XML IN A NUTSHELL

A Desktop Quick Reference

Elliotte Rusty Harold & W. Scott Means

O'REILLY®

XML in a Nutshell

by Elliotte Rusty Harold and W. Scott Means

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Contributor: Stephen Spainhour

Editors: Laurie Petrycki and John Posner

Production Editor: Ann Schirmer

Cover Designer: Ellie Volckhausen

Printing History:

January 2001: First Edition.

Library of Congress Cataloging-in-Publication Data can be found at: http://www.oreilly.com/catalog/xmlnut.

ISBN: 0-596-00058-8
[M] [8/01]

Table of Contents

Preface

XML is one of the most important developments in document syntax in the history of computing. In the last few years it has been adopted in fields as diverse as law, aeronautics, finance, insurance, robotics, multimedia, hospitality, travel, art, construction, telecommunications, software design, agriculture, physics, journalism, theology, retail, and medieval literature. XML has become the syntax of choice for newly designed document formats across almost all computer applications. It's used on Linux, Windows, Macintosh, and many other computer platforms. Mainframes on Wall Street trade stocks with one another by exchanging XML documents. Children playing games on their home PCs save their documents in XML. Sports fans receive real-time game scores on their cell phones in XML. XML is simply the most robust, reliable, and flexible document syntax ever invented.

XML in a Nutshell is a comprehensive guide to the rapidly growing world of XML. It covers all aspects of XML, from the most basic syntax rules, to the details of DTD creation, to the APIs you can use to read and write XML documents in a variety of programming languages.

What This Book Covers

There are hundreds of formally established XML applications from the W3C and other standards bodies, such as OASIS and the Object Management Group. There are even more informal, unstandardized applications from individuals and corporations, such as Microsoft's Channel Definition Format and John Guajardo's Mind Reading Markup Language. This book cannot cover them all, any more than a book on Java could discuss every program that has ever been or might ever be written in Java. This book focuses primarily on XML itself. It covers the fundamental rules that all XML documents and authors must adhere to, whether a web designer uses SMIL to add animations to web pages or a C++ programmer uses SOAP to serialize objects into a remote database.

This book also covers generic supporting technologies that have been layered on top of XML and are used across a wide range of XML applications. These technologies include:

XLinks
: An attribute-based syntax for hyperlinks between XML and non-XML documents that provide the simple, one-directional links familiar from HTML, multidirectional links between many documents, and links between documents you don't have write access to.

XSLT
: An XML application that describes transformations from one document to another, in either the same or different XML vocabularies.

XPointers
: A syntax for identifying particular parts of an XML document referred to by a URI; often used in conjunction with an XLink.

XPath
: A non-XML syntax used by both XPointers and XSLT for identifying particular pieces of XML documents. For example, an XPath can locate the third `address` element in the document, or all elements with an `email` attribute whose value is `elharo@metalab.unc.edu`.

Namespaces
: A means of distinguishing between elements and attributes from different XML vocabularies that have the same name; for instance, the title of a book and the title of a web page in a web page about books.

SAX
: The Simple API for XML, an event-based Java application programming interface implemented by many XML parsers.

DOM
: The Document Object Model, a tree-oriented API that treats an XML document as a set of nested objects with various properties.

All these technologies, whether defined in XML (XLinks, XSLT, and Namespaces) or in another syntax (XPointers, XPath, SAX, and DOM), are used in many different XML applications.

This book does not specifically cover XML applications that are relevant to only some users of XML. These include:

SVG
: Scalable Vector Graphics is a W3C-endorsed standard used for encoding line drawings in XML.

MathML
: The Mathematical Markup Language is a W3C-endorsed standard XML application used for embedding equations in web pages and other documents.

CML

The Chemical Markup Language was one of the first XML applications. It describes chemistry, solid-state physics, molecular biology, and the other molecular sciences.

RDF

The Resource Description Framework is a W3C-standard XML application used for describing resources, with a particular focus on the sort of metadata one might find in a library card catalog.

CDF

The Channel Definition Framework is a nonstandard, Microsoft-defined XML application used to publish web sites to Internet Explorer for offline browsing.

Occasionally we use one or more of these applications in an example, but we do not cover all aspects of the relevant vocabulary in depth. While interesting and important, these applications (and hundreds more like them) are intended primarily for use with special software that knows their format intimately. For instance, graphic designers do not work directly with SVG. Instead, they use their customary tools, such as Adobe Illustrator, to create SVG documents. They may not even know they're using XML.

This book focuses on standards that are relevant to almost all developers working with XML. We investigate XML technologies that span a wide range of XML applications, not those that are relevant only within a few restricted domains.

Organization of the Book

Part I, *XML Concepts*, introduces you to the fundamental standards that form the essential core that all XML applications and software must adhere to. It teaches you about well-formed XML, DTDs, namespaces, and Unicode as quickly as possible.

Part II, *Narrative-Centric Documents*, explores technologies that are used mostly for narrative XML documents, such as web pages, books, articles, diaries, and plays. You'll learn about XSLT, CSS, XSL-FO, XLinks, XPointers, and XPath.

One of the most unexpected developments in XML was its enthusiastic adoption of data-heavy structured documents such as spreadsheets, financial statistics, mathematical tables, and software file formats. Part III, *Data-Centric XML*, explores the use of XML for such data-intensive documents. This part focuses on the tools and APIs needed to write software that process XML, including SAX, the Simple API for XML, and the W3C's Document Object Model.

Finally, Part IV, *Reference*, is a series of quick-reference chapters that form the core of any Nutshell handbook. These chapters give you detailed syntax rules for the core XML technologies, including XML, DTDs, XPath, XSLT, SAX, and DOM. Turn to this section when you need to quickly find out the precise syntax for something you know you can do but don't remember exactly how to do.

Conventions Used in This Book

Body text, like the text you're reading now, is written in Garamond.

`Constant width` is used for:

- Code examples and fragments.
- Anything that might appear in an XML document, including element names, tags, attribute values, entity references, and processing instructions.
- Anything that might appear in a program, including keywords, operators, method names, class names, and literals.

`Constant-width bold`

- User input.
- Signifies emphasis should be deleted.

`Constant-width italic` is used for:

- Replaceable elements in code statements.

Italic is used for:

- New terms where they are defined.
- Pathnames, filenames, and program names. (However, if the program name is also the name of a Java class, it is written in constant-width font, like other class names.)
- Host and domain names (*www.xml.com*).
- URLs (*http://ibiblio.org/xml/*).

Significant code fragments, complete programs, and documents are generally placed into a separate paragraph like this:

```
<?xml version="1.0"?>
<?xml-stylesheet href="person.css" type="text/css"?>
<person>
  Alan Turing
</person>
```

XML is case sensitive. The `PERSON` element is not the same thing as the `person` or `Person` element. Case-sensitive languages do not always allow authors to adhere to standard English grammar. It is usually possible to rewrite the sentence so the two do not conflict, and when possible we have endeavored to do so. However, on rare occasions when there is simply no way around the problem, we let standard English come up the loser.

Finally, although most of the examples used here are toy examples unlikely to be reused, a few have real value. Please feel free to reuse them or any parts of them in your own code. No special permission is required. As far as we are concerned, they are in the public domain (though the same is definitely not true of the explanatory text).

Request for Comments

We enjoy hearing from readers with general comments about how this book could be better, specific corrections, or topics you would like to see covered. You can reach the authors by sending email to *elharo@metalab.unc.edu* and *smeans@enterprisewebmachines.com*. Please realize, however, that we each receive several hundred pieces of email a day and cannot respond to every one personally. For the best chances of getting a personal response, please identify yourself as a reader of this book. And please send the message from the account you want us to reply to and make sure that your Reply-to address is properly set. There's nothing quite so frustrating as spending an hour or more carefully researching the answer to an interesting question and composing a detailed response, only to have it bounce because the correspondent sent the message from a public terminal and neglected to set the browser preferences to include their actual email address.

The information in this book has been tested and verified, but you may find that features have changed (or you may even find mistakes). We believe the old saying, "If you like this book, tell your friends. If you don't like it, tell us." We're especially interested in hearing about mistakes. As hard as the authors and editors worked on this book, inevitably there are a few mistakes and typographical errors that slipped by us. If you find a mistake or a typo, please let us know so we can correct it. You can send any errors you find, as well as suggestions for future editions, to:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
1-800-998-9938 (in the United States or Canada)
1-707-829-0515 (international/local)
1-707-829-0104 (fax)

We have a web site for the book, where we list errata, examples, and any additional information. You can access this site at:

http://www.oreilly.com/catalog/xmlnut

Before reporting errors, please check this web site to see if we already posted a fix. To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

Many people were involved in the production of this book. The original editor, John Posner, got this book rolling and provided many helpful comments that substantially improved the book. When John moved on, Laurie Petrycki shepherded this book to its completion. Stephen Spainhour deserves special thanks for

his work on the reference section. His efforts in organizing and reviewing material helped create a better book. We'd like to thank Matt Sergeant and Didier P. H. Martin for their thorough technical review of the manuscript and thoughtful suggestions.

We'd also like to thank everyone who has worked so hard to make XML such a success over the last few years and thereby give us something to write about. There are so many of these people that we can only list a few. In alphabetical order we'd like to thank Tim Berners-Lee, Jon Bosak, Tim Bray, James Clark, Charles Goldfarb, Jason Hunter, Michael Kay, Brett McLaughlin, David Megginson, David Orchard, Walter E. Perry, Simon St. Laurent, C. M. Sperberg-McQueen, James Tauber, B. Tommie Usdin, and Mark Wutka. Our apologies to everyone we unintentionally omitted.

Elliotte would like to thank his agent, David Rogelberg, who convinced him that it was possible to make a living writing books like this rather than working in an office. The entire Sunsite crew (now *ibiblio.org*) has also helped him to communicate better with his readers in a variety of ways over the last several years. All these people deserve much thanks and credit. Finally, as always, he offers his largest thanks to his wife, Beth, without whose love and support this book would never have happened.

Scott would most like to thank his lovely wife, Celia, who has already spent way too much time as a "computer widow." He would also like to thank his daughter Selene for understanding why Daddy can't play with her when he's "working," and Skyler for just being himself. Also, he'd like to thank the team at Enterprise Web Machines for helping him make time to write. Finally, he would like to thank John Posner for getting him into this and Laurie Petrycki for working with him when things got tough.

Elliotte Rusty Harold
elharo@metalab.unc.edu

W. Scott Means
smeans@enterprisewebmachines.com

CHAPTER 1

Introducing XML

XML, the Extensible Markup Language, is a W3C-endorsed standard for document markup. It defines a generic syntax used to mark up data with simple, human-readable tags. It provides a standard format for computer documents. This format is flexible enough to be customized for domains as diverse as web sites, electronic data interchange, vector graphics, genealogy, real estate listings, object serialization, remote procedure calls, and voice mail systems.

You can write your own programs that interact with, massage, and manipulate data in XML documents. If you do, you'll have access to a wide range of free libraries in a variety of languages that can read and write XML so that you can focus on the unique needs of your program. Or you can use off-the-shelf software like web browsers and text editors to work with XML documents. Some tools are able to work with any XML document. Others are customized to support a particular XML application in a particular domain like vector graphics and may not be of much use outside that domain. But in all cases, the same underlying syntax is used, even if it's deliberately hidden by more user-friendly tools or restricted to a single application.

What XML Offers

XML is a meta-markup language for text documents. Data is included in XML documents as strings of text, and the data is surrounded by text markup that describes the data. A particular unit of data and markup is called an *element*. The XML specification defines the exact syntax this markup must follow: how elements are delimited by tags, what a tag looks like, what names are acceptable for elements, where attributes are placed, and so forth. Superficially, the markup in an XML document looks much like that in an HTML document, but some crucial differences exist.

Most importantly, XML is a *meta-markup language*. That means it doesn't have a fixed set of tags and elements that are always supposed to work for everyone in

all areas of interest. Attempts to create a finite set of such tags are doomed to failure. Instead, XML allows developers and writers to define the elements they need as they need them. Chemists can use tags that describe elements, atoms, bonds, reactions, and other items encountered in chemistry. Real estate agents can use elements that describe apartments, rents, commissions, locations, and other items needed in real estate. Musicians can use elements that describe quarter notes, half notes, G clefs, lyrics, and other objects common in music. The *X* in XML stands for *Extensible*. Extensible means that the language can be extended and adapted to meet many different needs.

Although XML is flexible in the elements it allows to be defined, it is strict in many other respects. It provides a grammar for XML documents that regulates placement of tags, where tags appear, which element names are legal, how attributes are attached to elements, and so forth. This grammar is specific enough to allow development of XML parsers that can read and understand any XML document. Documents that satisfy this grammar are said to be *well-formed*. Documents that are not well-formed are not allowed any more than a C program containing a syntax error would be. XML processors reject documents that contain well-formedness errors.

To enhance interoperability, individuals or organizations may agree to use only certain tags. These tag sets are called *XML applications*. An XML application is not a software application that uses XML, like Mozilla or Microsoft Word. Rather, it's an application of XML to a particular domain, such as vector graphics or cooking.

The markup in an XML document describes the document's structure. It lets you see which elements are associated with which other elements. In a well-designed XML document, the markup also describes the document's semantics. For instance, the markup can indicate that an element is a date, a person, or a bar code. In well-designed XML applications, the markup says nothing about how the document should be displayed. That is, it does not say that an element is bold, italicized, or a list item. XML is a structural and semantic markup language, not a presentation language.*

The markup permitted in a particular XML application can be documented in a document type definition (DTD). The DTD lists all legal markup and specifies where and how the markup may be included in a document. Particular document instances can be compared to the DTD. Documents that match the DTD are said to be *valid*. Documents that do not match are *invalid*. Validity depends on the DTD; whether a document is valid or invalid depends on which DTD you compare it to.

Not all documents need to be valid. For many purposes, a well-formed document is enough. DTDs are optional in XML. On the other hand, DTDs may not always be sufficient. The DTD syntax is limited and does not allow you to make many

* A few XML applications, like XSL Formatting Objects, are designed to describe text presentation. However, these are exceptions that prove the rule. Although XSL-FO describes presentation, you'd never write an XSL-FO document directly. Instead, you'd write a more semantically marked-up XML document, then use an XSL Transformations stylesheet to change the semantic-oriented XML into presentation-oriented XML.

useful statements such as, "This element contains a number" or "This string of text is a date between 1974 and 2032." If you're writing programs to read XML documents, you may want to add code to verify statements like these, just as you would if you were writing code to read a tab-delimited text file. The difference is that XML parsers present you with the data in a much more convenient format to work with and do more of the work for you before you have to resort to your own custom code.

What XML Is Not

XML is a markup language, and only a markup language. It's important to remember this fact. The XML hype has become so extreme that some people expect XML to do everything up to, and including, washing the family dog.

First of all, XML is not a programming language. There's no such thing as an XML compiler that reads XML files and produces executable code. You might define a scripting language that uses a native XML format and is interpreted by a binary program, but even this application would be unusual.* XML can be used as an instruction format for programs that make things happen. A traditional program, for example, may read a text config file and take different actions, depending on what it sees in the file. There's no reason why a config file can't be written in XML instead of unstructured text. Indeed, some recent programs are beginning to use XML config files. But in all cases the program, not the XML document, takes action. An XML document simply *is*. It does not *do* anything.

Furthermore, XML is not a network-transport protocol. XML, like HTML, won't send data across the network. However, data sent across the network using HTTP, FTP, NFS, or some other protocol might be in an XML format. XML can be the format for data sent across the network, but again, software outside the XML document must actually do the sending.

Finally, to mention the example in which the hype most often obscures reality, XML is not a database. You won't replace an Oracle or MySQL server with XML. A database can contain XML data as a VARCHAR, a BLOB, or a custom XML datatype; but the database itself is not an XML document. You can store XML data in a database or on a server or retrieve data from a database in an XML format, but to do so you need to run software written in a real programming language like C or Java. To store XML in a database, software on the client side sends the XML data to the server using an established network protocol like TCP/IP. Software on the server side receives the XML data, parses it, and stores it in the database. To retrieve an XML document from a database, you generally pass through a middleware product like Enhydra that makes SQL queries against the database and formats the result set as XML before returning it to the client. Indeed, some databases may integrate this software code into their core server or provide plug-ins, such as the Oracle XSQL servlet, to do it. XML serves very well as a ubiquitous, platform-independent transport format in these scenarios. However, XML is not the database and shouldn't be used as one.

* At least one XML application, XSL Transformations, has been proven to be Turing complete.

Portable Data

XML offers the tantalizing possibility of truly cross-platform, long-term data formats. It's long been the case that a document written by one piece of software on one platform is not necessarily readable on a different platform, by a different program on the same platform, or even by a future or past version of the same software on the same platform. When the document can be read, all the information may not necessarily come across. Much of the data from the original moon landings in the late 1960s and early 1970s is now effectively lost. Even if you can find a tape drive that reads the now obsolete tapes, nobody knows what format the data on the tapes is stored in!

XML is an incredibly simple, well-documented, straightforward data format. XML documents are text, and any tool that can read a text file can read an XML document. Both XML data and markup are text, and the markup is present in the XML file as tags. You don't have to wonder whether every eighth byte is random padding, guess whether a four-byte quantity is a two's complement integer or an IEEE 754 floating point number, or try to decipher which integer codes map to which formatting properties. You can read the tag names directly to see exactly what's in the document. Similarly, since tags define element boundaries, you aren't likely to get tripped up by unexpected line ending conventions or the number of spaces mapped to a tab. All the important details about the document's structure are explicit. You don't have to reverse engineer the format or rely on questionable, and often unavailable, documentation.

A few software vendors may want to lock in their users with undocumented, proprietary binary file formats. However, in the long run we are all better off if we can use the cleanly documented, well-understood, easy to parse, text-based formats that XML provides. XML allows documents and data to move from one system to another with a reasonable hope that the receiving system can make sense out of it. Furthermore, validation lets the receiving side ensure that it gets what it expects. Java promised portable code. XML delivers portable data. In many ways, XML is the most portable and flexible document format designed since the ASCII text file.

How XML Works

Example 1-1 shows a simple XML document. This particular XML document might appear in an inventory control system or a stock database. It marks up the data with tags and attributes describing the color, size, bar code number, manufacturer, and product name.

Example 1-1: An XML Document

```
<?xml version="1.0"?>
<product barcode="2394287410">
  <manufacturer>Verbatim</manufacturer>
  <name>DataLife MF 2HD</name>
  <quantity>10</quantity>
  <size>3.5"</size>
```

Example 1-1: An XML Document (continued)

```
  <color>black</color>
  <description>floppy disks</description>
</product>
```

This document is text and might well be stored in a text file. You can edit this file with any standard text editor, such as BBEdit, UltraEdit, Emacs, or vi. You do not need a special XML editor; in fact, we find that most general-purpose XML editors are far more trouble than they're worth and much harder to use than a simple text editor.

Then again, this document might not be a file at all. It might be a record in a database. It might be assembled on the fly by a CGI query to a web server and exist only in a computer's memory. It might even be stored in multiple files and assembled at runtime. Even if it isn't in a file, however, the document is a text document that can be read and transmitted by any software capable of reading and transmitting text.

Programs that actually try to understand the contents of the XML document, that is, do not merely treat it as any other text file, use an *XML parser* to read the document. The parser is responsible for dividing the document into individual elements, attributes, and other pieces. It passes the contents of the XML document to the application piece by piece. If at any point the parser detects a violation of XML rules, it reports the error to the application and stops parsing. In some cases the parser may read past the original error in the document so it can detect and report other errors that occur later in the document. However, once it has detected the first error, it no longer passes along the contents of the elements and attributes it encounters to the application.

Individual XML applications normally dictate precise rules about which elements and attributes are allowed where. You wouldn't expect to find a `G_Clef` element when reading a biology document, for instance. Some of these rules can be specified precisely using a DTD. A document may contain either the DTD itself or a pointer to a URI where the DTD is found. Some XML parsers notice these details and compare the document to its DTD as they read it to see if the document satisfies the specified constraints. Such a parser is called a *validating parser*. A violation of those constraints is a *validity error*, and the whole process of checking a document against a DTD is called *validation*. If a validating parser finds a validity error, it reports it to the application on whose behalf it parses the document. This application can then decide whether it wishes to continue parsing the document. However, validity errors, unlike well-formedness errors, are not necessarily fatal; an application may choose to ignore them. Not all parsers are validating parsers. Some merely check for well-formedness.

The application that receives data from the parser may be:

- A web browser, such as Netscape or Internet Explorer, that displays the document to a reader
- A word processor, such as StarOffice Writer, that loads the XML document for editing
- A database server, such as Oracle, that stores XML data in a database

- A drawing program, such as Corel Draw, that interprets XML as two-dimensional coordinates for the contents of a picture
- A spreadsheet, such as Gnumeric, that parses XML to find numbers and functions used in a calculation
- A personal finance program, such as Microsoft Money, that sees XML as a bank statement
- A syndication program that reads the XML document and extracts the headlines for today's news
- A program that you wrote in Java, C, Python, or some other language that does exactly what you want it to do
- Almost anything else

XML is an *extremely* flexible format for data. It can be used in all of these scenarios and many more. These examples are real. In theory, any data that can be stored in a computer can be stored in XML format. In practice, XML is suitable for storing and exchanging any data that can be plausibly encoded as text. Its use is unsuitable only for multimedia data, such as photographs, recorded sound, video, and other very large bit sequences.

The Evolution of XML

XML is a descendant of the Standard Generalized Markup Language (SGML). The language that would eventually become SGML was invented by Charles Goldfarb, Ed Mosher, and Ray Lorie at IBM in the 1970s and developed by several hundred people around the world until its eventual adoption as ISO standard 8879 in 1986. SGML was intended to solve many of the same problems XML solves. It was and is a semantic and structural markup language for text documents. SGML is extremely powerful and achieved some success in the U.S. military and government, the aerospace sector, and other domains that needed ways of managing technical documents that were tens of thousands of pages long efficiently.

SGML's biggest success was HTML, an SGML application. However, HTML is just one SGML application. It does not have or offer the full power of SGML. Since HTML restricts authors to a finite set of tags designed to describe web pages in a fairly presentationally oriented way, it's really little more than a traditional markup language that has been adopted by web browsers. It simply doesn't lend itself to use beyond the single application of web page design. You would not use HTML to exchange data between incompatible databases or send updated product catalogs to retailer sites, for example. HTML is useful for creating web pages, but it isn't capable of much more than that.

The obvious choice for other applications that took advantage of the Internet, but were not simple web pages, was SGML. SGML's main problem is its complexity. The official SGML specification is more than 150 very technical pages. It covers many special cases and unlikely scenarios. It is so complex that almost no software has ever implemented it fully. Programs that implemented or relied on different subsets of SGML were often incompatible with one another. The special feature one program considered essential would be considered extraneous fluff and omitted by the next program.

In 1996 Jon Bosak, Tim Bray, C. M. Sperberg-McQueen, James Clark, and several others began work on a "lite" version of SGML. This version retained most of SGML's power, but trimmed many features that were redundant, too complicated to implement, confusing to end users, or that had simply not been proven useful over the previous 20 years of experience with SGML. The result, in February 1998, was XML 1.0, and it was an immediate success. Many developers who knew they needed a structural markup language but couldn't bring themselves to accept SGML's complexity adopted XML wholeheartedly. It was ultimately used in domains ranging from legal court filings to hog farming.

However, XML 1.0 was just the beginning. The next standard out of the gate was Namespaces in XML, an effort to allow conflict-free use of markup from different XML applications in the same document. A web page about books, for example, could have a `title` element that referred to the page's title and `title` elements that referred to the book's title, and the two would not conflict.

The Extensible Stylesheet Language, an XML application that transforms other XML documents into a form that is viewable in web browsers, was the next development. This language soon split into XSL Transformations (XSLT) and XSL Formatting Objects (XSL-FO). XSLT has become a general-purpose language for transforming one XML document into another for web page display and other purposes. XSL-FO is an XML application that describes the layout of both printed and web pages. This application rivals PostScript for its power and expressiveness.

However, XSL is not the only option for styling XML documents. The Cascading Stylesheet Language (CSS) was already in use for HTML documents when XML was invented, and it was a reasonable fit to XML, as well. With the advent of CSS Level 2, the W3C made styling XML documents an explicit goal for CSS and gave it equal importance to HTML. The preexisting Document Style Sheet and Semantics Language (DSSSL) was also adopted from its roots in the SGML world to style XML documents for print and use on the Web.

The Extensible Linking Language (XLL) defined more powerful linking constructs that could connect XML documents in a hypertext network, vastly overpowering HTML's `A` tag. It also divided into two separate standards: XLink, which described connections between documents, and XPointer, which addressed the individual parts of an XML document. At this point, it was noticed that both XPointer and XSLT were developing fairly sophisticated, yet incompatible, syntaxes to do exactly the same thing: identify particular elements of an XML document. Consequently, the addressing parts of both specifications were split off and combined into a third specification, XPath.

A similar phenomenon occurred when it was noticed that XML 1.0, XSLT, XML Schemas, and the Document Object Model (DOM) all had similar, but subtly different, conceptual models of the structure of an XML document. For instance, XML 1.0 considers a document's root element as its root, while XSLT uses a more abstract root that includes the root element and several other pieces. Thus the W3C XML Core Working Group began work on an XML Information Set that all these standards could rely on and refer to.

Another piece of the puzzle was a uniform interface for accessing the contents of the XML document from inside a Java, JavaScript, or C++ program. The simplest

API was to merely treat the document as an object that contained other objects. Indeed, work was already underway inside and outside the W3C to define such a Document Object Model for HTML. Expanding this effort to cover XML was not difficult.

Outside the W3C, Peter Murray-Rust, David Megginson, Tim Bray, and other members of the *xml-dev* mailing list recognized that XML parsers, while all compatible in the documents they could parse, were incompatible in their APIs. This observation led to the development of the Simple API for XML, SAX. SAX2 was released in 2000 to add greater configurability, namespace support, and several optional features.

Development of extensions to the core XML specification continues. Future directions include:

XFragment
: An effort to make sense out of XML document pieces that may not be considered well-formed documents in isolation.

XML Schemas
: An XML application that can describe the allowed content of documents conforming to a particular XML vocabulary.

XHTML
: A reformulation of HTML as a well-formed, modular, potentially valid XML application.

XML Query Language
: A language for finding the elements in a document that meet specified criteria.

Canonical XML
: A standard algorithm used for determining whether two XML documents are the same after throwing away insignificant details, such as whether single or double quotes are used around attribute values.

XML Signatures
: A standard means of digitally signing XML documents, embedding signatures in XML documents, and authenticating the resulting documents.

Many new extensions of XML remain to be invented. XML has proven itself a solid foundation for many other technologies.

CHAPTER 2

XML Fundamentals

This chapter shows you how to write simple XML documents. You'll see that an XML document is built from text content marked up with text tags, such as <SKU>, <Record_ID>, and <author>, that look superficially like HTML tags. However, in HTML you're limited to about a hundred predefined tags that describe web page formatting; in XML you can create as many tags as you need. Furthermore, these tags usually describe the type of content they contain, rather than formatting or layout information. In XML you don't say that something is a paragraph or an unordered list. You say that it's a book, a biography, or a calendar.

Although XML is looser than HTML in which tags it allows, it is much stricter about where those tags are placed and how they're written. In particular, all XML documents must be *well-formed*. Well-formedness rules specify constraints such as, "All opened tags must be closed" and "Attribute values must be quoted." These rules are unbreakable, which makes parsing XML documents easier and writing them a little harder, but still allows an almost unlimited flexibility of expression.

XML Documents and XML Files

An XML document contains text, never binary data. It can be opened with any program that knows how to read a text file. Example 2-1 is close to the simplest XML document imaginable. It is nonetheless a well-formed XML document. XML parsers can read and understand it (at least as far as a computer program can be said to understand anything).

Example 2-1: A Very Simple, Yet Complete, XML Document

```
<person>
  Alan Turing
</person>
```

In the most common scenario, this document would be the entire content of a file named *person.xml*, or perhaps *2-1.xml*. However, XML is not picky about the filename. As far as the parser is concerned, this file could be called *person.txt*, *person*, or *Hey you, there's some XML in this here file!* Your operating system may not like these names, but an XML parser won't care. The document might not even be in a file. It could be a record or a field in a database. A CGI program could generate it on the fly in response to a browser query. It could even be stored in more than one file, though that's unlikely for such a simple document. If it's served by a web server, it will probably be assigned the MIME media type application/xml or text/xml. However, specific XML applications may use more specific MIME media types, such as application/mathml+xml, application/XSLT+xml, image/svg+xml, text/vnd.wap.wml, or text/html (in special cases).

Elements, Tags, and Character Data

The document in Example 2-1 is composed of a single *element* whose type is `person`. This element is delimited by the *start tag* `<person>` and the *end tag* `</person>`. Everything between the element's start tag and end tag (exclusive) is the element's *content*. The content of this element is the text string:

```
Alan Turing
```

The whitespace is part of the content, though many applications will choose to ignore it. `<person>` and `</person>` are *markup*. The string `Alan Turing` and its surrounding whitespace are *character data*. The tag is the most common form of markup in an XML document, but there are other kinds we'll discuss later.

Tag Syntax

Superficially, XML tags look like HTML tags. Start tags begin with a < and end tags begin with a </. Both of these are followed by the name of the element and are closed by a >. However, unlike HTML tags, you are allowed to make up new XML tags as you go along. To describe a person, use `<person>` and `</person>` tags. To describe a calendar, use `<calendar>` and `</calendar>` tags. The names of the tags reflect the type of content inside the element, not how that content will be formatted.

Empty elements

There's a special syntax for *empty elements*, elements without content. These elements can be represented by tags that begin with < but end with />. For instance, in XHTML, an XMLized reformulation of standard HTML, the line break and horizontal rule elements are written as `<br/>` and `<hr/>` instead of `<br>` and `<hr>`. These elements are exactly equivalent to `<br></br>` and `<hr></hr>`, however. You decide which form to use for empty elements. However, what you cannot do in XML and XHTML (unlike HTML) is use only the start tag, such as `<br>` or `<hr>`, without using matching end tags. That would be a well-formedness error.

Case sensitivity

XML, unlike HTML, is case sensitive. `<Person>` is not the same as `<PERSON>`, and neither is the same as `<person>`. If you open an element with a `<person>` tag,

you can't close it with a </PERSON> tag. You're free to use upper- or lowercase, or both, as you choose. You just have to be consistent within any one element.

XML Trees

XML documents are trees. To explain this concept, let's look at a slightly more complicated XML document. Example 2-2 is a person element containing information suitably marked up to show its meaning.

Example 2-2: A More Complex XML Document Describing a Person

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

Parents and children

This XML document is still composed of one person element. However, now this element doesn't merely contain undifferentiated character data. It contains four *child elements*: a name element and three profession elements. The name element contains two child elements of its own, first_name and last_name.

The person element is called the *parent* of the name element and the three profession elements. The name element is the parent of the first_name and last_name elements. The name element and the three profession elements are sometimes called one another's *siblings*. The first_name and last_name elements are also siblings.

As in human society, any one parent may have multiple children. However, unlike human society, XML gives each child exactly one parent, not two. Each element (with one exception that we'll note shortly) has exactly one parent element. That is, it is completely enclosed by another element. If an element's start tag is inside an element, then its end tag must also be inside that element. Overlapping tags, as in this common example from HTML, are prohibited in XML. Since the em element begins inside the strong element, it must also finish inside the strong element.

The root element

Every XML document has one element without a parent. This is the first element in the document that contains all other elements. In Example 2-1 and Example 2-2 the person element fills this role and is called the document's *root element*. It is sometimes also called the *document element*. Every well-formed XML document has exactly one root element. Since elements may not overlap and since all elements except the root have exactly one parent, XML documents form a data

structure that programmers call a *tree*. Figure 2-1 diagrams this relationship for Example 2-2. Each gray box represents an element. Each black box represents character data. Each arrow represents a containment relationship.

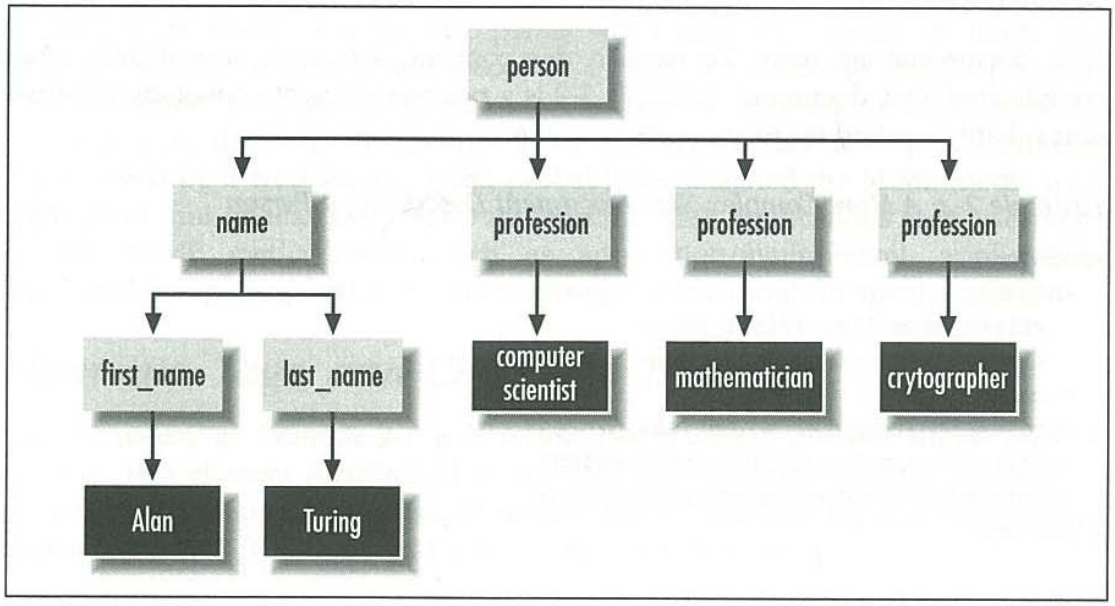

Figure 2-1: A tree diagram for Example 2-2

Mixed Content

In Example 2-2, the contents of the `first_name`, `last_name`, and `profession` elements were character data: text that does not contain any tags. The contents of the `person` and `name` elements were child elements and some whitespace that most applications ignore. This dichotomy between elements that contain only character data and elements that contain only child elements (and possibly a little whitespace) is common in data-oriented documents. However, XML can also be used for free-form, narrative documents, such as business reports, magazine articles, student essays, short stories, and web pages, as shown in Example 2-3.

Example 2-3: A Narrative-Organized XML Document

```
<biography>
  <name><first_name>Alan</first_name> <last_name>Turing</last_name>
  </name> was one of the first people to truly deserve the name
  <emphasize>computer scientist</emphasize>. Although his contributions
  to the field are too numerous to list, his best-known are the
  eponymous <emphasize>Turing Test</emphasize> and
  <emphasize>Turing Machine</emphasize>.

  <definition>The <term>Turing Test</term> is to this day the standard
  test for determining whether a computer is truly intelligent. This
  test has yet to be passed. </definition>

  <definition>The <term>Turing Machine</term> is an abstract finite
  state automaton with infinite memory that can be proven equivalent
  to any any other finite state automaton with arbitrarily large memory.
```

Example 2-3: A Narrative-Organized XML Document (continued)

```
  Thus what is true for a Turing machine is true for all equivalent
  machines no matter how implemented.
  </definition>

  <name><last_name>Turing</last_name></name> was also an accomplished
  <profession>mathematician</profession> and
  <profession>cryptographer</profession>. His assistance
  was crucial in helping the Allies decode the German Enigma
  machine. He committed suicide on <date><month>June</month>
  <day>7</day>, <year>1954</year></date> after being
  convicted of homosexuality and forced to take female
  hormone injections.
</biography>
```


This document's root element is `biography`. The `biography` contains `name`, `definition`, `profession`, and `emphasize` child elements. It also contains a lot of raw character data. Some elements, such as `last_name` and `profession`, contain only character data. Others, such as `name`, contain only child elements. Still others, such as `definition`, contain both character data and child elements. These elements are said to contain *mixed content*. Mixed content is common in XML documents used for anything organized as a written narrative, such as articles, essays, stories, books, novels, reports, and web pages. Mixed content is less common and harder to work with in the computer-generated and processed XML documents used for database exchange, object serialization, or persistent file formats. One of XML's strengths is the ease with which it can be adapted to the very different requirements of human-authored and computer-generated documents.

Attributes

XML elements can have attributes. An attribute is a name-value pair attached to the element's start tag. Names are separated from values by an equals sign, and possibly optional whitespace. Values are enclosed in single or double quotation marks. For example, this `person` element has a `born` attribute with the value `1912/06/23` and a `died` attribute with the value `1954/06/07`:

```
<person born="1912/06/23" died="1954/06/07">
  Alan Turing
</person>
```

This next element is exactly the same to an XML parser. It simply uses single quotations instead of double quotations and puts some extra whitespace around the equals signs:

```
<person born = '1912/06/23' died = '1954/06/07'>
  Alan Turing
</person>
```

The whitespace around the equals signs is purely a matter of personal aesthetics. The single quotes may be useful in cases in which the attribute value itself contains a double quote.

Example 2-4 shows how attributes might be used to encode much of the same information given in Example 2-2's data-oriented document.

Example 2-4: An XML Document That Describes a Person Using Attributes

```
<person>
  <name first="Alan" last="Turing"/>
  <profession value="computer scientist"/>
  <profession value="mathematician"/>
  <profession value="cryptographer"/>
</person>
```

This example raises the question of when and whether one should use child elements or attributes to hold information. This is a subject of heated debate. Some informaticians maintain that attributes are for metadata about the element, while elements are for the information itself. Others point out that it's not always easy to identify what is data and what is metadata. Indeed the answer may depend on the uses to which the information is put.

The fact that each element may have no more than one attribute with a given name is undisputed. This is unlikely to be a problem for a birth or death date, but it would be an issue for a profession, name, address, or anything else of which an element might plausibly have more than one. Furthermore, attributes are quite limited in structure. An attribute's value is simply a text string. The division of a date into a year, month and day with slashes is at the limit of the substructure that can be reasonably encoded in an attribute. Consequently, an element-based structure is much more flexible and extensible. Nonetheless, attributes are certainly more convenient in some applications. Ultimately, if you're designing your own XML vocabulary, you decide when to use elements and when to use attributes.

Attributes are also useful in narrative-oriented documents, as Example 2-5 demonstrates. Here what belongs to elements or attributes is perhaps a little more obvious. The narrative's raw text is presented as character data inside elements. Additional information that annotates the data is presented as attributes. This information includes source references, image URLs, hyperlinks, and birth and death dates. Even here, however, there's more than one way to do it. For instance, the footnote numbers could be attributes of the footnote element rather than character data.

Example 2-5: A Narrative XML Document That Uses Attributes

```
<biography xmlns:xlink="http://www.w3.org/1999/xlink/namespace/">

  <image source="http://www.turing.org.uk/turing/pi1/bus.jpg"
  width="152" height="345"/>
  <person born='1912/06/23'
  died='1954/06/07'><first_name>Alan</first_name>
  <last_name>Turing</last_name> </person> was one of the first people
  to truly deserve the name <emphasize>computer scientist</emphasize>.
  Although his contributions to the field were too numerous to list,
  his best-known are the eponymous <emphasize xlink:type="simple"
  xlink:href="http://cogsci.ucsd.edu/~asaygin/tt/ttest.html">Turing
```

Example 2-5: A Narrative XML Document That Uses Attributes (continued)

```
Test</emphasize> and <emphasize  xlink:type="simple"
xlink:href="http://mathworld.wolfram.com/TuringMachine.html">Turing
Machine</emphasize>.

<last_name>Turing</last_name> was also an accomplished
<profession>mathematician</profession> and
<profession>cryptographer</profession>. His assistance
was crucial in helping the Allies decode the German Enigma
machine.<footnote source="The Ultra Secret, F.W. Winterbotham,
1974">1</footnote>

He committed suicide on <date><month>June</month> <day>7</day>,
<year>1954</year></date> after being convicted of homosexuality
and forced to take female hormone injections.<footnote
source="Alan Turing: the Enigma, Andrew Hodges, 1983">2</footnote>

</biography>
```


XML Names

The XML specification can be quite legalistic and picky at times, but it tries to be efficient when possible. One way it tries to increase its efficiency is by reusing the same rules for different objects when possible. For example, the rules for XML element names are also the rules for XML attribute names and the names of several less common constructs. Generally, these names are referred to as *XML names*.

Element and other XML names may contain any alphanumeric character. These characters include the standard English letters A through Z and a through z, as well as the digits 0 through 9. XML names may also include non-English letters, numbers, and ideograms, such as ö, ç, ϑ, and ψ. They may also include these three punctuation characters:

_ Underscore

\- Hyphen

. Period

XML names may not contain other punctuation characters, such as quotation marks, apostrophes, dollar signs, carets, percent symbols, and semicolons. The colon is allowed, but its use is reserved for namespaces, as discussed in Chapter 4, *Namespaces*. XML names may not contain whitespace of any kind, whether a space, a carriage return, a line feed, or a nonbreaking space.

XML names may start only with letters, ideograms, or the underscore character. They may not start with a number, a hyphen, or a period. There is no limit to the length of an element or other XML name. The following elements are all well-formed:

- `<Drivers_License_Number>98 NY 32</Drivers_License_Number>`
- `<month-day-year>7/23/2001</month-day-year>`
- `<first_name>Alan</first_name>`
- `<_4-lane>I-610</_4-lane>`

- `<téléphone>011 33 91 55 27 55 27</téléphone>`
- `<ονομα>Melina Merkouri</ονομα>`

These elements are not well-formed:

- `<Driver's_License_Number>98 NY 32</Driver's_License_Number>`
- `<month/day/year>7/23/2001</month/day/year>`
- `<first name>Alan</first name>`
- `<4-lane>I-610</4-lane>`

Entity References

The character data inside an element may not contain a raw unescaped opening angle bracket <. This character is always interpreted as the beginning of a tag. If you need to use this character in your text, you can escape it using the `<` *entity reference*. When a parser reads the document, it replaces the `<` entity reference with the actual < character. However, it does not confuse `<` with the start of a tag. For example:

```
<SCRIPT LANGUAGE="JavaScript">
  if (location.host.toLowerCase().indexOf("ibiblio") &lt; 0) {
    location.href="http://www.ibiblio.org/xml/";
  }
</SCRIPT>
```

The character data inside an element may not contain a raw unescaped ampersand & either. This character is always interpreted as the beginning of an entity or character reference. However, the ampersand may be escaped using the `&` entity reference like this:

```
<publisher>O'Reilly &amp; Associates</publisher>
```

Entity references such as `&` and `<` are considered markup. When an application parses an XML document, it replaces this particular markup with the actual characters the entity reference refers to. The result, O'Reilly & Associates in the previous example, is called *parsed character data*; that is, the character data that's left after the document is parsed and the entities are resolved.

XML predefines exactly five entity references:

`<`
: The less-than sign, or opening angle bracket (<)

`&`
: The ampersand (&)

`>`
: The greater-than sign, or closing angle bracket (>)

`&quot;`
: The straight, double quotation marks (")

`&apos;`
: The apostrophe, or single quote (')

Only < and & must be used in place of the literal characters in element content. The other references are optional. " and ' are useful inside attribute values, where a raw " or ' might be misconstrued as ending the attribute value. For example, this image tag uses the ' entity reference to fill in the apostrophe in O'Reilly:

```
<image source='oreilly_koala3.gif' width='122' height='66'
       alt='Powered by O&apos;Reilly Books'
/>
```

Although misinterpreting an unescaped greater-than sign > as closing a tag it wasn't meant to close is impossible, > is allowed for symmetry with <.

In addition to these five predefined entity references, you can define others in the document type definition. We'll discuss how to do this in Chapter 3, *Document Type Definitions.*

CDATA Sections

When an XML document includes samples of XML or HTML source code, the < and & characters in those samples must be encoded as < and &. The more sections of literal code a document includes and the longer they are, the more tedious this encoding can become. To facilitate the process, you can enclose each sample of literal code in a *CDATA section.* A CDATA section is set off by a <![CDATA[and]]>. Everything between the <![CDATA[and the]]> is treated as raw character data. Less-than signs don't start tags. Ampersands don't start entity references. Everything is simply character data, not markup.

For example, in a Scalable Vector Graphics (SVG) tutorial written in XHTML, you might see something like this:

```
<p>You can use a default <code>xmlns</code> attribute to avoid
having to add the svg prefix to all your elements:</p>
<![CDATA[
  <svg xmlns="http://www.w3.org/2000/svg"
       width="12cm" height="10cm">
    <ellipse rx="110" ry="130" cx="1cm" cy="1cm" />
    <rect x="4cm" y="1cm" width="3cm" height="6cm" />
  </svg>
]]>
```

The SVG source code was included directly in the XHTML file without having to carefully replace each < with <. The result is a sample SVG document, not an embedded SVG picture, as would happen if this example were not placed inside a CDATA section.

The only thing that cannot appear in a CDATA section is the CDATA section end delimiter]]>.

CDATA sections are convenient for human authors, not programs. Parsers are not required to tell you whether a particular block of text came from a CDATA section, from normal character data, or from character data that contained entity references such as < and &. By the time you get access to the data, these differences may have been washed away.

Comments

XML documents can be commented so coauthors can leave notes for each other documenting why they've done what they've done or items that remain to be done. XML comments are syntactically similar to HTML comments; they begin with `<!--` and end with the first occurrence of `-->`. For example:

```
<!-- I need to verify and update these links when I get a chance. -->
```

The double hyphen `--` should not appear anywhere inside the comment until the closing `-->`. In particular, a three-hyphen close like `--->` is specifically forbidden.

Comments may appear anywhere in the document's character data. They may also appear before or after the root element. (Comments are not elements, so this does not violate the tree structure or the one-root element rules for XML.) However, comments may not appear inside a tag or another comment.

XML parsers may or may not pass along information included in comments. They are certainly free to drop them out if they choose. Do not write documents or applications that depend on the availability of comments. Comments are strictly for making the raw source code of an XML document more legible to human readers. They are not intended for any computer program. If you need such a feature for a computer program, use a *processing instruction* instead.

Processing Instructions

In HTML comments are sometimes abused to support nonstandard extensions. For instance, the contents of the `style` element are sometimes enclosed in a comment to protect it from display by a nonscript-aware browser. The Apache web server parses comments in *.shtml* files to recognize server-side includes. Unfortunately these documents may not survive being passed through various HTML editors and processors with their comments and associated semantics intact. Worse yet, it's possible for an innocent comment to be misconstrued as input to the application.

XML provides the processing instruction as an alternative means of passing information to particular applications that may read the document. A processing instruction begins with `<?` and ends with `?>`. Immediately following the `<?` is an XML name called the *target*, possibly the name of the application for which this processing instruction is intended, or perhaps just an identifier for this particular processing instruction. The rest of the processing instruction contains text in a format appropriate for the applications the instruction is intended for.

For example, HTML uses a Robots `META` tag to tell search engines and other robots whether and how they should index a page. The following processing instruction has been proposed as an equivalent for XML documents:

```
<?robots index="yes" follow="no"?>
```

The target of this processing instruction is `robots`. The syntax of this particular processing instruction is two pseudoattributes, one named `index` and one named `follow`, whose values are either `yes` or `no`. The semantics of this particular processing instruction are that if the `index` attribute has the value `yes`, then this page will be indexed by a search engine robot. If `index` has the value `no`, then it

won't be. Similarly, if `follow` has the value `yes`, then links from this document will be followed. Otherwise, they won't be.

Other processing instructions may have totally different syntaxes and semantics. For instance, processing instructions can contain an unlimited amount of text. PHP includes large programs in processing instructions:

```
<?php
  mysql_connect("database.unc.edu", "clerk", "password");
  $result = mysql("CYNW", "SELECT LastName, FirstName FROM Employees
    ORDER BY LastName, FirstName");
  $i = 0;
  while ($i < mysql_numrows ($result)) {
     $fields = mysql_fetch_row($result);
     echo "<person>$fields[1] $fields[0] </person>\r\n";
     $i++;
  }
  mysql_close();
?>
```


Processing instructions are markup, but not elements. Consequently, like comments, processing instructions may appear anywhere in an XML document outside of a tag, including before or after the root element. The most common processing instruction, `xml-stylesheet` attaches stylesheets to documents. It always appears before the root element, as Example 2-6 demonstrates. In this example, the `xml-stylesheet` processing instruction tells browsers to apply the CSS stylesheet *person.css* to this document before showing it to the reader.

Example 2-6: An XML Document with an xml-stylesheet Processing Instruction

```
<?xml-stylesheet href="person.css" type="text/css"?>
<person>
  Alan Turing
</person>
```

The processing instruction name xml, in any combination of case, (that is, xml, XML, Xml, etc.) is resolved for use by the W3C. Otherwise, you're free to pick any legal XML name for your processing instructions.

The XML Declaration

XML documents should, but do not have to, begin with an *XML declaration*. The XML declaration looks like a processing instruction with the name `xml` and `version`, `standalone`, and `encoding` attributes. Technically, it's not a processing instruction though, just the XML declaration; nothing more, nothing less. Example 2-7 demonstrates.

Example 2-7: A Very Simple XML Document with an XML Declaration

```
<?xml version="1.0" encoding="US-ASCII" standalone="yes"?>
<person>
  Alan Turing
</person>
```

XML documents do not have to have an XML declaration. However, if an XML document does have an XML declaration, then the declaration must be the first thing in the document. It must not be preceded by comments, whitespace, or processing instructions. The reason is that an XML parser uses the first five characters (`<?xml`) to make reasonable guesses about the encoding, such as whether the document uses a single- or multibyte character set. An invisible Unicode byte order mark is the only thing that may precede the XML declaration. We'll discuss this further in Chapter 5, *Internationalization.*

version

The `version` attribute of the XML declaration always has the value 1.0. It is plausible that at some future time this value may change, and then the `version` attribute will be used to distinguish between documents that adhere to XML 1.0 and documents that adhere to a later version of the specification. However, it's also plausible that this will never happen. XML 1.0 is a fairly robust specification designed for extensibility. Most efforts to improve it—namespaces and schemas, for example—are built as layers on top of the infrastructure that XML 1.0 provides. They do not require breaking existing parsers or documents. Parsers that read XML 1.0 documents can still read documents that use namespaces, schemas, or hypothetical future developments. They may not understand the extra semantics that these new developments offer, but they can still read the documents. XML is designed to be both forward and backward compatible.

encoding

So far we've been a little cavalier about encodings. We've said that XML documents are composed of pure text; but we haven't said what encoding that text uses. Is it ASCII? Latin-1? Unicode? Something else?

The short answer to this question is "Yes." The long answer is that, by default, XML documents are assumed to be encoded in the UTF-8 variable length encoding of the Unicode character set. This encoding is a strict superset of ASCII, so pure ASCII text files are also UTF-8 documents. However, most XML processors can handle a much broader range of encodings. All you have to do is give the name of the encoding in the XML declaration. Example 2-8 shows how you'd indicate that a document was written in the ISO 8859-1 (Latin-1) character set. This set includes characters needed for many non-English Western European languages, such as ö and ç.

Example 2-8: An XML Document Encoded in Latin-1

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

The `encoding` attribute is optional in an XML declaration. If it is omitted, the Unicode character set is assumed. The parser may use the first several bytes of the file to guess which form of Unicode is in use.

Chapter 5 discusses the different encodings and proper handling of non-English XML documents in much greater detail.

standalone

If the `standalone` attribute has the value `no`, then an application may have to read an external DTD (that is, a DTD in a file other than the one it's reading now) to determine the proper values for parts of the document. For instance, a DTD may provide default values for attributes a parser is required to report, though they aren't actually present in the document.

Documents that do not have DTDs, like all the documents in this chapter, can use the value `yes` for the `standalone` attribute. Documents that do have DTDs can also use the value `yes` for the `standalone` attribute if the DTD doesn't change the content of the document or if the DTD is purely internal. Chapter 3 provides details for documents with DTDs.

The `standalone` attribute is optional in an XML declaration. If it is omitted, the value `no` is assumed.

Checking Documents for Well-Formedness

Every legal XML document, without exception, must be well-formed. This means it must adhere to a number of rules, including:

- Every start tag must have a matching end tag
- Elements may not overlap
- There must be exactly one root element
- Attribute values must be quoted
- An element may not have two attributes with the same name
- Comments and processing instructions may not appear inside tags
- No unescaped `<` or `&` signs may occur in the element's or attribute's character data

This list is not exhaustive. A document can be malformed in many ways. You'll find a complete list in Chapter 18, *XML 1.0 Reference*. Some of these malformations involve constructs we have not yet discussed, such as DTDs. Others are very unlikely to occur if you follow the examples in this chapter (for example, including whitespace between the opening `<` and the element name in a tag).

Whether the error is small or large, likely or unlikely, an XML parser reading a document must report it. It may report multiple well-formedness errors it detects in the document. However, the parser is not allowed to fix the document and make a best faith effort of providing what it thinks the author really meant. It can't fill in missing quotes around attribute values, insert an omitted end tag, or ignore the comment that's inside a start tag. The parser is required to return an error. The objective here is to avoid the bug-for-bug compatibility wars that plagued early web browsers and continues to this day. Consequently, before you publish an XML document, whether that document is a web page, input to a database or something else, you should check it for well-formedness.

The simplest way to check the document is by loading it into a web browser that understands XML documents, such as Opera or Mozilla. If the document is well-formed, the browser will display it. If it isn't, then it will show an error message.

Instead of loading the document into a web browser, you can use an XML parser directly. Most XML parsers are not intended for end users; they are class libraries designed to be embedded into a more user-friendly program like Internet Explorer. They provide a minimal command-line interface, if that; and that interface is often not well documented. Nonetheless, running a batch of files through a command-line interface is sometimes faster than loading each of them into a web browser. Once you learn about DTDs, you can use the same tools to validate documents.

Many XML parsers are available in a variety of languages. Here we'll demonstrate checking well-formedness with the Apache XML Project's Xerces-J, which you can download from *http://xml.apache.org/xerces-j/index.html*. This open source package is written in pure Java, so it should run across all major platforms. The procedure should be similar for other parsers, though details vary.

To use this parser you'll first need to install a Java 1.1 or later compatible virtual machine. Virtual machines for Windows, Solaris, and Linux are available from *http://java.sun.com/*. To install Xerces-J just add the *xerces.jar* and *xercesSamples.jar* files to your Java class path. In Java 2 you can simply put those *.jar* files into your *jre/lib/ext* directory.

The class that actually checks files for well-formedness is called `sax.SAXCount`. It's run from a Unix shell or DOS prompt, like any other standalone Java program. The command-line arguments are the URLs to or filenames of the documents you want to check. Here's the result of running *SAXCount* against the original version of Example 2-5. The first line of output locates the file's first problem. The rest of the output is an irrelevant stack trace:

```
D:\xian\examples\02>java sax.SAXCount 2-5.xml
[Fatal Error] 2-5.xml:3:30: The value of attribute "height" must not
contain the '<' character.
Stopping after fatal error: The value of attribute "height" must not
contain the '<' character.
at org.apache.xerces.framework.XMLParser.reportError(XMLParser.java:1282)
at org.apache.xerces.framework.XMLDocumentScanner.reportFatalXMLError(XM
LDocumentScanner.java:644)
at org.apache.xerces.framework.XMLDocumentScanner.scanAttValue(XMLDocume
ntScanner.java:519)
at org.apache.xerces.framework.XMLParser.scanAttValue(XMLParser.
java:1932)
at org.apache.xerces.framework.XMLDocumentScanner.scanElement(XMLDocumen
tScanner.java:1800)
at org.apache.xerces.framework.XMLDocumentScanner$ContentDispatcher.disp
atch(XMLDocumentScanner.java:1223)
at org.apache.xerces.framework.XMLDocumentScanner.parseSome(XMLDocumentS
canner.java:381)
at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:1138)
at org.apache.xerces.framework.XMLParser.parse(XMLParser.java:1177)
at sax.SAXCount.print(SAXCount.java:135)
at sax.SAXCount.main(SAXCount.java:331)
```

As you can see, the program found an error. In this case the error message wasn't particularly helpful. The actual problem wasn't that an attribute value contained a < character. It was that the closing quote was missing from the attribute value. Still that information was sufficient to locate and fix the problem. Despite the long list of output in this example, *SAXCount* reports only the first error in the file, so you may have to run it multiple times until all mistakes are found and fixed. Once we fixed Example 2-5 to make it well-formed, *SAXCount* simply reported how long it took to parse the document and what it saw when it did:

```
D:\xian\examples\02>java sax.SAXCount 2-5.xml
2-5.xml: 140 ms (17 elems, 12 attrs, 0 spaces, 564 chars)
```

Once the document has been made well-formed, it can be passed to a web browser, a database, or whatever other program is waiting to receive it. Almost any nontrivial document crafted by hand contains well-formedness mistakes. That's why it's important to check your work before you publish it.