UNIT II

What Is DOM, Anyway?

The Document Object Model (DOM) provides a way of representing an XML document in memory so that it can be manipulated by your software. DOM is a standard application programming interface (API) that makes it easy for programmers to access elements and delete, add, or edit content and attributes. DOM was proposed by the World Wide Web Consortium (W3C) in August of 1997 in the User Interface Domain. The Activity was eventually moved to the Architecture Domain in November of 2000. Here’s a good place to start looking for DOM-related information: http://www.w3.org/DOM

DOM by itself is just a specification for a set of interfaces defined by W3C. In fact, the DOM interfaces are defined independent of any particular programming language. You can write DOM code in just about any programming language, such as Java, ECMAScript (a standardized version of JavaScript/JScript), or C++. There are DOM APIs for each of these languages. W3C uses the Object Management Group’s (OMG) Interface Definition Language (IDL) to define DOM in a language-neutral way. Language-specific bindings, or DOM interfaces, exist for these languages. The DOM specification itself includes bindings for Java and ECMAScript, but third parties have defined bindings for many other languages.

Any number of organizations provide implementations in accordance with the DOM specification. An implementation is a complete set of APIs for a given programming language that supports the DOM specification. You might suspect that commercial software vendors would sell DOM implementations, but it turns out that there are several opensource and freely available implementations. These implementations are well documented and of high quality. They are commonly used in production software with very good results. This is a result of a well-written specification by W3C. Due to the availability of high-quality free implementations, few if any implementations are sold for profit.

We will look at some of these implementations along with sample code throughout this chapter.

What DOM Is Not

From the preceding discussion, it might be clear to you what the DOM is, but it is also important to highlight what the DOM is not. Here is a brief summary:

- DOM is not a mechanism for persisting, or storing, objects as XML documents. Think of it the other way: DOM is an object model for representing XML documents in your code.

PARSING XML USING DOCUMENT OBJECT MODEL

- DOM is not a set of data structures; rather it is an object model describing XML documents.
- DOM does not specify what information in a document is relevant or how information should be structured.
- DOM has nothing to do with COM, CORBA, or other technologies that include the words object model.
Disadvantages of Using DOM

Although DOM is a W3C specification with support for a variety of programming languages, it’s not necessarily the best solution for all problems. One of the big issues is that DOM can be memory intensive. As mentioned earlier, when an XML document is loaded, the entire document is read in at once. A large document will require a large amount of memory to represent it. Other parsing methods, such as SAX, don’t read in the entire document, so they are better in terms of memory efficiency for some applications.

Some have argued that the DOM API is too complex. Although this is somewhat subjective, it is true that DOM is not practical for small devices such as PDAs and cellular phones. With the rapid proliferation of these devices and demand for greater functionality, XML will very likely play a role in this market. In these cases, DOM as specified by the W3C might not be the best way to go. Fortunately, there are smaller, simpler APIs for XML manipulation that follow the spirit, if not the letter, of DOM. Some of these alternative APIs are discussed later in this chapter.

Of course, everything is relative. If you want to write a quick-and-dirty program without the need for a lot of functionality, you might not require a sophisticated API at all. If all you want to do is generate a relatively simple XML document, you can always write out XML directly and avoid DOM entirely. However, as any veteran programmer knows, that quick-and-dirty code you wrote the midnight before the demo somehow always finds its way into production and becomes a maintenance nightmare!

DOM Levels

The DOM working group works on phases (or levels) of the specification. At the time of this writing, three levels are in the works. The DOM Level 1 and Level 2 specifications are W3C recommendations. This means that the specifications are final and can be implemented without fear of things changing. Level 1 allows traversal of an XML document as well as the manipulation of the content in that document. Level 2 extends Level 1 with additional features such as namespace support, events, ranges, and so on. Level 3 is currently a working draft. This means that it is under active development and subject to change. Details of the developments can be found at the DOM working group Web site (www.w3.org/DOM).

DOM Core

The DOM core is available in DOM Level 1 and beyond. It permits you to create and manipulate XML documents in memory. As mentioned earlier, DOM is a tree structure that represents elements, attributes, and content. As an example, let’s consider a simple XML document, as shown in Listing 7.1.

LISTING 7.1 Simple XML Document

```xml
<purchase-order>
<customer>James Bond</customer>
<merchant>Spies R Us</merchant>
<items>
<item>Night vision camera</item>
</items>
</purchase-order>
```
Parents, Children, and Siblings

In formal computer science literature, lots of different terms are used to describe the parts of a tree structure. You may have run into words such as root, branches, and leaves. This is a bit abstract and doesn’t describe relationships very well, so the DOM specification uses the words parents, children, and siblings to represent nodes and their relationships to one another.

Parent nodes may have zero or more child nodes. Parent nodes themselves may be the child nodes of another parent node. The ultimate parent of all nodes is, of course, the root node. Siblings represent the child nodes of the same parent. These abstract descriptions of nodes are mapped to elements, attributes, text, and other information in an XML document.

DOM interfaces contain methods for obtaining the parent, children, and siblings of any node. The root node has no parent, and there will be nodes that have no children or siblings. After all, the tree has to start and end somewhere!

DOM Interfaces

As mentioned earlier, the DOM interfaces are defined in IDL so that they are language neutral. The DOM specification goes into excruciating detail with respect to the interfaces. Of course, it must—what good is a spec if it is incomplete? A few fundamental interfaces are the most important. If you understand how these interfaces work, you can
solve most problems without learning the entire spec inside and out.

The fundamental interfaces are listed in Table 7.1, along with a brief description of each.

**Table 7.1 Fundamental Interfaces**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>The primary interface for the DOM. It can be an element, attribute, text, and so on, and contains methods for traversing a DOM tree.</td>
</tr>
<tr>
<td>NodeList</td>
<td>An ordered collection of Nodes.</td>
</tr>
<tr>
<td>NamedNodeMap</td>
<td>An unordered collection of Nodes that can be accessed by name and used with attributes.</td>
</tr>
<tr>
<td>DocumentFragment</td>
<td>A Node representing a piece of a document. It’s useful for extracting or inserting a fragment into a document.</td>
</tr>
<tr>
<td>Element</td>
<td>A Node representing an XML element.</td>
</tr>
<tr>
<td>Attr</td>
<td>A Node representing an XML attribute.</td>
</tr>
<tr>
<td>CharacterData</td>
<td>A Node representing character data.</td>
</tr>
<tr>
<td>Text</td>
<td>A CharacterData node representing text.</td>
</tr>
<tr>
<td>Comment</td>
<td>A CharacterData node representing a comment.</td>
</tr>
<tr>
<td>DOMException</td>
<td>An exception raised upon failure of an operation.</td>
</tr>
<tr>
<td>DOMImplementation</td>
<td>Methods for creating documents and determining whether an implementation has certain features.</td>
</tr>
</tbody>
</table>
The diagram in Figure 7.2 shows the relationships among the interfaces described in Table 7.1.

**Figure 7.2**
Interface relationships.

A number of extended interfaces are not mandatory but may be available in some implementations. These interfaces are beyond DOM Level 1 and are discussed later in this chapter. You can determine whether these interfaces are supported by calling the `hasFeature()` method of the `DOMImplementation` interface. You can use the arguments “XML” and “2.0” for the feature and version parameters of the `hasFeature()` method. For a detailed explanation, refer to the DOM specification on the W3C Web site.

The extended interfaces are listed in Table 7.2, along with a brief description of each.

**Table 7.2 Extended Interfaces**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDATASection</td>
<td>Text representing CDATA</td>
</tr>
<tr>
<td>DocumentType</td>
<td>A node representing document type</td>
</tr>
<tr>
<td>Notation</td>
<td>A node with public and system IDs of a notation</td>
</tr>
<tr>
<td>Entity</td>
<td>A node representing an entity that’s either parsed or unparsed</td>
</tr>
<tr>
<td>EntityReference</td>
<td>A node representing an entity reference</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>A node representing an XML processing instruction</td>
</tr>
</tbody>
</table>

### Java Bindings

The DOM working group supplies Java language bindings as part of the DOM specification. The specification and Java language bindings are available at the W3C Web site. These bindings are sets of Java source files containing Java interfaces, and they map exactly to the DOM interfaces described earlier. The package org.w3c.dom contains the Java interfaces but does not include a usable implementation. In order to make the interfaces do something useful, you will need an implementation, or a **parser**.

A number of DOM implementations are available for Java. Two of the most popular are Java APIs for XML Processing (JAXP), developed by Sun Microsystems, and Xerces,
developed as part of the Apache XML project. Both JAXP and Xerces are freely available in source and binary (.class) form. JAXP is available on the Sun Web site at http://java.sun.com/xml/xml _jaxp.html, and Xerces is available on the XML Apache Web site at http://xml.apache.org/xerces2-j/index.html.

LISTING 7.2 SimpleWalker.java

```java
package com.madhu.xml;
import java.io.;
import org.w3c.dom.;
import javax.xml.parsers.;
public class SimpleWalker {
    protected DocumentBuilder docBuilder;
    protected Element root;
    public SimpleWalker() throws Exception {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        docBuilder = dbf.newDocumentBuilder();
        DOMImplementation domImp = docBuilder.getDOMImplementation();
        if (domImp.hasFeature("XML", "2.0")) {
            System.out.println("Parser supports extended interfaces");
        }
    }
    public void parse(String fileName) throws Exception {
        Document doc = docBuilder.parse(new FileInputStream(fileName));
        root = doc.getDocumentElement();
        System.out.println("Root element is " + root.getNodeName());
    }
    public void printAllElements() throws Exception {
        printElement("", root);
    }
    public void printElement(String indent, Node aNode) {
        System.out.println(indent + "<" + aNode.getNodeName() + ">");
        Node child = aNode.getFirstChild();
        while (child != null) {
            printElement(indent + ", child);
            child = child.getNextSibling();
            System.out.println(indent + ">");
        }
    }
    public static void main(String args[]) throws Exception {
        SimpleWalker sw = new SimpleWalker();
        sw.parse(args[0]);
        sw.printAllElements();
    }
}

UNIT 2
LISTING 7.3 library.xml—Sample XML File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <fiction>
    <book>Moby Dick</book>
    <book>The Last Trail</book>
  </fiction>
  <biography>
    <book>The Last Lion, Winston Spencer Churchill</book>
  </biography>
</library>
```

DOM Traversal and Range

Traversal and range are features added in DOM Level 2. They are supported by Apache Xerces. You can determine whether traversal is supported by calling the `hasFeature()` method of the `DOMImplementation` interface. For traversal, you can use the arguments “Traversal” and “2.0” for the feature and version parameters of the `hasFeature()` method.

Traversals

Traversal is a convenient way to walk through a DOM tree and select specific nodes. This is useful when you want to find certain elements and perform operations on them.

Traversal Interfaces

The traversal interfaces are listed in Table 7.4, along with a brief description of each.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeIterator</td>
<td>Used to walk through nodes linearly. Represents a subtree as a linear list.</td>
</tr>
<tr>
<td>TreeWalker</td>
<td>Represents a subtree as a tree view.</td>
</tr>
<tr>
<td>NodeFilter</td>
<td>Can be used in conjunction with NodeIterator and TreeWalker to select specific nodes.</td>
</tr>
<tr>
<td>DocumentTraversal</td>
<td>Contains methods to create NodeIterator and TreeWalker instances.</td>
</tr>
</tbody>
</table>

Range

A range consists of two boundary points corresponding to the start and the end of the range. A boundary point’s position in a Document or DocumentFragment tree can be characterized by a node and an offset. The node is the container of the boundary point and its
position. The container and its ancestors are the ancestor containers of the boundary point and its position. The offset within the node is the offset of the boundary point and its position. If the container is an Attr, Document, DocumentFragment, Element, or EntityReference node, the offset is between its child nodes. If the container is a CharacterData, Comment, or ProcessingInstruction node, the offset is between the 16-bit units of the UTF-16 encoded string contained by it.

The boundary points of a range must have a common ancestor container that is either a Document, DocumentFragment, or Attr node. That is, the content of a range must be entirely within the subtree rooted by a single Document, DocumentFragment, or Attr node. This common ancestor container is known as the root container of the range. The tree rooted by the root container is known as the range’s context tree. The container of a boundary point of a range must be an Element, Comment, ProcessingInstruction, EntityReference, CDATASection, Document, DocumentFragment, Attr, or Text node. None of the ancestor containers of the boundary point of a range can be a DocumentType, Entity, or Notation node.

Range Interfaces

The range interfaces are listed in Table 7.5, along with a brief description of each.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>This interface describes a range and contains methods to define, delete, insert content.</td>
</tr>
<tr>
<td>DocumentRange</td>
<td>This interface creates a range.</td>
</tr>
</tbody>
</table>

Parsing XML Using SAX

What Is SAX, Anyway?
SAX is an API that can be used to parse XML documents. A parser is a program that reads data a character at a time and returns manageable pieces of data. For example, a parser for the English language might break up a document into paragraphs, words, and punctuation. In the case of XML, the important pieces of data include elements, attributes, text, and so on. This is what SAX does.
SAX provides a framework for defining event listeners, or handlers. These handlers are written by developers interested in parsing documents with a known structure. The handlers are registered with the SAX framework in order to receive events. Events can include start of document, start of element, end of element, and so on. The handlers contain a number of methods that will be called in response to these events. Once the handlers are defined and registered, an input source can be specified and parsing can begin.
What SAX Is Not

SAX by itself is just an API, and a number of implementations are available from many of the familiar sources. The most commonly used parsers are Xerces from the Apache XML project and Java API for XML Processing (JAXP) from Sun Microsystems. A good list of parsers can be found at http://www.xmlsoftware.com. SAX was originally developed in Java, but similar implementations are available in other languages as well. There are implementations for Perl, Python, and C++, for example. You can find more information at http://www.meeginson.com/SAX/applications.html.

SAX vs. DOM

As you know, DOM is an in-memory tree structure of an XML document or document fragment. DOM is a natural object model of an XML document, but it’s not always practical. Large documents can take up a lot of memory. This is overkill if all you want to do is find a small piece of data in a very large document.

SAX is, in many ways, much simpler than DOM. There is no need to model every possible type of object that can be found in an XML document. This makes the API easy to understand and easier to use. DOM contains many interfaces, each containing many methods. SAX is comprised of a handful of classes and interfaces. SAX is a much lower-level API when compared with DOM. For these reasons, SAX parsers tend to be smaller than DOM implementations. In fact, many DOM implementations use SAX parsers under the hood to read in XML documents.

SAX is an event-based API. Instead of loading an entire document into memory all at once, SAX parsers read documents and notify a client program when elements, text, comments, and other data of interest are found. SAX parsers send you events continuously, telling you what was found next.

The DOM parses XML in space, whereas SAX parses XML in time. In essence, the DOM parser hands you an entire document and allows you to traverse it any way you like. This can take a lot of memory, so SAX can be significantly more efficient for large documents. In fact, you can process documents larger than available system memory, but this is not possible with DOM. SAX can also be faster, because you don’t have to wait for the entire document to be loaded. This is especially valuable when reading data over a network.

In some cases, you might want to build your own object model of an XML document because DOM might not describe your specific document efficiently or in the way you would like. You could solve the problem by loading a document using DOM and translating the DOM object model into your own object model. However, this can be very inefficient, so SAX is often a better solution.

Disadvantages

SAX is not a perfect solution for all problems. For instance, it can be a bit harder to visualize compared to DOM because it is an event-driven model. SAX parsing is “single pass,” so you can’t back up to an earlier part of the document any more than you can back up from a serial data stream. Moreover, you have no random access at all. Handling parent/child relationships can be more challenging as well.

Building XML-Based Applications

PART II
Another disadvantage is that the current SAX implementations are read-only parsers. They do not provide the ability to manipulate a document or its structure (this feature may be added in the future). DOM is the way to go if you want to manipulate a document in memory.

There is no formal specification for SAX. The interfaces and behavior are defined through existing code bases. This means there is no way to validate a SAX parser or to determine whether it works correctly. In the words of Dave Megginson, “It’s more like English Common Law rather than the heavily codified Civil Code of ISO or W3C specifications.” Even considering these limitations, SAX does its job well. It’s lightweight, simple, and easy to use. If all you want to do is read XML, SAX will probably do what you need.

**SAX Versions**

The first version, SAX 1.0, was released in May 1998. It provided the basic functionality needed to read elements, attributes, text, and to manage errors. There was also some DTD support. The details of SAX 1.0 can be found at http://www.megginson.com/SAX/SAX1/index.html.

The current version, SAX 2.0, was released two years later in May 2000. Many of the SAX 2.0 interfaces are departures from SAX 1.0. Older interfaces are included, but deprecated, for backward compatibility. Adapters are included for using SAX 1.0 parsers with SAX 2.0, and vice versa. SAX 2.0 also includes support for namespaces and extensibility through features and properties. Documentation is improved as well.

**SAX Basics**

To illustrate how SAX works, let’s say you have a simple document, like this one:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fiction>
  <book author="Herman Melville">Moby Dick</book>
</fiction>
```

If you want to parse this document using SAX, you would build a content handler by creating a Java class that implements the ContentHandler interface in the org.xml.sax package. Convenience adapters are available that simplify some of this. Once you have a content handler, you simply register it with a SAX XMLReader, set up the input source, and start the parser. Next, the methods in your content handler will be called when the parser encounters elements, text, and other data. Specifically, the events generated by the preceding example will look something like this: start document, start element: fiction, start element: book (including attributes), characters: Moby Dick, end element: book, end element: fiction, end document.

As you can see, the events reported follow the content of the document in a linear sequence. There are a number of other events that might be generated in response to processing.
instructions, errors, and comments. We will look at these in the examples that follow.

**SAX Packages**

The SAX 2.0 API is comprised of two standard packages and one extension package. The standard packages are org.xml.sax and org.xml.helpers. The org.xml.sax package contains the basic classes, interfaces, and exceptions needed for parsing documents. There, you will find most of the interfaces needed to create handlers for various types of events. We will use many of these classes and interfaces in the sample code later in this chapter. A summary of the org.xml.sax package is shown in Table 8.1.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>Interface for a list of XML attributes.</td>
</tr>
<tr>
<td>ContentHandler</td>
<td>Receives notification of the logical content of a document.</td>
</tr>
<tr>
<td>DocumentHandler</td>
<td>Deprecated. This interface has been replaced by the SaxContentHandler interface, which includes namespace support.</td>
</tr>
<tr>
<td>DTDHandler</td>
<td>Receives notification of basic DTD-related events.</td>
</tr>
<tr>
<td>EntityResolver</td>
<td>Basic interface for resolving entities.</td>
</tr>
<tr>
<td>ErrorHandler</td>
<td>Basic interface for SAX error handlers.</td>
</tr>
<tr>
<td>Locator</td>
<td>Interface for associating a SAX event with a document location.</td>
</tr>
<tr>
<td>Parser</td>
<td>Deprecated. This interface has been replaced by the SaxXMLReader interface, which includes namespace support.</td>
</tr>
<tr>
<td>XMLFilter</td>
<td>Interface for an XML filter.</td>
</tr>
<tr>
<td>XMLReader</td>
<td>Interface for reading an XML document using callback.</td>
</tr>
<tr>
<td>Classes</td>
<td></td>
</tr>
<tr>
<td>HandlerBase</td>
<td>Deprecated. This class works with the deprecated DocumentHandler interface.</td>
</tr>
<tr>
<td>DataSource</td>
<td>A single input source for an XML entity.</td>
</tr>
<tr>
<td>Exceptions</td>
<td></td>
</tr>
<tr>
<td>SAXException</td>
<td>Encapsulates a general SAX error or warning.</td>
</tr>
<tr>
<td>SAXNotSupportedException</td>
<td>Exception class for an unrecognized identifier.</td>
</tr>
<tr>
<td>SAXNotSupportedException</td>
<td>Exception class for an unsupported operation.</td>
</tr>
<tr>
<td>SAXParseException</td>
<td>Encapsulates an XML parse error or warning.</td>
</tr>
</tbody>
</table>

The org.xml.sax.helpers package contains additional classes that can simplify some
### Table 8.2 The `org.xml.sax.helpers` Package

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeListImpl</td>
<td>Deprecated. This class implements a deprecated interface. \ AttributeList that has been replaced by Attributes, which is implemented in the AttributesImpl helper class.</td>
</tr>
<tr>
<td>AttributesImpl</td>
<td>Default implementation of the Attributes interface.</td>
</tr>
<tr>
<td>DefaultHandler</td>
<td>Default base class for SAX2 event handlers.</td>
</tr>
<tr>
<td>LocatorImpl</td>
<td>Provides an optional convenience implementation of Locator.</td>
</tr>
<tr>
<td>NamespaceSupport</td>
<td>Encapsulate namespace logic for use by SAX drivers.</td>
</tr>
<tr>
<td>ParserAdapter</td>
<td>Adapts a SAX1 Parser as a SAX2 XMLReader.</td>
</tr>
<tr>
<td>ParserFactory</td>
<td>Deprecated. This class works with the deprecated Parser interface.</td>
</tr>
<tr>
<td>XMLFilterImpl</td>
<td>Base class for deriving an XML filter.</td>
</tr>
<tr>
<td>XMLReaderAdapter</td>
<td>Adapts a SAX2 XMLReader as a SAX1 Parser.</td>
</tr>
<tr>
<td>XMLReaderFactory</td>
<td>Factory for creating an XML reader.</td>
</tr>
</tbody>
</table>

### Table 8.3 The `org.xml.sax.ext` Package

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeclHandler</td>
<td>SAX2 extension handler for DTD declaration events</td>
</tr>
<tr>
<td>LexicalHandler</td>
<td>SAX2 extension handler for lexical events</td>
</tr>
</tbody>
</table>

### Listing 8.1 SAXDemo.java

```java
package com.madhu.xml;

import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;

public class SAXDemo extends DefaultHandler {
    public void startDocument() {
        System.out.println("***Start of Document***");
    }
}'''
```
public void endDocument() {
    System.out.println("***End of Document***");
}

public void startElement(String uri, String localName, String qName, Attributes attributes) {
    System.out.print("<" + qName);
    int n = attributes.getLength();
    for (int i=0; i<n; i++) {
        System.out.print("" + attributes.getQName(i) + ";" + attributes.getValue(i) + ";");
    }
    System.out.println(">");
}

public void characters(char[] ch, int start, int length) {
    System.out.println(new String(ch, start, length).trim());
}

public void endElement(String namespaceURI, String localName, String qName) throws SAXException {
    System.out.println("</" + qName + ">");
}

public static void main(String args[]) throws Exception {
    if (args.length != 1) {
        System.err.println("Usage: java SAXDemo <xml-file>");
        System.exit(1);
    }

    SAXDemo handler = new SAXDemo();
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser parser = factory.newSAXParser();
    parser.parse(new File(args[0]), handler);
}
TABLE 8.4 The ContentHandler Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>characters()</td>
<td>Receives notification of character data</td>
</tr>
<tr>
<td>endDocument()</td>
<td>Receives notification of the end of a document</td>
</tr>
<tr>
<td>endElement()</td>
<td>Receives notification of the end of an element</td>
</tr>
<tr>
<td>endPrefixMapping()</td>
<td>Ends the scope of a prefix-URI mapping</td>
</tr>
<tr>
<td>ignorableWhitespace()</td>
<td>Receives notification of ignorable whitespace in element content</td>
</tr>
<tr>
<td>processingInstruction()</td>
<td>Receives notification of a processing instruction</td>
</tr>
<tr>
<td>setDocumentLocator()</td>
<td>Receives an object for locating the origin of SAX document events</td>
</tr>
<tr>
<td>skippedEntity()</td>
<td>Receives notification of a skipped entity</td>
</tr>
<tr>
<td>startDocument()</td>
<td>Receives notification of the beginning of a document</td>
</tr>
<tr>
<td>startElement()</td>
<td>Receives notification of the beginning of an element</td>
</tr>
<tr>
<td>startPrefixMapping()</td>
<td>Begins the scope of a prefix-URI namespace mapping</td>
</tr>
</tbody>
</table>

TABLE 8.5 The ErrorHandler Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error()</td>
<td>Receives notification of a recoverable error</td>
</tr>
<tr>
<td>fatalError()</td>
<td>Receives notification of a nonrecoverable error</td>
</tr>
<tr>
<td>warning()</td>
<td>Receives notification of a warning</td>
</tr>
</tbody>
</table>

Entity References

SAX parsers will resolve entity references automatically. However, there are cases when you might want to resolve an entity reference yourself. In the following example, we will define an entity for hardcover books. It will be referenced as &hc; and defined in our DTD. If we use an HTTP URL to define the entity, the SAX parser will go out to the network to resolve it. What we want to do here is resolve the entity using a local file. We can accomplish this using an EntityResolver. The source code for SAXEntity.java is
Transforming XML with XSL

XSL Technologies

XSL has two independent languages:
• The XSL Transformation Language (XSLT)
• The XSL Formatting Object Language (XSL-FO)

XSLT is used to convert an XML document to another format. XSL-FO provides a way of describing the presentation of an XML document. Both technologies use a supporting XML technology, XPath. XPath defines a standard mechanism for accessing elements within a document.

What is XSL?
• XSL is a language that allows one to describe a browser how to process an XML file.
• XSL can convert an XML file into another XML with different format.
• XSL can convert an XML file into a non-XML file.
  • The most common type of XSL processing is to convert XML file into HTML file which can be displayed by browsers. We will focus on this use of XSL.
• XSL is the bridge between XML and HTML.
• We can use XSL to have different HTML formats for the same data represented in XML.
• Separating data (contents) from style tags (display commands).
How does it work?

The XML source document is parsed into an XML source tree
You use XPath to define templates that match parts of the source tree
You use XSLT to transform the matched part and put the transformed information into the result tree
The result tree is output as a result document
Parts of the source document that are not matched by a template are typically copied unchanged

Simple XSLT

<xsl:for-each select="//book"> loops through every book element, everywhere in the document
<xsl:value-of select="title"/> chooses the content of the title element at the current location
<xsl:for-each select="//book">
  <xsl:value-of select="title"/>
</xsl:for-each>
chooses the content of the title element for each book in the XML document

Using XSL to create HTML

Our goal is to turn this:
<?xml version="1.0"?>

<library>
    <book>
        <title>XML</title>
        <author>Gregory Brill</author>
    </book>
    <book>
        <title>Java and XML</title>
        <author>Brett McLaughlin</author>
    </book>
</library>

Into HTML that displays something like

this: Book Titles:
    • XML
    • Java and XML

Book Authors:
    • Gregory Brill
    • Brett McLaughlin

Note that we’ve grouped titles and authors separately

What we need to do?

We need to save our XML into a file (let’s call it books.xml)

We need to create a file (say, books.xsl) that describes how to select elements from books.xml and embed them into an HTML page

We do this by intermixing the HTML and the XSL in the books.xsl file

We need to add a line to our books.xml file to tell it to refer to books.xsl for formatting information

books.xml, revised
Desired HTML

```html
<html>
<head>
<title>Book Titles and Authors</title>
</head>
<body>
<h2>Book titles:</h2>
<ul>
  <li>XML</li>
  <li>Java and XML</li>
</ul>
<h2>Book authors:</h2>
<ul>
  <li>Gregory Brill</li>
  <li>Brett McLaughlin</li>
</ul>
</body>
</html>
```

**XSL outline**

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html> ...
  </xsl:template>
</xsl:stylesheet>
```

**Selecting titles and authors**

- Notice that XSL can rearrange the data; the HTML result can present information in a different order than the XML.
All of books.xml

```xml
<?xml version="1.0"?>
<library>
  <book>
    <title>XML</title>
    <author>Gregory Brill</author>
  </book>
  <book>
    <title>Java and XML</title>
    <author>Brett McLaughlin</author>
  </book>
</library>
```

All of books.xsl

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>Book Titles and Authors</title>
      </head>
      <body>
        <h2>Book titles:</h2>
        <ul>
          <xsl:for-each select="/book">
            <li>
              <xsl:value-of select="title "/>
            </li>
          </xsl:for-each>
        </ul>
        <h2>Book authors:</h2>
        <ul>
          <xsl:for-each select="/book">
            <li>
              <xsl:value-of select="author "/>
            </li>
          </xsl:for-each>
        </ul>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
xsl:value-of

<xsl:value-of select="XPath expression"/> selects the contents of an element and adds it to the output stream

The select attribute is required

Notice that xsl:value-of is not a container, hence it needs to end with a slash

Example (from an earlier slide):

<h1> <xsl:value-of select="message"/> </h1>

xsl:for-each

xsl:for-each is a kind of loop statement

The syntax is

<xsl:for-each select="XPath expression">
    Text to insert and rules to apply
</xsl:for-each>

Example: to select every book (//book) and make an unordered list (<ul>) of their titles (title), use:

<ul>
    <xsl:for-each select="//book">
        <li>  <xsl:value-of select="title"/>  </li>
    </xsl:for-each>
</ul>
Filtering output

You can filter (restrict) output by adding a criterion to the select attribute’s value:

```xml
<ul>
  <xsl:for-each select="//book">
    <li>
      <xsl:value-of select="title[../author='Terry Pratchett']"/>
    </li>
  </xsl:for-each>
</ul>
```

This will select book titles by Terry Pratchett

Here is the filter we just used:

```xml
<xsl:value-of
    select="title[../author='Terry Pratchett']"/>
```

author is a sibling of title, so from title we have to go up to its parent, book, then back down to author

This filter requires a quote within a quote, so we need both single quotes and double quotes

Legal filter operators are:

- `=`
- `!=
- `&lt;`
- `&gt;`

Numbers should be quoted, but apparently don’t have to be

**xsl:if**

xsl:if allows us to include content if a given condition (in the test attribute) is true

Example:

```xml
<xsl:for-each select="//book">
  <xsl:if test="author='Terry Pratchett'">
    <li>
      <xsl:value-of select="title"/>
    </li>
  </xsl:if>
</xsl:for-each>
```

This does work correctly!
**xsl:choose**

The xsl:choose ... xsl:when ... xsl:otherwise construct is XML’s equivalent of Java’s switch ... case ... default statement

The syntax is:

```xml
<xsl:choose>
  <xsl:when test="some condition">
    ... some code ...
  </xsl:when>
  <xsl:otherwise>
    ... some code ...
  </xsl:otherwise>
</xsl:choose>
```

**xsl:sort**

You can place an xsl:sort inside an xsl:for-each

The attribute of the sort tells what field to sort on

Example:

```xml
<ul>
  <xsl:for-each select="/book">
    <xsl:sort select="author"/>
    <li> <xsl:value-of select="title"/> by
      <xsl:value-of select="author"/> </li>
  </xsl:for-each>
</ul>
```

This example creates a list of titles and authors, sorted by author

**xsl:apply-templates**

The <xsl:apply-templates> element applies a template rule to the current element or to the current element’s child nodes

If we add a select attribute, it applies the template rule only to the child that matches

If we have multiple <xsl:apply-templates> elements with select attributes, the child nodes are processed in the same order as the <xsl:apply-templates> elements

UNIT 2
Modeling XML with DataBase

XML Database Solutions

A large number of XML database solutions are available, and they generally come in two flavors: database mapping and native XML support.

XML Database Mapping

The first type of XML database solution provides a mapping between the XML document and the database fields. The system dynamically converts SQL result sets to XML documents. Depending on the sophistication of the product, it may provide a graphical tool to map the database fields to the desired XML elements. Other tools support a configuration file that defines the mapping. These tools continue to store the information in relational database management system (RDBMS) format. They simply provide an XML conversion process that is normally implemented as a server-side Web application.
 modeling databases in xml

in this section, you'll learn how to model a database in xml using java. when we
model a database, we provide an external representation of the database contents. for
our sample program, we'll utilize a database that contains information on rental
properties. we'll model the rental property database as an xml document. figure 10.3
shows the desired output.
the XML data binding features of Java Architecture for XML Binding (JAXB). JAXB provides a framework for representing XML documents as Java objects. Using the JAXB framework, we can guarantee that the documents processed by our system are well formed. Also, we have the option of validating the XML data against a schema.

In the JAXB framework, we can parse XML documents into a suitable Java object. This technique is referred to as **unmarshaling**. The JAXB framework also provides the capability to generate XML documents from Java objects, which is referred to as **marshaling**.

The process is illustrated in the Figure 10.4.

is easier to use and a more efficient technique for processing XML documents than the SAX or DOM API. Using the SAX API, you have to create a custom content handler for each XML document structure. Also, during the development of the content, you have to create and manage your own state machine to keep track of your place in the document. For very complex XML documents, the development process is very cumbersome. Using JAXB, an application can parse an XML document by simply unmarshaling the data from an input stream.

JAXB is similar to DOM in that we can create XML documents programmatically and perform validation. However, the hindrance with DOM is the complex API. If we have an XML tree, using the DOM API, we have to traverse through the tree to retrieve elements. However, with JAXB, we retrieve the data from the XML document by simply calling a method on an object. JAXB allows us to define Java objects that map to XML documents, so we can easily retrieve data. The JAXB framework also ensures the type safety of the data.
JAXB SOLUTION

1. Review the database schema.
2. Construct the desired XML document.
3. Define a schema for the XML document.
4. Create the JAXB binding schema.
5. Generate the JAXB classes based on the schema.
6. Develop a Data Access Object (DAO).
7. Develop a servlet for HTTP access.

FIGURE 10.5
The rental property application architecture.

Reviewing the Database Schema

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop_num</td>
<td>NUMBER</td>
</tr>
<tr>
<td>name</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>street_address</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>city</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>state</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>zip_code</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>size_sq</td>
<td>NUMBER</td>
</tr>
<tr>
<td>bed_count</td>
<td>NUMBER</td>
</tr>
<tr>
<td>bath_count</td>
<td>NUMBER</td>
</tr>
<tr>
<td>monthly_rent</td>
<td>NUMBER</td>
</tr>
<tr>
<td>voice_phone</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>fax_phone</td>
<td>VARCHAR2</td>
</tr>
</tbody>
</table>
Constructing the Desired XML Document

The desired output XML document describes the rental property. However, the XML document does not use the exact field names listed in the database schema. Instead, the XML document provides a custom mapping of the database fields to XML element names. Table 10.4 contains the mapping.

<table>
<thead>
<tr>
<th>Database Field</th>
<th>XML Element Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop_num</td>
<td>&lt;prop_id&gt;</td>
</tr>
<tr>
<td>name</td>
<td>&lt;name&gt;</td>
</tr>
<tr>
<td>street_address</td>
<td>&lt;street&gt;</td>
</tr>
<tr>
<td>city</td>
<td>&lt;city&gt;</td>
</tr>
<tr>
<td>state</td>
<td>&lt;state&gt;</td>
</tr>
<tr>
<td>zip_code</td>
<td>&lt;postal_code&gt;</td>
</tr>
<tr>
<td>size_sq</td>
<td>&lt;square_footage&gt;</td>
</tr>
<tr>
<td>bed_count</td>
<td>&lt;bedrooms&gt;</td>
</tr>
<tr>
<td>bath_count</td>
<td>&lt;bath&gt;</td>
</tr>
<tr>
<td>monthly_rent</td>
<td>&lt;price&gt;</td>
</tr>
<tr>
<td>voice_phone</td>
<td>&lt;phone&gt;</td>
</tr>
<tr>
<td>fax_phone</td>
<td>&lt;fax&gt;</td>
</tr>
</tbody>
</table>
Defining a Schema for the XML Document

Based on the desired document format, we can create a schema definition. In this section, we will define the Document Type Definition (DTD). The DTD schema format was chosen because JAXB 1.0 (early access) only supports DTDs. In the future, JAXB is supposed to support the formal XML Schema definition.

LISTING 10.1  

```xml
<install_dir>/ch10_xmlldb/rental_property.dtd

<!ELEMENT rental_property_list (rental_property)*>
<!ELEMENT rental_property (prop_id, name, address, square_footage, bedrooms, bath, price, contact)>
<!ELEMENT prop_id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, city, state, postal_code)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postal_code (#PCDATA)>
<!ELEMENT square_footage (#PCDATA)>
<!ELEMENT bedrooms (#PCDATA)>
```

```xml
<rental_property>
  <prop_id>1</prop_id>
  <name>The Meadows</name>
  <address>
    <street>251 Eisenhower Blvd</street>
    <city>Houston</city>
    <state>TX</state>
    <postal_code>77033</postal_code>
  </address>
  <square_footage>500.0</square_footage>
  <bedrooms>1.0</bedrooms>
  <bath>1.0</bath>
  <price>600</price>
  <contact>
    <phone>555-555-1212</phone>
    <fax>555-555-1414</fax>
  </contact>
</rental_property>
```
Creating the JAXB Binding Schema

Using the JAXB binding schema, we can define the names of the generated Java classes, map element names to specific properties in the Java class, and provide the mapping rules for attributes. The following code example informs the JAXB system that the element `<rental_property_list>` should be mapped to a Java class and that it is the root element for the XML document:

```
<element name="rental_property_list" type="class" root="true"/>
```

There’s no requirement to define a mapping for every element in the XML document. JAXB uses a default binding schema that will create properties in the Java class based on the XML element name.

The binding schema also allows us to define a conversion rule for elements. For example, the numerical data for the rental property, such as price, square footage, and number of rooms, is always represented in the DTD as text data (#PCDATA). This is one of the limitations of the DTD format. However, by using JAXB, we can specify that a given element should be converted to a Java primitive type or class. In the following code example, we inform JAXB to convert the values of `<square_footage>`, `<bedrooms>`, and `<bath>` to the double type; also, `<price>` is converted to an instance of the `java.math.BigDecimal` class:

```
<element name="square_footage" type="value" convert="double"/>
<element name="bedrooms" type="value" convert="double"/>
<element name="bath" type="value" convert="double"/>  
<element name="price" type="value" convert="BigDecimal"/>  
<conversion name="BigDecimal" type="java.math.BigDecimal"/>
```

We can also use the binding schema to define enumerated types, constructors, and interfaces. However, in the JAXB 1.0 early access version, constructors are not yet implemented.

The binding schema includes a section for controlling the output of the generated Java source code. For example, we can inform the system to use a given package name. The following code defines the package name as `xmlunleashed.ch10.jaxb`:

```
<options package="xmlunleashed.ch10.jaxb"/>
```

See the JAXB specification for details on the binding schema file format.

**LISTING 10.2** `<install_dir>\ch10_xmldb\rental_property.xjs`

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
```
Generating the JAXB Classes Based on Schemas

Now we are ready to generate the Java source files based on our schemas. JAXB provides a schema compiler for generating the Java source files. The schema compiler takes as input the DTD and the JAXB binding schema. Figure 10.6 illustrates the process.

Developing a Data Access Object (DAO)

A Data Access Object (DAO) provides access to the backend database. The goal of the DAO design pattern is to provide a higher level of abstraction for database access. The DAO encapsulates the complex JDBC and SQL calls. The DAO provides access to the backend database via public methods. The DAO converts a result set to a collection of objects. The objects model the data stored in the database. The application interaction with a DAO is shown in Figure 10.8.

UNIT 2
By using a DAO, the implementation details of the database are hidden from the application clients. The implementation details include the database schema and database vendor. This follows closely with the design principle of encapsulation. A benefit of using the DAO is improved application maintenance. If the database schema changes, such as a column name being modified, we only have to update the DAO. No modifications are required to the client programs. Also, if we decide to change the database implementation from Sybase to Oracle, modifications are only required to the DAO. The clients can continue to use the DAO without any modification. The DAO design pattern is widely used in the industry and is documented in Sun’s J2EE Patterns Catalog, found at java.sun.com/j2ee.

In our solution, we’ll create a DAO called RentalPropertyDAO. This version of the DAO will only provide the method `getRentalProperties()`. Later in the chapter, we’ll provide additional methods. The `getRentalProperties()` method submits a SQL query to the database and converts the result set to a collection of JAXB RentalProperty objects.

**Developing a Servlet for HTTP Access**

At this point, we have constructed the RentalPropertyDAO Data Access Object. This DAO is capable of retrieving information from a database and providing a collection of objects. Thanks to the JAXB framework, these objects can be marshaled into XML.