Java and the Apache XML Project

CHAPTER

17

IN THIS CHAPTER

This chapter is a tour through the emerging world of Apache, specifically the Xerces Java XML parser. The chapter introduces the Xerces download component, its integrated parser, documentation, and samples. Then it focuses on the critical packages and shows how to construct working applications, using both the Document Object Model (DOM) and Simple API for XML (SAX) models. You may use these samples as frameworks for further development. Along the way, the chapter introduces every important class and interface, so that by the end of the chapter, you will be adept in the construction of XML applications.

We assume that you have at least an intermediate comfort level with Java, that you understand the concepts of paths and classpaths, that you have utilized Java packages, classes, and interfaces, and that you have experience writing, compiling, and running applications. If you meet these requirements, and are comfortable with previous chapters, then hop on board.

17.1 Apache Background

Apache is a story that warms the hearts of Internet traditionalists. Sometimes confused with IBM (thanks to the influential corporation's mass acceptance of its software), Apache is actually a pure not-for-profit, open-source endeavor. Formed in 1995 by a half dozen Webmasters to consciously develop "a cog for the Internet," Apache emerged as the most widely accepted HTTP server—possibly the most successful piece of shareware ever released in terms of market share. Their triumph has ensured that at least one standard, the HTTP protocol, remains simple and approachable, safeguarded from proprietary interests.

The Apache Software Foundation (at `http://www.apache.org`) now boasts 60+ members whose open-source vision has embraced emerging standards to provide practical, zero-cost implementations for technologies ranging from Perl to PHP to XML. This chapter, of course, focuses on the XML technologies (and trust us, all the others are just as fun as this one!).

The Apache project features the Xerces XML parsers (available in Java and C++) but also hosts a broad realm of XML technologies. Developers can access additional tools that assist Web publishing, SOAP development, and formatting. The following is a brief list of XML subprojects, taken from the Apache XML Web site (`http://xml.apache.org`).

Xerces: XML parsers in Java, C++ (with Perl and COM bindings)

Xang: Rapid development of dynamic server pages, in JavaScript

Xalan: XSLT stylesheet processors, in Java and C++

SOAP: Simple Object Access Protocol

FOP: XSL formatting objects, in Java

Crimson: Java XML parser derived from the Sun Project X Parser

Cocoon: XML-based Web publishing, in Java

Batik: Java-based toolkit for Scalable Vector Graphics (SVG)

AxKit: XML-based Web publishing, in mod_perl

Many of these projects support recent additions to the XML set of standards. The Apache-Xerces parser, for instance, has provided XML Schema functionality since early in its inception; Xerces version 1.1 (released in May 2000) supported the working draft specification and has been updated regularly. Xerces has been fully XML Schema-compliant since Xerces version 1.1.3 (save for minor limitations, which are well documented at `http://xml.apache.org/xerces-j/releases.html`).

Note that we have referred to a singular parser, but a visit to `http://xml.apache.org` reveals links to *two* different parsers: Xerces Java 1 and Xerces Java 2. Xerces Java 2, or simply Xerces2, is much more recent, a complete rewrite of the existing version 1 codebase. Xerces2 has a custom Xerces Native Interface (XNI), and its source code is said to be "much cleaner, more modular, and easier to maintain" than Xerces1. Xerces2 also implements the latest W3 XML Schema standards. Table 17.1 contains a matrix of implemented standards for both parsers.

Because the features are nearly parallel, your choice between the two parsers rests primarily on your desire for customization. Will you need access to code for adjustment or extension (possibly to implement late W3 features yourself)? Xerces2 might be your best choice; but extend your test schedule appropriately because Xerces2 might be a bit less stable and reliable (and check back to the `http://xml.apache.org` Web site often for updates). Xerces2 now receives the majority of attention from Apache developers. For purposes of this chapter,

TABLE 17.1 A Comparison of Xerces Parsers

Supported Standards	*Xerces Java 1*	*Xerces Java 2*
Current Version (8/2002)	1.4.4	2.0.2
XML Recommendation	1.0 Recommendation	1.0, Second Edition
XML Namespaces	Recommendation	Recommendation
Document Object Model	DOM Level 1 and 2	DOM Level 2 Core, Events, Traversal, and Range Recommendations
		DOM Level 3 Core, Abstract Schemas, Load, and Save Working Drafts
Simple API for XML (SAX)	SAX Level 1 and 2	SAX Level 2 Core, Extension
Java APIs for XML Processing (JAXP)	JAXP 1.1	JAXP 1.1
XML Schema	1.0	1.0, Structures and Datatypes Recommendation, DOM Level 3 revalidation

we use Xerces2. When we refer to Xerces or "the parser," understand that we explicitly mean Xerces2.

17.2 Java Xerces on Your Computer

We will now take you through the process of downloading and exploring the Xerces2 package. By the end of this section, you should know the basic layout of the package and have successfully run several Xerces example applications.

17.2.1 Downloading Java Xerces 2 Parser

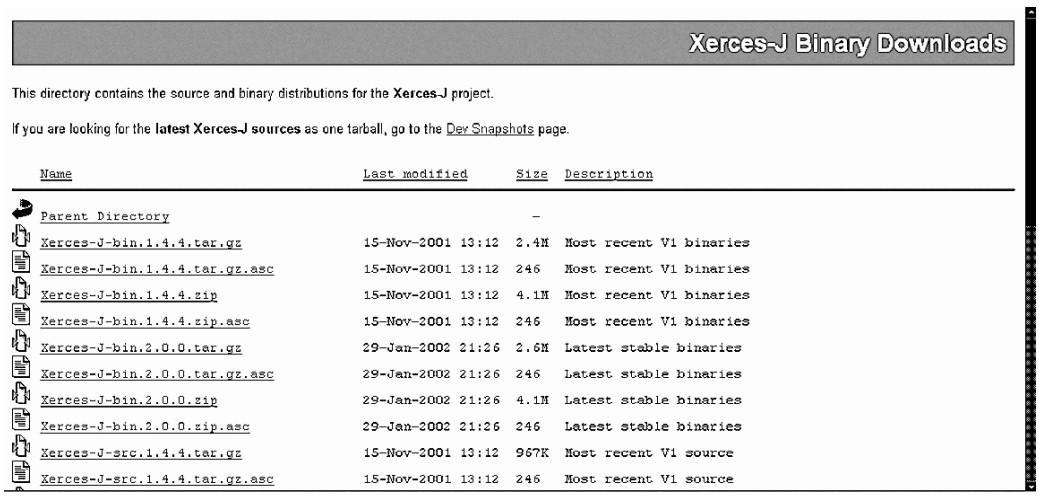
The download location for the Java Xerces2 parser is at

`http://xml.apache.org/dist/xerces-j/`

or you can click Xerces Java 2 on `http://xml.apache.org`, then click Download. A menu of options appears as shown in Figure 17.1.

Figuring out which file to download is easy. The working parser, packaged with supporting API documentation, is found in `bin` packages (`bin` being short for binaries or executables). Next, match the version you want to employ. There should be one 1.x.x version and one 2.x.x version. Only the latest of each parser version is available through this main page; older versions can be found at the bottom of the page (or with a little digging, if this page has changed).

The `bin` package is only one of three options. `src` packages contain the full source code for each version. The `tools` package includes the `ant` source-code compiler, the `junit` test package, and `xerces` and `xalan` JAR files. If you intend to customize code (by downloading the `src` package), you might find these tools quite helpful. Again, the version numbering applies, just as for the `bin` packages.



Xerces-J Binary Downloads

This directory contains the source and binary distributions for the **Xerces-J** project.

If you are looking for the **latest Xerces-J sources** as one tarball, go to the Dev Snapshots page.

Name	Last modified	Size	Description
Parent Directory		-	
Xerces-J-bin.1.4.4.tar.gz	15-Nov-2001 13:12	2.4M	Most recent V1 binaries
Xerces-J-bin.1.4.4.tar.gz.asc	15-Nov-2001 13:12	246	Most recent V1 binaries
Xerces-J-bin.1.4.4.zip	15-Nov-2001 13:12	4.1M	Most recent V1 binaries
Xerces-J-bin.1.4.4.zip.asc	15-Nov-2001 13:12	246	Most recent V1 binaries
Xerces-J-bin.2.0.0.tar.gz	29-Jan-2002 21:26	2.6M	Latest stable binaries
Xerces-J-bin.2.0.0.tar.gz.asc	29-Jan-2002 21:26	246	Latest stable binaries
Xerces-J-bin.2.0.0.zip	29-Jan-2002 21:26	4.1M	Latest stable binaries
Xerces-J-bin.2.0.0.zip.asc	29-Jan-2002 21:26	246	Latest stable binaries
Xerces-J-src.1.4.4.tar.gz	15-Nov-2001 13:12	967K	Most recent V1 source
Xerces-J-src.1.4.4.tar.gz.asc	15-Nov-2001 13:12	246	Most recent V1 source

FIGURE 17.1 *The Xerces download page.*

TIP

There are many download options once you have selected your type and version of Xerces. The two of interest end with '`.zip`' and '`.tar.gz`'. Windows users recognize '`.zip`', just as Unix users know well the `tar`ed, "GunZipped" format. The latter format is actually a *double-compressed* file and almost always smaller (and quicker to download) than its `.zip` counterpart. What many Windows users do not realize is that *they too* can open `.tar.gz` files, using the familiar WinZip utility, so there is no reason not to download the smallest package for your version.

17.2.2 Exploring the Xerces Package

After you have downloaded and unzipped your Xerces package, you will see a top-level layout of files and folders (Xerces version 2.0) as in Figure 17.2.

Table 17.2 contains a brief summary of contents. Explore these folders and files for a few minutes.

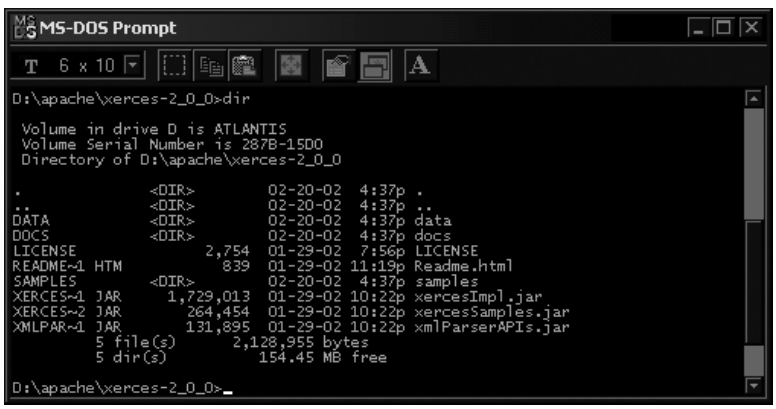


FIGURE 17.2 *The Xerces2 project folder.*

TABLE 17.2 Xerces2 Download Package, Top-level Files and Folders

Xerces2 Top-level Files and Directories	*Contents*
`docs/`	Contains the API documentation, including Javadocs
`samples/`	A plethora of source code samples demonstrating DOM and SAX as well as XML UI and IO and the custom XNI Xerces features

continues

TABLE 17.2 *(continued)*

Xerces2 Top-level Files and Directories	*Contents*
`data/`	Contains XML, Document Type Definition (DTD), and XML schema document files for the sample apps
`License`	The Apache-Xerces license
`Readme.html`	Links to the official API documentation
`xercesImpl.jar`	All the parser class files
`xmlParserAPIs.jar`	Implementation for all the standard APIs (DOM Level 2, SAX 2.0 R2 PR1, and the javax.xml.parsers of JAXP 1.1)
`xercesSamples.jar`	Compiled samples (class files) from the samples/folder

NOTE

For experienced users of Xerces who might have expected to see a `xerces.jar` file (from the official documentation): "Xerces formerly came with a file called `xerces.jar`. This file contained all of the parser's functionality. Two files are now included: `xercesImpl.jar`, our implementation of various APIs, and `xmlParserAPIs.jar`, the APIs themselves. This was done so that, if your XSLT [XSL Transformations] processor ships with APIs at the same level as those supported by Xerces-J, you can avoid putting `xmlParserAPIs.jar` on your classpath."

17.2.3 Running the Samples

Installation and execution of Java applications has always been a breeze. Certainly you know by now that you must have Java installed on your system, which simply means that a virtual machine is accessible on your path. Visit `http://java.sun.com` to locate and download a Java virtual machine/Java Runtime Environment (JRE) or a full Java Development Kit (JDK); most packages have an extractor or installation program to take care of all details. After you have Java installed, running Xerces samples is a piece of cake.

NOTE

Xerces Java samples run in JDK 1.x and Java 2 (Java 1.2.x, 1.3, and later) except those with a user interface (the samples in the `samples\ui` folder), which require Java 2 because they were written in Swing 1.1.

The Xerces parser includes over a dozen complete code samples. This is code you can reuse immediately to see quick results, and these samples reveal techniques that go even beyond this chapter. The following list contains some of the more useful sample classes, with brief descriptions summarized from the official documentation:

- `dom.Counter`: Shows the duration and count of elements, attributes, ignorable whitespaces, and characters appearing in the document. Three durations are shown: the parse time, the first traversal of the document, and the second traversal of the tree.
- `dom.DOMAddLines`: Illustrates how to add lines to the DOM `Node` and override methods from `DocumentHandler`, how to turn off ignorable whitespaces by overriding `ignorableWhitespace`, how to use the SAX `Locator` to return row position (line number of DOM element), and how to attach user-defined objects to `Nodes` by using the `setUserData` method.
- `dom.GetElementsByTagName`: Illustrates how to use the `Document#getElementsByTagName()` method to quickly and easily locate elements by name. This sample is also a DOM filter.
- `dom.Writer`: Illustrates how to traverse a DOM tree to print a document that is parsed.
- `dom.ASBuilder`: Illustrates how to preparse XML schema documents and build schemas, and how to validate XML instances against those schemas, using DOM Level 3 classes.
- `sax.Counter`: Illustrates how to register a SAX2 `ContentHandler` and receive the callbacks to print information about the document. The output of this program shows the time and count of elements, attributes, ignorable whitespaces, and characters appearing in the document.
- `sax.DocumentTracer`: Provides a complete trace of SAX2 events for files parsed. This is useful for making sure a SAX parser implementation faithfully communicates all information in the document to the SAX handlers.
- `sax.Writer`: Illustrates how to register a SAX2 `ContentHandler` and receive the callbacks to print a document that is parsed.
- `socket.DelayedInput`: Delays the input to the SAX parser to simulate reading data from a socket where data is not always immediately available.
- `socket.KeepSocketOpen`: Provides a solution to the problems of sending multiple XML documents over a single socket connection or sending other types of data after the XML document without closing the socket connection.
- `ui.TreeView`: Allows exploration of XML nodes via a tree panel in a Java Swing application. This sample also reveals document messages sent as the tree is traversed.

The names of the samples in the preceding list are used to invoke the corresponding code in the `xercesSamples.jar` package. If all three Xerces JAR files are placed in the classpath, running the `sax.DocumentTracer` sample, for example, is as simple as typing the following anywhere on the command line:

```
java sax.DocumentTracer my.xml
```

in which `my.xml` is any XML file (in the current path). If you do not want to add all three JAR files to your classpath, use the `-cp` option, with each JAR file listed afterward (separated by semicolons), as shown in Figure 17.3.

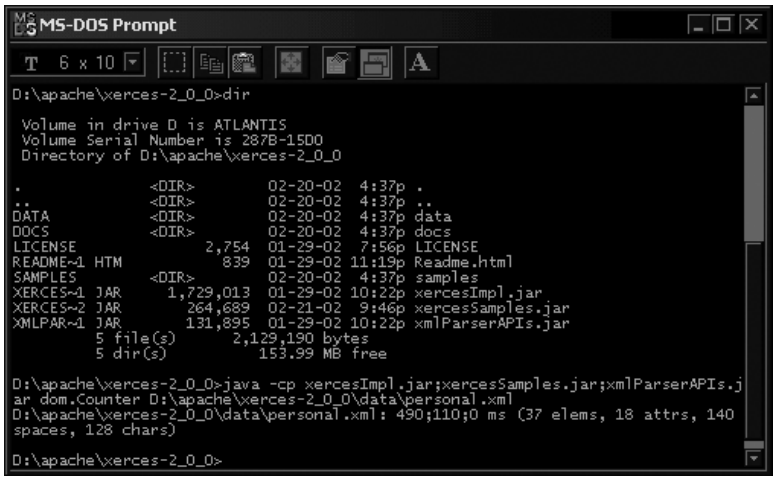


FIGURE 17.3 *Execution and output of first Xerces samples.*

NOTE

If you have a Java virtual machine version 1.3 or later, find the `ext` folder. For Windows users, this is typically:

```
C:\Program Files\JavaSoft\JRE\1.4\lib\ext\
```

Here you can place JAR files for global access, just as if you had placed each JAR file in your classpath. This makes using the `-cp` option unnecessary.

We conclude this section with one more sample, just for fun (and also because it might come in useful for message tracing in the future). This is the UI example from the sample set, called

"`TreeView`," which we are going to feed our `address.xml` file (make sure the `address.xsd` file is also in the current path). At the command line, enter the following line (presuming, this time, that you *do not* have the Xerces JAR files in the classpath):

```
java -cp xercesImpl.jar;xmlParserAPIs.jar;xercesSamples.jar
    ui.TreeView address.xml
```

Within a few seconds, a brand-new window appears, similar to the window shown in Figure 17.4.

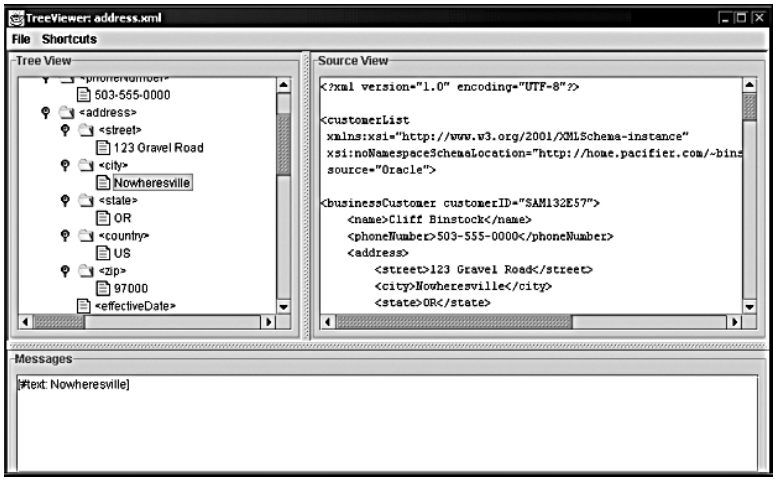


FIGURE 17.4 *Output of Xerces* `TreeView` *sample.*

In this window, you can expand the entire tree, refresh the view, and load new documents. (You *must* load an XML file from the command line initially, though; otherwise, the window will not appear.)

17.3 "Hello Apache"

In this section, you gain a little momentum, and a success or two, before digging into the nitty-gritty of Xerces. You use basic DOM methods to create a simple XML document, and then you reverse the process and parse an existing XML document.

17.3.1 Your First Parser

Listing 17.1 is a complete program that creates a few XML elements and then displays the serialized XML. The name of the file is `HelloApache.java`.

LISTING 17.1 HelloApache Example

```
import java.io.StringWriter;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.apache.xerces.dom.DocumentImpl;
import org.apache.xerces.dom.DOMImplementationImpl;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.Serializer;
import org.apache.xml.serialize.XMLSerializer;

public class HelloApache
{
  public static void main (String[] args)
  {
    try
    {
      Document doc = new DocumentImpl();

      // Create Root Element
      Element root = doc.createElement("BOOK");

      // Create 2nd level Element and attach to the Root Element
      Element item = doc.createElement("AUTHOR");
      item.appendChild(doc.createTextNode("Bachelard.Gaston"));
      root.appendChild(item);

      // Create one more Element
      item = doc.createElement("TITLE");
      item.appendChild(doc.createTextNode
                       ("The Poetics of Reverie"));
      root.appendChild(item);

      item = doc.createElement("TRANSLATOR");
      item.appendChild(doc.createTextNode("Daniel Russell"));
      root.appendChild(item);

      // Add the Root Element to Document
      doc.appendChild(root);

      //Serialize DOM
      OutputFormat format    = new OutputFormat (doc);
      // as a String
      StringWriter stringOut = new StringWriter ();
      XMLSerializer serial   = new XMLSerializer (stringOut,
                                                  format);
```

```
      serial.serialize(doc);

      // Display the XML
      System.out.println(stringOut.toString());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```


Naturally, you could shorten the number of `import` statements, but type these out at first to get comfortable with the locations of important classes.

To compile and execute this example, type these statements at the command line:

```
javac -classpath xercesImple.jar;xmlParserAPIs.jar;.
    HelloApache.java
java -cp xercesImple.jar;xmlParserAPIs.jar;.HelloApache
```

Hopefully, your results match the screen shown in Figure 17.5.

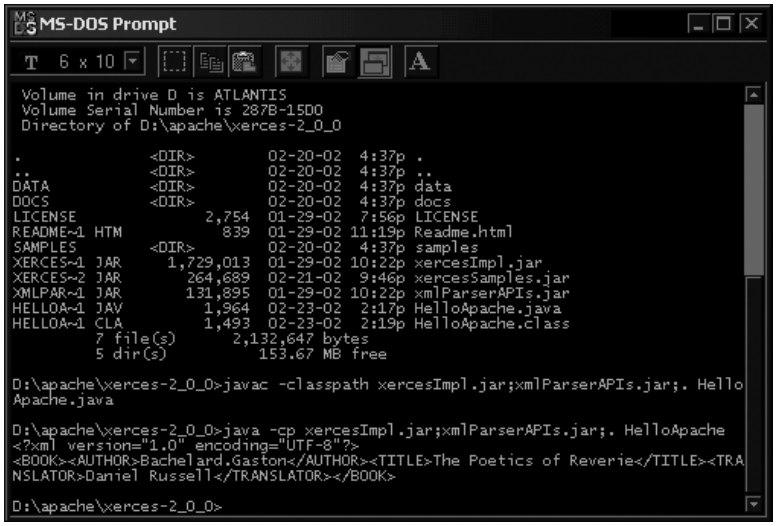


FIGURE 17.5 *Output of* `HelloApache` *example.*

WARNING

Be sure to check that no other XML parsers or packages are located on your classpath (which is why explicit definition of the classpath is used in our example) or in your JRE/JDK's `lib/ext` folder (which is another virtual classpath). Because Xerces2 uses `org.w3.*`-defined interfaces and permits use of other XML packages (such as JAXP), conflicts are possible. If compilation or execution fails, this should be the first thing that you verify. Know thy classpath.

Those familiar with other XML APIs recognize the `Document` and `Element` interfaces (mini-API references for both can be found in Section 17.5). Both interfaces permit calls of `appendChild` to attach DOM elements, and the `Document` interface is most often used to create new DOM elements (that is, `createElement`, `createTextNode`).

But first you must create the DOM Document. The easiest way to do this is by following the example code:

```
Document doc = new org.apache.xerces.dom.DocumentImpl();
```

This might be considered cheating by purists because, theoretically, specific implementation classes should not be created directly. Instead, they would prefer indirect creation via factory classes and interfaces. Xerces2 provides these as well:

```
javax.xml.parsers.DocumentBuilderFactory dbf =
     javax.xml.parsers.DocumentBuilderFactory.newInstance();
javax.xml.parsers.DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.newDocument();
```

WARNING

Beware: If you have installed more than one XML parser implementation, *you are not guaranteed to receive a Xerces2 Document in the* `javax.xml.parsers.DocumentBuilder`*'s* `newDocument` *call*. If you can be certain that the Xerces parser is the only XML parser accessible by your virtual machine, all should go well. But if you must have multiple XML parsers installed, use the Xerces-specific `Document` instantiator call. (There are also other workarounds—see the Xerces DOM samples for more hints.)

Also, notice the four lines of I/O calls that create the XML output string. You can find the `XMLSerializer` class in the `org.apache.xml.serialize` package (this, along with other utility

classes, is introduced in Tables 17.3 through 17.10), which also includes HTML, XHTML, and simple text serializers. These serializer classes can output to either `java.io.OutputStream`s or `java.io.Writer`s (designated by the first parameter to the `*Serializer` constructor). The second parameter of the `XMLSerializer` constructor is an `org.apache.xml.serialize.OutputFormat` object, which typically takes an entire `Document` object (this is a very interesting class—you can take great control of the style and content of the output). See the API documentation for the classes referenced here to see your range of I/O options.

17.3.2 Parsing "Hello Apache"

So now that you have seen how to build a simple XML document in Xerces2, take a brief look at how to load and parse an existing XML document. Listing 17.2 is a small program that takes an XML document's file name as a command-line argument and outputs the file's contents.

LISTING 17.2 HelloApache2 Example

```
import java.io.StringWriter;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.Serializer;
import org.apache.xml.serialize.DOMSerializer;
import org.apache.xml.serialize.SerializerFactory;
import org.apache.xml.serialize.XMLSerializer;

public class HelloApache2
{
  public static void main (String[] args)
  {
    try
    {
      DocumentBuilderFactory factory =
          DocumentBuilderFactory.newInstance();
      DocumentBuilder builder = factory.newDocumentBuilder();
      Document doc = builder.parse(args[0]);

      OutputFormat format = new OutputFormat (doc);
      StringWriter stringOut = new StringWriter ();
      XMLSerializer serial = new XMLSerializer (stringOut,
                                                format);
      serial.serialize(doc);
```

continues

```
      System.out.println(stringOut.toString());
    }
    catch (FactoryConfigurationError e)
    {
      System.out.println
          ("Unable to get a document builder factory: " + e);
    }
    catch (ParserConfigurationException e)
    {
      System.out.println("Parser was unable to be configured: "
                         + e);
    }
    catch (SAXException e)
    {
      System.out.println("Parsing error: " + e);
    }
    catch (IOException e)
    {
      System.out.println("I/O error: " + e);
    }
  }
}
```

Compile `HelloApache2.java` exactly as before, with the explicit classpath specified (or set as you prefer—just do not forget that there are `two` Xerces JAR files that must be made available to the compiler).

If you run your `address.xml` file through `HelloApache2`, remembering that `address.xsd` must be in the same path, you get the somewhat cluttered output shown in Figure 17.6.

You can tell that all your content is present, and if you save this output to a different file, Web browsers or other XML interpreters can read this output. However, if you want a cleaner output, you can use the `IndentPrinter` class from the `org.apache.xml.serialize` package (and a full example is in the included sample class `xerces-2_0_0\samples\xni\PSVIWriter.java`) for some helpful hints. As of Xerces2 version 2.0.0, the only way to output XML cleanly with line breaks and tabs is to write your own serializer; several of the samples do this.

The code specifically responsible for the parsing is found in these three lines:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(args[0]);
```

This should look familiar, because it uses the `DocumentBuilder` and `Factory` classes. But whereas you used the `Document` object to *create* XML elements, you now *receive* a whole XML document, using the `parse` function in `DocumentBuilder`. `DocumentBuilder`'s `parse`

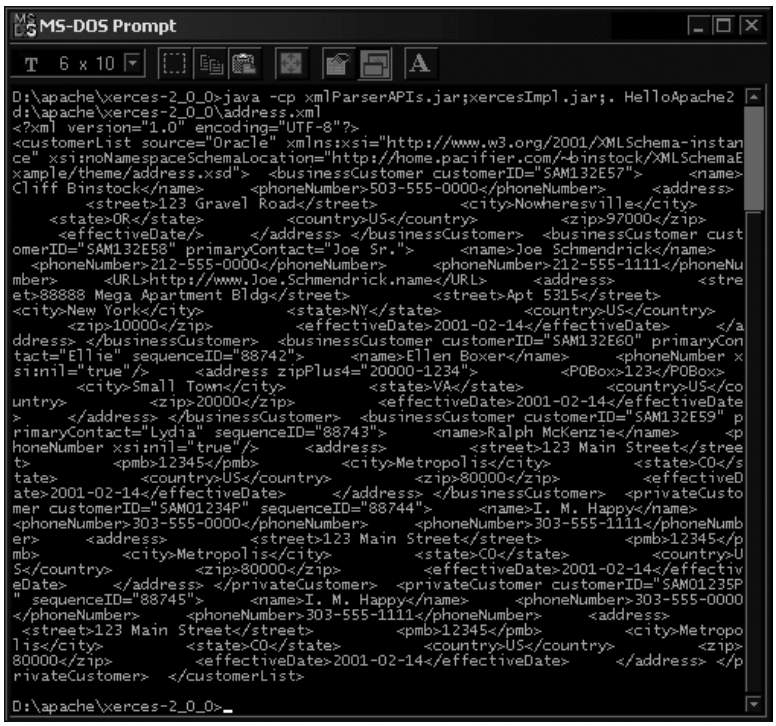


FIGURE 17.6 *Output of* `HelloApache2` *example.*

method can receive either a `java.io.File` or an `org.xml.sax.InputSource` object (the latter can itself accept `java.io.InputStreams` or `java.io.Readers`).

In addition, take note of the four types of exceptions that can be thrown. The first two in Listing 17.2, `javax.xml.parsers.FactoryConfigurationError` and `javax.xml.parsers.ParserConfigurationException`, alert you to major setup or configuration problems; you can then alert the system administrator to verify that only the correct Xerces JAR and class files are fed into the virtual machine. These exceptions might only need to be caught in the initialization stages. After the first successful parse, you probably only need to worry about the other, more common parsing exceptions: `org.xml.sax.SAXException` and `java.io.IOException` (which you should catch in *every* instance of XML document construction and parsing).

WARNING

The DOM parser encapsulates an inner SAX parser, so be prepared to catch SAX exceptions even in pure DOM applications.

17.4 Critical Xerces Packages

You have seen a decent handful of new classes so far, in several different packages. Before going deeper, we want to try to wrap up what you have seen and what you are about to see with a summarized subset of the Xerces API.

Listed in Tables 17.3 through 17.10 are the critical packages, classes, interfaces, and exceptions of the Java Xerces2 parser (with descriptions extracted from the API documentation). Take a look at these names and read the table a few times—if you become familiar with these types, you will soon feel comfortable building applications with Java Xerces. (Note that this is not an exhaustive list, so see the official API reference for a full package, class, interface, and exception list.)

The `java.xml.parsers` package lets you construct and access the actual parsers and document builders.

The `org.w3c.dom` package includes all the DOM Level 2 XML elements, which are listed in Table 17.4.

TABLE 17.3 The `java.xml.parsers` Package

Class	*Descriptions*
`DocumentBuilder`	Defines the API to obtain DOM `Document` instances from an XML document or to create a new DOM `Document`
`DocumentBuilderFactory`	Enables applications to obtain a parser that produces `DOM` object trees from XML documents
`SAXParser`	Defines the API that wraps an `XMLReader` implementation class
`SAXParserFactory`	Enables applications to configure and obtain a SAX-based parser to parse XML documents
Exception	*Description*
`ParserConfigurationException`	Indicates a serious configuration error
Error	*Description*
`FactoryConfigurationError`	Thrown when a problem with configuration of the parser factories exists

TABLE 17.4 The `org.w3c.dom` Package

Interface	*Description*
`Attr`	Represents an attribute in an `Element` object.
`CDATASection`	Used to escape blocks of text containing characters that would otherwise be regarded as markup.
`CharacterData`	Extends `Node` with a set of attributes and methods for accessing character data in the DOM.
`Comment`	Inherits from `CharacterData` and represents the content of a comment, that is, all the characters between the starting '`<!--`' and ending '`-->`'.
`Document`	Represents the entire HTML or XML document.
`DocumentFragment`	A lightweight or minimal `Document` object used to represent portions of an XML `Document` larger than a single node.
`DocumentType`	Each document has a doctype attribute whose value is either null or a `DocumentType` object ("`DocumentType`" as in DTD).
`DOMImplementation`	Provides methods for performing operations independent of a particular document object model instance.
`Element`	The majority of objects (apart from text) in a document are `Element` nodes.
`Entity`	Represents an entity, either parsed or unparsed, in a XML document.
`EntityReference`	May be inserted into the structure model when an entity reference is in the source document or when the user wants to insert an entity reference.
`NamedNodeMap`	Objects implementing the `NamedNodeMap` interface are used to represent collections of nodes that can be accessed by name.
`Node`	The primary datatype for the entire Document Object Model and each of its constituent elements.
`NodeList`	Provides the abstraction of an ordered collection of nodes.
`Notation`	Represents notation declared in the DTD.
`ProcessingInstruction`	Used to place processor-specific information in the text of the document.
`Text`	Inherits from `CharacterData` and represents the textual content (called "character data" in XML) of an `Element` or `Attr`.
Exception	*Description*
`DOMException`	Encapsulates a DOM error.

Use the DOM helper classes in `org.w3c.dom.traversal` in Table 17.5 to perform advanced iteration down your XML trees.

The essence of XML processing with SAX is to create an object that implements special interfaces (which receive SAX messages) and handle only the XML elements that interest you. Table 17.6 lists the primary interfaces you can implement.

TABLE 17.5 The `org.w3c.dom.traversal` Package

Interface	*Description*
`DocumentTraversal`	Contains methods that create `Iterators` to traverse a node and its children.
`NodeFilter`	Filters are objects that know how to filter out nodes (very useful, because the DOM does not provide any filters. These are quite easy to write, as well).
`NodeIterator`	Used for stepping through a set of nodes.
`TreeWalker`	Used to navigate a document tree or subtree using the view of the document.

TABLE 17.6 The `org.w3c.sax` Package

Interface	*Description*
`Attributes`	Interface for a list of XML attributes.
`ContentHandler`	Receives notification of the logical content of a document.
`DTDHandler`	Receives notification of basic DTD-related events.
`EntityResolver`	Basic interface for resolving entities.
`ErrorHandler`	Basic interface for SAX error handlers.
`Locator`	Interface for associating a SAX event with a document location.
`XMLFilter`	Interface for an XML SAX filter—used to build an additional bridge between an `XMLReader` parser and an event-handling client.
`XMLReader`	Interface for reading an XML document by using callbacks.
Class	*Description*
`InputSource`	A single input source for an XML entity

continues

TABLE 17.6 *(continued)*

Exception	*Description*
`SAXException`	Encapsulates a general SAX error or warning.
`SAXNotRecognizedException`	Exception class for an unrecognized identifier.
`SAXNotSupportedException`	Exception class for an unsupported operation.
`SAXParseException`	Encapsulates an XML parse error or warning.

The classes in Table 17.7 help ease the burden of building Java XML SAX applications. Feel free to use them. Because they are part of the standard API, they are supported by all recent Java XML implementations (which means that if you move from Apache-Xerces to something else, your code hardly needs to change).

Use the classes and interfaces in `org.apache.xml.serialize`, shown in Table 17. 8, to construct or utilize output facilities.

TABLE 17.7 The `org.w3c.sax.helpers` Package

Class	*Description*
`AttributesImpl`	Default implementation of the `Attributes` interface.
`DefaultHandler`	Default base class for SAX2 event handlers.
`LocatorImpl`	Optional convenience implementation of `Locator`.
`NamespaceSupport`	Encapsulates namespace logic for use by SAX drivers.
`ParserAdaptor`	Adapts a SAX1 parser as a SAX2 `XMLReader`.
`XMLFilterImpl`	Base class for deriving an XML filter (to sit between an `XMLReader` parser and an event-handling client).
`XMLReaderAdaptor`	Adapts a SAX2 `XMLReader` as a SAX1 parser.
`XMLReaderFactory`	Factory for creating an XML reader.

TABLE 17.8 The `org.apache.xml.serialize` Package

Interface	*Description*
`DOMSerializer`	Interface for a DOM serializer implementation
`Serializer`	Interface for a DOM serializer implementation, factory for DOM and SAX serializers, and static methods for serializing DOM documents

continues

TABLE 17.8 *(continued)*

Class	*Description*
`HTMLSerializer`	Implements an HTML/XHTML serializer supporting both DOM and SAX pretty serializing.
`IndentPrinter`	Extends printer and adds support for indentation and line-wrapping.
`OutputFormat`	Specifies an output format to control the serializer.
`Printer`	Responsible for sending text to the output stream or writer.
`TextSerializer`	Implements a text serializer supporting both DOM and SAX serializing.
`XHTMLSerializer`	Implements an XHTML serializer supporting both DOM and SAX pretty serializing.
`XMLSerializer`	Implements an XML serializer supporting both DOM and SAX pretty serializing.

Additional useful classes are listed in Table 17.9.

If you need direct access to Xerces parsers, study the API for the `org.apache.xerces.parsers` package to access the classes listed in Table 17.10.

TABLE 17.9 The `org.apache.xerces` Package

Class	*Description*
`DOMUtil`	Very useful helper functions—can get the first or last child elements; given a `org.w3c.dom.Node`, can get the name for a `Node`, a `Node`'s siblings, and much more
`EncodingMap`	A convenience class that handles conversions between Internet Assigned Numbers Authority (IANA) encoding names and Java encoding names, and vice versa
`URI`	A class to represent a Uniform Resource Identifier (URI)—designed to handle the parsing of URIs and provide access to the various constituent components (scheme, host, port, userinfo, path, query string, and fragment)

TABLE 17.10 The `org.apache.xerces.parsers` Package

Class	*Description*
`AbstractDOMParser`	The base class of all DOM parsers
`AbstractSAXParser`	The base class of all SAX parsers

continues

TABLE 17.10 *(continued)*

Class	*Description*
`DOMASBuilderImpl`	The Abstract Schema DOM builder class
`DOMBuilderImpl`	The Xerces DOM builder class
`DOMParser`	The main Xerces DOM parser class
`SAXParser`	The main Xerces SAX parser class

17.5 Xerces Java DOM In-depth

This section plunges into the Java DOM, using Xerces to construct a couple of larger applications. It also presents the critical packages and classes you will most likely use while building XML DOM applications. Included are APIs for the three most important DOM interfaces, along with sample code that explores many of their interesting methods.

17.5.1 The Document Interface

Xerces2 implements the DOM Level 2 API, which builds on the original Level 1 core. As you know, DOM is most useful when a new XML document must be created from scratch or when a parsed document must be saved in memory, presumably to be manipulated at a later time. The common Xerces DOM API elements are found in the package `org.w3.dom` and its subpackages.

The `org.w3.dom` package (whose interfaces are described in the previous section) is the starting point from which to construct Java DOM code. Look through Table 17.4, which describes the package `org.w3c.dom`, and see if you can locate which three interfaces *you* think are most important to your task of building DOM applications.

As you saw in Listing 17.2, an object that implements the `Document` interface is returned from the `DocumentBuilder` object via a call to either the `newDocument` or `parse` method. Once you have an object of type `Document`, you may call any of the methods in Table 17.11 to create actual XML elements (the `create*` methods) or access elements (the `get*` methods).

TABLE 17.11 The `org.w3c.dom.Document` Interface

Return Value	*Method Name (parameters)* *Explanation*
`Attr`	`createAttribute(String name)` '`Attr`' is short for '`Attribute`'—here you can create an `Attr` with a name of your choice and then set the value of the `Attr` with a call to `setValue` (`String`).

continues

TABLE 17.11 *(continued)*

Return Value	*Method Name (parameters)* *Explanation*
`Attr`	`createAttributeNS(String namespaceURI, String qualifiedName)` If you want to qualify the `Attribute` with a particular namespace URI, use this method.
`CDATASection`	`createCDATASection(String data)` Create a `CDATASection` and set the value of its data by using this method.
`Comment`	`createComment(String data)` Create a `Comment` and set the value of its data by using this method.
`DocumentFragment`	`createDocumentFragment()` A `DocumentFragment` object is a lightweight version of an XML document. If you want to move portions of a document around, you can use a `DocumentFragment` object to designate this purpose instead of using just a general `Node` object.
`Element`	`createElement(String tagName)` Here is how we commonly create XML document elements, which you can manipulate with one of many method calls. (Remember, `Element` extends the node interface, so all these methods are available.)
`Element`	`createElementNS(String namespaceURI, String qualifiedName)` If you want to qualify the `Element`'s *type name* with a particular namespace URI, use this method.
`EntityReference`	`createEntityReference(String name)` An `EntityReference` object points to another `Entity` somewhere in the XML document. Think of it as a shortcut.
`ProcessingInstruction`	`createProcessingInstruction(String target, String data)` Processing instructions are parser-specific commands (in XML documents, they are wrapped between '`<?`' and '`?>`'). Create them and set their target and data here.
`Text`	`createTextNode(String data)` An `Element` can have one `Text` node. Create and set data here.
`DocumentType`	`getDoctype()` This returns the Document Type Declaration associated with this document.

continues

TABLE 17.11 *(continued)*

Return Value	*Method Name (parameters)* *Explanation*
`Element`	`getDocumentElement()` This returns the root `Element` of the document, which you can then access.
`Element`	`getElementById(String elementId)` This returns an `Element` based on its ID (ID is a specific XML type).
`NodeList`	`getElementsByTagName(String tagname)` All `Elements` with a given tag name are returned in a `NodeList`, in the order in which they are encountered in a pre-order traversal of the `Document` tree.
`NodeList`	`getElementsByTagNameNS(String namespaceURI, String localName)` Same as `getElementsByTagName`, except only `Elements` with a matching namespace are returned in the `NodeList`.
`DOMImplementation`	`getImplementation()` Returns the `DOMImplementation` object that handles this document. Useful for testing whether the DOM implementation supports certain features through the returned interface's `hasFeature (String feature, String version)` method.
`Node`	`importNode(Node importedNode, boolean deep)` Imports a `Node` from another document to this document. The source node is not altered, and the node returned from this call has no parent node.
Plus all methods from `org.w3c.dom.Node`.	

The last line in Table 17.11 is important. The `Document` interface also extends a lower interface: `Node`. As you have probably guessed, `Node` is the second of the three critical `org.w3c.dom` interfaces. Before you get overwhelmed with the APIs, however, practice with a little bit of code by creating a more interesting XML document, with many of the features described in the `Document` interface in Table 17.11.

17.5.2 Creating DOM Documents

In this section, you create your first XML elements. You do this by using a `Document` object that is returned from the `DocumentBuilder` (just like `HelloApache2`). Once you have the `Document` object, you can use that object to create most XML nodes, such as `Text`, `CDATA`, and of course `Elements` themselves.

Every XML node has a corresponding method in the `Document` object. For example, to create a `Comment`, you call `createComment`, with the parameter being the actual comment text. (There is a caveat, with `Attributes`. See Listing 17.3, and we will explain afterward.)

When instantiated, all XML components must be attached to one another; because they are separate objects, they must be able to refer to one another. For instance, once you have created an `Element` and then a `Text` node for that `Element`, you must link them together. You do this by using the `appendChild` method, available in each object. Eventually the root `Element` itself must be attached to something: the `Document`. Then you can serialize out the `Document` object, as in Listing 17.2.

To add a twist, the code also specifies a namespace for this `Element`; you will see the effects after you compile and run the example. Also, instead of using a `StringWriter`, as in `HelloApache2`, you use a `FileOutputStream`. This lets you persist the finalized DOM document out to an XML file, which you can then access and transmit as you prefer (see the `XMLSerializer` class API in the official documentation to discover all the output options you have available).

LISTING 17.3 `HelloApacheDOM` Example

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.Comment;
import org.w3c.dom.CDATASection;
import org.w3c.dom.Text;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.Serializer;
import org.apache.xml.serialize.XMLSerializer;
import java.io.FileOutputStream;

public class HelloApacheDOM
{
  public static void main (String[] args)
  {
    try
    {
      javax.xml.parsers.DocumentBuilderFactory dbf =
        javax.xml.parsers.DocumentBuilderFactory.newInstance();
      javax.xml.parsers.DocumentBuilder db =
        dbf.newDocumentBuilder();
      Document doc = db.newDocument();

      // Create the parent Element object, and add a Comment
      Element root = doc.createElementNS
           ("http://www.galtenberg.net", "books:BOOK");
```

```
  Comment comment = doc.createComment
       ("Publisher 'Beacon Press' address is unknown");
  root.appendChild(comment);

  // Create a child Element with its own Text
  Element item =
      doc.createElementNS("http://www.galtenberg.net",
                          "books:AUTHOR");
  Text text = doc.createTextNode("Bachelard.Gaston");
  item.appendChild(text);
  root.appendChild(item);

  // Do the same as above, but this time add Attributes
  item = doc.createElementNS("http://www.galtenberg.net",
       "books:TITLE");
  text = doc.createTextNode("The Poetics of Reverie");
  Attr attrib = doc.createAttributeNS
       ("http://www.galtenberg.net", "books:ISBN");
  attrib.setValue("0-8070-6413-0");
  // Attributes are attached differently
  item.setAttributeNodeNS(attrib);
  item.appendChild(text);
  root.appendChild(item);

  item = doc.createElementNS("http://www.galtenberg.net",
       "books:TRANSLATOR");
  attrib = doc.createAttributeNS("http://www.galtenberg.net"
       ,"books:ORIGINAL-LANGUAGE");
  attrib.setValue("French");
  item.setAttributeNodeNS(attrib);
  item.appendChild(doc.createTextNode("Daniel Russell"));
  root.appendChild(item);

  item = doc.createElementNS("http://www.galtenberg.net",
       "books:EXCERPT");
  CDATASection cdata = doc.createCDATASection
       ("One does not dream with taught ideas.");
  item.appendChild(cdata);
  root.appendChild(item);

  doc.appendChild(root);
  OutputFormat format  = new OutputFormat (doc);
  FileOutputStream fs  = new FileOutputStream
    ("d:\\helloapache.xml");
  XMLSerializer serial = new XMLSerializer (fs, format);
  serial.serialize(doc);
}
catch (Exception e)
```

continues

```
    {
      e.printStackTrace();
    }
  }
}
```

When you get into this example, you will see that the fundamentals of building XML DOM applications are pretty straightforward. Decide which XML node you would like to utilize, import that interface, declare a type, use the `Document` object to create the node, fill in its data, and attach it to the appropriate `Element`, like this:

```
Element root = doc.createElementNS ("http://www.galtenberg.net",
     "books:BOOK");
Comment comment = doc.createComment
     ("Publisher 'Beacon Press' address is unknown");
root.appendChild(comment);
```

How else could you have created this `Element` (maybe without the namespace)? Look back to Table 17.11, which describes the `Document` interface. Yes, you could call `doc.createElement("BOOK");`.

Now compile and execute your sample. Out on your `D:` drive (make sure to change the code if this is not what you want), you now see the file `helloapache.xml`. When you open it in a Web browser, it should look like Figure 17.7.

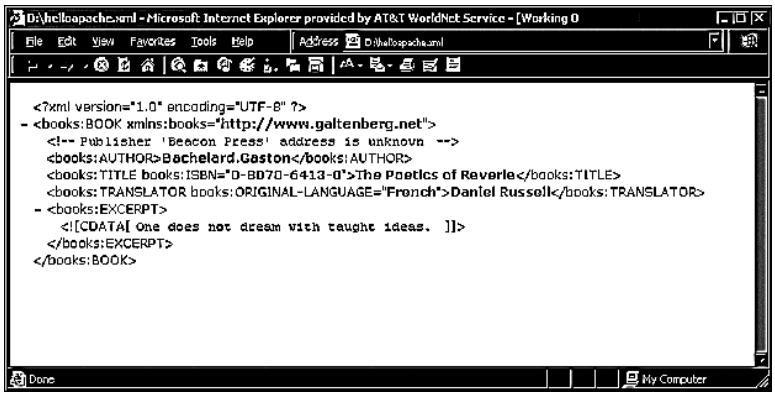


FIGURE 17.7 *Viewing output of* `HelloApacheDOM` *example in Web browser.*

17.5.3 The Element Interface

As mentioned earlier, there is one caveat in the `HelloApacheDOM` example, which is a good lesson if you are new to the XML APIs. When you created an XML attribute (using the `Attr`

interface), you did the normal things: You asked the `Document` to return you an `Attr` object and then you filled in the attribute's value. But unlike with other XML components, you did not call `appendChild` on an `Element` as follows:

```
Element item = doc.createElementNS("http://www.galtenberg.net",
     "books:TRANSLATOR");
Attr attrib = doc.createAttributeNS("http://www.galtenberg.net",
     "books:ORIGINAL-LANGUAGE");
attrib.setValue("French");
item.setAttributeNodeNS(attrib);
```

Instead of calling `appendChild` on `item`, you called `setAttributeNodeNS` with your `Attr` object. `Elements` have their own interface, which deals primarily with XML attributes. Table 17.12 contains `org.w3c.dom.Element`, which is the other critical interface with which you should become acquainted.

TABLE 17.12 The `org.w3c.dom.Element` Interface

Return Value	*Method Name (parameters)*
`String`	`getAttribute(String name)`
`Attr`	`getAttributeNode(String name)`
`Attr`	`getAttributeNodeNS(String namespaceURI, String localName)`
`String`	`getAttributeNS(String namespaceURI, String localName)`
`NodeList`	`getElementsByTagName(String name)`
`NodeList`	`getElementsByTagNameNS(String namespaceURI, String localName)`
`String`	`getTagName()`
`boolean`	`hasAttribute(String name)`
`boolean`	`hasAttributeNS(String namespaceURI, String localName)`
`void`	`removeAttribute(String name)`
`Attr`	`removeAttributeNode(Attr oldAttr)`
`void`	`removeAttributeNS(String namespaceURI, String localName)`
`void`	`setAttribute(String name, String value)`
`Attr`	`setAttributeNode(Attr newAttr)`
`Attr`	`setAttributeNodeNS(Attr newAttr)`
`void`	`setAttributeNS(String namespaceURI, String qualifiedName, String value)`
Plus all methods from `org.w3c.dom.Node`	

NOTE

We omitted the explanation for the methods in Table 17.12 because the naming and parameter patterns are pretty basic. If you want to explore these methods in detail, see the official API documentation.

17.5.4 The Node Interface

That leaves one more critical DOM interface to master. Nearly all the XML types returned from the methods in `Document` (`Attr`, `CDATASection`, `CharacterData`, `Comment`, `DocumentFragment`, `DocumentType`, `Element`, `Entity`, `EntityReference`, `ProcessingInstruction`, `Text`) extend the interface node. You already know the first method well. The rest of the interface methods (from the API documentation) are listed in Table 17.13.

TABLE 17.13 The `org.w3c.dom.Node` Interface

Return Value	*Method Name (parameters)* *Explanation*
`Node`	`appendChild(Node newChild)` Adds the node `newChild` to the end of the list of children of `this` node.
`Node`	`cloneNode(boolean deep)` Returns a duplicate of this node; that is, serves as a generic copy constructor for nodes.
`NamedNodeMap`	`getAttributes()` A `NamedNodeMap` containing the attributes of this node (if an `Element`), or `null` otherwise.
`NodeList`	`getChildNodes()` A `NodeList` that contains all children of this node.
`Node`	`getFirstChild()` The first child of this node.
`Node`	`getLastChild()` The last child of this node.
`String`	`getLocalName()` Returns the local part of the qualified name of this node.
`String`	`getNamespaceURI()` The namespace URI of this node, or `null` if unspecified.

continues

TABLE 17.13 *(continued)*

Return Value	*Method Name (parameters)* *Explanation*
`Node`	`getNextSibling()` The node immediately following this node.
`String`	`getNodeName()` The name of this node, depending on its type.
`short`	`getNodeType()` A code representing the type of the underlying object. (See the official API documentation for a listing of the potential types.)
`String`	`getNodeValue()` The value of this node, depending on its type.
`Document`	`getOwnerDocument()` The `Document` object associated with this node.
`Node`	`getParentNode()` The parent of this node.
`String`	`getPrefix()` The namespace prefix of this node, or `null` if unspecified.
`Node`	`getPreviousSibling()` The node immediately preceding this node.
`boolean`	`hasAttributes()` Returns whether this node (if an element) has any attributes.
`boolean`	`hasChildNodes()` Returns whether this node has any children.
`Node`	`insertBefore(Node newChild, Node refChild)` Inserts the node `newChild` before the existing child node `refChild`.
`boolean`	`isSupported(String feature, String version)` Tests whether the DOM implementation implements a specific feature and that feature is supported by this node.
`void`	`normalize()` Puts all text nodes in the full depth of the subtree underneath this node, including attribute nodes, into a "normal" form where only structure (for example, elements, comments, processing instructions, CDATA sections, and entity references) separates text nodes. That is, there are neither adjacent text nodes nor empty text nodes.

continues

TABLE 17.13 *(continued)*

Return Value	*Method Name (parameters)* *Explanation*
`Node`	`removeChild(Node oldChild)` Removes the child node indicated by `oldChild` from the list of children and returns it.
`Node`	`replaceChild(Node newChild, Node oldChild)` Replaces the child node `oldChild` with `newChild` in the list of children and returns the `oldChild` node.
`void`	`setNodeValue(String nodeValue)`
`void`	`setPrefix(String prefix)`

Note how the `Node` interface only deals with "its own kind" (other `Node`s and `Node`-utility classes). Because it is the base XML DOM interface, this makes sense. All of the methods listed in Table 17.13 are available in the XML node classes, so we highly recommend that you get comfortable with `Node`.

17.5.5 An Advanced DOM Example

There is one more DOM example in this chapter to demonstrate the accessing, parsing, and traversing of an XML document. But this time, you generate your own custom report.

In this example, instead of outputting everything you find in the document, you sort through the `Document` object to find the components that interest you. To do this, you use the `NamedNodeMap` and `NodeList` utility classes.

Think of these as their `java.util` counterparts. `Map`s, which are hashtables, have a key and value (use the key to *retrieve* the value). `List`s are just linked lists of objects (traverse them until there are no more objects). The `NamedNodeMap` and `NodeList` interfaces are quite simple; see the API documentation for the `org.w3c.dom` package.

One other thing to remember: The specific XML objects such as those of type `Text` and `Comments` and `CDATA` were attached as *children* to their parent `Element`. They must be accessed in the same way. The `NamedNodeMap` and `NodeList` hold `Element`s; you must go one level deeper to access your data. Also, you can check the *type* of the `Node` with a call to `getNodeType` (a list of the possible types can be found in the `Node` interface API) to confirm that the child you have accessed is the correct one.

Listing 17.4 shows one possible implementation of a program that generates a report using the XML data created from Listing 17.3.

LISTING 17.4 HelloApacheDOM2 Example

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;

public class HelloApacheDOM2
{
  public static void main (String[] args)
  {
    try
    {
      javax.xml.parsers.DocumentBuilderFactory dbf =
        javax.xml.parsers.DocumentBuilderFactory.newInstance();
      javax.xml.parsers.DocumentBuilder db =
        dbf.newDocumentBuilder();
      Document doc = db.parse(args[0]);

      // Display the root Element
      Element root = doc.getDocumentElement();
      System.out.println("\nDocument Element: Name = " +
        root.getNodeName() +", Value = "+ root.getNodeValue());

      // Traverse through list of the root Element's Attributes
      NamedNodeMap nnm = root.getAttributes();
      System.out.println("# of Attributes: " + nnm.getLength());
      for (int x = 0; x < nnm.getLength(); x++)
      {
        Node n = nnm.item(0);
        System.out.println("Attribute: Name = "
          + n.getNodeName() + ", Value = " + n.getNodeValue());
      }

      // Retrieve author name (Text is a child!)
      NodeList elementList = root.getElementsByTagName
        ("books:AUTHOR");
      String authorName =
          elementList.item(0).getFirstChild().getNodeValue();

      // Do the same for the title
      elementList = root.getElementsByTagName("books:TITLE");
      String bookName =
          elementList.item(0).getFirstChild().getNodeValue();

      // Pull the quote out
      elementList = doc.getElementsByTagName("books:EXCERPT");
```

continues

```
      for (int x = 0; x < elementList.getLength(); x++)
      {
        // This node is books:EXCERPT
        Node node = elementList.item(x);

        // Access CDATA underneath (remember, it's a child)
        Node childNode = node.getFirstChild();
        if (childNode.getNodeType() != Node.CDATA_SECTION_NODE)
          throw new Exception ("This element is not CDATA!");
        System.out.println("\nBook excerpt:");

        String value = childNode.getNodeValue();
        System.out.println(value);
        System.out.println("   —" +authorName+ ", " + bookName);
      }
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Compiling and executing this code should result in the information shown in Figure 17.8 (make sure to add the path to the XML file as an argument).

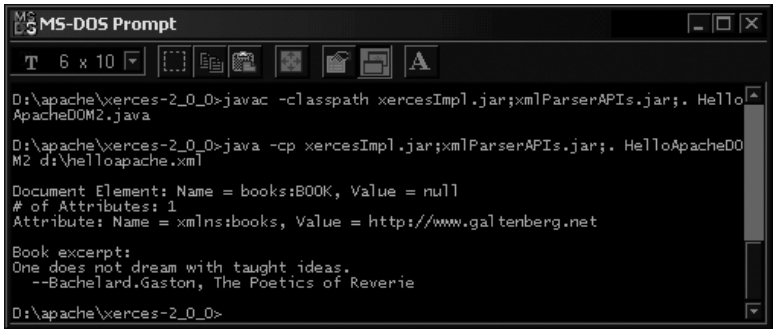


FIGURE 17.8 *Viewing output of* `HelloApacheDOM2` *example.*

17.5.6 DOM Helpers and DOM Level 3

Before moving on from DOM, know that there are also subpackages that provide advanced DOM Level 2 functionality. Take a look at the packages listed in Table 17.14 when you are comfortable with the previous samples and interfaces.

TABLE 17.14 Java Packages for Advanced DOM Functionality

Advanced DOM Package	*Functionality*
`org.w3c.dom.events`	Contains five interfaces (`DocumentEvent, Event, EventListener, EventTarget, MutationEvent`) to provide a generic event system. Useful for defining specific parsing and traversal functionality.
`org.w3c.dom.html`	Contains dozens of interfaces that let you build an HTML document just as you would an XML document; this functionality not only supports the DOM Level 0 specification but may also simplify common and frequent HTML operations.
`org.w3.dom.ranges`	Contains two interfaces (`DocumentRange, Range`) that let you identify and manipulate a range of document content.
`org.w3.dom.traversal`	Contains four interfaces (`DocumentTraversal, NodeFilter, NodeIterator, TreeWalker`) that let you dynamically identify, traverse, and filter a selected range of document content.

Also, if you want to explore DOM Level 3 functionality, see the `org.apache.xerces.dom3` package (and its subpackages) for the latest interfaces and behaviors. Note that this is a parser-specific package whose APIs might change, but if you require the abstract schema and load-and-save features described in the DOM Level 3 working drafts, at least you have this foothold. Make sure to keep current with new Xerces updates. (There are two Xerces discussion groups you can join to receive the latest news, bug reports, fixes, and releases. Click Mailing Lists on the main panel at `http://xml.apache.org` for subscription instructions.)

17.6 Java Xerces SAX In-depth

In this section, we switch over to the SAX XML mindset and write more Java code to explore this new paradigm. By the end of this section, you will have nearly completed your tour of the Xerces parser's XML capabilities, and you will be ready to build complete XML applications on your own.

17.6.1 The `ContentHandler` Interface

The premise of Java SAX is quite simple, but people marvel at the richness of its features. SAX is the epitome of interface- or contract-based development via events. Simply implement one or more of the SAX interfaces, tell the parser you want to be notified when certain XML nodes are found, and designate an XML document. As the document is parsed, your methods are called: one call, or event, for each new XML node discovered. And Apache-Xerces makes it as simple as it sounds.

The SAX paradigm has crystal-clear advantages and disadvantages you need to be aware of before you write your code. Most prominently, in SAX, the XML document's components are not *stored* anywhere (whereas in DOM, *everything* is stored—and accessible via a `Document`-derived object). When a new `Element` is reached, you are handed its name and `Attributes` (if you have requested this behavior of the parser). Store the data, set a flag, skip it—do whatever you want. But when a new `Element` is found and your method is called again, the old `Element` has been completely forgotten (unless your code has taken the data and copied it elsewhere, such as in a report or database).

So remember the fundamentals: Implement one or more SAX interfaces and write code for the XML components you are interested in. Take a look at the most important of the SAX interfaces: `org.w3c.sax.ContentHandler`. Study the methods in Table 17.15, because they are the events that occur as an XML document is parsed.

TABLE 17.15 The `org.w3c.sax.ContentHandler` Interface

Method	*Description*
`void`	`characters(char[] ch, int start, int length)` Here is where you are notified of the content of `Elements` if you implement this method. Caution: Test this event well—sometimes only partial chunks are sent. And *make sure you use the start and length parameters*—often the character buffer contains the whole document (or a large part).
`void`	`endDocument()` When the end of the document has been reached, you receive this event. You receive this event only once, and no other events will follow—this is a good place to do clean-up or finalization.
`void`	`endElement(String namespaceURI, String localName, String qualifiedName)` You receive one of these notifications when the end of the `Element` being parsed is reached. There is one `endElement` call for every `startElement` call.
`void`	`endPrefixMapping(String prefix)` This corresponds to the `startPrefixMapping` call. This event (if appropriate) happens after the `Element`'s `endElement` event.
`void`	`ignorableWhitespace(char[] ch, int start, int length)` If you want to be notified when the parser hits unnecessary whitespace (for custom behavior or error-handling), implement this method.
`void`	`processingInstruction(String target, String data)` Every time a processing instruction (other than an XML or text declaration) is reached, you are notified. (Remember that processing instructions are commands between '`<?`' and '`?>`'.)

continues

TABLE 17.15 *(continued)*

Method	*Description*
`void`	`setDocumentLocator(Locator locator)` Use this method to receive a handy object for locating the origin of SAX document events.
`void`	`skippedEntity(String name)` The parser must notify you if it could not locate a particular DTD (or if the parser doesn't do validation).
`void`	`startDocument()` You receive this notification when parsing begins. This is a good place to do initialization.
`void`	`startElement(String namespaceURI, String localName, String qualifiedName, Attributes atts)` This is an important method to implement. Here you are given the namespace and name and all attributes associated with the `Element`.

`void`	`startPrefixMapping(String prefix, String uri)`
	If you want to implement custom behavior when new prefixes are discovered, implement this method. In the example case of
	`<books:BOOK xmlns:books = "http://www.galtenberg.net">`
	`prefix` would equal '`books`' and `uri` would equal '`http://www.galtenberg.net`'.

Remember that when you implement an interface in Java, you must provide at least a body for every single method. We will show you how to work around this in Listing 17.6, but for now, send your very own `ContentHandler` through the SAX parser and see what comes out.

In Listing 17.5, we are simply going to report when parse events occur, with a `println` statement. Observe where each of the important types is imported from. Note that the `characters` and `ignorableWhitespace` events are handled differently (because their parameters are `char[]`s instead of `String`s). Also pay attention to the parameters passed to each method; some methods have no parameters and thus are essentially pure events.

LISTING 17.5 HelloApacheSAX Example

```
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.Attributes;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

public class HelloApacheSAX implements ContentHandler
{
  public void characters (char[] ch, int start, int length)
  {
    System.out.print("New Characters: ");
    for (int i = 0; i < length; i++)
      System.out.print(ch[start + i]);
    System.out.print("\n");
  }
  public void endDocument ()
  {
    System.out.println("End Document");
  }
  public void endElement (String namespaceURI, String localName,
                          String qualifiedName)
  {
    System.out.println("End Element:  Name = " + localName);
  }
  public void endPrefixMapping (String prefix)
  {
    System.out.println("End Prefix Mapping:  Prefix = "
                       + prefix);
  }
  public void ignorableWhitespace (char[] ch, int start,
                                   int length)
  {
    System.out.print("Ignorable Whitespace: ");
    for (int i = 0; i < length; i++)
      System.out.print(ch[start + i]);
    System.out.print("\n");
  }
  public void processingInstruction (String target, String data)
  {
    System.out.println("Processing Instruction:  Target = "
        + target + ", Data = " + data);
  }
  public void setDocumentLocator (Locator l)
  {
    System.out.println("\nSet Document Locator");
```

```
  }
  public void skippedEntity (String name)
  {
    System.out.println("Skipped entity:  Name = " + name);
  }
  public void startDocument ()
  {
    System.out.println("Start Document");
  }
  public void startElement (String namespace, String localName,
                    String qualifiedName, Attributes attribs)
  {
    System.out.println("Start Element:  Name = " + localName);
  }
  public void startPrefixMapping (String prefix, String uri)
  {
    System.out.println("Start Prefix Mapping:  Prefix = " +
        prefix + ", URI = " + uri);
  }
  public static void main (String[] args)
  {
    try
    {
      XMLReader parser;
      parser = XMLReaderFactory.createXMLReader();

      parser.setContentHandler(new HelloApacheSAX());

      parser.parse(args[0]);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```


After a new parser is created, you simply inform it that *your class* handles notifications with the following call. (It could have been *any* class as long as it implemented the `ContentHandler` interface.)

```
parser.setContentHandler(new HelloApacheSAX());
```

The `XMLReader` is the actual SAX parser. If you are wondering why we did not use the parser and factory from the `javax.xml.parsers` package, the short answer is that the `SAXParser` type does not accept a basic `ContentHandler`-derived object. It expects a `DefaultHandler`-derived object (explained in Listing 17.6). But this is still the same parser: The `SAXParser` object

actually *contains* an `XMLReader` object. You will use the `SAXParser` in the next sample. Simply know that the `XMLReader` can accept any one (or more) of the critical SAX interfaces.

If you compile and run our faithful `helloapache.xml` file through your new parser, the output should look like the screen shown in Figure 17.9.

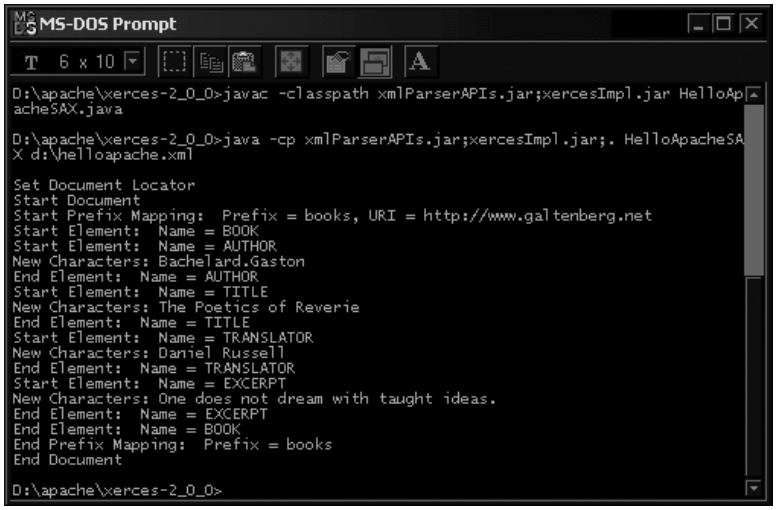


FIGURE 17.9 *Output of the* `HelloApacheSAX` *Example.*

We only want to display element content, so as soon as the AUTHOR element is handled, it is discarded and the parser moves right on to TITLE, and so on. Execution is extremely rapid, and very little memory is used. With SAX and Xerces, you can write a full XML application in mere minutes.

Can we make this any easier? You bet! Instead of implementing the full menu of `ContentHandler` methods, you can simply extend a class, `DefaultHandler` (in the `org.xml.sax.helpers` package), which already implements these methods (although they do not do anything). Now implement methods only for events that interest you. If you only care about catching `processingInstructions`, for example, you need implement only that single method.

`org.xml.sax.helpers.DefaultHandler` implements four SAX interfaces, `ContentHandler`, `DTDHandler`, `EntityResolver`, and `ErrorHandler`, described in Table 17.6 when we introduced the critical SAX packages. You have seen `ContentHandler`—the other three interfaces have only a couple of methods each, for performing lower-level handling (see the API documentation for details). You might be interested in examining `org.xml.sax.ErrorHandler`, because it can also be used in DOM parsing applications. Just add code similar to the following:

```
ErrorHandler handler = object that implements ErrorHandler;
DocumentBuilder builder = object that implements DocumentBuilder;
builder.setErrorHandler(handler);
```

For nearly all SAX applications, `DefaultHandler` is your base class of choice. In Listing 17.6, we use `DefaultHandler` in collaboration with the SAXParser class from the `javax.xml.parsers` package. (Remember, in the future, if you want to specify exact SAX interface-implementations, access the `XMLReader` within the SAX parser.)

Also, you will be catching the full complement of parser exceptions. These should look familiar, because you used them in the `HelloApache2` sample in Listing 17.6.

To make this final example more challenging, there are two additional assigned tasks:

- Use the `address.xml` document to print a report, but this time, only display names and ID information.
- Validate the document's XML schema (found in `address.xsd`).

The first task should be a piece of cake. You need to write just a bit of logic for the `startElement`, `endElement`, and `characters` methods. (And now, because you are using `DefaultHandler`, you do not have to write code for methods you will not use.)

Validating the XML schema is a bit more complicated. But the ride has been pretty smooth to this point, and Xerces makes schemas a breeze, too.

Xerces introduces the notion of properties and features, which are very similar to Java and Visual Basic properties. There is a key and value for each one. Simply set the data for whichever property or feature you care about. The only difference between properties and features is that features are Boolean, like on and off switches: Turn features on or off as you choose. Features can be of any type and are useful for both setting and retrieving specific parser settings.

To set features or properties for SAX processing, access the `XMLReader` object and call either `setFeature` or `setProperty`, with the appropriate key and value. You will find these calls in the `main` function block.

LISTING 17.6 `HelloApacheSAX2` Example

```
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
```

continues

```
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;
import java.io.IOException;

public class HelloApacheSAX2 extends DefaultHandler
{
  // A flag which indicates we've reached the Element
  //   we're interested in
  private boolean bName = false;

  // The three methods of the ErrorHandler interface
  public void error (SAXParseException e)
  {
    System.out.println("\n***Error*** " + e);
  }
  public void warning (SAXParseException e)
  {
    System.out.println("\n***Warning*** " + e);
  }
  public void fatalError (SAXParseException e)
  {
    System.out.println("\n***Fatal Error*** " + e);
  }

  public void startDocument ()
  {
    System.out.println("\n***Start Document***");
  }
  public void endDocument ()
  {
    System.out.println("\n***End Document***");
  }

  // There are many ways to filter out Elements -
  //   this is an elementary example
  public void startElement (String namespace, String localName,
                      String qualifiedName, Attributes attribs)

  {
    if (qualifiedName == "privateCustomer" ||
        qualifiedName == "businessCustomer")
    {
      System.out.println ("\nNew " + qualifiedName + " Entry:");

      for (int i = 0; i < attribs.getLength(); i++)
      {
        System.out.println(attribs.getQName(i) + ": " +
            attribs.getValue(i));
```

```
    }
  }
  else if (qualifiedName == "name")
  {
    bName = true;
  }
}

// Only print characters from the 'name' elements
public void characters (char[] ch, int start, int length)
{
  if (bName == true)
  {
    System.out.print("Name: ");
        for (int i = 0; i < length; i++)
      System.out.print(ch[start + i]);
        System.out.print("\n");
  }
}

// Regardless of what Element we're on, we're done with 'name'
public void endElement (String namespaceURI, String localName,
                        String qualifiedName)
{
  bName = false;
}

public static void main (String[] args)
{
  try
  {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser parser = factory.newSAXParser();

    DefaultHandler handler = new HelloApacheSAX2();
    XMLReader reader = parser.getXMLReader();

    // set parser features
    try
    {
      reader.setFeature
          ("http://xml.org/sax/features/validation", true);
      reader.setFeature
          ("http://apache.org/xml/features/validation/schema",
           true);
      reader.setFeature ("http://apache.org/xml/features/
      validation/warn-on-undeclared-elemdef", true);
```

continues

```
        reader.setProperty
        ("http://apache.org/xml/properties/schema/external-
        noNamespaceSchemaLocation", "address.xsd");
      }
      catch (SAXException e)
      {
        System.out.println
        ("Warning: Parser does not support schema validation");
      }

      parser.parse(args[0], handler);

    }
    catch (FactoryConfigurationError e)
    {
      System.out.println("Factory configuration error: " + e);
    }
    catch (ParserConfigurationException e)
    {
      System.out.println("Parser configuration error: " + e);
    }
    catch (SAXException e)
    {
      System.out.println("Parsing error: " + e);
    }
    catch (IOException e)
    {
      System.out.println("I/O error: " + e);
    }
  }
}
```

You can see that you are using a different method to retrieve your SAX parser, but `XMLReader` is there too (and vital for setting features and properties).

Make sure that both `address.xml` and `address.xsd` are available in your current path as you compile and run the example. Output should look similar to the screen shown in Figure 17.10.

You also need a socket open to the Internet, because `address.xsd` references another XSD document from the book.

Did you notice that execution took longer this time? Maybe you even saw the lag between the '`***StartDocument***`' message and the actual parsing. XML schema validation is time-consuming. Performance should improve over time as later Xerces parsers are released, but there is always a fair amount of overhead (just as there is overhead in using Java rather than a compiled language).

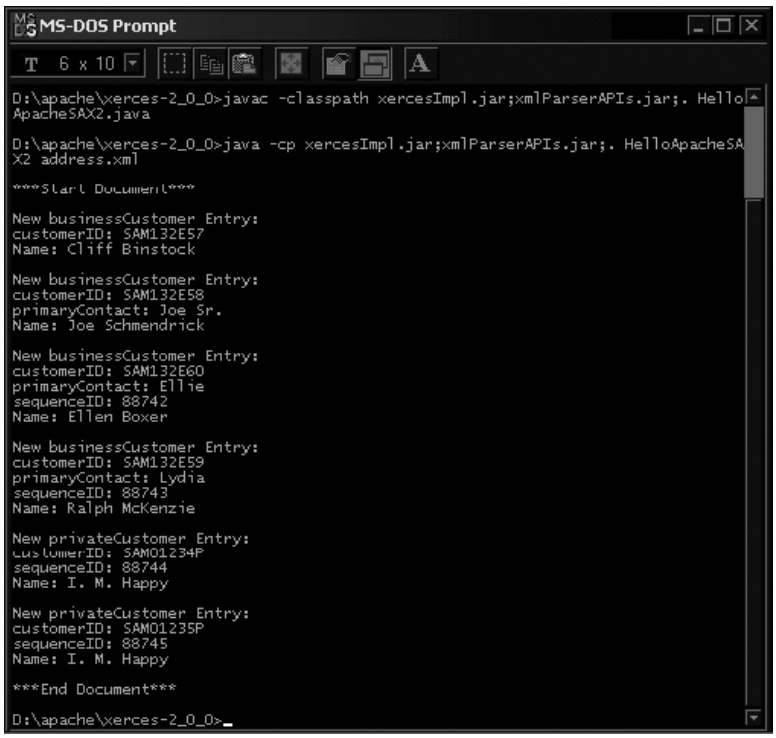


FIGURE 17.10 *Viewing output of* `HelloApacheSAX2` *example.*

But you succeeded in both your tasks. You will of course want to implement a more robust method of filtering `Elements`, and you will not want to hard-code schema locations in your `reader.setProperty` calls.

There are multiple ways to specify Schema functionality and file locations (in DOM as well as SAX—the code is virtually the same). In fact, entire sections in the API documentation are devoted to the dozens of properties and features. Study these well, and remember that they are subject to updates over time, so keep up with the latest Xerces releases.