1

Xerces

XML parsing is the foundational building block for every other tool we'll be looking at in this book. You can't use Xalan, the XSLT engine, without an XML parser because the XSLT stylesheets are XML documents. The same is true for FOP and its input XSL:FO, Batik and SVG, and all the other Apache XML tools. Even if you as a developer aren't interacting with the XML parser directly, you can be sure that each of the tools you describe makes use of an XML parser.

XML parsing technology is so important that the ASF has two XML parsing projects: Xerces and Crimson. The reason for this is historical. When the ASF decided to create http://xml.apache.org, both IBM and Sun had Java-based XML parsers that they wanted to donate to the project. The IBM parser was called XML for Java (XML4J) and was available in source code from IBM's AlphaWorks Website. The Sun parser was originally called Project X. The code base for IBM's XML4J parser became the basis for Xerces, and the code base for Project X became the basis for Crimson. The goal of the parsing project was to build a best-of-breed parser based on the ideas and experience of XML4J and Project X. This did not happen right away; it wasn't until late in 2000 that a second-generation Xerces effort was begun.

Throughout this chapter and the rest of this book, we'll use Xerces for Java 2 (hereafter Xerces) as our parser. Xerces for Java 2 is replacing both Xerces for Java 1 and Crimson. At the time of this writing, the plan is for a version of Xerces to be the reference implementation for XML parsing in the Sun Java Developer's Kit (JDK). Xerces is a fully featured XML parser that supports the important XML standards:

- XML 1.0, Second Edition
- XML Namespaces
- SAX 2.0
- DOM Level 1
- DOM Level 2 (Core, Events, Range, and Traversal)

- Java APIs for XML Parsing 1.2
- XML Schema 1.0 (Schema and Structures)

The current release of Xerces (2.4.0) also has experimental support for:

- XML 1.1 Candidate Recommendation
- XML Namespaces 1.1 Candidate Recommendation
- DOM Level 3 (Core, Load/Save)

A word about experimental functionality: one of the goals of the Xerces project is to provide feedback to the various standards bodies regarding specifications that are under development. This means the Xerces developers are implementing support for those standards before the standards are complete. Work in these areas is always experimental until the specification being implemented has been approved. If you need functionality that is documented as experimental, you may have to change your code when the final version of the specification is complete. If the functionality you need is implemented only in an experimental package, be aware that the functionality may change or be removed entirely as the standards process continues. A good example is abstract support for grammars (both DTDs and XML Schema), which was supposed to be part of DOM Level 3. However, the DOM Working Group decided to cease work on this functionality, so it had to be removed from Xerces. This is a rare and extreme occurrence, but you should be aware that it has happened. Most situations are less severe, such as changes in the names and signatures of APIs.

Prerequisites

You must understand a few basics about XML and related standards in order to make good use of the material in this chapter. Following is a quick review. If you need more information, *XML in a Nutshell, Second Edition* by Eliotte Rusty Harold and W. Scott Means is a good source for the relevant background. Let's begin with the following simple XML file:

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <book xmlns="http://sauria.com/schemas/apache-xml-book/book"
 3:   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4:   xsi:schemaLocation=
 5:    "http://sauria.com/schemas/apache-xml-book/book
 6:     http://www.sauria.com/schemas/apache-xml-book/book.xsd"
 7:   version="1.0">
 8:   <title>Professional XML Development with Apache Tools</title>
 9:   <author>Theodore W. Leung</author>
10:   <isbn>0-7645-4355-5</isbn>
11:   <month>December</month>
12:   <year>2003</year>
13:   <publisher>Wrox</publisher>
14:   <address>Indianapolis, Indiana</address>
15: </book>
```

Like all XML files, this file begins with an XML declaration (line 1). The XML declaration says that this is an XML file, the version of XML being used is 1.0, and the character encoding being used for this file is UTF-8. Until recently, the version number was always 1.0, but the W3C XML Working Group is in the

process of defining XML 1.1. When they have finished their work, you will be able to supply 1.1 in addition to 1.0 for the version number. If there is no encoding declaration, then the document must be encoded using UTF-8. If you forget to specify an encoding declaration or specify an incorrect encoding declaration, your XML parser will report a fatal error. We'll have more to say about fatal errors later in the chapter.

Well-Formedness

The rest of the file consists of data that has been marked up with tags (such as <title> and <author>). The first rule or prerequisite for an XML document is that it must be *well-formed*. (An XML parser is required by the XML specification to report a fatal error if a document isn't well-formed.) This means every start tag (like <book>) must have an end tag (</book>). The start and end tag, along with the data in between them, is called an *element*. Elements may not overlap; they must be nested within each other. In other words, the start and end tag of an element must be inside the start and end tag of any element that encloses it. The data between the start and end tag is also known as the *content* of the element; it may contain elements, characters, or a mix of elements and characters. Note that the start tag of an element may contain attributes. In our example, the book element contains an xsi:schemaLocation attribute in lines 4-5. The value of an attribute must be enclosed in either single quotes (') or double quotes ("). The type of the end quote must match the type of the beginning quote.

Namespaces

In lines 2-4 you see a number of namespace declarations. The first declaration in line 2 sets the default namespace for this document to http://sauria.com/schemas/apache-xml-book/book. *Namespaces* are used to prevent name clashes between elements from two different grammars. You can easily imagine the element name title or author being used in another XML grammar, say one for music CDs. If you want to combine elements from those two grammars, you will run into problems trying to determine whether a title element is from the book grammar or the CD grammar.

Namespaces solve that problem by allowing you to associate each element in a grammar with a namespace. The namespace is specified by a URI, which is used to provide a unique name for the namespace. You can't expect to be able to retrieve anything from the namespace URI. When you're using namespaces, it's as if each element or attribute name is prefixed by the namespace URI. This is very cumbersome, so the XML Namespaces specification provides two kinds of shorthand. The first shorthand is the ability to specify the default namespace for a document, as in line 2. The other shorthand is the ability to declare an abbreviation that can be used in the document instead of the namespace URI. This abbreviation is called the *namespace prefix*. In line 3, the document declares a namespace prefix xsi for the namespace associated with http://www.w3.org/2001/XMLSchema-instance. You just place a colon and the desired prefix after xmlns.

Line 4 shows how namespace prefixes are used. The attribute schemaLocation is prefixed by xsi, and the two are separated by a colon. The combined name xsi:schemaLocation is called a *qualified name* (QName). The prefix is xsi, and the schemaLocation portion is also referred to as the *local part* of the QName. (It's important to know what all these parts are called because the XML parser APIs let you access each piece from your program.)

Default namespaces have a lot of gotchas. One tricky thing to remember is that if you use a default namespace, it only works for elements—you must prefix any attributes that are supposed to be in the default namespace. Another tricky thing about default namespaces is that you have to explicitly define a

default namespace. There is no way to get one "automatically". If you don't define a default namespace, and then you write an unprefixed element or attribute, that element or attribute is in no namespace at all.

Namespace prefixes can be declared on any element in a document, not just the root element. This includes changing the default namespace. If you declare a prefix that has been declared on an ancestor element, then the new prefix declaration works for the element where it's declared and all its child elements.

You may declare multiple prefixes for the same namespace URI. Doing so is perfectly allowable; however, remember that namespace equality is based on the namespace URI, not the namespace prefix. Thus elements that look like they should be in the same namespace can actually be in different namespaces. It all depends on which URI the namespace prefixes have been bound to. Also note that certain namespaces have commonly accepted uses, such as the xsi prefix used in this example. Here are some of the more common prefixes:

Namespace Prefix	Namespace URI	Usage
xsi	http://www.w3.org/2001/XMLSchema-instance	XML Schema Instance
xsd	http://www.w3.org/2001/XMLSchema	XML Schema
xsl	http://www.w3.org/1999/XSL/Transform	XSLT
fo	http://www.w3.org/1999/XSL/Format	XSL Formatting Objects
xlink	http://www.w3.org/1999/xlink	XLink
svg	http://www.w3.org/2000/svg	Scalable Vector Graphics
ds	http://www.w3.org/2000/09/xmldsig#	XML Signature
xenc	http://www.w3.org/2001/04/xmlenc#	XML Encryption

Validity

The second rule for XML documents is *validity*. It's a little odd to say "rule" because XML documents don't have to be valid, but there are well defined rules that say what it means for a document to be valid. Validity is the next step up from well-formedness. Validity lets you say things like this: Every book element must have a title element followed by an author element, followed by an isbn element, and so on. Validity says that the document is valid according to the rules of some grammar. (Remember diagramming sentences in high-school English? It's the same kind of thing we're talking about here for valid XML documents.)

Because a document can only be valid according to the rules of a grammar, you need a way to describe the grammar the XML document must follow. At the moment, there are three major possibilities: DTDs, the W3C's XML Schema, and OASIS's Relax-NG.

DTDs

The XML 1.0 specification describes a grammar using a document type declaration (DTD). The language for writing a DTD is taken from SGML and doesn't look anything like XML. DTDs can't deal with namespaces and don't allow you to say anything about the data between a start and end tag. Suppose you have an element that looks like this:

```
<quantity>5</quantity>
```

Perhaps you'd like to be able to say that the content of a <quantity> element is a non-negative integer. Unfortunately, you can't say this using DTDs.

XML Schema

Shortly after XML was released, the W3C started a Working Group to define a new language for describing XML grammars. Among the goals for this new schema language were the following:

- Describe the grammar/schema in XML.
- Support the use of XML Namespaces.
- Allow rich datatypes to constrain element and attribute content.

The result of the working group's effort is known as *XML Schema*. The XML Schema specification is broken into two parts:

- *XML Schema Part 1: Structures* describes XML Schema's facilities for specifying the rules of a grammar for an XML document. It also describes the rules for using XML Schema in conjunction with namespaces.
- *XML Schema Part 2: Datatypes* covers XML Schema's rich set of datatypes that enable you to specify the types of data contained in elements and attributes. There are a lot of details to be taken care of, which has made the specification very large.

If you're unfamiliar with XML Schema, *XML Schema Part 0: Primer* is a good introduction.

Relax-NG

The third option for specifying the grammar for an XML document is Relax-NG. It was designed to fulfill essentially the same three goals that were used for XML Schema. The difference is that the resulting specification is much simpler. Relax-NG is the result of a merger between James Clark's TREX and MURATA Makoto's Relax. Unfortunately, there hasn't been much industry support for Relax-NG, due to the W3C's endorsement of XML Schema. Andy Clark's Neko XML tools provide basic support for Relax-NG that can be used with Xerces. We'll cover the Neko tools a bit later in the chapter.

Validity Example

Let's go back to the example XML file. We've chosen to specify the grammar for the book.xml document using XML Schema. The xsi:schemaLocation attribute in lines 4-5 works together with the default namespace declaration in line 2 to tell the XML parser that the schema document for the namespace http://sauria.com/schemas/apache-xml-book/book is located at http://www.sauria.com /schemas/apache-xml-book/book.xsd. The schema is attached to the namespace, not the document.

There's a separate mechanism for associating a schema with a document that has no namespace (xsi:noNamespaceSchemaLocation). For completeness, here's the XML Schema document that describes book.xml.

```
 1: <?xml version="1.0" encoding="UTF-8"?>
 2: <xs:schema
 3:   targetNamespace="http://sauria.com/schemas/apache-xml-book/book"
 4:   xmlns:book="http://sauria.com/schemas/apache-xml-book/book"
 5:   xmlns:xs="http://www.w3.org/2001/XMLSchema"
 6:   elementFormDefault="qualified">
 7:   <xs:element name="address" type="xs:string"/>
 8:   <xs:element name="author" type="xs:string"/>
 9:   <xs:element name="book">
10:     <xs:complexType>
11:       <xs:sequence>
12:         <xs:element ref="book:title"/>
13:         <xs:element ref="book:author"/>
14:         <xs:element ref="book:isbn"/>
15:         <xs:element ref="book:month"/>
16:         <xs:element ref="book:year"/>
17:         <xs:element ref="book:publisher"/>
18:         <xs:element ref="book:address"/>
19:       </xs:sequence>
20:       <xs:attribute name="version" type="xs:string" use="required"/>
21:
22:     </xs:complexType>
23:   </xs:element>
24:   <xs:element name="isbn" type="xs:string"/>
25:   <xs:element name="month" type="xs:string"/>
26:   <xs:element name="publisher" type="xs:string"/>
27:   <xs:element name="title" type="xs:string"/>
28:   <xs:element name="year" type="xs:short"/>
29: </xs:schema>
```

Entities

The example document is a single file; in XML terminology, it's a single *entity*. Entities correspond to units of storage for XML documents or portions of XML documents, like the DTD. Not only is an XML document a tree of elements, it can be a tree of entities as well. It's important to keep this in mind because entity expansion and retrieval of remote entities can be the source of unexpected performance problems. Network fetches of DTDs or a common library of entity definitions can cause intermittent performance problems. Using entities to represent large blocks of data can lead to documents that look reasonable in size but that blow up when the entities are expanded. Keep these issues in mind if you're going to use entities in your documents.

XML Parser APIs

Now that we've finished the XML refresher, let's take a quick trip through the two major parser APIs: SAX and DOM. A third parser API, the STreaming API for XML (STAX), is currently making its way through the Java Community Process (JCP).

A parser API makes the various parts of an XML document available to your application. You'll be seeing the SAX and DOM APIs in most of the other Apache XML tools, so it's worth a brief review to make sure you'll be comfortable during the rest of the book.

Let's look at a simple application to illustrate the use of the parser APIs. The application uses a parser API to parse the XML book description and turn it into a JavaBean that represents a book. This book object is a domain object in an application you're building. The file Book.java contains the Java code for the Book JavaBean. This is a straightforward JavaBean that contains the fields needed for a book, along with getter and setter methods and a toString method:

```
 1: /*
 2:  *
 3:  * Book.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: public class Book {
11:     String title;
12:     String author;
13:     String isbn;
14:     String month;
15:     int year;
16:     String publisher;
17:     String address;
18:
19:     public String getAddress() {
20:         return address;
21:     }
22:
23:     public String getAuthor() {
24:         return author;
25:     }
26:
27:     public String getIsbn() {
28:         return isbn;
29:     }
30:
31:     public String getMonth() {
32:         return month;
33:     }
34:
35:     public String getPublisher() {
36:         return publisher;
37:     }
38:
39:     public String getTitle() {
40:         return title;
41:     }
42:
43:     public int getYear() {
```

```
44:         return year;
45:     }
46:
47:     public void setAddress(String string) {
48:         address = string;
49:     }
50:
51:     public void setAuthor(String string) {
52:         author = string;
53:     }
54:
55:     public void setIsbn(String string) {
56:         isbn = string;
57:     }
58:
59:     public void setMonth(String string) {
60:         month = string;
61:     }
62:
63:     public void setPublisher(String string) {
64:         publisher = string;
65:     }
66:
67:     public void setTitle(String string) {
68:         title = string;
69:     }
70:
71:     public void setYear(int i) {
72:         year = i;
73:     }
74:
75:     public String toString() {
76:         return title + " by " + author;
77:     }
78: }
```

SAX

Now that you have a JavaBean for Books, you can turn to the task of parsing XML that uses the book vocabulary. The SAX API is event driven. As Xerces parses an XML document, it calls methods on one or more event-handler classes that you provide. The following listing, SAXMain.java, shows a typical method of using SAX to parse a document. After importing all the necessary classes in lines 8-14, you create a new XMLReader instance in line 19 by instantiating Xerces' SAXParser class. You then instantiate a BookHandler (line 20) and use it as the XMLReader's ContentHandler and ErrorHandler event callbacks. You can do this because BookHandler implements both the ContentHandler and ErrorHandler interfaces. Once you've set up the callbacks, you're ready to call the parser, which you do in line 24. The BookHandler's callback methods build an instance of Book that contains the information from the XML document. You obtain this Book instance by calling the getBook method on the bookHandler instance, and then you print a human-readable representation of the Book using toString.

```
 1: /*
 2:  *
 3:  * SAXMain.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11:
12: import org.apache.xerces.parsers.SAXParser;
13: import org.xml.sax.SAXException;
14: import org.xml.sax.XMLReader;
15:
16: public class SAXMain {
17:
18:     public static void main(String[] args) {
19:         XMLReader r = new SAXParser();
20:         BookHandler bookHandler = new BookHandler();
21:         r.setContentHandler(bookHandler);
22:         r.setErrorHandler(bookHandler);
23:         try {
24:             r.parse(args[0]);
25:             System.out.println(bookHandler.getBook().toString());
26:         } catch (SAXException se) {
27:             System.out.println("SAX Error during parsing " +
28:                 se.getMessage());
29:             se.printStackTrace();
30:         } catch (IOException ioe) {
31:             System.out.println("I/O Error during parsing " +
32:                 ioe.getMessage());
33:             ioe.printStackTrace();
34:         } catch (Exception e) {
35:             System.out.println("Error during parsing " +
36:                 e.getMessage());
37:             e.printStackTrace();
38:         }
39:     }
40: }
```

The real work in a SAX-based application is done by the event handlers, so let's turn our attention to the BookHandler class and see what's going on. The following BookHandler class extends SAX's DefaultHandler class. There are two reasons. First, DefaultHandler implements all the SAX callback handler interfaces, so you're saving the effort of writing all the implements clauses. Second, because DefaultHandler is a class, your code doesn't have to implement every method in every callback interface. Instead, you just supply an implementation for the methods you're interested in, shortening the class overall.

```
1: /*
2:  *
3:  * BookHandler.java
4:  *
```

```
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.util.Stack;
11:
12: import org.xml.sax.Attributes;
13: import org.xml.sax.SAXException;
14: import org.xml.sax.SAXParseException;
15: import org.xml.sax.helpers.DefaultHandler;
16:
17: public class BookHandler extends DefaultHandler {
18:     private Stack elementStack = new Stack();
19:     private Stack textStack = new Stack();
20:     private StringBuffer currentText = null;
21:     private Book book = null;
22:
23:     public Book getBook() {
24:         return book;
25:     }
26:
```

We'll start by looking at the methods you need from the ContentHandler interface. Almost all ContentHandlers need to manage a stack of elements and a stack of text. The reason is simple. You need to keep track of the level of nesting you're in. This means you need a stack of elements to keep track of where you are. You also need to keep track of any character data you've seen, and you need to do this by the level where you saw the text; so, you need a second stack to keep track of the text. These stacks as well as a StringBuffer for accumulating text and an instance of Book are declared in lines 18-21. The accessor to the book instance appears in lines 23-25.

The ContentHandler callback methods use the two stacks to create a Book instance and call the appropriate setter methods on the Book. The methods you're using from ContentHandler are startElement, endElement, and characters. Each callback method is passed arguments containing the data associated with the event. For example, the startElement method is passed the localPart namespace URI, and the QName of the element being processed. It's also passed the attributes for that element:

```
27:     public void startElement(
28:         String uri,
29:         String localPart,
30:         String qName,
31:         Attributes attributes)
32:         throws SAXException {
33:         currentText = new StringBuffer();
34:         textStack.push(currentText);
35:         elementStack.push(localPart);
36:         if (localPart.equals("book")) {
37:             String version = attributes.getValue("", "version");
38:             if (version != null && !version.equals("1.0"))
39:                 throw new SAXException("Incorrect book version");
40:             book = new Book();
41:         }
42:     }
```

The startElement callback basically sets things up for new data to be collected each time it sees a new element. It creates a new currentText StringBuffer for collecting this element's text content and pushes it onto the textStack. It also pushes the element's name on the elementStack for placekeeping. This method must also do some processing of the attributes attached to the element, because the attributes aren't available to the endElement callback. In this case, startElement verifies that you're processing a version of the book schema that you understand (1.0).

You can't do most of the work until you've encountered the end tag for an element. At this point, you will have seen any child elements and you've seen all the text content associated with the element. The following endElement callback does the real heavy lifting. First, it pops the top off the textStack, which contains the text content for the element it's processing. Depending on the name of the element being processed, endElement calls the appropriate setter on the Book instance to fill in the correct field. In the case of the year, it converts the String into an integer before calling the setter method. After all this, endElement pops the elementStack to make sure you keep your place.

```
43:
44:       public void endElement(String uri, String localPart,
45:         String qName)
46:         throws SAXException {
47:         String text = textStack.pop().toString();
48:         if (localPart.equals("book")) {
49:         } else if (localPart.equals("title")) {
50:             book.setTitle(text);
51:         } else if (localPart.equals("author")) {
52:             book.setAuthor(text);
53:         } else if (localPart.equals("isbn")) {
54:             book.setIsbn(text);
55:         } else if (localPart.equals("month")) {
56:             book.setMonth(text);
57:         } else if (localPart.equals("year")) {
58:             int year;
59:             try {
60:                 year = Integer.parseInt(text);
61:             } catch (NumberFormatException e) {
62:                 throw new SAXException("year must be a number");
63:             }
64:             book.setYear(year);
65:         } else if (localPart.equals("publisher")) {
66:             book.setPublisher(text);
67:         } else if (localPart.equals("address")) {
68:             book.setAddress(text);
69:         } else {
70:             throw new SAXException("Unknown element for book");
71:         }
72:         elementStack.pop();
73:     }
74:
```

The characters callback is called every time the parser encounters a piece of text content. SAX says that characters may be called more than once inside a startElement/endElement pair, so the implementation of characters appends the next text to the currentText StringBuffer. This ensures that you collect all the text for an element:

```
75:     public void characters(char[] ch, int start, int length)
76:         throws SAXException {
77:         currentText.append(ch, start, length);
78:     }
79:
```

The remainder of BookHandler implements the three public methods of the ErrorHandler callback interface, which controls how errors are reported by the application. In this case, you're just printing an extended error message to System.out. The warning, error, and fatalError methods use a shared private method getLocationString to process the contents of a SAXParseException, which is where they obtain position information about the location of the error:

```
80:     public void warning(SAXParseException ex) throws SAXException {
81:         System.err.println(
82:             "[Warning] " + getLocationString(ex) + ": " +
83:             ex.getMessage());
84:     }
85:
86:     public void error(SAXParseException ex) throws SAXException {
87:         System.err.println(
88:             "[Error] " + getLocationString(ex) + ": " +
89:             ex.getMessage());
90:     }
91:
92:     public void fatalError(SAXParseException ex)
93:         throws SAXException {
94:         System.err.println(
95:             "[Fatal Error] " + getLocationString(ex) + ": " +
96:             ex.getMessage());
97:         throw ex;
98:     }
99:
100:    /** Returns a string of the location. */
101:    private String getLocationString(SAXParseException ex) {
102:        StringBuffer str = new StringBuffer();
103:
104:        String systemId = ex.getSystemId();
105:        if (systemId != null) {
106:            int index = systemId.lastIndexOf('/');
107:            if (index != -1)
108:                systemId = systemId.substring(index + 1);
109:            str.append(systemId);
110:        }
111:        str.append(':');
112:        str.append(ex.getLineNumber());
113:        str.append(':');
114:        str.append(ex.getColumnNumber());
115:
116:        return str.toString();
117:
118:    }
119:
120: }
```

DOM

Let's look at how you can accomplish the same task using the DOM API. The DOM API is a tree-based API. The parser provides the application with a tree-structured object graph, which the application can then traverse to extract the data from the parsed XML document. This process is more convenient than using SAX, but you pay a price in performance because the parser creates a DOM tree whether you're going to use it or not. If you're using XML to represent data in an application, the DOM tends to be inefficient because you have to get the data you need out of the DOM tree; after that you have no use for the DOM tree, even though the parser spent time and memory to construct it. We're going to reuse the class Book (in Book.java) for this example.

After importing all the necessary classes in lines 10-17, you declare a String constant whose value is the namespace URI for the book schema (lines 19-21):

```
 1: /*
 2:  *
 3:  * DOMMain.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11:
12: import org.apache.xerces.parsers.DOMParser;
13: import org.w3c.dom.Document;
14: import org.w3c.dom.Element;
15: import org.w3c.dom.Node;
16: import org.w3c.dom.NodeList;
17: import org.xml.sax.SAXException;
18:
19: public class DOMMain {
20:     static final String bookNS =
21:         "http://sauria.com/schemas/apache-xml-book/book";
22:
```

In line 24 you create a new DOMParser. Next you ask it to parse the document (line 27). At this point the parser has produced the DOM tree, and you need to obtain it and traverse it to extract the data you need to create a Book object (lines 27-29):

```
23:     public static void main(String args[]) {
24:         DOMParser p = new DOMParser();
25:
26:         try {
27:             p.parse(args[0]);
28:             Document d = p.getDocument();
29:             System.out.println(dom2Book(d).toString());
30:
31:         } catch (SAXException se) {
32:             System.out.println("Error during parsing " +
33:             se.getMessage());
34:             se.printStackTrace();
```

```
35:         } catch (IOException ioe) {
36:             System.out.println("I/O Error during parsing " +
37:             ioe.getMessage());
38:             ioe.printStackTrace();
39:         }
40:     }
41:
```

The dom2Book function creates the Book object:

```
42:     private static Book dom2Book(Document d) throws SAXException {
43:         NodeList nl = d.getElementsByTagNameNS(bookNS, "book");
44:         Element bookElt = null;
45:         Book book = null;
46:         try {
47:             if (nl.getLength() > 0) {
48:                 bookElt = (Element) nl.item(0);
49:                 book = new Book();
50:             } else
51:                 throw new SAXException("No book element found");
52:         } catch (ClassCastException cce) {
53:             throw new SAXException("No book element found");
54:         }
55:
```

In lines 43-54, you use the namespace-aware method getElementsByTagNameNS (as opposed to the non-namespace-aware getElementsByTagName) to find the root book element in the XML file. You check the resulting NodeList to make sure a book element was found before constructing a new Book instance.

Once you have the book element, you iterate through all the children of the book. These nodes in the DOM tree correspond to the child elements of the book element in the XML document. As you encounter each child element node, you need to get the text content for that element and call the appropriate Book setter. In the DOM, getting the text content for an element node is a little laborious. If an element node has text content, the element node has one or more children that are text nodes. The DOM provides a method called normalize that collapses multiple text nodes into a single text node where possible (normalize also removes empty text nodes where possible). Each time you process one of the children of the book element, you call normalize to collect all the text nodes and store the text content in the String text. Then you compare the tag name of the element you're processing and call the appropriate setter method. As with SAX, you have to convert the text to an integer for the Book's year field:

```
56:         for (Node child = bookElt.getFirstChild();
57:             child != null;
58:             child = child.getNextSibling()) {
59:             if (child.getNodeType() != Node.ELEMENT_NODE)
60:                 continue;
61:             Element e = (Element) child;
62:             e.normalize();
63:             String text = e.getFirstChild().getNodeValue();
64:
65:             if (e.getTagName().equals("title")) {
66:                 book.setTitle(text);
67:             } else if (e.getTagName().equals("author")) {
```

```
68:                 book.setAuthor(text);
69:             } else if (e.getTagName().equals("isbn")) {
70:                 book.setIsbn(text);
71:             } else if (e.getTagName().equals("month")) {
72:                 book.setMonth(text);
73:             } else if (e.getTagName().equals("year")) {
74:                 int y = 0;
75:                 try {
76:                     y = Integer.parseInt(text);
77:                 } catch (NumberFormatException nfe) {
78:                     throw new SAXException("Year must be a number");
79:                 }
80:                 book.setYear(y);
81:             } else if (e.getTagName().equals("publisher")) {
82:                 book.setPublisher(text);
83:             } else if (e.getTagName().equals("address")) {
84:                 book.setAddress(text);
85:             }
86:         }
87:         return book;
88:     }
89: }
```

This concludes our review of the SAX and DOM APIs. Now we're ready to go into the depths of Xerces.

Installing Xerces

Installing Xerces is relatively simple. The first thing you need to do is obtain a Xerces build. You can do this by going to http://xml.apache.org/dist/xerces-j, where you'll see a list of the current official Xerces builds. (You can ignore the Xerces 1.X builds.)

The Xerces build for a particular version of Xerces is divided into three distributions. Let's use Xerces 2.4.0 as an example. The binary distribution of Xerces 2.4.0 is in a file named Xerces-J-bin.2.4.0.*xxx*, where *xxx* is either .zip or .tar.gz, depending on the kind of compressed archive format you need. Typically, people on Windows use a .zip file, whereas people on MacOS X, Linux, and UNIX of various sorts use a .tar.gz file. There are also *.xxx*.sig files, which are detached PGP signatures of the corresponding *.xxx* file. So, Xerces-J-bin.2.4.zip.sig contains the signature file for the Xerces-J-bin2.4.zip distribution file. You can use PGP and the signature file to verify that the contents of the distribution have not been tampered with.

In addition to the binary distribution, you can download a source distribution, Xerces-J-src.2.4.zip, and a tools distribution, Xerces-J-tools-2.4.0.zip. You'll need the tools distribution in order to build the Xerces documentation.

We'll focus on installing the binary distribution. Once you've downloaded it, unpack it using a zip-file utility or tar and gzip for the .tar.gz files. Doing so creates a directory called xerces-2.4.0 in either the current directory or the directory you specified to your archiving utility. The key files in this directory are

- **data**—A directory containing sample XML files.
- **docs**—A directory containing all the documentation.
- **Readme.html**—The jump-off point for the Xerces documentation; open it with your Web browser.
- **samples**—A directory containing the source code for the samples.
- **xercesImpl.jar**—A jar file containing the parser implementation.
- **xercesSamples.jar**—A jar file containing the sample applications.
- **xml-apis.jar**—A jar file containing the parsing APIs (SAX, DOM, and so on).

You must include xml-apis.jar and xercesImpl.jar in your Java classpath in order to use Xerces in your application. There are a variety of ways to accomplish this, including setting the CLASSPATH environment variable in your DOS Command window or UNIX shell window. You can also set the CLASSPATH variable for the application server you're using.

Another installation option is to make Xerces the default XML parser for your JDK installation. This option only works for JDK 1.3 and above.

JDK 1.3 introduced an Extension Mechanism for the JDK. It works like this. The JDK installation includes a special extensions directory where you can place jar files that contain extensions to Java. If JAVA_HOME is the directory where your JDK has been installed, then the extensions directory is <JAVA_HOME>\jre\lib\ext using Windows file delimiters and <JAVA_HOME>/jre/lib/ext using UNIX file delimiters.

If you're using JDK 1.4 or above, you should use the Endorsed Standards Override Mechanism, not the Extension Mechanism. The JDK 1.4 Endorsed Standards Override Mechanism works like the Extension Mechanism, but it's specifically designed to allow incremental updates of packages specified by the JCP. The major operational difference between the Extension Mechanism and the Endorsed Standards Override Mechanism is that the directory name is different. The Windows directory is named <JAVA_HOME>\jre\lib\endorsed, and the UNIX directory is named <JAVA_HOME>/jre/lib/endorsed.

Development Techniques

Now that you have Xerces installed, let's look at some techniques for getting the most out of Xerces and XML. We're going to start by looking at how to set the Xerces configuration through the use of features and properties. We'll look at the Deferred DOM, which uses lazy evaluation to improve the memory usage of DOM trees in certain usage scenarios. There are two sections, each on how to deal with Schemas/Grammars and Entities. These are followed by a section on serialization, which is the job of producing XML as opposed to consuming it. We'll finish up by examining how the Xerces Native Interface (XNI) gives us access to capabilities that are not available through SAX or DOM.

Xerces Configuration

The first place we'll stop is the Xerces configuration mechanism. There are a variety of configuration settings for Xerces, so you'll need to be able to turn these settings on and off.

Xerces uses the SAX features and properties mechanism to control all configuration settings. This is true whether you're using Xerces as a SAX parser or as a DOM parser. The class org.apache.xerces .parsers.DOMParser provides the methods setFeature, getFeature, setProperty, and getProperty, which are available on the class org.xml.sax.XMLReader. These methods all accept a String as the name of the feature or property. The convention for this API is that the name is a URI that determines the feature or property of interest. Features are boolean valued, and properties are object valued. The SAX specification defines a standard set of feature and property names, and Xerces goes on to define its own. All the Xerces feature/property URIs are in the http://apache.org/xml URI space under either features or properties. These URI's function in the same ways as Namespace URI's. They don't refer to anything—they are simply used to provide an extensible mechanism for defining unique names for features.

The configuration story is complicated when the JAXP (Java API for XML Parsing) APIs come into the picture. The purpose of JAXP is to abstract the specifics of parser instantiation and configuration from your application. In general, this is a desirable thing because it means your application doesn't depend on a particular XML parser. Unfortunately, in practice, this can mean you no longer have access to useful functionality that hasn't been standardized via the JCP. This is especially true in the case of parser configuration. If you're using the SAX API, you don't have much to worry about, because you can pass the Xerces features to the SAX setFeature and setProperty methods, and everything will be fine. The problem arises when you want to use the DOM APIs. Up until DOM Level 3, the DOM API didn't provide a mechanism for configuring options to a DOM parser, and even the mechanism described in DOM Level 3 isn't sufficient for describing all the options Xerces allows. The JAXP API for DOM uses a factory class called DOMBuilder to give you a parser that can parse an XML document and produce a DOM. However, it doesn't have the setFeature and set Property methods that you need to control Xerces-specific features. For the foreseeable future, if you want to use some of the features we'll be talking about, you'll have to use the Xerces DOMParser object to create a DOM API parser.

Validation-Related Features

A group of features relate to validation. The first of these is http://apache.org/xml/features /validation/dynamic. When this feature is on, Xerces adopts a laissez faire method of processing XML documents. If the document provides a DTD or schema, Xerces uses it to validate the document. If no grammar is provided, Xerces doesn't validate the document. Ordinarily, if Xerces is in validation mode, the document must provide a grammar of some kind; in non-validating mode, Xerces doesn't perform validation even if a grammar is present.

Most people think there are two modes for XML parsers—validating and non-validating—on the assumption that non-validating mode just means not doing validation. The reality is more complicated. According to the XML 1.0 specification (Section 5 has all the gory details), there is a range of things an XML parser may or may not do when it's operating in non-validating mode. The list of optional tasks includes attribute value normalization, replacement of internal text entities, and attribute defaulting. Xerces has a pair of features designed to make its behavior in non-validating mode slightly more predictable. You can prevent Xerces from reading an external DTD if it's in non-validating mode, using the http://apache.org/xml/features/nonvalidating/load-external-dtd* feature. This means the parsed document will be affected only by definitions from an internal DTD subset (a DTD in the document). It's also possible to tell Xerces not to use the DTD to default attribute values or to compute their types. The feature you use to do this is http://apache.org/xml/features/nonvalidating/load-dtd-grammar.

Error-Reporting Features

The next set of features controls the kinds of errors that Xerces reports. The feature http://apache.org/xml/features/warn-on-duplicate-entitydef generates a warning if an entity definition is duplicated. When validation is turned on, http://apache.org/xml/features/validation/warn-on-duplicate-attdef causes Xerces to generate a warning if an attribute declaration is repeated. Similarly, http://apache.org/xml/features/validation/warn-on-undeclared-elemdef causes Xerces to generate a warning if a content model references an element that has not been declared. All three of these properties are provided to help generate more user-friendly error messages when validation fails.

DOM-Related Features and Properties

Three features or properties affect Xerces when you're using the DOM API. To understand the first one, we have to make a slight digression onto the topic of ignorable whitespace.

Ignorable whitespace is the whitespace characters that occur between the end of one element and the start of another. This whitespace is used to format XML documents to make them more readable. Here is the book example with the ignorable whitespace shown in gray:

```
 1: <?xml version="1.0" encoding="UTF-8"?>¶
 2: <book xmlns="http://sauria.com/schemas/apache-xml-book/book"
 3:   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4:   xsi:schemaLocation=
 5:    "http://sauria.com/schemas/apache-xml-book/book
 6:     http://www.sauria.com/schemas/apache-xml-book/book.xsd"
 7:   version="1.0">¶
 8:  <title>XML Development with Apache Tools</title>¶
 9:  <author>Theodore W. Leung</author>¶
10:  <isbn>0-7645-4355-5</isbn>¶
11:  <month>December</month>¶
12:  <year>2003</year>¶
13:  <publisher>Wrox</publisher>¶
14:  <address>Indianapolis, Indiana</address>¶
15: </book>
```

An XML parser can only determine that whitespace is ignorable when it's validating. The SAX API makes the notion of ignorable whitespace explicit by providing different callbacks for characters and ignorableWhitespace. The DOM API doesn't have any notion of this concept. A DOM parser must create a DOM tree that represents the document that was parsed. The Xerces feature http://apache.org/xml/features/dom/include-ignorable-whitespace allows you control whether Xerces creates text nodes for ignorable whitespace. If the feature is false, then Xerces won't create text nodes for ignorable whitespace. This can save a sizable amount of memory for XML documents that have been pretty-printed or highly indented.

Frequently we're asked if it's possible to supply a custom DOM implementation instead of the one provided with Xerces. Doing this is a fairly large amount of work. The starting point is the property http://apache.org/xml/properties/dom/document-class-name, which allows you to set the name of the class to be used as the factory class for all DOM objects. If you replace the built-in Xerces DOM with your own DOM, then any Xerces-specific DOM features, such as deferred node expansion, are disabled, because they are all implemented within the Xerces DOM.

Xerces uses the SAX ErrorHandler interface to handle errors while parsing using the DOM API. You can register your own ErrorHandler and customize your error reporting, just as with SAX. However, you may want to access the DOM node that was under construction when the error condition occurred. To do this, you can use the http://apache.org/xml/properties/dom/current-element-node to read the DOM node that was being constructed at the time the parser signaled an error.

Other Features and Properties

Xerces uses an input buffer that defaults to 2KB in size. The size of this buffer is controlled by the property http://apache.org/xml/properties/input-buffer-size. If you know you'll be dealing with files within a certain size range, it can help performance to set the buffer size close to the size of the files you're working with. The buffer size should be a multiple of 1KB. The largest value you should set this property to is 16KB.

Xerces normally operates in a mode that makes it more convenient for users of Windows operating systems to specify filenames. In this mode, Xerces allows URIs (Uniform Resource Identifiers) to include file specifications that include backslashes (\) as separators, and allows the use of DOS drive letters and Windows UNC filenames. Although this is convenient, it can lead to sloppiness, because document authors may include these file specifications in XML documents and DTDs. The http://apache.org/xml/features/standard-uri-conformant feature turns off this convenience mode and requires that all URIs actually be URIs.

The XML 1.0 specification recommends that the character encoding of an XML file should be specified using a character set name specified by the Internet Assigned Numbers Authority (IANA). However, this isn't required. The feature http://apache.org/xml/features/allow-java-encodings allows you to use the Java names for character encodings to specify the character set encoding for a document. This feature can be convenient for an all-Java system, but it's completely non-interoperable with non-Java based XML parsers.

Turning on the feature http://apache.org/xml/features/disallow-doctype-decl causes Xerces to throw an exception when a DTD is provided with an XML document. It's possible to launch a denial-of-service attack against an XML parser by providing a DTD that contains a recursively expanding entity definition, and eventually the entity expansion overflows some buffer in the parser or causes the parser to consume all available memory. This feature can be used to prevent this attack. Of course, DTD validation can't be used when this flag is turned on, and Xerces is operating in a mode that isn't completely compliant with the XML specification.

Unfortunately, there are other ways to launch denial-of-service attacks against XML parsers, so the Xerces team has created a SecurityManager class that is part of the org.apache.xerces.util package. The current security manager can be accessed via the http://apache.org/xml/properties/security-manager property. It lets you replace the security manager with your own by setting the value of the property to an instance of SecurityManager. At the time of this writing, SecurityManager provides two JavaBean properties, entityExpansionLimit and maxOccurNodeLimit Setting entityExpansionLimit is another way to prevent the entity expansion attack. The value of this property is the number of entity expansions the parser should allow in a single document. The default value for entityExpansionLimit is 100,000. The maxOccurNodeLimit property controls the maximum number of occur nodes that can be created for an XML Schema maxOccurs. This is for the case where maxOccurs is a number, not unbounded. The default value for this property is 3,000.

Deferred DOM

One of the primary difficulties with using the DOM API is performance. This issue manifests itself in a number of ways. The DOM's representation of an XML document is very detailed and involves a lot of objects. This has a big impact on performance because of the time it takes to create all those objects, and because of the amount of memory those objects use. Developers are often surprised to see how much memory an XML document consumes when it's represented as a DOM tree.

To reduce the overhead of using the DOM in an application, the Xerces developers implemented what is called *deferred node expansion*. This is an application of lazy evaluation techniques to the creation of DOM trees. When deferred node expansion is turned on, Xerces doesn't create objects to represent the various parts of an XML document. Instead, it builds a non-object oriented set of data structures that contain the information needed to create the various types of DOM nodes required by the DOM specification. This allows Xerces to complete parsing in a much shorter time than when deferred node expansion is turned off. Because almost no objects are created, the memory used is a fraction of what would ordinarily be used by a DOM tree.

The magic starts when your application calls the appropriate method to get the DOM Document node. Deferred node expansion defers the creation of DOM node objects until your program needs them. The way it does so is simple: If your program calls a DOM method that accesses a node in the DOM tree, the deferred DOM implementation creates the DOM node you're requesting and all of its children. Obviously, the deferred DOM implementation won't create a node if it already exists. A finite amount of work is done on each access to an unexpanded node.

The deferred DOM is especially useful in situations where you're not going to access every part of a document. Because it only expands those nodes (and the fringe defined by their children) that you access, Xerces doesn't create all the objects the DOM specification says should be created. This is fine, because you don't need the nodes you didn't access. The result is a savings of memory and processor time (spent creating objects and allocating memory).

If your application is doing complete traversals of the entire DOM tree, then you're better off not using the deferred DOM, because you'll pay the cost of creating the non-object-oriented data structures plus the cost of creating the DOM objects as you access them. This results in using more memory and processor time than necessary.

The deferred DOM implementation is used by default. If you wish to turn it off, you can set the feature http://apache.org/xml/features/dom/defer-node-expansion to false. If you're using the JAXP DocumentBuilder API to get a DOM parser, then the deferred DOM is turned off.

Schema Handling

Xerces provides a number of features that control various aspects of validation when you're using XML Schema. The most important feature turns on schema validation: http://apache.org/xml/features/validation/schema. To use it, the SAX name-spaces property (http://xml.org/sax/features/namespaces) must be on (it is by default). The Xerces validator won't report schema validation errors unless the regular SAX validation feature (http://xml.org/sax/features/validation) is turned on, so you must make sure that both the schema validation feature and the SAX validation feature are set to true.

Here's the SAXMain program, enhanced to perform schema validation:

```
 1: /*
 2:  *
 3:  * SchemaValidateMain.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11:
12: import org.apache.xerces.parsers.SAXParser;
13: import org.xml.sax.EntityResolver;
14: import org.xml.sax.SAXException;
15: import org.xml.sax.SAXNotRecognizedException;
16: import org.xml.sax.SAXNotSupportedException;
17: import org.xml.sax.XMLReader;
18:
19: public class SchemaValidateMain {
20:
21:     public static void main(String[] args) {
22:         XMLReader r = new SAXParser();
23:         try {
24:             r.setFeature("http://xml.org/sax/features/validation",
25:              true);
26:             r.setFeature(
27:                 "http://apache.org/xml/features/validation/schema",
28:                  true);
29:         } catch (SAXNotRecognizedException snre) {
30:             snre.printStackTrace();
31:         } catch (SAXNotSupportedException snre) {
32:             snre.printStackTrace();
33:         }
34:         BookHandler bookHandler = new BookHandler();
35:         r.setContentHandler(bookHandler);
36:         r.setErrorHandler(bookHandler);
37:         EntityResolver bookResolver = new BookResolver();
38:         r.setEntityResolver(bookResolver);
39:         try {
40:             r.parse(args[0]);
41:             System.out.println(bookHandler.getBook().toString());
42:         } catch (SAXException se) {
43:             System.out.println("SAX Error during parsing " +
44:                 se.getMessage());
45:             se.printStackTrace();
46:         } catch (IOException ioe) {
47:             System.out.println("I/O Error during parsing " +
48:                 ioe.getMessage());
49:             ioe.printStackTrace();
50:         } catch (Exception e) {
51:             System.out.println("Error during parsing " +
52:                 e.getMessage());
```

```
53:             e.printStackTrace();
54:         }
55:     }
56: }
```

Additional Schema Checking

The feature http://apache.org/xml/features/validation/schema-full-checking turns on additional checking for schema documents. This doesn't affect documents using the schema but does more thorough checking of the schema document itself, in particular particle unique attribute constraint checking and particle derivation restriction checks. This feature is normally set to false because these checks are resource intensive.

Schema-Normalized Values

Element content is also normalized when you validate with XML Schema (only attribute values were normalized in XML 1.0). The reason is that simple types can be used as both element content and attribute values, so element content must be treated the same as attribute values in order to obtain the same semantics for simple types. In Xerces, the feature http://apache.org/xml/features/validation/schema/normalized-value controls whether SAX and DOM see the Schema-normalized values of elements and attributes or the XML 1.0 infoset values of elements and attributes. If you're validating with XML Schema, this feature is normally turned on.

Reporting Default Values

In XML Schema, elements and attributes are similar in another way: They can both have default values. The question then arises, how should default values be reported to the application? Should the parser assume the application knows what the default value is, or should the parser provide the default value to the application? The only downside to the parser providing the default value is that if the application knows what the default value is, the parser is doing unnecessary work. The Xerces feature http://apache.org/xml/features/validation/schema/element-default allows you to choose whether the parser reports the default value. The default setting for this feature is to report default values. Default values are reported via the characters callback, just like any other character data.

Accessing PSVI

Some applications want to access the Post Schema Validation Infoset (PSVI) in order to obtain type information about elements and attributes. The Xerces API for accomplishing this has not yet solidified, but it exists in an experimental form in the org.apache.xerces.xni.psvi package. If your application isn't accessing the PSVI, then you should set the feature http://apache.org/xml/features/validation/schema/augment-psvi to false so you don't have to pay the cost of creating the PSVI augmentations.

Overriding schemaLocation Hints

The XML Schema specification says that the xsi:schemaLocation and xsi:noNamespaceSchemaLocation attributes are hints to the validation engine and that they may be ignored. There are at least two good reasons your application might want to ignore these hints. First, you shouldn't believe a document that purports to tell your application what schema it should use to validate the document. When you wrote your application, you had a particular version of an XML Schema in mind. The incoming document is supposed to conform to that schema. But a number of problems can crop up if you believe the incoming document when it claims to know what schema to use. The author of the incoming document may have

used a different or buggy version of the schema you're using. Worse, the author of the incoming document may intentionally specify a different version of the schema in an attempt to subvert your application.

The second reason you may choose to ignore these hints is that you might want to provide a local copy of the schema so the validator doesn't have to perform a network fetch of the schema document every time it has to validate a document. If you're in a server environment processing thousands or even millions of documents per day, the last thing you want is for the Xerces validator to be doing an HTTP request to a machine somewhere on the Internet for each document it has to validate. Not only is this terrible for performance, but it makes your application susceptible to a failure of the machine hosting the schema. Fortunately, Xerces has a pair of properties you can use to override the schemaLocation hints. The first property is http://apache.org/xml/properties/schema/external-schemaLocation; it overrides the xsi:schemaLocation attribute. The value of the property is a string that has the same format as the xsi:schemaLocation attribute: a set of pairs of namespace URIs and schema document URIs. The other property is http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation; it handles the xsi:noNamespaceSchemaLocation case. Its value has the same format as xsi:noNamespaceSchemaLocation, a single URI with the location of the schema document.

Grammar Caching

If you're processing a large number of XML documents that use a single DTD, a single XML schema, or a small number of XML schemas, you should use the grammar-caching functionality built in to Xerces. You can use the http://apache.org/xml/properties/schema/external-schemaLocation or http://apache.org/xml/properties/schema/external-noNamespaceSchemaLocation properties to force Xerces to read XML schemas from a local copy, which improves the efficiency of your application. However, these properties work at an entity level (in a later section, you'll discover that you could use entity-handling techniques to accomplish what these two properties do).

Even if you're reading the grammar from a local file, Xerces still has to read the grammar file and turn it into data structures that can be used to validate an XML document, a process somewhat akin to compilation. This process is very costly. If your application uses a single grammar or a small fixed number of grammars, you would like to avoid the overhead of processing the grammar multiple times. That's the purpose of the Xerces grammar-caching functionality.

Xerces provide two styles of grammar caching: *passive caching* and *active caching*. Passive caching requires little work on the part of your application. You set a property, and Xerces starts caching grammars. When Xerces encounters a grammar that it hasn't seen before, it processes the grammar and then caches the grammar data structures for reuse. The next time Xerces encounters a reference to this grammar, it uses the cached data structures.

Here's a version of the book-processing program that uses passive grammar caching:

```
1: /*
2:  *
3:  * PassiveSchemaCache.java
4:  *
5:  * Example from "Professional XML Development with Apache Tools"
6:  *
7:  */
8: package com.sauria.apachexml.ch1;
```

```
 9: import java.io.IOException;
10:
11: import org.apache.xerces.parsers.SAXParser;
12: import org.xml.sax.SAXException;
13: import org.xml.sax.SAXNotRecognizedException;
14: import org.xml.sax.SAXNotSupportedException;
15: import org.xml.sax.XMLReader;
16:
17: public class PassiveSchemaCache {
18:
19:     public static void main(String[] args) {
20:         System.setProperty(
21:          "org.apache.xerces.xni.parser.Configuration",
22:          "org.apache.xerces.parsers.XMLGrammarCachingConfiguration");
```

Lines 20-22 contain the code that turns on passive grammar caching. All you have to do is set the Java property org.apache.xerces.xni.parser.Configuration to a configuration that understands grammar caching. One such configuration is org.apache.xerces.parsers.XMLGrammarCachingConfiguration. After that, the code is essentially the same as what you are used to seeing. This shows how easy it is to use passive grammar caching. Add three lines and you're done.

```
23:
24:         XMLReader r = new SAXParser();
25:         try {
26:             r.setFeature("http://xml.org/sax/features/validation",
27:              true);
28:             r.setFeature(
29:                 "http://apache.org/xml/features/validation/schema",
30:                  true);
31:         } catch (SAXNotRecognizedException snre) {
32:             snre.printStackTrace();
33:         } catch (SAXNotSupportedException snre) {
34:             snre.printStackTrace();
35:         }
36:         BookHandler bookHandler = new BookHandler();
37:         r.setContentHandler(bookHandler);
38:         r.setErrorHandler(bookHandler);
39:
40:         for (int i = 0; i < 5; i++)
41:             try {
42:                 r.parse(args[0]);
43:                 System.out.println(bookHandler.getBook().toString());
44:             } catch (SAXException se) {
45:                 System.out.println("SAX Error during parsing " +
46:                     se.getMessage());
47:                 se.printStackTrace();
48:             } catch (IOException ioe) {
49:                 System.out.println("I/O Error during parsing " +
50:                     ioe.getMessage());
51:                 ioe.printStackTrace();
52:             } catch (Exception e) {
53:                 System.out.println("Error during parsing " +
54:                     e.getMessage());
55:                 e.printStackTrace();
```

```
56:             }
57:     }
58:
59: }
```

Although passive caching is easy to use, it has one major drawback: You can't specify which grammars Xerces can cache. When you're using passive caching, Xerces happily caches any grammar it finds in any document. If you're processing a high volume of documents, let's say purchase orders, then you probably are using only one grammar, and you probably don't want the author of those purchase order documents to be the one who determines which grammar file is used (and possibly cached).

The solution to this problem is to use active grammar caching. Active grammar caching requires you to do more work in your application, but in general it's worth it because you get complete control over which grammars can be cached, as well as control over exactly which grammar files are used to populate the grammar caches.

When you're using active caching, you need to follow two steps. First, you create a grammar cache (an instance of org.apache.xerces.util.XMLGrammarPoolImpl) and load it by pre-parsing all the grammar files you want to cache. Then you call Xerces and make sure it's using the cache you just created.

Here's a program that makes use of active caching:

```
 1: /*
 2:  *
 3:  * ActiveSchemaCache.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9: import java.io.IOException;
10:
11: import org.apache.xerces.impl.Constants;
12: import org.apache.xerces.parsers.SAXParser;
13: import org.apache.xerces.parsers.StandardParserConfiguration;
14: import org.apache.xerces.parsers.XMLGrammarPreparser;
15: import org.apache.xerces.util.SymbolTable;
16: import org.apache.xerces.util.XMLGrammarPoolImpl;
17: import org.apache.xerces.xni.XNIException;
18: import org.apache.xerces.xni.grammars.Grammar;
19: import org.apache.xerces.xni.grammars.XMLGrammarDescription;
20: import org.apache.xerces.xni.parser.XMLConfigurationException;
21: import org.apache.xerces.xni.parser.XMLInputSource;
22: import org.apache.xerces.xni.parser.XMLParserConfiguration;
23: import org.xml.sax.SAXException;
24: import org.xml.sax.XMLReader;
25:
26:
27: public class ActiveSchemaCache {
28:     static final String SYMBOL_TABLE =
29:         Constants.XERCES_PROPERTY_PREFIX +
30:         Constants.SYMBOL_TABLE_PROPERTY;
31:
```

```
32:     static final String GRAMMAR_POOL =
33:         Constants.XERCES_PROPERTY_PREFIX +
34:         Constants.XMLGRAMMAR_POOL_PROPERTY;
35:
36:     SymbolTable sym = null;
37:     XMLGrammarPoolImpl grammarPool = null;
38:     XMLReader reader = null;
39:
40:    public void loadCache() {
41:         grammarPool = new XMLGrammarPoolImpl();
42:         XMLGrammarPreparser preparser = new XMLGrammarPreparser();
43:         preparser.registerPreparser(XMLGrammarDescription.XML_SCHEMA,
44:             null);
45:         preparser.setProperty(GRAMMAR_POOL, grammarPool);
46:         preparser.setFeature(
47:             "http://xml.org/sax/features/validation",
48:             true);
49:         preparser.setFeature(
50:             "http://apache.org/xml/features/validation/schema",
51:             true);
52:         // parse the grammar...
53:
54:         try {
55:             Grammar g =
56:                 preparser.preparseGrammar(
57:                     XMLGrammarDescription.XML_SCHEMA,
58:                     new XMLInputSource(null, "book.xsd", null));
59:         } catch (XNIException xe) {
60:             xe.printStackTrace();
61:         } catch (IOException ioe) {
62:             ioe.printStackTrace();
63:         }
64:
65:     }
66:
```

The loadCache method takes care of creating the data structures needed to cache grammars. The cache itself is an instance of org.apache.xerces.util.XMLGrammarPoolImpl, created in line 41. The object that knows the workflow of how to preprocess a grammar file is an instance of XMLGrammarPreparser, so in line 42 you create an instance of XMLGrammarPreparser.

XMLGrammarPreparsers need to know which kind of grammar they will be dealing with. They have a method called registerPreparser that allows them to associate a string (representing URIs for particular grammars) with an object that knows how to preprocess a specific type of grammar. This means a single XMLGrammarPreparser can preprocess multiple types of grammars (for example, both DTDs and XML schemas). In this example, you're only interested in allowing XML schemas to be cached, so you register XML schemas with the preparser (lines 43-44). If you're registering either XML schemas or DTDs with a preparser, then you can pass null as the second argument to registerPreparser. Otherwise, you have to provide an instance of org,apache.xerces.xni.grammarsXMLGrammarLoader, which can process the grammar you're registering.

Now you're ready to associate a grammar pool with the preparser. This is done using the preparser's setProperty method and supplying the appropriate values (line 45). XMLGrammarPreparser provides a feature/property API like the regular SAX and DOM parsers in Xerces. The difference is that when you set a feature or property on an instance of XMLGrammarPreparser, you're actually setting the feature or property on all XMLGrammarLoader instances that have been registered with the preparser. So the next two setFeature calls (in lines 46-51) tell all registered XMLGrammarLoaders to validate their inputs and to do so using XML Schema if possible. Note that implementers of XMLGrammarLoader aren't required to implement any features or properties (just as with SAX features and properties).

Once all the configuration steps are complete, all that is left to do is to call the preparseGrammar method for all the grammars you want loaded into the cache. Note that you need to use the XMLInputSource class from org.apache.xni.parser to specify how to get the grammar file. This all happens in lines 54-63.

How do you make use of a loaded cache? It turns out to be fairly simple, but it means a more circuitous route to creating a parser. The XMLParserConfiguration interface has a setProperty method that accepts a property named http://apache.org/xml/properties/internal/grammar-pool, whose value is a grammar pool the parser configuration should use. The constructors for the various Xerces parser classes can take an XMLParserConfiguration as an argument. So, you need to get hold of a parser configuration, set the grammar pool property of that configuration to the grammar pool that loadCache created, and then create a SAX or DOM parser based on that configuration. Pretty straightforward, right?

The first thing you need is an XMLParserConfiguration. You can use the Xerces supplied org.apache.xerces.parsers.StandardParserConfiguration because you aren't doing anything else fancy:

```
67:     public synchronized Book useCache(String uri) {
68:         Book book = null;
69:         XMLParserConfiguration parserConfiguration =
70:             new StandardParserConfiguration();
```

Next you need to set the grammar pool property on the parserConfiguration to be the grammarPool created by loadCache:

```
71:
72:         String grammarPoolProperty =
73:             "http://apache.org/xml/properties/internal/grammar-pool";
74:         try {
75:             parserConfiguration.setProperty(grammarPoolProperty,
76:                 grammarPool);
```

In this example you're using a SAX parser to process documents. The constructor for the Xerces SAX parser takes an XMLParserConfiguration as an argument, so you just pass the parserConfiguration as the argument, and now you have a SAXParser that's using the grammar cache!

```
77:             parserConfiguration.setFeature(
78:                 "http://xml.org/sax/features/validation",
79:                 true);
80:             parserConfiguration.setFeature(
81:                 "http://apache.org/xml/features/validation/schema",
82:                 true);
83:         } catch (XMLConfigurationException xce) {
84:             xce.printStackTrace();
```

```
85:         }
86:
87:         try {
88:             if (reader == null)
89:                 reader = new SAXParser(parserConfiguration);
```

Something else is going on here: each instance of ActiveCache has a single SAXParser instance associated with it. You create an instance of SAXParser only if one doesn't already exist. This cuts down on the overhead of setting up and tearing down parser instances all the time.

One other detail. When you reuse a Xerces parser instance, you need to call the reset method in between usages. Doing so ensures that the parser is ready to parse another document:

```
90:             BookHandler bookHandler = new BookHandler();
91:             reader.setContentHandler(bookHandler);
92:             reader.setErrorHandler(bookHandler);
93:             reader.parse(uri);
94:             book = bookHandler.getBook();
95:             ((org.apache.xerces.parsers.SAXParser)reader).reset();
96:         } catch (IOException ioe) {
97:             ioe.printStackTrace();
98:         } catch (SAXException se) {
99:             se.printStackTrace();
100:        }
101:        return book;
102:    }
103:
104:    public static void main(String[] args) {
105:      ActiveSchemaCache c = new ActiveSchemaCache();
106:      c.loadCache();
107:        for (int i = 0; i < 5; i++) {
108:            Book b = c.useCache("book.xml");
109:            System.out.println(b.toString());
110:        }
111:
112:      }
113: }
```

The Xerces grammar-caching implementation uses hashing to determine whether two grammars are the same. If the two grammars are XML schemas, then they are hashed according to their targetNamespace. If the targetNamespaces are the same, the grammars are considered to be the same. For DTDs, it's more complicated. There are three conditions:

- If their publicId or expanded SystemIds exist, they must be identical.
- If one DTD defines a root element, it must either be the same as the root element of the second DTD, or it must be a global element in the second DTD.
- If neither DTD defines a root element, they must share a global element between the two of them.

If you're using the grammar-caching mechanism to cache DTDs, be aware that it can only cache external DTD subsets (DTDs in an external file). In addition, any definitions in an internal DTD subset (DTD within the document) will be ignored.

Entity Handling

Earlier in the chapter we mentioned that we'd be looking at a mechanism that can do the same job as the Xerces properties for xsi:schemaLocation and xsi:noNamespaceSchemaLocation. That mechanism is the SAX entity resolver mechanism. Although it isn't Xerces specific, it's very useful, because all external files are accessed as entities in XML. The entity resolver mechanism lets you install a callback that is run at the point where the XML parser tries to resolve an entity from an ID into a physical storage unit (whether that unit is on disk, in memory, or off on the network somewhere). You can use the entity resolver mechanism to force all references to a particular entity to be resolved to a local copy instead of a network copy, which simultaneously provides a performance improvement and gives you control over the actual definition of the entities.

Let's look at how to extend the example program to use an entity resolver:

```
 1: /*
 2:  *
 3:  * EntityResolverMain.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11:
12: import org.apache.xerces.parsers.SAXParser;
13: import org.xml.sax.EntityResolver;
14: import org.xml.sax.SAXException;
15: import org.xml.sax.SAXNotRecognizedException;
16: import org.xml.sax.SAXNotSupportedException;
17: import org.xml.sax.XMLReader;
18:
19: public class EntityResolverMain {
20:
21:     public static void main(String[] args) {
22:         XMLReader r = new SAXParser();
23:         try {
24:             r.setFeature("http://xml.org/sax/features/validation",
25:                 true);
26:             r.setFeature(
27:                 "http://apache.org/xml/features/validation/schema",
28:                 true);
29:         } catch (SAXNotRecognizedException e1) {
30:             e1.printStackTrace();
31:         } catch (SAXNotSupportedException e1) {
32:             e1.printStackTrace();
33:         }
34:         BookHandler bookHandler = new BookHandler();
```

```
35:         r.setContentHandler(bookHandler);
36:         r.setErrorHandler(bookHandler);
37:         EntityResolver bookResolver = new BookResolver();
38:         r.setEntityResolver(bookResolver);
```

The EntityResolver interface originated in SAX, but it's also used by the Xerces DOM parser and by the JAXP DocumentBuilder. All you need to do to make it work is create an instance of a class that implements the org.xml.sax.EntityResolver interface and then pass that object to the setEntityResolver method on XMLReader, SAXParser, DOMParser, or DocumentBuilder.

```
39:         try {
40:             r.parse(args[0]);
41:             System.out.println(bookHandler.getBook().toString());
42:         } catch (SAXException se) {
43:             System.out.println("SAX Error during parsing " +
44:                 se.getMessage());
45:             se.printStackTrace();
46:         } catch (IOException ioe) {
47:             System.out.println("I/O Error during parsing " +
48:                 ioe.getMessage());
49:             ioe.printStackTrace();
50:         } catch (Exception e) {
51:             System.out.println("Error during parsing " +
52:                 e.getMessage());
53:             e.printStackTrace();
54:         }
55:     }
56: }
```

The real work happens in a class that implements the EntityResolver interface. This is a simple interface with only one method, resolveEntity. This method tries to take an entity that is identified by a Public ID, System ID, or both, and provide an InputSource the parser can use to grab the contents of the entity:

```
 1: /*
 2:  *
 3:  * BookResolver.java
 4:  *
 5:  * This file is part of the "Apache XML Tools" Book
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch2;
 9:
10: import java.io.FileReader;
11: import java.io.IOException;
12:
13: import org.xml.sax.EntityResolver;
14: import org.xml.sax.InputSource;
15: import org.xml.sax.SAXException;
16:
17: public class BookResolver implements EntityResolver {
18:     String schemaURI =
19:         "http://www.sauria.com/schemas/apache-xml-book/book.xsd";
20:
```

```
21:     public InputSource resolveEntity(String publicId,
22:         String systemId)
23:         throws SAXException, IOException {
24:         if (systemId.equals(schemaURI)) {
25:             FileReader r = new FileReader("book.xsd");
26:             return new InputSource(r);
27:         } else
28:             return null;
29:     }
30:
31: }
```

The general flow of a resolveEntity method is to look at the publicId and/or systemId arguments and decide what you want to do. Once you've made your decision, your code then accesses the physical storage (in this case, a file) and wraps it up in an InputSource for the rest of the parser to use. In this example, you're looking for the systemId of the book schema (which is the URI supplied in the xsi:schemaLocation hint). If the entity being resolved is the book schema, then you read the schema from a local copy, wrap the resulting FileReader in an InputSource, and hand it back.

You could do a variety of things in your resolveEntity method. Instead of storing entities in the local file system, you could store them in a database and use JDBC to retrieve them. You could store them in a content management system or an LDAP directory, as well. If you were reading a lot of large text entities over and over again, you could build a cache inside your entity resolver so the entities were read only once and after that were read from the cache.

Remember, though, at this level you're dealing with caching the physical storage structures, not logical structures they might contain. Even if you use the EntityResolver mechanism in preference to Xerces' xsi:schemaLocation overrides, you still aren't getting as much bang for your buck as if you use the grammar-caching mechanism. At entity-resolver time, you're caching the physical storage and saving physical retrieval costs. At grammar-caching time, you're saving the cost of converting from a physical to a logical representation. If you're going to do logical caching of grammars, it doesn't make much sense to do physical caching of the grammar files. There are plenty of non-grammar uses of entities, and these are all fair game for speedups via the entity resolver mechanism.

Entity References

In most cases, entities should be invisible to your application—it doesn't matter whether the content in a particular section of an XML document came from the main document entity, an internal entity, or an entity stored in a separate file. Sometimes your application does want to know, particularly if your application is something like an XML editor, which is trying to preserve the input document as much as possible.

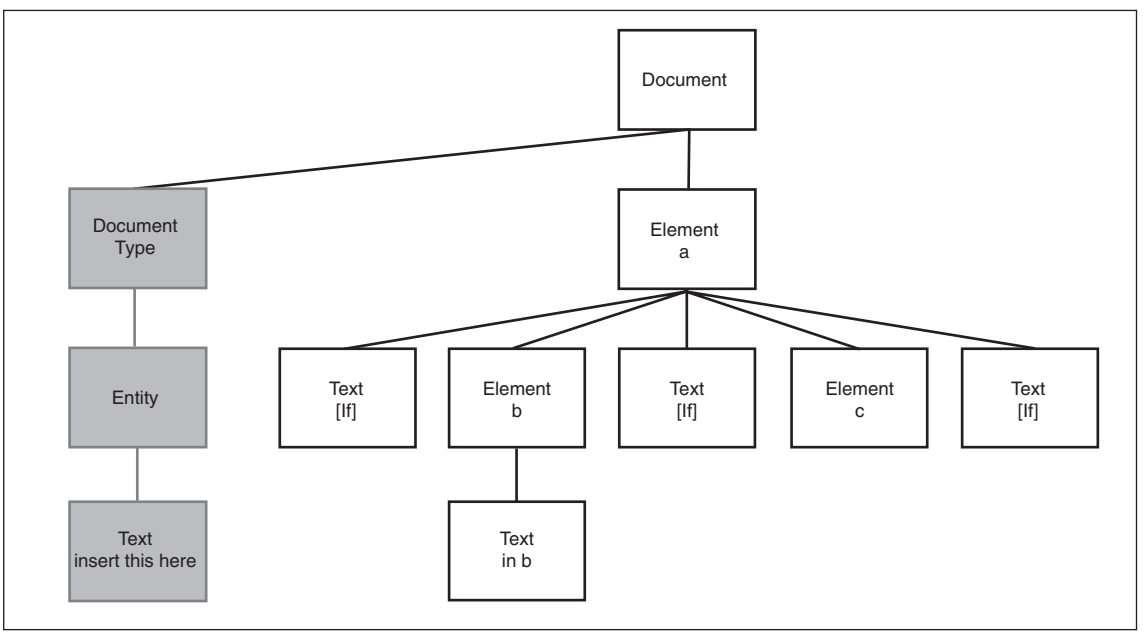
SAX provides the org.xml.sax.ext.LexicalHandler extension interface, which you can use to get callbacks about events you don't get via the ContentHandler callbacks. Among these callbacks are startEntity and endEntity, which are called at the start and end of any entity (internal or external) in the document. Ordinarily, startEntity and endEntity only report general entities and parameter entities (SAX says a parser doesn't have to report parameter entities, but Xerces does). Sometimes you'd like to know other details about the exact physical representation of a document, such as whether one of the built-in entities (&, >, <, ", or ') was used, or whether a character reference (&#XXXX) was used.

Xerces provides two features that cause startEntity and endEntity to report the beginning and end of these two classes of entity references. The feature http://apache.org/xml/features/scanner/notify-builtin-refs causes startEntity and endEntity to report the start and end of one of the built-in entities, and the feature http://apache.org/xml/features/scanner/notify-char-refs makes startEntity and endEntity report the start and end of a character reference.

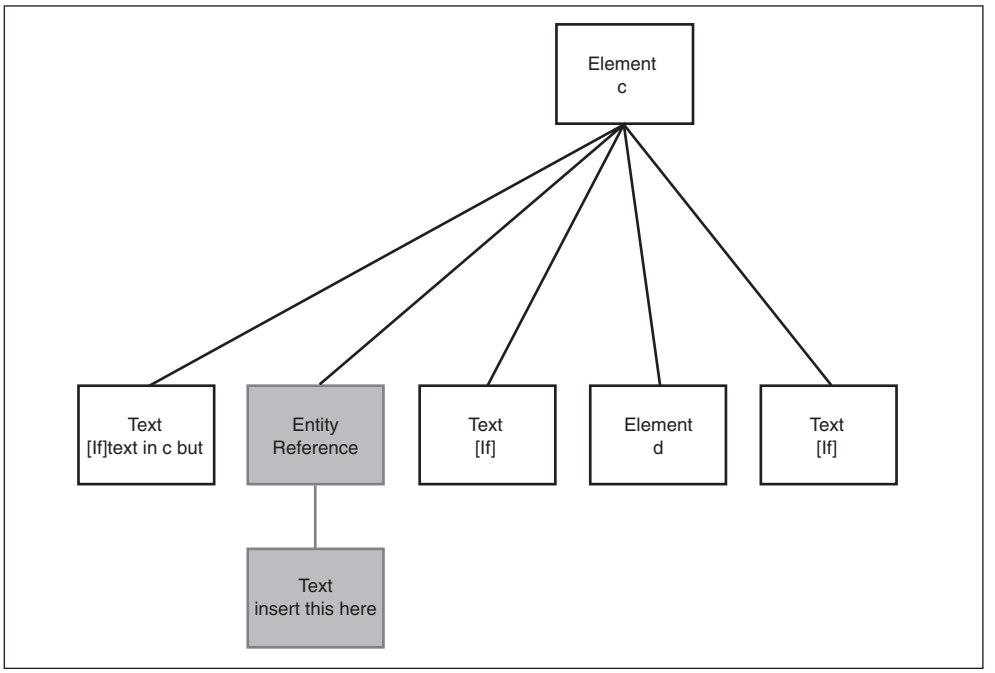
The DOM has its own challenges when dealing with entities. Consider this XML file:

```
 1: <?xml version="1.0" ?>
 2: <!DOCTYPE a [
 3: <!ENTITY boilerplate "insert this here">
 4: ]>
 5: <a>
 6:  <b>in b</b>
 7:  <c>
 8:   text in c but &boilerplate;
 9:   <d/>
10:  </c>
11: </a>
```

When a DOM API parser constructs a DOM tree, it creates an Entity node under the DocumentType node. The resulting DOM tree looks like this, with the DocumentType, Entity, and Text nodes shaded in gray. The Entity node has a child, which is a text node containing the expansion text for the entity. So far, so good.



If you look closely at the diagram, you see that the part of the DOM tree for element c has been omitted. Here's the rest of it, starting at the Element node for c.



Xerces created an EntityReference node as a child of the Element node (and in the correct order among its siblings). That EntityReference node then has a child Text node that includes the text expanded from the entity. This is useful if you want to know that a particular node was an entity reference in the original document. However, it turns out to be inconvenient if you don't care whether some text originated as an entity, because your code has to check for the possibility of EntityReference nodes as it traverses the tree. If you don't care about the origin of the text, then you can set the feature http://apache.org/xml/features/dom/create-entity-ref-nodes to false, and Xerces won't insert the EntityReference nodes. Instead, it will put the Text node where the EntityReference node would have appeared, thus simplifying your application code.

Serialization

Most of the classes included with Xerces focus on taking XML documents, extracting information out of them, and passing that information on to your application via an API. Xerces also includes some classes that help you with the reverse process—taking data you already have and turning it into XML. This process is called *serialization* (not to be confused with Java serialization). The Xerces serialization API can take a SAX event stream or a DOM tree and produce an XML 1.0 or 1.1 document. One major improvement in XML 1.1 is that many more Unicode characters can appear in an XML 1.1 document; however, this makes it necessary to have a separate serializer for XML 1.1. There are also serializers that can take an XML document and serialize it using rules for HTML, XHTML, or even text files.

The org.apache.xml.serialize package includes five different serializers. All of them implement the interfaces org.apache.xml.serialize.Serializer and org.apache.xml.serialize.DOMSerializer as well as the ContentHandler, DocumentHandler, and DTDHandler classes from org.xml.sax and the DeclHandler and LexicalHandler classes from org.xml.sax.ext. The five serializers are as follows:

- **XMLSerializer** is used for XML 1.0 documents and, of course, obeys all the rules for XML 1.0.
- **XML11Serializer** outputs all the new Unicode characters allowed by XML 1.1. If the XML that you're outputting happens to be HTML, then you should use either the HTMLSerializer or the XHTMLSerializer.
- **HTMLSerializer** is used to output a document as HTML. It knows which HTML tags can get by without an end tag.
- **XHTMLSerializer** is used to output a document as XHTML, It serializes the document according to the XHTML rules.
- **TextSerializer** outputs the element names and the character data of elements. It doesn't output the DOCTYPE, DTD, or attributes.

Here are some of the differences in formatting when outputting HTML:

- The HTMLSerializer defaults to an ISO-8859-1 output encoding.
- An empty attribute value is output as an attribute name with no value at all (not even the equals sign). Also, attributes that are supposed to be URIs, as well as the content of the SCRIPT and STYLE tags, aren't escaped (embedded ", ', <, >, and & are left alone).
- The content of A and TD tags isn't line-broken.
- Most importantly, the HTMLSerializer knows that not all tags are closed in HTML. HTMLSerializer's list of the tags that do not require closing is as follows: AREA, BASE, BASEFONT, BR, COL, COLGROUP, DD, DT, FRAME, HEAD, HR, HTML, IMG, INPUT, ISINDEX, LI, LINK, META, OPTION, P, PARAM, TBODY, TD, TFOOT, TH, THEAD, and TR.

The XHTML serializer outputs HTML according to the rules for XHTML. These rules are:

- Element/attribute names are lowercase because case matters in XHTML.
- An attribute's value is always written if the value is the empty string.
- Empty elements must have a slash (/) in an empty tag (for example,
).
- The content of the SCRIPT and STYLE elements is serialized as CDATA.

Using the serializer classes is fairly straightforward. The serialization classes live in the package org.apache.xml.serialize. All the serializers are constructed with two arguments: The first argument is an OutputStream or Writer that is the destination for the output, and the second argument is an OutputFormat object that controls the details of how the serializer formats its input. OutputFormats are constructed with three arguments: a serialization method, which is a string constant taken from org.apache.xml.serialize.Method; a string containing the desired output character encoding; and a boolean that tells whether to indent the output. You can also construct an OutputFormat using a DOM Document object.

Before we get into the details of OutputFormat, let's look at how to use the serializers in a program. We'll look at a SAX-based version first:

```
 1: /*
 2:  *
 3:  * SAXSerializerMain.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11:
12: import org.apache.xerces.parsers.SAXParser;
13: import org.apache.xml.serialize.Method;
14: import org.apache.xml.serialize.OutputFormat;
15: import org.apache.xml.serialize.XMLSerializer;
16: import org.xml.sax.SAXException;
17: import org.xml.sax.SAXNotRecognizedException;
18: import org.xml.sax.SAXNotSupportedException;
19: import org.xml.sax.XMLReader;
20:
21: public class SAXSerializerMain {
22:
23:     public static void main(String[] args) {
24:         XMLReader r = new SAXParser();
25:         OutputFormat format =
26:             new OutputFormat(Method.XML,"UTF-8",true);
27:         format.setPreserveSpace(true);
28:         XMLSerializer serializer =
29:             new XMLSerializer(System.out, format);
30:         r.setContentHandler(serializer);
31:         r.setDTDHandler(serializer);
32:         try {
33:             r.setProperty(
34:                 "http://xml.org/sax/properties/declaration-handler",
35:                 serializer);
36:             r.setProperty(
37:                 "http://xml.org/sax/properties/lexical-handler",
38:                 serializer);
39:         } catch (SAXNotRecognizedException snre) {
40:             snre.printStackTrace();
41:         } catch (SAXNotSupportedException snse) {
```

```
42:             snse.printStackTrace();
43:         }
44:         try {
45:             r.parse(args[0]);
46:         } catch (IOException ioe) {
47:             ioe.printStackTrace();
48:         } catch (SAXException se) {
49:             se.printStackTrace();
50:         }
51:     }
52: }
```

Note that you set up the serializer (in this case, an XMLSerializer) and then plug it into the XMLReader as the callback handler for ContentHandler, DTDHandler, DeclHandler, and LexicalHandler.

A SAX version of the serializers might not seem interesting at first glance. Remember that SAX allows you to build a pipeline-style conglomeration of XML processing components that implement the org.xml.sax.XMLFilter interface. The SAX version of the serializers can be the last stage in one of these pipelines. You can also write applications that accept the various SAX handlers as callbacks and that then call the callbacks as a way of interfacing to other SAX components. Combining this approach with the serializer classes is way to use SAX to generate XML from non-XML data, such as comma-delimited or tab-delimited files.

The DOM version is a little more straightforward:

```
 1: /*
 2:  *
 3:  * DOMSerializerMain.java
 4:  *
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11:
12: import org.apache.xerces.parsers.DOMParser;
13: import org.apache.xml.serialize.Method;
14: import org.apache.xml.serialize.OutputFormat;
15: import org.apache.xml.serialize.XMLSerializer;
16: import org.w3c.dom.Document;
17: import org.xml.sax.SAXException;
18:
19: public class DOMSerializerMain {
20:
21:     public static void main(String[] args) {
22:         DOMParser p = new DOMParser();
23:
24:         try {
25:             p.parse(args[0]);
26:         } catch (SAXException se) {
27:             se.printStackTrace();
28:         } catch (IOException ioe) {
```

```
29:             ioe.printStackTrace();
30:         }
31:         Document d = p.getDocument();
32:         OutputFormat format =
33:             new OutputFormat(Method.XML,"UTF-8",true);
34:         format.setPreserveSpace(true);
35:         XMLSerializer serializer =
36:             new XMLSerializer(System.out, format);
37:         try {
38:             serializer.serialize(d);
39:         } catch (IOException ioe) {
40:             ioe.printStackTrace();
41:         }
42:     }
43: }
```

Here you construct the OutputFormat and serializer and then pass the DOM Document object to the serializer's serialize method.

OutputFormat options

Now that you've seen examples of how to use the serializers, let's look at OutputFormat in more detail. A number of properties control how a serializer behaves. We'll describe some of the more important ones below in JavaBean style, so the property encoding has a getEncoding method and a setEncoding method.

Property	Description
String encoding	The IANA name for the output character encoding.
String[] cDataElements	An array of element names whose contents should be output as CDATA.
int indent	The number of spaces to indent.
boolean indenting	True if the output should be indented.
String lineSeparator	A string used to separate lines.
int lineWidth	Lines longer than lineWidth characters are too long and are wrapped/indented as needed.
String[] nonEscapingElements	An array of element names whose contents should not be output escaped (no character references are used).
boolean omitComments	True if comments should not be output.
boolean omitDocumentType	True if the DOCTYPE declaration should not be output.
boolean omitXMLDeclaration	True if the XML declaration should not be output.
boolean preserveEmptyAttributes	If false, then in HTML mode, empty attribute are output as the attribute name only, with no equal sign or empty quotes.
boolean preserveSpace	True if the serializer should preserve space that already exists in the input.

The following set of methods deals with the DOCTYPE declaration:

Method	Description
String getDoctypePublic()	Gets the public ID of the current DOCTYPE.
String getDoctypeSystem()	Gets the system ID of the current DOCTYPE.
void setDocType(String publicId, String systemID)	Sets the public ID and system ID of the current DOCTYPE.

One least caveat on the use of the serializers: serializers aren't thread safe, so you have to be careful if you're going to use them in a multithreaded environment.

At the time of this writing, the W3C DOM Working Group is working on the DOM Level 3 Load/Save specification, which includes a mechanism for saving a DOM tree back to XML. This work has not been finalized and applies only to DOM trees. It's definitely worth learning the Xerces serializers API, because they also work with SAX. It's also worthwhile because the current (experimental) implementation of DOM Level 3 serialization in Xerces is based on the org.apache.xml.serialize classes.

XNI

The first version of Xerces used a SAX-like API internally. This API allowed you to build both a SAX API and a DOM API on top of a single parser engine. For Xerces version 2, this API was extended to make it easier to build parsers out of modular components. This extended and refactored API is known as the Xerces Native Interface (XNI). XNI is based on the idea of providing a streaming information set. The XML Infoset specification describes an abstract model of all the information items present in an XML document, including elements, attributes, characters, and so on. XNI takes the streaming/callback model used by SAX and expands the callback classes and methods so that as much of the information set as possible is available to applications that use XNI. As an example, XNI retains the encoding information for external entities and passes it along to the application. It also captures the information in the XML declaration and makes it available. XNI lets you build XML processors as a pipeline of components connected by the streaming information set.

SAX was designed primarily as a read-only API. XNI provides a read-write model. This allows the streaming information set to be augmented as it passes from component to component. One important application is in validating XML schema, which causes the XML infoset to be augmented with information—such as datatypes—obtained during validation. The read/write nature of XNI is accomplished by adding an additional argument to each callback method. This argument is an instance of org.apache.xerces.xni.Augmentations, which is a data structure like a hash table that allows data to be stored and retrieved via String keys.

Most developers never look at the XNI interfaces, because they can do everything they want via the SAX, DOM, or JAXP APIs. But for those looking to exploit the full power of Xerces, digging into the details of XNI is necessary. We'll provide a basic overview of the pieces of XNI and how they fit together, and show an example based on accessing the PSVI.

XNI Basics

An XNI-based parser contains two pipelines that do all the work: the document pipeline and the DTD pipeline. The pipelines consist of instances of XMLComponent that are chained together via interfaces that represent the streaming information set. Unlike SAX, which has a single pipeline, XNI divides the pipeline in two: one pipeline for the content of the document and a separate pipeline for dealing with the information DTD. The pipeline interfaces live in org.apache.xerces.xni:

Interface	Purpose
XMLDocumentHandler	The major interface in the document content pipeline. This should be familiar to anyone familiar with SAX.
XMLDocumentFragmentHandler	The document content pipeline can handle document fragments as well. To do this, you need to connect stages using XMLDocumentFragmentHandler instead of XMLDocumentHandler.
XMLDTDHandler	The major interface in the DTD pipeline. It handles everything except parsing the content model part of element declarations.
XMLDTDContentModelHandler	Provided for applications that want to parse the content model part of element declarations.
XMLString	A structure used to pass text around within XNI. You must copy the text out of an XMLString if you want use it after the XNI method has executed. XMLStrings should be treated as read-only.
XNIException	An Exception class for use with the XNI layer.
Augmentations	A data structure like a hash table, for storing augmentations to the stream information set. The set of augmentations is an argument to almost every XNI method in the content and DTD pipelines.
QName	An abstraction of XML QNames.
XMLAttributes	An abstraction for the set of attributes associated with an element.
XMLLocator	A data structure used to hold and report the location in the XML document where processing is occurring / has failed.
XMLResourceIdentifier	A data structure representing the public ID, system ID, and namespace of an XML resource (XML Schema, DTD, or general entity).
NamespaceContext	An abstraction representing the stack of namespace contexts (like variable scopes) within an XML document.

XMLString, XNIException, Augmentations, QName, XMLAttributes, XMLLocator, XMLResourceIdentifier, and NamespaceContext are all used by one of the four major interfaces (XMLDocumentHandler, XMLDocumentFragmentHandler, XMLDTDHandler, and XMLDTDContentModelHandler).

If you look at the XMLComponent interface, you'll see that it really just defines methods for setting configuration settings on a component. Not surprisingly, it uses a feature and property interface reminiscent of SAX. The biggest addition is a pair of methods that return an array of the features/properties supported by the component. What may surprise you is that the interface doesn't say anything about the callback interfaces for the pipeline. This is intentional, because not all components are in all pipelines—that's part of the rationale for breaking up the pipeline interfaces, so that components can implement the smallest set of functionality they require.

To implement a real component that can be a part of a pipeline, you need more interfaces. These interfaces are found in org.apache.xerces.xni.parser. The callback interfaces define what it means to be a recipient or sink for streaming information set events. Components that act as sinks sit at the end of the pipeline. That means you need interfaces for components at the start of the pipeline and for components in the middle. Components at the start of the pipeline are sources of streaming information set events, so they need to be connected to an event sink. The interface for these components has a pair of methods that let you get and set the sink to which the source is connected. There are three of these source interfaces, one for each of the major pipeline interfaces (XMLDocumentFragmentHandler is considered minor because document fragments appear so infrequently):

- **XMLDocumentSource** for XMLDocumentHandler
- **XMLDTDSource** for XMLDTDHandler
- **XMLDTDContentModelSource** for XMLDTDContentModelHandler

Now, defining interfaces for components in the middle is easy. These components must implement both the source and sink (handler) interfaces for the pipeline. That gives XMLDocumentFilter, which implements XMLDocumentSource and XMLDocumentHandler. XMLDTDFilter and XMLDTDContentModelFilter are defined in a similar way.

At this point it's a little clearer what an XNI pipeline is. Using the DocumentHandler as an example, a pipeline is an instance of XMLDocumentSource connected to some number of instances of XMLDocumentFilter that are chained together. The last XMLDocumentFilter is connected to an instance of XMLDocumentHandler, which provides the final output of the pipeline. The instance of XMLDocumentSource takes the XML document as input. The next question you should be thinking about is how the pipeline is constructed, connected, and started up.

XNI Pipeline Interfaces

XNI provides interfaces you can use to take care of these matters. You aren't by any means required to do this—you could do it with custom code, but you'll probably find that you end up duplicating the functionality provided by XNI. The interfaces for managing XMLComponents are also found in org.apache.xerces.xni.parser. Let's call a pipeline of XMLComponents a *configuration*. The interface for managing a configuration is called XMLParserConfiguration. This interface extends XMLComponentManager, which provides a simple API for querying whether a set of components

supports a particular feature or property. XMLParserConfiguration adds APIs that let you do several categories of tasks:

- **Configuration**—This API provides methods to tell configuration clients the set of supported features and properties. It also adds methods for changing the values of features and properties.
- **Sink management**—There are methods that allow configuration clients to register sinks for the three major pipeline interfaces in the configuration. Clients can also ask for the currently registered sink on a per-interface basis.
- **Helper services**—XMLParserConfiguration assumes that configuration-wide services and data are used by the XMLComponents in the configuration. Examples of these services include error reporting as defined by the XMLErrorHandler interface and entity resolution as defined by the XMLEntityResolver interface.
- **Parsing kickoff**—XMLParserConfiguration provides methods for starting the process of parsing XML from an XMLInputSource.

Let's look back at the diagram of Xerces. On top of the XMLParserConfiguration sits a Xerces parser class. This class is a sink for XMLDocumentHandler, XMLDTDHandler, and XMLDTDContentModelHandler. It registers itself as the sink for the various parts of the pipeline. The implementation of the various callback methods takes care of translating between the XNI callback and the parser API being implemented. For a SAX parser, the translation is pretty straightforward, consisting mostly of converting QNames and XMLStrings into Java Strings. A DOM parser is little more difficult because the callbacks need to build up the nodes of the DOM tree in addition to translating the XNI types.

Remember that we said the diagram was simplified. The Xerces SAXParser and DOMParser are actually implemented as a hierarchy of subclasses, with functionality layered between the various levels of the class hierarchy. The reason for doing this is to allow developers to produce their own variants of SAXParser and DOMParser with as little work as necessary.

There's only one part of the diagram we haven't discussed. At bottom right is a section labeled *support components*. We've already talked a little about helper components when we discussed XMLParserConfiguration. In that discussion, we were looking at components that were likely to be used by any parser configuration we could think have. Other support components are used only by a particular parser configuration. These are used internally by the parser configuration but are known by some number of the XMLComponents in the pipelines. Examples of these kinds of components include symbol tables and components dedicated to managing the use of namespaces throughout the configuration. These support components are provided to the pipeline components as properties, so they are assigned URI strings that mark them as being for internal use and then set using the configuration-wide property-setting mechanism.

Xerces2 XNI Components

XNI as we've discussed it is really a framework. The interfaces describe how the pieces of the framework interact. You can think of Xerces2 as a very useful reference implementation of the XNI framework. If you're going to build an application using XNI, you may find it useful to reuse some of the components from the Xerces2 reference implementation. These components have the advantage of being heavily tested and debugged, so you can concentrate on implementing just the functionality you need. Here are some of the most useful components from Xerces2.

Document Scanner

The document scanner knows how to take an XML document and fire the callbacks for elements (and attributes), characters, and anything else you might encounter in an XML document. This is the workhorse component for any XNI application that is going to work with an XML document. Applications that just work with the DTD or schema may end up not using this class. The document scanner is implemented by the class org,apache.xerces.impl.XMLDocumentScannerImpl and uses the URI http://apache.org/xml/properties/internal/document-scanner as its property ID. To use it, you also need the DTD scanner, entity manager, error reporter, and symbol table.

DTD Scanner

If you're processing DTDs, either directly or indirectly, you need the DTD scanner. It knows the syntax of DTDs and fires XMLDTDHandler and XMLDTDContentModelHandler events as it processes the DTD. The DTD scanner is implemented by the class org.apache.xerces.impl.XMLDTDScannerImpl and uses the URI http://apache.org/xml/properties/internal/dtd-scanner as its property ID. To use it, you also need the entity manager, error reporter, and symbol table.

DTD Validator

Scanning DTDs is different from validating with them. After the DTD pipeline has scanned the DTD and assembled the necessary definitions, the document content pipeline needs to use those definitions to validate the document. That's where the DTD validator comes in. It takes the definitions created by the DTD pipeline and uses them to validate the document. The validator is inserted into the pipeline as a filter, after the document scanner. The DTD validator is implemented by the class org.apache.xerces.impl.dtd.XMLDTDValidator and uses the URI http://apache.org/xml/properties/internal/validator/dtd as its property ID. To use it, you also need the entity manager, error reporter, and symbol table.

Namespace Binder

The process of mapping namespace prefixes to namespace URIs is called *namespace binding*. It needs to occur after DTD validation has occurred because the DTD may have provided default values for one or more namespace attributes in the document. These namespace bindings are needed for schema validation, so the namespace binder is inserted as a filter after the DTD validator and before the schema validator. The namespace binder is implemented by the class org.apache.xerces.impl.XMLNamespaceBinder and uses the URI http://apache.org/xml/properties/internal/namespace-binder as its property ID. To use it, you also need the error reporter and the symbol table.

Schema Validator

The schema validator validates the document against an XML schema. It's inserted into the pipeline as a filter after the namespace binder. As it processes the document, it may augment the streaming information set with default and normalized simple type values. It may also add items to the PSVI via the augmentations. The schema validator is implemented by the class org.apache.xerces.impl.xs.XMLSchemaValidator and uses the URI http://apache.org/xml/properties/internal/validator/schema as its property ID. To use it, you also need the error reporter and the symbol table.

Error Reporter

The parser configuration needs a single mechanism that all components can use to report errors. The Xerces2 error reporter provides a single point for all components to report errors. It also provides some support for localizing the error messages and calling the XNI XMLErrorHandler callback. Localization works as follows. Each component is given a domain designated by a URI. The component then implements the org.apache.xerces.util.MessageFormatter interface to generate and localize its own error messages. This component is used by almost all the other Xerces2 components, so you need to have one of them in your configuration if you use any of them. The error reporter is implemented by the class org.apache.xerces.impl.XMLErrorReporter and uses the URI http://apache.org/xml/properties/internal/error-reporter as its property ID.

Entity Manager

Xerces2 provides an entity manager that handles the starting and stopping of entities within an XML document. This gives its clients (primarily the document scanner and DTD scanner) the illusion that there is a single entity, not multiple entities. The entity manager is implemented by the class org.apache.xerces.impl.EntityManager and uses the URI http://apache.org/xml/properties/internal/entity-manager as its property id. To use it, you also need the error reporter and the symbol table.

Symbol Table

XML parsers look at a lot of text when processing documents. Much of that text (element and attribute names, namespaces prefixes, and so on) is repeated in XML documents. Xerces2 tries to take advantage of that fact by providing a custom symbol table for strings in order to improve performance. The symbol table always returns the same java.lang.String reference for a given string value. This means components can compare strings by comparing these references, not by comparing the string values. So, not only does the symbol table save space, it helps replace expensive calls to String#equals() with calls to ==. This component is used by all the rest of the Xerces2 components, so your configuration needs one of them if you use any Xerces2 components. The symbol table is implemented by the class org.apache.xerces.util.SymbolTable and uses the URI http://apache.org/xml/properties/internal/symbol-table as its property ID.

Using the Samples

The Xerces distribution includes a number of sample programs, some of which can be very useful when you're developing programs using Xerces—especially when you've embedded Xerces into your application. Suppose you're trying to debug an application and the problem appears to be inside Xerces itself. You may be seeing exceptions thrown or getting answers you think are incorrect. One debugging method that can save a lot of time is to capture the XML that's being input to Xerces, save it a file, and drag out one of the samples to help you see what's going on.

Before you use any of the samples, you need to get to a command-line prompt on your operating system. Make sure that xml-apis.jar, xercesImpl.jar, and xercesSamples.jar are all on your classpath.

If you're working with SAX, the first place to go is to the SAX Counter sample. This sample parses your document and prints some statistics based on what it finds. To invoke Counter, type

```
java sax.Counter <options> <filename>
```

There are command-line options to turn on and off namespace processing, validation, and schema validation, and to turn on full checking of the schema document. If you omit the options and filename, you'll get a help screen describing all the options. The key reason to start with sax.Count is that if Xerces is throwing an exception, it will probably throw that exception when you run sax.Count. From there, you can try to figure out if the problem is with the XML file, your application, or Xerces (in which case you should send mail to xerces-j-user@xml.apache.org with a bug report).

There's a pair of DocumentTracer samples, one for SAX and one for XNI. These samples are in classes named sax.DocumentTracer and xni.DocumentTracer, respectively. Their job is to print out all the SAX or XNI callbacks as they are fired for your document. Occasionally these samples can be useful to help you figure out which callbacks are being passed which data—especially when you're tired and confused after a long day of programming. They can also help you debug namespace-related problems, because all the prefixes get expanded. The output of xni.DocumentTracer is more detailed and complete than that of sax.DocumentTracer, due to the higher fidelity of the XNI callbacks, but most of the time you'll want to use sax.DocumentTracer so you can see exactly what SAX sees.

If you're using the DOM, you can use the DOM Counter sample, which lives in dom.Counter. It does the same thing as sax.Counter, but it uses the DOM and therefore will probably exercise some of the same DOM code your application does.

CyberNeko Tools for XNI

Andy Clark is one of the Xerces committers and was the driving force behind the design of XNI. He's written a suite of tools called NekoXNI to showcase some of the things you can do with XNI. Even if you aren't interested in using XNI, you might want to have a look, because some of the tools are pretty useful. In this section, we'll look at a few of these tools.

NekoHTML

NekoHTML uses XNI to allow an application to process an HTML document as if it were an XML document. There are both SAX and DOM parsers in the org.cyberneko.html.parsers package. You use org.cyberneko.html.parsers.SAXParser just like the regular Xerces SAXParser; you can plug in your own ContentHandlers and so on using the regular SAX API. The org.cyberneko.html.parsers.DOMParser works like the Xerces DOMParser with one notable twist. Instead of using the Xerces XML DOM, it uses the Xerces HTML DOM, which means you get a DOM implementation that is aware of some of the rules of HTML. To use NekoHTML, you need to have nekohtml.jar in your classpath, in addition to the regular jars you need for Xerces. But if you need to process HTML, it's worth it.

ManekiNeko

Another interesting and useful component of NekoXNI is a validator for Relax-NG called ManekiNeko. This validator is based on James Clark's Jing validator for Relax-NG, and it works by creating a wrapper

that converts XNI events into the SAX events that Jing already understands. This wrapped version of Jing is then inserted into the appropriate spot in the XNI pipeline within an XMLParserConfiguration called JingConfiguration. For ease of use, Andy has again provided convenience classes that work just like the Xerces SAX and DOM parser classes. For a Relax-NG aware SAX parser, use org.cyberneko .relaxng.parsers.SAXParser; for a DOM parser, use org.cyberneko.relaxng.parsers.DOMParser. You must set the SAX validation and namespace features to true. You must also set a property that tells the Relax-NG validator where to find the Relax-NG schema to be used for validation, because Relax-NG doesn't specify a way of associating a schema with a document. This property is called http://cyberneko.org /xml/properties/relaxng/schema-location, and its value should be the URI for the schema file.

NekoPull

The last CyberNeko tool is NekoPull, the CyberNeko pull parser. The commonly used APIs for XML, SAX, and DOM are push APIs. Once your program asks the parser to parse a document, your application doesn't regain control until the parse completes. SAX calls your program code via its event callbacks, but that's about as good as it gets. With the DOM, you have to wait until the entire tree has been built before you can do anything.

The difficulty with SAX is that for any non-trivial XML grammar, you end up maintaining a bunch of stacks and a state machine that remembers where you are in the grammar at any point in the parse. It also makes it very hard to modularize your application. If you have an XML grammar where the elements are turned into objects of various classes, you have to do a lot of work to keep the event-handling code for each class associated with each class. You end up trying to create ContentHandlers that handle only the section of the grammar for a particular class, and then you have to build infrastructure to multiplex between these ContentHandlers. It can be done, but the process is tedious and error prone.

With the DOM, you can create a constructor that knows how to construct an instance of your class from an org.w3c.dom.Element node, and then you can pass the DOM tree around to instances of the various classes. You can handle contained objects by passing the right element in the DOM tree to the constructors for those contained object types. The disadvantage of the DOM is that you have to wait until the whole document is processed, even if you only need part of it. And, of course, there's the usual problem of how much memory DOM trees take up.

Pull-parsing APIs can give you the best of both worlds. In a pull-parsing API, the application asks the parser to parse the next unit in the XML document, regardless of whether that unit is an element, character data, a processing instruction, and so on. This means you can process the document in a streaming fashion, which is a benefit of SAX. You can also pass the parser instance around to your various object constructors. Because the parser instance remembers where it is in the document, the constructor can call the parser to ask for the next bits of XML, which should represent the data it needs to construct an object. Contained objects are handled just like the DOM case; you pass the parser instance (which again remembers its place) to the constructors for the contained objects. This is a much better API.

Let's walk through a pull implementation of the Book object building program:

```
1: /*
2:  *
3:  * NekoPullMain.java
4:  *
```

```
 5:  * Example from "Professional XML Development with Apache Tools"
 6:  *
 7:  */
 8: package com.sauria.apachexml.ch1;
 9:
10: import java.io.IOException;
11: import java.util.Stack;
12:
13: import org.apache.xerces.xni.XMLAttributes;
14: import org.apache.xerces.xni.XNIException;
15: import org.apache.xerces.xni.parser.XMLInputSource;
16: import org.cyberneko.pull.XMLEvent;
17: import org.cyberneko.pull.XMLPullParser;
18: import org.cyberneko.pull.event.CharactersEvent;
19: import org.cyberneko.pull.event.ElementEvent;
20: import org.cyberneko.pull.parsers.Xerces2;
21:
22: public class NekoPullMain {
23:
24:     public static void main(String[] args) {
25:         try {
26:             XMLInputSource is =
27:                 new XMLInputSource(null, args[0], null);
28:             XMLPullParser pullParser = new Xerces2();
29:             pullParser.setInputSource(is);
30:             Book book = makeBook(pullParser);
```

You start by instantiating an instance of the pull parser and setting it up with the input source for the document. Then you pass the parser, which is at the correct position to start reading a book, to a constructor function for the Book class.

```
31:             System.out.println(book.toString());
32:         } catch (IOException ioe) {
33:             ioe.printStackTrace();
34:         }
35:     }
36:
37:     private static Book makeBook(XMLPullParser pullParser)
38:         throws IOException {
39:         Book book = null;
40:         Stack textStack = new Stack();
```

When you ask the parser for the next bit of XML, you get back an event. That event is an object (a struct, really) that contains all the information about the piece of XML the parser saw. NekoPull includes event types for the document, elements, character data, CDATA, comments, text declaration, DOCTYPE declaration, processing instructions, entities, and namespace prefix mappings. The event types are determined by integer values in the type field of XMLEvent. Some of the events are bounded; that is, they correspond to a start/end pairing and are reported twice. The bounded events are DocumentEvent, ElementEvent, GeneralEntityEvent, CDATAEvent, and PrefixMappingEvent; a boolean field called start distinguishes start events from end events.

You loop and call pullParser's nextEvent method to get events until there aren't any more (or until you break out of the loop):

```
41:         XMLEvent evt;
42:         while ((evt = pullParser.nextEvent()) != null) {
43:             switch (evt.type) {
44:                 case XMLEvent.ELEMENT :
45:                     ElementEvent eltEvt = (ElementEvent) evt;
46:                     if (eltEvt.start) {
47:                         textStack.push(new StringBuffer());
48:                         String localPart = eltEvt.element.localpart;
49:                         if (localPart.equals("book")) {
50:                             XMLAttributes attrs = eltEvt.attributes;
51:                             String version =
52:                                 attrs.getValue(null, "version");
53:                             if (version.equals("1.0")) {
54:                                 book = new Book();
55:                                 continue;
56:                             }
57:                             throw new XNIException("bad version");
58:                         }
```

If you see a starting ElementEvent for the book element, you check the version attribute to make sure it's 1.0 and then instantiate a new Book object. For all starting ElementEvents, you push a new StringBuffer onto a textStack, just like for SAX. You do this to make sure you catch text in mixed content, which will be interrupted by markup. For example, in

```
<blockquote>
    I really <em>didn't</em> like what he had to say
</blockquote>
```

the text "I really" and "like what he had to say" belongs inside the blockquote element, whereas the text "didn't" belongs inside the em element. Keeping this text together is what the textStack is all about.

The real work of building the object is done when you hit the end tag, where you get an ending ElementEvent. Here you grab the text you've been collecting for this element and, based on the tag you're closing, call the appropriate Book setter method. You should be pretty familiar with this sort of code by now:

```
59:                     } else if (!eltEvt.empty) {
60:                         String localPart = eltEvt.element.localpart;
61:                         StringBuffer tos =
62:                             (StringBuffer) textStack.pop();
63:                         String text = tos.toString();
64:                         if (localPart.equals("title")) {
65:                             book.setTitle(text);
66:                         } else if (localPart.equals("author")) {
67:                             book.setAuthor(text);
68:                         } else if (localPart.equals("isbn")) {
69:                             book.setIsbn(text);
70:                         } else if (localPart.equals("month")) {
71:                             book.setMonth(text);
72:                         } else if (localPart.equals("year")) {
73:                             int year = 0;
74:                             year = Integer.parseInt(text);
75:                             book.setYear(year);
```

```
76:                         } else if (localPart.equals("publisher")) {
77:                             book.setPublisher(text);
78:                         } else if (localPart.equals("address")) {
79:                             book.setAddress(text);
80:                         }
```

When you see a CharactersEvent, you're appending the characters in the event to the text you're keeping for this element:

```
81:                         }
82:                         break;
83:                     case XMLEvent.CHARACTERS :
84:                         CharactersEvent chEvt = (CharactersEvent) evt;
85:                         StringBuffer tos =
86:                             (StringBuffer) textStack.peek();
87:                         tos.append(chEvt.text.toString());
88:                         break;
89:                 }
90:             }
91:             return book;
92:         }
93: }
```

As you can see, the style inside the constructor method is somewhat reminiscent of a SAX content handler. The difference is that when you get to contained objects, the code is dramatically simpler. You just have a bunch of methods that look like makeBook, except that as part of the processing of certain end ElementEvents, there's a call to the constructor function of another class, with the only argument being the pull parser.

As we're writing this, the first public review of JSR-173, the Streaming API for XML, has just begun. Perhaps by the time you're reading this, NekoXNI's pull parser will be implementing what's in that JSR.

At the moment, the NekoXNI tools are separate from Xerces, but there have been some discussions about incorporating all or some of the tools into the main Xerces distribution.

Practical Usage

We've covered a lot of ways you can use Xerces to get information out of XML documents and into your application. Here are two more practical usage tips.

Xerces isn't thread safe. You can't have two threads that execute a single Xerces instance at the same time. If you're in a multithreaded situation, you should create one instance of Xerces for each thread. If for some reason you don't want to do that, make sure the access to the parser instance is synchronized, or you'll run into some nasty problems. A common solution pattern for concurrent systems is to provide the thread with a pool of parser instances that have already been created.

That leads us into the second tip. If your application is processing many XML documents, you should try to reuse parser instances. Both the Xerces SAXParser and DOMParser provide a method called reset that you can use to reset the parser's internal data structures so the instance can be used to parse another

document. This saves the overhead of creating all the internal data structures for each document. When you combine this with grammar caching, you can get some nice improvements in performance relative to creating a parser instance over and over again.

Common Problems

This section addresses some common problems that people encounter when they use Xerces. Most of these issues aren't Xerces specific, but they happen so frequently that we wanted to address them.

- **Classpath problems—**It's a simple mistake but a surprisingly common one. Both xml-apis.jar and xercesImpl.jar must be on your classpath in order to use Xerces. Leaving one of them out will cause pain and suffering. If you want to use the samples, you need to include xercesSamples.jar on your classpath.

  The other thing to beware of is strange interactions between your classpath and either the JDK 1.3 Extension Mechanism or the JDK 1.4 Endorsed Standards Override Mechanism. If it looks like you aren't getting Xerces or the Xerces version that you think you're using, look for old versions of Xerces in these places. You can determine the version of Xerces by executing the following at your command line:

  ```
  java org.apache.xerces.impl.Version
  ```

  This command prints out the version of Xerces you're using. You can also call the static method org.apache.xerces.impl.Version#getVersion from inside a program to get the version string.

- **Errors not reported or always reported to the console—**If you don't provide an ErrorHandler, one of two behaviors will occur. In every version of Xerces prior to 2.3.0, if no ErrorHandler is registered, no error messages are displayed. You must register your own ErrorHandler if you want error messages to be reported. This problem confused a lot of people, so in version 2.3.0 the behavior was changed so that error messages are echoed to the console when no ErrorHandler is registered. In these versions of Xerces, you need to register your own ErrorHandler to turn off the messages to the console.

- **Multiple calls to characters—**In SAX applications, it's common to forget that the characters callback may be called more than once for the character data inside an element. Unless you buffer up the text by, say, appending it to a StringBuffer, it may look like your application is randomly throwing away pieces of character data.

- **When is ignorableWhitespace called?—**It's not enough that the definition of ignorable whitespace is confusing to people. The ignorableWhitespace callback is called for ignorableWhitespace only when a DTD is associated with the document. If there's no DTD, ignorableWhitespace isn't called. This is true even if there is an XML schema but no DTD.

- **Forgot validation switches—**Another common problem is forgetting to turn on the validation features. This is true both for DTD validation and for schema validation. A single feature must be turned on for DTD validation; but for schema validation you must have namespace support turned on in addition to the feature for schema validation. That's three properties. Make sure you have them all on.

- **Multiple documents in one file—**People like to try to put multiple XML documents into a single file. This isn't legal XML, and Xerces won't swallow it. You'll definitely see errors for that.

- **Mismatched encoding declaration—**The character encoding used in a file and the encoding name specified in the encoding declaration must match. The encoding declaration is the encoding="name" that appears after <? xml version="1.0" encoding="name"?> in an XML document. If the encoding of the file and the declared encoding don't match, you may see errors about invalid characters.
- **Forgetting to use namespace-aware methods—**If you're working with namespaces, be sure to use the namespace-aware versions of the methods. With SAX this is fairly easy because most people are using the SAX 2.0 ContentHandler, which has only the namespace-aware callback methods. If you're using DocumentHandler and trying to do namespaces, you're in the wrong place. You need to use ContentHandler. In DOM-based parsers, this is a little harder because there are namespace-aware versions of methods that have the letters *NS* appended to their names. So, Element#getAttributeNS is the namespace-aware version of the Element#getAttribute method.
- **Out of memory using the DOM—**Depending on the document you're working with, you may see out-of-memory errors if you're using the DOM. This happens because the DOM tends to be very memory intensive. There are several possible solutions. You can increase the size of the Java heap. You can use the DOM in deferred mode—if you're using the JAXP interfaces, then you aren't using the DOM in deferred mode. Finally, you can try to prune some of the nodes in the DOM tree by setting the feature http://apache.org/xml/features/dom/include-ignorable-whitespace to false.
- **Using appendChild instead of importNode across DOM trees—**The Xerces DOM implementation tries to enforce some integrity constraints on the contents of the DOM. One common thing developers want to do is create a new DOM tree and then copy some nodes from another DOM tree into it. Usually they try to do this using Node#appendChild, and then they start seeing exceptions like *DOMException: DOM005 Wrong document*, which is confusing. To copy nodes between DOM trees you need to use the Document#importNode method, and then you can call the method you want to put the node into its new home.

Applications

We've covered a lot of ground in this chapter, and yet we've hardly begun. XML parsing has so many applications that it's hard to show all the ways you might use it in your application. Here are a couple of ideas.

One place you end up directly interacting with the XML parser is in the kind of example we've been using through out this chapter: turning XML documents into domain-specific objects within your application. Although there are some proposals for tools that can do it for you, this is a task where you'll still see developers having direct interaction with the parser, at least for a little while longer.

Another application people use the parser for directly is filtering XML. When you have a very large XML document and you need only part of it, using SAX to cut out the stuff you don't want to deal with is a very viable solution.

XML parsers have a place as a document and schema development tool. They provide the means for you to create XML documents and grammars in many forms (DTDs, XML Schema, and Relax-NG) and verify that the grammars you've written do what you want and that your documents conform to those grammars.

The reality is that most developers are doing less with XML parsers directly. That's because lots of clever tool and application developers have leveraged the fundamental capability of XML parsing and used it to build tools that operate at a higher level. Although we hope you're excited about Xerces and all the cool things you can do with it, we hope you're even more excited by some of the tools that have already been built on top of it. Those tools are what the rest of this book is about.