Processing XML with Java

Elliotte Rusty Harold

Copyright 2001, 2002 Elliotte Rusty Harold

Welcome to Processing XML with Java, a complete tutorial about writing Java programs that read and write XML documents. This is the most comprehensive and up-to-date book about integrating XML with Java (and vice versa) you can buy. It contains over 1000 pages of detailed information on SAX, DOM, JDOM, JAXP, TrAX, XPath, XSLT, SOAP, and lots of other juicy acronyms. This book is written for Java programmers who want to learn how to read and write XML documents from their code. The paper version is published by Addison-Wesley, and can be found at fine bookstores everywhere including Amazon and Barnes & Noble. The list price is $54.95, but most bookstores are offering their usual discounts.

Normally, this is the point where I'd spend a few paragraphs describing just what's in the book and how important it is to your education, your career, and your love life; but this time I've done something a little different. The entire book is available online. You can read every chapter and every page so you can see for yourself how well this book answers your questions such as, "Why does SAX truncate the text in my documents after a few thousand characters?" "How do I serialize a DOM Document object in an implementation-independent way?" or, "Why doesn't my significant other understand the importance of a building a life size Millennium Falcon in our backyard?" Consequently, I'll forego the usual hype. Check the book out for yourself. The entire book is here at Cafe con Leche. You can read every word of it, all seventeen chapters and two appendixes. If you like it, please buy a copy. I promise it's cheaper than printing all 1100+ pages on your laser printer.

- Preface
- Acknowledgements
- 1 XML for Data
- 2 XML Protocols
- 3 Writing XML with Java
- 4 Converting Flat Files to XML
- 5 Reading XML
- 6 SAX
Examples

I've extracted out all the examples into individual files. You can download them as a zip archive if you like.

Contacting the Author

Your commentary and feedback is much desired, both on major issues (e.g. "Why don't you cover JAXB?") and minor ones ("Cat is misspelled in the first sentence of page 42."). Please send all feedback directly to me at elharo@metalab.unc.edu.

Prerequisites

This is not an introductory book. It assumes that you're completely familiar with Java including objects, classes, polymorphism, I/O, network programming, and more. It also assumes that you're familiar with XML at about the level of the XML Bible, 2nd Edition. In many ways this book picks up where that one left off.

Copying these Files

I'd like to ask that you not mirror these files on your own servers. It's difficult to keep multiple, unofficial mirrors in sync, and I don't want to deal with questions based on out of date copies. This site is hosted on the very fast, very well-connected IBiblio servers near the Internet backbone and should have enough bandwidth for everyone. If you want to save a copy on your local hard drive for your own use, feel free,
However, please don't pass out copies of your own copies to anyone else. Instead refer your friends and colleagues to this web site.

Colophon

This entire book was written in XML from start to finish. The specific XML application used is DocBook 4.2.0. I use the jEdit text editor on Windows and Linux to write. XInclude is used to merge the individual chapters and examples together. Michael Kay's SAXON XSLT processor and Norm Walsh's XSL stylesheets for Docbook 1.52.2 produce the HTML and XSL-FO output. I use FOP 0.20.4 to convert the XSL-FO files to PDFs. The DocBook source files were pulled into Adobe FrameMaker to layout the printed book.

Copyright 2001-2003 Elliotte Rusty Harold
elharo@metalab.unc.edu
Last Modified April 17, 2003
One night five developers, all of whom wore very thick glasses and had recently been hired by Elephants, Inc., the world’s largest purveyor of elephants and elephant supplies, were familiarizing themselves with the company’s order processing system when they stumbled into a directory full of XML documents on the main server. “What’s this?” the team leader asked excitedly. None of them had ever heard of XML before so they decided to split up the files between them and try to figure out just what this strange and wondrous new technology actually was.

The first developer, who specialized in optimizing Oracle databases, printed out a stack of FMPXMLRESULT documents generated by the FileMaker database where all the orders were stored, and began poring over them. “So this is XML! Why, it’s nothing novel. As anyone can see who’s able, an XML document is nothing but a table!”

“What do you mean, a table?” replied the second programmer, well versed in object oriented theory and occupied with a collection of XMI documents that encoded UML diagrams for the system. “Even a Visual Basic programmer could see that XML documents aren’t tables. Duplicates aren’t allowed in a table relation, unless this is truly some strange mutation. Classes and objects is what these document are. Indeed, it should be obvious on the very first pass. An XML document is an object and a DTD is a class.”

“Objects? A strange kind of object, indeed!” said the third developer, a web designer of some renown, who had loaded the XHTML user documentation for the order processing system into Mozilla. “I don’t see any types at all. If you think this is an object, then it’s your software I refuse to install. But with all those stylesheets there, it should be clear to anyone not sedated, that XML is just HTML updated!”
“HTML? You must be joking” said the fourth, a computer science professor on sabbatical from MIT, who was engrossed in an XSLT stylesheet that validated all the other documents against a Schematron schema. “Look at the clean nesting of hierarchical structures, each tag matching its partner as it should. I’ve never seen HTML that looks this good. What we have here is S-expressions, which is certainly nothing new. Babbage invented this back in 1882!”

“An S expression?” queried the technical writer, who was occupied with documentation for the project written in DocBook. “Maybe that means something to those in your learned profession. But to me, this looks just like a FrameMaker MIF file. However, locating the GUI is taking me awhile.”

And so they argued into the night, none of them willing to give an inch, all of them presenting still more examples to prove their points, none of them bothering to look at the others’ examples. Indeed, they’re probably still arguing today. You can even hear their shouts from time to time on xml-dev. Their mistake, of course, was in trying to force XML into the patterns of technologies they were already familiar with rather than taking it on its own terms. XML can store data, but it is not a database. XML can serialize objects, but an XML document is not an object. Web pages can be written in XML, but XML is not HTML. Functional (and other) programming languages can be written in XML, but XML is not a programming language. Books are written in XML, but that doesn’t make XML desktop publishing software.

XML is something truly new that has not been seen before in the world of computing. There have been precursors to it, and there are always fanatics who insist on seeing XML through database (or object, or functional, or S-expression) colored glasses. But XML is none of these things. It is something genuinely unique and new in the world of computing; and it can only be understood when you’re willing to accept it on its own terms, rather than forcing it into yesterday’s pigeon holes.

There are a lot of tools, APIs, and applications in the world that try to pretend XML is something more familiar to programmers; that it’s just a funny kind of database, or just like an object, or just like remote procedure calls. These APIs are occasionally useful in very restricted and predictable environments. However, they are not suitable for processing XML in its most general format. They work well in their limited domains, but they fail when presented with XML that steps outside the artificial boundaries they’ve defined. XML was designed to be extensible, but it’s a sad fact that many of the tools designed for XML aren’t nearly as extensible as XML itself.

This book is going to show you how to handle XML in its full generality. It pulls no punches. It does not pretend that XML is anything except XML, and it shows you how to design your programs so that they handle real XML in all its messiness: valid and invalid, mixed and unmixed, typed and untyped, and both all and none of these at the same time. To that end, this book focuses on those APIs that don’t try to hide the XML. In particular, there are three major Java APIs that correctly model XML, as opposed to modeling a particular class of XML documents or some narrow subset of XML. These are:

- SAX, the Simple API for XML
● DOM, the Document Object Model
● JDOM, a Java native API

These APIs are the core of this book. In addition I cover a number of preliminaries and supplements to the basic APIs including:

● XML syntax
● DTDs, schemas, and validity
● XPath
● XSLT and the TrAX API
● JAXP, a combination of SAX, DOM, and TrAX with a few factory classes

And, since we’re going to need a few examples of XML applications to demonstrate the APIs with, I also cover XML-RPC, SOAP, and RSS in some detail. However, the techniques this book teaches are hardly limited to just those three applications.

**Who You Are**

This book is written for experienced Java programmers who want to integrate XML into their systems. Java is the ideal language for processing XML documents. Its strong Unicode support in particular made it the preferred language for many early implementers. Consequently, more XML tools have been written in Java than in any other language. More open source XML tools are written in Java than in any other language. More programmers process XML in Java than in any other language.

*Processing XML with Java* will teach you how to:

● Save XML documents from applications written in Java
● Read XML documents produced by other programs
● Search, query, and update XML documents
● Convert legacy flat data into hierarchical XML
● Communicate with network servers that send and receive XML data
● Validate documents against DTDs, schemas, and business rules
● Combine functional XSLT transforms with traditional imperative Java code

This book is meant for Java programmers who need to do anything with XML. It teaches the fundamentals and advanced topics, leaving nothing out. It is a comprehensive course in processing XML with Java that takes developers from little knowledge of XML to designing sophisticated XML applications and parsing complicated documents. The examples cover a wide range of possible uses including file formats, data exchange, document transformation, database integration, and more.
What You Need to Know

This is not an introductory book with respect to either Java or XML. I assume you have substantial prior experience with Java and preferably some experience with XML. On the Java side, I will freely use advanced features of the language and its class library without explanation or apology. Among other things, I assume you are thoroughly familiar with:

- Object oriented programming including inheritance and polymorphism
- Packages and the CLASSPATH. You should not be surprised by classes that do not have `main()` methods or that are not in the default package.
- I/O including streams, readers, and writers. You should understand that `System.out` is a horrible example of what really goes on in Java programs.
- The Java Collections API including hash tables, maps, sets, iterators, and lists.

In addition, in one or two places in this book I’m going to use some SQL and JDBC. However, these sections are relatively independent of the rest of the book; and chances are if you aren’t already familiar with SQL, then you don’t need the material in these sections anyway.

What You Need to Have

XML is deliberately architecture, platform, operating system, GUI, and language agnostic (in fact, more so than Java). It works equally well on Mac OS, Windows, Linux, OS/2, various flavors of Unix, and more. It can be processed with Python, C++, Haskell, ECMAScript, C#, Perl, Visual Basic, Ruby, and of course Java. No byte order issues need concern you if you switch between PowerPC, X86, or other architectures. Almost everything in this book should work equally well on any platform that’s capable of running Java.

Most of the material in this book is relatively independent of the specific Java version. Java 1.4 bundles SAX, DOM, and a few other useful classes into the core JDK. However, these are easily installed in earlier JVMs as open source libraries from the Apache XML Project and other vendors. For the most part, I used Java 1.3 and 1.4 when testing the examples; and it’s possible that a few classes and methods have been used that are not available in earlier versions. In most cases, it should be fairly obvious how to backport them. All of the basic XML APIs except TrAX should work in Java 1.1 and later. TrAX requires Java 1.2 or later.
Up To Cafe con Leche
DOM is a read-write API. DOM documents are created not only by parsing text files, but also by creating new documents in memory out of nothing at all. These documents can then be serialized onto a stream or into a file. The abstract factory interface that creates new Document objects is called DOMImplementation. The Document interface has a dual purpose. First it represents XML documents themselves and provides access to their contents, document type declaration, and other properties. Secondly it too is an abstract factory responsible for creating the nodes that go in the document: elements, text, comments, processing instructions, etc. Each such node belongs exclusively to the document that created it and cannot be moved to a different document.

DOMImplementation

The DOMImplementation interface, shown in Example 10.1, is an abstract factory that is responsible for creating two things, new Document and DocumentType objects. It also provides the hasFeature() method discussed in the last chapter that tells you what features this implementation supports.

Example 10.1. The DOMImplementation interface
package org.w3c.dom;

public interface DOMImplementation {

    public DocumentType createDocumentType(
        String rootElementQualifiedName,
        String publicID, String systemID) throws DOMException;

    public Document createDocument(String rootElementNamespaceURI,
        String rootElementQualifiedName, DocumentType doctype)
        throws DOMException;

    public boolean hasFeature(String feature, String version);

}

For example, given a DOMImplementation object named impl, this chunk of code creates a new DocumentType object named svgDOCTYPE pointing to the Scalable Vector Graphics DTD:

    DocumentType svgDOCTYPE = impl.createDocumentType("svg",
        "-//W3C//DTD SVG 1.0//EN",
        "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd");

If the DTD does not have a public ID, you can just pass null for the second argument.

You can use this DocumentType object when constructing a new SVG Document object:

    Document svgDoc = impl.createDocument(
        "http://www.w3.org/2000/svg", "svg", svgDOCTYPE);

If svgDoc were serialized into a text file, it would look something like this (modulo insignificant white space):

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
    "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg"/>

Of course not all XML documents have DOCTYPE declarations or namespace URIs. If the document is merely well-formed, then you can just pass null for the doctype argument. If the document root element is not in a namespace, you can also pass null for the namespace URI. This code fragment creates an XML-RPC document with neither a document type declaration nor a namespace URI:
Document xmlrpc = impl.createDocument(null, "methodCall", null);

These Document objects, with or without DOCTYPE declarations, are not yet complete. In particular, they do not yet have any content beyond an empty root element. For that, you’ll have to use the methods of the Document interface to create nodes, and the methods of the Node interface to add these newly created nodes to the tree.
Locating a DOMImplementation

So far, I’ve deliberately avoided the crucial question of how one creates a DOMImplementation object in the first place. Since DOMImplementation is an interface, not a class, it cannot be instantiated directly through its own constructor. Instead you have to build it in one of three ways:

- Construct the implementation specific class
- Use the JAXP DocumentBuilder factory class.
- Use the DOM Level 3 DOMImplementationRegistry factory class.

Implementation Specific Class

Directly constructing an instance of the vendor class that implements DOMImplementation is the simplest of the three alternatives. However, the name of this class and how it’s created vary from one implementation to the next. For example, in Xerces the org.apache.xerces.dom.DOMImplementationImpl singleton class implements the DOMImplementation interface. The singleton object is retrieved via the getDOMImplementation() factory method like this:

```java
DOMImplementation impl = DOMImplementationImpl.getDOMImplementation();
```

However, if you were to switch to a different implementation, you’d need to change your source code and recompile. For example, in the Oracle XML parser for Java, oracle.xml.parser.v2.XMLDOMImplementation class implements the DOMImplementation interface and instances of this class are created with a no-args constructor, like this:

```java
DOMImplementation impl = new XMLDOMImplementation();
```

In both cases, the implementation specific object is assigned to a variable of type DOMImplementation. This allows the compiler to make sure you don’t accidentally use any implementation-specific methods in the object, or tie the code too tightly to one vendor. The implementation-dependent code should be limited to this one line.

JAXP DocumentBuilder

The JAXP DocumentBuilder class introduced in the last chapter has a getDOMImplementation
Locating a DOMImplementation

() method that can locate a local DOMImplementation class.

```java
public abstract DOMImplementation getDOMImplementation();
```

For example, this code fragment uses JAXP to create a new SVG Document object in memory:

```java
try {
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   DocumentBuilder builder = factory.newDocumentBuilder();
   DOMImplementation impl = builder.getDOMImplementation();

   DocumentType svgDOCTYPE = impl.createDocumentType("svg",
            "-//W3C//DTD SVG 1.0//EN",
            "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd");
   Document svgDoc = impl.createDocument(
            "http://www.w3.org/2000/svg", "svg", svgDOCTYPE
   );
   // work with the document...
}
```

If you only want to create a new Document, then DocumentBuilder also has a `newDocument()` method that shortcuts the DOMImplementation class:

```java
public abstract Document newDocument();
```

However, this method does not properly set a root element. Thus the documents created by `newDocument()` are at least initially malformed; and therefore I recommend that you don’t use this method. Just use JAXP to retrieve a DOMImplementation object and use its `createDocument()` method instead.

The specific implementation JAXP chooses is determined in the same way as described for locating a parser in the last chapter. That is, first JAXP reads the `javax.xml.parsers.DocumentBuilderFactory` system property, then the `lib/jaxp.properties` file, then the
Locating a DOMImplementation

META-INF/services/javax.xml.parsers.DocumentBuilderFactory file in all JAR files available to the runtime, and then finally a fallback class hardcoded into the JAXP implementation.

**DOM3 DOMImplementationRegistry**

The final option for locating the DOMImplementation is new in DOM Level 3 and only supported by Xerces 2 so far. This is the DOMImplementationRegistry class shown in Example 10.2.

**Example 10.2. The DOMImplementationRegistry class**

```java
package org.w3c.dom;

public class DOMImplementationRegistry {

    // The system property that specifies DOMImplementationSource class names.
    public static String PROPERTY = "org.w3c.dom.DOMImplementationSourceList";

    public static DOMImplementation getDOMImplementation(
        String features) throws ClassNotFoundException, InstantiationException, IllegalAccessException;

    public static void addSource(DOMImplementationSource s)
        throws ClassNotFoundException, InstantiationException, IllegalAccessException;

}
```

The `getDOMImplementation()` method returns a DOMImplementation object that supports the features given in the argument, or null if no such implementation can be found. For example, this code fragment requests a DOMImplementation that supports XML DOM Level 1, any version of the traversal module, and DOM Level 2 events:

```java
try {
    DOMImplementation impl = DOMImplementationRegistry.getDOMImplementation("XML 1.0 Traversal Events 2.0");
    if (impl != null) {
        DocumentType svgDOCTYPE = impl.createDocumentType("svg",    
            "-//W3C//DTD SVG 1.0//EN", 
            "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd");
        Document svgDoc = impl.createDocument(    
            http://www.cafeconleche.org/books/xmljava/chapters/ch10s02.html (3 of 5)2006-07-06 오전 10:19:48
```
Locating a DOMImplementation

```java
"http://www.w3.org/2000/svg", "svg", svgDOCTYPE
);
    // work with the document...

}
}
catch (Exception e) {
    System.out.println(e);
}

Be sure to check whether the implementation returned is null before using it. Many installations may not be able to support all the features you ask for.

DOMImplementationRegistry searches for DOMImplementation classes by looking at the value of the org.w3c.dom.DOMImplementationSourceList Java system property. This property should contain a white space separated list of DOMImplementationSource classes on the local system. This interface is summarized in Example 10.3.

Example 10.3. The DOMImplementationSource interface

```java
package org.w3c.dom;

public interface DOMImplementationSource {

    public DOMImplementation getDOMImplementation(String features);
}
```

DOMImplementationRegistry.getDOMImplementation() queries each source for its DOMImplementation. The double indirection (listing DOMImplementationSource classes rather than DOMImplementation classes) is necessary to allow DOMImplementationRegistry.getDOMImplementation() to return different classes of objects depending on which combination of features are requested.

The three exceptions that getDOMImplementation() throws—ClassNotFoundException, InstantiationException, and IllegalAccessException—shouldn’t be very common. The only way any of these can be thrown is if the org.w3c.dom.DOMImplementationSourceList system property includes the name of a class that can’t be found or one that is not a conforming instance of DOMImplementationSource.

However, like the other DOM Level 3 material discussed in this book, all of this is on the wrong side of the bleeding edge and cannot be expected to work in most existing implementations.
How to Use This Book

This book is organized as an advanced tutorial that can also serve as a solid and comprehensive reference. The first chapter covers the bare minimum material needed to start working with XML, though for the most part this is intended more as a review for readers who’ve already read other, more basic books than as a comprehensive introduction. The second chapter introduces RSS, XML-RPC, and SOAP, the XML applications we’ll be using for examples in the rest of the book. This is followed by two chapters on generating XML from your own programs (a subject which is all too often presented as a lot more complicated than it actually is). The first covers generating XML directly from code. The second covers converting legacy data in other formats to XML. The remaining bulk of the book is devoted to the major APIs for processing XML:

- The event based SAX API
- The tree-based DOM API
- The tree-based JDOM API
- XPath APIs for searching XML documents
- The TrAX API for XSLT processing

Finally, the book finishes with an appendix providing quick references to the main APIs.

If you have limited experience with XML, I suggest you read at least the first five chapters in order. From that point forward, if you have a particular API preference, you may begin with the part covering the major API you’re interested in:

- Chapters 6-8 cover SAX
- Chapters 9-13 cover DOM
- Chapters 14 and 15 cover JDOM

Once you’re comfortable with one or more of these APIs, you can read Chapters 16 and 17 on XPath and XSLT. However, those APIs and chapters do require some knowledge of at least one of the three major APIs.
The entire book is available online in plain vanilla HTML at my Cafe con Leche web site. You can find it at http://www.cafeconleche.org/books/xmljava/. Every word of this book is there. Nothing has been held back or left out. I do hope you’ll also find the printed book useful and choose to buy it—It’s certainly cheaper than the paper and toner you’d use up printing out all 1000+ pages from your laser printer—but you are by no means obligated to do so. My goal is to make this material as broadly available and useful as possible.

The online version has no protection other than copyright law and your own good will. You don’t need to register to read it, or download some special electronic key that becomes invalid when you buy a new laptop, and that probably won’t run on Linux or a Mac in the first place. I want people to read and use this book. I do not want to put up silly roadblocks that make it less useful than it could be. I will ask that as a courtesy you do not republish the online edition on your own server. Doing so makes it extremely difficult for me to keep the book up-to-date. If you want to save a few pages on your laptop so you can read this book on an airplane, I don’t really mind. However, please don’t pass out your own copies to anyone else. Instead refer your friends and colleagues to the web site or the printed book.
Some grammatical notes

The rules of English grammar were laid down, written in stone, and encoded in the DNA of grammar school teachers long before computers were invented. Unfortunately, this means that sometimes I have to decide between syntactically correct code and syntactically correct English. When I’m forced to do so, English normally loses. This means that sometimes a punctuation mark appears outside a quotation mark when you’d normally expect it to appear inside, a sentence begins with a lower case letter, or something similarly unsettling occurs. For the most part, I’ve tried to use various typefaces to make the offending phrase less jarring. In particular,

- *Italicized text* is used for emphasis, the titles of books and other cited works, words in languages other than English, words used to refer to the words themselves (for example, *BooBooisie* is a very funny word.), the first occurrence of an important term, Java system properties, host names, resolvable URLs.
- *Monospaced text* is used for XML and Java source code, namespace URLs, system prompts, and program output.
- *Italicized monospaced text* is used for pieces of XML and Java source code that should be replaced by some other text.
- *Bold text* is used for emphasis.
- *Bold monospaced text* is normally used for literal text the user types at a command line, as well as for emphasis in code.

It’s not just English grammar that gets a little squeezed either. The necessities of fitting code onto a printed page rather than a computer screen have occasionally caused me to deviate from the ideal Java coding conventions. The worst problem is line length. I can only fit 65 characters across the page in a line of code. To try and make maximum use of this space, I indent each block by two spaces and indent line continuations by one space, rather than the customary four spaces and two spaces respectively. Even so, I still have to break lines where I’d otherwise prefer not to. For example, I originally wrote this line of code for *Chapter 4*:

```
result.append("<Amount>" + amount + "</Amount>
               ");
```

However, to fit it on the page I had to split it into two pieces like this:

```
result.append("<Amount>");
result.append(amount + "</Amount>
               ");
```
Some grammatical notes

This case isn’t too bad, but sometimes even this isn’t enough and I have to remove indents from the front of the line that would otherwise be present. For example, this occasionally forces the indentation not to line up as prettily as it otherwise might, as in this example from Chapter 3:

```java
wout.write("xmlns='http://namespaces.cafeconleche.org/xmljava/ch3/'r\n")
```

The silver lining to this cloud is that sometimes the extra attention code gets when I’m trying to cut down its size results in better code. For example, in Chapter 4 I found I needed to remove a few characters from this line:

```java
OutputStreamWriter wout = new OutputStreamWriter(out, "UTF8");
```

On reflection I realized that nowhere did the program actually need to know that `wout` was an `OutputStreamWriter` as opposed to merely a `Writer`. Thus I could easily rewrite the offending line as follows:

```java
Writer wout = new OutputStreamWriter(out, "UTF8");
```

This follows the general object oriented principle of using the least specific type that will suit. This polymorphism makes the code more flexible in the future should I find a need to swap in a different kind of `Writer`. 
Contacting the Author

I always enjoy hearing from readers whether with general comments, specific ways I could improve the book, or questions related to the book’s subject matter. Since this book is being published in its entirety online, it is possible for me to reprint at least the online edition much faster than can be done with a traditional paper book. Thus corrections and errata are especially helpful since there’s a real chance for me to fix them. Before sending in a correction, please do check the online edition to see if I already know about and have fixed the problem.

Please send all comments, inquiries, bouquets, and brickbats to elharo@metalab.unc.edu. I get a lot of e-mail, so I can’t promise to answer them all; but I do try. It’s helpful if you use a subject line that clearly identifies yourself as a reader of this book. Otherwise, your message may get accidentally misidentified as spam I don’t want or bulk mail I don’t have time to read and get dropped in the bit bucket before I see it. Also, please make absolutely sure that your message uses the correct Reply-to address and that the address will be valid for at least several months after you send the message. There’s nothing quite so annoying as taking an hour or more to compose a detailed response to an interesting question, only to have it bounce because the reader sent the e-mail from a public terminal or changed their ISP. But please do write to me. I want to hear from you.

--
Elliotte Rusty Harold
Brooklyn, New York
June 7, 2002
Acknowledgements

Thomas Marlin provided me with the original Latin text of the Fibonacci problem you’ll find in Chapter 3.

Jason Hunter’s encyclopedic knowledge of the Java Servlet API was essential to the design and execution of the servlet code in this book. Donald Sizemore helped me get my servlets installed and running on IBiblio.

Luke Tymowski provided some of the RSS examples and helped me debug various problems with my Cobalt Qube.

Bruce Eckel and Chuck Allison helped me decipher the relative capabilities of Java and C++. Bruce Eckel also helped out with Python. Matt Sergeant and Brendan McKenna helped out with Perl. Philip Nelson, Robert A. Casola, and Rob Smith helped with Visual Basic. None of these people necessarily agree with what I wrote about those relative capabilities; (In fact, more often than not they vehemently disagree; de linguis non disputandum est); but I couldn’t have done it without them.

Although this is the sixth book I’ve written about XML, it is the first one I’ve written in XML. That could not have happened without Norm Walsh’s Docbook DTD and XSL stylesheets for Docbook.


Mike Champion, Andy Clark, Robert W. Husted, Anne T. Manes, Ron Weber, and John Wegis did yeomanlike service as technical reviewers. Their comments substantially improved the book.

As always, the folks at the Studio B literary agency were extremely helpful at all steps of the process. David Rogelberg, Sharon Rogelberg, and Stacey Barone should be called out for particular commendation.
This is my first book for Addison-Wesley, but it’s not going to be my last. They were all wonderful people to work with, and I look forward to working with them again. Mary T. O’Brien shepherded this book from contract to completion. Alicia Carey ably managed submissions and communications.

Finally, as always, my biggest thanks are due to my wife Beth without whose love and understanding this book could never have been completed.
XML was designed to be “SGML for the Web”. It was meant for the same sorts of narrative documents SGML and HTML had been used for previously: articles, books, short stories, poems, technical manuals, web pages, and so forth. Much to its inventors’ surprise, it achieved its first great successes not
Chapter 1. XML for Data

in the publishing and writing arenas it was intended for, but rather in the much more prosaic world of data formats. XML was enthusiastically adopted by programmers who needed a robust, extensible, standard format for data. For the most part, this was not narrative data like stories and articles, but record oriented data such as that found in databases. Uses included object serialization, financial records, vector graphics, remote procedure calls, and similar tasks. This chapter explores some of the flaws in traditional formats for such data and elucidates the features of XML that make it surprisingly well-suited for such tasks.

Motivating XML

If you’re reading this book you’re a developer. (At least I hope you are. Otherwise a lot of what I say isn’t going to make any sense :-) ) Doubtless over the course of your career you’ve written numerous programs that read and write files. And every time you wrote a new program you had to invent or learn a new file format. File formats I’ve personally had to deal with over the years include RTF, Word .doc files, tab delimited text, FITS, PDF, PostScript, and many more. You’ve probably encountered a few of these yourself. Doubtless, you’ve also seen many other formats.

If you’re like me you’ve learned to dread encountering a new file format. If it’s documented at all, the documentation is likely incomplete or worse yet misleading. Important details like byte order and line ending conventions are often left unspecified. Different tools that all claim to read and write the same format actually produce subtly different variants that are often incompatible in practice. When you think you’ve finally wrestled the last bug out of your code, you discover a file written by somebody else’s software that you can’t read; and you realize you’ve made one too many assumptions about the format, so you have to go back to the drawing board.

Consequently, when designing new file formats, developers have tended to gravitate toward the simplest formats they can imagine, often tab delimited text or comma separated values. Nonetheless, even these plain, undecorated formats often present unexpected problems. For example, should two tabs in a row be interpreted as the empty string, null, or the same as one tab? In fact, all three variations are used in practice. Java’s StringTokenizer class takes the last interpretation, two consecutive tabs are the same as one tab, even though this is the least common approach in actual data files, a fact which has surprised many Java programmers and led to not a few bugs in Java programs.[1]

A Thought Experiment

With all that in mind, let’s do a thought experiment. Imagine you’ve been tasked with writing a server side program that accepts orders over the Internet for an e-commerce site. The web server must send each completed order to the internal system, one order at a time. You’re responsible for writing the code on the server that sends the order to the internal system and for writing the code on the internal system that receives and processes the order. The only connection between the two systems is a TCP/IP
network; that is, you don’t have some sort of higher level API like JDBC that lets you move data
between the two systems. You need to invent a data format you can generate on one end and parse on
the other end that’s flexible enough to contain all the information in a typical order. This includes the
customer name, the product ordered, its price, the manufacturer’s stock keeping unit (SKU) number, the
address to ship to, the tax, and the shipping and handling charges. One possibility is to place each piece
of information on a separate line as shown in Example 1.1:

**Example 1.1. A plain text document indicating an order for 12 Birdsong Clocks, SKU 244**

c32
Chez Fred
Birdsong Clock
244
12
USD
21.95
135 Airline Highway
Narragansett
RI
02882
USD
263.40
7.0
USD
18.44
USPS
USD
8.95
USD
290.79

An alternative is to use a more complex and verbose XML format such as Example 1.2:

**Example 1.2. An XML document indicating an order for 12 Birdsong Clocks, SKU 244**

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order>
  <Customer id="c32">Chez Fred</Customer>
  <Product>
    <Name>Birdsong Clock</Name>
    <SKU>244</SKU>
    <Quantity>12</Quantity>
  </Product>
</Order>
```

http://www.cafeconleche.org/books/xmljava/chapters/ch01.html (3 of 10)2006-07-07 오후 2:06:35
Would you rather write the code to send and receive orders that are formatted as nice, simple linefeed delimited files as shown in **Example 1.1** or as complex, marked up XML documents such as **Example 1.2**? Both documents contain the same information. Most uninitiated developers prefer the first, simpler form. After all each piece of information is presented on a line by itself with no extraneous markup characters getting in the way. It’s my goal to convince you that contrary to most developers’ first intuition the second form is more robust, more extensible, and much easier to work with.

**Robustness**

Let’s consider robustness first. Suppose your program receives the order in **Example 1.3**:

**Example 1.3. A document indicating an order for 12 Birdsong Clocks, SKU 244?**

c32
Chez Fred
Birdsong Clock
12
244
USD
21.95
135 Airline Highway
Narragansett
RI
02882
USD
263.40
7.0
USD
18.44
Look’s the same as Example 1.1 doesn’t it? However, if you compare it very carefully with Example 1.3 you may notice that the 12 and the 244 have changed places. What used to be an order for 12 bird clocks may now be an order for 244 whoopee cushions. Maybe somebody will notice the problem before the order is shipped and maybe they won’t. Worse yet, the shipping charge and the total price got flipped around. This entire order now costs eight dollars and ninety-five cents. Again, maybe someone will notice the problem before it’s too late and maybe not. These sorts of problems aren’t theoretical. More than one e-commerce site has lost both revenue and customer goodwill by mispricing items.

In the XML version, this simply would not be an issue because each datum is marked up with what it means. You can freely reorder the quantity and the SKU or the shipping cost and the total price without any confusion about which is which. Example 1.4 demonstrates. What can be devastating mistakes in a traditional system are harmless in XML.

**Example 1.4. Still an order for 12 Birdsong Clocks, SKU 244**

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order>
  <Customer id="c32">Chez Fred</Customer>
  <Product>
    <Name>Birdsong Clock</Name>
    <Quantity>12</Quantity>
    <SKU>244</SKU>
    <Price currency="USD">21.95</Price>
  </Product>
  <ShipTo>
    <Street>135 Airline Highway</Street>
    <City>Narragansett</City> <State>RI</State> <Zip>02882</Zip>
  </ShipTo>
  <Subtotal currency='USD'>263.40</Subtotal>
  <Tax rate="7.0" currency='USD'>18.44</Tax>
  <Total currency='USD'>290.79</Total>
  <Shipping method="USPS" currency='USD'>8.95</Shipping>
</Order>
```

Some readers will be objecting at this point that you would never let a mistake like that through your
Chapter 1. XML for Data

system. After all you check every value for sensibility. You look up the SKU in the company database to make sure it matches the product name and price before completing an order. You check every return value from a method call to see if it’s null and you catch every exception. You write extensive tests to verify that each method is doing what you think it’s doing. You use a source code control system so you can always back out changes, and you never check code in until it’s passed all the regression tests. Every line of code is scrupulously documented. In fact, you write more documentation than actual code. And you’ve never, ever missed church on Sunday. In this case your name is Donald Knuth. The rest of us need a little more help making sure we don’t do something stupid.

Even if you are that conscientious, are you really willing to gamble on everyone else who sends or receives data from you being equally anal retentive? Wouldn’t it make more sense to use the most robust format possible so that when the inevitable errors do creep in, they’ll do less damage?

Of course, XML has a lot to offer the anal developer as well. When defining constraints such as “Every order must have a shipping address”, “the currency must be one of the three letter codes USD, CAN, or GBP” or “the total cost must be the sum of the unit price times the number of items, the tax, and the shipping”, it’s easiest to use a declarative language that specifies what the constraints are without elaborating the actual code to check these constraints. When your data is XML, you can use a declarative schema language to define and test such constraints. Indeed, you have a choice of several schema languages. The simplest and most broadly supported, the classic document type definition (DTD), allows you to verify that all required elements are present in the required order with any necessary attributes. The W3C XML schema language goes further and lets you constrain the contents of particular elements and attributes so that you can guarantee that the total price is a decimal number greater than 1.00. Schematron, the most powerful schema language of all, allows you to state multi-element constraints such as “the actual price must be less than or equal to the suggested retail price”. I’ll discuss all of these languages in more detail later in this chapter and the rest of the book. For now what you need to know is that you can list all the constraints on a document in a simple fashion and check those constraints without writing a lot of extra code to do so. You feed your documents through a validator before you act on them. Validation becomes a separate, modular and more maintainable part of the process. You can even change constraints or add new ones without recompiling your code.

Extensibility

Robustness isn’t the only advantage of the XML approach. The XML solution is also far more extensible. For example, suppose you suddenly discover a need to add a discount percentage to some products. The change to the XML is straightforward. Just add an extra element:

```
<Product>
  <Name>Birdsong Clock</Name>
  <Quantity>12</Quantity>
  <SKU>244</SKU>
  <Price currency="USD">21.95</Price>
</Product>
```
The change to the plain text file (or the equivalent binary file) is much less obvious. You can certainly add an extra line of data. However, then everything that follows it will be out of order. You could put the new information at the end of the document, but then it isn’t close to the item it logically belongs with. And suppose not all orders have discounts. Will there be blank lines for products that don’t have discounts? How will your program recognize that it’s supposed to convert an empty string into a zero discount rather than NaN or throwing an exception? This is not an insurmountable problem, but the simple solution is becoming more complex.

Now suppose someone wants to add a gift message field whose value can contain line breaks. Now the data can contain the delimiter character! You can probably escape the line breaks as \n or some such, and then escape the backslash character as \, but your nice simple solution is becoming quite a bit more complex. However, once again this is not a problem for XML as this solution demonstrates:

```
<GiftMessage>
  Happy Birthday Monica!

  Love Always,
  Tracy
</GiftMessage>
```

Throughout this example, I’ve assumed that each order is for exactly one product. That’s probably not true. Some customers will order multiple products at a time. Thus each order will contain between one and an indefinite number of products. Different products may even be going to different addresses. Do you break each individual item into a separate order document and repeat the customer information? If so how do you calculate the total shipping and total cost? Or do you allow multiple products in a single order? If so how do you tell where one product ends and the next begins? Again, none of these problems are unsolvable, but the simple solution proves more and more complex as the needs grow. The XML approach, by contrast, scales very well to expanded functionality in a very obvious way. Example 1.5 is an XML document that accomplishes all of the above. The boundaries between the individual parts are obvious.

Example 1.5. An XML document indicating an order for multiple products shipped to multiple addresses

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order>
  <Customer id="c32">Chez Fred</Customer>
  <Product>
    <Name>Birdsong Clock</Name>
  </Product>
```

http://www.cafeconleche.org/books/xmljava/chapters/ch01.html (7 of 10)2006-07-07 오후 2:06:35
This example still isn’t really complete. Many pieces are missing including the credit card information, billing address, and more. Real world examples are larger and more complex than can comfortably fit in a book. Adding these other parts would only stretch the flat format further and make the advantages of XML still more obvious. The more complex your data is, the more important it is to use a hierarchical format like XML rather than a flat format like tab or line-delimited text.

**Ease of Use**

Now here’s the real kicker: not only is the XML document far more robust. Not only is it much more extensible in the face of both expected and unexpected changes. Not only does it more easily adapt to
more complex structures. It is also easier for your programs to read! Writing a program to accept orders written in XML will be many times easier than writing a program to accept orders delivered in simple line delimited files. “How can that be?” you may be asking. After all, the program reading the XML document has to hunt for less than signs and quotation marks rather than just picking each piece of data off of a line. It has to make sure not to confuse any less than signs and quotation marks that may appear in the data itself with those in the markup. It has to deal with data that may extend across multiple lines. And in fact, there are many more possibilities not evident in this simple example that a real program has to handle.

Fortunately **none of this matters to you as a developer because you don’t have to do any of it.** Instead of writing the code to process XML documents directly, you let an XML parser do the hard work for you. A parser is a software library that knows how to read XML documents and handle all the markup it finds. The parser takes responsibility for checking documents for well-formedness and validity. Your own code reads the XML document only through the parser’s API. At this level, you can simply ask the parser to tell you what it saw in any particular element. Or you can ask the parser to tell you everything it sees as soon as it sees it. In either case, the parser just gives you the data after resolving all the markup. For instance, if you want to ask the parser what the total price was, it can tell you 290.79 and that this price has the currency USD. You don’t have to concern yourself with stripping off the markup around the information you want. Nor do you necessarily have to take the information in the order it appears in the input document. If you want the total price before the customer name, you can have it. If you just want to look at the price and ignore the rest of the order completely, you can do that too. You take the information in the form that’s convenient to you without worrying excessively about low level serialization details.

**Note**

One of the original ten goals for XML was that “It shall be easy to write programs which process XML documents.” Originally, this was interpreted as meaning that a “Desperate Perl Hacker” could write an XML parser in a weekend. Later it became clear that XML was simply too complex, even in its simplest form, for this goal to be met. However, the understanding of this requirement changed to mean that a typical programmer could use any of a number of free tools and libraries to process XML. Given this interpretation, the goal has most certainly been met.

The parser shields you from a lot of irrelevant details that you don’t really care about. These include:

- How text is encoded: in Unicode, ASCII, Latin-1, SJIS, or something else
- Whether carriage returns, line feeds, or both separate lines
- How reserved characters such as `<` are escaped when used in the plain text parts of the document
- Whether the byte order is big-endian or little-endian

None of these issues actually matter. None of them have any effect on what the data means or what the
format allows you to say. However, when designing a data format, you must answer all these questions. As soon as you’ve said, “The underlying format of the data is XML”, every one of these questions is answered. Some are answered by simply choosing one possible solution. (The less than sign is escaped as &lt;.) Others are answered by allowing all possibilities and letting the parser sort things out (line endings). In all cases, the design problem is greatly simplified by picking XML as the underlying format.

[1] This interpretation makes sense once you realize that java.util.StringTokenizer is designed for parsing Java source code, not for reading tab delimited data files. Nonetheless many programmers do use it for reading tab delimited data.
XML Syntax

This is not an introductory book about XML. I certainly expect that you have some experience with XML documents before now. Nonetheless, when writing programs to process XML it’s even more important to make sure that you are totally crystal clear about the exact terminology used when discussing XML. Therefore I’d like to take a few pages to briefly review the proper terminology for discussing XML, as well as to clarify a few points that are often confused or misunderstood.

XML Documents

The precise meaning of “XML document” is defined by the XML 1.0 specification published by the Worldwide Web Consortium (W3C). This specification provides a detailed BNF grammar defining exactly what is and is not an XML document. Anything that satisfies the document production in that BNF grammar and adheres to the fifteen well-formedness constraints is an XML document. Anything that does not is not an XML document.

Well-formedness is the minimum requirement for an XML document. A document that is not well-formed is not an XML document. Parsers cannot read it. A parser is not allowed to fix a malformed document. It cannot take a best-guess at what the document author intended. When a parser encounters a malformed document, it stops parsing and reports the error. It will not read any further in the document. Depending on which API you’re accessing the parser through, you may or may not have already received some information from the parts of the document before the error. However, under no circumstances will the parser give you any data from after the first well-formedness error in the document.

The detailed rules an XML document must follow aren’t so important here since the parser will check them for you. Very roughly an XML document must have a single root element. All start-tags must be matched by end-tags. All attribute values must be quoted. And only the Unicode characters that are legal in XML may be used in the document. (Almost all Unicode characters are legal in XML documents. The only ones really ruled out are the C0 controls like null, bell, and form feed.)

Note

Occasionally developers ask how they can parse a document that is almost, but not quite a well-formed XML document. For example, it may end with a form feed inserted by some Unix text editor to separate documents. Or perhaps it’s part of an
infinite stream of elements, the last of which is never seen so there’s no end-tag for the root element. Imagine, for example, weather observations or stock quotes being pushed across the Internet as XML elements.

The short answer is that you can’t parse these things because they are not XML documents, even if they use a lot of tags and attributes and other XML-like markup. The long answer is that you may be able to write a non-XML-aware program to preprocess the streams, fix up any well-formedness mistakes you see, and only then pass the fixed documents to the XML parser. However, the XML parser must receive a complete well-formed document. It cannot work with anything less.

There’s another way to look at XML documents besides simply as a sequence of characters that adheres to certain rules, and it’s one that sometimes makes sense, especially when writing programs that process XML documents. An XML document is a tree. It has a root node that contains various child nodes. Some of these child nodes have children of their own. Others are leaf nodes that have no children.

There are roughly five different kinds of nodes in an XML tree:

- **Root**
  - Also known as the document node, this is the abstract node that contains the entire XML document. Its children include comments, processing instructions, and the root element of the document.

- **Element**
  - An XML element with a name, a list of attributes, a list of in-scope namespaces, and a list of children.

- **Text**
  - The parsed character data between two tags (or any other kind of non-text node).

- **Comment**
  - An XML comment such as <!-- This needs to be fixed. -->. The contents of the comment are its data. A comment does not have any children.

- **Processing Instruction**
  - A processing instruction such as <?xml-stylesheet type="text/css" href="order.css"?> A processing instruction has a target and a value. It does not have any children.

Depending on context, some details of this tree structure can be understood differently. For example, some tree models consider parsed entities or CDATA sections to be additional kinds of nodes. Others
simply merge them into the tree structure as elements and text nodes. Some models allow one text node to follow another. Others require each text node to be the maximum contiguous run of text not interrupted by some other kind of node. Some models include the document type declaration and/or the XML declaration as a node. Others ignore them. Probably the most hotly debated point is how to handle attributes and namespaces. I chose to not consider them as nodes in the tree in their own right, treating them instead as properties of elements. Generally even those tree models such as XPath that do treat them as separate nodes still don’t make them children of the element they belong to. For now the details aren’t too important. The broad outline is the same for pretty much all the tree models.

Caution

There’s some argument about whether it really makes sense to talk about an XML document as having any independent existence separate from the text that makes up the document. After all, the XML 1.0 specification only defines concepts like document and element in terms of text strings. Later W3C specifications like the XML Information Set (Infoset) and the Document Object Model (DOM) do suggest a more abstract understanding of the components of an XML document. However, these specifications are much more controversial than XML 1.0 itself, and not as broadly implemented or accepted. For the purposes of writing programs that process XML, I do find it useful to consider XML documents more abstractly; and I will do so in this book. However, even here there’s a split depending on which API you choose. DOM is a very abstract model of XML documents that defines classes representing elements, attributes, comments, and more. SAX defines almost no such classes, however. It presents the content of an XML document almost exclusively as strings and arrays of characters.

XML Applications

An XML application is a specific XML vocabulary that contains particular elements and attributes. It is not a software program that somehow uses XML like the EditML Pro XML editor or the Mozilla web browser. XML applications limit the very flexible rules of XML to a finite set of elements of certain types. For example, DocBook is an XML application designed for technical manuscripts such as the book you’re reading now. Elements it defines include book, chapter, para, sect1, sect2, programlisting, and several hundred others. When writing a DocBook document, you have to use these elements; and you have to use them in certain ways. For instance, a sect2 element can be a child of a sect1 but not a child of a sect3 or a chapter. Scalable Vector Graphics (SVG) is an XML application for line art. Elements it defines include line, circle, ellipse, polygon, polyline, and so forth. All SVG documents are XML documents, but not all XML documents are SVG documents.

An XML application can have a schema that defines what is and is not a legal document for that application. Schemas can be written in a variety of languages including Document Type Definitions
(DTDs), the W3C XML Schema Language, RELAX NG, Schematron, and numerous others. Depending on the power of the schema language used, it may also be necessary to specify additional rules for the application in less-formal prose. For example, the XHTML 1.1 specification includes the requirement that “There must be a DOCTYPE declaration in the document prior to the root element. If present, the public identifier included in the DOCTYPE declaration must reference the DTD found in Appendix C using its Formal Public Identifier.” None of the common schema languages allow you to require anything about the DOCTYPE declaration.

An instance document is an instance of an XML application, whether formally defined or not. That is, it is an XML document with a root element and whatever other content it possesses that satisfies all the rules of some XML application. There are many possible instance documents for any one XML application, just as there are many programs that can be written in any one programming language.

Elements and Tags

The fundamental unit of XML is the element. You can write good XML documents without using any other XML construct. If for some reason you have a grudge against comments, processing instructions, attributes, or namespaces, you can pretend they don’t exist and still write well-formed XML documents. However, you must use elements. Every XML document has at least one element. You cannot write XML documents without using elements.

Logically every element has four key pieces:

- A name
- The attributes of the element
- The namespaces in scope on the element
- The content of the element

In addition, once schemas become more prevalent and parsers and APIs are revised to support them, it may also make sense to talk about the element’s type. For now, though, there’s not a lot of practical help to be gained by considering the type.

Furthermore, DOM and XPath also have mutually incompatible concepts of the value of an element. However, in both cases, the value is derived purely from the element content, so it’s not really a separate thing.

Syntactically, in the text form of an XML document, elements are delimited by tags. Start-tags begin with a < immediately followed by the element name. End-tags begin with a </ immediately followed by the element name. Both start and end-tags terminate with >. Everything in between the two tags is the content of the element. For example, this is a Quantity element with the content “12”:

<Quantity>12</Quantity>
XML Syntax

Tags and elements are closely related, but they are not the same thing. Be wary of books that confuse them. An element is the whole sandwich including bread, meat, cheese, pickles, and mayonnaise, while the tags are just the bread. An element is composed of a start-tag, followed by content, followed by an end-tag.

It is possible that an element may have no content. In this case it is called an empty element. For example, this is an empty Quantity element:

```
<Quantity></Quantity>
```

The start-tag butts right up against the end-tag. There is not even a single space character between them. By contrast, this next element is not empty because it does contain some white space, even if it doesn’t contain anything else:

```
<Quantity> </Quantity>
```

Besides start-tags and end-tags, there is one other kind of tag, the empty-element tag. An empty-element tag begins with a `<` followed by an element name like a start-tag. However, it ends with a `/>`. For example, this is an empty Quantity tag:

```
<Quantity/>
```

This tag both starts and ends a Quantity element. The content of this element is nothing, just like the content of `<Quantity></Quantity>`. Indeed `<Quantity/>` is just syntax sugar for `<Quantity></Quantity>`. They mean exactly the same thing. No application should treat these two constructs as different in any way. Indeed, most XML parsers and APIs won’t even tell you which form the element took in the source document. In both cases, what’s reported is an empty element with the name “Quantity”. How that element was represented is not important.

As well as text, an element can also contain one or more child elements. These are elements that are completely contained between the element’s start-tag and end-tag, and are not contained inside any other element also contained in the parent element. For example, this ShipTo element has four child elements: Street, City, State, and Zip:

```
<ShipTo>
  <Street>135 Airline Highway</Street>
  <City>Narragansett</City> <State>RI</State> <Zip>02882</Zip>
</ShipTo>
```

In addition to the four child elements, this ShipTo element also contains some white space; for example, the single space character between `</City>` and `<State>`. These spaces form text nodes
that are also counted among the element’s children. Text nodes like these that are composed of nothing but white space are sometimes called *ignorable white space*. This is an unfortunate turn of phrase. Sometimes you can ignore these nodes, but most of the time you can’t. The more proper term is *white space in element content*.[4]

All the elements contained inside an element are called the element’s *descendants*. Only the highest level are the children. The descendants include not only the children, but the children of the children, the children of the children’s children, and so forth. If you look at Example 1.2 again, you’ll see that the `Order` element has 15 descendant elements.

An element can also have *mixed content*. This is when an element contains both child elements and text nodes containing non-whitespace characters. For example, this variant `ShipTo` element has both the child elements you saw before as well as text nodes containing the strings “Chez Fred” and “Apt. 17D”:

```xml
<ShipTo>
  Chez Fred
  <Street>135 Airline Highway</Street>
  Apt. 17D
  <City>Narragansett</City> <State>RI</State> <Zip>02882</Zip>
</ShipTo>
```

Mixed content is very useful, indeed almost essential, for XML applications that contain narratives such as books and stories. Such applications include XHTML, DocBook, TEI, and XSL Formatting Objects. Mixed content is much less useful and much more cumbersome for data-oriented applications. XML documents that are intended for computers to read, as opposed to XML documents that are intended for humans to read, should use mixed content sparingly, if at all.

**Text**

XML documents are text. Each XML document is a sequence of characters. These characters are taken from the Unicode character set. However, XML documents can be written in any character set which your XML parser knows how to convert to Unicode, providing that it is properly specified in the document’s encoding declaration in the XML declaration.

**Caution**

Many developers have decided that they can make XML more efficient by defining a binary version. This tends to be based on some vague notion that binary formats are inherently smaller or faster than text formats. These developers rarely have any actual evidence to back up this claim, which is not surprising since it isn’t true. XML documents are routinely smaller and faster to read than the equivalent binary files.
in standard applications like Oracle, Microsoft Word, Microsoft Excel, and so forth. The fact is modern binary file formats are quite bloated, but disks have gotten so large that almost no one’s noticed or cared. Nonetheless, there seems to be a large pool of programmers who mistakenly believe:

1. File size matters.
2. They can compress better than gzip.
3. Human legible/human editable data doesn’t matter.

All three beliefs have been empirically proven false time and time again. Nonetheless, about once a month some developer somewhere announces that they’ve come up with yet another special purpose binary compression format for XML. These have proven completely pointless in practice. There is no actual benefit to these formats, and no one needs one. Worse yet, such a format substantially eliminates many of the existing benefits of XML.

Unicode is a character set with room for over one million different characters, though currently (Unicode 3.2) a few less than 100,000 are defined. Scripts covered by Unicode include Latin, Cyrillic, Greek, Hebrew, Arabic, Devanagari, the Han ideographs, and many more.

**Caution**

Contrary to what you may have heard, Unicode is not a two-byte character set and really never has been. Since there are more than a million different spaces for characters in Unicode, an arbitrary Unicode character cannot be represented by a single two-byte unsigned integer such as Java’s char data type. Prior to Unicode 3.1 all defined Unicode characters had code points less than 65,536, which fooled some developers into thinking they could get away with using two-byte chars. However, it’s long been known that more than 65,536 characters are actually used on Earth today and that Unicode would have to assign characters outside the Basic Multilingual Plane (the first 65,535 characters) to accommodate them. Although characters were not actually assigned code points greater than 65,536 until Unicode 3.1, the space for them has long been reserved. XML was designed by forward-thinkers who saw the problems ahead, and prepared for the eventual expansion of Unicode. Consequently XML documents can use the full range of all million-plus characters available in Unicode. Java’s designers weren’t as prescient though, and restricted the char data type to two-bytes. Consequently Java programmers need to go through some pretty nasty gyrations to adequately handle Unicode documents (including XML documents).

With a very few exceptions any character defined in Unicode can be used in the text content of an element or the value of an attribute. In brief, the exceptions are:
The C0 controls

The *non-printing characters* such as null and formfeed, between code points 0 and 31 (decimal). The carriage return, linefeed, and the horizontal tab are allowed.

The surrogate blocks

The *surrogate blocks* are two sets of 1024 code points each, which are used to extend Unicode beyond the Basic Multilingual Plane by allowing some characters to be represented as two surrogate characters. You can include surrogate pairs in an XML document in an encoding like UTF-16 that uses surrogate pairs. You just can’t treat an individual half of a surrogate pair as a character by itself.

The byte order mark

The *byte order mark*, also known as the zero-width non-breaking space, can be used at the beginning of a document to indicate the encoding and endianness of the document, but cannot be used elsewhere in the document.

All other characters are fair game, including some you probably shouldn’t be using anyway such as characters in the private use area and compatibility characters Unicode offers purely for interoperability with existing character sets.

The rules for characters used in the names of things (elements, attributes, entities, etc.) are a little stricter. In brief, only letters, digits, and ideographs defined in Unicode 2.0 can be used. In addition the punctuation marks -, ., _, and : are also legal. Digits, the hyphen, and the period cannot be the first character in a name. Other punctuation marks as well as new characters first defined in Unicode 3.0 and later are not allowed anywhere in a name. These are essentially the same rules used for naming variables, methods, and classes in Java. The major difference is that XML allows the hyphen and Java doesn’t (it’s reserved for the minus sign) while Java allows the dollar sign and XML doesn’t. XML also allows the colon, unlike Java. However, XML reserves this for use with namespaces. It should not be used as an arbitrary name character.

XML parsers faithfully preserve white space. A string containing only white space is not the same as a string containing nothing at all. A string with leading and trailing white space is not the same as the equivalent string with white space trimmed. Some specific XML applications may decide that white space is not significant in certain contexts. However, in generic XML all white space is significant and must be accounted for.

**Attributes**

Attributes are name value pairs associated with elements. The name of an attribute may be any legal XML name. The value may be any string of text, even potentially including characters like < and ". The
document author needs to escape such characters as &lt; and &quot;. However, the parser will resolve these references before passing the data to your application. The attribute value is enclosed in either single or double quotes, and the name is separated from the value by an equals sign. For example, this Subtotal element has a currency attribute with the value USD:

```xml
<Subtotal currency='USD'>393.85</Subtotal>
```

The quote marks are not part of the attribute value. Whether single or double quotes are used or whether there’s extra white space around the equals sign is not important. Most parsers don’t bother to report the difference. These two elements are also the same as the previous one:

```xml
<Subtotal currency="USD">393.85</Subtotal>
<Subtotal currency = "USD">393.85</Subtotal>
```

Attributes are unordered. There is no difference between these two elements:

```xml
<Tax rate="7.0" currency="USD">27.57</Tax>
<Tax currency="USD" rate="7.0">27.57</Tax>
```

When a parser tells you which attributes are attached to an element, it may or may not provide them in the same order they had in the input document. Some APIs report the attributes using an unordered data structure like a hash table. Others use an array or a list, but even in these cases there’s no guarantee that the order of the attributes in the list matches the order of the attributes in the start-tag.

Perhaps most surprisingly, attribute values whose type is not CDATA are normalized. This means that all leading and trailing white space is stripped from the value, and runs of white space characters are compressed to a single space. This does not apply to any of the attributes in the examples seen so far because untyped attributes are not normalized. However, once you add a DTD it is possible to declare that an attribute has type ID, IDREF, IDREFS, NMTOKEN, and several other types. Attributes of these types are always normalized before being passed to the client application.

**Note**

Tim Bray, one of the primary authors of XML 1.0, has admitted that normalization of attribute values was a mistake. In his words, “Why the $#%%!@! should attribute values be ‘normalized’ anyhow? This was a pure process failure: at no point during the 18-month development cycle of XML 1.0 did anyone stand up and say ‘why are you doing this?’ I’d bet big bucks that if someone had, the silly thing would have died a well-deserved death.” [5]

**XML Declaration**
Most XML documents begin with an *XML declaration*. An XML declaration has a `version` attribute with the value 1.0 and may have optional `standalone` and `encoding` attributes. For example, this XML declaration says that the document is written in XML 1.0 in the ISO-8859-1 (Latin-1) character set and does not require the parser to read the external DTD subset:

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

The `version` attribute always has the value 1.0. If XML 1.0 is ever revised, this may change to some other value. As I write this, there’s a hotly debated proposal at the W3C for a new version of XML code named “Blueberry” which would make XML marginally more compatible with Unicode 3.0 and later as well as making it easier to edit with some brain damaged IBM mainframe software that can’t handle files where lines end in carriage returns, line feeds, or both. If this gets adopted (and I for one hope it doesn’t) this may lead to a new value for the `version` attribute. However, for now, `version` is effectively fixed with the value 1.0.

The `encoding` attribute identifies the character set and encoding in which the document is written. Whatever the encoding is, one of the jobs of the parser is to convert the document to Unicode before passing it to the client application. Most APIs don’t offer any means of finding out what the original encoding was. You’ll simply receive Unicode strings from which all traces of the original encoding have been removed.

The `standalone` attribute specifies whether the XML parser may have to read parts of the DTD that are outside the instance document to correctly parse the file. This is mostly a hint for the parser. Some parser APIs may tell you what the value was, but you generally don’t need to worry about it. The parser either will or won’t read external entities as necessary. By the time your code gets hold of the document, all of this will have already been taken care of. You need not concern yourself with it.

**Comments**

XML comments are almost exactly like HTML comments. They begin with `<!--` and end with `-->`. For example, here’s a comment you might find in an order document:

```xml
<!-- Please make sure this order goes out ASAP! -->
```

Everything between the `<!--` and the `-->` should be ignored. In fact, most parsers and APIs do make the comments available to you if you want them, mostly so you can *round trip* documents (read them in and then write them back out again with everything still intact). However, beyond this use case, you really shouldn’t pay much attention to comments in your programs. Some HTML systems abuse comments to support server side includes or editor specific extensions. Since XML is much more flexible than HTML, however, you can use elements, attributes, or, as a last resort, processing instructions for these use cases.
Processing Instructions

Processing instructions are used to tell particular software how it should handle an XML document after the document has been parsed. Generally, processing instructions are used for meta-information that may apply to documents from many different domains and XML vocabularies. For instance, the most common processing instruction, xml-stylesheet, tells a browser or other formatter where it can find the stylesheet it should apply to the document. This can be used with DocBook documents, XHTML documents, Human Resources Markup Language documents, or the custom XML application you invented last Tuesday to catalog your baseball card collection. For another example, the Apache XML Project’s Cocoon application server reads cocoon-process processing instructions to figure out what processes to apply to a document before sending it to a user. This processing instruction tells Cocoon to replace the XInclude include elements with the contents of the documents they reference:

```xml
<?cocoon-process type="xinclude"?>
```

The basic syntax of a processing instruction is <?, followed immediately by an XML name identifying the target of the processing instruction, followed by white space and any data at all, followed by ?>.

Unlike elements or attributes, processing instructions can be added to a document without considering whether or not the DTD or schema allows it. Most schema languages do not consider the presence, absence, or structure of processing instructions when determining validity. Furthermore, unlike elements, processing instructions can appear before, after, or inside the root element. They are frequently placed in the document prolog, though they can appear in the document body or after the root element as well.

Most of the time, the processing instruction is not associated with any one XML application. For instance, an XML application may describe gene sequences, 16th century Italian love poetry, financial records, or vector graphics. However, each of these might need to be loaded into a Web browser which would apply a stylesheet to it. Processing instructions can be inserted into a document to support this without changing or affecting the normal document structure. In essence, processing instructions provide an out-of-band channel for passing information to software other than the program that would normally read a document.

XML parsers report the target and contents of processing instructions to the client application. However, they provide no further support for interpreting the data in the processing instruction. For instance, many processing instructions use a pseudo-attribute format like this:

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

However, as far as the XML parser is concerned, the data in this processing instruction is just a string that happens to contain some equals signs and quotation marks. These are not treated differently than
XML Syntax

any other character.[6] Both the syntax and semantics of the data is completely up to the application reading the document. Processing instructions are specifically for information that is not related to XML.

Entities

XML documents are not necessarily the same thing as XML files. A single XML document may be composed of several different files. Indeed, the pieces that make up an XML document may not be files at all, but may instead be records in a database, data sent out over the Internet by a web server in response to a CGI query, a small part of a much larger file, or something stranger still.

The individual storage units that make up any one XML document are called entities. Every XML document has at least one entity, the document entity. This is the storage unit, be it a file or something else, that holds the root element of the document. Every other entity in a document has a name. There are five such kinds of named entities, and they are classified according to three criteria:

Internal or external

The replacement text of an internal entity is defined as a string literal in the document’s DTD. The replacement text of an external entity is read out of a different file located via a URL.

Parsed or unparsed

A parsed entity contains XML. It is itself well-formed, and may even be a complete XML document if it has a root element. (Some entities that are only intended to be used as parts of other documents do not have root elements). You can think of a parsed entity as something that will be pasted right into the middle of an XML document, such that the resulting document would still be well-formed.

An unparsed entity can contain anything at all, including binary data. Unparsed entities are not pasted (even metaphorically) into XML documents. Instead a URL to the entity’s data is provided in an ENTITY declaration in the DTD. Then this entity is referenced in an attribute with the type ENTITY or ENTITIES in the document. An unparsed entity also has a notation that defines the type of the data in the unparsed entity (e.g. GIF image or C source code). Like the URL, the notation is also specified in the DTD rather than in the instance document. In practice, unparsed entities and notations are not much used.

General or parameter

A general entity is used within the instance document. A general entity reference begins with an &. A parameter entity is used within the DTD. A parameter entity reference begins with a %.

Since this book focuses on processing instance documents, we’ll consider general entities primarily.
Not all combinations are possible. In fact, there are exactly five kinds of named entities:

Internal parsed general entities

The familiar entity references like &amp; and &copy; that are defined completely in the DTD. For example, this declaration defines the copy entity as the text “Copyright”:

```xml
<!ENTITY copy "Copyright">
```

These entities are used in element content and attribute values.

External parsed general entities

External parsed general entities are just like internal parsed general entities except that their replacement text is read from a separate document rather than the DTD. The document is identified by a relative or absolute URL. For example, this declaration defines the legal entity as the content read from the URL http://www.example.com/legal.xml:

```xml
<!ENTITY legal SYSTEM "http://www.example.com/legal.xml">
```

The file such an entity is read from is just like another XML document except that it has a text declaration instead of an XML declaration, may not have a document type declaration, and might not have a single root element.

External unparsed general entities

External unparsed general entities refer to files containing non-XML, binary data. They are declared similarly to external parsed entities, but they also have a notation. For example, these definitions identify an unparsed entity named logo at the URL http://www.example.com/logo.png with the notation image/png:

```xml
<!NOTATION PNG SYSTEM "image/png">
<!ENTITY logo SYSTEM "http://www.example.com/logo.png" NDATA PNG>
```

Unparsed entities are referenced by attributes with type ENTITY or ENTITIES rather than by entity references. For example, such an attribute might be declared like this:

```xml
<!ELEMENT figure EMPTY>
<!ATTLIST figure logo ENTITY #REQUIRED>
```

Instances of the figure element would look like this:
The parser does not actually provide you with the contents of an unparsed entity. Instead it tells you the URI from which the data can be retrieved and the notation for that data. However, you have to use Java’s networking and I/O classes to get the data at that URI.

Internal parsed parameter entities

Internal parsed parameter entities are used purely within the DTD. The replacement text is provided by a string literal in the DTD. References to these entities begin with a percent sign. They’re often used to parameterize content models and attribute types. For example, the DocBook DTD defines the intermod.redecl.module parameter entity as the word IGNORE:

```xml
<!ENTITY % intermod.redecl.module "IGNORE"> 
```

Unlike a general entity reference, the %intermod.redecl.module; parameter entity reference can only be used in the DTD, not in the instance document. Since our focus is on instance documents, not DTDs, you won’t see a lot of these in this book.

External parsed parameter entities

External parsed parameter entities are used purely with the DTD. The replacement text is provided by a DTD fragment at a given URL. References to these entities begin with a percent sign. They often connect the different parts of a modular DTD into one coherent whole. For example, the DocBook DTD defines the dbpool parameter entity using a PUBLIC ID that loads the DTD fragment at the relative URL dbpoolx.mod:

```xml
<!ENTITY % dbpool PUBLIC
"-//OASIS//ELEMENTS DocBook XML Information Pool V4.1.2//EN"
"dbpoolx.mod"> 
```

Again, since our focus is on instance documents and not DTDs, you won’t see a lot of these in this book.

Namespaces

Namespaces are not part of XML 1.0. Namespaces were invented about a year after XML 1.0 was released to help sort out the rapidly expanding world of XML applications that all needed to be mixed together in the same documents. There are many good XML applications that don’t use them at all. For example, DocBook 4.1.2, the XML application in which this book was written, is completely namespace free as are XML-RPC and RSS 0.9.1. However, even if you can write very useful XML applications without thinking about namespaces, you’re going to encounter namespaces when you work with XML
applications designed by other developers. Consequently it’s important to have a solid understanding of them.

The key idea of namespaces is that each element is bound to a Uniform Resource Identifier (URI) (a URL in practice). If IBM only uses URIs in the ibm.com domain and Sun only uses URIs in the sun.com domain, then there won’t be any confusion between Sun’s Book element and IBM’s Book element, even if they’re used in the same document. Just look at the URIs to tell which is which.

**Note**

A URI identifies a resource, but it does not necessarily locate it. URIs include not only Uniform Resource Locators (URLs) but also Uniform Resource Names (URNs). For instance, a URN for this book based on its ISBN number is `urn:isbn:0201771861`; but this does not tell you where you can find a copy of the book. However, most developers agree that only absolute URLs should be used as namespace URIs, and most XML applications follow this suggestion.

The URIs are purely string identifiers. Even if the URI is a URL, the parser does not connect to the server and try to download the document that’s found there. Indeed there may not be any such document. When plugged into web browsers, namespace URLs often produce 404 Not Found errors. You can use namespaces in standalone systems without any network connection at all. You don’t even have to have access to DNS. For the same reason, two different URLs that point to the same page define two different namespaces. For example, the following URLs identify the same page but three different namespaces:

- http://ns.cafeconleche.org/Orders/
- http://ns.cafeconleche.org/Orders
- http://ns.cafeconleche.org/Orders/index.html

Since URIs contain many characters which are illegal in element names as well as being excessively long to type, short prefixes stand in for the URIs. The prefixes are separated from the local name by a colon. For instance, instead of the URI `http://www.w3.org/2001/XInclude` you might use the prefix `xinclude` or `xi`. An include element in the `http://www.w3.org/2001/XInclude` namespace would then be written as `xi:include`. This element has the prefix `xi`, the local name `include`, the qualified name `xi:include`, and the namespace URI `http://www.w3.org/2001/XInclude`.

`xmlns:prefix` attributes bind particular prefixes to particular URIs within the element where the attribute appears. For example, inside this Order element, the prefix `xi` is bound to the URI `http://www.w3.org/2001/XInclude`:

```xml
<Order xmlns:xi="http://www.w3.org/2001/XInclude"/>
```
Each prefix used in an element or attribute name must be bound to a URI. Failure to do this is a namespace well-formedness error. Although you can parse documents without considering namespaces, in practice most parsers and APIs check namespaces by default and a violation of namespace well-formedness is as serious as a violation of the rules of XML 1.0.

The prefix can change as long as the URI stays the same. For example, this element is the same as the previous one:

```xml
<Order xmlns:xinclude="http://www.w3.org/2001/XInclude">
  <xinclude:include href="order_details.xml"/>
</Order>
```

You can also define a default namespace that applies to elements without prefixes. For example, Example 1.6 places the Order element and all its descendants in the http://ns.cafeconleche.org/Orders/ namespace, even though none of them have prefixes.

**Example 1.6. An XML document that uses a default namespace**

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order xmlns="http://ns.cafeconleche.org/Orders/">
  <Customer id="c32">Chez Fred</Customer>
  <Product>
    <Name>Birdsong Clock</Name>
    <SKU>244</SKU>
    <Quantity>12</Quantity>
    <Price currency="USD">21.95</Price>
    <ShipTo>
      <Street>135 Airline Highway</Street>
      <City>Narragansett</City> <State>RI</State> <Zip>02882</Zip>
    </ShipTo>
  </Product>
  <Subtotal currency='USD'>263.405</Subtotal>
  <Tax rate="7.0" currency='USD'>18.44</Tax>
  <Shipping method="USPS" currency='USD'>8.95</Shipping>
  <Total currency='USD'>290.79</Total>
</Order>
```

Although it’s most common to place the namespace binding attributes on the root element, they can
appear on other elements deeper in the hierarchy. They can even override previous bindings in the ancestor elements. This is especially common with the binding of the default namespace. For instance, in Example 1.7 the Order, Customer, Product, Name, SKU, Quantity, Price, Subtotal, Tax, Shipping, and Total elements are all in the http://ns.cafeconleche.org/Orders/ namespace. However, the ShipTo, Street, City, State, and Zip elements are in the http://ns.cafeconleche.org/Address/ namespace.

**Example 1.7. An XML document that uses two default namespaces**

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order xmlns="http://ns.cafeconleche.org/Orders/">
  <Customer id="c32">Chez Fred</Customer>
  <Product>
    <Name>Birdsong Clock</Name>
    <SKU>244</SKU>
    <Quantity>12</Quantity>
    <Price currency="USD">21.95</Price>
    <ShipTo xmlns="http://ns.cafeconleche.org/Address/">
      <Street>135 Airline Highway</Street>
      <City>Narragansett</City> <State>RI</State> <Zip>02882</Zip>
    </ShipTo>
  </Product>
  <Subtotal currency='USD'>263.40</Subtotal>
  <Tax rate="7.0" currency='USD'>18.44</Tax>
  <Shipping method="USPS" currency='USD'>8.95</Shipping>
  <Total currency='USD' >290.79</Total>
</Order>
```

Although it’s less common, prefixes can also be attached to attribute names to indicate what namespace the attribute is in. For example, XLink uses this to distinguish between the XLink attributes such as type and href and attributes with the same names that might be used in elements that need to become XLinks. This ShipTo element is also a simple XLink to the recipient’s e-mail address:

```xml
<ShipTo xmlns="http://ns.cafeconleche.org/Address/"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xlink:type="simple" xlink:href="mailto:chezfred@yahoo.com">
  <GiftRecipient>Samuel Johnson</GiftRecipient>
  <Street>271 Old Homestead Way</Street>
  <City>Woonsocket</City> <State>RI</State> <Zip>02895</Zip>
</ShipTo>
```
Unprefixed attributes are never in any namespace. Unlike elements, they cannot be in the default namespace. Furthermore, they are not in the same namespace as the element to which they are attached. If an attribute does not have a prefix, it is not in a namespace.

On occasion namespace prefixes are used in attribute values, element content, and even in processing instructions. In these cases the nearest ancestor element that contains a binding for that prefix establishes what URI the prefix is mapped to. Inside an element with an `xmlns:prefix` attribute, we say that the namespace is *in scope* even if it isn’t obviously used anywhere in that element. Namespaces in scope on an element include not only those that the element itself declares but also those that are declared on that element’s ancestors. An element can redeclare a namespace prefix so that it’s mapped to a different URI on the element and the element’s children than in the element’s parent. Slightly more commonly, an element can change the default namespace that applies within the element and its content.

When writing software to process XML documents that use namespaces, you almost always want to make your code dependent on the URI, not the prefix. If you’re comparing two elements for equality, compare them by URI and local name, not prefix and local name. If you’re searching for an element of a certain type, look for an element with the right URI and local name, not the right prefix and local name.

---

[2] The well-formedness constraints specify requirements that are difficult or impossible to express in BNF form; for example, that “The Name in an element’s end-tag must match the element type in the start-tag.”

[3] A few parsers continue reading so they can report further errors after the first one. However, they only report errors, not content.

[4] Technically, whether or not white space only nodes are considered to be “white space in element content” depends on the content specification for the element given by the DTD. A white space only text node is only white space in element content when the content specification for the parent element in the DTD indicates that the parent element can only contain child elements but not mixed content. Since Example 1.2 doesn’t have a DTD, this can't possibly be white space in element content.


[6] JDOM and dom4j actually do provide special support for processing instructions written in this pseudo-attribute format. However, they both do a substantial amount of work in their own classes to support this interface, beyond what the parser provides.
Validity

Programmers have long known the value of verifiable preconditions on functions and methods. (A lot of us carelessly don’t use them, but that’s a topic for another book.) One of the important innovations of XML is the ability to place preconditions on the data the programs read, and to do this in a simple declarative way. XML allows you to say that every Order element must contain exactly one Customer element, that each Customer element must have an id attribute that contains an XML name token, that every ShipTo element must contain one or more Streets, one City, one State, and one Zip, and so forth. Checking an XML document against this list of conditions is called validation. Validation is an optional step but an important one.

There is more than one language in which you can express such conditions. Generically, these are called schema languages, and the documents that list the constraints are called schemas. Different schema languages have different strengths and weaknesses. The document type definition (DTD) is the only schema language built into most XML parsers and endorsed as a standard part of XML. However, because of the extensible nature of XML, many other schema languages have been invented that can easily be integrated with your systems.

DTDs

A DTD focuses on the element structure of a document. It says what elements a document may contain, what each element may and must contain in what order, and what attributes each element has.

Element Declarations

In order to be valid according to a DTD, each element used in the document must be declared in an ELEMENT declaration. For example, this is an ELEMENT declaration that says that Name elements contain #PCDATA, that is, text but no child elements.

```xml
<!ELEMENT Name (#PCDATA)>
```

Elements that can have children are declared by listing the names of their children in order, separated by commas. For example, this ELEMENT declaration says that an Order element contains a Customer element, a Product element, a Subtotal element, a Tax element, a Shipping element, and a Total element in that order:

```xml
<!ELEMENT Order (Customer, Product, Subtotal, Tax, Shipping, Total)>
```
The parenthesized list of things an element can contain is called the element’s *content model*. You can attach a question mark after an element name in the content model to indicate that the element is optional; that is, that either zero or one instance of the element may occur at that position. You can attach an asterisk after the element name to indicate that zero or more instances of the element may occur at that position, or a plus sign to indicate that one or more instances of the element must occur at that position. For example, this element declaration states that a *ShipTo* element must contain zero or one *GiftRecipient* elements, one or more *Street* elements, and exactly one *City*, *State*, and *Zip* elements in that order:

```
<!ELEMENT ShipTo (GiftRecipient?, Street+, City, State, Zip)>
```

You can use a vertical bar instead of a comma to indicate that either one or the other of the elements may appear. You can group collections of elements with parentheses to indicate that the entire group should be treated as a unit. You can suffix a *, ?, or + to the group to indicate that zero or more, zero or one, or one or more of those groups may appear at that point. Finally, you may replace the entire content model with the keyword *EMPTY* to specify that the element must not contain any content at all.

### Attribute Declarations

A DTD also specifies which attributes may and must appear on which elements. Each attribute is declared in an *ATTLIST declaration* which specifies:

- The element to which the attribute belongs
- The name of the attribute
- The type of the attribute
- The default value of the attribute

For example, this ATTLIST declaration says that every *Customer* element must have an attribute named *id* with type *ID*:

```
<!ATTLIST Customer id ID #REQUIRED>
```

DTDs define ten different types for attributes:

- **CDATA**
  - Any string of text; the default type for undeclared attributes in invalid documents
- **NM_TOKEN**
  - A string composed of one or more legal XML name characters. Unlike an XML name, a name
Validity

token may start with a digit.

NMTOKENS

A white space separated list of name tokens

ID

An XML name that is unique among ID type attributes in the document

IDREF

An XML name used as an ID attribute value on some element in the document

IDREFS

A white space separated list of XML names used as ID attribute values somewhere in the document

ENTITY

The name of an unparsed entity declared in an ENTITY declaration in the DTD

ENTITIES

A white space separated list of unparsed entities declared in the DTD

NOTATION

The name of a notation declared in a NOTATION declaration in the DTD

Enumeration

A list of all legal values for the attribute, separated by vertical bars. Each possible value must be an XML name token.

Most parsers and APIs will tell you what the type of an attribute is if you want to know, but in practice this knowledge is not very useful. W3C XML schema language schemas offer much more complete data typing for both elements and attributes, including not only these types but also the more customary data types like int and double.

DTDs allow four possible default values for attributes:

#REQUIRED

Each element in the instance document must provide a value for this attribute.

#IMPLIED

Each element in the instance document may or may not provide a value for this attribute. If an
element does not, then no default value is provided from the DTD.[2]

#FIXED "value"

The attribute always has the value that follows #FIXED in double or single quotes, whether or not it’s present in the instance document.

"value"

By default the attribute has the value specified in the DTD in single or double quotes. However, individual instances of the element may specify a different value.

Parsers may or may not tell you whether an attribute came from the instance document or was defaulted in from the DTD. It’s relatively rare that you care about this one way or the other. However, if you’re using a document that relies heavily on attribute values from DTDs, (e.g. for namespace declarations) make sure you’re using a parser that does read the external DTD subset.

**Example 1.8** is a complete DTD for order documents of the type shown in this chapter. It uses both ELEMENT and ATTLIST declarations.

**Example 1.8. A DTD for order documents**

```xml
<!ELEMENT Order (Customer, Product+, Subtotal, Tax, Shipping, Total)>
<!ELEMENT Customer (#PCDATA)>
<!ATTLIST Customer id ID #REQUIRED>
<!ELEMENT Product (Name, SKU, Quantity, Price, Discount?, ShipTo, GiftMessage?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT SKU (#PCDATA)>
<!ELEMENT Quantity (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ATTLIST Price currency (USD | CAN | GBP) #REQUIRED>
<!ELEMENT Discount (#PCDATA)>
<!ELEMENT ShipTo (GiftRecipient?, Street+, City, State, Zip)>
<!ELEMENT GiftRecipient (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Zip (#PCDATA)>
<!ELEMENT GiftMessage (#PCDATA)>
<!ELEMENT Subtotal (#PCDATA)>
<!ATTLIST Subtotal currency (USD | CAN | GBP) #REQUIRED>
<!ELEMENT Tax (#PCDATA)>
<!ATTLIST Tax currency (USD | CAN | GBP) #REQUIRED>
```

http://www.cafeconleche.org/books/xmljava/chapters/ch01s03.html (4 of 11)2006-07-07 오후 2:07:36
Validity

rate CDATA "0.0"
>

<!ELEMENT Shipping (#PCDATA)>
<!ATTLIST Shipping currency (USD | CAN | GBP) #REQUIRED
       method (USPS | UPS | Overnight) "UPS">

<!ELEMENT Total (#PCDATA)>
<!ATTLIST Total currency (USD | CAN | GBP) #REQUIRED>

Document Type Declarations

Documents are associated with particular DTDs using *document type declarations*. This is a document
type declaration that points to the DTD in Example 1.8:

<!DOCTYPE Order SYSTEM "order.dtd">

The document type declaration is placed in the instance document’s prolog, after the XML declaration
but before the root element start-tag. For example,

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Order SYSTEM "order.dtd">
<Order>
  ...
</Order>
```

This does assume that the DTD can be found in the same directory where the document itself is. If you
prefer you can use an absolute URL instead. For example,

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Order SYSTEM "http://www.ibiblio.org/xml/dtds/order.dtd">
<Order>
  ...
</Order>
```

Even though Example 1.5 satisfies all the conditions expressed in Example 1.8, it is not valid because it
does not have a document type declaration pointing to that DTD.

Caution

The acronym DTD is correctly used only to mean “document type definition”. It
should never be used to mean “document type declaration”. The document type
declaration may contain or point to the document type definition (or both); but it is
not the same thing.
DTDs are not just about validation. They can also affect the content of the instance document itself. In particular, they can:

- Define entities
- Define notations
- Provide default values for attributes

Assuming you’re using a validating parser, there is little reason to care about how such things happen. The entities the DTD defines will be resolved before you see them. The notations will be applied to the appropriate elements and entities. A default attribute value will be just one more attribute in an element’s list of attributes. Some APIs may tell you what entity a particular element came from or whether an attribute value was defaulted from the DTD or present in the instance document. However, most of the time you simply do not need to know this.

**Schemas**

The W3C XML Schema Language (schemas for short, though it’s hardly the only schema language) addresses several limitations of DTDs. First schemas are written in XML instance document syntax, using tags, elements, and attributes. Secondly, schemas are fully namespace aware. Thirdly, schemas can assign data types like integer and date to elements, and validate documents not only based on the element structure but also on the contents of the elements.

Example 1.9 shows a schema for order documents. Where `order.dtd` uses an ELEMENT declaration, `order.xsd` uses an `xsd:element` element. Where `order.dtd` uses an ATTLIST declaration, `order.xsd` uses an `xsd:attribute` element.

But `order.xsd` doesn’t just repeat the same constraints found in `order.dtd`. It also assigns types and ranges to the elements. For instance, it requires that all the money elements—Tax, Shipping, Subtotal, Total, and Price—contain a decimal number such as 9.85, 7.2, or -3.25. If one of these elements contained text that was not a decimal number such as “France”, then the validator would notice and report the problem. DTDs cannot detect mistakes like this. A DTD can note that there is no Price element where one is expected, but it cannot determine that the Price element does not actually give a price.

**Example 1.9. order.xsd: a schema for order documents**

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

8 If one of these elements contained text that was not a decimal number such as “France”, then the validator would notice and report the problem. DTDs cannot detect mistakes like this. A DTD can note that there is no Price element where one is expected, but it cannot determine that the Price element does not actually give a price.
<xsd:sequence>
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base="xsd:string">
          <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Product" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="SKU" type="xsd:positiveInteger"/>
        <xsd:element name="Quantity" type="xsd:positiveInteger"/>
        <xsd:element name="Price" type="MoneyType"/>
        <xsd:element name="Discount" type="xsd:decimal" minOccurs="0"/>
        <xsd:element name="ShipTo">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="GiftRecipient" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
              <xsd:element name="Street" type="xsd:string"/>
              <xsd:element name="City" type="xsd:string"/>
              <xsd:element name="State" type="xsd:string"/>
              <xsd:element name="Zip" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="GiftMessage" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="Subtotal" type="MoneyType"/>
<xsd:element name="Tax"/>
<xsd:complexType>
  <xsd:simpleContent>
    <xsd:extension base="MoneyType">
      <xsd:attribute name="rate" type="xsd:decimal"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<xsd:element name="Shipping">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="MoneyType">
        <xsd:attribute name="method" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name="Total" type="MoneyType"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:complexType name="MoneyType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:decimal">
      <xsd:attribute name="currency" type="xsd:string"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
</xsd:schema>

There are multiple ways to indicate that a document should satisfy a known schema. The most common is an xsi:noNamespaceSchemaLocation attribute on the root element of the instance document. The xsi prefix is bound to the http://www.w3.org/2001/XMLSchema-instance URI. For example,

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Order xsi:noNamespaceSchemaLocation="order.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  ...
</Order>
```

Some parsers also provide ways to specify a schema from inside a program, for instance by setting
Validity

various properties. I’ll discuss this more when we get to programmatic validation in Chapter 7.

Schemas are still pretty bleeding edge technology at the time of this writing. (May, 2002). There are only a few parsers that provide incomplete implementations of the full W3C XML Schemas 1.0 specification. Nonetheless, developers have been clamoring for this functionality (if not necessarily this syntax) for some time so schemas seem likely to achieve broad adoption relatively quickly.

For the moment, schema support is limited to simple validation, much as DTD support is. A schema-aware parser will read an XML document, compare what it sees there to a schema, and return a boolean result: either the document satisfies the schema or it does not. In the event the document fails to satisfy the schema, the parser might give you a line number and a more detailed error message about exactly what the problem is, but that’s it. More complete use of schemas, in which parsers tell you what the type of any element is so you can, for example, convert elements with type xsd:int to actual Java ints, are still a matter for research and experiment.

Schematron

Rick Jelliffe’s Schematron is a radically different approach to an XML schema language. Whereas other languages are conservative (everything not permitted is forbidden) Schematron is liberal (everything not forbidden is permitted). Furthermore, Schematron is based on XPath so it can check cooccurrence constraints between elements and attributes; e.g. that the content of the total price element must be equal to the sum of the content of the subtotal, tax, and shipping elements. Finally, Schematron can be implemented as an XSLT stylesheet rather than requiring special software.

Example 1.10 shows a Schematron schema for order documents. To keep the example smaller, I did not test absolutely everything I could. Instead, I took advantage of Schematron’s liberality to test only those conditions that neither DTDs nor schemas can validate; for instance, that the total price is the sum of the subtotal, the tax, and the shipping. I haven’t necessarily lost anything by doing this. I can validate a single document against multiple different kinds of schemas. For instance, orders could first be checked against the DTD, then checked against a W3C XML Schema Language schema, and only checked against this Schematron schema if they passed the first two tests.

Example 1.10. order.sct: a Schematron schema for order documents

```xml
<?xml version="1.0"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <title>A Schematron Schema for Orders</title>
  <pattern>
    <rule context="Order">
      <!-- Due to round-off error, floating point numbers should rarely be compared for direct equality. For this purpose, it's enough if they're accurate
```
XML is not by itself Turing complete so there are still some limits to what you can express in a Schematron schema. For instance, you can’t sum up the Quantity times the Price for each Product element and make sure that equals the Subtotal. However, Schematron is still a lot more powerful than other schema languages.

Schematron is implemented in a very unusual fashion. First you run your Schematron schema through an XSLT processor using a skeleton stylesheet Jelliffe provides. This produces a new XSLT stylesheet. In essence, this compiles the Schematron schema into an XSLT stylesheet. The compiler is itself written in XSLT. You then transform all your instance documents using the compiled schema. If any of the assertions fail, the output will contain the assertion message. Otherwise it will contain just the XML declaration. For example, using Michael Kay’s Saxon XSLT processor, to validate Example 1.2 against Example 1.10:

```
C:\XMLJAVA> saxon order.sct skeleton1-5.xsl>order_sct.xsl
C:\XMLJAVA> saxon order.xml order_sct.xsl
<?xml version="1.0" encoding="utf-8"?>
```

Schematron is the idiosyncratic product of one person. It is therefore not a standard part of any major parsers, unlike DTDs and the W3C XML Schema Language. However, it’s not particularly difficult to install Jelliffe’s Schematron validation software into most systems. Since Schematron is implemented in XSLT, all you need is a good API to access an XSLT engine. I’ll take this up again in the final chapter when I discuss APIs for XSLT.
Validity

The Last Mile

Schematron is powerful, but there are still some checks it cannot perform. In particular, it cannot perform any checks that require information external to the document and the schema. For example, it cannot verify that the page at a referenced URL is reachable. It cannot verify that a file exists on the local file system. It cannot compare the SKUs, names, and prices in an order document with their values in a remote database. None of the extant schema languages allow you to state conditions like these.

Java can do all of these things. The java.net.URL class can easily test whether a URL is live. The exists() method of the java.io.File class is a simple test for whether a file is where you think it is. JDBC is a whole API remote database access. However, unlike the more limited constraints of DTDs, the W3C XML Schema Language, or even Schematron, simply listing the conditions is not enough. To test such conditions, you have to write the code that tests them. Nobody’s done the hard work for you. There will always be some constraints you need a full-blown programming language to check. Indeed doing exactly this will be a major focus of the remainder of this book.

One thing you can learn from the existing languages is the clean way they separate validation from processing. If you design your own validation layer, you should do that too. Perform all validation before the document is processed for its contents. If possible, separate the constraints from the code that checks them.

[7] This is really a bad choice of terminology. Nothing is being implied here. A more accurate keyword would be #OPTIONAL. However #IMPLIED is what XML gives us.

[8] It would be possible to go further and require that each money item be a positive number with two decimal digits of precision such as 9.85 but not 7.2 or -3.25, but for now I wanted to keep this example smaller.
For the most part this book is going to focus on XML documents used as input to and output from various kinds of programs. In many cases it’s entirely possible that the XML documents will be both written and read by software and that no human being ever even looks at the documents. However, on occasion people do need to load XML documents into a browser or print them on paper so they can read them. For this purpose XML documents are rather coarse, especially for non-programmers. To pretty them up, you can attach a style sheet to the document that specifies how each element should be presented. There are two main languages used for this purpose today, Cascading Style Sheets (CSS) and the Extensible Stylesheet Language (XSL).

CSS

CSS is a very straightforward, non-XML, declarative language. CSS rules attach style properties to elements. Each rule has a selector specifying which elements it applies to. The simplest selector is merely an element name such as Order or Price. This is followed by a pair of braces containing the style properties to apply to the selected elements. Each property has a name such as font-weight or display and a value that’s appropriate for that property. The name and value are separated by a colon. For example, this rule says that the Customer element should be bold faced:

Customer {font-weight: bold}

CSS rules often set multiple properties for a single element. Individual properties are separated by semicolons. For example, this rule says that the Order element (and all its descendants) should have the font-family serif and the font-size 16 points.

Order {font-family: serif; font-size: 16pt}

Most of the properties set on an element such as Order are inherited by all its descendants such as Customer, Price, and State. However, if a descendant element sets a different value for an inherited property, then that value overrides the inherited value. For example, this rule sets the font-family for the gift message to ZapfChancery or any script font if ZapfChancery is not available on the local system. It overrides the choice of a serif font that GiftMessage inherits from its ancestor Order element.

GiftMessage {font-family: ZapfChancery, script}

The selector syntax can be adjusted to apply to multiple elements at one time by separating the names by commas. For example, this rule specifies that five different elements all have the value block for the display property. This means each will be separated from the previous and following elements by a line break.

Street, Subtotal, Tax, Shipping, Total {display: block}

CSS also allows you to select elements according to attributes, parentage, siblings, ID, link status, and more. An asterisk can be used to stand in for any element.

Example 1.11 is a complete CSS stylesheet for order documents such as Example 1.2. It adds several new features including setting the display property to none to hide the SKU element and using the :before selector and the content property to add a little boiler-plate text in front of several elements. However, although useful, these facilities are limited. You still
can’t reorder the elements, and the content property is limited to plain text, no markup.

**Example 1.11. A CSS stylesheet for order documents**

```css
Order {font-family: serif; font-size: 16pt;
    display: block; line-height: 20pt;
    margin-left: 0.25in
 }
ShipTo {margin-left: 0.5in; display: block }
ShipTo:before {content: "Ship to:"; margin-left: -0.25in }
Product {font-size: 12pt; display: block }
Customer {font-weight: bold; display: block }
GiftMessage {font-family: ZapfChancery, script}
Street, Subtotal, Tax, Shipping, Total {display: block}
Quantity:before {content: "Quantity: "}
SKU {display: none}
Subtotal:before {content: "Subtotal: "$}
Price:before {content: "Unit Price: "$}
Tax:before {content: "Tax: "$}
Shipping:before {content: "Shipping: "$}
Total:before {content: "Total: "$}
```

**Associating Style Sheets with XML Documents**

A few browsers may allow the user to specify which stylesheet to apply to a document. In general, however, an XML document that will be read by people must carry an xml-stylesheet processing instruction in its prolog that indicates which stylesheet should be applied to that document. This processing instruction has two pseudo-attributes, type and href. The type pseudo-attribute identifies the MIME media type of the stylesheet, text/css for Cascading Style Sheets. The href pseudo-attribute specifies the relative or absolute URL where the style sheet can be found. For example, this xml-stylesheet processing instruction says that a CSS style sheet named order.css can be found in the same directory where the XML document itself was found:

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

Figure 1.1 shows **Example 1.2** loaded into Mozilla after the stylesheet in **Example 1.11** has been attached. Loaded into Opera or Netscape 6, the results would be similar. Internet Explorer 5.5 and earlier have much weaker support for CSS, and would not do nearly as good a job formatting the XML. Netscape 4.x and earlier have absolutely no support for displaying XML documents.

**Figure 1.1. The clock order styled by CSS**
This isn’t bad. A browser with full support for CSS Level 2 does let you do a lot, but there are still numerous issues. The recipient’s name should be included in the ship-to information. The subtotal, tax, shipping, and total dollar amounts should really be aligned. And the product name, quantity, and price should probably be in a multi-row table with one row for each item in the order. Although CSS does support tables, it requires that the markup in the XML document already be structured in a very tabular fashion, which it isn’t here.

**XSL**

CSS is limited to specifying how the text content of each element is styled. It presents pretty much the entire content of the XML document in pretty much the order it was present in the stylesheet. It cannot add boilerplate text such as the return policy or extract just the elements needed for an address label. It is limited to specifying the appearance of elements already present in the XML document without changing their order or content.

XSL, by contrast, can reorder elements, delete certain elements, add content that wasn’t present in the original document, combine multiple documents into a single result, and more. Because of its ability to add text to what’s in the XML document before the document is shown to the user, XSL is much more suited for many of the examples in this book that contain only the raw data without a lot of extraneous text. XSL is a much more powerful stylesheet language than CSS. However, it is also comparatively difficult to learn.

XSL is divided into two complementary parts, XSL Transformations (XSLT) and XSL Formatting Objects (XSL-FO). Unlike CSS, both XSLT and XSL-FO are XML applications. XSLT stylesheets and XSL-FO documents are well-formed XML documents.
XSLT is a Turing-complete functional programming language designed specifically for describing transformations from one XML format into another. (A functional programming language is one with no side effects, in which the order of evaluation of statements makes no difference to the final result. The classic examples of functional languages are Scheme and Lisp.) An XSLT processor reads an XML document and an XSLT stylesheet, and transforms the document according to the instructions found in the stylesheet. It then outputs the transformed document. XSLT is designed for XML-to-XML transformations, but it can transform XML to HTML, XML to plain text, or XML to any other text format such as TeX or troff.

XSL-FO is an XML application that describes the layout of the various elements on a printed page. It has elements that represent paragraphs, list items, margins, and so forth. An XSL-FO processor converts an XSL-FO document into some other format that can be printed or viewed such as PDF, TeX, SVG, or plain text. This book was actually written in an XML application called DocBook. An XSL style sheet converted the source DocBook files into XSL-FO document, which was then further processed to produce the PDF document that was sent to the printer.

XSLT is based on the notion of templates. The XSLT processor compares nodes in an input document tree to templates in the stylesheet. When it finds a match, it follows the instructions in that template. These instructions can include XML to output, details about what to copy from the input into the output, and directions for which nodes to process next.

Example 1.12 is an XSLT stylesheet that describes how order documents can be converted into XSL Formatting Objects. This particular stylesheet formats the products as a table, and the rest of the elements as paragraphs. The products are written in a 12-point font while the rest of the document uses 16-point fonts.

### Example 1.12. An XSLT stylesheet for order documents

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <!-- Try to make the output look half decent -->
    <xsl:output indent="yes"/>

    <xsl:template match="Order">
        <fo:root>
            <fo:layout-master-set>
                <fo:simple-page-master master-name="only">
                    <fo:region-body margin-left="0.5in"
                        margin-top="0.5in"/>
                </fo:simple-page-master>
            </fo:layout-master-set>

            <fo:page-sequence master-reference="only">
                <fo:flow flow-name="xsl-region-body">
                    <xsl:apply-templates select="Customer"/>
                    <xsl:apply-templates select="ShipTo"/>
                    <fo:table font-size="12pt"
                        space-before="24pt" space-after="24pt">
                        <fo:table-column column-width="2in"/>
                        <fo:table-column column-width="1in"/>
                        <fo:table-column column-width="1in"/>
                    </fo:table>
                </fo:flow>
            </fo:page-sequence>
        </fo:root>
    </xsl:template>
</xsl:stylesheet>
```
<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Unit Price</th>
<th>Subtotal</th>
</tr>
</thead>
</table>

<xsl:apply-templates select="Product"/>

<xsl:apply-templates select="Tax"/>
<xsl:apply-templates select="Shipping"/>
<xsl:apply-templates select="Total"/>

<fo:flow>
  
</fo:flow>
The output vocabulary used in this stylesheet is XSL Formatting Objects. The actual document produced by transforming Example 1.5 with this style sheet is shown in Example 1.13 (with a few allowances for white space). This document would then be fed into an XSL-FO processor such as the Apache XML Project’s open source FOP to convert it to some other format, in this case PDF.

**Example 1.13. An XSL-FO document for the clock order**

```xml
<?xml version="1.0" encoding="utf-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="only">
      <fo:region-body margin-left="0.5in" margin-top="0.5in"/>
    </fo:simple-page-master>
  </fo:layout-master-set>
  <fo:page-sequence master-reference="only">
    <fo:flow flow-name="xsl-region-body">
      Ship to:
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

http://www.cafeconleche.org/books/xmljava/chapters/ch01s04.html (6 of 9) 2006-07-07 2:08:38
Chez Fred

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
<th>Unit Price</th>
<th>Subtotal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Birdsong Clock</td>
<td>12</td>
<td>$21.95</td>
<td>$263.4</td>
</tr>
<tr>
<td>Brass Ship's Bell</td>
<td>1</td>
<td>$144.95</td>
<td>$144.95</td>
</tr>
</tbody>
</table>
Since this book focuses more on using XML for computer-to-computer communication rather than computer-to-human or human-to-human communication, I’m not going to spend very many pages on style sheets. Indeed, this is the last time you’re going to see them for the next fifteen chapters. However, you should at least be aware that XSLT is an extremely powerful approach for certain kinds of problems. Indeed once you’ve gained a little facility with XSLT, you’ll notice that it is often easier to write an XSLT style sheet to solve a problem than a classic procedural program in Java. Even when XSLT can’t do everything you need, you may find that it can solve a large part of your problem. XML systems are often designed as a chain of steps: first validation using a DTD or schema, then transformation using XSLT, and finally whatever custom processing you wish to apply using Java. Many apparently hard problems become much simpler and easier to tackle when broken down in this fashion. I’ll explore this further when we return to XSLT in Chapter 17.
XML is a standard textual markup language suitable for encoding almost any sort of data. It works very well for both unstructured narrative data written by people and for the record-oriented data common in computer applications. About the only thing it’s not really suitable for are bitmapped things such as photographs and recorded sound.

Logically an XML document is made up of nested elements. Each element has a name, a set of attributes and some content. The content can include plain text and/or other elements. The attributes are name value pairs associated with the element. Each document has a single topmost element called the root or document element. Since all non-root elements nest completely inside other elements, an XML document has a natural tree structure. Besides elements and text nodes, XML documents can also contain comments, processing instructions, an XML declaration, and a document type declaration.

Syntactically elements are delimited by tags that look like `<Quantity>`, `</Quantity>`, and `<Quantity/>`. `<Quantity>` is a start-tag that must be matched by the corresponding end-tag `</Quantity>`. The content of the `Quantity` element comes in-between these two tags. `<Quantity/>` is an empty-element tag that represents a `Quantity` element with no content. Attributes are indicated by `name="value"` pairs inside start-tags and empty-element tags. For example, `<Quantity number="17"/>` is an empty `Quantity` element that has a `number` attribute with the value 17.

Physically, an XML document is divided into storage units called entities. These entities can be files, database records, data structures in memory, or something else. The document entity contains the root element of the document. Parsed entities contain XML markup and that will be merged to form the entire document. Parsed entities are located via general entity references such as `&anaconda;` in the document entity or another parsed entity. Unparsed entities contain non-XML, possibly binary data that will be identified by ENTITY type attributes in the document.

Every XML document must be well-formed. Among other things this means, every start-tag must have a matching end-tag, every attribute value must be quoted, and only certain characters can be used in element names. If a document is not well-formed, it is not an XML document; and XML parsers will not accept it. Beyond well-formedness, documents that have a schema may be (but do not have to be) valid. A valid document adheres to all the constraints listed in the schema. Schema languages include Document Type Definitions (DTDs), the W3C XML Schema Language, and the XPath-based Schematron.

Since XML markup normally focuses on the structure and semantics of the contained information,
before a document can be shown to a human reader, it must first be associated with a style sheet that tells the browser or other tool how to format the document for display to a person. The two most popular style languages are Cascading Style Sheets (CSS) and the Extensible Stylesheet Language (XSL). CSS is a non-XML declarative language for applying simple styles such as font-weight to elements of certain types. XSL is actually two separate XML applications, the XSL-FO page description language and the XSLT Turing-complete functional language. An XSLT style sheet is used to transform a source XML document into the XSL-FO vocabulary. However, XSLT can also be used to transform to other XML vocabularies such as XHTML.