <salary>62,000</salary>
  <area>3</area>
</employee>
```

2.

Now enter the following, and save it as `EmployeeToXML.xsl`:

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

<xsl:template match="/">
  <xsl:choose>
    <xsl:when test="number(employee/@FullSecurity) ">
      <xsl:copy-of select="/" />
```

```
</xsl:when>
<xsl:otherwise>
  <employee FullSecurity="{employee/@FullSecurity}">
    <xsl:for-each select="employee/* [not(self::salary)]">
      <xsl:copy-of select="."/>
    </xsl:for-each>
  </employee>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

3.

When FullSecurity is "1", as above, the output is as follows:

```
>msxsl employee.xml EmployeeToXML.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="1">
<name>John Doe</name>
<department>Widget Sales</department>
<phone>(555)555-5555<extension>2974</extension></phone>
<salary>62,000</salary>
<area>3</area>
</employee>
```

4.

Now change FullSecurity to "0", and resave employee.xml. The new result is as follows:

```
>msxsl employee.xml EmployeeToXML.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="0">
<name>John Doe</name>
<department>Widget Sales</department>
<phone>(555)555-5555<extension>2974</extension></phone>
<area>3</area>
</employee>
```

## How It Works

The first thing our template does is to check the FullSecurity attribute. If it's true, we call `<xsl:copy-of>` with the select attribute set to `"//"`. This copies the entire document root, including all of its children and attributes, to the result tree.

However, if the attribute is false, we need to copy all of the nodes except for the `<salary>` element to the result tree. This is done using the `<xsl:for-each>` element, to loop through all of the child elements of the `<employee>` element that are not the `<salary>` element. (If you look at the select attribute, it selects all children of the `<employee>` element, using `"employee/*"`. But it then further filters that, to select only the ones that aren't `<salary>`, using `"[not(self::salary)]"`.)

*The not() XPath function is a special Boolean function, that returns the reverse of whatever is passed to it. That is, if the expression would have evaluated to true, then not() will return false, and vice versa. In this case, we're creating an XPath expression that will create a node-set, containing all of the nodes which are not a <salary> node.*

*There is also an XPath function called true(), which always returns true, and one called false(), which always*

returns false. These functions are most often useful when debugging stylesheets.

This is one of the few places where the self axis is used, and it might seem a little strange. What we are saying is "if there is not a <salary> element in the self axis", and since the only node which is ever in the self axis is the context node, we're really checking this node's own name! Instead, we could have written that XPath as "[not(local-name(.) = 'salary')]",] which would have had the same effect. However, XSLT programmers use the construct we used in our stylesheet quite often, so it's good to see it in action, so that you won't be confused if you ever see it in a real-life stylesheet.

Each of these nodes is then copied to the result tree using <xsl:copy-of>.

## <xsl:copy>

Similar to <xsl:copy-of> is the <xsl:copy> element. It allows much more flexibility when copying sections of the source tree to the result tree:

```
<xsl:copy use-attribute-sets="att set names">
```

Instead of providing a select attribute, to indicate which section of the source tree to copy, <xsl:copy> simply copies the context node. And, unlike the <xsl:copy-of> construct, children and attributes of the context node are not automatically copied to the result tree. However, the contents of the <xsl:copy> element provide a template where you specify the attributes and children of the node which will go into the result tree. For example, if we have our good old <name> XML:

```
<?xml version="1.0"?>
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

we could create a template that looked like this:

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

The output of this would be:

```
<name/>
```

since none of the children would be copied. However, we could modify the template to look like this:

```
<xsl:template match="name">
  <xsl:copy>
    <blah><xsl:value-of select=". "/></blah>
  </xsl:copy>
</xsl:template>
```

in which case the output would look like this:

```
<name><blah>
  John
  Fitzgerald Johansen
  Doe
</blah></name>
```

Remember, when you use ". ." for the select attribute in <xsl:value-of>, you get the contents of the context node plus any descendants.

What we have done in the second example is to copy the context node, which in this case is the <name> element, created a child element called <blah>, and populated it with the text from <name> and its children.

## Try It Out-<xsl:copy> in Action

Let's redo the style sheet from the last Try It Out, using <xsl:copy> instead of <xsl:copy-of>, and using a fancy tactic called **recursion**.

Recursion is a methodology whereby a function can call itself, which can then call itself again, etc. Of course, in the case of XSLT, instead of creating recursive [functions](#), we would create recursive *templates*. These recursive templates are used quite a lot in XSLT, so it would be a good idea to start learning about them.

1.

Since the best way to understand recursion is to see it in action, let's create the main template for this stylesheet:

```
<xsl:template match="/ | * | @*">
<xsl:copy>
  <xsl:apply-templates select="* | @* | text()" />
</xsl:copy>
</xsl:template>
```

This template applies to the document root, as well as any elements, and any attributes. When an XSL processor starts going through the source tree, it will match the document root against this template. It will then be copied to the result tree, from the <xsl:copy>, and <xsl:apply-templates> will be called.

The XSL processor will then look through the source tree for further nodes, and when it finds any element or attribute, it will match against this template again. The new node will also be copied to the result tree, <xsl:apply-templates> will be called again, and any elements and attributes which are found will once more match against this template.

The template will keep getting called until it processes a node that doesn't have any child elements or attributes. Then it will return back to the template which called it and find further nodes to process, until it has processed all elements and attributes. (Because of the built-in template for text nodes we mentioned earlier, the XSLT engine will automatically add the text nodes' contents to the result tree.)

2.

As we can see, if we were to just put this one template in our stylesheet, all elements and attributes, along with their associated text, would be copied to the result tree, as-is, every time. However, we need to keep that <salary> element from being copied when FullSecurity is false. In order to do that, we can create a second template, which will match against this specific case:

```
<xsl:template match="salary[not(number(parent::employee/@FullSecurity))]">
  <!--do not process this node-->
</xsl:template>
```

To explain that match attribute, the template matches any <salary> element, if the FullSecurity attribute of the <employee> parent element is false. And, since the template doesn't do anything, the node is not processed. We could also have accomplished this by simply using an empty element, like this:

```
<xsl:template match="employee[not(number(@FullSecurity))]/salary"/>
```

However, with the comment included in our version things are clearer.

In the case where FullSecurity is false, the <salary> element matches both templates. However, one of the rules we mentioned earlier is that the most explicit match wins. In this case, our first template matches against any element, whereas the second matches against this specific element, so the second one wins.

3.

Putting it all together produces the following stylesheet, which you should save as EmployeeToXML2.xsl:

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

<xsl:template match="/ | * | @*">
  <xsl:copy>
    <xsl:apply-templates select="* | @* | text()" />
  </xsl:copy>
</xsl:template>

<xsl:template match="salary[not(number(parent::employee/@FullSecurity))]">
  <!--do not process this node-->
</xsl:template>
</xsl:stylesheet>
```

4.

Make sure the FullSecurity attribute in your XML is "1", and then run it through MSXSL. The results should look like this:

```
>msxsl employee.xml EmployeeToXML2.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="1">
<name>John Doe</name>
<department>Widget Sales</department>
<phone>(555)555-5555<extension>2974</extension></phone>
<salary>62,000</salary>
<area>3</area>
</employee>
```

5.

Now change FullSecurity to "0" and re-run it. The results will look like this:

```
>msxsl employee.xml EmployeeToXML2.xsl
<?xml version="1.0" encoding="UTF-8"?>
<employee FullSecurity="0">
<name>John Doe</name>
<department>Widget Sales</department>
<phone>(555)555-5555<extension>2974</extension></phone>
<area>3</area>
</employee>
```

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Sorting the Result Tree

Sorting in XSLT is accomplished by adding one or more `<xsl:sort>` children to an `<xsl:apply-templates>` element, or an `<xsl:for-each>` element:

```
<xsl:sort select="XPath expression"
  lang="lang"
  data-type="text or number"
  order="ascending or descending"
  case-order="upper-first or lower-first"/>
```

The select attribute chooses the element/attribute/etc. by which you want the XSL processor to sort. If more than one `<xsl:sort>` child is added, then the output is sorted first by the element/attribute/etc. in the first `<xsl:sort>`, and if there are any duplicates they are sorted by the element/attribute/etc. in the second, and so on.

If the XSL processor goes through all of the `<xsl:sort>` elements, and there are still two or more items with identical results, they are inserted into the result tree in the order in which they appear in the source tree.

The data-type attribute specifies whether the data you are sorting is numeric or textual in nature. For example, consider the numbers 1, 10, 5, and 11. If we sort these with data-type specified as "text", the result will be 1, 10, 11, 5. But if we sort numerically, with data-type specified as "number", the result will be 1, 5, 10, 11. The default is "text".

The order attribute specifies whether the result should be sorted in ascending or descending order. The default is ascending.

The case-order attribute specifies whether uppercase letters should come first, or lowercase letters should come first. For example, if case-order is "upper-first", then "A B a b" would be sorted as "A a B b", and if case-order is "lower-first", it would be sorted as "a A b B". The default value for case-order depends on the lang attribute, which specifies the language this document is in. (When lang is set to "en", the default case-order is upper-first).

## Try It Out-<xsl:sort> in Action

To demonstrate sorting with `<xsl:sort>`, let's sort our simple names XML.

1.

If you don't still have it, recreate PeopleElems.xml, using the following XML:

```
<?xml version="1.0"?>
<people>
  <name>
    <first>John</first>
    <middle>Fitzgerald</middle>
    <last>Doe</last>
  </name>
  <name>
    <first>Franklin</first>
    <middle>D.</middle>
    <last>Roosevelt</last>
  </name>
  <name>
    <first>Alfred</first>
    <middle>E.</middle>
    <last>Neuman</last>
  </name>
</people>
```

```
<name>
  <first>John</first>
  <middle>Q.</middle>
  <last>Public</last>
</name>
<name>
  <first>Jane</first>
  <middle></middle>
  <last>Doe</last>
</name>
</people>
```

2.

The following XSLT stylesheet will take that XML and sort it, first by last name, then by first. Save it as sort.xsl:

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

<xsl:template match="/">
<people>
  <xsl:for-each select="people/name">
    <xsl:sort select="last"/>
    <xsl:sort select="first"/>
    <xsl:copy-of select="."/>
  </xsl:for-each>
</people>
</xsl:template>
</xsl:stylesheet>
```

Since ascending order is the default, we didn't bother to specify it.

3.

Running this through MSXSL produces the following:

```
>msxsl PeopleElems.xml sort.xsl
<?xml version="1.0" encoding="UTF-8"?>
<people>
<name>
  <first>Jane</first>
  <middle></middle>
  <last>Doe</last>
</name>
<name>
  <first>John</first>
<middle>Fitzgerald Johansen</middle>
<last>Doe</last>
</name>
<name>
<first>Alfred</first>
  <middle>E.</middle>
<last>Neuman</last>
</name>
<name>
  <first>John</first>
  <middle>Q.</middle>
  <last>Public</last>
</name>
<name>
<first>Franklin</first>
  <middle>D.</middle>
```

```
<last>Roosevelt</last>
</name>
</people>
```

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Modes

The `<xsl:template>` and `<xsl:apply-templates>` elements both have a mode attribute, which we have conveniently ignored up until now. What does the mode attribute do?

Modes are handy when you need to process the same section of XML more than once. It allows you to create these templates and only call the one you want.

Using modes is trivial. When you create the template, you specify the mode attribute, giving it a name to describe this mode. To call the template, use `<xsl:apply-templates>` as before, but with the addition of the mode attribute, specifying the same mode. Now, in addition to matching the templates against the source tree, the XSL processor will also make sure that the only templates called are ones which are part of this mode. For example, if we have the following XSLT:

```
<xsl:apply-templates select="name" mode="TOC"/>
```

and the following templates:

```
<xsl:template match="name" mode="TOC"/>
<xsl:template match="name" mode="body"/>
<xsl:template match="name"/>
```

only the first template will be instantiated, even though all of these templates match against the same section of the source tree.

Keep in mind that the built-in templates also apply when modes are being used. So, along with the default template that we saw earlier,

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

there is also a built-in template which works with modes, similar to

```
<xsl:template match="* | /" mode="mode">
  <xsl:apply-templates mode="mode"/>
</xsl:template>
```

where "mode" refers to the current mode. In other words, if an XSLT processor is currently working in "TOC" mode, and comes across an element that has no templates defined for it, the processor will apply a template like the following:

```
<xsl:template match="* | /" mode="TOC">
  <xsl:apply-templates mode="TOC"/>
</xsl:template>
```

## Try It Out? Modes in Action

One common example given for modes is transforming an XML document to a web page. A template running under one mode can be used for a table of contents, and a template matching against the same section of the document can

be run under another mode for the actual body of the document.

To demonstrate this, we'll use some XML to represent a chapter from one of the most brilliant XML books ever written.

1.

Type in the following, and save it as chapter.xml:

```
<?xml version="1.0"?>
<chapter>
  <title>XSLT</title>
  <section>What is XSL?
    <paragraph><important>Extensible Stylesheet Language</important>, as the name implies, is an XML-based language used to create <important>stylesheets</important>. An XSL engine uses these stylesheets to...</paragraph>
    <paragraph>There are actually two...</paragraph>
  </section>
  <section>Why is XSLT So Important for E-Commerce?
    <paragraph>To get an idea of the power of XSLT...</paragraph>
    <paragraph>Unfortunately, the chances are...</paragraph>
  </section>
</chapter>
```

This is a bit shorter than the text of the actual chapter, but it's enough to demonstrate what we're doing. We want to take this and transform it to HTML, with a table of contents we can use to navigate to any section.

2.

Our first step is to set up the template to match against the document root, which will output the main structure of the HTML file:

```
<xsl:template match="/">
<html>
<head><title><xsl:value-of select="chapter/title"/></title></head>
<body>
<h1><xsl:value-of select="chapter/title"/></h1>

<h2>Table of Contents</h2>
<ol>
  <xsl:apply-templates select="chapter/section" mode="TOC"/>
</ol>

<xsl:apply-templates select="chapter/section" mode="body"/>

</body>
</html>
</xsl:template>
```

The two important pieces of code are the ones where we call `<xsl:apply-templates>` to process the main body of our document. The same portion of the source tree will be processed twice by two separate templates, once for each mode.

3.

Next we need to create those two templates. The first will output our table of contents:

```
<xsl:template match="section" mode="TOC">
<li><a href="#{concat('#section', position())}">
  <xsl:value-of select="text()" /></a>
</li>
</xsl:template>
```

We're using two new XPath functions here: `position()` and `concat()`. The `position()` function returns a node's numeric position within a node-set; that is, for the first node in a node-set, `position()` would return "1", and for the second "2". The `concat()` function takes two strings, and concatenates them together (that is, combines them into one string). In this case, we're taking the string "#section", and adding it to the value returned from the `position()` function, to create strings similar to "#section1", "#section2", etc.

The next effect is that we are creating an HTML list item for each section in the body, using the `position()` function to make sure that every section is uniquely identified.

4.

The second template will output the main body of the document:

```
<xsl:template match="section" mode="body">
<a name="{concat('section', position())}"><h2>
  <xsl:value-of select="text()"/></h2>
</a>
<xsl:apply-templates/>
</xsl:template>
```

It matches against the `<section>` elements again, but works in "body" mode. This template creates an `<h2>` heading, with the title of the section, and wraps it in an `<a>` to identify it, so that the links from the table of contents will work. Then `<xsl:apply-templates>` is called, to take care of the rest of the work in creating this section.

5.

The two following templates are very simple ones, which just transform the `<paragraph>` elements to HTML `<p>` elements, and the `<important>` elements to HTML `<b>` elements:

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

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

6.

And finally, we come to the last template in the stylesheet, which matches against any PCDATA which is a direct child of a `<section>` element; that is, its title. This is done so that these titles won't make it into the body of the section itself.

The problem is those default templates, which all XSL processors implement. Remember that one of these default templates matches against any text that isn't matched against any other templates, and outputs that text to the result tree. This means that when we call `<xsl:apply-templates>` from within the body template, we would get the PCDATA which is a direct child of a `<section>` element inserted into the result tree as well, because of this default template. The template to hide these titles is as follows:

```
<xsl:template match="/chapter/section/text()"><!--do nothing--></xsl:template>
```

7.

Putting all of this together produces the following stylesheet:

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

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

<xsl:template match="/">
<html>
<head><title><xsl:value-of select="chapter/title"/></title></head>
<body>
<h1><xsl:value-of select="chapter/title"/></h1>

<h2>Table of Contents</h2>
<ol>
  <xsl:apply-templates select="chapter/section" mode="TOC"/>
</ol>

<xsl:apply-templates select="chapter/section" mode="body"/>

</body>
</html>
</xsl:template>

<xsl:template match="section" mode="TOC">
  <li><a href="{concat('#section', position())}"><xsl:value-of select="text()"/></a></li>
</xsl:template>

<xsl:template match="section" mode="body">
  <a name="{concat('section', position())}"><h2><xsl:value-of select="text()"/></h2></a>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="paragraph">
  <p><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match="important">
  <b><xsl:apply-templates/></b>
</xsl:template>

<xsl:template match="/chapter/section/text()"><!--do nothing--></xsl:template>
</xsl:stylesheet>
```

## 8.

Save this as chapter.xsl, and run it through MSXSL like this:

```
>msxsl chapter.xml chapter.xsl -o chapter.html
```

You will get an HTML file called chapter.html, which contains the information from this chapter. In a browser, it will look similar to this:

The screenshot shows a Microsoft Internet Explorer window titled "XSLT - Microsoft Internet Explorer". The menu bar includes File, Edit, View, Favorites, Tools, and Help. The toolbar includes Back, Forward, Stop, Refresh, Favorites, Search, History, and Stop. The main content area displays the following text:

**XSLT**

**Table of Contents**

1. [What is XSLT?](#)
2. [Why is XSLT So Important for E-Commerce?](#)

**What is XSLT?**

Extensible Stylesheet Language, as the name implies, is an XML-based language used to create **style sheets**. An XSL engine uses these style sheets to...

There are actually two...

**Why is XSLT So Important for E-Commerce?**

To get an idea of the power of XSLT...

Unfortunately, the chances are...

[\*\*PREVIOUS\*\*](#)

[\*\*< Free Open Study >\*\*](#)

[\*\*NEXT ▶\*\*](#)

# Variables, Constants, and Named Templates

Anyone who has worked with programming languages is familiar with **variables** and **constants**. Variables are places in your program where you store information, while constants are values which are predetermined when the program is written, and can never be changed. For example, in Java we could create a variable to store a person's age and a constant to store the value of Pi like this:

```
int nAge = 30;
final float cfPI = 3.14;
```

The nAge variable can then be changed any time we need to in our application, whereas cfPI can never be changed, but will always be 3.14:

```
nAge = 42; //this is allowed
cfPI = 42; //this is not allowed
```

Constants can be useful in a variety of situations. For example, if you were designing an application to print out reports to a printer, you might need to know throughout the application how many lines per page you're outputting, or what fonts to use. You could just manually put the values in wherever you need them, or you could define constants to store the values, and then use those. The usefulness of constants would become immediately apparent the first time you needed to change one of those values and recompile your program: instead of searching through all of the code for your application, and manually replacing each occurrence of the value, you would simply need to change the value of your constant, in one place. XSLT also provides us with mechanisms for adding constants and variables to stylesheets, using `<xsl:variable>` and named templates.

## `<xsl:variable>`

The `<xsl:variable>` element allows us to add simple constants to stylesheets. For example, we could define a variable for Pi like so:

```
<xsl:variable name="cfPI">3.14</xsl:variable>
```

This variable can then be accessed anywhere you would use a regular XPath expression, using a dollar sign followed by the variable name. For example:

```
<math pi="{$cfPI}" />
```

or:

```
<xsl:value-of select="$cfPI" />
```

But wait, there's more! `<xsl:variable>` can also contain XML markup, and even XSLT elements! For example:

```
<xsl:variable name="space">
  <xsl:text> </xsl:text>
</xsl:variable>

<xsl:variable name="name">
  <name>
```

```
<xsl:value-of select="/name/first"/>
<xsl:copy-of select="$space"/>
<xsl:value-of select="/name/last"/>
</name>
</xsl:variable>

<!--this gets the value of the $name variable, including any XML markup--&gt;
&lt;xsl:copy-of select="$name"/&gt;</pre>
```

Notice that \$name is allowed to reference \$space. One important note, however, is that variables are not allowed to reference themselves, and neither are circular references allowed. (These would both be the XSLT equivalent of infinite loops. An XSL engine should catch this for you, to prevent your machine from crashing.) For example, the following *are not allowed* in XSLT:

```
<!--variables are not allowed to reference themselves-->
<xsl:variable name="name">
  <name><xsl:value-of select="$name"/></name>
</xsl:variable>

<!--circular references are not allowed either-->
<xsl:variable name="A">
  <b><xsl:value-of select="$B"/></b>
</xsl:variable>

<xsl:variable name="B">
  <a><xsl:value-of select="$A"/></a>
</xsl:variable>
```

As an alternative syntax, `<xsl:variable>` can have a `select` attribute. In this case, the value of the variable is the result from the XPath expression supplied. When `<xsl:variable>` has a `select` attribute, it is not allowed to have any content. For example:

```
<xsl:variable name="name" select="/people/name"/>
```

In all cases, whether the variable contains straight text or gets its value from the source tree, the value of the variable is fixed. This means that you can't change the value of a variable after it has been declared. (Refer back to the "XSLT Has No Side Effects" section earlier for the reasons behind this.) Programmers who are used to imperative languages will have a much easier time if they don't think in terms of variables when programming with XSLT, but treat `<xsl:variable>` like a constant.

When working with variables, it's important to remember the **binding**. That is, *where* in our stylesheet the constant can be referenced. (In other programming languages, this is often called the *scope*.) Consider the following example:

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

  <xsl:variable name="age">30</xsl:variable>

  <xsl:template match="/">
    <xsl:variable name="name">Fred</xsl:variable>
    <xsl:value-of select="concat($name, ' is ', $age)"/>
  </xsl:template>
</xsl:stylesheet>
```

When run with any XML file, this produces the following:

>**msxsl anyfile.xml scope.xsl**

Fred is 30

The \$age variable is called a **global** variable, because it is defined outside of any templates, which makes it accessible anywhere in the stylesheet. \$name, on the other hand, is a **local** variable, because it is defined inside a template, and therefore only accessible inside that template. It is also possible to create variables in different bindings with the same name. For example:

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

<xsl:variable name="name">Fred</xsl:variable>

<xsl:template match="/">
  <xsl:variable name="name">Barney</xsl:variable>
  <xsl:value-of select="$name"/>
</xsl:template>
</xsl:stylesheet>
```

In that <xsl:value-of>, which \$name are we selecting? The answer is that the variable with the most restrictive binding is used. In this instance, the \$name of "Fred" is global, and the \$name of "Barney" is local to this template, so the second one is used. Running this stylesheet in MSXML against any XML file produces the following:

```
>msxsl anyfile.xml binding.xsl
```

```
Barney
```

## Named Templates

Variables can be a powerful tool in XSLT stylesheets, but sometimes we need a bit more flexibility. Perhaps we need the functionality of a variable, but we also need to change the results returned from the variable depending on the source tree, similar to a template.

The problem with templates, though, from what we have learned so far, is that they work by matching against the source tree, so we can't just explicitly call a template whenever we want it. Furthermore, if we had one particular template which we wanted to instantiate for a number of different nodes in the source tree, we would have to define separate, identical templates for each node we match against, or else create a complex XPath expression for the template's match attribute, saying "match against this node, or that node, or this other node?".

Luckily, XSLT introduced the concept of **named templates**, which makes both of the above statements untrue. We create them using the same <xsl:template> element we have been using, but instead of using the match attribute, we specify the name attribute. These templates are then called with the simple <xsl:call-template> element. For example, consider the following XSLT:

```
<xsl:template match="/">
  <xsl:for-each select="date">
    <xsl:call-template name="bold"/>
  </xsl:for-each>

  <xsl:for-each select="customer">
    <xsl:call-template name="bold"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="bold">
  <b><xsl:value-of select="."/></b>
</xsl:template>
```

We have a simple template, which just outputs the contents of the context node, wrapped in an HTML <b> element. But this template is called both for <date> elements and for <customer> elements in the source tree; in fact, the bold

template in this case doesn't care *what* elements it is operating on.

If you wish, you can create templates with both the match and the name attributes; this will create a normal template which is instantiated through <xsl:apply-templates>, but that can also be called explicitly if desired.

## Parameters

At times we may not be able to make our templates generic enough for every case. For example, consider a template for outputting a name into the result tree:

```
<xsl:template name="name">
  First name: <xsl:value-of select="first"/>
  Last name: <xsl:value-of select="last"/>
</xsl:template>
```

In this case, we're assuming that the first name will always be in a <first> element, and the last name will always be in a <last> element. But what if we have an input document with names represented in different places in different ways? It might be better if we could specify to the template which values to use for the first name, and which values to use for the last.

The <xsl:param> element allows us to give our templates parameters, just like we give functions parameters in other programming languages. We could rewrite the previous template like this:

```
<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First name: <xsl:value-of select="$first"/>
  Last name: <xsl:value-of select="$last"/>
</xsl:template>
```

Now we're getting the values from our parameters. But, since those parameters are just getting their information from the <first> and <last> elements like before, how have we bought ourselves any benefits? The answer is that the information put inside the <xsl:param> element is only the **default value** for that parameter. So if we call our name template, without giving it any parameters, it will use the values from the <first> and <last> elements; if we call it with parameters, it will use the values we give it, instead of the defaults.

We pass parameters to the templates using the <xsl:with-param> element.

Stylesheets can also have parameters, by specifying top-level <xsl:param> elements. However, the XSLT specification doesn't specify how those parameters should be passed to the XSL processor, so the mechanics will vary from processor to processor. (For command-line XSLT engines, this is typically done through command-line parameters.)

## Try It Out-Parameters in Action

Let's put the above name template into an actual stylesheet, and see it in action.

1.

For our XML, we'll use the following format, which you should save as person.xml:

```
<?xml version="1.0"?>
<person>
  <name>
    <first>John</first>
    <last>Doe</last>
  </name>
</person>
```

2.

And here is our stylesheet, which you should save as person.xsl:

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

<xsl:template match="/">
  <xsl:for-each select="person/name">
    <xsl:call-template name="name"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First: <xsl:value-of select="$first"/>
  Last: <xsl:value-of select="$last"/>
</xsl:template>
</xsl:stylesheet>
```

Because the first and last names are contained in the `<first>` and `<last>` elements, just like our named template is expecting, we don't need to supply it with any parameters.

3.

Running this through MSXSL produces the following:

```
>msxsl person.xml person.xsl
```

```
First: John
Last: Doe
```

4.

Now let's change our XML document slightly, and resave it:

```
<?xml version="1.0"?>
<person>
  <name>
    <given>John</given>
    <family>Doe</family>
  </name>
</person>
```

5.

And also change our stylesheet just a bit, and resave it:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8"/>
<xsl:template match="/">
  <xsl:for-each select="/person/name">
    <xsl:call-template name="name">
      <xsl:with-param name="first">
        <xsl:value-of select="given"/>
      </xsl:with-param>
      <xsl:with-param name="last">
        <xsl:value-of select="family"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:for-each>
```

```
</xsl:template>

<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First: <xsl:value-of select="$first"/>
  Last: <xsl:value-of select="$last"/>
</xsl:template>
</xsl:stylesheet>
```

Since the first and last names aren't where the template expects to find them anymore, we pass it the parameters instead.

6.

Running this through MSXSL produces the following:

```
>msxsl person.xml person.xsl
```

```
First: John
Last: Doe
```

which, you may notice, is the same output as last time.

7.

Finally, let's modify our stylesheet one last time:

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

<xsl:template match="/">
  <xsl:for-each select="/person/name">
    <xsl:call-template name="name">
      <xsl:with-param name="first">Fred</xsl:with-param>
      <xsl:with-param name="last">Garvin</xsl:with-param>
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>

<xsl:template name="name">
  <xsl:param name="first"><xsl:value-of select="first"/></xsl:param>
  <xsl:param name="last"><xsl:value-of select="last"/></xsl:param>

  First: <xsl:value-of select="$first"/>
  Last: <xsl:value-of select="$last"/>
</xsl:template>
</xsl:stylesheet>
```

This is just to demonstrate that the values passed to the parameters don't have to come from the source tree at all; in this case, we just pass it the strings "Fred" and "Garvin", and it uses those for its values.

8.

Running this through MSXSL produces the following:

```
>msxsl person.xml person.xsl
```

```
First: Fred
Last: Garvin
```

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

This chapter has introduced Extensible Stylesheet Language Transformations, XSLT, a powerful template-based language that can be used to transform XML documents into, well, almost anything. We've covered:

- What XSLT is, and why it is important when working with XML
- What XPath is, and why a language like this is so necessary when working with XSLT
- Most of the XSLT elements you'll ever need when creating your own stylesheets

In all, we've learned how to put XSLT to use to style our XML documents in a number of exciting ways. The [next chapter](#) introduces Document Type Definitions, or DTDs.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 5: Document Type Definitions

## Overview

As we have seen in the first few chapters, the rules for XML are straightforward. It doesn't take much to create well-formed XML documents to describe any information that we want. We have also learned that when we create our XML documents, we can categorize our documents into groups of similar document types, based on the elements and attributes they contain. We learned that the elements and attributes that make up a document type are known as the document's vocabulary. In [Chapter 3](#), we learned how to use multiple vocabularies within a single document using namespaces. However, by this time, you may be wondering how to define your own types of documents and be able to check that certain documents follow the rules of your vocabulary.

Suppose you are developing an Internet site that utilizes our <name> sample from [Chapter 1](#). In our <name> sample, we created a simple XML document that allowed us to enter the first, middle and last name of a person. In our sample, we used the name "John Fitzgerald Doe". Now suppose that users of your Internet site are sending you information that does not match the vocabulary you developed. How could you verify that the content within the XML document is valid? You could write some code within your web application to check that each of the elements is correct and in the correct order, but what if you want to modify the type of documents you can accept? You would have to update your application code, possibly in many places. This isn't much of an improvement from the text documents we discussed in [Chapter 1](#).

The need to validate documents against a vocabulary is common in markup languages. In fact, it is so common that the creators of XML included a method for checking validity in the XML Recommendation. A document is **valid** if its XML content complies with a definition of allowable elements, attributes, and other document pieces. By utilizing special **Document Type Definition** syntaxes, or DTDs, you can check the content of a document type with special parsers. The XML Recommendation separates parsers into two categories-validating and nonvalidating. Validating parsers, according to the Recommendation, must implement validity checking using DTDs. Therefore, if we have a validating parser, we can remove the content checking code from our application and depend on the external processor.

*Although you will learn everything you need to know about DTDs in this chapter, you may like to see the XML Recommendation and its discussion of DTDs for yourself. If so, you can look it up at <http://www.w3.org/TR/REC-xml.html#dt-doctype>.*

In this chapter we will learn:

- How to create DTDs
- How to validate an XML document against a DTD
- How to use DTDs to create XML documents from multiple files

Important

Please note: all the code and files you need for each example in this chapter are available in the download for this book at <http://www.wrox.com/>.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Running the Samples

We have talked about some of the benefits of DTDs, but it will probably help us if we look at an example DTD before we move on. In order to see how the DTD works we will define a vocabulary for our <name> example from [Chapter 1](#).

## Preparing the Ground

We will need a program that can validate an XML document against a DTD. Although Microsoft's MSXML parser does support DTD validation, Internet Explorer, by default, does not. Although, MSXML 4 is nearing completion, at the time of writing they are still beta software. Instead of using Internet Explorer, as we have been doing in our previous samples, we will use a separate program that utilizes the Xerces Java Parser from the Apache Software Foundation. This will allow us to utilize the same parser for the next few chapters. For the sake of simplicity, this chapter will assume you have used the path C:\Wrox as the base path for the examples.

### Install JRE

Because our validation program was written in Java we will need to download and install the Java Runtime Environment (JRE) from Sun Microsystems. If you already have the JRE, or Java Development Kit (JDK) installed you can skip to the next step. We will need to use the Java Platform 2 Standard Edition (version 1.2 or newer).

Installing the JRE is very easy; unfortunately, it is a large download (anywhere from 5-15 megabytes depending on the version). Go to <http://java.sun.com/j2se/> and click to download the latest Java Runtime Environment. You can choose to download the Java Development Kit instead, if you like—it is a much larger download though. When we start using SAX later in the book, we will need a copy of the JDK, so you may choose to download it now.

Follow the instructions on the web page. Once you have downloaded the file, follow the directions to install it. You should write down the location of the folder you install it to, as you will need this later.

### Install the Wrox Package

Once you have downloaded and installed the JRE or JDK you will need to download and install the Wrox validation program. To download the program, go to <http://www.wrox.com/> and download the file validate.zip from the folder for this chapter. The download is a zip archive that will contain all of the files you will need including the Xerces Java Archive (JAR) file. Extract the file to the folder where you have saved your XML documents, in our examples we will use C:\Wrox.

### Modify the Path and Class Path

Finally, we need to be able to find our Java Runtime Environment and our validation classes. To make this easier we will add two system properties PATH and CLASSPATH. Often you will have these two properties already; in that case, we will simply modify them. Modifying system properties is different for every system, so you may need to consult your system's documentation.

On Windows 95, Windows 98, and Windows ME you can modify the path by editing your C:\Autoexec.bat file.

1.

Open the file and add the following lines to the end of the file:

```
SET PATH=%PATH%;C:\Program Files\JavaSoft\JRE\1.3.1\bin;C:\Wrox
```

```
SET CLASSPATH=%CLASSPATH%;C:\Wrox\classes;C:\Wrox\xerces.jar
```

Replace the paths above with the correct paths for your installation.

2.

Once you have added the lines, save and close the file.

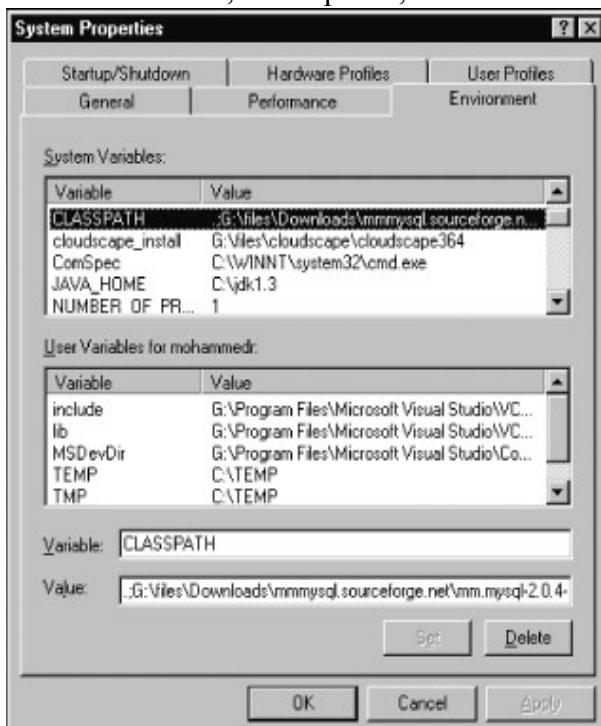
3.

Restart Windows.

4.

On Windows NT and Windows 2000 you can modify the system property by opening the Control Panel and clicking on the System icon:

On Windows NT, once opened, click on the Environment tab. You should see the following window:



On Windows 2000 click on the Advanced tab and then click the Environment Variables button. You should see the following window:



5.

The PATH variable should be located within the System Variables section. Select it and then modify the value by adding

```
C:\Program~1\Javasoft\JRE\1.3.1\bin;C:\Wrox
```

Make sure to include a semicolon between any existing value and the value that we are adding. Replace the paths above with the correct paths for your installation.

6.

The CLASSPATH variable should be located within the User Variables section. Select it, if it is there, otherwise add a new CLASSPATH variable and then modify the value by adding

```
C:\Wrox\classes;C:\Wrox\xerces.jar
```

Again, make sure to include a semicolon between any existing value and the value that we are adding. Also, be sure to replace the paths above with the correct paths for your installation.

At this point, our installation should be complete. We should now have the JRE installed (or the JDK) and ready to use from anywhere on our system. The ability to run Java programs will come in very handy as we make our way through XML as there are many free tools for XML that were built using Java. We should also have our validation program installed and ready for use. We are now ready to validate our XML documents against a DTD-all we need now is a DTD.

### Try It Out-What's in a Name?

In this example, we will embed a DTD that defines our <name> vocabulary directly within our XML document. Later we will see how separating the definition from our XML document can be useful in distributed environments.

1.

Open a text editor, such as Notepad, and type in the following document, making sure you include the spaces as shown. It may be easier to open the name.xml sample from [Chapter 1](#) and modify it as much of the content will remain the same or you may wish to download amended file from the code download for this book at <http://www.wrox.com/>:

```
<?xml version="1.0"?>
<!DOCTYPE name [
  <!ELEMENT name (first, middle, last)>
  <!ELEMENT first (#PCDATA)>
  <!ELEMENT middle (#PCDATA)>
  <!ELEMENT last (#PCDATA)>
]>
<name>
  <first>John</first>
  <middle>Fitzgerald</middle>
  <last>Doe</last>
</name>
```

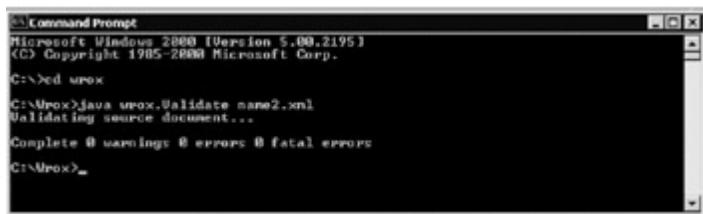
Save the file as name2.xml.

2.

We are ready to begin validating our XML documents against DTDs. Open a command prompt and change directories to the folder where your name2.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate name2.xml
```

You should see the following output:



If you received a Java error, you may need to check the installation. Sometimes this may simply be an issue with the PATH or CLASSPATH. If you think this might be the problem, try moving our PATH and CLASSPATH additions earlier in the list of paths. Also check that you use the correct case when you type wrox.Validate as Java is casesensitive.

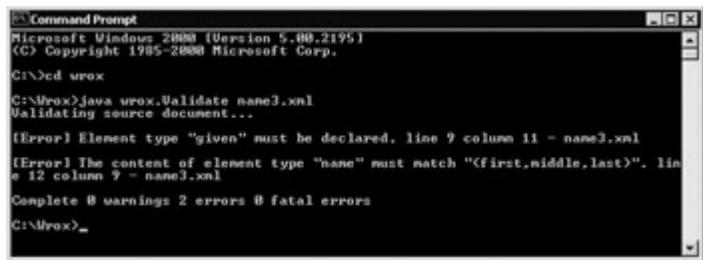
If the output suggests that the validation completed, but that there was an error in the document, correct the error (the parser will report the line number and column number of the error) and try again. When editing XML by hand it is common to make errors when you first begin. Soon you will be able to see an error and correct it preemptively.

### 3.

Change the name of the <first> element to <given> within the name2.xml document:

```
<?xml version="1.0"?>
<!DOCTYPE name [
    <!ELEMENT name (first, middle, last)>
    <!ELEMENT first (#PCDATA)>
    <!ELEMENT middle (#PCDATA)>
    <!ELEMENT last (#PCDATA)>
]>
<name>
    <given>John</given>
    <middle>Fitzgerald</middle>
    <last>Doe</last>
</name>
```

Save the file as name3.xml and try validating again. This time the program should tell us that there were errors:



The program told us that the element "given" was undeclared and that the content of our XML document didn't match what we specified in the DTD.

## How It Works

This Try It Out used our DTD to check that the content within our XML document matched our vocabulary. Internally, parsers will handle these checks in different ways. At the most basic level, the parser will read the DTD declarations and store them in memory. Then as it is reading the document it will validate each element that it encounters against the matching declaration. If it finds an element or attribute that does not appear within the declarations or appears in the wrong position, or if it finds a declaration that has no matching XML content, it will raise a validity error.

Let's break the DTD down into smaller pieces so that we can see some of what we will be learning later:

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

As we have seen in all of our XML documents, we begin with the XML declaration. Again, this is optional but it is highly recommended that you include it to avoid XML version conflicts later.

```
<!DOCTYPE name [
```

Immediately following the XML header is the **Document Type Declaration**, commonly referred to as the DOCTYPE. This informs the parser that there is a DTD associated with this XML Document.

#### Important

You may have noticed that the acronym for Document Type Declaration is also DTD. This is a source of confusion for many new XML users. Luckily, you will almost never see the acronym "DTD" used in reference to the Document Type Declaration. Whenever you see the acronym DTD in this book it will refer to the Document Type Definition as a whole.

If the document contains a DTD the Document Type Declaration must appear at the start of the document (preceded only by the XML header)-it is not permitted anywhere else within the document. The DOCTYPE declaration has an exclamation mark (!) at the start of the element name. If you remember your wellformedness rules from [Chapter 2](#), you may already be telling anyone who will listen that you have discovered an error in your XML book. This is not an error, as the XML Recommendation indicates that **declaration elements** must begin with an exclamation mark. Declaration elements may only appear as part of the DTD. They may not appear within the main XML content. The syntax for DTDs is entirely different from the XML syntax that we have already learned. As we progress through this chapter, you will need to learn many new rules for creating your DTDs. The syntax for DTDs is inherited from SGML.

This is actually a relatively simple DOCTYPE declaration; we will look at some more advanced DOCTYPE declaration features later.

```
<!ELEMENT name (first, middle, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
```

Directly following the DOCTYPE declaration is the body of the DTD. This is where we declare elements, attributes, entities, and notations. In this DTD, we have declared several elements, as that is all we needed for the vocabulary of our <name> document. In the same way as the DOCTYPE declaration, the element declarations also begin with an exclamation mark. At this point, you may begin to notice that the syntax for DTDs is very different from the rules for basic XML documents. The primary reason for this is their foundation in SGML.

```
] >
```

Finally, we close the DTD using a closing bracket and a closing angle bracket. This effectively ends our definition and our XML document immediately follows.

*Now that you have seen a DTD, and have seen a validating parser in action, you may feel ready to create DTDs for all of your XML documents. It is important to remember, however, that validation uses more processing power, even for a small document. Because of this, there may be many circumstances where you do not need a DTD. For example, if you are only using XML documents that are created by your company, or are machine generated (not hand typed), you can be relatively sure that they follow the rules of your vocabulary. In such cases, checking validity may be unnecessary. In fact, it may negatively affect your overall application performance.*

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)

# Sharing Vocabularies

In reality, most DTDs will be much more complex than our first example. Because of this, it is often better to share vocabularies, and use DTDs that are widely accepted. Before you start creating DTDs, it is good to know where you can find existing DTDs. Sharing DTDs not only removes the burden of having to create the declarations; it also allows you to more easily integrate with other companies and XML developers that use shared vocabularies.

Many individuals and industries have developed DTDs that are de facto standards. Scientists use the Chemical Markup Language (CML) DTD to validate documents they share. In the mortgage industry, many businesses utilize the Mortgage Industry Standards Maintenance Organization's (MISMO) DTD when exchanging information. XHTML, the XML version of HTML 4.01 maintains three DTDs; transitional, strict, and frameset. These three DTDs specify the allowed vocabulary for XHTML. Using these, browser developers can ensure that XHTML content is valid before attempting to display it. Can you imagine if every browser maker developed his or her own vocabulary for HTML? The Web would barely function.

There are several good places to check, when trying to find a DTD for a specific industry. The first place to look is, of course, your favorite search engine. Most often, this will turn up good results. Another great place to check is the *Cover Pages*. The *Cover Pages* is a priceless resource of XML information maintained by Robin Cover and can be found at

<http://www.oasis-open.org/cover/xml.html>. In addition to these excellent pages you might also want to check the Dublin Core Initiative, which is an online resource dedicated to creating interoperable standards. The address is <http://www.dublincore.org>. RosettaNet is a consortium of companies that are also working on developing ebusiness standards. You can learn more at <http://www.rosettanet.org>.

The xml?dev mailing list is another excellent resource. You can subscribe to the mailing list at <http://lists.xml.org/archives/>. Be warned, however, that the list is not for the faint of heart; hundreds of mails per week go through this list. The archives of xml?dev are full of announcements from companies and groups that have developed DTDs for widespread use. If you can't find a DTD for your application, create one. If you feel it may be useful to others in your industry, announce it on the list.

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)

# Anatomy of a DTD

Now that we have seen a DTD, let's look at each of the DTD declarations in more detail. DTDs can be broken down into five basic parts:

- Document Type Declaration
- 
- Element Declarations
- 
- Attribute Declarations
- 
- Notation Declarations
- 
- Entity Declarations

## The Document Type Declaration

Perhaps saying that the Document Type Declaration is part of the DTD is a bit incorrect. In fact, it is that which declares that we are specifying a DTD at all and where to find the rest of the definition. In our first example, our Document Type Declaration was simple.

```
<!DOCTYPE name [...]>
```

The basic structure of the Document Type Declaration will always begin in the same way, with `<!DOCTYPE`. There must be some whitespace following the word DOCTYPE as there is after all element names. Also, whitespace is not allowed to appear in between DOCTYPE and the opening "`<!`".

After the whitespace, the name of the XML document's root element must appear. It must appear *exactly* as it will in the document, including any namespace prefix. Because our document's root element was `<name>`, the word "name" follows the opening `<!DOCTYPE` in our declaration.

*Remember, XML is case sensitive. Therefore, any time you see a name in XML it is case sensitive. When we say the name must appear exactly as it will in the document, this includes character case. We will see this throughout our DTD; any reference to XML names implies case sensitivity.*

Following the name of the root element, you have several options for specifying the rest of the Document Type Declaration. In our `<name>` example, our element declarations appeared within our XML Document, between the [ and ] of our Document Type Declaration. Any declarations within the XML document, as in our sample, are the **internal subset**. The XML declaration also allows you to refer to an external DTD. Declarations that appear in an external DTD are the **external subset**. You can refer to an external DTD in one of two ways

- 
- System identifiers
- 
- Public identifiers

## System Identifiers

A **system identifier** allows you to specify the physical location of a file on your system. It is comprised of two parts: the keyword SYSTEM followed by a URI reference to a document with a physical location.

```
<!DOCTYPE name SYSTEM "name.dtd" [...]>
```

This can be a file on your local hard-drive, for example, or, perhaps, a file on your intranet or the Internet. To indicate that you are using a system identifier, you must type the word SYSTEM after the name of the root element in your declaration. Immediately following the SYSTEM keyword is the URI reference to the location of the file in quotation marks. Some valid system identifiers include:

```
<!DOCTYPE name SYSTEM "file:///c:/name.dtd" [...]>
```

```
<!DOCTYPE name SYSTEM "http://sernaferna.com/hr/name.dtd" [...]>
```

```
<!DOCTYPE name SYSTEM "name.dtd">
```

Notice that in the last example we have omitted the [ and ]. This is perfectly normal. Specifying an internal subset is optional. You can use an external and internal subset at the same time. We will look into this later in this chapter. If you do specify an internal subset, it will appear between the [ and ], immediately following the system identifier.

When used in the Document Type Declaration, a system identifier allows you to refer to an external file containing DTD declarations. We will see how to use an external DTD in our next Try It Out but before we do, we'll look at public identifiers.

## Public Identifiers

**Public identifiers** provide a second mechanism to locate DTD resources.

```
<!DOCTYPE name PUBLIC "-//Wrox Press//DTD Name example from Chapter 1//EN">
```

Much like the system identifier, the public identifier begins with a keyword, PUBLIC, followed by a specialized identifier. However, instead of a reference to a file, public identifiers are used to identify an entry in a catalog. According to the XML specification, public identifiers can follow any format, however, there is a commonly used format called **Formal Public Identifiers**, or **FPIs**. The syntax for a FPI is defined in the document ISO 9070. ISO 9070 also defines the process for registration and recording of formal public identifiers.

*The International Standards Organization, or ISO, is a group which designs government approved standards. These standards are numbered and available from the ISO for a fee. Often you can find interpretations of the standards in books or online. You can learn more about the ISO by going to their web site at <http://www.iso.ch/>.*

The syntax for FPIs follows the basic structure:

```
-//Owner//Class Description//Language//Version
```

The delimiter // separates each section. Generally, there are five sections of a FPI:

- 
- Registration Indicator
- 
- Owner
-

## Class Name and Description

- 
- Language
- 
- Display Version

The [first section](#), shown above as a hyphen (-), indicates whether the public identifier has been registered according to the registration procedure outlined in the ISO 9070 standard. If it has been registered a '+' is used and if not, the '-' is used. The second section is the name of the owner of the public identifier. In this case, because we are writing this sample for a book published by Wrox Press, we could use 'Wrox' as the owner. The third section is a short description of what the FPI actually identifies. In this case, we are writing a DTD for our name example, so we have indicated that our class is a DTD. Next, we need to specify the language of our DTD. This is done using a two character language constant defined in ISO 639. Finally, you may include a display version although this is optional and rarely used.

At the most basic level, public identifiers function similarly to namespace names. Public identifiers, however, cannot be used to combine two different vocabularies in the same document. Because of this, namespaces are much more powerful and are the preferred method.

Following the identifier string, you may include an optional URI reference as well. This allows the processor to find a copy of the document if it cannot resolve the public identifier (many processors cannot resolve public identifiers). The URI reference is a separate quoted string that follows the FPI. A valid document type declaration that uses a public identifier might look like:

```
<!DOCTYPE name PUBLIC "-//Wrox//DTD Name example from Chapter 1//EN" "name.dtd">
```

To summarize, the above declaration assumes we are defining a document type for a document whose root element is <name>. The definition has a public identifier of

-//Wrox//DTD Name example from [Chapter 1](#)//EN

and, in case that cannot be resolved, we have provided a relative URI to a file called name.dtd. In the above example, we haven't included an internal subset.

We have mentioned catalogs, registered, and unregistered public identifiers, but are these concepts commonly used in XML development? Yes. In fact, many web browsers, when identifying the versions of an XHTML document, utilize the public identifier mechanism. For example, many XHTML web pages will utilize the public identifier W3C//DTD XHTML 1.0 Strict//EN to identify the DTD associated with the document. When the web browser reads the file, it may use a built-in DTD that corresponds to the public identifier instead of downloading a copy from the Web. This allows web browsers to cache the DTD locally, reducing processing time. When you are developing your applications, you can use this same strategy. Using public identifiers simply gives you a way to identify your vocabulary, just as namespaces do.

Now that we have learned how to use public and system identifiers, let's try to create an external DTD file, and associate it with our XML document. Remember, we said that you may have an internal subset, an external subset, or both. When using an internal subset your DTD declarations will appear within your XML document. When using an external subset, your DTD declarations will appear in a separate external file.

## Try It Out-The External DTD

We have alluded to the idea that external DTDs can be very useful. By utilizing an external DTD, we can easily share

our vocabulary with others in our industry, or even our own company. By this same logic, we can utilize vocabularies that others have already developed, by referring to external files they have created. Therefore, let's reconfigure our <name> example so that our DTD is defined external to our XML document.

1.

We will begin by creating a new document to form our external DTD. Open Notepad and type in the following:

```
<!ELEMENT name (first, middle, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
```

Save the file as name4.dtd. Save it in the same folder as your name3.xml document.

2.

Now let's reopen our name3.xml document from earlier. Modify it as follows:

```
<?xml version="1.0"?>
<!DOCTYPE name PUBLIC "-//Wrox Press//DTD Name example from Chapter 1//EN" "name4.dtd">
<name>
  <first>John</first>
  <middle>Fitzgerald</middle>
  <last>Doe</last>
</name>
```

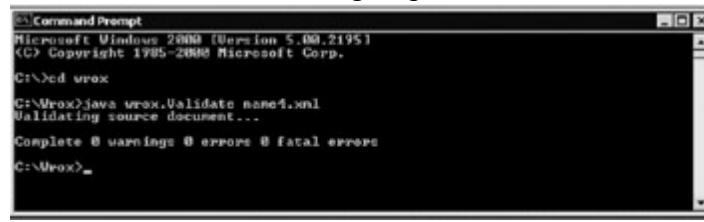
Make sure you also have changed the element <given> back to <first> after our last Try It Out. Save the modified file as name4.xml.

3.

Open a command prompt and change directories to the folder where your name4.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate name4.xml
```

You should see the following output which shows that our validation has been successful:



If you received any errors, check that you have typed everything correctly and try again.

## How It Works

In this Try It Out, we used an external DTD to check our XML content. As you may have guessed, the syntax for the DTD changed very little. The main difference between the internal DTD and external DTD was that there was no DOCTYPE declaration within the external DTD. The DOCTYPE declaration is always located within the XML document. In addition, within our name4.xml document there was no internal subset. Instead, we used a public identifier to indicate what DTD our validation program should use.

Of course, our validation program has no way to resolve public identifiers. The processor instead used the optional URI reference that we included to find the correct DTD for validation. In this case, our XML parser must find the file identified as "name4.dtd". Because this is a relative URL reference (it does not contain a web site address or drive letter), our parser will begin looking in the current directory-where the XML document it is parsing is located. The

XML Recommendation does not specify how parsers should handle relative URL references; however, most XML parsers will treat the path of the XML document as the base path, just as ours has done. It is important that you check your XML parser's documentation before you use relative URL references.

*There are many situations where using external DTDs can be very beneficial. For example, because the DTD appears in only one place it is easier to make changes. If the DTD is repeated in each XML file, upgrading can be much more difficult. Later in the chapter, we will look at XML documents and DTDs that are comprised of many files using entities. You must remember, however, that looking up the DTD file will take extra processing time. In addition, if the DTD file is located on the Internet, you will have to wait for it to download. Often, it is better to keep a local copy of the DTD for validation purposes. If you are maintaining a local copy, you should check for changes to the DTD at the original location.*

## Element Declarations

Throughout the beginning of this chapter, we have seen element declarations in use, but have not yet looked at the element declaration in detail. When using a DTD to define the content of an XML document, you must declare each element that appears within the document. As we will soon see, DTDs may also include declarations for optional elements, elements that may or may not appear in the XML document.

```
<!ELEMENT name (first, middle, last)>
```

Element declarations can be broken down into three basic parts:

- The ELEMENT declaration
- The element name
- The element content model

As we have seen with the DOCTYPE declaration, the ELEMENT declaration is used to indicate to the parser that we are about to define an element. Much like the DOCTYPE declaration, the ELEMENT declaration begins with an exclamation mark. The declaration can appear only within the context of the DTD.

Immediately following the ELEMENT keyword is the name of the element that you are defining. Just as we saw in the DOCTYPE, the element name must appear exactly as it will within the XML document, including any namespace prefix.

*The fact that you must specify the namespace prefix within DTDs is a major limitation. Essentially this means that the user is not able to choose a prefix but must use the prefix defined within the DTD. This limitation exists because the W3C completed the XML Recommendation before finalizing how namespaces would work. As we will see in the [next chapter](#), XML Schema is not limited in this way.*

The content model of the element appears after the element name. An element's **content model** defines the allowable content within the element. An element may be empty, or its content may consist of text, element children, or a combination of children and text. This is essentially the crux of the DTD, where the entire document's structure is defined. As far as the XML Recommendation is concerned, there are four kinds of content model:

- Empty
-

- Element

- Mixed

- Any

Let's look at each of these content models in more detail.

## Empty Content

If you remember from [Chapter 2](#) we learned that some elements may or may not have content:

```
<parody></parody>
<parody/>
```

The `<parody>` element, from [Chapter 2](#), sometimes had content and was sometimes empty. Some elements within our XML documents may **never** need to contain content. In fact, there are many cases when it wouldn't make sense for an element to contain text or elements. Consider the `<br/>` tag in XHTML (it functions in the same way as a `<br>` tag within HTML). The `<br/>` tag allows you to insert a line break in a web page. It would not make much sense to include text within the `<br/>` element. Moreover, no elements would logically fit into a `<br/>` tag. This is a perfect candidate for an empty content model.

To define an element with an empty content model you simply include the word EMPTY following the element name in the declaration:

```
<!ELEMENT br EMPTY>
```

It is important to remember that this will require that the element be empty within the XML document. You should not declare elements that **may** contain content using the EMPTY keyword. For example, the `<parody>` element may or may not contain other elements. As we will see, elements that are not declared with an empty content model may still be empty. Because the `<parody>` element may contain elements, we must declare the element by using an element content model rather than the EMPTY keyword.

## Element Content

As we have already seen, many elements in XML contain other elements. In fact, this is one of the primary reasons for creating XML. We have already seen an element declaration that has element content:

```
<!ELEMENT name (first, middle, last)>
```

In our earlier example, our `<name>` element had, as its children, a `<first>`, `<middle>`, and `<last>` element. When defining an element with element content you simply include the allowable elements within parentheses. Therefore, if our `<name>` element was only allowed to contain a `<first>` element, our declaration would read:

```
<!ELEMENT name (first)>
```

Each element that we specify as a child within our element's content must also be declared within the DTD. Therefore, in our above example, we would also need to define the `<first>` element later for our DTD to be complete. The processor needs this information, so that it knows how to handle the `<first>` element when it is encountered. You may put the ELEMENT declarations in any order you like. As you may have guessed, the element name in our content model must appear exactly as it will in the document, including a namespace prefix if any.

Of course, even in our small example at the start of the chapter our element had more than one child. This will often be the case. There are two fundamental ways of specifying that an element will contain more than one child:

- Sequences
- Choices

## Sequences

Often the elements within your documents must appear in distinct order. If this is the case, you should define your content model using a **sequence**. When specifying a sequence of elements, simply list the element names separated by commas. Again, this will be within the parentheses that immediately follow the name of the element we are declaring. All of our examples that had more than one element have used a sequence when declaring the content model:

```
<!ELEMENT name (first, middle, last)>
```

In the above, our declaration indicates that the `<name>` element must have exactly three children: `<first>`, `<middle>`, and `<last>` and that they must appear in that order. Just as we saw with a single element content model, each of the elements that we specify in our sequence must also be declared in our DTD.

If our XML document were missing one of the elements within our sequence, or if our document contained more than three elements, the parser would raise an error. If all of the elements were included within our XML document, but appeared in another order such as `<last>`, `<middle>`, `<first>` our processor would raise an error.

It is also important to note that in an element-only content model, whitespace doesn't matter. Therefore, using the above declaration, the allowable content for our `<name>` element might appear as:

```
<name>
  <first>John</first>
  <middle>Fitzgerald</middle>
  <last>Doe</last>
</name>
```

Because the whitespace within our element's content doesn't matter, we could also have the content appear as:

```
<name><first>John</first>
<middle>Fitzgerald</middle><last>Doe</last>
</name>
```

The spacing of the elements in an element only content model is only for readability. It has no significance to the validation program.

## Choices

Although we have used sequences throughout this chapter, there may be many circumstances where a sequence doesn't allow us to model our element content. Suppose we needed to allow one element or another, but not both. Obviously, we would need a choice mechanism of some sort. For example, let's say we were designing a DTD that could be used to catalog our music library. Within our catalog, we had several `<item>` elements. Within each `<item>` element we wanted to allow the four choices `<CD>`, `<cassette>`, `<record>`, and `<MP3>`. We could declare our `<item>` element as follows:

```
<!ELEMENT item (CD | cassette | record | MP3)>
```

This declaration would allow our <item> element to contain one <CD> or one <cassette> or one <record> or one <MP3> element. If our <item> element were empty, or if it contained more than one of these elements the parser would raise an error.

The choice content model is very similar to the sequence content model. Instead of separating the elements by commas, however, you must use the vertical bar (|) character. The vertical bar functions as an exclusive OR. An exclusive OR allows one and only one element out of the possible choices.

## Combining Sequences and Choices

We have learned enough to create a DTD for the examples we have seen so far in this chapter. Many XML documents will need to leverage much more complex rules. In fact, using a simple choice or sequence may not allow us to model our document completely. Suppose we wanted to add a <fullname> element to our <name> content model. The new <fullname> element would allow us to specify the entire name at once. Of course, if we specified a <fullname> element, we wouldn't need the <first>, <middle>, and <last> elements.

When creating our <name> declaration we would need to specify that the content may include either a <fullname> element **or** the <first>, <middle>, and <last> sequence of elements, but not both. The XML Recommendation allows us to mix sequences and choices. Because of this, we can declare our model as follows:

```
<!ELEMENT name (fullname | (first, middle, last))>
```

As in our earlier examples, we have enclosed the entire content model with parentheses. In the above example, however, we have a second set of parentheses within the content model. It is good to think of this as a content model within a content model. The inner content model, in the above, is a sequence specifying the elements <first>, <middle>, and <last>. The inner content model appears exactly as it was in our first example. The XML Recommendation allows us to have content models within content models within content models, and so on, infinitely.

The processor handles each inner content model much like a simple mathematical equation. Because the processor handles each model individually, it can treat each model as a separate entity. This allows us to use models in sequences and choices. In the example above, we had a choice between an element and a sequence content model. You could easily create a sequence of sequences, or a sequence of choices, or a choice of sequences-almost any other combination you can think of.

## Mixed Content

The XML Recommendation specifies that any element with text in its content is a **mixed content model** element. Within mixed content models, text may appear by itself or it may be interspersed between elements. In everyday usage, people will refer to elements that can contain only text as "text-only elements" or "text-only content".

The rules for mixed content models are similar to the element content model rules we learned in the [last section](#). We have already seen some examples of the simplest mixed content model-text only:

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

In the above declaration we have specified the keyword #PCDATA within the parentheses of our content model. You may remember from [Chapter 2](#) that PCDATA is an abbreviation for Parsed Character DATA. This simply indicates that there will be parsed character data within our content model. An example element that adheres to the above declaration might look like:

```
<artist>Weird Al Yankovic</artist>
```

Mixed content models may also contain elements interspersed within the text. An XHTML `<p>` tag (functions exactly the same as an HTML `<p>` tag) is a classic example of an element with mixed content:

```
<p>Without question, <b>Weird Al</b> is the <i>greatest</i> singer <b>of all time!</b></p>
```

In the above sample, we have a `<p>` element. Within the `<p>` element, we have interspersed the text with elements such as the `<i>` and the `<b>`.

There is only one way to declare a mixed content model within DTDs. In the mixed content model, we must use the choice mechanism when adding elements. This means that each element within our content model must be separated by the vertical bar (`|`) character:

```
<!ELEMENT p (#PCDATA | b | i)*>
```

In the sample above, we have attempted to describe the `<p>` element (actually, this definition is horribly lacking—there are many more allowable children of a *real* `<p>` element). Notice that we have used the choice mechanism to describe our content model; a vertical bar separates each element. We cannot use commas to separate the choices.

When including elements in the mixed content model, the `#PCDATA` keyword must always appear first in the list of choices. This allows validating parsers to immediately recognize that it is processing a mixed content model, rather than an element content model. Unlike element-only content models, you are not allowed to have inner content models in a mixed declaration.

You should also notice that we have included the `*` outside of the parentheses of our content model. When you are including elements within your mixed content model you are required to include the `*` at the end of your content model. The `*` character is known as a **cardinality indicator**. We will learn more about cardinality indicators shortly.

Because we are using a repeated choice mechanism, we have no control over the order or number of elements within our mixed content. We may have an unlimited number of `<b>` elements, an unlimited number of `<i>` elements, and any amount of text. All of this may appear in any order within our `<p>` element. This is a major limitation of DTDs. In the [next chapter](#), you will learn how XML Schema has improved validation of mixed content models.

To simplify, every time you declare elements within a mixed content model, they *must* follow four rules:

- They must use the choice mechanism to separate elements.
- The `#PCDATA` keyword must appear first in the list of elements.
- There must be no inner content models.
- The `*` cardinality indicator *must* appear at the end of the model.

## Any Content

Finally, we can declare our element utilizing the ANY keyword:

```
<!ELEMENT catalog ANY>
```

In the above, the ANY keyword indicates that any elements declared within the DTD may be used within the content of the <catalog> element, and that they may be used in any order any number of times. The ANY keyword does not allow you to include elements in your XML document that are not declared within the DTD. In addition, any character data may appear within the <catalog> element. Because DTDs are used to restrict content, the ANY keyword is not very popular as it does very little to restrict the allowed content of the element you are declaring.

### Try It Out—"I Want a New DTD"

You are probably ready to try to build a much more complex DTD with all of this newfound knowledge. Unfortunately, our current examples are too simple to really test our skills. Because of this, we are going to need a more complex subject. Weird Al Yankovic is, perhaps, the most complex subject we are likely to find. Let's take our <CD> example from [Chapter 2](#) and create a DTD for our vocabulary. In addition, because we are enormous Weird Al fans, we will extend our original example to allow us to catalog our entire Weird Al music library. In this Try It Out, we will start with the basics and add more features in following examples.

1.

Open the file cd.xml from [Chapter 2](#) and add the highlighted sections:

```
<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM "al.dtd">
<catalog>
  <CD>
    <artist>"Weird Al" Yankovic</artist>
    <title>Dare to be Stupid</title>
    <genre>parody</genre>
    <date-released>1990</date-released>
    <song>
      <title>Like A Surgeon</title>
      <length>
        <minutes>3</minutes>
        <seconds>33</seconds>
      </length>
      <parody>
        <title>Like A Virgin</title>
        <artist>Madonna</artist>
      </parody>
    </song>
    <song>
      <title>Dare to be Stupid</title>
      <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
      </length>
      <parody></parody>
    </song>
  </CD>
</catalog>
```

Save the file as al.xml.

Notice that we have added a document type declaration that refers to an external system file called al.dtd. Also, notice that the root element in our document and the element name within our declaration are the same.

2.

Open a new file in Notepad and save it in the same folder as al.xml. This file will be where we define our DTD in the same way as we did earlier with name4.dtd.

3.

Let's begin writing our DTD. Because we have a sample XML document, we can base most of our

declarations on the text that we have. Many of us were taught that, when programming, we should plan and design first, then implement. Building a DTD based on an existing sample, however, is by far the easiest method available. When designing a DTD it is much easier to create a sample and let the document evolve before the vocabulary is set in stone. Of course, you must remember that some elements may not appear in your sample (such as some elements in choice content models).

In our XML document, <catalog> is the root element. That is the easiest place to start so we will begin by declaring it in our DTD:

```
<!ELEMENT catalog ()>
```

We haven't specified a content model. Looking at our sample document, we can see that the <catalog> element contains a <CD> element. The <CD> element appears to be the only child so this content model should be easy to define:

```
<!ELEMENT catalog (CD)>
```

Of course, because we have specified CD in our content model, we know that we must declare it in our DTD. Let's do that now:

```
<!ELEMENT CD (artist, title, genre, date-released, song, song)>
```

Our <CD> element contains several children <artist>, <title>, <genre>, <date-released> and two <song> elements. Specifying the two <song> elements as we have done is a little clumsy, especially because we don't know how many songs to expect for a given CD. We will improve this content model a little later. Now we must declare each of these elements within our DTD:

```
<!ELEMENT artist (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date-released (#PCDATA)>
```

The first four elements are all text-only elements. The XML Recommendation actually considers text-only a mixed content model.

```
<!ELEMENT song (title, length, parody)>
```

Even though we have two <song> elements within our <CD> content model, both function the same. Because of this, we only declare the <song> element once. In fact, the XML Recommendation does not allow you to declare two elements with the same name inside a DTD. Because of this each time you use an ELEMENT declaration within the DTD it must have the same content model.

Because we have already declared a <title> element, we do not need to declare it again. Instead we will move on to the <length> element:

```
<!ELEMENT length (minutes, seconds)>
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
```

At this point, declaring the elements should begin to feel repetitive. Finally, we will declare the <parody> element.

```
<!ELEMENT parody (title, artist)>
```

We have already declared the <title> and <artist> elements so at this point we have completed the DTD.

The final DTD should look like:

```
<!ELEMENT catalog (CD)>
<!ELEMENT CD (artist, title, genre, date-released, song, song)>
<!ELEMENT artist (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date-released (#PCDATA)>
<!ELEMENT song (title, length, parody)>
<!ELEMENT length (minutes, seconds)>
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
<!ELEMENT parody (title, artist)>
```

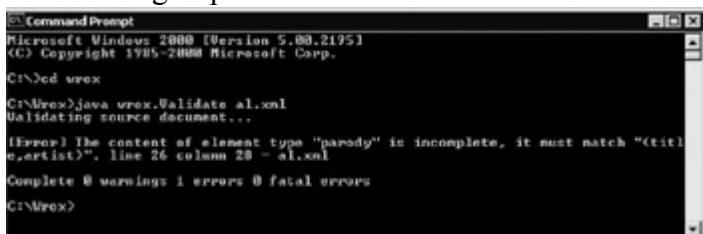
Save the file as al.dtd.

4.

Open a command prompt and change directories to the folder where your al.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate al.xml
```

The resulting response should be:



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>cd wrox
C:\wrox>java wrox.Validate al.xml
Validating source document...
[Error] The content of element type "parody" is incomplete, it must match "(title, artist)". line 26 column 28 - al.xml
Complete 0 warnings 1 errors 0 fatal errors
C:\wrox>
```

If you typed everything correctly, you should have received the above error. Why did we get this error? Looking back at our XML document reveals the problem: in our second <song> element, the song we described wasn't a parody. Because it wasn't a parody we didn't include a <title> or <artist> element. Our processor raised an error because it was expecting to see <title> and <artist> elements. It expected to see these elements because we included them in the sequence content model in our parody element declaration.

How can we fix the problem? Unfortunately, we can't yet. What we need is a way to tell the processor that the (title, artist) sequence may appear once or not at all. We must learn how to tell the processor how many times the elements will appear.

## Cardinality

An element's **cardinality** defines how many times it will appear within a content model. Each element within a content model may have an indicator of how many times it will appear following the element name. DTDs allow four indicators for cardinality:

Indicator	Description
[none]	As we have seen in all of our content models thus far, when no cardinality indicator is used it indicates that the element must appear once and only once. This is the default behavior for elements used in content models.
Indicator	Description

?	Indicates that the element may appear either once or not at all.
+	Indicates that the element may appear one or more times.
*	Indicates that the element may appear zero or more times.

Let's look at these indicators in action:

```
<!ELEMENT parody (title?, artist?)>
```

Suppose we used cardinality indicators when declaring the content for our `<parody>` element. By including a '?' when specifying our title and artist elements, we inform the processor that these elements may appear once or not at all within our content model. If we were to validate the document again, the parser would not raise validity errors if the `<title>` element and `<artist>` elements were missing. With our new declaration, the only allowable `<parody>` elements include:

```
<parody>
  <title>Like A Virgin</title>
  <artist>Madonna</artist>
</parody>
<parody>
  <title>Like A Virgin</title>
</parody>
<parody>
  <artist>Madonna</artist>
</parody>
<parody></parody>
```

In each of the above cases, we can see that the `<title>` element may or may not appear. In addition, the `<artist>` element may or may not appear. In the final example, we see that the `<parody>` element can be completely empty, where neither element appears. An important point to remember is that, because we used the '?' cardinality indicator, both the `<title>` and `<artist>` elements may appear within our `<parody>` element at most one time. It is also important to notice that when both elements do appear, they must be in the order that we defined within our sequence.

#### Important

It is important to remember that the cardinality indicator only affects the content model where it appears. Even though we specify that the `<title>` element within the `<parody>` content model may appear once or not at all, this does not affect the declaration of the `<title>` element, or any other use of the `<title>` element in our DTD. It helps to remember that no cardinality indicator means that you are using the default (once and only once). The `<title>` element is used in three places within our DTD, but the only time where it may appear once or not at all is within our `<parody>` content model.

We could now get our example working by changing our content model as we have seen. Does the new model really represent what we wanted though? Of course, because it lets us have an empty `<parody>` element, it solves the problem at hand. In reality though, are there any circumstances where we would want to know only the title or only the artist of the parodied song? Probably not. We need to define our content model so that we will get both the `<title>` and `<artist>` elements or neither one.

Luckily for us, the XML Recommendation allows us to apply cardinality indicators to content models as well. Remember, that earlier we said that a content model would appear within parentheses, and that content models may contain inner content models. Because of this, we can change our <parody> declaration one more time:

```
<!ELEMENT parody (title, artist)?>
```

The ? indicator is functioning exactly as it did earlier. This time, however, we are indicating that the entire sequence may appear once or not at all. When we declare the <parody> element using this new model, the only allowable <parody> elements include:

```
<parody>
  <title>Like A Virgin</title>
  <artist>Madonna</artist>
</parody>
<parody></parody>
```

This solves our problem completely as it represents our desired content model perfectly. Before we go back to our example though, let's look at some other ways we could spruce up our catalog DTD.

While we were designing our DTD, originally, we noticed that we also had a problem with the declaration of our <CD> element. Within its content model, we had to specify that there were only two songs. This was a major problem! As promised, we can now fix this problem using a cardinality indicator.

Let's examine what it is that we want to accomplish. We know that every CD we document will have at least one song. We also know that each CD will usually have more than one song on it. We don't know how many songs each CD will have, and, in fact, each CD will probably have a different number of songs. We need to use a cardinality indicator that specifies the <song> element may appear one or more times within our <CD> element. The '+' indicator does just that:

```
<!ELEMENT CD (artist, title, genre, date-released, song+)>
```

With the above declaration, we can document one or more songs per CD, which is exactly what we want to do.

Perhaps the largest deficiency remaining in our catalog DTD is that it isn't much of a catalog. Currently our DTD only allows one <CD> element to appear as a child of the <catalog> element. This won't let us get very far in documenting our entire music library. We need to indicate that the <CD> element may appear zero, once, or many times. Fortunately the '\*' cardinality indicator does just that. We could improve our DTD by changing our earlier <catalog> declaration:

```
<!ELEMENT catalog (CD*)>
```

Because cardinality indicators can affect content models as well, we could have just as easily changed our declaration to:

```
<!ELEMENT catalog (CD)*>
```

Just as we saw earlier with the '?' indicator, cardinality indicators may appear on elements or content models. In the first example the '\*' indicator tells the processor that the CD element may appear zero or more times. In the second example the '\*' indicator specifies that the entire content model may appear zero or more times. Because there is only one element within the content model, these two declarations function in exactly the same way.

This might be good enough for our own music library, but it would be nice to have some more options. Some of us (in fact, most of us) may have MP3s, cassette tapes, and record albums lying around that we would like to catalog. It

would be nice if we could include a <CD>, a <cassette>, a <record> or an <MP3> element within our catalog. We already know how to create a choice content model:

```
<!ELEMENT catalog (CD | cassette | record | MP3)>
```

Unfortunately, this declaration limits us to **one** <CD> or **one** <cassette> or **one** <record> or **one** <MP3> within our catalog. This seems like a job for a cardinality indicator. The '\*' indicator will work perfectly in this situation. Consider the following declaration:

```
<!ELEMENT catalog (CD | cassette | record | MP3)*>
```

We have placed the '\*' outside of the parentheses. This indicates that we want the entire content model to appear zero or more times. In this case, though, we have a choice content model. This means that we will have the choice between these four elements zero or more times. This definition allows us to have as many <CD>, <cassette>, <record> and <MP3> elements as we want-and they can appear in any order.

When we specify cassette, record, and MP3 within our content model, we need to declare the <cassette>, <record> and <MP3> elements within our DTD. They should have the same content as our <CD> element; when we modify our DTD, we can just copy the <CD> element's declaration:

```
<!ELEMENT cassette (artist, title, genre, date-released, song+)>
<!ELEMENT record (artist, title, genre, date-released, song+)>
<!ELEMENT MP3 (artist, title, genre, date-released, song+)>
```

## Try It Out-"I Want a New DTD"-Part 2

Now that we have learned how to correct and improve our DTD, let's get down to business and integrate the changes we have been going over.

1.

Open the file al.dtd and modify the highlighted sections:

```
<!ELEMENT catalog (CD | cassette | record | MP3)*>
<!ELEMENT CD (artist, title, genre, date-released, song+)>
<!ELEMENT cassette (artist, title, genre, date-released, song+)>
<!ELEMENT record (artist, title, genre, date-released, song+)>
<!ELEMENT MP3 (artist, title, genre, date-released, song+)>
<!ELEMENT artist (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date-released (#PCDATA)>
<!ELEMENT song (title, length, parody)>
<!ELEMENT length (minutes, seconds)>
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
<!ELEMENT parody (title, artist)?>
```

Save the file as al2.dtd.

2.

Of course, now that we have created a new DTD file, we will need to update our XML document to refer to it. Open al.xml and modify the DOCTYPE declaration so that it refers to our new DTD and save the file as al2.xml.

```
<!DOCTYPE catalog SYSTEM "al2.dtd">
```

3.

Open a command prompt and change directories to the folder where your al2.xml document is located. At the prompt type the following and then press the Enter key:

> **java wrox.Validate al2.xml**

Your output should show a complete validation without errors.

If you received any errors this time, check that you have typed everything correctly and try again.

## How It Works

This Try It Out implements much of what we learned throughout this section. To sum it up, we set out to design a DTD that described a catalog for all of our Weird Al albums. We used an assortment of complex content models so that our DTD would reflect various XML documents. Of course, when we first began designing our DTD we didn't include many options (in fact, there were even a couple of errors). After we had the basic structure designed, we modified our DTD to correct some problems and even insert some features. The design strategy is very common among XML developers.

### Important

Some XML designers have taken this design strategy a step further. Instead of relying only on an example XML document, they will use complex UML diagrams or other types of visual aid. As we will see in the [next chapter](#), new syntaxes have evolved based on this strategy of using an example document to describe the vocabulary. For instance, Examplotron uses a syntax in which the example essentially is the declaration. More information on Examplotron can be found at <http://examplotron.org/>

Now that we have a firm grasp on how to declare elements within our DTD, we will turn our attention to attributes.

## Attribute Declarations

Attribute declarations are in many ways similar to element declarations. Instead of declaring content models for allowable elements, however, DTDs allow you to declare a list of allowable attributes for each element. These lists are called ATTLIST declarations.

```
<!ELEMENT catalog (CD | cassette | record | MP3)*>
<!ATTLIST catalog name CDATA #IMPLIED>
```

Above, we have an element declaration for our `<catalog>` element from our catalog example. Following the element declaration, we have an ATTLIST declaration that declares the allowable attributes of our `<catalog>` element. This particular ATTLIST declares only one attribute, `name`, for the `<catalog>` element.

An ATTLIST declaration can be broken down into three basic parts:

- The ATTLIST keyword
- The associated element's name
-

## The list of declared attributes

Just as we have seen in all of our other declarations, the ATTLIST declaration begins with an exclamation mark to indicate that it is part of the DTD. Immediately following the ATTLIST keyword is the name of the associated element. In our example above, the name of our associated element is catalog. We indicated that we were building a list of attributes for a <catalog> element by including the word catalog immediately following the ATTLIST keyword.

Following the ATTLIST name, we declared each attribute in the list. An ATTLIST declaration may include any number of attributes. Each attribute in the list can be broken down into three parts:

- The attribute name
- 
- The attribute type
- 
- The attribute value declaration

Let's look at each section of name attribute declaration:

```
name CDATA #IMPLIED
```

In the above declaration, the name of the attribute is, ironically, name. We have declared that our name attribute may contain character data by using the CDATA keyword-this is our attribute's type. Finally, we have declared that our attribute has no default value, and further, that this attribute does not need to appear within our element using the #IMPLIED keyword. The third part of this attribute declaration is known as the **value declaration**-it controls how the XML parser will handle the attribute's value. We will look at value declaration options in more detail a little later.

## Attribute Names

We learned, in [Chapter 2](#), that attribute names are very similar to element names. We must follow the basic XML naming rules when declaring an attribute. In addition to the basic naming rules, we must also make sure that we don't have duplicate names within our attribute list. Remember duplicate attribute names are not allowed within a single element.

As far as DTDs are concerned, namespace declarations, such as: xmlns:xhtml="<http://www.w3.org/1999/xhtml>" are also treated as any other attributes. Although the Namespace Recommendation treats xmlns attributes only as declarations, DTDs must declare them in an ATTLIST declaration if they are used. Again, this is because the W3C finalized the syntax for DTDs before the Namespace Recommendation was completed.

To declare an attribute name you simply type the name exactly as it will appear in the XML document, including any namespace prefix.

## Attribute Types

When declaring attributes, you can specify how the processor should handle the character data that appears in the value. So far, we haven't seen anything like this in DTDs. Within our element declarations, we could specify that an element contained text-but we couldn't specify how the processor should treat the value. Because of this, several powerful XML features are available using attributes.

Let's look at the different attribute types:

Type	Description
CDATA	Indicates that the attribute value is character data
ID	Indicates that the attribute value uniquely identifies the containing element
IDREF	Indicates that the attribute value is a reference, by ID, to a uniquely identifiable element
IDREFS	Indicates that the attribute value is a whitespace separated list of IDREF values
ENTITY	Indicates that the attribute value is a reference to an external unparsed entity (we will learn more about entities later). The unparsed entity might be an image file or some other external resource such as an MP3 or binary file
ENTITIES	Indicates that the attribute value is a whitespace separated list of ENTITY values
NMTOKEN	Indicates that the attribute value is a name token. An NMTOKEN is a string of character data comprised of standard name characters
NMTOKENS	Indicates that the attribute value is a whitespace separated list of NMTOKEN values
Enumerated List	Apart from using the default types, you can also declare an enumerated list of possible values for the attribute value.

As we saw in our example, the attribute type immediately follows the attribute name. Let's look at each of these types in more detail.

## CDATA

CDATA is the default attribute type. As the most basic of the data types, a processor will do no additional type checking on a CDATA attribute. Of course, the XML wellformedness rules still apply; but as long as the content is well formed, a validating parser will accept any text as CDATA.

## ID, IDREF and IDREFS

Attributes of type ID can be used to uniquely identify an element within an XML document. Once you have uniquely identified the element, you can later use an IDREF to refer to that element. As we will soon see, identifying elements is paramount in many XML technologies such as XSLT and XPath. Many of you may have already seen an ID mechanism in action. Within HTML, many elements can be identified with an ID attribute. Often JavaScript code will access the elements by their ID.

There are several rules to remember when using ID attributes:

- The value of an ID attribute must follow the rules for XML names
- The value of an ID attribute must be unique within the entire XML Document
-

Only one attribute of type ID may be declared per element

- 

The attribute value declaration for an ID attribute must be #IMPLIED or #REQUIRED

Let's look at an example document:

```
<?xml version="1.0"?>
<!DOCTYPE names [
    <!ELEMENT names (name)*>
    <!ELEMENT name (first, middle?, last)>
    <!ATTLIST name social ID #REQUIRED>
    <!ELEMENT first (#PCDATA)>
    <!ELEMENT middle (#PCDATA)>
    <!ELEMENT last (#PCDATA)>
]>
<names>
    <name social="_111-22-3333">
        <first>John</first>
        <middle>Fitzgerald</middle>
        <last>Doe</last>
    </name>
    <name social="_777-88-9999">
        <first>Jane</first>
        <middle>Mary</middle>
        <last>Doe</last>
    </name>
</names>
```

Are our values for the social attributes within our sample document valid? If we look at the internal subset, we should see that we have declared the social attribute as an ID attribute. The first thing you should notice about our ID values is that they begin with underscores. Remember that XML names must begin with a letter, an underscore or a colon. If we had simply used the numeric social security number it would have been an invalid ID because XML names are not allowed to begin with numeric digits. Prepending each value with an underscore makes each value legal. As we look through our XML document, we only find two ID values and they are unique. It is important to remember that *any* attribute value of type ID must be unique—even if the attribute is part of an element with a different name. We haven't declared more than one ID attribute type in a single element. When we did declare our social attribute, we made sure to include the #REQUIRED keyword.

When we use IDREF attributes, the rules are similar:

- 

The value of an IDREF attribute must follow the rules for XML names

- 

The value of an IDREF attribute must match the value of some ID within the XML document

Often we need to refer to a list of elements. For example, imagine we needed to have a list of all of the people that worked in an office. We might want to use an IDREFS attribute store with a list of whitespace separated IDREF values that referred to the social ID attributes defined in our <name> element.

## ENTITY and ENTITIES

Attributes may also include references to **unparsed entities**. An unparsed entity is an entity reference to an external file that the processor cannot parse. For example, in XHTML images are unparsed entities; instead of actually including the image inside the markup, we use <img> tags to refer to the external resource. In XML we can declare reusable references inside our DTD using an ENTITY declaration. We haven't seen ENTITY declarations yet, so we

will cover this in more detail a little later in this chapter.

For now, let's cover the rules for ENTITY attribute types. In ENTITY attributes, you must refer to an ENTITY that has been declared somewhere in the DTD. In addition, because you are referring to an ENTITY, the value must follow the rules for XML names. Consider the following attribute declaration:

```
<!ATTLIST CD image ENTITY #IMPLIED>
```

By declaring an image attribute as we have done, we can then refer to an ENTITY within our XML document:

```
<CD image="PolkaPartyCoverArt">
```

Our image attribute refers to an ENTITY that is named PolkaPartyCoverArt. This assumes that we have declared the ENTITY somewhere in our DTD. You should also notice that our value follows the rules for XML names, it begins with a letter and contains valid name characters.

The [ENTITIES](#) attribute type is simply a whitespace separated list of ENTITY values. Therefore, we could declare the following:

```
<!ATTLIST CD images ENTITIES #IMPLIED>
```

A valid use of the above declaration might look like:

```
<CD images="PolkaPartyCoverArt.front  
PolkaPartyCoverArt.back">
```

The ENTITY names are still valid (remember it is legal to use a period in an XML name) and they are separated by whitespace. In fact, a line feed and several spaces appear between the two values. This is legal-the XML processor doesn't care *how much* whitespace separates two values. The processor considers spaces, tabs, line feeds, and carriage return characters whitespace.

## NMTOKEN and NMTOKENS

Often we will need to have attributes that refer to a name. This might be a reference to an element name, an entity name, an attribute name or even a person's name. The NMTOKEN type allows us to use any name as a value. As long as the value follows the rules for an XML name then the processor will treat it as valid. You do not need to declare the name that an NMTOKEN attribute uses; it only has to follow the rules for XML names.

The one exception is that an NMTOKEN value may begin with any name character. When we learned the rules for XML names, we learned that our names are not allowed to begin with a numerical digit. NMTOKEN values are not required to adhere to this rule.

Suppose we added a type attribute to our <catalog> element to allow us to specify what kind of catalog we created:

```
<!ATTLIST catalog type NMTOKEN #IMPLIED>
```

The following value would be allowable:

```
<catalog type="music">
```

As we have seen with other attribute types, NMTOKENS is simply a whitespace separated list of NMTOKEN values. We could declare the type attribute to allow multiple catalog types as follows:

```
<!ATTLIST catalog type NMOKENS #IMPLIED>
```

The following value would be allowable:

```
<catalog type="music movies books">
```

We haven't declared any of these values within our DTD; they simply follow the rules for NMOKEN values.

## Enumerated Attribute Types

Clearly, the ability to check types within attribute values is indispensable. Suppose you only want to allow a certain set of values in your attribute? We could use our existing types to restrict our attribute value, but it may not give us enough control. Take our <name> example, for instance. Let's say we wanted to add an attribute called title. We could use the new title attribute to indicate how the person should be addressed. We might expect to see the values, "Mr.", "Mrs.", "Ms.", "Miss", "Dr.", "Rev". All of these values are character data, so we could use the CDATA type. Of course, we could also receive the value "42", because it is character data. This isn't what we want at all. Instead, we could use the NMOKEN attribute type because all of our choices are valid NMOKEN values. Of course, this would also allow values like "Polka". We need to limit the values that are allowed for our attribute with even greater control.

An **enumerated list** allows us to do just that. When we declare our attribute, we can specify a list of allowable values. Let's see what a declaration for our title attribute would look like using an enumerated list:

```
<!ATTLIST name title (Mr. | Mrs. | Ms. | Miss | Dr. | Rev) #IMPLIED>
```

Again, the whitespace within the declaration does not matter. Here we see that we have listed out all of our possible values within parentheses. All of our possible values are separated by the vertical bar character (|). This declaration indicates that the value of our title attribute must match one (and only one) of the listed values. Each item in the list must be a valid NMOKEN value. Remember the NMOKEN type functions much like an XML name, but NMOKEN values may begin with numerical digits.

Some *valid* uses of the new title attribute would be:

```
<name title="Mr.">  
<name title="Miss">
```

Some *invalid* values would be:

```
<name title="Sir">  
<name title="DR.">
```

The first value is invalid because it attempts to use a value that is not in the list. The second value is not valid because, although "Dr." appears in the list of allowed values, "DR." does not. Remember that, because XML is case sensitive the values in your list will be case sensitive as well.

Within the XML Recommendation, there are actually two kinds of enumerated attribute types. The first, as we have just seen, uses valid names for the list of possible values. The second uses valid NOTATION values for the list. Much like an ENTITY, a NOTATION must be declared within the DTD. NOTATION declarations are often used in conjunction with external files, as they are typically used to identify file types. We will look into NOTATION declarations at the end of this chapter. For now, just be aware that this type of declaration is allowed:

```
<!ATTLIST image type NOTATION (jpeg | gif | bmp) #REQUIRED>
```

Again, we list the possible values within the parentheses. In a NOTATION enumeration though, simply insert the keyword NOTATION before the list of possible values. This will indicate to the processor that it should check that each value within the list is a valid NOTATION that has been declared within the DTD. This example assumes that our DTD contains declarations for the three notation types jpeg, gif, and bmp. By using a NOTATION enumeration, the parser may pass along additional information about the enumerated type. Again, we will see some of the benefits of using notations later in the chapter.

## Attribute Value Declarations

Within each attribute declaration, you must also specify how the value will appear in the document. Often we will want to provide a default value for our attribute declaration. At times we may simply want to require that the attribute be specified in the document, still other times we may need to require that the value of the attribute be fixed at a given value. Each attribute can be declared with these properties.

The XML Recommendation allows us to specify that the attribute:

- Has a default value
- Has a fixed value
- Is required
- Is implied

## Default Values

Sometimes we need to provide a value for an attribute even if it hasn't been included in the XML document. By specifying a **default value** for our attributes, we can be sure that it will be included in the final output. As the document is being processed, a validating parser will automatically insert the attribute with the default value if the attribute has been omitted. If the attribute has been included in the document, the parser will use the included attribute. Remember, only validating parsers make use of the information within the DTD, and therefore the default value will only be used by a validating parser. The ability to specify default values for attributes is one of the most valuable features within DTDs.

Specifying a default attribute is easy; simply include the value in quotation marks after the attribute type:

```
<!ATTLIST name title (Mr. | Mrs. | Ms. | Miss | Dr. | Rev) "Mr.">
```

We have modified our title attribute so that it uses a default value. The default value we have selected is "Mr.". When a validating parser is reading our <name> element, if the title attribute has been omitted, the parser will automatically insert the attribute title with the value "Mr.". If the parser does encounter a title attribute within the <name> element, it will use the value that has been specified within the document.

When specifying a default value for your attribute declarations you must be sure that the value you specify follows the rules for the attribute type that you have declared. For example, if your attribute type is NMTOKEN then your default value must be a valid NMTOKEN. If your attribute type is CDATA then your default value can be any wellformed XML character data.

You are not permitted to specify a default value for attributes of type ID. This may seem strange at first, but actually makes a good deal of sense. If a validating parser inserted the default value into more than one element, the ID would no longer be unique throughout the document. Remember, an ID value must be unique-if two elements had an ID attribute with the same value the document would not be valid.

## Fixed Values

There are some circumstances where an attribute's value must always be fixed. When an attribute's value can never change we use the #FIXED keyword followed by the **fixed value**. Fixed values operate much like default values. As the parser is validating the file, if the fixed attribute is encountered then the parser will check that the fixed value and attribute value match. If they do not match, the parser will raise a validity error. If the parser does not encounter the attribute within the element, it will insert the attribute with the fixed value.

A common use for fixed attributes is specifying version numbers. Often DTD authors will fix the version number for a specific DTD:

```
<!ATTLIST catalog version CDATA #FIXED "1.0">
```

Like default values, when specifying values in fixed attribute declarations, you must be sure that the value you specify follows the rules for the attribute type you have declared. In addition, you may not specify a fixed value for an attribute of type ID.

## Required Values

When you specify that an attribute is **required**, it must be included within the XML document. Often a document must have the attribute to function properly, at other times it is simply a matter of exercising control over the document content. Imagine you are developing a program that can display images. Within your XML document, you have <image> elements that refer to external image files. In order for your application to display the images, it needs to know what kind of image is being referenced; and without this information, the file cannot be displayed.

```
<!ATTLIST image format NOTATION (jpeg | gif | bmp) #REQUIRED>
```

In the example above, we have specified that the format attribute must appear within the <image> element in the document. If our parser encounters an <image> element without a format attribute as it is processing the document it will raise an error.

To declare that an attribute is required, simply add the keyword #REQUIRED immediately after the attribute type. When declaring that an attribute is required, you are not permitted to specify a default value.

## Implied Values

In most cases the attribute you are declaring won't be required, and often won't even have a default or fixed value. In these circumstances, the attribute may or may not occur within the element. These attributes are called **implied attributes**, as there is sometimes no explicit value available. When the attributes do occur within the element, a validating parser will simply check that the value specified within the XML document follows the rules for the declared attribute type. If the value does not follow the rules, the parser will raise a validity error.

When declaring an attribute you must always specify a value declaration. If the attribute you are declaring has no default value, has no fixed value, and is not required then you must declare that the attribute is **implied**. You can declare that an attribute is implied by simply adding the keyword #IMPLIED after the attribute's type declaration:

```
<!ATTLIST CD images ENTITIES #IMPLIED>
```

## Specifying Multiple Attributes

So far, our ATTLIST declarations have been limited. In each of the above examples, we have only declared a single attribute. This is good, but many elements will need more than one attribute. No problem, the ATTLIST declaration allows you to declare more than one attribute. For example:

```
<!ATTLIST catalog version CDATA #FIXED "1.0"
    name CDATA #IMPLIED >
```

In the above ATTLIST declaration for our `<catalog>` element, we have included a version and a name attribute. Our version attribute is a fixed character data attribute; our name attribute is also character data but is optional. When declaring multiple attributes, as we have in this example, simply use white space to separate the two declarations. In the above, we have used a line feed and have aligned the attribute declarations with some extra spaces. This type of formatting is common when declaring multiple attributes. While, you can declare more than one attribute within an ATTLIST declaration, you are only permitted to declare one ATTLIST for each ELEMENT declaration.

## Try It Out—"I Want a New DTD"-Part 3

Now that we have seen some common attribute declarations, let's revisit our music catalog example and add some improvements. As we can now declare attributes, we'll include several that can be used within our catalog. We will add a version attribute, a name attribute, and a gender attribute.

1.

Let's begin by opening our `a12.xml` file. We will modify our DOCTYPE declaration again, and add some attributes and then save the file as `a13.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM "a13.dtd">
<catalog version="1.0" name="My Music Library">
    <CD>
        <artist>"Weird Al" Yankovic</artist>
        <title>Dare to be Stupid</title>
        <genre>parody</genre>
        <date-released>1990</date-released>
        <song>
            <title>Like A Surgeon</title>
            <length>
                <minutes>3</minutes>
                <seconds>33</seconds>
            </length>
            <parody>
                <title>Like A Virgin</title>
                <artist gender="female">Madonna</artist>
            </parody>
        </song>
        <song>
            <title>Dare to be Stupid</title>
            <length>
                <minutes>3</minutes>
                <seconds>25</seconds>
            </length>
            <parody></parody>
        </song>
    </CD>
</catalog>
```

2.

Now that we have modified our XML document. Let's declare these new attributes within our DTD. Open `a12.dtd`, make the following modifications and save the file as `a13.dtd`:

```
<!ELEMENT catalog (CD | cassette | record | MP3)*>
```

```
<!ATTLIST catalog version CDATA #FIXED "1.0"
      name CDATA #IMPLIED >
<!ELEMENT CD (artist, title, genre, date-released, song+)>
<!ELEMENT cassette (artist, title, genre, date-released, song+)>
<!ELEMENT record (artist, title, genre, date-released, song+)>
<!ELEMENT MP3 (artist, title, genre, date-released, song+)>
<!ATTLIST CD images ENTITIES #IMPLIED>
<!ATTLIST cassette images ENTITIES #IMPLIED>
<!ATTLIST record images ENTITIES #IMPLIED>
<!ATTLIST MP3 images ENTITIES #IMPLIED>
<!ELEMENT artist (#PCDATA)>
<!ATTLIST artist gender (male | female) "male">
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date-released (#PCDATA)>
<!ELEMENT song (title, length, parody)>
<!ELEMENT length (minutes, seconds)>
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
<!ELEMENT parody (title, artist)?>
```

3.

Open a command prompt and change directories to the folder where your al3.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate al3.xml
```

You should see that complete validation statement again.

If you received any errors this time, check that you have typed everything correctly and try again.

## How It Works

In this Try It Out example we added several ATTLIST declarations to our DTD. We added the attributes version and name to our <catalog> element. We could use the version attribute to indicate to an application what version DTD this catalog matches. Using the name attribute, we can provide a friendly description of our whole catalog.

We also added an images attribute to our <CD>, <cassette>, <record>, and <MP3> elements. Notice that we had to declare the ATTLIST for the images attribute once for each element. The images attribute was designed to be type **ENTITIES**. We haven't learned how to declare **ENTITIES** yet, so we didn't use these attributes in our al3.xml document. We declared that the images attribute is implied, because it may or may not appear within our document.

Finally, we modified our <artist> element, adding the gender attribute. The gender attribute allows us to determine the gender of our artist. Because there are only two choices for the value of the gender attribute, we decided to use an enumerated list. We also set the default value to "male" because our favorite artist is male and we don't want to type it repeatedly. You should notice that we didn't include the gender attribute when referring to Weird Al. Because we have omitted the gender attribute, the processor, as it is parsing, will automatically insert the attribute with the default value. When we referred to Madonna, however, we needed to include the gender attribute because she is female and the default value was "male".

---

[◀ PREVIOUS](#)

[\*\*< Free Open Study >\*\*](#)

[NEXT ▶](#)

# Entities

In [Chapter 2](#), we learned that we could escape characters, or use entity references to include special characters within our XML document. We learned that there are five **entities** built into XML that allow us to include characters that have special meaning in XML documents. In addition to these built-in entities, we also learned that we could utilize character references to include characters that are difficult to type, such as the © character:

```
<value>5 &gt; 4</value>  
  
<copyright>&#169; 2001 Wrox Press</copyright>
```

In our first example, we have included an &gt; entity reference within our element content. This allows us to include a "<" character without our XML parser treating it as the start of a new element. In our second example we have included an &#169; character reference within our element content. This allows us to include the "©" character, by specifying the character's Unicode value.

In fact, entities are not limited to simple character references within our XML documents. Entities can be used throughout the XML document to refer to sections of replacement text, other XML markup, and even external files. We can separate entities into four primary types, each of which may be used within an XML Document:

- Built-in Entities
- 
- Character Entities
- 
- General Entities
- 
- Parameter Entities

Let's look at each of these in more detail.

*In fact, technically, an entity is any part of an XML document. For example, the root element within an XML document is called the **document entity**. Of course, you cannot use these entities as you can use the four entity types we have listed so their usefulness is, therefore, limited.*

## Built-in Entities

We have already seen that there are five entities that can be used within an XML document by default.

- &amp;#xA0;the & character
- &lt;?the < character
- &gt;?the > character

- &apos;?the ' character
- &quot;?the " character

These five entities are often called **built-in entities**, because, according to the XML Recommendation, all XML parsers must support the use of these five entities by default. You are not required to create declarations for them in the DTD. We will soon see that there are other kinds of entities that require you to declare them first within the DTD before they are used within the document.

## References to Built-in Entities

To use an entity, we must include an entity reference within our document. An **entity reference**, as the name implies, *refers* to an entity that represents a character, text, or even an external file. A reference to a built-in entity follows the pattern:

```
&quot;
```

The reference begins with the ampersand (&) character. Immediately following the ampersand is the name of the entity to which we are referring, in this case quot. At the end of the reference is a semicolon (;). Whitespace is not allowed anywhere within the reference.

In general, you may use entity references anywhere you could use normal text within the XML document. For example, you can include entity references within element contents and attribute values. You can also use entity references within your DTD within text values, such as default attribute values. Although the built-in entities allow you to include characters that are used in markup, they cannot be used in place of XML markup. For example, the following is *legal*:

```
<artist name="&quot;Weird Al&quot; Yankovic"/>
```

Here, we used the &quot; built in entity so that we could include the quotation mark as part of "Weird Al" Yankovic's name. This is allowed because it is within the attribute value. On the other hand, the following would be *illegal*:

```
<company name=&quot;Way Moby&quot;/>
```

In the above, the &quot; entity is used in place of actual quotation marks. As an XML parser processed the element, it would encounter the "&" after the "=" and immediately raise a well-formedness error. The XML within the document is first checked for well-formedness errors *and then* entity references are resolved. Many XML parsers will check the well-formedness of a specific section of an XML Document and then begin replacing entities within that section. This can be very useful in large documents. You should consult your XML parser's documentation for more information. It is also important to note that you may not use entities within names of elements or attributes.

## Character Entities

**Character entities**, much like the five built-in entities, are not declared within the DTD. Instead, they can be used in the document within element and attribute content, without any declaration. References to character entities are often used for characters that are difficult to type, or for non-ASCII characters.

## References to Character Entities

Again, to utilize a character entity within your document you must include an entity reference. The syntax for character entity references is very similar to the five built in entities:

&#169;

As you can see from the above, the primary difference in character entity references is that there is no entity name. The reference begins with the ampersand (&) character. However, instead of an entity name, we have a hash mark (#) followed by a number, in this case "169", which is the Unicode value for the "©" character. At the end of the reference is a semicolon (;). Just as we saw in our references to built-in entities, whitespace is not allowed anywhere within the character entity reference.

You may also refer to a character entity by utilizing the hexadecimal Unicode value for the character:

&#xA9;

Here, we have used the hexadecimal value "A9" in place of the decimal value "169". When the value you are specifying is hexadecimal, you must include a lowercase "x" before the value, so that the XML parser will know how it should handle the reference. In fact, it is much more common to utilize the hexadecimal form, because the Unicode specification lists characters using hexadecimal values.

Just as we saw with built-in entity references, character entity references may be used anywhere you could use normal text, such as element content and attribute values. You can also use them within your DTD. You cannot use character entities in place of actual XML markup, or as part of the names of elements or attributes.

#### Important

Does this mean that, by using character references, you can include *any* Unicode character in your XML document? Not exactly. Actually, you are only permitted to include characters specified within the XML Recommendation, which was based on Unicode 3.0. As the Unicode specification has evolved, the need to utilize more characters in XML has also grown. Recently, XML experts, in a series of discussions known as the "Blueberry Debates", considered expanding the list of allowable XML characters. In order to expand the list it may require an XML version change?this is why it is important that you include the XML version in the header at the start of your documents ? to ensure that they are backwards compatible. The current list of allowable XML characters can be found in the XML Recommendation at <http://www.w3.org/TR/REC-xml#NT-Char> and <http://www.w3.org/TR/REC-xml#CharClasses>. If an XML parser encounters a character (or character entity reference) that is not allowed, the parser should immediately raise a fatal error. Illegal characters are considered well-formedness errors.

## General Entities

General entities function very similarly to the five built-in entities, however, general entities must be declared within the DTD before they can be used within the XML document. Most commonly, XML developers use **general entities** to

create reusable sections of **replacement text**. Instead of representing only a single character, general entities may represent characters, paragraphs, and even entire documents. Throughout this section, we will learn the many uses of general entities.

There are two ways to declare general entities within the DTD. You can specify the value of the entity directly in the declaration, or you may refer to an external file. Let's begin by looking at an **internal entity declaration**:

```
<!ENTITY producer "Produced by Way Moby, 1985">
```

Just as we have seen with our earlier ELEMENT and ATTLIST declarations, the ENTITY declaration begins with an exclamation mark. Following the ENTITY keyword is the name of the entity, in this case producer. We will use this name when referring to the entity elsewhere in the XML document. The name must follow the rules for XML names, just as we have seen throughout this chapter. After the entity name, in the above declaration, is a line of replacement text. Whenever an XML parser encounters a reference to this entity, it will substitute the replacement text at the point of the reference. The above example is an internal entity declaration because the replacement text appears directly within the declaration in the DTD.

In the above example, our replacement text value is "Produced by Way Moby, 1985". Way Moby is the record label that produces many of Weird Al's albums. We declared this text in an entity as we might use it often within our XML document. General entity values are not limited to simple characters or text values, however. Within a general entity, the replacement text may consist of any well-formed XML content. The only exception to this rule is that you are not required to have one root element within the replacement text. All of the following are examples of *legal* general entity values:

```
<!ENTITY weirdal "&quot;Weird Al&quot; Yankovic">
<!ENTITY parody "<genre>parody</genre>">
<!ENTITY johndoe "<first>John</first>
    <middle>Fitzgerald</middle>
    <last>Doe</last>">
```

Notice that we have included entity references within our replacement text. Just as we said, entity references may be used within your DTDs in place of normal text (default attribute values and entity replacement text values). Also, notice that our values may or may not have a root element, or may have no elements at all. Although we have included entity references within our replacement text, we should note that an entity is not permitted to contain a reference to itself either directly or indirectly. The following declarations are *not legal*:

```
<!ENTITY startSong "<song>">
<!ENTITY endSong "</song>">
<!ENTITY recursive "This is &recursive;">
```

The first two examples are not legal because they are not well-formed. In the first declaration, we have specified the start of a `<song>` element but have not included the closing tag. In the second declaration, we only have the closing tag of a `<song>` element. You are not permitted to begin an element in one entity and end it in another?each entity must be wellformed on its own. The third entity contains a reference to itself within its replacement text. When an entity refers to itself, it is known as a **recursive entity reference**.

You might want to store your replacement text in an external file instead of including it within the DTD. This can be very useful when you have a large section of replacement text. As we have seen, there are no limits on the length of replacement text. Your DTD can quickly become cluttered by sections of replacement text, making it more difficult to read. When declaring your entities, instead of declaring the replacement text internally, you can refer to external files. When the replacement text for an entity is stored externally, the entity is declared using an **external entity declaration**. For example, we could declare our entities as follows:

```
<!ENTITY lyrics SYSTEM "lyrics.txt">
```

or

```
<!ENTITY lyrics PUBLIC "?//Wrox//Text Lyrics for an album//EN" "lyrics.txt">
```

Just as we saw with our Document Type Declaration, when referring to external files we can use a system identifier or a public identifier. When we use a public identifier, we may also include an optional URI reference, which we have done in our example above.

In each of these declarations, we have referred to an external file named lyrics.txt. As an XML parser is processing the DTD, if it encounters an external entity declaration it may open the external file and parse it. If the XML parser is a validating parser, it must open the external file, parse it, and be able to utilize the content when it is referenced. If the XML parser is not a validating parser, it may or may not attempt to parse the external file.

#### Important

The XML Recommendation makes the distinction between validating and non-validating parsers primarily to make it easier to create XML parsers that conform to the XML specification. Many XML parsers don't include the ability to validate a document against a DTD because of the additional processing or programming time it requires. Many of these same parsers will include the ability to use external entities, however, because of the added functionality. If you are using a non-validating parser, you should check the documentation to see if it can parse external entities.

Remember, just as we saw with our internal entity declaration, the replacement text must be well-formed XML (with the exception of requiring a single root element). If the parser encounters a well-formedness error within the external file, it will raise the error and discontinue parsing.

Sometimes, you will need to refer to an external file that you do not want the XML Parser to parse. For example, if the external file you are referring to is binary, such as an image or MP3, the parser will surely raise an error when it attempts to parse it. You can indicate that an external entity declaration should not be parsed by including the NDATA keyword and notation type at the end of the declaration:

```
<!ENTITY albumCover SYSTEM "file:///c:/Wrox/image.bmp" NDATA bmp >
<!ENTITY albumCover PUBLIC "?//Wrox Press//Image Bitmap image//EN"
"file:///c:/Wrox/image.bmp" NDATA bmp >
```

Notice that within the above ENTITY declarations we have again used the external entity declaration form by including either a system or public identifier. In addition, we have included the keyword NDATA and the notation type bmp. The keyword NDATA is an abbreviation for "notation data". It is used to indicate that the entity will not be parsed; instead, the entity is simply used to provide a reference, or **notation**, to the external file. Following the NDATA keyword was the notation type bmp. The notation type is used to indicate the kind of file we are referring to. The type must be declared within the DTD using a NOTATION declaration, which we will learn about in detail later.

## References to General Entities

Now that we have seen how to declare entities within our DTD, let's look at how to refer to them within our document:

```
&lyrics;
```

The entity reference looks very similar to the built-in entity references we learned about earlier. Again, the reference begins with the ampersand (&) character. Immediately following the ampersand is the name of the entity to which we are referring, in this case lyrics. At the end of the reference is a semicolon (;). Whitespace is not allowed anywhere within the reference. You may refer to any general entity that you have declared within your DTD, as we have above. When the parser encounters the reference, it will include the replacement text that is declared within the DTD or the external file to which the entity declaration refers.

The only exception to this rule is that you cannot create an entity reference to an unparsed external entity, such as an image or MP3 file. Obviously, if the external data is not parsed, it cannot be used as replacement text within the XML document.

Now that we have seen the basics of how to declare and refer to general entities, let's look at an example that uses them.

#### Try It Out?"I Want a New DTD"?Part 4

Let's rework our catalog example so that each of our <song> elements can contain the lyrics to the song we are including in our XML markup. Of course, typing in the lyrics for every song might become cumbersome, so we will create some external entities to refer to files on the Web. Because Weird Al is our favorite artist, most of our lyrics files will come from the web site <http://www.yankovic.org/Category/PLyrics.html>. This means that you will need a live Internet connection to test this example and the ones that follow.

1.

Let's begin by declaring an element for our lyrics. We will also need to declare the new entities within our DTD. Open a13.dtd, make the following modifications and save the file as a14.dtd:

```
<!ELEMENT catalog (CD | cassette | record | MP3)*>
<!ATTLIST catalog version CDATA #FIXED "1.0"
          name CDATA #IMPLIED >
<!ELEMENT CD (artist, title, genre, date?released, song+)>
<!ELEMENT cassette (artist, title, genre, date?released, song+)>
<!ELEMENT record (artist, title, genre, date?released, song+)>
<!ELEMENT MP3 (artist, title, genre, date?released, song+)>
<!ATTLIST CD images ENTITIES #IMPLIED>
<!ATTLIST cassette images ENTITIES #IMPLIED>
<!ATTLIST record images ENTITIES #IMPLIED>
<!ATTLIST MP3 images ENTITIES #IMPLIED>
<!ELEMENT artist (#PCDATA)>
<!ATTLIST artist gender (male | female) "male">
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date?released (#PCDATA)>
<!ELEMENT song (title, length, parody, lyrics)>
<!ELEMENT length (minutes, seconds)>
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
<!ELEMENT parody (title, artist)?>
<!ELEMENT lyrics (#PCDATA)>
<!ENTITY LikeASurgeon SYSTEM
"http://www.yankovic.org/WeirdAl/c_Dare_To_Be_Stupid/like_a_surgeon.txt">
<!ENTITY DareToBeStupid SYSTEM
"http://www.yankovic.org/WeirdAl/c_Dare_To_Be_Stupid/dare_to_be_stupid.txt">
```

2.

Next, open a13.xml from our last example. We will add the <lyrics> element to the songs in our catalog, and utilize references to our newly-defined entities. We will also need to change our Document Type Declaration to refer to our new DTD. Once you have completed these modifications, save the file as a14.xml:

```
<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM "al4.dtd">
<catalog version="1.0" name="My Music Library">
  <CD>
    <artist>"Weird Al" Yankovic</artist>
    <title>Dare to be Stupid</title>
    <genre>parody</genre>
    <date?released>1990</date?released>
    <song>
      <title>Like A Surgeon</title>
      <length>
        <minutes>3</minutes>
        <seconds>33</seconds>
      </length>
      <parody>
        <title>Like A Virgin</title>
        <artist gender="female">Madonna</artist>
      </parody>
      <lyrics>&LikeASurgeon; </lyrics>
    </song>
    <song>
      <title>Dare to be Stupid</title>
      <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
      </length>
      <parody></parody>
      <lyrics>&DareToBeStupid; </lyrics>
    </song>
  </CD>
</catalog>
```

### 3.

Make sure that you are connected to the Internet. Open a command prompt and change directories to the folder where your al4.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate al4.xml
```

This should validate in the way we have come to expect.

If you receive the message "Connection refused: connect" it is because the application could not retrieve the lyrics documents from the Web. Remember, our entity declarations refer to files stored on the Internet; in order to retrieve them you must have an active Internet connection. If you received any other errors, check that you have typed everything correctly and try again.

### 4.

Just to prove that the text has been retrieved from the Web and inserted into our XML document, open up al4.xml in Internet Explorer. Here's a section of what you should see:



## How It Works

In this Try It Out, we introduced a <lyrics> element into our catalog. In order to save on some typing we created entities to refer to files stored on the Internet that contain the lyrics. As our XML parser is processing the file, it encounters the entity declarations, reads the system identifier and attempts to retrieve the files. Once it retrieves the files, it parses the content and stores a copy in memory so that it can replace any references to the entities in our document with the correct replacement text.

It is important to note that we chose to use text files on the Internet?not HTML files. Remember the parser must parse the external document and check it for wellformedness. Most HTML files on the Web are not wellformed XML; we chose text files because we wanted to make sure that the file would not create a wellformedness error when it was parsed. In addition, the ELEMENT declaration for our lyrics element specifies that it contains only #PCDATA. If our XML parser encountered an <html> element within the <lyrics> element, even as the result of an entity's replacement text, it would raise a validity error because we have not declared an <html> element within our DTD.

*Earlier we mentioned that validation uses more processing power, and that this may be a drawback to using DTDs. Likewise, using external entities may also decrease your application performance. You may have noticed a significant performance decrease in our last example. Because external files must be opened and read, and often downloaded from the Internet, you should consider the pros and cons of using external entities before dividing your DTD into separate modules.*

## Parameter Entities

**Parameter entities**, much like general entities, allow you to create reusable sections of replacement text. So far, we have seen that we can refer to entities within element and attribute content, and within specific places within the DTD, such as default attribute values and within entity replacement text. Instead of declaring parameter entities for use with general content, you may only use parameter entities within the DTD. Unlike other kinds of entities, the replacement text within a parameter entity can be made up of DTD declarations or pieces of declarations.

Parameter entities can also be used to build DTDs from multiple files. This is often helpful when different groups work on DTDs. In addition, as we mentioned at the beginning of the chapter, this allows you to reuse DTDs and portions of DTDs in your own XML documents. When XML documents or DTDs are broken up into multiple files, they are said to be **modular**.

Parameter entity declarations are very similar to general entity declarations:

```
<!ENTITY % defaultTitle "Mr.">
```

Here we have declared an internal parameter entity named defaultTitle. We can tell that this is a parameter entity because of the percent sign (%) before the name of the entity. This is the primary difference between the format of parameter entity declarations and general entity declarations. Notice that there is a space between the ENTITY keyword and the percent sign?also there is a space between the percent sign and the name of the entity. This whitespace is required.

Like general entities, parameter entities may also refer to external files utilizing a system or public identifier. For example:

```
<!ENTITY % module SYSTEM "module.dtd">
```

or

```
<!ENTITY % module PUBLIC "?//Wrox Press//DTD External module//EN" "module.dtd">
```

Parameter entity declarations that refer to external files must refer to files that can be parsed by the XML parser. Earlier we learned that general entity declarations may use the keyword NDATA to indicate that the external file should not be parsed. Parameter entity declarations cannot use the NDATA keyword.

## References to Parameter Entities

When referring to a parameter entity within a DTD, the syntax will change slightly. Instead of using an ampersand (&) you must use a percent sign (%). For example:

```
%module;
```

As we can see, the reference consists of a percent sign (%), followed by the entity name, followed by a semi-colon (;). References to parameter entities are only permitted within the DTD.

Consider our <catalog> example, in which we declared the content models for our <CD>, <cassette>, <record>, and <MP3> elements:

```
<!ELEMENT CD (artist, title, genre, date?released, song+)>
<!ELEMENT cassette (artist, title, genre, date?released, song+)>
<!ELEMENT record (artist, title, genre, date?released, song+)>
<!ELEMENT MP3 (artist, title, genre, date?released, song+)>
```

The content model for each of these element declarations is the same. We could save ourselves some typing by utilizing a parameter entity for the content model.

```
<!ENTITY % albumContentModel "(artist, title, genre, date?released, song+)">
```

In the above, we have declared a parameter entity called albumContentModel. We can tell that this is a parameter entity because of the percent sign in between the keyword ENTITY and the name. We know that this is an internal parameter entity because our replacement text is included in the declaration. Utilizing a parameter entity reference, we can modify our earlier element declarations:

```
<!ELEMENT CD %albumContentModel;>
<!ELEMENT cassette %albumContentModel;>
<!ELEMENT record %albumContentModel;>
<!ELEMENT MP3 %albumContentModel;>
```

In a simple case such as this one, our DTD becomes much easier to read. We can quickly see that all four of these elements have exactly the same content model. In addition, if we need to make change to the content model, we only have to modify the ENTITY declaration.

## Try It Out?"I Want a New DTD"?Part 5

Let's take what we have just learned and use it within our catalog DTD. This will allow us to reduce the number of repeated content models within our DTD.

1.

Let's begin by making the appropriate modifications to our DTD file. Open a4.dtd, make the highlighted modifications and save the file as a5.dtd:

```
<!ELEMENT catalog (CD | cassette | record | MP3)*>
<!ATTLIST catalog version CDATA #FIXED "1.0"
                    name CDATA #IMPLIED >
<!ENTITY % albumContentModel "(artist, title, genre, date?released, song+)">
<!ELEMENT CD %albumContentModel;>
<!ELEMENT cassette %albumContentModel;>
```

```
<!ELEMENT record %albumContentModel;>
<!ELEMENT MP3 %albumContentModel;>
<!ATTLIST CD images ENTITIES #IMPLIED>
<!ATTLIST cassette images ENTITIES #IMPLIED>
<!ATTLIST record images ENTITIES #IMPLIED>
<!ATTLIST MP3 images ENTITIES #IMPLIED>
<!ELEMENT artist (#PCDATA)>
<!ATTLIST artist gender (male | female) "male">
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date?released (#PCDATA)>
<!ELEMENT song (title, length, parody, lyrics)>
<!ELEMENT length (minutes, seconds)>
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
<!ELEMENT parody (title, artist)?>
<!ELEMENT lyrics (#PCDATA)>
<!ENTITY LikeASurgeon SYSTEM
"http://www.yankovic.org/WeirdAl/c_Dare_To_Be_Stupid/like_a_surgeon.txt">
<!ENTITY DareToBeStupid SYSTEM
"http://www.yankovic.org/WeirdAl/c_Dare_To_Be_Stupid/dare_to_be_stupid.txt">
```

## 2.

Next, we will need to change our XML file to refer to our new DTD. This is the only change we will need to make within our XML document. Open a14.xml from our last example, change the Document Type Declaration to refer to our new DTD, and save the file as a15.xml:

```
<!DOCTYPE catalog SYSTEM "a15.dtd">
```

## 3.

Open a command prompt and change directories to the folder where your a15.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate a15.xml
```

Your output should show a full validation without errors.

If you receive the message "Connection refused: connect" it is because the application could not retrieve the lyrics documents from the Web. If you do not have an active Internet connection, you can remove the lyric entities and entity references for the sake of this example. If you received any other errors, check that you have typed everything correctly and try again.

## How It Works

In this last Try It Out, we were able to simplify several of our ELEMENT declarations by utilizing a parameter entity for the content model. Just as we have seen throughout this section, parameter entities allow us to reuse DTD declarations or pieces of declarations. As the parser attempts to process the content model for our <CD> declaration, it encounters the parameter entity reference. It replaces the entity reference with the replacement text specified in the ENTITY declaration. It is important to note that the declaration of a parameter entity must occur in the DTD before any references to that entity.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Notation Declarations

Throughout this chapter, we have seen references to **notation declarations**. So far, we have seen how to use notations within specialized ATTLIST declarations. We have also seen that we can use notations when declaring unparsed entities. Often, we will need to refer to external resources, such as image files and databases, which cannot be processed by an XML parser. To make use of the external files we will need an application that can utilize the data. Notation declarations allow us to associate types of external resources with external applications that can handle them.

The basic format of a notation declaration is:

```
<!NOTATION bmp SYSTEM "file:///c:/windows/paint.exe">
```

In the above, we have indicated that this is a notation declaration by beginning our element with an exclamation mark and by using the NOTATION keyword. Following the NOTATION keyword is the name of our notation, in this case bmp. Notation names must follow the same rules as element names. Immediately after the notation name, we must include a SYSTEM or PUBLIC identifier that allows the processor to locate an external application for handling the notation. In this case, we have used a SYSTEM identifier that refers to paint.exe.

A notation, such as the one we have declared, could be used whenever we need to refer to a bitmap file. Because our XML processor cannot parse bitmap files, we will need to use an external program for displaying or editing them. The external program could be any program you like, even web applications. When the parser encounters a usage of the notation name, it will simply provide the path to the application.

## Try It Out-"I Want a New DTD"-Part 6

Let's look at our media library example one last time. Now that we understand entities and notations we should be able to complete our DTD.

1.

Let's begin by opening our a15.dtd file, add some declarations and save the file as a16.dtd:

```
<!NOTATION bmp SYSTEM "explorer.exe">
<!ENTITY DareToBeStupidCoverPhoto SYSTEM "cover1.bmp" NDATA bmp>
<!ELEMENT catalog (CD | cassette | record | MP3)*>
<!ATTLIST catalog version CDATA #FIXED "1.0"
      name CDATA #IMPLIED >
<!ENTITY % albumContentModel "(artist, title, genre, date-released, song+)">
<!ELEMENT CD %albumContentModel;>
<!ELEMENT cassette %albumContentModel;>
<!ELEMENT record %albumContentModel;>
<!ELEMENT MP3 %albumContentModel;>
<!ATTLIST CD images ENTITIES #IMPLIED>
<!ATTLIST cassette images ENTITIES #IMPLIED>
<!ATTLIST record images ENTITIES #IMPLIED>
<!ATTLIST MP3 images ENTITIES #IMPLIED>
<!ELEMENT artist (#PCDATA)>
<!ATTLIST artist gender (male | female) "male">
<!ELEMENT title (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT date-released (#PCDATA)>
<!ELEMENT song (title, length, parody, lyrics)>
<!ELEMENT length (minutes, seconds)>
```

```
<!ELEMENT minutes (#PCDATA)>
<!ELEMENT seconds (#PCDATA)>
<!ELEMENT parody (title, artist)?>
<!ELEMENT lyrics (#PCDATA)>
<!ENTITY LikeASurgeon SYSTEM
"http://www.yankovic.org/WeirdAl/c_Dare_To_Be_Stupid/like_a_surgeon.txt">
<!ENTITY DareToBeStupid SYSTEM
"http://www.yankovic.org/WeirdAl/c_Dare_To_Be_Stupid/dare_to_be_stupid.txt">
<!ENTITY WeirdAl '"Weird Al" Yankovic'>
```

## 2.

Now that we have modified our DTD. Let's use the changes within our XML document. Open al5.xml, make the following modifications and then save the file as al6.xml:

```
<<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM "al6.dtd">
<catalog version="1.0" name="My Music Library">
  <CD images="DareToBeStupidCoverPhoto">
    <artist>&WeirdAl;</artist>
    <title>Dare to be Stupid</title>
    <genre>parody</genre>
    <date-released>1990</date-released>
    <song>
      <title>Like A Surgeon</title>
      <length>
        <minutes>3</minutes>
        <seconds>33</seconds>
      </length>
      <parody>
        <title>Like A Virgin</title>
        <artist gender="female">Madonna</artist>
      </parody>
      <lyrics>&LikeASurgeon;</lyrics>
    </song>
    <song>
      <title>Dare to be Stupid</title>
      <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
      </length>
      <parody></parody>
      <lyrics>&DareToBeStupid;</lyrics>
    </song>
  </CD>
</catalog>
```

## 3.

Open a command prompt and change directories to the folder where your al6.xml document is located. At the prompt type the following and then press the Enter key:

```
> java wrox.Validate al6.xml
```

Your output should show that the validation completed successfully once more.

If you received any errors this time, check that you have typed everything correctly and try again.

## How It Works

We have finally completed a DTD for our music library. Although it is short, it uses some very complex features of DTDs. In this final Try It Out example, we added a NOTATION and several ENTITY declarations to our DTD; we also modified our XML document to utilize these new features.

We began by adding a notation declaration to our DTD. The notation declaration allowed us to refer to external bitmap files for the covers. We also associated bitmap files with explorer.exe, because we know that explorer.exe can open and display the external files. Whenever this notation is used, the parser will report the associated program as well. Our application can use this information to provide a view of the external files. Once we had created a notation declaration for bitmap images, we were able to create an unparsed entity that referred to an external image file. We declared an unparsed entity called DareToBeStupidCoverPhoto that referenced an external file called cover1.bmp. In this case, our processor doesn't care that the file doesn't exist. If we were using a parser that used the notation information, it might raise a warning.

Finally, we declared a parsed general entity called WeirdAl. Knowing that our music library is filled with albums by Weird Al, we decided to create an entity to avoid retyping. Instead, we refer to the entity in the XML document and the parser inserts the replacement text.

---

[!\[\]\(9df8d770acdc32acf90176ef0d17a455\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(2e08eb129f6b838fa1c004896cdd1efd\_img.jpg\) NEXT >](#)

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)

# Developing DTDs

Most of the DTDs we developed within this chapter were relatively simple. As you begin developing DTDs for your XML documents, you may find it is difficult to present the DTDs in a linear order. Most of our declarations flowed in order but often you will not be sure in what order your DTD declarations should occur. Don't worry, apart from entities that are used within the DTDs, declarations can appear in any order. While you have ample freedom in the order you choose, it is common to keep associated declarations near one another. For example, in most DTDs, an ATTLIST declaration will immediately follow the ELEMENT declaration with which it is associated.

As the flow of the DTDs becomes difficult to follow, it is important to document your declarations. You may use XML comments and processing instructions within a DTD, following rules similar to usage in XML content. Comments and processing instructions may appear in the internal or external subsets, but they cannot appear within markup declarations.

For example, the following is valid:

```
<!--name : allows you to name the catalog -->
<!ATTLIST catalog name CDATA #IMPLIED>
```

The following is not valid:

```
<!ATTLIST catalog
  <!--name : allows you to name the catalog -->
  name CDATA #IMPLIED>
```

When developing DTDs you should also note that comments and processing instructions are never declared.

As we have already seen, developing a DTD is easiest when you have an example XML document. What should you do if you have a very long example file with many elements? The best strategy is to divide the DTD into pieces or modules. The best way to do this is by using external parameter entities. Instead of designing the whole DTD at once, try to create DTDs for subsections of your vocabulary and then use parameter entity references when testing. Once you have your DTD working, you can combine the modules to increase performance. By dividing your DTD in this way, you can quickly identify and fix errors.

[◀ PREVIOUS](#)[< Free Open Study >](#)[NEXT ▶](#)

# DTD Limitations

Throughout this chapter, we have seen some of the many benefits of using DTDs. They allow us to validate our content without application specific code, allow us to supply default values for attributes, and even create modular XML documents. Throughout your XML career, you will use existing DTDs and often design your own. Because of XML's strong SGML foundation, much of the early XML development focused on the markup of technical documents. Since that time, XML has been used in areas no one ever expected. While this was a great achievement for the XML community, it began to reveal some limitations of DTDs.

Some limitations of DTDs include:

- DTD syntax is different from XML syntax
- Poor support for XML namespaces
- Poor data typing
- Limited content model descriptions

Before we look at these limitations in more detail, it is important to reiterate: even with their limitations, DTDs are a fundamental part of the XML Recommendation. DTDs will continue to be used in many diverse situations, even as other methods of describing documents emerge.

## DTD Syntax

Throughout this chapter, we have learned a new syntax for expressing DTD declarations. This syntax is different from the generic XML syntax we learned in the first few chapters. Why is the syntax so different? Early on, we learned that XML is based on SGML. Because many of the developers turning to XML used SGML, the creators of XML chose to adopt the DTD syntax that was originally developed for SGML.

This proved to be both a benefit and a limitation within XML. Initially this made migration from SGML to XML a snap. Many users had already developed DTDs for their SGML documents. Instead of having to completely redesign their vocabularies, they could reuse what they had already done with minimal changes. As support for XML grew, new XML tools and standards were developed that allowed users to manipulate their XML data. Unfortunately, these tools were meant for generic XML, not for DTDs.

## XML Namespaces

Throughout this chapter, we have seen the limitations of namespaces in DTDs. Whenever element or attribute names are declared the namespace prefix and colon must be included in the declaration. In addition, DTDs must treat namespace declarations as attributes. This is because the completion of the XML Recommendation occurred before the syntax for XML namespaces was finalized. Forcing users to declare namespace prefixes in advance defeats the purpose of namespaces altogether. Merging documents from multiple namespaces when the prefixes are predefined can be hazardous as well.

## Data Typing

As XML developers began using DTDs to model more complex data (such as databases and programming objects) the need for stronger data types emerged. Within DTDs the only available data types are limited to use in attribute declarations, and even then provide only a fraction of the needed functionality. There is no method for constraining the data within a text-only element to a specific type. For example, if you were modeling a database and needed to specify that data within a specific element needed to be numeric, you couldn't use DTDs.

## Limited Content Model Descriptions

In addition to needing more advanced data types, limitations in content model descriptions soon became apparent. Developers wanted the capability to mimic object inheritance in their XML content models. Developers also found the cardinality operators limiting. DTDs lack strict control on the number of times an element occurs.

---

[PREVIOUS](#)

[< Free Open Study >](#)

[NEXT](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

By utilizing DTDs, we can easily validate our XML documents against a defined vocabulary of elements and attributes. This reduces the amount of code needed within our application. Instead, a validating XML parser can be used to check that the contents of our XML document are valid according to the declarations within our DTD. DTDs allow us to exercise much more control over our document content than simple wellformedness checks.

In this chapter, we learned:

- How to validate a document against a DTD
- How to create element declarations
- How to create attribute declarations
- How to create entity declarations
- How to create notation declarations
- How to specify an XML document and DTD using external files

We also learned that DTDs have several limitations. In the [next chapter](#), we will see how these limitations have been addressed in newer standards, such as XML Schemas.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 6: XML Schemas

## Overview

In the [last chapter](#), we learned that many people use DTDs to validate their XML documents. This keeps us from needing to write application specific code to check that our documents are valid. We also saw some of the limitations of DTDs. Since the inception of XML, several new formats have been developed that allow us to define the content of our vocabulary.

In 1999, the W3C began to develop XML Schemas in response to the growing need for a more advanced format for describing XML documents. Already, work had begun on several efforts that were intended to better model the types of document that were being created by XML developers. The W3C's effort took the best of these early technologies and then added additional features that could be used to fully describe any well-formed XML document. During the development, several members of the W3C designed simpler schema languages with fewer features outside of the W3C. We will discuss some of the more popular schema languages in the [next chapter](#).

XML Schemas reached recommendation status in May 2001. Since that time, support for XML Schemas has rapidly grown, and they have quickly made progress towards fulfilling their original potential. At this point, there are a variety of parsers that allow you to validate an XML document against an XML Schema definition.

### Important

So, what is a schema? A schema is any type of model document that defines the structure of something. In this case, our "something" is an XML document. Many of you may have dealt with database schemas—the documents that are used to define the structure of the database tables and fields. In fact, DTDs are forms of schemas. Throughout this book, we have been using the term "vocabulary" where we could have used the word "schema". So, what is an XML Schema? This is where it gets confusing. The term **XML Schema** is used to refer to the specific W3C XML Schema technology. W3C XML Schemas, much like DTDs allow you to describe the structure for an XML document. Properly, when referring to this technology, you should capitalize the "S" in "Schema". XML Schema definitions are also commonly referred to as XSD.

In this chapter we will learn:

- The benefits of XML Schemas
- How to create and use XML Schemas
- Some of the basic features of XML Schemas

•  
How to document our XML Schemas

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Benefits of XML Schemas

At this point, you already have invested time in learning DTDs. You know the syntax and can create complex, even modular, definitions for your vocabulary. Although XML Schemas are the next great thing, it may be helpful for you to understand some of the benefits of XML Schemas before jumping in.

- XML Schemas are created using XML, not an alternative SGML syntax
- XML Schemas fully support the Namespace Recommendation
- XML Schemas allow you to validate text element content based on built-in and user-defined data types.
- XML Schemas allow you to more easily create complex and reusable content models
- XML Schemas allow you to model concepts such as object inheritance and type substitution

Let's look at some of these in more detail.

## XML Schemas Use XML Syntax

In the [last chapter](#), we spent most of our time learning the DTD syntax. The syntax, as we learned, doesn't follow the basic rules for XML well-formedness. Instead, you must declare your DTDs using a separate syntax. When defining an XML Schema, the syntax is entirely in XML; although you still have to learn the rules for which elements and attributes are required in given declarations, you can use XML tools that have no understanding of the rules to process XML Schema documents. As we learn new XML technologies through this book, we will see how to apply them to any XML document. Unfortunately, powerful tools, such as XSLT, often cannot be used on DTDs.

## XML Schema Namespace Support

Because XML Schemas were finalized after the Namespace Recommendation, the concept of namespaces was available to be included in the design (for a refresher on namespaces, review [Chapter 3](#)). Unlike DTDs, which do not support the full functionality of namespaces, XML Schemas enable you to define vocabularies that utilize namespace declarations. More importantly, XML Schemas allow you to mix namespaces in XML documents with less rigidity. For example, when designing an XML Schema, it is not necessary to specify namespace prefixes, as you must in DTDs. Instead, the XML Schema leaves that decision to the end-user.

## XML Schema Datatypes

When we were developing our DTDs, we could specify that an element had mixed element, or empty content. Unfortunately, when our elements contained only text we could not exercise any constraints on the type of data that was allowed. Attribute declarations gave us more control, but even then, our types were very limited.

### Important

XML Schema divides data types into two broad categories: simple and complex. Elements that may contain attributes or other elements are **complexType**s. Attributes values and elements that contain only text content are **simpleType**s.

XML Schemas allow you to specify the type of textual data allowed within attributes and elements using simpleType declarations. By utilizing these types, for example, you could specify that an element may contain only dates, or only positive numbers, or numbers within a certain range. Many commonly used simpleTypes are built into XML Schemas. This is, perhaps, the single most important feature within XML Schemas. By allowing you to specify allowable type of data within an element, you can more rigidly control documents that are intended to represent databases, programming languages, and objects within programming languages. We will look at simpleTypes and complexTypes later in this chapter.

## XML Schema Content Models

In order to reuse a content model within a DTD we had to utilize parameter entities. As we saw, this can lead to situations that make it difficult to reuse parts of the DTD. XML Schemas provide several mechanisms for reusing content models. In addition to the simple models we created in DTDs, XML Schemas can model complex programming concepts. We will see examples of this in the [next chapter](#) when we look at type inheritance and element substitution. The advanced features of XML Schemas allow you to build content models upon content models, modifying the definition in each step.

---

[!\[\]\(ab7c4117441648a8b8a4a2f050c1af0f\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(4052e1cdc9dfdaa7f672aaf6d1203e08\_img.jpg\) NEXT >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Do We Still Need DTDs?

Wait a second. Why did we spend all of [Chapter 5](#) learning about DTDs if we were just going to turn around and teach you a better way to validate documents? DTDs are extremely useful even with the advent of XML Schemas. Although XML Schemas provide better features for describing documents, as well as an easier syntax, they provide no ENTITY functionality. In many XML documents and applications the ENTITY declaration is of paramount importance. On the merits of this feature alone, DTDs will live a long and happy life.

DTDs also have a special prominence, in that they are the only definition mechanism that is embedded within the XML Recommendation itself. This allows DTDs to be embedded directly in the XML documents they are describing, unlike all other syntaxes, which require a separate file. Parsers that support DTDs are trained to use the embedded declarations, while non-validating parsers are allowed to ignore the declarations. Because DTDs also inherit most of their behavior from SGML, much of the early and ongoing work of describing XML document types has been and will be done using DTDs.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Basic XML Schemas

As we progress through this chapter, you should begin to see more benefits of XML Schemas, which provide many features that were never possible using DTDs. We have divided the topic of XML Schemas into two chapters, "XML Schemas" and "[Advanced XML Schemas](#)". Throughout this chapter, we will focus on the basic parts of XML Schemas that are similar to DTDs. We will also learn about some of the data type mechanisms. In the [next chapter](#), we will learn some of the advanced features available within XML Schemas.

*Although you will learn how to design and use XML Schemas in this chapter, you may like to see the XML Schema Recommendation for yourself. The XML Schema Recommendation is divided into three parts: an introduction to XML Schema concepts at <http://www.w3.org/TR/xmlschema-0/>; a document which defines all of the structures used in XML Schemas at <http://www.w3.org/TR/xmlschema-1/>; and a document which describes XML Schema Datatypes at <http://www.w3.org/TR/xmlschema-2/>.*

## The XML Schema Document

Although most XML Schemas are stored within a separate XML document, the XML Schema Recommendation takes great pains to ensure that the XML Schema can be defined using alternate methods as well. In this respect, XML Schemas function very similarly to external DTDs, an XML document contains a reference to the XML Schema that defines its vocabulary. An XML document that adheres to a particular XML Schema is an XML Schema **instance** document.

As we saw in the [last chapter](#), validating a document against its vocabulary requires the use of a special parser. The XML Schema Recommendation calls these parsers **schema validators**. Not only do schema validators render a verdict on the document's schema validity, many also provide type information to the application. This set of type information is called the **Post Schema Validation Infoset (PSVI)**. The PSVI contains all of the information in the XML document and a basic summary of everything declared in the schema.

## Running the Samples

We have talked about some of the benefits of XML Schemas, but it will probably help us if we see an entire XML Schema before we look at each part in detail. In order to see how the XML Schema works we will modify our name example from the [previous chapter](#). Throughout this chapter, we will be using the same program to validate our documents that we used in [Chapter 5](#). Because our program is based on the Java version of Xerces, it is also capable of checking an instance document against an XML Schema.

### Important

At the time of this writing, support for XML Schemas is not as widespread as the support for DTDs. XML Schemas are a much newer technology, though. A list of XML Schema tools can be found on the XML Schema homepage at <http://www.w3.org/XML/Schema#Tools>.

## Try It Out—What's in a Name?

In this example, we will create an XML Schema that defines our name vocabulary. We will also see how to refer to the XML Schema from the instance document. At the end of this example, we will break down each step to see how it works.

Let's begin by creating the XML Schema. Simply open a text editor, such as Notepad, and copy the following. When you are finished, save the file as name5.xsd. Throughout this chapter, we will assume that you are saving your documents within the C:\Wrox directory. If you save your documents in another directory, you will need to substitute that path wherever you see C:\Wrox.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
<element name="name">
<complexType>
<sequence>
<element name="first" type="string"/>
<element name="middle" type="string"/>
<element name="last" type="string"/>
</sequence>
<attribute name="title" type="string"/>
</complexType>
</element>
</schema>
```

## 2.

Next, we'll need to create the instance document. This document will be very similar to our name4.xml example from the [previous chapter](#). Instead of referring to a DTD, we will be referring to our newly created XML Schema. Copy the following; when you are finished, save the file as name5.xml.

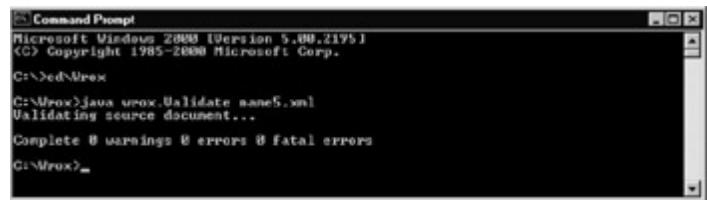
```
<?xml version="1.0"?>
<name
  xmlns="http://www.wrox.com/name"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wrox.com/name name5.xsd"
  title="Mr.">
  <first>John</first>
  <middle>Fitzgerald</middle>
  <last>Doe</last>
</name>
```

## 3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your name5.xml and name5.xsd documents are located. At the prompt type the following and then press the Enter key:

**C:\Wrox> java wrox.Validate name5.xml**

You should see the following output, confirming that our document has validated successfully against our schema:



If you received a Java error, you may need to check the installation of the program. Sometimes this may simply be an issue with the PATH or CLASSPATH. Review the instructions given in the [last chapter](#) and if you think this might be the problem, try moving our PATH and CLASSPATH additions earlier in the list of paths. Also check that you use the correct case when you type wrox.Validate as Java is case-sensitive.

If the output suggests that the validation completed, but that there was an error in the document, correct the error and try again.

4.

If you would like to see what happens if there is an error, simply modify your name5.xml document and try validating again.

## How It Works

In this Try It Out example, we created an XML Schema for our name vocabulary. We used the XML Schema to check if our instance document was schema valid. In order to connect the two documents we included a reference to the XML Schema within our instance document. The internal process by which schema validators compare the document structure against the vocabulary varies greatly. At the most basic level, the schema validator will read the declarations within the XML Schema. As it is parsing the instance document, it will validate each element that it encounters against the matching declaration. If it finds an element or attribute that does not appear within the declarations, or if it finds a declaration that has no matching XML content it will raise a schema validity error.

Let's break the XML Schema down into smaller pieces so that we can see some of what we will be learning later:

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

As we have seen in all of our XML documents, we begin with the XML declaration. Again, this is optional but it is highly recommended that you include it to avoid XML version conflicts later.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
```

The root element within our XML Schema is the <schema> element. Within the <schema> element, we have our namespace declaration. We indicated that the namespace of our <schema> element is <http://www.w3.org/2001/XMLSchema>. This namespace represents the Recommendation version of XML Schema.

Within the <schema> element, we have also included a targetNamespace attribute that indicates that we are developing a vocabulary for the namespace <http://www.wrox.com/name>. Remember this is just a unique name that we chose, the URL does not necessarily point to anything. We also declared a namespace that matches our targetNamespace with the prefix target. If we need to refer to any declarations within our XML Schema we will need this declaration, so we have included it just in case. Again, you are not required to use target as your prefix, you could choose any prefix you like.

We also included the attribute elementFormDefault with the value qualified. Essentially, this controls the way namespaces are used within our corresponding XML document. For now, it is best to get into the habit of adding this attribute with the value qualified, as it will simplify our XML documents. We will look at what this means a little later in the chapter.

```
<element name="name">
```

Within our <schema> element, we have an <element> declaration. Within this <element> declaration, we specified that the name of the element is name. In this example, we have chosen to specify our content by including a <complexType> definition within our <element> declaration.

```
<complexType>
<sequence>
<element name="first" type="string"/>
<element name="middle" type="string"/>
<element name="last" type="string"/>
</sequence>
<attribute name="title" type="string"/>
</complexType>
```

Because our <name> element, in our example, contains the elements <first>, <middle>, and <last> it is considered a complexType. Our <complexType> definition allows us to specify the allowable elements and their order.

Just as we did in our DTD, we must declare our content using a content model. In DTDs we could use sequences and choices when specifying our content model. In our example, we have indicated that we are using a sequence by including a <sequence> definition. Our <sequence> declaration contains three <element> declarations. Within these declarations, we have specified that their type is string. This indicates that the elements must adhere to the XML Schema simpleType string, which allows any textual content.

In addition, within our <complexType> definition, we included an <attribute> declaration. This <attribute> declaration appears at the end of the <complexType> definition, after any content model information. By declaring a title attribute, we can easily specify how we should address the individual described by our <name> XML.

Before we move on, let's just take a quick look at our instance document:

```
<name  
    xmlns="http://www.wrox.com/name"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://www.wrox.com/name name5.xsd"  
    title="Mr.">
```

Within the root element of our instance document, we have two namespace declarations. The first indicates that our default namespace is <http://www.wrox.com/name>. This namespace matches the targetNamespace that we declared within our XML Schema. We also declare the namespace <http://www.w3.org/2001/XMLSchema-instance>. The XML Schema Recommendation allows you to include several attributes from this namespace within your instance document.

Our instance document includes the attribute schemaLocation. This attribute tells our schema validator where to find our XML Schema document for validation. The schemaLocation attribute is declared within the namespace <http://www.w3.org/2001/XMLSchema-instance> so we have prefixed the attribute with the prefix xsi. The value of schemaLocation attribute is <http://www.wrox.com/name> name5.xsd. This is known as a namespace/file location pair; it is the namespace of our XML document and the URL of the XML Schema that describes our namespace. The XML Schema Recommendation allows you to declare several namespace/file location pairs within a single schemaLocation attribute—simply separate the values with whitespace. This is useful when your XML document uses multiple namespaces.

The schemaLocation attribute is only a hint for the processor to use—the processor may not use the provided location at all. For example, the validator may have a local copy of the XML Schema that it uses, instead of loading the file specified, in order to decrease processor usage. If your XML Schema has no targetNamespace you must refer to the XML Schema using the noNamespaceSchemaLocation attribute within our instance document.

This has been an extremely brief overview of some difficult concepts in XML Schemas. This Try It Out is intended to give you an overall context for what we will be learning throughout the chapter.

*This chapter won't list all of the elements available with XML Schemas, but will introduce the more common ones that you're likely to encounter. Furthermore, not all of the attributes are listed for some of the elements, but again, only the more common ones. For in-depth coverage of all of the XML Schema features, and their use, see Professional XML Schemas, by Stephen Mohr et al. Wrox Press (ISBN 1-861005-47-4).*

## <schema>

As we have already seen, the <schema> element is the root element within an XML Schema. The <schema> element allows us to declare namespace information as well as defaults for declarations throughout the document. The XML Schema recommendation also allows us to include a version attribute that can help to identify the XML Schema and the version of our vocabulary.

```
<schema targetNamespace="URI"  
       attributeFormDefault="qualified or unqualified"  
       elementFormDefault="qualified or unqualified"  
       version="version number">
```

## The XML Schema Namespace

In our first example, we declared the namespace <http://www.w3.org/2001/XMLSchema> within our <schema> element. This allows us to indicate that the <schema> element is part of the XML Schema vocabulary. Remember, because XML is case-sensitive, namespaces are case-sensitive. If the namespace does not match <http://www.w3.org/2001/XMLSchema> the schema validator should reject the document. For example, you could use any of the following <schema>:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">  
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">  
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

As we learned in [Chapter 3](#), the namespace prefix is insignificant—it is only a shortcut to the namespace declaration. You will usually see one of these three variations. The XML Schema Recommendation, itself, uses the prefix xs. Which you use is a matter of personal preference.

## Target Namespaces

The primary purpose of XML Schemas is to declare vocabularies. These vocabularies can be identified by a namespace that is specified in the targetNamespace attribute. It is important to realize that not all XML Schemas will have a targetNamespace. Many XML Schemas define vocabularies that will be reused in another XML Schema, or vocabularies that will be used in documents where the namespace is not necessary.

When declaring a targetNamespace, it is important to include a matching namespace declaration. Like the XML Schema namespace, you can choose any prefix you like, or you can use a default namespace declaration. The namespace declaration will be used when you are referring to declarations within the XML Schema. We will see what this means in more detail later in the section "*Referring to an Existing Global Element*".

Some possible targetNamespace declarations include:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"  
        targetNamespace="http://www.wrox.com/name"  
        xmlns:target="http://www.wrox.com/name">  
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"  
        targetNamespace="http://www.wrox.com/name" xmlns="http://www.wrox.com/name">
```

## Declaration Defaults

The <schema> element also allows us to modify the defaults for the declarations that appear within the XML Schema. We can modify these defaults by including the attributes:

- elementFormDefault
- attributeFormDefault

The elementFormDefault and attributeFormDefault attributes allow you to control the default qualification form for elements and attributes in the instance documents. Within the instance document, elements and attributes may be

qualified or unqualified. An element or attribute is **qualified** if it has an associated namespace URI. For example, the following elements are qualified:

```
<name xmlns="http://www.wrox.com/name">
  <first>John</first>
  <middle>Fitzgerald</middle>
  <last>Doe</last>
</name>
<n:name xmlns:n="http://www.wrox.com/name">
  <n:first>John</n:first>
  <n:middle>Fitzgerald</n:middle>
  <n:last>Doe</n:last>
</n:name>
```

Even though our first example doesn't have a namespace prefix, it still has an associated namespace URI, <http://www.wrox.com/name>, so it is qualified *but not* prefixed. Each of the children elements is also qualified because of the default namespace declaration in the `<name>` element. Again, these elements have no prefixes. In the second example, all of the elements are qualified *and* prefixed.

Unqualified elements have no associated namespace. For example:

```
<n:name xmlns:n="http://www.wrox.com/name">
  <first>John</first>
  <middle>Fitzgerald</middle>
  <last>Doe</last>
</n:name>
```

The `<name>` element is qualified, but the `<first>`, `<middle>`, and `<last>` elements are not. The `<first>`, `<middle>`, and `<last>` elements have no associated namespace declaration (default or otherwise) and, therefore, they are unqualified. This mix of qualified and unqualified elements may seem strange; nevertheless, it is the default behavior. The default value for both `elementFormDefault` and `attributeFormDefault` is unqualified.

Even though the value of the `elementFormDefault` attribute is unqualified, some elements must be qualified regardless. For example, the `<name>` element must *always* be qualified in the instance document. This is because it was declared globally within our XML Schema (we will look at global and local declarations in detail in the [next section](#)). In the above example, this is exactly what we have done. We have qualified the `<name>` element with a namespace but not the `<first>`, `<middle>`, and `<last>` elements.

Most of your documents should qualify all of their elements. In some cases, you will need to create a document that uses both qualified and unqualified elements. For example, XSLT and SOAP documents may contain both qualified and unqualified elements. Therefore, it is considered best practice that, unless you have a very specific need to mix qualified and unqualified elements, you should always include the `elementFormDefault` attribute with the value `qualified`.

## <element>

When declaring an element, we are actually performing two primary tasks: specifying the element name and defining the allowable content.

```
<element
  name="name of the element"
  type="global type"
  ref="global element declaration"
  form="qualified or unqualified"
  minOccurs="non negative number"
  maxOccurs="non negative number or 'unbounded'"
  default="default value"
```

fixed="fixed value">

According to the XML Schema recommendation, an element's allowable content is determined by its **type**. As we have already seen, element types are divided into simpleTypes and complexTypes. XML Schemas allow you to specify an element's type in one of three ways:

- Creating a local type
- Using a global type
- Referring to an existing global element

In addition to these two methods, you may also reuse elements by referring to an existing global element. In this case, you include a reference to the global element declaration. Of course, you do not need to specify a type in your reference; the type of the element is included in the global element declaration.

## Global vs. Local

Before we can understand these different methods for declaring elements, we must learn the difference between global and local declarations. XML Schema declarations can be divided into two broad categories: global declarations and local declarations. **Global declarations** are declarations that appear as direct children of the `<schema>` element. Global element declarations can be reused throughout the XML Schema. **Local declarations** do not have the `<schema>` element as their direct parent and are only valid in their specific context. Let's look at our first example again:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
    <!-- The first element declaration is global because it is a direct child of the
        schema element -->
    <element name="name">
        <complexType>
            <sequence>
                <!-- These element declarations are local because they are children of the
                    sequence element within a complexType definition -->
                <element name="first" type="string"/>
                <element name="middle" type="string"/>
                <element name="last" type="string"/>
            </sequence>
            <attribute name="title" type="string"/>
        </complexType>
    </element>
</schema>
```

In this XML Schema, we have four element declarations. The first declaration, our `<name>` element, is a global declaration because it is a direct child of the `<schema>` element. The declarations for the `<first>`, `<middle>`, and `<last>` elements are considered local because the declarations are not direct children of the `<schema>` element. The declarations for the `<first>`, `<middle>`, and `<last>` elements are only valid within the `<sequence>` declaration—they cannot be reused elsewhere in the XML Schema.

## Creating a Local Type

Of the three methods of element declaration, creating a local type should seem the most familiar. We used this model when we declared our `<name>` element in our example. To create a local type, you simply include the type

declaration as a child of the element declaration:

```
<element name="name">
  <complexType>
    <!-- type information -->
  </complexType>
</element>
```

or:

```
<element name="name">
  <simpleType>
    <!-- type information -->
  </simpleType>
</element>
```

In these examples, we see that an element declaration may contain a `<complexType>` definition or a `<simpleType>` definition. We will look at the insides of these declarations a little later.

## Using a Global Type

Often, many of our elements will have the same content. Instead of declaring duplicate local types throughout our schema, we can create a global type. Within our element declarations, we can refer to a global type by name. In these examples, our element declaration refers to a global type named `string`. This type is a member of the XML Schema vocabulary. When referring to the type we must include the namespace prefix, just as we would for elements:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="first" type="string"/>
```

In our first declaration, the XML Schema namespace is the default namespace; because there is no prefix, we do not need a prefix when referring to the `string` type.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="first" type="xs:string"/>
```

In the second declaration, however, the XML Schema namespace is declared using the prefix `xs`. Therefore, in order to refer to the `string` type, we must include the namespace prefix `xs`.

### Try It Out—Creating Reusable Global Types

Creating global types within our XML Schema is straightforward. Let's convert our `<name>` example to use a named global type rather than a local type.

1.

We will begin by making the necessary changes to our XML Schema. Open the file `name5.xsd` and make the following changes. When you are finished, save the file as `name6.xsd`.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
  <complexType name="NameType">
    <sequence>
      <element name="first" type="string"/>
      <element name="middle" type="string"/>
      <element name="last" type="string"/>
    </sequence>
```

```
<attribute name="title" type="string"/>
</complexType>
<element name="name" type="target:NameType"/>
</schema>
```

2.

Before we can schema validate our document, we must modify it so that it refers to our new XML Schema. Open the file name5.xml and change the xsi:schemaLocation attribute, as follows. When you are finished, save the file as name6.xml.

```
xsi:schemaLocation="http://www.wrox.com/name name6.xsd"
```

3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your name6.xml and name6.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate name6.xml
```

This should validate correctly as we saw in the last Try It Out.

## How It Works

We had to make minor modifications to our schema in order to create a reusable complexType. First, we moved our <complexType> definition from within our <element> declaration to our <schema> element. Remember, a declaration is global if it is a direct child of the <schema> element. Once we had made our <complexType> definition global, we needed to add a name attribute so that we could refer to it later. We named our <complexType> definition NameType so it would be easy to identify later.

Once we declared our NameType <complexType>, we modified our <name> element declaration to refer to it. We added a type attribute to our element declaration with the value target:NameType. Keep in mind; we had to include the namespace prefix target when referring to the type, so the validator knew what namespace it should look in.

## Referring to an Existing Global Element

Referring to global types allows us to easily reuse content model definitions within our XML Schema. Often, we may want to reuse entire element declarations, instead of just the type. XML Schemas allow you to reuse global element declarations within your content model. To refer to a global element declaration, simply include a ref attribute and specify the name of the element as the value.

```
<element ref="target:first"/>
```

Again, the name of the element must be qualified with the namespace prefix. Notice that when we refer to a global element declaration we have no type attribute and no local type declaration. Our element declaration will use the type of the <element> declaration to which we are referring.

## Try It Out–Referring to Global Element Declarations

Let's modify our last example so that we can see how to create and refer to global element declarations.

1.

Again, we will begin by making the necessary changes to our XML Schema. Open the file name6.xsd and make the following changes. When you are finished, save the file as name7.xsd.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
```

```
elementFormDefault="qualified">
<element name="first" type="string"/>
<element name="middle" type="string"/>
<element name="last" type="string"/>
<complexType name="NameType">
  <sequence>
    <element ref="target:first"/>
    <element ref="target:middle"/>
    <element ref="target:last"/>
  </sequence>
  <attribute name="title" type="string"/>
</complexType>
<element name="name" type="target:NameType"/>
</schema>
```

## 2.

Before we can schema validate our XML document, we must modify it so that it refers to our new XML Schema. Open the file name6.xml and change the xsi:schemaLocation attribute, as follows. When you are finished, save the file as name7.xml.

```
xsi:schemaLocation="http://www.wrox.com/name name7.xsd"
```

## 3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your name7.xml and name7.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate name7.xml
```

Again, this should validate with no errors.

## How It Works

In this Try It Out, we utilized references to global element declarations within our content model. First, we moved the declarations for our <first>, <middle>, and <last> elements from within our <complexType> definition to our <schema> element making them global. After we created our global declarations, we inserted references to the elements within our <complexType>. In each reference, we prefixed the global element name with the prefix target.

At this point, it might help to examine what our schema validator is doing in more detail. As our schema validator is processing our instance document, it will first encounter the root element, in this case <name>. When it encounters the <name> element, it will look it up in the XML Schema. When attempting to find the declaration for the root element, the schema validator will only look through the global element declarations.

### Important

In this case, there are four global element declarations: <first>, <middle>, <last>, and <name>. Any one of these could be used as the root element within an instance document; in our example, we are using the <name> element as our instance document root element. Although the XML Schema Recommendation allows us to have multiple global <element> declarations, we are still only allowed to have one root element in our instance document.

Once the schema validator finds the matching declaration, it will find the associated type (in this case it is a global <complexType> definition NameType). It will then validate the content of the <name> element within the instance against the content model defined in the associated type. When the schema validator encounters the <element>

reference declarations, it will import the global <element> declarations into the <complexType> definition, as if they had been included directly.

Now that we have seen some of the basics of how elements are declared, let's look briefly at some of the features element declarations offer. Later in the chapter, we will look at complexType definitions and content models in more depth.

## Naming Elements

Specifying a name in your element declaration is very straightforward. Simply include the name attribute and specify the desired name as the value. The name must follow the rules for XML names that we have already learned. In the [last chapter](#), when creating names in DTDs, we saw that we had to include a namespace prefix, if one were going to be used in the instance document. Because XML Schemas are namespace aware, this is unnecessary. Simply specify the name of the element; the schema validator will be able to understand any prefix that is used within the instance document. The following are examples of *valid* element names:

```
<element name="first" type="string"/>
<element name="record" type="string"/>
```

The following are examples of *invalid* element names:

```
<element name="2ndElement" type="string"/>
<element name="album:CD" type="string"/>
```

The first of these examples is invalid because it begins with a number. Remember, XML names may include numerical digits, periods (.), hyphens (-), and underscores (\_), but they must begin with a letter or an underscore (\_). The second of these examples is invalid because it contains a colon (:). Since the inception of namespaces, the colon may only be used to indicate a namespace prefix. As we said earlier, the prefix must not be included in the element name.

## Element Qualified Form

The form attribute allows you to override the default for element qualification. Remember, if an element is qualified it must have an associated namespace when it is used in the instance document. You can specify whether the element must be qualified by setting the value of the form attribute to qualified or unqualified. If you do not include a form attribute, the schema validator will use the value of the elementFormDefault attribute declared in the <schema> element.

## Cardinality

In the [last chapter](#), we learned that when we are specifying elements in our content models we could modify their cardinality. Cardinality represents the number of occurrences of a specific element within a content model. In XML Schemas, we can modify an element's cardinality by specifying the minOccurs and maxOccurs attributes within the element declaration.

### Important

It is important to note that the minOccurs and maxOccurs attributes are not permitted within global element declarations. Instead, you should use these attributes within the element references in your content models.

Within DTDs, we had very limited options when specifying cardinality. Using cardinality indicators, we could declare that an element would appear once and only once, once or not at all, one or more times, or zero or more times. This

seems to cover the basics, but many times we need more control. XML Schemas improved the model by allowing you to specify the minimum and maximum separately.

Some possible uses of the minOccurs and maxOccurs attributes include:

```
<element name="element1" type="string" minOccurs="2" maxOccurs="2"/>
<element ref="target:element2" maxOccurs="10"/>
<element name="element3" type="string" minOccurs="0" maxOccurs="unbounded"/>
```

The default value for the minOccurs attribute is 1. The default value for the maxOccurs attribute is also 1. You can use the two attributes separately or in conjunction. Both attributes accept numbers that are not negative for their value. The maxOccurs attribute also allows the value unbounded, which indicates that there is no limit to the number of occurrences. The only additional rule you must adhere to when specifying minOccurs and maxOccurs, is that the value of maxOccurs must be greater than, or equal to, the value for minOccurs.

In our first example above we have declared that the element <element1> must appear within our instance document a minimum of two times and a maximum of two times. In the second example, we have declared our element using a reference to the global <element2> declaration. Even though it is declared using the ref attribute, we are permitted to use the minOccurs and maxOccurs attributes to specify the element's cardinality. In this case, we have included a maxOccurs attribute with the value 10. We have not included a minOccurs attribute, so a schema validator would use the default value, 1. In the final example, we have specified that <element3> may or may not appear within our instance document because the minOccurs attribute has the value 0. We have also indicated that it may appear an infinite number of times because the value of maxOccurs is unbounded.

## Default and Fixed Values

When designing the DTD for our media catalog in the [last chapter](#), we made use of attribute default and fixed values. In XML Schemas, we can declare default and fixed values for elements as well as attributes. When declaring default values for elements we must specify a default text value. You are not permitted to specify a default value for an element whose content model will contain other elements, unless the content model is mixed.

When working with DTDs, we saw that we could provide a value for an attribute even if it wasn't included in the XML document. In XML Schemas, we can provide default values for attributes and elements. By specifying a default value for our element, we can be sure that the schema validator will treat the value as if it were included in the XML document, even if it is omitted. To specify a default value, simply include the default attribute with the desired value. Suppose our <name> elements were being used to design the "Doe" family tree. We might want to make "Doe" the default for the last name element:

```
<element name="last" type="string" default="Doe"/>
```

In this example we have declared that our element <last> will have the default value "Doe". Because of this, when a schema validator encounters the last element, if there is no content, it will insert the default value. For example, if the schema validator encounters:

```
<last></last>
```

or:

```
<last/>
```

it would treat the element as follows:

```
<last>Doe</last>
```

It is important to note that the element must appear within the document, and it must be empty. If the element does not appear within the document, or if the element already has content, the default value will not be used.

In the [last chapter](#), in addition to default values, we also learned that attributes may have fixed values. Again, in XML Schemas elements and attributes may have fixed values. There are some circumstances where you may want to make sure that an element's value does not change, such as an element whose value is used to indicate a version number. When an element's value can never change, simply include a fixed attribute with the fixed value. As the schema validator is processing the file, if an element whose declaration contains a fixed value is encountered, then the parser will check that the content and fixed attribute value match. If they do not match, the parser will raise a schema validity error. If the element is empty, then the parser will insert the fixed value.

To specify a fixed value, simply include the fixed attribute with the desired value:

```
<element name="version" type="string" fixed="1.0"/>
```

In the above example, we have specified that the `<version>` element, if it appears, must contain the value 1.0, which is a valid string value (the type of our `<version>` element is string). The following elements would be *legal*:

```
<version>1.0</version>
<version></version>
<version/>
```

As the schema validator is processing the file, it will accept the latter of these examples. Because they are empty elements, it will treat them as if the value 1.0 had been included. The following value is *not legal*:

```
<version>2.0</version>
```

When specifying fixed or default values you must ensure that the value you specify is allowable content for the type that you have declared for your element declaration. Again, your default and fixed values are not permitted to contain element content. Therefore, your element must have a simpleType or a mixed content declaration. You are not permitted to use default and fixed values at the same time within a single element declaration.

## Element Wildcards

Often we will want to include elements in our XML Schema without exercising as much control. Suppose we wanted our element to contain any of the elements declared in our namespace. Suppose we wanted to specify that our element may contain any elements from another namespace. This is common when designing XML Schemas. Declarations that allow you to include any element from a namespace are called **element wildcards**.

To declare an element wildcard, use the `<any>` declaration:

```
<any
  minOccurs="non negative number"
  maxOccurs="non negative number or 'unbounded'"
  namespace="allowable namespaces"
  processContents="lax or skip or strict">
```

The `<any>` declaration can only appear within a content model. You are not allowed to create global `<any>` declarations, instead you must use them within content models. When specifying an `<any>` declaration you can specify the cardinality just as you would an `<element>` declaration. By specifying the `minOccurs` or the `maxOccurs` attributes you can control how many wildcard occurrences are allowed within your instance document.

The <any> declaration allows you to control what namespace or namespaces the elements are allowed to come from, which is controlled by including the namespace attribute. The namespace attribute allows several values:

Value	Description
#any	Allows elements from all namespaces to be included as part of the wildcard
##other	Allows elements from namespaces other than the targetNamespace to be included as part of the wildcard
##targetNamespace	Allows elements from only the targetNamespace to be included as part of the wildcard
##local	Allows any well-formed elements that are not qualified by a namespace to be included as part of the wildcard
Whitespace separated list of allowable namespace URIs	Allows elements from any listed namespaces to be included as part of the wildcard. Possible list values also include ##targetNamespace and ##local.

For example, suppose that we wanted to allow any well-formed XML content from any namespace within our <name> element. Within the content model for our NameType complexType we could include an element wildcard:

```
<complexType name="NameType">
<sequence>
  <element ref="target:first"/>
  <element ref="target:middle"/>
  <element ref="target:last"/>
  <!-- allow any element from any namespace -->
  <any namespace="#any"
    processContents="lax"
    minOccurs="0"
    maxOccurs="unbounded"/>
</sequence>
<attribute name="title" type="string"/>
</complexType>
```

In the above example, we have included an <any> element wildcard. By setting the namespace attribute to ##any, we have specified that elements from all namespaces can be included as part of the wildcard. We have also included cardinality attributes to indicate the number of allowed elements. In this case, we have specified that there may be any number of elements because we have set the value of maxOccurs to unbounded. We have also indicated that there may be none at all by setting the value of the minOccurs attribute to 0. Therefore, our content model must contain a <first>, <middle>, and <last> element in sequence, followed by any number of elements from any namespace.

When the schema validator is processing an element that contains a wildcard declaration, it will validate the instance documents in one of three ways. If the value of the processContents attribute is set to skip, the processor will skip any wildcard elements in the instance document. If the value of processContents attribute is set to lax, then the processor will attempt to validate the wildcard elements if it has access to an XML Schema that defines them. If the value of the processContents attribute is set to strict (the default), the processor will attempt to validate the wildcard elements if it has access to an XML Schema that defines them. Unlike the lax setting, however, if the XML Schema for the wildcard elements cannot be found then the schema validator will raise a validity error.

## <complexType>

So far, we have seen the basics of declaring elements. In each of our examples, we utilized a `<complexType>` definition. Let's look at our type definitions in more detail. As we have already learned, our element content is controlled, either directly or indirectly, by `<simpleType>` and `<complexType>` definitions. Within our `<complexType>` definition, we can specify the allowable element content for our declaration.

```
<complexType
    mixed="Boolean expression"
    name="Name of complexType">
```

All of our examples so far have used either a local or global `<complexType>` to specify the content model for our `<name>` element declaration.

```
<element name="name">
    <complexType>
        <sequence>
            <element name="first" type="string"/>
            <element name="middle" type="string"/>
            <element name="last" type="string"/>
        </sequence>
        <attribute name="title" type="string"/>
    </complexType>
</element>
```

When we created a local declaration, we did not include a name attribute in our `<complexType>` definition. Local `<complexType>` definitions are **never** named; in fact, they are called **anonymous complex types**. As we have already seen, however, global `<complexType>` definitions are **always** named, so that they can be identified later.

Apart from the content models we have seen, `<complexType>` definitions can also be used to create mixed and empty content models. Mixed content models, as we have seen, allow us to include both text and element content within a single content model. To create a mixed content model in XML Schemas, simply include the mixed attribute with the value true in your `<complexType>` definition.

```
<element name="p">
    <complexType mixed="true">
        <choice minOccurs="0" maxOccurs="unbounded">
            <element name="b" type="string"/>
            <element name="i" type="string"/>
        </choice>
    </complexType>
</element>
```

In this example, we have declared a `<p>` element, which can contain an infinite number of `<b>`, and `<i>` elements. Text can be interspersed throughout these elements. An allowable `<p>` element might look like:

```
<p>Without question, <b>Weird Al</b> is the <i>greatest</i> singer <b>of all time!</b></p>
```

In the above `<p>` element, we have textual content interspersed throughout the elements we have declared within our content model. As the schema validator is processing the above example, it will ignore the textual content and instead perform standard validation on the elements. Therefore, because the `<p>` element is declared using a mixed content model, the schema validator will check only the elements within its content. Because the elements `<b>` and `<i>` may appear repeatedly, the above example is valid.

To declare an empty content model in a `<complexType>` definition, you simply create the `<complexType>` definition without any `<element>` or content model declarations. Consider the following declarations:

```
<element name="emptyElement">
  <complexType>
  </complexType>
</element>
<element name="emptyElement">
  <complexType/>
</element>
```

Each of these declares an element named emptyElement. In both cases, the <complexType> definition is empty, indicating the emptyElement, when used in our instance document, must also be empty. For example, the following elements would be valid instances of the above declarations:

```
<emptyElement/>
<emptyElement></emptyElement>
```

Although we haven't looked at attribute declarations in XML Schemas, it should be mentioned that <complexType> definitions also contain <attribute> declarations, if the type we are defining contains attributes. We will look at this in more detail later in this chapter.

## <group>

Within our XML Schema, we can declare our elements within <complexType> definitions as we have done. However, XML Schemas also allow you to define reusable groups of elements. By creating a global <group> declaration, you can easily reuse and combine entire content models.

```
<group
  name="name of global group">
```

Just as we have seen with global <complexType> definitions, all global <group> declarations must be named. Simply specify the name attribute with the desired name. Again, the name that you specify must follow the rules for XML names and it should not include a prefix. The basic structure of a global <group> declaration follows:

```
<group name="NameGroup">
  <!-- content model goes here -->
</group>
```

## Try It Out—Using a Global Group

Let's redesign our schema so that we can create a reusable global <group> declaration.

1.

Again, we will begin by making the necessary changes to our XML Schema. Open the file name7.xsd and make the following changes. When you are finished, save the file as name8.xsd.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
  <group name="NameGroup">
    <sequence>
      <element name="first" type="string"/>
      <element name="middle" type="string"/>
      <element name="last" type="string"/>
    </sequence>
  </group>
```

```
<element name="name">
<complexType>
<group ref="target:NameGroup"/>
<attribute name="title" type="string"/>
</complexType>
</element>
</schema>
```

## 2.

Before we can schema validate our XML document, we must modify it so that it refers to our new XML Schema. Open the file name7.xml and change the xsi:schemaLocation attribute as shown below. When you are finished, save the file as name8.xml.

```
xsi:schemaLocation="http://www.wrox.com/name name8.xsd"
```

## 3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your name8.xml and name8.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate name8.xml
```

It should validate with no errors

## How It Works

In this Try It Out, we have modified our XML Schema to use a global <group> declaration. First, we created a global <group> declaration named NameGroup. Within our declaration, we specified the allowable content model for our <name> element. Within the <complexType> definition for our <name> element, instead of including element declarations, we created a <group> reference declaration. When referring to the global <group> declaration, we included a ref attribute with the value target:NameGroup. Notice that our <attribute> declaration still appeared within our <complexType> definition and not within our <group> declaration. This should give you some indication of the difference between a <group> and a <complexType> definition. A <complexType> definition defines the allowable content for a specific element or type of element. A <group> declaration simply allows you to create a reusable group of elements, which can be used within content model declarations in your XML Schema.

As our schema validator is processing our instance document, it will process the <name> element similarly to our earlier examples. When it encounters the <name> element, it will look it up in the XML Schema. Once it finds the declaration, it will find the associated type (in this case it is a local <complexType> definition). When the schema validator encounters the <group> reference declaration, it will treat the items within the group as if they had been included directly within the <complexType> definition.

## Content Models

We have already seen that we can use <complexType> and <group> declarations to specify an element's allowable content. XML Schemas provide greater flexibility than DTDs when specifying an element's content model. In XML Schemas you can specify an element's content model using

- A <sequence> declaration
- A <choice> declaration
-

A reference to a global <group> declaration

- An <all> declaration

Using these four primary declarations we can specify the content model of our type in a variety of ways. Each of these declarations may contain:

- Inner content models
- Element declarations
- Element wildcards

We will see how each of these is used throughout this section of the chapter.

## <sequence>

Just as we saw in our DTD content models, specifying our content model using a sequence of elements is very simple. In fact, our first example used a <sequence> declaration when defining the allowable children of our <name> element.

```
<sequence  
    minOccurs="non negative number"  
    maxOccurs="non negative number or 'unbounded'">
```

The <sequence> declaration allows you to specify minOccurs and maxOccurs attributes that apply to the overall sequence. You can modify the cardinality, or how many of this sequence of elements will occur, by changing the values of these attributes. The minOccurs and maxOccurs attributes function exactly as they did within our element declarations.

We have already seen that the <sequence> declaration may contain <element> declarations within it. In addition to <element> declarations, it may contain element wildcards or inner <sequence>, <choice>, or <group> references. We may have sequences within sequences within sequences, or we may have choices within sequences that are in turn within groups—almost any combination you can imagine.

A sample sequence might appear as:

```
<complexType>  
    <sequence>  
        <element name="first" type="string"/>  
        <element name="middle" type="string"/>  
        <element name="last" type="string"/>  
    </sequence>  
    <attribute name="title" type="string"/>  
</complexType>
```

By utilizing a <sequence> to specify your content model, you indicate that the elements must appear within your instance document in the **sequence**, or order, specified. For example, because our complexType definition utilizes a sequence the following would be **legal**:

```
<first>John</first>
<middle>Fitzgerald</middle>
<last>Doe</last>
```

The following would be *illegal*:

```
<last>Doe</last>
<middle>Fitzgerald</middle>
<first>John</first>
```

This example would not be allowable because the elements do not appear in the order that we have specified within our complexType definition.

## <choice>

The basic structure of our <choice> declaration looks very much like our <sequence> declaration.

```
<choice
  minOccurs="non negative number"
  maxOccurs="non negative number or 'unbounded'">
```

Again, we can specify minOccurs and maxOccurs attributes to modify the cardinality of our <choice> declaration. The <choice> declaration function is also similar to its DTD counterpart. The <choice> declaration allows you to specify multiple declarations. Within an instance document, however, only one of the declarations may be used. For example, suppose we had declared the content model of our <name> element using a <choice> declaration:

```
<element name="name">
  <complexType>
    <choice>
      <element name="first" type="string"/>
      <element name="middle" type="string"/>
      <element name="last" type="string"/>
    </choice>
    <attribute name="title" type="string"/>
  </complexType>
</element>
```

If we had declared our content model as we have in the above example, then within our instance document we could include only the <first> element, only the <middle> element, or only the <last> element. We could not include more than one of the elements within the instance.

Just as we saw in our <sequence> declaration, our <choice> declaration may contain <element> declarations, element wildcards, and inner <sequence>, <choice>, or <group> references.

## <group>

The <group> reference declaration allows us to refer to global element groups within our content model. Don't confuse a <group> reference, which appears within a content model, with a global <group> declaration that defines a reusable group of elements. We have already seen that global element groups function very similarly to global type definitions; they allow us to define content models that can be grouped together and reused within other content models.

```
<group
  ref="global group definition"
  minOccurs="non negative number"
  maxOccurs="non negative number or 'unbounded'">
```

Within a content model, the <group> is used by creating a reference to an already declared group. This can be done by including a ref attribute and specifying the name of the global <group> declaration. As we saw with element references, the global group reference must be prefixed with the appropriate namespace prefix.

An example could be:

```
<group name="NameGroup">
  <sequence>
    <element name="first" type="string"/>
    <element name="middle" type="string"/>
    <element name="last" type="string"/>
  </sequence>
</group>
<element name="name">
  <complexType>
    <group ref="target:NameGroup"/>
    <attribute name="title" type="string"/>
  </complexType>
</element>
```

Here the group reference within our <complexType> definition has a ref attribute with the value target:NameGroup. This refers to the global group declaration named NameGroup. We must prefix the name with a namespace prefix, in this case target, so we can identify the namespace in which the NameGroup declaration appears.

Again, we can specify minOccurs and maxOccurs attributes to modify the cardinality of our <group> reference. However, the <group> reference may not contain element children. Of course, as we have already seen, the global <group> declaration which it is referring to will contain element children that define the content model.

## <all>

The <all> declaration allows us to declare that the elements within our content model may appear in any order.

```
<all
  minOccurs="0 or 1"
  maxOccurs="1">
```

In order to use the <all> mechanism, however, we must adhere to several rules. First, the <all> declaration must be the only content model declaration that appears as a child of a <complexType> definition. Secondly, the <all> declaration may only contain <element> declarations as its children. It is not permitted to contain <sequence>, <choice>, or <group> declarations. Finally, the <all> declaration's children may appear once at most in the instance document. This means that within the <all> declaration the values for are limited to 0 or 1.

Is the <all> declaration useful? Even though there are additional restrictions, the <all> declaration can be very useful. The <all> declaration is commonly used when the expected content is known, but not the order. For example, suppose that an XML document was created and updated by several applications. In our hypothetical, as information is collected, it is added to the end of the XML document. All of the information is the same; it just appears in any order.

*The <all> element has many limitations, as we mentioned above. These limitations ensure that schema validators could easily understand the <all> element. Without these restrictions, it would be very difficult to write software to validate XML Schemas that contained <all> declarations.*

Suppose we had declared our <name> content model using the <all> mechanism:

```
<element name="name">
  <complexType>
```

```
<all>
  <element name="first" type="string"/>
  <element name="middle" type="string"/>
  <element name="last" type="string"/>
</all>
<attribute name="title" type="string"/>
</complexType>
</element>
```

Notice that the `<all>` element is the only content model declaration within our `<complexType>` (`<attribute>` declarations do not count as content model declarations). Also, notice that our `<all>` declaration contains only `<element>` declarations as its children. Each element can appear once and only once (because the default value for `minOccurs` and `maxOccurs` is 1). By declaring our content model as we have above, we can validate our element content but still allow our elements to appear in any order. The allowable content for a `<name>` element might include:

```
<first>John</first>
<middle>Fitzgerald</middle>
<last>Doe</last>
<first>John</first>
<last>Doe</last>
<middle>Fitzgerald</middle>
```

As long as all of the elements we have specified appear, they can appear in any order. In the second example, the `<middle>` element was added last (maybe it wasn't known initially). Because we have declared our content model using `<all>`, this is still allowable.

## Try It Out—"Weird" XML Schema

Once again, our knowledge has surpassed our early examples. In order to use all of the XML Schema features that we have learned we will turn to a more complex subject. Let's create an XML Schema for our music catalog. Not only will this provide ample opportunity to use the functionality we have learned thus far, it will also allow us to compare and contrast a DTD and its XML Schema counterpart.

1.

Let's begin by creating the XML Schema. Simply open a text editor, such as Notepad, and copy the following. When you are finished, save the file as al7.xsd in your C:\Wrox directory.

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  elementFormDefault="qualified">

  <element name="catalog">
    <complexType>
      <choice maxOccurs="unbounded">
        <element name="CD">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="cassette">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="record">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
      </choice>
    </complexType>
  </element>
</schema>
```

```
</complexType>
</element>
<element name="MP3">
  <complexType>
    <group ref="target:AlbumGroup"/>
  </complexType>
</element>
</choice>
</complexType>
</element>
<group name="AlbumGroup">
  <sequence>
    <element name="artist" type="string"/>
    <element name="title" type="string"/>
    <element name="genre" type="string"/>
    <element name="date-released" type="string"/>
    <element name="song" type="target:SongType" maxOccurs="unbounded"/>
  </sequence>
</group>

<complexType name="SongType">
  <sequence>
    <element name="title" type="string"/>
    <element name="length" type="target:LengthType"/>
    <element name="parody" type="target:ParodyType"/>
    <element name="lyrics" type="string"/>
  </sequence>
</complexType>

<complexType name="LengthType">
  <sequence>
    <element name="minutes" type="string"/>
    <element name="seconds" type="string"/>
  </sequence>
</complexType>

<complexType name="ParodyType">
  <sequence minOccurs="0">
    <element name="title" type="string"/>
    <element name="artist" type="string"/>
  </sequence>
</complexType>

</schema>
```

## 2.

Next, we'll need to create the instance document. This document will be very similar to our catalog samples from [Chapter 5](#). Instead of referring to a DTD, we will be referring to our newly created XML Schema. To start with we will not include any attributes, we will add these later in the chapter. Also, we will not be using any entities. Copy the following; when you are finished, save the file as al7.xml.

```
<?xml version="1.0"?>
<catalog
  xmlns="http://www.wrox.com/catalog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wrox.com/catalog al7.xsd">
  <CD>
    <artist>"Weird Al" Yankovic</artist>
    <title>Dare to be Stupid</title>
    <genre>parody</genre>
    <date-released>1990</date-released>
    <song>
      <title>Like A Surgeon</title>
```

```
<length>
    <minutes>3</minutes>
    <seconds>33</seconds>
</length>
<parody>
    <title>Like A Virgin</title>
    <artist>Madonna</artist>
</parody>
<lyrics></lyrics>
</song>
<song>
    <title>Dare to be Stupid</title>
    <length>
        <minutes>3</minutes>
        <seconds>25</seconds>
    </length>
    <parody></parody>
    <lyrics></lyrics>
</song>
</CD>
</catalog>
```

### 3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your al7.xml and al7.xsd documents are located. At the prompt type the following and then press the Enter key:

**C:\Wrox> java wrox.Validate al7.xml**

It should validate with no warnings.

## How It Works

Let's break down each section of our <schema>, to figure out what is going on:

```
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.wrox.com/catalog"
    xmlns:target="http://www.wrox.com/catalog"
    elementFormDefault="qualified">
```

As we have seen in our earlier examples, we begin our XML Schema with the <schema> element. Again, we are careful to specify the correct namespace for XML Schemas. We have also included a targetNamespace attribute to indicate the namespace for our vocabulary. We have also added a namespace declaration so that we can refer to items in our targetNamespace later. Finally, we have included the attribute elementFormDefault with the value qualified.

```
<element name="catalog">
<complexType>
    <choice maxOccurs="unbounded">
        <element name="CD">
            <complexType>
                <group ref="target:AlbumGroup"/>
            </complexType>
        </element>
        <element name="cassette">
            <complexType>
                <group ref="target:AlbumGroup"/>
            </complexType>
        </element>
        <element name="record">
            <complexType>
```

```
<group ref="target:AlbumGroup"/>
</complexType>
</element>
<element name="MP3">
<complexType>
<group ref="target:AlbumGroup"/>
</complexType>
</element>
</choice>
</complexType>
</element>
```

Next, we created a global <element> declaration for our <catalog> element. Remember, our <catalog> element must be declared globally because we will be using it as our root element within our instance document. As our schema validator processes our instance document, it will encounter the <catalog> element. The schema validator will then open our XML Schema document based on the xsi:schemaLocation attribute hint and find the global declaration for the <catalog> element.

We specified the type of our <catalog> element, by declaring a local <complexType> within our <element> declaration. Within our <complexType> definition, we used a <choice> content model. We specified that the choice could occur an unbounded amount of times. The possible element choices within our content model included <CD>, <cassette>, <record>, and <MP3>. Because each of the elements had duplicate content models, we simply created local <complexType> definitions that referred to a global <group> declaration named AlbumGroup.

```
<complexType>
<group ref="target:AlbumGroup"/>
</complexType>
```

In order to refer to the global <group> declaration, we needed to prefix the group name with the namespace prefix for our targetNamespace.

```
<group name="AlbumGroup">
<sequence>
<element name="artist" type="string"/>
<element name="title" type="string"/>
<element name="genre" type="string"/>
<element name="date-released" type="string"/>
<element name="song" type="target:SongType" maxOccurs="unbounded"/>
</sequence>
</group>
```

The <group> declaration for our AlbumGroup was very straightforward. It listed the allowable elements for the content model within a <sequence> declaration. We should point out that, when declaring our <song> element, we specified its type by referring to the global <complexType> definition named SongType. We also specified that the <song> element might occur an unbounded number of times.

```
<complexType name="SongType">
<sequence>
<element name="title" type="string"/>
<element name="length" type="target:LengthType"/>
<element name="parody" type="target:ParodyType"/>
<element name="lyrics" type="string"/>
</sequence>
</complexType>
```

Our SongType <complexType> definition was very simple. We listed the allowable elements within its content model. We again specified the types for <length> and <parody> elements by referring to global <complexType> definitions.

```
<complexType name="LengthType">
<sequence>
<element name="minutes" type="string"/>
<element name="seconds" type="string"/>
</sequence>
```

```
</complexType>

<complexType name="ParodyType">
  <sequence minOccurs="0">
    <element name="title" type="string"/>
    <element name="artist" type="string"/>
  </sequence>
</complexType>
```

Our LengthType and ParodyType `<complexType>` definitions introduced nothing new. We did remember to specify that the `<title>`, `<artist>` sequence of elements might occur once or not at all.

```
</schema>
```

That completed the XML Schema for our music catalog. We will continue to add features to this XML Schema as we work our way through the rest of the chapter.

## **<attribute>**

We have spent most of this chapter looking at how to create element declarations. Of course, this is only the very first step when creating our XML Schemas. Within XML Schemas, attribute declarations are similar to element declarations. In the examples for our `<name>` element, we have already seen an attribute declaration for the title attribute.

```
<attribute
  name="name of the attribute"
  type="global type"
  ref="global attribute declaration"
  form="qualified or unqualified"
  use="optional or prohibited or required"
  default="default value"
  fixed="fixed value">
```

As we saw with element declarations, there are three primary methods for declaring attributes:

- Creating a local type
- Using a global type
- Referring to an existing global attribute

Unlike elements types, which are divided into simpleTypes and complexTypes, attribute declarations are restricted to simpleTypes. Remember, complexTypes are used to define types that will contain attributes or elements; simpleTypes are used to restrict text only content. Because an attribute can only contain text, we can only use simpleTypes to define their allowable content.

### **Creating a Local Type**

Creating a local type for an `<attribute>` declaration is similar to creating a local type for an `<element>` declaration. To create a local type, you simply include the type declaration as a child of the `<attribute>` element:

```
<attribute name="title">
  <simpleType>
    <!-- type information -->
  </simpleType>
```

```
</element>
```

In this example, we see that an attribute declaration may contain only a `<simpleType>` definition. We will look at the insides of these declarations a little later.

## Using a Global Type

Just as we saw with our `<element>` declarations, many of our attributes will have the same type of value. Instead of declaring duplicate local types throughout our schema, we can create a global `<simpleType>` definition. Within our attribute declarations, we can refer to a global type by name.

```
<attribute name="title" type="string"/>
<xss:attribute name="title" type="xss:string"/>
```

In these examples, our attribute declaration refers to a global type named string. This type is a member of the XML Schema vocabulary. When referring to the type we must include the namespace prefix, just as we would for element declarations. The first declaration should look very familiar—we used it in our `<name>` XML Schemas throughout the beginning of this chapter. It simply declares an attribute, title, which must contain a value of type string.

## Referring to an Existing Global Attribute

Referring to global `<simpleType>` definitions allows us to reuse attribute types within our XML Schema. Often, we may want to reuse entire attribute declarations, instead of just the type. XML Schemas allow you to reuse global attribute declarations within your `<complexType>` definition. To refer to a global attribute declaration, include a `ref` attribute in your declaration and specify the name of the global attribute as the value.

```
<attribute ref="target:title"/>
```

Again, the name of the attribute must be qualified with the namespace prefix. Notice that when we refer to a global attribute declaration we have no type attribute and no local type declaration. Our attribute declaration will use the type of the `<attribute>` declaration to which we are referring.

Unfortunately, reusing global attribute declarations can create problems in your instance documents because of namespaces. Each attribute that you declare globally **must** be qualified by a namespace. Remember, default namespace declarations do not apply to attributes. Because of this, you **must** have a namespace prefix for attributes that have been declared globally. Instead of dealing with these issues, most XML Schema authors utilize global attributeGroups when they need to reuse declarations. We will look at attributeGroups a little later.

## Naming Attributes

Just as we saw in our element declarations, attribute names must follow the rules for XML names that we have already learned. In the [last chapter](#), when creating names in DTDs, we saw that we had to include a namespace prefix, if one were going to be used in the instance document. Because XML Schemas are namespace aware, this is unnecessary. Simply specify the name of the attribute; the schema validator will be able to understand any prefix that is used within the instance document.

## Attribute Qualified Form

The form attribute allows you to override the default for attribute qualification. Attribute qualification functions very similarly to element qualification. If an attribute is qualified, it must have an associated namespace when it is used in the instance document. Remember, default namespaces do not apply to attributes in your instance document. Therefore, you can only qualify an attribute with a namespace prefix.

You can specify whether the attribute must be qualified by setting the value of the form attribute to qualified or unqualified. If you do not include a form attribute, the schema validator will use the value of the attributeFormDefault attribute declared in the <schema> element. Again, any attribute that is declared globally must be qualified, regardless of the form and attributeFormDefault values. Unlike elements, it is very common to have unqualified attributes within an instance document.

## Attribute Use

When declaring an attribute, we can specify that it is required, optional, or prohibited in the instance document. To control how an attribute will be used, simply include the use attribute within the <attribute> declaration and specify the appropriate value. It should also be noted that you cannot include a use attribute in a global <attribute> declaration.

By setting the value of the use attribute to prohibited, you can ensure that an attribute may not appear within your instance document. Developers commonly use prohibited attribute declarations in conjunction with attribute wildcards. Using this model, you can specify that you want to allow a large group of attributes and subsequently disallow specific attributes within the group. If you specify that an attribute is required, it must appear within the instance document. If the attribute is omitted the schema validator will raise a validity error.

Most attributes are optional—therefore, the default value for use is optional. By declaring that an attribute is optional, you indicate that it may or may not appear in the instance document. If you specify a default in your attribute declaration, the value of use cannot be required or prohibited.

## Default and Fixed Values

We have already seen that XML Schemas allow you to declare default and fixed values for elements. Declaring default and fixed values for attributes functions exactly the same. To specify a default value, simply include the default attribute with the desired value.

```
<attribute name="title" type="string" default="Mr." />
```

In the above declaration, we have specified that the default value for the title attribute is "Mr.". If the schema validator finds that the title attribute has been omitted, it will insert the attribute and set the value to "Mr."

Fixed values operate much like default values. As the schema validator is processing the file, if it encounters a fixed attribute then the parser will check that the content and fixed attribute value match. If they do not match, the parser will raise a schema validity error. If the element or content is omitted, the parser will insert the attribute with the fixed value.

To specify a fixed value, simply include the fixed attribute with the desired value:

```
<attribute name="version" type="string" fixed="1.0" />
```

When specifying fixed or default values you must ensure that the value you specify is allowable content for the type that you have declared for your attribute declaration. You are not permitted to use default and fixed values at the same time within a single attribute declaration.

## Attribute Wildcards

Earlier in the chapter, we learned about element wildcards—declarations that allowed us to include any elements from a specific namespace, or list of namespaces, within our content model. We often want to declare similar behavior for attributes. Declarations that allow you to include any attribute from a namespace are called **attribute wildcards**.

To declare an attribute wildcard, use the <anyAttribute> declaration:

```
<anyAttribute  
namespace="allowable namespaces"  
processContents="lax or skip or strict">
```

The `<anyAttribute>` declaration can only appear within a `<complexType>` or `<attributeGroup>` declaration. You are not allowed to create global `<anyAttribute>` declarations. The `<anyAttribute>` declaration allows you to control what namespace or namespaces contain attributes that may be used. Allowable namespaces are specified by including the namespace attribute. The namespace attribute allows several values:

Value	Description
<code>##any</code>	Allows attributes from all namespaces to be included as part of the wildcard
<code>##other</code>	Allows attributes from namespaces other than the targetNamespace to be included as part of the wildcard
<code>##targetNamespace</code>	Allows attributes from only the targetNamespace to be included as part of the wildcard
<code>##local</code>	Allows attributes that are not qualified by a namespace to be included as part of the wildcard
Whitespace separated list of allowable namespace URIs	Allows attributes from any listed namespaces to be included as part of the wildcard. Possible list values also include <code>##targetNamespace</code> and <code>##local</code> .

Suppose that we wanted to allow any unqualified attributes, as well as any attributes from the <http://www.w3.org/XML/1998/namespace> namespace:

```

<complexType>
  <sequence>
    <!-- content model -->
  </sequence>
  <anyAttribute namespace="##local http://www.w3.org/XML/1998/namespace"
                processContents="lax"/>
</complexType>

```

By including an attribute wildcard we can achieve this. Notice that the value of the namespace attribute is a whitespace separated list with the values `##local` and <http://www.w3.org/XML/1998/namespace>.

*The namespace <http://www.w3.org/XML/1998/namespace> contains the `xml:lang` and `xml:space` attributes. These attributes are commonly used to add information about the language or spacing of an XML document.*

When the schema validator is processing an element that contains an attribute wildcard declaration, it will validate the instance documents in one of three ways. If the value of the `processContents` attribute is set to `skip`, the processor will skip any wildcard attributes in the element. If the value of `processContents` attribute is set to `lax`, then the processor will attempt to validate the wildcard attributes if it has access to an XML Schema that defines them. If the value of the `processContents` attribute is set to `strict` (the default), the processor will attempt to validate the wildcard attributes if it has access to an XML Schema that defines them. Unlike the `lax` setting, however, if the XML Schema for the wildcard attributes cannot be found then the schema validator will raise a validity error.

## Try It Out—"Weird" XML Schema—Adding Attributes

Now that we have seen all of the various options for attribute declarations, let's update our catalog schema. In this example we will add two attributes to our `<catalog>` element.

1.

We will begin by making the necessary changes to our XML Schema. Open the file al7.xsd and make the following changes. Because we will only need to change the declaration for the <catalog> element, that is all we have shown. We will add two attribute declarations after the content model. The rest of the XML Schema will remain the same. When you are finished, save the file as al8.xsd.

```
<element name="catalog">
<complexType>
<choice maxOccurs="unbounded">
<element name="CD">
<complexType>
<group ref="target:AlbumGroup"/>
</complexType>
</element>
<element name="cassette">
<complexType>
<group ref="target:AlbumGroup"/>
</complexType>
</element>
<element name="record">
<complexType>
<group ref="target:AlbumGroup"/>
</complexType>
</element>
<element name="MP3">
<complexType>
<group ref="target:AlbumGroup"/>
</complexType>
</element>
</choice>
<attribute name="version" type="string" fixed="1.0"/>
<attribute name="name" type="string" use="required"/>
</complexType>
</element>
```

2.

Before we can validate our document, we must modify it so that it refers to our new XML Schema. We will also need to add attributes to our <catalog> element. Open the file al7.xml and make the following changes to the <catalog> element—the rest of the file will remain the same. When you are finished, save the file as al8.xml.

```
<catalog
xmlns="http://www.wrox.com/catalog"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.wrox.com/catalog al8.xsd"
version="1.0" name="My Music Library">
```

3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your al8.xml and al8.xsd documents are located. At the prompt type the following and then press the Enter key:

**C:\Wrox> java wrox.Validate al8.xml**

This should validate with no errors.

## How It Works

In this Try It Out we added two attributes to our <catalog> element. We did this by adding the attribute declarations after the content model of the anonymous complexType definition within the element declaration. Let's look at each of these attribute declarations in more detail

```
<attribute name="version" type="string" fixed="1.0"/>
```

Our first attribute declaration defined the version attribute. We indicated that its value must be type string—meaning any text value is allowed. When we declared the attribute, we included a fixed attribute with the value 1.0. This means that if the version attribute appears within our document then it must have the value 1.0. If the version attribute is omitted, our schema validator will insert the attribute with the value 1.0.

```
<attribute name="name" type="string" use="required"/>
```

In our second attribute declaration, we defined the name attribute. Again, we have indicated that the attribute value must be type string. In this attribute declaration, we have included a use attribute with the value required. This means that the name attribute must appear within our instance document. If the attribute is omitted the schema validator will raise an error.

Remember, within the instance document attributes may appear in any order. In addition, no attribute may appear more than once in a single element.

## <attributeGroup>

We have seen that, by creating a global <group> declaration, we can define reusable groups of elements. In addition to element groups, XML Schema also allows us to define attribute groups.

```
<attributeGroup  
  name="name of global attribute group">
```

Often we will need to use the same set of attributes for many elements. In such a case, it is easier in to create a global attribute group that can be reused in our <complexType> definitions. In DTDs this was not possible without using parameter entities.

The <attributeGroup> declaration is very similar to the <group> declaration. Like <group> declarations, all global <attributeGroup> declarations must be named. Simply specify the name attribute with the desired name. Again, the name that you specify must follow the rules for XML names and it should not include a prefix. The basic structure of a global <attributeGroup> declaration follows:

```
<attributeGroup name="NameAttGroup">  
  <!-- attribute declarations go here -->  
</attributeGroup>
```

Instead of allowing content model declarations like the <group> declarations we saw earlier the chapter, the <attributeGroup> declaration allows <attribute> declarations as children. It also allows attribute wildcards and references to global <attribute> and <attributeGroup> declarations.

Although <attributeGroup> declarations may include references to other global <attributeGroup> declarations as part of the content model, group declarations may not recursively refer to themselves. For example, the following is an illegal <attributeGroup> declaration:

```
<attributeGroup name="AttGroup1">  
  <attributeGroup ref="target:AttGroup1"/>  
</attributeGroup >
```

This is illegal as well:

```
<attributeGroup name="AttGroup1">  
  <attributeGroup ref="target:AttGroup2"/>
```

```
</attributeGroup >
<attributeGroup name="AttGroup2">
    <attributeGroup ref="target:AttGroup1"/>
</attributeGroup >
```

This second declaration is illegal, because the declaration indirectly refers to itself.

To use an `<attributeGroup>`, simply include an `<attributeGroup>` reference declaration within a `<complexType>` or global `<attributeGroup>` declaration. To specify which `<attributeGroup>` you are referring to, include the `ref` attribute with the name of the global `<attributeGroup>` declaration as the value. Just as we have seen with our other references, you need to specify the namespace when referring to the global declaration. To do this, include the namespace prefix in the value.

## Try It Out—"Weird" XML Schema—Using a Global Attribute Group

Let's redesign our schema so that we can create a reusable global `<attributeGroup>` declaration. We will move our `name` attribute declaration into our `attributeGroup`. This way, if we want to add `name` attributes to other elements, we can simply reuse the global `attributeGroup`.

1.

We will begin by making the necessary changes to our XML Schema. Open the file `a18.xsd` and make the following changes. When you are finished, save the file as `a19.xsd`.

```
<?xml version="1.0"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.wrox.com/catalog"
    xmlns:target="http://www.wrox.com/catalog"
    elementFormDefault="qualified">

    <attributeGroup name="NameAttGroup">
        <attribute name="name" type="string" use="required"/>
    </attributeGroup>

    <element name="catalog">
        <complexType>
            <choice maxOccurs="unbounded">
                <element name="CD">
                    <complexType>
                        <group ref="target:AlbumGroup"/>
                    </complexType>
                </element>
                <element name="cassette">
                    <complexType>
                        <group ref="target:AlbumGroup"/>
                    </complexType>
                </element>
                <element name="record">
                    <complexType>
                        <group ref="target:AlbumGroup"/>
                    </complexType>
                </element>
                <element name="MP3">
                    <complexType>
                        <group ref="target:AlbumGroup"/>
                    </complexType>
                </element>
            </choice>
            <attribute name="version" type="string" fixed="1.0"/>
            <attributeGroup ref="target:NameAttGroup"/>
        </complexType>
    </element>
</schema>
```

```
</element>

<group name="AlbumGroup">
  <sequence>
    <element name="artist" type="string"/>
    <element name="title" type="string"/>
    <element name="genre" type="string"/>
    <element name="date-released" type="string"/>
    <element name="song" type="target:SongType" maxOccurs="unbounded"/>
  </sequence>
</group>

<complexType name="SongType">
  <sequence>
    <element name="title" type="string"/>
    <element name="length" type="target:LengthType"/>
    <element name="parody" type="target:ParodyType"/>
    <element name="lyrics" type="string"/>
  </sequence>
</complexType>

<complexType name="LengthType">
  <sequence>
    <element name="minutes" type="string"/>
    <element name="seconds" type="string"/>
  </sequence>
</complexType>

<complexType name="ParodyType">
  <sequence minOccurs="0">
    <element name="title" type="string"/>
    <element name="artist" type="string"/>
  </sequence>
</complexType>

</schema>
```

## 2.

Before we can validate our XML document against our schema, we must modify it so that it refers to our new XML Schema. Open the file al8.xml and change the xsi:schemaLocation attribute, as follows. When you are finished, save the file as al9.xml.

```
xsi:schemaLocation="http://www.wrox.com/catalog al9.xsd"
```

## 3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your al9.xml and al9.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate al9.xml
```

This should validate with no errors.

## How It Works

In this Try It Out, we have modified our XML Schema to use a global <attributeGroup> declaration. First, we created a global <attributeGroup> declaration named NameAttGroup. Within our declaration, we included the declaration for our name attribute. Within the <complexType> definition for our <catalog> element, instead of including the attribute declaration, we created an <attributeGroup> reference declaration. When referring to the global

<attributeGroup> declaration, we included a ref attribute with the value targetNameAttGroup.

As our schema validator is processing our instance document, it will process the <catalog> element similar to our earlier examples. When it encounters the <catalog> element, it will look it up in the XML Schema. Once it finds the declaration, it will find the associated type (in this case it is a local <complexType> definition). When the schema validator encounters the <attributeGroup> reference declaration, it will treat the name <attribute> declaration within the group as if it had been included directly within the <complexType> definition.

## Datatypes

Throughout this chapter, we have seen how to declare elements and attributes using <complexType> definitions. At the start of the chapter, however, we promised that we would see how to define the allowable content for text-only elements and attribute values. It's time that we made good on that promise.

The XML Schema Recommendation allows you to use

- Built-in datatypes
- User defined datatypes

### Built-in Datatypes

In our examples throughout this chapter, we have used the string type for our text only content. The string type is a primitive data type that allows any textual content. XML Schemas provide a number of simpleTypes that allow you to exercise greater control over textual content in your XML document. The following table lists all of the simpleTypes built into XML Schemas:

Type	Description
string	Any character data
normalizedString	A whitespace normalized string where all spaces, tabs, carriage returns, and line feed characters are converted to single spaces
token	A string that does not contain sequences of two or more spaces, tabs, carriage returns or line feed characters
byte	A numeric value from -128 to 127
unsignedByte	A numeric value from 0 to 255
base64Binary	Base 64 encoded binary information
hexBinary	Hexadecimal encoded binary information
integer	A numeric value representing a whole number
positiveInteger	An integer whose value is greater than 0

negativeInteger	An integer whose value is less than 0
nonNegativeInteger	An integer whose value is 0 or greater
nonPositiveInteger	An integer whose value is less than or equal to 0
int	A numeric value from -2147483648 to 2147483647
unsignedInt	A numeric value from 0 to 4294967295
long	A numeric value from -9223372036854775808 to 9223372036854775807
unsignedLong	A numeric value from 0 to 18446744073709551615
short	A numeric value from -32768 to 32767
unsignedShort	A numeric value from 0 to 65535
decimal	A numeric value that may or may not include a fractional part
float	A numeric value that corresponds to the IEEE single-precision 32-bit floating-point type defined in the standard IEEE 754-1985. (IEEE 754-1985 can be found at <a href="http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html">http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html</a> )
double	A numeric value that corresponds to the IEEE double-precision 64-bit floating point type defined in the standard IEEE 754-1985
Boolean	A logical value, including true, false, 0, and 1
time	An instant of time that occurs daily as defined in section 5.3 of ISO 8601. For example, 15:45:00.000 is a valid time value. (ISO 8601 can be found at <a href="http://www.iso.ch/market/8601.pdf">http://www.iso.ch/market/8601.pdf</a> )
dateTime	An instant of time including both a date and a time value as defined in section 5.4 of ISO 8601. For example, 1998-07-12T16:30:00.000 is a valid dateTime value
duration	A span of time as defined in section 5.5.3.2 of ISO 8601. For example, P30D is a valid duration value indicating a duration of 30 days
date	A date according to the Gregorian Calendar as defined in section 5.2.1 of ISO 8601. For example, 1995-05-07 is a valid date value
gMonth	A month in the Gregorian Calendar as defined in section 3 of ISO 8601. For example, --07-- is a valid gMonth value

gYear	A year in the Gregorian Calendar as defined in section 5.2.1 of ISO 8601. For example, 1998 is a valid gYear value
gYearMonth	A specific month and year in the Gregorian Calendar as defined in section 5.2.1 of ISO 8601. For example, 1998-07 is a valid gYearMonth value
gDay	A recurring day of the month as defined in section 3 of ISO 8601, such as the 12th day of the month. For example, ---12 is a valid gDay value
gMonthDay	A recurring day of a specific month as defined in section 3 of ISO 8601, such as the 12th day of July. For example, --07-12 is a valid gMonthDay value
Name	An XML Name
QName	A Qualified XML Name as defined in the Namespaces Recommendation
NCName	A Non-Colonized XML Name which does not include a namespace prefix or colon as defined in the Namespaces Recommendation
anyURI	A valid Uniform Resource Identifier
language	A language constant as defined in RFC 1766, such as en-US (RFC 1766 can be found at <a href="http://www.ietf.org/rfc/rfc1766.txt">http://www.ietf.org/rfc/rfc1766.txt</a> )

In addition to the types listed, the XML Schema Recommendation also allows the types defined within the XML Recommendation. These types include ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, and NMTOKENS. These types were covered in the [last chapter](#).

Just as we have used the string type throughout our examples, any of the above types can be used to restrict the allowable content within your elements and attributes. Suppose we wanted to modify the declarations of our <minutes> and <seconds> elements within our catalog XML Schema to ensure that the users of our XML Schema enter valid values. We could modify our declarations as follows:

```
<element name="minutes" type="nonNegativeInteger"/>
<element name="seconds" type="nonNegativeInteger"/>
```

Now, instead of allowing any textual content, we require that the user specify a numeric value that is greater than, or equal to, zero. We could have also used the duration built-in type as each of our values represent spans of time. We'll leave the declaration how it is, though, as specifying minutes and seconds as numeric values is easier than creating duration values. For a more in-depth look at these types, see [Appendix E](#), "Schema Datatypes", or see the XML Schema Recommendation at <http://www.w3.org/TR/xmlschema-2/>.

## Try It Out—"Weird" XML Schema—Using the Built-in XML Schema datatypes

Let's modify our catalog example, so that we can take advantage of the built-in XML Schema datatypes:

1.

We will begin by making the necessary changes to our XML Schema. Open the file a19.xsd and make the following changes. When you are finished, save the file as a110.xsd.

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

```
xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/catalog"
xmlns:target="http://www.wrox.com/catalog"
elementFormDefault="qualified">

<attributeGroup name="NameAttGroup">
  <attribute name="name" type="string" use="required"/>
</attributeGroup>

<element name="catalog">
  <complexType>
    <choice maxOccurs="unbounded">
      <element name="CD">
        <complexType>
          <group ref="target:AlbumGroup"/>
        </complexType>
      </element>
      <element name="cassette">
        <complexType>
          <group ref="target:AlbumGroup"/>
        </complexType>
      </element>
      <element name="record">
        <complexType>
          <group ref="target:AlbumGroup"/>
        </complexType>
      </element>
      <element name="MP3">
        <complexType>
          <group ref="target:AlbumGroup"/>
        </complexType>
      </element>
    </choice>
    <attribute name="version" type="float" fixed="1.0"/>
    <attributeGroup ref="target:NameAttGroup"/>
  </complexType>
</element>

<group name="AlbumGroup">
  <sequence>
    <element name="artist" type="string"/>
    <element name="title" type="normalizedString"/>
    <element name="genre" type="token"/>
    <element name="date-released" type="gYearMonth"/>
    <element name="song" type="target:SongType" maxOccurs="unbounded"/>
  </sequence>
</group>
<complexType name="SongType">
  <sequence>
    <element name="title" type="normalizedString"/>
    <element name="length" type="target:LengthType"/>
    <element name="parody" type="target:ParodyType"/>
    <element name="lyrics" type="string"/>
  </sequence>
</complexType>

<complexType name="LengthType">
  <sequence>
    <element name="minutes" type="nonNegativeInteger"/>
    <element name="seconds" type="nonNegativeInteger"/>
  </sequence>
</complexType>

<complexType name="ParodyType">
  <sequence minOccurs="0">
```

```
<element name="title" type="normalizedString"/>
<element name="artist" type="string"/>
</sequence>
</complexType>

</schema>
```

2.

Before we can schema validate our XML document, we must modify it so that it refers to our new XML Schema. Open the file al9.xml and change the xsi:schemaLocation attribute, as follows. When you are finished, save the file as al10.xml.

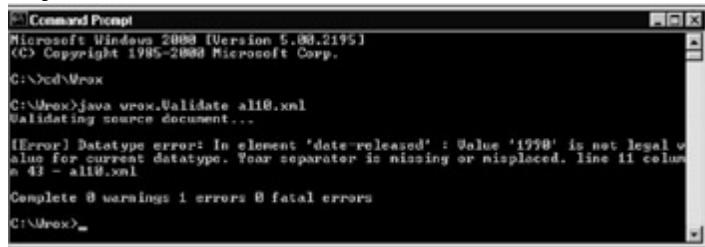
```
xsi:schemaLocation="http://www.wrox.com/catalog al10.xsd"
```

3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your al10.xml and al10.xsd documents are located. At the prompt type the following and then press the Enter key:

**C:\Wrox**

**> java wrox.Validate al10.xml**



You should see the following output:

Why did we get an error? Because we are using more restrictive types, we will need to update our instance document.

4.

Open the file al10.xml and modify the <date-released> element; when you are finished, save the file.

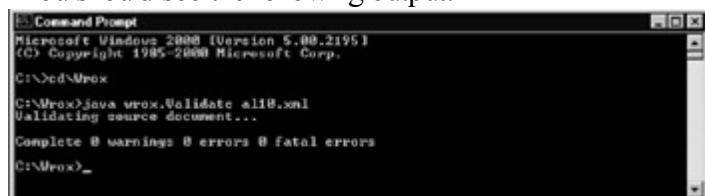
```
<date-released>1985-06</date-released>
```

5.

Now that we have fixed the error, we are ready to begin validating our XML instance document against our XML Schema. At the command prompt type the following and then press the Enter key:

**C:\Wrox> java wrox.Validate al10.xml**

You should see the following output:



In this Try It Out we used some of the XML Schema built-in datatypes. These data types allow us to exercise more control over the textual content within our instance documents. As we saw in the [last chapter](#) DTDs were not capable of advanced data typing. Let's look at some of the types we used in a little more detail.

```
<attribute name="version" type="float" fixed="1.0"/>
```

We began by changing the type of our `version` attribute from string to float. This is a perfect fit because our version number must always be a valid floating-point number.

```
<element name="title" type="normalizedString"/>
```

We modified all of our `<title>` element declarations to use the `normalizedString` type. This type allows us to guarantee that there will be no extra spaces or line feeds within the content of our `<title>` element.

```
<element name="genre" type="token"/>
```

Because our `genre` must be a single value, we decided to use the `token` type. The `token` type is somewhat less restrictive than the `name` type, but more restrictive than the `string` type.

```
<element name="date-released" type="gYearMonth"/>
```

We modified our `<date-released>` element so that it used the built-in type `gYearMonth`. What we didn't do was modify our instance document right away, and it caused an error when we schema validated our document the first time. Because the `gYearMonth` value is based on the international standard ISO 8601, we must follow specific rules when including the value. As we saw in the example in the datatypes table above, simple `gYearMonth` values follow the format:

CCYY-MM

where CC is the hundreds of years in the Gregorian calendar, YY represents the year, and MM represents the month. Therefore, the value 1990 was not valid because it did not include the month. Instead we changed the value to 1985-06 (the initial release of *Dare to be Stupid*), which included the hundreds of years in the Gregorian calendar, the year, and the two-digit month.

As our schema validator processes our document, not only is it checking that the element content models we have specified are correct, it is also checking that the textual data we have included in our elements is valid based on the type we specified.

## User Defined Datatypes

Although the XML Schema Recommendation includes a wealth of built-in datatypes—it doesn't include everything. As we are developing our XML Schemas, we will run into many elements and attribute values that require a type not defined in the XML Schema Recommendation. Consider our `<seconds>` element declaration. Although we have improved it by restricting its content to no negative numeric values, it will still accept unwanted values such as

```
<seconds>393</seconds>
```

According to our declaration, the value 393 is valid because it is a non-negative numeric value. Is this really what we wanted? What we need is a numeric type that allows the values from 0 to 59 (we chose 59 because if there are 60 or more seconds we would need to increment the value of our `<minute>` element). No such type exists within the XML Schema Recommendation so we must create a new type using a `<simpleType>` definition.

## <simpleType>

Often, when designing our XML Schemas we will need to design our own datatypes. We can create custom user defined datatypes using the <simpleType> definition.

```
<simpleType  
    name="name of the simpleType"  
    final="#all or list or union or restriction">
```

When we declare a simpleType, we must always base our declaration on an existing datatype. The existing datatype may be a built-in XML Schema datatype or it may be another user datatype. Because we must derive every <simpleType> definition from another datatype, <simpleType> definitions are often called **derived types**. There are three primary derived types:

- Restriction types
- List types
- Union types

In this section, we learn the basics of <simpleType> declarations and user-defined types. We will introduce advanced features in the [next chapter](#). In addition, [Appendix E](#), "Schema Datatypes" covers datatypes in detail. If you are looking for an in-depth treatment of all of the features and options, see *Professional XML Schemas* by Stephen Mohr et al. Wrox Press (ISBN 1-861005-47-4).

## <restriction>

The most common <simpleType> derivation is the restriction type. Restriction types are declared using the <restriction> declaration.

```
<restriction  
    base="name of the simpleType you are deriving from">
```

Fundamentally, a derived type, declared using the <restriction> declaration, is a subset of its base type. Facets control all simpleTypes within XML Schemas. A **facet** is a single property of a simpleType. For example, the built-in numeric type nonNegativeInteger was created by setting the facet minInclusive to zero. This specifies that the minimum value allowed for the type is zero. By constraining the facets of existing types, we can create our own, more restrictive, types.

There are twelve constraining facets:

Facet	Description
minExclusive	Allows you to specify the minimum value for the type which excludes the value you specify
minInclusive	Allows you to specify the minimum value for the type which includes the value you specify

maxExclusive	Allows you to specify the maximum value for the type which excludes the value you specify
maxInclusive	Allows you to specify the maximum value for the type which includes the value you specify
totalDigits	Allows you to specify the total number of digits in a numeric type
fractionDigits	Allows you to specify the number of fractional digits in a numeric type (for example the number of digits to the right of the decimal point)
length	Allows you to specify the number of items in a list type or the number of characters in a string type
minLength	Allows you to specify the minimum number of items in a list type or the minimum number of characters in a string type
maxLength	Allows you to specify the maximum number of items in a list type or the maximum number of characters in a string type
enumeration	Allows you to specify an allowable value in an enumerated list
whiteSpace	Allows you to specify how whitespace should be treated within the type
pattern	Allows you to restrict string types using regular expressions

Not all types use every facet. In fact, most types can only be constrained by a couple of facets. For a complete list of what constraining facets can be used when restricting the built-in XML Schema types, see [Appendix E](#).

Within a <restriction> declaration, we can specify our base type using the base attribute. The value of the base attribute is a reference to a global simpleType, or built-in XML Schema datatype. As we have seen with all references in our XML Schema, the reference is a namespace qualified value and, therefore, may need to be prefixed. The <restriction> declaration also allows you to derive your type from a local <simpleType> definition.

For example, you could create a restriction type that uses enumeration facets, to define the allowable titles in a title attribute:

```
<attribute name="title">
  <simpleType>
    <restriction base="string">
      <enumeration value="Mr."/>
      <enumeration value="Mrs."/>
      <enumeration value="Ms."/>
      <enumeration value="Miss"/>
      <enumeration value="Dr."/>
      <enumeration value="Rev"/>
    </restriction>
  </simpleType>
</attribute>
```

In this declaration, we have created a <restriction> declaration with the base type string. Within our restriction, we

have specified multiple enumeration facets, to create a list of all of the allowable values for our type.

## Try It Out—"Weird" XML Schemas—Creating a Restriction simpleType

As we saw in the *User Datatypes* section above, our <seconds> element should be more restrictive. Now that we know how to create our own <simpleType> definitions, let's create a <restriction> type for our <seconds> element:

1.

We will begin by making the necessary changes to our XML Schema. Open the file all0.xsd and make the following changes. We will only need to modify the <element> declaration for the <seconds> element. The rest of the XML Schema will remain the same. When you are finished, save the file as all1.xsd.

```
<element name="seconds">
<simpleType>
  <restriction base="nonNegativeInteger">
    <maxInclusive value="59"/>
  </restriction>
</simpleType>
</element>
```

2.

Before we can schema validate our XML document, we must modify it so that it refers to our new XML Schema. Open the file all0.xml and change the xsi:schemaLocation attribute, as follows. When you are finished, save the file as all1.xml.

```
xsi:schemaLocation="http://www.wrox.com/catalog all1.xsd"
```

3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your all1.xml and all1.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate all1.xml
```

This should validate without any errors

## How It Works

In this Try It Out, we modified our <seconds> element declaration. We created an internal <simpleType> definition that was a restriction derived from nonNegativeInteger. Because there are only 59 seconds within a minute, we wanted to allow numbers with a maximum value of 59. To enforce this rule we modified the maxInclusive facet of our simple type, setting the value to 59.

## <list>

Often we will need to create a list of items. Using a <list> declaration we can base our list items on a specific simpleType.

```
<list
  itemType="name of simpleType used for items">
```

When creating our <list> declaration we could specify the type of items in our list by including the itemType attribute. The value of the itemType attribute should be a reference to a global <simpleType> definition or built-in XML Schema datatype. Again, the reference is a namespace qualified value and, therefore, may need to be prefixed. The

<list> declaration also allows you to specify your itemType by creating a local <simpleType> definition.

When choosing the itemType you must remember you are creating a whitespace separated list. Because of this, your items cannot contain whitespace. Therefore, types that include whitespace cannot be used as itemTypes. A side effect of this limitation is that you cannot create a list whose itemType is itself a list.

For example:

```
<element name="ages">
  <simpleType>
    <list itemType="positiveInteger"/>
  </simpleType>
</element>
```

In the above declaration, we have created a list of positiveInteger values. A valid <ages> element might appear as:

```
<ages>4 9 12</ages>
```

### <union>

Finally, when creating your derived types, you may need to combine two or more types. By declaring a <union> type, you can validate your type against the allowable values in multiple types.

```
<union
  memberTypes="whitespace separated list of types">
```

When creating our <union> declaration we could specify the types we are combining by including the memberTypes attribute. The value of the memberTypes attribute should be a whitespace separated list of references to global <simpleType> definitions or built-in XML Schema datatypes. Again, these references are namespace qualified values and, therefore, may need to be prefixed. The <union> declaration also allows you to specify your memberTypes by creating local <simpleType> definitions.

Suppose you declared the following <simpleType> definitions:

```
<simpleType name="CatBreeds">
  <restriction base="string">
    <enumeration value="Abyssinian"/>
    <enumeration value="Himalayan"/>
    <enumeration value="Siamese"/>
  </restriction>
</simpleType>
<simpleType name="DogBreeds">
  <restriction base="string">
    <enumeration value="Springer-Spaniel"/>
    <enumeration value="Labrador"/>
    <enumeration value="Beagle"/>
  </restriction>
</simpleType>
```

In the above example, we have created two <simpleType> definitions that list various breeds of cats and dogs. Now, suppose you wanted to declare a <pet> element that would allow you to specify your breed of pet:

```
<element name="pet">
  <simpleType>
    <union memberTypes="target:CatBreeds target:DogBreeds"/>
  </simpleType>
```

```
</element>
```

In this declaration, we have created a union of our two simpleTypes CatBreeds and DogBreeds. With this declaration, our <pet> element can contain one of the values available within either CatBreeds or DogBreeds. Some **valid** <pet> elements include:

```
<pet>Beagle</pet>
<pet>Abyssinian</pet>
```

Some **invalid** <pet> elements include:

```
<pet>Terrier</pet>
<pet>Siamese Labrador</pet>
```

The first two <pet> elements both contain valid values. The third <pet> element is invalid because the value Terrier is not listed in either of our unioned types. The fourth <pet> element is invalid because, although it appears to contain two values—the schema validator will treat this as a single value. The value Siamese Labrador is not listed in either of our unioned types.

---

 PREVIOUS

[< Free Open Study >](#)

NEXT 

# Documenting XML Schemas

Throughout your programming career, and even in this book, you have heard that documenting your code is one of the best habits you can develop. The XML Schema Recommendation provides several mechanisms for documenting your code:

- Comments
- Attributes from other namespaces
- Annotations

Let's look at each of these documenting methods in more detail.

## Comments

In [Chapter 2](#), we learned that XML allows you to introduce comments in your XML documents. Because the XML Schema is, itself, an XML document you can freely intersperse XML comments throughout the declarations, as long as you follow the rules for XML well-formedness.

```
<!-- This complexType is inherited from our NameType and it should be used when the
person hasn't provided a middle name -->
<complexType name="RestrictedNameType">
  <complexContent>
    <restriction base="target:NameType">
      <sequence>
        <element name="first" type="string"/>
        <!-- Notice that we have omitted the middle element -->
        <element name="last" type="string"/>
      </sequence>
      <attribute name="title" type="string"/>
    </restriction>
  </complexContent>
</complexType>
```

In the above XML Schema fragment, we have placed two comments. The first comment simply introduces our complexType and when it should be used. If someone were reading our XML Schema, this would surely give him or her some guidance when creating his or her instance documents. We used the second comment to show what part of the declaration was restricted.

While these comments are useful for someone reading our XML Schema, many processors will not report XML Comments. Therefore, the document must be read by a human in order for the comments to be useful in all cases.

## Attributes from Other Namespaces

The XML Schema Recommendation provides a second mechanism for documenting your XML Schemas. All of the elements defined within the XML Schema vocabulary allow you to include any attribute from another namespace. You can use the alternative attributes to introduce descriptive data that is included with your element.

Suppose you declared a description attribute within the namespace <http://www.wrox.com/comments>. You could use this attribute throughout your XML Schema to include comments that are embedded within your elements.

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  elementFormDefault="qualified"
  xmlns:doc="http://www.w3.org/comment">
  <complexType name="SongType" doc:description="The SongType describes a song - it is used in AlbumType within the music catalog">
    <sequence>
      <element name="title"
        type="normalizedString"
        doc:description="The title of the song"/>
      <element name="length"
        type="target:LengthType"
        doc:description="The length of the song, seconds and minutes"/>
      <element name="parody"
        type="target:ParodyType"
        doc:description="The title and artist of the parodied song"/>
    </sequence>
  </complexType>
  ...
</schema>
```

In this example, we have included a namespace declaration for a fictitious vocabulary for comments. We assumed that within our fictitious namespace there was a declaration for the description attribute. Throughout our XML Schema document, we introduced descriptions of the items we were declaring by including the description attribute from the comment vocabulary.

As a schema validator processes the document, it will ignore all of the description attributes because they are declared in another namespace. The attributes can still be used to pass information on to other applications. In addition, the comments provide extra information for those reading our XML Schema.

## Annotations

The primary documenting features introduced in the XML Schema Recommendation are called annotations. **Annotations** allow you to provide documentation information, as well as additional application information.

```
<annotation
  id="ID">
```

The <annotation> declaration can appear as a child of most XML Schema declarations. The <annotation> declaration allows you to add two forms of information to your declarations:

- application information
- documentation information

Each <annotation> declaration may contain the elements <appinfo> and <documentation>. These elements may contain **any** XML content from **any** namespace. Each of these elements may also contain a source attribute. The source attribute is used to refer to an external file that may be used for application information or documentation

information. Typically, <appinfo> declarations are used to pass information such as example files, associated images, or additional information for validation. Annotations usually include <documentation> declarations to describe the features, or uses, of a particular declaration within the XML Schema.

Consider the following example:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
  <annotation>
    <appinfo source="file:///c:/Wrox/name-instance.xml"/>
    <documentation>
      <p>
        The name vocabulary was created for a Chapter 2 sample. We have upgraded it to
        an XML Schema. The appinfo of this <pre>&lt;annotation&gt;</pre>
        element points to a sample XML file. The sample should be used <i>only as an example</i>
      </p>
    </documentation>
  </annotation>
  <element name="name">
    <annotation>
      <documentation source="name.html"/>
    </annotation>
    <complexType>
      <sequence>
        <element name="first" type="string"/>
        <element name="middle" type="string"/>
        <element name="last" type="string"/>
      </sequence>
      <attribute name="title" type="string"/>
    </complexType>
  </element>
</schema>
```

Within this example XML Schema, we have two <annotation> declarations. Our first <annotation> declaration is contained within our <schema> element. It is used to add information that is applicable to the entire XML Schema document.

Within our first <annotation> declaration, we have included both <appinfo> and <documentation> element. We didn't include any content within our <appinfo> element. Instead, we included a source attribute that pointed to an example XML instance document. Of course, schema validator must be programmed to utilize the <appinfo> declaration. Many programs define different behavior for the <appinfo> declaration. Often the <appinfo> declaration contains additional validation information, such as other schema languages.

#### Important

The Schematron is another language for defining your vocabulary. We will look at it in more detail in the [next chapter](#). Schematron definitions, because they offer additional features, are often embedded directly within the <appinfo> declaration. Several processors have been written that can use Schematron in conjunction with XML Schemas. This is covered in detail within *Professional XML Schemas*, Wrox Press, ISBN 1-861005-47-4

The <documentation> declaration within our first annotation contains an HTML fragment that could be used when generating a user's manual for our XML Schema. Our second annotation included only a <documentation> declaration. Unlike the first <documentation> declaration, the second declaration was empty, and instead used the source attribute to refer to an external file called name.html.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

In this chapter, we've learned how to create XML Schemas that can be used to schema validate our XML documents. We started again with our simple name examples and then progressed to our more complex catalog examples.

We've learned:

- The advantages of XML Schemas over Document Type Definitions
- How to associate an XML Schema with an XML document
- How to declare element and attribute types
- How to declare groups and attribute groups
- How to specify allowable XML content using simpleTypes and complexTypes
- How to document your XML Schema

While we have not discussed all of the options available within XML Schemas, we have established a foundation upon which you can build many XML Schemas. However, we have been touting the power available within XML Schemas so, in the [next chapter](#), we're going to look at some of its more advanced features.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 7: Advanced XML Schemas

## Overview

In the [last chapter](#), we learned the fundamentals of creating and using XML Schemas. So far, we have seen how to declare elements and attributes in our XML Schemas. We have also learned how to create reusable groups and types. Throughout this chapter, we will look at some of the more advanced features of XML Schemas. We will learn how to apply object oriented programming principles to our XML Schemas. We will also see how we can create modular XML Schemas that are comprised of multiple files. In addition, we will look at some alternative schema formats at the end of this chapter.

In this chapter we'll learn:

- How to inherit one type from another
- How to substitute one type for another
- How to create XML Schemas from multiple documents
- How to declare Notations
- Alternative schema formats

Although this chapter covers the advanced features of XML Schemas, several of these topics are extremely complex, and well beyond the scope of this book. By the end of this chapter, you should have a good understanding of these topics. You may, however, want to learn more. As we mentioned in the [last chapter](#), *Professional XML Schemas*, Stephen Mohr *et al.* Wrox Press (ISBN 1-861005-47-4), covers these topics in detail. You can also find more information in the XML Schema Recommendation itself at <http://www.w3.org/TR/xmlschema-1/> and <http://www.w3.org/TR/xmlschema-2/>.

# Inheritance

We have already learned how to create and reuse types within our XML Schemas. The XML Schema Recommendation allows you to do much more than simply create and reuse types; the Recommendation allows you to create a hierarchy of types that lets you model complex object oriented relationships. The inheritance feature of XML Schema types was designed so that XML Schemas could be used to model data from modern programming languages.

Within object oriented programming languages such as C++, C#, Java, Object Pascal, or Visual Basic .NET, object types can inherit the behavior of the types they are derived from. This has some parallels with genetics?we each inherit physical traits from our parents, and indirectly from our grandparents, all the way back to the beginning.

We have already seen a form of inheritance in XML Schemas. When creating our simpleTypes, we derived our declarations from existing data types using the restriction mechanism. Our declarations inherited the traits of their base type. In many of our declarations, we restricted some of the facets to create our derived types. In fact, within XML Schemas, there are two primary models for creating inherited types:

- Extension
- Restriction

The XML Schema Recommendation allows you to extend and restrict simpleTypes and complexTypes. Utilizing inheritance in your type declarations can be very helpful. For example, it can reduce the size of your XML Schema by allowing you to reuse base types, and listing only what has been added or restricted. In XML Schemas, **base types** are complexType or simpleType definitions that are extended or restricted. In our earlier genetics analogy, your base type is your mother or father, whose base type is, in turn, their parent.

## Extension

The extension mechanism allows us to derive new types from existing ones. The new types will be extensions of the types from which they are derived?they will contain all of the original type information and will add new information. Within XML schemas you can extend simpleType and complexType definitions.

### Extending `<complexType>` Definitions

When extending a base `<complexType>` definition, you add elements or attributes. Your base `<complexType>` definition should be a global, named type. Consider the declaration for our `<name>` element in the [last chapter](#):

```
<complexType name="NameType">
<sequence>
  <element name="first" type="string"/>
  <element name="middle" type="string"/>
  <element name="last" type="string"/>
</sequence>
<attribute name="title" type="string"/>
</complexType>
```

One of the methods we used when declaring the allowable content for a `<name>` element was to create a global `<complexType>` definition named NameType. Suppose we wanted to declare additional allowed elements or attributes within this type.

```
<complexType name="ExtendedNameType">
<complexContent>
  <extension base="target:NameType">
    <sequence>
      <element name="age" type="positiveInteger"/>
      <element name="birthdate" type="dateTime"/>
    </sequence>
    <attribute name="gender" type="string"/>
  </extension>
</complexContent>
</complexType>
```

In this extension, we have specified that the resulting type can have element content and may have attributes (using the `<complexContent>` declaration). Within our `<extension>` declaration, we have specified that we are basing our new, extended, type on the `NameType` declaration. We did this by setting the value of the `base` attribute to `target:NameType`. If our schema validator encountered an element that was declared using the `ExtendedNameType` it would allow the elements `<first>`, `<middle>`, `<last>`, `<age>` and `<birthdate>` in its content, as well as the `title` and `gender` attributes. Effectively, the schema validator treats the extended type as if it had been declared as follows:

```
<complexType name="ExtendedNameType">
<sequence>
  <element name="first" type="string"/>
  <element name="middle" type="string"/>
  <element name="last" type="string"/>
</sequence>
<sequence>
  <element name="age" type="positiveInteger"/>
  <element name="birthdate" type="dateTime"/>
</sequence>
<attribute name="title" type="string"/>
<attribute name="gender" type="string"/>
</complexType>
```

Notice that the `<sequence>` declarations are not combined; instead, they are interpreted as two separate sequences that appear one after the other. Keep in mind, this combined example is only how the schema validator interprets the newly created type, it is never actually declared this way. It is important to understand that any extensions you create will be appended to the existing content model. Because of this, you cannot insert elements anywhere in the base type's content model. Also, when extending a `complexType`, you cannot override the properties of existing elements or attributes?you can only add new ones.

## Extending `<simpleType>` Definitions

Often we need to declare an element that has simple content but also contains attributes. In order to do this we must extend a `<simpleType>` definition. In order to extend a `<simpleType>` definition we create a new `<complexType>` that uses a `<simpleType>`, or an XML Schema built-in type, as its base type. Suppose we wanted to add a `gender` attribute to our `<artist>` element from the [last chapter](#). We declared our original `<artist>` element as follows:

```
<element name="artist" type="string"/>
```

We declared that our `<artist>` element must contain string content. In the [last chapter](#), we learned that text-only content is constrained by `simpleTypes` or built-in XML Schema types. We also learned that in order to add attributes to our element we must use a `<complexType>` definition. Because of this when we are extending a `simpleType` we must use a `<complexType>` definition:

```
<element name="artist">
<complexType>
  <simpleContent>
    <extension base="string">
```

```
<attribute name="gender" type="string"/>
</extension>
</simpleContent>
</complexType>
</element>
```

In the above declaration, we have created an anonymous complexType definition for our <artist> element. Within the <complexType> definition, we have specified that our element will still have simple content (text only content), but that we are extending what the simpleType can do. We have done this with the <simpleContent> and <extension> declarations respectively. Within the <extension> declaration, we have included the attribute base with the value string. This indicates that the base simpleType that we are extending is the XML Schema built-in type string. Within the <extension> declaration, we have added a single attribute declaration for the gender attribute. When declaring a simpleType extension as we have done, you are not permitted to include element declarations of any kind.

The following examples would be valid according to the above declaration:

```
<artist>"Weird Al" Yankovic</artist>
<artist gender="male">"Weird Al" Yankovic</artist>
<artist gender="male"/>
```

In the first example, we have our favorite artist's name, but no gender attribute. In the second we have both a gender attribute and text content. In the last example we have only the gender attribute?there is no text within the element. All of these are allowable. Remember, because we declared our extension using the <simpleContent> element our type couldn't contain other elements.

## Restriction

Just as we have seen with the extension mechanism, the restriction mechanism allows us to derive new types from existing ones. You can use the restriction mechanism to restrict complexType and simpleType definitions, although, as we saw in the [last chapter](#), it is far more common to restrict simpleType definitions. It should be noted that, although you can use extension to add attributes to a simpleType making it a complexType?the process is not reversible. You cannot use restriction to make a complexType into a simpleType.

### Restricting <complexType> Definitions

When restricting <complexType> definition we start with a base type and create the derivation by removing elements or attributes. Then we need to go a step further; when creating your restricted <complexType> definition you must declare again only the elements you want to keep. By default attributes will automatically be included in the newly restricted type. In this way, restricting <complexType> definitions is far more difficult than extending them.

The rules for restricting complexTypes are very involved. Instead of listing out all of the conditions and exceptions, we will focus on two basic rules: First, you cannot introduce anything new when restricting a complexType. Essentially this means that you cannot add elements or attributes that didn't exist in the base type. You must also be careful, when modifying existing declarations, that the modifications are permitted. Secondly, you cannot remove anything that must appear in the base type. For example, if our base type declares that an element has a minOccurs value of 1 (the default), it cannot be removed in our restriction. This rule was created so that applications that are designed to handle the base type can also handle the restricted type without raising an error.

If it is this difficult to create complexType restrictions, why would anyone do it? The main benefit is the ability to limit a definition without completely changing the type. This allows you to still utilize the type for substitution. It is also used when you are utilizing types that were created by someone else. You can limit the allowable behavior but still create valid documents based on their definitions.

At the time of this writing, support for complexType restriction is not widespread. In fact, many schema validators do

not implement all of the rules defined in the XML Schema Recommendation. Because of this, complexType restrictions are seldom used.

## Restricting <simpleType> Definitions

We learned how to restrict <simpleType> definitions in the [last chapter](#). We restrict <simpleType> definitions by constraining their facets. In the [last chapter](#), we restricted a simpleType that was based on an XML Schema built-in type. Let's look at an example restriction whose base type is a <simpleType> definition we created:

```
<simpleType name="IntList">
  <list itemType="positiveInteger"/>
</simpleType>
<simpleType name="FiveIntList">
  <restriction base="target:IntList">
    <maxLength value="5"/>
  </restriction>
</simpleType>
```

In the above examples, we have declared a simpleType named IntList. Our IntList simpleType allows us to create a whitespace separated list of positiveInteger. In the second declaration we have created a restriction of our IntList type name FiveIntList. Our FiveIntList simpleType applies only one facet, maxLength, which we use to limit the number of items in the list. By setting the value of our maxLength facet to 5 we limit our list to five values.

---

[!\[\]\(4ba7d9059354d2582333ef83ee8ae741\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(e1442b240a0816b50186e5b3943fa7d7\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Substitution

Why should we bother with the complexities of inheritance if we can declare new types instead of deriving our definitions from old ones? With inherited types comes type substitution. XML Schemas allow you to use derived types in place of base types within your XML Schema. Using one type in place of another is called **type substitution**.

In addition to type substitution, XML Schemas allow you to substitute one element for another. **Element substitution** is very similar to type substitution in that it allows you to replace one element with another. For example, suppose you have a `<name>` element and a `<person>` element that have the same type. Element substitution would allow you to use a `<person>` element anywhere you could use a `<name>` element.

## Type Substitution

Type substitution is founded on the concept of type inheritance that we learned in the [last section](#). Inherited types, as we learned, are directly connected to their base types. In fact, the inherited type is a subset of the base type, or vice versa.

Type substitution is controlled within the instance document using the `xstype` attribute. The `xstype` attribute is declared within the namespace <http://www.w3.org/2001/XMLSchema-instance>. By including the `xstype` attribute within your instance document, you can override the type declared within your XML Schema. As the schema validator processes your document, if it encounters the `xstype` attribute, it will validate the element's content against the type specified in the value of the attribute.

*Throughout this section we are introducing and explaining how to use the `xstype` attribute. Actually, the attribute is named type; we are simply using the namespace prefix `xsi`, which we declare within our examples. This indicates that the type attribute comes from the namespace <http://www.w3.org/2001/XMLSchema-instance>. This is the same namespace that contains the declarations for the `schemaLocation` and `noNamespaceSchemaLocation` attributes we saw in the [last chapter](#). Within instance documents, you should always use the prefix `xsi` to refer to items from the <http://www.w3.org/2001/XMLSchema-instance> namespace. Although it is not required, it is the most commonly used prefix.*

The value of the `xstype` attribute is a namespace qualified type (as we have seen within our element declarations). The specified type must be derived from the type declared within the XML Schema. This may seem a little confusing; let's return to our `<name>` example to see how type substitution is accomplished.

## Try It Out-Type Substitution

In this example, we will create a new XML Schema that defines our `<name>` vocabulary. We will also include the declaration for our `ExtendedNameType`. Within the instance document, we will create several `<name>` elements within a root `<names>` element, so that we can see examples of type substitution. At the end of this example, we will break down each step to see how it works.

1.

Let's begin by creating the XML Schema. Simply open a text editor, such as Notepad and copy the following. When you are finished, save the file as `name9.xsd` in the `C:\Wrox` directory you created for the examples in the last two chapters.

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
```

```
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
    <!-- Our NameType from our earlier examples -->
    <complexType name="NameType">
        <sequence>
            <element name="first" type="string"/>
            <element name="middle" type="string"/>
            <element name="last" type="string"/>
        </sequence>
        <attribute name="title" type="string"/>
    </complexType>
    <!-- The new ExtendedNameType that inherits from NameType -->
    <complexType name="ExtendedNameType">
        <complexContent>
            <extension base="target:NameType">
                <sequence>
                    <element name="age" type="positiveInteger"/>
                    <element name="birthdate" type="dateTime"/>
                </sequence>
                <attribute name="gender" type="string"/>
            </extension>
        </complexContent>
    </complexType>
    <!-- We will use the names element as the root in our instance document -->
<element name="names">
    <complexType>
        <sequence>
            <!-- We allow an unbounded number of name elements -->
            <element name="name" type="target:NameType" maxOccurs="unbounded"/>
        </sequence>
    </complexType>
</element>
</schema>
```

## 2.

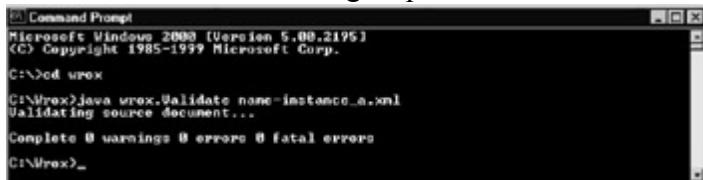
Next, we'll need to create the instance document. This document will be very similar to our name.xml samples from the [last chapter](#). In this document, however, we will have a <names> root element and we will have several <name> elements for content. This will allow us to see some of the different methods of type substitution. Copy the following; when you are finished, save the file as name9.xml.

```
<?xml version="1.0"?>
<names
    xmlns="http://www.wrox.com/name"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.wrox.com/name name9.xsd">
    <name title="Mr.">
        <first>John</first>
        <middle>Fitzgerald</middle>
        <last>Doe</last>
    </name>
    <name title="Mr." gender="male" xsi:type="ExtendedNameType">
        <first>Weird</first>
        <middle>Al</middle>
        <last>Yankovic</last>
        <age>41</age>
        <birthdate>1959-10-23T13:56:00.000-08:00</birthdate>
    </name>
    <name title="Mrs." xsi:type="NameType">
        <first>Jane</first>
        <middle></middle>
        <last>Doe</last>
    </name>
</names>
```

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your name9.xml and name9.xsd documents are located. At the prompt type the following and then press the Enter key:

C:\Wrox> java wrox.Validate name9.xml

You should see the following output:



The screenshot shows a Microsoft Windows 2000 Command Prompt window. The title bar says "Command Prompt". The window content is as follows:  
Microsoft Windows 2000 [Version 5.00.2195]  
(C) Copyright 1985-1999 Microsoft Corp.  
C:\>cd wrox  
C:\wrox>java wrox.Validate name-instance\_a.xml  
Validating source document...  
Complete 0 warnings 0 errors 0 fatal errors  
C:\wrox>

If the output suggests that the validation completed, but that there was an error in the document, correct the error and try again.

## How It Works

In this Try It Out example, we utilized the type substitution mechanism of XML Schemas. Within our instance document we utilized the xsi:type attribute to override the type that our schema processor should used to validate the content of our <name> elements.

```
<name title="Mr.">  
...  
</name>
```

In our first <name> element we didn't include the xsi:type attribute. Therefore, as our schema validator processed the element it used the type that was declared in the XML Schema. Within our XML Schema, we declared that the <name> element would use the type NameType.

```
<name title="Mr." gender="male" xsi:type="ExtendedNameType">  
...  
</name>
```

In our second <name> element we did include the xsi:type attribute. We specified that the schema validator should utilize the type ExtendedNameType when validating the content of our name element. This is called a **type override**. It is important to remember that the value of the xsi:type attribute must be qualified with a namespace. In this case, however, we didn't need to include a namespace prefix because the default namespace declared within our instance document contains the type ExtendedNameType.

As we have already seen, the type specified in our xsi:type attribute must be inherited from the type that is declared in our XML Schema. Let's break this down further. In our XML Schema, we declared our <name> element as follows:

```
<element name="name" type="target:NameType" maxOccurs="unbounded"/>
```

We specified that the type of our element was NameType, which we declared within our XML Schema. We also declared an inherited type, ExtendedNameType.

```
<complexType name="ExtendedNameType">  
<complexContent>  
  <extension base="target:NameType">  
    <sequence>  
      <element name="age" type="positiveInteger"/>  
      <element name="birthdate" type="dateTime"/>  
    </sequence>  
    <attribute name="gender" type="string"/>  
  </extension>  
</complexContent>  
</complexType>
```

Because this type is inherited from our NameType complexType, it is a valid value for an xsi:type attribute. Notice that within our second <name> element we followed the content model declared within our ExtendedNameType.

```
<name title="Mr." gender="male" xsi:type="ExtendedNameType">
<first>Weird</first>
<middle>Al</middle>
<last>Yankovic</last>
<age>42</age>
<birthdate>1959-10-23T13:56:00.000-08:00</birthdate>
</name>
```

We included the <age> and <birthdate> elements because they are required within our ExtendedNameType. When overriding simple and complex types within the instance document, you must follow the content model of the type you specify as the value of your xsi:type attribute.

```
<name title="Mrs." xsi:type="NameType">
<first>Jane</first>
<middle></middle>
<last>Doe</last>
</name>
```

In the final <name> element we included an xsi:type attribute with the value NameType. Of course, this is unnecessary because it is the type declared within our XML Schema. This is intended to show that it is possible to have an xsi:type attribute with the same type that is declared within the XML Schema even though it is redundant.

#### Important

You should be aware that a user could possibly use a type override when they create an XML instance document that uses your XML Schema. It is important that you consider this as you are developing applications, as a user may legally override the type that your document is expecting.

## When to Use Type Substitution

Now that we have explained the "How" of type substitution, we should mention the "Why". As we have already mentioned inheritance allows you to model object oriented programming structures. Type substitution allows you to leverage those structures. Many of the programs that have been written to translate binary objects to XML utilize the type-override mechanism to denote the type of object being described. For example:

```
<object xsi:type="java.awt.Applet">
  <!-- additional object elements or properties of the Applet -->
</object>
```

In the above, we have an <object> element whose type is overridden. In the base XML Schema, <object> might be the only element declared. If all of the types within the XML Schema derive from a single base, then the Java hierarchy could easily be duplicated. This simplifies what would otherwise be extremely complex because the user or application is only concerned with a single element. All other information is stored only within the complexType declarations.

## Element Substitution

The XML Schema Recommendation also introduces a model for replacing one element with another. Like type substitution, element substitution is also dependent upon the type inheritance features of XML Schemas. However, element substitution is declared and controlled within the XML Schema, unlike type substitution, which uses override declarations in the instance document. This allows you to provide options to the end-user of your XML Schema without them needing to understand the details of type overrides.

In order to substitute one element for another we must use global element declarations. In addition to using global element declarations, we must also declare that the interchangeable elements are part of a substitution group. We can do this by including the substitutionGroup attribute within our <element> declarations. Each substitution group has a primary element declaration known as the **head element**. Within the XML Schema, anywhere that the head element is referenced, another element within the substitution group can be used instead.

In order to get a clear picture of how the element substitution mechanism works, let's reconfigure our last example to utilize element substitution.

## Try It Out - Element Substitution

In this example, we will create an XML Schema that defines our <name> vocabulary. We will base this on our last examples. Within the instance document, we will substitute several elements in place of our <name> element. At the end of this example, we will break down each step to see how it works.

1.

Let's begin by modifying the XML Schema. Open the file name9.xsd and make the following changes. When you are finished, save the file as name10.xsd .

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.wrox.com/name" xmlns:target="http://www.wrox.com/name"
elementFormDefault="qualified">
    <!-- Global elements used for element substitution -->
    <element name="name" type="target:NameType"/>
    <element name="person"
        type="target:NameType"
        substitutionGroup="target:name"/>
    <element name="ageName"
        type="target:ExtendedNameType"
        substitutionGroup="target:name"/>
    <!-- Our NameType from our earlier examples -->
    <complexType name="NameType">
        <sequence>
            <element name="first" type="string"/>
            <element name="middle" type="string"/>
            <element name="last" type="string"/>
        </sequence>
        <attribute name="title" type="string"/>
    </complexType>
    <!-- The new ExtendedNameType that inherits from NameType -->
    <complexType name="ExtendedNameType">
        <complexContent>
            <extension base="target:NameType">
                <sequence>
                    <element name="age" type="positiveInteger"/>
                    <element name="birthdate" type="dateTime"/>
                </sequence>
                <attribute name="gender" type="string"/>
            </extension>
        </complexContent>
    </complexType>
    <!-- We will use the names element as the root in our instance document -->
    <element name="names">
        <complexType>
            <sequence>
                <!-- We declare our name element using a reference to a global declaration -->
                <element ref="target:name" maxOccurs="unbounded"/>
            </sequence>
        </complexType>
    </element>
</schema>
```

## 2.

Next, we'll need to modify our instance document. In this document, instead of using the xsitype attribute for our substitution mechanism, we will substitute different elements. This will allow us to see some of the different kinds of element substitution. Open the file name9.xml and make the following modifications. When you are finished, save the file as name10.xml.

```
<?xml version="1.0"?>
<names
  xmlns="http://www.wrox.com/name"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wrox.com/name name10.xsd">
  <name title="Mr.">
    <first>John</first>
    <middle>Fitzgerald</middle>
    <last>Doe</last>
  </name>
  <ageName title="Mr." gender="male">
    <first>Weird</first>
    <middle>Al</middle>
    <last>Yankovic</last>
    <age>41</age>
    <birthdate>1959-10-23T13:56:00.000-08:00</birthdate>
  </ageName>
  <person title="Mrs.">
    <first>Jane</first>
    <middle></middle>
    <last>Doe</last>
  </person>
</names>
```

## 3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your name10.xml and name10.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate name10.xml
```

It should validate with no errors.

## How It Works

In this Try It Out example, we utilized the element substitution mechanism of XML Schemas. Unlike the first Try It Out in this chapter, our <names> element within our instance document contains several different elements. These various elements are substituted for the <name> element declared within our XML Schema.

```
<element name="name" type="target:NameType"/>
```

We inserted several global <element> declarations before our type declarations. Remember, that the element substitution mechanism only works with global <element> declarations. Our first declaration defines the <name> element. This declaration appears as it has in many of our examples thus far. We will use this global declaration as our head element.

```
<element name="person"
  type="target:NameType"
  substitutionGroup="target:name"/>
<element name="ageName"
  type="target:ExtendedNameType"
  substitutionGroup="target:name"/>
```

Following the declaration for our <name> element we have created global declarations for the <person> and <ageName> elements. Within each of these declarations, we have included the substitutionGroup attribute. We

specified our global <name> element as the value, indicating that it is the head element for this group of element substitutions.

It is important to notice that we can use different types for our substitutable elements. The <person> element declaration has the type NameType. The <ageName> element is type ExtendedNameType. We have used these types to demonstrate allowable substitutions. Although we don't use a restricted type here, you may also use restricted types when using element substitution.

#### Important

An element may only be part of a substitution group if it has the same type or a type inherited from the same type as the head element of the substitution group.

Finally, we declared the content of our <names> root element. Instead of listing all of the different element possibilities within our content model, we simply included a reference to our global <name> element. Remember, the global <name> element was used as the head of our substitution group. This means, within our instance document, we can utilize any of the elements from our substitution group in place of the <name> element.

```
<element ref="target:name" maxOccurs="unbounded"/>
```

Within our instance document, we use one of each element. Of course, when designing your XML Schemas, you can use substitutions as often as you like.

## When to Use Element Substitution

As you may have already noticed, the net result of the element substitution and type substitution mechanisms is very similar. In the Type Substitution section, we learned that many programs that model object oriented programming structures utilize type substitution. Obviously, the element substitution mechanism could be used in much the same way. More often, however, element substitution is used in circumstances when the reader needs a choice between two (or more) elements, as we have done above.

For example, suppose you wanted to create very descriptive element names for your documents:

```
<ThePrimaryApplicantHomeAddress>
```

This is a very descriptive element name! It makes it very easy to understand what the element is at a glance-- but imagine if you had to type this hundreds of times a day--you might not be so impressed. Instead of eliminating the element altogether you could keep it and create an abbreviated element.

```
<PAHA>
```

This isn't descriptive at all--but it is definitely shorter. We have simply taken the first letter of each word to create the abbreviated name (you can use any method you like). Within our XML Schema we would declare the <PAHA> element as follows:

```
<element name="PAHA"
        type="target: ThePrimaryApplicantHomeAddressType"
        substitutionGroup="target: ThePrimaryApplicantHomeAddress"/>
```

This way either element can be used throughout our document. The user can choose their preferred method. Providing abbreviated element names is common in Internet and Palm XML applications. This ultimately makes your XML Schema more user-friendly.

# Creating a Schema from Multiple Documents

To keep things simple, when declaring our XML Schemas, we have used a single schema document. The XML Schema Recommendation introduces mechanisms for combining XML Schemas and reusing definitions. As we saw in [Chapter 5](#), reusing existing definitions is good practice - it saves us time creating the documents and increases our document's interoperability.

*Several XML Schema authors have already begun developing libraries of commonly used types that can be reused in your XML Schemas. The W3C type library can be found at <http://www.w3.org/2001/03/XMLSchema/TypeLibrary.xsd>.*

The XML Schema Recommendation provides three declarations for use with multiple XML Schema Documents:

- `<import>`
- `<include>`
- `<redefine>`

Let's look at each of these features in more detail:

## <import>

The `<import>` declaration, as the name implies, allows us to import global declarations from other XML Schemas. The `<import>` declaration is used primarily for combining XML Schemas that have different targetNamespaces. By importing the declarations, the two XML Schemas can be used in conjunction within an instance document. It is important to note that the `<import>` declaration only allows us to *refer* to declarations within other XML Schemas. In the [next section](#) we will learn about the `<include>` declaration which *includes* the declarations as if they had been declared. The `<include>` declaration can only be used for XML Schemas with the same targetNamespace.

```
<import
  namespace="URI"
  schemaLocation="URI">
```

The `<import>` declaration is always declared globally within an XML Schema (it must be a direct child of the `<schema>` element). This means that the `<import>` declaration applies for the entire XML Schema. When importing declarations from other namespaces the schema validator will attempt to look up the namespace declaration based on the `schemaLocation` attribute specified within the corresponding `<import>` declaration. Of course, as we saw in the [last chapter](#), the `schemaLocation` attribute serves only as a hint to the processor. The processor may elect to use another copy of the XML Schema. If the schema validator cannot locate the XML Schema, for any reason, it may raise an error or proceed with lax validation.

In order to get a better idea of how this works, we need a sample XML Schema that uses the `<import>` declaration. Let's combine the examples that we have been working with for the past two chapters. Within the XML Schema for our music catalog, we will import the declarations from our `<name>` vocabulary. We will use the imported `<name>` declarations to better define our musical artists.

# Try It Out?Importing XML Schema Declarations

In this example we will take the basics of the catalog XML Schema we used in the [last chapter](#) and introduce an <import> declaration. We will import the name vocabulary that we have been developing. We will need to modify our instance document to reflect the changes in our XML Schemas.

1.

Let's begin by modifying our catalog vocabulary. We will need to import the name vocabulary and use the imported types. Open the file all1.xsd and make the following changes. When you are finished, save the file as all2.xsd.

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  xmlns:name="http://www.wrox.com/name"
  elementFormDefault="qualified">

  <!-- Notice that we use a full file URL here -->
  <import namespace="http://www.wrox.com/name"
  schemaLocation="file:///c:/wrox/name10.xsd" />

  <attributeGroup name="NameAttGroup">
    <attribute name="name" type="string" use="required"/>
  </attributeGroup>

  <element name="catalog">
    <complexType>
      <choice maxOccurs="unbounded">
        <element name="CD">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="cassette">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="record">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="MP3">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
      </choice>
      <attribute name="version" type="float" fixed="1.0"/>
      <attributeGroup ref="target:NameAttGroup"/>
    </complexType>
  </element>

  <group name="AlbumGroup">
    <sequence>
      <!-- We use the imported NameType for the artist element -->
      <element name="artist" type="name:NameType"/>
      <element name="title" type="normalizedString"/>
      <element name="genre" type="token"/>
      <element name="date-released" type="gYearMonth"/>
      <element name="song" type="target:SongType" maxOccurs="unbounded"/>
    </sequence>
  </group>
</schema>
```

```
</sequence>
</group>

<complexType name="SongType">
  <sequence>
    <element name="title" type="normalizedString"/>
    <element name="length" type="target:LengthType"/>
    <element name="parody" type="target:ParodyType"/>
    <element name="lyrics" type="string"/>
  </sequence>
</complexType>

<complexType name="LengthType">
  <sequence>
    <element name="minutes" type="nonNegativeInteger"/>
    <element name="seconds">
      <simpleType>
        <restriction base="nonNegativeInteger">
          <maxInclusive value="59"/>
        </restriction>
      </simpleType>
    </element>
  </sequence>
</complexType>

<complexType name="ParodyType">
  <sequence minOccurs="0">
    <element name="title" type="normalizedString"/>
    <!-- We use the imported NameType for the artist element -->
    <element name="artist" type="name:NameType"/>
  </sequence>
</complexType>

</schema>
```

## 2.

Now that we have modified our XML Schema document, let's create an instance document that reflects the changes. This document will appear to be very similar to our all1.xml from the [last chapter](#). Only the <artist> elements will change. Open the file all1.xml and make the following changes. When you are finished, save the file as all2.xml.

```
<?xml version="1.0"?>
<catalog
  xmlns="http://www.wrox.com/catalog"
  xmlns:n="http://www.wrox.com/name"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.wrox.com/catalog all2.xsd"
  version="1.0" name="My Music Library">
  <CD>
    <artist title="Mr.">
      <n:first>Alfred</n:first>
      <n:middle>Matthew</n:middle>
      <n:last>Yankovic</n:last>
    </artist>
    <title>Dare to be Stupid</title>
    <genre>parody</genre>
    <date-released>1985-06</date-released>
    <song>
      <title>Like A Surgeon</title>
      <length>
        <minutes>3</minutes>
        <seconds>33</seconds>
      </length>
      <parody>
        <title>Like A Virgin</title>
```

```
<artist title="Mrs.">
  <n:first>Madonna</n:first>
  <n:middle></n:middle>
  <n:last></n:last>
</artist>
</parody>
<lyrics></lyrics>
</song>
<song>
  <title>Dare to be Stupid</title>
  <length>
    <minutes>3</minutes>
    <seconds>25</seconds>
  </length>
  <parody></parody>
  <lyrics></lyrics>
</song>
</CD>
</catalog>
```

3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your all2.xml and all2.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate all2.xml
```

It should validate with no errors.

## How It Works

In this Try It Out, we imported one XML Schema into another. We used the `<import>` declaration because the two XML Schemas were designed for different targetNamespaces. Within our first XML Schema, we declared a single global complexType that could be used to describe names. In our second XML Schema, we were forced to do some more work:

```
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  xmlns:n="http://www.wrox.com/name"
  elementFormDefault="qualified">
```

The first addition that we had to make was an XML namespace declaration in the root element. We added a namespace declaration for the namespace <http://www.wrox.com/name>. We needed to add this declaration so that we could refer to items declared within the namespace later in our XML Schema.

Secondly, we added an `<import>` declaration.

```
<import namespace="http://www.wrox.com/name"
  schemaLocation="file:///c:/wrox/name10.xsd" />
```

This `<import>` declaration is straightforward. We are importing the declarations from the <http://www.wrox.com/name> namespace, which is located in the file c:\wrox\name10.xsd. This declaration allows us to reuse the declarations from our name10.xsd XML Schema within our all2.xsd XML Schema. If you are using another schema validator, you should check the documentation for special rules when referring to external files. For example, the Xerces parser (which our wrox.Validate program is based on) handles relative URL references differently in different versions. This is why we have used the fully qualified path in our schemaLocation attribute.

Finally, we modified the type attributes within our artist `<element>` declarations.

```
<element name="artist" type="name:NameType"/>
```

Notice that we used the namespace prefix we declared within the root element when referring to the NameType declaration from our name10.xsd file.

Once we made these changes, we had to create a new, compliant instance document. The major difference (apart from the schemaLocation) was the modified content of our <artist> elements.

```
<artist title="Mr .">
  <name:first>Alfred</name:first>
  <name:middle>Matthew</name:middle>
  <name:last>Yankovic</name:last>
</artist>
```

This might seem a little more confusing than you would expect. Because we declared that the elementFormDefault of both XML Schemas was qualified, we are required to qualify all of our elements with namespace prefixes (or a default namespace declaration). Therefore, we are required to qualify all of the elements with their namespace.

Although the <artist> element is declared with a type from our name10.xsd file, the <element> declaration is located within the all2.xsd XML Schema. For this reason, the <artist> element is a member of the <http://www.wrox.com/catalog> namespace. The title attribute doesn't need to be qualified because we didn't modify the attributeFormDefault within our XML Schemas. The <first>, <middle>, and <last> elements are all declared within the <http://www.wrox.com/name> namespace; therefore we must qualify them with the name prefix we declared in the root element of our instance document.

## <include>

The <include> declaration functions very similarly to the <import> declaration. Unlike the <import> declaration, however, the <include> declaration allows us to combine XML Schemas that are designed for the same targetNamespace (or no targetNamespace) much more effectively. When a schema validator encounters an <include> declaration it treats the global declarations from the secondary XML Schema as if they had been declared in the XML Schema that contains the <include> declaration. This subtle difference makes quite a difference when you are using many modules to define your targetNamespace.

```
<include
  schemaLocation="URI">
```

Notice, within the <include> declaration, there is no namespace attribute. Unlike the <import> declaration, the <include> declaration can be used only on documents with the same targetNamespace, or no targetNamespace. Because of this, a namespace attribute would be redundant. Again, the schemaLocation attribute allows you to specify the location of the XML Schema you are including. Just as we saw with the <import> declaration, the schemaLocation functions as a validator hint. If the schema validator cannot locate a copy of the XML Schema, for any reason, it may raise an error or proceed with lax validation.

In order to demonstrate the <include> declaration we are going to need an example that utilizes two XML Schema documents with the same targetNamespace. In order to do this we will break our music catalog into two parts?the catalog declaration and the song declaration.

## Try It Out?Including XML Schema Declarations

In order to get a clear picture of how to use the include mechanism, we will divide our all2.xsd XML Schema into two parts, and include one in the other. This is known as dividing an XML Schema into modules?separate files that make up the overall XML Schema.

1.

We will begin by creating a new XML Schema that declares the type SongType. In order to create the

declarations, you can simply copy and paste the declarations from all2.xsd. Once you have created the file, save it as song.xsd in the C:\Wrox folder.

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  xmlns:name="http://www.wrox.com/name"
  elementFormDefault="qualified">

  <!-- We will need to import the name vocabulary in this XML Schema as well -->
  <import namespace="http://www.wrox.com/name"
    schemaLocation="file:///c:/wrox/name10.xsd" />

  <complexType name="SongType">
    <sequence>
      <element name="title" type="normalizedString"/>
      <element name="length" type="target:LengthType"/>
      <element name="parody" type="target:ParodyType"/>
      <element name="lyrics" type="string"/>
    </sequence>
  </complexType>

  <complexType name="LengthType">
    <sequence>
      <element name="minutes" type="nonNegativeInteger"/>
      <element name="seconds">
        <simpleType>
          <restriction base="nonNegativeInteger">
            <maxInclusive value="59"/>
          </restriction>
        </simpleType>
      </element>
    </sequence>
  </complexType>

  <complexType name="ParodyType">
    <sequence minOccurs="0">
      <element name="title" type="normalizedString"/>
      <!-- We use the imported NameType for the artist element -->
      <element name="artist" type="name:NameType"/>
    </sequence>
  </complexType>
</schema>
```

## 2.

Now that we have created our song.xsd XML Schema, we need to make a couple of modifications to the all2.xsd XML Schema. We will begin by inserting an <include> declaration. Secondly, we will remove the SongType declaration and its associated types. Open the file all2.xsd, make the following modifications and save the file as all3.xsd.

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  xmlns:name="http://www.wrox.com/name"
  elementFormDefault="qualified">

  <!-- Notice that we use a full file URL here -->
  <import namespace="http://www.wrox.com/name" />
```

```
    schemaLocation="file:///c:/wrox/name10.xsd" />

<!-- Notice that we use a full file URL here -->
<include schemaLocation="file:///c:/wrox/song.xsd" />

<attributeGroup name="NameAttGroup">
    <attribute name="name" type="string" use="required"/>
</attributeGroup>

<element name="catalog">
    <complexType>
        <choice maxOccurs="unbounded">
            <element name="CD">
                <complexType>
                    <group ref="target:AlbumGroup"/>
                </complexType>
            </element>
            <element name="cassette">
                <complexType>
                    <group ref="target:AlbumGroup"/>
                </complexType>
            </element>
            <element name="record">
                <complexType>
                    <group ref="target:AlbumGroup"/>
                </complexType>
            </element>
            <element name="MP3">
                <complexType>
                    <group ref="target:AlbumGroup"/>
                </complexType>
            </element>
        </choice>
        <attribute name="version" type="float" fixed="1.0"/>
        <attributeGroup ref="target:NameAttGroup"/>
    </complexType>
</element>

<group name="AlbumGroup">
    <sequence>
        <!-- We use the imported NameType for the artist element -->
        <element name="artist" type="name:NameType"/>
        <element name="title" type="normalizedString"/>
        <element name="genre" type="token"/>
        <element name="date-released" type="gYearMonth"/>
        <!-- Notice that we still use the target: prefix to refer to SongType -->
        <element name="song" type="target:SongType" maxOccurs="unbounded"/>
    </sequence>
</group>

<!-- SongType, LengthType, and ParodyType have been removed -->

</schema>
```

### 3.

Before we can schema validate our document, we must modify it so that it refers to our new XML Schema. Open the file all2.xml and change the xsi:schemaLocation attribute, as follows. When you are finished, save the file as all3.xml.

```
xsi:schemaLocation="http://www.wrox.com/catalog all3.xsd"
```

### 4.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your all3.xml and all3.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate all3.xml
```

1.

It should validate with no errors.

## How It Works

Dividing complex XML Schemas into modules can be an excellent design technique. In this Try It Out, we divided our catalog vocabulary into two modules. We declared these modules in separate XML Schema documents each with <http://www.wrox.com/catalog> as the targetNamespace. Because the two documents utilized the same targetNamespace we simply used an <include> declaration to combine them.

```
<include schemaLocation="file:///c:/wrox/song.xsd" />
```

As the schema validator processes all3.xsd it includes the declarations with all3.xsd with the declarations for all3.xsd as if they had been declared in one document. Because of this, we were able to use the SongType <complexType> definition as if it were declared within catalog.xsd.

```
<element name="song" type="target:SongType" maxOccurs="unbounded"/>
```

Because we didn't introduce any namespace complexities, the instance document didn't need to change to support the new modular design.

*What happens when the XML Schema you are including has no targetNamespace? Declarations within XML Schemas that have no targetNamespace are treated differently. These declarations are known as **Chameleon Components**. Chameleon Components take on the targetNamespace of the XML Schema that includes them.*

## <redefine>

Often XML Schema developers need to include a declaration from another XML Schema, but need the declaration to be modified in some way at the same time. Instead of including the declaration and then modifying it in a separate declaration the XML Schema Recommendation allows us to do this in one step using the <redefine> declaration.

```
<redefine  
schemaLocation="URI">
```

The <redefine> declaration enables us to redefine the content of groups, complexTypes and simpleTypes. When redefining complexTypes and simpleTypes, simply include the new type definition within the <redefine> element. Creating the redefined types is very similar to creating inherited types. The only difference is that you must specify the type you are redefining as the base of the derivation. Instead of creating a new type, however, the schema validator will simply use the redefinition in place of the old type.

Although we are allowed to redefine many declarations within a single XML Schema, we cannot redefine a single declaration more than once. We'll reconfigure our last Try It Out example to demonstrate how to use the <redefine> declaration.

## Try It Out?Redefining XML Schema Declarations

In our last example, we included and used the SongType within our all3.xsd XML Schema. Suppose that we wanted to modify the SongType as it was included. Let's redefine the SongType declaration, by adding an attribute named hasVideo. We will use this attribute within our instance document to indicate whether the particular song had a

music video associated with it.

1.

Let's begin by modifying our all3.xsd XML Schema. First, let's remove the <include> declaration we added in the last Try It Out. Secondly, let's create a <redefine> declaration for our SongType. Open the file all3.xsd, make the following modifications and save the file as all4.xsd.

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.wrox.com/catalog"
  xmlns:target="http://www.wrox.com/catalog"
  xmlns:name="http://www.wrox.com/name"
  elementFormDefault="qualified">

  <!-- Notice that we use a full file URL here -->
  <import namespace="http://www.wrox.com/name"
  schemaLocation="file:///c:/wrox/name10.xsd" />

  <!-- The include declaration has been removed -->

  <!-- We will redefine our SongType and add an attribute -->
  <redefine schemaLocation="file:///c:/wrox/song.xsd">
    <complexType name="SongType">
      <complexContent>
        <extension base="target:SongType">
          <attribute name="hasVideo" type="boolean"/>
        </extension>
      </complexContent>
    </complexType>
  </redefine>

  <attributeGroup name="NameAttGroup">
    <attribute name="name" type="string" use="required"/>
  </attributeGroup>

  <element name="catalog">
    <complexType>
      <choice maxOccurs="unbounded">
        <element name="CD">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="cassette">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="record">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
        <element name="MP3">
          <complexType>
            <group ref="target:AlbumGroup"/>
          </complexType>
        </element>
      </choice>
      <attribute name="version" type="float" fixed="1.0"/>
      <attributeGroup ref="target:NameAttGroup"/>
    </complexType>
  </element>
```

```
<group name="AlbumGroup">
<sequence>
    <!-- We use the imported NameType for the artist element -->
    <element name="artist" type="name:NameType"/>
    <element name="title" type="normalizedString"/>
    <element name="genre" type="token"/>
    <element name="date-released" type="gYearMonth"/>
    <!-- Notice that we still use the target: prefix to refer to SongType -->
    <element name="song" type="target:SongType" maxOccurs="unbounded"/>
</sequence>
</group>

<!-- SongType, LengthType, and ParodyType have been removed -->

</schema>
```

## 2.

Next, we'll need to modify the instance document to make use of our newly added hasVideo attribute. We will also need to modify the xsi:schemaLocation attribute to refer to our new XML Schema. Open the file al13.xml, make the following modifications and save the file as al14.xml.

```
<?xml version="1.0"?>
<catalog
    xmlns="http://www.wrox.com/catalog"
    xmlns:n="http://www.wrox.com/name"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.wrox.com/catalog al14.xsd"
    version="1.0" name="My Music Library">
    <CD>
        <artist title="Mr.">
            <n:first>Alfred</n:first>
            <n:middle>Matthew</n:middle>
            <n:last>Yankovic</n:last>
        </artist>
        <title>Dare to be Stupid</title>
        <genre>parody</genre>
        <date-released>1985-06</date-released>
        <song hasVideo="true">
            <title>Like A Surgeon</title>
            <length>
                <minutes>3</minutes>
                <seconds>33</seconds>
            </length>
            <parody>
                <title>Like A Virgin</title>
                <artist title="Mrs.">
                    <n:first>Madonna</n:first>
                    <n:middle></n:middle>
                    <n:last></n:last>
                </artist>
            </parody>
            <lyrics></lyrics>
        </song>
        <song hasVideo="true">
            <title>Dare to be Stupid</title>
            <length>
                <minutes>3</minutes>
                <seconds>25</seconds>
            </length>
            <parody></parody>
            <lyrics></lyrics>
        </song>
    </CD>
</catalog>
```

3.

We are ready to begin validating our XML instance document against our XML Schema. Open a command prompt and change directories to the folder where your all4.xml and all4.xsd documents are located. At the prompt type the following and then press the Enter key:

```
C:\Wrox> java wrox.Validate all4.xml
```

It should validate fine with no errors.

## How It Works

In this Try It Out, we redefined our SongType declaration, as it was included within our all4.xsd XML Schema. This allowed us to add the attribute hasVideo to each of our <song> elements within our instance document. The real work within this example was the <redefine> declaration:

```
<redefine schemaLocation="file:///c:/wrox/song.xsd">
<complexType name="SongType">
<complexContent>
<extension base="target:SongType">
<attribute name="hasVideo" type="boolean"/>
</extension>
</complexContent>
</complexType>
</redefine>
```

The <redefine> declaration, works very similar to the <include> declaration. Again notice that there is no namespace attribute; all of our redefined declarations must come from XML Schemas with the same targetNamespace (or no namespace). Just as we did with our <include> declaration we specified the file song.xsd within our schemaLocation attribute.

Within the <redefine> declaration, we have declared a <complexType>. Notice that the <complexType> definition is named using the same name as the type we are redefining. As we have already learned, the <redefine> element allows you to create inherited types. Therefore, we continue the <complexType> definition as an extended <complexType> derived from the SongType declaration from our song.xsd XML Schema.

Within the extension, we added a single declaration for the hasVideo attribute. We declared that the attribute would be of type boolean. Remember from the [last chapter](#), the boolean type allows us to specify values of true or false.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Notations

In [Chapter 5](#) we learned about notations and how to declare them in DTDs. The XML Schema Recommendation also provides the ability to declare notations. Remember, notations are used to define external file types. For example, a notation can be used whenever we need to refer to an external bitmap file. Because our XML processor cannot parse bitmap files, we will need to use an external program for displaying or editing them. The external program could be any program you like, even web applications. When the parser encounters a usage of the notation name, it will simply provide the path to the application.

```
<notation
  name="name of the notation"
  public="Public Identifier"
  system="System Identifier">
```

Within XML Schemas, notations must be declared globally (the `<notation>` element must be a direct child of the `<schema>` element). Just as we saw in our DTD NOTATION declarations, our XML Schema `<notation>` declaration has a global name. In addition to the name the `<notation>` provides public and system attributes that can be used to specify the public identifier and system identifier respectively. Just as we saw in the DTD chapter, the public identifier is optional.

In [Chapter 5](#), we learned that the public identifier follows a specific public identifier format called Formal Public Identifiers. Within XML Schemas you can continue to use Formal Public Identifiers, however, it is also common to utilize MIME types within the public attribute. System identifiers function identically to their DTD counterparts, both must be valid URIs.

*MIME types, or Multipart Internet Mail Extension types, are used to identify file types on the World Wide Web. Some common MIME types for XML include text/xml and application/xml.*

In [Chapter 5](#), we declared a NOTATION for bitmap images:

```
<!NOTATION bmp SYSTEM "file:///c:/windows/paint.exe">
```

Using an XML Schema, we could declare the same notation as follows:

```
<notation name="bmp" system="file:///c:/windows/paint.exe">
```

In each of these examples, we have created system references to specific programs in specific folders. These programs may or may not exist on our system. Often internet or Intranet application paths are used instead, as you can better control the existence of the application.

Specifying an application path can be dangerous. Suppose someone creates an XML Schema that references the Windows program command.com. A malicious developer could utilize this as a back door to execute commands on a remote system. You should exercise caution when utilizing notations and file paths within your XML Schemas.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Alternative Schema Formats

Throughout the past three chapters, we have focused on DTDs and XML Schemas. As we have already mentioned, these are not the only formats that are used for describing a document's vocabulary. They are, however, the most widely used as they are the only recommendations created and published by the World Wide Web Consortium (W3C). Some of the most important alternative formats are:

- XDR
- Schematron
- Examplotron
- RELAX NG

Let's look at each of these in more detail.

## XDR

XDR, or XML-Data Reduced, is a format that is widely used by Microsoft. XDR is a simplified version of XML-Data. XDR Schemas can be used to describe and validate an XML document against a formal definition. XDR Schemas, like W3C XML Schemas, are written in an XML syntax and provide many features similar to W3C XML Schemas. Originally, XDR Schemas were used in many Microsoft products, such as MSXML 3.0, SQL 2000, and BizTalk, but while XDR is still supported, recently W3C XML Schemas have become the preferred format for describing vocabularies. More information about XDR can be found at

<http://www.w3.org/TR/1998/NOTE-XML-data-0105/> and  
<http://msdn.microsoft.com/library/en-us/xmlsdk30/htm/xmconxmlschemadevelopersguide.asp>

## Schematron

Schematron is a schema language written by Rick Jelliffe. Instead of listing out all of the allowable elements and attributes in your vocabulary you create assertions or rules for your XML Documents. For example, suppose that you wanted to state that the `<middle>` element could only appear if both the `<first>` and `<last>` elements were in the document. You could create a rule for this. Because you do not develop a vocabulary with Schematron you must design your schema with a very different mindset. At the same time, however, you can create many complex rules that are not available in any other language. For this reason, many developers use Schematron in conjunction with W3C XML Schemas as they complement one another extremely well. You can find more information about Schematron at its homepage <http://www.ascc.net/xml/resource/schematron/> or the SourceForge project <http://sourceforge.net/projects/schematron>.

## Examplotron

Examplotron was created by Eric van der Vlist primarily to prove the concept of using an example document instead of a schema. In his effort to prove his theory, he created an extremely simple, yet effective, language. Examplotron takes the design strategy of creating your schema based on an existing instance document to the next level. By simply

inserting some key attributes, you can create schemas for many documents. Unfortunately, Examplotron has many limitations and cannot model some complex content models at all. For more information about Examplotron, go to <http://examplotron.org/>.

## RELAX NG

RELAX NG is the result of merging two popular schema languages: RELAX and TREX. The RELAX NG language is very similar to the W3C XML Schema language. RELAX NG has a much stronger foundation in mathematical models, which allows programmers to create highly optimized validation tools. In addition, RELAX NG omits many features that make W3C XML Schemas difficult to learn. RELAX NG, like W3C XML Schemas, is written in an XML Syntax and requires you to define the allowable elements and attributes within your instance documents. RELAX NG can be found on the Web at <http://relaxng.org>.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

At this point, you should begin to recognize the power of XML Schemas, but you should also be aware of the complexity of XML Schemas. Even though we have covered the bulk of XML Schemas there are still many pitfalls and complexities we have not seen.

In this chapter, we have learned some of the advanced features of XML Schemas including:

- The extension and restriction mechanisms for complex and simple types
- How to substitute inherited types and use element substitution
- The import, include, and redefine methods for combining XML Schemas
- How to declare notations
- Alternative schema languages

This chapter has concentrated on an advanced subset of XML Schemas. Using the information in this chapter, you can now develop extremely advanced XML Schemas. As you continue through your XML career, you may need to learn some of the obscure features and rules that we have not covered. *Professional XML Schemas*, which we mentioned earlier, is a thorough reference for just this purpose. As we move through the rest of the book, try to figure out what the XML Schema definitions would look like for the examples you encounter.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 8: The Document Object Model (DOM)

## Overview

Now that we've got our information in XML format, we want to be able to do things with that information in the applications we write. We need not only to access it, but to change it and add to it, and even create brand new documents from scratch. The **Document Object Model (DOM)** provides a means for working with XML documents (and other types of document) in code, and a way to interface with that code in the programs we write.

For example, the DOM enables us to create documents and parts of documents, navigate through the document, move, copy, and remove parts of the document, and add or modify attributes. In this chapter you will learn how to work with the DOM to achieve such tasks, as well as seeing:

- What the DOM is, and why it was created
- What interfaces are, and how they differ from objects
- What XML-related interfaces exist in the DOM, and what they can do
- What exceptions are

The DOM specification is being built level by level, with each new level providing new functionality. That is, when the W3C produced the first DOM Recommendation, it was **DOM Level 1**. Level 1 was then added to, producing **Level 2**. At the time of writing, DOM Level 2 was the currently released Recommendation, and DOM Level 3 was in working draft form, so in this chapter we'll be discussing the DOM Level 2.

You can find the DOM Level 2 specification at <http://www.w3.org/TR/DOM-Level-2/>.

*Since the DOM is really for programmers, some programming experience is assumed for this chapter; **Object Oriented Programming** experience would be helpful but isn't essential.*

The DOM specification is language and platform-agnostic, meaning that a DOM implementation could be created in any programming language, on any platform. Since we've already been using Internet Explorer and MSXML, the examples in this chapter will use MSXML's DOM implementation.

# What is the DOM?

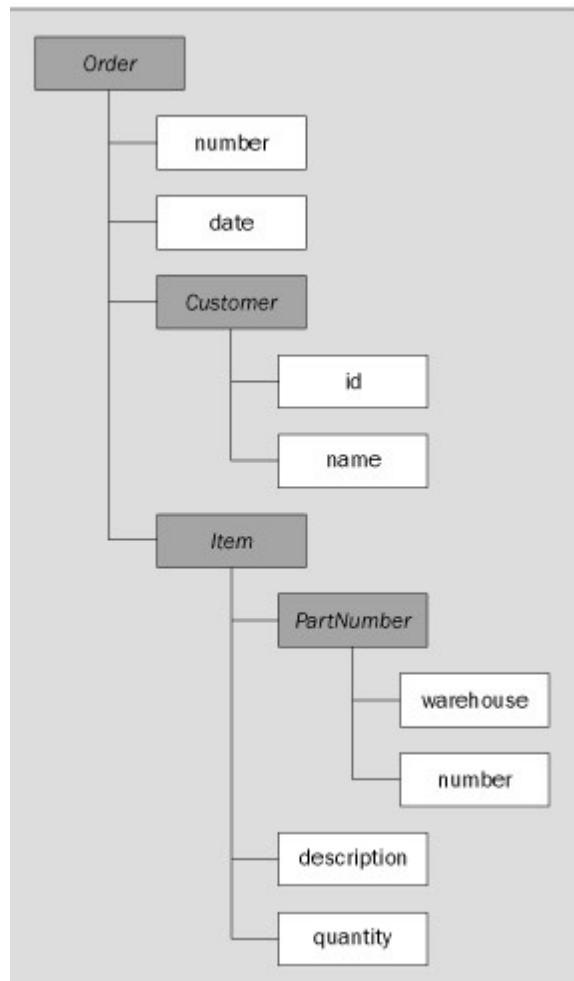
In [Chapter 1](#) we looked at the concept of an object model, and how using one can make working with information easier. We also noted the fact that an XML document is structured very much like an object model: it is hierarchical, with nodes potentially having other nodes as children.

## XML As an Object Model

For example, we could take our <order> XML, which looked like this:

```
<?xml version="1.0"?>
<order number="312597">
  <date>2000/1/1</date>
  <customer id="216A">Company A</customer>
  <item>
    <part-number warehouse="Warehouse 11">E16-25A</part-number>
    <description>Production-Class Widget</description>
    <quantity>16</quantity>
  </item>
</order>
```

and structure it as an object model, such as the following:



Some of the elements have been made into *objects*, as shown by the shaded boxes, and some (including attributes) have been made into *properties* of those objects, as shown by the white boxes. If we were writing code to deal with an order, this object model would make it easier to process that information, and would probably even include

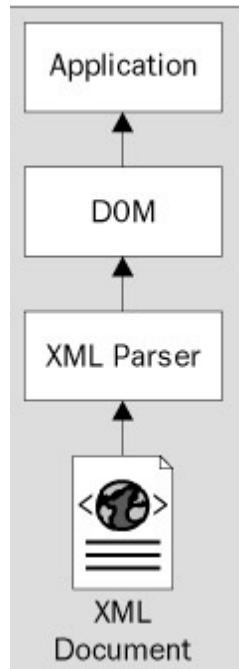
*methods* to provide some functionality for us. (Of course, if you were to create an object model to encapsulate this information, you might design a totally different one!)

For example, we might provide a method on the object model called `fulfill()`, which would fulfill the order, or `saveToDatabase()`, which could take all of the information and save it to our database.

## The XML DOM

You may have noticed that the object model above would only work for this specific document type. To work with other document types, we would have to create a new object model for each one. While these proprietary object models will be useful in many situations, there are also situations where a more generic approach is needed. That is, we need an object model that can model *any* XML document, regardless of how it is structured. The **Document Object Model (DOM)** takes this generic approach.

The DOM is usually added as a layer between the XML parser and the application that needs the information in the document, meaning that the parser reads the data from the XML document and then feeds that data into a DOM. The DOM is then used by a higher-level application. The application can do whatever it wants with this information, including putting it into another proprietary object model, if so desired.



So, in order to write an application that will be accessing an XML document through the DOM, you need to have an XML parser and a DOM implementation installed on your machine. (There is an alternative, SAX, which we'll look at in the following chapter, but for now we'll concentrate on using the DOM.) Some DOM implementations, such as the one that ships with Internet Explorer, have the parser built right in, while others can be configured to sit on top of one of many parsers.

We'll be using some of IE5's built-in XML capabilities for the Try It Outs in this chapter; IE 5 or later ships with a library called **MSXML**, which includes a DOM implementation (Level 1 plus some extensions). Check out <http://www.xmlsoftware.com/parsers/> for others.

Most of the time, when working with the DOM, the developer will never even have to know that an XML parser is involved, because the parser is at a lower level than the DOM, and will be hidden away.

### DOMString

In order to ensure that all DOM implementations work the same, the DOM specifies the `DOMString` data type. This is a sequence of 16-bit units (characters) which is used anywhere that a string is expected.

In other words, the DOM specifies that all strings must be UTF-16 (refer back to [Chapter 2](#) if you need a reminder of what this is). Although the DOM specification uses this DOMString type anywhere it's talking about strings, this is just for the sake of convenience; a DOM implementation doesn't actually need to make any type of DOMString object available.

Many programming languages, such as Java, JavaScript, and Visual Basic, work with strings in 16-bit units natively, so anywhere a DOMString is specified these programming languages could use their native string types. On the other hand, C and C++ can work with strings in 8-bit units or in 16-bit units, so care must be taken to ensure that you are always using the 16-bit units in these languages.

---

[!\[\]\(2a618e2de01841f89add750b30fda589\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(2539adf4ffbdeb2ca29a9741efe7b69a\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# What Are Interfaces?

Although the name "Document Object Model" has the word "object" in it, the DOM doesn't really deal with objects very much; it defines a set of **interfaces** that objects must conform to. Since that's the case, we'd better take a look at what interfaces are, and what they're good for.

We already know what objects are. They encapsulate our data, and the code to work with that data. But sometimes, when we're modeling our data, we don't want to create a specific object, but a *type* of object, in other words a collection of methods and properties that one or more objects may support. For example, I might want to create an object type for a musician, which I would call Musician (I'm nothing if not boring and predictable). This type would have a property for the type of instrument the musician plays (maybe instrument, which would return a string containing the name of the instrument), and methods to play the music (maybe startSong(), crescendo(), improvise(), etc.).

But, the thing about musicians is that anyone can be a musician. There are people who devote their lives to being musicians, and there are people who play music as a hobby. Someone can be a musician and, at the same time, be a teacher, or a bank teller, or the President of the United States. What we need is a way to apply our object types to any object. That's what an interface does. It's a contract to support certain properties and methods, which can be applied to an object.

*Different programming languages may or may not use the word "interface", or have a specific mechanism for providing interfaces, but the same concept can be applied in any language.*

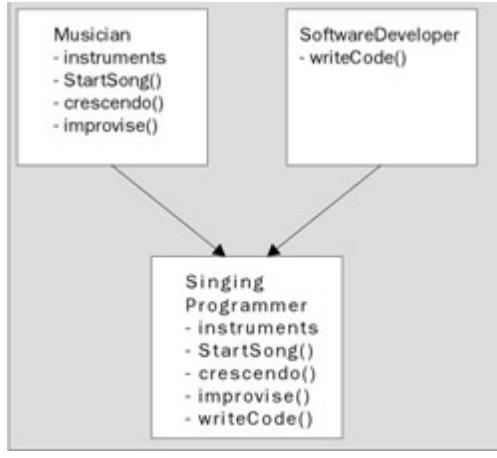
## Implementing Interfaces

If we were to create a type of object called SingingProgrammer, we could program that object to **implement** the Musician interface. This means we're declaring that our SingingProgrammer object type will provide all of the properties and methods of the Musician interface. Anyone using a SingingProgrammer object would then know that the object had an instruments property, and a startSong() method, in addition to whatever other properties and methods the SingingProgrammer object supports (such as annoyCoWorkers(), for example).

Of course, we would still have to write all of the needed code for the SingingProgrammer object type's implementation of Musician; it's not enough to say that SingingProgrammer implements Musician, we have to then write the code for all of the methods and all of the properties. This has the added benefit that the way our SingingProgrammer object implements startSong() might be entirely different from the way a SingingBankTeller object implements startSong(), although both will deliver the same behavior.

## Implementing Multiple Interfaces

But we aren't just limited to one interface per object. I could also create a SoftwareDeveloper interface, with a writeCode() method, and have the SingingProgrammer object implement that interface in addition to the Musician interface. People using our SingingProgrammer object would then know that it is not only a musician, but a software developer as well, and with all of the functionality of both.



If we had written these interfaces and objects in Java, we could write code like the following:

```
//create a SingingProgrammer object
SingingProgrammer oMyObject = new SingingProgrammer();

//call methods of the Musician and Software Developer interfaces
oMyObject.improvise();
oMyObject.writeCode();
```

This declares a SingingProgrammer object, and calls various methods from both of the interfaces that SingingProgrammer implements. Since we know that SingingProgrammer implements these interfaces, we know that these methods will be present, as will all of the other methods and properties available from a SingingProgrammer object. (In Java, if an object declares that it implements an interface, the Java compiler won't let the program compile until we write code for *all* of the properties and methods of the interface.)

We could also write code like the following:

```
//create a Musician variable, which references a SingingProgrammer object
Musician oMyObject = (Musician) new SingingProgrammer();

//call Musician methods
oMyObject.improvise();

//this won't work
oMyObject.writeCode();
```

This may be a bit confusing if you haven't programmed with interfaces before. We have created a variable of type Musician, and that variable is referencing a SingingProgrammer object. Or, to put it another way, we have created a SingingProgrammer object, but when accessing the object through this variable, we are only allowed to access the methods and properties available from the Musician interface. (That's why we can't call the writeCode() method.) We can't access any methods or properties of the other interfaces associated with the SingingProgrammer object. This Musician variable doesn't have to reference a SingingProgrammer object, but can reference *any* type of object that implements the Musician interface.

The point is: a SingingProgrammer object isn't *just* a SingingProgrammer object. It is also a Musician object, and a SoftwareDeveloper object. If I were to write a function in Java, which took a Musician object as a parameter, then I could pass it a SingingProgrammer object, because a SingingProgrammer object is a Musician object.

*Taking an object of one type, such as our SingingProgrammer object, and treating it as if it were another type of object, such as a Musician, is called casting.*

Because the DOM uses interfaces exclusively for defining its **Application Programming Interface (API)**, a lot of flexibility is provided for people creating DOM implementations; they can use any programming language they wish,

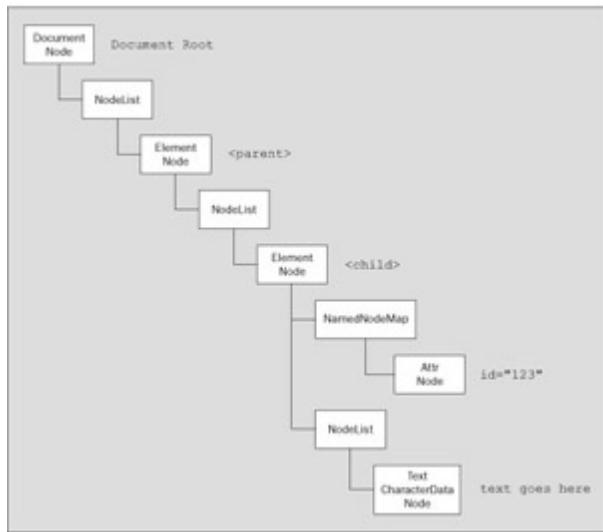
and still create W3C DOM-compliant implementations.

## DOM Interfaces

To get an idea of what interfaces are involved in the DOM, let's take a very simple XML document, such as this one:

```
<parent>
  <child id="123">text goes here</child>
</parent>
```

and see how it would look as represented by the DOM:



Each box represents an object that will be created; the names in the boxes are the interfaces that will be implemented by each object. For example, we have an object to represent the whole document, and objects to represent each of the elements and nodes. Each object implements a number of appropriate interfaces, such as Text, CharacterData, and Node for the object that represents the "text goes here" character data.

As you can see, most of the objects above implement the Node interface. Node is a fundamental interface in the DOM. Almost all of the objects you will be dealing with will extend this interface, which makes sense, since any part of an XML document is a node.

It is important to remember, when working with the DOM, that any changes you make to the structure of the XML document are only made in memory, and not to the original document. That is, if you create an XML file on your C: drive, called C:\blah.xml, and then load that document into the DOM and modify the information, C:\blah.xml will remain unchanged, unless you somehow explicitly save your changes back to the document.

## DOM Implementations

The Document Object Model is not just an API for working with XML documents; it can also be used for working with HTML documents, CSS stylesheets, and a variety of other documents might be introduced in future levels of the DOM. In fact, DOM implementations can be specialized to work only with XML documents or only with HTML documents, or they can be built to work with a number of types of documents. For example, a DOM which is shipped as part of an XML parser would probably only have the XML-specific APIs implemented, whereas a DOM which is shipped as part of a web browser would probably have the HTML and CSS-specific APIs included, to allow programmatic access to those types of documents.

The combination of the DOM (with the HTML API) scripting languages and HTML is what makes **Dynamic HTML** (or **DHTML**) possible, because the contents of an HTML document are exposed through the object model. When a web page is loaded into a DHTML-aware web browser, objects are created for each and every element on the page. This allows the web page writer to insert script on the page, to call methods and properties on

those objects.

For example, if we have an HTML form like this:

```
<form id="frmScratchForm" name="frmScratchForm">
  <input name="rdoRadio" type="radio" checked>First <br>
  <input name="rdoRadio" type="radio">Second
</form>
```

we can write JavaScript code to select the second radio button (the index is zero-based so 0 refers to the first item , 1 to the second, etc.) like this:

```
document.frmScratchForm.rdoRadio[1].checked = true;
```

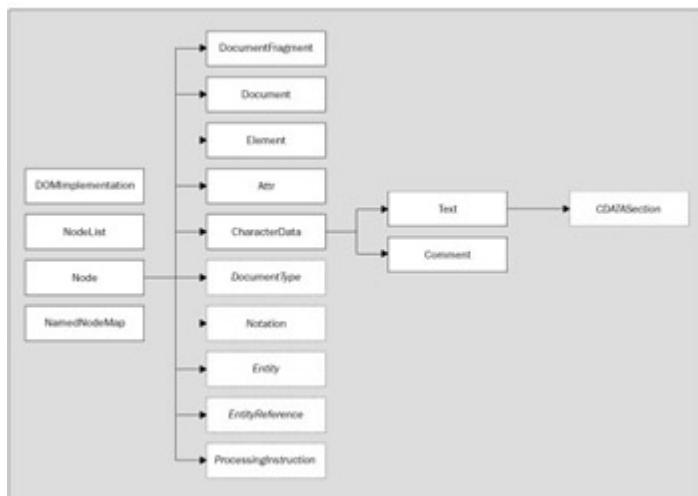
So even though the first radio button will be selected when the page loads, we can change it programmatically.

Because there are different types of DOM implementation, the DOM provides the **DOM Core**, a core set of interfaces for working with basic documents, and a number of optional modules for working with other documents: **DOM HTML**, **DOM CSS**, etc. These modules are sets of additional interfaces that can be implemented. The DHTML example above used objects from the DOM HTML module.

For the rest of this chapter, we're going to be concentrating on the DOM Core.

## The DOM Core

The DOM Core provides the following interfaces:



These core interfaces are further broken down into [Fundamental Interfaces](#) and [Extended Interfaces](#).

- The Fundamental Interfaces must be implemented by all DOM implementations, even ones that will only be working with non-XML documents.
- The Extended Interfaces only need to be implemented by DOM implementations that will be working with XML.

This begs the question why the Extended Interfaces were included in the DOM Core, instead of being in a separate DOM XML, but that's the way it is. Since this is a book on XML, we will only study the DOM Core interfaces, but many of the concepts you learn will be useful if you ever need to learn one of the optional modules.

## Extending Interfaces

As you're reading through the interface descriptions in this chapter, keep in mind that the properties and methods listed for each interface are just the *minimum* required in any DOM implementation. That means someone writing an implementation of the DOM can **extend** these interfaces with their own properties and methods, and most DOM implementations do extend the basic interface.

### What Do We Mean By Extending Interfaces?

With our interfaces we can define generic types of objects, and by implementing the interfaces any object can be of that type.

Let's go back to our Musician example. The more I think about it, the more I realize that there are a lot of different types of musician out there. What if we wanted an interface for a specific type of musician, like a JazzMusician interface? Jazz musicians might do things other musicians might not, like free-form solos, but they will also do anything that a regular musician would do.

In this case, what we want to do is *extend* the Musician interface. (Again, the terminology may differ from language to language.) When we create our JazzMusician interface, we declare that it does everything a Musician does, plus whatever extra properties and methods we choose to add, like maybe a freeFormSolo() method. Like all other aspects of interfaces, the way we extend an interface is entirely language-dependent, meaning that it would work differently in C++ than in Java or Visual Basic.

But in all cases, the net result is the same: if an object is implementing the JazzMusician interface, it must implement all of the properties and methods of that interface, plus all of the properties and methods of Musician. That object would not only be a JazzMusician object, but also a Musician object.

### Extending Interfaces in the DOM

The most fundamental interface we'll meet with the DOM is the **Node interface**. This interface represents a single node in the document tree. Since most of the objects in the DOM are nodes, most of the interfaces extend the Node interface. This means that the objects implementing these interfaces have all of the properties and methods of Node, plus whatever additional properties and methods are needed.

There's no limit to how many levels deep your extensions can go. For example, you can see from our diagram of the DOM interfaces that there's a CDATASection interface, which extends the Text interface, which in turn extends the CharacterData interface, which in turn extends the Node interface. So any object which implements CDATASection is also by definition a Text object, a CharacterData object, and a Node object.

### Vendor-Specific Extensions

A different type of extension is one that is supplied by a vendor, for their specific DOM implementation. That is, a method or a property which is not part of the DOM specification, but was added by the vendor.

A good example of extensions provided for DOM implementations is the ability to load and save XML documents. The DOM specification doesn't specify any way to do this, so it is completely implementation-specific as to how you would load an XML document into the DOM, or save it to disk. The MSXML DOM, which we'll be using in this chapter, provides some extensions for this purpose:

- The load() method will load an XML document from a URL or a stream
- The loadXML() method will load an XML document from a string

- The save() method will save an XML document as a Unicode string to your hard drive, in the location you specify
- The xml property returns the entire contents of the XML document, in a string

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Using the DOM

The DOM is a pretty flexible technology, when it comes to working with XML documents. It not only allows us to pull information out of a document, it allows us to modify that information, and add to it. We could even create an XML document from scratch, if we wanted to.

In the next few sections, we'll see

- How to navigate around the objects in the DOM, and get information out of an XML document
- How to add nodes to a document, or create one from scratch
- How to remove nodes from a document
- How to move nodes around

## Important

There is a complete listing of all the properties and methods of every interface in DOM Level 2 in [Appendix A](#).

But before we get too far into how the DOM works, we should cover how **exceptions** are handled in the DOM.

## Exceptions

A number of programming languages provide a mechanism called **exceptions**, for handling error situations. These are a way to signal that a problem has occurred. The DOM also uses exceptions for its error handling.

Exceptions are defined by creating a special type of object. When a situation arises which will cause an exception, this exception object is **thrown**, and then **caught** by an **exception handler**.

Different programming languages deal with exceptions differently. For example, in Java exception handling would look like this:

```
try
{
    //code here that might raise an exception

    //OR, raise one ourselves
    throw objException;
}
catch(ObjectType e)
{
    //deal with the exception
}
```

Any code that might raise an exception is included in a try block, and the code to handle the exception is in a catch block. If you need to, you can explicitly throw an exception, using the throw keyword.

Exception handling in Visual Basic 6 looks quite different:

```
On Error Resume Next  
'code which might raise an exception goes here  
If Err.Number <> 0 Then  
    'an error occurred; deal with it  
End If  
On Error GoTo 0
```

Visual Basic uses a special object, called Err, which holds information about an error. There are some default actions that VB performs when an error occurs, so On Error Resume Next tells VB not to handle the error, because we want to do it ourselves, and to keep on going with the processing. We then check the Err object's number property, to see if there is indeed an error, and if so deal with it. Finally, On Error GoTo 0 tells VB to go ahead and start handling errors in its own way again.

So, as you can see, exception handling can differ greatly from language to language. Also, not all languages have this kind of exception handling; this makes working with the DOM a little bit different.

For our examples, we'll be working with JavaScript on a web page in IE 5 or later. Earlier versions of the JavaScript language have no specified exception handling capabilities, while later versions support exception handling through the same try ? catch syntax as Java. (JavaScript error handling is supported in Internet Explorer versions 5 and later, and Netscape Navigator versions 6 and later.)

## DOMException

The DOM defines an interface that is implemented by any exception objects that are thrown: the DOMException interface. This is actually a very simple interface; it has only one property, code, which is a number indicating what type of error has occurred.

In the DOM Level 2 Recommendation there are 15 possible values for code, with other codes reserved by the W3C for future use. (For a full list of the codes and their meanings, see [Appendix A](#).)

For example, if we were working with the DOM in Java, we might write error-handling code such as the following:

```
try  
{  
    sValue = oDOM.documentElement.firstChild.nodeValue;  
}  
catch(DOMException e)  
{  
    if(e.code == NOT_FOUND_ERR)  
    {  
        //the node wasn't found in the document  
    }  
}
```

Since we know that the DOM objects will throw DOMException objects when an exception occurs, we can use the code property to figure out what type of error was raised.

## Try It Out-Our Test Page

In order to test out what we learn about the DOM, we'll be using MSXML 3. Using JavaScript within a web page, we can load and work with XML documents using the DOM. MSXML 3 doesn't support Level 2 of the DOM, however, it only supports Level 1 with some extensions.

We'll create a simple HTML page that we can use to work with the DOM. Type the following into Notepad (or your favorite HTML editor), and save it to your hard drive as dom.html:

```
<html>
<head><title>DOM Demo</title>

<script language="JavaScript">

var oDOM;
oDOM = new ActiveXObject ("MSXML.DOMDocument");
oDOM.async = false;
oDOM.load("domnode.xml");

//our code will go here...

</script>

</head>
<body>
    <p>This page demos some of the DOM capabilities.</p>
</body>
</html>
```

## How It Works

This page needs an XML document, in this case domnode.xml, in order to work. As we create each Try It Out, we'll cover what needs to go into that document.

The HTML page itself doesn't actually do anything, except display the text This page demos some of the DOM capabilities. All of the work will be done in that `<script>` block, and any results we want to see will be displayed in message boxes. We'll always write our code after the comment that reads, "our code will go here?".

The JavaScript code loads our XML document into the DOM by using a property called `async`, and a method called `load()`. In our code above, we simply supplied the name of the XML document to `load()`, so it must reside in the same directory as our HTML page. The `async` property we used specifies to MSXML whether it should load the XML document **synchronously**, or **asynchronously**. When a document is loaded asynchronously, this means that the `load()` method will return right away, and load the document in the background. (MSXML provides a property called `readyState`, which will indicate when the document has finished loading, or various other states in between when it starts and when it finishes.) This will allow us to write code that runs while MSXML is loading the document. On the other hand, if we need to work with the data right away, we probably want to load the data synchronously, meaning that `load()` won't return until the document is finished loading, which is what we specified in this case.

With this page written, we're now ready to start looking at how the DOM works, beginning with the basics: how to get information out.

## Retrieving Information from the DOM

As you'll recall from previous chapters, XML documents are trees of information, and the relationships between nodes are expressed as parent/child, ancestor/descendant, etc. The DOM allows access to the nodes in a tree by exposing properties that work with these concepts.

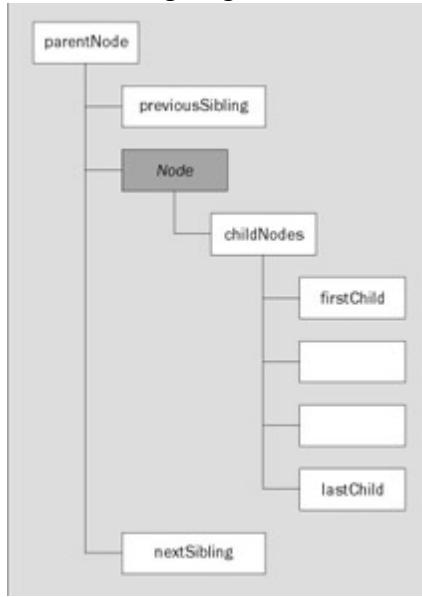
The Node interface provides a number of properties for dealing with direct relationships; these properties are the `parentNode`, `firstChild`, `lastChild`, `previousSibling`, and `nextSibling` properties, all of which return a Node, or the `childNodes` property, which returns a NodeList. (A NodeList contains a collection of Node objects.)

Not all nodes can have children (attributes, for example), and even if a node can have children it might not, but when that happens any properties which are supposed to return children will just return null. (Or, in the case of `childNodes`,

will return a NodeList with no nodes.)

Though `Node` is implemented in most DOM objects, it has some properties and methods that may not be appropriate for certain node types. These are just included for the sake of convenience, so that if you're working with a variable of type `Node`, you will have access to some of the functionality of the other interfaces, without having to cast to one of those types.

The following diagram shows a node, and the node that would be returned from each of these properties:



To get at the children of the node, we access the `childNodes` property. Or, if we want the first or the last child, there are properties that directly return these nodes, which is easier than having to navigate through the `childNodes` property. (If a node has only one child, `firstChild` and `lastChild` will both return that node.)

The `parentNode` property returns the node to which this node belongs in the tree, and `previousSibling` and `nextSibling` return the two nodes that are children of that parent node, and on either side of the node we're working with.

## The `hasChildNodes()` Method

If you just want to check if the node has children at all, there is a method named `hasChildNodes()`, which returns a Boolean value indicating whether there are any. (Note that this includes text nodes, so even if an element has only text for a child, `hasChildNodes()` will return true.) For example, we could write code like the following, so that if a node has any children, a message box will pop up with the name of the first one:

```
if (objNode.hasChildNodes ())  
{  
    alert (objNode.firstChild.nodeName);  
}
```

## The `ownerDocument` Property

Also, since every node must belong to a document, there's also a property called `ownerDocument`, which returns an object implementing the `Document` interface to which this node belongs. Almost all of the objects in the DOM implement the `Node` interface, so this allows you to find the owner document from any object in the DOM.

## Try It Out-Navigating the Tree

Let's put our HTML test page to its first use, and write some code to navigate the DOM.

1.

First, create the following XML document, and save it to your hard drive, as domnode.xml:

```
<root>
<DemoElement DemoAttribute="stuff">This is the PCDATA</DemoElement>
</root>
```

2.

Modify dom.html as follows, and save it as dom2.html:

```
<script language="JavaScript">

var oDOM;
oDOM = new ActiveXObject("MSXML.DOMDocument");
oDOM.async = false;
oDOM.load("domnode.xml");

//our code will go here...
alert(oDOM.firstChild.firstChild.firstChild.nodeValue);

</script>
```

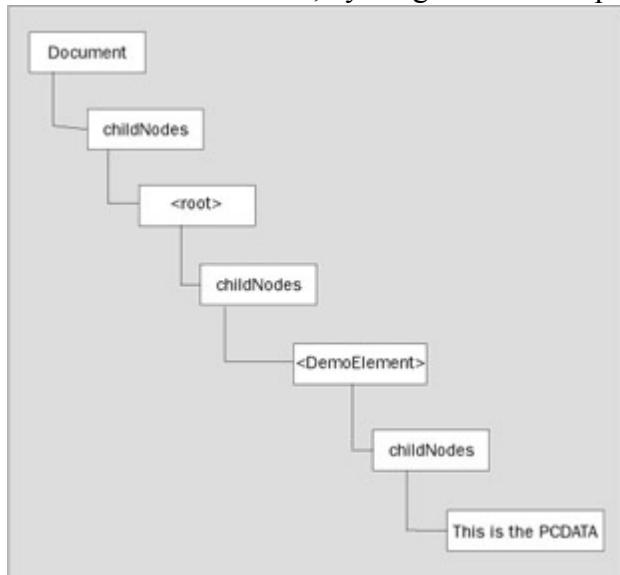
3.

Then view the page in IE 5 or later. This will pop up a message box:



## How It Works

Remember from our previous discussion that each Node object provides a NodeList, called childNodes, which contains all of that node's children. Also remember that we can directly access the first node in that list, using the firstChild property. So, in the code above, we navigate all the way from the document root to the text of the <DemoElement> element, by using the firstChild property. Graphically, this document can be illustrated as follows:



## Getting Information About a Node

The Node interface has several properties that let us get information about the node in question.

## The `nodeType` Property

The `nodeType` property returns a number, which indicates what type of node you're dealing with (all of the possible values for `nodeType` are listed in [Appendix A](#)). For example, you could check to see if you're working with an Element like:

```
if (objNode.nodeType == 1)
```

Luckily for us, most DOM implementations will include predefined constants for these node types. For example, a constant might be defined called `NODE_ELEMENT`, with the value of 1, meaning that we could write code like this:

```
if (objNode.nodeType == NODE_ELEMENT)
```

This makes it easier to tell what we are checking for, without having to remember that `nodeType` returns "1" for an element. ([Appendix A](#) lists all the DOM node types.)

## The `attributes` Property

A good example of a property of Node that doesn't apply to every node type is the [attributes](#) property, which is *only* applicable if the node is an element. The [attributes](#) property returns a `NamedNodeMap`, containing any attributes of the node. If the node is not an element, or is an element with no attributes, the [attributes](#) property returns null.

*A `NamedNodeMap` is similar to a `NodeList`, except that the nodes are not ordered. We usually access the nodes in a `NodeList` by their position, whereas we access the nodes in a `NamedNodeMap` by their name. We'll talk about these two interfaces in a later section.*

## The `nodeName` and `nodeValue` Properties

Two pieces of information that you will probably want from any type of node are its name and its value, and Node provides the `nodeName` and `nodeValue` attributes to retrieve this information. `nodeName` is **read-only**, meaning that you can get the value from the property but not change it, and `nodeValue` is **read-write**, meaning that you can change the value of a node if desired. We've already seen the `nodeValue` in action, in the previous Try It Out.

The values returned from these properties differ from node-type to node-type (this is covered in more detail in [Appendix A](#)). For example, for an element, `nodeName` will return the name of the element, but for a text node, `nodeName` will return the string "#text", since PCDATA nodes don't really have a name.

If we have a variable named `oNode` referencing an element like <first>John</first>, then we can write code like this:

```
alert(oNode.nodeName);  
//pops up a message box saying "first"  
  
oNode.nodeName = "FirstName";  
//will raise an exception! nodeName is read-only  
  
alert(oNode.nodeValue);  
//pops up a message box saying "null"
```

The result of that second alert may surprise you; why does it return "null", instead of "John"? The answer is that the text inside an element is not part of the element itself; it actually belongs to a text node, which is a child of the element node.

If we have a variable named oText, which points to the text node child of this element, we can write code like this:

```
alert(oText.nodeName);
//pops up a message box saying "#text"

alert(oText.nodeValue);
//pops up a message box saying "John"

oText.nodeValue = "Bill";
//this is allowed, the element is now <first>Bill</first>
```

## Try It Out-Accessing Element Information with Node

To demonstrate some of these concepts, we'll reuse our previous XML document, and look at some of the information returned from the Node interface.

1.

After the "our code will go here" comment in dom2.html, delete the code from the previous Try It Out.

2.

Now add the following code:

```
//our code will go here...
var oMainNode;
oMainNode = oDOM.documentElement.firstChild;
alert(oMainNode.nodeName);
```

The documentElement property is a special property of the Document interface, which returns the <root>node.

3.

Now save the file as dom3.html, and open it in IE5. The result will be a message box, with the name of the first child node of <root>: in this case DemoElement:



4.

Now change the last line of the previous code to print the node's value, instead of its name, like this:

```
//our code will go here...
var oMainNode;
oMainNode = oDOM.documentElement.firstChild;
alert(oMainNode.nodeValue);
```

5.

Save the HTML as dom4.html, and refresh the page in the browser. This pops up a message box with the word null; why is that?



Remember that this is an element we're talking about. The text in that element is contained not in the element itself, but in a Text child.

Important

An element doesn't have any values of its own, only children.

## NodeList and NamedNodeMap

Many of the properties and methods in the DOM will return a collection of Nodes, instead of just one, which is why the NodeList and NamedNodeMap interfaces were created.

NodeList is actually a very simple interface; there is only one property and one method:

- The length property returns the number of items in the NodeList.
- The item() method returns a particular item from the list. As a parameter, it takes the **index** of the Node you want.

Items in a NodeList are numbered starting at 0, not at 1. That means that if there are five items in a NodeList, the length property will return 5, but to get at the first item you would call item(0), and to get the fifth item you would call item(4). So the last Node in the NodeList is always at position length - 1. If you call item() with a number that's out of the range of this NodeList, it will return null.

A node list is always "live"; that means that if you add nodes to, or remove nodes from, the document, a node list will always reflect those changes. For example, if we got a node list of all elements in the document with a name of "first", then appended an element named "first", the node list would automatically contain this new element, without us having to ask it to recalculate itself.

The NamedNodeMap interface is used to represent an unordered collection of nodes. Items in a NamedNodeMap are usually retrieved by name.

It should come as no surprise that there is a getNamedItem() method, which takes a string parameter specifying the name of the node, and returns a Node object.

Even though the items in a NamedNodeMap are not ordered, you still might want to iterate through them one by one. For this reason, NamedNodeMap provides a length property and an item() method, which work the same as length and item() on the NodeList interface. But the DOM specification is clear that this "*does not imply that the DOM specifies an order to these Nodes*". (You can see this for yourself at <http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210/core.html#ID-1780488922>). This means that you might not get items out of the NamedNodeMap in the same order that they appear in the document.

## Try It Out-Accessing Items in a NodeList

To see how a NodeList works, we'll first need a group of nodes.

1.

Save the following XML to your hard drive as domodelist.xml:

```
<?xml version="1.0"?>
<simple>
  <name>John</name>
  <name>David</name>
  <name>Andrea</name>
  <name>Ify</name>
  <name>Chaulene</name>
  <name>Cheryl</name>
  <name>Shurnette</name>
  <name>Mark</name>
  <name>Carolyn</name>
  <name>Agatha</name>
</simple>
```

2.

We can load this XML into MSXML as in our previous examples.

3.

Modify dom.html as follows:

```
<html>
<head><title>DOM Demo</title>

<script language="JavaScript">

  var oDOM;
  oDOM = new ActiveXObject("MSXML.DOMDocument");
  oDOM.async = false;
  oDOM.load("domodelist.xml");

  //our code will go here...

</script>

</head>
<body>
  <p>This page demos some of the DOM capabilities.</p>
</body>
</html>
```

4.

Now we can get a list of all of the nodes named name by using this code which you should add:

```
var oNodeList;
oNodeList = oDOM.getElementsByTagName("name");
```

The `getElementsByTagName()` function returns a NodeList, containing all of the descendant elements of a node that have the specified tag. In this case, the NodeList will contain all of the elements which descent from the document root, and which are called "name".

5.

And we can then get the text from the second `<name>` element like so:

```
alert(oNodeList.item(1).firstChild.nodeValue);
```

6.

Save the file as dom5.html and view it in IE 5. This produces a message box like this:



*If you are puzzled why item(1) returns the second child, remember that it is a ten member list numbered from 0 to 9, so item(1) is the second list member.*

### Try It Out-Accessing Nodes in a NamedNodeMap

The Node interface has an [attributes](#) property, which returns a NamedNodeMap, so we'll write some simple JavaScript code to work with the map:

1.

Save the following as nodemap.xml, to the same directory as dom.html:

```
<root first="John" last="Doe" middle="Fitzgerald Johansen"/>
```

2.

Next open up the dom.html web page we've been working with, and change it to load the new XML file, like so:

```
<html>
<head><title>DOM Demo</title>
<script language="JavaScript">

var oDOM;
oDOM = new ActiveXObject("MSXML.DOMDocument");
oDOM.async = false;
oDOM.load("nodemap.xml");

//our code will go here...
</script>

</head>
<body>
<p>This page demos some of the DOM capabilities.</p>
</body>
</html>
```

3.

First of all, we'll get our NamedNodeMap to contain the attributes of this node, which happens to be the root element. Add the following code right after the "our code will go here" comment:

```
var oMap;
oMap = oDOM.documentElement.attributes;
```

4.

We could then get the value of the first attribute like so:

```
alert(oMap.getNamedItem("first").nodeValue);
```

5.

Save this as dom6.html. Open it in IE5 and you will see the following message box:



## Creating XML Documents Through the DOM

When creating an XML document from scratch, or even adding nodes to an existing document, most of the work is done through the Document interface. A Document object is often the only type of object you can create yourself. That is, if you were starting from scratch, you would not be able to create a Node object or a DOMException object, but you would be able to create a Document object.

This interface provides **factory methods** that can be used to create other objects. These methods are named `createNodeType()`, where `NodeType` is the type of node you want to create, for example `createElement()` or `createAttribute()`. This makes adding nodes to an XML document a two-step process:

- First, create the node using one of the Document factory methods
- Second, append the child in the appropriate spot

*In addition to the `createNodeType()` methods, there are also `createNodeTypeNS()` methods-these work the same, but create the nodes in a particular namespace.*

The Document interface represents an entire XML document. It extends the Node interface, so any Node properties and methods will also be available from a Document object. For Document, the node will be the document root-not the root element. Remember that for an XML document, the document root is a conceptual entity that contains everything else in the document, including the root element.

All nodes must belong to one, and only one, document. Even if a node is not currently part of the tree-structure of the document, such as one that has been created but not yet added, it still belongs to the document. That is, if we use one of the Document interface's methods to create a node, the node won't immediately be a part of the document's tree. However, that node will belong to the document that created it. This also means that a node cannot be moved from one document to another, or created from one document and added to another.

Once a new node has been created, it must be appended to the document. The Node interface provides the `appendChild()` and `insertBefore()` methods to do this.

### The `appendChild()` Method

The appendChild() method takes an object-implementing Node as a parameter, and just appends it to the end of the list of children. You might append one node to another like this:

```
oParentNode.appendChild(oChildNode);
```

The oChildNode node is now the last child node of oParentNode, regardless of what type of node it is.

## The insertBefore() Method

To have more control over where the node is inserted, you could call insertBefore(). This takes two parameters: the node to insert, and the "reference node", or the one before which you want the new child inserted. The following will add the same oChildNode to the same oParentNode, but the child will be added as the second last child:

```
oParentNode.insertBefore(oChildNode, oParentNode.lastChild);
```

## Try It Out-Creating an XML Document from Scratch

Let's see how to create an XML document programmatically.

1.

To start, we'll go back to our original dom.html document, which looked like this:

```
<html>
<head><title>DOM Demo</title>

<script language="JavaScript">
  var oDOM;
  oDOM = new ActiveXObject("MSXML.DOMDocument");
  oDOM.async = false;
  oDOM.load("domnode.xml");

  //our code will go here...

</script>

</head>
<body>
  <P>This page demos some of the DOM capabilities.</P>
</body>
</html>
```

Delete the lines:

```
  oDOM.async = false;
  oDOM.load("domnode.xml");
```

2.

We're creating a Document object called oDom. We'll use that object to create an element and a text node.

Insert the following lines of code right after the comment:

```
  //our code will go here...
  var oNode, oText;
  oNode = oDOM.createElement("root");
  oText = oDOM.createTextNode("root PCDATA");
```

The createElement() method takes the name of the element to be created as its parameter, and createTextNode() takes as its parameter the text we want to go into the node.

3.

With these objects created, we can now add the element to our document (which will make it the root element), and add the text node to that element. Add the following code right after the code you've already entered:

```
oDOM.appendChild(oNode);
oNode.appendChild(oText);
alert(oDOM.xml);
```

The first command adds the <root> tags and the second the PCDATA.

The xml property we call in that last line is a Microsoft-specific extension to the DOM, which returns the entire XML document as a string.

4.

Save the HTML as dom7.html and view it with IE 5 or later. The following message box will appear:



5.

Now let's add an attribute to that node. Add the following lines of code right before the alert, as shown:

```
oDOM.appendChild(oNode);
oNode.appendChild(oText);

var oAttr;
oAttr = oDOM.createAttribute("id");

//set the attribute's value
oAttr.nodeValue = "123";

//append the attribute to the element
oNode.attributes.setNamedItem(oAttr);

alert(oDOM.xml);
```

Save this as dom8.html.

The createAttribute() method takes the name of the attribute as its parameter, so we've created an attribute named id, given it the value "123", then added that attribute to the node. (Later we'll see an easier way of doing this, using the setAttribute method.)

The setNamedItem() method belongs to the NamedNodeMap interface, and is used for inserting items into the node map-or, in this case, adding an attribute to an element.

Our XML now looks like this:



Thus, we have created an entire XML document, all from code.

## Removing Nodes from a Document

Sometimes, instead of adding to a document, we want to remove information from it. The Node interface provides the `removeChild()` method, which takes a reference to the child you want to remove, and returns that object back to you, in case you want to use it somewhere else. Even though the node is removed from the tree, it still belongs to the document, although if we were to remove the child and then save the document, it would be lost. So we could remove the last child of any node, and keep it in a variable, like this:

```
oOldChild = oParent.removeChild(oParent.lastChild);
```

## Try it Out-Removing a Node From a Document

For this example, we'll reuse our `domnode.xml` document.

1.

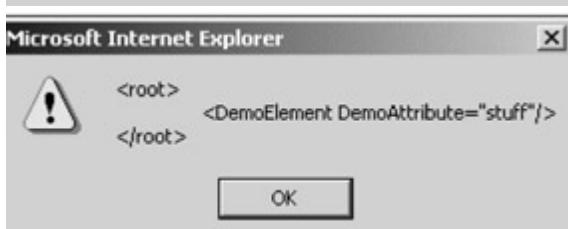
Open `dom.html`, and delete any code after the "our code will go here" comment.

2.

Add the following code and save the file as `dom9.html`:

```
//our code will go here...
var oNode, oRemovedNode;
oNode = oDOM.documentElement.firstChild;
oRemovedNode = oNode.removeChild(oNode.firstChild);
alert(oRemovedNode.nodeValue);
alert(oDOM.xml);
```

We're removing the PCDATA child of `<DemoElement>` from the document. Two message boxes will be shown, as follows:



## How It Works

As mentioned earlier, the `removeChild()` method returns the child that was removed. We used this functionality to get the text of our PCDATA, even after we had removed it. We then show the entire XML document, to illustrate that the PCDATA is really gone.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Advanced DOM Topics

So far, we've seen how to get information out of an XML document, create a document from scratch or add nodes to a document, and how to remove nodes from a document. The next topics will discuss some more advanced things we can do with the PCDATA in our XML documents, as well as a couple of handy extension methods.

## Working With Text

As you're well aware, working with XML documents involves a lot of work with text: sometimes in PCDATA in the XML document, and sometimes in other places, like attribute values, or comments. The DOM defines two interfaces for this purpose:

- A CharacterData interface, which has a number of properties and methods for working with text
- A Text interface, which extends CharacterData, and is used specifically for PCDATA in the XML document

Because CharacterData extends Node, both CharacterData objects and Text objects are also Node objects.

## Handling Complete Strings

The simplest way to get or set the PCDATA in a CharacterData object is simply to get it from the data property. This sets or returns the whole string in one chunk.

There is also a length property, which returns the number of Unicode characters in the string.

*When dealing with strings in CharacterData objects, note that the characters in the string are numbered starting at 0, not 1. So in the string "Hi", "H" would be letter 0, and "i" would be letter 1.*

So, if we have a Text node object named oText containing the string "John", then:

```
alert(oText.length);
```

pops up a message box saying 4, and :

```
alert(oText.data);
```

pops up a message box saying John.

## Handling Substrings

If you only want a part of the string, there is a substringData() method, which takes two parameters:

- The offset at which to start taking characters, starting with 0
-

The number of characters to take

If you specify more characters than are available in the string, substringData() just returns the number of characters up until the end and stops.

For example, if we have a CharacterData object named oText, and the contents of that object are "This is the main string", then:

```
alert(oText.substringData(12, 4));
```

would pop up a message box saying main, and:

```
alert(oText.substringData(12, 2000));
```

would pop up a message box saying main string, since the DOM will stop reading characters when it gets to the end of the string.

## Modifying Strings

Adding text to the end of a string is done with the appendData() function, which takes a single string parameter, containing the text to add to the end of the existing text.

If we used the same oText node as above, then:

```
oText.appendData(".");
```

would change the contents to "This is the main string." with the period added.

But since we sometimes need to add data to the middle of a string, there is also the insertData() method, which takes two parameters:

- The offset at which to start inserting characters
- The string you wish to insert

The following code would change the data to "This is the groovy main string":

```
oText.insertData(12, "groovy ");
```

Deleting characters from the string is done via the deleteData() method, which you use exactly the same as the substringData() method. So calling:

```
oText.deleteData(12, 7);
```

on the CharacterData node we've been working with would change the string back to "This is the main string." removing the text "groovy".

And finally, if you want to replace characters in a string with other characters, instead of calling deleteData() and then insertData(), you could simply use replaceData(). This method takes three arguments:

-

The offset position at which to start replacing

- The number of characters to replace
- The string to replace them with

Note that the number of characters you're inserting doesn't have to be the same as the number of characters you're replacing; it can be shorter or longer.

If we still have the same oText node containing "This is the main string.", we could do the following:

```
oText.replaceData(8, 8, "a");
```

which would replace "the main" with "a", thus changing the string to "This is a string".

## Splitting Text

The Text interface only adds one method to the ones inherited from CharacterData, which is the splitText() method. This takes one Text object and splits it into two, which are siblings of each other. The method takes one parameter, which is the offset at which to make the split.

The result is that the first Text node will contain the text from the old node until (but not including) the offset point, and the second Text node will contain the rest of the text from the old node. If the offset is equal to the length of the string, the first Text node will contain the old string as it was, and the new node will be empty; and if the offset is greater than the string's length, a DOMException will be raised.

We could, therefore, write code like this:

```
oText.splitText(12);
```

And the result would look something like this:



Of course, if we were to save this XML document like that, our change would be lost, since the PCDATA would then just become one string again. splitText() comes in most handy when you are going to be inserting other elements in the middle of the text.

## Try It Out?Putting splitText() to Practical Use

To demonstrate this, let's take the following HTML paragraph:

```
<p>This is the main string</p>
```

and make that word "main" bold using the DOM.

1.

Save the following as domtext.xml:

```
<?xml version="1.0"?>
<p>This is the main string</p>
```

2.

Extend our usual dom.html document as follows, to create a Node object to hold this paragraph:

```
<html>
<head><title>DOM Demo</title>

<script language="JavaScript">

var oDOM;
oDOM = new ActiveXObject("MSXML.DOMDocument");
oDOM.async = false;
oDOM.load("domtext.xml");

//our code will go here...
var oNode;
oNode = oDOM.documentElement;

</script>

</head>
<body>
<p>This page demos some of the DOM capabilities.</p>
</body>
</html>
```

We're going to take one Text node, and change it into three nodes:

○

A Text node

○

A child <b> node

○

Another Text node

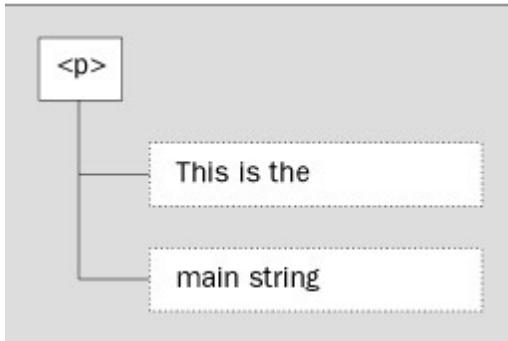
3.

Our first step is to break the node into two separate text nodes. Add the following to the HTML file:

```
var oNode;
oNode = oDOM.documentElement;

var oText;
oText = oNode.firstChild;
oText.splitText(12);
```

Our <p> node now looks like this:

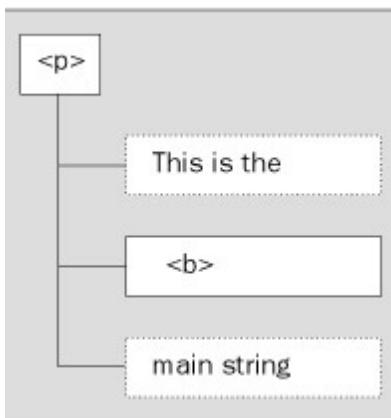


4.

Next, we'll create the <b> element, and insert it as a child:

```
var oBElement;  
oBElement = oDOM.createElement("b");  
oNode.insertBefore(oBElement, oNode.lastChild);
```

Now our <p> node looks like this:

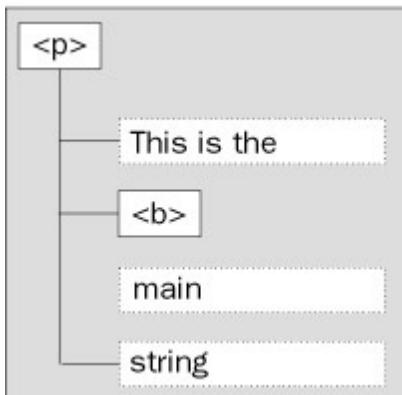


5.

We now want to split the second Text node, so that we can strip the word "main" out and insert it as a child of the <b> element:

```
oText = oNode.lastChild;  
oText.splitText(4);
```

Which makes our <p> node look like this:



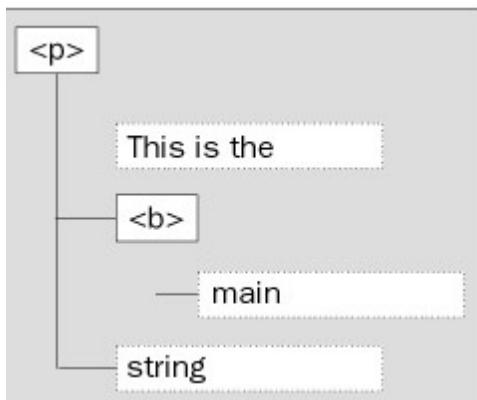
6.

And, finally, we want to remove the second Text node, and append it to the <b> element:

```
oText = oNode.removeChild(oNode.childNodes.item(2));
```

```
oNode.childNodes.item(1).appendChild(oText);  
alert(oNode.xml);
```

Our final result looks like this:



7.

Now save the file as dom10.html and open it in the browser. The message box contains the following XML:



And you thought splitText() would be difficult!

## DOM Extensions

As mentioned earlier, the DOM Recommendation specifies only the *base* that a DOM implementation must provide. Implementers are free to add to that base, to provide additional functionality.

We've already seen some of the extensions provided by MSXML to load an XML document: load() can retrieve an XML document from a URL, and can do so either synchronously or asynchronously, by using the async property. The loadXML() method can load an XML document from a string, such as

```
oDOM.loadXML ("<mydoc/>")
```

MSXML also provides a save() method, which can save an XML document to the hard drive, and we've already been using the xml property, which returns the entire document as a string.

Every DOM implementation will have to provide similar extension methods and properties, since the DOM Level 2 Recommendation doesn't specify any standard way to load an XML document into the DOM (other than by creating it from scratch), or any standard way to save an XML document. Although using these extensions is non-portable, since they will only work for a particular DOM implementation, they are unavoidable.

In addition to these extensions, MSXML also provides a couple of convenience extensions for retrieving the data from an XML document. These extensions are not necessary for working with the DOM, as the loading and saving methods are, but can be handy. So if you want to write code which would work with any DOM implementation, you might want to avoid these extensions, but if you know that you will only be working with this particular DOM implementation, they can make your code easier to write, and even make it execute more quickly.

## The selectNodes() And selectSingleNode() Methods

We've seen how to navigate through the DOM using the properties and methods supplied by the Node interface, such as firstChild, lastChild, childNodes, etc. We have also seen the getElementsByTagName() method, which will return a NodeList containing all of the descendant elements with a certain name.

The drawback to using the Node properties is that we have to navigate quite a way through the document. For example, recall our earlier domnode.xml document:

```
<root>
  <DemoElement DemoAttribute="stuff">This is the PCDATA</DemoElement>
</root>
```

In order to get at the PCDATA in the <DemoElement> element, we had to write code like this:

```
alert(oDOM.firstChild.firstChild.firstChild.nodeValue);
```

This isn't very pretty. In fact, it would be a lot easier if we could give the DOM an XPath expression, containing the node or nodes that we want, and have it give us back the relevant data. This is what the selectNodes() and selectSingleNode() methods provide: They take a string, containing an XPath expression, and return a Node (for the selectSingleNode() method) or NodeList (for the selectNodes() method) object.

*Note that Microsoft isn't the only DOM vendor who is working on this type of extension. The DOM implementation from Oracle also has similar selectNodes() and selectSingleNode() methods, which return nodes specified by an XPath expression.*

For example, we could get at the same PCDATA using selectSingleNode(), as follows:

```
alert(oDOM.selectSingleNode("/root/DemoElement").nodeValue)
```

Since these two extension methods use XPath to specify the desired nodes, we have a lot more control over the results. For example, using getElementsByTagName() we could get back a NodeList containing all of the <name> elements:

```
oDOM.getElementsByTagName("name")
```

however, using selectNodes() we could filter that, to only return <name> elements that have a first attribute with a value of "John".

```
oDOM.selectNodes("//name[@first='John']")
```

Also, getElementsByTagName(), as its name implies, can only return elements, whereas selectSingleNode() and selectNodes() can return any node types. We could use selectNodes() to get back a NodeList containing all of the first attributes in a document like so:

```
oDOM.selectNodes("//@first")
```

## Try It Out?The Extensions In Action

For this example, we'll demonstrate some of the flexibility provided by selectNodes() and selectSingleNode().

1.

Enter the following into Notepad, and save it to your hard drive as people.xml. Make sure you save it to the same directory as dom.html.

```
<?xml version="1.0"?>
<people>
  <programmers>
    <name>John</name>
```

```
<name>Bill</name>
<name>Sally</name>
<name>Joanne</name>
</programmers>
<managers>
  <name>Fred</name>
  <name>Mary</name>
  <name>Wilma</name>
  <name>Sally-Anne</name>
  <name>John</name>
</managers>
</people>
```

2.

Open dom.html, and change the line of code that loads the XML, to load our new document.

```
var oDOM;
oDOM = new ActiveXObject("MSXML.DOMDocument");
oDOM.async = false;
oDOM.load("people.xml");
```

3.

First, we'll write some code to list all of the programmers in this file. (Rather than popping up a message box for each and every one, we'll create a string that contains all of the names, and pop that up in a message box. That will be a little less annoying.)

Add the following code after the "our code will go here" comment:

```
var oNodes, sNames, i;

oNodes = oDOM.selectNodes("/people/programmers/name")
sNames = new String();
for(i = 0; i < oNodes.length - 1; i++)
{
    sNames = sNames + oNodes.item(i).firstChild.nodeValue;
    sNames = sNames + ", ";
}
sNames = sNames + oNodes.item(i).firstChild.nodeValue;
alert(sNames);
```

4.

Now let's also use the selectSingleNode() method to get the name of the second manager in our list. Add the following after the code you just entered:

```
alert(oDOM.selectSingleNode("/people/managers/name[1]").firstChild.nodeValue);
```

5.

Save this as dom11.html, and load it in the browser. You should get two message boxes, as follows:



## How It Works

For our first piece of code, we used `selectNodes()` to return a list of all of the `<name>` elements under the `<programmers>` element. We could have done the same with `getElementsByTagName()`, but it would have been a bit messier:

```
oNodes = oDOM.documentElement.firstChild.getElementsByTagName ("name");
```

For our second piece of code, we used `selectSingleNode()` to return the second `<name>` element under the `<managers>` element. Again, we could have used the standard DOM interfaces to do this, but it would have been a bit messier:

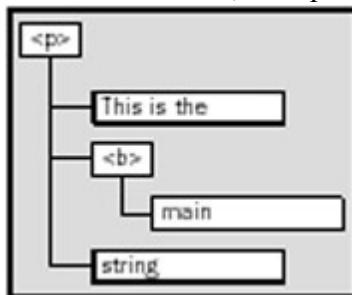
```
oDOM.documentElement.childNodes.item(1).childNodes.item(1).firstChild.nodeValue
```

## The Text Property

Another extension added to MSXML is the `text` property. This property will return all of the PCDATA for the current node, as well as any children. For example, consider the following:

```
<p>This is the <b>main</b> string</p>
```

As we saw before, the `<p>` element has three children, as pictured here:



In order to show all of the text under the `<p>` element, we would have to combine the first text child, the text child of the `<b>` element, and the second text child. However, with the `text` property of MSXML, we could do this in one operation, similar to this:

oPElement.text

## Try It Out?Using The Text Property

Let's take that example and put it into a real Try It Out, to see that it really works.

1.

Open Notepad, and type in the following XML:

```
<p>This is the <b>main</b> string</p>
```

Save it as string.xml, to the same directory as dom.html.

2.

Open dom.html and change it so that the XML document being loaded is the string.xml document.

```
<script language="JavaScript">
```

```
var oDOM;  
oDOM = new ActiveXObject("MSXML.DOMDocument");  
oDOM.async = false;  
oDOM.load("string.xml");  
  
//our code will go here...  
</script>
```

3.

Add the following line of code:

```
//our code will go here...  
alert(oDOM.documentElement.text);
```

4.

Save your changes as dom12.html, and load the page into the browser. You will get a message box, as follows:



## How It Works

When we call the text property, it recursively gets the text not only of the current node but also of any child and descendent nodes as well.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

The DOM is one of the more widely recognized and easiest to use specifications to come out of the W3C concerning XML. Many, if not most, programmers who will be working with XML will be using the DOM to get at the data, or to create new XML documents.

In this chapter we have learned all we need to use the DOM to process XML documents, including:

- How to read the information in an XML document
- How to add to that information, or even create XML documents from scratch
- Some handy extension functions we can use, if we are able to settle on one particular DOM implementation

Because the DOM is creating all of these objects in memory, one for each and every node in the XML document, DOM implementations can be quite large, and processing XML documents via the DOM can take up a lot of memory. In the [next chapter](#), we'll be studying another way that we can get information out of our documents. If the DOM is too slow, or takes up too much memory, we can use the Simple API for XML (SAX) instead.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 9: The Simple API for XML (SAX)

## Overview

And now for something *slightly* different. When it comes to analyzing XML documents and extracting information out of them, the DOM isn't the only game in town. There's another API called **SAX**, which turns out to be very good at the things that the DOM isn't so good at (although, as you might expect, it's not so good at the things the DOM is good at). The two approaches should be regarded as complementary, and any XML programmer should be conversant with both of them.

In this chapter you will learn:

- What SAX is, and why it was created
- Where you can get SAX
- How to use the two main SAX 2.0 interfaces: ContentHandler and ErrorHandler
- How to use SAX in both Java and Microsoft Visual Basic
- When to use SAX

I'm duty bound at this point to issue a programming warning. This chapter contains explicit illustrations of programming ? using Java and Microsoft Visual Basic in particular?and anyone who is uncomfortable with this may like to skip to another chapter now. (In fact, unless you plan to use XML with Java or Visual Basic, you'll probably never come across SAX.) As with the [last chapter](#) (in fact more so) we're going to have to assume some programming experience, although I promise I'll steer clear of the weird stuff. If you want to try any of the Java examples, you're also going to need some kind of Java development environment. If you want to try out any of the Visual Basic examples towards the end of the chapter, you're going to have to shell out some hard cash for Microsoft Visual Basic; the code in this chapter has been tested under version 6 of Visual Basic. You won't, however, be needing a browser.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# What is SAX, and Why Was it Invented?

The **Simple API for XML**, or SAX, was developed in order to enable more efficient analysis of large XML documents. The problem with the DOM is that before you can use it to traverse a document, it has to build up a massive in-memory map of the document. This takes up space, and more importantly time. If you're trying to extract a small amount of information from the document, this can be extremely inefficient.

Let's illustrate this by a simple analogy.

## The Fabulous Lost Treasure of the Xenics

Imagine that you're an archaeologist searching a labyrinth of catacombs somewhere to find the Fabulous Lost Treasure Of The Xenics (FLTDX). Unfortunately, in recent years, you've spent more time on the interview circuit than in the field, and, as a result, you're no longer in good enough shape to go rummaging around in damp tunnels with low ceilings. (Although you'd like to be able to stroll in and pick up the treasure for the benefit of the world's cameras, of course.) Instead, you send in your athletic young assistant, Indy. Or is it Lara? All these kids look the same to you?

We have two approaches here to locating the FLTDX. Firstly, Indy/Lara could map out the entire catacombs and contents for you. Once you've got such a map, you could find your way to the treasure without any trouble. However, this is a process that could, frankly, take years, and our evil arch-rival (there's always one) may well get to the treasure before us. Instead, what we could do is fit our assistants with a radio mike, and send them off, telling them to map everything as they go, but also to report back to us on everything they find.

On our headset, we'd probably hear something like:

"Er ? pile of flints ? spear ? ornamental vase ? small votive bust ? Fabulous Lost Treasure Of The Xenics ?"

"STOP!"

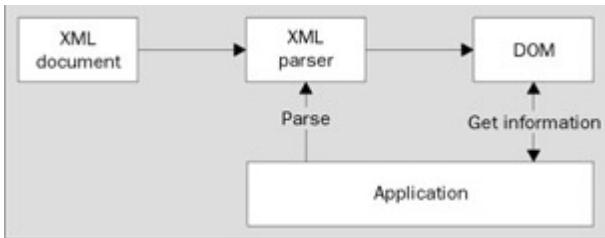
## So What Does That Have to Do with SAX?

If only life was always that simple. Well, maybe XML can be. Let's interpret the analogy. The catacombs are, of course, supposed to represent an XML document. Indy/Lara is our parser, and the map that they are writing is the in-memory image of the XML document.

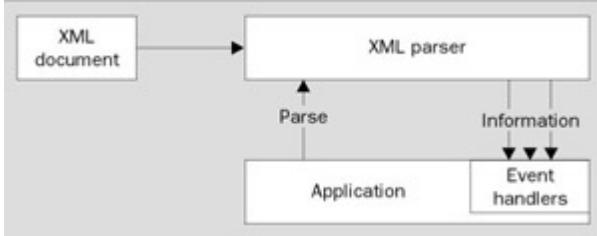
In the first instance, they were providing us with a DOM view of the document. As we saw, this can be time-consuming, although if we need that complete map (for instance, if we wished to build a copy of the catacombs for our new theme-park venture, or if we also wanted to find the Amazing Missing Treasure Of The Vrmls) it's the best way to proceed.

However, if all we want to do is locate specific parts of the document, the second approach is more appropriate. This is the way SAX works, and we call it **event-driven**. Rather than parse the document into the DOM and then use the DOM to navigate around the document, we tell the parser to raise **events** whenever it finds something, just like Indy/Lara giving us a running commentary over the headset.

If we were to show the two approaches diagrammatically, DOM would look something like this:



Whereas the SAX approach is more like this:



Now anyone who has done user-interface work will be familiar with this kind of "tell me when something happens" approach. After all, the best way to deal with a user clicking on buttons is to have a bit of code that's triggered whenever the user does something, rather than having it in a constant loop saying "What are you doing now?" over and over again. However, it's a little different from the run of the mill for document analysis. The good news is that it's not rocket science, either, and once you've got used to the idea of event-driven programming, you'll be doing it all the time, I promise you.

## A Brief History of SAX

The extraordinary thing about SAX is that it isn't owned by anyone. It doesn't belong to any consortium, standards body, company or individual. So it doesn't survive because so-and-so says that you must use it in order to comply with standard X, or because company Y is dominant in the marketplace. It survives because it's simple and it works.

SAX arose out of discussions on the XML-DEV mailing list (now hosted by OASIS at <http://www.oasis-open.org/>, and you can read the archives at <http://lists.xml.org/archives/xml-dev/>) aimed at resolving incompatibilities between different XML parsers (this was back in the infancy of XML in late 1997). David Megginson took on the job of coordinating the process of specifying a new API, and then declared the SAX 1.0 specification frozen on 11 May 1998. A whole series of SAX 1.0-compliant parsers then began to emerge, both from vast corporations (IBM and Sun, for example) and from enterprising individuals (such as James Clark). All of these parsers were freely available for public download.

Eventually, a number of shortcomings in the specification became apparent, and David Megginson and his chums got back to work, finally producing the SAX 2.0 specification on 5 May 2000. David Megginson continues to coordinate SAX development, and his site (<http://www.megginson.com>) is well worth a visit for up-to-date information.

Not all the SAX 1.0 parser writers took up the challenge of upgrading to SAX 2.0. This is probably because there were already several big contenders in the fray: Sun Java API for XML Parsing (JAXP), the Apache project's Xerces parser, and another newcomer, who we'll meet in a minute or two. All of these in fact have full support for DOM, too. The Apache software actually originates from IBM, who donated a release of the source from their xml4j project. From that point on, xml4j and Apache diverged, and both products are pursuing a vigorous life of their own. For the purposes of this chapter, we will use the Apache Xerces parser.

SAX is actually specified as a set of Java interfaces, which, up until recently, meant that if you were going to do any serious work with it, you were going to be looking at doing some Java programming, using JDK 1.1 or later. There are parsers emerging for other languages (Perl and Python, for example), but until late in 2000, there was no agreed standard for C++ or COM. Actually, there still isn't an agreed C++ or COM standard, but that's largely irrelevant, as on October 27, 2000, Microsoft released version 3 of their XML Parser and SDK, which included full COM support for SAX 2.0. At the time of writing, version 4.0 has just been released. It is highly likely that this implementation will become the de facto COM (and hence Visual Basic and probably C++) standard.

In deference to the SAX tradition, however, we're going to start with some Java examples. Once we've explored the Java implementation of SAX in some detail, we'll repeat the task with a Visual Basic application using the Microsoft COM classes. Unfortunately, we have to use full-blown Visual Basic here, rather than VBScript, because of the event-driven nature of SAX. We appreciate that Microsoft Visual Basic is not a free download, like the other tools that we've used in this chapter, but it's so important that it's worth a look at anyway.

## Where to Get SAX

I don't know about you, but I'm itching to try all this out. However, before we can do anything, we need to get hold of some software. Here's what we need:

- For our parser, we're going to use Apache Xerces, version 1.4.3. This is available from <http://xml.apache.org/xerces-j>. You should already have this installed from the work we did in [Chapter 5](#).
- We'll also need our Java development kit, release 1.1 or later. If you don't happen to have one lying around and didn't download the full JDK when going through [Chapter 5](#), your best bet is to download the Sun Java 2 Platform, Standard Edition from <http://java.sun.com/j2se/1.3>. However, this is pretty massive download (30 Mbytes or so), and if you're pushed for bandwidth, JDK 1.1 is quite acceptable. This is still available from <http://java.sun.com/products/jdk/1.1>, although even this is a good 8 Mbytes, so you might want to get this ready well in advance.
- Finally, if we want to try out Visual Basic SAX, we'll need Version 3 of the Microsoft XML parser and SDK; you can get these from <http://msdn.microsoft.com/xml/general/xmlparser.asp>.

A full list of other SAX-aware parsers can be found on <http://www.megginson.com/SAX/>.

# Using the SAX Interfaces

It's time to get to work and actually try some of this out. First of all, you need to install the JDK, if you haven't already, which should be easy. JDK pretty much installs itself. Under Windows, for example, it comes as a self-extracting executable, and all you have to do is follow the instructions on screen. You should already have Xerces installed and on your CLASSPATH from the work we did in [Chapter 5](#), if not refer to the instructions given there.

## How to Receive SAX Events

One final thing before we get stuck in. You may be wondering how we're going to be receiving these events. Remember the discussion on **interfaces** in the [last chapter](#)? If not, it might be a good time to take a look again to refresh your memory. What we're going to do is write a Java class that **implements** one of the SAX interfaces. If we pass a reference to an instance of that class into our parser, the parser can then use that interface to talk back to us.

We specify that a class **implements** an interface by declaring it like this:

```
public class MyClass implements ContentHandler
```

MyClass is the name of my new class, and ContentHandler is the name of the interface. Actually, this is the most important interface in SAX, as it is the one that defines the callback methods for content related events (that is, events about elements, attributes, and their contents). So what we're doing here is creating a class that contains methods that a SAX-aware parser knows about.

The ContentHandler interface contains a whole series of methods, most of which in the normal course of events we don't really want to be bothered with. Unfortunately, when you implement an interface, you have to provide implementations of *all* the methods defined in that interface. However, SAX provides us with a default, empty, implementation of them, called DefaultHandler. So rather than *implement* ContentHandler, we can instead *extend* DefaultHandler, like this:

```
public class MyClass extends DefaultHandler
```

We can then pick and choose which methods we want to provide our own implementations, in order to trap specific events. This is called **overriding** the methods, and it works like this.

If we leave things as they are, the base class (DefaultHandler in this case) provides its own implementation of them for use by MyClass. So if, for instance, there's a method in the ContentHandler interface called doSomething, whenever another piece of code invokes the doSomething method of MyClass's implementation of ContentHandler, the method invoked is actually DefaultHandler.doSomething. This is because DefaultHandler is providing a default implementation of doSomething. This is called "inheriting an implementation".

However, if we provide our own implementations of the methods, then they are used instead. In our example above, the method invoked would now be MyClass.doSomething. This might do something totally different from DefaultHandler's implementation.

Actually, DefaultHandler is a really hard-working class, because it also provides default implementations of the three other core SAX interfaces: ErrorHandler, DTDHandler, and EntityResolver. We'll come across them in a little while, but for the time being, we'll focus on ContentHandler.

The most important methods in ContentHandler are as follows:

Method	Description
startDocument	Receive notification of the start of a document
endDocument	Receive notification of the end of a document
startElement	Receive notification of the start of an element
endElement	Receive notification of the end of an element
characters	Receive notification of character data

The first pair of methods that we might want to override are startDocument and endDocument. I think we're ready for a small Try It Out.

## Try It Out?Minimal Parsing Using SAX

1.

In order to do this example, we'll need an XML document. Create one as follows:

```
<?xml version="1.0"?>
<bands>
    <band type="progressive">
        <name>King Crimson</name>
        <guitar>Robert Fripp</guitar>
        <saxophone>Mel Collins</saxophone>
        <bass>Boz</bass>
        <drums>Ian Wallace</drums>
    </band>
    <band type="punk">
        <name>X-Ray Spex</name>
        <vocals>Poly Styrene</vocals>
        <saxophone>Laura Logic</saxophone>
        <guitar>Someone else</guitar>
    </band>
    <band type="classical">
        <name>Hilliard Ensemble</name>
        <saxophone>Jan Garbarek</saxophone>
    </band>
    <band type="progressive">
        <name>Soft Machine</name>
        <organ>Mike Ratledge</organ>
        <saxophone>Elton Dean</saxophone>
        <bass>Hugh Hopper</bass>
        <drums>Robert Wyatt</drums>
    </band>
</bands>
```

A motley bunch, if ever there was one. Save this as bands.xml in a convenient directory. We're going to start by writing a piece of software that will print out the name of every element found.

2.

Now let's create our Java class, in a separate document. The class will be called BandReader. In order to be able to use names like DefaultHandler, rather than org.xml.sax.helpers.DefaultHandler, we need to add the following lines at the start of our code:

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;

import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
```

This will ensure that the Java compiler will know where to look for classes that it can't find locally.

3.

Now here comes our class declaration and main() method:

```
public class BandReader extends DefaultHandler
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("Here we go ...");
        BandReader readerObj = new BandReader();
        readerObj.read(args[0]);
    }
}
```

The main method declaration is standard Java: this is the piece of code that will be executed when we start the class. It prints out a message, creates a new instance of the class that it resides in, and invokes a method called read().

The void part of the declaration, incidentally, means that the method doesn't return a value to its caller, and the throws Exception part means that if anything happens that it can't cope with, it passes the exception back so the caller can deal with it. We'll see more of exceptions later.

4.

The read() method is where things get interesting:

```
public void read (String fileName) throws Exception
{
    XMLReader readerObj =
XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
    readerObj.setContentHandler (this);
    readerObj.parse (fileName);
}
```

The first line of this method creates an XMLReader object using a factory helper object. This is, in fact, the only place where we explicitly refer to the fact that it's the Xerces parser that we're using. So we could in fact substitute the qualified name of another parser here if we happened to have another one handy.

Having got our XML reader, we need to tell it which event handlers it is to use. For the time being, we're only going to handle content-related events, so we need to specify an implementation of ContentHandler. Since our class implements ContentHandle (because it extends DefaultHandler), this will do just fine. We refer to the current instance of our class as "this", and we pass this as the parameter to setContentHandler.

5.

Finally, we need to tell Xerces which document to parse, by invoking the parse() method.

6.

All that's left to do is to write the methods that will catch the events coming back from the parser. These are pretty simple (even if it may not feel that way just now if you're new to Java). There's one to catch the "start of document" event:

```
    public void startDocument() throws SAXException
{
    System.out.println("Starting ...");
}
```

There's one to catch the "end of document" event:

```
    public void endDocument() throws SAXException
{
    System.out.println("... Finished");
}
```

And there's one to catch the start of each element:

```
    public void startElement(String uri, String localName, String qName,
Attributes atts) throws SAXException
{
    System.out.println("Element is " + qName);
}
}
```

So here's our entire BandReader class:

```
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

public class BandReader extends DefaultHandler
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("Here we go ...");
        BandReader readerObj = new BandReader();
        readerObj.read(args[0]);
    }

    public void read (String fileName) throws Exception
    {
        XMLReader readerObj =
XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
        readerObj.setContentHandler (this);
        readerObj.parse (fileName);
    }

    public void startDocument() throws SAXException
    {
        System.out.println("Starting ...");
    }

    public void endDocument() throws SAXException
    {
        System.out.println("... Finished");
    }

    public void startElement(String uri, String localName, String qName, Attributes atts)
```

```
throws SAXException
{
    System.out.println("Element is " + qName);
}
}
```

7.

Save this as file BandReader.java. As you will have probably realized by now, this program doesn't have one of those clever flashy user interfaces, so we'll need to run it from a terminal session. In Windows, we can do this by opening a command prompt and changing the directory to the one where we've saved our files. Now let's compile it. (Incidentally, I'm assuming from here on that we're using the JavaSoft JDK that I described earlier on in the chapter. And remember, if you haven't changed your PATH variable, you may have to type the absolute path to the java executable javac.)

```
javac BandReader.java
```

8.

Now let's run it:

```
java BandReader bands.xml
```

9.

Here's what we see:

```
Here we go ...
```

```
Starting ...
Element is bands
Element is band
Element is name
Element is guitar
Element is saxophone
Element is bass
Element is drums
Element is band
Element is name
Element is vocals
Element is saxophone
Element is guitar
Element is band
Element is name
Element is saxophone
Element is band
Element is name
Element is organ
Element is saxophone
Element is bass
Element is drums
... Finished
```

## How It Works

We instantiated an XML reader object, using the helper factory:

```
XMLReader readerObj =
XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
```

Then we told it which object to raise the events with via the ContentHandler interface, and which file to parse:

```
readerObj.setContentHandler (this);
readerObj.parse (fileName);
```

Then it simply scanned through the document, raising the appropriate events at the start and end of the document:

```
public void startDocument() throws SAXException
{
    System.out.println("Starting ...");
```

```
}
```

```
public void endDocument() throws SAXException
{
    System.out.println("... Finished");
}
```

and at the start of each element:

```
public void startElement(String uri, String qName, Attributes
atts) throws SAXException           throws SAXException
{
    System.out.println("Element is " + qName);
}
```

## Extracting Character Data

Believe it or not, at this point we've actually covered most of the principles of SAX. However, it would be fair to say that we haven't actually used it for anything remotely interesting or useful. As you may have noticed, all we've managed to extract so far are the names of the tags. What would be more interesting is extracting the actual *character data* between those tags.

To do that, we need to implement a further method in the ContentHandler interface: characters. The declaration of characters looks like:

```
public void characters(char[] chars, int start, int len) throws SAXException
```

This method gets fired whenever the parser encounters a chunk of character data. That's not quite as straightforward as it might seem, as there's no obligation on the parser writer to deliver all the character data between two tags as a single block. (If you think about it, this is actually quite reasonable?after all, the string might turn out to be extremely long, and it could make for a very clumsy parser implementation.) From an application point of view, this just means that you may need to build up your string over a number of character events.

The three parameters are, respectively, the array of characters, the start position in the array, and the number of characters to read from the array.

Let's see how this might work in practice.

## Try It Out?Extracting Characters Using SAX

For this example, we'll stick with the same XML file as last time, bands.xml. This time, however, we're going to extract the names of all the saxophonists that we come across. (It's like using SAX to find sax, if you see what I mean.)

1.

Let's start off by copying BandReader.java to a new file, SaxFinder.java. We'll also need to change every occurrence of BandReader in the file to SaxFinder.

2.

Next, we need to add a couple of private member variables to our SaxFinder class, just before the declaration of the main method:

```
public class SaxFinder extends DefaultHandler
{
    private StringBuffer saxophonist = new StringBuffer();
    private boolean isSaxophone = false;

    public static void main(String[] args) throws Exception
```

The first one is the string buffer that's going to hold the name of each saxophone player as we receive character events from SAX. The second one is a flag that we're going to use to keep track of whether or not we're currently inside a <saxophone> element.

3.

We need to make some changes to the startElement method:

```
public void startElement(String uri, String localName, String qName,  
Attributes atts) throws SAXException  
{  
    if (qName.equals("saxophone"))  
    {  
        isSaxophone = true;  
        saxophonist.setLength(0);  
    }  
  
    else  
        isSaxophone = false;  
}
```

What we're doing here is checking to see if we're at the start of a <saxophone> element. If we are, then we set our flag, and clear out anything left behind in our string buffer, so that it's ready to hold our saxophonist's name as we build up the string.

4.

We also need to add an endElement method. We didn't need one previously, because we were only interested in the *start* of each element. Here's the new method:

```
public void endElement(String uri, String localName, String qName)  
    throws SAXException  
{  
    if (isSaxophone)  
    {  
        System.out.println("Saxophonist is " + saxophonist.toString());  
        isSaxophone = false;  
    }  
}
```

This simply checks to see if we've just finished a <saxophone> element. If we have, it outputs the saxophonist's name from our string buffer, and then clears down the flag.

5.

The last change that we need to make is to add a characters method:

```
public void characters(char[] chars, int start, int len)  
    throws SAXException  
{  
    if (isSaxophone)  
        saxophonist.append(chars, start, len);  
}
```

This method checks to see if we're currently looking at a saxophonist, and, if we are, appends the incoming characters to the end of our string buffer. Remember that we can't rely on SAX to give us the complete string in one go. There's no reason why we shouldn't get three events with characters like this, for example:

Andy S

hep  
pard

instead of a single one with Andy Sheppard.

Just to recap, here's the full code of SaxFinder:

```
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

public class SaxFinder extends DefaultHandler
{
private StringBuffer saxophonist = new StringBuffer();
private boolean isSaxophone = false;

public static void main(String[] args) throws Exception
{
    System.out.println("Here we go ...");
    SaxFinder readerObj = new SaxFinder();
    readerObj.read(args[0]);
}

public void read (String fileName) throws Exception
{
    XMLReader readerObj =
XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
    readerObj.setContentHandler (this);
    readerObj.parse (fileName);
}

public void startDocument() throws SAXException
{
    System.out.println("Starting ...");
}

public void endDocument() throws SAXException
{
    System.out.println("... Finished");
}

public void startElement(String uri, String localName, String qName,
                        Attributes atts) throws SAXException
{
    if (qName.equals("saxophone"))
    {
        isSaxophone = true;
        saxophonist.setLength(0);
    }

    else
        isSaxophone = false;
}

public void endElement(String uri, String localName, String qName)
                      throws SAXException
{
    if (isSaxophone)
    {
        System.out.println("Saxophonist is " + saxophonist.toString());
        isSaxophone = false;
    }
}

public void characters(char[] chars, int start, int len)
                  throws SAXException
```

```
{  
    if (isSaxophone)  
        saxophonist.append(chars, start, len);  
}  
}  
6.
```

Save the file and compile it in the same way as before:

```
javac SaxFinder.java
```

7.

Let's see what happens when we run it:

```
Here we go ...
```

```
Starting ...  
Saxophonist is Mel Collins  
Saxophonist is Laura Logic  
Saxophonist is Jan Garbarek  
Saxophonist is Elton Dean  
... Finished
```

## How It Works

The SAX parser searches through our document, finding our elements. The only things that we're interested in are the elements enclosed in <saxophone>?</saxophone> tags. We collect any character data that SAX throws at us between these tags and concatenate it into a string:

```
if (isSaxophone)  
    saxophonist.append(chars, start, len);
```

Then we output it once we've received the closing tag:

```
System.out.println("Saxophonist is " + saxophonist.toString());
```

## Extracting Attributes

By this point, I would imagine that you're beginning to get more than a little curious about a couple of aspects of the startElement declaration that we've so far glossed over. Just to remind ourselves, this is what it looks like:

```
public void startElement(String uri, String localName, String qName,  
                         Attributes atts) throws SAXException
```

The two things lurking in this declaration that we haven't mentioned yet are that [Attributes](#) parameter, and that SAXException thing. We'll take a look at SAXException in a little while, when we discuss error handling, but for the time being we'll concentrate on [Attributes](#).

There are four principal methods available on this object:

- getLength returns an integer giving the number of attributes in the list.
- getQName returns the qualified name of the attribute at a specified position in the list (starting from 0).
- getValue returns the value of an attribute; we can either specify which attribute by giving it a string containing the attribute name, or by specifying its position in the list (starting from 0).
- getType returns the type of an attribute; again we can either specify which attribute by name or by position.

The various types are described in [Appendix F](#).

Let's see if we can make good use of these in developing our application still further.

## Try It Out?Extracting Attributes

In this example, we're going to take the example from the last Try It Out, and extend it to list the names of all the bands that our saxophone players in bands.xml belong to, and what type of band they are. Here's how we do it.

1.

Copy the Java class SaxFinder.java to BandFinder.java. Don't forget to change all the instances of SaxFinder to BandFinder.

2.

Add three more member variables to our class:

```
public class BandFinder extends DefaultHandler
{
    private StringBuffer saxophonist = new StringBuffer();
    private boolean isSaxophone = false;
    private StringBuffer bandName = new StringBuffer();
    private boolean isName = false;
    private String bandType = new String();
```

BandName and isName are exact analogues of saxophonist and isSaxophonist, and we're going to use them to extract the name of the band from between the <name>...</name> tags. We're going to use bandType to hold the type of the band, as extracted from the attribute associated with the <band> tag.

3.

Our startElement method needs a little work doing to it. First of all, let's add a couple of lines to the beginning:

```
public void startElement(String uri, String localName, String qName,
                         Attributes atts) throws SAXException
{
    if (qName.equals("band"))
        bandType = atts.getValue("type");
```

Here, we're checking to see if we've got a <band> tag. If so, we know that we can extract its type from the type attribute, so this is what we do, using the getValue method. (You might be wondering what would happen if there wasn't a type attribute. As it happens, the [next section](#) is devoted to just this topic.)

Next, we need to add some code to handle <name> tags. We can pretty much clone this from the code for <saxophone>:

```
public void startElement(String uri, String localName, String qName,
                         Attributes atts) throws SAXException
{
    if (qName.equals("band"))
        bandType = atts.getValue("type");

    else if (qName.equals("name"))
    {
        isName = true;
        isSaxophone = false;
        bandName.setLength(0);
    }

    else if (qName.equals("saxophone"))
```

```
{  
    isName = false;  
    isSaxophone = true;  
    saxophonist.setLength(0);  
}  
  
else  
{  
    isName = false;  
    isSaxophone = false;  
}  
}  
4.
```

There's also a small change to endElement, to output the name of the band as well as the saxophonist. Then we clear out the flag that's used to keep track of the fact that we're in the middle of the band name element:

```
public void endElement(String uri, String localName, String qName)  
    throws SAXException  
{  
    if (isSaxophone)  
    {  
        System.out.println("The saxophonist in " + bandName.toString()  
            + " (" + bandType + ") is " + saxophonist.toString());  
        isSaxophone = false;  
    }  
    if (isName)  
        isName = false;  
}  
5.
```

Finally, there's some more cloning to be done in characters:

```
public void characters(char[] chars, int start, int len)  
    throws SAXException  
{  
    if (isName)  
        bandName.append(chars, start, len);  
  
    if (isSaxophone)  
        saxophonist.append(chars, start, len);  
}
```

Here's the complete BandFinder code:

```
import org.xml.sax.helpers.XMLReaderFactory;  
import org.xml.sax.XMLReader;  
import org.xml.sax.SAXException;  
import org.xml.sax.Attributes;  
import org.xml.sax.helpers.DefaultHandler;  
  
public class BandFinder extends DefaultHandler  
{  
    private StringBuffer saxophonist = new StringBuffer();  
    private boolean isSaxophone = false;  
    private StringBuffer bandName = new StringBuffer();  
    private boolean isName = false;  
    private String bandType = new String();  
  
    public static void main(String[] args) throws Exception  
    {  
        System.out.println("Here we go ...");  
        BandFinder readerObj = new BandFinder();  
        readerObj.read(args[0]);  
    }
```

```
public void read (String fileName) throws Exception
{
    XMLReader readerObj =
    XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
    readerObj.setContentHandler (this);
    readerObj.parse (fileName);
}

public void startDocument() throws SAXException
{
    System.out.println("Starting ...");
}

public void endDocument() throws SAXException
{
    System.out.println("... Finished");
}

public void startElement(String uri, String localName, String qName,
                        Attributes atts) throws SAXException
{
    if (qName.equals("band"))
        bandType = atts.getValue("type");

    else if (qName.equals("name"))
    {
        isName = true;
        isSaxophone = false;
        bandName.setLength(0);
    }

    else if (qName.equals("saxophone"))
    {
        isName = false;
        isSaxophone = true;
        saxophonist.setLength(0);
    }

    else
    {
        isName = false;
        isSaxophone = false;
    }
}

public void endElement(String uri, String localName, String qName)
                      throws SAXException
{
    if (isSaxophone)
    {
        System.out.println("The saxophonist in " + bandName.toString()
                           + " (" + bandType + ") is " + saxophonist.toString());
        isSaxophone = false;
    }

    if (isName)
        isName = false;
}

public void characters(char[] chars, int start, int len)
                  throws SAXException
{
    if (isName)
        bandName.append(chars, start, len);
```

```
    if (isSaxophone)
        saxophonist.append(chars, start, len);
}
}
```

## 6.

Save and compile the file as before. Then all that's left to do is to take it for a drive:

Here we go ...

Starting ...

```
The saxophonist in King Crimson (progressive) is Mel Collins
The saxophonist in X-Ray Spex (punk) is Laura Logic
The saxophonist in Hilliard Ensemble (classical) is Jan Garbarek
The saxophonist in Soft Machine (progressive) is Elton Dean
... Finished
```

## How It Works

We can extract the data for band names in the same way as we extract the names of their saxophonists:

```
if (isName)
    bandName.append(chars, start, len);
```

We can extract the attribute type from the attribute list passed in to startElement:

```
public void startElement(String name, AttributeList atts)
    throws SAXException
{
    if (name.equals("band"))
        bandType = atts.getValue("type");
```

## Error Handling

By now you'll have noticed that all the methods in the ContentHandler interface throw exceptions of type SAXException if anything goes wrong that can't be handled locally. If one of these exceptions gets thrown, it gets caught by the parser, which reports the error and shuts down. The same mechanism is used for reporting errors in parsing the XML.

So if, for example, we were to supply it with an incorrect bands.xml file, like this:

```
<?xml version="1.0"?>
<bands>
    <band type="progressive">
        <name>King Crimson</name>
        <guitar>Robert Fripp
        <saxophone>Mel Collins</saxophone>
        <bass>Boz</bass>
        <drums>Ian Wallace</drums>
    </band>
    <band type="punk">
        <name>X-Ray Spex</name>
        <vocals>Poly Styrene</vocals>
        <saxophone>Laura Logic</saxophone>
        <guitar>Someone else</guitar>
    </band>
    <band type="classical">
        <name>Hilliard Ensemble</name>
        <saxophone>Jan Garbarek</saxophone>
    </band>
    <band type="progressive">
        <name>Soft Machine</name>
        <organ>Mike Ratledge</organ>
```

```
<saxophone>Elton Dean</saxophone>
<bass>Hugh Hopper</bass>
<drums>Robert Wyatt</drums>
</band>
</bands>
```

we'd see something like this (because we missed off the closing guitar tag):

```
Here we go ...
Starting ...
The saxophonist in King Crimson (progressive) is Mel Collins
org.xml.sax.SAXParseException: The element type "guitar" must be terminated by the
matching end-tag "</guitar>".
    at org.apache.xerces.framework.XMLParser.reportError(XMLParser.java:1056)
    at
org.apache.xerces.framework.XMLDocumentScanner.reportFatalXMLError(XMLDocumentScanner.ja
va:635)
    at
org.apache.xerces.framework.XMLDocumentScanner.abortMarkup(XMLDocumentScanner.java:684)
    at
org.apache.xerces.framework.XMLDocumentScanner$ContentDispatcher.dispatch(XMLDocumentSca
nner.java:1188)
    at
org.apache.xerces.framework.XMLDocumentScanner.parseSome(XMLDocumentScanner.java:381)
    at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:948)
    at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:987)
    at BandFinder.read(BandFinder.java:31)
    at BandFinder.main(BandFinder.java:21)
Exception in thread "main"
```

We can make use of this mechanism ourselves, by explicitly throwing a SAXException when something has gone wrong:

```
if (something bad happened)
    throw new SAXException ("Something bad happened");
```

Let's see how this works in practice.

## Try It Out?Throwing Exceptions

For this example, let's say that we are particularly obsessive, and we want to insist that each of the bands described in our XML must have a type attribute.

*For something this simplistic, we actually do this using validation with Document Type Definitions (DTDs) or XML Schemas, but there are some complex requirements that are extremely hard?if not impossible?to impose using just a schema. For more on DTDs see [Chapter 5](#), and for XML Schemas, see [Chapters 6 and 7](#).*

1.

Copy BandFinder.java to BandValidator.java, and change every occurrence of BandFinder to BandValidator.

2.

Let's make a small change to our startElement method, so that it checks to see if the type attribute is non-null:

```
public void startElement(String name, AttributeList atts)
    throws SAXException
{
    if (name.equals("band"))
    {
        bandType = atts.getValue("type");

        if (bandType == null)
            throw new SAXException("Band type not specified");
```

```
}

else if (name.equals("name"))
```

So as soon as a band with no type attribute is encountered, the exception is thrown and the execution of the method ceases. The parser will catch the exception, report the errors and stop.

*Incidentally, if you're not familiar with Java, take care with the double "==" in the line where we're checking to see if the band type is null. Java uses this to denote a comparison, rather than an assignment (which just uses one "=").*

### 3.

Now let's repair the damage we did to bands.xml last time, and then change it so that one of the bands has no type attribute:

```
<?xml version="1.0"?>
<bands>
    <band>
        <name>King Crimson</name>
        <guitar>Robert Fripp</guitar>
        <saxophone>Mel Collins</saxophone>
        <bass>Boz</bass>
        <drums>Ian Wallace</drums>
    </band>
    <band type="punk">
        <name>X-Ray Spex</name>
        <vocals>Poly Styrene</vocals>
        <saxophone>Laura Logic</saxophone>
        <guitar>Someone else</guitar>
    </band>
    <band type="classical">
        <name>Hilliard Ensemble</name>
        <saxophone>Jan Garbarek</saxophone>
    </band>
    <band type="progressive">
        <name>Soft Machine</name>
        <organ>Mike Ratledge</organ>
        <saxophone>Elton Dean</saxophone>
        <bass>Hugh Hopper</bass>
        <drums>Robert Wyatt</drums>
    </band>
</bands>
```

Well, after all, the Crims *are* pretty hard to categorize ?

### 4.

Save both files and compile BandValidator.java. This is what we see if we now run it:

Here we go ...

```
Starting ...
Exception in thread "main" org.xml.sax.SAXException: Band type not specified
    at BandValidator.startElement(bandvalidator.java:58)
    at org.apache.xerces.parsers.SAXParser.startElement(SAXParser.java:1371)
    at org.apache.xerces.validators.common.XMLValidator.callStartElement(XML
Validator.java:828)
    at
org.apache.xerces.framework.XMLDocumentScanner$ContentDispatcher.dispatch (XMLDocumentSca
nner.java:1222)
    at
org.apache.xerces.framework.XMLDocumentScanner.parseSome (XMLDocumentScanner.java:380)
    at org.apache.xerces.framework.XMLParser.parse (XMLParser.java:908)
    at org.apache.xerces.framework.XMLParser.parse (XMLParser.java:947)
    at BandValidator.read (bandvalidator.java:34)
    at BandValidator.main (bandvalidator.java:24)
```

So the XML is rejected as being incorrect. We've used the parser's existing error-catching mechanism by throwing a SAXException at it.

## More About Errors?Using the Locator Object

In the last Try It Out, we raised the possibility of using SAX to provide sophisticated validation of an XML document. However, our last effort, whilst it did report the error to us, didn't give us much in the way of information about *where* it had actually occurred. For the validation to be at all meaningful, we need to tell the user where the problem occurred. This is where the Locator object comes in.

The parser may provide the Locator object, so that we can use it in our event methods to find out where we are in the document. I use the word "may" because, according to the SAX specification, the parser isn't obliged to provide a locator. However, in practice most of them do, although the results from different parsers aren't entirely consistent.

How do we get hold of this Locator object? What we do is implement yet another method of the ContentHandler interface (we'll have implemented the lot by the time we've finished, at which point the DefaultHandler implementation is going to be a waste of time). This new method is called setDocumentLocator, and it's declared like this:

```
public void setDocumentLocator(Locator loc)
```

All we need to do in our implementation is copy the reference to the object to a local member variable. We can then use it whenever we want to find out where we are. So what methods can we use on Locator?

- We can use the method getPublicId to get the public identifier of the current document event: generally speaking, this will be null
- The method getSystemId gets us the system identifier: this is usually the file name of the XML document that we're parsing
- The method getLineNumber returns us the current line number
- The method getColumnNumber returns us the current column number

Let's try this out.

## Try It Out?Using the Locator

In this example, we'll be using the Locator to refine our reporting of the validation error that we found in the last example. We'll stick with our BandValidator class.

1.

First of all, we need to add another import:

```
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.helpers.DefaultHandler;
import org.xml.sax.Locator;
```

2.

Having done that, let's add a member variable to hold our reference to the Locator:

```
public class BandValidator extends DefaultHandler  
{  
    private StringBuffer saxophonist = new StringBuffer();  
    private boolean isSaxophone = false;  
    private StringBuffer bandName = new StringBuffer();  
    private boolean isName = false;  
    private String bandType = new String();  
    private Locator locatorObj;
```

3.

Next, let's add that implementation of setDocumentLocator to the end of the document:

```
public void setDocumentLocator(Locator loc)  
{  
    locatorObj = loc;  
}
```

(I told you that there wasn't much to it.)

4.

All we need to do now is put it to use, in our implementation of startElement:

```
public void startElement(String name, AttributeList atts)  
    throws SAXException  
{  
    if (name.equals("band"))  
    {  
        bandType = atts.getValue("type");  
  
        if (bandType == null)  
        {  
            if (locatorObj != null)  
                System.err.println ("Error in " +  
                    locatorObj.getSystemId() + " at line " +  
                    locatorObj.getLineNumber() + ", column " +  
                    locatorObj.getColumnNumber());  
            throw new SAXException("Band type not specified");  
        }  
    }  
}
```

Notice that we check to see that the Locator object is non-Null, in case we've picked an inferior parser that doesn't support Locator.

5.

Fortunately, Xerces does. Save and compile BandValidator.java, and this is what you should see when it's run:

Here we go ...

```
Starting ...  
Error in file:///c:/saxtest/bands.xml at line 3, column 11  
Exception in thread "main" org.xml.sax.SAXException: Band type not specified  
    at BandValidator.startElement(bandvalidator.java:58)  
    at org.apache.xerces.parsers.SAXParser.startElement(SAXParser.java:1371)  
    at org.apache.xerces.validators.common.XMLValidator.callStartElement(XML  
Validator.java:828)  
    at  
org.apache.xerces.framework.XMLDocumentScanner$ContentDispatcher.dispatch (XMLDocumentSc  
anner.java:1222)  
    at  
org.apache.xerces.framework.XMLDocumentScanner.parseSome (XMLDocumentScanner.java:380)
```

```
at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:908)
at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:947)
at BandValidator.read(bandvalidator.java:34)
at BandValidator.main(bandvalidator.java:24)
```

The Locator object tells us where we currently are in the parsing process. Provided that the parser supports this, we can get hold of a reference to a Locator object by implementing the setLocatorObject method.

## Even More About Errors?Catching Parsing Errors

We still haven't finished with errors, because we need to consider the other side of the coin. What if we want to do our own handling of parser errors?

It probably won't come as a great surprise to find out that what we do is implement some methods of an interface. However, these new methods aren't part of ContentHandler, they're part of another interface altogether, called ErrorHandler. As it happens, DefaultHandler provides us with a rudimentary implementation of this one as well, although it doesn't actually do anything apart from throw a SAXException to print out a trace of the call stack, like the one we saw in the last example.

The new methods are as follows:

- warning?this gives us a warning that something untoward has happened. This would be a condition that isn't considered an "error" or "fatal error" by the W3C XML 1.0 recommendation.
- error?this reports a recoverable error. This would be an "error" as defined in section 1.2 of the W3C XML 1.0 recommendation, such as a validation error in the case of a validating parser.
- fatal?this reports a non-recoverable error. This would be a "fatal error" as defined in section 1.2 of the W3C XML 1.0 recommendation, such as the XML not being well-formed.

In order to make use of this, we need to pass a reference to the main object to a method on the Parser interface called setErrorHandler. This is the exact analogue of the call to setContentHandler that we used to tell the parser who to send parsing events to.

## Try It Out?Catching Parsing Errors

In this example, we'll extend our BandValidator class so that we catch parser errors and report their location. So, as well as overriding some of DefaultHandler's implementation of ContentHandler, we're going to be overriding some of its implementation of ErrorHandler.

1.

First of all, we need to add another import:

```
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXParseException;
```

2.

Next, let's turn our parser into a validating one. We do this by means of a call to setFeature. This is a method

that permits extensibility in the parser, by allowing you to specify the URI of an optional feature that you want to set. Here's what we do:

```
public void read (String fileName) throws Exception
{
    XMLReader readerObj =
XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");

    try
    {
        readerObj.setFeature("http://xml.org/sax/features/validation", true);
    }

    catch (SAXException e)
    {
        System.err.println("Cannot activate validation");
    }
}
```

3.

Now we tell the parser who to send the error events to:

```
    readerObj.setContentHandler (this);
    readerObj.setErrorHandler (this);
    readerObj.parse (fileName);
}
```

4.

Next, we need to add the three new methods to catch the errors. These will override the minimal implementations provided by DefaultHandler:

```
public void warning(SAXParseException exception) throws SAXException
{
    System.err.println("Warning in " + exception.getSystemId() +
                       " at line " + exception.getLineNumber() + ", column " +
                       exception.getColumnNumber());
}

public void error(SAXParseException exception) throws SAXException
{
    System.err.println("Error in " + exception.getSystemId() +
                       " at line " + exception.getLineNumber() + ", column " +
                       exception.getColumnNumber());
}

public void fatalError(SAXParseException exception) throws SAXException
{
    System.err.println("Fatal error in " + exception.getSystemId() +
                       " at line " + exception.getLineNumber() + ", column " +
                       exception.getColumnNumber());
    throw exception;
}
```

All we're doing here is printing out the location of the error, taken from the incoming SaxParseException object, and then?in the case of fatalError?rethrowing the error back to the parser. It's worth noting that if the parser doesn't support Locator, it's not going to provide us with anything meaningful in these methods on SAXParseException.

5.

Let's go back to our XML and make a couple of changes. First of all, we're going to add a reference to a DTD, and secondly, we're going to make a small but significant change to the last line:

```
<?xml version="1.0"?>
<!DOCTYPE bands SYSTEM "bands.dtd" >
<bands>
```

```
<band type="progressive">
  <name>King Crimson</name>
  <guitar>Robert Fripp</guitar>
  <saxophone>Mel Collins</saxophone>
  <bass>Boz</bass>
  <drums>Ian Wallace</drums>
</band>
<band type="punk">
  <name>X-Ray Spex</name>
  <vocals>Poly Styrene</vocals>
  <saxophone>Laura Logic</saxophone>
  <guitar>Someone else</guitar>
</band>
<band type="classical">
  <name>Hilliard Ensemble</name>
  <saxophone>Jan Garbarek</saxophone>
</band>
<band type="progressive">
  <name>Soft Machine</name>
  <organ>Mike Ratledge</organ>
  <saxophone>Elton Dean</saxophone>
  <bass>Hugh Hopper</bass>
  <drums>Robert Wyatt</drums>
</band>
</bands>
```

## 6.

Here's the DTD (bands.dtd) that we're going to use:

```
<!ELEMENT bands (band*)>

<!ELEMENT band (name, (guitar | organ), saxophone, bass, drums) >
<!ATTLIST band type CDATA #REQUIRED >

<!ELEMENT name      (#PCDATA) >
<!ELEMENT guitar    (#PCDATA) >
<!ELEMENT organ     (#PCDATA) >
<!ELEMENT saxophone (#PCDATA) >
<!ELEMENT bass      (#PCDATA) >
<!ELEMENT drums     (#PCDATA) >
```

## 7.

If you run BandValidator, you should see a much more descriptive output of where the errors have occurred, and what type of error they are. Notice that processing doesn't stop until the final fatal exception "The element type "bands" must be terminated by the matching end-tag "</bands>".

The parser uses the ErrorHandler interface to inform applications of error events. We can therefore catch parser errors by implementing the methods in this interface.

## Other Methods in ContentHandler

There are one or two other methods in the ContentHandler interface that we should look at briefly.

### ignorableWhitespace

The first one is ignorableWhitespace, and for the most part, yes, you can ignore this one. This method has the following definition:

```
public void ignorableWhitespace(char[] chars, int start, int len)
```

```
throws SAXException
```

The sharper-eyed among you will spot that this is very similar to our old friend characters. It's all to do with DTDs, or rather, a particular side-effect of using one. Here's when and why you might want to override this method.

- - A DTD can state that an element can only contain children and not PCDATA
  - 
  - However, the element can still contain whitespaces (spaces, tabs and newlines) for readability
  - 
  - If it does, a validating parser will report these characters by means of an ignorableWhiteSpace event rather than a characters one
  - 
  - A non-validating parser, on the other hand, won't be able to tell whether the characters are ignorable whitespace or PCDATA, so it will report all characters by means of characters events

The upshot of this is that, unless you are particularly interested in reproducing the structure of the document exactly, you can safely ignore ignorableWhiteSpace events.

## processingInstruction

The only other event method in ContentHandler is processingInstruction. Unsurprisingly, this method is used to catch processing instructions, which you'll probably remember from [Chapter 2](#). These are things like:

```
<?someApplication someParam?>
```

that act as instructions for some application processing the XML. The declaration of processingInstruction is as follows:

```
public abstract void processingInstruction(String target, String data)  
throws SAXException
```

In this case, target would be someApplication, and data would be someParam.

I'm sure I don't need to remind you at this point that the XML declaration at the start of an XML document is *not* really a processing instruction, and as such it won't cause you to receive a processingInstruction event. Or at least, if it does, then switch to another parser, quickly.

---

[PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# The Microsoft Way with SAX

Version 3 of the MSXML SDK contains Microsoft's first official implementation of SAX 2.0. It doesn't, incidentally, include an implementation of SAX 1.0, unlike other parsers that have come via the 1.0 route, and still retain the old interfaces, albeit deprecated. If the public domain way of doing things like SAX is to bundle an API up as a set of Java interfaces, then the Microsoft way (at least in the days before .NET gains universal acceptance) is to bundle it up as a set of COM objects. There isn't space here to go into COM in any great detail; for the purposes of this example, you can think of a COM component as just another type of object with interfaces, like the ones we've encountered already. There's more information on COM in the Wrox book *Beginning ATL 3.0 COM programming*, by Richard Grimes et al. ISBN 1-861001-20-7.

In order to try out this SDK, we're going to have to use a Microsoft language and development environment, and I've picked Microsoft Visual Basic 6 for our example. Microsoft Visual C++ would be equally valid, as it supports COM just as well as VB. This section does assume some knowledge of Visual Basic.

What we're going to do next is re-implement the final example from the [previous section](#) as a form-based Visual Basic application. As we do so, we'll find that there are a lot of similarities, and one or two significant differences. First of all, we need to install the Microsoft parser and SDK. This is very straightforward, and all you need to do is follow the instructions.

## Try it Out-A Microsoft SAX Application

1.

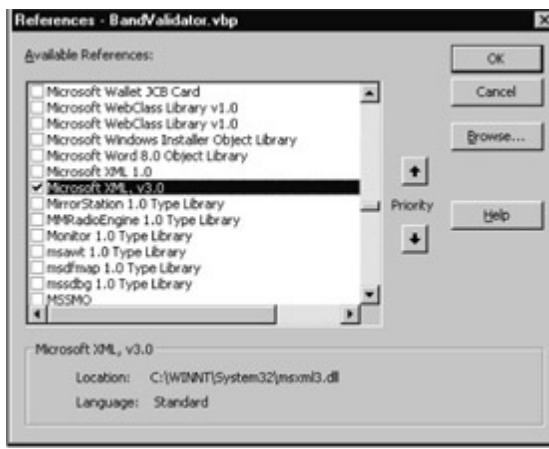
We start off by creating a new standard EXE type Visual Basic project, called (inevitably) BandValidator. This is what the form looks like:



The drive selection box is called Drive1, the directory selection box is called Dir1 and the file selection box is called File1. The big button in the middle is called cmdParse, and the list box underneath it is called lstResults.

2.

Before we do anything else, we need to add a reference to the Microsoft XML Parser DLL to the project. We do this by selecting Project/References from the main menu:



3.

Let's start off coding by adding some simple stuff to the main form class to help navigate to our XML file:

```
Private Sub Dir1_Change()
    File1.Path = Dir1.Path
End Sub

Private Sub Drive1_Change()
    Dir1.Path = Drive1.Drive
End Sub

Private Sub File1_Click()
    cmdParse.Enabled = True
End Sub
```

4.

Next, we need to add the code that gets triggered when we press the magic Parse button. This is what it looks like:

```
Private Sub cmdParse_Click()
    Dim strPath As String
    Dim oContentHandler As ContentHandlerImpl
    Dim oErrorHandler As ErrorHandlerImpl
    Dim oReader As SAXXMLReader

    On Error Resume Next

    strPath = File1.Path + "\" + File1.FileName

    Set oReader = New SAXXMLReader

    Set oContentHandler = New ContentHandlerImpl
    Set oErrorHandler = New ErrorHandlerImpl

    Set oReader.contentHandler = oContentHandler
    Set oReader.errorHandler = oErrorHandler

    Set oContentHandler.m_oReader = oReader

    oReader.parseURL (strPath)
End Sub
```

This is actually very similar to what goes on in the Java version. First of all, having constructed the path of the file we want to analyze, we create an instance of our XML reader object, oReader. Then, we pass it a reference to our content and error handler objects (we'll see what they look like shortly). Skipping over the next line for the time being, we move on to parsing the file itself, using the parseURL method on the reader.

## 5.

Now let's create that class module called ContentHandlerImpl. This is going to be an implementation of the IVBSAXContentHandler interface. As you may have already guessed, this is Microsoft's version of the ContentHandler Java class. We have to implement the interface here, because we can't extend a default implementation in Visual Basic. Unfortunately, we have to implement the entire interface, which means that there are several methods that don't do anything at all. Here's how the code starts:

```
Option Explicit
```

```
Implements IVBSAXContentHandler
```

```
Public m_oReader As IMXReaderControl
Private m_oLocator As IVBSAXLocator
Private m_strSaxophonist As String
Private m_bIsSaxophone As Boolean
Private m_strBandName As String
Private m_bIsName As Boolean
Private m_strBandType As String
```

Most of those member variables should look familiar to you from the Java code, with the exception of the first one, m\_oReader. If you recall, we set that up in the main form class, in this line:

```
Set oContentHandler.m_oReader = oReader
```

So what's going on here? The Microsoft implementation of SAXXMLReader has another interface, called IMXReaderControl, which you can use to control the flow of execution during parsing. IMXReaderControl has three methods: abort, resume, and suspend. We'll see abort in action shortly, but the other two are quite interesting, because they allow for the possibility that the parsing process might have to wait until some external process completes, asynchronously. This is one of the areas where Microsoft has enhanced the original SAX concept.

Here's the code for the startDocument and endDocument methods:

```
Private Sub IVBSAXContentHandler_startDocument()
    Form1.lstResults.AddItem ("Starting ...")
End Sub

Private Sub IVBSAXContentHandler_endDocument()
    Form1.lstResults.AddItem ("... finished")
End Sub
```

And here's startElement:

```
Private Sub IVBSAXContentHandler_startElement(strNamespaceURI As String,
    strLocalName As String, strQName As String, ByVal oAttributes As
    MSXML2.IVBSAXAttributes)
Dim typeAttr As Long

On Error Resume Next

If (strQName = "band") Then
    typeAttr = oAttributes.getIndexFrom QName("type")

    If (Err.Number <> 0) Then
        Form1.lstResults.AddItem ("Error in " + m_oLocator.systemId() + " at line "
+ Str(m_oLocator.lineNumber()) + ", column " + Str(m_oLocator.columnNumber()))
        Form1.lstResults.AddItem ("Band type not specified")
        m_oReader.abort
    End If
End If
```

```
End If

m_strBandType = oAttributes.getValue(typeAttr)

ElseIf (strQName = "name") Then
    m_bIsName = True
    m_bIsSaxophone = False
    m_strBandName = ""

ElseIf (strQName = "saxophone") Then
    m_bIsName = False
    m_bIsSaxophone = True
    m_strSaxophonist = ""

Else
    m_bIsName = False
    m_bIsSaxophone = False
End If
End Sub
```

This is all pretty similar to the Java implementation, except for this line:

```
Form1.lstResults.AddItem ("Error in " + m_oLocator.systemId() + " at
line " + Str(m_oLocator.lineNumber()) + ", column " +
Str(m_oLocator.columnNumber()))
Form1.lstResults.AddItem ("Band type not specified")
m_oReader.abort
```

where, instead of throwing an exception (which we can't do in Visual Basic), we invoke the abort method of our reader control object. Note the use of the locator object, as we did in Java, to identify the location in the file being parsed.

endElement is analogous to the Java version:

```
Private Sub IVBSAXContentHandler_endElement(strNamespaceURI As String,
    strLocalName As String, strQName As String)
If (m_bIsSaxophone) Then
    Form1.lstResults.AddItem ("The saxophonist in " + m_strBandName + " (" +
m_strBandType + ") is " + m_strSaxophonist)
    m_bIsSaxophone = False
End If

If (m_bIsName) Then m_bIsName = False
End Sub
```

As is characters:

```
Private Sub IVBSAXContentHandler_characters(strChars As String)
If (m_bIsName) Then m_strBandName = m_strBandName + strChars

If (m_bIsSaxophone) Then m_strSaxophonist = m_strSaxophonist + strChars
End Sub
```

The only method left that we need to do anything with is the equivalent of setDocumentLocator:

```
Private Property Set IVBSAXContentHandler_documentLocator(ByVal RHS As
    MSXML2.IVBSAXLocator)
Set m_oLocator = RHS
End Property
```

So that's our implementation of ContentHandler complete.

## 6.

Now let's create our other class module, ErrorHandlerImpl. This contains just three methods: one to handle errors, one to handle fatal errors, and one to handle "ignorable warnings" (the idea of a warning being "ignorable" is a Microsoft extension to the specification-go figure). All three implementations are similar here, and this is our entire class module:

```
Option Explicit
```

```
Implements IVBSAXErrorHandler
```

```
Private Sub IVBSAXErrorHandler_error(ByVal oLocator As MSXML2.IVBSAXLocator,  
strErrorMessage As String, ByVal nErrorCode As Long)  
    Form1.lstResults.AddItem ("Error: " + strErrorMessage + " at line " +  
Str(oLocator.lineNumber) + ", column " + Str(oLocator.columnNumber))  
End Sub
```

```
Private Sub IVBSAXErrorHandler_fatalError(ByVal oLocator As MSXML2.IVBSAXLocator,  
strErrorMessage As String, ByVal nErrorCode As Long)  
    Form1.lstResults.AddItem ("Fatal error: " + strErrorMessage + " at line " +  
Str(oLocator.lineNumber) + ", column " + Str(oLocator.columnNumber))  
End Sub
```

```
Private Sub IVBSAXErrorHandler_ignorableWarning(ByVal oLocator As MSXML2.IVBSAXLocator,  
strErrorMessage As String, ByVal nErrorCode As Long)  
    Form1.lstResults.AddItem ("Warning: " + strErrorMessage + " at line " +  
Str(oLocator.lineNumber) + ", column " + Str(oLocator.columnNumber))  
End Sub
```

And that's our application complete. Let's try running it:



So that's pretty cool. What if we tinker with the XML a bit to make it invalid, by removing the </guitar> from Robert Fripp?



That's our implementation of IVBSAXErrorHandler coming into its own.

What happens if we remove the band type attribute from King Crimson?



That's the abort method on the reader control object stopping the parser process.

So it seems that we can do pretty much anything in Microsoft SAX that we can do in Java SAX. Actually, there are a number of things that Microsoft can do that Java can't. For instance, although the SAX specification doesn't mandate it, Microsoft's implementation also has an XML *writer* interface, which makes it a great tool for carrying out complex transformations.

# Good SAX and Bad SAX

Now that we're thoroughly familiar with SAX, this is a good point to review what SAX is good at and what it isn't so good at, so that we can decide when to use it and when to use another approach, such as the DOM.

As we've seen, SAX is great for analyzing and extracting content from XML documents. Let's look at what makes it so good:

- It's simple: you only need to implement three or perhaps four interfaces to get going.
- It doesn't load the whole document into memory, so it doesn't take up vast amounts of space. Of course, if your application is using SAX to build up its own in-memory image of the document, it's likely to end up taking a similar amount of space as the DOM would have done (unless your image is a lot more efficient than the DOM!).
- The parser itself typically has a smaller footprint than that of its DOM cousin. In fact, DOM implementations are often built on top of SAX.
- It's quick, because it doesn't need to read in the whole document before you start work on it.
- It focuses on the real content rather than the way that it's laid out.
- It's great at filtering data, and letting you concentrate on the subset that you're interested in.

So why don't we use it for everything, then? Here are a few drawbacks:

- You get the data in the order that SAX gives it to you. You have absolutely no control over the order in which the parser searches. As we have seen in our Try It Outs, this means that you may need to build up the data that you need over several event invocations. This can be a problem if you're doing particularly complex searches.
- SAX programming requires fairly intricate state keeping, which is prone to errors.
- If you're interested in analyzing an entire document, DOM is much better, because you can traverse your way around the DOM in whichever direction you want, as many times as you want.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

SAX is an excellent API for analyzing and extracting information from large XML documents without incurring the time and space overheads associated with the DOM.

In this chapter, we learnt how to use SAX to catch events passed to us by a parser, by implementing a known SAX interface, ContentHandler. We used this to extract some simple information from an XML document.

We also looked at error handling, and found out how to implement sophisticated intelligent parsing, reporting errors as we did so. In addition we looked at how to supplement the error handling mechanisms in the parser by using the Locator object.

We compared the new Microsoft implementation of SAX with the traditional Java approach, and found some interesting similarities and differences.

Finally, we discussed the strengths and weaknesses of SAX.

Stay with us as we find out about SOAP.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 10: **SOAP**

## Overview

So far we've covered what XML is, how to create well-formed and valid XML documents, and even seen a couple of programmatic interfaces into XML documents, namely the DOM and SAX. We also discussed the fact that XML isn't really a language on its own; it's a meta-language, to be used in the creation of other languages.

This chapter will take a slightly different turn. Rather than discussing XML itself, we'll be discussing an application of XML: the **Simple Object Access Protocol**, or **SOAP**. SOAP is a protocol that allows objects on one computer to call and make use of objects on other computers. In other words, SOAP is a means of performing distributed computing.

In this chapter we'll learn:

- What a **Remote Procedure Call (RPC)** is, and what RPC protocols exist currently
- Why SOAP can provide more flexibility than previous RPC protocols
- Why most SOAP implementations would want to use HTTP as a transport protocol, and how HTTP works under the hood
- How to format SOAP messages
- What **Web Services** are

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Running the Examples

In order to run some of the examples in this chapter, you will need Internet Information Server (IIS), or Personal Web Server (PWS) installed on your machine.

The good news is that if you're running Windows, with the exception of Windows 3.1 and earlier versions, you can install Microsoft's **Personal Web Server (PWS)** for free. The downside is that, while PWS is fine for internal intranet web sites, it's definitely not suitable for high-use Internet websites as it's not robust enough or secure enough. However, it is great for developing server-side script before deploying it to the full Windows web server called **Internet Information Services (IIS)**?this is supplied with Windows NT Server and Windows 2000 editions.

The steps needed to install a web server on Windows depend on which operating system is being used. Windows 2000 Server and Advanced Server users will already have IIS installed by default. If you're running Windows 2000 Professional, then note that IIS comes with the operating system, but is not installed by default. Instead, you'll need to go to Control Panel, Add/Remove programs and select Add/Remove Windows Components, under which you'll find the option to install Internet Information Services (IIS). (You can load IIS onto Windows 2000 Server and Advanced Server the same way, if the person who installed the operating system didn't include it.)

Windows NT 4 Server and Workstation users will need to install **NT Option Pack**. For Server users, this installs the full IIS server version, while Workstation users will have the PWS version installed. The NT Option pack can be obtained both on CD and from the Microsoft web site at:

<http://www.microsoft.com/ntserver/nts/downloads/recommended/NT4OptPk/default.asp>

Finally, Windows 98 users will find PWS on the Windows disk.

We'll look next at all the steps necessary to install PWS on a Windows 98 machine. You should find that most of the steps are similar on other versions of Windows, with just a few minor variations.

## Installing PWS On Windows 98

The set up files for PWS are on the Windows 98 CD in the directory \add-ons\pws\.

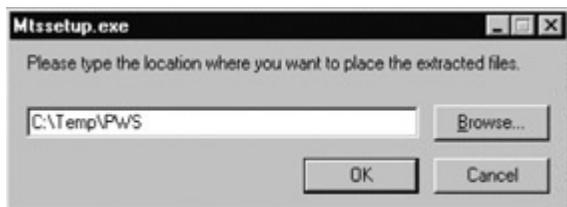
We need to copy these files from the PWS directory on the CD to the local hard drive?the C:\Temp directory would be a good place.

Due to changes in Windows 98 operating system, one of the files, MtsSetup.dll, is out-of-date and needs to be replaced with one downloaded from the Microsoft web site. (This step only needs to be performed for Windows 98 users.) This file can be downloaded from:

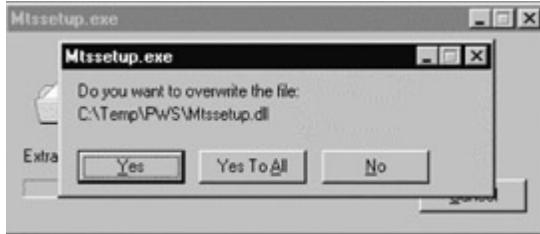
<http://download.microsoft.com/download/transaction/Patch/1/W95/EN-US/Mtssetup.exe>

Save the executable file to your hard drive in the C:\Temp directory, and then run it. First, the license agreement will appear. Read it, and if you agree, click Yes to start the install.

Next, in the text box that appears, either enter or browse to the directory into which the PWS directory was copied from the CD, then click OK.



When the box shown below pops up asking if you want to overwrite the existing Mtsetup.dll file with the new one you just downloaded, click Yes.



That completes the update. Now we can start the PWS installation process by running setup.exe in C:\Temp\PWS from Run on the Start menu.

We should then see an introductory screen. Click the Next button and we'll be given a choice of whether we want a Minimum, Typical or Custom install. Unless hard drive space is at a premium (that is, you have less than 50 Mb free) then go for the Typical install, which requires about 32 Mb of hard drive space.

In the next screen, we are allowed to choose the directory where the web directories and web pages for the server will be stored. The default directory is fine unless you have limited space on your C: drive, in which case any hard drive is acceptable. (If you're running Windows 2000, you will not be given this option, and the default directory will be used.) We're not given the option of selecting an FTP or application directory, since the Windows 98 PWS does not support either of these. Once the directory is selected, click Next and the PWS installation will commence.



Once the install has finished, click Finish and re-start the computer.

## Managing PWS Under Windows 98

Web servers are a little different to a normal program. Normally, you load or start a program by clicking its icon. When you've stopped using it, you close it down. However, web servers usually start when the operating system starts, unless you specify otherwise. They run in the background, and are termed a service rather than a normal program.

PWS comes with its own basic console, which gives us control over the web service. For example, if we want to stop the service we use the console's Stop button. To start it running again, we need to start it using the Start button in the console. Note that when the service is stopped, any attempts to load web pages from the server will fail. The

console also gives us the ability to create new web directories and check usage statistics, such as how many visitors per day have visited our site.

After PWS has installed, the PWS tray icon, shown here, will be placed in the Windows task bar at the bottom of your desktop.

Double-clicking this will open up the PWS console. Alternatively, you can also access the console from an option on the Start menu. Windows 2000 users can find the Personal Web Manager under Administrative Tools in the Control Panel. NT4 users will find it under Start, Windows NT 4.0 Option Pack, Microsoft Internet Information Server, Internet Service Manager. Note that this may vary with your operating system.

Open up the PWS console now.



From the main section of the console, we can glean a lot of information. First, below the line "Web publishing is on. Your home page is available at:" we can see the name given to the server. In my case, the name is set automatically as <http://Paul>. If you click on this <http://myservername> link, where myservername is the server name for your web server, it will open up your default browser with the current home page for the server.

Underneath this, we are told what the physical path for the root directory (labeled home directory) of our server is. If you selected the default options when PWS was installed, then, like mine shown above, yours will be <C:\Inetpub\wwwroot>.

If we put a file in the root directory of our server called myPage.htm, then typing:

<http://myservername/myPage.htm>

into our browser will load the myPage.htm page. Create a new directory under the root directory, <C:\Inetpub\wwwroot>, called SOAP. This is the directory you will be using for the examples in this chapter. If you were to put a file called myPage2.htm in there, then typing:

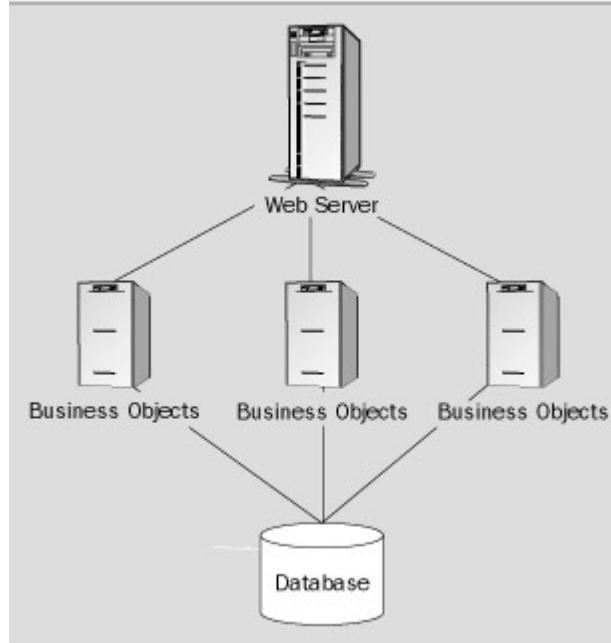
<http://myservername/SOAP/myPage2.htm>

into your web browser would cause myPage2.htm in SOAP to be loaded.

In addition, if the web server is on the same machine as the browser, then you can type <http://localhost> instead of <http://myservername>.

# What Is an RPC?

It is often necessary to design **distributed systems**, where the code to run an application is spread across multiple computers. For example, to create a large transaction processing system, you might have separate servers for business logic objects, one for presentation logic objects, a database server, etc., which all need to talk to each other.



In order for a model like this to work, code on one computer needs to call code on another computer. For example, the code in the web server might need a list of orders, for display on a web page, in which case it would call code in the Business Objects to provide that list of orders. That code in turn might need to talk to the database. When code on one computer calls code on another computer, this is called a **Remote Procedure Call**, usually abbreviated **RPC**.

In order to make an RPC, you need to know a number of things:

- Where does the code you want to call reside? If you want to execute a particular piece of code, you need to know where that code is!
- Does the code need any parameters? If so, what type of parameters? For example, if you want to call a remote procedure to add two numbers together, that procedure would need to know what numbers it's adding.
- Will the procedure return any data? If so, in what format? For example, a procedure to add two numbers would probably return a third number, which would be the result of the calculation.

In addition, you need to deal with networking issues, packaging any data for transport across the wire, and a number of other issues. For this reason, a number of RPC **protocols** have been developed.

### Important

A **protocol** is a set of rules that allow different applications, or even different computers, to communicate. For example, **TCP** (Transmission Control Protocol) and **IP** (Internet Protocol) are protocols that allow computers on the Internet to talk to each other, because they set up rules about how data should be passed, how computers are addressed, etc.

These protocols specify how to provide an address for the remote computer, how to package up data to be sent to the remote procedures and how to get any return data back, how to initiate the call, how to deal with errors, and all of the other details that need to be addressed to allow multiple computers to communicate with each other. (Such RPC protocols often piggy-back on other protocols; for example, an RPC protocol might specify that TCP/IP must be used as its network transport.)

## What RPC Protocols Exist?

There are a number of protocols that exist for performing remote procedure calls, but the most common are **DCOM** and **IIOP** (both of which are extensions of other technologies: **COM** and **CORBA**, respectively), and **Java RMI**. Each of these protocols provides the functionality that you need to perform remote procedure calls, although each also has its drawbacks.

The following sections will discuss these protocols, and those drawbacks, although we won't really go too much into technical details. We'll just take it for granted that these protocols provide the functionality needed to perform remote procedure calls.

### DCOM

Microsoft developed a technology called the **Component Object Model**, or **COM** (see <http://www.microsoft.com/com/default.asp>), to help facilitate **component-based software**. That is, software that can be broken down into smaller, separate, components, which can then be shared across an application, or even across multiple applications. COM provides a standard way of writing objects so that they can be discovered at run-time, and used by any application running on the computer. What's more, COM objects are language-independent. That means that you can write a COM object in virtually any programming language, C, C++, Visual Basic, etc., and that object can talk to any other COM object, even if it was written in a different language.

A good example of COM in action is Microsoft Office: because much of the functionality in Office has been provided through COM objects, it is easy for one Office application to make use of another. For example, since Excel's functionality is exposed through COM objects, we might create a Word document that contains an embedded Excel spreadsheet.

However, this functionality is not limited to Office applications; we could also write our own application that makes use of Excel's functionality to perform complex calculations, or uses Word's spell checking component. This would allow me to write my applications quicker, since I wouldn't have to write the functionality for a spell-checking component or a complex math component myself. By extension, we could also write our own shareable components for use in others' applications.

COM is a handy technology to use when creating reusable components, but it doesn't tackle the problem of distributed applications. In order for your application to make use of a COM object, that object must reside on the same computer as your application. For this reason, Microsoft developed a technology called **Distributed COM**, or **DCOM** (see <http://www.microsoft.com/com/tech/dcom.asp>). DCOM extends the COM programming model, and allows applications to call COM objects that reside on remote computers. To an application, calling a remote object from a server using DCOM is just as easy as calling a local object on the same PC using COM?as long as the

necessary configuration has been done ahead of time.

However, as handy as COM and DCOM are for writing component-based software and distributed applications, they have one major drawback: both of these technologies are Microsoft-specific. The COM objects we write, or that we want to use, will only work on computers running Microsoft Windows. And, even though we can call remote objects over DCOM, those objects also must be running on computers running Microsoft Windows.

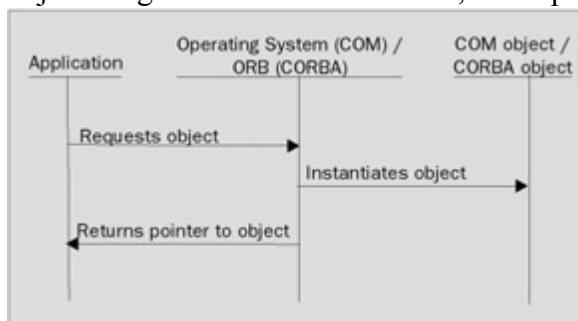
*There have been DCOM implementations written for other, non-Microsoft, operating systems, but they haven't reached large acceptance. In practice, when someone wants to develop a distributed application on non-Microsoft platforms, they use one of the other RPC protocols.*

For some people, this may not be a problem. For example, if I'm developing an application for my company, and we have already standardized on Microsoft Windows for our employees, then using a Microsoft-specific technology might be fine. For others, however, this limitation means that DCOM is just not an option.

## IIOP

Prior even to Microsoft's work on COM, the **Object Management Group (OMG)** (<http://www.omg.org/>) had developed a technology to solve the same problems that COM and DCOM try to solve, but in a platform-neutral way. They called this technology the **Common Object Request Broker Architecture**, or **CORBA** (<http://www.corba.org/>). Just as with COM, CORBA objects can be written in virtually any programming language, and any CORBA object can talk to any other, even if it was written in a different language. CORBA works similarly to COM, the main difference being who supplies the underlying plumbing for the technology.

For COM objects, the underlying COM functionality is provided by the operating system (Windows), whereas with CORBA, an **Object Request Broker**, or **ORB**, provides the underlying functionality. For example, to instantiate an object using either COM or CORBA, the steps are similar:



While the concepts are the same, using an ORB instead of the operating system to provide the base object services gives one important advantage: it makes CORBA platform-independent. Any vendor who creates an ORB can create versions for Windows, Unix, Linux, etc.

Furthermore, the OMG created the **Internet Inter-ORB Protocol (IIOP)**, which allows communication between different ORBs. This means that you not only have platform-independence, you also have ORB independence, as you can combine ORBs from different vendors, and have remote objects talking to each other over IIOP (as long as you stay away from any vendor-specific extensions to IIOP).

## Java RMI

Both DCOM and IIOP provide very similar functionality: they provide a language-independent way to call objects that reside on remote computers. IIOP goes a step further than DCOM, and allows for components to be run on different platforms. However, there is already a language that is specifically designed to be "write once, run anywhere": Java.

Java provides the **Remote Method Invocation (RMI)**, see <http://java.sun.com/products/jdk/rmi/>) system for distributed computing. Since Java objects can be run from any platform, the idea behind RMI is to just write

everything in Java, and then have those objects communicate with each other.

Although Java can be used to write CORBA objects that can be called over IIOP, or even to write COM objects using certain non-standard Java language extensions, using RMI for your distributed computing can provide a smaller learning curve because the programmer isn't required to learn about CORBA and IIOP. Since all of the objects involved are using the same programming language, any data types are simply the built-in Java data types, and Java exceptions can be used for error handling. Finally, there is one ability in RMI that isn't available in DCOM or IIOP: it can transfer code with every call. That is, even if the remote computer we're calling doesn't have the code it needs, we can send it, and yet still have the remote computer perform the processing.

However, the obvious drawback to Java RMI is that it ties the programmer to one programming language, Java, for all of the objects in the distributed system.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# The New RPC Protocol: SOAP

To these existing protocols, we can now add one more: the **Simple Object Access Protocol**, or **SOAP**. According to the current SOAP specification it is "a lightweight protocol for exchange of information in a decentralized, distributed environment". In other words, it is a standard way to send information from one computer to another using XML to represent the information.

*At the time of writing the current version 1.1 of SOAP can be found at <http://www.w3.org/TR/SOAP/>. Version 1.2 is still at Working Draft stage at the W3C, and can be viewed at <http://www.w3.org/TR/soap12/>.*

In a nutshell, the SOAP specification defines a protocol where all information sent from computer to computer is marked up in XML, and the information is usually transmitted via HTTP.

*Technically, SOAP messages don't have to be sent via HTTP. Any networking protocol, such as SMTP or FTP, could be used. However, in practice HTTP will probably be the most common protocol used for SOAP. We'll be discussing the reasons for this later.*

Let's take a look at some of the advantages of SOAP over the other protocols we've discussed so far:

- It's platform-, language-, and vendor-neutral. Because SOAP is implemented using XML and (usually) HTTP, it is easy to process and send SOAP requests in any language, on any platform, without having to depend on tools from a particular vendor.
- It's easy to implement. SOAP was designed to be less complex than the other protocols?hence the word "simple" in its name. A SOAP server could be implemented using nothing more than a web server and an ASP page, or a CGI script.
- It's firewall safe. Assuming that you use HTTP as your network protocol, SOAP messages can be passed across a firewall, without having to perform extensive configuration. However, if a firewall administrator does want to filter out SOAP messages, there is a way to do that as well, which we'll discuss later.

## Try It Out?Creating an RPC Server in ASP

Throughout this chapter, we're going to build a remote procedure that will be available via SOAP. I've chosen a very simple procedure to write, one that simply adds two numbers together and returns the result, because what the service does isn't as important to us as how to build the SOAP plumbing, which we'll add bit-by-bit over the course of the chapter.

To start with, we're going to create a simple ASP page, which accepts two numbers, adds them, and returns the results in XML.

1.

To begin with, we need to create our ASP page. Call your page addNumbers1.asp. (The "1" is there because we'll be creating new versions of this page as we go, to add in more and more SOAP functionality.)

2.

This page will pull information from the **Query String**. This is information that is appended to the end of a

URL. For example, in the URL:

<http://localhost/SOAP/addNumbers1.asp?numOne=12&numTwo=12>

the URL itself is "<http://localhost/SOAP/addNumbers1.asp>", while "?numOne=12&numTwo=12" is the query string. The ASP object model makes getting this information easy. The main code for the page will be as follows:

```
Response.ContentType = "text/xml"

Dim sNumber1, sNumber2
Dim nResult

sNumber1 = Request.QueryString.Item("numOne")
sNumber2 = Request.QueryString.Item("numTwo")

On Error Resume Next
nResult = CLng(sNumber1) + CLng(sNumber2)

If Err.number <> 0 Then
    Response.Write ErrorXML()
Else
    Response.Write SuccessXML(nResult)
End If
```

We've set the content type of our result to "text/xml", which will indicate to the browser calling this page that the results will be XML. We've then pulled the information from the query string, and stored it in a couple of variables (sNumber1 and sNumber2). Finally, we added those two numbers together, checking to make sure that there is no error. If there is an error, we call a function called ErrorXML(), and send the results to the output. Otherwise, we call a function called SuccessXML(), passing it the results of our addition, and send that to the output.

### 3.

Finally, we just need to add those two functions, ErrorXML() and SuccessXML().

```
Function ErrorXML()
Dim sXML

sXML = "<Error><Reason>Invalid numbers</Reason></Error>"

ErrorXML = sXML
End Function

Function SuccessXML(sResult)
Dim sXML

sXML = "<addNumbers>" & vbCrLf
sXML = sXML & "<result>" & sResult & "</result>" & vbCrLf
sXML = sXML & "</addNumbers>"

SuccessXML = sXML
End Function
```

### 4.

Your final ASP page will look like this:

```
<%@ Language=VBScript %>
<%
Response.ContentType = "text/xml"
```

```
Dim sNumber1, sNumber2
Dim nResult

sNumber1 = Request.QueryString.Item("numOne")
sNumber2 = Request.QueryString.Item("numTwo")

On Error Resume Next
nResult = CLng(sNumber1) + CLng(sNumber2)

If Err.number <> 0 Then
    Response.Write ErrorXML()
Else
    Response.Write SuccessXML(nResult)
End If

Function ErrorXML()
    Dim sXML

    sXML = "<Error><Reason>Invalid numbers</Reason></Error>"

    ErrorXML = sXML
End Function

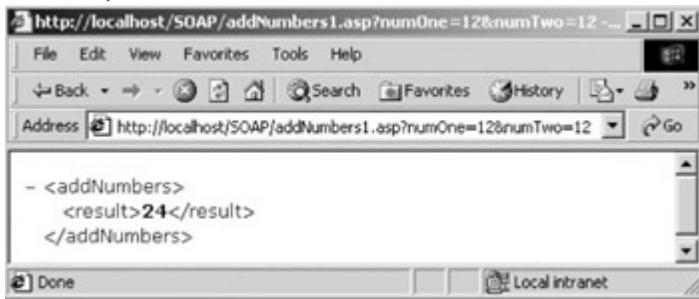
Function SuccessXML(sResult)
    Dim sXML

    sXML = "<addNumbers>" & vbCrLf
    sXML = sXML & "<result>" & sResult & "</result>" & vbCrLf
    sXML = sXML & "</addNumbers>"

    SuccessXML = sXML
End Function
%>
```

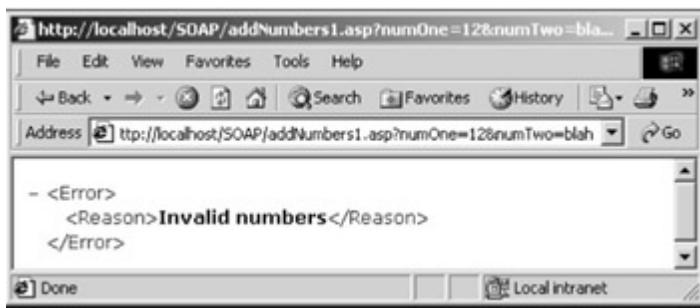
## 5.

To try this page out, open a browser, and type in the URL, appending the query string numOne=12&numTwo=12. For example, on my machine that URL is the one shown earlier: <http://localhost/SOAP/addNumbers1.asp?numOne=12&numTwo=12>, since the addNumbers1.asp file is in a SOAP subdirectory of my InetPub/wwwroot directory, which is where the web server looks for web files. On your machine the URL might be slightly different, although the query string will be the same. In your browser, the results should look like:



## 6.

Now try it again, but this time pass a non-numeric value for one of the variables you're passing. For example, if you use <http://localhost/SOAP/addNumbers1.asp?numOne=12&numTwo=blah> you will get the following results:



## How It Works

This page simply pulls two values from the query string, casts them as numbers, and adds them together. The results are returned as XML. If either of the two values isn't numeric, meaning that they can't be added together, a different XML document is written back to the client, indicating that there was a problem.

Note that this ASP page isn't limited to being called from a browser. For example, we could use the `load()` method of MSXML to call this ASP page, and get back the results, similar to this example in Visual Basic:

```
Sub Main()
    Dim xdDoc as MSXML.DOMDocument

    Set xdDoc = New MSXML.DOMDocument
    xdDoc.async = False
    xdDoc.Load "http://localhost/SOAP/addNumbers1.asp?numOne=12&numTwo=12"

    If xdDoc.documentElement.nodeName = "Error" Then
        MsgBox "Unable to add numbers together"
    Else
        MsgBox xdDoc.selectSingleNode("/addNumbers/result").Text
    End If

    Set xdDoc = Nothing
End Sub
```

We pass a URL, including the query string, to `load()`, making sure that we get the results synchronously (in other words, we wait for a response from the server before continuing), and then check the results. If the root element is named "Error", then we know something went wrong, otherwise we can get the results out of our XML.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# How SOAP Works

As we mentioned before, SOAP messages are basically XML documents, sent across HTTP. SOAP specifies:

- Rules on how the RPC should be sent. Although the SOAP specification says that any network protocol can be used, there are specific rules included in the specification for HTTP, since that's the protocol most people will probably use.
- The overall structure of the XML that is sent. This is called the **envelope**. Any information to be sent back and forth over SOAP is contained within this envelope.
- Rules on how data is represented in this XML. These are called the **encoding rules**.

The following sections will go into more detail on these concepts.

## The Network Transport

As mentioned earlier, the SOAP specification allows any network transport to be used to send the SOAP messages. For example, you could use IBM MQSeries or Microsoft Message Queue (MSMQ) to send SOAP messages asynchronously over a queue, or even use SMTP to send SOAP messages via e-mail. However, the most common protocol used will probably be HTTP, which is what we'll concentrate on in this chapter.

### HTTP

Many readers may already be somewhat familiar with the HTTP protocol, since it is used every time you request a web page in your browser. Since most SOAP implementations will use HTTP as their underlying protocol, we should take a look at how it works under the hood.

The **Hypertext Transfer Protocol, HTTP**, is a **request/response** protocol. This means that when you make an HTTP request, at its most basic, the following steps occur:

- A connection to the HTTP server is opened
- A request is sent to the server
- Some processing is done by the server
- A response from the server is sent back
- The connection is closed

An HTTP message contains two parts: a set of **headers**, followed by an optional **body**. The headers are simply text,

where each header is separated from the next by a new line, while the body might be text or binary information. The body is separated from the headers by two new lines.

For example, suppose I attempt to load an HTML page, located at <http://www.sernaferna.com/samplepage.html>, into my browser. My browser, in this case Internet Explorer 5.5, would send a request similar to the following to the [www.sernaferna.com](http://www.sernaferna.com) server:

```
GET /samplepage.htm HTTP/1.1
Accept: /*
Accept-Language: en-ca
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.sernaferna.com
```

The first line of our request specifies the method that is to be performed by the HTTP server. HTTP defines a few types of request, but we have specified "GET", indicating to the server that we want to get the resource indicated, which in this case is "/samplepage.html". (Another common method is "POST", which we'll discuss in a moment.) This line also specifies that we're using the HTTP/1.1 version of the protocol. There are a number of other headers there as well, which specify to the web server a few pieces of information about the browser, like what types of information it can receive, which are

- Accept, which tells the server what MIME types this browser accepts. (In this case, "\*/\*", meaning any MIME types.)
- Accept-Language, which tells the server what language this browser is using. Servers can potentially use this information to customize the content sent back. In this case, the browser is specifying that it is the Canadian (ca) dialect of the English (en) language.
- Accept-Encoding, which specifies to the server if the content can be encoded before being sent to the browser. In this case, the browser has specified that it can accept documents that are encoded using gzip or deflate.

For a GET, there is no body in the HTTP request message.

In response, the server would send something similar to the following:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Fri, 06 Jul 2001 15:30:52 GMT
Content-Type: text/html
Last-Modified: Thu, 05 Jul 2001 15:19:57 GMT
Content-Length: 98
```

```
<html>
<head><title>Hello world</title></head>
<body>
<p>Hello world</p>
</body>
</html>
```

Again, there is a set of HTTP headers, followed by the body. In this case, some of the headers sent by the HTTP server were:

-

A status code, 200, to indicate that the request was successful. The HTTP specification (<ftp://ftp.isi.edu/in-notes/rfc2616.txt>) defines a number of valid status codes that can be sent in an HTTP response, such as the famous (or infamous) 404 code, which means that the resource being requested could not be found.

- A Content-Type header, indicating what type of content is contained in the body of the message. A client application (such as a web browser) would use this header to decide what to do with the item; for example, if the content type were that of a .wav file, the browser might load an external program to play it, or give the user the option of saving it to the hard drive instead.
- A Content-Length header, which indicates how long the body of the message will be.

The GET method is the most common HTTP method used in regular every-day surfing. The second most common is the POST method. When you do a POST, information is sent to the HTTP server in the body of the message. For example, when you fill out a form on a web page, and click the submit button, the web browser will usually POST that information to the web server, which processes it before sending back the results.

Suppose we create an HTML page, which includes a form, like this:

```
<html>
<head>
<title>Test form</title>
</head>
<body>
<form action="acceptform.asp" method="POST">
Enter your first name: <input name="txtFirstName"><br>
Enter your last name: <input name="txtLastName"><br>
<input type='submit'>
</form>
</body>
</html>
```

This form will POST any information to a page called acceptform.asp, in the same location as this HTML file, similar to the following:

```
POST /acceptform.asp HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, */*
Referer: http://www.sernaferna.com/myform.htm
Accept-Language: en-ca
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.sernaferna.com
Content-Length: 33

txtFirstName=John&txtLastName=Doe
```

So, while the GET method provides for surfing the Internet, it's the POST method that allows for things like e-commerce, since information can be passed back and forth.

## Why HTTP for SOAP?

Earlier we mentioned that most SOAP implementations would probably use HTTP as their transport. Why is that? Here are a few reasons:

-

HTTP is already a widely implemented, and well understood, protocol

- 
- The request/response paradigm lends itself to RPC well
- 
- Most firewalls are already configured to work with HTTP
- 
- HTTP makes it easy to build in security, with **Secure Sockets Layer (SSL)**

## **Widely Implemented**

One of the primary reasons for the explosive growth of the Internet was the availability of the World Wide Web, which runs over the HTTP protocol. There are millions of web servers in existence, serving up HTML and other content over HTTP, and many, many companies using HTTP for e-commerce.

HTTP is a relatively easy protocol to implement, which is one of the reasons that the Web works as smoothly as it does. If HTTP had been hard to implement, a number of implementers would have probably gotten it wrong, meaning that some web browsers wouldn't have worked with some web servers.

Using HTTP for SOAP implementations will therefore be easier than other network protocols would have been. Alternatively, SOAP implementations can piggy-back on existing web servers; in other words, use their HTTP implementation. This means that you don't have to worry about the HTTP implementation at all!

## **Request/Response**

Most of the time, when a client is making an RPC call, it is going to need some kind of response back. For example, if we make a call to our remote procedure that adds numbers together, we would need the result of the calculation back, or it wouldn't be a very useful procedure to call! In other instances, we may not need data back from the RPC call, but we may still need confirmation back that the procedure executed successfully. For example, if we are submitting an order to a back-end database, there may not be data to return, but we should know whether the submission failed or succeeded.

HTTP's request/response paradigm lends itself easily to this type of situation. For our "addition" remote procedure, we

- 
- Open a connection to the server providing the SOAP service
- 
- Send the numbers which need to be added together
- 
- Process the numbers
- 
- Get back the result
- 
- Close the connection

In actual fact, the SOAP specification says that SOAP messages are one-way, instead of two-way. This would mean that there would have to be two separate messages sent: one from the "client" to the "server" with the numbers to add,

and one from the "server" back to the "client" with the result of the calculation. Luckily, the SOAP specification also says that when a request/response protocol, such as HTTP, is used, these two messages can be combined in the request/response of the protocol.

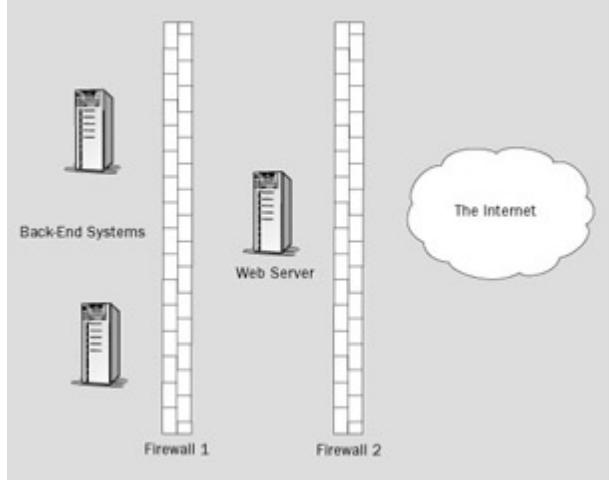
## Firewall-Ready

Most companies protect themselves from outside hackers by placing a **firewall** between their internal systems and the external Internet.

### Important

**Firewalls** are designed to protect a network by blocking certain types of network traffic. Most firewalls allow HTTP traffic (the type of network traffic which would be generated by browsing the Web), but disallow other types of traffic.

These firewalls protect the company's data, but they also make it more difficult to provide web-based services to the outside world. For example, consider a company that is selling goods over the Web. This web-based service would need certain information, like what items are available in stock, which it would have to get from the company's internal systems. In order to provide this service, the company would probably have to create an environment such as the following:



This is a very common configuration, in which the web server is placed between two firewalls. (This section, between the two firewalls, is often called a **Demilitarized Zone**, or **DMZ**) Firewall 1 protects the company's internal systems, and must be carefully configured to allow the proper communication between the web server and the internal systems, without letting any other traffic get through. Firewall 2 is configured to let traffic through between the web server and the Internet, but no other traffic.

This protects the company's internal systems, but makes it a bit more difficult for the developers who are creating this web-based service, because of the added complexity added by these firewalls-especially for the communication between the web server and the back-end servers. However, because firewalls are configured to let HTTP traffic go through, it is much easier to provide the necessary functionality if all of the communication between the web server and the other servers uses this protocol

*This isn't just limited to SOAP, either. Some of the RPC protocols discussed earlier can also use HTTP as a network transport.*

## Security

Because there is already an existing security model for HTTP, the **Secure Sockets Layer (SSL)**, it is very easy to make transactions over HTTP secure. In fact, SSL is so common that there are even hardware accelerators available, to speed up SSL transactions!

## Using HTTP for SOAP

Using HTTP for SOAP messages is very easy. There are only two things you need to do with the client:

- For the HTTP method, use POST. The body of the message will be an XML document, describing the SOAP request (described next).
- Add an extra HTTP header, SOAPAction, which specifies what remote procedure is being called.

In addition, the SOAP specification states that the server must return a proper HTTP status code, in the HTTP headers it returns to the client. So, if the SOAP request is not successful, the SOAP server must return a status code of 500 (this is an HTTP "internal server error"), and if the request is successful, the SOAP server must return an HTTP success status code (usually 200, although there are a number of success status codes defined in the HTTP specification).

For example, consider the following:

```
POST /soap.asp HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword, /*
Accept-Language: en-ca
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.sernaferna.com
Content-Length: 242
SOAPAction: "http://www.sernaferna.com/soap/addNumbers#addNumbers"

<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <m:addNumbers xmlns:m="http://www.sernaferna.com/soap/addNumbers/">
      <numOne>12</numOne>
      <numTwo>24</numTwo>
    </m:addNumbers>
  </Body>
</Envelope>
```

Don't worry about the format of the XML yet—we'll cover that next.

The SOAPAction header indicates the remote procedure being called, using a URI. In some cases, the URL of the HTTP request is a unique value, which specifies the remote procedure being called, in which case an empty string can be used for the SOAPAction, like this:

```
SOAPAction: ""
```

Or, if you wish, you can even leave the SOAPAction completely blank, like this:

```
SOAPAction:
```

meaning that you aren't specifying the SOAP remote procedure at all in the HTTP headers. In this case, the SOAP server would have to look at the XML in the body of the request, to determine which remote procedure is being called.

The SOAPAction header can also be used to filter SOAP messages that are passing through a firewall. The firewall administrator could configure the firewall to look at this header, and only let through messages destined for particular URIs, or even filter out all SOAP traffic. Without this header, the only way to filter the traffic would be to look at the body of every HTTP POST, trying to figure out if the message is intended for an unauthorized SOAP service, which could significantly slow down traffic going through the firewall.

However, since the SOAPAction header is not mandatory, at best it can be considered a "hint" to the server that will process the SOAP message. You should not count on it being present.

## Try It Out-Using HTTP POST to Call Our RPC

Instead of passing our information via the query string, we can also pass information to the server in an HTTP POST. This would be most helpful when passing complex information, rather than just a couple of numbers, as we've been doing. In fact, for really complex data, we could even use XML to pass the information!

1.

Create a new ASP page, called addNumbers2.asp. You can copy and paste in the code from addNumbers1.asp, since most of the code will remain the same.

2.

The big change that will be in this page is that the information will now be loaded from the HTTP POST body, instead of from the query string. We will pass a simple XML document through the POST body, which looks like this:

```
<addNumbers>
<numOne>12</numOne>
<numTwo>24</numTwo>
</addNumbers>
```

In order to do this, we'll use MSXML on the server to pull that XML out of the POST body, and then get the information that we need.

Modify the code in your ASP page as follows (the changed code is highlighted):

```
Response.ContentType = "text/xml"
Response.AddHeader "SOAPAction", "addNumbersResult"

Dim xdRequestXML
Dim sNumber1, sNumber2
Dim nResult

Set xdRequestXML = Server.CreateObject("MSXML.DOMDocument")
xdRequestXML.load Request

sNumber1 = xdRequestXML.selectSingleNode("/addNumbers/numOne").text
sNumber2 = xdRequestXML.selectSingleNode("/addNumbers/numTwo").text

On Error Resume Next
nResult = CLng(sNumber1) + CLng(sNumber2)

If Err.number <> 0 Or Request.ServerVariables("HTTP_SOAPAction") <> "addNumbers" Then
    Response.Write ErrorXML()
Else
    Response.Write SuccessXML(nResult)
End If
```

Notice that the bulk of the code remains the same, except that we are getting the information from the POST body's XML, and we're filtering for this one particular RPC call, using the SOAPAction header. (In IIS, we get HTTP headers from the ServerVariables collection-but we have to add "HTTP\_" to the beginning of the header name, because it's a custom header.) Also, we're sending a SOAPAction HTTP header back to the client, since the return is also a SOAP message. In the next Try It Out, we'll use better URIs for the SOAPAction headers.

Keep the ErrorXML() and SuccessXML() functions the same.

3.

We'll write a simple HTML page to test this. This page will use MSXML to post the information to the ASP page, so it will need IE 5 or higher to run.

```
<html><head><title>POST Tester</title>
<script language="JavaScript">
function doPost()
{
    var xdDoc, xhHTTP, sXML

    sXML = "<addNumbers><numOne>";
    sXML += numOne.value;
    sXML += "</numOne><numTwo>";
    sXML += numTwo.value;
    sXML += "</numTwo></addNumbers>";

    xdDoc = new ActiveXObject("MSXML.DOMDocument");
    xdDoc.loadXML(sXML);

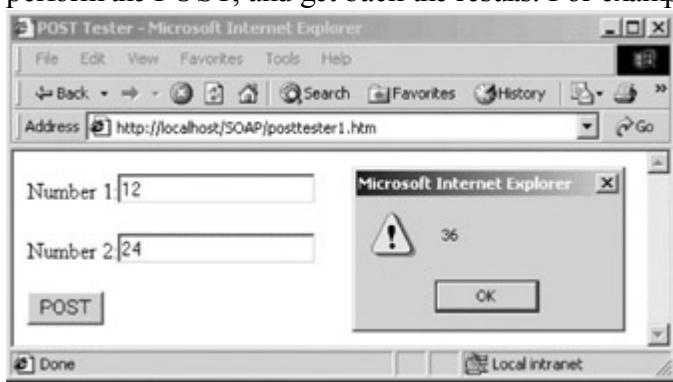
    xhHTTP = new ActiveXObject("MSXML2.XMLHTTP");
    xhHTTP.open("POST", "http://localhost/SOAP/addNumbers2.asp", false);
    xhHTTP.setRequestHeader("SOAPAction", "addNumbers");
    xhHTTP.send(xdDoc);

    xdDoc = xhHTTP.responseXML;
    if(xdDoc.documentElement.nodeName == "Error")
    {
        alert("invalid numbers passed");
    }
    else
    {
        alert(xdDoc.selectSingleNode("/addNumbers/result").text);
    }
}
</script>
</head>
<body>
<p>Number 1:<input id=numOne name=numOne></p>
<p>Number 2:<input id=numTwo name=numTwo></p>
<input type="button" value="POST" id=btnPost name=btnPost onclick="doPost()">
</body></html>
```

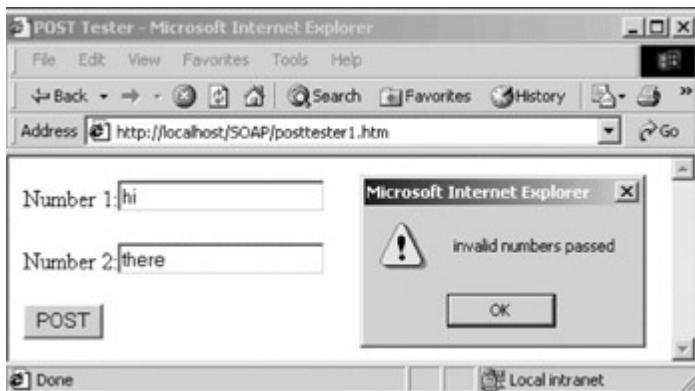
The important code, the JavaScript, is highlighted.

4.

Save this as posttester1.htm, and open it up in IE. Fill in the two text boxes, and click the POST button to perform the POST, and get back the results. For example:



shows a successful result, and



shows the results when non-numeric values are entered.

## How It Works

This example makes use of MSXML's XMLHTTP object, which can make HTTP requests. To start with, we needed to open the connection, which we did using the following line of code:

```
xhHTTP.open("POST", "http://localhost/SOAP/addNumbers2.asp", false);
```

We passed the open() method three parameters:

- The HTTP method we wish to use. In this case, "POST".
- The URL to which we're sending the request.
- A Boolean parameter, to indicate if this should be an asynchronous operation. We specified false, meaning that we want XMLHTTP to make the request synchronously.

Once we opened the connection, we added the SOAPAction header, with the following line of code:

```
xhHTTP.setRequestHeader("SOAPAction", "addNumbers");
```

Next, we called the send() method, to POST our information, and passed it a DOMDocument object (xdDoc), containing our information, packaged up in XML form.

Once the send() method returned, we pulled the HTTP response from the responseXML property, which returns a DOMDocument. (There is also a responseText property, which would return us the raw text. This text would only include the body of the HTTP response, though, not the headers. In this case, the text would be an XML stream.)

The information is being posted to the server in a format similar to this:

```
POST /SOAP/addNumbers2.asp HTTP/1.1
Accept: /*
accept-language: en-ca
referer: http://localhost/SOAP/posttester1.htm
content-type: text/xml
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: localhost
Content-Length: 65
```

```
<addNumbers><numOne>12</numOne><numTwo>24</numTwo></addNumbers>
```

while the information is being returned in a format similar to this:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Fri, 06 Jul 2001 17:48:34 GMT
Content-Length: 48
Content-Type: text/xml
```

```
<addNumbers>
<result>36</result>
</addNumbers>
```

So far we have created a system in which messages are passed back-and-forth, via HTTP POST, in which all of the data is encoded in XML. This is very handy, but we're still missing one crucial piece, for this to be proper SOAP: the XML message must be contained in a standard SOAP **envelope**.

## The Envelope

When data is being sent to a SOAP server, it must be represented in a particular way, so that the server will be able to understand it. The SOAP specification outlines a simple XML document type, which is used for all SOAP messages. The basic structure of that document is as follows:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <head-ns:someHeaderElem xmlns:head-ns="some URI"
      soap:mustUnderstand="1"
      soap:actor="some URI"/>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <some-ns:someElem xmlns:some-ns="some URI"/>
    <!-- OR -->
    <SOAP-ENV:Fault>
      <faultcode/>
      <faultstring/>
      <faultactor/>
      <detail/>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As you can see, there aren't a lot of elements involved in a SOAP message. In fact, there are three main ones, **<Envelope>**, **<Header>**, and **<Body>**, along with a few elements used to describe error conditions. Of these elements, only **<Envelope>** and **<Body>** are mandatory; **<Header>** is optional, and **<Fault>** and its child elements are only required when an error occurs. In addition, all of the attributes (`encodingStyle`, `mustUnderstand`, and `actor`) are optional.

### <Envelope>

Other than the fact that it resides in SOAP's envelope namespace, <http://schemas.xmlsoap.org/soap/envelope/>, the **<Envelope>** element doesn't really need any explanation. It simply provides the root element for the XML document, and is usually used to include any namespace declarations. The next couple of sections will talk about the other elements available, as well as the attributes.

### <Body>

The **<Body>** element contains the main body of the SOAP message. The actual RPC calls are made using direct children of the **<Body>** element (which are called **body entries**). For example, consider the following:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
  <m:addNumbers xmlns:m="http://www.sernaferna.com/soap/add/">
    <m:numOne>12</m:numOne>
    <m:numTwo>24</m:numTwo>
  </m:addNumbers>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this case, we're making one RPC call, to a procedure called `addNumbers`, in the <http://www.sernaferna.com/soap/add/> namespace. This procedure takes two parameters, `numOne` and `numTwo`, which are presumably the numbers to be added. Direct child elements of the `<Body>` element must reside in a namespace other than the SOAP namespace. This namespace is what the SOAP server will use to uniquely identify this procedure, so that it knows what code to run. When the procedure is done running, a SOAP message will be sent back (via the HTTP response) with the results. The `<Body>` of that message might look similar to this:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
  <m:addNumbersResult xmlns:m="http://www.sernaferna.com/soap/add/">
    <m:result>24</m:result>
  </m:addNumbersResult>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

That is, the response is just another SOAP message, using the same XML structure as the request.

There can be multiple RPC calls within one `<Body>` element.

## <Header>

The `<Header>` element is used when you need to add additional information to your SOAP message, besides what is defined by the procedure you're calling. For example, suppose you have created a system whereby orders can be placed into your database using SOAP messages, and have defined a standard SOAP message format that anyone communicating with your system must use. You might use a SOAP header for authentication information, so that only authorized persons or systems can make use of your system.

When there is a `<Header>` element, it must be the first child of the `<Envelope>` element. Functionally, the `<Header>` element works very much like the `<Body>` element; it's simply a placeholder for other elements, in namespaces other than the SOAP envelope namespace, each of which is a SOAP message, which is to be performed in conjunction with the main SOAP message(s) in the body. These elements are called **header entries**. There are also some optional attributes you can include on those header entries: `mustUnderstand`, and `actor`.

### The `mustUnderstand` Attribute

The `mustUnderstand` attribute specifies whether it is absolutely necessary for the SOAP server to process the message in the header. A value of "1" indicates that the header entry is mandatory, whereas a value of "0" indicates that the header entry is optional. For example, consider the following:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header xmlns:some-ns="http://www.sernaferna.com/soap/headers/">
  <some-ns:authentication mustUnderstand="1">
    <UserID>User ID goes here...</UserID>
  </some-ns:authentication>

  <some-ns:log mustUnderstand="0">
    <additional-info>Info goes here...</additional-info>
  </some-ns:log>

  <some-ns:log>
    <additional-info>Info goes here...</additional-info>
  </some-ns:log>
```

```
</SOAP-ENV:Header>
<SOAP-ENV:Body xmlns:body-ns="http://www.sernaferna.com/soap/whatever">
  <body-ns:mainRPC>
    <additional-info/>
  </body-ns:mainRPC>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This SOAP message contains three header entries: one for authentication, and two for logging purposes.

For the <authentication> header entry, we specified a value of "1" for mustUnderstand. This means that the SOAP server must process the header entry. If the SOAP server doesn't understand this header entry, it must reject the entire SOAP message—it is not allowed to process the entries in the SOAP body. In this way, we're forcing proper authentication to be used, otherwise our SOAP server won't even process the SOAP messages sent to it.

In the second header entry, we specified a value of "0" for mustUnderstand, which makes this header entry optional. This means that if the SOAP server doesn't understand this particular header entry, it can still go ahead and process the SOAP body anyway.

Finally, in the third header entry the mustUnderstand attribute was left out. In this case, the header entry is optional, just as if we had specified the mustUnderstand attribute with a value of "0".

## The actor Attribute

In some cases a SOAP message may pass through a number of applications on a number of computers before it gets to its final destination. You might send a SOAP message to Computer A, which might then send that message on to Computer B. Computer A would be called a **SOAP intermediary**.

In these cases you can specify that some SOAP headers must be performed by a specific intermediary, by using the actor attribute. The value of the attribute is a URI, which uniquely identifies each intermediary. Or, if you want the header entry to be executed by the next intermediary in the line, regardless of what computer it's on, you can use the special URI defined in the SOAP specification, "<http://schemas.xmlsoap.org/soap/actor/next>".

When a SOAP server processes a header entry, because it was instructed to by the actor attribute, it is not allowed to pass that header on to the next SOAP intermediary. (However, this rule is easy to get around, if necessary, because the SOAP specification also says that a similar header entry could be inserted in its place; so you could process the SOAP header entry, and then put in another header entry exactly the same.)

## <Fault>

Any time computers are involved, things can go wrong, and there may be times when a SOAP server is not able to process a SOAP message, for whatever reason. Perhaps a resource needed to perform the operation isn't available, or invalid parameters were passed, or the server doesn't understand the SOAP request in the first place. In these cases, the server will return fault codes to the client, to indicate these errors.

Fault codes are sent using the same format as other SOAP messages. However, in this case the <Body> element will have only one child, a <Fault> element. Children of the <Fault> element will contain details of the error. A SOAP message indicating a fault might look similar to this:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Database error</faultstring>
      <faultactor>some URI</faultactor>
      <detail>
        <e:addNumbersFault xmlns:e="some URI">
          <e:dbError>1001</e:dbError>
        </e:addNumbersFault>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<e:message>invalid columnname</e:message>
</e:addNumbersFault>
</detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The <faultcode> element contains a unique identifier, which identifies this particular type of error. The SOAP specification defines four such identifiers:

Fault Code	Description
VersionMismatch	A SOAP message was received that specified a version of the SOAP protocol that this server doesn't understand. (This would happen by specifying a different namespace for the <Envelope> element than the one we've been using so far.)
MustUnderstand	The SOAP message contained a mandatory header, which the SOAP server didn't understand.
Client	Indicates that the message was not properly formatted. That is, the client made a mistake when creating the SOAP message.
Server	Indicates that the server had problems processing the message, even though the contents of the message were formatted properly. For example, perhaps a database was down.

Keep in mind that the identifier is actually namespace-qualified, using the <http://schemas.xmlsoap.org/soap/envelope/namespace>.

The <faultstring> element simply contains a human-readable string, containing a description of the error.

The <faultactor> element contains a URI which specifies which SOAP intermediary caused the fault.

The <detail> element can be used to contain additional, application-specific information about the error. Just like the <Body> element, the actual information is included in namespace-qualified child elements of the <detail> element.

### Try It Out-Using a Proper SOAP Envelope for addNumbers

Our last Try It Out gave us almost all of the benefits of SOAP: It will work easily with a firewall, and all of the information is passed over HTTP in XML, meaning that we could implement our remote procedure using any language, on any platform, and we can call it from any language, from any platform. However, our solution is still a little proprietary. In order to make our procedure more universal, we need to go one further step, and use a SOAP envelope for our XML.

To call the addNumbers procedure, we'll use the following SOAP message:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<an:addNumbers xmlns:an="http://www.sernaferna.com/soap/addNumbers/">
  <an:numOne>12</an:numOne>
  <an:numTwo>24</an:numTwo>
</an:addNumbers>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

And, for our response, we'll send the following XML back to the client:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<an:addNumbersResponse xmlns:an="http://www.sernaferna.com/soap/addNumbers/">
<an:response>36</an:response>
</an:addNumbersResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This gives our SOAP messages the XML structure. The only extra thing we need is a proper SOAPAction header. Since both of the body entries are in the <http://www.sernaferna.com/soap/addNumbers/> namespace we'll use that to create the following two SOAP actions:

```
http://www.sernaferna.com/soap/addNumbers#addNumbers
http://www.sernaferna.com/soap/addNumbers#addNumbersResponse
```

Also, we need to handle our errors using a SOAP envelope as well. We'll use the following format for our errors:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<SOAP-ENV:Fault>
<faultcode>soap:FAULTCODE</faultcode>
<faultstring>ERROR DESCRIPTION</faultstring>
<faultactor>http://www.sernaferna.com/SOAP/addNumbers3.asp</faultactor>
<detail>
<an:addNumbersErrorDetail
      xmlns:an="http://www.sernaferna.com/soap/addNumbers/">
<an:numOne>XXX</an:numOne>
<an:numTwo>XXX</an:numTwo>
</an:addNumbersErrorDetail>
</detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

1.

To start, create a new ASP page, called addNumbers3.asp. To make things easier, you might want to copy and paste all of the code from addNumbers2.asp to this new page, and just make changes as required.

2.

Next, we need to change the SuccessXML() function, so that the XML it returns is a proper SOAP envelope, with our data contained in the body. Modify the function as follows:

```
Function SuccessXML(sResult)
Dim sXML

sXML = "<SOAP-ENV:Envelope xmlns:SOAP-ENV="
sXML = sXML & "'http://schemas.xmlsoap.org/soap/envelope/'>"
sXML = sXML & vbCrLf & "<SOAP-ENV:Body>" & vbCrLf
sXML = sXML & "<an:addNumbersResponse "
sXML = sXML & "xmlns:an='http://www.sernaferna.com/soap/addNumbers'>"
sXML = sXML & vbCrLf
sXML = sXML & "<response>" & sResult & "</response>" & vbCrLf
sXML = sXML & "</an:addNumbersResponse>" & vbCrLf
sXML = sXML & "</SOAP-ENV:Body>" & vbCrLf
sXML = sXML & "</SOAP-ENV:Envelope>"
```

```
SuccessXML = sXML
End Function
```

### 3.

In addition, we also need to return a proper SOAP envelope when an error occurs. Modify ErrorXML() as follows:

```
Function ErrorXML(sFaultCode, sFaultString, sNumOne, sNumTwo)
Dim sXML

sXML = "<SOAP-ENV:Envelope xmlns:SOAP-ENV="
sXML = sXML & "'http://schemas.xmlsoap.org/soap/envelope/'>"
sXML = sXML & vbCrLf & "<SOAP-ENV:Body>" & vbCrLf
sXML = sXML & "<SOAP-ENV:Fault>" & vbCrLf
sXML = sXML & "<faultcode>soap:" & sFaultCode & "</faultcode>" & vbCrLf
sXML = sXML & "<faultstring>" & sFaultString & "</faultstring>" & vbCrLf
sXML = sXML & "<faultactor>http://www.sernaferna.com/"
sXML = sXML & "SOAP/addNumbers3.asp</faultactor>" & vbCrLf
sXML = sXML & "<detail>" & vbCrLf
sXML = sXML & "<an:addNumbersDetail >
sXML = sXML & "xmlns:an='http://www.sernaferna.com/soap/addNumbers/'>"
sXML = sXML & vbCrLf & "<an:numOne>" & sNumOne & "</an:numOne>" & vbCrLf
sXML = sXML & "<an:numTwo>" & sNumTwo & "</an:numTwo>" & vbCrLf
sXML = sXML & "</an:addNumbersDetail>" & vbCrLf
sXML = sXML & "</detail>" & vbCrLf
sXML = sXML & "</SOAP-ENV:Fault>" & vbCrLf
sXML = sXML & "</SOAP-ENV:Body>" & vbCrLf
sXML = sXML & "</SOAP-ENV:Envelope>"
ErrorXML = sXML
End Function
```

Notice that the function now takes a number of parameters, since some of the information in this XML is dynamic, based on the type of error that occurred. This will be a lot more useful to users of our procedure than a message that says "Invalid numbers" every single time.

### 4.

Finally, we need to modify the main code in our ASP page, deal with the new XML format, and pass the proper parameters to ErrorXML(), when needed. The finished addNumbers3.asp is shown below:

```
<%@ Language=VBScript %>
<%
Response.ContentType = "text/xml"
Response.AddHeader "SOAPAction",
"http://www.sernaferna.com/soap/addNumbers#addNumbersResult"

Dim xdRequestXML
Dim sNumber1, sNumber2
Dim nResult

Set xdRequestXML = Server.CreateObject("MSXML.DOMDocument")
xdRequestXML.load Request

sNumber1 = xdRequestXML.documentElement.firstChild.firstChild.firstChild.text
sNumber2 = xdRequestXML.documentElement.firstChild.firstChild.lastChild.text

On Error Resume Next
nResult = CLng(sNumber1) + CLng(sNumber2)

If Err.number <> 0 Then
    Response.Write ErrorXML("Client", "Invalid numbers", sNumber1, sNumber2)
ElseIf Request.ServerVariables("HTTP_SOAPAction") <>
"http://www.sernaferna.com/soap/addNumbers#addNumbers" Then
    Response.Write ErrorXML("Client", "Procedure not understood",
Request.ServerVariables("SOAPAction"), sNumber2)
```

```
Else
    Response.Write SuccessXML(nResult)
End If

Function ErrorXML(sFaultCode, sFaultString, sNumOne, sNumTwo)
    Dim sXML

    sXML = "<SOAP-ENV:Envelope xmlns:SOAP-ENV="
    sXML = sXML & "'http://schemas.xmlsoap.org/soap/envelope/'>"
    sXML = sXML & vbCrLf & "<SOAP-ENV:Body>" & vbCrLf
    sXML = sXML & "<SOAP-ENV:Fault>" & vbCrLf
    sXML = sXML & "<faultcode>soap:" & sFaultCode & "</faultcode>" & vbCrLf
    sXML = sXML & "<faultstring>" & sFaultString & "</faultstring>" & vbCrLf
    sXML = sXML & "<faultactor>http://www.sernaferna.com/"
    sXML = sXML & "SOAP/addNumbers3.asp</faultactor>" & vbCrLf
    sXML = sXML & "<detail>" & vbCrLf
    sXML = sXML & "<an:addNumbersDetail "
    sXML = sXML & "xmlns:an='http://www.sernaferna.com/soap/addNumbers/'>"
    sXML = sXML & vbCrLf & "<an:numOne>" & sNumOne & "</an:numOne>" & vbCrLf
    sXML = sXML & "<an:numTwo>" & sNumTwo & "</an:numTwo>" & vbCrLf
    sXML = sXML & "</an:addNumbersDetail>" & vbCrLf
    sXML = sXML & "</detail>" & vbCrLf
    sXML = sXML & "</SOAP-ENV:Fault>" & vbCrLf
    sXML = sXML & "</SOAP-ENV:Body>" & vbCrLf
    sXML = sXML & "</SOAP-ENV:Envelope>"
    ErrorXML = sXML
End Function

Function SuccessXML(sResult)
    Dim sXML

    sXML = "<SOAP-ENV:Envelope xmlns:SOAP-ENV="
    sXML = sXML & "'http://schemas.xmlsoap.org/soap/envelope/'>"
    sXML = sXML & vbCrLf & "<SOAP-ENV:Body>" & vbCrLf
    sXML = sXML & "<an:addNumbersResponse "
    sXML = sXML & "xmlns:an='http://www.sernaferna.com/soap/addNumbers'>"
    sXML = sXML & vbCrLf
    sXML = sXML & "<response>" & sResult & "</response>" & vbCrLf
    sXML = sXML & "</an:addNumbersResponse>" & vbCrLf
    sXML = sXML & "</SOAP-ENV:Body>" & vbCrLf
    sXML = sXML & "</SOAP-ENV:Envelope>"

    SuccessXML = sXML
End Function
%>
```

When accessing the XML sent from the client, I simply used the `firstChild` and `lastChild` properties of the `Node` interface, rather than dealing with the intricacies of the namespaces in the XML document.

## 5.

Finally, we need to modify our HTML test page, to pass the proper SOAP message to `addNumbers3.asp`. Create a new HTML page, called `posttester2.htm`, and copy and paste the code from `posttester1.htm`. The only code we need to change is the `doPost()` function, to pass the proper XML format (and change the SOAP action). Modify `doPost()` as follows:

```
function doPost()
{
    var xdDoc, xhHTTP, sXML

    sXML = "<SOAP-ENV:Envelope xmlns:SOAP-ENV="
    sXML = sXML + "'http://schemas.xmlsoap.org/soap/envelope/'>\n";
    sXML = sXML + "    <SOAP-ENV:Body>\n";
    sXML = sXML + "        <an:addNumbers"
    sXML = sXML + "            xmlns:an='http://www.sernaferna.com/soap/addNumbers'>\n";
```

```
sXML = sXML + "      <an:numOne>";
sXML = sXML + numOne.value;
sXML = sXML + "</an:numOne>\n";
sXML = sXML + "      <an:numTwo>";
sXML = sXML + numTwo.value;
sXML = sXML + "</an:numTwo>\n";
sXML = sXML + "    </an:addNumbers>\n";
sXML = sXML + "  </SOAP-ENV:Body>\n";
sXML = sXML + "</SOAP-ENV:Envelope>";

xdDoc = new ActiveXObject("MSXML.DOMDocument");
xdDoc.loadXML(sXML);

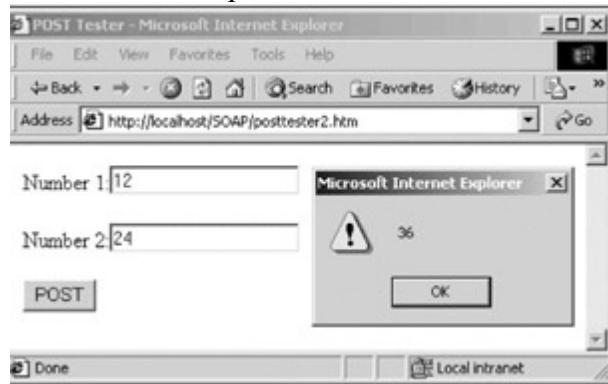
xhHTTP = new ActiveXObject("MSXML2.XMLHTTP");
xhHTTP.open("POST", "http://localhost/SOAP/addNumbers3.asp", false);
xhHTTP.setRequestHeader("SOAPAction",
"http://www.sernaferna.com/soap/addNumbers#addNumbers");
xhHTTP.send(xdDoc);

xdDoc = xhHTTP.responseXML;
if(xdDoc.documentElement.firstChild.firstChild.nodeName == "Fault")
{
    alert("invalid numbers passed");
}
else
{
    alert(xdDoc.documentElement.firstChild.firstChild.firstChild.text);
}
}
```

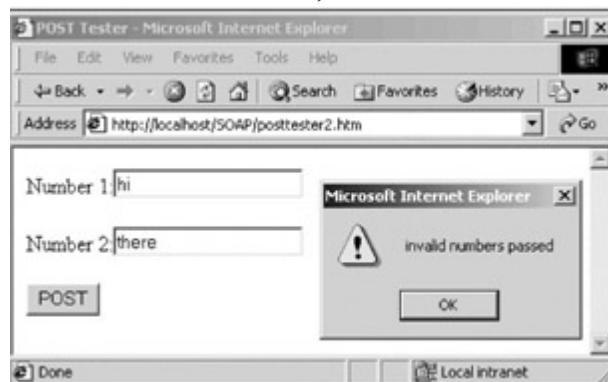
Notice that, in addition to changing the format of the XML we're sending to the server, we also need to change the way we're accessing the XML returned from the server. Again, I simply used the `firstChild` property, rather than `selectSingleNode()`, to avoid the intricacies of the namespaces in our XML document.

## 6.

Once you have saved your changes, open `posttester2.htm` in IE. It should work exactly as the previous version. For example:



shows a successful result, and



shows the results when non-numeric values are entered.

This seems like a lot of work, simply to add two numbers together! However, realize that we have created, from scratch, all of the plumbing necessary to create an entire SOAP service. Implementing a more difficult SOAP service, such as some type of order-processing system, would require the same level of plumbing, even though the functionality being provided would be much more difficult.

*In addition, there are a number of SOAP toolkits available, meaning that you won't necessarily have to write the SOAP plumbing by hand, like this, every time you want to use SOAP to send messages from one computer to another. However, now, when you use those toolkits, you'll understand what's going on under the hood.*

## Encoding Rules

So far, we've been sending pretty simple pieces of data around in our SOAP message: a couple of numbers to be added together, and another number that is the result of the addition. In some cases, we are sending a bit of text, such as in the <faultstring> element when there's an error. In real life, the data we need to send in our SOAP messages will often be much more complex.

In other areas of computer science, programmers are used to dealing with complex data types, such as arrays, enumerations, structures, etc. In order for SOAP to be useful to a broad range of programmers, there needed to be a way to send this kind of information in a SOAP body as well, so a set of **encoding rules** was created in the SOAP specification. These rules allow complex data types and objects to be serialized as XML in order to be sent in SOAP messages.

It is important to note, however, that these rules are not mandatory. You are free to use these rules in your SOAP messages, if you wish, but you are also free to send your XML without using the encoding rules outlined in the SOAP specification, or even to invent your own encoding rules. In fact, the SOAP specification defines the encodingStyle attribute, which specifies which set of encoding rules your SOAP message is using. The value of the encodingStyle attribute is a URI, which uniquely identifies your encoding rules. (The encoding rules outlined in the SOAP specification are identified by the URI <http://schemas.xmlsoap.org/soap/encoding/>.)

The encodingStyle attribute can be attached to any element in a SOAP message. For example, in this document the encoding rules attribute applies to the entire SOAP message

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
    <SOAP-ENV:Body>  
        <an:addNumbersResult xmlns:an="http://www.sernaferna.com/soap/addNumbers/">  
            <an:result>36</an:result>  
        </an:addNumbersResult>  
    </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

while in this message, the encoding rules only apply to the body:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">  
    <SOAP-ENV:Body SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">  
        <an:addNumbersResult xmlns:an="http://www.sernaferna.com/soap/addNumbers/">  
            <an:result>36</an:result>  
        </an:addNumbersResult>  
    </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

and in this message, the encoding rules only apply to the <an:addNumbersResult> procedure call:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">  
    <SOAP-ENV:Body>  
        <an:addNumbersResult xmlns:an="http://www.sernaferna.com/soap/addNumbers/">
```

```
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">"
<an:result>36</an:result>
</an:addNumbersResult>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

We won't go into details on the encoding rules outlined in the SOAP specification, but they're there if you need them. (The SOAP specification is available at <http://www.w3.org/TR/SOAP/>.)

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# What Are Web Services?

Chances are, if you've heard the hype about SOAP, you've also heard the hype about **Web Services**. What are they, and how do they relate to SOAP? Well, SOAP just solves part of the distributed computing problem: it allows you to make remote procedure calls. However, in order to make use of a procedure which is available through SOAP, you have to know where that procedure is located (its URL), as well as exactly how to pass the information to that procedure?how to format the body of the SOAP message. You may also need other information about the procedure, such as what information it returns.

Web Services provide this functionality. At the time of writing, the W3C was still going over what different pieces needed to be defined for web services, and had set up an **XML Protocol Activity**, which you can access at <http://www.w3.org/2000/xp/>. The different pieces needed to define web services may include SOAP, XML Schema, a language called **WSDL** to describe web services, a protocol called **UDDI** to allow publishing and discovery of web services, and even other protocols and languages. For more information on Web Services, refer to the Wrox Press book, *Professional XML Web Services*, ISBN 1-861005-09-1.

## WSDL

The **Web Services Description Language (WSDL)** is an XML-based language that provides a contract between a web service and the outside world. To understand this better, let's go back to our discussion of COM and CORBA.

The reason that COM and CORBA objects can be so readily shared is that they have defined contracts with the outside world. This contract defines the methods an object provides, as well as the parameters to those methods, and their return values. Interfaces for both COM and CORBA are written in variants of the **Interface Definition Language**, or **IDL**. Code can then be written to look at an object's interface, and figure out what functions are provided. In practice, this dynamic investigation of an object's interface often happens at design time, as a programmer is writing the code that calls another object. A programmer would find out what interface an object supports, and then write code that properly calls that interface.

Web Services have a similar contract with the outside world, except that the contract is written in WSDL, instead of IDL. This WSDL document outlines what messages this SOAP server expects, in order to provide services, as well as what messages it returns. Again, in practice, WSDL would probably be used at design time. A programmer would use WSDL to figure out what procedures are available from the SOAP server, and what format of XML is expected by that procedure, and then write the code to call it.

Or, to take things a step further, programmers might never have to look at WSDL directly, or even deal with the underlying SOAP protocol. There are already a number of SOAP "toolkits" available, which can hide the complexities of SOAP from you. If you point one of these toolkits at a WSDL document, it can automatically generate code to make the SOAP call for you! At that point, working with SOAP is as easy as calling any other local object on your machine.

*Two popular SOAP toolkits are the **Web Services Toolkit** from IBM, available from their developerWorks web site (<http://www.ibm.com/developerworks>), and the **SOAP Toolkit** from Microsoft, available from their Microsoft Developer Network web site (<http://msdn.microsoft.com>).*

## UDDI

The **Universal Discovery, Description, and Integration** protocol (**UDDI**) is a protocol which allows web

services to be registered, so that they can be discovered by programmers and other web services.

For example, if I'm going to be doing business with another company over the Web, and I know that they have a UDDI server, I can query that server to find out what web services are provided. The UDDI service will return an XML document, naturally, describing all of the available services.

Many of the SOAP toolkits available, such as IBM's Web Services Toolkit, provide tools to work with UDDI.

---

[!\[\]\(a0881e932d41dc60794092e2732cdd17\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(355b02fa81e9a4492d7a00d807c966ba\_img.jpg\) NEXT >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

In this chapter, we have looked at SOAP, an XML-based protocol for performing remote procedure calls. We have studied how this protocol is used, and even put it to practice by creating a SOAP-based addition service.

Because SOAP is based on easy-to-implement and standardized technologies such as XML and HTTP, it has the potential to become a very universal protocol indeed. In fact, most of the hype surrounding SOAP concerns its interoperability. At least initially, companies providing SOAP software are concentrating on making their software as interoperable as possible with the software from other companies, instead of creating proprietary changes to the standard.

With the backing of companies such as Microsoft, IBM, DevelopMentor, Lotus, UserLand Software, Sun Microsystems, and Canon, SOAP is sure to be a widely implemented technology, and the web services built on top of SOAP also have huge potential for creating widely accessible functionality over the Web.

In the [next chapter](#), we'll look at how we can display XML documents in a browser, using Cascading Style Sheets.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 11: Displaying XML

## Overview

To style or not to style, that is the question. This chapter covers the methods you can choose from if you decide to style your XML for effective presentation on the Web. It not only looks at **Cascading Style Sheets** (CSS), but also aims to help you to understand how to design your XML (which is structural in nature) to take best advantage of this presentational tool.

We'll be looking at:

- How the concept of stylesheets came into being and was developed to solve various issues that display of marked up documents presented
- How CSS is used with HTML/XHTML
- How CSS can be used with XML, including transforming your XML through XSLT and applying CSS to the output
- The future of CSS with XML

*While a number of basic concepts will be introduced here, CSS covers a broad area, with two "official" Recommendations (CSS Level 1 and Level 2) and a working draft for the third generation of CSS (CSS Level 3), covering some 800 pages worth of documentation. For more detailed information, check out the W3C's web styling home page at <http://www.w3.org/Style/>.*

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# The Need for Style(sheets)

Where do you draw the line between the structure of a document and its presentation? This is one of the conundrums that plagues both web and XML designers to this day, as the distinctions between what we mean by the "logical" structure and the "presentation" tend to shift from situation to situation, but by understanding the effective separation between the two you can create much more modular, maintainable code?both XML and otherwise.

HTML was not originally intended to be quite the powerhouse language that it has become over the years. Indeed, at first, it was basically a way of marking up physics abstracts?summaries of articles on the network by the various physicists at CERN. The first HTML "browser" reflected this: you could give the title, the various headings, and sundry citation elements in the language (the logical structure of the document) and they would show up in a fixed font on the screen; the presentation may consist of bolding or underlining elements if the system offered the capability, but the *presentation* aspect of the abstract was negligible at best.

This changed with the NCSA Mosaic browser, which made it possible to provide basic graphical markup on the page, as well as include pictures. The pictures were perhaps the most important part of this, since the program itself determined the text presentation automatically. Still, the web pages involved went from being geekish novelties to looking almost like a page from a magazine (albeit very simple magazines at first).

It was this similarity that no doubt accounted for the phenomenal growth of the Web. While primitively interactive, the web pages that the new breed of programmer/graphic artist built began to blur the distinction between structure and presentation. The `<em>` (emphasis) tag was replaced with the `<i>` (italics) tag, the `<strong>` tag with the `<b>` (bold) tag, the latter being strongly established typographic terms. Because most documents are not physics abstracts, the language was stretched and pushed and punctured to provide an increasing number of tags, and the two dominant players of the time, Netscape and Microsoft, each routinely added proprietary tags to the rapidly evolving HTML specification to gain a competitive edge in the fast growing browser sphere.

In the wake of these browser wars, the unfortunate losers were those web designers and developers who were trying to stay current with the "standards" yet still cater to the demands of their users. Many of the "innovations" being pushed by the browser manufacture's involved giving the web designer (or in a few cases the end user) specialized tags that could set some kind of behavior on a block of text. Some of these were truly obnoxious?the epileptic seizure inducing `<blink>` tags, the dizzying `<marquee>`?but in most cases the tags actually did have a certain degree of usefulness.

Perhaps one of the most widely used (and abused) tags was the `<font>` tag, which made it possible to set the relative size, font?face, and color of a block of text. While this tag certainly made it possible to create web pages that far more closely resembled magazine presentation, it had the very real problem of thoroughly confusing the structure of a page?the various headings, paragraph elements and marginalia?with the visual appearance of the page. For instance, consider two sets of elements:

```
<h1>This is the primary document title</h1>
```

```
<div><font size="7" face="New Times Roman">This is the primary document title.</font></div>
```

The first line provides a logical view of a primary header element, without knowing anything about the fonts or other decorative devices an application can look at the first element and know it is a primary header. On many web browsers, the second expression looks identical to the first, but it is contextually poor. That same application, in trying to interpret the meaning of the expression, may possibly be able to guess based upon a combination of font size, face and location, but it is just that ... a guess. The logical structure of the document has broken down here.

It is worth noting here that at the time the <font> tag was introduced, it was, short of the user physically setting the display attributes for himself or herself in whatever browser was used, the only real way to change font characteristics. Even today, if it is used intelligently, the font tag can be used to support changing the media characteristics of a specific element, provided that the element itself retains its logical meaning. In the example above, if the <h1> element had included the font tag, then the <font> tag in turn acts as a styling element:

```
<h1><font size="7" face="New Times Roman">This is a primary title</font></h1>
```

is good practice, while

```
<div><font size="7" face="New Times Roman">This is a primary title</font></div>
```

is not, since the <div> element contains no real logical information.

This distinction between logical structure and media presentation may seem like an academic argument, but, in fact, it makes a profound difference when you are attempting to scale web sites up from a few web pages to a few thousand. By being able to create a consistent, uniform means of painting the visual (or aural) vision of a document, you can devote far more effort to creating, storing, and manipulating the real content, and far less on the presentation of that content.

---

[!\[\]\(a9216c18e6ea0011f80493b3c4eb3a73\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(ba7af7ad2557bbd5060b27ee071341d0\_img.jpg\) NEXT](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Enter Cascading Style Sheets

CSS was an attempt to marry the logical or structural markup characteristics of HTML with the media presentation characteristics of Postscript. Like another stylesheet language (XSLT), CSS is rule-based. The CSS-enabled browser applies the rule to the elements that are indicated in a specific stylesheet, a document, which may be either embedded within its host HTML document or contained separately.

While CSS was meant to ameliorate at least one problem—the difficulty of providing consistent information about how a given header font was sized or which font was used—it has actually found a place within the XML community itself. In some respects, in fact, the combination of XML and CSS is a natural, since XML provides pure structure about a document with no real concept of media presentation, while CSS provides pure media presentation, generally with only a very fuzzy notion of how the document is structured. More about CSS and XML later.

CSS were first proposed in early 1995 by the Style working group of the W3C, whose homepage is located at <http://www.w3.org/Style/CSS/>. The group had five primary purposes (though not necessarily prioritized in this order):

- To slow down (if not stop altogether), the barrage of new tags that were being introduced every time one of the major browser vendors wanted to get a competitive edge in the marketplace.
- To separate the logical structure of a web page from the media presentation of that web page, making the source documents more semantically rich in the process.
- To establish a uniform set of media attributes that could be used by all browser vendors, which would at least cut down on the very real danger of fragmentation that seemed to be all but assured in the browser sphere.
- To create a way of establishing meta tags that could more appropriately describe a specific set of functionality.
- Finally, to create a means for working with media beyond the web page, including printed paper, hand-held devices, phones, speech devices, and so on.

For instance, you can define with CSS an alternative media description of the paragraph `<p>` element:

```
<style>
  h1 {font-family:Impact; font-size:52pt;}
  p {font-family: Helvetica; font-size:11pt;}
</style>

<p>Warning, Will Robinson!</p>
```

The block of code does several things. The `<style>` element declares that all lines of code within the style utilize the CSS stylesheet language, and as such should not be interpreted by the browser as text. In essence, the `<style>` element "escapes" the content out of the HTML environment and into the CSS one, just as `<script>` escapes JavaScript code.

Each line within the style block is a **rule**, something that describes the characteristics of an HTML or XML tag. <style> elements can contain multiple rules. A rule in turn is made up of two parts: a **selector** that describes what tags the expression matches, along with a **descriptor** that contains zero or more CSS **properties**, contained with a set of brackets {}. For example:

```
h1 {font-family:Impact; font-size:52pt;}
```

has a selector of "h1" and a descriptor {font-family:Centaur; font-size:52pt}, which in turn is made up of the two CSS properties font-family and font-size that have the respective property values: "Impact" (a font name) and "52pt" (52 points, a font size). There are minor variations of this syntax. For instance, you can specify the background texture of certain elements (such as a heading) using the url notation:

```
h1 {font-family:Impact; font-size:52pt; color:black;  
background-image:url(images/cssPaper.gif) }
```

where url(images/cssPaper.jpg) contains a reference to the file cssPaper.jpg in the folder images, relative to the folder of the current document. This is the result when viewed in the browser:



The W3C Cascading Style Sheet language has taken a long time to filter through the Internet. Microsoft Internet Explorer 3.0 was the first browser to support a partial implementation of the language. Ironically, it never achieved full compliance with even the first of the CSS Recommendations (of which there are currently two, with the third in Working Draft status) until the recent release of Internet Explorer 6.0. Likewise, the Netscape 4.0 browser had a very strange implementation of CSS, achieved something closer with Netscape 4.7, and with Netscape 6.0 is compliant with CSS Level 1 and most of CSS 2. Finally, it's worth noting that even across platforms the support for CSS has been inconsistent: Internet Explorer 5.0 on the Macintosh has supported almost all of the CSS 2 Recommendation for some time now, but 100% CSS 1 compliance has only just been achieved for Windows with Internet Explorer 6.0.

There are a number of browsers out there that support the CSS standard-sort of. To provide a general benchmark (and to help me write this chapter), I chose to look at three of the most common: Internet Explorer 6.0, Opera 5.12 and Mozilla 0.9.4. It is worth noting carefully that Opera and Mozilla, at the time of writing this chapter, are still in beta development, so you should confirm any comments that are made concerning support or lack of adherence to the CSS standard for your browser of choice. I chose not to test against the Amaya browser from the W3C (see <http://www.w3.org/Amaya/>), because while that is a benchmark browser, it is considered primarily a "concept client" for seeing whether a specific implementation may work, rather than a commercial browser in widespread use.

It is safe to say that most browsers released within the last six months now fully implement CSS level 1, five years after the CSS 1 specification was first published. Support for CSS level 2 is considerably more limited, and only the benchmark Amaya browser offers even a portion of the CSS level 3 set (which is, admittedly, still a Working Draft, not a full formal Recommendation). Put another way, the full CSS Recommendation as it stands as of writing this book may be completely supported by 2007, or later-a seeming lifetime on the Internet. However, if the success of CSS 1 is any indication, this slow and steady (if sometimes contentious) route will likely result in a standard that will last long after most proprietary "extensions" have become curious relics.

# Linking Stylesheets in HTML

There is something of a chicken and egg concept when dealing with CSS: understanding how global Cascading Stylesheets link into HTML or XML can make it easier to understand how CSS properties work, but at the same time you need to understand the basics of CSS to create such stylesheets. The following section looks at some linking examples that use CSS?for more information about what is specifically happening, at the actual rule level, you'll want to read on to the later sections of this chapter.

You can define a fairly comprehensive set of style rules in HTML, but if you have to explicitly declare them every time you create a new document, the ability to reuse them in a modular fashion diminishes dramatically. Fortunately, you can in fact link your HTML or XHTML document to external files, making it possible to specify one CSS document that is referenced uniformly by all XHTML documents in the web site. XHTML 1.0 linking follows the same linking convention as HTML 4.01 (which is only natural, as XHTML is simply an XML compliant version of the HTML 4.01 specification) in using the `<link>` element.

The `<link>` element was intended to be considerably more exhaustive than it finally ended up being. As originally envisioned, `<link>` served as an associational mechanism to point to any external resource that the web page may end up needing. In practice, most browsers only really support a link element that has a rel attribute set to "stylesheet" (Mozilla used to support rel links to home, tables of contents, and so forth, but in the most recent build that functionality seems dormant). In the following Try It Out exercise, you can see the effects of using `<link>` to add CSS formatting through an external file:

## Try It Out? Adding an External Stylesheet to an HTML or XHTML Document

1.

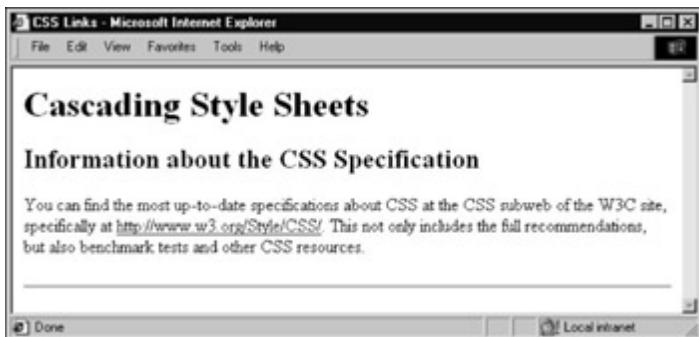
Create the following document in a folder, and call it `cssLinks.htm`:

```
<html>
<head>
  <link href="cssChapterFormat.css" rel="stylesheet" type="text/css"/>
  <title>CSS Links</title>
</head>

<body>
  <h1>Cascading Style Sheets</h1>
  <h2>Information about the CSS Specification</h2>
  <p>You can find the most up-to-date specifications about CSS at the CSS
    subweb of the W3C site, specifically at
    <a href="http://www.w3.org/Style/CSS/">http://www.w3.org/Style/CSS/</a>.
    This not only includes the full recommendations, but also benchmark
    tests and other CSS resources.
  </p>
  <hr/>
</body>
</html>
```

2.

View the file `cssLinks.htm` in a browser. It should look something like this:



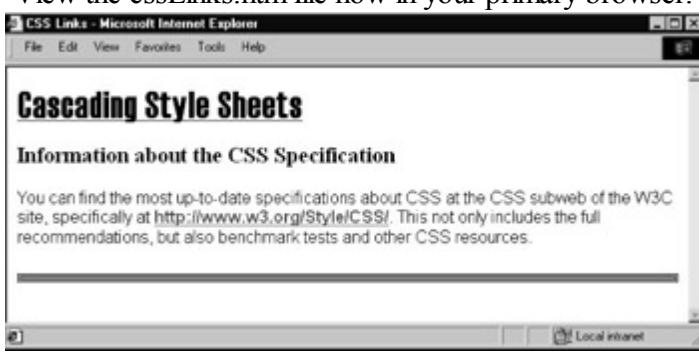
3.

In a text editor, create another file called cssChapterFormat.css, as follows, and place it in the same folder as cssLinks.htm.

```
h1 { font-family:Haettenschweiler, Helvetica Heavy, serif;  
font-size:26pt;  
text-decoration:underline;  
}  
h2 { font-family:Times New Roman, serif;  
font-size:16pt;  
font-weight:bold;  
}  
p { font-family: Arial, Helvetica, sans-serif;  
font-size:12pt;  
}  
  
a { text-decoration:underline;  
color:red;  
font-weight:bold;  
}  
  
hr { border:double 3px blue;  
height:8px;  
}
```

4.

View the cssLinks.htm file now in your primary browser. It should be similar to this:



You may also want to try seeing it in different browsers, to get an idea about how each browser handles the same problem.

## How It Works

Linking follows the same inheritance rules that internal stylesheets follow, with the caveat that a linked document will always have a lower inheritance priority (so will always be overridden) if an internal stylesheet exists. Additionally, it is possible to load more than one linked stylesheet, with conflicting CSS properties always resolving to the last stylesheet loaded.

Linking to CSS documents is a convenience in HTML, but it becomes a very necessary capability when stylesheets are applied to XML. However, because XML tags do not contain any predefined meaning, the mechanism used to apply stylesheets for XML is very different from the use of the <link> element. To link with CSS, you actually have to go outside of the element structure and apply a processing instruction to inform the browser how to display it, of the form:

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

The following Try It Out is very similar to the one above, but with an XML rather than HTML source.

## Try It?Adding an External Stylesheet to an XML Document

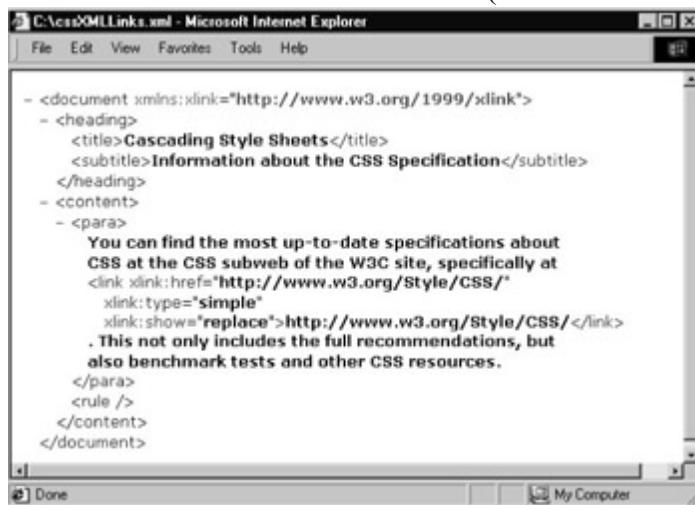
1.

In your text editor, create the following document and save it in a folder, calling it cssXMLLinks.xml:

```
<document xmlns:xlink="http://www.w3.org/1999/xlink">
<heading>
  <title>Cascading Style Sheets</title>
  <subtitle>Information about the CSS Specification</subtitle>
</heading>
<content>
  <para>You can find the most up-to-date specifications about CSS at the
    CSS subweb of the W3C site, specifically at
    <link xlink:href="http://www.w3.org/Style/CSS/" xlink:type="simple"
      xlink:show="replace">http://www.w3.org/Style/CSS/</link>. This not only
      includes the full recommendations, but also benchmark tests and other
      CSS resources.
  </para>
  <rule/>
</content>
</document>
```

2.

View the document in an XML viewer (such as Internet Explorer). The results will look something like this:



3.

At the top of the CSSXMLLinks.xml file, add the following line:

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

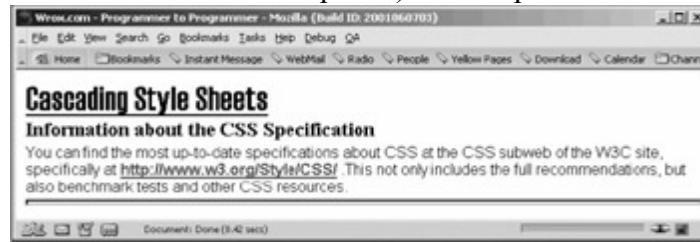
4.

Create a new document in the same folder called CSSDocumentXML.css, and put the following code into it:

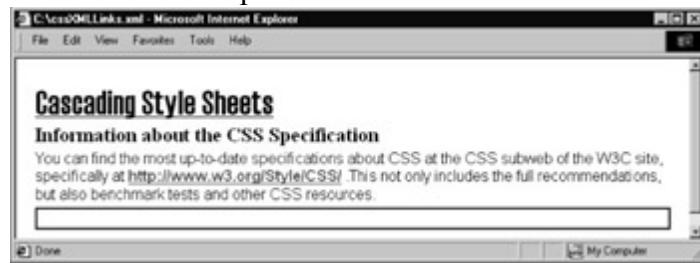
```
document {display:block;padding:10px;}  
heading title  
{display:block;  
font-family:Haettenschweiler, serif;  
font-size:26pt;  
text-decoration:underline;  
margin-bottom:5px;  
}  
heading subtitle  
{display:block;  
font-family:Times New Roman, serif;  
font-size:16pt;  
font-weight:bold;  
margin-bottom:3px;  
}  
content {display:block;}  
para {display:block;  
font-family: Arial, Helvetica, sans-serif;  
font-size:12pt;  
margin-bottom:5px;  
}  
  
link {display:inline;  
text-decoration:underline;  
color:red;  
font-weight:bold;  
}  
  
rule {border:inset 2px blue;  
height:2px;  
width:100%;  
display:block;  
}
```

## 5.

Save it and view the CSSXMLLinks.xml file in Internet Explorer 4+, Mozilla 0.8+, or Opera 5+ (this code will not work with Netscape 4.x). The output in Mozilla will look much like:



while the same output in IE will look like:



Note that the CSS support in Internet Explorer is not, in general, as robust as that of Mozilla. For instance, you can create an XLink reference that will act like a link (<a>) tag in Mozilla, but this isn't supported in the IE model?the user defined <link> element will style correctly, but it won't actually do anything.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# The Basics of HTML/XHTML CSS

So what is so "cascading" about Cascading Style Sheets? Think of rice paddies. In many parts of the world, rice paddies are set up in terraces that permit water, applied to the top, to cascade down to any terrace that sits below it. In time the water will eventually fill the whole valley evenly. However, only those terraces that are at or below the level of the water's source will fill?if there are terraces above this one, they won't receive any water at all.

A CSS property is the water to an element's terrace. The property fills that element in which it is first defined, and any element (or text) that this element *contains*: anything between the element's opening and closing tags. However, the property will not fill beyond that element, nor will it affect anything up the chain?any element that contains the element in question.

Note that the idea of containment is central to XML. You have to explicitly define a closing element for each opening element, even if that closing element is itself. In that particular case, an element is closed according to some fairly complex rules that in general boil down to just before the start of the next similar container. Thus, a paragraph `<p>` element will close either when a `</p>` terminating tag is encountered, another `<p>` tag is next, an `<h1>?<h9>` tag is next, and so forth.

Cascading works on four distinct levels;element styles, document styles, external stylesheets, and browser styles. While there are exceptions, if a property is assigned in an element and a document, or a document and an external stylesheet, the property value with the more specific scope will be the one applied. For instance, if the foreground color for a document was set to blue, but the foreground color for an element was set to red, then any elements or text within the element will be red, while anything outside of the element will be blue. This will be covered in greater detail throughout this section.

Here's a question, however: what if you never define a value for a given CSS property in a web page? This is where default browser settings come into play. Most browsers define specific default values for all CSS properties, and use these whenever that property is not explicitly defined somewhere in the cascade. In the case of the Netscape, Mozilla, Opera and Internet Explorer browsers, some style properties are accessible through dialogs (implying, of course, that users can change their default CSS settings, something that should *always* be factored in web design). Additionally, the Opera browser gives users the ability to assign an external stylesheet to their browsers, handling all cases where properties are not specifically defined by the web developer.

This has a number of consequences when working with CSS:

- Never assume that a given element will have the same style across all browsers.
- Never make critical information a part of CSS without insuring it exists in the source document as well.
- Users can override styles with their own built-in stylesheets.
- Before posting a page to a site, disable all CSS references so that you can see what your page will look like in a viewer that doesn't have CSS capabilities (one of the biggest arguments for using document level stylesheets or above is the ease of disabling them compared to in?element style attributes).

In the next sections we'll work our way up from the most locally applied style declarations at the element level to the

most generalized at the document level.

## Element Styles and the Style Attribute

The HTML style attribute is used to assign CSS style properties to an element in either HTML or XHTML (for the purpose of discussion, this chapter treats them as one unless otherwise indicated). The style defines a local (unnamed) rule, which includes the descriptor text, but assumes that the selector is in fact the current element. For instance, the following paragraph:

```
<p style="font-size:11pt; font-family:Centaur;">Understanding the nature of reality, the characteristics of being or self, is the primary purpose of the discipline of ontology.</p>
```

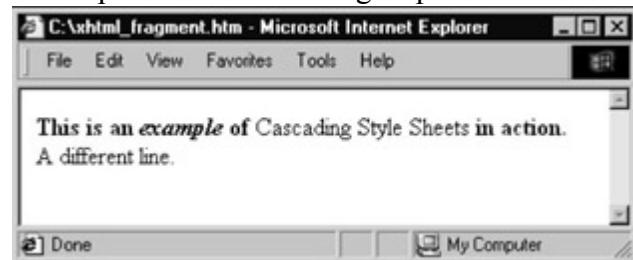
```
<p>Ontology is currently a topic of immense interest among researchers who are exploring the concepts of meaning and knowledge as they apply to computer systems.</p>
```

includes a style attribute that creates a temporary rule which affects the CSS properties of font-size and font-family for the paragraph and anything inside it. This means that the font-family of the first paragraph will be "Centaur" (a serifed font) uniformly throughout the paragraph, but because the second paragraph isn't contained within the first paragraph, it will instead utilize the CSS font-family and font-size properties defined either at the document (or external stylesheet) level.

In essence, when you set a stylesheet property in CSS on an XHTML element, what you are doing is changing that characteristic for the element in question as well as everything that the element contains. As another example, consider the following XHTML fragment:

```
<div style="font-weight:bold">
  This is an <i>example</i> of <span style="font-weight:normal">
    Cascading Style Sheets</span> in action.
</div>
<div>
  A different line.
</div>
```

which produces the following output:



The first line sets the font-weight property of the enclosing `<div>` to the value "bold", indicating that all contents should be made bold. In the expression "`<i>example</i>`", the bold weight remains in force?the italics tag has inherited the bold font-weight property from its container. However, the expression "`<span style="font-weight:normal">Cascading Style Sheets</span>`" sets the property of its `<div>` element and children back to the normal state. This only holds within the defined span, it doesn't affect the following text. Similarly, the next major `<div>` element doesn't see the changed font-weight property at all.

Stylesheet properties can be aggregated on a single line by listing them as semi?colon separated characteristics. For instance, to set a block of text so that it is both bold and red, you'd use the CSS font-weight and color attributes:

```
<div style="font-weight:bold; color:red">
  This is an example of Cascading Style Sheets in action.
</div>
```

Each of the properties cascade in the same way, and you don't need to change all of the properties simultaneously for any subordinate elements. Thus, to keep the font-weight set to bold but to change the color of "Cascading Style Sheets" to blue, the code would look like this:

```
<div style="font-weight:bold; color:red">
    This is an example of
    <span style="color:blue">Cascading Style Sheets</span>
    in action.
</div>
```

Because of the container arrangement involved with CSS, you can actually apply styles at the <body> level, and they will cascade down as the default styles. For instance, the following XHTML document sets the default font-family for the document to Arial, or Helvetica if that isn't supported, or the system sans-serif font if that isn't supported (this code is available in the code download as example1.htm):

```
<html>
    <head>
        <title>CSS Example 1</title>
    </head>

    <body style="font-family:Arial,Helvetica,sans-serif">
        <h1>CSS Examples</h1>
        <div style="font-weight:bold;">This is an example of
            <span style="font-weight:normal">Cascading Style Sheets</span>
            in action.</div>
        <p>Here's a paragraph.</p>
        <p style="font-family:Times New Roman, Times, serif">
            Here's another paragraph, in a serifed font.</p>
    </body>
</html>
```

This allows for fairly fine grain manipulation of style attributes, but in some respects is not all that much better than the use of the older <font> tag, save that you have even more opportunities to blur the distinction between logical and presentational markup. You can effectively preserve the underlying structure of a document in a browser that doesn't understand the CSS markup, but it is no more scalable than using <font> was.

In order to better control your presentation styles, you should think seriously about defining all of your style rules within a single <style> document element. By doing so, you can radically simplify the amount of maintenance that you need to perform on a web page, and also make it much easier to change the look and feel of a page dynamically, an ability which lays at the core of what was described for a few years as dynamic HTML.

## Centralizing with the <style> Element

The <style> element makes it possible to bring together collections of style attributes that may in turn be applied to a specific element or class of elements. In effect, you can think about the <style> element as being a container for style rules, a separate language that exists outside of HTML proper. Rules are defined using the CSS?selector syntax, as follows:

```
<style>
    selector1 {property1:value1;property2:value2;}
    selector2 {property3:value3; }
    selector3 {} // comment
</style>
```

Selectors can take a number of different forms:

- - The name of an element.** For instance, the p selector will match all <p> paragraph tags.
  - 
  - The name of a class.** An element can have a class attribute that contains the name of the relevant class (or classes, you can have more than one, so long as they are whitespace separated). For instance, the .warning class (the "." period is mandatory) will apply to all elements that have the attribute class="warning".
  - 
  - Wildcard character.** The "\*" character matches all elements in a document. This is useful when needing to assign a consistent font to all elements, for instance. Note that if any rule is given that has a more specific selector (which would be anything other than another "\*") then the more specific rule supersedes the wildcard rule.
  - 
  - Pseudoclasses.** CSS also defines certain pseudoclasses that perform some action beyond rendering the page. For instance, the a:active rule describes how the text of an anchor/link <a> element behaves when the mouse is depressed on the text.
  - 
  - Combined expressions.** Two or more selectors joined together with a space (such as "p b") will apply the rule only to the last elements that are descendants of the preceding elements (for example, applying stylesheets only to those bold <b> tags that are contained within paragraph tags <p>).
  - 
  - ID Hashes.** You can specify an element using an id rather than an element name, as long as you precede the id with a hash. Thus if you have an element <p> with an id attribute of "greeting" then the selector "#greeting" will match this element.
  - 
  - Predicate expressions.** A selector can have a predicate expression (a boolean expression within square brackets) that chooses only those elements for which the predicate is true. For instance, p[id="important"] will select only those paragraph elements which have an attribute named id that has a value "important".

For instance, if you wanted to define all paragraph elements so that they were rendered in a sans-serif font such as Helvetica, you could create a simple rule to do so:

```
<style>
p {font-family:Helvetica}
</style>
```

While you can define just one tag within a stylesheet in this manner, typically stylesheets will contain the definitions of a number of rules. For instance, you could have the following define the basic characteristics of a standard web page:

```
<style>
body
{margin-left:0.5in;margin-right:0.5in;font-family:Helvetica;background-color:lightGray}
h1 {font-size:24pt;}
h2 {font-size:18pt;}
h3 {font-size:14pt;}
p {font-size:11pt;}
</style>
```

The rule definitions (the parts of a rule that are within the "{}" brackets) are whitespace independent?it doesn't matter whether spaces, tabs, or carriage returns are used with the brackets themselves. This makes it easier to format stylesheets visually:

```
<style>
  body {
    margin-left:0.5in;
    margin-right:0.5in;
    font-family:Helvetica;
    background-color:lightGray;
  }
  ...
</style>
```

All <p> tags in the document will then display in Helvetica, except for those <p> tags that explicitly set their font-family to some other font name. Note that the changes will take place regardless of where the stylesheet itself is located in the document?the <style> element does not have to be given prior to the rest of the document to affect it.

You can have more than one rule within a given stylesheet for the same element. If you do have more than one, all the rules for that element act together according to the following guidelines:

1.

If the rules have no properties in common, then the rule affecting the specified element is made up of the union of all rule definitions for that element. For instance:

```
<style>
  p {color:red; font-family:Helvetica}
  p {font-size:11pt;}
</style>
```

is the same as:

```
<style>
  p {color:red; font-family:Helvetica; font-size:11pt;}
</style>
```

2.

If the rules have properties in common, then the last rule defined for a property takes precedence. For instance,

```
<style>
  p {font-size:14pt;}
  p {font-size:11pt;}
</style>
```

is the same as:

```
<style>
  p {font-size:11pt;}
</style>
```

3.

Style rules transcend stylesheets?if you have more than one stylesheet in a document (or if you have one internal and one linked stylesheet), the browser should treat them as if they were all aggregated into a single stylesheet, with internal style rules having precedence over external rules.

Selectors likewise have their own language. For instance, while the typical selector is simply the name of the element, you can similarly specify container relationships. For instance, if you wanted to make italic elements *<i>* red, you'd use the stylesheet:

```
<style>
h1 i {color:red}
</style>
```

However, if you only wanted the *<i>* tag within an *<h1>* primary heading tag to be red but to have the normal behavior everywhere else, you can specify this relationship as:

```
<style>
h1 i {color:red}
</style>
```

Note that this relationship only indicates a container/contained relationship?the *<em>* tag does not have to be an immediate child of the *<h1>* tag, but only a descendant. You can use this to good effect in a multi?level list, as shown below. Other than in the first list case, the *<li>* list elements are in fact not children, but grandchildren, of the preceding *<li>* elements, yet the cascade still works. Note that in this particular case, all list elements at any level still inherit the top?most font-family property, as illustrated in the Try It Out section below.

## Try It Out? Creating Multi?Level Cascades

1.

With a text editor, create the following document and save it in a folder, calling it *cssLists.htm*:

```
<html>
<head>
    <title>Cascading List Elements</title>
    <style>
    </style>
</head>
<body>
    <h1>Cascading List Elements</h1>
    <ul>
        <li>First Level List
            <ul>
                <li>Second Level List
                    <ul>
                        <li>Third Level List
                            <ul>
                                <li>Fourth Level List</li>
                            </ul>
                        </li>
                    </ul>
                </li>
            </ul>
        </li>
    </ul>
</body>
</html>
```

2.

View the HTML file in a browser. It should look something like this:



3.

In the <style> block, add the following highlighted CSS code and save as cssLists2.htm

```
<html>
<head>
    <title>Cascading List Elements</title>
    <style>
        ul li {font-size:14pt;font-family:Helvetica}
        ul li li {font-size:12pt;}
        ul li li li {font-size:10pt;}
        ul li li li li {font-size:8pt;}
    </style>
</head>
```

```
<body>
    <h1>Cascading List Elements</h1>
    <ul>
        <li>First Level List
            <ul>
                <li>Second Level List
                    <ul>
                        <li>Third Level List
                            <ul>
                                <li>Fourth Level List</li>
                            </ul>
                        </li>
                    </ul>
                </li>
            </ul>
        </li>
    </ul>
</body>
</html>
```

4.

View the new html file cssLists2.htm. It should look something like this:



5.

Set the CSS font-style property of the rule ul li li to "italic" to see how different properties cascade.

```
<style>
```

```
ul li {font-size:14pt; font-family:Helvetica}
ul li li {font-size:12pt; font-style:italic}
ul li li li {font-size:10pt;}
ul li li li li {font-size:8pt;}
</style>
```

6.

Save the file as cssLists3.htm and view it in your browser. You should see the following:



This ability to set up cascades lets you design rules that work in very specialized circumstances. However, the biggest problem that you deal with in creating generalized rules on HTML elements comes when the same elements (such as a set of nested lists, or a series of paragraph tags) are used in different ways in different parts of the document. For instance, one part of the document may use lists for building nested trees (such as those that Internet Explorer uses to display XML documents) but it may be used elsewhere for a table of contents. This is one of the reasons that CSS introduces the concept of **classes**.

## Working with Classes

How many times have you ever needed an element for citations? Or a dictionary definition? Elements for both of these purposes have existed in HTML because of its origins, but neither is exactly commonly used.

*Just for reference, the dictionary elements <dl>, <dt>, and <dd> describe a dictionary definition listing, a definition title, and a definition description, respectively, while the <cite> element lets you include an explicit citation in your web page.*

On the other hand, if you're writing a book on programming, then it might be handy to have both such constructs as well as a <caption> element, a <figure> element, and an <example> element, not to mention straightforward elements like <chapter>, <section>, <subsection>, <sidebar>, <pullquote>, and so forth, none of which exists in HTML. One answer to this conundrum of needing something that HTML does not provide is to create a new XML schema and attach <style> elements to each node in the schema (something discussed in the [next section](#)), but if you still want to write HTML code that will work for 99.995% of all the browsers on the planet, XML doesn't have quite the same heft and reach (though that is changing, thankfully).

The class attribute was devised as an interim solution to this particular problem, and like CSS itself predates XML by a couple of years. The idea behind class is simple?instead of assigning a rule to an existing HTML element, you could instead assign a rule to an arbitrary named class, then associate the class with one or more elements. For instance, suppose that you wanted to add an arbitrary note to your HTML that would set the left margin a half?inch in, and add a pointer graphic to the left of the block itself.

```
<html>
  <head>
    <title>Pointer Note</title>
    <style>
      .note {font-family:Arial,Helvetica; margin-left:0.34in;}
      .note:before {content:url("pointer.gif") "Note:"}
      .note:first-line {margin-left:-0.34in;}
    </style>
  </head>
```

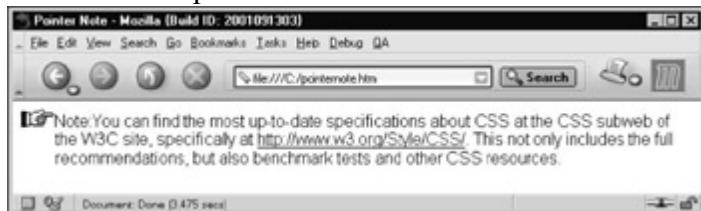
```
<body>

<div class="note">You can find the most up-to-date specifications about
CSS at the CSS subweb of the W3C site, specifically at
<a href="http://www.w3.org/Style/CSS/">http://www.w3.org/Style/CSS/</a>.
This not only includes the full recommendations, but also benchmark
tests and other CSS resources.
</div>

</body>
</html>
```

*Note: to run this code effectively, you will need to save a graphic called pointer.gif in the same folder as this file. There's one available to you in the code download for this chapter.*

Currently, IE 5+ and IE 6 do not recognize the :before or :first-line extension (which lets you place content before the start of an HTML or XHTML element), nor does Opera, but Mozilla does. The screenshot below of the output in Mozilla illustrates the note, along with its associated hand pointer. Internet Explorer will print the note text indented, but without the pointer.



Creating a class involves two steps. The first requires declaring the class in a stylesheet as you would any other selector, except that class names always start with a period ("."). Thus, declaring a simple note would look like this:

```
<style>
  .note { font-family:Arial,Helvetica;
          margin-left:0.34in;
        }
</style>
```

The .note class is then applied to an element through the class attribute:

```
<div class="note">This is a note.</div>
```

Unless explicitly declared otherwise, a class can be applied to any element, and the CSS properties within that class will override the same properties that are implicit in the class itself.

```
<h1 class="note">This is a Header 1 Note.</h1>
<p class="note">This is a Paragraph Note.</p>
```

To limit a class so that it only applies to one specific element type, include the name of that element in the selector declaration:

```
<style>
  h1.note { font-family:Arial,Helvetica;
            margin-left:0.34in;
            font-size:24pt;
            color:blue;
          }
  p.note { font-family:Arial,Helvetica;
            margin-left:0.34in;
          }
```

```
    font-size:11pt;
    color:red;
}
</style>
```

This actually displays a note differently depending upon the element that the class is applied to. A note class attached to an `<h1>` tag will be blue 24 pt Helvetica, while the same note class applied to a `<p>` tag will be red 11pt Helvetica (or Arial, if viewed on a Windows system). Put another way, the note class imposes its own semantic meaning to the tag.

An element can actually have more than one class applied to it. For instance, suppose that you had the note class, but you also had one class for comments directed to the student (called student) and another for comments directed to the teacher (called, not surprisingly, teacher). You can apply both the note and the student classes by giving the names of the classes separated by whitespace. For instance, the class attribute that would include references to both note and student would look like:

```
<p class="note student">This is a note directed to students.</p>
```

In general, the following rule applies:

**Important**

You can apply any number of classes to an element by separating them with whitespace, as `class="class1 class2 class3 [etc.]"`. Overlapping properties defined in later classes will supercede those in earlier classes.

Let's put this to the test in our next Try It Out.

## Try It Out?Adding Multiple Classes to Elements

1.

With a text editor, create the following document and save it in a folder, naming it `studentTeacher1.htm`. View the file in your favorite browser:

```
<html>
<head>
  <title>Student Teacher 1 </title>
  <style>
  </style>
</head>

<body>

  <p>You can find the most up-to-date specifications about CSS at the CSS
  subweb of the W3C site, specifically at
  <a href="http://www.w3.org/Style/CSS/">http://www.w3.org/Style/CSS/</a>.
  This not only includes the full recommendations, but also benchmark
  tests and other CSS resources.
</p>

  <p>Make sure that you indicate that there is more than just the
  recommendation at this web site.
</p>
</body>
</html>
```

2.

In the <style> block, add a new note class, then add note attributes to both paragraphs. Save as studentTeacher2.htm and view.

```
<html>
<head>
    <title>Student Teacher 2 </title>
    <style>
        .note {font-family:Arial,Helvetica;
               margin-left:0.34in;
               color:green;
        }
    </style>
</head>

<body>
    <p class="note" >You can find the most up-to-date specifications about
       CSS at the CSS subweb of the W3C site, specifically at
       <a href="http://www.w3.org/Style/CSS/">http://www.w3.org/Style/CSS/</a>.
       This not only includes the full recommendations, but also benchmark
       tests and other CSS resources.
    </p>

    <p class="note" >Make sure that you indicate that there is more than
       just the recommendation at this web site.
    </p>
</body>
</html>
```

3.

Create a graphic bullet (such as the hand in the earlier Try It Out, pointer.gif). Save this in the same folder as your HTML pages.

4.

In the <style> block, add two new sub classes .note:before and .note:first-line. Save as studentTeacher3.htm and view, noting the difference in different browsers (try this with Opera and Mozilla, especially).

```
<html>
<head>
    <title>Student Teacher 1 </title>
    <style>
        .note {font-family:Arial,Helvetica;
               margin-left:0.34in;
               color:green;
        }
        .note:before {content:url("pointer.gif") "Note:"}
        .note:first-line {margin-left:-0.34in;}
    </style>
</head>

<body>
    <p class="note" >You can find the most up-to-date specifications about
       CSS at the CSS subweb of the W3C site, specifically at
       <a href="http://www.w3.org/Style/CSS/">http://www.w3.org/Style/CSS/</a>.
       This not only includes the full recommendations, but also benchmark
       tests and other CSS resources.
    </p>

    <p class="note" >Make sure that you indicate that there is more than
       just the recommendation at this web site.
    </p>
</body>
</html>
```

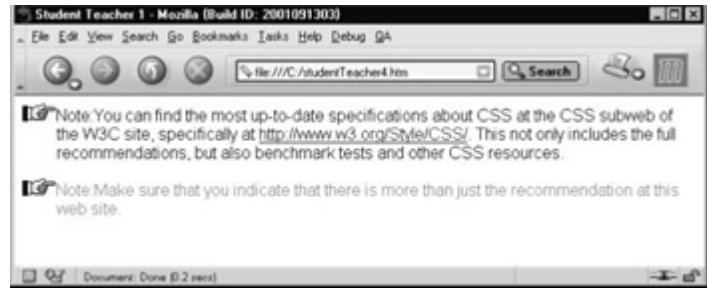
5.

Finally, in the <style> block, add two classes student and teacher, then add a student and a teacher class to the indicated paragraphs. Save as studentTeacher4.htm and view in various browsers.

```
<html>
<head>
    <title>Student Teacher 1 </title>
    <style>
        .note {font-family:Arial,Helvetica;
            margin-left:0.34in;
            color:green;}
        .note:before {content:url("pointer.gif") "Note:"}
        .note:first-line {margin-left:-0.34in;}
        .student {color:black;}
        .teacher {color:gray;}
    </style>
</head>
<body>
    <p class="note student" >You can find the most up-to-date specifications
        about CSS at the CSS subweb of the W3C site, specifically at
        <a href="http://www.w3.org/Style/CSS/">http://www.w3.org/Style/CSS/</a>.
        This not only includes the full recommendations, but also benchmark
        tests and other CSS resources.
    </p>

    <p class="note teacher" >Make sure that you indicate that there is
        more than just the recommendation at this web site.
    </p>
</body>
</html>
```

StudentTeacher4.htm illustrates how an element can have more than one class associated with it. Here's what it looks like in Mozilla:



## How It Works

In general, when more than one class is assigned to an element, the last CSS properties that are declared in the list are the ones put into effect. Thus, in the "note teacher" combination, the note class has an assigned color of green, while the teacher has an assigned color of gray. Because teacher came after note in the listing, the element will be gray; however, if the order had been inverted to "teacher note" the resulting color becomes green instead.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Deploying CSS with XML

There is a certain amount of justification in saying that XML owes at least some of its existence to the limitations of CSS. Cascading Style Sheets provided an alternative mechanism that made it possible to create alternative "tags" in HTML documents (both HTML 4.01 and XHTML), but these classes were awkward to work with and had no real cohesiveness (they couldn't be validated, for instance). Finally, classes had no real knowledge of or ability to affect the contents of the element that they were attached to.

Of course, many SGML experts had been pushing for a lightweight version of SGML for years before CSS came on the scene. The XML language, which provided a far more consistent, structured framework, effectively solved one of the major problems that CSS had been designed to address—adding alternative meanings to web pages—yet they introduced another for which CSS was ideally suited.

An XML document contains nothing but logical structure and content. Indeed, it is sometimes handy to see XHTML as an XML language that programs a browser—in essence, a device language. Because XML doesn't have any specific "devices" that it targets (it is, after all, a meta language for building device languages, among other things), a web browser wouldn't know what to do with it. However, with CSS you can create a set of definitions that tell the browser how to interpret a given XML document as markup.

A CSS stylesheet for XML differs somewhat from the XHTML variety in that the mechanism to associate a stylesheet with an XML document has to be imposed from outside the containing document element itself. Specifically, in order to apply a stylesheet to a document, you have to use the same processing directive that is used for XSL, with the exception that the type attribute is set to "text/css" instead of "text/xsl":

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

```

This directive instructs the XML parser (which is the same, typically, as the XHTML browser) to use the name of elements or the name of classes attached to those elements via attributes as rules for output. For instance, consider a basic XML article (in this case *Will Web Services Break the Semantic Web?*, by Kurt Cagle):

```
<?xml-stylesheet type="text/css" href="article.css"?>
<document xmlns:xlink="http://www.w3.org/1999/xlink">
<banner>Recursions and Ruminations</banner>
<page>
  <header>
    <title>Will Web Services Break the Semantic Web?</title>
    <author>Kurt Cagle</author>
    <primaryLink>http://www.kurtcagle.net/articles/brokenSemantics.xml
      </primaryLink>
  </header>
  <body>
    <para>The Semantic Web has been a central concept in web development
      since Tim Berners Lee's seminal paper on the subject, <jump
      xlink:href="http://www.w3.org/DesignIssues/Semantic.html"
      xlink:type="simple" xlink:show="replace">A roadmap to the Semantic
      Web</jump> in 1998 (with a more popular description of the Semantic
      Web in <jump xlink:href="http://www.scientificamerican.com/2001/
      0501issue/0501berners-lee.html" xlink:type="simple"
      xlink:show="replace">Scientific American</jump> in 1999. In it he
      posited that while the World Wide Web has become largely readable to
      people, no consistent mechanism existed for being able to create
      associations between different types of data that could be used to

```

make inferences. The ability to identify a block of data as being a "thing" of a certain type is crucial to being able to manipulate that thing. Thus, ultimately, the purpose of the Semantic Web (as he saw it) was to provide an architecture that would not only make it possible to objectify the web but to do so in a way that required relatively little human intervention.

</para>

<para>The original vision that Berners-Lee had of this Semantic web depended upon a standard that is still relatively obscure today, though it is growing in popularity. The Resource Description Framework language, or RDF, can seem fairly abstruse -- it's primary purpose is to provide a framework for describing the associations between one or more resources on the Web. A simple relationship, <term>equivalence</term> for instance, could be used to indicate that an XML element in one schema (such as a <element>lastname</element>) was equivalent to a <element>familyName</element> element in a different schema. By having this relationship clarified, it means that if a program could read <element>lastname</element> elements, they could work with <element>familyName</element> elements in the same way, even if the rest of the schema was very different.

</para>

<para>One consequence of such a relationship mechanism is that the disparate data stores can be used to aggregate and filter data in a much more transparent fashion. Through such a system, for instance, you can create an aggregate bibliography of everything a person may have written on the Web, for instance, making it possible to build a comprehensive portfolio of their works in ways that would be almost impossible to do on the Web. It also makes patterns far more obvious, as an RDF-centric Internet tends to emphasize the relationships between entities, rather than the specific characteristics of the entities themselves.

</para>

<!-- additional content -->

</body>

</page>

</document>

The processing directive at the top of the document informs the browser that the stylesheet for this document is of type "text/css", which means that it should be interpreted as CSS text, and that the document should pull in the CSS file article.css to apply the styles:

```
document {display:block;}
page { display:block;
        font-family:Arial;
        margin:0.5in;
        width:400px
    }
banner { display:block;
        position:absolute;
        left:0;
        top:0;
        width:100%;
        height:0.35in;
        background-color:navy;
        color:white;
        font-family:Brush Script, Fantasy;
        font-size:16pt;
    }
header title{ width:400px;
        color:navy;
        padding:5px;
        font-family:Brush Script, Fantasy;
```

```
font-size:32pt;
display:block;
}
header author{ color:navy;
padding:5px;
font-family:Brush Script,Fantasy;
font-size:18pt;
display:block;
}
header author:before {content:"by "}
body {font-family:Times New Roman,Times,serif; display:block;}
para {display:block; margin-bottom:1%; }
jump {display:inline; font-size:110%; }
jump:link {color:blue; }
jump:hover {color:red; }
jump:visited {color:navy; }
element { display:inline;
font-family:Courier New,courier,fixed;
font-size:10pt; }
element:before {content:"\003C"}
element:after {content:"\003E"}
note { font-family:Arial,Helvetica;
margin-left:0.34in;
display:block;
margin-bottom:2%;
font-size:9pt; }
note:before {content:url("pointer.gif") "Note:"}
note:first-line {margin-left:-0.34in; }
primaryLink {display:none; }
```

This creates the following result, when viewed in Mozilla:



At first glance, the format used in the code looks slightly different because class names are not in evidence. However, the language is exactly the same: the selectors are designed for matching elements, not classes. Indeed, while it is possible to apply classes to XML output, this is in fact not recommended in general because it is both redundant and potentially conflicting, since an XML schema may very well define a class attribute that has absolutely nothing to do with CSS.

Another fairly significant difference between CSS formatted for HTML as opposed to that for XML is that most HTML elements have intrinsically defined **flow models**. The canvas that XHTML and XML are both painted on works in a flow fashion-the browser defines a specific flow direction that indicates how both individual characters and blocks of content are painted. For instance, most European languages flow from left-to-right, while most Middle-Eastern languages flow from right-to-left (and Japanese and Chinese flow from top-to-bottom, with a

rightward bias).

The flow models use this ordering along with specifications indicating what kind of container the content flows into. There are four primary containers according to the CSS Level 2 specification, (plus a number of secondary containers, not mentioned here) that are set by the CSS display property:

- - block: The block value creates a bounding box around the content, so that any new content ends up appearing outside (usually below) the bounding block in question. The <div> or <p> HTML elements are the prototypical block elements.
  - 
  - inline: An inline element appears as part of the flow, rather than being in a separate bounding box. <span> or <b> elements are the prototypical inline elements.
  - 
  - run-in: A run-in element combines the capabilities of a block and inline element. If the element immediately following a run-in element is a block (which doesn't float and isn't absolutely positioned) then the run-in box becomes the first inline element of the box; otherwise it acts as a block element.
  - 
  - table: The need to display tables occurs often in both XHTML and XML, but because a <table> element has no explicit meaning in an XML document, it (and its associated component properties, such as table-row and table-cell) has been made into a CSS property.
  - 
  - none: The display:none property indicates to the renderer that the content of that element should not be sent to the page (see the *Hiding Content* section, later in this chapter).

Typically, one of the major tasks in writing CSS for XML is in explicitly setting up the display type of each element. The display property is one of the few CSS properties where the child elements do not inherit the parent's properties-the reason for this has to do with the problem of having XML elements that do not have any explicit CSS rule associated with them. If all such elements inherited the block value implicitly, then any text with no rule will appear on its own line. On the other hand, with the default (inline) when an element is not defined it can still be output to the browser without interrupting the flow of text.

## Integrating Position

While the display property dictates the type of flow model being used, the position property sets whether positioning coordinates (such as left or top) apply relative to the parent container the content is in or are given as absolute coordinates. The position property can also take a number of values:

- - static: The static position is the default behavior, and basically sets the position of the element in question to the next appropriate place in the flow. In most browser models, static positioning is the most efficient, but you can't change coordinates of a static element once initially drawn unless you set the position property to "relative" first.
  - 
  - relative: Relative elements start out identically to static elements, but you can change their coordinate values. All coordinate positions are given relative to the starting point of the parent containing element, not relative to the page as a whole.
  -

absolute: Use absolute coordinates when you want to place an element on a page relative to the upper left-hand corner of the browser display window. If no coordinates are given, the element is positioned in their default flow position.

Note that some display and position modes are somewhat contradictory. For instance, while you can have an inline element with an absolute set position, the element acts in this case like a block element.

You can use absolute positions to set an element on a page so that it works independently of the flow of the rest of the page. For example, in the article given above, a banner runs across the top of the page that is actually outside of the margins set for the <page> element. This was accomplished by setting the position of the <banner> element to absolute, assigning the value 0 to both left and top properties, and setting the width and height to 100% of the page and 0.35 in. respectively.

The CSS model is sometimes described as being 2.5 D. In addition to being able to set the position of an element on the display plane, it is also possible to position an element above or below the working plane through the z-index property. The basic document is assumed to have a default z-index value of 0, but by setting z-index to a positive value you can cause the content of that element to "float" in front, obscuring what lies beneath. On the other hand, by setting the z-index to a negative value, it will appear behind the rest of the text.

## Hiding Content

Sometimes you don't want content to appear on a page. For instance, in the *Broken Semantics* article, one element, <primaryLink>, contains contents that are not appropriate to be displayed, since this link tells where the master version of the article is located. You can hide this link in one of two ways: by either setting the display property of the element to "none" or by setting the visibility property to "hidden". However, while these may seem the same, in fact they do two very different things.

The display:none setting effectively removes the element from the page flow altogether. Elements with display:none cannot receive any events, although you can usually manipulate the CSS attributes from within the relevant supported scripting model.

The visibility:hidden technique, on the other hand, keeps the content in the flow model but does not draw the characters themselves. If the display property is set to "inline" and visibility is set to "hidden" (as opposed to "visible"), there would be an empty region where the text would have been. The element would still receive any mouse or keyboard events that it would receive otherwise.

## Creating Tables in CSS

Being able to create tables using CSS solves one of the thornier problems plaguing the use of XML. Tables were introduced into HTML because they solved a very real need: a significant amount of all data can be most easily expressed in tabular form. Unfortunately, tables are somewhat problematic, because they contain no semantic information; a <tr>, or table row, element begs the question "A row of what?"

This becomes an even bigger issue with XML, since you no longer have pre-defined <table>, <tr>, <td>, etc. elements to format your content. The set of display:table properties performs this function and solves the problem of semantics at the same time. The elements in a data set need not know that they are being expressed in a table format, since this information is carried at the CSS level. Tables move from logical to presentation structures, where they belong.

One of the really quirky things about writing about XML is that after a while things tend to become very self-referential. I needed a table to illustrate the various table display properties, so, it seemed only logical that the example should in fact be the very table that describes itself. A simple XML document (tableproperties.xml) describes the contents of the table:

```
<?xml-stylesheet type="text/css" href="tableproperties.css"?>
<cssproperties>
  <headers>
    <header id="propName">Property Name</header>
    <header id="propDesc">Property Description</header>
    <header id="analog">HTML Analog</header>
  </headers>
  <properties>
    <property>
      <propName>table</propName>
      <propDesc>Provides the containing element for a property</propDesc>
      <analog>table</analog>
    </property>
    <property>
      <propName>table-row</propName>
      <propDesc>Creates a standard row for table cells</propDesc>
      <analog>tr</analog>
    </property>
    <property>
      <propName>table-header-group</propName>
      <propDesc>Creates the head element that column header elements may be
            displayed in</propDesc>
      <analog>thead</analog>
    </property>
    <property>
      <propName>table-row-group</propName>
      <propDesc>Creates the general body of the table, where all of the data
            is found.</propDesc>
      <analog>tbody</analog>
    </property>
    <property>
      <propName>table-footer-group</propName>
      <propDesc>Creates a footer for the table.</propDesc>
      <analog>tfoot</analog>
    </property>
    <property>
      <propName>table-column</propName>
      <propDesc>Creates a specialized column element.</propDesc>
      <analog>col</analog>
    </property>
    <property>
      <propName>table-column-group</propName>
      <propDesc>Creates a group of columns.</propDesc>
      <analog>colgroup</analog>
    </property>
    <property>
      <propName>table-cell</propName>
      <propDesc>Creates a cell within a row.</propDesc>
      <analog>td</analog>
    </property>
    <property>
      <propName>table-cell</propName>
      <propDesc>It can also create a cell in a table header or table
            footer.</propDesc>
      <analog>th</analog>
    </property>
    <property>
      <propName>table-caption</propName>
      <propDesc>Creates a caption for the table.</propDesc>
      <analog>caption</analog>
    </property>
  </properties>
</cssproperties>
```

By applying the tableproperties.css stylesheet listed below, which maps various elements from the tableproperties.xml file to their corresponding table format, you can actually create a table showing the properties of tables:

```
cssproperties {display:table; border:inset 3px black; }
headers {display:table-header-group; }
header { display:table-cell;
          padding:5px;
          background-color:navy;
          color:white;
          font-weight:bold;
          border:outset 2px black;
      }
properties {display:table-row-group}
property {display:table-row; }
propName { display:table-cell;
          padding:5px;
          border-bottom:solid 1px lightGray;
      }
propDesc { display:table-cell;
          padding:5px;
          border-bottom:solid 1px lightGray;
      }
analog { display:table-cell;
          border-bottom:solid 1px lightGray;
      }
analog:before {content:"\003C"}
analog:after {content:"\003E"}
```

which looks like this when viewed in Mozilla:

A screenshot of a Mozilla browser window displaying a table. The table has three columns: 'Property Name', 'Property Description', and 'HTML Analog'. The rows list various CSS properties and their corresponding HTML elements or descriptions.

Property Name	Property Description	HTML Analog
table	Provides the containing element for a property	<table>
table-row	Creates a standard row for table cells	<tr>
table-header-group	Creates the head element that column.header elements may be displayed in	<thead>
table-row-group	Creates the general body of the table, where all of the data is found	<tbody>
table-footer-group	Creates a footer for the table	<tfoot>
table-column	Creates a specialized column element	<col>
table-column-group	Creates a group of columns	<colgroup>
table-cell	Creates a cell within a row	<td>
table-cell	It can also create a cell in a table header or table footer	<th>
table-caption	Creates a caption for the table	<caption>

As an important caveat, the table properties currently work with both the Opera 5, Netscape 6.1 and Mozilla 0.92 and higher browsers (and with the Gecko in Netscape 6.1), but not with Internet Explorer 6.0. So you should generally only utilize this in situations where you know your clients are going to be using one of the other browsers. Indeed, this naturally leads into a discussion about CSS and XSLT, since one of the primary limitations that CSS currently faces is the fact that the leading Internet browser (Internet Explorer) only provides full support for CSS level 1 (the various table properties are all part of CSS level 2, which IE 6 only partially supports). What this often means in practice is that, at least with regard to XML, the preferable solution to produce output is to use XSLT on the server to generate CSS-enhanced XHTML.

## Working with Complementary Stylesheet Languages: CSS and XSLT

When CSS was first debated as a stylesheet language, it was (and is) not the only contender. Another faction on the W3C pushed hard for a very different approach to stylesheets, one based upon Document Style Semantics and Specification Language (DSSSL), which was the stylesheet language for SGML. This language was designed to

perform more transformative structuring of content, making it possible to create alternative tags that could then be transformed into a new output grammar that would more precisely describe a web or print page in traditional markup language terms that were more consistent with Postscript than HTML.

This new language was called the Extensible Stylesheet Language, or XSL, and it consisted of an XML based transformation language called XSLT (XSL for Transformations) and a page description language, XSL-FO (XSL Formatting Objects). Somewhere along the line, however, XSLT became a superstar.

XSLT is covered extensively earlier in this book, in [Chapter 4](#). Its ability to transform XML content from one schema instance to another meant that people quickly adopted it to handle data transformations, integration of resources, and far more. Meanwhile, XSL-FO has limped along, used in a few fairly limited circumstances, although increasingly finding a place in the print production arena.

XSLT and CSS can actually act as complementary technologies. One role that XSLT has is as a way of mapping one structure to a different format. Especially because not all browsers currently support the more powerful capabilities of CSS level 2, it is sometimes preferable to transform an XML structure into XHTML, and either embed or reference classes in the resulting output that can be transformed.

There are a few problems that are not easy to solve just using CSS; the table example given earlier would not work in Internet Explorer 6.0. However, you can use XSLT to transform the content into an XHTML table.

## Try It Out-XML with an XSLT Stylesheet

Suppose, though, that in addition to just duplicating the table, you wanted to have each row's background alternate in color between white and light blue, making the table easier to read. We'll create an XSLT stylesheet to demonstrate how this can be accomplished by assigning the relevant classes dynamically according to the row position.

1.

Create the following document in your text editor and save it as transformTable.xsl.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output media-type="text/html" method="xml"/>

<xsl:template match="/">
<html>
<head>
<title>Table Properties</title>
</head>
<body>
<xsl:apply-templates select="cssproperties"/>
</body>
</html>
</xsl:template>

<xsl:template match="cssproperties">
<style>
.cssproperties {table; border: inset 3px black; }
.headers {display:table-header-group; }
.header { display:table-cell;
padding:5px;
background-color:navy;
color:white;
font-weight:bold;
border: outset 2px black; }
.properties {display:table-row-group}
.property {display:table-row; }
.propName {display:table-cell;padding:5px; }
.propDesc {display:table-cell;padding:5px; }
```

```
.analog {display:table-cell;}  
.row0 {background-color:lightBlue;}  
.row1 {background-color:white;}  
</style>  
  
<table class="cssproperties">  
  <xsl:apply-templates select="headers"/>  
  <xsl:apply-templates select="properties"/>  
</table>  
</xsl:template>  
  
<xsl:template match="headers">  
  <thead class="{name(.)}">  
    <xsl:apply-templates select="header"/>  
  </thead>  
</xsl:template>  
  
<xsl:template match="header">  
  <th class="{name(.)}"><xsl:value-of select="."/></th>  
</xsl:template>  
  
<xsl:template match="properties">  
  <tbody class="{name(.)}">  
    <xsl:apply-templates select="property"/>  
  </tbody>  
</xsl:template>  
  
<xsl:template match="property">  
  <tr class="{name(.)} row{position() mod 2}">  
    <xsl:apply-templates select="*" mode="properties"/>  
  </tr>  
</xsl:template>  
  
<xsl:template match="*" mode="properties">  
  <td class="{name(.)}">  
    <xsl:apply-templates select="."/>  
  </td>  
</xsl:template>  
  
<xsl:template match="analog" mode="properties">  
  <td class="{name(.)}">  
    &lt;<xsl:apply-templates select="."/>&gt;  
  </td>  
</xsl:template>  
  
</xsl:stylesheet>
```

## 2.

Now we need to apply this stylesheet to our XML, so open up the tableproperties.xml file we created earlier and alter the line:

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

to

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

Save the file as tableproperties\_xsl.xml in the same folder as your transformTable.xsl file.

## 3.

Now open up tableproperties.xml in IE and you should see results like this:



## How It Works

The stylesheet is invoked by the line:

```
<xsl:stylesheet type="text/xsl" href="transformTable.xsl"?>
```

in our XML file.

The interaction between CSS classes and XSLT are demonstrated in a number of places throughout this code. For instance, the template match for the property node,

```
<xsl:template match="property">
  <tr class="{name(.)} row{position() mod 2}">
    <xsl:apply-templates select="*" mode="properties"/>
  </tr>
</xsl:template>
```

uses attribute-based XPath evaluation (within the {} brackets) to write a class attribute with two class names. The first-"`{name(.)}`"-actually uses the name of the node itself as the class name, while the second class name- "`row{position() mod 2}`" queries the XSLT position() function to get the row number then applies modulus 2 to retrieve a value of either 0 (for even rows) or 1 (for odd rows). So, the fifth row would have a class attribute value of

```
<tr class="property row1">
```

Through the intelligent division of class properties, you can effectively create a very dynamic interface while simultaneously manage to maintain visual consistency between web pages generated via XSLT. You can also use the XSLT document() function to encapsulate stylesheet sets in a single XML document, and then incorporate them into your transformed output dynamically. Our next Try It Out gives an example of how this might be done.

## Try It Out-Applying Different Themes with CSS and XSLT

Here we'll create two CSS stylesheets that are stylistically similar, but one uses a red theme while the other uses a blue theme. We'll then incorporate them into a single, referenceable XML document with each stylesheet referenced by an appropriate id.

1.

Open up your text editor and create the document shown below. Save it as themes.xml:

```
<stylesheets>
<style id="blue">//<! [CDATA[
.cssproperties {table;border:inset 3px black;}
.headers {display:table-header-group;}
.header { display:table-cell;
padding:5px;
background-color:navy;
color:white;
```

```
        font-weight:bold;
        border:outset 2px black;
    }
.properties {display:table-row-group}
.property {display:table-row;}
.propName {display:table-cell;padding:5px;}
.propDesc {display:table-cell;padding:5px;}
.analog {display:table-cell;}
.row0 {background-color:lightBlue;}
.row1 {background-color:white;}
]]></style>

<style id="red">//<![CDATA[
.cssproperties {table;border:inset 3px red;}
.headers {display:table-header-group;}
.header { display:table-cell;
    padding:5px;
    background-color:maroon;
    color:white;
    font-weight:bold;
    border:outset 2px black;
}
.properties {display:table-row-group}
.property {display:table-row;}
.propName {display:table-cell;padding:5px;}
.propDesc {display:table-cell;padding:5px;}
.analog {display:table-cell;}
.row0 {background-color:orange;}
.row1 {background-color:white;}
]]></style>
</stylesheets>
```

## 2.

By putting multiple stylesheets into a single XML document, you can retrieve a particular "theme" using parameters. Save the following code that uses this method as transformTable2.xsl in the same folder as themes.xml:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output media-type="text/html" method="xml"/>
<xsl:param name="theme" select="'red'"/>
<xsl:template match="/">
<html>
    <head>
        <title>Table Properties</title>
    </head>
    <body>
        <xsl:apply-templates select="cssproperties"/>
    </body>
</html>
</xsl:template>

<xsl:template match="cssproperties">
<style>
    <xsl:value-of
        select="document('themes.xml')/stylesheets/style[@id=$theme]"/>
</style>
<table class="cssproperties">
    <xsl:apply-templates select="headers"/>
    <xsl:apply-templates select="properties"/>
</table>
</xsl:template>

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

```
<thead class="{name(.)}">
  <xsl:apply-templates select="header"/>
</thead>
</xsl:template>

<xsl:template match="header">
  <th class="{name(.)}"><xsl:value-of select="."/>/</th>
</xsl:template>

<xsl:template match="properties">
  <tbody class="{name(.)}">
    <xsl:apply-templates select="property"/>
  </tbody>
</xsl:template>

<xsl:template match="property">
  <tr class="{name(.)} row{position() mod 2}">
    <xsl:apply-templates select="*" mode="properties"/>
  </tr>
</xsl:template>

<xsl:template match="*" mode="properties">
  <td class="{name(.)}">
    <xsl:apply-templates select="."/>/
  </td>
</xsl:template>

<xsl:template match="analog" mode="properties">
  <td class="{name(.)}">
    &lt;<xsl:apply-templates select="."/>/&gt;
  </td>
</xsl:template>

</xsl:stylesheet>
```

### 3.

We now need to reference our new stylesheet in the XML file so open up tableproperties\_xsl.xml and change the line:

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

to:

```
<?xml-stylesheet type="text/xsl" href="transformTable2.xsl"?>
```

and save the file as tableproperties\_xsl2.xml.

### 4.

View the XML file in IE and you should see that the formatting from the red themed stylesheet appears. To prove the point, you might like to change the value red in the line:

```
<xsl:param name="theme" select="'red'"/>
```

to blue and view the differences.

## How It Works

The XSLT \$theme parameter can here take the value of "red" or "blue", which will in turn extract the appropriate style block from the themes.xml file, retrieved via the document() function:

```
<xsl:template match="cssproperties">
<style>
<xsl:value-of
  select="document('themes.xml')/stylesheets/style[@id=$theme]"/>
</style>
<table class="cssproperties">
  <xsl:apply-templates select="headers"/>
  <xsl:apply-templates select="properties"/>
</table>
</xsl:template>
```

This makes it possible to do any number of layout changes:

- changing color themes or backgrounds graphics
- changing the positions of various elements on the screen
- adding interactive capabilities (such as behaviors in Internet Explorer 5+)
- or pulling in more sophisticated encoding (such as Scalable Vector Graphics (SVG) graphic integration in Mozilla)

Indeed, by making the effort to separate the logical transformation layer (done via XSLT) from the presentation layer (handled largely by CSS) you are able to lay the foundations for extremely dynamic graphical user interfaces that are exclusively web based, that can change, in response to people's access levels, the time of day or their moods, and that more importantly can also change in response to the kind of data that is being presented.

---

[!\[\]\(7366eb85cf62ce28da681a7ca672d910\_img.jpg\) PREVIOUS](#)

[<Free Open Study>](#)

[!\[\]\(523a4ffe59ab1f393c3d9e6aa0682222\_img.jpg\) NEXT](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# CSS and the Future of XML

Cascading Style Sheets exist in an awkward counterpoint to the tag and attribute nature of XML itself and, to a certain extent, its syntax can be seen as being a legacy application of an earlier technology that is fast becoming obsolete. However, the need that CSS fills is a very real one: assigning a media representation to the logical structures of XML and XHTML. In the CSS 3 Working Group documents, this role is being explored to its logical conclusion.

Part of the thrust of CSS 3 is to take the complex language that has evolved over the last six years and make it more modular. By breaking the specification into distinct building blocks, it becomes possible to customize CSS for various user agents (in other words, web browsers, mobile phones, PDAs, and so forth) so that they can more effectively handle implementations within system limitations. As with HTML migrating to XHTML, by breaking the CSS implementations into XML-ized components makes it easier to update the capabilities of CSS more or less in real time, rather than waiting, sometimes for years, for the W3C committee to agree to add a particular property to the specification.

CSS is becoming an abstract grammar that more adequately defines media representation. The SVG language is built predominantly on various CSS properties, and it is likewise affecting the future evolution of the language. Similarly, CSS is being tied into the creation of speech capable (and speech aware) interfaces. CSS 3 currently defines a very rich speech interface that makes it possible to control most aspects of spoken markup code.

Ultimately, CSS may very well end up disappearing entirely as a distinct language, a trend that is already becoming apparent. Instead, it is moving into a framework mode where CSS properties (and the manner in which they interoperate with other XML standards) simply become yet another part of the XML language.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

In working with CSS, then, the principle ideas to remember from this chapter are:

- **Separating Presentation from Structure.** The purpose of CSS is to make the presentation of the web pages independent of the HTML, XHTML or XML that underlies it.
- **Work from General to Specific.** Start your pages with a well thought-out stylesheet and use these global styles as much as possible to maintain portability and ease of development. Use embedded stylesheets for customization on this general theme, and use the style attribute only if you can't justify creating an explicit class.
- **Use Complementary Style Technology.** XSLT is a transformative technology that converts an XML structure into an XHTML one. Use CSS globally (through classes) to keep the media presentation as decoupled from the resultant structure as possible.
- **Remember Non-CSS Browsers.** Before you post a web page, remove the CSS and see if it is still intelligible. If it's not, then you may have too much structure creeping into your CSS.

Cascading Style Sheets provide a means of taking the confusing mix of logic and presentation that has defined HTML over the years and make it more capable of being used in a logical capacity alone. However, the primary reason that HTML has such staying power is that it is primarily a legacy language that people have learned, that vendors have created tools for, and that the architecture has been designed for. As the Web moves increasingly into an XML-based dialect, CSS will become increasingly important as a means not just of adding some additional media characteristics to an already media rich language, but also to becoming *the* media language, a general form language that can be used in a wide number of devices. Significantly, even cell phone manufacturers are moving beyond the Wireless Markup Language (WML) and moving toward the combination of XHTML and CSS for their next generation systems. CSS is an important part of the set of XML technologies, and will become even more so with time.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 12: XML and Databases

## Overview

By now, it should be painfully obvious that XML is all about data, whether it be data for an e-commerce transaction, or data in a document, such as in XHTML. XML provides a near perfect way to communicate textual data, to people and to computers: it's structured, there are ways of ensuring that the structure is adhered to, and it can easily be transmitted and transformed into other formats. So how does XML fit in with traditional databases?

In the world of computer science, you can rarely throw a proverbial rock without hitting such a database. It's inevitable that at some point or another, you're probably going to be dealing with one.

This chapter will discuss some of the issues involved in integrating XML into existing database-oriented applications or, conversely, integrating databases into your new applications. It will cover:

- An overview of what a relational database is, including the concept of normalization
- Some ideas for building an application to take advantage of your existing databases
- Some ideas for potentially building XML right into a database
- A look at what a couple of the major relational database vendors are doing to incorporate XML into their products

You may already be somewhat familiar with the concept of a database: they were invented as a way to store large amounts of information, and quickly retrieve it. Since decades have gone into database design and best practices, a thorough discussion of databases is not within the scope of this chapter, and indeed, this book. The chapter starts out with an explanation of relational databases; if you are already familiar with them you can skip this first part of the chapter if you wish, and move straight on to the [Making Use of What You've Got](#) section.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# What You Need

You can't really do any useful work with a database unless you write some code. For this chapter, we'll be writing code using Active Server Pages (ASP), ActiveX Data Objects (ADO), and an Access database.

You will have already installed ASP and ADO when installing IIS or PWS for the SOAP chapter. For our Access database, we'll be using the Northwind sample application, which is installed with a number of Microsoft products, and downloadable from Microsoft's Office web site, at <http://office.microsoft.com>. Do a search for Nwind.exe, and then download the file. For the purposes of these exercises, install the database to: C:\Program Files\Microsoft Office\Office\Samples\.

*If you happen to have a copy of SQL Server installed, it also comes with the Northwind database installed. If you wish, you can use that database, instead of the Access version.*

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Databases, Yesterday and Today

There are a number of different types of Database Management Systems (DBMS), such as **hierarchical** and **object oriented**, but by far the most common type in use is the **relational** database. In a **Relational Database Management System (RDBMS)**, information is structured in tables that are arranged in a similar way to spreadsheets, in other words in **rows** and **columns**. Since the number of relational databases in the world far exceeds the number of all the others combined, we will focus on them and their use of XML in this chapter.

The key when arranging a relational database is to concentrate on the *data* itself, not on the *applications* that will be using that data. This makes the database much more flexible and open to changes. Arranging the data according to the application might make that application run more smoothly, but it could hinder the data from growing into other areas, and indeed, hinder the application from changing out of its initial design.

The best way to understand is to see an example, so let's see some relational tables. We'll take the following information:

Order Number:	123587
Account Number:	125692
First Name:	John
Middle Name:	Fitzgerald Johansen
Last Name:	Doe
Home Phone:	(555)555-1212
Work Phone:	(555)555-2121
Item:	E16-25A
Description:	Production Class Widget
Quantity:	16
Date:	2000/1/1

and arrange it into some tables like so:

Parts Table	
Item	description
E16-25A	Production Class Widget
B25-25A	Automatic Farstie Maker
E14-25B	Economy Class Widget

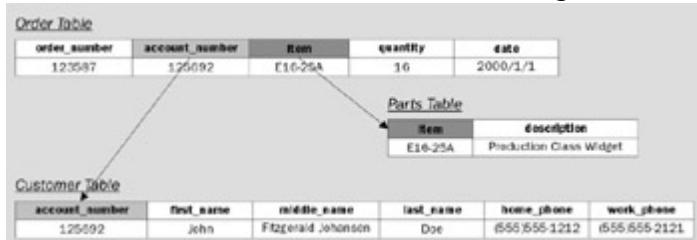
Customer Table					
account_number	first_name	middle_name	last_name	home_phone	work_phone
125692	John	Fitzgerald	Johansen	Doe	(555)555-1212
125693	Jane			Smith	(555)555-1111
125694	Alfred	E.	Neuman	(555)555-9999	(555)555-0000

Order Table					
order_number	account_number	item	quantity	date	
123587	125692	E16-25A	16	2000/1/1	
123588	125692	E16-25A	1	2000/2/1	
123589	125694	E16-25A	20	2000/2/1	
123590	125693	B25-25A	5	2000/2/1	

Our information is laid out in a grid, with rows and columns. The information for one "part", or for one "customer", or for one "order", gets grouped together in a *row*, which is often called a **record** in database terms. Each individual piece of data in that record, such as an "account number" or a "middle name", gets its own *column*, which is also sometimes called a **field**.

As you can see, although the information has been split into multiple tables, we still have enough information in each table to combine it back together if necessary. For example, we could ask for the information for order 123587 from the Order table; then we could use the account\_number column in that table to go to the Customer table, and get our customer information, and the item column to get the information from the Parts table.



## SQL

To facilitate working with relational databases, a programming language was invented to talk to them: **Structured Query Language**, or **SQL**. (Depending on whom you talk to, this might be pronounced "S-Q-L", "sequel", "squeal", or any number of other variations.)

The SQL language has been standardized over the years, notably in SQL92 and SQL99. The intention is that a basic SQL statement that works in one relational database will work with any other relational database. However, to date no RDMS supports the whole of the SQL99 standard, and most database vendors also add their own extensions to the language. Like XSL, SQL is a *declarative* language, not a *procedural* one.

A typical SQL statement looks like this:

```
SELECT first_name FROM Customer WHERE account_number = '125692'
```

You'll notice that the account number is within single quote marks, indicating that it is being modeled as a string rather than a number. This is deliberate; for data such as this, it's better to design for flexibility rather than for speed. It allows, for example, for an alphanumeric account number, should the business rules change in the future.

This statement will return the value from the first\_name column in the Customer table, in the row where the account number is 125692. If there were more than one row with this value for the account number, the SELECT statement would return more than one value. The results from a SELECT statement are in a tabular form, just like relational tables, so in this case the statement would only return one column, and one row for each record in the table with an account number of 125692.

SQL is not case-sensitive, meaning that you could write the word "SELECT" as "SELECT", "select", "Select", and so on. However, many (if not most) programmers tend to write their SQL keywords in all uppercase, to distinguish them from the column names and table names.

One of the problems we often encounter when working with databases is that of **uniqueness**. If we ask for the information from account number 125692, we should only get at most one record returned. If there were multiple accounts with an account number of 125692, it would potentially cause confusion, and we would no longer be able to trust the data in our database. Another way to phrase this is that we would lose **database integrity**.

For this reason, databases have a special type of column (or group of columns) called the **primary key**, which is a column whose value must be unique from record to record within the table. That is, if one record in the table has a value of "1" in its primary key, no other record in the table can have that value for its primary key. Databases enforce this primary key functionality for you, so if you try to insert a new record into the database, with a value in the primary key that is already used, the DBMS will not allow the insert. A table can have only one primary key.

If we specified the account\_number column in the Customer table to be the primary key, and tried to insert another row into the table with the value of 125692 for that column, the database would reject it. This means that we would attain our uniqueness goal, and we could trust that our SQL statement would only ever return at most one record, no more.

## Retrieving Data from Multiple Tables

We can also gather information for order number 123587 from across multiple tables, like the following.

```
SELECT Order.order_number, Order.account_number, Customer.first_name,
       Customer.middle_name, Customer.last_name, Customer.home_phone,
       Customer.work_phone, Order.item, Parts.description, Order.quantity,
       Order.date
  FROM Order, Customer, Parts
 WHERE Order.order_number = '123587'
       AND Customer.account_number = Order.account_number
       AND Parts.item = Order.item
```

The query gets data from all three tables, and the result is a tabular structure containing all of the information from the three tables. Since there could be columns in different tables that have the same name, SQL allows you to prefix the column name with its corresponding table name. So, instead of just specifying the item column, we can specify that we want the item column from the Order table, by saying "Order.item". Since this can involve a lot of typing, and make your SQL statements harder to read, SQL also allows you to create a shorter **alias** for a table name, such as the following:

```
SELECT o.order_number, o.account_number, c.first_name,
       c.middle_name, c.last_name, c.home_phone,
       c.work_phone, o.item, p.description, o.quantity,
       o.date
  FROM Order o, Customer c, Parts p
 WHERE o.order_number = '123587'
       AND c.account_number = o.account_number
       AND p.item = o.item
```

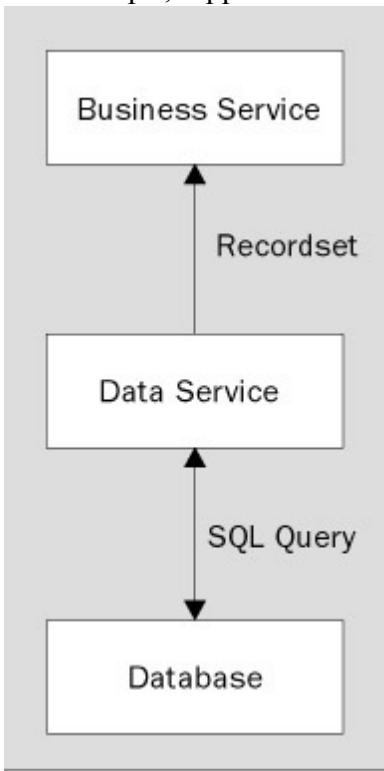
After the name of each table, but before the comma, we specify the alias.

Many technologies that are capable of capturing the results of a SQL query, such as Microsoft's **ActiveX Data Objects**, or **ADO**, also capture the name of each column in the result set, to make the programmer's life easier. This resultset might look something like this (only the first five fields are shown):

Customer Table					
account_number	first_name	middle_name	last_name	home_phone	work_phone
125692	John	Fitzgerald Johansen	Doe	(555)555-1212	(555)555-2121

As you can see, the name of each column gets carried over to the results. However, if we wish, we can also create aliases for our column names, so that the result from our query is not as tightly coupled with our data structure.

For example, suppose we have a setup similar to the following:



That is, we have a database, a Data Service that talks to that database, and a Business Service that works with the recordset returned by the Data Service. The Data Service might perform a SQL statement similar to the following:

```
SELECT first_name, middle_name, last_name  
FROM Customer  
WHERE account_number = '125692'
```

This will produce a recordset similar to this

first_name	middle_name	last_name
John	Fitzgerald Johansen	Doe

Unfortunately, if our data ever changes the result being returned in our recordset will change as well. For example, suppose a company's data modeling standards change, and it is decided to rename these columns to FName, MName, and LName. We would then need to modify the Data Service, which talks to the database, and change any SQL statements that included these columns.

However, we would also have to rewrite the Business Service, since it is expecting a recordset with certain column names. To avoid this, we could alias our column names, so that the recordset is returned in the old format, like this:

```
SELECT FName first_name, MName middle_name, LName last_name  
FROM Customer  
WHERE account_number = '125692'
```

Many developers make it a point to always alias column names in their SQL queries, partially to avoid this problem, and partially to make the column names in their recordsets more readable. For example, if your database has a column to represent a customer's first name called "CFN", it might be more readable to alias that column as

"CustomerFirstName".

## Joins

Getting data from more than one table at a time is such a common occurrence that there is a special SQL syntax you can use for that purpose, called a **join** (because it conceptually joins multiple tables together and treats them as one for the query). Using this join syntax, we could rewrite the above SQL statement like this:

```
SELECT o.order_number, o.account_number, c.first_name, c.middle_name,
       c.last_name, c.home_phone, c.work_phone, o.item, p.description,
       o.quantity, o.date
  FROM Parts p INNER JOIN Order o ON p.item = o.item
    INNER JOIN Customer c ON o.account_number = c.account_number
 WHERE o.order_number = '123587'
```

A join can make your SQL statements easier to read, and can also provide better performance.

*Note that in the code above we used an "inner join". This is the most common type of join. A full explanation of the different types of join available in SQL is beyond the scope of this chapter. If you want to learn more about SQL, a good introductory text is "Beginning SQL Programming", Wrox Press, ISBN: 1861001800. For a more comprehensive look try "An Introduction to Database Systems" by C. J. Date, Addison-Wesley, ISBN: 0201385902.*

## Normalization

We had to put that last SQL statement through a lot of hoops to gather data from across all three tables. Why did we bother to break the information up like that, instead of just putting it all in one table? The reason is that we don't want to have to enter all of a customer's information into the database every time that customer places an order. With our method, all we have to do in the Order table is refer to an existing record in the Customer table. If we were to enter all of the information every time, we would not only get a lot of duplication in our database, but we would also increase the likelihood that information would be entered incorrectly. (If John Doe places five different orders, and we have to enter John Doe's information for each of those orders, then it's five times as likely that we might make a mistake when entering that information.)

This also makes it easier to write other applications to make use of this data in the future. Because we have designed our database with the data in mind, not the application, it's easier to write applications to make use of that data, in ways we had never anticipated. For example, we might write applications to create mailing lists, based on all of the customers in our Customer table, and another to take care of inventory, based on our Parts table, etc., even though neither of these applications would care about the orders in our Order table.

There are a number of common rules that database developers use to organize their data into tables, which over time have proven to make it much easier to build fast and flexible queries into that data, as well as making changes to the structure easier. Using these rules to structure your tables is called **normalization**. For example, some of the rules of normalization include:

- Always break related sets of information into their own tables. In our example, we have information about an order, a part, and a customer, so we have broken the data into three tables accordingly.
- Don't have repeating data in one table. That is, if a customer can have more than one phone number, you should not create a phone1 column and a phone2 column, but instead should have a separate table for phone numbers, and one that links phone numbers to customers. In this way, a customer can have as many phone numbers as needed. (In our data, we broke this rule, by having a home\_phone and a work\_phone column, because we **denormalized** the data. We'll discuss denormalization next.)

- Every table should have a primary key.
- Data that is not uniquely identified by the primary key should be broken out into a separate table.

That last point might need a little more explaining. Imagine that we had a table for corporate customers, which contained the following information:

CorporateCustomer Table				
account_number	name	company_name	company_city	phone
165467	Tony Ruberto	123 Someplace Drive	Toronto	(555)111-2222
654644	Jeremy Weber	86 The Place	New York	(555)333-4444
136841	Gerard Prucknicki	123 Someplace Drive	Toronto	(555)111-5555

*Note that this is fictional data, and that you won't find it in the Northwind database.*

We have here information on a corporate customer, and each customer has an account number. However, the account\_number field doesn't really have anything to do with the company\_address field-the company address depends more on the company\_name. We would fix this by breaking information on the customer's company into a separate table, like this:

CorporateCustomer Table				
account_number	name	company_id	phone	
165467	Tony Ruberto	1	(555)111-2222	
654644	Jeremy Weber	2	(555)333-4444	
136841	Gerard Prucknicki	1	(555)111-5555	

Customer Table			
company_id	name	address	city
1	Serna Ferna Inc.	123 Someplace Drive	Toronto
2	Ferna Serna & Sons	86 The Place	New York

Not only does this better represent the information, we have also managed to eliminate duplication, for the cases where multiple corporate customers work for the same company.

In addition to normalization, there is also a technique called **denormalization**, in which we do the opposite, and bring information that should have gone into a separate table into a less generic one. For example, in our Customer table, we could have broken the fields for the phone numbers into a separate Phone table. But after so much experience working with databases, Database Analysts have realized that sometimes the extra flexibility isn't going to buy you anything, so it would be quicker to put everything into one table.

*There are also technical considerations involved. For example, in most database systems the optimal number of tables in a multi-table join is four or less; after this, the database starts to slow down when trying to retrieve the information.*

Although there are some rules and guidelines you can follow to normalize and denormalize a database, such as putting the data into something called "**third normal form**" (3NF), and then denormalizing any tables that need it, it really takes a lot of experience and know-how to get it just right. This is why companies with large databases often have dedicated Database Analysts, whose job is to manage the databases, and assist software developers who need to connect to the data in those databases.

Important

If your company has a DBA, and you're going to be writing software that accesses the company's database, remember what a valuable resource that DBA is! You might be surprised how well a DBA can help you optimize your SQL statements, to squeeze every ounce of speed out of your database, and thus your application.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

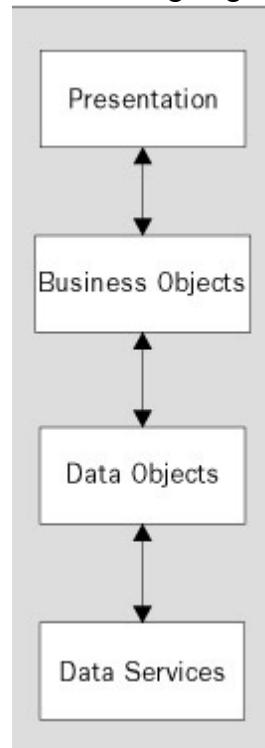
# Making Use of What You've Got

If you're going to be integrating XML into your *existing* infrastructure, chances are you've already got a database in place somewhere, if not more than one. In fact, there was probably a lot of time and effort spent in optimizing those databases for speed, and architecting the data in such a way that it makes sense for your business and is easily retrievable.

In these cases, it's usually a good choice to take advantage of those databases, building an **n-tier** application, in which the logic for the application is broken up into logical layers.

## n-Tier Architecture

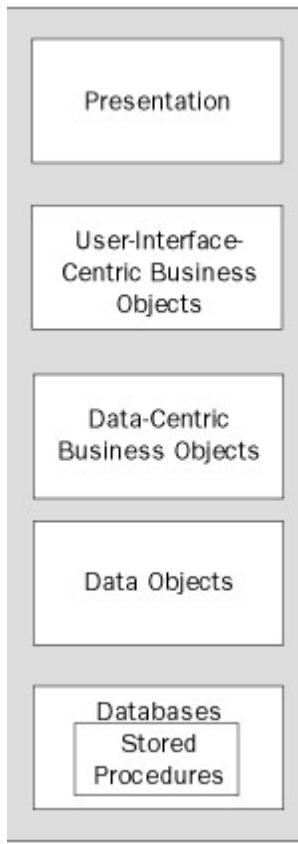
The following diagram illustrates this type of architecture:



Such applications commonly have the following logical layers:

- 
- **Data Services**, where all data for the application is stored (usually a database, but not necessarily)
- 
- **Data Objects**, which handle communication between the database and the Business Objects
- 
- **Business Objects**, which take care of the business logic in your application, and are responsible for communications between the presentation and data tiers
- 
- **Presentation**, which is responsible for dealing with communication between the user and the business logic tier

These logical layers don't necessarily reflect the *physical* deployment of the application. These components could be distributed across several machines, or could all be on the same machine. For large, complex applications, some or all of the layers might be broken down into further layers, giving way to more tiers, such as this:



This is a common design, although different software designers may use different names for the tiers involved.

Stored procedures are a common tool used in databases. They are compiled SQL statements, bundled together like programs in the database itself. They provide much faster execution than calling the database with normal SQL statements, since they are already compiled.

The multi-tiered approach has numerous advantages:

- Reduced complexity for developers. For example, the developers who are writing the Data Objects need extensive knowledge about the database(s) involved, how to optimize SQL statements, and so on. They need to know how to interface with the business layer but not how the business logic of the application works; the Business Objects take care of all of that. By the same token, the developers writing the Business Objects don't need to know about the complexities of the Data Objects layer, they only need to worry about their business logic and how to interface with the data layer. (Even if the same developers are writing objects in both layers, the reduced complexity will help them. It means they only have to concentrate on one task at a time, instead of having all of the logic mixed together into one component.) The complexities of one layer are hidden from other layers.
- Provision for change. For example, we could replace our back-end database, and modify our Data Objects to make use of that new database, but our Business Objects would never have to be touched. They wouldn't know or care that the database had changed. By the same token, if the business rules ever change, we can modify the Business Object concerned, without ever having to change the Data Objects at all.
- More scalable than traditional client-server applications. For example, expensive database connections can be

pooled and shared between Data Objects, meaning that you could potentially serve more clients than you have database connections, as opposed to the client-server model where every client needs a dedicated connection. In addition, when an application grows, extra servers can be added to the environment, to share the load for Business Objects and Data Services objects, which is not as easy for 2-tier client-server applications.

- Increased reusability. With an n-tier architecture, it's very easy to reuse components in other applications. For example, it would be possible to make use of the Data Objects tier in numerous applications, which means that code wouldn't have to be rewritten for each, even though the Business Objects for each application might be vastly different.
- Greater security. The user never accesses the data directly, always via the business tier. This can help to prevent data from being corrupted inadvertently (or deliberately). That is, if there isn't a data object to perform some operation on the data, then users won't be able to perform that operation. As a caveat to this, although it provides better security against accidental damage, it is not suitable protection against a skilled and malicious programmer.

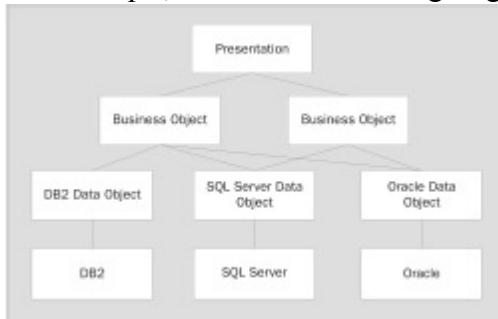
## Using XML in an n-Tier Application

Normally, the Data Objects tier would return information to the Business Objects tier in some kind of data-centric form. **ActiveX Data Objects (ADO)** are commonly used to do this, by code running on Microsoft operating systems. ADO provides a number of objects that can be used to retrieve data from almost any kind of database, including the **Recordset** object, which encapsulates the results of a SQL query. The Data Objects perform the SQL queries, obtain this recordset, and return it to the Business Objects, which then pull out and deal with the data they need.

So what does all of this have to do with XML? Well, if we're doing our job, that Presentation tier is going to be using XML for its data needs. (After all, this is the tier that changes the most often, making it the logical candidate to take advantage of the latest and greatest technologies.) Also, if we're really on the ball, our Business Objects will also be using XML, both to communicate with the Presentation tier, and to communicate with each other. So why not go all the way, and have our Data Objects return XML as well, instead of recordsets? When updating the database, the Business Objects could pass XML to the Data Objects, which would parse the XML and pull out the appropriate data to insert into the databases.

This means that, potentially, any time one object communicated with another, it would use XML as its common language. No longer would the Business Object programmers need to know ADO, or any other data-access technology, all they would need to know is XML. The more objects you have communicating with each other, the more you'll appreciate this simplicity.

For example, consider the following diagram:



There are quite a number of objects in there! Some of these objects might be on different servers. Some might be written in different programming languages. Some might not be using compiled languages at all; the Presentation tier, for example, could be Active Server Pages or CGI scripts creating HTML pages from a web server, or even

DHTML, with JavaScript making use of the XML on the client. But, since we're using XML throughout the application, these differences aren't important to us; all we need to know is the format of our XML documents, and our objects can be created independently of each other.

Think about the ramifications of this. Previously, when our Data Objects were returning ADO recordsets, we would have had to write our Business Objects to understand ADO and a standard called the **Component Object Model (COM)**, since ADO is a COM-based technology. This is great if you're already a COM developer, but if you're a Java developer, you might consider this quite a hassle!

But now, since we're just passing XML back and forth, we have gained language-independence for our objects. We could write some of our Data Objects in Visual Basic, using ADO to retrieve the data from some of the databases, and some in Java, using **JDBC (Java DataBase Connectivity)** to retrieve the data from other databases. These Data Objects could then pass XML to Business Objects that are written in completely different languages. After they'd done their work, they could pass more XML on to the Presentation tier. This could be as simple as an HTML page, with an IE 5 data island containing the XML and some scripting code to work with the data on the page.

Not only have we gained language-independence, but we have also gained some server-independence. Because we're just passing XML back-and-forth, and XML is purely text, we have a lot more freedom in putting different objects on different servers, and using lower-level protocols such as HTTP to pass the information back and forth. If we weren't using XML, we would have one of three choices:

- Put all of our objects on the same server. This would basically reduce us to glorified client/server computing.
- Lock ourselves into a particular technology for distributed computing. The two most common are **CORBA** and Microsoft's **DCOM**.
- Create a proprietary solution. This would provide the most flexibility, but would also take the most development. You would have to write all of the code to call a remote object, in addition to the business logic code needed for your application.

*CORBA (Common Object Request Broker Architecture) is a standard for communicating between distributed objects. It can execute objects in different languages on different platforms, all over the network. DCOM (Distributed COM) is a similar standard, but it only works with Microsoft-based objects. For more information on this topic, see [Chapter 10](#) on SOAP.*

In either case, we would still not have the flexibility that XML allows us. For example, since CORBA implementations from different vendors don't work very well together, we would have to limit ourselves to a single CORBA vendor. Or, if we went the DCOM route, we would have to make all of our objects COM objects, which would limit us to servers running Microsoft operating systems.

Using the XML method, we can use whichever technologies make the most sense: perhaps we can use COM+ on Windows 2000 for some servers, but use Java objects on other servers running UNIX.

So what are the drawbacks to using XML, instead of other technologies like ADO recordsets? The main ones are speed and transmission time. Transforming all of our data to and from XML may provide more flexibility, but it will also reduce the speed of our applications, and the larger file size of our XML documents will add to transmission times. For this reason, you should take this consideration into mind when designing your systems; if you know that all of the objects in your system will be written in Visual Basic, then passing the data around in ADO recordsets will provide the speed, and the extra flexibility of XML is not needed. The same would hold true if all of the objects were to be written in Java, which provides JDBC objects to do the same thing as ADO does for COM-aware languages. On the other hand, when working with distributed applications, the overhead of communicating over a network can be quite noticeable, meaning that the time it takes to transform your data back-and-forth from XML, or transmit

slightly larger files, will often be unnoticeable compared to the network latency.

## Try It Out-Returning XML from a Data Service Object

To put some of this into practice, we'll create a simple data service that can pull information out of a database, and return that information in XML. However, rather than require you to have a full-fledged programming language at hand, we'll "fake it", creating our data service as an ASP page. Luckily, the concepts apply regardless of what programming language or environment you're developing in.

For a database, we'll be using the **Northwind** sample Access database, from Microsoft. We'll connect to Northwind using ADO, which was installed when you installed IIS or PWS, along with Access drivers, for accessing Access databases.

1.

Under the Inetpub\wwwroot directory, create a new directory called databases. Create a new file in that directory, called GetCustomers.asp.

2.

Add the usual ASP text at the beginning of the file, like this:

```
<%@ Language=VBScript %>  
<%
```

3.

This ASP file will return data in XML format; therefore, we need to indicate to the client that it will be receiving XML so we need to set the ContentType property of the Response object accordingly.

```
Response.ContentType = "text/xml"
```

4.

ADO provides a Connection object, which we can use to connect to the database, as well as a Recordset object, which will store the data that is returned from our SQL query. Declare the variables, and create the SQL query, as follows:

```
Dim cnnNorthWind, rsReturn  
Dim sSQLStatement  
  
sSQLStatement = "SELECT * FROM Customers"
```

5.

Next, we'll make the connection to the database, and perform the actual SQL query. The following code will set some properties on the Connection object, make the connection, and then execute the SQL, using the Execute() method.

```
On Error Resume Next  
Set cnnNorthWind = Server.CreateObject("ADODB.Connection")  
  
'This line is wrapped to fit on the page (taking up 3 lines here), but it should 'all be  
on one line in the ASP file. "Data" and "Source", and "Persist" and  
'"Security" should be separated by a space.  
cnnNorthWind.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data  
Source=C:\Program Files\Microsoft Office\Office\Samples\Nwind.mdb;Persist  
Security Info=False"  
  
cnnNorthWind.CursorLocation = 3 'adUseClient
```

```
cnnNorthWind.Open
Set rsReturn = cnnNorthWind.Execute(sSQLStatement)
Set cnnNorthWind = Nothing

If Err.number <> 0 Then
    Response.Write "<Error>" & vbCrLf
    Response.Write vbTab & "<Description>" & Err.Description & _
                  "</Description>" & vbCrLf
    Response.Write "</Error>" & vbCrLf
    Response.End
End If
```

Notice the second line of code here, where we set the `ConnectionString` property. A **connection string** is a string that tells ADO how to connect to the database: where it is located, what ODBC or OLE DB drivers to use, and other information that ADO needs to know about. The most important part of this query string is the `Source`; if your copy of `Nwind.mdb` is in a different location from `C:\Program Files\Microsoft Office\Office\Samples`, you'll need to modify the query string to point to the relevant location on your system.

## 6.

Now that we have retrieved the data from our database, we're ready to start creating our XML. Add the following code:

```
Response.Write "<?xml version='1.0' encoding='ISO-8859-1'?>" & vbCrLf
Response.Write "<Customers>" & vbCrLf

While Not rsReturn.EOF
    Response.Write vbTab & "<Customer ID='"
    Response.Write rsReturn("CustomerID")
    Response.Write "'>" & vbCrLf

    Response.Write vbTab & vbTab & "<CompanyName>" & rsReturn("CompanyName") & _
                  "</CompanyName>" & vbCrLf
    Response.Write vbTab & vbTab & "<ContactName>" & rsReturn("ContactName") & _
                  "</ContactName>" & vbCrLf
    Response.Write vbTab & vbTab & "<ContactTitle>" & rsReturn("ContactTitle") & _
                  "</ContactTitle>" & vbCrLf
    Response.Write vbTab & vbTab & "<Address>" & rsReturn("Address") & _
                  "</Address>" & vbCrLf
    Response.Write vbTab & vbTab & "<City>" & rsReturn("City") & _
                  "</City>" & vbCrLf
    Response.Write vbTab & vbTab & "<Region>" & rsReturn("Region") & _
                  "</Region>" & vbCrLf
    Response.Write vbTab & vbTab & "<PostalCode>" & rsReturn("PostalCode") & _
                  "</PostalCode>" & vbCrLf
    Response.Write vbTab & vbTab & "<Country>" & rsReturn("Country") & _
                  "</Country>" & vbCrLf
    Response.Write vbTab & vbTab & "<Phone>" & rsReturn("Phone") & _
                  "</Phone>" & vbCrLf
    Response.Write vbTab & vbTab & "<Fax>" & rsReturn("Fax") & _
                  "</Fax>" & vbCrLf

    Response.Write vbTab & "</Customer>" & vbCrLf
    Response.Flush

    rsReturn.MoveNext
Wend

Response.Write "</Customers>"
```

I've specified "ISO-8859-1" for the `encoding` attribute of the XML Declaration, since this is the character set used by default by IIS.

7.

Finally, we just need to release our Recordset COM object. Strictly speaking, this isn't necessary, since IIS will release it for us when the ASP page is finished executing, but it's good practice to do so. The final version of the page should look like this:

```
<%@ Language=VBScript %>
<%
Response.ContentType = "text/xml"

Dim cnnNorthWind, rsReturn
Dim sSQLStatement

sSQLStatement = "SELECT * FROM Customers"

On Error Resume Next
Set cnnNorthWind = Server.CreateObject("ADODB.Connection")
cnnNorthWind.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Microsoft Office\Office\Samples\Nwind.mdb;Persist Security Info=False"
cnnNorthWind.CursorLocation = 3 'adUseClient
cnnNorthWind.Open
Set rsReturn = cnnNorthWind.Execute(sSQLStatement)
Set cnnNorthWind = Nothing

If Err.number <> 0 Then
    Response.Write "<Error>" & vbCrLf
    Response.Write vbTab & "<Description>" & Err.Description & _
                  "</Description>" & vbCrLf
    Response.Write "</Error>" & vbCrLf
    Response.End
End If

Response.Write "<?xml version='1.0' encoding='ISO-8859-1'?>" & vbCrLf
Response.Write "<Customers>" & vbCrLf

While Not rsReturn.EOF
    Response.Write vbTab & "<Customer ID='"
    Response.Write rsReturn("CustomerID")
    Response.Write "'>" & vbCrLf

    Response.Write vbTab & vbTab & "<CompanyName>" & rsReturn("CompanyName") & _
                  "</CompanyName>" & vbCrLf
    Response.Write vbTab & vbTab & "<ContactName>" & rsReturn("ContactName") & _
                  "</ContactName>" & vbCrLf
    Response.Write vbTab & vbTab & "<ContactTitle>" & rsReturn("ContactTitle") & _
                  "</ContactTitle>" & vbCrLf
    Response.Write vbTab & vbTab & "<Address>" & rsReturn("Address") & _
                  "</Address>" & vbCrLf
    Response.Write vbTab & vbTab & "<City>" & rsReturn("City") & _
                  "</City>" & vbCrLf
    Response.Write vbTab & vbTab & "<Region>" & rsReturn("Region") & _
                  "</Region>" & vbCrLf
    Response.Write vbTab & vbTab & "<PostalCode>" & rsReturn("PostalCode") & _
                  "</PostalCode>" & vbCrLf
    Response.Write vbTab & vbTab & "<Country>" & rsReturn("Country") & _
                  "</Country>" & vbCrLf
    Response.Write vbTab & vbTab & "<Phone>" & rsReturn("Phone") & _
                  "</Phone>" & vbCrLf
    Response.Write vbTab & vbTab & "<Fax>" & rsReturn("Fax") & _
                  "</Fax>" & vbCrLf

    Response.Write vbTab & "</Customer>" & vbCrLf
    Response.Flush

    rsReturn.MoveNext
```

Wend

```
Response.Write "</Customers>"
```

```
Set rsReturn = Nothing
```

```
%>
```

8.

Since we put the page into the Inetpub\wwwroot\databases directory, you can view it at <http://localhost/databases/GetCustomers.asp>. It will look something like this in the browser:



## How It Works

Using ADO, we have connected to a database, retrieved some data, converted it to XML, and returned it to the client. Of course, using Access as a back-end database won't provide the kind of scalability that most applications will require, however, the techniques used here will apply to any database.

In fact, if you have a copy of SQL Server handy, with the Northwind database installed, you can run this ASP page from that database, rather than from the Access database, simply by changing the connection string, and leaving the rest of the code as-is! For example, if I had a database server named MyServer, a user named "username", and a password of "password", I could use the following connection string:

```
Provider=SQLOLEDB.1;Initial Catalog=Northwind;Data
Source=MyServer;UID=username;PWD=password;Network Library=DBMSSOCN
```

Once we have the data from the database, in our Recordset object, we write out the root element's start-tag, and create a loop to get information out of each record. Finally, after we have iterated through each record, we write out the root element's end-tag.

In addition, if an error occurs while trying to retrieve our data, we write out a simple XML document with the error's description, so that no matter what, our client will receive well-formed XML as a response.

## Try It Out-A Simpler Data Service

When writing data services, you may often find yourself using the method in the Try It Out above, whereby you write out your XML one tag at a time, inserting data at the appropriate points. However, if you're going to be using ADO's Recordset object, there is an easier way to do this: there is a save() method provided, which can be used to persist the information in the Recordset to a file, or to a stream. In addition, the Recordset object can be told to save the information in XML format! Also, since ASP's Response object is a stream, we can actually take the information from our Recordset, and pass it straight through to the client in XML form!

1.

In the same directory where you put GetCustomers.asp, create a new file called GetCustomers2.asp. Insert the following code into that file:

```
<%@ Language=VBScript %>
<%
Response.ContentType = "text/xml"

Dim cnnNorthWind, rsReturn
Dim sSQLStatement

sSQLStatement = "SELECT * FROM Customers"

On Error Resume Next
Set cnnNorthWind = Server.CreateObject("ADODB.Connection")
cnnNorthWind.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program Files\Microsoft Office\Office\Samples\Nwind.mdb;Persist Security Info=False"
cnnNorthWind.CursorLocation = 3 'adUseClient
cnnNorthWind.Open
Set rsReturn = cnnNorthWind.Execute(sSQLStatement)
Set cnnNorthWind = Nothing
If Err.number <> 0 Then
    Response.Write "<Error>" & vbCrLf
    Response.Write vbTab & "<Description>" & Err.Description & "</Description>" &
    vbCrLf
    Response.Write "</Error>"
    Response.End
End If

rsReturn.Save Response, 1

Set rsReturn = Nothing
%>
```

Much of this code is the same as it was for GetCustomers.asp, so feel free to copy and paste the code from one file to the other. The important line of code is the highlighted one, which calls the save() method, tells it where to persist the information (the Response object), and in what format to save it (1, which is short for adPersistXML).

## 2.

You can view this page by going to <http://localhost/databases/GetCustomers2.asp>. You'll notice that the format of the XML returned by ADO is much different from the XML we created before! It will look something like this in your browser:



I have collapsed the `<s:Schema>` element, which comes before the actual data, for readability.

## How It Works

The code to return our XML is now much simpler; we simply call the save() method, and let ADO do all of the work for us! However, the XML that is returned is now a lot more complex. It looks something like this:

```
<xml>
<s:Schema/>
<rs:data>
  <z:row ColumnName="data from DB"/>
</rs:data>
</xml>
```

The <s:Schema> element contains an XDR schema for the data in the document, which is in the <rs:data> element. Each record in the Recordset object is in a <zrow> element, and each column in the row is in an attribute of that element.

### Important

Note that the root element of this document is called "xml", which, as we learned in [Chapter 2](#), is illegal! Unfortunately, MSXML allows XML element names that break this well-formedness rule, and allows elements to be named like this. However, you should never take advantage of this fact, and create XML documents with elements starting with the letters "x", "m", and "T"-if you do, your documents will only ever be allowed by MSXML, and no other XML parsers, meaning that you will have lost one of the main benefits of XML: portability.

Because the Response object in ASP is a stream, we were able to pass this XML straight into it, without having to really touch it.

---

[!\[\]\(4fbc7a62936a056d4cb8b68398cd96b0\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(73498c0aeca4df0a50f49410beee4811\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Integrating XML into the Database Itself

Suppose we're creating an application from scratch. We're dealing with a lot of data, so we're probably going to need a database, but we're savvy enough to be using XML as much as possible. How can we combine the best of both worlds?

We'll take a look at a common way to do this, mapping the data from XML to relational tables and vice versa, and we'll also look at taking denormalization to the extreme, and storing XML strings right in the database.

## Mapping XML to a Relational Structure

One of the difficulties of working with both XML and a traditional relational database is that the way you structure data for each is fundamentally different. When you're working with a relational database, you're dealing with data in columns, rows, and tables, and with the relations between those tables. On the other hand, when you're working with XML, you're dealing with very hierarchical information, along with the information attached to that information (attributes).

If you're building an application from scratch, you may even have to model your data twice: once for the database, and once for your XML documents. If you're going to be modifying an existing application, your job will probably be made much simpler, because you'll only have to model the data one way: mapping an existing database into new hierarchical XML structures, or mapping XML documents to new relational database tables.

There aren't a lot of hard and fast rules about mapping data back-and-forth from relational tables to XML hierarchies. For example, consider our Order table:

Order Table				
order number	account number	item	quantity	date
123587	129922	E1625A	16	2000/1/1
123588	129922	E1625A	1	2000/2/1
123589	129994	E1625A	20	2000/2/1
123590	129693	B2625A	5	2000/2/1

Off the top of my head, I can come up with a number of ways of structuring this as an XML document, such as

```
<?xml version="1.0"?>
<orders>
  <order number="123587">
    <account>129922</account>
    <item>E1625A</item>
    <quantity>16</quantity>
    <date>2000/1/1</date>
  </order>
  <order number="123588">
    <account>129922</account>
    <item>E1625A</item>
    <quantity>1</quantity>
    <date>2000/2/1</date>
  </order>
  <!--etc.-->
</orders>
```

or

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

```
<order number="123587"
       account="12992"
       item="E1625A"
       quantity="16"
       date="2000/1/1"/>
<order number="123588"
       account="12992"
       item="E1625A"
       quantity="1"
       date="2000/2/1"/>
<!--etc.-->
</orders>
```

or any number of other ways. Any or all of these ways of structuring the data might make sense for my application.

Of course, to a certain extent the way I structure my XML isn't too important, either. After all, we always have XSLT to fall back on—if I structure the data in such a way that it's not easy to work with for certain applications, that XML can always be transformed to a different format. Furthermore, I can return XML in one format, for certain purposes, but create entirely different XML documents, from the same database, for use in other situations.

For this reason, I think it's more important to get the structure of your relational database perfect, than it is to get the structure of your XML documents perfect. Imperfectly designed XML will need to be transformed, causing performance hits, but improperly designed databases have the potential to cause enormous headaches down the road, when it's hard to get out our data or change the data that needs to be stored.

On the other hand, sometimes a database doesn't need to be designed with this robustness in mind. (We'll see an example of this coming up.) In this case, you might want to consider an alternative to a well-normalized database.

## Storing XML in a Database

Consider a forms-based application where your users will be entering many pieces of information for every application, but most of that information only makes sense in the context of the application as a whole. For example, we might want to do a search to find the account number for the person who placed an order, but we'd never do a search to find out that customer's middle name; we only ever need to see the middle name when we're working with the whole order.

Perhaps we could store all of the information in one table in the relational database; any searchable fields (for example: account number, order number) would get their own column, and then another column would be used to store all of the XML (as a string) for the entire order:

Order Table		
order_number	account_number	order_xml
123587	125692	<?xml version="1.0"?><Order number="123587"><Account number="125692"><FirstName>John</FirstName><MiddleName>Fitzgerald</MiddleName><LastName>Doe</LastName><HomePhone>(555) 555-1212</HomePhone><WorkPhone>(555) 555-2121</WorkPhone></Account><Item number="E16-25A"><Description>Production Class Widget</Description><Quantity>16</Quantity></Item><Date>2000/1/1</Date></Order>

Since we have one XML document representing each order, we can get all the information for an entire application from one field. Getting all this information might be as simple as using:

```
SELECT order_xml FROM order WHERE order_number = 123587
```

In most cases, updating an order also involves modifying just one field, since you rarely need to change the columns

that are searchable.

## Using the Database

Normally, this approach works best when your database is a **staging database**, or a temporary holding place for the data, until the time when it can be put into the real back-end database. This is because the requirements for a staging database are often less stringent than for other databases. As you can see from the diagram above, the data is not normalized at all, but is stored in one large chunk. The retrieval of the data is an all-or-nothing exercise that makes it much easier to return the data for the application that is using it, eliminating joins and complex SQL statements. However, it does make it harder to write other applications that might want to use *parts* of the data, since those parts aren't directly exposed. An example might be a reporting application that could list all of the orders placed by a particular customer in the last month.

*The most common example of an application that can make use of this paradigm is a forms-based web application. A user goes through your web site, entering information on each page. Upon clicking submit at the end of each page, the data is stored in your staging database to keep it safe. When the user gets to the last page, and performs the final submit, the data from your staging database is submitted to the real database, for processing by your back-end systems.*

When it does come time to move the data to the real database, a Data Object would read the XML from the staging database, create SQL INSERT statements with it, and insert it into the database that way. Your application gets all of the advantages of dealing with the data in an XML format it understands in the front end, and your company gets the advantages of having a fully normalized relational database in the back end.

In many cases, you don't even need to keep the data in the staging database for too long, since it will be permanently stored in your back-end databases for posterity. Not only is the database simple, it also becomes easier to maintain, since you don't have to worry too much about running out of space. Perhaps you can delete the row as soon as the data is submitted to the back-end databases, or maybe you can schedule an automatic job to run every night, and delete any data that's more than 30-days old.

Before you decide to take this drastic approach and store all your XML data in this way, you'll want to take a good hard look at what you're doing and whether this will meet your needs. Many Database Analysts would probably faint at a table like this! You might also want to consider less drastic measures, where you are only breaking certain sections of the data into their own XML fields. Perhaps you could store an address in its own column, in XML format, for example?

On the other hand, if this type of layout can meet your needs, you'll have all of the benefits of an n-tier architecture, but your Data Objects have one less task to perform (translating your data into XML format). It's only when we need to move the data up to a higher-level database that we need to worry about translation.

### Important

The less work your objects have to do, the more scalable your application becomes, and the more users you can serve. but remember: this increase in speed comes with a decrease in flexibility.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Database Vendors and XML

With both XML and databases being data-centric technologies, are they in competition with each other? Quite the contrary! XML is best used to communicate data, and a database is best used to store and retrieve data; this makes the two *complementary*, rather than *competitive*.

For this reason, database vendors are realizing the power and flexibility of XML, and are building support for XML right into their products. This potentially makes the programmer's job easier when writing Data Objects, since there is one less step to perform: instead of retrieving data from the database, and transforming it to XML, you can potentially retrieve data from the database already in XML format!

## Important

XML will never replace the database, but the two will become more closely integrated with time.

We'll be looking at two database vendors who were quick to market with XML integration technologies: Oracle, and Microsoft. Both companies offer a suite of tools that can be used for XML development, when communicating with a database and otherwise.

*Note that in the following sections there are code segments, but no Try It Outs at all. This is because we considered it perhaps too much to expect you, the reader, to install these tools for one example.*

## Microsoft's XML Technologies

Microsoft has been big on XML since the very beginning. There is extensive documentation on XML at **MSDN** (**Microsoft Developer Network**), the online site devoted to developers working with Microsoft technologies. The XML-related information is gathered together at <http://msdn.microsoft.com/xml/>. There's also copious amounts of sample code, just there for the, erm, "borrowing".

### MSXML

The first, and most obvious, form of XML support from Microsoft is that Internet Explorer comes bundled with the **MSXML** COM-based parser. MSXML 3 (that is shipping with IE 6) provides validating and non-validating modes, as well as support for XML namespaces, SAX 2, and XSLT. MSXML 4, which is still in beta stage at the time of writing, also includes support for the XML Schemas Recommendation.

*Remember that you can install these parsers with IE 5 if you wish, but MSXML 4 installs in Replace Mode only.*

### Visual Basic Code Generator

Because MSDN is big on code samples, there are a number of sample applications available for download from the site. One good example is a Visual Basic code generator, which can read XDR schema documents, and produce Visual Basic classes to match the schema (it's located at <http://msdn.microsoft.com/xml/articles/generat.asp>). In effect, you can build the basics of an object model, based on an XML document type, automatically. This tool isn't supported by Microsoft?it's only provided for convenience?but the sourcecode is available so you can always change anything that doesn't quite work the way you want it to.

However, remember that the tool is based on an earlier schema language from Microsoft, called **XML-Data Reduced**, or **XDR**, not the W3C's XML Schema language.

## SQL Server

And finally, there's the XML support that's built into SQL Server, Microsoft's Relational Database Management System.

### Important

All of this is available natively in SQL Server 2000, or as a downloadable Technology Preview for SQL Server 7.

SQL Server provides the ability to perform a SQL query through an HTTP request, via an ISAPI filter for Internet Information Server (Microsoft's web server). So we might perform a query like the following (replacingservername with the name of your web server, and databasename with the name of the database you're querying):

```
http://servername/databasename?sql=SELECT+last_name+FROM+Customer+FOR+XML+RAW
```

Or, if we don't want to be passing our complex SQL statements in a URL, we can also create an **XML template** file to store the query. It would look something like this:

```
<root>
  <sql:query xmlns:sql="urn:schemas-microsoft-com:xml-sql">
    SELECT last_name FROM Customer FOR XML RAW
  </sql:query>
</root>
```

Notice the words FOR XML RAW added to the end of that SQL statement. This is a language enhancement Microsoft has added to allow SQL queries to natively return XML.

If this template were named lastname.xml, we'd execute the SQL by using the following URL:

```
http://servername/databasename/lastname.xml
```

And this query would return XML similar to the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <row last_name="Doe" />
  <row last_name="Smith" />
  <row last_name="Johnson" />
</root>
```

FOR XML RAW instructs SQL Server to format the XML in its own way. There is also a FOR XML EXPLICIT clause, which allows you to tell SQL Server how to format the XML. This makes the SQL queries very complex, though, because you have to tell SQL Server not only what data, and from where, but also how you want the data formatted. So if you're using FOR XML EXPLICIT, you will probably want to use the XML templates.

As we'll see in the upcoming example, templates can take parameters as well, to make your queries dynamic.

Not only can you get data from SQL Server in XML, you can put it in using **SQL Update Grams**. These are XML files containing the information you want to put in the database (in a certain format, of course).

## Putting SQL Server to Work

For this example, let's go back to our large SQL statement, which joined together information from across all three of

our tables.

Because the SELECT statement is so long, we're probably not going to want to pass it over in the query string, so let's create an XML template to store the results, which we'll call order.xml:

```
<?xml version="1.0"?>
<root>
<sql:query xmlns:sql="urn:schemas-microsoft-com:xml-sql"><! [CDATA[
SELECT Order.order_number, Order.account_number, Customer.first_name,
       Customer.middle_name, Customer.last_name, Customer.home_phone,
       Customer.work_phone, Order.item, Parts.description, Order.quantity,
       Order.date
FROM Order, Customer, Parts
WHERE Order.order_number = '123587'
      AND Customer.account_number = Order.account_number
      AND Parts.item = Order.item
FOR XML RAW
]]></sql:query>
</root>
```

Notice that I have also wrapped the SQL query in a CDATA section. This isn't strictly necessary, for our query, since there are no & or < characters, but sometimes it's just better to be safe than sorry.

We would call this query something like this:

```
http://servername/databasename/order.xml
```

But wait a second: this SQL query is always going to return the same information. Why would we even bother with all of this: we could just save the result and ask for that? We need a way to choose the order number at runtime, so that we can get the information from any order. We do this like so:

```
<?xml version="1.0"?>
<root>
<sql:query ordnum='' xmlns:sql="urn:schemas-microsoft-com:xml-sql"><! [CDATA[
SELECT Order.order_number, Order.account_number, Customer.first_name,
       Customer.middle_name, Customer.last_name, Customer.home_phone,
       Customer.work_phone, Order.item, Parts.description, Order.quantity,
       Order.date
FROM Order, Customer, Parts
WHERE Order.order_number = ?
      AND Customer.account_number = Order.account_number
      AND Parts.item = Order.item
]]></sql:query>
</root>
```

As you can see, there is an extra attribute on our `<sql:query>` element for the order number, which has no value. There is also a question mark in our WHERE clause. This will be a parameter to our template. We can then call the template like this:

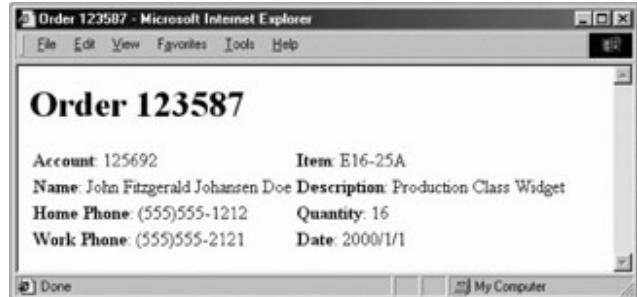
```
http://servername/databasename/order.xml?ordnum='123587'
```

Next, let's ensure that the results of this query will be viewable in IE 5, by adding a style sheet PI to the document (which we'll assume is stored in the same folder):

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="html.xsl"?>
<root>
<sql:query ordnum='' xmlns:sql="urn:schemas-microsoft-com:xml-sql"><! [CDATA[
SELECT Order.order_number, Order.account_number, Customer.first_name,
       Customer.middle_name, Customer.last_name, Customer.home_phone,
```

```
Customer.work_phone, Order.item, Parts.description, Order.quantity,  
Order.date  
FROM Order, Customer, Parts  
WHERE Order.order_number = ?  
    AND Customer.account_number = Order.account_number  
    AND Parts.item = Order.item  
]]></sql:query>  
</root>
```

IE5 will recognize that PI, fetch the XSL style sheet (htm.xsl), and render the document on the client. We might end up with a web page like this:



With a simple XML file, which is accessible through an ISAPI filter, we have effectively created an order-viewing system! All we need to know is the order number, and we can get the information from SQL Server formatted in XML and, once on the client, can use XSL to re-format the data as HTML.

## Oracle's XML Technologies

Like Microsoft, Oracle provides a number of tools to work with XML, in its **XML Developer's Kit (XDK)**. However, since Microsoft had some XML support built right into the database, it took them much longer to get their database-integration tools to market, whereas Oracle had these tools out to market very quickly indeed.

*Oracle's Technology Network has a good web site devoted to the XML Developer's Kit, which is located at <http://technet.oracle.com/tech/xml/>. You will need to register in order to download the XDK, or see some of the documentation, but registration is free.*

### XML Parsers

The first tool available from Oracle is the XML parser. Oracle provides parsers written in Java, C, C++, and PL/SQL. These parsers provide:

- A DOM interface
- A SAX interface
- Both validating and non-validating support
- Support for namespaces
- Fully compliant support for XSLT

Like MSXML, the Oracle XML parsers provide extension functions to the DOM, `selectSingleNode()` and

selectNodes(), which function just like the Microsoft methods.

## Code Generators

Oracle offers Java and C++ class generating applications, just like the Visual Basic class building application Microsoft offers. However, these generators work from DTDs, not schemas, meaning that they are already fully conformant to the W3C specifications. Also, since these tools are part of the XDK, they are fully supported by Oracle, instead of just being sample code.

## XML SQL Utility for Java

Along with these basic XML tools, the XDK also provides the **XML SQL Utility for Java**. This tool can generate an XML document from a SQL query, either in text form or in a DOM.

The XML results may differ slightly from the Microsoft SQL Server FOR XML clause, but they're just as easy to use. For example, the following SQL query:

```
SELECT last_name FROM customer
```

might return XML like the following:

```
<?xml version="1.0"?>
<ROWSET>
  <ROW id="1">
    <last_name>Doe</last_name>
  </ROW>
  <ROW id="2">
    <last_name>Smith</last_name>
  </ROW>
  <ROW id="3">
    <last_name>Johnson</last_name>
  </ROW>
</ROWSET>
```

So, instead of including the information in attributes of the various `<row>` elements, Oracle decided to include the information in separate child elements.

And, just like Microsoft's enhancements to SQL Server 2000, Oracle's XML SQL Utility for Java can take in XML documents, and use the information to update the database.

## XSQL Servlet

Microsoft decided to make SQL queries available over HTTP by providing an ISAPI filter for IIS. Oracle, on the other hand, decided to create the Java **XSQL Servlet** instead. Since it is a Java Servlet, it will run on any web server that supports servlets?which is most of them.

This servlet takes in an XML document that contains SQL Queries, like the XML templates used by SQL Server. The servlet can optionally perform an XSLT transformation on the results (if a stylesheet is specified), so the results can potentially be any type of file that can be returned from an XSLT transformation, including XML and HTML. (To accomplish this, the XSQL Servlet makes use of the XML parser mentioned above.) Because it's a servlet, it will run on any web server that has a Java Virtual Machine and can host servlets.

For example, consider the following XML example document from Oracle's Technology Network web site, which might be used by the XSQL Servlet:

```
<?xml version="1.0"?>
<?xmlstylesheet type="text/xsl" href="rowcol.xsl"?>
<query connection="demo"
    find="%"
    sort="ENAME"
    null-indicator="yes" >
    SELECT * FROM EMP
        WHERE ENAME LIKE '%{@find}%' 
            ORDER BY {@sort}
</query>
```

When this query is run, the servlet will replace the <query> element with the results of the query. In this case, there are two variables indicated: find and sort, which are used to specify the search and sort criteria respectively. These variables could be specified by passing them to the servlet, for example, using the following URL:

<http://localhost/xsql/demo/emp.xsql?find=T&sort=EMPNO>

Notice also the <?xml-stylesheet?> PI. Since we have included this, the servlet will transform the results of the query, using the specified XSLT stylesheet (rowcol.xsl).

## Putting Oracle to Work

Just to compare the two technologies, we could redo our SQL Server query with Oracle's tools. First of all, we'll need our XSQL file, which stores the query:

```
<?xml version="1.0"?>
<?xmlstylesheet type="text/xsl" href="html.xsl"?>
<query ordnum='1'><![CDATA[
SELECT Order.order_number, Order.account_number, Customer.first_name,
       Customer.middle_name, Customer.last_name, Customer.home_phone,
       Customer.work_phone, Order.item, Parts.description, Order.quantity,
       Order.date
FROM Order, Customer, Parts
WHERE Order.order_number = {@ordnum}
      AND Customer.account_number = Order.account_number
      AND Parts.item = Order.item
]]></query>
```

This is very much like the Microsoft version, except that we treat the parameter slightly differently (highlighted in the code), and we don't need a namespace for the <query> element. We would also call this the same way as we call the Microsoft version:

<http://servername/databasename/order.xsql?ordnum='123587'>

However, there is a difference between the way the Microsoft and Oracle technologies work. The XSL transformation that will take place for this XML is performed by the servlet, on the server. This means that we don't necessarily need IE 5 as our client browser, but can use *any* web browser.

Performing the XSL transformation on the server might slow our server down a bit, because it has one more thing to do. However, it will provide much more flexibility, both with the client browsers that can access the information, and with the power we have in our XSLT stylesheets.

# Who Needs a Database Anyway?

So far, most of this chapter has assumed that you are using your XML knowledge to build an "application" of some kind: an order processing application, for example. We've been talking about n-tier architecture, and building objects to do the work. But what if you're not building applications at all? Perhaps you're putting together an online magazine, for example, or a news site-something that is completely centered around content, and not so much around changing the content regularly, or transactions.

In this case, what you're probably most concerned with is getting that content served up to your users as quickly as possible. When they go to your site, they don't want to have to wait for the information to get to them, it should be instantaneous. This means that your web server should be doing as little as possible. When the user requests a page, the web server should be able to just give it out, and then move on to do something else.

Traditionally, web sites could get better scalability by using static HTML pages in place of dynamically generated pages, because the web server doesn't have to do any work with the page, just hand it out. With the addition of XLink and XPointer to the XML family of technologies (both of which are covered in the [next chapter](#)), we could potentially do the same thing with XML-based sites: serve up static XML documents and CSS stylesheets to clients, and have the clients render the XML for display. If the clients understand XLink, they would then be able to further link to the other XML documents on your site, the way that HTML pages can link to one another today with HTML hyperlinks.

However, this approach may be a little bit "pie in the sky" at this point in history-there aren't a lot of browsers that can display XML documents formatted with CSS stylesheets, and even fewer browsers that understand XLink. That brings up our other option: using XSLT to transform our XML into HTML.

When deciding how to do this, we have a few choices:

- We could pass the XML and XSL documents to the client, and have the browser perform the transformation. (For this to work, the client-side browser must be able to perform XSLT transformations.)
- We could perform the transformation server-side, using a servlet, or Active Server Pages. Writing an ASP page to call an XSL transformation only takes a few lines of code.
- We could perform a batch transformation of all of our XML documents to HTML, and store those static HTML documents on the web server.

Depending on your situation (including your browser-base), all of these could be viable options, so feel free to use whichever methodology works best for your application. And as we move forward, and web browsers get smarter about XML, you might find yourself leaning more and more toward the first choice, making less work for your web server.

The point is you don't always need a database. Part of the fun of software development is figuring out which technologies you need, and which you don't. And by all means, feel free to mix and match! It's not too far fetched to imagine applications with a lot of content stored in static XML documents, and other information coming from databases.

# XML Databases

Most of this chapter has focused on integrating XML with traditional databases. The general idea has been to store the data in the database, and transform it to and from XML when needed.

However, there is a fundamental difference between relational databases and XML documents: relational databases are centered on tables, columns, and rows, whereas XML documents are hierarchical in nature, using the notions of parents, children, and descendants. This means that there may be cases where the structure of an XML document won't fit nicely into the relational model.

For such cases, you might want to consider a specialized **XML database**. Instead of transforming back-and-forth from XML, like SQL Server and Oracle, an XML database stores data natively in XML. Storing your data in XML format can offer a couple of benefits:

- The speed of your application can be greatly increased. When all of the components in your application need to deal with the data in XML format, an XML database eliminates the need to transform the data back-and-forth—it starts out in XML, and stays in XML.
- Many of the XML databases coming out provide functionality to query the database using XPath queries, just as relational databases today use SQL. Since XML is hierarchical in nature, XPath can provide a more natural query language than SQL, which is more focused with the tables, rows, and columns of relational databases.

There are already a number of such XML databases available. The features and focus of each product can vary greatly, meaning that not all of these products are in direct competition with each other. Some of the XML databases you might encounter include:

- Extensible Information Server, from eXcelon (<http://www.exeloncorp.com>)
- GoXML, from XML Global (<http://www.xmlglobal.com/prod/db/>)
- dbXML, an open source product from dbXML Group (<http://www.dbxml.com>)
- Tamino XML Database, from Software AG (<http://www.softwareag.com/tamino/>)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

This chapter hasn't been an exhaustive look into XML and databases, but hopefully it has started to get you thinking about how the two can be used together effectively.

*A more comprehensive study is included in Professional XML, 2nd Edition (ISBN 1861005059) from Wrox Press.*

We have seen:

- A little bit of how normalization works, and why it is important
- Some ways that the Data Objects in an n-tier architecture can be used to keep XML as your communication format throughout the application, instead of dealing with other data-transmission technologies
- How Microsoft and Oracle have begun the work of integrating XML and the traditional database, and the tools they provide with that aim
- Why you might want to consider a native XML database, in cases where your applications need XML, XML, and nothing but XML.

In the [next chapter](#), we'll be examining a way that you can link your XML documents to each other, similar in concept to HTML hyperlinks, as well as some ways that you can point to or retrieve pieces of an XML document.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Chapter 13: Linking and Querying XML

## Overview

This [last chapter](#) of our book will look at initiatives in linking and querying XML, both areas where the W3C has been working towards making Recommendations.

We'll be looking at ways to link XML documents together, using technologies called [XLink](#) and [XPointer](#), covering:

- HTML hyperlinks, and their limitations
- How to create the same type of links that HTML provides using XLink simple links
- How to create links which point to multiple resources, along with rules for traversing those links, using XLink extended links
- How to create links which only reference certain sections of an XML document, instead of the entire document, using XPointer

At the end of this chapter we'll take a briefer look at [XQuery](#), a potentially important new query language for XML.

As you probably knew even before you picked up this book, XML is a cutting edge technology. Some specifications, like the XML specification itself, have been recommended by the W3C, meaning that they are ready to be implemented, whereas others are still in various stages of revision. In the case of the two linking specifications we'll be looking here, XLink is a W3C Recommendation, while, at the time of writing, XPointer was still at the "Last Call" stage and XQuery is still at the working draft stage. The impact of working with these incomplete specifications is that there is little or no software that implements them, at the time of writing, even for XLink, which is a full Recommendation. So, the sample code and Try It Outs in this chapter are for example only; you probably won't actually be able to try the samples with real software. Hopefully, with web browsers such as Mozilla, which is already starting to incorporate some XLink functionality, the sample code in this chapter will become much more useful in the near future.

# Linking

One of the reasons for the phenomenal growth of the Internet is the ability for documents to **link** to other documents. If I create a web page about guitars, for example, I can include links on my page that point to my favorite guitar makers, my favorite musicians, etc. Even if the sites that I'm linking to have nothing to do with me, or are halfway around the world, I am able to create these links, because HTML provides a mechanism for doing so (called hyperlinks).

In this section, we'll be looking at similar ways to link XML documents together, using the XLink and XPointer technologies mentioned above. The XLink specification can be found at <http://www.w3.org/TR/xlink/>, and the XPointer specification can be found at <http://www.w3.org/TR/xptr/>.

## HTML Linking

Let's start by looking at how we can link documents in HTML and the limitations of HTML linking. We can then look at how XLink and XPointer overcome these limitations.

One of the reasons for the phenomenal growth of the World Wide Web is the ability for any HTML page to link to another page via a hyperlink. This is what makes the Web a "web", since all web pages can theoretically be connected to all other web pages.

These hyperlinks work very simply: a user sees the hyperlink on a web page, clicks it, and a new page replaces the current one. Or, as variations on this theme, perhaps the new document opens up a new window, or just replaces the page in one of the frames on the current page. The action performed when the user selects the link is also configurable to a certain degree. For example, in some browsers you can right-click a link and choose specifically to open the link in a new window, instead of having the link replace the current page in the current window.

HTML hyperlinks are constrained in a few ways:

- A link only involves *two* resources: the resource which contains the link, and the resource to which the link points. (Remember, a resource is anything which has identity; in this case, the initiating resource will always be an HTML document.)
- These HTML hyperlinks are always *one-way* (although most web browsers provide a Back button, to provide the appearance of a two-way link).
- HTML links are embedded in HTML documents, meaning that the source has to be a marked up document.

The diagram below illustrates the two resources involved: the string link in the originating page (blah.html), and the page being retrieved (blah2.html):



The code to do this in HTML would look something like this:

```
<p>This page contains a <a href="blah2.html">link</a> to  
another page.</p>
```

In this case, the resource being pointed to is an *entire* HTML page, but sometimes a link points to just a *section* of an HTML page. To achieve this, we might modify our link above to this:

```
<p>This page contains a <a href="blah2.html#some-section">link</a> to another page.</p>
```

This points the web browser not only to a certain page, but also to a specific section of that page. Of course, even though the link points to that one section, the whole page is loaded into the browser?it is just scrolled to the spot in the document where the hyperlink points. In the HTML world, this is much more intuitive than just having a little piece of a document showing up, since it allows the user to scroll back and forth through the entire document if they want to do so.

However, in order for this to work, the page that we are linking to must also be an HTML document. Furthermore, it must contain special markup (an anchor) to identify the section we want to point to, similar to this:

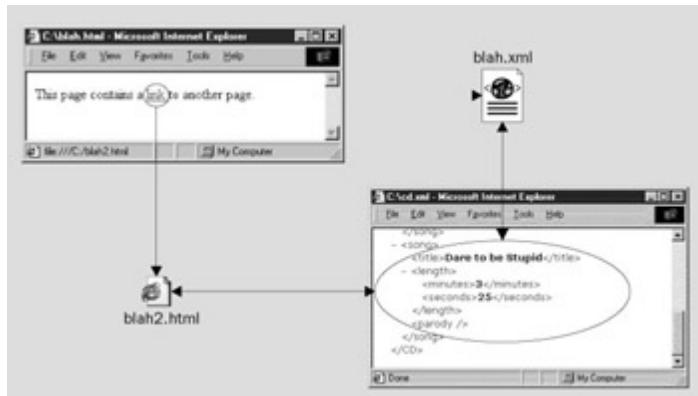
```
<a name="some-section">The link will point to here</a>
```

## XML Linking

The XML Linking Working Group at the W3C (you can find them on the Web at <http://www.w3.org/XML/Linking>) has drafted two specifications to provide these kinds of linking capabilities for XML, and even extend them to allow links in XML to do things that HTML links can't:

- [XLink](#) provides a mechanism for defining links, including links which involve multiple resources, and multiple traversal directions between those resources
- [XPointer](#) provides functionality for pointing to document fragments from XML documents

Using XLink and XPointer together, we can create much more complex links, like the one pictured here:



Here we have a link involving four resources instead of two; XLink would be used to link each file together, while XPointer would be used to point to just a section of, for example, the **cd.xml** file. As you can see from the arrows, not all of the resources are related in one-way relationships. There are a couple of resources that are two-way: for example, not only can we traverse from **blah.xml** to the portion of XML in **cd.xml**, but we can also traverse from that portion of XML back to the **blah.xml** file. In fact, we can do any of the following:

- Traverse from the link in **blah.html** to **blah2.html**
- Traverse from the link in **blah.html** to **blah.xml**
- Traverse from **blah2.html** to a section of **cd.xml**, or from the section of **cd.xml** to **blah2.html**
- Traverse from **blah.xml** to a section of **cd.xml**, or from the section of **cd.xml** to **blah.xml**

What's important to realize, though, is that we're just talking about *one link*?these aren't separate links in each file pointing to each other. In fact, the information about this link doesn't necessarily have to reside in *any* of these files; with XLink, we can define all of this information in a completely separate XML file, if we wish.

The XML Linking Working Group is mainly designing links for use in hypertext applications?that is, displaying XML documents in browser-type applications, the way that HTML documents are displayed today. However, both XLink and XPointer could also be used in other, non-browser-oriented, applications. For example, word processing applications could use XLink to link related documents together, or an order processing system could use XLink to link customers with the orders those customers have placed.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# XLink

XLink provides a number of **global attributes**, which can be attached to any XML element, to define a link.

In this section, we'll study:

- The global attributes provided by XLink, and how they work
- The XLink concept of **simple links**, and the concept of **extended links**, which are not available in HTML links
- What an **arc** is, and how it works

As a demonstration, let's consider our familiar `<name>` example:

```
<name xmlns="http://sernaferna.com/name">
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

We might create an application in which a user could search for information on an employee, and get back information in this format. Now let's suppose that John Doe has a home page, located at <http://sernaferna.com/homepages/doe/>, which we want to link to from our XML. We could do so by adding some XLink attributes to the `<last>` element, like this:

```
<name xmlns="http://sernaferna.com/name"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last xlink:type="simple"
        xlink:href="http://sernaferna.com/homepages/doe/">Doe</last>
</name>
```

Even though the `<last>` element is in the <http://sernaferna.com/name> namespace, we can make it an XLink linking element by adding the type and href global attributes from the XLink namespace.

In the next couple of sections, we'll examine these XLink attributes, and how they work.

## XLink Attributes

In order to provide the functionality mentioned above, XLink provides a number of global attributes that can be added to any element. This is in contrast to HTML, where only a few restricted elements serve as links (such as `<a>` and `<img>`).

*The namespace used for these global attributes is <http://www.w3.org/1999/xlink>. For the purposes of this chapter, however, we will usually only use the xlink: prefix to denote XLink attributes; the namespace declaration will be assumed.*

The attributes are as follows:

-

type specifies the type of XLink element being created

- href specifies the URI used to retrieve a resource
- role is used to specify the function of a link's resource in a machine-readable fashion
- title is used to specify the function of a link's resource in a human-readable fashion
- show specifies how the resource should be displayed, when retrieved
- actuate specifies when the specified resource should be retrieved

The from and to attributes specify link directions, since XLink links are not necessarily one-way links

## The type Attribute

Which attributes can be used on an element depends on the value of type, which is mandatory on any XML element being used to define a link. There are six types:

- simple
- extended
- locator
- arc
- resource
- title

The XLink specification provides a handy table, that summarizes which attributes are required (R), which are optional (O), and which are not allowed (X) for each XLink type:

	simple	extended	locator	arc	resource	title
type	R	R	R	R	R	R
href	O	X	R	X	X	X

<b>role</b>	O	O	O	O	O	X
<b>title</b>	O	O	O	O	O	X
<b>show</b>	O	X	X	O	X	X
<b>actuate</b>	O	X	X	O	X	X
<b>from</b>	X	X	X	O	X	X
<b>to</b>	X	X	X	O	X	X

Because the type attribute defines the type of XLink element being created, its value is used to refer to that type of element. That is, we refer to an element with xlink:type="simple" as a "simple-type element", and to an element with xlink:type="title" as a "title-type element".

Simple-type and extended-type elements are used to define an entire link, while the other types of elements will be children of one of these elements. (Simple and extended links, and the differences between the two, will be covered later.) The locator, arc, resource, and title-type elements are all used in the context of an extended link, to provide more information about the link:

- Locator-type elements are used to indicate remote resources taking part in the link
- Arc-type elements are used to indicate the rules for traversing from one resource to another within the link
- Resource-type elements define the various resources taking part in the link
- Title-type elements can provide human-readable titles for a link

## The href Attribute

The href XLink attribute is called the **locator attribute**, and it does exactly what its name implies: provides a URI where a resource can be located. Since there can be multiple resources involved in an XLink link, there can also be multiple locator attributes (on multiple elements) to provide the URIs to those resources.

The syntax for the href attribute is the same as that for the href attribute in HTML's <a> element. For example, this HTML:

```
<a href="http://sernaferna.com/home.htm">
```

is similar to this XLink example:

```
<myelement xlink:href="http://sernaferna.com/home.htm">
```

For each of these href attributes a simple URL is provided, which points to the resource being identified.

## Semantic Attributes (role and title)

The role and title attributes are used to label the functions of the various resources in the link: role in a machine-readable way, and title in a human-readable way.

- The **role attribute** provides a machine-readable label for a resource. In other words, role is used to give a resource a name which is easily understandable by a computer program.

- The value for the role attribute must be a QName, thus allowing you to distinguish roles you define from roles others define, by means of namespaces. For example:

```
<customer xmlns:order="http://www.sernaferna.com/order"
           xlink:type="resource"
           xlink:role="order:customer">
    John Doe
</customer>
```

In this example, we have a <customer> element, which is also a resource-type XLink element. We have given it the role of order:customer, so that any other XLink elements which need to refer to this resource can do so. Also, since we use QNames for the role attribute's value, we don't have to worry about collisions with others defining a role of customer, since our customer role is differentiated by the URI that the order: namespace prefix is bound to.

The **title attribute** value is just a string that describes this link. The XLink specification doesn't stipulate any functionality that an application should perform with the information in title; it might not even be used at all.

One possible use for it is to create a **tool tip**, which appears if a user hovers their mouse over the link. Consider the following link:

```
<element xmlns:name="http://www.sernaferna.com/name"
          xlink:type="simple"
          xlink:href="http://www.somewhere.com"
          xlink:role="name:first"
          xlink:title="Retrieve first name">
    Click here!
</element>
```

In Mozilla, which supports simple-type XLinks, the code produces this result:



## Behavior Attributes (actuate and show)

XLink allows you some flexibility in defining exactly how links work; that is, *when* the link is traversed, and *how* to treat the resource when it is retrieved.

- - The **actuate attribute** specifies when the resource should be retrieved. XLink provides three possible values for this attribute?onLoad, onRequest, and undefined?but you are allowed to provide other values using a QName:
    -

onLoad works in a similar way to the HTML <img> tag, in that the resource is retrieved as soon as the document is loaded

- 

onRequest works more like the HTML <a> tag, in that the resource is not retrieved until the user requests it

- 

If undefined is specified, the application is free to deal with the link in any way it feels appropriate

Applications can define other behaviors for actuate using a QName. For example, we might define an application that does order processing for our company, and displays a list of outstanding orders to each user. Since this list is dynamic, we could decide that we need the list to be refreshed every five minutes, which could be done with syntax similar to the following:

```
<JobList xmlns:op="http://sernaferna.com/OrderProcessing"  
         xlink:type="simple"  
         xlink:show="replace"  
         xlink:actuate="op:EveryFiveMinutes"  
      />
```

Of course, this example depends on us writing the order processing application to understand and recognize the op:EveryFiveMinutes value for the actuate attribute. If a QName is specified that the application doesn't recognize, it must treat it as if it was undefined.

- - The **show attribute** specifies how to display the resource when it is loaded. XLink provides four possible values for the attribute:
    - new indicates that the resource should be displayed in a separate window; this is like indicating target="\_blank" in HTML's <a> element
    - 
    - replace indicates that the current document should be replaced by the new document, just like the default action for an HTML <a> element
    - 
    - embed indicates that the XLink element should be replaced by the resource, just like an image replaces the <img> element's contents in HTML
    - 
    - undefined?again, the application is free to deal with the link in any way it feels appropriate

As with the actuate attribute, you can supply your own QName for this attribute, and if the application doesn't recognize it, it must be treated the same as undefined. For example, you might define your own value for show which would cause the resource to be popped up in a little window of its own, perhaps to display a definition for a term.

## The from and to Attributes

We mentioned earlier that XLink links can contain multiple resources, and that links can be traversed in multiple directions (For example from ResourceA to ResourceB, or from ResourceB to ResourceA). The from and to attributes are used to define the directionality for a link.

Consider the following:

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:op="http://sernaferna.com/OrderProcessingSystem"
            xlink:type="extended">
  <salesperson xlink:type="locator"
    xlink:href="http://sernaferna.com/JohnDoe.xml"
    xlink:role="op:salesperson"
    xlink:title="Salesperson"/>
  <order xlink:type="locator"
    xlink:href="http://sernaferna.com/order256.xml"
    xlink:role="op:order"
    xlink:title="Order"/>
  <GetOrder xlink:type="arc"
    xlink:from="op:salesperson"
    xlink:to="op:order"
    xlink:role="op:GetOrder"/>
</PartNumber>
```

The <GetOrder> element defines an arc. This arc will go from the resource with a role of op:salesperson to the resource with a role of op:order. In this case, the arc will be from the resource defined in the <salesperson> element to the resource defined in the <order> element.

## Link Types

Now that we've looked at the various attributes, let's put them into practice. XLink defines two main types of links: a **simple link**, and an **extended link**.

A simple link is what we've been working with all along in HTML. It deals with only two resources, and is unidirectional. (That is, it goes from the "FromResource" to the "ToResource", but not the other way around.)

Extended links, on the other hand, provide extra functionality that HTML links don't; for example, they can involve more than two resources, and they allow you to specify complex traversal rules between the various resources. While simple links usually only involve a single XML element, with the XLink attributes added, extended links consist of a number of elements, which are all children or descendants of an extended-type link element.

The next sections will go into more detail about simple and extended links, including all of the elements that are available (or mandatory) for an extended link.

### Simple Links

#### Important

Simple links provide the same functionality as the hyperlinks provided in HTML: they are one-way links, involving only two resources (the source and the destination).

To specify a simple link, we set the type attribute's value to "simple", and specify where to get the destination resource using the href attribute.

The following two links, in HTML and in XML using XLink, provide the same basic functionality, which is to replace the current document with the one being linked to:

```
<a href="http://somewhere.com">Click me!</a>
```

```
<SomeElement xlink:type="simple" xlink:href="http://somewhere.com/">
    Click me!
</SomeElement>
```

The following two links also provide the same basic functionality, which is to open up a document, in a new window, and to provide a title for the user, which would probably be shown if the mouse is hovered over the link. First the HTML version:

```
<a href="http://somewhere.com/"
    title="Click to go somewhere"
    target="_blank">
    Click me!
</a>
```

and then the XML version:

```
<SomeElement xlink:type="simple"
    xlink:href="http://somewhere.com"
    xlink:title="Click to go somewhere"
    xlink:show="new">
    Click me!
</SomeElement>
```

Simple links are called **inline links**, because the link's own content (in this case the Click me! PCDATA) serves as one of the resources. As we will see in the [next section](#), not all XLink links have to be inline, but all *simple* ones do.

## Try It Out?Creating Simple Links

1.

To try out our knowledge of simple links, let's create a quick XML document that provides some information about a person:

```
<?xml version="1.0"?>
<person>
    <name>???</name>
    <picture>???</picture>
    <homepage>???</homepage>
</person>
```

We'll assume that we would have a CSS style sheet to display this document in an XLink-aware browser. As you can see, there are placeholders (???) in here, where the actual information will go, so let's start to fill them in.

2.

For the <picture> element, we'll assume that there is a JPEG picture in this directory, called picture.jpg, which is a picture of the person we're describing; we can easily pull that into the document. However, the people looking at our XML document might be on slow modems; they shouldn't have to download our picture unless they really want to see it. So we'll set the picture to only load when they request it, and instead display some text to let them know that this is the case:

```
<?xml version="1.0"?>
<person xmlns:xlink="http://www.w3.org/1999/xlink">
    <name>???</name>
    <picture xlink:type="simple"
        xlink:href="picture.jpg"
        xlink:actuate="onRequest"
        xlink:show="embed"
```

```
xlink:title="Click to see picture!">
Click here to see a picture!
</picture>
<homepage>???</homepage>
</person>
```

This will display the text Click here to see a picture!, but when the user clicks that text it will be replaced with the picture itself.

3.

Since so many people have home pages, we'll also include a spot in here to point to this person's home page.

This link will function just like an HTML <a> element:

```
<?xml version="1.0"?>
<person xmlns:xlink="http://www.w3.org/1999/xlink">
<name>???</name>
<picture xlink:type="simple"
          xlink:href="picture.jpg"
          xlink:actuate="onRequest"
          xlink:show="embed"
          xlink:title="Click to see picture!">
    Click here to see a picture!
</picture>
<homepage xlink:type="simple"
          xlink:href="http://www.sernaferna.com/homepages/personal.htm"
          xlink:actuate="onRequest"
          xlink:show="replace">
    Click here for the homepage.
</homepage>
</person>
```

4.

Finally, we need to take care of the <name> element. For this example, let's say we happen to know there's an XML file, name.xml, which is in the same directory as our XML document, and which contains all of the information we need. So, let's pull that information in as well:

```
<?xml version="1.0"?>
<person xmlns:xlink="http://www.w3.org/1999/xlink">
<name xlink:type="simple"
      xlink:href="name.xml"
      xlink:actuate="onLoad"
      xlink:show="embed"/>
<picture xlink:type="simple"
          xlink:href="picture.jpg"
          xlink:actuate="onRequest"
          xlink:show="embed"
          xlink:title="Click to see picture!">
    Click here to see a picture!
</picture>
<homepage xlink:type="simple"
          xlink:href="http://www.sernaferna.com/homepages/personal.htm"
          xlink:actuate="onRequest"
          xlink:show="replace">
    Click here for the homepage.
</homepage>
</person>
```

This pulls the information right into our document; it's similar to specifying an external file in HTML's src attribute on the <SCRIPT> element. In other words, as far as the browser is concerned this is one big document, as if the contents of name.xml had been typed into this document, instead of the <name> element. (For readers familiar with C or C++, this is very much the same as using the #include directive.)

*Interestingly, because Mozilla partially implements XLink, parts of this Try It Out will work with it.*

The link to [www.sernaferna.com](http://www.sernaferna.com) will work, but the other two will not, as Mozilla hasn't implemented XLinks where the show attribute is set to "embed".

## Extended Links

### Important

An extended link is one that associates any number of resources.

An extended link can be inline (like a simple link) if the link's own content serves as one of the resources, or it can be **out of line**, meaning that all of the link's resources are remote. This out of line functionality allows us to define links in separate, external files, without having to make any modifications to the documents that serve as our resources. So, for example, you could create a link for your application from **Document A** to **Document B**, even if you don't have access to modify either document! In order to do this, you would create a third document, which defines all of the necessary links, which would be processed by the application.

As an example of where extended links might be useful, consider a corporate web site. If you decided to include a simple footer at the bottom of each page, saying something like "Copyright ( 2001 My Company Inc.", you could create an XML file with an extended link to include that file in all of the others on the site, without having to change any of those other files.

Extended links are defined in an extended-type element by this attribute:

```
xlink:type="extended"
```

They have one or more child elements, which define the local and remote resources participating in the link, traversal rules for those resources, and human-readable labels for the link.

Extended-type elements can also have the semantic attributes (role and title), in which case they apply to the *whole* link. However, as we'll see, we can apply these attributes to *specific* resources within our link as well.

### Try It Out?Creating an Extended Link

For this example, we'll look at a fictitious application that can display information about a part number to a user. The user can then navigate to the last order that was placed for that part number, and to the salesperson who made that order.

Unfortunately, we don't yet have enough information to create a whole extended link?we'll pick up that information in the next few sections. So for now, we'll just have to content ourselves with the extended-type element itself.

5.

First, we'll create the XML document, parts.xml, which will contain the link:

```
<PartNumber>
<item>
  <part-number>E16-25A</part-number>
  <description>Production-Class Widget</description>
</item>
</PartNumber>
```

6.

Next, we'll change the <PartNumber> element, to be an extended-type link element:

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
            xlink:type="extended">
<item>
```

```
<part-number>E16-25A</part-number>
<description>Production-Class Widget</description>
</item>
</PartNumber>
```

## How It Works

We have declared <PartNumber> to be an extended-type element. Our link doesn't yet have any functionality, or even any resources, but we'll add those as we expand this example.

## Locator-type Elements

### Important

Locator-type elements are child elements of extended-type elements, used to indicate the remote (out of line) resources taking part in an extended link.

Along with the type attribute, locator-type elements must also include the locator attribute (href), which specifies the URI pointing to the resource in question. If desired, it can also include the role and title attributes, but they are not required.

The following is an example of a locator-type element, with all of the available XLink attributes specified:

```
<salesperson xlink:type="locator"
              xlink:href="http://sernaferna.com/JohnDoe.xml"
              xlink:role="op:salesperson"
              xlink:title="Salesperson"/>
```

## Try It Out?Adding Remote Resources to the Extended Link

With this new information, we can now add a couple of resources to our extended link. We'll add links to the salesperson that placed the order, and the order itself.

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:op="http://sernaferna.com/OrderProcessingSystem"
            xlink:type="extended">
  <item>
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
  </item>
  <salesperson xlink:type="locator"
              xlink:href="http://sernaferna.com/JohnDoe.xml"
              xlink:role="op:salesperson"
              xlink:title="Salesperson"/>
  <order xlink:type="locator"
        xlink:href="http://sernaferna.com/order256.xml"
        xlink:role="op:order"
        xlink:title="Order"/>
</PartNumber>
```

## How It Works

Our link now has two external resources, and specifies the URIs where these resources can be found, for example:

```
<salesperson xlink:type="locator"
              xlink:href="http://sernaferna.com/JohnDoe.xml"
```

Because we have given them roles, we need to declare the namespace being used for our QNames:

```
  xmlns:op="http://sernaferna.com/OrderProcessingSystem"
```

We still don't know the relationship between these resources, but we'll take care of that next.

## Arcs

### Important

The rules for traversing from one resource to another in a link are grouped together into an entity called an arc, which is provided by an arc-type element.

Arc-type elements define the direction in which the link must traverse, using the from and to attributes, and the behavior the link will follow when it retrieves these resources, using the show and actuate attributes.

Arc-type elements can even define the role and title attributes, which will define not the resource being retrieved, but the resource in the context of this arc. For example, in our previous example we have a resource with a title of Salesperson, and one with a title of Order. So, if we defined an arc going from the Order resource to this Salesperson resource, we might title the arc Salesperson's name. However, if the arc was defined as going the other way, we might call it Last order processed, as that makes more sense for this arc.

### Important

Every available XLink attribute is allowable with an arc-type element, except for href. An arc-type element is the only place where the XLink from and to attributes are used.

The following is an example of an arc-type element, specifying all of the available attributes:

```
<GetPhoto xlink:type="arc"
    xlink:from="myapp:name"
    xlink:to="myapp:photo"
    xlink:show="replace"
    xlink:actuate="onRequest"
    xlink:role="myapp:GetPhoto"
    xlink:title="Show photo"/>
```

This specifies that the GetPhoto element acts as a link, and has a title of "Show photo". As we mentioned, if this document is being shown in some type of browser, the title might be displayed when the from resource is hovered over with the mouse. When the link is activated, the link element with role myapp:name should be replaced with the link element with role myapp:photo. This link is identified as myapp:GetPhoto. The myapp:name and myapp:photo link elements will be defined elsewhere.

## Try It Out?Adding Arcs to the Extended Link

So far we have two resources in our extended link, but we haven't defined how our application is allowed to traverse between these resources. For our purposes, it's OK to go either way, so we'll define two arcs, one for each direction:

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:op="http://sernaferna.com/OrderProcessingSystem"
    xlink:type="extended">
<item>
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
</item>
<salesperson xlink:type="locator"
    xlink:href="http://sernaferna.com/JohnDoe.xml"
    xlink:role="op:salesperson"
    xlink:title="Salesperson"/>
<order xlink:type="locator"
    xlink:href="http://sernaferna.com/order256.xml"
    xlink:role="op:order"
    xlink:title="Order"/>
<GetOrder xlink:type="arc"
    xlink:from="op:salesperson"
```

```
xlink:to="op:order"
xlink:show="replace"
xlink:actuate="onRequest"
xlink:role="op:GetOrder"
xlink:title="Last order processed."/>
<GetSalesperson xlink:type="arc"
    xlink:from="op:order"
    xlink:to="op:salesperson"
    xlink:show="replace"
    xlink:actuate="onRequest"
    xlink:role="op:GetSalesperson"
    xlink:title="Salesperson's name"/>
</PartNumber>
```

## How It Works

With these arcs defined, we can go from an order to a salesperson's name, and from a salesperson's name to an order. In both cases, the new resource will replace the current one, since we have specified replace for the show attribute, and the link won't be traversed until the user requests it, because we've specified onRequest for the actuate attribute.

## Resource-type Elements

As we mentioned earlier, extended links can be inline if at least one of the resources specified is local.

### Important

Resource-type elements are used to create local resources, by putting an element declared as a resource-type under the extended-type element.

Resource-type elements can only have the role and title attributes, both of which are optional. (The href, show, and actuate attributes wouldn't really make sense for a local resource.)

This makes resource-type elements very simple:

```
<item xlink:type="resource"
    xlink:role="order:item"
    xlink:title="Item">
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
</item>
```

Everything in the `<item>` element, including its children, is part of the resource.

## Try It Out?Adding a Local Resource to the Extended Link

For our example, we'll create a local resource to describe a part number. We'll then have that part number link to the op:order resource, leaving in place the links between the op:order resource and the op:name resource:

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:op="http://sernaferna.com/OrderProcessingSystem"
    xlink:type="extended">
    <item xlink:type="resource"
        xlink:role="op:item"
        xlink:title="Item">
        <part-number>E16-25A</part-number>
        <description>Production-Class Widget</description>
    </item>
    <salesperson xlink:type="locator"
        xlink:href="http://sernaferna.com/JohnDoe.xml"
        xlink:role="op:salesperson"
```

```
xlink:title="Salesperson"/>
<order xlink:type="locator"
      xlink:href="http://sernaferna.com/order256.xml"
      xlink:role="op:order"
      xlink:title="Order"/>
<GetOrder xlink:type="arc"
          xlink:from="op:salesperson"
          xlink:to="op:order"
          xlink:show="replace"
          xlink:actuate="onRequest"
          xlink:role="op:GetOrder"
          xlink:title="Last order processed."/>
<GetSalesperson xlink:type="arc"
                xlink:from="op:order"
                xlink:to="op:salesperson"
                xlink:show="replace"
                xlink:actuate="onRequest"
                xlink:role="op:GetSalesperson"
                xlink:title="Salesperson's name"/>
<GetItemOrder xlink:type="arc"
              xlink:from="op:item"
              xlink:to="op:order"
              xlink:show="new"
              xlink:actuate="onRequest"
              xlink:role="op:GetItemOrder"
              xlink:title="Last order placed for this item"/>
</PartNumber>
```

## How It Works

What we've effectively done is create a local resource, which will go on the main page in the user's browser, and which describes a part number.

We have defined not only a local resource, but also an arc to open up an order from that resource. If the user clicks on that part number, a new window will open up, with the order information, and the user can then navigate back and forth in that new window between the order and the salesperson's name XML documents.

The end result might look something like this:



Or like this:



## Title-type Elements

### Important

Title-type elements are simple elements, which can be used in lieu of the title attribute if so desired.

The benefit of using title-type elements instead of the title attribute is that elements can include other elements as children. One common example might be to add HTML markup to the title. This means we could rewrite:

```
<GetItemOrder xlink:type="arc"  
    xlink:title="Last order placed for this item"/>
```

as:

```
<GetItemOrder xlink:type="arc">  
    <description xlink:type="title">  
        Last <b>order</b> placed for this <i>item</i>.  
    </description>  
</GetItemOrder>
```

By using a title-type element (<description>) we can include <b> and <i> child elements to make the link title more attractive.

## Defaulting XLink Attributes

Because there are so many XLink global attributes, the XLink specification assumes that you will be using DTDs to provide default values for them. This can prove to be a great time-saver, and also makes your documents more readable.

For example, suppose you have a <definition> element that you will always be using in your documents to point to a definition somewhere. The link will be simple, and will just pop up the definition in a little window. You might create a <definition> element like this:

```
<definition xmlns:show="http://www.sernaferna.com/show"  
    xlink:type="simple"  
    xlink:href="definition1.xml"  
    xlink:actuate="onRequest"  
    xlink:show="show:popup"  
    xlink:title="Click for definition">  
    widget  
</definition>
```

This defines a link from the word "widget" to the definition of that word. But, as you can see, most of the attributes will be exactly the same for every instance of the <definition> element. For this reason, we could give them all defaults in our DTD, such as the following:

```
<!ELEMENT definition ANY>  
<!ATTLIST definition
```

```
xmlns:show    CDATA      #FIXED "http://www.sernaferna.com/show"
xmlns:xlink   CDATA      #FIXED "http://www.w3.org/1999/xlink"
xlink:type    (simple)   #FIXED "simple"
xlink:href    CDATA      #IMPLIED
xlink:title   CDATA      #FIXED "Click for definition"
xlink:show    (show:popup) #FIXED "show:popup"
xlink:actuate (onLoad
               |onRequest
               |undefined) #FIXED "onRequest"
>
```

This makes creating an instance of <definition> as easy as this:

```
<definition xlink:href="definition1.xml">widget</definition>
```

which is much easier to read.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# XPointer: Pointing to Document Fragments

So far we've covered how we can create links that involve numerous resources, with varying arcs to and from those resources, and even how we can control the behavior of those arcs. But we're still dealing with entire documents. What about those times when we only need part of a document: a few elements, or even a few characters, from an XML document?

The **XPointer** specification provides a method for pointing to pieces of an XML document. These XPointer expressions can then be appended to URIs, making them useful in the context of links. For example, instead of creating a link to the names.xml document, you might create a link that points to the first `<name>` element in names.xml, or to all `<name>` elements where the child `<first>` element contains the value "John", etc.

This may seem very similar to the types of expressions you learned how to write in [Chapter 4](#), using XPath. In fact, XPointer is built on XPath, so everything you learned before still applies, but it also adds some extensions to basic XPath expressions. These extensions to XPath allow XPointer expressions to:

- Locate information by string matching
- Be appended to URIs, to expand the power and flexibility of links
- Address not only entire nodes in an XML document, but pieces of nodes

As an example of the last point, we can write an XPath expression to return the `<name>` element, but we could write an XPointer expression to return the first few characters of the text in the `<name>` element. Or we could define a spot in the document before the first `<name>` element, and another spot after the first `<middle>` element, and take everything in between these two spots.

Because of this extended functionality, XPointer no longer refers to "nodes"; instead, the specification generalizes the term further to be called a **location**. These locations can be of any node type which was allowed in XPath, but can also be one of two new types: **points** and **ranges**. Accordingly, instead of dealing in node-sets, XPointer deals in **location sets**. (More on these terms coming up.)

Most importantly, XPointer provides a syntax for appending an XPointer expression to the end of a URI, where this expression specifies the piece of the document you want. So not only can we say "get this document from this location", but we can go even further and say "get this *piece* of this document from this location".

*When working with XPointer, you will always be dealing with XML documents: by contrast, XLink can create links to any type of documents, XML or otherwise.*

## Appending XPointer Expressions to URIs

As we mentioned earlier, we can currently create hyperlinks to particular sections of HTML documents by specifying a URI like this:

`http://www.sernaferna.com/default/page.htm#section-one`

In order for this to work, the HTML document, page.htm, has to have an anchor in it, like this:

```
<a name="section-one">
```

XPointer uses a similar syntax to return pointers to parts of an XML document. For example, consider the following:

```
http://www.sernaferna.com/default/page.xml#xpointer(//section-one)
```

This URI points to a file called page.xml, but an XPointer expression is appended to the end, enclosed between the beginning "xpointer(" string and the closing ")". This expression is really just an XPath expression with a few XPointer extensions. When the XML document is retrieved, the XPointer-aware application would only be concerned with <section-one> elements, just as if we had used the following in XSLT:

```
<xsl:value-of select="//section-one"/>
```

In other words, we could have a page.xml document such as the following:

```
<?xml version="1.0"?>
<document>
  <section-one>
    <name>John Doe</name>
  </section-one>
  <section-two>
    <company>Serna Ferna Inc.</company>
  </section-two>
</document>
```

along with a simple link, defined as follows:

```
<SectionOneName xlink:type="simple"
  xlink:href="http://www.sernaferna.com/default/page.xml#xpointer(//section-
  one)"
  xlink:actuate="onLoad"
  xlink:show="embed"
  xmlns:xlink="http://www.w3.org/1999/xlink"/>
```

The net result would be a <SectionOneName> which would be similar to this:

```
<SectionOneName>
  <section-one>
    <name>John Doe</name>
  </section-one>
</SectionOneName>
```

Of course, the key difference between this XPointer expression and the URL to the HTML document above is that the HTML document needed an anchor; for XPointers to work, we don't need to modify the XML document at all.

## Using Multiple XPointer Expressions

In cases where we want to use multiple XPointer expressions, we can chain them together. XPointer reads the expressions from left to right; if one expression fails, the next one is evaluated. The results from the first expression to be evaluated are taken, and the rest ignored (note that an expression returning an empty location set is not the same as failing).

Consider the following example:

```
xpointer(id("section-one")) xpointer(//*[@id="section-one"])
```

If there is a DTD associated with this XML document, which defines an ID attribute, then the first XPointer expression (`id("section-one")`) will be evaluated and the results will be returned. However, if there is no DTD associated with this XML document, the expression will fail. The reason for this is that we can't have ID attributes without a DTD, because without the DTD the processor does not know which attributes are IDs. In that case, the second expression would be evaluated, which would return a location set of any elements with an attribute *named id* that have a value of "section-one". Again, that location set could possibly be empty, which is not the same as failing.

## Try It Out-Retrieving a Piece of a Document

In our extended link Try It Outs, we had separate XML documents containing the information for an order and the name of the salesperson who completed that order. Let's modify that example.

1.

Suppose our order XML (`order256.xml`) was structured like this:

```
<?xml version="1.0"?>
<order>
  <name>
    <first>John</first>
    <middle/>
    <last>Doe</last>
  </name>
  <item>Production-Class Widget</item>
  <quantity>16</quantity>
  <date>
    <m>1</m>
    <d>1</d>
    <y>2000</y>
  </date>
  <customer>Sally Finkelstein</customer>
</order>
```

Why, it includes the salesperson's name right in it! There's no need to get a separate document for it.

2.

We can then restructure our extended link like the following:

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:op="http://sernaferna.com/OrderProcessingSystem"
            xlink:type="extended">
  <item xlink:type="resource"
        xlink:role="op:item"
        xlink:title="Item">
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
  </item>
  <salesperson xlink:type="locator"
               xlink:href="http://sernaferna.com/order256.xml#xpointer(/order/name)"
               xlink:role="op:salesperson"
               xlink:title="Salesperson"/>
  <order xlink:type="locator"
        xlink:href="http://sernaferna.com/order256.xml"
        xlink:role="op:order"
        xlink:title="Order"/>
  <GetOrder xlink:type="arc"
            xlink:from="op:salesperson"
            xlink:to="op:order"
```

```
xlink:show="replace"
xlink:actuate="onRequest"
xlink:role="op:GetOrder"
xlink:title="Last order processed."/>
<GetSalesperson xlink:type="arc"
    xlink:from="op:order"
    xlink:to="op:salesperson"
    xlink:show="replace"
    xlink:actuate="onRequest"
    xlink:role="op:GetSalesperson"
    xlink:title="Salesperson's name"/>
<GetItemOrder xlink:type="arc"
    xlink:from="op:item"
    xlink:to="op:order"
    xlink:show="new"
    xlink:actuate="onRequest"
    xlink:role="op:GetItemOrder"
    xlink:title="Last order placed for this item"/>
</PartNumber>
```

## How It Works

What we have done, in effect, is create two resources (Salesperson and Order) which point to the same document; traversing from one resource to the other would really just be hiding most of the document (only showing the salesperson's name), or un-hiding it again (and showing the entire order document).

## XPointer Schemes

In our URL, we specified a syntax like this for our XPointers:

```
#xpointer(expression)
```

The expression is easy-it's just XPath with some XPointer additions-but what does the xpointer really mean? It's actually called a **scheme**.

### Important

A scheme identifies what type of XML document we are getting our information from.

The only scheme included in the XPointer specification is the xpointer scheme we've been using; it applies to any XML document.

We could create schemes for other specific document types. For example, there is an XML format for describing graphics, called SVG (Scalable Vector Graphics), and a scheme could be developed explicitly for retrieving parts of an SVG document. We could also create schemes for our own, proprietary, XML document types.

However, in order to use any scheme other than xpointer, you would need software that not only understood XPointers, but also understood your particular scheme.

## XPointer Shorthand Syntaxes

In addition to the full XPointer syntax, there are also a couple of shorthand syntaxes available:

- 
- 
- 

The bare names syntax

## The child sequence syntax

For these examples, we'll create a simple XML document, with an ID attribute:

```
<?xml version="1.0"?>
<!DOCTYPE XPointerSample [
  <!ELEMENT XPointerSample (name)>
  <!ELEMENT name (first, middle, last)>
  <!ELEMENT first (#PCDATA)>
  <!ELEMENT middle (#PCDATA)>
  <!ELEMENT last (#PCDATA)>
  <!ATTLIST first id ID #REQUIRED>
]>
<XPointerSample>
  <name>
    <first id="section-one">John</first>
    <middle>Fitzgerald Johansen</middle>
    <last>Doe</last>
  </name>
</XPointerSample>
```

We'll assume this file is stored on a web server, at the location <http://www.sernaferna.com/default/page.xml>.

## Full Syntax

One common usage of XPointer is likely to be retrieving an element from an XML document based on its ID. The full syntax for the href attribute would be like this:

```
http://www.sernaferna.com/default/page.xml#xpointer(id("section-one"))
```

This will retrieve an element which has an id attribute, with a value of "section-one"; in this case, the <first> element.

## Bare Names Syntax

The bare names syntax allows us to retrieve an element by its ID like this:

```
http://www.sernaferna.com/default/page.xml#section-one
```

which has the same effect as the XPointer expression above. You may notice that this syntax is the same as the earlier URI we saw for retrieving an HTML document. Part of the reason that XPointer provides this shorthand syntax is to provide a mechanism that is analogous to HTML's method of retrieving a document. (The other reason for providing this shorthand is to encourage the use of IDs.)

## Child Sequence Syntax

The third way we can specify XPointer expressions at the end of a URI is to use a child sequence:

```
http://www.sernaferna.com/default/page.xml#/1/1/2
```

The numbers after the "/" characters are child element numbers of the previously selected elements. In other words, this expression means "the second child element of the first child element of the first child element". When used right at the beginning of the child sequence, /1 means the root element. The child sequence and bare names syntaxes can only retrieve elements from the XML document; for other node-types, such as attributes or text nodes, you need to use the full XPointer syntax. This XPointer expression is equivalent to:

[http://www.sernaferna.com/default/page.xml#xpointer\(/\\*\[1\]/\\*\[1\]/\\*\[2\]\)](http://www.sernaferna.com/default/page.xml#xpointer(/*[1]/*[1]/*[2]))

which says to select the second child of the first child of the document root or, in this case, the <middle> element.

Instead of specifying /1 as the starting point of a child sequence, we can also append a child sequence to a bare name expression, such as in the following:

<http://www.sernaferna.com/default/page.xml#section-one/3>

This will take the third child element of the element with an ID of section-one.

#### Important

An obvious drawback to ordinal references (in other words, using numbers as in the examples above) is that if the structure of the XML file changes, any links into it will be made invalid. This might not be the case had the links used an actual XPath expression. There are actually very few cases where you would want to use the child sequence syntax, rather than referring to elements and attributes by name.

## Locations, Points and Ranges

We're already used to the idea of nodes; these represent elements, attributes, Processing Instructions, etc. We're also familiar with node-sets, which are collections of nodes. However, as mentioned earlier, XPointer adds the concepts of a **location** and a **location set**.

A location can be any node type that is allowed in XPath, or one of the positional declarations known as **points** and **ranges**. A **location set** is an ordered list of locations.

XPointer allows us even greater flexibility than XPath, because we don't have to retrieve whole nodes, but can retrieve pieces of nodes (for example, the first few characters of a text node, or the last few characters of the text node in one element and the first few characters of the text node from the next element.) The way we can do this is by selecting ranges, which is a part of the document between two points. Let's look at the concepts of points and ranges.

## Points

#### Important

A point is simply a spot in the XML document. It is defined using the usual XPointer expressions.

There are two pieces of information needed to define a point: a container node and an index. Points are located between bits of XML; that is, between two elements or between two characters in a CDATA section. Whether the point refers to characters or elements depends on the nature of the container node. An index of zero indicates the point before any child nodes, and a non-zero index  $n$  indicates the point immediately after the  $n$ th child node. (So an index of 5 indicates the point right after the 5th child node.)

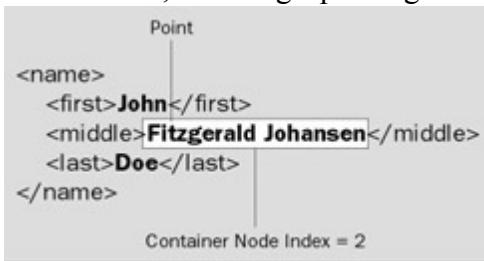
When the container node is an element (or the document root), the index becomes an index of the child elements, and the point is called a node-point. In the following diagram, the container node is the <name> element, and the index is 2. This means the point indicates a spot right after the second child element of <name>, which is the <middle> element:



In XPointer, we can use the start-point() or end-point() functions, to return a starting point or ending point for a location. For example, the XPointer syntax for the point above would be:

```
#xpointer(start-point(//name/*[2]))
```

If the container is any other node-type, the index refers to the characters of the string value of that node, and the point is called a character-point. In the following diagram, the container node is the PCDATA child of <middle>, and the index is 2, indicating a point right after the "i" and right before the "t" of Fitzgerald:



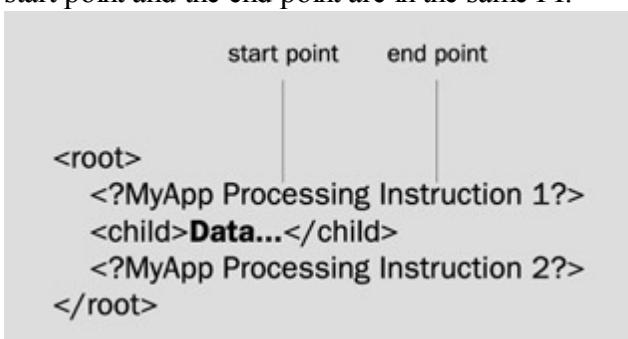
## Ranges

### Important

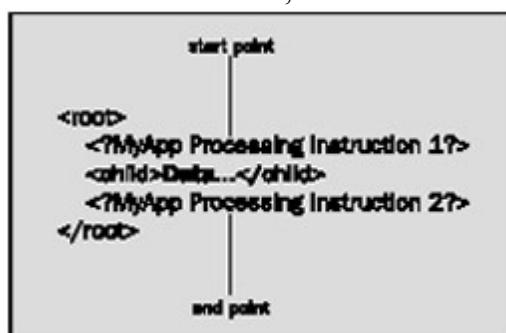
A range is defined by two points-a start point and an end point-and consists of all of the XML structure and content between those two points.

The start point and end point must both be in the same document. If the start point and the end point are equal, the range is a collapsed range. However, a range can't have a start point that is *later* in the document than the end point.

If the container node of either point is anything other than an element node, text node, or document root node, then the container node of the other point must be the same. For example, the following range is valid, because both the start point and the end point are in the same PI:



whereas this one is not, because the start point and end point are in different PIs:



The concept of a range is the reason that the XPath usage of nodes and node-sets weren't good enough for XPointer; the information contained in a range might include only parts of nodes, which XPath can't handle.

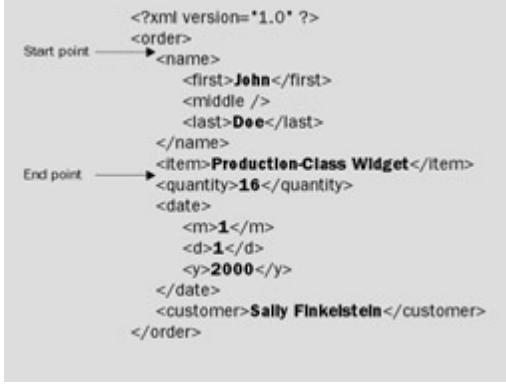
## How Do We Select Ranges?

XPointer adds the range-to() function, which we can insert in our XPointer expressions to specify a range. It's used as follows:

```
xpointer(/order/name/range-to(order/item))
```

The beginning of this XPath expression should be pretty familiar to us—we're selecting the <name> element, which is a child of the <order> element, which is a child of the document root. We then have what looks like another step in the expression, starting with a "/" character, but instead of specifying a child or axis name, we specify the range-to() function. This is an extension of the XPath syntax, and XPointer software will know that this means we're now specifying the end point in our range.

This selects a range where the start point is just before the <name> element, and the end point is just after the <item> element:



In other words, the <name> and <item> elements would both be part of this range, which would be the only member in the location set.

## Ranges with Multiple Locations

This is pretty easy when the expressions on either side of the range-to() function return a single location, but what about when the expressions return multiple locations in their location sets? Well then things get a bit more complicated. Let's create an example, and work our way through it.

Consider the following XML:

```
<people>
  <person name="John">
    <phone>(555) 555-1212</phone>
    <phone>(555) 555-1213</phone>
  </person>
  <person name="David">
    <phone>(555) 555-1214</phone>
  </person>
  <person name="Andrea">
    <phone>(555) 555-1215</phone>
    <phone>(555) 555-1216</phone>
    <phone>(555) 555-1217</phone>
  </person>
  <person name="Ify">
    <phone>(555) 555-1218</phone>
    <phone>(555) 555-1219</phone>
```

```
</person>
<!--more people could follow-->
</people>
```

We have a list of people, and each person can have one or more phone numbers. Now consider the following XPointer:

```
xpointer("//person/range-to(phone[1]))
```

As you can see, the first expression will return a number of `<person>` elements, and the second expression will return the first `<phone>` element. XPointer tackles this as follows:

1.

It evaluates the expression on the left side of the `to` keyword, and saves the returned location set. In this case, it will be a location set of all of the `<person>` elements.

```
<person name="John"/>
<person name="David"/>
<person name="Andrea"/>
<person name="Ify"/>
```

2.

Using the first location in that set as the context location, XPath then evaluates the expression on the right side of the `to` keyword. In this case, it will select the first `<phone>` child of the first `<person>` element in the location set on the left.

```
<person name="John"/> ----- <phone>(555)555-1212</phone>
<person name="David"/>
<person name="Andrea"/>
<person name="Ify"/>
```

3.

For each location in this second location set, XPointer adds a range to the result, with the start point at the beginning of the location in the first location set, and the end point at the end of the location in the second location set. In this case, only one range will be created, since the second expression only returned one location.

```
Range
----->
<person name="John"/> ----- <phone>(555)555-1212</phone>
<person name="David"/>
<person name="Andrea"/>
<person name="Ify"/>
```

4.

Steps 2 and 3 are then repeated for each location in the first location set, with all of the additional ranges being added to the result. So, as a result of the XPointer above, we would end up with the following pieces of XML selected in our document:

```
<person name="John">
  <phone>(555) 555-1212</phone>
<person name="David">
  <phone>(555) 555-1214</phone>
<person name="Andrea">
  <phone>(555) 555-1215</phone>
<person name="Ify">
  <phone>(555) 555-1218</phone>
```

## XPointer Function Extensions to XPath

Of course, to deal with these new concepts, XPointer adds a few functions to the ones supplied by XPath. We won't go into their details here, but the following is a brief description of the new functions (you can get full details from the XPointer working draft, at <http://www.w3.org/TR/xptr/>):

- We already mentioned the start-point() and end-point() functions. start-point() takes a location set as a parameter, and returns a location set containing the start points of all of the locations in the location set. For example, start-point("//cityName[1]") would return the start point of the first <cityName> element in the document, and start-point("//cityName") would return a set containing the start points of all of the <cityName> elements in the document. end-point() works exactly the same, but returns end points.
- The range() and range-inside() functions return ranges. range() takes a location set as a parameter, and returns another location set. For each location in the input, it creates a range for that location which it adds to the output. range-inside() works similarly, taking in and returning location sets, except that the range it creates for each location is only for the *contents* of the location, not the entire thing.
- The here() function returns the element which contains the XPointer. That is, if we define an XPointer which points to a specific piece of an XML document, here() returns the element which contains that piece.
- The origin() function returns the element from which a user or program initiated traversal of a link. It is used in the context of out of line XLink links.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Querying

Along with XLink and XPointer, there is also an **XQuery** Working Group in place at the W3C, which is creating a query language, **XQuery**, that can retrieve data from XML documents across the Web. XQuery is intended to be as universal as the SQL language is for querying relational databases. (The working group's page at <http://www.w3.org/XML/Query> says that their goal is to provide "*the needed interaction between the web world and the database world. Ultimately, collections of XML files will be accessed like databases.*") Work on this query language is influenced by both SQL and XPath.

There are a number of documents under work at the W3C from this working group. Aside from the working draft for the language itself, at <http://www.w3.org/TR/xquery/>, there is a **Requirements** working draft (<http://www.w3.org/TR/xmlquery-req>), as well as a **Use Cases** working draft (<http://www.w3.org/TR/xmlquery-use-cases>). These two documents outline what XQuery should eventually be designed to do. There is also a working draft for the **Formal Semantics** of the language (<http://www.w3.org/TR/query-semantics/>), and a working draft for the **Data Model** (<http://www.w3.org/TR/query-datamodel/>).

This seems like a lot of working drafts for one language, but work on XQuery is considered to be fundamental, and it is expected that this language will be implemented very widely. In fact, some of the work on XQuery is considered so integral to XML that it is being incorporated right into the next version of the XPath language, which will be XPath 2.0 - the Data Model document mentioned above is actually for both XQuery and XPath 2.0.

## XQuery Syntaxes

One major difference between XQuery and other query languages, such as SQL and XPath, is that XQuery will have multiple syntaxes. At the minimum it will have a syntax in XML and another more human-readable syntax. The human-readable syntax is defined in the working draft itself, while the XML syntax, called **XQueryX**, is defined in a separate working draft, at <http://www.w3.org/TR/xqueryx>.

A quick glance at the XQueryX working draft shows that the XML syntax for XQuery will be much more verbose than the human-readable syntax. For example, suppose we have a document that conforms to the following DTD (from the XML Query Use Cases document):

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book year CDATA #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last   (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher (#PCDATA )>
<!ELEMENT price   (#PCDATA )>
```

The XQuery Working draft gives an example query to list each publisher in the XML document, and the average price of its books, which is

```
FOR $p IN distinct(document("bib.xml")//publisher)
LET $a := avg(document("bib.xml")//book[publisher = $p]/price)
RETURN
    <publisher>
```

```
<name> {$p/text()} </name>
<avgprice> {$a} </avgprice>
</publisher>
```

Since XQuery is only in Working Draft stage, the final syntax could change, meaning that the query above may not end up being proper XQuery syntax.

This statement is doing the following:

- - Creating a variable, named p, which will contain a list of all of the distinct <publisher> elements from the document bib.xml. That is, if there are multiple <publisher> elements which contain the text "Wrox Press", the p variable will only contain one of them, and ignore the rest.
  - For each of the <publisher> elements in the p variable, another variable, called a, will be created, which will contain the average price of all of the books associated with this publisher. It does this by:
    - Getting a node-set of all of the <price> elements, which are a child of a <book> element whose <publisher> element has the same value as the current value of p.
    - Passing this node-set to the avg() function, which will return the average value.

Once the publisher's name, and the average price of its books, have been discovered, an XML fragment similar to

```
<publisher>
<name>Publisher's name</name>
<avgprice>Average price</avgprice>
</publisher>
```

will be returned.

This is very similar to the types of queries we can create using SQL. However, in the XML syntax, this query is much more verbose:

```
<q:query xmlns:q="http://www.w3.org/2001/06/xqueryx">
  <q:flwr>
    <q:forAssignment variable="$p">
      <q:function name="distinct">
        <q:step axis="SLASHSLASH">
          <q:function name="document">
            <q:constant datatype="CHARSTRING">bib.xml</q:constant>
          </q:function>
          <q:identifier>publisher</q:identifier>
        </q:step>
      </q:function>
    </q:forAssignment>
    <q:letAssignment variable="$a">
      <q:function name="avg">
        <q:step axis="CHILD">
          <q:function name="document">
            <q:constant datatype="CHARSTRING">bib.xml</q:constant>
          </q:function>
```

```
<q:step axis="CHILD">
  <q:predicatedExpr>
    <q:identifier>book</q:identifier>
    <q:predicate>
      <q:function name="EQUALS">
        <q:identifier>publisher</q:identifier>
        <q:variable>$p</q:variable>
      </q:function>
    </q:predicate>
  </q:predicatedExpr>
  <q:identifier>price</q:identifier>
</q:step>
</q:step>
</q:function>
</q:letAssignment>
<q:return>
  <q:elementConstructor>
    <q>tagName>
      <q:identifier>publisher</q:identifier>
    </q>tagName>
  <q:elementConstructor>
    <q>tagName>
      <q:identifier>name</q:identifier>
    </q>tagName>
  <q:step axis="CHILD">
    <q:variable>$p</q:variable>
    <q:nodeKindTest kind="TEXT" />
  </q:step>
</q:elementConstructor>
<q:elementConstructor>
  <q>tagName>
    <q:identifier>avgprice</q:identifier>
  </q>tagName>
  <q:variable>$a</q:variable>
</q:elementConstructor>
</q:elementConstructor>
</q:return>
</q:flwr>
</q:query>
```

Phew, that's a lot of text! But keep in mind that the XML syntax isn't primarily meant to be read by humans—that's what the human-readable syntax in the XQuery Working Draft is for. XQueryX is meant to be generated by, and processed by, software, so making the query so explicit in the XML syntax will hopefully make it easier to process by these tools.

A full discussion of XQuery is out of scope for this chapter, but keep an eye on the working drafts listed above for more information.

# Summary

This chapter has introduced you to a few technologies that are still very new?in fact, two aren't even full W3C Recommendations yet!

*The XPointer specification was scheduled to be finished its Last Call period January 29, 2001, but at time of writing was still in Last Call. So by the time you read this, it could very well have reached Recommendation status.*

XLink and XPointer provide some powerful capabilities for linking together XML documents, especially for use in browser-type applications, although there is functionality here that could be used in any XML application that deals with multiple documents.

We've seen:

- How to create simple links in XLink, which mirror the functionality provided by HTML
- How to create complex (extended) links using XLink, even when you don't have access to modify the documents in question
- How default attribute values in a DTD can greatly simplify XML documents that use XLink
- How XPointer can be used to work with only those portions of an XML document that are really needed
- How different types of XML documents can be specified for XPointer using different schemes (when they become available)

We've now covered all of the technologies that we're going to look at in this book.

We started by understanding what well-formed XML is, and learned how XML Namespaces can be used to keep different vocabularies separate, even if they happen to have items with overlapping names. We covered the use of XSLT to transform XML data into other formats, and we learned how DTDs and XML Schemas provide a way to define how an XML document should be structured, and how to specify default values. These technologies are particularly helpful when sharing documents with other developers.

The DOM and SAX were introduced as ways of accessing XML programmatically, and we learned how CSS can be used to apply style to our logically structured data. Other areas we looked at were how XML can be integrated with database applications, in n-tier architectures and how we might link resources using XLink and XPointer.

Let's move on now to reinforce our new found knowledge with practical case studies, which will help solidify our knowledge through looking at some working code.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Case Study 1: Using XSLT to Build Interactive Web Applications

## Overview

This case study is for people who want to roll up their sleeves and try out building a real and practical business solution with XSLT; a simple online address book. If you want to see what can be done with XSLT to build a complete interactive application, then this case study is for you. Once you have that hands-on experience you will be able to move from the sample provided to building an application that is meaningful to your business. To help you do this there are a couple of Try It Out sections that will lead you through modifying the stylesheet.

This chapter will look at the components and architecture of an online address book that is being delivered to the user as a web application. Along with the components we'll look at some of the considerations of the interactive process between the user and the web server, the creation of dynamic content in web pages, and some of the common languages that are used to build web applications. There is also a small but critical component built in Java to consider. The role that Java plays at the application level will be compared to the role it plays at the system level, but don't let that scare you. There is no need to understand or write Java for this application.

The intent of this chapter is not only to show some XSLT in practical use, but also to encourage you to get up to speed actually writing XSLT that provides a useful application. You can simply read the code fragments and descriptions to see how a complete application can be built with XSLT, or you can get actual experience writing XSLT by downloading and installing the application and then modifying the stylesheet yourself. One of the most powerful features of this application is that you can both learn more by changing the application and can also build a completely new application that will serve a useful purpose in your business.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Advantages of Using XSL to Build Interactive Web Applications

The best single reason for using XSL for web applications is that it's easy. It's easy to design a web page, to incorporate end-user application data, and to make complex user interface behavior.

It's almost deceptively easy. When seasoned web application developers look at the XSLT code used in this case study they ask, "What's the big deal?" There really doesn't appear to be anything magical or very complex or challenging about it at all.

It's easy because the XSLT handles all of the conditional processing that is usually done in Java, Visual Basic, or Perl code. It handles all of the data access and filtering usually done by SQL scripts or database connections or special parsing of text files. It's easy to design, because the HTML page presentation layout is echoed in the node layout of the XSLT templates. It's easy to use because XSLT is the same thing as the XML source data so it can manipulate and display the data in one uniform syntax and structure using XPath expressions. It works with completely standard XML and XSLT files without special extensions. There is no compiling so changes made to the XSLT stylesheet can be seen immediately with a simple refresh of the browser.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Possible Business Uses for XSLT-Driven Web Applications

This case study uses XML documents and XSLT stylesheets to create an online address book. This is a read-only application that will allow users to enter the first few characters of a person's last name and see a list of people that match that search criteria. Since many people have the same last name and the spelling may not be known, a list of matches to the search criteria is displayed without showing all the details for each person on that list. The intent is for the user to scan down the list looking for the exact person they want, and then click on one of those names in the list to display that person's address, employer, and occupation.

However, there is no reason this can't be used in the same way to allow your customers to look up products in your inventory. If, for instance, you would like to make an application that will show house listings held by a real estate broker, the very same basic XSLT stylesheet will do that too. Of course, you would have to change the names of the document nodes to fit that purpose along with the actual labels shown on the HTML page, but all the basic lookup functionality could be used from the original address book XSL stylesheet.

Or perhaps you need to build a web application that allows people to search for automobiles for sale at a particular car dealer. Again, this application will do that without much modification. Searches for make, model, year, and price can be swapped for the last name search in the address book application.

How about a travel agency showing vacation packages along with the usual sunny scenes of balmy beaches? This application framework will do that too including all the travel-related images. Perhaps you need to create a catalog of bathroom plumbing fixtures? The possibilities are endless.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Using XML in an XSLT-Driven Application

Let's look at how this application uses XML in the first place. After all, most use of XML has to do with Business-to-Business (B2B) transactions or retrieving relatively small XML files to be displayed as HTML in a web browser or WAP cell phone. In the B2B case, some documents such as purchase orders may have to be parsed or somehow broken up or reformatted to be useful to an order entry system at the destination business of the B2B document transfer. XSLT is the perfect solution for this problem and the XSLT transform will be executed in some kind of sophisticated B2B software that has been custom designed for this purpose. But this isn't what we're doing here with XML.

The other popular use of XML is to store documents in a presentation-neutral way. The feature of that is that the documents or parts of the documents can then be retrieved and made readable by an XSL stylesheet that is customized to the consumer of the information. For instance, if a document contains customer information, one stylesheet might just show the shipping address to the shipping department's browsers. Another stylesheet might just show the customer's payment history to the accounting department. And yet another stylesheet might just show the customers' kids' birthdays on a Wireless Application Protocol (WAP) cell phone used by a sales person on the street. But this also isn't quite what we're doing here with XML.

What we are doing here with XML is using it as a way to encode the user interface input solely for the purpose of being able to control the application by using the XSLT transform engine. What that means is the XML document has to be created on the fly in the web server each time the user submits a request from the browser to the server. Now this may sound difficult, but, in fact, the job gets much easier when you see an XML document as really just a collection of data that lives in memory.

The way that data in memory is organized is controlled by the DOM, which was covered in [Chapter 8](#). It forms the definition of the object that contains the XML document, so what we have now is just a small amount of information in memory that represents an XML document, not a file or printed page. This makes the application fast because it's never parsed or stored on disk or moved across a wire from one business to another, which is important when you realize we're talking about making this document on the fly each time the user submits a request from their web browser. In fact, this document only has to move from the Java code that created it to the XSLT transform engine, and both of those functions are running in the same Java servlet that is the kernel of the whole application. In programming terms, the movement of the document is nothing more than passing a reference to a method. There's one other unique feature to using XML in this manner, which is that the document has a very short lifespan: once the XSLT transform engine consumes it, it is thrown away entirely.

This isn't the whole story about how XML is used in this application, though. As a matter of simplicity, an XML document stored on the server's hard disk is used to store the business data for the application. In the case of the sample application, the business data is 500 names and addresses in an address book. This application uses an XML document in a file as a sort of poor man's database.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Application Business Requirements

This address book serves as a contact list of people who have made donations to election campaigns. Political action and special interest groups can use this application to contact individuals known to support political activities.



To make looking up a name easier, the user can enter just the first few characters of the person's last name. The result of that action is to display a list of people whose last names match the user's search entry. However, that list itself can be quite long which would make for a lot of browser scrolling and the possibility of a lengthy download time. We can fix that possibility by sending just a small number of items on the list to the browser along with some links that will display the next or previous set of items on the list. Along with that paging list, the user needs to see a display of where they are in the list in terms of the page currently being displayed and the total number of people selected in the list. Not only that, the Next and Previous links should not be displayed as active links if there are no more items above or below the page that is currently on the display.

Once the right person is found in the list, the user can click on the name to see the details of the address. At this point both the list and the details for that selected person will be displayed for consistency. This kind of user interface is easier for the user to understand than one where the list is removed entirely from the display when a single person is selected. However, the selected person should be highlighted in the list display to avoid confusion of which person is selected that matches to the address details, and if there isn't any specific person selected, the address details area should either be empty or show a message such as "No item selected". If either a new search is requested or the list is paged up or down, the address details of the last selected person should not be displayed.

A few other nice features are thrown in just to make the user interface easy to use and the underlying XML visible for debugging and learning:

- 

- An error message is displayed if the user does not enter more than one character in the search edit box.

- 

- A message is displayed if the user entered a search that didn't result in finding any matching last names.

- 

- The user can enter a partial name in uppercase or lowercase to get the same results. Unfortunately, there is no easy case conversion in XSLT currently.

-

The last input to the search edit box should be preserved in the display on subsequent views.

- A separate page displays various XML trees in raw tag-delimited text form. The trees include the browser request document, the entire XML source file and the portion of that file that is the results of the search criteria filter.

*The source of this address list is partial campaign contribution data filed by American presidential candidates with the Federal Election Commission, covering the period from January 1 - December 31, 1999. The database can be obtained from <http://www.fec.gov/finance/ftpdet.htm>. To boil down a large table to a manageable size, only the first record of a group of last names with a given frequency were selected (this was done using SQL, not XSLT due to the large size of the database). That's why there are plenty of Allen's but no Zelda's. All the resulting last names are the most common surnames in the US.*

---

[!\[\]\(5f540fc8d334abb49dd9de479f8a5e01\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(b7baf1cfd2f174dedd1a0d4484eff59f\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Separating the Application Server from the End-User Application

In this case study, Java is used to build an application server component, not an application that will be presented to the end user, which will be written entirely in XSLT. By "application server", I mean the methods that will deliver an end-user application that has been written entirely in XSLT. By "component", I mean an installable piece of software that will run inside the Tomcat servlet container, or any other servlet container that provides a compatible execution environment such as JRun from Allaire (<http://www.allaire.com/index.cfm>) or Caucho Resin (<http://www.caucho.com/>). As such, the application server has no previous knowledge of the application being presented to the user any more than a web server has any knowledge about the content of the files it is serving. So the target requirements of this Java servlet is to allow an as yet undefined application written in XSLT to successfully execute using the facilities of a web server and web browsers.

This is a really large and general statement that the servlet that we'll use for this case study does not totally satisfy. There are plenty of types of user interface information that are not handled by this servlet such as HTTP headers. There are also plenty of kinds of XSLT application architecture that are not handled, but one overriding requirement of this application server is to keep it simple. This servlet is not the best ever XSLT application server, because there are things that it won't do, but this servlet is completely adequate for delivering small but useful applications, which can be completely constructed using XSLT without ever needing to modify the Java code to do something specific for a specific end user application. This section is most useful for developers familiar with Java but it isn't necessary to read it if you just want to learn XSL.

## User Interface Connection to the XSLT Transform

To make this application work there must be some method of telling the XSLT transform what the user has done as far as clicking on a link or submitting an HTML form. Not only that, there also needs to be a way of presenting the data the application will work on to the XSLT transform. When searching for a method to address these needs, you will find lots of tutorials and case studies that go to great lengths to build documents with a large amount of Java or ASP or Visual Basic code. Many applications use procedural code that filters or sort the XML by calling functions directly on a DOM instance. That certainly works but it puts part of the application-specific processing in that procedural code. Then the resulting DOM is handed to the XSLT transformer where more application-specific processing is done. This creates a diffused technology where Java produces part of the application process and XSLT produces another part of the application process.

That approach is far too complicated and confusing. If XSLT is going to be used to do the work, then the Java or other procedural language should not be any part of the application-specific details if the intent is to use XSL to build the end-user application.

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Application Server Architecture

Since this application is really a dynamic web application, some of the details of the web server, the Java servlet that forms the kernel of the process and the XSLT transform engine should be understood. Although you really can just install the application and start to work with XSLT immediately, a little background on how the system level components work will give you a better understanding of what the server is doing.

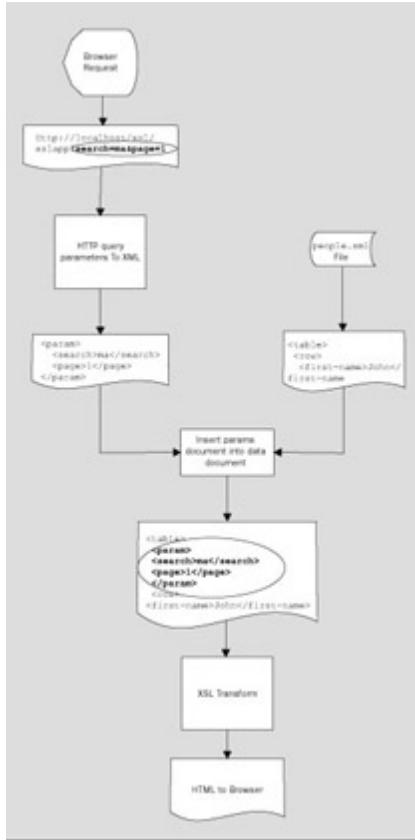
Let's start with the Apache Tomcat web application server. Tomcat is an Apache project (<http://www.apache.org>) whose purpose is the implementation of Sun Microsystems' Java Servlet specification. It is free to download and use in a commercial application. This forms the real platform that connects your computer to a web browser either on your computer or any other computer on a network. Its job is to listen for network requests from a web browser and return an HTML page to the requesting browser. The URL of the request contains the file name of the page along with any HTTP query parameters that are specific to the application. In this case study, the requested page will be xslapp in the xsl directory. As it turns out, there is no file named xslapp but Tomcat knows how to direct a request for that file name to a Java servlet named xslapp.class that will build the HTML page that Tomcat will send back to the browser. Tomcat requires a Java Runtime Environment (JRE) installation on your computer, which you should have installed from our work in earlier chapters (if you haven't, you can find installation instructions in [Chapter 5](#)). Since Tomcat listens for network requests, its job could be foiled if you have a personal firewall running on your computer. Tomcat will start listening at port 8080 but will fail to start if a personal firewall prevents use of that port number.

Some of Tomcat's configurations are controlled by the file TOMCAT\_HOME/webapps/xsl/WEB\_INF/web.xml. This file does not come with Tomcat but is included in the software for this case study, and installed the first time that you run the application. Details of this file will be discussed shortly.

Once the case study application is deployed inside Tomcat by following the installation instructions, the xslapp servlet along with the people.xml document file and the addressbook.xsl template file are put in place in the Tomcat directory structure. All of these files are included in the Web Application Resource file, xsl.war.

## Servlet Overview

This servlet has been developed to run in the Apache Tomcat web server and servlet engine. Java servlets will run in servers other than Tomcat. However, the Web Application Resource (WAR) file that this case study is packaged in requires the Java servlet version 2.3 which may not be supported by all servlet engines. Tomcat is the servlet container that provides the web server functionality of listening at a port for HTTP requests and passes them to a servlet method for the purpose of servicing that request. Tomcat (now becoming Catalina) is an open source project so it is freely available for download and use from <http://jakarta.apache.org>. Its intent is to be a reference implementation of the servlet specification.



The previous diagram gives you a road map to the functionality provided by the Java servlet and Tomcat servlet container. All the documents are DOM instances in memory. They never exist as a document with tags in a file. The document representing the browser request (which isn't really a document but an HTTP request) and the HTML document sent to the browser do exist as byte streams on the network.

The Java part of this project does three things:

- Capture the user's actions and make them available to the XSL transform.
- Open the XML document file that contains the user application data. This file is found at `TOMCAT_HOME/webapps/xsl/people.xml` and contains the name and address information for the address book application. That XML file is parsed into a DOM instance that becomes the source document for the XSL transform engine.
- Run the transform and return the HTML document directly to the browser.

Of these, the first provides the key functionality that will create the document that contains the user input information.

Since this is a web-based application, the user's actions are always represented in a browser request for a specific URL no matter if the address is typed into the location edit box or a link gets clicked or an input form is submitted. The details of that request for a URL are encoded in the HTTP query parameters (in other words, everything after the question mark in the URL). For example, if the URL is:

`http://localhost?search=Smith&page=1&debug=on`

then the query parameters are the `search=Smith&page=1&debug=on` part of the whole URL. This amounts to all the knowledge we have about the user's actions when the mouse and keyboard were used on the web browser, which has to be somehow made available to the XSLT transform. To do this, the Java code builds an XML document that contains a node for every query parameter with each node having the name of the query parameter and the value of

that parameter. So the resulting document in this case would look like:

```
<param>
  <search>Smith</search>
  <page>1</page>
  <debug>on</debug>
</param>
```

The next thing the Java does is physically open an XML file that is used for the application data. In this case it happens to be 500 names and addresses. This file is parsed into a DOM instance and then the document above is inserted into it like this:

```
<table>
  <param>
    <search>Smith</search>
    <page>1</page>
    <debug>on</debug>
  </param>
  <row>
    <l_name>Smith</l_name>
    <f_name>Bruce</f_name>
    <title>Mr.</title>
    <addr1>4422 Bryan Station Road</addr1>
    <addr2 />
    <city>Lexington</city>
    <state>KY</state>
    <zip>40516</zip>
    <employer>"Addington, Inc."</employer>
    <occupation>Coal Executive</occupation>
    <contrib_id>258</contrib_id>
  </row>
</table>
```

Finally, this document is sent to the XSLT transformer and the result returned to the browser as HTML.

The implication of this is that the document node of <param> is completely made up from the hyperlinks and form inputs the application developer puts in the HTML code used in the XSLT templates. There is no other reference to this document information and the Java code has no explicit knowledge of these nodes.

## Servlet Reuse Without Modification

The real benefit is that you can take this Java servlet exactly as it is and use it to build your own custom application with your own XML document file and XSLT stylesheet file. You don't have to know anything about Java or ever touch this servlet to use it for completely different purposes than this sample address book. For instance, let's say you wanted a way for your sales people to look up customers with overdue balances so they know how to handle purchase requests from those customers. All you would need to do is build an XML document of those customers with a schema of your choosing, perhaps containing their names, amount due and last payment amount and date. A separate data export process in your accounting system could rebuild that file daily so the XML document would always have relatively current information. Then you would build a custom XSLT stylesheet that could look up customers by name and show their account details.

More than one instance of this servlet can be used simultaneously with each one serving a different application. All of that is handled in the TOMCAT\_HOME/webapps/xsl/WEB-INF/web.xml file. Each servlet instance and therefore application instance has two nodes in this file. One declares the servlet instance and its properties. The other node sets the URL mapping between the address that is entered in the browser and a particular servlet instance. Here are the nodes for the single address book end-user application.

```
<servlet>
  <!-- the name reference for the servlet container (Tomcat) -->
  <servlet-name>xslapp</servlet-name>

  <!-- the class name referring to the file xslapp.class -->
  <servlet-class>xslapp</servlet-class>

  <!-- set the file name for the XSL stylesheet in TOMCAT_HOME/webapps/xsl -->
  <init-param>
    <param-name>stylesheet</param-name>
    <param-value>addressbook.xsl</param-value>
  </init-param>

  <!-- set the file name for the XML document in TOMCAT_HOME/webapps/xsl -->
  <init-param>
    <param-name>document</param-name>
    <param-value>people.xml</param-value>
  </init-param>

  <!-- display the raw XML before it gets transformed -->
  <init-param>
    <param-name>debug</param-name>
    <param-value>false</param-value> <!-- set to 'true' to display XML -->
  </init-param>
</servlet>
```

Notice that the XSL stylesheet and XML document file names for the address book are declared above. Also there is no need to make the servlet name the same as the class name as it is here.

```
<servlet-mapping>
  <servlet-name>xslapp</servlet-name>
  <url-pattern>/xslapp</url-pattern>
</servlet-mapping>
```

This node sets the file name in the URL and associates it with the servlet name xslapp in the previous <servlet> node. So now the URL is <http://localhost/xsl/xslapp>. Notice that it is not <http://localhost/xsl/xslapp.class>.

Now if you wanted two simultaneous applications that use the same servlet but serve different end-user applications, here is the complete web.xml document to do that:

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

<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
<servlet>
  <!-- the name reference for the servlet container (Tomcat) -->
  <servlet-name>xslapp</servlet-name>

  <!-- the class name referring to the file xslapp.class -->
  <servlet-class>xslapp</servlet-class>

  <!-- set the file name for the XSL stylesheet in TOMCAT_HOME/webapps/xsl -->
  <init-param>
    <param-name>stylesheet</param-name>
    <param-value>addressbook.xsl</param-value>
  </init-param>

  <!-- set the file name for the XML document in TOMCAT_HOME/webapps/xsl -->
  <init-param>
```

```
<param-name>document</param-name>
<param-value>people.xml</param-value>
</init-param>

<!-- display the raw XML before it gets transformed -->
<init-param>
    <param-name>debug</param-name>
    <param-value>false</param-value> <!-- set to 'true' to display XML -->
</init-param>
</servlet>

<servlet>
    <!-- the name reference for the servlet container (Tomcat) -->
    <servlet-name>myapplication</servlet-name>

    <!-- the class name referring to the file xslapp.class -->
    <servlet-class>xslapp</servlet-class>

    <!-- set the file name for the XSL stylesheet in TOMCAT_HOME/webapps/xsl -->
    <init-param>
        <param-name>stylesheet</param-name>
        <param-value>mystylesheet.xsl</param-value>
    </init-param>

    <!-- set the file name for the XML document in TOMCAT_HOME/webapps/xsl -->
    <init-param>
        <param-name>document</param-name>
        <param-value>mydata.xml</param-value>
    </init-param>

    <!-- display the raw XML before it gets transformed -->
    <init-param>
        <param-name>debug</param-name>
        <param-value>false</param-value> <!-- set to 'true' to display XML -->
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>xslapp</servlet-name>
    <url-pattern>/xslapp</url-pattern>
</servlet-mapping>

<servlet-mapping>
    <servlet-name>myapplication</servlet-name>
    <url-pattern>/myapp</url-pattern>
</servlet-mapping>
</web-app>
```

In both the original application and the new application called myapplication, the reference is to the class xslapp. However, the differences are the name of the XSLT and XML files and the URL virtual file requested by the browser which is /myapp for the new application.

This application uses the Saxon XSLT processor developed by Michael Kay which was chosen for its performance and robustness. Michael Kay is the author of *XSLT Programmer's Reference, 2nd Edition*, ISBN 1-861005-06-7, also published by Wrox Press. Another popular XSLT transform engine is Xalan from the Apache project.

# Installing the Application

This application needs three components to successfully install:

- The Java run time environment
- The Tomcat web and servlet server
- 

The application specific files from code download for this book. These files are all compressed into the xsl.war file, which contains all of the following application code files:

a.

web.xml. This file contains configuration data for Tomcat and the servlet.

b.

people.xml. The document file containing names and addresses for 500 people.

c.

addressbook.xsl. The XSL stylesheet file.

d.

xslapp.class. The actual servlet file.

e.

crimson.jar. The XML parser used by the Saxon XSL transform engine.

f.

saxon.jar. The XSLT processor.

g.

xslapp.java. The Java source file for the servlet.

h.

spacer.gif

i.

blueball.gif

We're assuming that if you've got this far in the book you already have a Java run time installation, but then if you're like me you might have flipped to this chapter at the back of the book and not had a need for this component yet. If you don't have Java runtime installed already, you can find instructions on how install it in [Chapter 5](#).

*Please note: although this application has been installed repeatedly on Windows 2000 Server and Professional and Solaris 8, it has not been tested on other versions of Windows or other flavors of UNIX.*

Follow these steps to get the application installed and running.

1.

Download Tomcat version 4.x from <http://jakarta.apache.org/site/binindex.html>. Look for the 4.x version in the "Release Builds" section. The exact current release version changes periodically. I am using jakarta-tomcat-4.0.zip for Windows. Don't use version 3.x as it will only work with a parser version that is different from the parser version the XSLT engine requires.

2.

Install Tomcat either with the setup program or by unpacking the files into a convenient directory such as C:\java\tomcat. I suggest avoiding a directory path that includes imbedded spaces such as C:\Program Files\... as whitespace can be interpreted as the end of a file specifying string in Java or UNIX. Set an environment variable that tells Tomcat where your Java run time installation resides such as JAVA\_HOME=c:\java\java1\_3. There is more information on setting environment variables in [Chapter 5](#).

3.

Configure the web server listening port if you already have a web server running. If you don't, ignore this item. When Tomcat is started it will use port 8080 by default. That is the standard port for all Java Web Servers and Tomcat so if you already have a server running that listens at that port you will have to stop it first. You can configure the port that the Tomcat server listens on in TOMCAT\_HOME/conf/server.xml, where TOMCAT\_HOME is the name of the directory that you installed Tomcat on. Search for the string 8080 to find the element.

*This application starts on port 8080, which is not the standard port used by web browsers. This was done intentionally just in case you already have a web server running on your machine. In the case of office firewall servers that try to thwart electronic break-ins, the port 8080 may not be allowed to communicate with the public internet. The standard port that browsers use and is always open through firewalls is port 80. You can make this case study application work directly on port 80 so it will be seen through all firewalls by editing the configuration file as described above. The details about this configuration will be discussed later. Also, if you have a personal firewall running on the machine you want to install this application on, you must configure it to allow the application server to run by opening port 8080.*

4.

Drop the xsl.war file from the code download for this book into Tomcat's web applications directory at TOMCAT\_HOME/webapps. Note that the TOMCAT\_HOME/webapps/xsl directory does not exist when you first install Tomcat but will be created from the xsl.war file when Tomcat is first started. That means you can't change anything in the application until after the first time it runs.

5.

Start the server with startup.bat on Windows or startup.sh on UNIX in TOMCAT\_HOME/bin from that directory, not from your hard drive root directory or your shell home or other directory. To start the server in Windows, open a DOS or command window that will show the standard prompt C:\. Change directory to TOMCAT\_HOME\bin. Then type startup. If you have installed Tomcat using the installation program, you should also be able to start Tomcat with the Start Tomcat option in the Start menu on Windows machines.

6.

Point your browser at <http://localhost:8080/xsl/xslapp> to start the application. Expect a delay when the application first starts as it has to decompress the files from the xsl.war file. This URL assumes you have the default installation on the computer you are running the browser on.

# Address Book Stylesheet Design

This stylesheet has to contain all the code that controls the user interaction, accesses the application data, and formats the HTML page presentation. Not only that, it also has to be written in a way that is easy to write and understand and, of course, comply with the XSLT language syntax and structure requirements.

The tree diagram below will give you a better feeling of the overall naming and functioning of the major nodes in the application. This diagram doesn't show every little detail of what goes into the final HTML page but will give you an overall road map. What's even more important is to get the notion that an application's complete user interface can be modeled as a tree. Some of the more interesting features of this model are that there is more than one page in this application and error- or data-dependent conditions are tested and responded to right in the node that displays the results. Every time the transform is run, all the nodes of this tree are generated from it with the exception of the nodes that are siblings shown with a dotted line. Only one of those siblings will be generated, depending on what the user requested.

This application also covers the case for which more than one page can be generated from a single stylesheet and single source document. I've noted this in the diagram by the dotted line that chooses the child node of the display root. Either the user page with its form and list will be displayed or the XML debugging page with its display category links will be displayed. There is also the possibility of a condition where no people are found matching the user's last name criteria or where no specific person has been selected in order to display the address details. Those conditions are also noted as nodes connected with dotted lines. One conditional node shows the No records found message or the scrolling list of people that match the search criteria. The other conditional node either shows the No item selected message or the address details display node with all its labels and formatting.

Not all the details of all the nodes and code in the stylesheet are included in this tree diagram. To reduce the clutter, only the larger nodes with their functional descriptions are mentioned. The intent is to form a road map so you can know what part of the stylesheet is related to what other part and to the entire user interface.



# User Interface Components

The page layout is broken up into smaller user interface components. Those functions would naturally follow the application requirements for the user interface and are graphically related in the diagram above.

- **Search criteria input form.** This allows the user to enter a few leading characters of a person's last name and submit that to start a search.
  - **Search results list.** One of the considerations about this is what to do if there are a large number of search results. Should they all be displayed on one page and let that page potentially become very long? Or should there be a user interface technique to allow paging through that long list? I chose the latter because it will

nicely fit any number of search results items into a single browser-sized page without scrolling. I also chose that type of user interface because it's in wide use and forms a good opportunity for using XSLT to build some non-trivial logic.

- **Details display** for a single selected person from the search list. This is really the easiest to do because there is no paging, it only shows one person at a time and there is no editing allowed because this is a read-only application.
- **XML debugging page** for easy viewing of the XML source document. This page is the fun part because it doesn't need scrolling one page at a time or fancy layout. It just needs to select which part of the transform's source document to display and lay out the raw XML, tags and all. It also has a very small demonstration of JavaScript on it.

Finally, these functional components need to be laid out in some kind of coherent presentation using an HTML table. In this application I don't attempt to make one component disappear to show another. For instance, when a single person is selected from the list, the list component isn't removed from the page. The challenge is to keep all three or four of these functional areas going at the same time without resorting to frames or Java applets that run in the browser. It also produces a browser-independent application.

The fourth user interface component, the XML debugging page, is shown below.



There is one XSLT template for each of the four components in the breakdown listed above. These templates are inserted into one HTML table that serves to lay out the whole page. Here is an outline of the HTML for the page:

```
<html>
<head>
    <title>XSL Application</title>
    <style type="text/css">
        td { font-family: Arial, Helvetica, sans-serif; font-size: 10pt }
        #selected { background-color: #CCCCCC; font-weight: bold }
    </style>
</head>
<body vlink="blue">
    <center><b>XSLT On-Line Address Book</b></center>
    <!-- **** user page layout table ***** -->
    <table border="1" height="300">
        <tr>
            <td width="50%" valign="top">
                <!-- ***** form and list display ***** -->
                <xsl:call-template name="form"/>
                <xsl:call-template name="list"/>
            </td>
```

```
<td width="50%" valign="top">
<br/><br/>
<!-- ***** details display ***** --&gt;
&lt;xsl:call-template name="detail"/&gt;
&lt;/td&gt;
&lt;/tr&gt;
&lt;/table&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>
```

The thing to notice here is the use of the `<xsl:call-template>` elements used in the body of the `<td>` element. In this stylesheet they are used to import the body of some other named templates. By setting it up this way, we can isolate each one of the user interface components into their own named templates. This technique makes the work easier because we can focus on just one component at a time and not have to see any code surrounding it. Coding is also easier because you can drop the content of a template right into another layout structure with a single tag, which is how the `<xsl:call-template>` element is used inside the HTML table above. You can read more details about how named templates and `<xsl:call-template/>` works in [Chapter 4](#).

That whole section of HTML also has to be put in a template somewhere. That ends up being the single rules-based template that matches the source document root: `<xsl:template match="/">`. This "match" template is different from the named templates that will be called from the `<xsl:call-template/>` elements. The "match" template is guaranteed to execute because it matches the document root and overrides the default template. The default template is the one that passes everything (except comments) through from the source document to the HTML output page. For brevity, I won't show the HTML within the `<xsl:template match="/">` element right now but it will all come together shortly.

Besides the user interface components, coding the application is made easier by using a few top-level variables. This XSLT element allows you to assign a name to a tree value or node-set value for later use. In this application the variables that are most useful represent the two parts of the document the Java servlet built. Remember that there was a small node that encapsulated the browser request parameters and another node-set that had the 500 people's names and addresses. Here is a review of a small part of that input document:

```
<table>
<!-- ***** HTTP query parameters ***** --&gt;
&lt;param&gt;
&lt;search&gt;Sm&lt;/search&gt;
&lt;page&gt;1&lt;/page&gt;
&lt;debug&gt;on&lt;/debug&gt;
&lt;/param&gt;
<!-- unfiltered source document of 500 row nodes --&gt;
&lt;row&gt;
&lt;l_name&gt;Smith&lt;/l_name&gt;
&lt;f_name&gt;Bruce&lt;/f_name&gt;
&lt;title&gt;Mr.&lt;/title&gt;
&lt;addr1&gt;4422 Bryan Station Road&lt;/addr1&gt;
&lt;addr2 /&gt;
&lt;city&gt;Lexington&lt;/city&gt;
&lt;state&gt;KY&lt;/state&gt;
&lt;zip&gt;40516&lt;/zip&gt;
&lt;employer&gt;"Addington, Inc."&lt;/employer&gt;
&lt;occupation&gt;Coal Executive&lt;/occupation&gt;
&lt;contrib_id&gt;258&lt;/contrib_id&gt;
&lt;/row&gt;
...
&lt;/table&gt;</pre>
```

## The \$params Variable

The HTTP query parameters node is captured in an XSLT variable by:

```
<xsl:variable name="param" select="/table/param"/>
```

That means any time we write \$param in our code we get:

```
<search>Sm</search>
<page>1</page>
<debug>on</debug>
```

The \$params variable is very useful when we get to actually controlling the user interface behavior by using the information in the HTTP request sent from the browser. As an example of using this parameter, to just get the string value that the user entered in the search criteria field would take:

```
<xsl:value-of select="$param/search"/>
```

## The \$searchlist Variable

This useful variable will hold the node-set of all the people that match the search criteria. This variable is a bit trickier because it contains the XPath expression that does the actual filter of people whose last name meets the user's search criteria. This XPath filter expression forms the actual crux of the most basic functionality of this application, which allows the user to select only a small part of a large document for display. Actually, that is a two step process on the part of the user and the application. The first step is to create a list (a node-set) of people whose last names match the user's search criteria. Then the second step is to filter that node-set down to a single <row> node that contains a specific person's name and address. The \$searchlist variable is only concerned with the first step of filtering. Here is a short sample of the node-set that is produced if the search input is "Sm". It is what you get when you reference \$searchlist later on in the code.

```
<row>
  <contrib_id>37872</contrib_id>
  <l_name>Small</l_name>
  <f_name>Aaron M.</f_name>
  <addr1>657 N Lake Drive</addr1>
  <city>Lakewood</city>
  <state>NJ</state>
  <zip>08701</zip>
  <employer>Info Requested</employer>
  <occupation>Info Requested</occupation>
</row>
<row>
  <contrib_id>37904</contrib_id>
  <l_name>Smith</l_name>
  <f_name>A. J. C.</f_name>
  <addr1>630 Park Avenue</addr1>
  <city>New York</city>
  <state>NY</state>
  <zip>10021</zip>
  <employer>Marsh & McClellan Co.</employer>
  <occupation>Chairman</occupation>
</row>
```

There is an excellent discussion about XPath filter expressions in [Chapter 4](#). The details of how a filter expression works by matching a literal string to the string value of a node is worth reading again if you're not familiar with it. But filtering by matching a literal string to a node's value isn't quite what we need to do in this application. There are two additional problems to take care of. The first is that we want to get the nodes that have a string value that starts with the user's search criteria rather than an exact match to the search criteria. The XPath starts-with() function will do that kind of a match. Here is the XSL node that does this trick:

```
<xsl:variable name="searchlist" select="/table/row[starts-with(l_name, 'Sm')]"/>
```

But that element will only work for the hard coded "Sm" search criteria. What we really want is the user's search criteria that can be found in the \$params variable, so now the variable that will return the search results for the user's criteria is:

```
<xsl:variable name="searchlist" select="/table/row[starts-with(l_name, $param/search)]"/>
```

What's left to take care of is the case sensitivity of the filter expression including the starts-with() function. There is no case conversion function in the present XSLT 1.0 version so a character translation function will be used to take care of the problem. All that's needed is to apply a character translation to both parameters in the starts-with() function such that each lowercase character in the alphabet is converted to the corresponding uppercase character. The variable now looks like:

```
<xsl:variable name="searchlist"
  select="/table/row[starts-with(
    translate(l_name,
      'abcdefghijklmnopqrstuvwxyz', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'),
    translate($param/search, 'abcdefghijklmnopqrstuvwxyz',
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ'))]"/>
```

This does end up with a rather long-winded looking expression but it shows how versatile and functional XPath expressions can be. This variable is now the real workhorse of selecting names from all the nodes in the document. It is equivalent to the SQL statement:

```
select * from table where l_name like 'myVariable%'
```

## The \$app Variable

This variable simply holds a string value for the servlet name that will be a part of every link and HTML form action attribute. Again, by keeping this simple information in a variable, it's easy to change in just one place rather than everywhere in the style sheet where the file name that Tomcat uses to refer to the servlet appears. This is especially handy when you want to map the servlet to a different URL, which would be done in TOMCAT\_HOME/webapps/xsl/WEB-INF/web.xml. As a practical matter, this technique gives you the possibility of using a different XML document and XSL stylesheet but still using the same servlet mapping for each set of XML and XSL files. The top-level element that controls this is:

```
<xsl:variable name="app" select="'xslapp'"/>
```

To use this in a hypertext link it would be written:

```
<a href="{{$app}}?debug="" target="new">XML</a>
```

*The use of the curly braces above allows the inclusion of an XPath expression. If you need to review the details of using curly braces, they can be found in [Chapter 4](#).*

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Putting the Layout and Variables Together

Now that we've seen the overall HTML layout, the use of named templates (although the actual templates haven't been discussed yet), and the three top level variables, those pieces can all be assembled into the stylesheet along with the logic that selects whether to show the user page or the XML debugging page.

```

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

<!-- **** variable param for all user input **** -->
<xsl:variable name="param" select="/table/param"/>

<!-- *** variable app identify the servlet name *** -->
<xsl:variable name="app" select="'xslapp'"/>

<!-- ** variable searchlist is results of filter ** -->
<xsl:variable name="searchlist"
  select="/table/row[starts-with(
    translate(l_name, 'abcdefghijklmnopqrstuvwxyz', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'),
    translate($param/search, 'abcdefghijklmnopqrstuvwxyz',
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ'))]"/>

<!-- ***** the main output template *****-->
<xsl:template match="/">
  <html>
    <head>
      <title>XSL Application</title>
      <style type="text/css">
        td { font-family: Arial, Helvetica, sans-serif; font-size: 10pt;};
        #selected { background-color: #CCCCCC; font-weight: bold }
      </style>
    </head>
    <body vlink="blue">
      <center><b>XSLT On-Line Address Book</b></center>
      <xsl:choose>
        <!-- **** show XML for debugging ***** -->
        <xsl:when test="$param/debug">
          <xsl:call-template name="debug"/>
        </xsl:when>
        <xsl:otherwise>
          <!-- **** user page layout table ***** -->
          <table border="1" height="300">
            <tr>
              <!-- ** form and list display ** -->
              <td width="50%" valign="top">
                <xsl:call-template name="form"/>
                <xsl:call-template name="list"/>
              </td>
              <!-- ***** details display ***** -->
              <td width="50%" valign="top">
                <br/><br/>
                <xsl:call-template name="detail"/>
              </td>
            </tr>
          </table>
        </xsl:otherwise>
      </xsl:choose>
    </body>
  </html>
</xsl:template>

```

```
</xsl:otherwise>
</xsl:choose>
</body>
</html>
</xsl:template>

<!-- 4 named templates are inserted here. See details below -->

</xsl:stylesheet>
```

This is the complete stylesheet: top-level variables along with the complete match template. Notice that an `<xsl:choose>` element has been used to select either the debug template or the contents of the user page with its HTML layout table. Also, notice that the HTML `<head>` element along with "XSLT On-Line Address Book" display is outside of the `<xsl:choose>` element. That forms the basic application level templating technique that will display the heading on both the user and debugging pages.

## Debug Page Selection Details

When the user clicks on the XML link on the user page, none of its layout or components will be displayed at all. Instead, some fairly raw formatted XML including the `<param>` node and the node-set of the search results will be displayed without any fancy HTML tables. This introduces the first conditional display that will be facilitated by the `param/debug` node that was created by the servlet when the browser request contained the debug parameter like this:

`http://localhost:8080/xsl/xslapp?debug=search`.

The decision of which page to display is done by testing the `param/debug` node to see if it is present using the `<xsl:when test="$param/debug">` element. Therefore, the purpose of this whole `<xsl:choose>` node is to completely abandon the user page with its HTML table for the crude output of the raw XML for the debugging page. Going back to the XSLT basics, remember that the `<xsl:otherwise>` element will output whatever is enclosed in it if none of the `<xsl:when>` test conditions have been satisfied. Both the `<xsl:when>` and the `<xsl:otherwise>` elements must be enclosed in the `<xsl:choose>` element which allows you to select one of a number of possible test conditions to be included in the template's output. You may want to review the `<xsl:choose>` element, which was covered in [Chapter 4](#).

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Detailed Named Template Descriptions

Now that the top-level template and parameters have been discussed, the named templates that actually do the work of displaying the page can be considered in detail. These are the form, list, detail, and debug named templates.

## Form Template

This template (which generates the search input field, Find button, and XML link) is placed in the main output by the `<xsl:call-template name="form"/>` element, which you can see in the match template code above. It's the form template's responsibility to display the user input form for getting the search criteria. A couple of other functions it provides are an error message if the user doesn't enter any characters and a simple link to display the XML for debugging. An added detail for user convenience is returning the form from the last request with the last search value already present in the edit box.

Let's go on to consider each of the sub-components of the form template in turn.

### Error Message Node

If we start at the top of the template we begin with the error message for a situation where user input is missing:

```
<xsl:if test="string-length($param/search) < 1 and $param/search">
  <tr bgcolor="lightblue">
    <td align="center" colspan="4">
      <font color="red">At least one character required!</font>
    </td>
  </tr>
</xsl:if>
```

This `<xsl:if>` element test uses the `string-length()` function to determine if the user has typed in at least one character. However, the `and` operator is also included to find out if there is a `<search>` element at all in the `<param>` node. The reason for this is that without the test for the presence of `$param/search`, the error message would be shown as if there were no `<search>` element at all, which is the case when the user first enters the address in the location box without mentioning any query parameters at all. In other words, clicking the Find button without entering at least one character in the search box is considered an error condition but just entering the URL of the application without any parameters is not.

### User Entry Form Using HTML `<form>` Element

The next sub-component of the form template displays the actual user entry form along with a link that launches a new browser window to display the actual XML document that is getting transformed.

### Stateless Web Application

First there is one design detail to be noted. This is a stateless application. The HTTP parameters are not kept on the server between requests so every parameter that needs to be remembered from one request to the next must be a part of a hyperlink or an HTML form input element. That's to say that the `<param>` node of the source document will disappear completely if there are no HTTP parameters in the browser request. As a case in point, the XML debugging link in the user page form area actually may have a URL that looks like:

`http://localhost:8080/xsl/xslapp?page=10&search=m&debug=`

just for the purpose of opening a new browser window. That means the current known values for the page and search parameters are copied into this hyperlink. The XSLT code that does this is:

```
<a href="{{$app}}?page={{$param/page}}&search={{$param/search}}&debug=" target="new">  
XML  
</a>
```

The overall result is that the values of the page and search parameters that were set by previous browser requests are maintained by receiving them in one request and then sending them back out to the browser in the response page. The persistence of these parameters is caused by embedding them on the page being sent to the browser by including them in the hyperlinks. The parameters can also be preserved from one request to the next by putting them in input fields in an HTML <form> element.

```
<tr bgcolor="lightblue">  
  <form action="{{$app}}">  
    <td>Name </td>  
    <td>  
      <!-- show the search string in the value attribute -->  
      <input type="text" name="search" size="3" value="{{$param/search}}"/></td>  
    <td>  
      <input type="submit" value="Find"/>  
  
      <!-- page is always set to 0 when a new search is requested -->  
      <input type="hidden" name="page" value="0"/>  
    </td>  
    <td>  
      <!-- ***** show debugging link ***** -->  
      <a href="{{$app}}?page={{$param/page}}&search={{$param/search}}&debug=" target="new">XML</a>  
    </td>  
  </form>  
</tr>
```

Here again the attribute template between the curly braces is used to fill in the form's action attribute and the value to be shown pre-filled in for the search input box.

## List Statistics Display

The last sub-component is the list statistics display showing the total number of items found in the search along with the item numbers for the current page from the list:

```
<xsl:variable name="nextOrMax">  
  <xsl:choose>  
    <xsl:when test="$param/page + 10 < count($searchlist)">  
      <xsl:value-of select="$param/page + 10"/>  
    </xsl:when>  
    <xsl:otherwise>  
      <xsl:value-of select="count($searchlist)"/>  
    </xsl:otherwise>  
  </xsl:choose>  
</xsl:variable>  
  
<xsl:if test="$param/search and string-length($param/search) > 0">  
  Items <xsl:value-of select="$param/page + 1"/> - <xsl:value-of select="$nextOrMax"/> of  
<xsl:value-of select="count($searchlist)"/>  
</xsl:if>
```

This complex looking node starts out by defining a variable named \$nextOrMax that will hold either the number of the bottom item shown or the current value of the page parameter: \$param/page + 10. This test is against the count() function applied to the \$searchlist variable. Then, in the <xsl:if> element, we test to make sure that the user has entered search criteria into the input field, and if so, that the value of the parameter \$nextOrMax is actually displayed with the <xsl:value-of> element.

## Complete Form Template

Here is the entire template with its sub-components along with some HTML tables for layout.

```
<xsl:template name="form">
<table width="100%" cellpadding="0" cellspacing="0">

    <!-- error message if search string is empty -->
    <xsl:if test="string-length($param/search) < 1 and $param/search">
        <tr bgcolor="lightblue">
            <td align="center" colspan="4">
                <font color="red">At least one character required!</font>
            </td></tr>
    </xsl:if>

    <!-- ***** user input form ***** -->
    <tr bgcolor="lightblue">
        <td><form action="{{$app}}">
            <td>Name </td>
            <td>
                <input type="text" name="search" size="3" value="{{$param/search}}"/>
            </td>
            <td>
                <input type="submit" value="Find"/>
            <!-- **** reset the page parameter to 0 **** -->
            <input type="hidden" name="page" value="0"/>
        </td>
        <td>
            <!-- ***** show debugging link ***** -->
            <a href="{{$app}}?page={{$param/page}}&search={{$param/search}}&debug="target="new">XML</a>
        </td>
        </form>
    </tr><tr><td colspan="4" align="center">

        <!-- ***** display list statistics ***** -->
        <xsl:variable name="nextOrMax">
            <xsl:choose>
                <xsl:when test="$param/page + 10 < count($searchlist)">
                    <xsl:value-of select="$param/page + 10"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="count($searchlist)"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>
        <xsl:if test="$param/search and string-length($param/search) > 0">
            Items <xsl:value-of select="$param/page +1"/> - <xsl:value-of select="$nextOrMax"/> of <xsl:value-of select="count($searchlist)"/>
        </xsl:if>
    </td></tr></table>
</xsl:template>
```

## List Template

The next template shows the list of the search results as a page-at-a-time list and is probably the most complex of all the templates. It must know what page it is on in the list and whether to show the Next Items or Previous Items as active links or grayed out inactive links. It also has to make each item a link with a parameter keyed to that item to show the address details and then it has to visually highlight a person's name when that link is clicked. Like the form template, the list template is also broken into a number of small nodes that each serve a specific function and result in a specific part of the list's display. This technique of laying out the nodes in the template along with all their logic is what makes developing complex user interfaces an easy task using XSLT.

## Suppress List and Empty Results Message

Here is the <xsl:choose> element that decides if there even is a need to show the list. It responds to the possibility that the user's selection criteria resulted in no items to display or the possibility that there was no valid user input:

```
<xsl:variable name="total_rows" select="count($searchlist) "/>

<xsl:choose>
  <xsl:when test="$searchlist and string-length($param/search) > 0">
    <table cellspacing="0">
      <!-- list display nodes. See detail below -->
    </table>
  </xsl:when>
  <xsl:otherwise>
    <center><b>No records found!</b></center>
  </xsl:otherwise>
</xsl:choose>
```

A feature of this node is that it does not even output the HTML <table> element in the page if it would end up being empty because some browsers will not display properly if a <table> element does not have the proper children <tr> and <td> elements.

## Next and Previous Links

The Next 10 items link is constructed in the following way:

```
<tr bgcolor="lightblue">
  <td align="center" colspan="2">
    <xsl:choose>
      <xsl:when test="$total_rows>$param/page + 10">
        <a href="{$app}?page={$param/page + 10}&search={$param/search}">
          Next 10 items</a>
      </xsl:when>
      <xsl:otherwise>
        <font color="gray">Next 10 items</font>
      </xsl:otherwise>
    </xsl:choose>
  </td>
</tr>
```

We test to see whether the total number of rows in the results returned by the query is greater than the value of the page parameter plus ten (the page parameter is set to 0 when the user first clicks on the Find button). If it is, then we know that we have more rows to display than can fit on one page, so we generate a link to the next page of results by adding 10 to the value of the page parameter. If not, then we generate a grayed-out, inactive link.

The Previous 10 items link is constructed similarly:

```
<tr bgcolor="lightblue"><td align="center" colspan="2">
  <xsl:choose>
    <xsl:when test="$param/page > 9">
```

```
<a href="{{$app}?page={$param/page - 10}&search={$param/search}">
    Previous 10 items</a>
</xsl:when>
<xsl:otherwise>
    <font color="gray">Previous 10 items</font>
</xsl:otherwise>
</xsl:choose>
</td></tr>
```

Here, we test to see if the page parameter is greater than nine. If it is, we know that there must be a page of information prior to the page we're on currently, so we generate a link to it, by subtracting ten from the page parameter. If not, then we generate a grayed out, inactive link.

## Row Iteration

The next node is task row iteration; it does the real work of looping through the rows for the current list page displaying the person's last and first names, identifying each row with a unique identifier and controlling the color to present on each row.

```
<xsl:for-each select="$searchlist[position() > $param/page and position() < $param/page + 11]">
    <!-- ***** details described below *** -->
</xsl:for-each>
```

This entire node is expanded to become a set of nodes (viewed as rows in the list) by the use of the `<xsl:for-each>` element. But rather than look through the entire source document, only the nodes in the temporary node-set in the `$searchlist` variable will be considered. That was the variable that was instantiated at the top of the stylesheet and used a filter expression including the user's search criteria as set in the search parameter. The `select` attribute in the `<xsl:for-each>` element includes a filter expression that will only select those nodes whose position in the `$searchlist` is both greater than the current value of the `page` parameter and less than the `page` parameter + 11. So this is the exact mechanism that creates the paging of the list and is expressed in a single although somewhat congested `<xsl:for-each>` `select` attribute.

## Selected Item Highlighting

A highlight is applied to the currently selected row by applying a CSS style to the `<td>` elements in that row. To do that, a couple of variables are set up inside the row iterating template. The first variable (`isSelectedItem`) is a Boolean-valued variable that indicates if the current row is the one the user selected. The second variable (`style`) is a string-valued variable that contains the name of the CSS definition for the visual highlighting.

```
<!-- ***** Row iterator ***** -->
<xsl:for-each select="$searchlist[position() > $param/page and position() < $param/page + 11]">

    <!-- ** set this value for each row iteration *** -->
    <xsl:variable name="isSelectedItem" select="contrib_id = $param/id"/>

    <!-- **for use on id attribute for <td> below *** -->
    <xsl:variable name="style">
        <xsl:if test="$isSelectedItem">selected</xsl:if>
    </xsl:variable>
    ...
</xsl:for-each>
```

Then setting the style of the `<td>` elements that make up the last name and first name columns does the actual implementation of the visual highlighting. By controlling the `<td>` element's `id` attribute, the appropriate style for the selected person is set from the CSS style definition. Here is one `<td>` node:

```
<tr>
  <!-- ***** last name column ***** -->
  <td width="100" id="{{$style}}>
...
</td>
...
</tr>
```

## Item Row Display

Having taken care of the selected item's color, what remains is the actual text to display in the row along with the necessary link's query parameters. This is done using both the attribute value template for the href attribute and also the `<xsl:value-of>` element to put the actual text value of the person's last and first names into the table data cell `<td>` element.

```
<tr>
  <!-- ***** last name column ***** -->
  <td width="100" id="{{$style}}>
    <xsl:if test="$selectedItem">
      <xsl:text> </xsl:text>
    </xsl:if>
    <xsl:if test="not($selectedItem)">
      <xsl:text> </xsl:text>
    </xsl:if>
    <a href="{{$app}}?id={{contrib_id}}&search={{$param/search}}&page={{$param/page}}>
      <xsl:value-of select="l_name"/></a>

    <!-- ***** first name column ***** -->
    <td width="100" id="{{$style}}>
      <xsl:value-of select="f_name"/>
    </td></tr>
```

Since this is also enclosed in the `<xsl:for-each>` elements, all the abbreviated path references are in the context of the `<xsl:for-each>` so they don't need a full path description. One more detail is the `<xsl:text>` elements following the blue ball image, which is there just to place a space between the name and the ball.

## Complete List Template

Putting all the nodes together for the list produces this template:

```
<xsl:template name="list">
  <xsl:variable name="total_rows" select="count($searchlist)"/>

  <xsl:choose>
    <xsl:when test="$searchlist and string-length($param/search) > 0">
    <!-- don't allow empty <table>. Netscape fails -->

      <table cellspacing="0" border="1">

        <!-- ***** Scroll previous page ***** -->
        <tr bgcolor="lightblue">
          <td align="center" colspan="2">
            <xsl:choose>
              <xsl:when test="$param/page > 9">
                <a href="{{$app}}?page={{$param/page - 10}}&search={{$param/search}}>
                  Previous 10 items</a>
              </xsl:when>
              <xsl:otherwise>
                <font color="gray">Previous 10 items</font>
              </xsl:otherwise>
            </xsl:choose>
          </td>
        </tr>
        <tr>
          <td>
```

```
</xsl:choose>
</td></tr>

<!-- ***** Row iterator ***** -->
<xsl:for-each select="$searchlist[position() > $param/page and position() < $param/page + 11]">

<!-- ** set this value for each row iteration *** -->
<xsl:variable name="isItemSelected" select="contrib_id = $param/id"/>

<!-- ** use on id attribute for <td> ***** -->
<xsl:variable name="style">
    <xsl:if test="$isItemSelected">selected</xsl:if>
</xsl:variable>

<tr>
    <!-- ***** last name column ***** -->
    <td width="100" id="{{$style}}>
        <xsl:if test="$isItemSelected">
            <xsl:text> </xsl:text>
        </xsl:if>
        <xsl:if test="not($isItemSelected)">
            <xsl:text> </xsl:text>
        </xsl:if>
    <a href="{{$app}}?id={{contrib_id}}&search={{$param/search}}&page={{$param/page}}>
        <xsl:value-of select="l_name"/></a>
    </td>

    <!-- *** first name column *** -->
    <td width="100" id="{{$style}}>
        <xsl:value-of select="f_name"/>
    </td>

    <td width="100" id="{{$style}}> <!-- *** width of first name column *** -->
        <xsl:value-of select="occupation"/>
    </td>
</tr>
</xsl:for-each>
<!-- ***** Scroll next page ***** -->
<tr bgcolor="lightblue"><td align="center" colspan="2">
    <xsl:choose>
        <xsl:when test="$total_rows>$param/page + 10">
            <a href="{{$app}}?page={{$param/page + 10}}&search={{$param/search}}>
                Next 10 items</a>
        </xsl:when>
        <xsl:otherwise>
            <font color="gray">Next 10 items</font>
        </xsl:otherwise>
    </xsl:choose>
</td></tr>

</table>
</xsl:when>
<xsl:otherwise><center><b>No records found!</b></center>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
```

## Try It Out - Adding a Field to the List

This is a good point to get hands-on experience of modifying the stylesheet. Here is a possible scenario for a change in requirements: There obviously is information about each person's occupation but it can't be seen until that person is

selected. However, the user may want to scan down the list looking for particular occupations attempting to target affluent individuals; corporation presidents or CEOs might fit the bill!

To do this, you need to add a column to the HTML table, show the occupation from the node-set held in the \$searchlist variable and adjust the colspan attribute for the Next Items and Previous Items links.

1.

Open the XSL stylesheet file in any text editor. The file location is  
TOMCAT\_HOME/webapps/xsl/addressbook.xsl

2.

Find the list display template. You can search for <xsl:template name="list">. In that template look for the <td> node that displays the person's first name:

```
<!-- *** first name column *** -->
<td width="100" id="{{$style}}>
  <xsl:value-of select="f_name"/>
</td>
```

3.

Add the following code directly under that <td> node:

```
<td width="100" id="{{$style}}>
  <xsl:value-of select="occupation"/>
</td>
```

4.

Adjust the width of the Next and Previous list scrolling links from 2 columns to 3 columns. They are both in a table data element that looks like:

```
<td align="center" colspan="2">
```

Change the colspan attribute from 2 to 3.

5.

Save the file and refresh your browser.

## Try It Out - Sorting the List

This is a more challenging modification to the stylesheet than adding a field to the display list. Suppose that our users liked being able to see occupation details but still didn't like the fact that they have to page through the entire list looking for all the corporation presidents or CEOs. The users wanted all the people with those occupations listed together. This new requirement can be met by sorting the node-set in the \$searchlist parameter. The <xsl:sort> element would do this nicely except that it will only work as the first child of a <xsl:for-each> or a <xsl:template match="..."> element.

Since we don't use any match templates that option is out. However, we do use an <xsl:for-each> element to iterate through the rows in the list. It looked very attractive to just put an <xsl:sort select="occupation"/> element in the <xsl:for-each> node but the results from doing that aren't particularly useful, because it only sorts the current list page being displayed.

To get around this problem, we need to make an entirely new variable. Its nodes can be instantiated from a new <xsl:for-each> element and put the <xsl:sort select="occupation"/> element inside of that. The new variable will have the same contents as the \$searchlist variable but now sorted by occupation. Then that new variable can be referenced by the list iteration <xsl:for-each> node. Here is how to do it:

1.

Open the XSL stylesheet file in any text editor. The file location is  
TOMCAT\_HOME/webapps/xsl/addressbook.xsl

2.

Find the list display template. You can search for <xsl:template name="list">

3.

Add the following code directly under that template tag:

```
<xsl:variable name="sorted-list">
<xsl:for-each select="$searchlist">
  <xsl:sort select="occupation"/>
  <xsl:copy-of select="current()" />
</xsl:for-each>
</xsl:variable>
```

4.

Now find this tag:

```
<xsl:for-each select="$searchlist[position() > $param/page and position() < $param/page + 11]">
```

5.

Replace the \$searchlist variable reference with \$sorted-list/row so the element looks like this:

```
<xsl:for-each select="$sorted-list/row[position() > $param/page and position() < $param/page + 11]">
```

6.

Save the file and refresh your browser. Page through the list to see the sorted occupations.

## How It Works

The \$sorted-list variable is filled by iterating through the <row> nodes in the \$searchlist variable. As that is happening, the nodes are being sorted by the <occupation> node. At the same time the sorted <row> nodes are actually being placed into the \$sorted-list body with this element:

```
<xsl:copy-of select="current()"/>
```

What we see here is that the XPath function current() will return the node that <xsl:for-each> is currently pointing to. As the <xsl:for-each> element loops through the node-set that has been selected for it, each node is available to the current() function in turn. Using <xsl:copy-of> here ensures a 'deep' copy of all of the children under the current node.

## Details Display Template

The last template that the user would normally see shows the name and address details of the item selected in the list. This is the easiest to do because there is no row iteration, things to click on or conditional displays.

The first thing to do is set up a variable that only contains the node-set containing the data about the selected person. Therefore, the XPath filter expression would test against the id parameter that was created in this HTML anchor node:

```
<a href="{{$app}}?id={{contrib_id}}. . .">
<xsl:value-of select="l_name"/>
```

</a>

The tree to be filtered in the \$searchlist parameter has already been filtered down from the source document. So our person variable looks like:

```
<xsl:variable name="person" select="$searchlist[contrib_id = $param/id]"/>
```

One last thing to check for is the presence of the id parameter altogether. If it isn't there at all because the user hasn't selected a single person from the search results yet, the entire details area should not be displayed. This check for the id parameter is implemented in the <xslchoose> element, the first element in the template.

```
<xsl:template name="detail">
  <xsl:choose>
    <!-- if a single person has been selected, include the following -->
    <xsl:when test="$param/id" > <!-- is the id parameter present? -->

      <!-- variable holds just the details data nodes for the selected person -->
      <xsl:variable name="person" select="$searchlist[contrib_id = $param/id]"/>
      <table width="100%">
        <tr>
          <!-- display the title bar with first and last names -->
          <td colspan="2" bgcolor="silver" align="center"><b>
            <xsl:value-of select="$person/f_name"/><xsl:text> </xsl:text>
            <xsl:value-of select="$person/l_name"/></b>
          </td>
        </tr>
        <tr>
          <td>Address</td>
          <td><xsl:value-of select="$person/addr1"/></td>
        </tr>
        <tr>
          <td>City</td>
          <td><xsl:value-of select="$person/city"/></td>
        </tr>
        <tr>
          <td>State</td>
          <td><xsl:value-of select="$person/state"/></td>
        </tr>
        <tr>
          <td>Zip</td>
          <td><xsl:value-of select="$person/zip"/></td>
        </tr>
        <tr>
          <td>Employer</td>
          <td><xsl:value-of select="$person/employer"/></td>
        </tr>
        <tr>
          <td>Occupation</td>
          <td><xsl:value-of select="$person/occupation"/></td>
        </tr>
      </table>
    </xsl:when>

    <!-- no person has been selected -->
    <xsl:otherwise>
      <center><b>No item selected</b></center>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

# XML Debugging Template

If you want to see the actual XML that is contained in the various variables, this handy page sorts them out and displays them depending on the variable that the user selects. It also has a link to start a small JavaScript that runs on the browser (not in the XSLT) that will close the browser window. This feature shows the basic use of JavaScript inside an XSLT template. The overall template layout starts with the JavaScript and the links to select the desired section for debugging:

```
<script language="javascript">
<xsl:text>
function closeWindow()
{
    window.close();
}</xsl:text>
</script>

<font face="arial">
<b>
<a href="{{$app}}?debug=parameters&search={{$param/search}}&page={{$param/page}}>
Parameters</a>
<br/>
<a href="{{$app}}?debug=search&search={{$param/search}}&page={{$param/page}}>
Search List</a>
<br/>
<a href="{{$app}}?debug=doc&search={{$param/search}}&page={{$param/page}}>
Input Document</a>
<br/>
</b>
<br/>
<a href="{{$app}}" onClick="closeWindow()">Close Window</a>
```

There are two tricks being used here. The first is the use of the debug parameter to select a specific XML display or just display the page without any XML selection displayed at all. It works by using two different `<xsl:choose>` elements. The first one was discussed when the top-level rules template was first introduced. It has a test criteria that just looks for the presence of the debug parameter using the `<xsl:when test="$param/debug">` element. Notice that this doesn't test the \$param/debug element for a value, it just tests for its presence. Then in the `<xsl:choose>` node below, each `<xsl:when>` node tests for the \$param/debug node to have a specific value. So, if the \$param/debug node is present but doesn't have any specific value, just the menu appears without a specific XML display.

The other XSLT trick to pay attention to here is the use of the `<xsl:text>` element. This allows you to embed unaltered text inside the HTML `<script>` node. Without the `<xsl:text>` element, the actual JavaScript will appear outside of the `<script>` node. The JavaScript is called in the normal way with the `onClick` attribute of the HTML `<a>` tag. The rest of the template is an `<xsl:choose>` node with `<xsl:when>` elements that make their selection according to the value for the \$param/debug element which comes from the `<a href="...">` tag described in the paragraph above:

```
<xsl:choose>
<!-- **** display parameters node *** --&gt;
&lt;xsl:when test="$param/debug = 'params'"&gt;
&lt;center&gt;&lt;b&gt;Parameters&lt;/b&gt;&lt;/center&gt;
&lt;xmp&gt;
&lt;xsl:copy-of select="/table/param"/&gt;
&lt;/xmp&gt;
&lt;/xsl:when&gt;

<!-- *** display the search results list *** --&gt;
&lt;xsl:when test="$param/debug = 'search'"&gt;
&lt;center&gt;&lt;b&gt;Search Results&lt;/b&gt;&lt;/center&gt;
&lt;xmp&gt;
&lt;xsl:copy-of select="$searchlist"/&gt;</pre>

```

```
</xmp>
</xsl:when>

<!-- *** display the entire source doc *** -->
<xsl:when test="$param/debug = 'doc'">
    <center><b>Source Document</b></center>
    <xmp>
        <xsl:copy-of select="/" />
    </xmp>
</xsl:when>

<!-- ***** $param/debug has no value ***** -->
<xsl:otherwise>
    <center><b>No XML Tree Selected</b></center>
</xsl:otherwise>
</xsl:choose>
```

The combination of the HTML tag `<xmp>` with the XSLT element `<xsl:copy-of>` shows all elements in the document or temporary tree held in the variables. The `<xsl:copy-of>` element puts everything out to the HTML page as if it was a document file you can read. Then the surrounding `<xmp>` tags tells the browser to not try to interpret those tags that came from the source document but just display them as plain text. Without the `<xmp>` tags, the browser won't show the XML tag elements at all. Remember that the `<xsl:copy-of>` element does a "deep copy" meaning that it will copy every element it finds under the root of its select attribute including all its children, and their children, and so on. The result is a true view of everything that exists in the node or temporary tree held in the variable.

The complete debugging page template listing:

```
<xsl:template name="debug">
    <script language="javascript">
        <xsl:text>
            function closeWindow()
            {
                window.close();
            }
        </xsl:text>
    </script>
    <font face="arial">
        <b><a href="{{$app}}?debug=parameters&search={$param/search}&page={$param/page}">Parameters</a><br/>
        <a href="{{$app}}?debug=search&search={$param/search}&page={$param/page}">Search List</a><br/>
        <a href="{{$app}}?debug=doc&search={$param/search}&page={$param/page}">Input Document</a><br/>
        </b><br/>
        <a href="{{$app}}" onClick="closeWindow()">Close Window</a>
    </font>
    <xsl:choose>
        <!-- ***** display parameters node ***** -->
        <xsl:when test="$param/debug = 'params'">
            <center><b>Parameters</b></center>
            <xmp>
                <xsl:copy-of select="/table/param"/>
            </xmp>
        </xsl:when>

        <!-- *** display the search results list *** -->
        <xsl:when test="$param/debug = 'search'">
            <center><b>Search Results</b></center>
            <xmp>
                <xsl:copy-of select="$searchlist"/>
            </xmp>
        </xsl:when>
    </xsl:choose>
```

```
<!-- *** display the entire source doc ***** -->
<xsl:when test="$param/debug = 'doc'">
    <center><b>Source Document</b></center>
    <xmp>
        <xsl:copy-of select="/" />
    </xmp>
</xsl:when>
<!-- **** $param/debug has no value **** -->
<xsl:otherwise>
    <center><b>No XML Tree Selected</b></center>
</xsl:otherwise>
</xsl:choose>
</font>
</xsl:template>
```

This concludes the detailed description of the XSLT stylesheet used for this application. Next a few tips and tricks about coding errors, error messages, possible uses of the document() function along with the \$\_real-path and \$\_session-id elements for external referencing, file referencing, and session storage.

---

[!\[\]\(86b969eb085e8da909edc9382837f521\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(df9eda3dc665fca10e2367369eb7d7ec\_img.jpg\) NEXT >](#)

# Suggestions for Enhancement of the Application

One major feature missing from this application is a user log in. However, there is a relatively painless way that this can also be built right in the existing stylesheet by using the `<xsl:document>` element and the `document()` function to read and write files. What this implies is that the transform can have multiple source documents, not just the primary document, and can redirect the output of a node to an output file. Without going into the details of these transform functions, there is one important piece of information that is needed to do this and that is the real file path to the running servlet. That is, opening and saving files requires some knowledge by the stylesheet of just where it is running. This may be available when the primary source document is read from a file but in this case the primary source document comes from a DOM so the transform has no way of knowing where it is in the operating system's file space.

Another application enhancement is keeping user session information on the server between browser requests. This technique of using stateful sessions is widely used in Java Server Pages (JSP) and Microsoft's ASP but hasn't been touched on in this application. However, this can also be done by reading and writing XML files that are unique to a particular user session.

Addressing these needs, I have provided two top level parameters that have information about the file path and the session ID. These two parameters must be declared directly within the `<xsl:stylesheet>` node as top level elements, not under a specific template.

```
<xsl:param name="_real-path" select="'no-path-found'"/>
<xsl:param name="_session-id" select="'no-session-found'"/>
```

*Note that these parameter names cannot be changed as they map directly to the servlet code.*

You can read a previously saved session document using these parameters with:

```
<xsl:variable name="session" select="document(concat('file:',
substring-after($_real-path, 'C:'), 'sessions/', $_session-id, '.xml'))"/>
```

and you can write that session document with this element:

```
<xsl:document href="{concat(substring-after($_real-path, 'C:'),
'sessions/', $_session-id, '.xml')}">
. . .
</xsl:document>
```

Note that this code assumes you are running under Windows where the C: will be interpreted by the transform engine to be protocol descriptor, not part of the file path. The methods that determine how to use the href attribute are implemented in the Java URISolver class. This is an area under development so it may not be reliable for all transform engines and operating systems. However, I have installed a custom URISolver in more advanced servlets. The results have completely enabled the use of `<xsl:document>` and also `<xsl:import>`, the element used to modularize stylesheets. What's more, in any operating system you will need a periodic timed process to delete these session files that are over an hour old.

There are many other possibilities to enhance the stylesheet by importing other stylesheets or reading other document files that can contain HTML as headers or footers. Again, this can be done by using the `$_real-path` parameter to know where you are in file space. This will work no matter where you install the servlet or Tomcat or what operating

system you are using.

## Servlet Enhancements

The detail of servlet design is outside of the scope of this chapter but the sourcecode is included with adequate documentation for Java developers who would like to modify it. For experienced Java servlet developers, quite a few potential enhancements will be obvious. For instance, both the source document and the prepared stylesheet can be cached and only reloaded if their source file timestamp has changed. My experiments with this have improved performance over ten times for the exact same document and stylesheet. Furthermore, adding more transforms can create XML pipelines with intermediate results stored in sessions, create access to relational and XML databases, connect to e-mail and get fast logging, just to mention a few possibilities.

Both the higher speed caching servlet and database connection servlet are available on the author's web site at <http://www.xlroot.com>.



[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

By examining this application closely and playing with the two Try It Outs, you should have a good enough understanding of the XSLT template to be able to build true interactive web applications completely in XSLT. Although this case study is limited by the size and performance of a file-based XML document, the same principals would apply if the source document were to be obtained from a relational database, which would far exceed an XML file for search performance. As a practical matter, a truly fast performing application running against a large database could be built using a number of XSLT transforms in a flexible architecture that allowed for easy pipelining of the output of one transform into a database request and the output of that SQL query into other transforms for further processing or display.

An overall architectural concept about using the XSLT transform engine to build applications is that the transform should be used to implement business rules logic, not to access and control large amounts of data. By viewing the transform as a process for building complex user interaction behavior, the need for data to drive that interaction is minimal, it just needs enough data to get the job done for the immediate needs. For instance, all the data this application really needs is enough to efficiently show a number of list pages. It doesn't really need access to the entire XML source document for the application as it is presently designed. If some outside data source were to deliver just enough data to populate a few pages of the list, the entire application would perform much better. However, in the name of simplicity the approach used in this application is much easier to understand and control in a single XSL transform.

Finally, the best possible value you can get from this case study is through digging in and modifying this XSLT code to suit your own purposes, so why not give it a try?

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Case Study 2: XML Web Services

XML is a powerful tool for representing and exchanging data. Developers and businesses can rely on the standard format of XML to easily exchange data between applications and systems. Sometimes the format of an XML document needs to be massaged a bit before it can be exchanged easily, and in those cases, XSLT can be used to transform XML. Vocabularies of XML, like XSL, SOAP, or XHTML, have been created and have extended XML far beyond its original application. Web Services in particular are based upon the use of several XML dialects, like SOAP, to perform data exchanges in ways that will truly reshape the way we think about programming.

The example in this case study was developed to show a Web Service in action, but it also serves as an example of how XML can be used in several different ways within a single application. In general, this example shows how:

- XML document files stored on the server can serve as a data store for an application
- XML messages function as a standard for communication between systems
- XSLT can be used as the engine of a Web Service

To be more specific, this case study demonstrates the following techniques:

- Using ASP as a listener for SOAP requests
- Using XSLT to process SOAP requests
- Using external XML documents as the source of data for a transformation
- Writing Web Service clients using JavaScript

## Overview

Web Services are functional components that are accessible using standard Internet protocols, such as HTTP. Vendors such as Microsoft and IBM have released development tools that simplify the process of building Web Service clients and servers. In some cases, the tools make the migration from existing code to Web Service almost transparent. In fact, the next generation of development tools from these and other vendors will make coding a web service so simple that the real challenge is placed on computer book authors: come up with an example that is both relevant to a wide range of developers and at least mildly interesting.

## What Should We Build?

The example we will build in this case study is a cookbook exposed as a Web Service. A traditional cookbook would allow you to browse recipes by category, searching for some elaborate dish to create. This cookbook,

however, is designed for the less ambitious chef. Our Web Service assumes that you are a developer (or architect, or college student, insert your favorite procrastinating profession here) up at 4a.m in search of something to eat, and find that your refrigerator is empty except for a handful of unrelated items (for example, eggs and soy sauce). What can you make with eggs and soy sauce? Our Web Service will answer that question for you. The Web Service will implement a single method: GetRecipes. The GetRecipes method accepts a list of ingredients, and returns one or more recipes that can be made with what you have available.

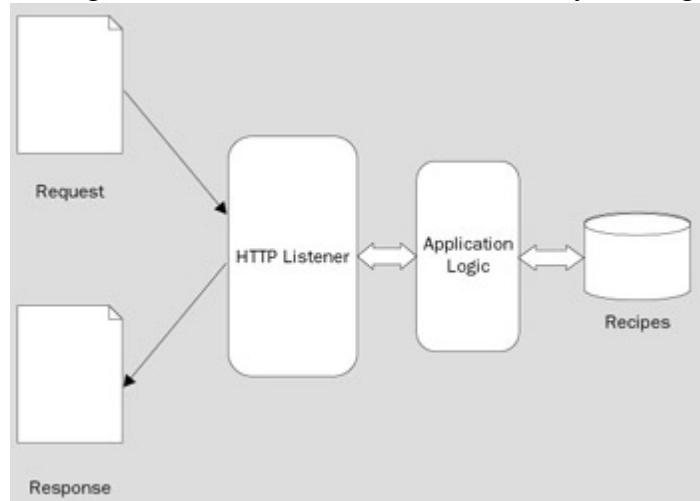
## How Should We Build It?

Now that we know what our Web Service will do, we must decide how we want to build it. In general, we want to stick with standard XML and avoid any specific vendor extensions (such as scripting), building a system that allows us to be flexible in the future.

Let's take a look at what we need to build our Web Service, and then we will decide how to address those needs from a technical standpoint. In order to build our recipe service, we need to:

- Search through a list of recipes and return ones that match our ingredients
- Request recipe lists from the Web Service using a SOAP client application
- Accept SOAP requests and return SOAP responses over HTTP

The figure below shows a model of what our system might look like.



## Building the Server

Let's take a look at each piece of the system, starting with the recipe data itself. As we assemble the components of our Web Service, we will decide how best to implement each portion. Note that all of the code in this case study assumes that is located in a virtual directory called `recipes` on your web server.

### An XML Cookbook

Our Web Service is going to provide a list of recipes based on the SOAP request it receives, so we need to store our cookbook somewhere on the server. There are a number of ways we could store the data: anything from a simple ASCII text file to a fully featured relational database could serve our purpose. As we have seen in earlier chapters, XML is a great way to represent data in a format that is easily accessible and interchangeable, so let's stick with that model and use an XML document file as our database.

For our example, the cookbook is contained inside one XML document file, recipes.xml, which is located on the server. The document contains a single <cookbook> element as the document element, and the <cookbook> contains one or more <recipe> elements. Every <recipe> has information about the ingredients needed to prepare the dish, as well as the steps involved in creating it. There is also additional information, such as the <name> and <author>.

The code below shows a small version of our cookbook, one that contains only three simple recipes, but that will be enough data for a test of the service.

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Cookbook XML Case Study -->
<cookbook xmlns='http://www.wrox.com/recipes/'>
  <recipes>
    <recipe>
      <name>Peanut Butter and Jelly Sandwiches</name>
      <author>Calvin Dix</author>
      <ingredients>
        <ingredient>peanut butter</ingredient>
        <ingredient>jelly</ingredient>
        <ingredient>bread</ingredient>
      </ingredients>
      <directions>
        <direction>Take two slices of bread.</direction>
        <direction>Using a knife, apply peanut butter to the first slice.</direction>
        <direction>Apply jelly to the second slice.</direction>
        <direction>Put the slices together.</direction>
      </directions>
    </recipe>
    <recipe>
      <name>Macaroni and Cheese</name>
      <author>Alexander Dix</author>
      <ingredients>
        <ingredient>macaroni</ingredient>
        <ingredient>cheese packet</ingredient>
      </ingredients>
      <directions>
        <direction>Boil water.</direction>
        <direction>Place the macaroni into the boiling water.</direction>
        <direction>Stir the macaroni for 5 minutes.</direction>
        <direction>Drain all the water from the macaroni.</direction>
        <direction>Mix in the cheese packet.</direction>
      </directions>
    </recipe>
    <recipe>
      <name>Hard Boiled Eggs</name>
      <author>Jennifer Dix</author>
      <ingredients>
        <ingredient>eggs</ingredient>
      </ingredients>
      <directions>
        <direction>Boil water.</direction>
        <direction>Place the egg(s) into the boiling water for 5 minutes.</direction>
        <direction>Reduce heat to low and leave eggs in water for 15
minutes.</direction>
        <direction>Empty the water and replace it with cold water.</direction>
        <direction>Leave the eggs in the water for 5 minutes.</direction>
        <direction>Place eggs in the refrigerator to cool.</direction>
      </directions>
    </recipe>
  </recipes>
</cookbook>
```

Now that we have our data in place, we need to decide what the interface to this data will be.

## GetRecipes

In order to build our Web Service, we need to define what it is going to do in more detail. We have already stated that our service will accept a list of ingredients, and return a list of recipes. The data stored in recipes.xml contains both types of data, so to keep things simple we will use that as our model for representing data in our messages.

The Web Service will implement one method, GetRecipes. In pseudo-code, it will look something like this:

```
Function GetRecipes( array of ingredients ) Returns array of recipes
```

What does an array of ingredients, or of recipes, look like? In recipes.xml, we have already defined how an array of ingredients will be represented: using the <ingredients> element. Similarly, recipes.xml contains an element called <recipes>. This element represents an array of <recipe> elements, and so it is this element that we will use in the response.

The code below shows an example of a SOAP request that our Web Service will handle from the client:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
    <R:GetRecipes xmlns:R='urn://www.wrox.com/recipes/'>
        <R:ingredients>
            <R:ingredient>eggs</R:ingredient>
        </R:ingredients>
    </R:GetRecipes>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP message elements, <Envelope> and <Body>, are boilerplate code. The GetRecipe method is our call, and the <ingredients> element it contains is our list of ingredients. Now, let's look at the response:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
    <R:GetRecipesResponse xmlns:R='urn://www.wrox.com/recipes/'>
        <recipes xmlns="urn://www.wrox.com/recipes/">
            <r:recipe xmlns:r="http://www.wrox.com/recipes/">
                <r:name>Hard Boiled Eggs</r:name>
                <r:author>Jennifer Dix</r:author>
                <r:ingredients>
                    <r:ingredient>eggs</r:ingredient>
                </r:ingredients>
                <r:directions>
                    <r:direction>Boil water.</r:direction>
                    <r:direction>Place the egg(s) into the boiling water for 5
minutes.</r:direction>
                    <r:direction>Reduce heat to low and leave eggs in water for 15
minutes.</r:direction>
                    <r:direction>Empty the water and replace it with cold water.</r:direction>
                    <r:direction>Leave the eggs in the water for 5 minutes.</r:direction>
                    <r:direction>Place eggs in the refrigerator to cool.</r:direction>
                </r:directions>
            </r:recipe>
        </recipes>
    </R:GetRecipesResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Just like our request, the message contains data modeled from recipes.xml. Now, let's examine what it takes to get from a request to a valid response.

## External Documents

All the information that we need to build the SOAP response is not contained in the SOAP request. Like any real world Web Service, the service code needs to access some external logic or data in order to process the request. In our case, the external data that we need is inside recipes.xml, our XML cookbook.

There are a couple of different approaches we could take at this point. For example, we could:

- Execute code in another language, such as Java or VBScript, to read the recipe list from our file
- Use extension elements of the XSLT processor to access the file
- Stick with standard XSLT functions to access the data

It might be simpler and involve writing less code to use an extension or escape to another language. One of the purposes of this case study, however, is to see how far we can take an XML-only solution, so we would like to avoid any XSLT extensions that lock us into a particular processor. Luckily for us, it is a common problem to have data required for the transformation in an XML document other than the source. Often in XSLT, multiple documents might be needed to process the transformation. In those cases, the XSLT document function can be used.

*Using an XML document file for our data store and accessing it in this manner is an approach that will not scale well to very large data sets. If you are building a Web Service that has steep requirements for data, it is better to leverage the capabilities of a relational database. If your data set is small, however, you may find it much easier to work with XML files as we do in this case study.*

### document() function

XSLT has a variety of built-in functions that can be used in expressions. These functions provide valuable features to the language, such as formatting, type conversion, and evaluation. The document() function is one of these functions.

The document() function allows you to refer to an external XML document within an XSLT stylesheet. Using the document function, you can then access the tree of that document and use it to perform additional steps in the transformation. You use the document() function by passing it the path to the external document, which is relative to the current stylesheet. For example, the code below loads the tree of the document myData.xml as a node-set and returns it into the XSLT variable myData.

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

### Try It Out?document() Function

Before we use it in our Web Service, let's try out the document() function to see how an external XML document can be used inside a transformation. In this case, we have two XML documents: one with a list of stores, and one with a list of regions. We will use the document() function to look up regional information for each store from the regional document, working with that data as an external XML document.

1.

We start with our source XML document, stores.xml, which contains a list of stores:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="regions.xsl"?>
<stores>
  <store>
    <name>Willie's Market</name>
    <manager>Willie Dewit</manager>
    <region>S</region>
  </store>
  <store>
    <name>The Country Store</name>
    <manager>Jessica Long</manager>
    <region>W</region>
  </store>
  <store>
    <name>L'Apelier</name>
    <manager>Ray Budd</manager>
    <region>MA</region>
  </store>
</stores>
```

## 2.

We also need our external document, regions.xml, which contains the regional information:

```
<?xml version="1.0"?>
<regions>
  <region>
    <code>MA</code>
    <name>Mid-Atlantic</name>
    <phone>412-555-8000</phone>
  </region>
  <region>
    <code>S</code>
    <name>South</name>
    <phone>704-555-1342</phone>
  </region>
  <region>
    <code>W</code>
    <name>West</name>
    <phone>399-555-8080</phone>
  </region>
</regions>
```

## 3.

The next step is to create the template that will transform one of our stores into HTML that contains more detailed regional information:

```
  <xsl:template match="store">
    <P>Store Name:
    <xsl:value-of select="name" />
    </P>
    <xsl:variable name="regionCode" select="region" />
    <xsl:variable name="regions"
      select="document('regions.xml')//*[local-name()='region']" />
    <xsl:for-each select="$regions">
      <xsl:if test="$regionCode=code" >
        <P>Contact:
        <xsl:value-of select="phone" />
      </P>
    </xsl:if>
  </xsl:for-each>
</xsl:template>
```

Notice the match attribute on the <xsl:template> element. This value indicates what elements in the source XML should be associated with this template. During the transformation, any elements in the source document that are named store will trigger the above template. The document() function returns the node-set contained in the file regions.xml, and that node-set is placed into the variable regions.

4.

We then place this template inside our stylesheet, regions.xsl:

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

<xsl:template match="/">
  <html>
    <head><title>Regional Information</title></head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>

<xsl:template match="store">
  <p>Store Name:<br/>
  <xsl:value-of select="name" />
  </p>
  <xsl:variable name="regionCode" select="region" />
  <xsl:variable name="regions"
    select="document('regions.xml')/*[local-name()='region']" />
  <xsl:for-each select="$regions">
    <xsl:if test="$regionCode=code" >
      <p>Contact:<br/>
        <xsl:value-of select="phone" />
      </p>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

5.

Finally, we trigger our transformation, and we get the results below:

```
<html>
  <head><title>Regional Information</title></head>
  <body>
    <p>Store Name: Willie's Market</p>
    <p>Contact: 704-555-1342</p>
    <p>Store Name: The Country Store</p>
    <p>Contact: 399-555-8080</p>
    <p>Store Name: L'Apelier</p>
    <p>Contact: 412-555-8000</p>
  </body>
</html>
```

Our stylesheet, recipes.xsl, uses the document() function in the same way that it is used in this example. Data located in the external document is combined with data in the source XML document to make a single transformation. Now that we know how to work with the document function, let's write the XSLT that will handle our request.

## Processing the Request

Once we have received a SOAP request, we must process it. Depending on how a Web Service is implemented, the code that processes a request could be written in almost any language. Frameworks like Microsoft's .NET or Apache Axis provide "glues" that translate between application code and SOAP requests and responses.

The "glues" that are provided by these implementations perform translations of some kind between SOAP XML and applications, but those translations performs the same actions as XSLT transformations. If we choose, we can write the entire process in XSLT, effectively coding our Web Service logic entirely in XSLT.

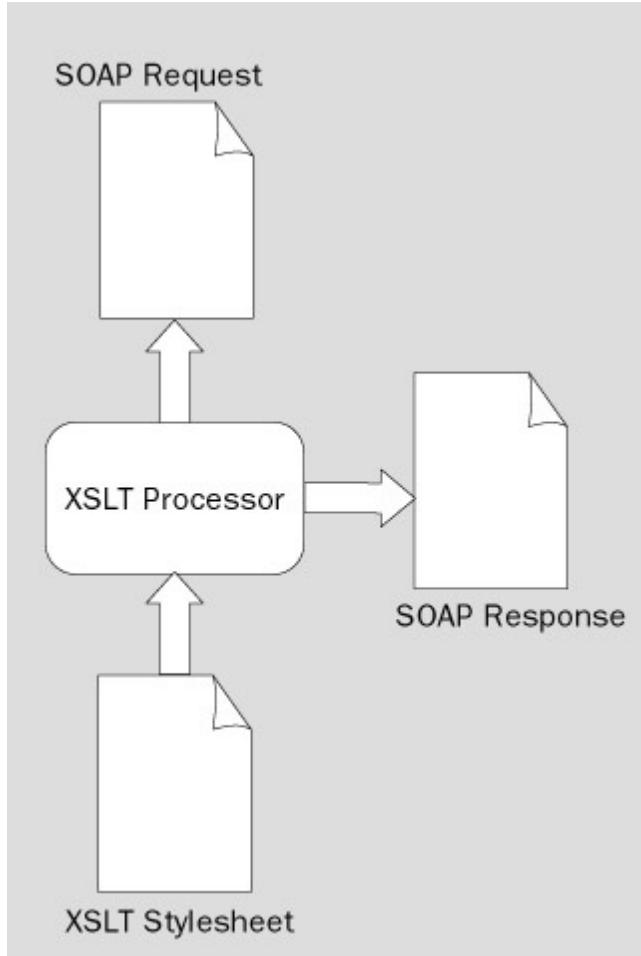
## XSLT Advantages

Just because we can write a Web Service in XSLT does not mean that we should. There are plenty of ways to write a Web Service, and there should be some motivating factor that pushes us to use XSLT as a method of implementing a Web Service. In fact, there are several advantages to building a Web Service in XSLT:

- XSLT processors are available for many platforms, so using standard XSLT gives you portable Web Service code.
- XSLT is intended for XML transformations, so the language lends itself to working with SOAP content directly.
- When using XSLT as your engine, you have complete control over the responses you generate. Most toolkits for developing Web Services remove your code from directly accessing the XML of the message. With direct access to the code that does the transformation from the SOAP messages, you have direct control over the SOAP responses.

## SOAP Transformations

The figure below illustrates how we use XSLT to transform the SOAP request, just as if it were any other type of source XML document.



The SOAP request is the source for our transformation, and the SOAP response is the output. The missing piece of this transformation is, of course, the XSLT stylesheet that will perform this translation. In general, this stylesheet needs templates that will be able to:

- Validate the general structure of the SOAP message
- Return faults when errors occur
- Extract the body blocks and pass them to a specialized template for processing

Next we will take a look at what that stylesheet looks like.

*For a review of the XSLT used in the rest of this case study, refer to [Chapter 4, XSLT](#), for complete coverage of the XSLT syntax.*

### **recipes.xsl**

Let's build the stylesheet that will process our requests. We start with the standard `xsl:stylesheet` element, but we also use another XSLT element, `xsl:output`. The `xsl:output` element tells the processor what encoding to use when generating the output. The attribute `omit-xml-declaration` tells the processor not to generate the XML declaration (`<?xml version='1.0'?>`) as part of the transformation. We will see why in a moment.

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

```
indent="yes" encoding="utf-8"
omit-xml-declaration="yes"/>
```

Next, we start our templates. If we are going to transform SOAP requests, one of the possibilities that we must handle is receiving a bad message. When an error occurs at a Web Service that is accepting SOAP requests, the service is responsible for returning a SOAP Fault. To help with this, we provide a couple of templates. The first is the ThrowFault template, which will return a fault given a string and a code.

The ThrowFault template uses an XSLT element called, ironically, <xsl:element>. Using <xsl:element> is the equivalent of including a literal element in your stylesheet, but it allows you to defer to the processor the burden of selecting namespace prefixes for your elements.

```
<!-- ThrowFault -->
<xsl:template name="ThrowFault">
  <xsl:param name="code"/>
  <xsl:param name="str"/>

  <xsl:element name="Fault"
    namespace="http://schemas.xmlsoap.org/soap/envelope/">
    <xsl:element name="faultcode"
      namespace="http://schemas.xmlsoap.org/soap/envelope/">
      <xsl:value-of select="$code"/>
    </xsl:element>
    <xsl:element name="faultstring"
      namespace="http://schemas.xmlsoap.org/soap/envelope/">
      <xsl:value-of select="$str"/>
    </xsl:element>
  </xsl:element>
</xsl:template>
```

In order to make things even easier, we include two other templates that use ThrowFault to generate specific faults for versioning problems and for an unknown request. These are ThrowVersionMismatchFault and ThrowNoOperationFault, respectively.

```
<!-- ThrowVersionMismatchFault -->
<xsl:template name="ThrowVersionMismatchFault">
  <xsl:call-template name="ThrowFault">
    <xsl:with-param name="code">SOAP:VersionMismatch</xsl:with-param>
    <xsl:with-param name="str">The version did not match the 1.1
specification.</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<!-- ThrowNoOperationFault -->
<xsl:template name="ThrowNoOperationFault">
  <xsl:call-template name="ThrowFault">
    <xsl:with-param name="code">SOAP:Client.NoOperation</xsl:with-param>
    <xsl:with-param name="str">The requested operation does not
exist.</xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

With our fault support templates written, we can now write our template that will process the SOAP message.

```
<!-- ProcessMesage -->
<xsl:template name="ProcessMessage">
  <xsl:variable name="Envelope" select="child::*[local-name() = 'Envelope']"/>
  <xsl:text disable-output-escaping="yes">&#60;&#63;xml version="1.0"
encoding="UTF-8"&#63;&#62;</xsl:text>
```

```
<SOAP:Envelope xmlns:SOAP='http://schemas.xmlsoap.org/soap/envelope/'  
    SOAP:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'  
    xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'  
    xmlns:xsd='http://www.w3.org/1999/XMLSchema'  
    xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'>  
    <SOAP:Body>  
        <xsl:choose>  
            <xsl:when test="$Envelope">  
                <xsl:for-each select="$Envelope">  
                    <xsl:choose>  
                        <xsl:when test="self::s:*"  
                            xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">  
                            <xsl:variable name="Body"  
                                select="child::*[local-name() = 'Body']"/>  
                            <xsl:for-each select="$Body">  
                                <xsl:choose>  
                                    <xsl:when test="namespace-uri($Body) =  
'http://schemas.xmlsoap.org/soap/envelope/'">  
                                        <xsl:variable name="payload" select="child::*"/>  
                                        <xsl:choose>  
                                            <xsl:when test="local-name($payload)='GetRecipes'">  
                                                <xsl:call-template name="ProcessPayload">  
                                                    <xsl:with-param name="payload" select="$payload"/>  
                                                </xsl:call-template>  
                                            </xsl:when>  
                                            <xsl:otherwise>  
                                                <xsl:call-template name="ThrowNoOperationFault" />  
                                            </xsl:otherwise>  
                                        </xsl:choose>  
                                    </xsl:when>  
                                    <xsl:otherwise>  
                                        <xsl:call-template name="ThrowVersionMismatchFault" />  
                                    </xsl:otherwise>  
                                </xsl:choose>  
                            </xsl:for-each>  
                        </xsl:when>  
                        <xsl:otherwise>  
                            <xsl:call-template name="ThrowVersionMismatchFault"/>  
                        </xsl:otherwise>  
                    </xsl:choose>  
                </xsl:for-each>  
            </xsl:when>  
            <xsl:otherwise>  
                <xsl:call-template name="ThrowVersionMismatchFault"/>  
            </xsl:otherwise>  
        </xsl:choose>  
    </SOAP:Body>  
</SOAP:Envelope>  
</xsl:template>
```

All the templates so far have just provided us with a way to handle SOAP messages in general. This is significant, because we now have the beginnings of a framework for Web Services. All the code that has preceded this section was written to handle the basics of Web Services. Now, we need to write the templates that will answer the request. The ProcessMessage template calls our ProcessPayload template in order to generate the response payload. ProcessPayload is where all the application specific transformation occurs, so we could rewrite the code in ProcessPayload and create an entirely new Web Service. In this case, we use `<xsl:element>` to output the literal element GetResponsePayload, which is the name of our response body block.

```
<xsl:template name="ProcessPayload" >  
    <xsl:param name="payload" />
```

```
<xsl:element name="GetRecipeResponse"
    namespace="urn://www.wrox.com/recipes/" >
<xsl:variable name="recipeList"
    select="document('recipes.xml')/*[local-name()='recipe']"/>
<xsl:for-each select="$payload">
    <xsl:variable name="ingredientList"
        select="/*[local-name()='ingredient']" />

    <xsl:element name="recipes" namespace="urn://www.wrox.com/recipes/" >
        <xsl:for-each select="$recipeList">
            <xsl:variable name="recipe" select=". " />
            <xsl:variable name="IsRecipeOk">
                <xsl:call-template name="IsRecipeOk">
                    <xsl:with-param name="recipe"
                        select="$recipe" />
                    <xsl:with-param name="ingredientList"
                        select="$ingredientList" />
                </xsl:call-template>
            </xsl:variable>
            <xsl:if test="$IsRecipeOk = '1'" >
                <xsl:copy-of select="$recipe" />
            </xsl:if>
        </xsl:for-each>
    </xsl:element>
</xsl:for-each>
</xsl:element>
</xsl:template>
```

For each recipe in the cookbook, we call the template IsRecipeOk. That template is a Boolean test that compares our list of ingredients to the recipe. If there is a match, it returns a 1, and it returns a 0 if the recipe is not a match.

```
<xsl:template name="IsRecipeOk">
    <xsl:param name="recipe" />
    <xsl:param name="ingredientList" />

    <xsl:for-each select="$recipe" >
        <xsl:variable name="recipeIngredients"
select="child::*[local-name()='ingredients']/child::*[local-name()='ingredient']" />
        <xsl:variable name="foundIngredients">
            <xsl:call-template name="GetIngredientCount">
                <xsl:with-param name="recipe" select="$recipe" />
                <xsl:with-param name="ingredientList" select="$ingredientList" />
            </xsl:call-template>
        </xsl:variable>
        <xsl:variable name="listCount" select="count($recipeIngredients)" />
        <xsl:if test="number($foundIngredients) >= number($listCount)">1</xsl:if>
    </xsl:for-each>
</xsl:template>
```

In order to check the recipe, we need to determine if all of the ingredients of the recipe appear in our ingredient list from the request. The best way to do this is to count the total number of ingredients that we have in the recipe, and count the number of the ingredients that appear in our list. If they are the same, then we have a match. If they are not, the recipe requires some ingredient we do not have. The template that does this calculation is GetIngredientsCount, which uses **recursion** (which was discussed in [Chapter 4](#)) to walk the list of ingredients for a match.

```
<xsl:template name="GetIngredientCount">
    <xsl:param name="recipe" />
    <xsl:param name="ingredientList" />

    <xsl:for-each select="$recipe" >
```

```
<xsl:choose>
  <xsl:when test="$ingredientList" >
    <xsl:variable name="curr" select="$ingredientList[1]" />
    <xsl:variable name="rest">
      <xsl:call-template name="GetIngredientCount">
        <xsl:with-param name="recipe" select="$recipe" />
        <xsl:with-param name="ingredientList"
          select="$ingredientList[position() != 1]" />
      </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="check"
      select="child::*[local-name() = 'ingredients']/child::*[local-name() = 'ingredient'][string()
      ) = $curr]" />
    <xsl:choose>
      <xsl:when test="$check">
        <xsl:value-of select="$rest + 1" />
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$rest" />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:when>
  <xsl:otherwise>0</xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
</xsl:template>
```

The final template, which is actually the first template to get triggered during the transformation, is the only template in our stylesheet that is not named. This template uses the match attribute to trigger on the root of the source document, and it then passes the entire source document (in our case, the SOAP request) to the ProcessMessage template. After this template, we can close the `<xsl:stylesheet>` as well.

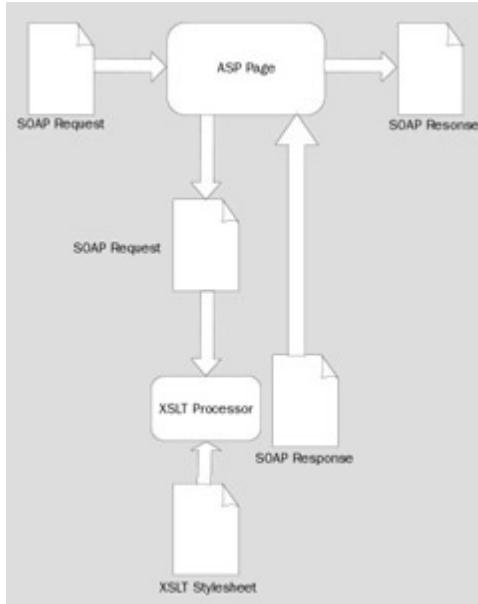
```
<!-- Process the whole document as a SOAP request -->
<xsl:template match="/">
  <xsl:call-template name="ProcessMessage" />
</xsl:template>
</xsl:stylesheet>
```

All those templates make up for a long stylesheet, but the advantage is that other than the listener and the XSLT processor, our Web Service is written entirely in XSLT, and can be ported to most platforms just by replacing the listener and processor.

## Listening for Requests

We have all the XSLT code in place now to transform a SOAP request. However, in order for our service to process SOAP requests, it has to listen for them. SOAP messages can be transported over any protocol, but the most popular choice for a transport binding (and the one defined by the SOAP specification) is HTTP. That means that we need to listen for incoming POSTs of SOAP messages. For our purposes, we will create a short ASP page to serve as our listener. The listener could be written in JSP, or it could be a CGI application, it really doesn't matter. All you need is some code executing that listens for incoming requests and can trigger the transformation.

The figure below shows how we can combine our XSLT processing with our ASP listener to create our server implementation.



Let's take a look at the code for our ASP listener. The page serves no purpose beyond listening for SOAP requests and triggering a transformation. It is the output of the transformation that is the SOAP response. To perform the transformation, we will use the Microsoft XML Parser object, DOMDocument.

The DOMDocument object has a number of methods and properties, exposing the full range of features that you might want in a DOM parser. For our purposes, we really only need to be concerned with two methods: load and transformNode.

The complete source for the ASP listener (`recipes.asp`) is shown below. As you can see, it takes very little code to read the incoming SOAP request and process it using a transformation.

```
<%@Language="VBSCRIPT"%>
<%
    Option Explicit

    Dim objSource
    Dim objXSL
    Dim strOutput

    Response.ContentType = "text/xml"

    Set objSource = Server.CreateObject( "MSXML2.DOMDocument.3.0" )
    Set objXSL = Server.CreateObject( "MSXML2.DOMDocument.3.0" )

    objSource.load Request
    objXSL.load Server.MapPath( "recipes.xsl" )

    strOutput = objSource.transformNode( objXSL )

    Response.Write strOutput

    Set objSource = Nothing
    Set objXSL = Nothing
%>
```

With only the code listed above, we have a functioning listener for our Web Service. We create two instances of the DOMDocument object, one to serve as the source, and the other to serve as the transformation code. The source is the incoming SOAP message, which is contained in the ASP Request object. Once that is loaded into the objSource variable, we also load `recipes.xsl` to perform the transformation. The `transformNode()` method does just that, and it returns the SOAP response, which we then write to the ASP Response object.

With the ASP listener in place, our server is complete. We have data, message processing, and a listener to handle the message transport. Now let's build a client that can access our recipes Web Service.

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Building the Client

With our Web Service completed, we need to create a client that will access the recipe list. There are a number of vendor toolkits that could help us here as well, such as the Microsoft SOAP Toolkit 2.0, or Apache's Java SOAP implementation. To keep things simple, however, we will build a simple web client that uses JavaScript and Microsoft's XMLHTTP objects to build our SOAP request.

*For a review of this technique, see the SOAP chapter of this book for additional client examples. The client shown below follows the same principles. For more information on using a vendor toolkit for Web Service clients and servers, see the Wrox Press book, Professional XML Web Services, ISBN 1-861005-09-1.*

In our client, we need to be able to select one or more ingredients, and then trigger the request to the Web Service. Also, we need to be able to display the response returned by the Web Service in a friendly way as part of our page. The picture below shows what our client page will look like:



Now, let's take a look at the code needed to make this page, recipes.htm:

```

<html>
<head>
<title>Recipe Client</title>
</head>

<XML id="xslSource" src="showrecipes.xsl"></XML>

```

The code above looks like any of a million pages, except for the <XML> tag. This includes in this page a block of XML content that will not be displayed. In this case, the XML content that we are including is another XSL stylesheet, one that we will use to format the SOAP response. We will look at the contents of that file in a moment, but first let's examine the rest of the client.

```

<SCRIPT FOR="cmdLookup" EVENT="onclick">
  var doc, web, strXML;

  strXML = "<s:Envelope xmlns:s='http://schemas.xmlsoap.org/soap/envelope/'>\n\t";
  strXML = strXML + "<s:Body>";
  strXML = strXML + "<r:GetRecipes>";

```

```
strXML = strXML + " xmlns:r='urn://www.wrox.com/recipes/'>";
strXML = strXML + "<r:ingredients>";
```

The code above shows a JavaScript handler for the Get Recipes button on the client page. This is triggered whenever the button is clicked. Whenever that happens, we build a SOAP request, send it to the server, and process the response. First, we build the envelope and payload elements. The next step is to determine what ingredients we should send in our request. Our ingredient list is contained in the list box on the page, and one or more of the ingredients are selected. The code below loops through the contents of the list box, adding one ingredient element to the payload for each one that is selected. The list box, named listIngredients, is accessed through the browser's document object model with the document.listform.listIngredients syntax.

```
// Each line like this adds one ingredient
var list, i;

for ( i = 0; i < document.listform.listIngredients.length; i++ )
{
    if ( document.listform.listIngredients.options[i].selected == true )
    {
        strXML = strXML + "<r:ingredient>";
        strXML = strXML + document.listform.listIngredients.options[i].value;
        strXML = strXML + "</r:ingredient>";
    }
}

strXML = strXML + "</r:ingredients>";
strXML = strXML + "</r:GetRecipes>";
strXML = strXML + "</s:Body>";
strXML = strXML + "</s:Envelope>";
```

The next step is to load the string XML into a DOMDocument object. This will allow us to verify that it is valid XML.

```
doc = new ActiveXObject("MSXML2.DOMDocument.3.0");
doc.loadXML(strXML);
```

Next, with all the message data in place inside the doc variable, we can create our XMLHTTP object. XMLHTTP is a COM object that is part of the Microsoft XML Parser, MSXML. Using XMLHTTP, we are able to post XML data over HTTP. In our case, the variable web is our XMLHTTP object. We open the connection to the server, and send the request.

```
web = new ActiveXObject("MSXML2.XMLHTTP");
web.open("POST", "http://localhost/recipes/recipes.asp", false);
web.send(doc);
```

The last step in our client is to read the response as XML and load it into another DOMDocument object so that we can parse and transform it. The response is returned to us as the responseXML property of the XMLHTTP object. Once the response XML can be parsed, we can do whatever we need with the data. In our case, we will transform it using the showrecipes.xsl stylesheet that we referred to earlier on the page. That stylesheet will format the response into the HTML shown previously in the client page image. That is done by setting the innerHTML property of that page element to the output of the transformation, as shown here:

```
doc = web.responseXML;

var strDisplay;
strDisplay = doc.transformNode( xslSource );

result.innerHTML = strDisplay;
```

```
</SCRIPT>
```

That is all the scripting code we need in order to create our client. The rest of the page, shown below, contains just the HTML needed to create the look of the page and define its elements.

```
<body bgcolor="#336699">
<table border="0" cellspacing="0" style="border-collapse: collapse"
bordercolor="#111111" width="100%" cellpadding="3">
<tr>
<td width="100%" bgcolor="#000000" align="center"><b>
<font color="#FFFFFF" size="6">WhatCanIMakeForDinner.com</font></b></td>
</tr>
<tr><td width="100%">&nbsp;</td></tr>
<tr>
<td width="100%"><font color="#FFFFFF">Select one or more recipes from
the list below, and see what you can make for dinner!</font>
</td>
</tr>
<tr>
<td width="100%" align="left" valign="top">&nbsp;</td>
</tr>
</table>
<table border="0" cellspacing="0" style="border-collapse: collapse"
bordercolor="#111111" width="100%" cellpadding="3">
<tr>
</tr>
<tr>
<td bordercolor="#111111" width="100%" cellpadding="0">
<table border="0" cellspacing="0" style="border-collapse: collapse"
bordercolor="#111111" width="100%" cellpadding="0">
<tr>
<td width="100%" align="left" valign="top">
<table border="0" cellspacing="0" style="border-collapse: collapse"
bordercolor="#111111" width="100%" cellpadding="0" height="54">
<tr>
<td height="28" valign="top" align="left">
<input type="button" value="Get Recipes" name="cmdLookup">
</td>
<td width="100%" height="28">&nbsp;</td>
</tr>
<tr>
<td width="100%" height="26" align="left" valign="top">
<form name="listform">
<select size=10 multiple name="listIngredients">
<option value="apples">apples</option>
<option value="macaroni">macaroni</option>
<option value="cheese packet">cheese packet</option>
<option value="peanut butter">peanut butter</option>
<option value="jelly">jelly</option>
<option value="bread">bread</option>
<option value="eggs">eggs</option>
<option value="butter">butter</option>
<option value="soy sauce">soy sauce</option>
<option value="tomatoes">tomatoes</option>
</select>
</form></td>
</tr>
</table>
</td>
<td width="83%" align="left" valign="top"><DIV id="result"/></td>
</tr>
```

```
</table>
</td>
</tr>
</table>
</tr>
<tr>
    <td width="100%">&nbsp;</td>
</tr>
<tr>
    <td width="100%" bgcolor="#000000">&nbsp;</td>
</tr>
</table>
</body>
</html>
```

With our page complete, the only thing left to examine is showrecipes.xsl stylesheet.

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

    <!-- showrecipes.xsl -->
    <!-- Written by Chris Dix, 2001 -->

    <xsl:template match="/">
        <xsl:variable name="recipeList" select="//*[local-name() = 'recipe']" />
        <table border="1" bordercolor="#111111" cellspacing="0"
            cellpadding="3" style="font-size:10pt; font-family:Verdana;
            color:black;">
            <xsl:if test="not($recipeList)">
                <tr><td>No match</td></tr>
            </xsl:if>
            <xsl:for-each select="$recipeList">
                <tr><td>
                    <p><B><xsl:value-of select="child::*[local-name() = 'name']"/></B> by
<xsl:value-of select="child::*[local-name() = 'author']"/></p>
                    <xsl:variable name="ingredientList"
                        select="child::*[local-name() = 'ingredients']" />
                    <xsl:variable name="directionList"
                        select="child::*[local-name() = 'directions']" />
                    <xsl:for-each select="$ingredientList">
                        <xsl:variable name="ingItems"
                            select="child::*[local-name() = 'ingredient']" />
                        <xsl:for-each select="$ingItems">
                            <p>    <xsl:value-of select="."/></p>
                        </xsl:for-each>
                    </xsl:for-each>
                    <xsl:for-each select="$directionList">
                        <xsl:variable name="dirItems"
                            select="child::*[local-name() = 'direction']" />
                        <xsl:for-each select="$dirItems">
                            <p><b><xsl:value-of select="position()"/></b>. <xsl:value-of
select="."/></p>
                        </xsl:for-each>
                    </xsl:for-each>
                </td></tr>
            </xsl:for-each>
        </table>
    </xsl:template>

</xsl:stylesheet>
```

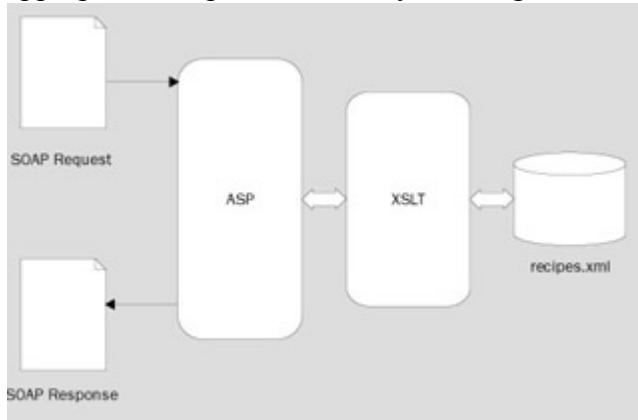
Compared to some stylesheets, this is relatively tame. The stylesheet loads all the <recipe> elements in the source

document into the variable recipeList. Using <xsl:for-each>, we can then walk through the list and convert each <recipe> element into HTML. In the same manner, we access the list of ingredients and directions for each of those recipes and display them as well.

An interesting side effect of all this is that the showrecipes.xsl Stylesheet that we use to display the recipes in HTML can be applied equally to a SOAP response from our Web Service and the entire recipes.xml data store. Previously, we had used XSLT as the engine of our Web Service. Here, it is put to a more conventional use, transforming the response of the Web Service into HTML that can be displayed to the user of the service.

## How We Built It

Now that we have implemented our Web Service, we can reverse-engineer it for documentation and fill in the appropriate components in our system diagram:



In summary, the system we have created for this case study is made up of the following components:

1. An XML document containing a list of recipes
2. An XSLT stylesheet that transforms a SOAP request for recipes into a SOAP response
3. An ASP page that listens for and accepts SOAP requests
4. A JavaScript client that uses the Web Service

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Summary

In this case study, we developed a Web Service that returns a list of possible recipes when provided a list of available ingredients. Using XML, XSLT, and a little script code, we are able to deliver a valuable service to night owls around the world. Of course, this exact example is not something you will be faced with implementing on your next project, but the problem is a common one: build a Web Service around data stored in XML. Most of the XSLT we have written for the Web Service is not specific to this particular service, and it could be used to build other Web Services with just a little modification.

We have taken a different approach to Web Services in this case study, one that relies almost completely on XML. Although we needed script code at both the client and server to build our Web Service application, that code is extraneous and could have been written in any number of languages. The core of our application is written entirely in XML: recipe data is stored in an XML document, XML messages are exchanged between the client and server using SOAP, and XML serves as the logic of our service in the form of XSLT. By following this approach and using standard XML to build applications, you will be able to use more of the capabilities that XML provides.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Appendix A: The XML Document Object Model

This appendix lists all of the interfaces in the DOM Level 2 Core, both the [Fundamental Interfaces](#) and the [Extended Interfaces](#), including all of their properties and methods. Examples of how to use some of these interfaces were given in [Chapter 8](#).

*Further information on these interfaces can be found at:  
<http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210/core.html>*

## Fundamental Interfaces

The DOM Fundamental Interfaces are interfaces that *all* DOM implementations must provide, even if they aren't designed to work with XML documents.

### DOMException

An object implementing the DOMException interface is raised whenever an error occurs in the DOM.

Property	Description
code	An integer, representing which <b>exception code</b> this DOMException is reporting.

The code property can take the following values:

Exception Code	Integer Value	Description
INDEX_SIZE_ERR	1	The index or size is negative, or greater than the allowed value.
DOMSTRING_SIZE_ERR	2	The specified range of text does not fit into a DOMString.
HIERARCHY_REQUEST_ERR	3	The node is inserted somewhere it doesn't belong.
WRONG_DOCUMENT_ERR	4	The node is used in a different document than the one that created it, and that document doesn't support it.
INVALID_CHARACTER_ERR	5	A character has been passed which is not valid in XML.

NO_DATA_ALLOWED_ERR	6	Data has been specified for a node which does not support data.
NO_MODIFICATION_ALLOWED_ERR	7	An attempt has been made to modify an object which doesn't allow modifications.
NOT_FOUND_ERR	8	An attempt was made to reference a node which does not exist.
NOT_SUPPORTED_ERR	9	The implementation does not support the type of object requested.
INUSE_ATTRIBUTE_ERR	10	An attempt was made to add a duplicate attribute.
INVALID_STATE_ERR	11	An attempt was made to use an object which is not, or is no longer, useable.
SYNTAX_ERR	12	An invalid or illegal string was passed.
INVALID_MODIFICATION_ERR	13	An attempt was made to modify the type of the underlying object.
NAMESPACE_ERR	14	An attempt was made to create or change an object in a way which is incompatible with namespaces.
INVALID_ACCESS_ERR	15	A parameter was passed or an operation attempted which is not supported by the underlying object.

## Node

The Node interface is the base interface upon which most of the DOM objects are built, and contains methods and attributes which can be used for all types of nodes. The interface also includes some helper methods and attributes which only apply to particular types of nodes.

Property	Description
nodeName	The name of the node. Will return different values, depending on the nodeType, as listed in the next table.
nodeValue	The value of the node. Will return different values, depending on the nodeType, as listed in the next table.
nodeType	The type of node. Will be one of the values from the next table.
parentNode	The node that is this node's parent.

childNodes	A NodeList containing all of this node's children. If there are no children, an empty NodeList will be returned, not NULL.
firstChild	The first child of this node. If there are no children, this returns NULL.
lastChild	The last child of this node. If there are no children, this returns NULL.
previousSibling	The node immediately preceding this node. If there is no preceding node, this returns NULL.
nextSibling	The node immediately following this node. If there is no following node, this returns NULL.
attributes	A NamedNodeMap containing the attributes of this node. If the node is not an element, this returns NULL.
ownerDocument	The document to which this node belongs.
namespaceURI	The namespace URI of this node. Returns NULL if a namespace is not specified.
prefix	The namespace prefix of this node. Returns NULL if a namespace is not specified.
localName	Returns the local part of this node's QName.

The values of the nodeName and nodeValue properties depend on the value of the nodeType property, which can return one of the following constants:

nodeType <b>property constant</b>	nodeName	nodeValue
ELEMENT_NODE	Tag name	NULL
ATTRIBUTE_NODE	Name of attribute	Value of attribute
TEXT_NODE	#text	Content of the text node
CDATA_SECTION_NODE	#cdata-section	Content of the CDATA section
ENTITY_REFERENCE_NODE	Name of entity referenced	NULL
ENTITY_NODE	Entity name	NULL
PROCESSING_INSTRUCTION_NODE	Target	Entire content excluding the target
COMMENT_NODE	#comment	Content of the comment

DOCUMENT_NODE	#document	NULL
DOCUMENT_TYPE_NODE	Document type name	NULL
DOCUMENT_FRAGMENT_NODE	#document-fragment	NULL
NOTATION_NODE	Notation name	NULL

Method	Description
insertBefore(newChild, refChild)	Inserts the newChild node before the existing refChild. If refChild is NULL, inserts the node at the end of the list. Returns the inserted node.
replaceChild(newChild, oldChild)	Replaces oldChild with newChild. Returns oldChild.
removeChild(oldChild)	Removes oldChild from the list, and returns it.
appendChild(newChild)	Adds newChild to the end of the list, and returns it.
hasChildNodes()	Returns a Boolean; true if the node has any children, false otherwise.
cloneNode(deep)	Returns a duplicate of this node. If the Boolean deep parameter is true, this will recursively clone the sub-tree under the node, otherwise it will only clone the node itself.
normalize()	If there are multiple adjacent Text child nodes (from a previous call to Text.splitText()) this method will combine them again. It doesn't return a value.
supports(feature, version)	Indicates whether this implementation of the DOM supports the feature passed. Returns a Boolean, true if it supports the feature, false otherwise.

## Document

An object implementing the Document interface represents the entire XML document. This object is also used to create other nodes at run-time.

The Document interface extends the Node interface.

Property	Description
doctype	Returns a DocumentType object, indicating the document type associated with this document. If the document has no document type specified, returns NULL.

implementation	The DOMImplementation object used for this document.
documentElement	The root element for this document.
Method	Description
createElement (tagName)	Creates an element, with the name specified.
createDocumentFragment ()	Creates an empty DocumentFragment object.
createTextNode (data)	Creates a Text node, containing the text in data.
createComment (data)	Creates a Comment node, containing the text in data.
createCDATASection (data)	Creates a CDATASection node, containing the text in data.
createProcessingInstruction (target, data)	Creates a ProcessingInstruction node, with the specified target and data.
createAttribute (name)	Creates an attribute, with the specified name.
createEntityReference (name)	Creates an entity reference, with the specified name.
getElementsByTagName (tagname)	Returns a NodeList of all elements in the document with this tagname. The elements are returned in document order.
importNode (importedNode, deep)	Imports a node importedNode from another document into this one. The original node is not removed from the old document, it is just cloned. (The Boolean deep parameter specifies if it is a deep or shallow clone: deep-sub-tree under node is also cloned, shallow-only node itself is cloned.) Returns the new node.
createElementNS (namespaceURI, qualifiedName)	Creates an element, with the specified namespace and QName.
createAttributeNS (namespaceURI, qualifiedName)	Creates an attribute, with the specified namespace and QName.
getElementsByTagNameNS (namespaceURI, localName)	Returns a NodeList of all the elements in the document which have the specified local name, and are in the namespace specified by namespaceURI.
getElementByID (elementID)	Returns the element with the ID specified in elementID. If there is no such element, returns NULL.

Note

All of the createXXX() methods return the node created.

# DOMImplementation

The DOMImplementation interface provides methods which are not specific to any particular document, but to any document from this DOM implementation. You can get a DOMImplementation object from the implementation property of the Document interface.

Method	Description
hasFeature(feature, version)	Returns a Boolean, indicating whether this DOM implementation supports the feature requested. version is the version number of the feature to test.
createDocumentType(qualifiedName, publicID, systemID, internalSubset)	Creates a DocumentType object, with the specified attributes.
createDocument(namespaceURI, qualifiedName, doctype)	Creates a Document object, with the document element specified by qualifiedName. The doctype property must refer to an object of type DocumentType.

# DocumentFragment

A document fragment is a temporary holding place for a group of nodes, usually with the intent of inserting them back into the document at a later point.

The DocumentFragment interface extends the Node interface, without adding any additional properties or methods.

# NodeList

A NodeList contains an ordered group of nodes, accessed via an integral index.

Property	Description
length	The number of nodes contained in this list. The range of valid child node indices is 0 to length - 1 inclusive.
item(index)	Returns the Node in the list at the indicated index. If index is the same as or greater than length, returns NULL.

# Element

Provides properties and methods for working with an element.

The Element interface extends the Node interface.

Property	Description
tagName	The name of the element.
Method	Description

<code>getAttribute (name)</code>	Returns the value of the attribute with the specified name, or an empty string if that attribute does not have a specified or default value.
<code>setAttribute (name, value)</code>	Sets the value of the specified attribute to this new value. If no such attribute exists, a new one with this name is created.
<code>removeAttribute (name)</code>	Removes the specified attribute. If the attribute has a default value, it is immediately replaced with an identical attribute, containing this default value.
<code>getAttributeNode (name)</code>	Returns an Attr node, containing the named attribute. Returns NULL if there is no such attribute.
<code>setAttributeNode (newAttr)</code>	Adds a new attribute node. If an attribute with the same name already exists, it is replaced. If an Attr has been replaced, it is returned, otherwise NULL is returned.
<code>removeAttributeNode (oldAttr)</code>	Removes the specified Attr node, and returns it. If the attribute has a default value, it is immediately replaced with an identical attribute, containing this default value.
<code>getElementsByTagName (name)</code>	Returns a NodeList of all descendants with the given node name.
<code>getAttributeNS (namespaceURI, localName)</code>	Returns the value of the specified attribute, or an empty string if that attribute does not have a specified or default value.
<code>setAttributeNS (namespaceURI, qualifiedName, value)</code>	Sets the value of the specified attribute to this new value. If no such attribute exists, a new one with this namespace URI and QName is created.
<code>removeAttributeNS (namespaceURI, localName)</code>	Removes the specified attribute. If the attribute has a default value, it is immediately replaced with an identical attribute, containing this default value.
<code>getAttributeNodeNS (namespaceURI, localName)</code>	Returns an Attr node, containing the specified attribute. Returns NULL if there is no such attribute.
<code>setAttributeNodeNS (newAttr)</code>	Adds a new Attr node to the list. If an attribute with the same namespace URI and local name exists, it is replaced. If an Attr object is replaced, it is returned, otherwise NULL is returned.
<code>getElementsByTagNameNS (namespaceURI, localName)</code>	Returns a NodeList of all of the elements matching these criteria.

## NamedNodeMap

A named node map represents an unordered collection of nodes, retrieved by name.

Property	Description
<code>length</code>	The number of nodes in the map.

Method	Description
getNamedItem(name)	Returns a Node, where the nodeName is the same as the name specified, or NULL if no such node exists.
setNamedItem(arg)	The arg parameter is a Node object, which is added to the list. The nodeName property is used for the name of the node in this map. If a node with the same name already exists, it is replaced. If a Node is replaced it is returned, otherwise NULL is returned.
removeNamedItem(name)	Removes the Node specified by name, and returns it.
item(index)	Returns the Node at the specified index. If index is the same as or greater than length, returns NULL.

Method	Description
getNamedItemNS(namespaceURI, localName)	Returns a Node, matching the namespace URI and local name, or NULL if no such node exists.
setNamedItemNS(arg)	The arg parameter is a Node object, which is added to the list. If a node with the same namespace URI and local name already exists, it is replaced. If a Node is replaced it is returned, otherwise, NULL is returned.
removeNamedItemNS(namespaceURI, localName)	Removes the specified node, and returns it.

## Attr

Provides properties for dealing with an attribute.

The Attr interface extends the Node interface.

Property	Description
name	The name of the attribute.
specified	A Boolean, indicating whether this attribute was specified (true), or just defaulted (false).
value	The value of the attribute.
ownerElement	An Element object, representing the element to which this attribute belongs.

## CharacterData

Provides properties and methods for working with character data.

The CharacterData interface extends the Node interface.

Property	Description
data	The text in this CharacterData node.
length	The number of characters in the node.
substringData(offset, count)	Returns a portion of the string, starting at the offset. Will return the number of characters specified in count, or until the end of the string, whichever is less.
appendData(arg)	Appends the string in arg to the end of the string.
insertData(offset, arg)	Inserts the string in arg into the middle of the string, starting at the position indicated by offset.
Method	Description
deleteData(offset, count)	Deletes a portion of the string, starting at the offset. Will delete the number of characters specified in count, or until the end of the string, whichever is less.
replaceData(offset, count, arg)	Replaces a portion of the string, starting at the offset. Will replace the number of characters specified in count, or until the end of the string, whichever is less. The arg parameter is the new string to be inserted.

## Text

Provides an additional method for working with text nodes.

The Text interface extends the CharacterData interface.

Method	Description
splitText(offset)	Separates this single Text node into two adjacent Text nodes. All of the text up to the offset point goes into the first Text node, and all of the text starting at the offset point to the end goes into the second Text node.

## Comment

Encapsulates an XML comment.

The Comment interface extends the CharacterData interface, without adding any additional properties or methods.

# Extended Interfaces

The DOM Extended Interfaces need only be provided by DOM implementations that will be working with XML documents.

## CDATASection

Encapsulates an XML CDATA section.

The CDATASection interface extends the Text interface, without adding any additional properties or methods.

## ProcessingInstruction

Provides properties for working with an XML processing instruction (PI).

The ProcessingInstruction interface extends the Node interface.

Property	Description
target	The PI target, in other words the name of the application to which the PI should be passed.
data	The content of the PI.

## DocumentType

Provides properties for working with an XML document type. Can be retrieved from the doctype property of the Document interface. (If a document doesn't have a document type, doctype will return NULL.)

DocumentType extends the Node interface.

Property	Description
name	The name of the DTD (Document Type Definition).
entities	A NamedNodeMap containing all entities declared in the DTD (both internal and external). Parameter entities are not contained, and duplicates are discarded, according to the rules followed by validating XML parsers.
notations	A NamedNodeMap containing the notations contained in the DTD. Duplicates are discarded.
publicID	The public identifier of the external subset.
systemID	The system identifier of the external subset.

internalSubset	The internal subset, as a string.
----------------	-----------------------------------

## Notation

Provides properties for working with an XML notation. Notations are read-only in the DOM.

The Notation interface extends the Node interface.

Property	Description
publicID	The public identifier of this notation. If the public identifier was not specified, returns NULL.
systemID	The system identifier of this notation. If the system identifier was not specified, returns NULL.

## Entity

Provides properties for working with parsed and unparsed entities. Entity nodes are read-only.

The Entity interface extends the Node interface.

Property	Description
publicID	The public identifier associated with the entity, or NULL if none is specified.
systemID	The system identifier associated with the entity, or NULL if none is specified.
notationName	For unparsed entities, the name of the notation for the entity. NULL for parsed entities.

## EntityReference

Encapsulates an XML entity reference.

The EntityReference extends the Node interface, without adding any properties or methods.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Appendix B: XPath Reference

XPath is a W3C Recommendation, describing a syntax for selecting a set of nodes from an XML document. Version 1.0 of XPath reached Recommendation status on 16 November 1999. The requirements document for version 2.0 was a working draft at the time of writing. XPath is an essential part of the XSLT Recommendation.

An XPath location path contains one or more "location steps", separated by forward slashes (/). Each location step has the following form:

```
axis-name::node-test [predicate] *
```

In plain English, this is an axis name, then two colons, then a node test, and finally zero or more predicates each contained in square brackets. A predicate can contain literal values (for example, 4, 'hello'), operators (+, -, =, etc.), and other XPath expressions. XPath also defines a set of functions for use in predicates.

The XPath axis defines a part of the document, from the perspective of the "context node". This node serves as the 'starting point' for selecting the result set of the XPath expression. The node test makes a selection from the nodes on the given axis. By adding predicates, it is possible to select a subset from these nodes. If the expression in the predicate returns true, the node remains in the selected set, otherwise it is removed.

In this reference we will list the XPath axes, node tests and functions. For each entry, we will list whether it is implemented in version 1.0 of the specification. Also, we will list in which versions of the Microsoft implementation the feature was implemented. Other implementations are not listed.

You can find an online version of the reference at: <http://www.vbxml.com/xsl/xpathRef.asp>.

We chose to list details on the several MSXML implementations, because Microsoft has chosen to ship several partial implementations of the specification. Although the latest version (MSXML 3.0) is a complete implementation, in many environments the older implementations are still in use. MSXML 2.0 is the version shipped with Internet Explorer 5, causing it to be the most widely installed XSLT implementation (and sadly one of the most incomplete as well). MSXML 2.6 is a version that was released as a preview, but was shipped with certain versions of BizTalk server. Therefore some developers are forced to work with this version.

Other implementations, such as Xalan and Saxon, are not listed here. You can more or less trust their latest versions to implement version 1.0 of the implementation and you will normally not be forced to use a specific version.

## Axes

Below, I list each axis with a description of the nodes it selects. The primary node type of an axis indicates what kind of nodes are selected by the literal node test or the \* node test (see under the *literal name* node test for an example). For some axes, XPath defines a shorthand syntax. The form of this syntax and its primary node type are listed for every axis.

### **ancestor**

<b>Description:</b>	Contains the context node's parent node, its parent's parent node, etc., all the way up to the document root. If the context node is the root node, the ancestor node is empty.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

## ancestor-or-self

<b>Description:</b>	Identical to the ancestor axis, but including the context node itself.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

## attribute

<b>Description:</b>	Contains all attributes on the context node. The axis will be empty unless the context node is an element.
<b>Primary node type:</b>	Attribute
<b>Shorthand:</b>	@
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview) (erroneously returns namespace declarations as well) MSXML 3.0

## child

<b>Description:</b>	Contains all direct children of the context node (that is the children, but not any of the children's children).
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	Default axis if no axis is given
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5) (only shorthand syntax) MSXML 2.6 (January 2000 preview)MSXML 3.0

## descendant

<b>Description:</b>	All children of the context node, including all children's children recursively.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	//
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5) (only shorthand syntax) MSXML 2.6 (January 2000 preview)MSXML 3.0

## descendant-or-self

<b>Description:</b>	Identical to the descendant axis, but including the context node itself.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

## following

<b>Description:</b>	Contains all nodes that come after the context node in the document order. This means that the opening tag of the node must come after the closing tag of the context node, and therefore excludes the descendants of the context node.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 3.0

## following-sibling

<b>Description:</b>	Contains all siblings (children of the same parent node) of the context node that come after the context node in document order.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 3.0

## namespace

<b>Description:</b>	Contains all namespaces available on the context node. This includes the default namespace and the xml namespace (these are automatically declared in any document). The axis will be empty unless the context node is an element.
<b>Primary node type:</b>	Namespace
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 3.0

## parent

<b>Description:</b>	Contains the direct parent node (and only the direct parent node) of the context node, if there is one.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	..
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE5) (only shorthand syntax) MSXML 2.6 (January 2000 preview)MSXML 3.0

## preceding

<b>Description:</b>	Contains all nodes that come before the context node in the document order. This includes only elements that are already closed (their closing tag comes before the context node in the document), and therefore excludes all ancestors of the context node.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 3.0

## preceding-sibling

<b>Description:</b>	Contains all siblings (children of the same parent node) of the context node that come before the context node in document order.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	None
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 3.0

## self

<b>Description:</b>	Contains only the context node itself.
<b>Primary node type:</b>	Element
<b>Shorthand:</b>	.
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5) (only shorthand syntax) MSXML 2.6 (January 2000 preview)MSXML 3.0

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Node tests

A node test describes a test performed on each node of an axis to decide whether it should be included in the result set. Appending a predicate can later filter this resultset.

\*

<b>Description:</b>	Returns true for all nodes of the primary type for the axis.
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5)MSXML 2.6 (January 2000 preview)MSXML 3.0

## comment()

<b>Description:</b>	Returns true for all comment nodes.
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5)MSXML 2.6 (January 2000 preview)MSXML 3.0

## literal name

<b>Description:</b>	Returns true for all nodes of that name of the primary node type. If the node test is 'PERSON', it returns true for all nodes <PERSON> (if the primary node type is Element).
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5)MSXML 2.6 (January 2000 preview)MSXML 3.0

## node()

<b>Description:</b>	Returns true for all nodes, except attributes and namespaces.
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5)MSXML 2.6 (January 2000 preview)MSXML 3.0

## processing-instruction( name? )

<b>Description:</b>	Returns true for all processing instruction nodes. If a name parameter is passed (the question mark means that it is optional), it returns true only for processing instruction nodes of that name.
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5) (called pi() in MSXML2)MSXML 2.6 (January 2000 preview)MSXML 3.0

## text()

<b>Description:</b>	Returns true for all text nodes.
<b>Implemented:</b>	W3C 1.0 specification (recommendation)MSXML 2.0 (IE 5)MSXML 2.6 (January 2000 preview)MSXML 3.0

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Functions

To filter a subset from the resultset of nodes that was selected with an axis and node test, we can append a predicate within square brackets. The expression within the brackets can use literal values (numbers, strings, etc.), XPath expressions, and a number of functions described by the XPath specification.

Each function is described below by a line of this form:

return-type **function-name** (parameters)

For each parameter, we display the type (object, string, number, node-set) and where necessary a symbol indicating if the parameter is optional (?) or can occur multiple times (+). The type object means that any type can be passed.

If an expression is passed as a parameter, it is first evaluated and (if necessary) converted to the expected type before passing it to the function.

boolean **boolean** ( object )

Converts anything passed to it to a Boolean.

`boolean(attribute::name)` will return true if the context node has a name attribute.

Parameter:

object

Numbers result in true if they are not zero or NaN.

Strings result in true if their length is non-zero.

Node-sets return true if they are non-empty.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

number **ceiling** ( number )

Rounds a passed number to the smallest integer that is not smaller than the passed number.

`ceiling(1.1)` returns 2

Parameter:

number

The number that must be rounded.

Implemented:

W3C 1.0 specification (recommendation)MSXML 3.0

string **concat** ( string1, string2+ )

Concatenates all passed strings to one string.

concat('con', 'c', 'a', 't') returns concat

Parameters:

string1

The first string.

string2

All following strings.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

boolean **contains** ( string1, string2 )

Returns true if string1 contains string2.

contains('Teun Duynstee', 'uy') returns true.

Parameters:

string1

The source string.

string2

The string that must be searched for.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

## number **count** ( node-set )

Returns the number of nodes in the passed node-set.

`count(child::*[@name])` returns the number of child elements of the context node that have a name attribute.

Parameter:

node-set

The node-set that is to be counted.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

## boolean **false** ( )

Always returns false. This may seem useless, but XPath does not define a True and False literal value, so the function can be used to construct expressions like:

`starts-with(@name, 'T') = false()`

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

## number **floor** ( number )

Rounds a passed number to the largest integer that is not larger than the passed number.

`floor(2.9)` returns 2

`floor(-1.1)` returns -2

Parameter:

number

The number that must be rounded.

Implemented:

W3C 1.0 specification (recommendation)MSXML 3.0

## node-set **id** ( string )

Returns the element identified by the passed identifier. Note that this will only work in validated documents, because for non-validated documents the parser has no way of knowing which attributes represent ID values.

Parameter:

string

The ID value.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0

boolean **lang** ( string )

Returns true if the language of the context node is the same as the passed language parameter. The language of the context node can be set using the `xml:lang` attribute on itself or any of its ancestors. This feature of XML isn't used frequently.

`lang('en')` returns true for English language nodes.

Parameter:

string

Language identifier.

Implemented:

W3C 1.0 specification (recommendation)MSXML 3.0

number **last** ( )

Returns the index number of the last node in the current context node-set.

`child::*[position() = last()-1]` selects the penultimate child element of the context node.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.0 (IE5) (called `end()` in MSXML2) MSXML 2.6 (January 2000 preview) (does not work when used on the descendant axis) MSXML 3.0

string **local-name** ( node-set? )

Returns the local part of the name of the first node (in document order) in the passed node-set. For example, the local part of an `<xsl:value-of>` element is `value-of`.

Parameter:

node-set

If no node-set is specified, the current context node is used.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

string **name** ( node-set? )

Returns the name of the passed node. This is the fully qualified name, including namespace prefix.

Parameter:

node-set

If no node-set is specified, the current context node is used.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

string **namespace-uri** ( node-set? )

Returns the full URI that defines the namespace of the passed node.

namespace-uri(@href) in an XHTML document might return '<http://www.w3.org/Profiles/XHTML-transitional>'

Parameter:

node-set

If no node-set is specified, the current context node is used.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

string **normalize-space** ( string? )

Returns the whitespace-normalized version of the passed string. This means that all leading and trailing whitespace gets stripped and all sequences of whitespace get combined to one single space.

normalize-space(' some text ') would return 'some text'

Parameter:

string

If no string is passed, the current node is converted to a string.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

boolean **not** ( boolean )

Returns the inverse of the passed value.

not(@name) returns true if there is no name attribute on the context node.

Parameter:

boolean

An expression that evaluates to a Boolean value.

Implemented

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

number **number** ( object? )

Converts parameter to a number.

number(' -3.6 ') returns the number -3.6

The number() function does not use any localized settings, so you must only use this conversion when the format of the numeric data is language-neutral.

Parameter:

object

If nothing is passed, the current context node is used.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview) MSXML 3.0

number **position** ( )

Returns the position of the current context node in the current context node-set.

position() returns 1 for the first node in the context node-set.

Implemented:

W3C 1.0 specification (recommendation) MSXML 2.6 (January 2000 preview) MSXML 3.0

number **round** ( number )

Rounds a passed number to the nearest integer.

round(1.5) returns 2, round(-1.7) returns -2

Parameter:

number

The number that must be rounded.

Implemented:

W3C 1.0 specification (recommendation) MSXML 3.0

boolean **starts-with** ( string1, string2 )

Returns true if string1 starts with string2.

starts-with(@name, 'T') returns true if the value of the name attribute starts with a capital T.

Parameters:

string1

The string that must be checked.

string2

The substring that must be searched for.

Implemented:

W3C 1.0 specification (recommendation) MSXML 2.6 (January 2000 preview) (somehow this fails to work in the test attribute of an <xslif> element) MSXML 3.0

string **string** ( object? )

Converts the passed object to a string value.

Parameter:

object

If nothing is passed, the result is an empty string.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

number **string-length** ( string? )

Returns the number of characters in the passed string.

string-length('Teun Duynstee') returns 13

Parameter:

string

If nothing is passed, the current context is converted to a string.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

string **substring** ( string, number1, number2? )

Returns the substring from the passed string starting at the number1 character, with the length of number2. If no number2 parameter is passed, the substring runs to the end of the passed string.

substring('Teun Duynstee', 6) returns 'Duynstee'

Parameters:

string

The string that will be used as source for the substring.

number1

Start location of the substring.

number2

Length of the substring.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

**string `substring-after` ( string1, string2 )**

Returns the substring following the first occurrence of string2 inside string1. For example, the return value of `substring-after('2000/3/22', '/')` would be `3/22`.

Parameters:

`string1`

The string that serves as source.

`string2`

The string that is searched in the source string.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

**string `substring-before` ( string1, string2 )**

Returns the string part preceding the first occurrence of the string2 inside the string1. For example, the return value of `substring-before('2000/3/22', '/')` would be `2000`.

Parameters:

`string1`

The string that serves as source.

`string2`

The string that is searched in the source string.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

**number `sum` ( node-set )**

Sums the values of all nodes in the set when converted to number.

`sum(student/@age)` returns the sum of all age attributes on the student elements on the child axis of the context node.

Parameter:

node-set

The node-set containing all values to be summed.

Implemented:

W3C 1.0 specification (recommendation)MSXML 3.0

string **translate** ( string1, string2, string3 )

Translates characters in string1 to other characters. Translation pairs are specified by string2 and string3. For example, translate('A Space Odissei', 'i', 'y') would result in A Space Odyssey, and translate('abcdefg', 'aceg', 'ACE') would result in AbCdEf. The final g gets translated to nothing, because the string3 has no counterpart for that position in the string2.

Parameters:

string1

String to be translated character by character.

string2

String defining which characters must be translated.

string3

String defining what the characters from the string2 should be translated to.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

boolean **true** ( )

Always returns true.

Implemented:

W3C 1.0 specification (recommendation)MSXML 2.6 (January 2000 preview)MSXML 3.0

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Appendix C: XSLT Reference

This reference appendix describes the elements and functions that are part of XSLT. For the XPath functions that can also be used with XSLT, see [Appendix B](#).

The XSLT 1.0 specification became a W3C Recommendation on 16 November 1999. Version 1.1 was a Working Draft at the time of writing, as were the requirements for version 2.0. As in [Appendix B](#), we will describe not only the functionality as described in the specifications, but also in which releases of the MSXML library the feature is implemented, since some older partial implementations of this library can still be found in many environments.

This is not done for the many other implementations, such as Xalan and Saxon. Distributions of XSLT processors will normally come with a description of the conformance to the XSLT 1.0 specification.

Both the attributes on XSLT elements and the parameters of XSLT functions can be of several types. At the end of this appendix, you will find a list of the types used in the elements and functions of XSLT.

You can find an online version of this reference at: <http://www.vbxml.com/xsl/XSLTRef.asp>.

## Elements

The XSLT stylesheet is itself an XML document, using a number of special elements in its own namespace. This namespace is <http://www.w3.org/1999/XSL/Transform>, but in this appendix (and the rest of this book) we simply use the prefix xsl.

For each element we give a short description of its use, describe the attributes that can or must be used on the element, and indicate where in the stylesheet the element can occur (as a child of which other elements).

### **<xsl:apply-imports>**

For calling a template from an imported stylesheet that was overruled in the importing stylesheet. This is normally used if you want to add functionality to a standard template that you imported using `<xsl:import>`.

Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 3.0
Can contain:	No other elements
Can be contained by:	<code>&lt;xsl:attribute&gt;</code> , <code>&lt;xsl:comment&gt;</code> , <code>&lt;xsl:copy&gt;</code> , <code>&lt;xsl:document&gt;</code> , <code>&lt;xsl:element&gt;</code> , <code>&lt;xsl:fallback&gt;</code> , <code>&lt;xsl:for-each&gt;</code> , <code>&lt;xsl:if&gt;</code> , <code>&lt;xsl:message&gt;</code> , <code>&lt;xsl:otherwise&gt;</code> , <code>&lt;xsl:param&gt;</code> , <code>&lt;xsl:processing-instruction&gt;</code> , <code>&lt;xsl:template&gt;</code> , <code>&lt;xsl:variable&gt;</code> , <code>&lt;xsl:when&gt;</code>

### **<xsl:apply-templates>**

Used to pass the context on to another template. The select attribute specifies which nodes should be transformed now; the processor decides which templates will be used.

#### Attributes:

select (optional)	Expression describing which nodes in the source document should be transformed next. Defaults to child::*.	
	Type:	node-set-expression
	Attribute Value Template:	no
mode (optional)	By adding a mode attribute, the processor will transform the indicated source document nodes using only templates with this same mode attribute. This allows us to process the same source node in different ways.	
	Type:	qname
	Attribute Value Template:	no
Implemented	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0	
Can contain	<xsl:sort>, <xsl:with-param>	
Can be contained by	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:attribute>

Generates an attribute in the destination document. It should be used in the context of an element (either a literal, <xsl:element>, or some other element that generates an element in the output). It must occur before any text or element content is generated.

#### Attributes:

name (required)	The name of the attribute.	
	Type:	qname
	Attribute Value Template:	yes
namespace (optional)	The namespace (the default uses the namespace of the element the attribute is placed on).	

	Type:	uri-reference
	Attribute Value Template:	yes
Implemented:	W3C 1.0 specification (recommendation) MSXML 2.0 (IE5) MSXML 2.6 (January 2000 preview) MSXML 3.0	W3C 1.1 specification (working draft)
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:call-template>, <xsl:choose>, <xsl:copy>, <xsl:copy-of>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:attribute-set>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:attribute-set>

Used to define a set of attributes that can then be added to an element as a group by specifying the <xsl:attribute-set> element's name attribute value in the use-attribute-sets attribute on the <xsl:element> element.

### Attributes:

name (required)	Name that can be used to refer to this set of attributes.	
	Type:	qname
	Attribute Value Template:	no
use-attribute-sets (optional)	For including an existing attribute set in this attribute set.	
	Type:	qnames
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 3.0	
Can contain:	<xsl:attribute>	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:call-template>

Used to call a template by name. Causes no context switch (change of context node) as <xsl:apply-templates> and <xsl:for-each> do. The template you call by name will still be processing the same context node as your current template. This element can be used to reuse the same functionality in several templates.

**Attributes:**

<b>name (required)</b>	Name of the template you want to call.	
	Type:	qname
	Attribute Value Template:	no
<b>Implemented:</b>	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
<b>Can contain:</b>	<xsl:with-param>	
<b>Can be contained by:</b>	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

**<xsl:choose>**

For implementing the choose/when/otherwise construct. Compare to Case/Select in Visual Basic or switch in C and Java.

<b>Implemented:</b>	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0
<b>Can contain:</b>	<xsl:otherwise>, <xsl:when>
<b>Can be contained by:</b>	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>

**<xsl:comment>**

For generating a comment node in the destination document.

<b>Implemented:</b>	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0
<b>Can contain:</b>	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:call-template>, <xsl:choose>, <xsl:copy>, <xsl:copy-of>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:text>, <xsl:value-of>, <xsl:variable>
<b>Can be contained by:</b>	<xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:template>, <xsl:variable>, <xsl:when>

## <xsl:copy>

Generates a copy of the context node in the destination document. Does not copy any children or attributes.

Attributes:

use-attribute-sets (optional)	For adding a set of attributes to the copied node.	
	Type:	qnames
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.0 (IE 5) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:copy-of>

Copies a full tree, including attributes and children, to the destination document. If multiple nodes are matched by the select attribute, all of the sub-trees are copied. If you have an XML fragment stored in a variable, <xsl:copy-of> is the handiest element to send the variables content to the output.

Attributes:

select (required)	XPath expression leading to the nodes to be copied.	
	Type:	expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	No other elements	

Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>
----------------------	--

## <xsl:decimal-format>

Declares a decimal format, which controls the interpretation of a format pattern used by the format-number() function. This includes defining the decimal separator and the thousands separator.

Attributes:

name (optional)	The name of the defined format.	
	Type:	qname
	Attribute Value Template:	no
decimal-separator (optional)	The character that will separate the integer part from the fraction part. Default is a dot (.).	
	Type:	char
	Attribute Value Template:	no
grouping-separator (optional)	The character that will separate the grouped numbers in the integer part. Default is a comma (,).	
	Type:	char
	Attribute Value Template:	no
infinity (optional)	The string that should appear if a number equals infinity. Default is the string 'Infinity'.	
	Type:	string
	Attribute Value Template:	no
minus-sign (optional)	The character that will be used to indicate a negative number. Default is minus (-).	
	Type:	char
	Attribute Value Template:	no
NaN (optional)	The string that should appear if a number is Not a Number. Default is the string 'NaN'.	

	Type:	string
	Attribute Value Template:	no
percent (optional)	Character that will be used as the percent sign. Default is %.	
	Type:	char
	Attribute Value Template:	no
per-mille (optional)	Character that will be used as the per-thousand sign. Default is the Unicode character #x2030, which looks like ?.	
	Type:	char
	Attribute Value Template:	no
zero-digit (optional)	The character used as the digit zero. Default is 0.	
	Type:	char
	Attribute Value Template:	no
digit (optional)	The character used in a pattern to indicate the place where a leading zero is required. Default is 0.	
	Type:	char
	Attribute Value Template:	no
pattern-separator (optional)	The character that is used to separate the negative and positive patterns (if they are different). Default is semicolon (;).	
	Type:	char
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:document>

Switches the target of the result tree to another document. All output nodes instantiated within the <xsl:document> element will appear in the document indicated by the href attribute. All other attributes are identical to the attributes on <xsl:output>. Note that this element is not part of the XSLT 1.0 specification; it is an XSLT 1.1 extension.

#### Attributes:

method (optional)	xml is default.  html will create empty elements like   and use HTML entities like &agrave;;  text will cause no output escaping to happen at all (no entity references in output).	
	Type:	xml html text qname-but-not-ncname
	Attribute Value Template:	yes
version (optional)	The version number that will appear in the XML declaration of the output document.	
	Type:	token
	Attribute Value Template:	yes
encoding (optional)	The encoding of the output document.	
	Type:	string
	Attribute Value Template:	yes
omit-xml-declaration (optional)	Specifies if the resulting document should contain an XML declaration (<?xml version="1.0"?>).	
	Type:	yes no
	Attribute Value Template:	yes
standalone (optional)	Specifies whether the XSLT processor should output a standalone document declaration.	
	Type:	yes no
	Attribute Value Template:	yes
doctype-public (optional)	Specifies the public identifier to be used in the DTD.	
	Type:	string

	Attribute Value Template:	yes
doctype-system (optional)	Specifies the system identifier to be used in the DTD.	
	Type:	string
	Attribute Value Template:	yes
cdata-section-elements (optional)	Specifies a list of elements that should have their content escaped by using a CDATA section instead of entities.	
	Type:	qnames
	Attribute Value Template:	yes
indent (optional)	Specifies to addition of extra whitespace for readability.	
	Type:	yes no
	Attribute Value Template:	yes
media-type (optional)	To specify a specific MIME type while writing out content.	
	Type:	string
	Attribute Value Template:	yes
Implemented:	W3C 1.1 specification (working draft)	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:element>

Generates an element with the specified name in the destination document.	
Attributes:	
name (required)	Name of the element (this may include a prefix bound to a namespace in the stylesheet).

	Type:	qname
	Attribute Value Template:	yes
namespace (optional)	To overrule the namespace that follows from the prefix in the name attribute (if any).	
	Type:	uri-reference
	Attribute Value Template:	yes
use-attribute-sets (optional)	To add a predefined set of attributes to the element.	
	Type:	qnames
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:fallback>

Can be used to specify actions to be executed if the action of its parent element is not supported by the processor.

Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>

## <xsl:for-each>

For looping through the node selected by the XPath expression in the select attribute. The context is shifted to the current node in the loop.

#### Attributes:

select (required)	Expression that selects the nodes to loop through.	
	Type:	node-set-expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.0 (I E5) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:sort>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:if>

Executes the contained elements only if the test expression returns true (or a filled node-set).

#### Attributes:

test (required)	The expression that is tested. If it returns true or a non-empty node-set, the content of the <xsl:if> element is executed.	
	Type:	boolean-expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.0 (I E5) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:sort>, <xsl:text>, <xsl:value-of>, <xsl:variable>	

Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>
----------------------	--

## <xsl:import>

Imports the templates from an external stylesheet document into the current document. The priority of these imported templates is very low, so if a template in the importing document is implemented for the same pattern, it will always prevail over the imported template. The imported template can be called from the overriding template using <xsl:apply-imports>.

Attributes:

href (required)	Reference to the stylesheet to be imported.	
	Type:	uri-reference
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
Can contain:	No other elements	
Can be contained by	<xslstylesheet>, <xslttransform>	

## <xsl:include>

Includes templates from an external document as if they where part of the importing document. This means that templates from the included stylesheet have the same priority as they would have had if they were part of the including stylesheet. An error occurs if a template with the same match and priority attributes exists in both the including and included stylesheets.

Attributes:

href (required)	Reference to the stylesheet to be imported.	
	Type:	uri-reference
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.6 (January 2000 preview)MSXML 3.0	
Can contain:	No other elements	

Can be contained by:	<xslstylesheet>, <xslttransform>
----------------------	----------------------------------

## <xsl:key>

Can be used to create index-like structures that can be queried from the key() function. It is basically a way to describe name/value pairs inside the source document (like a Dictionary object in VB, a Hashtable in Java, or an associative array in Perl). However, in XSLT, more than one value can be found for one key and the same value can be accessed by multiple keys.

Attributes:

name (required)	The name that can be used to refer to this key.	
	Type:	qname
	Attribute Value Template:	no
match (required)	The pattern defines which nodes in the source document can be accessed using this key. In the name/value pair analogy, this would be the definition of the value.	
	Type:	pattern
	Attribute Value Template:	no
use (required)	This expression defines what the key for accessing each value would be. Example: if an element PERSON is matched by the match attribute and the use attribute equals "@name", the key() function can be used to find this specific PERSON element by passing the value of its name attribute.	
	Type:	expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xslstylesheet>, <xslttransform>	

## <xsl:message>

To issue error messages or warnings. The content of the element is the message. What the XSLT processor does with the message depends on the implementation. You could think of displaying it within a message box or logging to the error log.

Attributes:

terminate (optional)	If terminate is set to yes, the execution of the transformation is stopped after issuing the message.	
	Type:	yes no
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:namespace-alias>

Used to make a certain namespace appear in the destination document without using that namespace in the stylesheet. The main use of this element is in generating new XSLT stylesheets.

### Attributes:

stylesheet-prefix (required)	The prefix for the namespace that is used in the stylesheet.	
	Type:	prefix #default
	Attribute Value Template:	no
result-prefix (required)	The prefix for the namespace that must replace the aliased namespace in the destination document.	
	Type:	prefix #default
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:number>

For outputting the number of a paragraph or chapter in a specified format. It has very flexible features, to allow for different numbering rules.

### Attributes:

level (optional)	<p>The value <b>single</b> counts the location of the nearest node matched by the count attribute (along the ancestor axis) relative to its preceding siblings of the same name. Typical output: chapter number.</p> <p>The value <b>multiple</b> will count the location of all the nodes matched by the count attribute (along the ancestor axis) relative to their preceding siblings of the same name. Typical output: paragraph number of form 4.5.3.</p> <p>The value <b>any</b> will count the location of the nearest node matched by the count attribute (along the ancestor axis) relative to their preceding nodes (not only siblings) of the same name. Typical output: bookmark number</p>	
	Type:	single multiple any
	Attribute Value Template:	no
count (optional)	Specifies the type of node that is to be counted.	
	Type:	pattern
	Attribute Value Template:	no
from (optional)	Specifies the starting point for counting.	
	Type:	pattern
	Attribute Value Template:	no
value (optional)	Used to specify the numeric value directly instead of using 'level', 'count' and 'from'.	
	Type:	number-expression
	Attribute Value Template:	no
format (optional)	How to format the numeric value to a string (1 becomes 1, 2, 3, ...; a becomes a, b, c,).	
	Type:	string
	Attribute Value Template:	yes

lang (optional)	Language used for alphabetic numbering	
	Type:	token
	Attribute Value Template:	yes
letter-value (optional)	Some languages have traditional orders of letters specifically for numbering. These orders are often different from the alphabetic order.	
	Type:	alphabetic traditional
	Attribute Value Template:	yes
grouping-separator (optional)	Character to be used for group separation.	
	Type:	char
	Attribute Value Template:	yes
grouping-size (optional)	Number of digits to be separated. grouping-separator=";" and grouping-size="3" causes: 1;000;000.	
	Type:	number
	Attribute Value Template:	yes
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:otherwise>

Content is executed if none of the <xsl:when> elements in an <xsl:choose> is matched.	
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>

Can be contained by:	<xsl:choose>
----------------------	--------------

## <xsl:output>

Top level element for setting properties regarding the output style of the destination document. The <xsl:output> element basically describes how the translation from a created XML tree to a character array (string) happens.

### Attributes:

method (optional)	xml is default  html will create empty elements like   and use HTML entities like &agrave;;  text will cause no output escaping to happen at all (no entity references in output.)	
	Type:	xml html text qname-but-not-ncname
	Attribute Value Template:	no
version (optional)	The version number that will appear in the XML declaration of the output document.	
	Type:	token
	Attribute Value Template:	no
encoding (optional)	The encoding of the output document.	
	Type:	string
	Attribute Value Template:	no
omit-xml-declaration (optional)	Specifies if the resulting document should contain an XML declaration (<?xml version="1.0"?>)	
	Type:	yes no
	Attribute Value Template:	no
standalone (optional)	Specifies whether the XSLT processor should output a standalone document declaration.	
	Type:	yes no
	Attribute Value Template:	no
doctype-public (optional)	Specifies the public identifier to be used in the DTD	

	Type:	string
	Attribute Value Template:	no
doctype-system (optional)	Specifies the system identifier to be used in the DTD	
	Type:	string
	Attribute Value Template:	no
cdata-section-elements (optional)	Specifies a list of elements that should have their content escaped by using a CDATA section instead of entities.	
	Type:	qnames
	Attribute Value Template:	no
indent (optional)	Specifies the addition of extra whitespace for readability	
	Type:	yes no
	Attribute Value Template:	no
media-type (optional)	To specify a specific MIME type while writing out content.	
	Type:	string
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) (No support for methods html and text) MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:param>

Defines a parameter in a <xsl:template> or <xsl:stylesheet>.

Attributes:

name (required)	Name of the parameter	
	Type:	qname

	Attribute Value Template:	no
select (optional)	Specifies the default value for the parameter	
	Type:	expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:preserve-space>

Allows you to define which elements in the source document should have their whitespace content preserved. See also <xsl:strip-space>.

### Attributes:

elements (required)	In this attribute you can list the elements (separated by whitespace) for which you want to preserve the whitespace content.	
	Type:	tokens
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:processing-instruction>

Generates a processing instruction in the destination document.

### Attributes:

name (required)	The name of the processing instruction (the part between the first question mark and the first whitespace of the processing instruction)
-----------------	--

	Type:	ncname
	Attribute Value Template:	yes
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5) (Caution: the <xsl:processing-instruction> element is called <xslpi> in IE5) MSXML 2.6 (January 2000 preview)MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:call-template>, <xsl:choose>, <xsl:copy>, <xsl:copy-of>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:sort>

Allows specifying a sort order for <xsl:apply-templates> and <xsl:for-each> elements. Multiple <xsl:sort> elements can be specified for primary and secondary sorting keys.

### Attributes:

select (optional)	Expression that indicates which should be used for the ordering.	
	Type:	string-expression
	Attribute Value Template:	no
lang (optional)	To set the language used while ordering (in different languages the rules for alphabetic ordering can be different).	
	Type:	token
	Attribute Value Template:	yes
data-type (optional)	To specify alphabetic or numeric ordering.	
	Type:	text number qname-but-not-ncname
	Attribute Value Template:	yes
order (optional)	Specifies ascending or descending ordering.	
	Type:	ascending descending
	Attribute Value Template:	yes

case-order (optional)	Specifies if uppercase characters should order before or after lowercase characters. Note that case insensitive sorting is not supported.	
	Type:	upper-first lower-first
	Attribute Value Template:	yes
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:apply-templates>, <xsl:for-each>	

## <xsl:strip-space>

Allows you to define which elements in the source document should have their whitespace content stripped. See also <xsl:preserve-space>.

### Attributes:

elements (required)	Specify which elements should preserve their whitespace contents.	
	Type:	tokens
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:stylesheet>

The root element for a stylesheet. Synonym to <xsl:transform>.

### Attributes:

id (optional)	A reference for the stylesheet.	
	Type:	id
	Attribute Value Template:	no

extension-element-prefixes (optional)	Allows you to specify which namespace prefixes are XSLT extension namespaces (like msxml).	
	Type:	Tokens
	Attribute Value Template:	no
exclude-result-prefixes (optional)	Namespaces that are only relevant in the stylesheet or in the source document, but not in the result document, can be removed from the output by specifying them here.	
	Type:	tokens
	Attribute Value Template:	no
version (required)	Version number	
	Type:	number
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0	
Can contain:	<xsl:attribute-set>, <xsl:decimal-format>, <xsl:import>, <xsl:include>, <xsl:key>, <xsl:namespace-alias>, <xsl:output>, <xsl:param>, <xsl:preserve-space>, <xsl:strip-space>, <xsl:template>, <xsl:variable>	
Can be contained by:	No other elements	

## <xsl:template>

Defines a transformation rule. Some templates are built-in and don't have to be defined. Refer to [Chapter 3](#) for more information about writing templates.

Attributes:

match (optional)	Defines the set of nodes on which the template can be applied.	
	Type:	pattern
	Attribute Value Template:	no
name (optional)	Name to identify the template when calling it using <xsl:call-template>.	
	Type:	qname

	Attribute Value Template:	no
priority (optional)	If several templates can be applied (through their match attributes) on a node, the priority attribute can be used to make a certain template prevail over others.	
	Type:	number
	Attribute Value Template:	no
mode (optional)	If a mode attribute is present on a template, the template will only be considered for transforming a node when the transformation was started by an <xsl:apply-templates> element with a mode attribute with the same value.	
	Type:	qname
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.0 (IE5) MSXML 2.6 (January 2000 preview) (except for the mode attribute) MSXML 3.0	
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:stylesheet>, <xsl:transform>	

## <xsl:text>

Generates a text string from its content. Whitespace is never stripped from an <xsl:text> element.		
Attributes:		
disable-output-escaping (optional)	If set to yes, the output will not be escaped: this means that a string "<" will be written to the output as "<" instead of "&lt;". This means that the result document will not be a well-formed XML document anymore.	
	Type:	yes no
	Attribute Value Template:	no
Implemented	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.0 (IE5) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	No other elements	

Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>
----------------------	--

## <xsl:transform>

Identical to <xsl:stylesheet>		
Attributes:		
id (optional)	A reference for the stylesheet.	
	Type:	id
	Attribute Value Template:	no
extension-element-prefixes (optional)	Allows you to specify which namespace prefixes are XSLT extension namespaces (like msxml).	
	Type:	tokens
	Attribute Value Template:	no
exclude-result-prefixes (optional)	Namespaces that are only relevant in the stylesheet or in the source document, but not in the result document, can be removed from the output by specifying them here.	
	Type:	tokens
	Attribute Value Template:	no
version (required)	Version number	
	Type:	number
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0	
Can contain:	<xsl:attribute-set>, <xsl:decimal-format>, <xsl:import>, <xsl:include>, <xsl:key>, <xsl:namespace-alias>, <xsl:output>, <xsl:param>, <xsl:preserve-space>, <xsl:strip-space>, <xsl:template>, <xsl:variable>	
Can be contained by:	No other elements	

## <xsl:value-of>

Generates a text string with the value of the expression in the select attribute.		
Attributes:		
select (required)	Expression that selects the node-set that will be converted to a string	
	Type:	string-expression
	Attribute Value Template:	no
disable-output-escaping (optional)	You can use this to output < instead of &lt; to the destination document. Note that this will cause your destination to become invalid XML. Normally used to generate HTML or text files.	
	Type:	yes no
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 2.0 (IE5)MSXML 2.6 (January 2000 preview)MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:template>, <xsl:variable>, <xsl:when>	

## <xsl:variable>

Defines a variable with a value. Note that in XSLT, the value of a variable cannot change?you can instantiate a variable using <xsl:variable>, but it cannot be changed afterwards. Refer to [Chapter 4](#) for more information on the use of variables.

Attributes:		
name (required)	Name of the variable	
	Type:	qname
	Attribute Value Template:	no
select (optional)	Value of the variable (if the select attribute is omitted, the content of the <xsl:variable> element is the value).	
	Type:	expression

	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) MSXML 2.6 (January 2000 preview)	W3C 1.1 specification (working draft) MSXML 3.0
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:attribute>, <xsl:comment>, <xsl:copy>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:otherwise>, <xsl:param>, <xsl:processing-instruction>, <xsl:stylesheet>, <xsl:template>, <xsl:transform>, <xsl:variable>, <xsl:when>	

## <xsl:when>

Represents one of the options for execution in a <xsl:choose> block.		
Attributes:		
test (required)	Expression to be tested.	
	Type:	boolean-expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) MSXML 2.0 (I E5)	W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0
Can contain:	<xsl:apply-imports>, <xsl:apply-templates>, <xsl:attribute>, <xsl:call-template>, <xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:document>, <xsl:element>, <xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:message>, <xsl:number>, <xsl:processing-instruction>, <xsl:text>, <xsl:value-of>, <xsl:variable>	
Can be contained by:	<xsl:choose>	

## <xsl:with-param>

Used to pass a parameter to a template using <xsl:apply-templates> or <xsl:call-template>. The template called must have a parameter of the same name defined using <xsl:param>.		
Attributes:		
name (required)	Name of the parameter.	
	Type:	qname

	Attribute Value Template:	no
select (optional)	XPath expression selecting the passed value.	
	Type:	expression
	Attribute Value Template:	no
Implemented:	W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0	
Can contain:	No other elements	
Can be contained by:	<xsl:apply-templates>, <xsl:call-template>	

[!\[\]\(3364eb1c40a3eed1f04e60d23fb9c78b\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(ad6d527f700f6f15a226df7a3ca27cea\_img.jpg\) NEXT](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Functions

Within expressions in an XSLT stylesheet, you can use all the XPath functions we saw in [Appendix A](#) and also a number of special XSLT functions. These functions are described here.

Each function is described by a line of this form:

return-type **function-name** (parameters)

For each parameter, we display the type (object, string, number, node-set) and where necessary a symbol indicating if the parameter is optional (?) or can occur multiple times (+). The type object means that any type can be passed.

If an expression is passed as a parameter, it is first evaluated and (if necessary) converted to the expected type before passing it to the function.

**node-set current ()**

Returns the current context node-set, outside the current expression. For MSXML2 you can use the context() function as a workaround. context(-1) is synonymous to current()

Implemented

W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 2.6 (January 2000 preview) MSXML 3.0

**node-set document ( object, node-set? )**

To get a reference to an external source document.

Parameters:

object

If of type String, this is the URL of the document to be retrieved. If a node-set, all nodes are converted to strings and all these URLs are retrieved in a node-set.

node-set

Represents the base URL from where relative URLs are resolved.

Implemented:

W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 3.0

**boolean element-available ( string )**

To query availability of a certain extension element.

Parameters:

string

Name of the extension element.

Implemented:

W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0

string **format-number** ( number, string1, string2? )

Formats a numeric value into a formatted and localized string.

Parameters:

number

The numeric value to be represented.

string1

The format string that should be used for the formatting.

string2

Reference to a <xsl:decimal-format> element to indicate localization parameters.

Implemented:

W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0

boolean **function-available** ( string )

To query availability of a certain extension function.

Parameter:

string

Name of the extension function.

Implemented:

W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0

node-set **generate-id** ( node-set? )

Generates a unique identifier for the specified node. Each node will cause a different ID, but the same node will always generate the same ID. You cannot be sure that the IDs generated for a document during multiple transformations will remain identical.

Parameter:

node-set

The first node of the passed node-set is used. If no node-set is passed, the current context is used.

Implemented:

W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0

node-set **key** ( string, object )

To get a reference to a node using the specified <xsl:key>.

Parameters:

string

The name of the referenced <xsl:key> .

object

If of type String, this is the index string for the key. If of type node-set, all nodes are converted to strings and all are used to get nodes back from the key.

Implemented:

W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0

object **system-property** ( string )

To get certain system properties from the processor.

Parameter:

string

The name of the system property. Properties that are always available are xslversion, xslvendor and xslvendor-url.

Implemented:

W3C 1.0 specification (recommendation)W3C 1.1 specification (working draft)MSXML 3.0

node-set **unparsed-entity-uri** ( string )

Returns the URI of the unparsed entity with the passed name.

Parameter:

string

Name of the unparsed entity.

Implemented:

W3C 1.0 specification (recommendation) W3C 1.1 specification (working draft) MSXML 3.0

## Inherited XPath Functions

Check [Appendix A](#) for information on the XPath functions. They can all be used in XSLT:

boolean()	ceiling()	concat()	contains()
count()	false()	floor()	id()
lang()	last	local-name()	name()
namespace-uri()	normalize-space()	not	number()
position()	round()	starts-with()	string()
string-length()	substring()	substring-after()	substring-before()
sum()	translate()	true()	

## Types

These types are used to specify the types of the attributes for the XSLT elements given in the tables above.

boolean	Can have values true and false.
char	A single character.
expression	A string value, containing an XPath expression.
id	A string value. Must be an XML name. The string value can be used only once as an id in any document.
language-name	A string containing one of the defined language identifiers. American English = EN-US.
name	A string value that conforms to the name conventions of XML. That means: no whitespace, should start with either a letter or an underscore (_).
names	Multiple name values separated by whitespace.
namespace-prefix	Any string that is defined as a prefix for a namespace.

ncname	A name value that does not contain a colon.
node	A node in an XML document. Can be of several types, including: element, attribute, comment, processing instruction, text node, etc.
node-set	A set of nodes in a specific order. Can be of any length.
node-set-expression	A string value, containing an XPath expression that returns nodes.
number	A numeric value. Can be either floating point or integer
object	Anything. Can be a string, a node, a node-set, anything
qname	Qualified name: the full name of a node. Made up of two parts: the local name and the namespace identifier.
qnames	A set of qname values, separated by whitespace.
string	A string value
token	A string value that contains no whitespace.
tokens	Multiple token values separated by whitespace.
uri-reference	Any string that conforms to the URI specification.

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Appendix D: Schema Element and Attribute Reference

In this appendix, we provide a full listing of all the elements that form part of the XML Schema Structures Recommendation (found at <http://www.w3.org/TR/xmlschema-1/>). The elements are given in alphabetical order, and each element is described, an example or two is given, followed by a table detailing all of those attributes that element can carry. Required attributes are denoted by bold text.

Note that the constraining facets are not included here, but in the next appendix, together with a listing of all the built-in datatypes to which they apply.

At the end of this appendix, we present a table of the XML Schema attributes that can be used in instance documents, and are in the XML Schema Instance namespace.

## all

The elements contained within this element, are allowed to appear in any order within the instance documents. The all element may be declared within a complexType or a group. It can contain element or annotation elements. Note that when using minOccurs and maxOccurs on element declarations within an all element, you cannot set a multiplicity higher than 1 with maxOccurs, although you can make an element optional with minOccurs. These limitations do not apply to these attributes when they are applied to the all element itself.

## Example

```
<xs:element name = "Rucksack">
  <xs:complexType>
    <xs:all>
      <xs:element name = "Sunglasses" type = "xs:string"
        minOccurs = "0" maxOccurs = "1" />
      <xs:element name = "Sweater" type = "xs:string" />
        minOccurs = "0" maxOccurs = "1" />
      <xs:element name = "Book" type = "xs:string" />
      <xs:element name = "Lunchbox" type = "xs:string" />
      <xs:element name = "Flask" type = "xs:string" />
    </xs:all>
  </xs:complexType>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.

maxOccurs	nonNegativeInteger or unbounded	The maximum number of times the model group can occur.
minOccurs	nonNegativeInteger	The minimum number of times the model group can occur.

**For more information:** see §3.8.2 of the Recommendation.

## annotation

The annotation element is used to provide informative/explanatory data to be read by machines or humans. It may contain appinfo, or documentation elements, which are used to contain instructions for the processing application and schema documentation comments respectively. It is contained by most elements (excluding itself); specific cases are detailed below.

## Example

An example of using annotation with documentation:

```
<xs:element name = "Person">
  <xs:annotation>
    <xs:documentation>
      Used to contain personal information. Note that the last name
      is mandatory, while the first name is optional.
    </xs:documentation>
  </xs:annotation>
  <!-- definition of Person element goes here -->
</xs:element>
```

An example of using annotation with appinfo:

```
<xs:element name="purchaseOrder" type="PurchaseOrderType">
  <xs:annotation>
    <xs:appinfo>
      <sch:pattern name="Top Level Purchase Order elements">
        <sch:rule context="/">
          <sch:assert test="self::purchaseOrder">
            The root element must be a "purchaseOrder"
          </sch:assert>
        </sch:rule>
      </sch:pattern>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

In this second example, the annotation element is used to contain a Schematron schema inside the appinfo element.

**For more information:** see §3.13.2 of the Recommendation.

## any

This is a wildcard element that acts as a placeholder for any element in a model group; the any element is never the direct child of an element element. The content may be validated against another namespace if desired. This is useful, for instance, if unspecified XHTML, or MathML content may be included within the instance document. It may

contain an annotation element, and can be contained by choice or sequence.

## Example

```
<xs:element name = "XHTMLSection">
  <xs:complexType>
    <xs:sequence>
      <xs:any namespace = "http://www.w3.org/1999/xhtml"
        minOccurs = "0" maxOccurs = "unbounded"
        processContents = "lax" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Here, an XHTMLSection element in an instance document can contain any well-formed markup that is valid in the XHTML namespace.

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
maxOccurs	nonNegativeInteger or unbounded	The maximum number of times the model group can occur.
minOccurs	nonNegativeInteger	The minimum number of times the model group can occur.
namespace	##any   ##other   List of (anyURI   ##targetNamespace   ##local)	##any means that the content can be of any namespace. ##other refers to any namespace other than the target namespace of the schema. Otherwise, a whitespace separated list of the namespaces of allowed elements, which can include ##targetNamespace to allow elements in the target namespace of the schema and ##local to allow elements in no namespace. The default is ##any.
processContents	lax   skip   strict	If lax, validation is performed if possible. If skip, then no validation occurs. If strict, validation is enforced, and the validator needs to be able to find declarations for the elements used. The default is skip.

**For more information:** see §3.10.2 of the Recommendation.

## anyAttribute

Like any, this element is a wildcard element, only this allows unspecified attributes to be present. Again, these can be validated against a specific namespace. For example, XML Schema allows elements to have any attributes as long as they're not in the XML Schema namespace or no namespace. You might find this useful to allow the use of any XLink attribute on an element. Can be contained by attributeGroup, complexType, extension, or restriction, and like most

elements it can contain an annotation.

## Example

```
<xs:element name = "SomeElement">
  <xs:complexType>
    <!-- content definition goes here-->
    <xs:anyAttribute namespace = "http://www.w3.org/1999/xlink" />
  </xs:complexType>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
namespace	##any   ##other   List of (anyURI   ##targetNamespace   ##local) "	The same rules and defaults apply as for the any element, described above.
processContents	skip   lax   strict	The same rules and defaults apply as for the any element, described above.

For more information: see §3.4.2 of the Recommendation.

## appinfo

This allows information to be supplied to an application reading the schema, perhaps containing unique identifiers, or additional tags to help an application perform further processing on the schema. It is always used inside the annotation element, as described above.

## Example

```
<xs:element name="purchaseOrder" type="PurchaseOrderType">
  <xs:annotation>
    <xs:appinfo>
      <sch:pattern name="Top Level Purchase Order elements">
        <sch:rule context="/">
          <sch:assert test="self::purchaseOrder">
            The root element must be a "purchaseOrder"
          </sch:assert>
        </sch:rule>
      </sch:pattern>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
-----------	-------------	-------------

source	anyURI	Specifies a URI where the parser can acquire the required appinfo content.
--------	--------	--

**For more information:** see §3.13.2 of the Recommendation.

## attribute

This is used to declare attributes and/or indicate the presence of an attribute. It is usually found within an attributeGroup or a complexType and so defines the attributes for that particular content model. It can also be used in an extension or restriction element, however, when deriving a new type, or inside the root schema element to create global attribute definitions that can be referenced from other declarations. The attribute element may contain an annotation. It may also contain an anonymous simpleType declaration, if there's no type attribute.

## Example

```

<xs:attribute name = "Amount">
  <xs:simpleType name="positiveDecimalN.2" >
    <xs:restriction base="xs:decimal" >
      <xs:minInclusive value="0" />
      <xs:fractionDigits value="2" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:element name = "Payment">
  <xs:complexType >
    <xs:attribute ref = "Amount" />
    <xs:attribute name = "currency" type = "xs:string" default = "US$"
                  use = "optional" />
  </xs:complexType>
</xs:element>

```

## Attributes

Attribute	Value Space	Description
default	string	A string containing the value of the attribute, if the attribute has not been specified in an instance document.
fixed	string	If present, the value of the attribute in an instance document must always match the value specified by fixed.
form	qualified unqualified	If qualified, then the attribute must be namespace qualified in the instance document. Note that if the form attribute is present on the attribute element then it overrides attributeFormDefault on the schema element.
id	ID	Gives a unique identifier to the element.
name	NCName	The name of the attribute, conforming to the XML NCNAME datatype.

ref	QName	Specifies a previously defined attribute name, allows us to inherit its properties.
type	QName	The datatype of the attribute.
use	optional   prohibited   required	If optional, then the attribute may be omitted in the instance document. If required, it must be included, and if prohibited, it cannot be included. The default is optional.

**For more information:** see §3.3.2 of the Recommendation.

## attributeGroup

This allows us to specify and refer to a group of attribute definitions for multiple use—that is, when we want more than one element to carry the same group of elements. It may contain annotation, attribute, attributeGroup, and anyAttribute. There are two ways that attributeGroup (and group) elements are used. Initially, we can define the group at the top level of the schema (as a child of the schema element), and secondly, we can reference that definition from within a complexType definition. Attribute group definitions can be nested, so attributeGroup can contain or be contained by another attributeGroup. It can also be referenced from within a redefine, extension, or restriction when creating a new type.

## Example

```

<xs:attributeGroup name = "myAttrGroup">
    <xs:attribute name = "weight" type = "xs:decimal" use = "optional" />
    <xs:attribute name = "height" type = "xs:decimal" use = "optional" />
</xs:attributeGroup>

<xs:element name = "Person">
    <xs:complexType>
        <xs:sequence>
            <!-- element content here -->
        </xs:sequence>
        <xs:attributeGroup ref = "myAttrGroup" />
    </xs:complexType>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
name	NCName	The name of this attribute group.
ref	QName	Reference to another attribute group; used when referring back to a previously defined group.

**For more information:** see §3.6.2 of the Recommendation.

## choice

When the choice element is used, we can specify that the contents specified within this tag are *mutually exclusive*. That is, one and only one of its immediate children can appear in the instance document. It may contain annotation, element, group, choice, sequence, or any elements, so we can nest content models. Similarly, it can be contained by choice, complexType, group, or sequence.

## Example

```
<xs:element name = "IceCream">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
        <xs:element name = "Strawberry" type = "xs:string" />
        <xs:element name = "Chocolate" type = "xs:string" />
      </xs:choice>
      <xs:choice>
        <xs:element name = "Cone" type = "xs:string" />
        <xs:element name = "Tub" type = "xs:string" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
maxOccurs	nonNegativeInteger or unbounded	The maximum number of times the model group can occur.
minOccurs	nonNegativeInteger	The minimum number of times the model group can occur.

For more information: see §3.8.2 of the Recommendation.

## complexContent

This element is used to extend or restrict complex types. It indicates that the resulting content model can carry attributes, and will contain element content or mixed content, or even be empty. This element is used inside a complexType, and can contain annotation, restriction, or extension.

## Example

```
<xs:complexType name="CAN_Address">
  <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="Province" type="xs:string" />
        <xs:element name="PostalCode" type="CAN_PostalCode"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
```

```
</xs:complexContent>
</xs:complexType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
mixed	boolean	If true, then the content is specified as being mixed. The default is false.

**For more information:** see §3.4.2 of the Recommendation.

## complexType

This specifies that the type is complex type—that is, it can have element content and/or carry attributes. Complex type definitions are the key to the creation of complex structures and content models in XML Schema. Anything that is more complex than simple character data is essentially a complex type. They are usually declared within an element element, or globally within the schema element, but they can also be used from within redefine. A complexType has an optional annotation element. It may be derived from another type, in which case it contains a simpleContent or complexContent element. Alternatively, it can specify a model group directly using group, all, choice, or sequence, followed by an attribute declaration using attribute, attributeGroup, or anyAttribute element.

## Example

```
<xs:element name = "ResearchPaper">
  <xs:complexType mixed = "true">
    <xs:sequence>
      <xs:element name = "Hypothesis" type = "xs:string" />
      <xs:element name = "Conclusion" type = "ConclusionType" />
    </xs:sequence>
    <xs:attribute name = "paperID" type = "xs:integer" />
  </xs:complexType>
</xs:element>
<xs:complexType name = "ConclusionType" block = "#all">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name = "accepted" type = "xs:boolean" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

## Attributes

Attribute	Value Space	Description
abstract	boolean	This specifies whether the complex type can be used to validate an element. If abstract is true, then it can't—you have to derive other types from it for it to be useful. Note that this behavior is distinct from using the abstract attribute on the element element (see below). The default is false.

block	#all   List of (extension   restriction)	Allows the schema author to prevent derived types being used in the instance document in place of this type (which can be done with the xsi:type attribute). extension and restriction, prevent the use of types derived by extension and restriction respectively, and #all prevents the use of any derived type.
final	#all   List of (extension   restriction)	This attribute restricts the derivation of a new datatype by extension or restriction within the schema. The values it takes act in the same way as those for block.
id	ID	Gives a unique identifier to the type.
mixed	boolean	Specifies whether or not the content of this datatype is mixed.
name	NCName	The name of the datatype specified.

For more information: see §3.4.2 of the Recommendation.

## documentation

Content is used in this to help document the schema, for example, on an element-by-element basis, to indicate how certain structures should be used and so forth. This facilitates automatic generation of documentation for a schema. Complex content is permitted within the documentation tag, so you could include, for example, well-formed XHTML. The documentation tag is always contained within annotation.

## Example

```
<xs:element name = "Person">
  <xs:annotation>
    <xs:documentation>
      Used to contain personal information. Note that the last name
      is mandatory, while the first name is optional.
    </xs:documentation>
  </xs:annotation>
  <!-- definition of Person element goes here -->
</xs:element>
```

## Attributes

Attribute	Value Space	Description
source	anyURI	Specifies the URI where the content of this element may be found. You don't need this attribute, if the content is specified within the documentation tag as in the example above.
xml:lang	language	Specifies the language, using a code defined by RFC 3066. Most languages can be identified by a simple two letter code.

For more information: see §3.13.2 of the Recommendation.

## element

Possibly the most important schema namespace element, this declares the elements that can occur in the instance document. It can contain a simpleType or a complexType depending on the content definition, and unique, key, or keyref elements to define identity constraints. As with most elements, it can also contain an annotation. Elements are declared within model groups using all, choice, or sequence, or can be declared globally as children of the schema element.

## Example

```
<xs:element name = "Customer">
  <xs:complexType>
    <xs:sequence>
      <xs:element name = "FirstName" type = "xs:string" />
      <xs:element name = "MiddleInitial" type = "xs:string" />
      <xs:element name = "LastName" type = "xs:string" />
    </xs:sequence>
    <xs:attribute name = "customerID" type = "xs:string" />
  </xs:complexType>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
abstract	boolean	Specifies that the element is abstract, and so it cannot appear in the instance document, but must be substituted with another element. The default is false.
block	#all   List of (substitution   extension   restriction)	Prevents derived types being used in place of this element, in the instance document (which can be done with the xsi:type attribute), and/or substituting another element in its place. extension and restriction prevent the use of types derived by extension and restriction respectively, and #all prevents the use of any derived type.
default	string	Allows us to specify a default value for the element, if a value is not included in the instance document.
final	#all   List of (extension   restriction)	This prevents the element being nominated as the head element in a substitution group, which has members derived by extension and/or restriction as appropriate.
fixed	string	If present in the instance document, the value of the element must always match the specified string value.
form	qualified   unqualified	If qualified, then the element must be namespace qualified in the instance document. The value of this attribute overrides whatever is specified by the elementFormDefault on the schema element.

id	ID	Gives a unique identifier to the type.
maxOccurs	nonNegativeInteger   unbounded	The maximum number of times the element can occur.
minOccurs	nonNegativeInteger	The minimum number of times the element can occur.
name	NCName	The name of the element.
nillable	boolean	If true, the element may have a nil value specified with xs:nil in the instance document. The default is false.
ref	QName	This attribute allows us to reference a globally defined element, using the value of that element's name attribute.
substitutionGroup	QName	The element becomes a member of the substitution group, specified by this attribute. Wherever the head element of the substitution group is used in a model group, we can substitute this element in its place.
type	QName	The type of the content of this element, which could be simple or complex.

**For more information:** see §3.3.2 of the Recommendation.

## extension

This element is used to extend a base type with further element or attribute declarations. When adding element content to a type, the extension element may contain one or more of element, group, choice or sequence. When adding attributes, it will contain the attribute, attributeGroup or anyAttribute element. Note that when extension is contained inside complexContent, then it can introduce new element and/or attribute content, whereas when it is inside a simpleContent element, then it can only be used to add attributes to a type.

## Example

Extending a complex type:

```
<xs:complexType name="CAN_Address">
  <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="Province" type="xs:string" />
        <xs:element name="PostalCode" type="CAN_PostalCode"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Extending a simple type to produce a complex type with simple content:

```
<xs:complexType name = "ConclusionType" block = "#all">
  <xs:simpleContent>
```

```
<xs:extension base="xs:string">
  <xs:attribute name = "accepted" type = "xs:boolean" />
</xs:extension>
</xs:simpleContent>
</xs:complexType>
```

## Attributes

Attribute	Value Space	Description
base	QName	Specifies the base internal or derived datatype that will be extended.
id	ID	Gives a unique identifier to the element.

**For more information:** see §3.4.2 of the Recommendation.

## field

Allows us to specify an XPath node for an element or attribute used to give a key or unique value (see the key, keyref, and unique elements). It can contain an annotation element, and is used within a key, keyref, or unique element.

## Example

```
<xs:element name = "Employees" minOccurs = "1" maxOccurs = "1" >
  <xs:complexType>
    <xs:sequence>
      <xs:element ref = "Employee" minOccurs = "1"
                  maxOccurs = "unbounded" />
    </xs:sequence>
  </xs:complexType>
  <xs:unique name = "employeeIdentificationNumber">
    <xs:selector xpath = "Employee" />
    <xs:field xpath = "@employeeID" />
  </xs:unique>
</xs:element>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
xpath	XPath	This attribute points to a value that's used to index the elements selected by the identity constraint, relative to those elements.

**For more information:** see §3.11.2 of the Recommendation.

## group

Allows us to define model groups that can be reused in different structures in our schema. The group element can be used in one of two ways: firstly, to define a named model group, and secondly, to reference a globally defined named

model group. The group element may contain an annotation element, and if it is being used to define a group, rather than reference one, then it contains one of all, choice, or sequence. It is used as a child of the schema element when creating a global model group definition, or within choice, sequence, complexType, or redefine when referencing a group.

## Example

```
<xs:element name = "Customer">
  <xs:complexType>
    <xs:group ref = "FirstOrLastNameGroup" />
  </xs:complexType>
</xs:element>

<xs:group name = "FirstOrLastNameGroup">
  <xs:choice>
    <xs:element name = "FirstName" type = "xs:string" />
    <xs:element name = "LastName" type = "xs:string" />
  </xs:choice>
</xs:group>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
maxOccurs	nonNegativeInteger   unbounded	The maximum number of times the element can occur.
minOccurs	nonNegativeInteger	The minimum number of times the element can occur.
name	NCNAME	Defines the name of the model group. In this case we are creating a named model group, so ref, minOccurs, and maxOccurs attributes are not permitted.
ref	QName	This points towards a previously defined model group. When using this attribute, we can't use name, but we can set occurrence constraints with minOccurs and/or maxOccurs.

**For more information:** see §3.7.2 of the Recommendation.

## import

This element imports a schema for another namespace. It's declared as a child of the root schema element, and has an optional annotation. A schema may import multiple other schemas.

## Example

```
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
            targetNamespace = "http://www.example.com/ECommerce"
            xmlns = "http://www.example.com/ECommerce"
            elementFormDefault = "qualified"
            xmlns:wrox = "http://www.wrox.com/ECommerce">
```

```
<xs:import schemaLocation = "http://file_Location/Products.xsd"
            namespace = "http://www.wrox.com/ECommerce" />
<xs:import schemaLocation = "http://file_Location/TypeLib.xsd"
            namespace = "http://www.wrox.com/ECommerce" />

<!-- rest of schema definition here -->

</xs:schema>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
namespace	anyURI	The target namespace of the imported data.
schemaLocation	anyURI	The location of the schema to import.

**For more information:** see §4.2.3 of the Recommendation.

## include

This element is used to include a schema from the same target namespace, or adopt one from no namespace. Like import, it is declared as a child of the root schema element, and may contain an annotation. A schema may include any number of include elements.

## Example

```
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema"
            targetNamespace = "http://www.example.com/ECommerce"
            xmlns = "http://www.example.com/ECommerce"
            elementFormDefault = "qualified">

    <xs:include schemaLocation = "http://location_of_schema/Products.xsd" />
    <xs:include schemaLocation = "http://location_of_schema/TypeLib.xsd" />

    <!-- rest of schema definition here -->

</xs:schema>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
schemaLocation	anyURI	The location of the schema to include.

**For more information:** see §4.2.1 of the Recommendation.

## key

The key element, along with its partner keyref, allows us to define a relationship between two elements. For example, element A might contain a key that is unique within a specified scope. Element B can then refer back to element A using a keyref element. A key is always defined inside an element. It contains selector and field elements to define the element that is the key, and the scope in which it applies. Like other elements, it can also contain an annotation.

## Example

```
<xs:key name = "KeyDepartmentByID">
  <xs:selector xpath = "Departments/Department" />
  <xs:field xpath = "@departmentID" />
</xs:key>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
name	NCName	The name of the key used.

**For more information:** see §3.11.2 of the Recommendation.

## keyref

The keyref element is used to specify a reference to a key (see the discussion of key above.) Like key, it is declared within an element element, and contains an optional annotation element, and selector and field elements.

## Example

```
<xs:keyref name = "RefEmployeeToDepartment" refer = "KeyDepartmentByID">
  <xs:selector xpath = "Employees/Employee" />
  <xs:field xpath = "Department/@refDepartmentID" />
</xs:keyref>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
name	NCName	The name of the key reference.
refer	QName	The name of the key to which this key reference refers.

**For more information:** see §3.11.2 of the Recommendation.

## list

A list is a special sort of simple type-it is a finite-length sequence of whitespace-separated atomic values. The

itemType attribute defines the item type of which the list consists. The itemType cannot be a list type, however. We cannot have a list of lists, or a list of a type that contains whitespace. A list is defined within a simpleType definition, and can contain an optional annotation and simpleType elements.

## Example

```
<xs:simpleType name="AgesList">
  <xs:list itemType="xs:integer" />
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
itemType	QName	A base datatype of which the elements of a list in an instance document consist.

For more information: see §3.14.2 of the Recommendation.

## notation

A notation is used to associate a particular file type with the location of a processing application. The name of the notation should correspond to a value that is declared as a NOTATION datatype (or rather, derived from this type, since it cannot be used directly in the schema). A notation is declared inside the root schema element, and can contain an optional annotation.

## Example

```
<xs:notation name="jpeg" public="image/jpeg" system="JPEG_Viewer.exe" />
<xs:notation name="png" public="image/png" system="PNG_Viewer.exe" />

<xs:simpleType name="notation.Image" >
  <xs:restriction base="xs:NOTATION">
    <xs:enumeration value="jpeg"/>
    <xs:enumeration value="png"/>
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
name	NCName	The name of the specified NOTATION datatype.
public	anyURI	Any URI; usually some relevant identifier, like a MIME type.

system	anyURI	Any URI; usually some local processing application.
--------	--------	---

**For more information:** see §3.12.2 of the Recommendation.

## redefine

This allows us to redefine complex types, simple types, model groups, or attribute groups from another external schema. The external schema needs to have the same target namespace as the one where redefine is used, however, or no namespace. Within the element, we refer to an existing type and amend it as necessary using extension or restriction. redefine is used within the root schema element, and may contain annotation, simpleType, complexType, group, or attributeGroup elements.

## Example

From one schema we have:

```
<xs:complexType name = "NameType">
  <xs:sequence>
    <xs:element name = "FirstName" type = "xs:string" />
    <xs:element name = "MiddleInitial" type = "xs:string" />
    <xs:element name = "LastName" type = "xs:string" />
  </xs:sequence>
</xs:complexType>
```

We can redefine this in another schema like so:

```
<xs:redefine schemaLocation = "http://file_location/firstSchema.xsd">
  <xs:complexType name = "NameType">
    <xs:complexContent>
      <xs:restriction base = "NameType">
        <xs:sequence>
          <xs:element name = "FirstName" type = "xs:string" />
          <xs:element name = "LastName" type = "xs:string" />
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:redefine>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
schemaLocation	anyURI	Specifies the location of the schema.

**For more information:** see §4.2.2 of the Recommendation.

## restriction

This element is used to constrain an existing complex or simple type, using various constraining elements. There are

three different situations where we might use restriction: to restrict a simple type, to restrict a complex type using simple content, or to restrict a complex type using complex content. Therefore, the restrict element may appear inside simpleType, simpleContent, or complexContent. In the first two situations, the element may contain a simpleType element, and one of the constraining facets-minExclusive, maxExclusive, minInclusive, maxInclusive, totalDigits, fractionDigits, length, minLength, maxLength, enumeration, whitespace, or pattern. When restricting a complex type, restriction may also contain attribute, attributeGroup, and anyAttribute, and if the restriction is inside a complexContent element, then it may also include group, all, choice, and sequence. The restriction element also has an optional annotation element.

## Example

Deriving a simple type:

```
<xs:simpleType name="Char">
  <xs:restriction base="xs:string">
    <xs:length value="1" />
  </xs:restriction>
</xs:simpleType>
```

Deriving complex type with simple content:

```
<xs:complexType name = "Person">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name = "age" type = "xs:integer" />
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name = "RestrictedPerson">
  <xs:simpleContent>
    <xs:restriction base="Person">
      <xs:attribute name = "age">
        <xs:simpleType>
          <xs:restriction base = "xs:integer">
            <xs:minInclusive value="1" />
            <xs:maxInclusive value="120" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
```

Deriving a complex type with complex content:

```
<xs:complexType name="ShortAddress">
  <xs:complexContent>
    <xs:restriction base="Address" >
      <xs:sequence>
        <xs:element name="Name" type="xs:string" />
        <xs:element name="Street" type="xs:string" minOccurs="1"
          maxOccurs="2" />
        <xs:element name="City" type="xs:string" />
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
base	QName	The base type from which the new type is derived.

**For more information:** see §3.4.2 and §3.14.2 of the Recommendation.

## schema

This is the parent element of all other schema elements. Details such as target namespace are specified here, as well as various other useful properties, detailed below. It may contain include, import, redefine, annotation, simpleType, complexType, group, attributeGroup, element, attribute, or notation.

## Example

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xss: schema xmlns:xss = "http://www.w3.org/2001/XMLSchema">
    <!--rest of content goes here-->
</xss: schema>
```

## Attributes

Attribute	Value Space	Description
attributeFormDefault	qualified   unqualified	Specifies the default attribute if the attribute is missing.
blockDefault	#all   List of extension   restriction   substitution	Allows us to block some or all of the derivations of datatypes using substitution groups, with the extension, restriction, or <a href="#">substitution</a> element, from being used in the schema; except where overridden by the block attribute of an element or complexType element in the schema.
elementFormDefault	qualified   unqualified	Similar to the attributeFormDefault attribute above, only applies to namespace qualification of elements instead.
finalDefault	#all   List of extension   restriction	Similar to blockDefault, only this blocks all derivations.
id	ID	Gives a unique identifier to the element.
Attribute	Value Space	Description
targetNamespace	anyURI	This is used to specify the namespace that this schema refers to.

version	token	Used to specify the version of the schema. This can take a token datatype.
xml:lang	language	Specifies the language, using a code defined by RFC 3066. Most languages can be identified by a simple two letter code.

### Note

**For more information:** see §3.15.2 of the Recommendation.

## selector

This is used within the context of key, keyref, or unique to define elements that have unique values. The position of the key, keyref or unique element indicates its scope within the document. It may contain annotation.

### Example

```
<xs:key name = "KeyDepartmentByID">
  <xs:selector xpath = "Departments/Department" />
  <xs:field xpath = "@departmentID" />
</xs:key>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
xpath	XPath	A <i>relative</i> XPath expression (relative to the element on which the identity constraint is defined) that specifies which elements the identity constraint applies to.

**For more information:** see §3.11.2 of the Recommendation.

## sequence

Items that are specified inside this element must also appear in the instance document in the same order. We can specify the frequency in which this sequence may appear within the parent node. It is contained within choice, complexType, group, or sequence and may contain annotation, element, group, choice, sequence, or any.

### Example

```
<xs:sequence>
  <xs:element name = "FirstName" type = "xs:string" />
  <xs:element name = "MiddleInitial" type = "xs:string" />
  <xs:element name = "LastName" type = "xs:string" />
</xs:sequence>
```

## Attributes

Attribute	Value Space	Description
-----------	-------------	-------------

<code>id</code>	<code>ID</code>	Gives a unique identifier to the element.
<code>maxOccurs</code>	<code>nonNegativeInteger</code> or <code>unbounded</code>	The maximum number of times the model group can occur.
<code>minOccurs</code>	<code>nonNegativeInteger</code>	The minimum number of times the model group can occur.

**For more information:** see §3.8.2 of the Recommendation.

## simpleContent

This specifies that the content of a datatype is simpleContent (no tagged data). We would normally restrict or extend it using an extension or restriction element. It is contained within complexType and may contain annotation, restriction, or extension.

### Example

```
<xs:complexType name="length1">
  <xs:simpleContent>
    <xs:extension base="xs:nonNegativeInteger">
      <xs:attribute name="unit" type="xs:NMTOKEN"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

## Attributes

Attribute	Value Space	Description
<code>id</code>	<code>ID</code>	Gives a unique identifier to the element.

**For more information:** see §3.4.2 of the Recommendation.

## simpleType

This declares or references a simple type (no element content). It can be contained within attribute, element, list, redefine, restriction, schema, or union, and may contain annotation, list, restriction, or union.

### Example

```
<xs:simpleType name="FixedLengthString">
  <xs:restriction base="xs:string">
    <xs:length value="120" />
  </xs:restriction>
</xs:simpleType>
<simpleType name="Size" >
  <restriction base="xs:string" >
    <enumeration value="S" />
    <enumeration value="M" />
    <enumeration value="L" />
    <enumeration value="XL" />
  </restriction>
</simpleType>
```

## Attributes

Attribute	Value Space	Description
final	#all   List of (union   restriction)	Restricts how new datatypes may be derived from this simple type.
id	ID	Gives a unique identifier to the element.
name	NCName	The name of the datatype that this element is defining.

**For more information:** see §3.14.2 of the Recommendation.

## union

This allows us to join numerous simple datatypes together. Specify a whitespace-separated list of datatypes and they will be joined together to form the new datatype. It is contained by simpleType and may contain annotation, or simpleType.

## Example

```
<xs:simpleType name="union.ShoeSizes">
    <xs:union memberTypes="list.size list.sizenum" />
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
memberTypes	List of QName	A whitespace-separated list of simple datatypes that we wish to join together to become a new simpleType.

**For more information:** see §3.14.2 of the Recommendation.

## unique

It allows us to specify that elements must have a unique value within a document, where the value might be their value, or the value of one of their ancestors or attributes, or a combination. Any unique datatype cannot have the same value more than once within the instance document, subject to the conditions specified in the selector or field elements. It is contained by element and may contain annotation, selector, or field.

## Example

```
<xs:unique name = "employeeIdentificationNumber">
    <xs:selector xpath = "Employees/Employee" />
    <xs:field xpath = "@employeeID" />
</xs:unique>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
name	NCName	Is simply a name for the identity constraint, subject to any constraints/exceptions applied by the selector and field elements.

For more information: see §3.11.2 of the Recommendation.

## XML Schema Instance Attributes

The XML Schema Instance namespace is declared in an instance document to refer to instance specific XML Schema attributes. (The namespace does not include any elements.) For example, the document can indicate to the parser the location of the schema to which it conforms using the schemaLocation attribute. The namespace is:

<http://www.w3.org/2001/XMLSchema-instance>, and is declared in the document element like this:

```
<element-name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

All the attributes detailed in the table below would be prefixed by xsi: in the above case.

Attribute	Type	Description
nil	boolean	Used to indicate that an element is valid despite having an empty value. Necessary for simple types, such as dates and numbers, for which empty values aren't valid. For example:  <OrderDate xsi:nil = "true"></OrderDate>
noNamespaceSchemaLocation	anyURI	Used to specify the location of a schema without a target namespace. For example:  xsi:noNamespaceSchemaLocation= "name.xsd"

Attribute	Type	Description
schemaLocation	list of anyURI	Used to specify the location of a schema with a target namespace. The namespace of the schema is specified first, then after a space there is the location of the schema. Multiple schema/namespace pairs can be given as a whitespace separated list. For example:  xsi:schemaLocation=" <a href="http://www.example.org">http://www.example.org</a> example.xsd"

type	QName	Allows us to override the current element type by specifying the qualified name of a type in an existing XML Schema. Note that the datatype has to be derived from the one that the element is declared with, and it can't have been blocked. For example:  <returnAddress xs:type="ipo:USAddress">
------	-------	---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Appendix E: Schema Datatypes Reference

In this appendix, we will give a quick reference to the W3C Recommendation for XML Schema, Part 2: Datatypes. Datatypes were separated out into a specification in their own right, so that other XML-related technologies as well as XML Schema (for example, Relax NG) can use them.

XML Schema defines a number of **built-in** types that we can use to indicate the intended type of content, and indeed to validate it. We can further restrict these types using **facets** to create our own datatypes, known as **derived types**. The second part of the XML Schema Recommendation defines two sorts of datatype:

- **Built-in types**, which are available to all XML Schema authors, and should be implemented by a conforming processor.
- **User-derived types**, which are defined in individual schema instances, and are particular to that schema (although it is possible to import these definitions into other definitions).

Remember, there are two sub-groups of built-in type:

- **Primitive types**, which are types in their own right. They are not defined in terms of other datatypes. Primitive types are also known as base types, because they are the basis from which all other types are built.
- **Derived types**, which are built from definitions of other datatypes.

In the first part of this appendix, we will provide a quick overview of all the XML built-in datatypes, both primitive and derived. In the second part, we will give details of all of the constraining facets of these datatypes that can be used to restrict the allowed value space thereby deriving new types. Finally, we have tables that illustrate which of these constraining facets can be applied to which datatype.

## XML Schema Built-in Datatypes

Here are the primitive types that XML Schema offers, from which we can derive other datatypes:

Primitive type	Description	Example
----------------	-------------	---------

string	<p>Represents any legal character strings in XML that matches the Char production in XML 1.0 Second Edition (<a href="http://www.w3.org/TR/REC-xml">http://www.w3.org/TR/REC-xml</a>).</p>	<p>Bob Watkins</p> <p>Note, if you need to use these characters in some text (in element content or an attribute value), you will need to escape them according to the XML rules:</p> <p>&amp;lt; or &amp;#60; for &lt; (an opening angled bracket)</p> <p>&amp;gt; or &amp;#62; for &gt; (a closing angled bracket)</p> <p>&amp;amp; or &amp;#30; for &amp; (an ampersand)</p> <p>&amp;apos; or &amp;#39; for ' (an apostrophe)</p> <p>&amp;quot; or &amp;#34; for " (a quotation mark)</p>
boolean	Represents binary logic, true or false.	<p>true, false, 1, 0</p> <p>(These are the only permitted values for this datatype.)</p>
decimal	Represents arbitrary precision decimal numbers.	<p>3.141</p> <p>The ASCII plus (+) and minus (-) characters are used to represent positive or negative numbers, for example: -1.23, +00042.00.</p>
float	Standard concept of real numbers corresponding to a single precision 32 bit floating point type.	<p>-INF, -1E4, 4.5E-2, 37, INF, NaN</p> <p>NaN denotes not a number</p> <p>INF denotes infinity</p>
Primitive type	Description	Example
double	Standard concept of real numbers corresponding to a double precision 64 bit floating point type.	<p>-INF, 765.4321234E11, 7E7, 1.0, INF, NaN</p> <p>NaN denotes not a number</p> <p>INF denotes infinity</p>

duration	<p>Represents a duration of time in the format PnYnMnDTnHnMnS, where:</p> <p>P is a designator that must always be present</p> <p>nY represents number of years</p> <p>nM represents number of months</p> <p>nD represents number of days</p> <p>T is the date/time separator</p> <p>nH is number of hours</p> <p>nM is number of minutes</p> <p>nS is number of seconds</p>	P1Y0M1DT20H25M30S  1 year and a day, 20 hours, 25 minutes and 30 seconds.  Limited forms of this lexical production are also allowed. For example, P120D denotes 120 days.
dateTime	<p>A specific instance in time in the format:</p> <p>CCYY-MM-DDThh:mm:ss where:</p> <p>CC represents the century</p> <p>YY represents the year</p> <p>MM represents the month</p> <p>DD represents the day</p> <p>T is the date/time separator</p> <p>hh represents hours</p> <p>mm represents minutes</p> <p>ss represents seconds (Fractional seconds can be added to arbitrary precision)</p> <p>There is also an optional time zone indicator.</p>	2001-04-16T15:23:15  Represents the 16th of April 2001, at 3:23 and 15 seconds in the afternoon.  (Note that the year 0000 is prohibited, and each of the fields CC, YY, MM and DD must be exactly 2 digits).
time	<p>Represents an instance of time that occurs every day in the format HH:MM:SS.</p> <p>Fractional seconds can be added to arbitrary precision and there is also an optional time zone indicator.</p>	14:12:30  Represents 12 minutes and thirty seconds past two in the afternoon.

date	<p>Represents a calendar date from the Gregorian calendar (the whole day) in the format CCYY-MM-DD. There is also an optional time zone indicator.</p> <p>This complies with the ISO Standard 8601.</p>	2001-04-16  Represents the 16th of April 2001.
gYearMonth	Represents a month in a year in the Gregorian calendar, in the format CCYY-MM.	1999-02  Represents February 1999.
gYear	<p>Represents a year in the Gregorian calendar in the format CCYY.</p> <p>There is also an optional time zone indicator and optional leading minus sign.</p>	1986  Represents 1986.
gMonthDay	<p>Represents a recurring day of a recurring month in the Gregorian calendar in the format --MM-DD.</p> <p>There is also an optional time zone indicator.</p>	--04-16  Represents the 16th of April, ideal for birthdays, holidays, and recurring events.
gDay	Represents a recurring day in the Gregorian calendar in the format --DD.	--16  Represents the sixteenth day of a month. Ideal for monthly occurrences, such as pay day.
gMonth	<p>Represents a recurring month in the Gregorian calendar in the format --MM--.</p> <p>There is also an optional time zone indicator.</p>	--12--  Represents December.
Primitive type	Description	Example
hexBinary	Represents hex-encoded arbitrary binary data.	0FB7
base64Binary	Represents Base64-encoded arbitrary binary data.	GpM7
anyURI	Represents a URI. The value can be absolute or relative, and may have an optional fragment identifier, so it can be a URI Reference.	<a href="http://www.example.com">http://www.example.com</a> <a href="mailto://info@example.com">mailto://info@example.com</a> mySchemafile.xsd

QName	<p>Represents any XML element together with a prefix bound to a namespace, both separated by a colon.</p> <p><i>The XML Namespace Recommendation can be found at: <a href="http://www.w3.org/TR/REC-xml-names/">http://www.w3.org/TR/REC-xml-names/</a>. Namespaces were discussed in <a href="#">Chapter 3</a></i></p>	xs:element
NOTATION	<p>Represents the NOTATION type from XML 1.0 Second Edition. Only datatypes derived from a NOTATION base type (by specifying a value for enumeration) are allowed to be used in a schema.</p> <p>Should only be used for attribute values.</p>	

In order to create new simple datatypes-known as **derived types**-you place further restrictions on an existing built-in type (or another simple type that has been defined). The type that you place the restrictions upon is known as the new type's **base-type**. Here is a list of the **built-in derived types**:

Derived type	Description	Base type	Example
normalizedString	Represents whitespace normalized strings. Whitespace normalized strings do not contain carriage return (#xD), line feed (#xA) or tab (#x9) characters.	string	Like this
token	Represents tokenized strings, they do not contain line feed or tab characters and contain no leading or trailing spaces, and no internal sequences of more than two spaces.	normalizedString	One Two Three
language	Natural language identifiers, as defined in RFC 1766, and valid values for xml:lang as defined in XML 1.0 Second Edition.	token	en-GB, en-US, fr
NMTOKEN	<b>XML 1.0 Second Edition NMTOKEN.</b>	token	small
Name	Represents XML Names.	token	for:example
NCName	Represents XML "non-colonized" Names, without the prefix and colon.	Name	Address
ID	Represents the ID attribute type from XML 1.0 Second Edition.	NCName	

IDREF	Represents the IDREF attribute type from XML 1.0 Second Edition.	NCName	
IDREFS	IDREFS attribute type from XML 1.0 Second Edition. (An aggregation with one and only one member type: ENTITY.)	A list with itemType IDREF	
ENTITY	Represents the ENTITY attribute type from XML 1.0 Second Edition.	NCName	Note that the ENTITY has to be declared externally to the schema in a DTD.
ENTITIES	Represents the ENTITIES attribute type from XML 1.0 Second Edition.  <u>ENTITIES</u> is a set of ENTITY elements separated by an XML whitespace character.	A list with itemType ENTITY  <i>All elements of an ENTITIES instance are of type ENTITY, which also forms some kind of base type</i>	Note that the <u>ENTITIES</u> list has to be declared externally to the schema in a DTD.

Derived type	Description	Base type	Example
integer	Standard mathematical concept of integer numbers.	decimal	-4, 0, 2, 7
nonPositiveInteger	Standard mathematical concept of a non-positive integer (includes 0).	integer	-4, -1, 0
negativeInteger	Standard mathematical concept of negative integers (does not include 0).	nonPositiveInteger	-4, -1
long	An integer between -9223372036854775808 and 9223372036854775807.	integer	-23568323, 52883773203895
int	An integer between -2147483648 and 2147483647.	long	-24781982, 24781924
short	An integer between -32768 and 32767.	int	-31353, -43, 345, 31347

byte	An integer between -128 and 127.	short	-127, -42, 0, 54, 125
nonNegativeInteger	A positive integer including zero.	integer	0, 1, 42
unsignedLong	A nonNegativeInteger between 0 and 18446744073709551615.	nonNegativeInteger	0, 356, 38753829383
unsignedInt	An unsignedLong between 0 and 4294967295.	unsignedLong	46, 4255774, 2342823723
unsignedShort	An unsignedInt between 0 and 65535.	unsignedInt	78, 64328
unsignedByte	An unsignedShort between 0 and 255.	unsignedShort	0, 46, 247
positiveInteger	An integer of 1 or higher.	nonNegativeInteger	1, 24, 345343

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

[< Free Open Study >](#)

NEXT &gt;

# Constraining Facets

The constraining facets defined in the XML Schema Datatypes specification are:

- length
- minLength
- maxLength
- pattern
- enumeration
- whitespace
- maxInclusive
- minInclusive
- maxExclusive
- minExclusive
- totalDigits
- fractionDigits

## **length**

This allows us to specify the exact length of a datatype. If the datatype is a string, then it specifies the number of characters in it. If it's a list, then it specifies the number of items in the list. It is always used inside a restriction element, and can in turn contain an annotation element.

## **Example**

```
<xss:simpleType name="USA_SSN">
  <xss:restriction base="xss:string">
```

```
<xs:length value="11" />
</xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If a simple type has its length facet's fixed attribute set to true, then it cannot be used to derive another simple type with a different length facet. Default is false.
id	ID	Gives a unique identifier to the type.
value	nonNegativeInteger	The actual length of the datatype.

For more information: see §4.3.1 of the Datatypes Recommendation.

## minLength

This sets the minimum length of a datatype. If the base type is string, then it sets the minimum number of characters. If it is a list, it sets the minimum number of members. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="USA_LicensePlate">
  <xs:restriction base="xs:string">
    <xs:minLength value="1" />
    <xs:maxLength value="9" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If true, then any datatypes derived from the one in which this is set cannot alter the value of minLength. The default is false.
id	ID	Gives a unique identifier to the type.
value	nonNegativeInteger	Sets the minimum length of the datatype, if applicable; must be a non-negative integer.

For more information: see §4.3.2 of the Datatypes Recommendation.

## maxLength

This sets the maximum length of a datatype. If the base type is string, then it sets the maximum number of characters. If it is a list, it sets the maximum number of members. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="USA_LicensePlate">
  <xs:restriction base="xs:string">
    <xs:minLength value="1" />
    <xs:maxLength value="9" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If fixed is true, then any datatypes derived from the one in which this is set, cannot alter the value of maxLength. The default is false.
id	ID	Gives a unique identifier to the type.
value	nonNegativeInteger	Sets the maximum length of the datatype, if applicable; must be a non-negative integer.

**For more information:** see §4.3.3 of the Datatypes Recommendation.

## pattern

This allows us to restrict any simple datatype by specifying a regular expression. It acts on the lexical representation of the type, rather than the value itself. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="USA_SSN">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{3}-[0-9]{2}-[0-9]{4}" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the type.
value	anySimpleType	The value contained within this attribute is any valid regular expression.

**For more information:** see §4.3.4 of the Datatypes Recommendation.

## enumeration

The enumeration element is used to restrict the values allowed within a datatype to a set of specified values. It is always used inside a restriction element to do this. It can contain an annotation element.

### Example

```
<xs:simpleType name="Sizes">
  <xs:restriction base="xs:string">
    <xs:enumeration value="S" />
    <xs:enumeration value="M" />
    <xs:enumeration value="L" />
    <xs:enumeration value="XL" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
id	ID	Gives a unique identifier to the element.
value	anySimpleType	One of the values of an enumerated datatype. Multiple enumeration elements are used for the different choices of value.

**For more information:** see §4.3.5 of the Datatypes Recommendation.

## whiteSpace

This dictates what (if any) whitespace transformation is performed upon the XML instances data, before validation constraints are tested. It is always used inside a restriction element to do this. It can contain an annotation element.

### Example

```
<xs:simpleType name="token">
  <xs:restriction base="xs:normalizedString">
    <xs:whiteSpace value="collapse" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If fixed is true, then any type derived from this one cannot set whiteSpace to a value other than the one specified. The default is false.
id	ID	Gives a unique identifier to the type.

value	collapse   preserve   replace	<p>preserve means that all whitespace is preserved as it is declared in the element. If replace is used, then all whitespace characters such as carriage return and tab and so on are replaced by single whitespace characters. collapse means that any series of whitespace characters are collapsed into a single whitespace character.</p> <p>Note that a type with its whiteSpace attribute set to preserve, cannot be derived from one with a value of replace or collapse, and similarly, one with a value of replace, cannot be derived from one with a value of collapse.</p>
-------	-------------------------------------	---

**For more information:** see §4.3.6 of the Datatypes Recommendation.

## maxInclusive

This sets the *inclusive* upper limit of an ordered datatype (number, date type or ordered list). So, the value stated here is therefore the highest value that can be used in this datatype. maxInclusive must be equal to or greater than any value of minInclusive and greater than the value of minExclusive. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="TheAnswer">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="42" />
    <xs:maxInclusive value="42" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If true, then any datatypes derived from this one cannot alter the value of maxInclusive; the default is false.
id	ID	Gives a unique identifier to the type.
value	anySimpleType	If the base datatype is numerical, this would be a number; if a date, then this would be a date.

**For more information:** see §4.3.7 of the Datatypes Recommendation.

## minInclusive

This sets the *inclusive* lower limit of an ordered datatype (number, date type, or ordered list). The value stated here is therefore the lowest value that can be used in this datatype. minInclusive must be equal to or less than any value of maxInclusive and must be less than the value of maxExclusive. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="TheAnswer">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="42" />
    <xs:maxInclusive value="42" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If true, then any datatypes derived from this one cannot alter the value of minInclusive; the default is false.
id	ID	Gives a unique identifier to the type.
value	anySimpleType	If the base datatype is numerical, this would be a number; if a date, then a date.

**For more information:** see §4.3.10 of the Datatypes Recommendation.

## maxExclusive

This sets the *exclusive* upper limit of an ordered datatype (number, date type, or ordered list). The maxExclusive value is therefore one higher than the maximum value that can be used. maxExclusive must be greater than or equal to the value of minExclusive and greater than the value of minInclusive. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="TheAnswer">
  <xs:restriction base="xs:integer">
    <xs:minExclusive value="42" />
    <xs:maxExclusive value="42" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If true, then any datatypes derived from this one cannot alter the value of maxExclusive; the default is false.
id	ID	Gives a unique identifier to the type.
value	anySimpleType	If the base datatype is numerical, this is a number; if a date, then it is a date.

**For more information:** see §4.3.8 of the Datatypes Recommendation.

## minExclusive

This sets the *exclusive* lower limit of an ordered datatype (number, date type or ordered list). The minExclusive value is therefore one lower than the lowest value the data can take. minExclusive must be less than the value of maxInclusive, and less than or equal to the value of maxExclusive. It is always used inside a restriction element to do this. It can contain an annotation element.

## Example

```
<xs:simpleType name="TheAnswer">
  <xs:restriction base="xs:integer">
    <xs:minExclusive value="42" />
    <xs:maxExclusive value="42" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
fixed	boolean	If true, then any datatypes derived from this one, cannot alter the value of minExclusive; the default is false.
id	ID	Gives a unique identifier to the type.
value	anySimpleType	If the base datatype is numerical, this would be a number; if a date, then a date.

**For more information:** see §4.3.9 of the Datatypes Recommendation.

## totalDigits

This facet applies to all datatypes derived from the decimal type. The value stated is the *maximum* number of decimal digits allowed for the entire number (which must always be a positive integer).

## Example

```
<xs:simpleType name="Datapoint">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="9" />
    <xs:fractionDigits value="3" />
  </xs:restriction>
</xs:simpleType>
```

## Attributes

Attribute	Value Space	Description
-----------	-------------	-------------

fixed	boolean	If true, then any datatypes derived from this one cannot alter the value of totalDigits; the default is false.
id	ID	Gives a unique identifier to the type.
value	positiveInteger	The actual value of the totalDigits attribute.

**For more information:** see §4.3.11 of the Datatypes Recommendation.

## **fractionDigits**

This facet applies to all datatypes derived from the decimal type. The value stated is the *maximum* number of digits in the fractional portion of the number (always a *non-negative* integer that is less than or equal to the value of totalDigits).

## Example

```
<xs:simpleType name="Datapoint">
    <xs:restriction base="xs:decimal">
        <xs:totalDigits value="9" />
        <xs:fractionDigits value="3" />
    </xs:restriction>
</xs:simpleType>
```

## Attributes

<b>Attribute</b>	<b>Value Space</b>	<b>Description</b>
fixed	boolean	If true, then any datatypes derived from this one cannot alter the value of totalDigits; the default is false.

<b>Attribute</b>	<b>Value Space</b>	<b>Description</b>
id	ID	Gives a unique identifier to the type.
value	nonNegativeInteger	The actual value of the value fractionDigits attribute. This cannot be any larger than the totalDigits value.

**For more information:** see §4.3.12 of the Datatypes Recommendation.

The two tables below, indicate which of these constraining facets may be applied to which datatypes, in order to derive new types. First, for the primitive built-in types:

string	X	X	X	preserve	X	X						
anyURI	X	X	X	collapse	X	X						
NOTATION	X	X	X	collapse	X	X						
QName	X	X	X	collapse	X	X						
<b>Binary Encoding Types</b>												
boolean				collapse	X							
hexBinary	X	X	X	collapse	X	X						
base64Binary	X	X	X	collapse	X	X						
Numeric Types												
decimal				collapse	X	X	X	X	X	X	X	X
float				collapse	X	X	X	X	X	X		
double				collapse	X	X	X	X	X	X		
<b>Date/Time Types</b>												
duration				collapse	X	X	X	X	X	X		
dateTime				collapse	X	X	X	X	X	X		
date				collapse	X	X	X	X	X	X		
time				collapse	X	X	X	X	X	X		
gYear				collapse	X	X	X	X	X	X		
gYearMonth				collapse	X	X	X	X	X	X		
<b>Datatypes</b>	<b>length</b>	<b>minLength</b>	<b>maxLength</b>	<b>whiteSpace</b>	<b>pattern</b>	<b>enumeration</b>	<b>minExclusive</b>	<b>maxExclusive</b>	<b>minInclusive</b>	<b>maxInclusive</b>	<b>totalDigits</b>	<b>fractionDigits</b>
gMonth				collapse	X	X	X	X	X	X		
gMonthDay				collapse	X	X	X	X	X	X		
gDay				collapse	X	X	X	X	X	X		

Second, and for the derived built-in types:



int				collapse	X	X	X	X	X	X	X	X	0
long				collapse	X	X	X	X	X	X	X	X	0
unsignedByte				collapse	X	X	X	X	X	X	X	X	0
unsignedShort				collapse	X	X	X	X	X	X	X	X	0
unsignedInt				collapse	X	X	X	X	X	X	X	X	0
unsignedLong				collapse	X	X	X	X	X	X	X	X	0

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Appendix F: SAX 2.0: The Simple API for XML

This appendix contains the specification of the SAX interface, version 2.0, some of which is explained in [Chapter 9](#). It is taken largely verbatim from the definitive specification to be found at <http://www.megginson.com/SAX/index.html>, with editorial comments added in italics.

The classes and interfaces are described in alphabetical order.

The SAX specification is in the public domain?see the web site quoted above for a statement of policy on copyright. Essentially the policy is: do what you like with it, copy it as you wish, but no one accepts any liability for errors or omissions.

The SAX distribution also includes two other "helper classes":

- LocatorImpl is an implementation of the Locator interface
- ParserFactory is a class that enables you to load a parser identified by a parameter at run-time

The documentation of these helper classes is not included here. For this, and for SAX sample applications, see the SAX distribution available from <http://www.megginson.com>.

SAX 2.0 contains complete namespace support, which is available by default from any XMLReader object. An XML reader can also optionally supply raw XML 1.0 names.

An XML reader is fully configurable: it is possible to attempt to query or change the current value of any feature or property. Features and properties are identified by fully-qualified URIs, and parties are free to invent their own names for new extensions.

The ContentHandler and [Attributes](#) interfaces are similar to the deprecated DocumentHandler and AttributeList interfaces, but they add support for namespace-related information. ContentHandler also adds a callback for skipped entities, and [Attributes](#) adds the ability to look up an attribute's index by name.

The following interfaces have been deprecated:

- org.xml.sax.Parser
- org.xml.sax.DocumentHandler
- org.xml.sax.AttributeList

org.xml.sax.HandlerBase

The following interfaces and classes have been added to SAX2:

- org.xml.sax.XMLReader (replaces Parser)
- org.xml.sax.XMLFilter
- org.xml.sax.ContentHandler (replaces DocumentHandler)
- org.xml.sax.Attributes (replaces AttributeList)
- org.xml.sax.SAXNotSupportedException
- org.xml.sax.SAXNotRecognizedException

## Class and Interface Hierarchies

The following diagrams show the class and interface hierarchies of SAX 2.0. We covered some of these classes in [Chapter 9](#), although many were left out as they are outside of the scope of what you will most likely need to know. However, this appendix covers them all, and further details can be found at the SAX web site.

### Class Hierarchy

```
-class java.lang.Object
  -class org.xml.sax.helpers.AttributeListImpl
    (implements org.xml.sax.AttributeList)
  -class org.xml.sax.helpers.AttributesImpl
    (implements org.xml.sax.Attributes)
  -class org.xml.sax.helpers.DefaultHandler
    (implements org.xml.sax.ContentHandler, org.xml.sax.DTDHandler,
     org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler)
  -class org.xml.sax.HandlerBase
    (implements org.xml.sax.DocumentHandler, org.xml.sax.DTDHandler,
     org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler)
  -class org.xml.sax.InputSource
  -class org.xml.sax.helpers.LocatorImpl
    (implements org.xml.sax.Locator)
  -class org.xml.sax.helpers.NamespaceSupport
  -class org.xml.sax.helpers.ParserAdapter
    (implements org.xml.sax.DocumentHandler, org.xml.sax.XMLReader)
  -class org.xml.sax.helpers.ParserFactory
    class java.lang.Throwable
    (implements java.io.Serializable)
  -class java.lang.Exception
    -class org.xml.sax.SAXException
      -class org.xml.sax.SAXNotRecognizedException
      -class org.xml.sax.SAXNotSupportedException
      -class org.xml.sax.SAXParseException
  -class org.xml.sax.helpers.XMLFilterImpl
    (implements org.xml.sax.ContentHandler, org.xml.sax.DTDHandler,
     org.xml.sax.EntityResolver, org.xml.sax.ErrorHandler, org.xml.sax.XMLFilter)
  -class org.xml.sax.helpers.XMLReaderAdapter
    (implements org.xml.sax.ContentHandler, org.xml.sax.Parser)
  -class org.xml.sax.helpers.XMLReaderFactory
```

### Interface Hierarchy

```
interface org.xml.sax.AttributeSet
interface org.xml.sax.ContentHandler
interface org.xml.sax.DocumentHandler
interface org.xml.saxDTDHandler
interface org.xml.sax.EntityResolver
interface org.xml.sax.ErrorHandler
interface org.xml.sax.Locator
interface org.xml.sax.Parser
interface org.xml.sax.XMLReader
interface org.xml.sax.XMLFilter
```

## org.xml.sax.Attributes (SAX 2?Replaces AttributeList)

Interface for a list of XML attributes?this interface allows access to a list of attributes in three different ways:

- By attribute index
- By namespace-qualified name
- By qualified (prefixed) name

The list will not contain attributes that were declared #IMPLIED but not specified in the start tag. It will also not contain attributes used as namespace declarations (xmlns\*) unless the <http://xml.org/sax/features/namespace-prefixes> feature is set to true (it is false by default).

If the namespace-prefixes feature (see above) is false, access by qualified name may not be available; if the <http://xml.org/sax/features/namespaces> feature is false, access by namespace-qualified names may not be available.

This interface replaces the now-deprecated SAX1 AttributeList interface, which does not contain namespace support. In addition to namespace support, it adds the getIndex methods (below).

The order of attributes in the list is unspecified, and will vary from implementation to implementation.

<b>getLength</b>  public int getLength()	Return the number of attributes in the list.  Once you know the number of attributes, you can iterate through the list.  <b>Returns:</b>  The number of attributes in the list.
<b>getURI</b>  public String getURI(int index)	Look up an attribute's namespace URI by index.  <b>Parameters</b>  index?the attribute index (zero-based).  <b>Returns:</b>  The namespace URI, or the empty string if none is available, or Null if the index is out of range.

<p><b>getLocalName</b></p> <pre>public String getLocalName(int index)</pre>	<p>Look up an attribute's local name by index.</p> <p><b>Parameters</b></p> <p>index?the attribute index (zero-based).</p> <p><b>Returns:</b></p> <p>The local name, or the empty string if namespace processing is not being performed, or Null if the index is out of range.</p>
<p><b>getQName</b></p> <pre>public String getQName(int index)</pre>	<p>Look up an attribute's XML 1.0 qualified name by index.</p> <p><b>Parameters</b></p> <p>index?the attribute index (zero-based).</p> <p><b>Returns:</b></p> <p>The XML 1.0 qualified name, or the empty string if none is available, or Null if the index is out of range.</p>
<p><b>getType</b></p> <pre>public String getType(int index)</pre>	<p>Look up an attribute's type by index.</p> <p>The attribute type is one of the strings "CDATA", "ID", "IDREF", "IDREFS", "NMOKEN", "NMOKENS", "ENTITY", "<a href="#">ENTITIES</a>", or "NOTATION" (always in uppercase).</p> <p>If the parser has not read a declaration for the attribute, or if the parser does not report attribute types, then it must return the value "CDATA" as stated in the XML 1.0 Recommendation (clause 3.3.3, "Attribute-Value Normalization").</p> <p>For an enumerated attribute that is not a notation, the parser will report the type as "NMOKEN".</p> <p><b>Parameters</b></p> <p>index?the attribute index (zero-based).</p> <p><b>Returns:</b></p> <p>The attribute's type as a string, or null if the index is out of range.</p>

<p><b>getValue</b></p> <pre>public String getValue(int index)</pre>	<p>Look up an attribute's value by index.</p> <p>If the attribute value is a list of tokens (IDREFS, ENTITIES, or NMTOKENS), the tokens will be concatenated into a single string with each token separated by a single space.</p> <p><b>Parameters</b></p> <p>index?the attribute index (zero-based).</p> <p><b>Returns:</b></p> <p>The attribute's value as a string, or Null if the index is out of range.</p>
<p><b>getIndex</b></p> <pre>public int getIndex(String uri, String localPart)</pre>	<p>Look up the index of an attribute by namespace name.</p> <p><b>Parameters</b></p> <p>uri?the namespace URI, or the empty string if the name has no namespace URI.</p> <p>localName?the attribute's local name.</p> <p><b>Returns:</b></p> <p>The index of the attribute, or -1 if it does not appear in the list.</p>
<p><b>getIndex</b></p> <pre>public int getIndex(String qName)</pre>	<p>Look up the index of an attribute by XML 1.0 qualified name.</p> <p><b>Parameters</b></p> <p>qName?the qualified (prefixed) name.</p> <p><b>Returns:</b></p> <p>The index of the attribute, or -1 if it does not appear in the list.</p>
<p><b>getType</b></p> <pre>public String getType(String uri, String localName)</pre>	<p>Look up an attribute's type by namespace name.</p> <p>See <code>getType(int)</code> for a description of the possible types.</p> <p><b>Parameters</b></p> <p>uri?the namespace URI, or the empty string if the name has no namespace URI.</p> <p>localName?the local name of the attribute.</p> <p><b>Returns:</b></p> <p>The attribute type as a string, or Null if the attribute is not in the list or if namespace processing is not being performed.</p>

<p><b>getType</b></p> <pre>public String getType(String qName)</pre>	<p>Look up an attribute's type by XML 1.0 qualified name. See <code>getType(int)</code> for a description of the possible types.</p> <p><b>Parameters</b></p> <p><code>qName</code>?the XML 1.0 qualified name.</p> <p><b>Returns:</b></p> <p>The attribute type as a string, or null if the attribute is not in the list or if qualified names are not available.</p>
<p><b>getValue</b></p> <pre>public String getValue(String uri, String localName)</pre>	<p>Look up an attribute's value by namespace name. See <code>getValue(int)</code> for a description of the possible values.</p> <p><b>Parameters</b></p> <p><code>uri</code>?the namespace URI, or the empty string if the name has no namespace URI. <code>localName</code>?the local name of the attribute.</p> <p><b>Returns:</b></p> <p>The attribute value as a string, or Null if the attribute is not in the list.</p>
<p><b>getValue</b></p> <pre>public String getValue(String qName)</pre>	<p>Look up an attribute's value by XML 1.0 qualified name. See <code>getValue(int)</code> for a description of the possible values.</p> <p><b>Parameters</b></p> <p><code>qName</code>?the XML 1.0 qualified name.</p> <p><b>Returns:</b></p> <p>The attribute value as a string, or Null if the attribute is not in the list or if qualified names are not available.</p>

## Interface org.xml.sax.AttributeList?Deprecated

An AttributeList is a collection of attributes appearing on a particular start tag. The parser supplies the DocumentHandler with an AttributeList as part of the information available on the `startElement` event. The AttributeList is essentially a set of name-value pairs for the supplied attributes; if the parser has analyzed the DTD it may also provide information about the type of each attribute.

## Interface for an Element's Attribute Specifications

The SAX parser implements this interface and passes an instance to the SAX application as the second argument of each `startElement` event.

The instance provided will return valid results only during the scope of the startElement invocation (to save it for future use, the application must make a copy: the AttributeListImpl helper class provides a convenient constructor for doing so).

An AttributeList includes only attributes that have been specified or defaulted: #IMPLIED attributes will not be included.

There are two ways for the SAX application to obtain information from the AttributeList. First, it can iterate through the entire list:

```
public void startElement (String name, AttributeList atts) {  
    for (int i = 0; i < atts.getLength(); i++) {  
        String name = atts.getName(i);  
        String type = atts.getType(i);  
        String value = atts.getValue(i);  
        [...]  
    }  
}
```

(Note that the result of getLength() will be zero if there are no attributes.)

As an alternative, the application can request the value or type of specific attributes:

```
public void startElement (String name, AttributeList atts) {  
    String identifier = atts.getValue("id");  
    String label = atts.getValue("label");  
    [...]  
}
```

The AttributeListImpl helper class provides a convenient implementation for use by parser or application writers.

#### **getLength**

```
public int getLength()
```

Return the number of attributes in this list.

The SAX parser may provide attributes in any arbitrary order, regardless of the order in which they were declared or specified. The number of attributes may be zero.

#### **Returns:**

The number of attributes in the list.

<p><b>getName</b></p> <pre>public String getName(int index)</pre>	<p>Return the name of an attribute in this list (by position).</p> <p>The names must be unique: the SAX parser shall not include the same attribute twice. Attributes without values (those declared #IMPLIED without a value specified in the starttag) will be omitted from the list.</p> <p>If the attribute name has a namespace prefix, the prefix will still be attached.</p> <p><b>Parameters:</b></p> <p>index?the index of the attribute in the list (starting at 0).</p> <p><b>Returns:</b></p> <p>The name of the indexed attribute, or Null if the index is out of range.</p>
<p><b>getType</b></p> <pre>public String getType(int index)</pre>	<p>Return the type of an attribute in the list (by position).</p> <p>The attribute type is one of the strings "CDATA", "ID", "IDREF", "IDREFS", "NMOKEN", "NMOKENS", "ENTITY", "<a href="#">ENTITIES</a>", or "NOTATION" (always in uppercase).</p> <p>If the parser has not read a declaration for the attribute, or if the parser does not report attribute types, then it must return the value "CDATA" as stated in the XML 1.0 Recommendation (clause 3.3.3, "Attribute-Value Normalization").</p> <p>For an enumerated attribute that is not a notation, the parser will report the type as "NMOKEN".</p> <p><b>Parameters:</b></p> <p>index?the index of the attribute in the list (starting at 0).</p> <p><b>Returns:</b></p> <p>The attribute type as a string, or null if the index is out of range.</p>
<p><b>getType</b></p> <pre>public String getType(String name)</pre>	<p>Return the type of an attribute in the list (by name).</p> <p>The return value is the same as the return value for getType(int).</p> <p>If the attribute name has a namespace prefix in the document, the application must include the prefix here.</p> <p><b>Parameters:</b></p> <p>name?the name of the attribute.</p> <p><b>Returns:</b></p> <p>The attribute type as a string, or Null if no such attribute exists.</p>

<p><b>getValue</b></p> <pre>public String getValue(int index)</pre>	<p>Return the value of an attribute in the list (by position). If the attribute value is a list of tokens (IDREFS, <a href="#">ENTITIES</a>, or NMTOKENS), the tokens will be concatenated into a single string separated by whitespace.</p> <p><b>Parameters:</b></p> <p>index?the index of the attribute in the list (starting at 0).</p> <p><b>Returns:</b></p> <p>The attribute value as a string, or Null if the index is out of range.</p>
<p><b>getValue</b></p> <pre>public String getValue(String name)</pre>	<p>Return the value of an attribute in the list (by name). The return value is the same as the return value for <code>getValue(int)</code>. If the attribute name has a namespace prefix in the document, the application must include the prefix here.</p> <p><b>Parameters:</b></p> <p>name?the name of the attribute.</p> <p><b>Returns:</b></p> <p>The attribute value as a string, or Null if no such attribute exists.</p>

## Interface org.xml.sax.ContentHandler (SAX 2?Replaces DocumentHandler)

*Every SAX application is likely to include a class that implements this interface, either directly or by subclassing the supplied class HandlerBase.*

### Receive notification of general document events

Receive notification of the logical content of a document?this is the main interface that most SAX applications implement: if the application needs to be informed of basic parsing events, it implements this interface and registers an instance with the SAX parser using the `setContentHandler` method. The parser uses the instance to report basic document-related events like the start and end of elements and character data.

The order of events in this interface is very important, and mirrors the order of information in the document itself. For example, all of an element's content (character data, processing instructions, and/or subelements) will appear, in order, between the `startElement` event and the corresponding `endElement` event.

This interface is similar to the now-deprecated SAX 1.0 DocumentHandler interface, but it adds support for namespaces and for reporting skipped entities (in non-validating XML processors).

Implementors should note that there is also a Java class ContentHandler in the `java.net` package; that means that it's probably a bad idea to do the following (more like a feature than a bug anyway, as `import ... *` is a sign of bad programming):

```
import java.net.*; import  
org.xml.sax.*;
```

### **setDocumentLocator**

```
public void setDocumentLocator(  
    Locator locator)
```

Receive an object for locating the origin of SAX document events.

SAX parsers are strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the ContentHandler interface.

The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.

Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.

#### **Parameters**

locator?an object that can return the location of any SAX document event.

### **startDocument**

```
public void startDocument()
```

Receive notification of the beginning of a document.

The SAX parser will invoke this method only once, before any other methods in this interface or in DTDHandler (except for setDocumentLocator).

#### **Throws:**

SAXException?any SAX exception, possibly wrapping another exception.

### **endDocument**

```
public void endDocument()
```

Receive notification of the end of a document.

The SAX parser will invoke this method only once, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.

#### **Throws:**

SAXException?any SAX exception, possibly wrapping another exception.

<p><b>startPrefixMapping</b></p> <pre>public void startPrefixMapping (String prefix, String uri)</pre>	<p>Begin the scope of a prefix-URI namespace mapping.</p> <p>The information from this event is not necessary for normal namespace processing: the SAX XML reader will automatically replace prefixes for element and attribute names when the <a href="http://xml.org/sax/features/namespaces">http://xml.org/sax/features/namespaces</a> feature is true (the default).</p> <p>There are cases, however, when applications need to use prefixes in character data or in attribute values, where they cannot safely be expanded automatically; the start/endPrefixMapping event supplies the information to the application to expand prefixes in those contexts itself, if necessary.</p> <p>Note that start/endPrefixMapping events are not guaranteed to be properly nested relative to each-other: all startPrefixMapping events will occur before the corresponding startElement event, and all endPrefixMapping events will occur after the corresponding endElement event, but their order is not otherwise guaranteed.</p> <p>There should never be start/endPrefixMapping events for the "xml" prefix, since it is pre-declared and immutable.</p> <p><b>Parameters</b></p> <p>prefix?the namespace prefix being declared.</p> <p>uri?the namespace URI the prefix is mapped to.</p> <p><b>Throws:</b></p> <p>SAXException?the client may throw an exception during processing.</p>
<p><b>endPrefixMapping</b></p> <pre>public void endPrefixMapping (String prefix)</pre>	<p>End the scope of a prefix-URI mapping.</p> <p>See startPrefixMapping for details. This event will always occur after the corresponding endElement event, but the order of endPrefixMapping events is not otherwise guaranteed.</p> <p><b>Parameters</b></p> <p>prefix?the prefix that was being mapped.</p> <p><b>Throws:</b></p> <p>SAXException?the client may throw an exception during processing.</p>

## **startElement**

```
public void startElement(String namespaceURI, String localName, String qName, Attributes atts)
```

Receive notification of the beginning of an element.

The parser will invoke this method at the beginning of every element in the XML document; there will be a corresponding endElement event for every startElement event (even when the element is empty). All of the element's content will be reported, in order, before the corresponding endElement event.

This event allows up to three name components for each element:

- the namespace URI
- the local name
- the qualified (prefixed) name

Any or all of these may be provided, depending on the values of the [http://xml.org/sax/features/namespaces](#) and the [http://xml.org/sax/features/namespace-prefixes](#) properties:

- The namespace URI and local name are required when the namespaces property is true (the default), and are optional when the namespaces property is false (if one is specified, both must be).
- The qualified name is required when the namespace-prefixes property is true, and is optional when the namespace-prefixes property is false (the default).

Note that the attribute list provided will contain only attributes with explicit values (specified or defaulted): #IMPLIED attributes will be omitted. The attribute list will contain attributes used for namespace declarations (xmlns\* attributes) only if the [http://xml.org/sax/features/namespace-prefixes](#) property is true (it is false by default, and support for a true value is optional).

### **Parameters**

uri?the namespace URI, or the empty string if the element has no namespace URI or if namespace processing is not being performed.

localName?the local name (without prefix), or the empty string if namespace processing is not being performed.

qName?the qualified name (with prefix), or the empty string if qualified names are not available.

atts?the attributes attached to the element. If there are no attributes, it shall be an empty [Attributes](#) object.

### **Throws:**

SAXException?any SAX exception, possibly wrapping another exception.

<b>endElement</b> <pre>public void endElement(String namespaceURI, String localName, String qName)</pre>	<p>Receive notification of the end of an element.</p> <p>The SAX parser will invoke this method at the end of every element in the XML document; there will be a corresponding startElement event for every endElement event (even when the element is empty).</p> <p>For information on the names, see startElement.</p> <p><b>Parameters</b></p> <p>uri?the namespace URI, or the empty string if the element has no namespace URI or if namespace processing is not being performed.</p> <p>localName?the local name (without prefix), or the empty string if namespace processing is not being performed.</p> <p>qName?the qualified XML 1.0 name (with prefix), or the empty string if qualified names are not available.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<b>characters</b> <pre>public void characters (char[] ch, int start, int length)</pre>	<p>Receive notification of character data.</p> <p>The parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity so that the locator provides useful information.</p> <p>The application must not attempt to read from the array outside of the specified range.</p> <p>Note that some parsers will report whitespace in element content using the ignorableWhitespace method rather than this one (validating parsers must do so).</p> <p><b>Parameters</b></p> <p>ch?the characters from the XML document.</p> <p>start?the start position in the array.</p> <p>length?the number of characters to read from the array.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>

<p><b>ignorableWhitespace</b></p> <pre>public void ignorableWhitespace (char[] ch, int start, int length)</pre>	<p>Receive notification of ignorable whitespace in element content.</p> <p>Validating parsers must use this method to report each chunk of whitespace in element content (see the W3C XML 1.0 Recommendation, section 2.10): non-validating parsers may also use this method if they are capable of parsing and using content models.</p> <p>SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the locator provides useful information.</p> <p>The application must not attempt to read from the array outside of the specified range.</p> <p><b>Parameters</b></p> <p>ch?the characters from the XML document.</p> <p>start?the start position in the array.</p> <p>length?the number of characters to read from the array.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<p><b>processingInstruction</b></p> <pre>public void processingInstruction (String target, String data)</pre>	<p>Receive notification of a processing instruction.</p> <p>The parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.</p> <p>A SAX parser must never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.</p> <p><b>Parameters</b></p> <p>target?the processing instruction target.</p> <p>data?the processing instruction data, or null if none was supplied. The data does not include any whitespace separating it from the target.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>

<b>skippedEntity</b>  public void skippedEntity (String name)	<p>Receive notification of a skipped entity.</p> <p>The parser will invoke this method once for each entity skipped. Non-validating processors may skip entities if they have not seen the declarations (because, for example, the entity was declared in an external DTD subset). All processors may skip external entities, depending on the values of the <a href="http://xml.org/sax/features/external-general-entities">http://xml.org/sax/features/external-general-entities</a> and the <a href="http://xml.org/sax/features/external-parameter-entities">http://xml.org/sax/features/external-parameter-entities</a> properties.</p> <p><b>Parameters</b></p> <p>name?the name of the skipped entity. If it is a parameter entity, the name will begin with '%', and if it is the external DTD subset, it will be the string "[dtd]".</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
---	---

## Interface org.xml.sax.DocumentHandler?Deprecated

*Every SAX application is likely to include a class that implements this interface, either directly or by subclassing the supplied class HandlerBase.*

### Receive notification of General Document Events

This is the main interface that most SAX applications implement: if the application needs to be informed of basic parsing events, it implements this interface and registers an instance with the SAX parser using the setDocumentHandler method. The parser uses the instance to report basic document-related events like the start and end of elements and character data.

The order of events in this interface is very important, and mirrors the order of information in the document itself. For example, all of an element's content (character data, processing instructions, and/or sub-elements) will appear, in order, between the startElement event and the corresponding endElement event.

Application writers who do not want to implement the entire interface can derive a class from HandlerBase, which implements the default functionality; parser writers can instantiate HandlerBase to obtain a default handler. The application can find the location of any document event using the locator interface supplied by the parser through the setDocumentLocator method.

<b>characters</b>  public void characters(char ch[], int start, int length) throws SAXException	<p>Receive notification of character data.</p> <p>The parser will call this method to report each chunk of character data. SAX parsers may return all contiguous character data in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the locator provides useful information.</p> <p>The application must not attempt to read from the array outside of the specified range <i>and must not attempt to write to the array</i>.</p> <p>Note that some parsers will report whitespace using the ignorableWhitespace() method rather than this one (validating parsers <i>must</i> do so).</p> <p><b>Parameters:</b></p> <ul style="list-style-type: none"><li>ch?the characters from the XML document.</li><li>start?the start position in the array.</li><li>length?the number of characters to read from the array.</li></ul> <p><b>Throws:</b></p> <ul style="list-style-type: none"><li>SAXException?any SAX exception, possibly wrapping another exception.</li></ul>
<b>endDocument</b>  public void endDocument() throws SAXException	<p>Receive notification of the end of a document.</p> <p>The SAX parser will invoke this method only once <i>for each document</i>, and it will be the last method invoked during the parse. The parser shall not invoke this method until it has either abandoned parsing (because of an unrecoverable error) or reached the end of input.</p> <p><b>Throws:</b></p> <ul style="list-style-type: none"><li>SAXException?any SAX exception, possibly wrapping another exception.</li></ul>

<b>endElement</b>  public void endElement(String name) throws SAXException	<p>Receive notification of the end of an element.</p> <p>The SAX parser will invoke this method at the end of every element in the XML document; there will be a corresponding startElement() event for every endElement() event (even when the element is empty).</p> <p>If the element name has a namespace prefix, the prefix will still be attached to the name.</p> <p><b>Parameters:</b></p> <p>name?the element type name.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<b>ignorableWhitespace</b>  public void ignorableWhitespace (char ch[], int start, int length) throws SAXException	<p>Receive notification of ignorable whitespace in element content.</p> <p>Validating parsers must use this method to report each chunk of ignorable whitespace (see the W3C XML 1.0 Recommendation, section 2.10); non-validating parsers may also use this method if they are capable of parsing and using content models.</p> <p>SAX parsers may return all contiguous whitespace in a single chunk, or they may split it into several chunks; however, all of the characters in any single event must come from the same external entity, so that the locator provides useful information.</p> <p>The application must not attempt to read from the array outside of the specified range.</p> <p><b>Parameters:</b></p> <p>ch?the characters from the XML document.</p> <p>start?the start position in the array.</p> <p>length?the number of characters to read from the array.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>

<p><b>processingInstruction</b></p> <pre>public void processingInstruction (String target, String data) throws SAXException</pre>	<p>Receive notification of a processing instruction.</p> <p>The parser will invoke this method once for each processing instruction found: note that processing instructions may occur before or after the main document element.</p> <p>A SAX parser should never report an XML declaration (XML 1.0, section 2.8) or a text declaration (XML 1.0, section 4.3.1) using this method.</p> <p><b>Parameters:</b></p> <p>target?the processing instruction target.</p> <p>data?the processing instruction data, or null if none was supplied.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<p><b>setDocumentLocator</b></p> <pre>public void setDocumentLocator (Locator locator)</pre>	<p>Receive an object for locating the origin of SAX document events.</p> <p>A SAX parser is strongly encouraged (though not absolutely required) to supply a locator: if it does so, it must supply the locator to the application by invoking this method before invoking any of the other methods in the DocumentHandler interface.</p> <p>The locator allows the application to determine the end position of any document-related event, even if the parser is not reporting an error. Typically, the application will use this information for reporting its own errors (such as character content that does not match an application's business rules). The information returned by the locator is probably not sufficient for use with a search engine.</p> <p>Note that the locator will return correct information only during the invocation of the events in this interface. The application should not attempt to use it at any other time.</p> <p><b>Parameters:</b></p> <p>locator?an object that can return the location of any SAX document event.</p>

<p><b>startDocument</b></p> <pre>public void startDocument() throws SAXException</pre>	<p>Receive notification of the beginning of a document.</p> <p>The SAX parser will invoke this method only once <i>for each document</i>, before any other methods in this interface or in DTDHandler (except for setDocumentLocator).</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<p><b>startElement</b></p> <pre>public void startElement (String name, AttributeList atts) throws SAXException</pre>	<p>Receive notification of the beginning of an element.</p> <p>The parser will invoke this method at the beginning of every element in the XML document; there will be a corresponding endElement() event for every startElement() event (even when the element is empty). All of the element's content will be reported, in order, before the corresponding endElement() event.</p> <p>If the element name has a namespace prefix, the prefix will still be attached. Note that the attribute list provided will contain only attributes with explicit values (specified or defaulted): #IMPLIED attributes will be omitted.</p> <p><b>Parameters:</b></p> <p>name?the element type name.</p> <p>atts?the attributes attached to the element, if any.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>

## Interface org.xml.sax.DTDHandler

*This interface should be implemented by the application if it wants to receive notification of events related to the DTD. SAX does not provide full details of the DTD, but this interface is available because without it, it would be impossible to access notations and unparsed entities referenced in the body of the document.*

*Notations and unparsed entities are rather specialized facilities in XML, so most SAX applications will not need to use this interface.*

### Receive notification of basic DTD-related events

If a SAX application needs information about notations and unparsed entities, then the application implements this interface and registers an instance with the SAX parser using the parser's setDTDHandler method. The parser uses the instance to report notation and unparsed entity declarations to the application.

The SAX parser may report these events in any order, regardless of the order in which the notations and unparsed entities were declared; however, all DTD events must be reported after the document handler's startDocument event,

and before the first startElement event.

It is up to the application to store the information for future use (perhaps in a hash table or object tree). If the application encounters attributes of type "NOTATION", "ENTITY", or "[ENTITIES](#)", it can use the information that it obtained through this interface to find the entity and/or notation corresponding with the attribute value.

The HandlerBase class provides a default implementation of this interface, which simply ignores the events.

<p><b>notationDecl</b></p> <pre>public void notationDecl (String name, String publicId, String systemId) throws SAXException</pre>	<p>Receive notification of a notation declaration event.</p> <p>It is up to the application to record the notation for later reference, if necessary.</p> <p>If a system identifier is present, and it is a URL, the SAX parser must resolve it fully before passing it to the application.</p> <p><b>Parameters:</b></p> <p>name ? the notation name.</p> <p>publicId?the notation's public identifier, or Null if none was given.</p> <p>systemId?the notation's system identifier, or Null if none was given.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<p><b>unparsedEntityDecl</b></p> <pre>public void unparsedEntityDecl (String name, String publicId, String systemId, String notationName) throws SAXException</pre>	<p>Receive notification of an unparsed entity declaration event.</p> <p>Note that the notation name corresponds to a notation reported by the notationDecl() event. It is up to the application to record the entity for later reference, if necessary.</p> <p>If the system identifier is a URL, the parser must resolve it fully before passing it to the application.</p> <p><b>Parameters:</b></p> <p>name?the unparsed entity's name.</p> <p>publicId?the entity's public identifier, or Null if none was given.</p> <p>systemId?the entity's system identifier (it must always have one).</p> <p>notationName?the name of the associated notation.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>

## Interface org.xml.sax.EntityResolver

When the XML document contains references to external entities, the URL will normally be analyzed automatically by the parser: the relevant file will be located and parsed where appropriate. This interface allows an application to override this behavior. This might be needed, for example, if you want to retrieve a different version of the entity from a local server, or if the entities are cached in memory or stored in a database, or if the entity is really a reference to variable information such as the current date.

When the parser needs to obtain an entity, it calls this interface, which can respond by supplying any InputSource object.

## Basic Interface for Resolving Entities

If a SAX application needs to implement customized handling for external entities, it must implement this interface and register an instance with the SAX parser using the parser's setEntityResolver method.

The parser will then allow the application to intercept any external entities (including the external DTD subset and external parameter entities, if any) before including them.

Many SAX applications will not need to implement this interface, but it will be especially useful for applications that build XML documents from databases or other specialized input sources, or for applications that use URI types other than URLs.

The following resolver would provide the application with a special character stream for the entity with the system identifier "<http://www.myhost.com/today>":

```
import org.xml.sax.EntityResolver;
import org.xml.sax.InputSource;

public class MyResolver implements EntityResolver {
    public InputSource resolveEntity (String publicId, String systemId)
    {
        if (systemId.equals("http://www.myhost.com/today")) {
            // return a special input source
            MyReader reader = new MyReader();
            return new InputSource(reader);
        } else {
            // use the default behaviour
            return null;
        }
    }
}
```

The application can also use this interface to redirect system identifiers to local URIs or to look up replacements in a catalog (possibly by using the public identifier).

The HandlerBase class implements the default behavior for this interface, which is simply always to return Null (to request that the parser use the default system identifier).

## resolveEntity

```
public InputSource resolveEntity(String  
publicId, String systemId) throws  
SAXException, IOException
```

Allow the application to resolve external entities.

The parser will call this method before opening any external entity except the top-level document entity (including the external DTD subset, external entities referenced within the DTD, and external entities referenced within the document element): the application may request that the parser resolve the entity itself, that it use an alternative URI, or that it use an entirely different input source.

Application writers can use this method to redirect external system identifiers to secure and/or local URIs, to look up public identifiers in a catalogue, or to read an entity from a database or other input source (including, for example, a dialog box).

If the system identifier is a URL, the SAX parser must resolve it fully before reporting it to the application.

### Parameters:

publicId?the public identifier of the external entity being referenced, or Null if none was supplied.

systemId?the system identifier of the external entity being referenced.

### Returns:

An InputSource object describing the new input source, or Null to request that the parser open a regular URI connection to the system identifier.

### Throws:

SAXException?any SAX exception, possibly wrapping another exception.

### Throws:

IOException?a Java-specific IO exception, possibly the result of creating a new InputStream or Reader for the InputSource.

## Interface org.xml.sax.ErrorHandler

*You may implement this interface in your application if you want to take special action to handle errors. There is a default implementation provided within the HandlerBase class.*

### Basic Interface for SAX Error Handlers

If a SAX application needs to implement customized error handling, it must implement this interface and then register an instance with the SAX parser using the parser's setErrorHandler method. The parser will then report all errors and warnings through this interface.

The parser shall use this interface instead of throwing an exception: it is up to the application whether to throw an exception for different types of errors and warnings. Note, however, that there is no requirement that the parser

continue to provide useful information after a call to fatalError (in other words, a SAX driver class could catch an exception and report a fatalError).

The HandlerBase class provides a default implementation of this interface, ignoring warnings and recoverable errors and throwing a SAXParseException for fatal errors. An application may extend that class rather than implementing the complete interface itself.

<p><b>error</b></p> <pre>public void error(SAXParseException exception) throws SAXException</pre>	<p>Receive notification of a recoverable error.</p> <p>This corresponds to the definition of "error" in section 1.2 of the W3C XML 1.0 Recommendation. For example, a validating parser would use this callback to report the violation of a validity constraint. The default behavior is to take no action.</p> <p>The SAX parser must continue to provide normal parsing events after invoking this method: it should still be possible for the application to process the document through to the end. If the application cannot do so, then the parser should report a fatal error even if the XML 1.0 Recommendation does not require it to do so.</p> <p><b>Parameters:</b></p> <p>exception?the error information encapsulated in a SAX parse exception.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>
<p><b>fatalError</b></p> <pre>public void fatalError(SAXParseException exception) throws SAXException</pre>	<p>Receive notification of a non-recoverable error.</p> <p>This corresponds to the definition of "fatal error" in section 1.2 of the W3C XML 1.0 Recommendation. For example, a parser would use this callback to report the violation of a well-formedness constraint.</p> <p>The application must assume that the document is unusable after the parser has invoked this method, and should continue (if at all) only for the sake of collecting additional error messages: in fact, SAX parsers are free to stop reporting any other events once this method has been invoked.</p> <p><b>Parameters:</b></p> <p>exception?the error information encapsulated in a SAX parse exception.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p>

<b>warning</b>	Receive notification of a warning.  SAX parsers will use this method to report conditions that are not errors or fatal errors as defined by the XML 1.0 Recommendation. The default behavior is to take no action.  The SAX parser must continue to provide normal parsing events after invoking this method: it should still be possible for the application to process the document through to the end.
	<b>Parameters:</b>  exception?the warning information encapsulated in a SAX parse exception.  <b>Throws:</b>  SAXException?any SAX exception, possibly wrapping another exception.

## Class org.xml.sax.HandlerBase?Deprecated

*This class is supplied with SAX itself: it provides default implementations of most of the methods that would otherwise need to be implemented by the application. If you write classes in your application as subclasses of HandlerBase, you need only code those methods where you want something other than the default behavior.*

### Default Base Class for Handlers

This class implements the default behavior for four SAX interfaces: EntityResolver, DTDHandler, DocumentHandler, and ErrorHandler.

Application writers can extend this class when they need to implement only part of an interface; parser writers can instantiate this class to provide default handlers when the application has not supplied its own.

Note that the use of this class is optional.

In the description below, only the behavior of each method is described. For the parameters and return values, see the corresponding interface definition.

<b>characters</b>  public void characters(char ch[], int start, int length) throws SAXException	By default, do nothing. Application writers may override this method to take specific actions for each chunk of character data (such as adding the data to a node or buffer, or printing it to a file).
<b>endDocument</b>  public void endDocument() throws SAXException	Receive notification of the end of the document.  By default, do nothing. Application writers may override this method in a subclass to take specific actions at the beginning of a document (such as finalizing a tree or closing an output file).

<b>endElement</b>  public void endElement(String name) throws SAXException	By default, do nothing. Application writers may override this method in a subclass to take specific actions at the end of each element (such as finalizing a tree node or writing output to a file).
<b>error</b>  public void error(SAXParseException e) throws SAXException	The default implementation does nothing. Application writers may override this method in a subclass to take specific actions for each error, such as inserting the message in a log file or printing it to the console.
<b>fatalError</b>  public void fatalError(SAXParseException e) throws SAXException	The default implementation throws a SAXParseException. Application writers may override this method in a subclass if they need to take specific actions for each fatal error (such as collecting all of the errors into a single report): in any case, the application must stop all regular processing when this method is invoked, since the document is no longer reliable, and the parser may no longer report parsing events.
<b>ignorableWhitespace</b>  public void ignorableWhitespace(char ch[], int start, int length) throws SAXException	By default, do nothing. Application writers may override this method to take specific actions for each chunk of ignorable whitespace (such as adding data to a node or buffer, or printing it to a file).
<b>notationDecl</b>  public void notationDecl(String name, String publicId, String systemId)	By default, do nothing. Application writers may override this method in a subclass if they wish to keep track of the notations declared in a document.
<b>processingInstruction</b>  public void processingInstruction(String target, String data) throws SAXException	By default, do nothing. Application writers may override this method in a subclass to take specific actions for each processing instruction, such as setting status variables or invoking other methods.
<b>resolveEntity</b>  public InputSource resolveEntity(String publicId, String systemId) throws SAXException	Always return Null, so that the parser will use the system identifier provided in the XML document.  This method implements the SAX default behavior: application writers can override it in a subclass to do special translations such as catalog lookups or URI redirection.
<b>setDocumentLocator</b>  public void setDocumentLocator( Locator locator)	By default, do nothing. Application writers may override this method in a subclass if they wish to store the locator for use with other document events.
<b>startDocument</b>  public void startDocument() throws SAXException	By default, do nothing. Application writers may override this method in a subclass to take specific actions at the beginning of a document (such as allocating the root node of a tree or creating an output file).

<b>startElement</b>	By default, do nothing. Application writers may override this method in a subclass to take specific actions at the start of each element (such as allocating a new tree node or writing output to a file).
<b>unparsedEntityDecl</b>	By default, do nothing. Application writers may override this method in a subclass to keep track of the unparsed entities declared in a document.
<b>warning</b>	The default implementation does nothing. Application writers may override this method in a subclass to take specific actions for each warning, such as inserting the message in a log file or printing it to the console.

## Class org.xml.sax.InputSource

An InputSource object represents a container for the XML document or any of the external entities it references (technically, the main document is itself an entity). The InputSource class is supplied with SAX: generally the application instantiates an InputSource and updates it to say where the input is coming from, and the parser interrogates it to find out where to read the input from.

The InputSource object provides three ways of supplying input to the parser: a system identifier (or URL), a Reader (which delivers a stream of Unicode characters), or an InputStream (which delivers a stream of uninterpreted bytes).

### A Single Input Source for an XML Entity

This class allows a SAX application to encapsulate information about an input source in a single object, which may include a public identifier, a system identifier, a byte stream (possibly with a specified encoding), and/or a character stream.

There are two places that the application will deliver this input source to the parser: as the argument to the Parser.parse method, or as the return value of the EntityResolver.resolveEntity method.

The SAX parser will use the InputSource object to determine how to read XML input. If there is a character stream available, the parser will read that stream directly; if not, the parser will use a byte stream, if available; if neither a character stream nor a byte stream is available, the parser will attempt to open a URI connection to the resource identified by the system identifier.

An InputSource object belongs to the application: the SAX parser shall never modify it in any way (it may modify a copy if necessary).

If you supply input in the form of a Reader or InputStream, it may be useful to supply a system identifier as well. If you do this, the URI will not be used to obtain the actual XML input, but it will be used in diagnostics, and more importantly to resolve any relative URIs within the document, for example entity references.

<b>InputSource</b>	Zero-argument default constructor.
public InputSource()	

<p><b>InputSource</b></p> <pre>public InputSource( String systemId)</pre>	<p>Create a new input source with a system identifier.</p> <p>Applications may use setPublicId to include a public identifier as well, or setEncoding to specify the character encoding, if known.</p> <p>If the system identifier is a URL, it must be fully resolved.</p> <p><b>Parameters:</b></p> <p>systemId?the system identifier (URI).</p>
<p><b>InputSource</b></p> <pre>public InputSource( InputStream byteStream)</pre>	<p>Create a new input source with a byte stream.</p> <p>Application writers may use setSystemId to provide a base for resolving relative URIs, setPublicId to include a public identifier, and/or setEncoding to specify the object's character encoding.</p> <p><b>Parameters:</b></p> <p>byteStream?the raw byte stream containing the document.</p>
<p><b>InputSource</b></p> <pre>public InputSource(Reader characterStream)</pre>	<p>Create a new input source with a character stream.</p> <p>Application writers may use setSystemId() to provide a base for resolving relative URIs, and setPublicId to include a public identifier.</p> <p>The character stream shall not include a byte order mark.</p>
<p><b>setPublicId</b></p> <pre>public void setPublicId (String publicId)</pre>	<p>Set the public identifier for this input source.</p> <p>The public identifier is always optional: if the application writer includes one, it will be provided as part of the location information.</p> <p><b>Parameters:</b></p> <p>publicId?the public identifier as a string.</p>
<p><b>getPublicId</b></p> <pre>public String getPublicId ()</pre>	<p>Get the public identifier for this input source.</p> <p><b>Returns:</b></p> <p>The public identifier, or null if none was supplied.</p>

<p><b>setSystemId</b></p> <pre>public void setSystemId ( String systemId)</pre>	<p>Set the system identifier for this input source.</p> <p>The system identifier is optional if there is a byte stream or a character stream, but it is still useful to provide one, since the application can use it to resolve relative URIs and can include it in error messages and warnings (the parser will attempt to open a connection to the URI only if there is no byte stream or character stream specified).</p> <p>If the application knows the character encoding of the object pointed to by the system identifier, it can register the encoding using the setEncoding method.</p> <p>If the system ID is a URL, it must be fully resolved.</p> <p><b>Parameters:</b></p> <p>systemId?the system identifier as a string.</p>
<p><b>getSystemId</b></p> <pre>public String getSystemId ()</pre>	<p>Get the system identifier for this input source.</p> <p>The getEncoding method will return the character encoding of the object pointed to, or Null if unknown.</p> <p>If the system ID is a URL, it will be fully resolved.</p> <p><b>Returns:</b></p> <p>The system identifier.</p>
<p><b>setByteStream</b></p> <pre>public void setByteStream ( InputStream byteStream)</pre>	<p>Set the byte stream for this input source.</p> <p>The SAX parser will ignore this if there is also a character stream specified, but it will use a byte stream in preference to opening a URI connection itself.</p> <p>If the application knows the character encoding of the byte stream, it should set it with the setEncoding method.</p> <p><b>Parameters:</b></p> <p>byteStream?a byte stream containing an XML document or other entity.</p>
<p><b>getByteStream</b></p> <pre>public InputStream getByteStream()</pre>	<p>Get the byte stream for this input source.</p> <p>The getEncoding method will return the character encoding for this byte stream, or Null if unknown.</p> <p><b>Returns:</b></p> <p>The byte stream, or Null if none was supplied.</p>

<b>setEncoding</b> <pre>public void setEncoding( String encoding)</pre>	Set the character encoding, if known. The encoding must be a string acceptable for an XML encoding declaration (see section 4.3.3 of the XML 1.0 Recommendation). This method has no effect when the application provides a character stream. <b>Parameters:</b> encoding?a string describing the character encoding.
<b>getEncoding</b> <pre>public String getEncoding()</pre>	Get the character encoding for a byte stream or URI. <b>Returns:</b> The encoding, or Null if none was supplied.
<b>setCharacterStream</b> <pre>public void setCharacterStream(Reader characterStream)</pre>	Set the character stream for this input source. If there is a character stream specified, the SAX parser will ignore any byte stream and will not attempt to open a URI connection to the system identifier. <b>Parameters:</b> characterStream?the character stream containing the XML document or other entity.
<b>getCharacterStream</b> <pre>public Reader getCharacterStream()</pre>	Get the character stream for this input source. <b>Returns:</b> The character stream, or Null if none was supplied.

## Interface org.xml.sax.Locator

*This interface provides methods that the application can use to determine the current position in the source XML document.*

## Interface for Associating a SAX Event with a Document Location

If a SAX parser provides location information to the SAX application, it does so by implementing this interface and then passing an instance to the application using the document handler's setDocumentLocator method. The application can use the object to obtain the location of any other document handler event in the XML source document.

Note that the results returned by the object will be valid only during the scope of each document handler method: the application will receive unpredictable results if it attempts to use the locator at any other time.

SAX parsers are not required to supply a locator, but they are very strongly encouraged to do so. If the parser supplies a locator, it must do so before reporting any other document events. If no locator has been set by the time

the application receives the startDocument event, the application should assume that a locator is not available.

<b>getPublicId</b>  public String getPublicId()	Return the public identifier for the current document event.  <b>Returns:</b>  A string containing the public identifier, or Null if none is available.
<b>getSystemId</b>  public String getSystemId()	Return the system identifier for the current document event.  If the system identifier is a URL, the parser must resolve it fully before passing it to the application.  <b>Returns:</b>  A string containing the system identifier, or Null if none is available.
<b>getLineNumber</b>  public int getLineNumber()	Return the line number where the current document event ends. Note that this is the line position of the first character after the text associated with the document event. In practice some parsers report the line number and column number where the event starts.  <b>Returns:</b>  The line number, or -1 if none is available.
<b>getColumnNumber</b>  public int getColumnNumber()	Return the column number where the current document event ends. Note that this is the column number of the first character after the text associated with the document event. The first column in a line is position 1.  <b>Returns:</b>  The column number, or -1 if none is available.

## Interface org.xml.sax.Parser?Deprecated

*Every SAX 1.0 parser must implement this interface. An application parses an XML document by creating an instance of a parser (that is, a class that implements this interface) and calling one of its parse() methods.*

### Basic Interface for SAX Parsers

All SAX parsers must implement this basic interface: it allows applications to register handlers for different types of events and to initiate a parse from a URI, or a character stream.

All SAX parsers must also implement a zero-argument constructor (though other constructors are also allowed).

SAX parsers are reusable but not re-entrant: the application may reuse a parser object (possibly with a different input source) once the first parse has completed successfully, but it may not invoke the parse() methods recursively within a parse.

<p><b>parse</b></p> <pre>public void parse( InputSource source) throws SAXException, IOException</pre>	<p>Parse an XML document.</p> <p>The application can use this method to instruct the SAX parser to begin parsing an XML document from any valid input source (a character stream, a byte stream, or a URI).</p> <p>Applications may not invoke this method while a parse is in progress (they should create a new parser instead for each additional XML document). Once a parse is complete, an application may reuse the same parser object, possibly with a different input source.</p> <p><b>Parameters:</b></p> <p>source?the input source for the top-level of the XML document.</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p> <p><b>Throws:</b></p> <p>IOException?an IO exception from the parser, possibly from a byte stream or character stream supplied by the application.</p>
<p><b>parse</b></p> <pre>public void parse(String systemId) throws SAXException, IOException</pre>	<p>Parse an XML document from a system identifier (URI).</p> <p>This method is a shortcut for the common case of reading a document from a system identifier. It is the exact equivalent of the following:</p> <pre>parse(new InputSource(systemId));</pre> <p>If the system identifier is a URL, it must be fully resolved by the application before it is passed to the parser.</p> <p><b>Parameters:</b></p> <p>systemId?the system identifier (URI).</p> <p><b>Throws:</b></p> <p>SAXException?any SAX exception, possibly wrapping another exception.</p> <p><b>Throws:</b></p> <p>IOException?an IO exception from the parser, possibly from a byte stream or character stream supplied by the application.</p>

<p><b>setDocumentHandler</b></p> <pre>public void setDocumentHandler(     DocumentHandler handler)</pre>	<p>Allow an application to register a document event handler.</p> <p>If the application does not register a document handler, all document events reported by the SAX parser will be silently ignored (this is the default behavior implemented by HandlerBase).</p> <p>Applications may register a new or different handler in the middle of a parse, and the SAX parser must begin using the new handler immediately.</p> <p><b>Parameters:</b></p> <p>handler?the document handler.</p>
<p><b>setDTDHandler</b></p> <pre>public void setDTDHandler( DTDHandler     handler)</pre>	<p>Allow an application to register a DTD event handler.</p> <p>If the application does not register a DTD handler, all DTD events reported by the SAX parser will be silently ignored (this is the default behavior implemented by HandlerBase).</p> <p>Applications may register a new or different handler in the middle of a parse, and the SAX parser must begin using the new handler immediately.</p> <p><b>Parameters:</b></p> <p>handler?the DTD handler.</p>
<p><b>setEntityResolver</b></p> <pre>public void setEntityResolver(     EntityResolver resolver)</pre>	<p>Allow an application to register a custom entity resolver.</p> <p>If the application does not register an entity resolver, the SAX parser will resolve system identifiers and open connections to entities itself (this is the default behavior implemented in HandlerBase).</p> <p>Applications may register a new or different entity resolver in the middle of a parse, and the SAX parser must begin using the new resolver immediately.</p> <p><b>Parameters:</b></p> <p>resolver?the object for resolving entities.</p>

<p><b>setErrorHandler</b></p> <p>public void setErrorHandler( ErrorHandler handler)</p>	<p>Allow an application to register an error event handler.</p> <p>If the application does not register an error event handler, all error events reported by the SAX parser will be silently ignored, except for fatalError, which will throw a SAXException (this is the default behavior implemented by HandlerBase).</p> <p>Applications may register a new or different handler in the middle of a parse, and the SAX parser must begin using the new handler immediately.</p> <p><b>Parameters:</b></p> <p>handler?the error handler.</p>
<p><b>setLocale</b></p> <p>public void setLocale(Locale locale)throws SAXException</p>	<p>Allow an application to request a locale for errors and warnings.</p> <p>SAX parsers are not required to provide localization for errors and warnings; if they cannot support the requested locale, however, they must throw a SAX exception. Applications may not request a locale change in the middle of a parse.</p> <p><b>Parameters:</b></p> <p>locale?a Java Locale object.</p> <p><b>Throws:</b></p> <p>SAXException?throws an exception (using the previous or default locale) if the requested locale is not supported.</p>

◀ PREVIOUS

[< Free Open Study >](#)

NEXT ▶

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Class org.xml.sax.SAXException

*This class is used to represent an error detected during processing either by the parser or by the application.*

## Encapsulate a General SAX Error or Warning

This class can contain basic error or warning information from either the XML parser or the application: a parser writer or application writer can subclass it to provide additional functionality. SAX handlers may throw this exception or any exception subclassed from it.

If the application needs to pass through other types of exceptions, it must wrap those exceptions in a SAXException or an exception derived from a SAXException.

If the parser or application needs to include information about a specific location in an XML document, it should use the SAXParseException subclass.

<b>getMessage</b>  <pre>public String getMessage()</pre>	<p>Return a detailed message for this exception.</p> <p>If there is an embedded exception, and if the SAXException has no detailed message of its own, this method will return the detailed message from the embedded exception.</p> <p><b>Returns:</b></p> <p>The error or warning message.</p>
<b>getException</b>  <pre>public Exception getException()</pre>	<p>Return the embedded exception, if any.</p> <p><b>Returns:</b></p> <p>The embedded exception, or null if there is none.</p>
<b>toString</b>  <pre>public String toString()</pre>	<p>Convert this exception to a string.</p> <p><b>Returns:</b></p> <p>A string version of this exception.</p>

# Class org.xml.sax.SAXParseException

*Extends SAXException. This exception class represents an error or warning condition detected by the parser or by the application. In addition to the basic capability of SAXException, a SAXParseException allows information to be retained about the location in the source document where the error occurred. For an application-detected error, this information might be obtained from the Locator object.*

## Encapsulate an XML Parse Error or Warning

This exception will include information for locating the error in the original XML document. Note that although the

application will receive a SAXParseException as the argument to the handlers in the ErrorHandler interface, the application is not actually required to throw the exception; instead, it can simply read the information in it and take a different action.

Since this exception is a subclass of SAXException, it inherits the ability to wrap another exception.

<p><b>SAXParseException</b></p> <pre>public SAXParseException( String message, Locator locator)</pre>	<p>Create a new SAXParseException from a message and a locator. This constructor is especially useful when an application is creating its own exception from within a DocumentHandler callback.</p> <p><b>Parameters:</b></p> <p>message-the error or warning message. locator-the locator object for the error or warning.</p>
<p><b>SAXParseException</b></p> <pre>public SAXParseException(String message, Locator locator, Exception e)</pre>	<p>Wrap an existing exception in a SAXParseException. This constructor is especially useful when an application is creating its own exception from within a DocumentHandler callback, and needs to wrap an existing exception that is not a subclass of SAXException.</p> <p><b>Parameters:</b></p> <p>message-the error or warning message, or Null to use the message from the embedded exception. locator-the locator object for the error or warning. e-any exception</p>
<p><b>SAXParseException</b></p> <pre>public SAXParseException(String message, String publicId, String systemId, int lineNumber, int columnNumber)</pre>	<p>Create a new SAXParseException. This constructor is most useful for parser writers. If the system identifier is a URL, the parser must resolve it fully before creating the exception.</p> <p><b>Parameters:</b></p> <p>message-the error or warning message. publicId-the public identifier of the entity that generated the error or warning. systemId-the system identifier of the entity that generated the error or warning. lineNumber-the line number of the end of the text that caused the error or warning. columnNumber-the column number of the end of the text that caused the error or warning.</p>

<p><b>SAXParseException</b></p> <pre>public SAXParseException(String message, String publicId, String systemId, int lineNumber, int columnNumber, Exception e)</pre>	<p>Create a new SAXParseException with an embedded exception.</p> <p>This constructor is most useful for parser writers who need to wrap an exception that is not a subclass of SAXException.</p> <p>If the system identifier is a URL, the parser must resolve it fully before creating the exception.</p> <p><b>Parameters:</b></p> <ul style="list-style-type: none"><li>message-the error or warning message, or Null to use the message from the embedded exception.</li><li>publicId-the public identifier of the entity that generated the error or warning.</li><li>systemId-the system identifier of the entity that generated the error or warning.</li><li>lineNumber-the line number of the end of the text that caused the error or warning.</li><li>columnNumber-the column number of the end of the text that caused the error or warning.</li><li>e-another exception to embed in this one.</li></ul>
<p><b>getPublicId</b></p> <pre>public String getPublicId()</pre>	<p>Get the public identifier of the entity where the exception occurred.</p> <p><b>Returns:</b></p> <p>A string containing the public identifier, or Null if none is available.</p>
<p><b>getSystemId</b></p> <pre>public String getSystemId()</pre>	<p>Get the system identifier of the entity where the exception occurred. <i>Note that the term "entity" includes the top-level XML document.</i></p> <p>If the system identifier is a URL, it will be resolved fully.</p> <p><b>Returns:</b></p> <p>A string containing the system identifier, or Null if none is available.</p>
<p><b>getLineNumber</b></p> <pre>public int getLineNumber()</pre>	<p>The line number of the end of the text where the exception occurred.</p> <p><b>Returns:</b></p> <p>An integer representing the line number, or -1 if none is available.</p>

### **getColumnNumber**

```
public int getColumnNumber()
```

The column number of the end of the text where the exception occurred.  
The first column in a line is position 1.

#### **Returns:**

An integer representing the column number, or -1 if none is available.

## **Class org.xml.sax.SAXNotRecognizedException (SAX 2)**

Exception class for an unrecognized identifier-an XML reader will throw this exception when it finds an unrecognized feature or property identifier; SAX applications and extensions may use this class for other, similar purposes.

### **SAXNotRecognizedException**

```
public SAXNotRecognizedException  
(String message)
```

Construct a new exception with the given message.

#### **Parameters**

message-the text message of the exception.

This class has also inherited a lot of methods from other classes. These are summarized below:

Methods inherited from class org.xml.sax.SAXException:

- getException
- getMessage
- toString

Methods inherited from class java.lang.Throwable:

- fillInStackTrace
- getLocalizedMessage
- printStackTrace

Methods inherited from class java.lang.Object:

- equals
- getClass
- hashCode

- notify
- 
- notifyAll
- 
- wait

## Class org.xml.sax.SAXNotSupportedException (SAX 2)

Exception class for an unsupported operation-an XMLReader will throw this exception when it recognizes a feature or property identifier, but cannot perform the requested operation (setting a state or value). Other SAX2 applications and extensions may use this class for similar purposes.

<b>SAXNotSupportedException</b>	Construct a new exception with the given message.
public SAXNotSupportedException (String message)	<b>Parameters</b>  message-the text message of the exception.

This class has also inherited a lot of methods from other classes. These are summarized below:

Methods inherited from class org.xml.sax.SAXException:

- 
- getException
- 
- getMessage
- 
- toString

Methods inherited from class java.lang.Throwable:

- 
- fillInStackTrace
- 
- getLocalizedMessage
- 
- printStackTrace

Methods inherited from class java.lang.Object:

- 
- equals
- 
- getClass
-

hashCode

•

notify

•

notifyAll

•

wait

## Interface org.xml.sax.XMLFilter (SAX 2)

This interface is like the reader, except it is used to read documents from a source other than a document or database. It can also modify events on the way to an application (extends XMLReader).

Interface for an XML filter-an XML filter is like an XML reader, except that it obtains its events from another XML reader rather than a primary source like an XML document or database. Filters can modify a stream of events as they pass on to the final application.

The XMLFilterImpl helper class provides a convenient base for creating SAX2 filters, by passing on all EntityResolver, DTDHandler, ContentHandler, and ErrorHandler events automatically.

<b>setParent</b>  public void setParent( XMLReader parent)	Set the parent reader.  This method allows the application to link the filter to a parent reader (which may be another filter). The argument may not be Null.  <b>Parameters</b>  parent-the parent reader.
<b>getParent</b>  public XMLReader getParent	Get the parent reader.  This method allows the application to query the parent reader (which may be another filter). It is generally a bad idea to perform any operations on the parent reader directly: they should all pass through this filter.  <b>Returns:</b>  The parent filter, or Null if none has been set.

## Interface org.xml.sax.XMLReader (SAX 2-Replaces Parser)

*Every SAX 2.0 parser must implement this interface for reading documents using callbacks. An application parses an XML document by creating an instance of a parser (that is, a class that implements this interface) and calling one of its parse() methods.*

Interface for reading an XML document using callbacks. XMLReader is the interface that an XML parser's SAX2 driver must implement. This interface allows an application to set and query features and properties in the parser, to register event handlers for document processing, and to initiate a document parse.

All SAX interfaces are assumed to be synchronous: the parse methods must not return until parsing is complete, and readers must wait for an event-handler callback to return before reporting the next event.

This interface replaces the (now deprecated) SAX 1.0 parser interface. The XMLReader interface contains two important enhancements over the old Parser interface:

- it adds a standard way to query and set features and properties
- it adds namespace support, which is required for many higher-level XML standards

There are adapters available to convert a SAX1 Parser to a SAX2 XMLReader and vice-versa.

<p><b>getFeature</b></p> <pre>public boolean getFeature(String name)</pre>	<p>Look up the value of a feature</p> <p>The feature name is any fully-qualified URI. It is possible for an XMLReader to recognize a feature name but be unable to return its value; this is especially true in the case of an adapter for a SAX1 Parser, which has no way of knowing whether the underlying parser is performing validation or expanding external entities.</p> <p><b>Parameters</b></p> <p>name-the feature name, which is a fully-qualified URI</p> <p><b>Returns:</b></p> <p>The current state of the feature (true or false)</p> <p><b>Throws:</b></p> <p>SAXNotRecognizedException</p> <p>SAXNotSupportedException</p>
<p><b>setFeature</b></p> <pre>public void setFeature(String name, boolean value)</pre>	<p>Set the state of a feature.</p> <p>The feature name is any fully-qualified URI. It is possible for an XMLReader to recognize a feature name but be unable to set its value; this is especially true in the case of an adapter for a SAX1 Parser, which has no way of affecting whether the underlying parser is validating, for example.</p> <p><b>Parameters</b></p> <p>name-the feature name, which is a fully-qualified URI</p> <p>state-the requested state of the feature (true or false)</p> <p><b>Throws:</b></p> <p>SAXNotRecognizedException</p> <p>SAXNotSupportedException</p>

<p><b>getProperty</b></p> <pre>public Object getProperty(String name)</pre>	<p>Look up the value of a property.</p> <p>The property name is any fully-qualified URI. It is possible for an XMLReader to recognize a property name but be unable to return its state; this is especially true in the case of an adapter for a SAX1 Parser.</p> <p><b>Parameters</b></p> <p>name-the feature name, which is a fully-qualified URI</p> <p><b>Returns:</b></p> <p>The current value of the property</p> <p><b>Throws:</b></p> <p>SAXNotRecognizedException</p> <p>SAXNotSupportedException</p>
<p><b>setProperty</b></p> <pre>public void setProperty(String name, Object value)</pre>	<p>Set the value of a property.</p> <p>The property name is any fully-qualified URI. It is possible for an XMLReader to recognize a property name but to unable to set its value; this is especially true in the case of an adapter for a SAX1 Parser.</p> <p><b>Parameters</b></p> <p>name-the feature name, which is a fully-qualified URI</p> <p>state-the requested value for the property</p> <p><b>Throws:</b></p> <p>SAXNotRecognizedException</p> <p>SAXNotSupportedException</p>

<b>setEntityResolver</b>  public void setEntityResolver(EntityResolver resolver)	<p>Allow an application to register an entity resolver.</p> <p>If the application does not register an entity resolver, the XMLReader will perform its own default resolution.</p> <p>Applications may register a new or different resolver in the middle of a parse, and the SAX parser must begin using the new resolver immediately.</p> <p><b>Parameters</b></p> <p>resolver-the entity resolver.</p> <p><b>Throws:</b></p> <p>java.lang.NullPointerException-if the resolver argument is Null.</p>
<b>getEntityResolver</b>  public EntityResolver getEntityResolver()	<p>Return the current entity resolver.</p> <p><b>Returns:</b></p> <p>The current entity resolver, or Null if none has been registered.</p>
<b>setDTDHandler</b>  public void setDTDHandler(DTDHandler handler)	<p>Allow an application to register a DTD event handler.</p> <p>If the application does not register a DTD handler, all DTD events reported by the SAX parser will be silently ignored.</p> <p>Applications may register a new or different handler in the middle of a parse, and the SAX parser must begin using the new handler immediately.</p> <p><b>Parameters</b></p> <p>handler-the DTD handler.</p> <p><b>Throws:</b></p> <p>java.lang.NullPointerException-if the handler argument is Null.</p>
<b>getDTDHandler</b>  public DTDHandler getDTDHandler	<p>Return the current DTD handler.</p> <p><b>Returns:</b></p> <p>The current DTD handler, or Null if none has been registered.</p>

<p><b>setContentHandler</b></p> <pre>public void setContentHandler (ContentHandler handler)</pre>	<p>Allow an application to register a content event handler.</p> <p>If the application does not register a content handler, all content events reported by the SAX parser will be silently ignored. Applications may register a new or different handler in the middle of a parse, and the SAX parser must begin using the new handler immediately.</p> <p><b>Parameters</b></p> <p>handler-the content handler.</p> <p><b>Throws:</b></p> <p>java.lang.NullPointerException-if the handler argument is Null.</p>
<p><b>getContentHandler</b></p> <pre>public getContentHandler</pre>	<p>Return the current content handler.</p> <p><b>Returns:</b></p> <p>The current content handler, or Null if none has been registered.</p>
<p><b>setErrorHandler</b></p> <pre>public void setErrorHandler( ErrorHandler handler)</pre>	<p>Allow an application to register an error event handler.</p> <p>If the application does not register an error handler, all error events reported by the SAX parser will be silently ignored; however, normal processing may not continue. It is highly recommended that all SAX applications implement an error handler to avoid unexpected bugs.</p> <p>Applications may register a new or different handler in the middle of a parse, and the SAX parser must begin using the new handler immediately.</p> <p><b>Parameters</b></p> <p>handler-the error handler.</p> <p><b>Throws:</b></p> <p>java.lang.NullPointerException-if the handler argument is Null.</p>
<p><b>getErrorHandler</b></p> <pre>public ErrorHandler getErrorHandler</pre>	<p>Return the current error handler.</p> <p><b>Returns:</b></p> <p>The current error handler, or Null if none has been registered.</p>

<p><b>parse</b></p> <pre>public void parse( InputSource input)</pre>	<p>Parse an XML document.</p> <p>The application can use this method to instruct the XML reader to begin parsing an XML document from any valid input source (a character stream, a byte stream, or a URI).</p> <p>Applications may not invoke this method while a parse is in progress (they should create a new XMLReader instead for each nested XML document). Once a parse is complete, an application may reuse the same XMLReader object, possibly with a different input source.</p> <p>During the parse, the XMLReader will provide information about the XML document through the registered event handlers.</p> <p>This method is synchronous: it will not return until parsing has ended. If a client application wants to terminate parsing early, it should throw an exception.</p> <p><b>Parameters</b></p> <p>source-the input source for the top-level of the XML document.</p> <p><b>Throws:</b></p> <p>SAXException-any SAX exception, possibly wrapping another exception.</p> <p>java.io.IOException-an IO exception from the parser, possibly from a byte stream or character stream supplied by the application.</p>
<p><b>parse</b></p> <pre>public void parse( String systemId)</pre>	<p>Parse an XML document from a system identifier (URI).</p> <p>If the system identifier is a URL, it must be fully resolved by the application before it is passed to the parser.</p> <p><b>Parameters</b></p> <p>systemId- the system identifier (URI).</p> <p><b>Throws:</b></p> <p>SAXException-any SAX exception, possibly wrapping another exception.</p> <p>java.io.IOException-an IO exception from the parser, possibly from a byte stream or character stream supplied by the application.</p>

All XMLReaders are required to recognize the <http://xml.org/sax/features/namespaces> and the <http://xml.org/sax/features/namespace-prefixes> feature names.

Some feature values may be available only in specific contexts, such as before, during, or after a parse.

Implementors are free (and encouraged) to invent their own features, using names built on their own URIs.

# Appendix G: Useful Web Resources

This appendix is a compilation of some of the most useful web-based resources relating to XML and the surrounding technologies.

## W3C web site

<http://www.w3.org>

## XML specification

<http://www.w3.org/TR/2000/REC-xml-20001006>

## Microsoft MSXML parser

<http://msdn.microsoft.com/downloads/webtechnology/xml/msxml.asp>

## Vivid Creations XML tools, including ActiveDOM

<http://www.vivid-creations.com>

## Apache Software Foundation's Xerces parser and Xalan XSLT processor

<http://xml.apache.org/>

## IBM xml4j parser

<http://alphaworks.ibm.com/tech/xml4j>

## Mike Kay's Saxon XSLT processor

<http://saxon.sourceforge.net/>

XML tools

<http://www.xmlsoftware.com/parsers/>

## CSS specifications

<http://www.w3.org/Style/CSS/#specs>

## XSL

<http://www.w3.org/Style/XSL>

## XSLT specification

<http://www.w3.org/TR/xslt>

## XSL Formatting Objects specification

<http://www.w3.org/TR/xsl>

## XSL tutorials

<http://www.xsldinfo.com/tutorials/>

[http://www.xslt.com/resources\\_tutorials.htm](http://www.xslt.com/resources_tutorials.htm)

## DOM

<http://www.w3.org/TR/DOM-Level-2/>

<http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210/core.html>

## SAX

<http://www.megginson.com/SAX/index.html>

## SOAP

<http://www.w3.org/TR/SOAP/>

## Java Development Kit

<http://java.sun.com/products/?frontpage-main>

## XML Namespaces specification

<http://www.w3.org/TR/1999/REC-xml-names-19990114/>

## URI syntax

<http://www.ietf.org/rfc/rfc2396.txt>

## URN syntax

<http://www.ietf.org/rfc/rfc2141.txt>

## URL syntax

<http://www.ietf.org/rfc/rfc1738.txt>

## XML Schema Part 0: Primer

<http://www.w3.org/TR/xmlschema-0>

## **XML Schema Part 1: Structures**

<http://www.w3.org/TR/xmlschema-1>

## **XML Schema Part 2: Datatypes**

<http://www.w3.org/TR/xmlschema-2>

## **XPath specification**

<http://www.w3.org/TR/xpath>

## **XPointer specification**

<http://www.w3.org/TR/xptr>

## **XLink specification**

<http://www.w3.org/TR/xlink/>

## **XML Query Working Group**

<http://www.w3.org/TR/xmlquery-req>

## **XML databases ? eXcelon**

<http://www.exceloncorp.com/>

## **Oracle's Technology Network and XML Developer's Kit**

<http://technet.oracle.com/tech/xml/>

## **XML-RPC protocol**

<http://www.xmlrpc.com/>

## **DevelopMentor**

<http://www.develop.com/soap/issues.htm>

## **RDF and Dublin Core standard specifications**

<http://www.w3.org/RDF/>

## **Tim Bray's "RDF and Metadata"**

<http://www.xml.com/pub/98/06/rdf.html>

## John Cowan's "RDF Made Easy"

<http://www.ccil.org/~cowan/XML/rdf-made-easy.ppt>

## Schematron web site

<http://www.ascc.net/xml/resource/schematron/schematron.html>

## Schematron XSLT style sheet

<http://www.ascc.net/xml/resource/schematron/schematron.xsl>

## Miloslav Nic's Schematron tutorials

<http://zvon.vscht.cz/HTMLonly/SchematronTutorial/General/contents.html>

## Visual Basic code generator for XML Schema documents

<http://msdn.microsoft.com/xml/articles/generat.asp>

## XML-DEV mailing list (now hosted by OASIS)

<http://www.oasis-open.org/>

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## Symbols

#PCDATA keyword  
mixed content model, [Mixed Content](#)  
&amp, & character, [Escaping Characters](#)  
&apos, ' character, [Escaping Characters](#), [Built-in Entities](#)  
&gt, > character, [Escaping Characters](#)  
&lt, < character, [Escaping Characters](#), [Built-in Entities](#)  
&mp, & character, [Built-in Entities](#)  
&nbs, white space, [Whitespace in PCDATA](#)  
&qt, > character, [Built-in Entities](#)  
&quot, character, [Escaping Characters](#), [Built-in Entities](#)  
( ), functions in XPath, [XPath Functions](#)  
\* cardinality indicator, [Cardinality](#)  
mixed content model, element declarations, [Mixed Content](#)  
\* node test, XPath, [\\*](#)  
\* wildcard character, CSS-selector syntax, [Centralizing with the <style> Element](#)  
\*, XPath wildcard, [<xselement>](#)  
+ cardinality indicator, [Cardinality](#)  
, document root in XPath, [The Document Root](#)  
// recursive descent operator, [Try It Out-Attributes and Attribute Sets in Action](#)  
<!--...-->, comments, [Comments](#)  
<?...?> tags, processing instructions, [Processing Instructions](#)  
<?xml...?> tags, XML declarations, [XML Declaration](#)  
<xsl  
for-each> element, [<xsl:for-each>](#), [<xsl:for-each>](#), [<xsl:for-each>](#)  
compared to <xsl\, [<xsl:for-each>](#)  
? cardinality indicator, [Cardinality](#)  
@ XPath pattern, match any attribute, [<xselement>](#)  
[ ], location paths, XPath, [Filtering XPath Expressions and Patterns](#)  
\  
(colon character), namespaces, [How XML Namespaces Work](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

**A**

abort method, IMXReaderControl interface, [The Microsoft Way with SAX](#)  
absolute value, position property, CSS, [Integrating Position](#)  
ActiveDOM, [Parsing XML](#)  
ActiveX Data Objects, see [ado](#)  
actor attribute  
<Header> element, SOAP, [The actor Attribute](#)  
actuate attribute, XLink, [XLink Attributes](#), [Behavior Attributes \(actuate and show\)](#)  
arc-type elements, [Arcs](#)  
address book stylesheet design  
setting the variables, [User Interface Components](#), [The \\$params Variable](#), [The \\$searchlist Variable](#), [The \\$app Variable](#)  
user interface components, [User Interface Components](#)  
ADO, [Retrieving Data from Multiple Tables](#)  
COM based technology, [Using XML in an n-Tier Application](#)  
Connection object, [Using XML in an n-Tier Application](#)  
Recordset object, [Using XML in an n-Tier Application](#)  
SQL queries, capturing results of, [Retrieving Data from Multiple Tables](#)  
alert() function, [Hierarchies in HTML](#)  
alias  
use of, [Retrieving Data from Multiple Tables](#)  
all declaration, XML Schemas content model, [<all>](#)  
minOccurs and maxOccurs attributes, [<all>](#)  
Amaya browser  
CSS Level 2 compliance, [Enter Cascading Style Sheets](#)  
American Standard Code for Information Interchange (ASCII), see [ascii](#)  
ancestor axis, XPath, [ancestor](#)  
ancestor-or-self axis, XPath, [ancestor-or-self](#)  
ancestors, [Hierarchies in XML](#)  
anchors, [Appending XPointer Expressions to URIs](#)  
annotation declarations, XML Schemas, [Annotations](#)  
anonymous complex types, [complexType](#)  
ANY keyword  
element declarations, [Any Content](#)  
any> declaration  
element wildcards, [Element Wildcards](#)  
values allowed by namespace attribute, [Element Wildcards](#)  
Apache Tomcat web application server, [Application Server Architecture](#)  
HTTP requests, [Servlet Overview](#)  
installing, [Installing the Application](#)  
requires a Java Runtime Environment (JRE) installation on your computer, [Application Server Architecture](#)  
Apache Xerces, [Parsing XML](#), [Where to Get SAX](#)  
Apache's Java SOAP implementation, [Building the Client](#)  
API  
SAX, [Chapter 9: The Simple API for XML \(SAX\)](#)  
appendChild method, Node interface, [The appendChild\(\) Method](#)  
appendData method, CharacterData interface, [Modifying Strings](#)

appinfo element, [Annotations](#)  
application server architecture  
online address book case study, [Application Server Architecture](#)  
applications  
communication between, using XML, [Using XML in an n-Tier Application](#)  
arc-type elements, XLink, [The type Attribute](#)  
adding to extended link, [Try It Out-Adding Arcs to the Extended Link](#)  
extended links, [Arcs](#)  
arcs, XLink, [XLink](#)  
ASCII, [Encoding](#)  
ATTLIST declarations, [Attribute Declarations](#)  
adding to DTD, [How It Works](#)  
Attr interface, DOM, [Attr](#)  
attribute axis, XPath, [attribute](#)  
attribute declarations, [Attribute Declarations](#), [<attribute>](#), [<attribute>](#)  
attribute names, [Attribute Names](#)  
attribute types, [Attribute Types](#)  
attribute wildcards, [Attribute Wildcards](#)  
default and fixed values, [Default and Fixed Values](#)  
local types, [Creating a Local Type](#)  
naming, [Naming Attributes](#)  
overriding default attribute qualification, [Attribute Qualified Form](#)  
references to unparsed entities, [ENTITY and ENTITIES](#)  
referring to existing global attribute, [Referring to an Existing Global Attribute](#)  
specifying multiple attributes, [Specifying Multiple Attributes](#)  
use attribute, [Attribute Use](#)  
using global type, [Using a Global Type](#)  
attribute names  
attribute declarations, [Attribute Names](#)  
attribute order, [Attributes](#)  
attribute types  
attribute declarations, [Attribute Types](#)  
table, [Attribute Types](#)  
XSLT elements, tabulated (Appendix), [Types](#)  
attribute value declarations, [Attribute Value Declarations](#)  
default values, [Default Values](#)  
fixed values, [Fixed Values](#)  
implied values, [Implied Values](#)  
required values, [Required Values](#)  
attribute wildcards, attribute declarations, [Attribute Wildcards](#)  
namespace attribute, [Attribute Wildcards](#)  
attributeFormDefault attribute  
schema element, [Declaration Defaults](#)  
attributeGroup declaration, XML Schemas, [<attributeGroup>](#)  
AttributeList interface  
deprecated in SAX 2.0, [Appendix F: SAX 2.0: The Simple API for XML](#), [Interface org.xml.sax.AttributeList-DDeprecated](#)  
interface for elements attribute specifications, [Interface org.xml.sax.AttributeList-DDeprecated](#)  
attributes  
adding, [Attributes](#)  
adding dynamically, [<xsl:attribute>](#) and [<xsl:attribute-set>](#)  
global attributes, [Namespaces and Attributes](#)  
listed for each XSLT element (Appendix), [Elements](#)  
namespaces \r c7att, [Namespaces and Attributes](#)  
naming, [Attributes](#)

reasons for using, [Why Use Attributes?](#)  
related groups of, [Related Groups of Attributes](#)  
sets, [Related Groups of Attributes](#)  
use of quotation marks, [Attributes](#)  
using, [Namespaces and Attributes](#)  
vs. elements \r c2attvsel, [Why Use Attributes?](#)  
without values, [Putting SQL Server to Work](#)  
attributes \r c2att, [Attributes](#)  
Attributes class, SAX  
getLength method, [Extracting Attributes](#)  
getType method, [Extracting Attributes](#)  
getValue method, [Extracting Attributes](#)  
get0Name method, [Extracting Attributes](#)  
Attributes interface, SAX 2  
additional features, [org.xml.sax.Attributes \(SAX 2-Replaces AttributeList\)](#)  
Attributes interface, SAX2  
additional features, [Appendix F: SAX 2.0: The Simple API for XML](#)  
getIndex() methods, [org.xml.sax.Attributes \(SAX 2-Replaces AttributeList\)](#)  
axes, XPath, [Axes](#)  
full list of axes (Appendix), [Axes](#)  
implementation in XPath 1.0 and MSXML, [Appendix B: XPath Reference](#)  
primary node types, [Axes](#)  
axis names, XPath, [XPath Axis Names](#)

---

[!\[\]\(c573702f2de1076cc60f68b0a12c779b\_img.jpg\) PREVIOUS](#)

[!\[\]\(65ffedf6f2ec9ba9bb28d3b96a62fc36\_img.jpg\)< Free Open Study >](#)

[!\[\]\(50ed95fe3e7f8434787bac1d58567866\_img.jpg\) NEXT >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## B

bare names syntax, XPointer, [XPointer Shorthand Syntaxes](#), [Bare Names Syntax](#)  
behavior attributes, XLink, [Behavior Attributes \(actuate and show\)](#)  
binary files, [Binary Files](#)  
binding, constants, [Variables, Constants, and Named Templates](#)  
bits, [Binary Files](#)  
block value, CSS primary containers, [Deploying CSS with XML](#)  
Body> element, SOAP, [<Body>](#)  
Boolean expressions  
conditional processing, [Conditional Processing with <xsl:if> and <xsl:choose>](#)  
boolean() function, XPath, [Functions](#)  
branches, [Hierarchies in XML](#)  
browser styles, cascading, [The Basics of HTML/XHTML/CSS](#)  
built in entities, [Built-in Entities](#)  
references to, [References to Built-in Entities](#)  
Business Objects layer, [n-Tier Architecture](#)  
business to business e-commerce, [e-Commerce](#)  
XSLT, [Why is XSLT So Important for e-Commerce?](#)  
business to customer transactions, [e-Commerce](#)  
B2B e-commerce, [e-Commerce](#)  
XSLT, [Why is XSLT So Important for e-Commerce?](#)  
B2C transactions, [e-Commerce](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## C

Cagle, Kurt, [Deploying CSS with XML](#)  
cardinality  
element declarations, [Cardinality](#)  
cardinality indicators, [Cardinality](#)  
\*, [Cardinality](#)  
+, [Cardinality](#)  
?, [Cardinality](#)  
content model, [Cardinality](#)  
elements, [Cardinality](#)  
table, [Cardinality](#)  
carriage return character, [End-of-Line Whitespace](#)  
Cascading Style Sheets, see [css](#)  
Cascading Stylesheets \t See CSS, [What are the Pieces that Make Up XML?](#)  
case study 1, online address book, [Possible Business Uses for XSLT-Driven Web Applications](#)  
case study 2, XML web services, [Case Study 2 - XML Web Services](#)  
case-order attribute  
xsl, [Sorting the Result Tree](#)  
case-sensitivity, [Case-Sensitivity](#)  
namespaces, [The XML Schema Namespace](#)  
potential problems, [Case-Sensitivity](#)  
casting, [Implementing Multiple Interfaces](#)  
catch blocks  
exception handling, [Exceptions](#)  
CDATA attribute type, [CDATA](#)  
CDATA sections \r c2cdata, [CDATA Sections](#)  
cdata-section-elements attribute  
xsl, [<xsl:output>](#)  
CDATASection interface, DOM, [CDATASection](#)  
ceiling() function, XPath, [Functions](#)  
Chameleon Components, [<include>](#)  
character code, [Important](#)  
Character Data \t See CDATA, [CDATA Sections](#)  
character data, extracting, [Extracting Character Data](#)  
character encoding, [Important](#)  
character entities  
references to, [References to Character Entities](#)  
character references, [Escaping Characters](#)  
character-points, XPointer, [Points](#)  
CharacterData interface, DOM, [Working With Text](#), [CharacterData](#)  
appendData method, [Modifying Strings](#)  
data property, [Handling Complete Strings](#)  
deleteData method, [Modifying Strings](#)  
insertData method, [Modifying Strings](#)  
length property, [Handling Complete Strings](#)  
replaceData method, [Modifying Strings](#)  
substringData method, [Handling Substrings](#)

characters method, ContentHandler interface, [How to Receive SAX Events](#), [Extracting Character Data](#)  
Chemical Markup Language (CML) DTD, [Sharing Vocabularies](#)  
Chemical Markup Language \t See CML, [Why "Extensible"?](#)  
child axis, XPath, [child](#)  
child elements  
default namespaces for, [Declaring Namespaces on Descendants](#)  
child sequence syntax, XPointer, [XPointer Shorthand Syntaxes](#), [Child Sequence Syntax](#)  
children, [Hierarchies in XML](#)  
choice declaration, XML Schemas content model, [<choice>](#)  
class attribute, CSS, [Working with Classes](#)  
assigning rule to, [Working with Classes](#)  
creating class and applying to element, [Working with Classes](#)  
CSS-selector syntax, [Centralizing with the <style> Element](#)  
classes, [Implementing Interfaces](#)  
classes, SAX 2.0, full list, [Appendix F: SAX 2.0: The Simple API for XML](#)  
CLASSPATH and PATH  
modifying, [Modify the Path and Class Path](#)  
CML, [Why "Extensible"?](#)  
code generators, Oracle, [Code Generators](#)  
collapsed ranges, XPointer, [Ranges](#)  
colon character ( : )  
) namespaces, [How XML Namespaces Work](#)  
color attribute  
CSS basics, HTML and XHTML, [Element Styles and the Style Attribute](#)  
columns, [Databases, Yesterday and Today](#)  
COM, [DCOM](#), [Using XML in an n-Tier Application](#)  
functionality in MS Office, [DCOM](#)  
MS specific, [DCOM](#)  
combined expressions, CSS-selector syntax, [Centralizing with the <style> Element](#)  
Comment interface, DOM, [Comment](#)  
comment() node test, XPath, [comment\(\)](#)  
comments  
displaying, [Comments](#)  
comments \r c2comm, [Comments](#)  
Common Object Request Broker Architecture, see [corba](#)  
complexType definitions  
element declarations, [<complexType>](#)  
empty content models, [<complexType>](#)  
sequence declaration, XML Schemas content model, [<sequence>](#)  
complexTypes, [Important](#)  
extending definitions, [Extending <complexType> Definitions](#), [Extending <complexType> Definitions](#)  
redefining contents, [<redefine>](#)  
restricting definitions, [Restricting <complexType> Definitions](#)  
Component Object Model, see [com](#)  
compression, [So What is XML? Why Use Attributes?](#)  
concat() function, Xpath, [Modes](#), [Functions](#)  
conditional processing, XSLT, [Conditional Processing with <xsl:if> and <xsl:choose>](#)  
using, [Conditional Processing with <xsl:if> and <xsl:choose>](#)  
Connection object, ADO, [Using XML in an n-Tier Application](#)  
ConnectionString property, [Using XML in an n-Tier Application](#)  
Execute method, [Using XML in an n-Tier Application](#)  
ConnectionString property, Connection object, ADO, [Using XML in an n-Tier Application](#)  
constants in XSLT, [Variables, Constants, and Named Templates](#)  
binding, [Variables, Constants, and Named Templates](#)  
constraining facets

enumeration facet, [Constraining Facets](#)  
fractionDigits facet, [Constraining Facets](#)  
length facet, [Constraining Facets](#)  
maxExclusive facet, [Constraining Facets](#)  
maxInclusive facet, [Constraining Facets](#)  
maxlength facet, [Constraining Facets](#)  
minExclusive facet, [Constraining Facets](#)  
minInclusive facet, [Constraining Facets](#)  
minlength facet, [Constraining Facets](#)  
pattern facet, [Constraining Facets](#)  
table, [`<restriction>`](#)  
totalDigits facet, [Constraining Facets](#)  
whiteSpace facet, [Constraining Facets](#)  
container node, XPointer, [Points](#)  
contains() function, XPath, [Functions](#)  
content models, DTDs  
any content, [Any Content](#)  
element content, [Element Content](#), [Sequences](#), [Choices](#), [Combining Sequences and Choices](#)  
element declarations, [Element Declarations](#), [Cardinality](#)  
empty content, [Empty Content](#)  
mixed content, [Mixed Content](#)  
content models, XML Schemas, [Content Models](#)  
all declaration, [`<all>`](#)  
choice declaration, [`<choice>`](#)  
group declaration, [`<group>`](#), [`<group>`](#)  
sequence declaration, [`<sequence>`](#)  
ContentHandler interface, SAX, [How to Receive SAX Events](#)  
characters method, [How to Receive SAX Events](#), [Extracting Character Data](#), [Extracting Attributes](#)  
DefaultHandler class, [How to Receive SAX Events](#)  
endDocument method, [How to Receive SAX Events](#)  
endElement method, [How to Receive SAX Events](#), [Extracting Character Data](#), [Extracting Attributes](#)  
ignorableWhitespace method, [ignorableWhitespace](#)  
processingInstruction method, [processingInstruction](#)  
setDocumentLocator method, [More About Errors?Using the Locator Object](#), [Try It Out?Using the Locator](#)  
startDocument method, [How to Receive SAX Events](#)  
startElement method, [How to Receive SAX Events](#), [Extracting Attributes](#), [Try It Out?Using the Locator](#)  
ContentHandler interface, SAX 2  
receives notification of general document events, [Interface org.xml.sax.ContentHandler \(SAX 2?Replaces DocumentHandler\)](#)  
ContentHandler interface, SAX2  
additional features, [Appendix F: SAX 2.0: The Simple API for XML](#)  
skippedEntity() method, [Interface org.xml.sax.ContentHandler \(SAX 2?Replaces DocumentHandler\)](#)  
ContentType property, Response object, [Using XML in an n-Tier Application](#)  
context node, XPath, [XPath](#)  
copying, [`<xsl:copy>`](#)  
templates, effect of, [How Do Templates Affect the Context Node?](#)  
context nodes  
XPath axis definition in terms of, [Appendix B: XPath Reference](#)  
copying source tree, [Copying Sections of the Source Tree to the Result Tree](#)  
CORBA, [IIOP](#), [Using XML in an n-Tier Application](#)  
platform-neutral, [IIOP](#)  
underlying functionality, [IIOP](#)  
count() function, XPath, [Functions](#)  
Cover Pages  
XML resource, [Sharing Vocabularies](#)

createAttribute method, Document interface, [Creating XML Documents Through the DOM](#)  
createElement method, Document interface, [Creating XML Documents Through the DOM](#)  
createNodeType method, Document interface, [Creating XML Documents Through the DOM](#)  
CSS, [Chapter 11: Displaying XML](#)  
displaying XML, [Chapter 11: Displaying XML](#)  
future of XML, [CSS and the Future of XML](#)  
purposes, [Enter Cascading Style Sheets](#)  
rule based, [Enter Cascading Style Sheets](#)  
support in IE, [Linking Stylesheets in HTML](#)  
XML, displaying, [What are the Pieces that Make Up XML?](#)  
CSS and XSLT  
applying different themes with CSS and XSLT, [Working with Complementary Stylesheet Languages: CSS and XSLT](#)  
XML with an XSLT stylesheet, [Working with Complementary Stylesheet Languages: CSS and XSLT](#)  
CSS and XSLT\rCSS\_XSLT, [Working with Complementary Stylesheet Languages: CSS and XSLT](#)  
CSS basics, HTML and XHTML  
adding multiple classes to elements, [Working with Classes](#)  
aggregating stylesheet properties on single line, [Element Styles and the Style Attribute](#)  
centralizing with <style> element, [Centralizing with the <style> Element](#)  
class attribute, [Working with Classes](#)  
creating multi-level cascades, [Centralizing with the <style> Element](#)  
CSS-selector syntax, [Centralizing with the <style> Element](#)  
element styles, [Element Styles and the Style Attribute](#)  
font-weight and color attributes, [Element Styles and the Style Attribute](#)  
setting stylesheet property on XHTML element, [Element Styles and the Style Attribute](#)  
style attribute, [Element Styles and the Style Attribute](#)  
CSS basics, HTML and XHTML\rCSS\_HTMLetc, [The Basics of HTML/XHTML CSS](#)  
CSS deploying with XML  
creating tables, [Creating Tables in CSS](#)  
display, [Creating Tables in CSS](#)  
hiding content, [Hiding Content](#)  
text/css stylesheet type, [Deploying CSS with XML](#)  
type attribute, [Deploying CSS with XML](#)  
CSS deploying with XML\rCSS\_XML, [Deploying CSS with XML](#)  
CSS formatted for HTML  
HTML elements have intrinsically defined flow models, [Deploying CSS with XML](#)  
CSS Level 1  
compliance, [Enter Cascading Style Sheets](#)  
CSS Level 2  
compliance, [Enter Cascading Style Sheets](#)  
CSS Level 2 specification  
four primary containers, [Deploying CSS with XML](#)  
CSS properties  
default browser settings, [The Basics of HTML/XHTML CSS](#)  
display property, [Integrating Position](#)  
position property, [Integrating Position](#)  
visibility property, [Hiding Content](#)  
CSS stylesheet for XML  
differences to stylesheet for XHTML, [Deploying CSS with XML](#)  
CSS-selector syntax  
defining multiple rules for same element, [Centralizing with the <style> Element](#)  
defining rules, [Centralizing with the <style> Element](#), [Centralizing with the <style> Element](#)  
current() function, XSLT, [Functions](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## D

data

binary, [Binary Files](#)

describing, [How Else Would We Describe Our Data?](#)

retrieving from multiple tables, [Retrieving Data from Multiple Tables](#)

structured, [What's a Document Type?](#)

Data Objects, [Using the Database](#)

Data Objects layer, [n-Tier Architecture](#)

Data service layer, [n-Tier Architecture](#)

data types

DOMString, [DOMString](#)

data-type attribute

xsl, [Sorting the Result Tree](#)

Database Management Systems, see [dbms](#)

databases

creating a simpler data service, [Using XML in an n-Tier Application](#)

data retrieval from multiple tables, [Retrieving Data from Multiple Tables](#)

integrating XML into \r c12xdata, [Integrating XML into the Database Itself](#)

integrity, [SQL](#)

joins, [Joins](#)

mapping XML to a relational structure, [Mapping XML to a Relational Structure](#)

n-tier architecture \r c12n, [n-Tier Architecture](#)

normalization, [Normalization](#)

primary keys, [SQL](#)

requirements for examples, [What You Need](#)

returning XML from a data service object, [Using XML in an n-Tier Application](#)

situations when not needed \r c12when, [Who Needs a Database Anyway?](#)

SQL \r c12sql, [SQL](#)

staging, [Using the Database](#)

uniqueness, [SQL](#)

vendors and XML \r c12vend, [Database Vendors and XML](#)

XML, storing, [Storing XML in a Database](#)

DBMS, [Databases, Yesterday and Today](#)

different types, [Databases, Yesterday and Today](#)

dbXML, [XML Databases](#)

DCOM, [DCOM, Using XML in an n-Tier Application](#)

DCOM, RPC protocol, [DCOM](#)

extension of COM, [DCOM](#)

MS specific, [DCOM](#)

declaration elements

must begin with exclamation mark, [How It Works](#)

declarative programming, [Declarative Programming](#)

vs. imperative, [Imperative Versus Declarative Programming](#)

default and fixed values, attribute declarations, [Default and Fixed Values](#)

default namespaces

canceling, [Canceling Default Namespaces](#)

child elements, [Declaring Namespaces on Descendants](#)

on specific elements, [Declaring Namespaces on Descendants](#)  
using, [Default Namespaces](#)  
default namespaces \r c7def, [Default Namespaces](#)  
default templates, XSLT, [Default Templates](#)  
for PIs (Processing Instructions) and comments, [Default Templates](#)  
for text and attribute nodes, [Default Templates](#)  
using, [Default Templates](#)  
default values  
attribute value declarations, [Default Values](#)  
default XLink attributes, [Defaulting XLink Attributes](#)  
DefaultHandler class, SAX, [How to Receive SAX Events](#)  
deleteData method, CharacterData interface, [Modifying Strings](#)  
Demilitarized Zone, [Firewall-Ready](#)  
denormalization, [Normalization](#)  
descendant axis, XPath, [descendant](#)  
descendant-or-self axis, XPath, [descendant-or-self](#)  
descendants, [Hierarchies in XML](#)  
descriptor, CSS rules, [Enter Cascading Style Sheets](#)  
details display, address book stylesheet design, [User Interface Components](#)  
DHTML, [Hierarchies in HTML](#), [DOM Implementations](#)  
DHTML (Dynamic HTML), [DOM Implementations](#)  
disable-output-escaping attribute  
xsl, [<xsl:text>](#)  
disable-output-escaping attribute  
xsl, [Getting Information from the Source Tree with <xsl:value-of>](#)  
display property, CSS, [Integrating Position](#)  
display\br/>table properties, CSS, [Creating Tables in CSS](#)  
displaying XML, [Chapter 11: Displaying XML](#)  
CSS, [Chapter 11: Displaying XML](#)  
need for stylesheets, [The Need for Style\(sheets\)](#)  
distributed objects, [Using XML in an n-Tier Application](#)  
Distributed COM, see [dcom](#)  
div tags  
HTML tags, [The Need for Style\(sheets\)](#)  
DOCTYPE declaration, [How It Works](#)  
always located within the XML document, [How It Works](#)  
simple example, [How It Works](#)  
Document interface, DOM, [Creating XML Documents Through the DOM](#), [Document](#)  
createAttribute method, [Creating XML Documents Through the DOM](#)  
createElement method, [Creating XML Documents Through the DOM](#)  
createNodeType method, [Creating XML Documents Through the DOM](#)  
factory methods, [Creating XML Documents Through the DOM](#)  
document object, [Hierarchies in HTML](#)  
Document Object Model, see [dom](#)  
Document Object Model \t See DOM, [Hierarchies in HTML](#)  
document root in XPath, [The Document Root](#)  
matching template against, [Modes](#)  
document styles, cascading, [The Basics of HTML/XHTML/CSS](#)  
Document Type Declaration, [The Document Type Declaration](#)  
public identifiers, [Public Identifiers](#)  
system identifier, [System Identifiers](#)  
document type definitions, see [dtlds](#)  
document types  
namespaces \r c7need, [Why Do We Need Namespaces?](#)

document types \r c1doc, [What's a Document Type?](#)

document() function, XPath, [XML Debugging Template](#)

document() function, XSLT, [document\(\) function](#), [Functions](#)

using, [Try It Out?document\(\) Function](#)

DocumentFragment interface, DOM, [DocumentFragment](#)

DocumentHandler interface

deprecated in SAX 2.0, [Appendix F: SAX 2.0: The Simple API for XML](#)

DocumentHandler interface, SAXreceives notification of general document events, [Interface org.xml.sax.DocumentHandler?Deprecated](#)

documenting XML Schemas, [Documenting XML Schemas](#)

annotations, [Annotations](#)

attributes from other namespaces, [Attributes from Other Namespaces](#)

comments, [Comments](#)

documents

elements, retrieving from, [Full Syntax](#)

linking to specific parts of \r c11xpoint, [XPointer: Pointing to Document Fragments](#)

namespaces, adding \r c7doc, [How XML Namespaces Work](#)

nodes, adding, [Creating XML Documents Through the DOM](#)

nodes, removing, [Removing Nodes from a Document](#)

transforming to web page, [Modes](#)

DocumentType interface, DOM, [DocumentType](#)

DOM, [Hierarchies in HTML](#), [What are the Pieces that Make Up XML?](#), [Chapter 8: The Document Object Model \(DOM\)](#)

full list of interfaces, their properties and methods, [Appendix A: The XML Document Object Model](#)

Oracle XML parsers' extensions to, [XML Parsers](#)

DOM (Document Object Model), [Chapter 8: The Document Object Model \(DOM\)](#), [DOM Interfaces](#)

CharacterData interface, [Working With Text](#)

Document interface, [Creating XML Documents Through the DOM](#)

DOMException interface, [DOMException](#)

error handling, [Exceptions](#)

extensions, [Extending Interfaces](#), [DOM Extensions](#)

implementations, [DOM Implementations](#)

Node interface, [DOM Interfaces](#)

owner documents, [The ownerDocument Property](#)

Text interface, [Working With Text](#)

trees, [Retrieving Information from the DOM](#)

XML, [The XML DOM](#)

XML documents, creating, [Creating XML Documents Through the DOM](#)

XML parsers, [The XML DOM](#)

DOM Core, [DOM Implementations](#), [The DOM Core](#)

Extended interfaces, [The DOM Core](#)

Fundamental interfaces, [The DOM Core](#)

DOM CSS, [DOM Implementations](#)

DOM Extended Interfaces, [Extended Interfaces](#)

DOM HTML, [DOM Implementations](#)

DOMDocument object, MSXML Parser object

load method, [Listening for Requests](#)

transformNode method, [Listening for Requests](#)

DOMException interface, [DOMException](#)

DOMImplementation interface, [DOMImplementation](#)

DOMString data type, [DOMString](#)

double quotation marks, attributes, [Attributes](#)

DTD anatomy

document type declaration, [The Document Type Declaration](#)

DTD limitations, [DTD Limitations](#)

data typing, [Data Typing](#)

limited content model descriptions, [Limited Content Model Descriptions](#)

syntax, [DTD Syntax](#)

XML namespaces, [XML Namespaces](#)

DTDHandler interface, SAX

receives notification of basic DTD-related events, [Receive notification of basic DTD-related events](#)

DTDs, [What are the Pieces that Make Up XML?](#), [Chapter 5: Document Type Definitions](#)

attribute declarations, [Attribute Declarations](#)

cardinality indicators, [Cardinality](#)

creating, [Try It Out?"I Want a New DTD"](#)

default XLink attributes, [Defaulting XLink Attributes](#)

developing, [Developing DTDs](#)

element declarations, [Element Declarations](#), [Mixed Content](#)

embedding a DTD in XML document, [Try It Out?What's in a Name?](#), [How It Works](#)

ENTITY functionality, [Do We Still Need DTDs?](#)

need for, [Do We Still Need DTDs?](#)

sharing vocabularies, [Sharing Vocabularies](#)

using external DTDs, [Try It Out?The External DTD](#)

validating documents, [Chapter 5: Document Type Definitions](#)

duplicate attribute error, [Attributes](#)

Dynamic HTML, see [dhtml](#)

dynamically creating elements in XSLT, [<xsl:element>](#)

---

[!\[\]\(5add88f38d25136a40ffcfd684d67791\_img.jpg\) PREVIOUS](#)

[!\[\]\(7206bbebc4c4f09551baa7bd85ca7009\_img.jpg\)< Free Open Study >](#)

[!\[\]\(f3546bcd5c89ed4dd86968a32a3180b1\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## E

e-commerce

XML, using in, [e-Commerce](#)

XSLT, importance of, [Why is XSLT So Important for e-Commerce?](#)

EBCDIC, [Unicode](#)

eight-bit ASCII encoding, [Encoding](#)

element class, CSS-selector syntax, [Centralizing with the <style> Element](#)

element content, [Hierarchies in XML](#)

element content, element declarations, [Element Content](#)

choices, [Choices](#)

combining sequences and choices, [Combining Sequences and Choices](#)

sequences, [Sequences](#)

element declarations

any content, [Any Content](#)

cardinality, [Cardinality](#)

complexType definition, [<complexType>](#)

content model, [Element Declarations](#)

default and fixed values, [Default and Fixed Values](#)

element content, [Element Content](#)

element wildcards, [Element Wildcards](#)

empty content model, [Empty Content](#)

mixed content model, [Mixed Content](#)

substitutionGroup attribute, [Element Substitution](#)

element element, [<element>](#)

form attribute, [Element Qualified Form](#)

Element interface, DOM, [Element](#)

ELEMENT keyword, [Element Declarations](#)

element rules \r c2ele, [Rules for Elements](#)

element styles, cascading, [The Basics of HTML/XHTML/CSS](#)

element substitution, [Element Substitution](#)

using, [When to Use Element Substitution](#)

element wildcards

any> declaration, [Element Wildcards](#)

element-available() function, XSLT, [Functions](#)

elementFormDefault attribute

schema element, [Declaration Defaults](#)

elements, [Tags and Text and Elements, Oh My!](#)

cardinality indicators, [Cardinality](#)

complexity of, [Why Use Attributes?](#)

content, [Hierarchies in XML, Tags and Text and Elements, Oh My!](#)

dynamically creating in XSLT, [<xsl:element>](#)

empty, [Empty Elements](#)

illegal PCDATA characters \r c2ill, [Illegal PCDATA Characters](#)

information, accessing with Node interface, [Try It Out-Accessing Element Information with Node](#)

intrusiveness, [Why Use Attributes?](#)

names, [Element Names](#)

namespaces, [Using Prefixes, How XML Namespaces Work, Default Namespaces, Declaring Namespaces on](#)

## Descendants

namespaces \r c7need, [Why Do We Need Namespaces?](#)  
parent and child relationships tabulated, [Elements](#)  
retrieving from documents, [Full Syntax](#)  
root, [An XML Document Can Have Only One Root Element](#)  
start and end tags, [Every Start-tag Must Have an End-tag](#)  
vs. attributes, [Namespaces and Attributes](#)  
vs. attributes \r c2attsel, [Why Use Attributes?](#)  
XSLT, attribute types tabulated, [Types](#)  
XSLT, containing as variables, [Variables, Constants, and Named Templates](#)  
XSLT, full list, [Elements](#)  
em tags, [Centralizing with the <style> Element](#)  
embedding DTDs, [Try It Out- What's in a Name?](#)  
empty content models  
complexType definitions, [<complexType>](#), [<complexType>](#)  
empty content, element declarations, [Empty Content](#)  
empty elements, [Empty Elements](#)  
EMPTY keyword, [Empty Content](#)  
enumeration facet  
attributes, [Constraining Facets](#)  
encoding  
Unicode, [Unicode](#)  
encoding \r c2enc, [Encoding](#)  
encoding rules, SOAP, [How SOAP Works](#), [Encoding Rules](#)  
encodingStyle attribute, SOAP, [Encoding Rules](#)  
end-tags, [Tags and Text and Elements, Oh My!](#), [Every Start-tag Must Have an End-tag](#)  
endDocument method, ContentHandler interface, [How to Receive SAX Events](#)  
endDocument method, IVBSAXContentHandler interface, [The Microsoft Way with SAX](#)  
endElement method, ContentHandler interface, [How to Receive SAX Events](#), [Extracting Character Data](#)  
Entities, [Entities](#)  
built in entities, [Built-in Entities](#)  
character entities, [Character Entities](#)  
general entities, [General Entities](#)  
parameter entities, [Parameter Entities](#)  
ENTITIES attribute type, [ENTITY and ENTITIES](#)  
ENTITY attribute type, [ENTITY and ENTITIES](#)  
ENTITY functionality  
not provided with XML Schemas, [Do We Still Need DTDs?](#)  
Entity interface, DOM, [Entity](#)  
ENTITY keyword, [General Entities](#)  
entity references, [Escaping Characters](#)  
illegal references, [General Entities](#)  
may be used within DTDs, [General Entities](#)  
EntityReference interface, DOM, [EntityReference](#)  
EntityResolver interface, SAX  
basic interface for resolving entities, [Interface org.xml.sax.EntityResolver](#)  
enumerated attribute types, [Enumerated Attribute Types](#)  
NMTOKEN values, [Enumerated Attribute Types](#)  
NOTATION values, [Enumerated Attribute Types](#)  
specify list of allowable values, [Enumerated Attribute Types](#)  
envelope, SOAP, [How SOAP Works](#), [The Envelope](#)  
structure of document, [The Envelope](#)  
using for addNumbers, [Try It Out-Using a Proper SOAP Envelope for addNumbers](#)  
Envelope> element, SOAP, [<Envelope>](#)  
Err object, Visual Basic, [Exceptions](#)

error handling, DOM  
exceptions, [Exceptions](#)

See also exceptions \t, [Exceptions](#)

error method, ErrorHandler interface, [Even More About Errors-Catching Parsing Errors](#)

error method, IVBSAXErrorHandler interface, [The Microsoft Way with SAX](#)

ErrorHandler interface, SAX

basic interface for SAX error handlers, [Basic Interface for SAX Error Handlers](#)

ErrorHandler interface, SAX

error method, [Even More About Errors-Catching Parsing Errors](#)

fatal method, [Even More About Errors-Catching Parsing Errors](#)

warning method, [Even More About Errors-Catching Parsing Errors](#)

errors, [Errors in XML](#), [Even More About Errors-Catching Parsing Errors](#), see also [exceptions](#)

parser errors, [Even More About Errors-Catching Parsing Errors](#)

errors, SAX

fatal errors, [Even More About Errors-Catching Parsing Errors](#)

Locator class, [More About Errors-Using the Locator Object](#)

non-recoverable errors, [Even More About Errors-Catching Parsing Errors](#)

recoverable errors, [Even More About Errors-Catching Parsing Errors](#)

escaping characters, [Escaping Characters](#)

Exampotron, [Exampotron](#)

exceptions, [Even More About Errors-Catching Parsing Errors](#)

exceptions, DOM, [Exceptions](#)

DOMException interface, [DOMException](#)

See also error handling \t, [Exceptions](#)

exceptions, Java, [How to Receive SAX Events](#)

exceptions, SAX

SAXException, [Error Handling](#)

Execute method, Connection object, ADO, [Using XML in an n-Tier Application](#)

Expat, [Parsing XML](#)

expressions, XPath, [XPath](#)

Extended Interfaces, DOM

full list of interfaces, properties and methods, [Extended Interfaces](#)

Extended interfaces, DOM Core, [The DOM Core](#)

extended links, [Link Types](#)

arc-type elements, [Arcs](#)

creating, [Try It Out-Creating an Extended Link](#)

remote resources, adding, [Try It Out-Adding Remote Resources to the Extended Link](#)

resource-type elements, [Resource-type Elements](#)

title-type elements, [Title-type Elements](#)

XLink, [XLink](#)

extended links \r c11extend, [Extended Links](#)

extended-type elements, XLink, [The type Attribute](#), [Extended Links](#)

extensibility

of XML, [Why "Extensible"?](#)

Extensible Information Server, [XML Databases](#)

Extensible Markup Language, see [xml](#)

Extensible Stylesheet Language, see [xsl](#)

Extensible Stylesheet Language \t See XSL, [What are the Pieces that Make Up XML?](#)

Extensible Stylesheet Language for Transformations \t See XSLT, [Chapter 4: XSLT](#)

extension declarations, [Extending <simpleType> Definitions](#)

external DTDs

benefits, [How It Works](#)

using, [Try It Out-The External DTD](#)

external entity declaration, [General Entities](#)

external stylesheets

adding to HTML or XHTML documents, [Linking Stylesheets in HTML](#)  
adding to XML document, [Linking Stylesheets in HTML](#)  
external stylesheets, cascading, [The Basics of HTML/XHTML CSS](#)  
external subset, Document Type Declaration, [The Document Type Declaration](#)  
extraneous white space, [Whitespace in Markup](#)

---

[!\[\]\(db338933d5c646f1ed2a0efa9c10cb79\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(e9bcaaa9c1242cad288636b8de1d63e1\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## F

facets

table of constraining facets, [<restriction>](#)

factory methods

Document interface, [Creating XML Documents Through the DOM](#)

false() function, XPath, [<xsl:copy-of>](#), [Functions](#)

fatal errors, [Errors in XML](#), [Even More About Errors?Catching Parsing Errors](#)

fatal method, ErrorHandler interface, [Even More About Errors?Catching Parsing Errors](#)

fatalError method, IVBSAXErrorHandler interface, [The Microsoft Way with SAX](#)

Fault> element, SOAP, [<Fault>](#)

faultcode> element, SOAP

unique identifiers, [<Fault>](#)

fields, [Databases, Yesterday and Today](#)

files

binary, [Binary Files](#)

text, [Text Files](#)

filters, using in XPath location paths, [Filtering XPath Expressions and Patterns](#)

fixed values

attribute value declarations, [Fixed Values](#)

floor() function, XPath, [Functions](#)

following axis, XPath, [following](#)

following-sibling axis, XPath, [following-sibling](#)

font tags, [The Need for Style\(sheets\)](#)

font-weight attribute, CSS, [Element Styles and the Style Attribute](#)

FOR XML EXPLICIT clause, [SQL Server](#)

FOR XML RAW clause, [SQL Server](#)

form attribute

element element, [Element Qualified Form](#)

form attribute, attribute declarations

overriding default attribute qualification, [Attribute Qualified Form](#)

form tags, [User Entry Form Using HTML <form> Element](#)

Formal Public Identifiers, see [fpi's](#)

format-number() function, XSLT, [Functions](#)

formatting XSLT output, [<xsl:output>](#)

FPIs, [Public Identifiers](#)

notation declarations, [Notations](#)

syntax, [Public Identifiers](#)

fractionDigits facet, [Constraining Facets](#)

attributes, [Constraining Facets](#)

from attribute, XLink, [XLink Attributes, The from and to Attributes](#)

arc-type elements, [Arcs](#)

fully qualified names, namespaces, [How XML Namespaces Work](#)

function-available() function, XSLT, [Functions](#)

functions

Xpath, usable within XSLT stylesheets, [Inherited XPath Functions](#)

XSLT special, full list (Appendix), [Functions](#)

functions in XPath, [XPath Functions](#)

recursion, [`<xsl:copy>`](#)

Fundamental Interfaces, DOM

full list of interfaces, properties and methods, [Appendix A: The XML Document Object Model](#)

Fundamental interfaces, DOM Core, [The DOM Core](#)

Node interface, [Extending Interfaces in the DOM](#)

---

[!\[\]\(2f117456a43f4be0cb4c5f1b6e2cdf3d\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(9f545d4f6d8b725e981277523c4230a8\_img.jpg\) NEXT >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## G

general entities

create reusable sections of replacement text, [General Entities](#)

references to, [References to General Entities](#)

generate-id() function, XSLT, [Functions](#)

GET method, HTTP, [HTTP](#)

getColumnNumber method, Locator class, [More About Errors-Using the Locator Object](#)

getElementsByTagName method, MSXML, [Try It Out-Accessing Items in a NodeList](#)

getIndex() methods

Attributes interface, SAX2, [org.xml.sax.Attributes \(SAX 2-Replaces AttributeList\)](#)

getLength method, Attributes class, [Extracting Attributes](#)

getLineNumber method, Locator class, [More About Errors-Using the Locator Object](#)

getNamedItem method, NamedNodeMap interface, [NodeList and NamedNodeMap](#)

getPublicId method, Locator class, [More About Errors-Using the Locator Object](#)

getSystemId method, Locator class, [More About Errors-Using the Locator Object](#)

getType method, Attributes class, [Extracting Attributes](#)

getValue method, Attributes class, [Extracting Attributes](#)

get0Name method, Attributes class, [Extracting Attributes](#)

global attributes, [Namespaces and Attributes](#)

XLink, [XLink](#)

global complex type definitions, [<complexType>](#)

global constants, [Variables, Constants, and Named Templates](#)

global type, attribute declarations, [Using a Global Type](#)

GoXML, [XML Databases](#)

group declaration, XML Schemas content model, [<group>](#)

group element, [<group>](#)

using a global group, [<group>](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## H

HandlerBase interface, SAX  
default base class for handlers, [Default Base Class for Handlers](#)  
deprecated, [Default Base Class for Handlers](#)  
hasChildNodes method, Node interface, [The hasChildNodes\(\) Method](#)  
head elements, [Element Substitution](#)  
Header> element, SOAP, [<Header>](#)  
mustUnderstand attribute, [The mustUnderstand Attribute](#)  
here() function, XPointer, [XPointer Function Extensions to XPath](#)  
hierarchies of information  
in HTML, [Hierarchies in HTML](#)  
in XML, [Hierarchies in XML](#)  
hierarchies of information \r c1hi, [Hierarchies of Information](#)  
href attribute, XLink, [XLink Attributes](#), [The href Attribute](#)  
HTML, [A Brief History of Markup](#)  
adding external stylesheets, [Linking Stylesheets in HTML](#)  
based on SGML, [A Brief History of Markup](#)  
case-insensitive, [Case-Sensitivity](#)  
DHTML, [DOM Implementations](#)  
DOM, [DOM Implementations](#)  
from XML, [Who Needs a Database Anyway?](#)  
hierarchies in, [Hierarchies in HTML](#)  
history, [The Need for Style\(sheets\)](#)  
limitations of, [So What is XML?](#)  
not extensible, [Why "Extensible"?](#)  
overlapping tags, [Tags Cannot Overlap](#)  
white space, [Whitespace in PCDATA](#)  
XML, difference from, [HTML and XML: Apples and Red Delicious Apples](#)  
HTML and XHTML CSS\rCSS\_HTMLetc, [The Basics of HTML/XHTML/CSS](#)  
HTML linking  
anchors, [Appending XPointer Expressions to URIs](#)  
constraints, [HTML Linking](#)  
pointing to section of page, [HTML Linking](#)  
HTML linking \r c11html, [HTML Linking](#)  
HTML tags  
a tags, [XML Debugging Template](#)  
describing in XML document, [CDATA Sections](#)  
div tags, [The Need for Style\(sheets\)](#)  
em tags, [Centralizing with the <style> Element](#)  
font tags, [The Need for Style\(sheets\)](#)  
form tags, [User Entry Form Using HTML <form> Element](#)  
li tags, [Centralizing with the <style> Element](#)  
p tags, [The Basics of HTML/XHTML/CSS](#)  
style attribute, [Element Styles and the Style Attribute](#)  
style tags, [Centralizing with the <style> Element](#)  
table tags, [Suppress List and Empty Results Message](#)  
td tags, [Suppress List and Empty Results Message](#), [Selected Item Highlighting](#)

tr tags, [Suppress List and Empty Results Message](#)

xmp tags, [XML Debugging Template](#)

HTTP, [HTTP](#)

GET method, [HTTP](#)

headers and body, [HTTP](#)

POST method, [HTTP](#), [Using HTTP for SOAP](#)

reasons for use with SOAP, [Why HTTP for SOAP?](#), [Widely Implemented](#), [Request/Response](#), [Firewall-Ready](#)

security, [Security](#)

request/response protocol, [HTTP](#)

SOAPAction header, [Using HTTP for SOAP](#)

SQL queries through, [SQL Server](#)

use by SOAP, [How SOAP Works](#)

using for SOAP, [Using HTTP for SOAP](#)

HTTP requests

online address book case study, [Servlet Overview](#)

hyperlinks, [A Brief History of Markup](#)

hyperlinks \r c11html, [HTML Linking](#)

Hypertext Markup Language, see [html](#)

Hypertext Transfer Protocol, see [http](#)

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## I

ID attribute type, [ID, IDREF and IDREFS](#)  
rules, [ID, IDREF and IDREFS](#)  
id hashes, CSS-selector syntax, [Centralizing with the <style> Element](#)  
id() function, XPath, [Functions](#)  
IDL, [WSDL](#)  
IDREF attribute type, [ID, IDREF and IDREFS](#)  
rules, [ID, IDREF and IDREFS](#)  
IDREFS attribute type, [ID, IDREF and IDREFS](#)  
ignorableWarning method, IVBSAXErrorHandler interface, [The Microsoft Way with SAX](#)  
ignorableWhitespace method, ContentHandler interface, [ignorableWhitespace](#)  
IIOP, [IIOP](#)  
IIOP, RPC protocol, [IIOP](#)  
IIS, [Running the Examples](#)  
full Windows web server, [Running the Examples](#)  
illegal PCDATA characters  
CDATA sections \r c2cdata, [CDATA Sections](#)  
escaping characters, [Escaping Characters](#)  
illegal PCDATA characters \r c2ill, [Illegal PCDATA Characters](#)  
imperative programming, [Imperative Programming](#)  
vs. declarative, [Imperative Versus Declarative Programming](#)  
implementation details  
XPath axes, [Appendix B: XPath Reference](#)  
XPath functions, [Functions](#)  
XPath node-tests, [Node tests](#)  
XSLT elements, [Elements](#)  
XSLT special functions, [Functions](#)  
implementations, inheriting in SAX, [How to Receive SAX Events](#)  
implied attributes, [Implied Values](#)  
implied values  
attribute value declarations, [Implied Values](#)  
import declaration, [<import>, <import>](#)  
targetNamespace attribute, [<import>](#)  
IMXReaderControl interface, SAX  
abort method, [The Microsoft Way with SAX](#)  
resume method, [The Microsoft Way with SAX](#)  
suspend method, [The Microsoft Way with SAX](#)  
include declaration, [<include>](#)  
targetNamespace attribute, [<import>, <include>](#)  
indent attribute  
xsl, [<xsl:output>](#)  
indexes, XPointer, [Points](#)  
infinite loops, [Variables, Constants, and Named Templates](#)  
information hierarchies \r c1hi, [Hierarchies of Information](#)  
inheritance  
extension, [Inheritance, Extending <complexType> Definitions, Extending <simpleType> Definitions](#)  
restriction, [Inheritance, Restriction, Restricting <complexType> Definitions, Restricting <simpleType> Definitions](#)

inline element, CSS primary containers, [Deploying CSS with XML](#)

inline links \r c11simp, [Simple Links](#)

InputSource interface, SAX

single source for an XML entity, [Class org.xml.sax.InputSource](#)

INSERT statement

moving data to real database, [Using the Database](#)

insertBefore method, Node interface, [The insertBefore\(\) Method](#)

insertData method, CharacterData interface, [Modifying Strings](#)

integrity in databases, [SQL](#)

inter-enterprise communications with XML, [Using XML in an n-Tier Application](#)

inter-object communications with XML, [Using XML in an n-Tier Application](#)

interactive web applications, [Case Study 1: Using XSLT to Build Interactive Web Applications](#)

advantages of using XSL, [Advantages of Using XSL to Build Interactive Web Applications](#)

Interface Definition Language, see [idl](#)

interfaces

compared to objects, [What Are Interfaces?](#)

deprecated and added, for SAX 2.0, [Appendix F: SAX 2.0: The Simple API for XML](#)

DOM, [DOM Interfaces](#)

DOM Core, [The DOM Core](#)

DOM, full list, [Appendix A: The XML Document Object Model](#)

extending, [Extending Interfaces](#)

implementing, [Implementing Interfaces](#)

multiple, [Implementing Multiple Interfaces](#)

SAX, [How to Receive SAX Events](#)

SAX 2.0, full list, [Appendix F: SAX 2.0: The Simple API for XML](#)

internal subset, Document Type Declaration, [The Document Type Declaration](#)

internationalization

Unicode, [Unicode](#)

Internet Explorer, [Parsing XML](#)

DOM, [The XML DOM](#)

MSXML, [The XML DOM](#)

XML files, opening in, [So What is XML?](#)

Internet Information Services, see [iis](#)

Internet Inter-ORB Protocol, see [iop](#)

ISAPI filters, [SQL Server](#)

ISO-8859-1, [Unicode](#)

item method, NamedNodeMap interface, [NodeList and NamedNodeMap](#)

item method, NodeList interface, [NodeList and NamedNodeMap](#)

itemType attribute

list declarations, XML Schemas, [<list>](#)

IVBSAXContentHandler interface, SAX, [The Microsoft Way with SAX](#)

documentLocator method, [The Microsoft Way with SAX](#)

endDocument method, [The Microsoft Way with SAX](#)

startDocument method, [The Microsoft Way with SAX](#)

startElement method, [The Microsoft Way with SAX](#)

IVBSAXErrorHandler interface, SAX

error method, [The Microsoft Way with SAX](#)

fatalError method, [The Microsoft Way with SAX](#)

ignorableWarning method, [The Microsoft Way with SAX](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## J

Java

compared to Visual Basic, [The Microsoft Way with SAX](#)

exception handling, [Exceptions](#)

exceptions, [How to Receive SAX Events](#)

interfaces, implementing, [Implementing Multiple Interfaces](#)

namespaces, [Using Prefixes](#)

Java DataBase Connectivity, see [jdbc](#)

Java Development Kit, see [jdk](#)

Java Platform 2 Standard Edition

using, [Install JRE](#)

Java RMI, [Java RMI](#)

Java RMI, RPC protocol, [Java RMI](#)

Java specific, [Java RMI](#)

Java Runtime Environment, see [jre](#)

Java servlet, functionality provided by

online address book case study, [Servlet Overview](#)

JavaScript

imperative programming, [Imperative Programming](#)

try and catch exception handling supported, [Exceptions](#)

XML parsers, preventing from being parsed by, [CDATA Sections](#)

JDBC, [Using XML in an n-Tier Application](#)

JDK, [Install JRE](#), [Where to Get SAX](#)

installing, [Install JRE](#)

joins, [Joins](#)

JRE, [Install JRE](#)

installing, [Install JRE](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## K-L

key() function, XSLT, [Functions](#)  
labels for resources, [Semantic Attributes \(role and title\)](#)  
lang() function, XPath, [Functions](#)  
last() function, XPath, [Functions](#)  
leaves, [Hierarchies in XML](#)  
length facet  
attributes, [Constraining Facets](#)  
li tags, [Centralizing with the <style> Element](#)  
line feed character, [End-of-Line Whitespace](#)  
linking, [Chapter 13: Linking and Querying XML](#)  
arcs, [Arcs](#)  
defining functionality of, [Behavior Attributes \(actuate and show\)](#)  
directionality, [The from and to Attributes](#)  
extended links, [Link Types](#), [Try It Out?Creating an Extended Link](#)  
extended links \r c11extend, [Extended Links](#)  
HTML \r c11html, [HTML Linking](#)  
inline links \r c11simp, [Simple Links](#)  
labeling resources, [Semantic Attributes \(role and title\)](#)  
out of line links \r c11extend, [Extended Links](#)  
simple links, [Link Types](#)  
simple links \r c11simp, [Simple Links](#)  
to specific parts of documents \r c11xpoint, [XPointer: Pointing to Document Fragments](#)  
XLink, [XML Linking](#), [XLink Attributes](#)  
XLink \r c11xlink, [XLink](#)  
XML \r c11xml, [XML Linking](#)  
XPointer, [XML Linking](#), [Appending XPointer Expressions to URIs](#), [XPointer Schemes](#), [XPointer Shorthand Syntaxes](#), [Locations, Points and Ranges](#), [Points](#), [Ranges](#), [XPointer Function Extensions to XPath](#)  
XPointer \r c11xpoint, [XPointer: Pointing to Document Fragments](#)  
linking stylesheets  
XHTML, [Linking Stylesheets in HTML](#)  
list declarations, XML Schemas, [<list>](#)  
itemType attribute, [<list>](#)  
literal name() node test, XPath, [literal name](#)  
load method, DOMDocument object, [Listening for Requests](#)  
load method, MSXML, [Vendor-Specific Extensions](#), [DOM Extensions](#), [The New RPC Protocol: SOAP](#)  
loadXML method, MSXML, [Vendor-Specific Extensions](#), [DOM Extensions](#)  
local complex type definitions, [<complexType>](#)  
local constants, [Variables, Constants, and Named Templates](#)  
local resources  
extended links, adding to, [Resource-type Elements](#)  
local type, attribute declarations, [Creating a Local Type](#)  
local-name() function, XPath, [<xsl:element>](#), [Functions](#)  
location paths, XPath, [XPath](#), [The Document Root](#)  
location steps and their components, [Appendix B: XPath Reference](#)  
specific, [Filtering XPath Expressions and Patterns](#)  
location sets, XPointer, [XPointer: Pointing to Document Fragments](#), [Locations, Points and Ranges](#)

locations, XPointer, [XPointer: Pointing to Document Fragments](#), [Locations, Points and Ranges](#)  
locator attribute, XLink, [The href Attribute](#)  
Locator class, [More About Errors?Using the Locator Object](#), [Try It Out?Using the Locator](#)  
getColumnNumber method, [More About Errors?Using the Locator Object](#)  
getLineNumber method, [More About Errors?Using the Locator Object](#)  
getPublicId method, [More About Errors?Using the Locator Object](#)  
getSystemId method, [More About Errors?Using the Locator Object](#)  
Locator interface, SAX  
associating a SAX event with a document location, [Interface org.xml.sax.Locator](#)  
locator-type elements, XLink, [The type Attribute](#), [Locator-type Elements](#)  
logical layers  
n-tier architecture, [n-Tier Architecture](#)

[!\[\]\(a5d7fd5fb5ce3676a2a1638c1950c46d\_img.jpg\) PREVIOUS](#)

[!\[\]\(df2c48210484d942935f17eb11d1e2ba\_img.jpg\)< Free Open Study >](#)

[!\[\]\(d570f88eab94bf84b6a0292d4bd10ccf\_img.jpg\) NEXT >](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## M

Macintosh computers  
newline characters, [End-of-Line Whitespace](#)  
main method, Java, [How to Receive SAX Events](#)  
markup  
history \r c1hist, [A Brief History of Markup](#)  
white space, [Whitespace in Markup](#)  
XML, containing as a variable, [Variables, Constants, and Named Templates](#)  
match attribute  
xsl, [How XSLT Stylesheets Work?Templates](#), [`<xsltemplate>`](#), [`<xsl:copy>`](#)  
MathML, [Why "Extensible"?](#)  
maxExclusive facet  
attributes, [Constraining Facets](#)  
maxInclusive facet  
attributes, [Constraining Facets](#)  
modifying, [How It Works](#)  
maxlength facet  
attributes, [Constraining Facets](#)  
maxOccurs and minOccurs attributes  
cardinality in element declarations, [Cardinality](#)  
metadata  
attributes, [Why Use Attributes?](#)  
binary files, [Binary Files](#)  
in text files, [A Brief History of Markup](#)  
method attribute  
xsl  
output> element, [`<xsl:output>`](#)  
methods, [Hierarchies in HTML](#)  
methods, overriding, [How to Receive SAX Events](#)  
methods, SAX 2.0  
interfaces and classes, [Appendix F: SAX 2.0: The Simple API for XML](#)  
Microsoft  
XML technologies \r c12msoft, [Microsoft's XML Technologies](#)  
Microsoft SAX application, [The Microsoft Way with SAX](#)  
IVBSAXContentHandler interface, [The Microsoft Way with SAX](#)  
XML parser, [The Microsoft Way with SAX](#)  
Microsoft SOAP Toolkit 2.0, [Building the Client](#)  
minExclusive facet  
attributes, [Constraining Facets](#)  
minInclusive facet, [Constraining Facets](#)  
attributes, [Constraining Facets](#)  
minlength facet, [Constraining Facets](#)  
attributes, [Constraining Facets](#)  
mixed content, [Hierarchies in XML](#)  
mixed content model  
element declarations, [Mixed Content](#)  
mixed content model, element declarations, [Mixed Content](#)

\* cardinality indicator, [Mixed Content](#)  
rules, [Mixed Content](#)  
text-only elements, [Try It Out?"I Want a New DTD"](#)  
mixed content models  
complexType definitions, [`<complexType>`](#)  
mode attribute  
xsl, [`<xsltemplate>`](#), [`<xsl:apply-templates>`](#), [Modes](#)  
modes, [Modes](#)  
transforming XML documents to web pages, [Modes](#)  
using, [Modes](#)  
modifying system properties  
PATH and CLASSPATH, [Modify the Path and Class Path](#)  
Mortgage Industry Standards Maintenance Organization's (MISMO) DTD, [Sharing Vocabularies](#)  
Mosaic browser, [The Need for Style\(sheets\)](#)  
Mozilla, [Chapter 13: Linking and Querying XML](#)  
MSXML  
DOM implementation, [The XML DOM](#)  
load method, [Vendor-Specific Extensions](#), [DOM Extensions](#), [The New RPC Protocol: SOAP](#)  
loadXML method, [Vendor-Specific Extensions](#), [DOM Extensions](#)  
save method, [Vendor-Specific Extensions](#)  
XMLHTTP object, [How It Works](#)  
MSXML parser, [MSXML](#)  
Replace Mode, [MSXML](#), [Associating Stylesheets with XML Documents Using Processing Instructions](#)  
XSLT, [MSXML](#)  
MSXML Parser object  
DOMDocument object, [Listening for Requests](#)  
MSXML SDK  
SAX implementation, [The Microsoft Way with SAX](#)  
MSXML 3 SP1, [MSXML](#)  
MSXML3 parser  
and other versions, implementation of XPath axes, [Appendix B: XPath Reference](#)  
and other versions, implementation of XPath functions, [Functions](#)  
and other versions, implementation of XPath node-tests, [Node tests](#)  
and other versions, implementation of XSLT elements, [Elements](#)  
and other versions, implementation of XSLT special functions, [Functions](#)  
MSXSL, [MSXML](#)  
multi-tier architecture, see [n-tier architecture](#)  
multiple interfaces, implementing, [Implementing Multiple Interfaces](#)  
multiple tables  
data retrieval, [Retrieving Data from Multiple Tables](#)  
multiple XPointer expressions  
using, [Using Multiple XPointer Expressions](#)  
mustUnderstand attribute  
<Header> element, SOAP, [The mustUnderstand Attribute](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## N

n-tier architecture, [n-Tier Architecture](#)  
advantages of, [n-Tier Architecture](#)  
layers, [n-Tier Architecture](#)  
XML, using in \r c12nxml, [Using XML in an n-Tier Application](#)  
n-tier architecture \r c12n, [n-Tier Architecture](#)  
name attribute  
xsl, [`<xsl:template>, <xsl:element>, <xsl:attribute> and <xsl:attribute-set>`](#)  
name() function, XPath, [`<xsl:element>, Functions`](#)  
named templates, [`<xsl:template>, Variables, Constants, and Named Templates`](#)  
NamedNodeMap interface, DOM, [NamedNodeMap](#)  
getNamedItem method,  [NodeList and NamedNodeMap](#)  
item method,  [NodeList and NamedNodeMap](#)  
length property,  [NodeList and NamedNodeMap](#)  
setNamedItem method, [Try It Out?Creating an XML Document from Scratch](#)  
namespace attribute  
values allowed in <any> declaration, [Element Wildcards](#)  
xsl, [`<xsl:element>, <xsl:attribute> and <xsl:attribute-set>`](#)  
namespace attribute, attribute declarations  
allowable values, [Attribute Wildcards](#)  
namespace axis, XPath, [namespace](#)  
Namespace Identifier, URNs, [URNs](#)  
Namespace Specific String, URNs, [URNs](#)  
namespace support  
XML Schemas and DTDs, [XML Schema Namespace Support](#)  
namespace-uri() function, XPath, [Functions](#)  
namespaces, [What are the Pieces that Make Up XML?](#)  
attributes \r c7att, [Namespaces and Attributes](#)  
case-sensitivity, [The XML Schema Namespace](#)  
concerned with vocabulary, not document type, [Using Prefixes](#)  
declaring on descendants, [Declaring Namespaces on Descendants](#)  
default, [Canceling Default Namespaces](#)  
default \r c7def, [Default Namespaces](#)  
fully qualified names, [How XML Namespaces Work](#)  
in XML \r c7name, [How XML Namespaces Work](#)  
Java, [Using Prefixes](#)  
namespace URIs, meaning of \r c7mean, [What Do Namespace URIs Really Mean?](#)  
notations \r c7not, [Do Different Notations Make Any Difference?](#)  
on specific elements, [Declaring Namespaces on Descendants](#)  
prefix approach \r c7prefix, [Using Prefixes](#)  
prefixes, [So Why Doesn't XML Just Use These Prefixes?](#)  
qualified names, [How XML Namespaces Work](#)  
reasons for using \r c7need, [Why Do We Need Namespaces?](#)  
reasons for using \r c7reas, [When Should I Use Namespaces?](#)  
support for in SAX 2.0, [Appendix F: SAX 2.0: The Simple API for XML](#)  
UName, [Default Namespaces](#)  
URLs \r c7url, [Why Use URLs for Namespaces, Not URNs?](#)

XHTML, [When Should I Use Namespaces?](#)  
xmlns attribute, [How XML Namespaces Work](#)  
XSLT, and xsl prefix, [Elements](#)  
naming attributes, [Attributes](#)  
naming attributes, attribute declarations, [Naming Attributes](#)  
naming elements, [Element Names](#)  
case-sensitivity, [Case-Sensitivity](#)  
NDATA keyword, [General Entities](#)  
nesting tags, [Tags Cannot Overlap](#)  
new line characters, [End-of-Line Whitespace](#)  
NIDs, URNs, [URNs](#)  
NMTOKEN attribute type, [NMTOKEN and NMTOKENS](#)  
NMTOKEN values  
may begin with numerical digits, [Enumerated Attribute Types](#)  
NMTOKENS attribute type, [NMTOKEN and NMTOKENS](#)  
Node interface, DOM, [DOM Interfaces, Extending Interfaces in the DOM](#), [Node](#)  
appendChild method, [The appendChild\(\) Method](#)  
attributes property, [The attributes Property](#), [Try It Out?Accessing Nodes in a NamedNodeMap](#)  
childNodes property, [Retrieving Information from the DOM](#)  
element information, accessing, [Try It Out?Accessing Element Information with Node](#)  
firstChild property, [Retrieving Information from the DOM](#)  
hasChildNodes method, [The hasChildNodes\(\) Method](#)  
insertBefore method, [The insertBefore\(\) Method](#)  
lastChild property, [Retrieving Information from the DOM](#)  
nextSibling property, [Retrieving Information from the DOM](#)  
nodeName property, [The nodeName and nodeValue Properties](#)  
nodeType property, [The nodeType Property](#)  
nodeValue property, [The nodeName and nodeValue Properties](#)  
ownerDocument property, [The ownerDocument Property](#)  
parentNode property, [Retrieving Information from the DOM](#)  
previousSibling property, [Retrieving Information from the DOM](#)  
removeChild method, [Removing Nodes from a Document](#)  
node() node test, XPath, [node\(\)](#)  
node-points, XPointer, [Points](#)  
node-set  
XPath, [XPath](#)  
node-tests, [Node tests](#)  
component of XPath location steps, [Appendix B: XPath Reference](#)  
full list (Appendix), [Node tests](#)  
implementation in XPath 1.0 and MSXML, [Node tests](#)  
NodeList interface, DOM, [NodeList](#)  
accessing items, [Try It Out?Accessing Items in a NodeList](#)  
item method, [NodeList and NamedNodeMap](#)  
length property, [NodeList and NamedNodeMap](#)  
nodes, [XPath](#)  
children, [Retrieving Information from the DOM](#), [The hasChildNodes\(\) Method](#)  
collections, [NodeList and NamedNodeMap](#)  
documents, [Creating XML Documents Through the DOM](#), [Removing Nodes from a Document](#)  
information, retrieving, [Getting Information About a Node](#)  
NamedNodeMaps, accessing, [Try It Out?Accessing Nodes in a NamedNodeMap](#)  
processing on several, [`<xsl:for-each>`](#)  
retrieving parts of, [Locations, Points and Ranges](#)  
Non-Breaking SSpace, [Whitespace in PCDATA](#)  
non-recoverable errors, [Even More About Errors?Catching Parsing Errors](#)  
non-validating parsers

external entities, [General Entities](#)  
non-validating processors  
skipped entities, [Interface org.xml.sax.ContentHandler \(SAX 2?Replaces DocumentHandler\)](#)  
none property, CSS, [Deploying CSS with XML](#)  
normalization, [Normalization](#)  
rules, [Normalization](#)  
staging database, [Using the Database](#)  
third normal form, [Normalization](#)  
normalize-space() function, XPath, [Functions](#)  
not() function, XPath, [`<xsl:copy-of>`](#), [Functions](#)  
notation declarations, [Notation Declarations](#), [Notations](#)  
FPIs, [Notations](#)  
public attribute, [Notations](#)  
system attribute, [Notations](#)  
NOTATION enumeration  
three notation types, [Enumerated Attribute Types](#)  
Notation interface, DOM, [Notation](#)  
NOTATION keyword, [Notation Declarations](#)  
NSS, URNs, [URNs](#)  
number() function, XPath, [Functions](#)

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## O

Object Management Group, see [omg](#)  
object models, [Hierarchies of Information](#)  
DOM (See DOM) \t, [DOM Implementations](#)  
XML, [XML As an Object Model](#)  
Object Request Broker  
provides underlying functionality for CORBA, [IIOP](#)  
objects  
communication with XML, [Using XML in an n-Tier Application](#)  
compared to interfaces, [What Are Interfaces?](#)  
distributed, [Using XML in an n-Tier Application](#)  
OMG, [IIOP](#)  
development of CORBA, [IIOP](#)  
omit-xml-declaration attribute  
xsl, [`<xsl:output>`](#)  
On Error Resume Next, Visual Basic, [Exceptions](#)  
online address book case study, [Possible Business Uses for XSLT-Driven Web Applications](#)  
address book stylesheet design, [Address Book Stylesheet Design](#), [User Interface Components](#)  
application business requirements, [Application Business Requirements](#)  
application server architecture, [Application Server Architecture](#), [Servlet Overview](#), [Servlet Reuse Without Modification](#)  
detailed named template descriptions, [Detailed Named Template Descriptions](#), [Form Template](#), [List Template](#), [Details Display Template](#), [XML Debugging Template](#)  
form template, [Error Message Node](#), [User Entry Form Using HTML <form> Element](#), [List Statistics Display](#), [Complete Form Template](#)  
user entry form, [Stateless Web Application](#)  
installing the application, [Installing the Application](#)  
list template, [Suppress List and Empty Results Message](#), [Next and Previous Links](#), [Row Iteration](#), [Selected Item Highlighting](#), [Item Row Display](#), [Complete List Template](#), [Try It Out - Adding a Field to the List](#), [Try It Out - Sorting the List](#)  
putting layout and variables together, [Putting the Layout and Variables Together](#), [Debug Page Selection Details](#)  
separating application server from end-user application, [Separating the Application Server from the End-User Application](#), [User Interface Connection to the XSLT Transform](#)  
servlet enhancements, [Servlet Enhancements](#)  
suggestions for enhancement of the application, [Suggestions for Enhancement of the Application](#)  
Oracle  
vs. SQL Server, [Putting Oracle to Work](#)  
XML technologies \r c12oracle, [Oracle's XML Technologies](#)  
Oracle XML parsers, [XML Parsers](#)  
order attribute  
xsl, [Sorting the Result Tree](#)  
order of attributes, [Attributes](#)  
ordinal references, XPointer, [XPointer Shorthand Syntaxes](#)  
org.xml.sax. interfaces and classes  
full list (Appendix), [Appendix F: SAX 2.0: The Simple API for XML](#)  
origin() function, XPointer, [XPointer Function Extensions to XPath](#)  
out of line links \r c11extend, [Extended Links](#)

output, XSLT, [<xsl:output>](#)  
overlapping tags, [Tags Cannot Overlap](#)

---

[!\[\]\(fd208c08d80e549fbd4498427638a3d0\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(fe984cf66499f01e6402fcf8e4b73247\_img.jpg\) NEXT >](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## P

parameter entities, [Parameter Entities](#)  
references to, [References to Parameter Entities](#)  
used to build DTDs from multiple files, [Parameter Entities](#)  
parameters, [Parameters](#)  
types, which can be passed by functions, [Types](#)  
using, [Parameters](#)  
which can be passed by XPath functions, [Functions](#)  
which can be passed by XSLT functions, [Functions](#)  
parent axis, XPath, [parent](#)  
parent/child relationships, XML, [Hierarchies in XML](#)  
parents, [Hierarchies in XML](#)  
Parsed Character DATA, see [pcdata](#)  
Parser interface, SAX  
basic interface for SAX parsers, [Interface org.xml.sax.Parser?Deprecated](#)  
Parser interface  
setErrorHandler method, [Even More About Errors?Catching Parsing Errors](#)  
parsers, [XML Parsers](#)  
Apache Xerces, [Where to Get SAX](#)  
comments, displaying, [Comments](#)  
errors, [Even More About Errors?Catching Parsing Errors](#)  
extraneous white space, [Whitespace in Markup](#)  
namespaces, [How XML Namespaces Work](#)  
parsing using SAX, [How to Receive SAX Events](#)  
relative URL references, [How It Works](#)  
validating and non-validating, [Chapter 5: Document Type Definitions](#)  
Xerces Java Parser, [Preparing the Ground](#)  
parsing XML \r c2pars, [Parsing XML](#)  
PATH and CLASSPATH  
modifying, [Modify the Path and Class Path](#)  
pattern facet, [Constraining Facets](#)  
attributes, [Constraining Facets](#)  
PCDATA, [Tags and Text and Elements, Oh My!](#)  
illegal characters \r c2ill, [Illegal PCDATA Characters](#)  
white space, [Whitespace in PCDATA](#)  
Personal Web Server, see [pws](#)  
PIs \t See processing instructions, [Processing Instructions](#)  
points, XPointer, [XPointer: Pointing to Document Fragments, Locations, Points and Ranges](#)  
character-points, [Points](#)  
defining, [Points](#)  
node-points, [Points](#)  
position property, CSS  
values, [Integrating Position](#)  
position() function, XPath, [Modes, Functions](#)  
POST method, HTTP, [HTTP, Using HTTP for SOAP](#)  
using to call RPC, [Try It Out?Using HTTP POST to Call Our RPC](#)  
Post Schema Validation InfoSet, see [psvi](#)

PRE> tag, [Whitespace in PCDATA](#)  
preceding axis, XPath, [preceding](#)  
preceding-sibling axis, XPath, [preceding-sibling](#)  
predicate expressions, CSS-selector syntax, [Centralizing with the <style> Element](#)  
predicates, XPath  
filtering node-test result sets, [Appendix B: XPath Reference](#)  
XPath functions usable in, [Functions](#)  
prefixes to define namespaces  
reasons why not used in XML \r c7nopre, [So Why Doesn't XML Just Use These Prefixes?](#)  
prefixes to define namespaces \r c7prefix, [Using Prefixes](#)  
Presentation layer, [n-Tier Architecture](#)  
primary keys, [SQL](#)  
priority attribute  
xsl, [Template Priority](#)  
processing instructions \r c2pi, [Processing Instructions](#)  
processing-instruction(name?) node test, XPath, [processing-instruction\( name? \)](#)  
ProcessingInstruction interface, DOM, [ProcessingInstruction](#)  
processingInstruction method, ContentHandler interface, [processingInstruction](#)  
programming  
imperative vs. declarative, [Imperative Versus Declarative Programming](#)  
side effects in, [XSLT Has No Side Effects](#)  
properly nesting tags, [Tags Cannot Overlap](#)  
properties, [Hierarchies in HTML](#)  
pseudo classes, CSS-selector syntax, [Centralizing with the <style> Element](#)  
PSVI, [The XML Schema Document](#)  
public identifiers, [Public Identifiers](#)  
function similarly to namespace names, [Public Identifiers](#)  
optional URI reference, [Public Identifiers](#)  
registered and unregistered, [Public Identifiers](#)  
PUBLIC keyword, [Public Identifiers](#)  
PWS, [Running the Examples](#)  
fine for internal intranet websites, [Running the Examples](#)  
installing on Windows 98, [Installing PWS On Windows 98](#)  
managing under Windows 98, [Managing PWS Under Windows 98](#)  
PWS console, [Managing PWS Under Windows 98](#)  
root directory, [Managing PWS Under Windows 98](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## Q-R

QName \t See qualified names, [How XML Namespaces Work](#)  
qualified and unqualified elements and attributes, [Declaration Defaults](#)  
qualified names, [How XML Namespaces Work](#)  
to define behaviors for actuate attribute, XLink, [Behavior Attributes \(actuate and show\)](#)  
querying, [Querying](#)  
quotation marks, attributes, [Attributes](#)  
range() function, XPointer, [XPointer Function Extensions to XPath](#)  
range-inside() function, [XPointer Function Extensions to XPath](#)  
range-to() function, XPointer, [How Do We Select Ranges?](#)  
ranges, XPointer, [XPointer: Pointing to Document Fragments, Locations, Points and Ranges](#)  
collapsed, [Ranges](#)  
selecting, [How Do We Select Ranges?](#)  
with multiple locations, [Ranges with Multiple Locations](#)  
ranges, XPointer \r c11range, [Ranges](#)  
RBDMS, [Databases, Yesterday and Today](#)  
structure, [Databases, Yesterday and Today](#)  
RDDL, [RDDL](#)  
read method, Java, [How to Receive SAX Events](#)  
records, [Databases, Yesterday and Today](#)  
Recordset object  
in SQL queries, [Using XML in an n-Tier Application](#)  
Recordset object, ADO, [Using XML in an n-Tier Application](#)  
save method, [Using XML in an n-Tier Application](#)  
recoverable errors, [Even More About Errors-Catching Parsing Errors](#)  
recursion, [`<xsl:copy>`](#)  
recursive descent operator //, [Try It Out-Attributes and Attribute Sets in Action](#)  
recursive entity reference, [General Entities](#)  
redefine declaration, [`<redefine>`](#)  
can create inherited types, [`<redefine>`](#)  
each declaration can only be redefined once, [`<redefine>`](#)  
registered public identifiers, [Public Identifiers](#)  
related groups of attributes, [Related Groups of Attributes](#)  
Relational Database Management System, see [rbdms](#)  
relational databases  
differences to XML documents, [XML Databases](#)  
relative value, position property, CSS, [Integrating Position](#)  
RELAX NG, [RELAX NG](#)  
Remote Method Invocation, see [java rmi](#)  
Remote Procedure Calls, see [rpc](#)  
remote resources, [Try It Out-Adding Remote Resources to the Extended Link](#)  
adding to extended link, [Try It Out-Adding Remote Resources to the Extended Link](#)  
removeChild method, Node interface, [Removing Nodes from a Document](#)  
replaceData method, CharacterData interface, [Modifying Strings](#)  
required values  
attribute value declarations, [Required Values](#)  
Resource Directory Description Language, see [rddl](#)

resource-type elements, XLink, [The type Attribute](#)  
adding local resource to extended link, [Try It Out-Adding a Local Resource to the Extended Link](#)  
extended links, [Resource-type Elements](#)  
resources, [What Exactly are URIs?](#)  
displaying, [Behavior Attributes \(actuate and show\)](#)  
extended links, adding to, [Resource-type Elements](#)  
labels for, [Semantic Attributes \(role and title\)](#)  
multiple, [The from and to Attributes](#)  
remote, adding to extended link, [Try It Out-Adding Remote Resources to the Extended Link](#)  
specifying when retrieved, [Behavior Attributes \(actuate and show\)](#)  
Response object, [Using XML in an n-Tier Application](#)  
ContentType property, [Using XML in an n-Tier Application](#)  
restriction declarations, [`<restriction>`](#)  
restriction simpleType  
creating, [Try It Out-"Weird" XML Schemas-Creating a Restriction simpleType](#)  
result tree, [What is XSL?](#)  
sections of, copying from source tree, [Copying Sections of the Source Tree to the Result Tree](#)  
resume method, IMXReaderControl interface, [The Microsoft Way with SAX](#)  
reusability  
n-tier architecture, [n-Tier Architecture](#)  
role attribute, XLink, [XLink Attributes](#), [Semantic Attributes \(role and title\)](#)  
arc-type elements, [Arcs](#)  
resource-type elements, [Resource-type Elements](#)  
root element, [An XML Document Can Have Only One Root Element](#)  
namespaces, [Declaring Namespaces on Descendants](#)  
round() function, XPath, [Functions](#)  
rows, [Databases, Yesterday and Today](#)  
RPC, [Remote Procedure Calls, Chapter 10: SOAP](#)  
making, [What Is an RPC?](#)  
RPC protocols  
DCOM, [DCOM](#)  
IIOP, [IIOP](#)  
Java RMI, [Java RMI](#)  
SOAP, [The New RPC Protocol: SOAP](#)  
run-in element, CSS primary containers, [Deploying CSS with XML](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## S

save method, MSXML, [Vendor-Specific Extensions](#), [DOM Extensions](#)  
save method, Recordset object, ADO, [Using XML in an n-Tier Application](#)  
SAX, [What are the Pieces that Make Up XML?](#), [Install JRE](#), [Chapter 9: The Simple API for XML \(SAX\)](#)  
Attributes class, [Extracting Attributes](#)  
attributes, extracting, [Extracting Attributes](#)  
benefits and limitations, [Good SAX and Bad SAX](#)  
character data, extracting, [Extracting Character Data](#)  
classes, interfaces and their methods, [Appendix F: SAX 2.0: The Simple API for XML](#)  
ContentHandler interface, [How to Receive SAX Events](#)  
DefaultHandler class, [How to Receive SAX Events](#)  
development, [What is SAX, and Why Was it Invented?](#)  
ErrorHandler interface, [Even More About Errors-Catching Parsing Errors](#)  
event-driven, [What is SAX, and Why Was it Invented?](#)  
exceptions, [Error Handling](#)  
implementations, inheriting, [How to Receive SAX Events](#)  
installing, [Where to Get SAX](#)  
JDK, [Where to Get SAX](#)  
methods, overriding, [How to Receive SAX Events](#)  
Microsoft application, [The Microsoft Way with SAX](#)  
parser errors, [Even More About Errors-Catching Parsing Errors](#)  
string buffers, [Extracting Character Data](#)  
validation, [More About Errors-Using the Locator Object](#)  
SAX parsers, [A Brief History of SAX](#)  
SAX 2.0 interface specification, [Appendix F: SAX 2.0: The Simple API for XML](#)  
SAXException, [Error Handling](#)  
SAXException interface, SAX  
encapsulate a general SAX error or warning, [Class org.xml.sax.SAXException](#)  
SAXNotRecognizedException interface, SAX, [Class org.xml.sax.SAXNotRecognizedException \(SAX 2\)](#)  
SAXNotSupportedException interface, SAX, [Class org.xml.sax.SAXNotSupportedException \(SAX 2\)](#)  
Saxon parser  
implements XPath 1.0, [Appendix B: XPath Reference](#)  
Saxon processor  
conformance to XSLT 1.0, [Appendix C: XSLT Reference](#)  
SaxParseException class, [Try It Out-Catching Parsing Errors](#)  
SAXParseException interface, SAX  
encapsulate an XML parse error or warning, [Class org.xml.sax.SAXParseException](#)  
scalability  
n-tier architecture, [n-Tier Architecture](#)  
Scalable Vector Graphics \t See SVG, [Why "Extensible"?](#)  
Schema datatypes reference, [Appendix E: Schema Datatypes Reference](#)  
constraining facets, [Constraining Facets](#)  
schema element, [<schema>](#)  
attributeFormDefault attribute, [Declaration Defaults](#)  
elementFormDefault attribute, [Declaration Defaults](#)  
import declaration, [<import>](#)  
notation element, [Notations](#)

targetNamespace attribute, [Target Namespaces](#)  
schema validators  
validating a document against its vocabulary, [The XML Schema Document](#)  
schemaLocation attribute, [<import>, <include>](#)  
schemas, [What are the Pieces that Make Up XML?](#), [What Do Namespace URIs Really Mean?](#)  
Examplotron, [Examplotron](#)  
RELAX NG, [RELAX NG](#)  
Schematron, [Schematron](#)  
XDR, [XDR](#)  
Schematron, [Schematron](#)  
schemes, XPointer, [XPointer Schemes](#)  
scope, constants, [Variables, Constants, and Named Templates](#)  
SDD \t See Standalone Document Declaration, [Standalone](#)  
search criteria input form, address book stylesheet design, [User Interface Components](#)  
search results list, address book stylesheet design, [User Interface Components](#)  
Secure Sockets Layer, see [ssl](#)  
security  
n-tier architecture, [n-Tier Architecture](#)  
select attribute  
xsl, [<xsl:apply-templates>, <xsl:for-each>, <xsl:copy-of>](#), [Sorting the Result Tree](#), [Variables, Constants, and Named Templates](#)  
SELECT statement  
example, [SQL](#)  
selectNodes() method, [XML Parsers](#)  
selector, CSS rules, [Enter Cascading Style Sheets](#)  
selectSingleNode() method, [XML Parsers](#)  
self axis, XPath, [self](#)  
semantic attributes, XLink, [Semantic Attributes \(role and title\)](#)  
extended links, [Extended Links](#)  
sequence declaration, XML Schemas content model, [<sequence>](#)  
complexType definitions, [<sequence>](#)  
minOccurs and maxOccurs attributes, [<sequence>](#)  
servlet overview  
online address book case study, [Servlet Overview](#)  
servlet reuse without modification  
online address book case study, [Servlet Reuse Without Modification](#)  
setDocumentLocator method, ContentHandler interface, [More About Errors-Using the Locator Object](#), [Try It Out-Using the Locator](#)  
setErrorHandler method, Parser interface, [Even More About Errors-Catching Parsing Errors](#)  
setNamedItem method, NamedNodeMap interface, [Try It Out-Creating an XML Document from Scratch](#)  
seven-bit ASCII encoding, [Encoding](#)  
SGML, [A Brief History of Markup](#)  
complexity of, [Why Use Attributes?](#)  
sharing vocabularies, [Sharing Vocabularies](#)  
shorthand syntaxes of XPointer \r c11short, [XPointer Shorthand Syntaxes](#)  
show attribute, XLink, [XLink Attributes](#), [Behavior Attributes \(actuate and show\)](#)  
arc-type elements, [Arcs](#)  
sibling/sibling relationships, XML, [Hierarchies in XML](#)  
side effects in programming, [XSLT Has No Side Effects](#)  
Simple API for XML, see [sax](#)  
simple content, [Hierarchies in XML](#)  
simple links, [Link Types](#)  
creating, [Try It Out-Creating Simple Links](#)  
simple links \r c11simp, [Simple Links](#)  
simple links, XLink, [XLink](#)

Simple Object Access Protocol, see [soap](#)  
simple-type elements, XLink, [The type Attribute](#)  
simpleContent declarations, [Extending <simpleType> Definitions](#)  
simpleType declarations  
creating user defined datatypes, [<restriction>](#)  
simpleTypes, [Important](#)  
creating user defined datatypes, [<simpleType>](#)  
extending definitions, [Extending <simpleType> Definitions](#), [Extending <simpleType> Definitions](#)  
facets, [<restriction>](#)  
redefining contents, [<redefine>](#)  
restricting definitions, [Restricting <simpleType> Definitions](#)  
table of all built into XML Schemas, [Built-in Datatypes](#)  
union declarations, [<union>](#)  
single quotation marks, attributes, [Attributes](#)  
skippedEntity() method  
ContentHandler interface, SAX2, [Interface org.xml.sax.ContentHandler \(SAX 2-Replaces DocumentHandler\)](#)  
SOAP, [Chapter 10: SOAP](#)  
advantages, [The New RPC Protocol: SOAP](#)  
creating an RPC server in ASP, [The New RPC Protocol: SOAP](#)  
encoding rules, [How SOAP Works, Encoding Rules](#)  
encodingStyle attribute, [Encoding Rules](#)  
envelope, [How SOAP Works, The Envelope](#)  
HTTP, [How SOAP Works, HTTP](#)  
messages are basically XML documents, [How SOAP Works](#)  
network transport, [The Network Transport](#)  
performing distributed programming, [Chapter 10: SOAP](#)  
reasons for using HTTP, [Why HTTP for SOAP?](#)  
RPC protocol, [The New RPC Protocol: SOAP](#)  
use of XML and HTTP, [The New RPC Protocol: SOAP](#)  
using a proper SOAP envelope for addNumbers, [Try It Out-Using a Proper SOAP Envelope for addNumbers](#)  
using HTTP, [Using HTTP for SOAP](#)  
using HTTP POST to call RPC, [Try It Out-Using HTTP POST to Call Our RPC](#), [How It Works](#)  
SOAP attributes  
actor attribute, [The actor Attribute](#)  
SOAP elements  
<Body> element, [<Body>](#)  
<Envelope> element, [<Envelope>](#)  
<Fault> element, [<Fault>](#)  
<faultcode> element, [<Fault>](#)  
<Header> element, [<Header>](#)  
SOAP examples  
requirements for running, [Running the Examples](#)  
SOAP requests  
error messages, [Using HTTP for SOAP](#)  
HTTP, [Listening for Requests](#)  
SOAP Toolkit  
available from, [WSDL](#)  
SOAPAction, HTTP header, [Using HTTP for SOAP](#)  
sorting in XSLT, [Sorting the Result Tree](#)  
source tree, [What is XSL?](#)  
copying to result tree, [Copying Sections of the Source Tree to the Result Tree](#)  
splitText method, Text interface, [Splitting Text](#)  
splitting text, [Splitting Text](#)  
SQL, [SQL](#)  
HTTP, making queries available over, [SQL Server](#), [XSQL Servlet](#)

joins, [Joins](#)  
queries, [Retrieving Data from Multiple Tables](#)  
standardization, [SQL](#)  
stored procedures, [n-Tier Architecture](#)  
XML documents, tool for generating from queries, [XML SQL Utility for Java](#)  
SQL \r c12sql, [SQL](#)  
SQL Server  
FOR XML EXPLICIT clause, [SQL Server](#)  
FOR XML RAW clause, [SQL Server](#)  
using, [Putting SQL Server to Work](#)  
vs. Oracle, [Putting Oracle to Work](#)  
XML support \r c12use, [SQL Server](#)  
SQL Update Grams, [SQL Server](#)  
sql  
query> element, [SQL Server](#), [Putting SQL Server to Work](#)  
SSL, [Security](#)  
staging databases, [Using the Database](#)  
standalone attribute  
XML declaration, [Standalone](#)  
xsl, [<xsl:output>](#)  
Standalone Document Declaration, [Standalone](#)  
Standard Generalized Markup Language, see [sgml](#)  
start-point() function, XPointer, [XPointer Function Extensions to XPath](#)  
start-tags, [Tags and Text and Elements, Oh My!, Every Start-tag Must Have an End-tag](#)  
startDocument method, ContentHandler interface, [How to Receive SAX Events](#)  
startDocument method, IVBSAXContentHandler interface, [The Microsoft Way with SAX](#)  
startElement method, ContentHandler interface, [How to Receive SAX Events](#)  
startElement method, IVBSAXContentHandler interface, [The Microsoft Way with SAX](#)  
starts-with() function, XPath, [The \\$searchlist Variable](#), [Functions](#)  
static value, position property, CSS, [Integrating Position](#)  
stored procedures, [n-Tier Architecture](#)  
string() function, XPath, [Functions](#)  
string-length() function, XPath, [XPath Functions](#), [Error Message Node](#), [Functions](#)  
strings  
handling, [Handling Complete Strings](#)  
modifying, [Modifying Strings](#)  
string buffers, [Extracting Character Data](#)  
sub-strings, [Handling Substrings](#)  
structured data, [What's a Document Type?](#)  
Structured Query Language, see [sql](#)  
style element  
container for style rules, [Centralizing with the <style> Element](#)  
stylesheets, [What is XSL?](#)  
default in Internet Explorer, [So What is XML?](#)  
need for, [The Need for Style\(sheets\)](#)  
sub-strings, handling, [Handling Substrings](#)  
substitution  
element substitution, [Element Substitution](#), [When to Use Element Substitution](#)  
type substitution, [Type Substitution](#), [When to Use Type Substitution](#)  
substring() function, XPath, [Functions](#)  
substring-after() function, XPath, [Functions](#)  
substring-before() function, XPath, [Functions](#)  
substringData method, CharacterData interface, [Handling Substrings](#)  
sum() function, XPath, [Functions](#)  
suspend method, IMXReaderControl interface, [The Microsoft Way with SAX](#)

SVG, [Why "Extensible"?](#)

system identifier

Document Type Declaration, [System Identifiers](#)

SYSTEM keyword plus URI reference, [System Identifiers](#)

SYSTEM keyword, [System Identifiers](#)

system-property() function, XSLT, [Functions](#)

[!\[\]\(c85043d66a66d81e5c4352703091696f\_img.jpg\) PREVIOUS](#)

[< Free Open Study >](#)

[!\[\]\(3b9782e7e71057f1b594fa78238d8d8a\_img.jpg\) NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## T

table element, CSS primary containers, [Deploying CSS with XML](#)  
table tags, [Suppress List and Empty Results Message](#)  
tables, [Databases, Yesterday and Today](#)  
data retrieval, [Retrieving Data from Multiple Tables](#)  
denormalization, [Normalization](#)  
joins, [Joins](#)  
normalization, [Normalization](#)  
tags  
case-sensitivity, [Case-Sensitivity](#)  
end, [Tags and Text and Elements, Oh My!, Every Start-tag Must Have an End-tag](#)  
onClik attribute, [XML Debugging Template](#)  
properly nesting, [Tags Cannot Overlap](#)  
start, [Tags and Text and Elements, Oh My!, Every Start-tag Must Have an End-tag](#)  
Tamino XML Database, [XML Databases](#)  
targetNamespace attribute  
Chameleon Components, [<include>](#)  
import declaration, [<import>](#)  
include declaration, [<import>](#), [<include>](#)  
schema element, [Target Namespaces](#)  
td tags, [Suppress List and Empty Results Message](#)  
id attribute, [Selected Item Highlighting](#)  
templates  
calling other templates, [<xsl:apply-templates>](#)  
context node, effect on, [How Do Templates Affect the Context Node?](#)  
declarative programming, [Declarative Programming](#)  
defining, [<xsl:template>](#)  
matching against document root, [Modes](#)  
modes, [Modes](#)  
named, [<xsl:template>](#), [Variables, Constants, and Named Templates](#)  
parameters, [Parameters](#)  
priority, [Template Priority](#)  
recursion, [<xsl:copy>](#)  
transforming XML documents to web pages, [Modes](#)  
xsl, [<xsl:for-each>](#)  
XSLT stylesheets, [How XSLT Stylesheets Work-Templates](#)  
text  
result tree, inserting text into, [<xsl:text>](#)  
splitting, [Splitting Text](#)  
text files, [Text Files](#)  
Text interface, DOM, [Working With Text](#), [Text](#)  
splitText method, [Splitting Text](#)  
text() node test, XPath, [text\(\)](#)  
text-only elements, [Mixed Content](#)  
third normal form, [Normalization](#)  
title attribute  
resource-type elements, XLink, [Resource-type Elements](#)

title attribute, XLink, [XLink Attributes](#), [Semantic Attributes \(role and title\)](#)  
arc-type elements, [Arcs](#)  
title property  
document object, [Hierarchies in HTML](#)  
title-type elements, XLink, [The type Attribute](#)  
extended links, [Title-type Elements](#)  
to attribute, XLink, [XLink Attributes](#), [The from and to Attributes](#)  
arc-type elements, [Arcs](#)  
tool tips, [Semantic Attributes \(role and title\)](#)  
top-level elements, [`<xsl:output>`](#)  
totalDigits facet  
attributes, [Constraining Facets](#)  
tr tags, [Suppress List and Empty Results Message](#)  
transformNode method, DOMDocument object, [Listening for Requests](#)  
translate() function, XPath, [Functions](#)  
trees, [Hierarchies in XML](#)  
traversing with DOM, [Retrieving Information from the DOM](#)  
true() function, XPath, [`<xsl:copy-of>`](#), [Functions](#)  
try blocks  
exception handling, [Exceptions](#)  
type attribute, [Type Substitution](#)  
type substitution, [Type Substitution](#)  
type attribute, CSS, [Deploying CSS with XML](#)  
type attribute, XLink, [XLink Attributes](#), [The type Attribute](#)  
type override, [Type Substitution](#)  
users may override, [Type Substitution](#)  
type substitution, [Type Substitution](#)  
using, [When to Use Type Substitution](#)

---

[!\[\]\(7295a990d21eb6900570b0b15c03302b\_img.jpg\) PREVIOUS](#)

[`< Free Open Study >`](#)

[!\[\]\(6f40a73d115e0c8c2724865825ccdb0b\_img.jpg\) NEXT](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## U

UDDI, [UDDI](#)

UName, [Default Namespaces](#)

Unicode, [Unicode](#)

uniqueness in databases, [SQL](#)

Universal Discovery, Description, and Integration, see [uddi](#)

universal name, see [uname](#)

Unix

newline characters, [End-of-Line Whitespace](#)

unparsed-entity-uri() function, XSLT, [Functions](#)

unregistered public identifiers, [Public Identifiers](#)

URIs

definition, [What Exactly are URIs?](#)

namespace URIs, meaning of \r c7mean, [What Do Namespace URIs Really Mean?](#)

using to define namespace prefixes, [So Why Doesn't XML Just Use These Prefixes?](#)

XPointer expression, appending to, [Appending XPointer Expressions to URIs](#)

URIs \r c7uri, [What Exactly are URIs?](#)

URLs, [URLs](#)

namespaces \r c7url, [Why Use URLs for Namespaces, Not URNs?](#)

URNs, [URNs](#)

use attribute

attribute declarations, [Attribute Use](#)

user defined datatypes

XML Schemas, [User Defined Datatypes](#)

UTF-16, [Unicode](#)

DOMString, [DOMString](#)

UTF-8, [Unicode](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## V

valid element names, [Element Names](#)  
validating and non-validating parsers, [Chapter 5: Document Type Definitions](#)  
validation, SAX, [More About Errors?Using the Locator Object](#)  
values, attributes, [Attributes](#)  
variables in XSLT, [Variables, Constants, and Named Templates](#)  
binding, [Variables, Constants, and Named Templates](#)  
version attribute  
xsl, [<xsl:output>](#)  
visibility property, CSS  
hiding content, [Hiding Content](#)  
Visual Basic  
compared to Java, [The Microsoft Way with SAX](#)  
exception handling, [Exceptions](#)  
Visual Basic code generator  
based on XDR, [Visual Basic Code Generator](#)  
vocabularies  
XML, [Why "Extensible"? What's a Document Type?](#)

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

&lt; PREVIOUS

&lt; Free Open Study &gt;

NEXT &gt;

# Index

## W

WAR, [Servlet Overview](#)

WAR file

case study in packaged in, [Servlet Overview](#)

warning method, ErrorHandler interface, [Even More About Errors?Catching Parsing Errors](#)

WD-xsl, [MSXML](#)

Web Application Resource, see [war](#)

web pages

transforming XML documents to, [Modes](#)

web resources, [Appendix G: Useful Web Resources](#)

web server

installing on Windows, [Running the Examples](#)

web servers

reducing load on with XML, [Reducing Server Load](#)

Web Services

functionality provided, [What Are Web Services?](#)

Web Services Description Language, see [wsdl](#)

Web Services Toolkit

available from, [WSDL](#)

web site content

with XML, [Web Site Content](#)

Weird Al Yankovic, Dare to be Stupid, [Tags and Text and Elements, Oh My!](#)

adding attributes, [Attributes, Try It Out?"Weird" XML Schema?Adding Attributes](#)

adding comments, [Comments](#)

adding markup to comments, [Comments](#)

adding processing instruction, [Is the XML Declaration a Processing Instruction?](#)

create DTD, [Try It Out?"I Want a New DTD", Try It Out?"I Want a New DTD"?Part 4, Try It Out?"I Want a New DTD"?Part 5, Notation Declarations](#)

create XML Schema, [Try It Out?"Weird" XML Schema, How It Works](#)

xsl\, [How It Works](#)

creating a restriction simpleType, [Try It Out?"Weird" XML Schemas?Creating a Restriction simpleType](#)

declaring, [Standalone](#)

improving DTD, [Try It Out?"I Want a New DTD"?Part 2, Try It Out?"I Want a New DTD"?Part 3](#)

using a global attribute group, [<attributeGroup>](#)

using built-in XML Schema datatypes, [Try It Out?"Weird" XML Schema?Using the Built-in XML Schema datatypes](#)

well-formed XML, [Chapter 2: Well-Formed XML](#)

WHERE clause, [Putting SQL Server to Work](#)

white space

end-of-line, [End-of-Line Whitespace](#)

in markup, [Whitespace in Markup](#)

preserving using <xsl\, [<xstext>](#)

white space \r c2white, [Whitespace in PCDATA](#)

white space stripping

new line characters, [End-of-Line Whitespace](#)

whiteSpace facet

attributes, [Constraining Facets](#)

wildcard character \*, CSS-selector syntax, [Centralizing with the <style> Element](#)

Windows

newline characters, [End-of-Line Whitespace](#)

windows-1252, [Unicode](#)

World Wide Web Consortium, [What Is the World Wide Web Consortium?](#)

Wrox validation program

installing, [Install the Wrox Package](#)

WSDL, [WSDL](#)

W3C \t See World Wide Web Consortium, [What Is the World Wide Web Consortium?](#)

W3C recommendations

linking and querying, [Chapter 13: Linking and Querying XML](#)

W3C Style working group, [Enter Cascading Style Sheets](#)

---

[◀ PREVIOUS](#)

[< Free Open Study >](#)

[NEXT ▶](#)

# Index

## X

Xalan processor

conformance to XSLT 1.0, [Appendix C: XSLT Reference](#)

implements XPath 1.0, [Appendix B: XPath Reference](#)

XDK, [Oracle's XML Technologies](#)

XML SQL Utility for Java, [XML SQL Utility for Java](#)

XDR, [Visual Basic Code Generator](#)

XDR schemas, [XDR](#)

Xerces, [Parsing XML](#)

XHTML

adding external stylesheets, [Linking Stylesheets in HTML](#)

linking stylesheets, [Linking Stylesheets in HTML](#)

namespace, [How XML Namespaces Work](#), [When Should I Use Namespaces?](#)

XML language that programs a browser, [Deploying CSS with XML](#)

XHTML and HTML CSS\rCSS\_HTMLetc, [The Basics of HTML/XHTML/CSS](#)

XHTML DTDs, [Sharing Vocabularies](#)

XLink, [What are the Pieces that Make Up XML?](#), [XML Linking](#)

actuate attribute, [Behavior Attributes \(actuate and show\)](#)

adding a local resource to an extended link, [Try It Out-Adding a Local Resource to the Extended Link](#)

adding arcs to extended link, [Try It Out-Adding Arcs to the Extended Link](#)

attributes, [XLink Attributes](#)

default attributes, [Defaulting XLink Attributes](#)

extended links, [Link Types](#), [Try It Out-Creating an Extended Link](#)

extended links \r c11extend, [Extended Links](#)

from attribute, [The from and to Attributes](#)

global attributes, [XLink](#)

global attributes \r c11attr, [XLink Attributes](#)

href attribute, [The href Attribute](#)

linking, [Chapter 13: Linking and Querying XML](#)

role attribute, [Semantic Attributes \(role and title\)](#)

show attribute, [Behavior Attributes \(actuate and show\)](#)

simple links, [Link Types](#)

simple links \r c11simp, [Simple Links](#)

title attribute, [Semantic Attributes \(role and title\)](#)

to attribute, [The from and to Attributes](#)

type attribute, [The type Attribute](#)

XLink \r c11xlink, [XLink](#)

XML, [Chapter 1: What is XML?](#)

attributes \r c2att, [Attributes](#)

comments \r c2comm, [Comments](#)

CSS stylesheets, [Deploying CSS with XML](#)

databases, [Storing XML in a Database](#), [Database Vendors and XML](#)

deploying CSS with\rCSS\_XML, [Deploying CSS with XML](#)

DOM, [XML As an Object Model](#), [The XML DOM](#)

errors, [Errors in XML](#)

example file, [So What is XML?](#)

extensibility of, [Why "Extensible"?](#)

hierarchies in, [Hierarchies in XML](#)

HTML, difference from, [HTML and XML: Apples and Red Delicious Apples](#)

HTML, transforming to, [Who Needs a Database Anyway?](#)

integrating into databases \r c12xdata, [Integrating XML into the Database Itself](#)

mapping to relational structure, [Mapping XML to a Relational Structure](#)

Microsoft technologies \r c12msoft, [Microsoft's XML Technologies](#)

n-tier architecture, using in \r c12xml, [Using XML in an n-Tier Application](#)

namespaces \r c7name, [How XML Namespaces Work](#)

Oracle technologies \r c12oracle, [Oracle's XML Technologies](#)

parsing \r c2pars, [Parsing XML](#)

problems in storing in relational databases, [XML Databases](#)

reasons for using \r c1reason, [What Does XML Buy Us?](#)

SAX, [Chapter 9: The Simple API for XML \(SAX\)](#)

SQL Server support \r c12use, [SQL Server](#)

SQL Update Grams, [SQL Server](#)

subset of SGML, [So What is XML?](#)

using in XSLT-driven web applications, [Using XML in an XSLT-Driven Application](#)

well-formed, [Chapter 2: Well-Formed XML](#)

where used \r c1use, [Where Is XML Used, and Where Can it Be Used?](#)

white space, [Whitespace in PCDATA](#)

XML \r c1xml, [So What is XML?](#)

XML databases, [XML Databases](#)

benefits, [XML Databases](#)

stores data natively in XML, [XML Databases](#)

XML debugging page, address book stylesheet design, [User Interface Components](#)

XML declaration, [XML Declaration](#)

character encoding, specifying, [Specifying A Character Encoding for XML](#)

not a processing instruction, [Is the XML Declaration a Processing Instruction?](#)

standalone attribute, [Standalone](#)

XML Developers Kit \t See XDK, [Oracle's XML Technologies](#)

XML document

adding external stylesheets, [Linking Stylesheets in HTML](#)

XML document types \t See document types, [Why Do We Need Namespaces?](#)

XML documents

creating through the DOM, [Creating XML Documents Through the DOM](#)

trees, [Retrieving Information from the DOM](#)

validating against XML Schemas, [`<import>`](#)

validation using SAX, [More About Errors-Using the Locator Object](#)

XML files

opening in Internet Explorer, [So What is XML?](#)

XML linking

two-way links, [XML Linking](#)

XML linking \r c11xml, [XML Linking](#)

XML Linking Working Group \r c11xml, [XML Linking](#)

XML parser, Oracle, [XML Parsers](#)

XML parsers, see [parsers](#)

DOM, [The XML DOM](#)

SAX, [A Brief History of SAX](#)

XML Protocol Activity, [What Are Web Services?](#)

XML Query Working Group, [Querying](#)

XML resources

Cover Pages, [Sharing Vocabularies](#)

xml-dev, [Sharing Vocabularies](#)

XML Schema built-in datatypes

derived types, [XML Schema Built-in Datatypes](#)

primitive types, [XML Schema Built-in Datatypes](#)

XML Schema datatypes, [XML Schema Datatypes](#)

XML Schema definitions, see [xsd](#)

XML Schema elements

appinfo element, [Annotations](#)

documentation element, [Annotations](#)

element element, [<element>](#)

group element, [<group>](#)

xs\, [Appendix D: Schema Element and Attribute Reference](#)

XML Schema elements and attributes

quick reference, [Appendix D: Schema Element and Attribute Reference](#)

XML Schema instance attributes, [XML Schema Instance Attributes](#)

XML Schema namespace

declaring, [The XML Schema Namespace](#)

XML Schema Recommendation, [Inheritance](#)

schema validators, [The XML Schema Document](#)

XML Schemas, [Chapter 6: XML Schemas](#), [Chapter 7: Advanced XML Schemas](#)

alternatives to XML Schemas, [Alternative Schema Formats](#)

attribute declarations, [<attribute>](#)

benefits, [Benefits of XML Schemas](#), [XML Schemas Use XML Syntax](#), [XML Schema Namespace Support](#), [XML Schema Datatypes](#)

built in datatypes, [Built-in Datatypes](#)

content models, [Content Models](#)

creating from multiple documents, [Creating a Schema from Multiple Documents](#), [<import>](#), [<include>](#), [<redefine>](#)

creating reusable global types, [Try It Out-Creating Reusable Global Types](#)

datatypes, [Datatypes](#)

declaration defaults, [Declaration Defaults](#)

declaring content model, [Running the Samples](#)

declaring elements, [<element>](#)

declaring namespaces, [<schema>](#), [The XML Schema Namespace](#)

declaring notations, [Notations](#)

defining attribute groups, [<attributeGroup>](#)

defining name vocabulary, [Running the Samples](#)

definition, [Chapter 6: XML Schemas](#)

dividing schemas into modules, [<include>](#)

documenting, [Documenting XML Schemas](#)

element declarations, [Naming Elements](#), [Element Qualified Form](#), [Cardinality](#), [Default and Fixed Values](#), [Element Wildcards](#), [<complexType>](#), [<group>](#)

ENTITY functionality not provided, [Do We Still Need DTDs?](#)

inheritance, [Inheritance](#)

list declarations, [<list>](#)

need for DTDs, [Do We Still Need DTDs?](#)

qualified and unqualified elements and attributes, [Declaration Defaults](#)

referring to an existing global element, [Referring to an Existing Global Element](#)

schemaLocation attribute, [<import>](#)

simpleTypes, [<simpleType>](#), [<restriction>](#), [<list>](#), [<union>](#)

specifying element type, [<element>](#), [Global vs. Local](#), [Creating a Local Type](#), [Using a Global Type](#)

substitution, [Substitution](#), [Type Substitution](#), [Element Substitution](#)

support for, [Running the Samples](#)

target namespaces, [Target Namespaces](#)

type attribute, [Type Substitution](#)

type override, [Type Substitution](#)

union declarations, [<union>](#)

user defined datatypes, [User Defined Datatypes](#), [<simpleType>](#)

XML declaration, [Running the Samples](#)

xsl\, [Running the Samples](#)

XML Schemas element

schema element, [`<schema>`](#)

XML SQL Utility for Java, [XML SQL Utility for Java](#)

XML template files, [SQL Server](#)

XML Web Services case study, [Case Study 2 - XML Web Services](#)

building the client, [Building the Client](#)

building the server, [Building the Server](#), [An XML Cookbook](#), [GetRecipes](#), [External Documents](#)

external documents, [document\(\) function](#)

combining XSLT processing with ASP listener, [Listening for Requests](#)

how we built it, [How We Built It](#)

listening for SOAP requests, [Listening for Requests](#)

overview, [Case Study 2 - XML Web Services](#)

processing the SOAP request, [Processing the Request](#), [XSLT Advantages](#), [SOAP Transformations](#), [recipes.xsl](#)

SOAP transformations, [SOAP Transformations](#)

requirements for building recipe service, [How Should We Build It?](#)

SOAP request, [GetRecipes](#)

SOAP response, [External Documents](#)

XML 1.0 specification, [What are the Pieces that Make Up XML?](#)

XML-Data Reduced, see [xdr](#)

XML-Data Reduced schemas, see [xdr schemas](#)

xml-dev

XML resource, [Sharing Vocabularies](#)

XMLFilter interface, SAX, [Interface org.xml.sax.XMLFilter \(SAX 2\)](#)

XMLHTTP object

create, [Building the Client](#)

responseXML property, [Building the Client](#)

XMLHTTP object, MSXML, [How It Works](#)

xmlns attribute, namespaces, [How XML Namespaces Work](#)

default namespaces, canceling, [Canceling Default Namespaces](#)

XMLReader class, [How to Receive SAX Events](#)

XMLReader interface, SAX

replaces Parser interface, [Interface org.xml.sax.XMLReader \(SAX 2-Replaces Parser\)](#)

XMLReaderFactory class, [How to Receive SAX Events](#)

xml4j, [Parsing XML](#)

xmp tags, [XML Debugging Template](#)

XPath, [What are the Pieces that Make Up XML?](#), [Chapter 4: XSLT](#), [XPath](#)

axis names, [XPath Axis Names](#)

complex expressions, [Dealing with Complex Expressions](#)

context node, [XPath](#)

document root, [The Document Root](#)

expressions, [XPath](#)

filtering expressions and patterns, [Filtering XPath Expressions and Patterns](#)

functions, [XPath Functions](#)

location path syntax, [Appendix B: XPath Reference](#)

location paths, [XPath](#), [The Document Root](#), [Filtering XPath Expressions and Patterns](#)

node-set, [XPath](#)

pattern, [`<xsl:template>`](#)

reference, [Appendix B: XPath Reference](#)

string-length() function, [XPath Functions](#)

XPointer extensions to, [XPointer Function Extensions to XPath](#)

XPath axes, see [axes](#), [xpath](#)

using, [XPath Axis Names](#)

XPath expressions, [XPath Basics](#)

filtering, [Filtering XPath Expressions and Patterns](#)

XPath functions

all usable within XSLT stylesheets, [Inherited XPath Functions](#)

concat() function, [Modes](#)

document() function, [XML Debugging Template](#)

false() function, [`<xsl:copy-of>`](#)

full list (Appendix), [Functions](#)

implementation in XPath 1.0 and MSXML, [Functions](#)

local-name() function, [`<xs:element>`](#)

name() function, [`<xsl:element>`](#)

not() function, [`<xsl:copy-of>`](#)

parameters which can be passed by different, [Functions](#)

position() function, [Modes](#)

starts-with() function, [The \\$searchlist Variable](#)

string-length() function, [XPath Functions](#), [Error Message Node](#)

true() function, [`<xsl:copy-of>`](#)

XPath node tests, see [node-tests](#)

XPath patterns, [XPath Basics](#)

@, match any attribute, [`<xs:element>`](#)

filtering, [Filtering XPath Expressions and Patterns](#)

XPath wildcard, \*, [`<xs:element>`](#)

XPointer, [What are the Pieces that Make Up XML?](#), [XML Linking](#)

bare name syntax, [XPointer Shorthand Syntaxes](#), [Bare Names Syntax](#)

child sequence syntax, [XPointer Shorthand Syntaxes](#), [Child Sequence Syntax](#)

elements, retrieving from documents, [Full Syntax](#)

expressions, [XPointer: Pointing to Document Fragments](#)

linking, [Chapter 13: Linking and Querying XML](#)

location sets, [XPointer: Pointing to Document Fragments](#), [Locations](#), [Points and Ranges](#)

locations, [XPointer: Pointing to Document Fragments](#), [Locations](#), [Points and Ranges](#)

multiple expressions, [Using Multiple XPointer Expressions](#)

pointing to document fragments, [XPointer: Pointing to Document Fragments](#)

points, [XPointer: Pointing to Document Fragments](#), [Locations](#), [Points and Ranges](#), [Points](#)

range-to() function, [How Do We Select Ranges?](#)

ranges, [XPointer: Pointing to Document Fragments](#), [Locations](#), [Points and Ranges](#), [How Do We Select Ranges?](#), [Ranges with Multiple Locations](#)

ranges \r c11range, [Ranges](#)

retrieving a piece of a document, [Try It Out-Retrieving a Piece of a Document](#)

schemes, [XPointer Schemes](#)

shorthand syntaxes \r c11short, [XPointer Shorthand Syntaxes](#)

URIs, appending expressions to, [Appending XPointer Expressions to URIs](#)

XPath, extensions to, [XPointer Function Extensions to XPath](#)

XPointer \r c11xpoint, [XPointer: Pointing to Document Fragments](#)

XQuery, [Chapter 13: Linking and Querying XML](#), [Querying](#)

syntaxes, [XQuery Syntaxes](#)

Working draft, [XQuery Syntaxes](#)

XQueryX, [XQuery Syntaxes](#)

xs\

all element, [Appendix D: Schema Element and Attribute Reference](#)

annotation element, [Appendix D: Schema Element and Attribute Reference](#)

any element, [Appendix D: Schema Element and Attribute Reference](#)

anyAttribute element, [Appendix D: Schema Element and Attribute Reference](#)

appInfo element, [Appendix D: Schema Element and Attribute Reference](#)

attribute element, [Appendix D: Schema Element and Attribute Reference](#)

attributeGroup element, [Appendix D: Schema Element and Attribute Reference](#)

choice element, [Appendix D: Schema Element and Attribute Reference](#)

complexContent element, [Appendix D: Schema Element and Attribute Reference](#)

complexType element, [Appendix D: Schema Element and Attribute Reference](#)  
documentation element, [Appendix D: Schema Element and Attribute Reference](#)  
element element, [Appendix D: Schema Element and Attribute Reference](#)  
extension element, [Appendix D: Schema Element and Attribute Reference](#)  
field element, [Appendix D: Schema Element and Attribute Reference](#)  
group element, [Appendix D: Schema Element and Attribute Reference](#)  
import element, [Appendix D: Schema Element and Attribute Reference](#)  
include element, [Appendix D: Schema Element and Attribute Reference](#)  
key element, [Appendix D: Schema Element and Attribute Reference](#)  
keyref element, [Appendix D: Schema Element and Attribute Reference](#)  
list element, [Appendix D: Schema Element and Attribute Reference](#)  
notation element, [Appendix D: Schema Element and Attribute Reference](#)  
redefine element, [Appendix D: Schema Element and Attribute Reference](#)  
restriction element, [Appendix D: Schema Element and Attribute Reference](#)  
schema element, [Appendix D: Schema Element and Attribute Reference](#)  
selector element, [Appendix D: Schema Element and Attribute Reference](#)  
sequence element, [Appendix D: Schema Element and Attribute Reference](#)  
simpleContent element, [Appendix D: Schema Element and Attribute Reference](#)  
simpleType element, [Appendix D: Schema Element and Attribute Reference](#)  
union element, [Appendix D: Schema Element and Attribute Reference](#)  
unique element, [Appendix D: Schema Element and Attribute Reference](#)

XSD, [Chapter 6: XML Schemas](#)

xsl\

nil attribute, [XML Schema Instance Attributes](#)

noNameSpaceSchemaLocation attribute, [XML Schema Instance Attributes](#)

schemaLocation attribute, [`<group>`](#), [XML Schema Instance Attributes](#)

type attribute, [XML Schema Instance Attributes](#)

XSL, [What is XSL?](#)

result tree, [What is XSL?](#)

source tree, [What is XSL?](#)

XML, displaying, [What are the Pieces that Make Up XML?](#)

XSL Formatting Objects, [What is XSL?](#)

XSL stylesheets

address book stylesheet design, [Address Book Stylesheet Design](#)

associating with XML document using processing instructions, [Associating Stylesheets with XML Documents Using Processing Instructions](#)

must be well-formed, [What is XSL?](#)

possible uses, [Using XML in an XSLT-Driven Application](#)

XSL transformations

Oracle, [Putting Oracle to Work](#)

XSL-FO

often used in print production, [Working with Complementary Stylesheet Languages: CSS and XSLT](#)

xsl\

apply-imports> element, [Elements](#)

apply-templates> element, [`<xsl:apply-templates>`](#), [`<xsl:apply-templates>`](#), [`<xsl:apply-templates>`](#),  
[`<xsl:apply-templates>`](#), [`<xsl:apply-templates>`](#), [`<xsl:element>`](#), [`<xsl:element>`](#), [Default Templates](#), [Modes](#),  
[`<xsl:apply-templates>`](#)

attribute-set> element, [`<xsl:attribute>`](#) and [`<xsl:attribute-set>`](#), [Related Groups of Attributes](#), [Try It Out-Attributes and Attribute Sets in Action](#), [`<xsl:attribute-set>`](#)

attribute> element, [`<xsl:attribute>`](#) and [`<xsl:attribute-set>`](#), [`<xsl:attribute>`](#) and [`<xsl:attribute-set>`](#), [`<xsl:attribute>`](#) and [`<xsl:attribute-set>`](#), [Try It Out-Attributes and Attribute Sets in Action](#), [`<xsl:attribute>`](#)

call-template element, [User Interface Components](#), [Form Template](#)

call-template> element, [Variables, Constants, and Named Templates](#), [`<xsl:call-template>`](#)

choose element, [Suppress List and Empty Results Message](#), [Details Display Template](#), [XML Debugging Template](#)

choose> element, [Conditional Processing with `<xsl:if>`](#) and [`<xsl:choose>`](#), [Conditional Processing with `<xsl:if>` and](#)

<xsl:choose>, <xsl:choose>  
comment> element, <xsl:comment>  
copy-of element, [How It Works](#), [XML Debugging Template](#)  
copy-of> element, [Copying Sections of the Source Tree to the Result Tree](#), <xsl:copy-of>, <xsl:copy-of>,  
<xsl:copy-of>, <xsl:copy-of>  
copy> element, <xsl:copy-of>, <xsl:copy>, <xsl:copy>, <xsl:copy>  
decimal-format> element, <xsl:decimal-format>  
document element, [Suggestions for Enhancement of the Application](#)  
document> element, <xsl:document>  
element element, [recipes.xsl](#)  
element> element, <xselement>, <xselement>, <xselement>, <xselement>  
fallback> element, <xsl:fallback>  
for-each element, [Item Row Display](#), [Try It Out - Sorting the List](#), [Building the Client](#)  
for-each elements, [Row Iteration](#)  
for-each> element, [Declarative Programming](#), <xsl:for-each>, <xsl:for-each>, <xsl:for-each>, <xsl:copy-of>,  
<xsl:for-each>  
if> element, [Conditional Processing with <xsl:if> and <xsl:choose>](#), [Conditional Processing with <xsl:if> and <xsl:choose>](#), <xsl:if>  
import element, [Suggestions for Enhancement of the Application](#)  
import> element, <xsl:import>  
include> element, <xsl:include>  
key> element, <xsl:key>  
message> element, <xsl:message>  
namespace-alias> element, <xsl:namespace-alias>  
number> element, <xsl:number>  
otherwise> element, [Conditional Processing with <xsl:if> and <xsl:choose>](#), <xsl:otherwise>  
output element, [recipes.xsl](#)  
output> element, <xsl:apply-templates>, <xsl:output>, <xsl:output>, <xsl:output>, <xsl:output>, <xsl:output>,  
<xsl:output>, <xsl:output>, <xsl:output>, <xsl:output>  
param> element, [Parameters](#), <xsl:param>  
preserve-space> element, <xsl:preserve-space>  
processing-instruction> element, <xsl:processing-instruction>  
sort element, [Try It Out - Sorting the List](#)  
sort> element, [Declarative Programming](#), [Sorting the Result Tree](#), <xsl:sort>  
strip-space> element, <xsl:strip-space>  
stylesheet element, [recipes.xsl](#)  
stylesheet> element, <xsl\_stylesheet>, <xsl\_stylesheet>, <xsl\_stylesheet>, <xsl:apply-templates>, <xsl\_stylesheet>  
template match element, [User Interface Components](#), [Try It Out - Sorting the List](#)  
template> element, [How XSLT Stylesheets Work-Templates](#), [XPath Basics](#), <xsl:template>, <xsl:template>,  
<xsl:template>, [Template Priority](#), <xsl:apply-templates>, <xsl:copy>, [Modes](#), [Variables](#), [Constants](#), [Named Templates](#), <xsl:template>  
compared to <xsl\>, <xsl:for-each>  
text element, [Item Row Display](#), [XML Debugging Template](#)  
text> element, <xsl:text>, <xsl:text>, <xsl:text>, <xsl:text>  
transform> element, <xsl\_stylesheet>, <xsl:transform>  
value-of element, [Item Row Display](#)  
value-of> element, [How XSLT Stylesheets Work-Templates](#), [Declarative Programming](#), [XPath Basics](#),  
<xsl\_stylesheet>, [Getting Information from the Source Tree with <xsl:value-of>](#), [Getting Information from the Source Tree with <xsl:value-of>](#), [Getting Information from the Source Tree with <xsl:value-of>](#), <xsl:text>, [Default Templates](#),  
<xsl:for-each>, <xsl:value-of>  
variable element, [The \\$params Variable](#)  
variable> element, [Variables](#), [Constants](#), [Named Templates](#), <xsl:variable>  
version attribute, <xsl\_stylesheet>  
when element, [XML Debugging Template](#)  
when> element, [Conditional Processing with <xsl:if> and <xsl:choose>](#), <xsl:when>

with-param> element, [Parameters](#), <xsl:with-param>  
XSLT, [Chapter 4: XSLT, Who Needs a Database Anyway?](#)  
attribute namespaces, [Namespaces and Attributes](#)  
attributes, dynamically adding, <xsl:attribute> and <xsl:attribute-set>  
business to business example, [Why is XSLT So Important for e-Commerce?](#)  
conditional processing, [Conditional Processing with <xsl:if> and <xsl:choose>](#), [Conditional Processing with <xsl:if> and <xsl:choose>](#), [Advantages of Using XSL to Build Interactive Web Applications](#)  
constants, [Variables, Constants, and Named Templates](#)  
declarative language, [Declarative Programming](#)  
default templates, [Default Templates](#)  
e-commerce, importance in, [Why is XSLT So Important for e-Commerce?](#)  
elements, [Appendix C: XSLT Reference](#)  
elements, containing as variables, [Variables, Constants, and Named Templates](#)  
elements, dynamically adding, <xsl:element>  
for use when mapping XML to relational structure, [Mapping XML to a Relational Structure](#)  
formatting output, <xsl:output>  
functions can be used in expressions, [document\(\) function](#)  
modes, [Modes](#)  
named templates, [Variables, Constants, and Named Templates](#)  
output, specifying, <xsl:output>  
parameters, [Parameters](#)  
recursive templates, <xsl:copy>  
self-standing, [What is XSL?](#)  
side effects, lack of, [XSLT Has No Side Effects](#)  
sorting, [Sorting the Result Tree](#)  
special functions, (Appendix), [Functions](#)  
stylesheet creation, <xsl:apply-templates>  
templates, [How Do Templates Affect the Context Node?](#)  
text, inserting into result tree, <xsl:text>  
top-level elements, <xsl:output>  
transforming XML into HTML, [Who Needs a Database Anyway?](#)  
variables, [Variables, Constants, and Named Templates](#)  
XML, displaying, [What are the Pieces that Make Up XML?](#)  
XSLT elements  
<xsl\|, [How XSLT Stylesheets Work- Templates](#), [Declarative Programming](#), [XPath Basics](#), <xsl:stylesheet>,  
<xsl:stylesheet>, <xsl:template>, <xsl:apply-templates>, <xsl:apply-templates>, <xsl:apply-templates>,  
<xsl:apply-templates>, <xsl:apply-templates>, [Getting Information from the Source Tree with <xsl:value-of>](#),  
<xsl:output>, <xsl:output>, <xsl:output>, <xsl:element>, <xsl:element>, <xsl:element>, <xsl:attribute>  
and <xsl:attribute-set>, <xsl:attribute> and <xsl:attribute-set>, <xsl:text>, [Default Templates](#), [Conditional Processing with <xsl:if> and <xsl:choose>](#), <xsl:for-each>, <xsl:for-each>,  
<xsl:for-each>, [Copying Sections of the Source Tree to the Result Tree](#), <xsl:copy-of>, <xsl:copy-of>,  
<xsl:copy-of>, <xsl:copy>, <xsl:copy>, [Sorting the Result Tree](#), [Modes](#), [Variables, Constants, and Named Templates](#), [Parameters](#), [Elements](#), <xsl:apply-templates>, <xsl:attribute>, <xsl:attribute-set>, <xsl:call-template>,  
<xsl:choose>, <xsl:comment>, <xsl:copy>, <xsl:copy-of>, <xsl:decimal-format>, <xsl:document>, <xsl:element>,  
<xsl:fallback>, <xsl:for-each>, <xsl:if>, <xsl:import>, <xsl:include>, <xsl:key>, <xsl:message>,  
<xsl:namespace-alias>, <xsl:number>, <xsl:otherwise>, <xsl:output>, <xsl:param>, <xsl:preserve-space>,  
<xsl:processing-instruction>, <xsl:sort>, <xsl:strip-space>, <xsl:stylesheet>, <xsl:template>, <xsl:text>,  
<xsl:transform>, <xsl:value-of>, <xsl:variable>, <xsl:when>, <xsl:with-param>  
xsl\|, [User Interface Components](#), [The \\$params Variable](#), [Form Template](#), [Error Message Node](#), [Suppress List and Empty Results Message](#), [Row Iteration](#), [Item Row Display](#), [Try It Out - Sorting the List](#), [How It Works](#), [Details](#)  
[Display Template](#), [XML Debugging Template](#), [Suggestions for Enhancement of the Application](#), [recipes.xsl](#), [Building the Client](#)  
XSLT engines, [Running the Examples](#)

declarative programming, [Declarative Programming](#)

XSLT functions

document() function, [document\(\) function](#)

XSLT stylesheets, [Running the Examples](#)

how they work, [How XSLT Stylesheets Work-Templates](#)

XSLT templates

user interface components, [User Interface Components](#)

XSLT-driven web applications

possible business uses, [Possible Business Uses for XSLT-Driven Web Applications](#)

using XML, [Using XML in an XSLT-Driven Application](#)

XML encodes the user interface input, [Using XML in an XSLT-Driven Application](#)

XSQL Servlet, Java, [XSQL Servlet](#)

---

[!\[\]\(bc7439098852759fdfbb8ee74617650b\_img.jpg\) PREVIOUS](#)

[\*< Free Open Study >\*](#)