23 XML

CHAPTER GOALS

To learn to use XML elements and attributes

To understand the concept of an XML parser

To read and write XML documents

To design Document Type Definitions for XML documents

CHAPTER CONTENTS

The **Extensible Markup Language** (XML) is a popular mechanism for encoding data. Independent of any programming language, XML allows you to encode complex data in a form that the recipient can easily parse. It is simple enough that a wide variety of programs can generate XML data. XML data has a nested structure, so you can use it to describe hierarchical data sets—for example, an invoice that contains many items, each of which consists of a product and a quantity. Because the XML format is standardized, libraries for parsing the data are widely available and—as you will see in this chapter—easy to use for a programmer.

23.1 XML Tags and Documents

The XML format uses a mixture of text and tags to describe data. *Tags* are enclosed in angle brackets <...>. An *element* is a unit of information that is delimited by a start-tag and a matching end-tag. An element can contain text and other elements. For example, `<city>Sunnyvale</city>` is an element with a text child, and

```
<address>
   <street>1195 W. Fairfield Rd.</street>
   <city>Sunnyvale</city>
   <state>CA</state>
</address>
```

is an element with three child elements. In the following sections, you will see why XML is more useful than a plain text format, how it is related to HTML, and which rules you need to follow when producing an XML document.

23.1.1 Advantages of XML

XML allows you to encode complex data, independent of any programming language, in a form that the recipient can easily parse.

To understand the advantages of using XML for encoding data, let's look at a typical example. We will encode product descriptions, so that they can be transferred to another computer. Your first attempt might be a naïve encoding like this:

```
Toaster
29.95
```

In contrast, here is an XML encoding of the same data:

```
<product>
   <description>Toaster</description>
   <price>29.95</price>
</product>
```

XML files are readable by computer programs and by humans.

The advantage of the XML version is clear: You can look at the data and understand what they mean. Of course, this is a benefit for the programmer, not for a computer program. A computer program has no understanding of what a "price" is. As a programmer, you still need to write code to extract the price as the content of the `price` element. Nevertheless, the fact that an XML document is comprehensible by humans is a huge advantage for program development.

XML-formatted data files are resilient to change.

A second advantage of the XML version is that it is *resilient to change*. Suppose the product data change, and an additional data item is introduced to denote the manufacturer. In the naïve format, the manufacturer might be added after the price, like this:

```
Toaster
29.95
General Appliances
```

A program that can process the old format might get confused when reading a sequence of products in the new format. The program would think that the price is followed by the name of the next product. Thus, the program needs to be updated to work with both the old and new data formats. As data get more complex, programming for multiple versions of a data format can be difficult and time-consuming.

When using XML, on the other hand, it is easy to add new elements:

```
<product>
   <description>Toaster</description>
   <price>29.95</price>
   <manufacturer>General Appliances</manufacturer>
</product>
```

Now a program that processes the new data can still extract the old information in the same way—as the contents of the `description` and `price` elements. The program need not be updated, and it can tolerate different versions of the data format.

23.1.2 Differences Between XML and HTML

If you know HTML, you may have noticed that the XML format of the product data looked somewhat like HTML code. However, there are some differences that we will discuss in this section.

Let's start with the similarities. The XML tag pairs, such as `<price>` and `</price>` look just like HTML tag pairs, for example `<li>` and `</li>`. Both in XML and in HTML, tags are enclosed in angle brackets `< >`, and a start-tag is paired with an end-tag that starts with a slash `/` character.

However, web browsers are quite permissive about HTML. For example, you can omit an end-tag `</li>` and the browser will try to figure out what you mean. In XML, this is not permissible. When writing XML, pay attention to the following rules:

- You *must* pay attention to the letter case of the tags; for example, in XML `<li>` and `<LI>` are different tags that bear no relation to each other.
- Every start-tag *must* have a matching end-tag. You cannot omit tags, such as `</li>`. A tag that ends in `/>` is both a start- and end-tag:

  ```
  <img src="hamster.jpeg"/>
  ```

  When the parser sees the `/>`, it knows not to look for a matching end-tag.
- Finally, attribute values must be enclosed in quotes. For example,

  ```
  <img src="hamster.jpeg" width=400 height=300/>
  ```

  is not acceptable. You must use

  ```
  <img src="hamster.jpeg" width="400" height="300"/>
  ```

Moreover, there is an important conceptual difference between HTML and XML. HTML has one specific purpose: to describe web documents. In contrast, XML is an *extensible* syntax that can be used to specify many different kinds of data. For example, the VRML language uses the XML syntax to describe virtual reality scenes. The MathML language uses the XML syntax to describe mathematical formulas. You can use the XML syntax to describe your own data, such as product records or invoices.

XML describes the meaning of data, not how to display them.

Most people who first see XML wonder how an XML document looks inside a browser. However, that is *not* generally a useful question to ask. Most data that are encoded in XML have nothing to do with browsers. For example, it would probably not be exciting to display an XML document with nothing but product records (such as the ones in the previous section) in a browser. Instead, you will learn in this chapter how to write programs that analyze XML data. XML does not tell you how to display data; it is merely a convenient format for representing data.

23.1.3 The Structure of an XML Document

An XML document starts out with an XML declaration and contains elements and text.

In this section, you will see the rules for properly formatted XML. In XML, text and tags are combined into a *document*. The XML standard recommends that every XML document start with a declaration

```
<?xml version="1.0"?>
```

Next, the XML document contains the actual data. The data are contained in a *root element*. For example,

```
<?xml version="1.0"?>
<invoice>
   more data
</invoice>
```

The `invoice` root element is an example of an XML element. An element has one of two forms:

<elementName> content </elementName>

or

<elementName/>

In the first case, the element has content—elements, text, or a mixture of both. A good example is a paragraph in an HTML document:

```
<p>Use XML for <strong>robust</strong> data formats.</p>
```

The `p` element contains

1. The text: "Use XML for "
2. A `strong` child element
3. More text: " data formats."

An element can contain text, child elements, or both (mixed content). For data descriptions, avoid mixed content.

For XML files that contain documents in the traditional sense of the term, the mixture of text and elements is useful. The XML specification calls this type of content *mixed content*. But for files that describe data sets—such as our product data—it is better to stick with elements that contain *either* other elements or text. Content that consists only of elements is called *element content*.

An element can have *attributes*. For example, the a element of HTML has an `href` attribute that specifies the URL of a hyperlink:

```
<a href="http://horstmann.com"> . . . </a>
```

Elements can have attributes. Use attributes to describe how to interpret the element content.

An attribute has a name (such as `href`) and a value. In XML, the value must be enclosed in single or double quotes.

An element can have multiple attributes, for example

```
<img src="hamster.jpeg" width="400" height="300"/>
```

And, as you have already seen, an element can have both attributes and content.

```
<a href="http://horstmann.com">Cay Horstmann's web site</a>
```

Programmers often wonder whether it is better to use attributes or child elements. For example, should a product be described as

```
<product description="Toaster" price="29.95"/>
```

or

```
<product>
   <description>Toaster</description>
   <price>29.95</price>
</product>
```

The former is shorter. However, it violates the spirit of attributes. Attributes are intended to provide information *about* the element content. For example, the `price` element might have an attribute `currency` that helps interpret the element content. The content 29.95 has a different interpretation in the element

```
<price currency="USD">29.95</price>
```

than it does in the element

```
<price currency="EUR">29.95</price>
```

You have now seen the components of an XML document that are needed to use XML for encoding data. There are other XML constructs for more specialized situations—see `http://www.xml.com/axml/axml.html` for more information. In the next section, you will see how to use Java to parse XML documents.

1. Write XML code with a `student` element and child elements `name` and `id` that describe you.
2. What does your browser do when you load an XML file, such as the `section_2/items.xml` file that is contained in the companion code for this book?
3. Why does HTML use the `src` attribute to specify the source of an image instead of `<img>hamster.jpeg</img>`?

Practice It Now you can try these exercises at the end of the chapter: R23.1, R23.2, R23.3.

HOW TO 23.1 Designing an XML Document Format

This How To walks you through the process of designing an XML document format. You will see in Section 23.4 how to formally describe the format with a document type definition. Right now, we focus on an informal definition of the document content. The "output" of this activity is a sample document.

Step 1 Gather the data that you must include in the XML document.

Write them on a sheet of paper. If at all possible, work from some real-life examples. For example, suppose you need to design an XML document for an invoice. A typical invoice has

- An invoice number
- A shipping address
- A billing address
- A list of items ordered

If possible, gather some actual invoices. Decide which features of the actual invoices you need to include in your XML document.

Step 2 Analyze which data elements need to be refined.

Continue refinement until you reach data values that can be described by single strings or numbers. Make a note of all data items that you discovered during the refinement process. When done, you should have a list of data elements, some of which can be broken down further and some of which are simple enough to be described by a single string or number.

For example, the "shipping address" actually contains the customer name, street, city, state, and ZIP code.

The "list of items ordered" contains items. Each item contains a product and the quantity ordered. Each product contains the product name and price.

Thus, our list now contains

- Address
- Name
- Street
- City
- State
- ZIP code
- List of items ordered
- Item
- Product
- Description
- Price
- Quantity

Keep breaking the data items down until each of them can be described by a single *string* or *number*. For example, an address cannot be described by a single string, but a city can be described by a single string.

Step 3 Come up with a suitable element name that describes the entire XML document.

This element becomes the root element. For example, the invoice data would be contained in an element named `invoice`.

Step 4 Come up with suitable element names for the top-level decomposition that you found in Step 1.

These become the children of the root element. For example, the `invoice` element has children

- `address`
- `items`

Step 5 Repeat this process to give names to the other elements that you discovered in Step 2.

As you do this, make a comprehensive example that shows all elements at work. For the invoice problem, here is an example:

```
<invoice>
   <address>
      <name>ACME Computer Supplies Inc.</name>
      <street>1195 W. Fairfield Rd.</street>
      <city>Sunnyvale</city>
      <state>CA</state>
      <zip>94085</zip>
   </address>
   <items>
      <item>
         <product>
            <description>Ink Jet Refill Kit</description>
            <price>29.95</price>
         </product>
         <quantity>8</quantity>
      </item>
      <item>
         <product>
            <description>4-port Mini Hub</description>
            <price>19.95</price>
         </product>
         <quantity>4</quantity>
      </item>
   </items>
</invoice>
```

Step 6 Check that the document doesn't have mixed content.

That is, make sure each element has as its children either additional elements or text, but not both. If necessary, add more child elements to wrap any text.

For example, suppose the `product` element looked like this:

```
<product>
   <description>Ink Jet Refill Kit</description>
   29.95
</product>
```

Perhaps someone thought it was "obvious" that the last entry was the price. However, following Programming Tip 23.2, it is best to wrap the price inside a `price` element, like this:

```
<product>
   <description>Ink Jet Refill Kit</description>
   <price>29.95</price>
</product>
```

Programming Tip 23.1

Prefer XML Elements over Attributes

Attributes are shorter than elements. For example,

```
<product description="Toaster" price="29.95"/>
```

seems simpler than

```
<product>
   <description>Toaster</description>
   <price>29.95</price>
</product>
```

There is the temptation to use attributes because they are "easier to type". But of course, you don't type XML documents, except for testing purposes. In real-world situations, XML documents are generated by programs.

Attributes are less flexible than elements. Suppose we want to add a currency indication to the value. With elements, that's easy to do:

```
<price currency="USD">29.95</price>
```

or even

```
<price>
   <currency>USD</currency>
   <amount>29.95</amount>
</price>
```

With attributes, you are stuck—you can't refine the structure. Of course, you could use

```
<product description="Toaster" price="USD 29.95"/>
```

But then your program has to parse the string `USD 29.95` and manually take it apart. That's just the kind of tedious and error-prone coding that XML is designed to avoid.

In HTML, there is a simple rule when using attributes. All strings that are not part of the displayed text are attributes. For example, consider a link.

```
<a href="http://horstmann.com">Cay Horstmann's web site</a>
```

The text inside the `a` element, `Cay Horstmann's web site`, is part of what the user sees on the web page, but the `href` attribute value `http://horstmann.com` is not displayed on the page.

Of course, HTML is a little different from the XML documents that you construct to describe data, such as product lists, but the same basic rule applies. Anything that's a part of your data should not be an attribute. An attribute is appropriate only if it tells something *about* the data but isn't a part of the data itself. If you find yourself engaged in metaphysical discussions to determine whether an item is part of the data or tells something about the data, make the item an element, not an attribute.

Programming Tip 23.2

Avoid Children with Mixed Elements and Text

The children of an element can be

1. Elements
2. Text
3. A mixture of both

In HTML, it is common to mix elements and text, for example

```
<p>Use XML for <strong>robust</strong> data formats.</p>
```

But when describing data sets, you should not mix elements and text. For example, you should not do the following:

```
<price>
   <currency>USD</currency>
   29.95
</price>
```

Instead, the children of an element should be either text

```
<price>29.95</price>
```

or elements

```
<price>
   <currency>USD</currency>
   <amount>29.95</amount>
</price>
```

There is an important reason for this design rule. As you will see later in this chapter, you can specify much stricter rules for elements that have only child elements than for elements whose children can contain mixed content.

23.2 Parsing XML Documents

A parser is a program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document.

To read and analyze the contents of an XML document, you need an XML **parser**. A parser is a program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document.

Two kinds of XML parsers are in common use. *Streaming parsers* read the XML input one token at a time and report what they encounter: a start-tag, text, an end-tag, and so on. In contrast, a *tree-based parser* builds a tree that represents the parsed document. Once the parser is done, you can analyze the tree.

A streaming parser reports the building blocks of an XML document. A tree-based parser builds a document tree.

Streaming parsers are more efficient for handling large XML documents whose tree structure would require large amounts of memory. Tree-based parsers, however, are easier to use for most applications—the **parse tree** gives you a complete overview of the data, whereas a streaming parser gives you the information in bits and pieces.

In this section, you will learn how to use a tree-based parser that produces a tree structure according to the DOM (Document Object Model) standard. The DOM standard defines interfaces and methods to analyze and modify the tree structure that represents an XML document.

In order to parse an XML document into a DOM tree, use the `DocumentBuilder` class from the `java.xml` package. To get a `DocumentBuilder` object, first call the static `newInstance` method of the `DocumentBuilderFactory` class, then call the `newDocumentBuilder` method on the factory object:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

A `DocumentBuilder` can read an XML document from a file, URL, or input stream. The result is a `Document` object, which contains a tree.

Once you have a `DocumentBuilder`, you can read a document. To read a document from a file, first construct a `File` object from the file name, then call the `parse` method of the `DocumentBuilder` class:

```
String fileName = . . .;
File f = new File(fileName);
Document doc = builder.parse(f);
```

If the document is located on the Internet, use a URL:

```
String urlName = . . .;
URL u = new URL(urlName);
Document doc = builder.parse(u);
```

You can also read a document from an arbitrary input stream:

```
InputStream in = . . .;
Document doc = builder.parse(in);
```

Once you have created a new document or read a document from a file, you can inspect and modify it.

An XPath describes a node or node set, using a notation similar to that for directory paths.

The easiest method for inspecting a document is the *XPath* syntax. In the DOM standard, a node is the common superclass for all components that make up an XML document. In particular, text sequences and elements are nodes. An XPath describes a node or set of nodes, using a syntax that is similar to directory paths. For example, consider the following XPath, applied to the document in Figure 1 and Figure 2:

```
/items/item[1]/quantity
```

This XPath selects the quantity of the first item, that is, the value 8. (In XPath, array positions start with 1. Accessing `/items/item[0]` would be an error.)

Figure 1
An XML Document

```
<?xml version="1.0"?>
<items>
   <item>
      <product>
         <description>Ink Jet Refill Kit</description>
         <price>29.95</price>
      </product>
      <quantity>8</quantity>
   </item>
   <item>
      <product>
         <description>4-port Mini Hub</description>
         <price>19.95</price>
      </product>
      <quantity>4</quantity>
   </item>
</items>
```

Similarly, you can get the price of the second product as

```
/items/item[2]/product/price
```

To get the number of items, use the XPath expression

```
count(/items/item)
```

In our example, the result is 2.

The total number of children can be obtained as

```
count(/items/*)
```

In our example, the result is again 2 because the `items` element has exactly two children.

To select attributes, use an `@` followed by the name of the attribute. For example,

```
/items/item[2]/product/price/@currency
```

would select the `currency` attribute of the price element if it had one.

Finally, if you have a document with variable or unknown structure, you can find out the name of a child with an expression such as the following:

```
name(/items/item[1]/*[1])
```

The result is the name of the first child of the first item, or `product`.

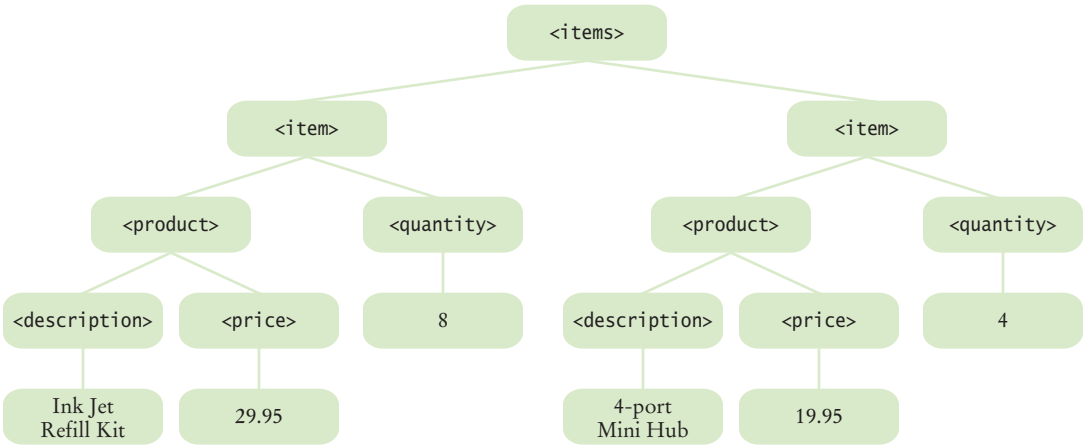

Figure 2
The Tree View of the Document

That is all you need to know about the XPath syntax to analyze simple documents. (See Table 1 for a summary.) There are many more options in the XPath syntax that we do not cover here. If you are interested, look up the specification (`http://www.w3.org/TR/xpath`) or work through the online tutorial (`http://www.zvon.org/xxl/XPathTutorial/General/examples.html`).

To evaluate an XPath expression in Java, first create an `XPath` object:

```
XPathFactory xpfactory = XPathFactory.newInstance();
XPath path = xpfactory.newXPath();
```

Then call the `evaluate` method, like this:

```
String result = path.evaluate(expression, doc)
```

Here, *expression* is an XPath expression and `doc` is the `Document` object that represents the XML document. For example, the statement

```
String result = path.evaluate("/items/item[2]/product/price", doc)
```

sets `result` to the string `"19.95"`.

Now you have all the tools that you need to read and analyze an XML document. The example program at the end of this section puts these techniques to work. (The program uses the `LineItem` and `Product` classes from Section 11.3.) The class `ItemListParser` can parse an XML document that contains a list of product descriptions. Its `parse` method takes the file name and returns an array list of `LineItem` objects:

```
ItemListParser parser = new ItemListParser();
ArrayList<LineItem> items = parser.parse("items.xml");
```

The `ItemListParser` class translates each XML element into an object of the corresponding Java class. We first get the number of items:

```
int itemCount = Integer.parseInt(path.evaluate("count(/items/item)", doc));
```

For each item element, we gather the product data and construct a `Product` object:

```
String description = path.evaluate(
      "/items/item[" + i + "]/product/description", doc);
double price = Double.parseDouble(path.evaluate(
      "/items/item[" + i + "]/product/price", doc));
Product pr = new Product(description, price);
```

Then we construct a `LineItem` object in the same way, and add it to the `items` array list.

Table 1 XPath Syntax Summary

Syntax Element	Purpose	Example
name	Matches an element	`item`
`/`	Separates elements	`/items/item`
`[`*n*`]`	Selects a value from a set	`/items/item[1]`
`@`*name*	Matches an attribute	`price/@currency`
`*`	Matches anything	`/items/*[1]`
`count`	Counts matches	`count(/items/item)`
`name`	The name of a match	`name(/items/*[1])`

Here is the complete source code:

section_2/ItemListParser.java

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.ArrayList;
4  import javax.xml.parsers.DocumentBuilder;
5  import javax.xml.parsers.DocumentBuilderFactory;
6  import javax.xml.parsers.ParserConfigurationException;
7  import javax.xml.xpath.XPath;
8  import javax.xml.xpath.XPathExpressionException;
9  import javax.xml.xpath.XPathFactory;
10 import org.w3c.dom.Document;
11 import org.xml.sax.SAXException;
12
13 /**
14    An XML parser for item lists.
15 */
16 public class ItemListParser
17 {
18    private DocumentBuilder builder;
19    private XPath path;
20
21    /**
22       Constructs a parser that can parse item lists.
23    */
24    public ItemListParser()
25          throws ParserConfigurationException
26    {
27       DocumentBuilderFactory dbfactory
28             = DocumentBuilderFactory.newInstance();
29       builder = dbfactory.newDocumentBuilder();
30       XPathFactory xpfactory = XPathFactory.newInstance();
31       path = xpfactory.newXPath();
32    }
33
34    /**
35       Parses an XML file containing an item list.
36       @param fileName the name of the file
37       @return an array list containing all items in the XML file
38    */
39    public ArrayList<LineItem> parse(String fileName)
40          throws SAXException, IOException, XPathExpressionException
41    {
42       File f = new File(fileName);
43       Document doc = builder.parse(f);
44
45       ArrayList<LineItem> items = new ArrayList<LineItem>();
46       int itemCount = Integer.parseInt(path.evaluate(
47             "count(/items/item)", doc));
48       for (int i = 1; i <= itemCount; i++)
49       {
50          String description = path.evaluate(
51                "/items/item[" + i + "]/product/description", doc);
52          double price = Double.parseDouble(path.evaluate(
53                "/items/item[" + i + "]/product/price", doc));
54          Product pr = new Product(description, price);
55          int quantity = Integer.parseInt(path.evaluate(
56                "/items/item[" + i + "]/quantity", doc));
```

```
57            LineItem it = new LineItem(pr, quantity);
58            items.add(it);
59         }
60         return items;
61      }
62   }
```

section_2/ItemListParserDemo.java

```
1   import java.util.ArrayList;
2
3   /**
4      This program parses an XML file containing an item list.
5      It prints out the items that are described in the XML file.
6   */
7   public class ItemListParserDemo
8   {
9      public static void main(String[] args) throws Exception
10     {
11        ItemListParser parser = new ItemListParser();
12        ArrayList<LineItem> items = parser.parse("items.xml");
13        for (LineItem anItem : items)
14        {
15           System.out.println(anItem.format());
16        }
17     }
18  }
```

Program Run

```
Ink Jet Refill Kit            29.95   8   239.6
4-port Mini Hub               19.95   4   79.8
```

4. What is the result of evaluating the XPath statement `/items/item[1]/product/price` in the XML document of Figure 2?
5. Which XPath statement yields the name of the root element of any XML document?

Practice It Now you can try these exercises at the end of the chapter: R23.10, P23.1, P23.4.

Common Error 23.1

XML Elements Describe Objects, Not Classes

When you convert XML documents to Java classes, you need to determine a class for each element type. A common mistake is to make a separate class for each XML element. For example, consider a slightly different invoice description, with separate shipping and billing addresses:

```
<invoice>
   <shipto>
      <name>ACME Computer Supplies Inc.</name>
      <street>1195 W. Fairfield Rd.</street>
      <city>Sunnyvale</city>
      <state>CA</state>
      <zip>94085</state>
   </shipto>
   <billto>
```

```
      <name>ACME Computer Supplies Inc.</name>
      <street>P.O. Box 11098</street>
      <city>Sunnyvale</city>
      <state>CA</state>
      <zip>94080-1098</zip>
   </billto>
   <items>
   . . .
   </items>
</invoice>
```

Should you have a class `Shipto` to match the `shipto` element and another class `Billto` to match the `billto` element? That makes no sense, because both of them have the same contents: elements that describe an address.

Instead, you should think of the XML element as the value of an instance variable and then determine an appropriate class. For example, an invoice object has instance variables

- `billto`, of type `Address`
- `shipto`, also of type `Address`

Note that you don't see the classes in the XML document. There is no notion of a class `Address` in the XML document describing an invoice. To make element classes explicit, you use an XML schema—see Special Topic 23.2 for more information.

23.3 Creating XML Documents

In the preceding section, you saw how to read an XML file into a `Document` object and analyze the contents of that object. In this section, you will see how to do the opposite—build up a `Document` object and then save it as an XML file. Of course, you can also generate an XML file simply as a sequence of `print` statements. However, that is not a good idea—it is easy to build an illegal XML document in this way, as when data contain special characters such as `<` or `&`.

Recall that you needed a `DocumentBuilder` object to read in an XML document. You also need such an object to create a new, empty document. Thus, to create a new document, first make a `DocumentBuilderFactory`, then a `DocumentBuilder`, and finally the empty document:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.newDocument(); // An empty document
```

The `Document` interface has methods to create elements and text nodes.

A document contains two kind of nodes, elements and text nodes. The DOM standard provides interfaces for these node types, as well as a common superinterface `Node` (see Figure 3). You use the `createElement` method of the `Document` interface to create the elements that you need:

```
Element priceElement = doc.createElement("price");
```

You set element attributes with the `setAttribute` method. For example,

```
priceElement.setAttribute("currency", "USD");
```

You have to work a bit harder for inserting text. First create a text node:

```
Text textNode = doc.createTextNode("29.95");
```

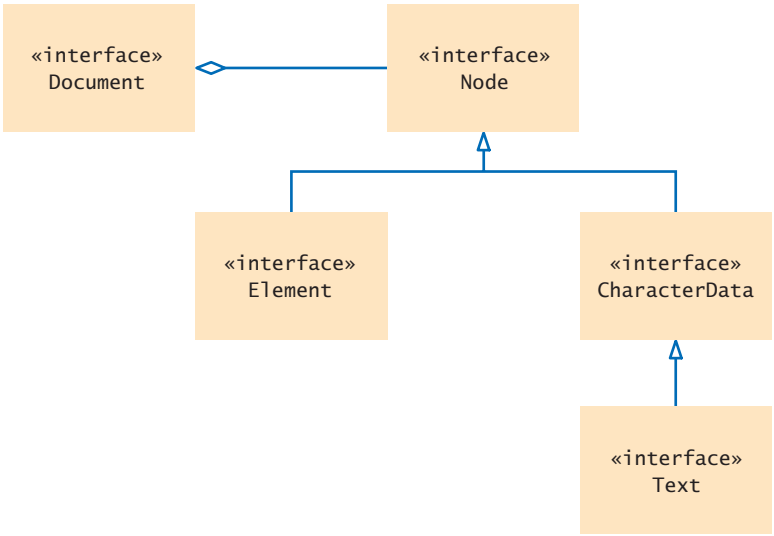

Figure 3 UML Diagram of DOM Interfaces Used in This Chapter

Then add the text node to the element:

```
priceElement.appendChild(textNode);
```

To construct the tree structure of a document, it is a good idea to use a set of helper methods. We start out with a helper method that creates an element with text:

```
private Element createTextElement(String name, String text)
{
   Text t = doc.createTextNode(text);
   Element e = doc.createElement(name);
   e.appendChild(t);
   return e;
}
```

Using this helper method, we can construct a `price` element like this:

```
Element priceElement = createTextElement("price", "29.95");
```

Next, we write a helper method to create a `product` element from a `Product` object:

```
private Element createProduct(Product p)
{
   Element e = doc.createElement("product");
   e.appendChild(createTextElement("description", p.getDescription()));
   e.appendChild(createTextElement("price", "" + p.getPrice()));
   return e;
}
```

This helper method is called from the `createItem` helper method:

```
private Element createItem(LineItem anItem)
{
   Element e = doc.createElement("item");
   e.appendChild(createProduct(anItem.getProduct()));
   e.appendChild(createTextElement("quantity", "" + anItem.getQuantity()));
   return e;
}
```

A helper method

```
private Element createItems(ArrayList<LineItem> items)
```

for the `items` element is implemented in the same way—see the program listing at the end of this section.

Now you build the document as follows:

```
ArrayList<LineItem> items = . . .;
doc = builder.newDocument();
Element root = createItems(items);
doc.appendChild(root);
```

Use an `LSSerializer` to write a DOM document.

Once you have built the document, you will want to write it to a file. The DOM standard provides the `LSSerializer` interface for this purpose. Unfortunately, the DOM standard uses very generic methods, which makes the code that is required to obtain a serializer object look like a "magic incantation":

```
DOMImplementation impl = doc.getImplementation();
DOMImplementationLS implLS
      = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();
```

Once you have the serializer object, you simply use the `writeToString` method:

```
String str = ser.writeToString(doc);
```

By default, the `LSSerializer` produces an XML document without spaces or line breaks. As a result, the output looks less pretty, but it is actually more suitable for parsing by another program because it is free from unnecessary white space.

If you want white space, you use yet another magic incantation after creating the serializer:

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

Here is an example program that shows how to build and print an XML document:

section_3/ItemListBuilder.java

```
1  import java.util.ArrayList;
2  import javax.xml.parsers.DocumentBuilder;
3  import javax.xml.parsers.DocumentBuilderFactory;
4  import javax.xml.parsers.ParserConfigurationException;
5  import org.w3c.dom.Document;
6  import org.w3c.dom.Element;
7  import org.w3c.dom.Text;
8
9  /**
10    Builds a DOM document for an array list of items.
11 */
12 public class ItemListBuilder
13 {
14    private DocumentBuilder builder;
15    private Document doc;
16
17    /**
18       Constructs an item list builder.
19    */
20    public ItemListBuilder()
21          throws ParserConfigurationException
22    {
```

```
23          DocumentBuilderFactory factory
24                = DocumentBuilderFactory.newInstance();
25          builder = factory.newDocumentBuilder();
26       }
27
28       /**
29          Builds a DOM document for an array list of items.
30          @param items the items
31          @return a DOM document describing the items
32       */
33       public Document build(ArrayList<LineItem> items)
34       {
35          doc = builder.newDocument();
36          doc.appendChild(createItems(items));
37          return doc;
38       }
39
40       /**
41          Builds a DOM element for an array list of items.
42          @param items the items
43          @return a DOM element describing the items
44       */
45       private Element createItems(ArrayList<LineItem> items)
46       {
47          Element e = doc.createElement("items");
48
49          for (LineItem anItem : items)
50          {
51             e.appendChild(createItem(anItem));
52          }
53
54          return e;
55       }
56
57       /**
58          Builds a DOM element for an item.
59          @param anItem the item
60          @return a DOM element describing the item
61       */
62       private Element createItem(LineItem anItem)
63       {
64          Element e = doc.createElement("item");
65
66          e.appendChild(createProduct(anItem.getProduct()));
67          e.appendChild(createTextElement(
68                "quantity", "" + anItem.getQuantity()));
69
70          return e;
71       }
72
73       /**
74          Builds a DOM element for a product.
75          @param p the product
76          @return a DOM element describing the product
77       */
78       private Element createProduct(Product p)
79       {
80          Element e = doc.createElement("product");
81
```

```
      e.appendChild(createTextElement(
            "description", p.getDescription()));
      e.appendChild(createTextElement(
            "price", "" + p.getPrice()));

      return e;
   }

   private Element createTextElement(String name, String text)
   {
      Text t = doc.createTextNode(text);
      Element e = doc.createElement(name);
      e.appendChild(t);
      return e;
   }
}
```

section_3/ItemListBuilderDemo.java

```
import java.util.ArrayList;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSSerializer;

/**
   This program demonstrates the item list builder. It prints the XML
   file corresponding to a DOM document containing a list of items.
*/
public class ItemListBuilderDemo
{
   public static void main(String[] args) throws Exception
   {
      ArrayList<LineItem> items = new ArrayList<LineItem>();
      items.add(new LineItem(new Product("Toaster", 29.95), 3));
      items.add(new LineItem(new Product("Hair dryer", 24.95), 1));

      ItemListBuilder builder = new ItemListBuilder();
      Document doc = builder.build(items);
      DOMImplementation impl = doc.getImplementation();
      DOMImplementationLS implLS
            = (DOMImplementationLS) impl.getFeature("LS", "3.0");
      LSSerializer ser = implLS.createLSSerializer();
      String out = ser.writeToString(doc);

      System.out.println(out);
   }
}
```

This program uses the `Product` and `LineItem` classes from Chapter 11. The `LineItem` class has been modified by adding `getProduct` and `getQuantity` methods.

Program Run

```
<?xml version="1.0" encoding="UTF-8"?><items><item><product>
<description>Toaster</description><price>29.95</price></product>
<quantity>3</quantity></item><item><product><description>Hair dryer
</description><price>24.95</price></product><quantity>1</quantity>
</item></items>
```

6. Suppose you need to construct a `Document` object that represents an XML document other than an item list. Which methods from the `ItemListBuilder` class can you reuse?
7. How would you write a document to the file `output.xml`?

Practice It Now you can try these exercises at the end of the chapter: R23.12, P23.10, P23.11.

HOW TO 23.2 Writing an XML Document

What is the best way to write an XML document? This How To shows you how to produce a `Document` object and generate an XML document from it.

Step 1 Provide the outline of a document builder class.

To construct the `Document` object from an object of some class, you should implement a class such as this one:

```
public class MyBuilder
{
   private DocumentBuilder builder;
   private Document doc;

   public Document build(SomeClass x) { . . . }
   . . .
   private Element createTextElement(String name, String text)
   {
      Text t = doc.createTextNode(text);
      Element e = doc.createElement(name);
      e.appendChild(t);
      return e;
   }
}
```

Step 2 Look at the format of the XML document that you want to create.

Consider all elements, except for those that only have text content. Find the matching Java classes. In the `ItemListBuilder` example, we ignore `quantity`, `description`, and `price` because they have text content. The remaining elements and their Java classes are

- `product` - `Product`
- `item` - `LineItem`
- `items` - `ArrayList<LineItem>`

Step 3 For each element in Step 2, add a helper method to your builder class.

Each helper method has the form

```
private Element createElementName(ClassForElement x)
```

For example,

```
public class MyBuilder
{
   . . .
   public Document build(ArrayList<LineItem> x) { . . . }
   private Element createProduct(Product x)  { . . . }
   private Element createItem(LineItem x)  { . . . }
   private Element createItems(ArrayList<LineItem> x)  { . . . }
}
```

Step 4 Implement the helper methods.

For each element, call the helper methods of its children. However, if a child has text content, call `createTextElement` instead.

For example, the `item` element has two children: `product` and `quantity`. The former has a helper method, and the latter has text content. Therefore, the `createItem` method calls `createProduct` and `createTextElement`:

```
private Element createItem(LineItem anItem)
{
   Element e = doc.createElement("item");
   e.appendChild(createProduct(anItem.getProduct()));
   e.appendChild(createTextElement("quantity", "" + anItem.getQuantity()));
   return e;
}
```

You may find it helpful to implement the helper methods "bottom up", starting with the simplest method (such as `createProduct`) and finishing with the method for the root element (`createItems`).

Step 5 Finish off your builder by writing a constructor and the `build` method.

```
public class MyBuilder
{
   public MyBuilder() throws ParserConfigurationException
   {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      builder = factory.newDocumentBuilder();
   }
   public Document build(ClassForRootElement x)
   {
      doc = builder.newDocument();
      doc.appendChild(createRootElementName(x));
      return doc;
   }
   . . .
}
```

Step 6 Use a class, such as the `LSSerializer`, to convert the `Document` to a string.

For example,

```
Invoice x = . . .;
InvoiceBuilder builder = new InvoiceBuilder();
Document doc = builder.build(x);
LSSerializer ser = . . .;
String str = ser.writeToString(doc);
```

Special Topic 23.1

Grammars, Parsers, and Compilers

Grammars are very important in many areas of computer science to describe the structure of computer programs or data formats. To introduce the concept of a grammar, consider this set of rules for a set of simple English language sentences:

1. A sentence has a noun phrase followed by a verb and another noun phrase.
2. A noun phrase consists of an article followed by an adjective list followed by a noun.
3. An adjective list consists of an adjective or an adjective followed by an adjective list.
4. Articles are "a" and "the".
5. Adjectives are "quick", "brown", "lazy", and "hungry".
6. Nouns are "fox", "dog", and "hamster".
7. Verbs are "jumps over" and "eats".

Here are two sentences that follow these rules:

- The quick brown fox jumps over the lazy dog.
- The hungry hamster eats a quick brown fox.

Symbolically, these rules can be expressed by a formal grammar:

```
<sentence> ::= <noun-phrase> <verb> <noun-phrase>
<noun-phrase> ::= <article> <adjective-list> <noun>
<adjective-list> ::= <adjective> |
   <adjective> <adjective-list>
<article> ::= a | the
<adjective> ::= quick | brown | lazy | hungry
<noun> ::= fox | dog | hamster
<verb> ::= jumps over | eats
```

Here the symbol `::=` means "can be replaced with" and `|` separates alternate choices. For example, `<article>` can be replaced with "a" or "the".
The grammar symbols, such as `<noun>`, happen to be enclosed in angle brackets just like XML tags, but they are different from tags. One purpose of a grammar is to produce strings that are valid according to the grammar by starting with the start symbol (`<sentence>` in this example) and applying replacement rules until the resulting string is free from symbols. See Table 2 for an example of the replacement process.

If you have a grammar and a string, such as "the hungry hamster eats a quick brown fox" or "a brown jumps over hamster quick lazy", you can parse the sentence: that is, check whether the sentence is described by the grammar rules and, if it is, show how it can be derived from the start symbol (see Table 2). Another way to show the derivation is to construct a parse tree (see Figure 4).

Table 2 Deriving a Sentence from a Grammar

String	Rule
`<sentence>`	Start
`<noun-phrase> <verb> <noun-phrase>`	1
`<noun-phrase> eats <noun-phrase>`	7
`<article> <adjective-list> <noun> eats <noun-phrase>`	2
`the <adjective-list> <noun> eats <noun-phrase>`	4

Table 2 Deriving a Sentence from a Grammar

String	Rule
the <adjective> <noun> eats <noun-phrase>	3
the hungry <noun> eats <noun-phrase>	5
the hungry hamster eats <noun-phrase>	6
the hungry hamster eats <article> <adjective-list> <noun>	2
the hungry hamster eats a <adjective-list> <noun>	4
the hungry hamster eats a <adjective> <adjective-list> <noun>	3
the hungry hamster eats a quick <adjective-list> <noun>	5
the hungry hamster eats a quick <adjective> <noun>	3
the hungry hamster eats a quick brown <noun>	5
the hungry hamster eats a quick brown fox	6

A parser is a program that reads strings and decides whether the input conforms to the rules of a certain grammar. Some parsers—such as the DOM XML parser—build a parse tree in the process or report an error message when a parse tree cannot be constructed. Other parsers—such as the SAX XML parser—call user-specified methods whenever a part of the input was successfully parsed.

The most important use for parsers is inside compilers for programming languages. Just as our grammar can describe (some) simple English language sentences, the valid "sentences" in a programming language can be described by a grammar. The actual grammar for the Java programming language occupies about fifteen pages in *The Java Language Specification* (`http://docs.oracle.com/javase/specs/jls/se7/html/index.html`). To give a flavor of a small subset of such a grammar, here is a grammar that describes arithmetic expressions.

```
<expression> ::= <term> |
   <expression> <additive-operator> <term>
<additive-operator> ::= + | -
<term> ::= <factor> |
   <term> <multiplicative-operator> <factor>
<multiplicative-operator> ::= * | /
<factor> ::= <integer> | ( <expression> )
<integer> ::= <digits> | - <digits>
<digits> ::= <digit> | <digit> <digits>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

An example of a valid expression in this grammar is

```
-2 * (3 + 10)
```

Try deriving this expression from the `<expression>` start symbol, as was done in Table 2 or Figure 4!

In a compiler, parsing the program source is the first step toward generating code that the target processor (the Java virtual machine in the case of Java) can execute. Writing a parser is a challenging and interesting task. You may at one point in your studies take a course in compiler construction, in which you learn how to write a parser and how to generate code from the parsed input. Fortunately, to use XML you don't have to know how the parser does its job.

You simply ask the XML parser to read the XML input and then process the resulting `Document` tree.

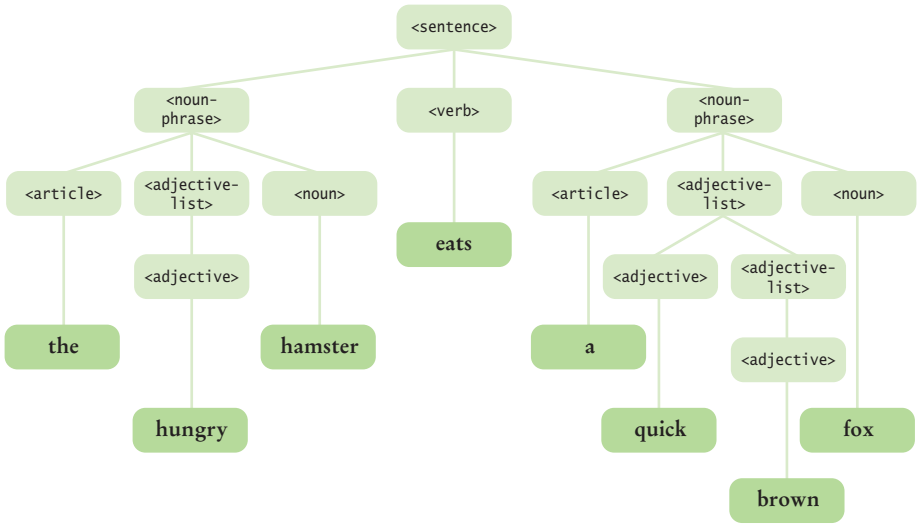

Figure 4 A Parse Tree for a Simple Sentence

23.4 Validating XML Documents

In this section you will learn how to specify rules for XML documents of a particular type. There are several mechanisms for this purpose. The oldest and simplest mechanism is a Document Type Definition (DTD), the topic of this section. We discuss other mechanisms in Special Topic 23.2.

23.4.1 Document Type Definitions

Consider a document of type `items`. Intuitively, `items` denotes a sequence of `item` elements. Each `item` element contains a `product` and a `quantity`. A `product` contains a `description` and a `price`. Each of these elements contains text describing the product's description, price, and quantity. The purpose of a DTD is to formalize this description.

A DTD is a sequence of rules that describes the valid child elements and attributes for each element type.

A DTD is a sequence of rules that describes

- The valid attributes for each element type
- The valid child elements for each element type

Let us first turn to child elements. The valid child elements of an element are described by an `ELEMENT` rule:

```
<!ELEMENT items (item*)>
```

This means that an item list must contain a sequence of 0 or more `item` elements.

As you can see, the rule is delimited by `<! . . .>`, and it contains the name of the element whose children are to be constrained (`items`), followed by a description of what children are allowed.

Table 3 Replacements for Special Characters

Character	Encoding	Name
<	`<`	Less than (left angle bracket)
>	`>`	Greater than (right angle bracket)
&	`&`	Ampersand
'	`&apos;`	Apostrophe
"	`&quot;`	Quotation mark

Next, let us turn to the definition of an `item` element:

```
<!ELEMENT item (product, quantity)>
```

This means that the children of an `item` element must be a `product` element, followed by a `quantity` element.

The definition for a product is similar:

```
<!ELEMENT product (description, price)>
```

Finally, here are the definitions of the three remaining elements:

```
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

The symbol `#PCDATA` refers to text, called "parsed character data" in XML terminology. The character data can contain any characters. However, certain characters, such as < and &, have special meaning in XML and need to be replaced if they occur in character data. Table 3 shows the replacements for special characters.

Table 4 Regular Expressions for Element Content

Rule Description	Element Content
`EMPTY`	No children allowed
(*E*`*`)	Any sequence of 0 or more elements *E*
(*E*`+`)	Any sequence of 1 or more elements *E*
(*E*`?`)	Optional element *E* (0 or 1 occurrences allowed)
(*E*$_1$, *E*$_2$, . . .)	Element *E*$_1$, followed by *E*$_2$, . . .
(*E*$_1$ | *E*$_2$ | . . .)	Element *E*$_1$ or *E*$_2$ or . . .
`(#PCDATA)`	Text only
`(#PCDATA |` *E*$_1$ `|` *E*$_2$ . . .)`*`	Any sequence of text and elements *E*$_1$, *E*$_2$, . . . , in any order
`ANY`	Any children allowed

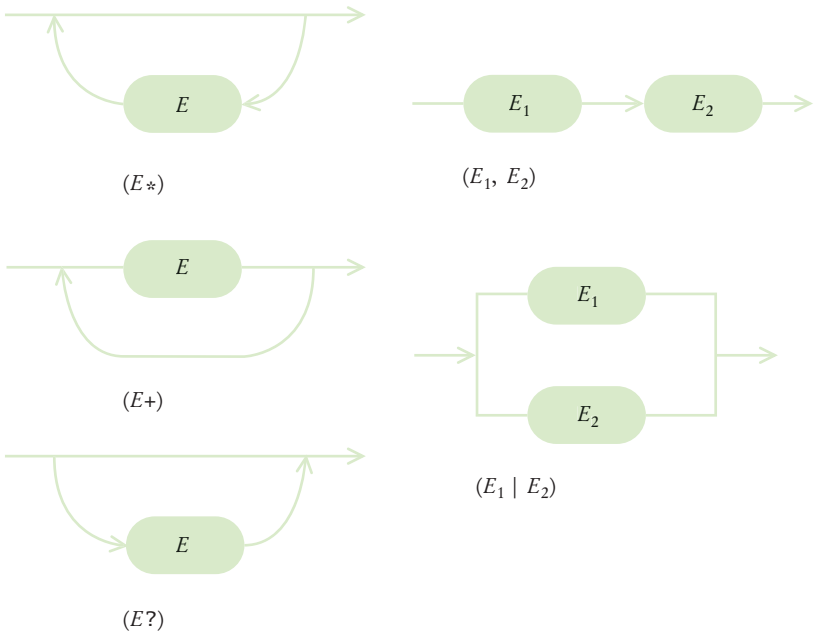

Figure 5
DTD Regular Expression Operations

The complete DTD for an item list has six rules, one for each element type:

```
<!ELEMENT items (item*)>
<!ELEMENT item (product, quantity)>
<!ELEMENT product (description, price)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Let us have a closer look at the descriptions of the allowed children. Table 4 shows the expressions used to describe the children of an element. The `EMPTY` reserved word is self-explanatory: an element that is declared as `EMPTY` may not have any children. For example, the HTML DTD defines the `img` element to be `EMPTY`—an image has only attributes, specifying the image source, size, and placement, and no children.

More interesting child rules can be formed with the **regular expression** operations (`* + ? , |`). (See Table 4 and Figure 5. Also see Special Topic 10.4 for more information on regular expressions.) You have already seen the `*` ("0 or more") and `,` (sequence) operations. The children of an `items` element are 0 or more `item` elements, and the children of an `item` are a sequence of `product` and `description` elements.

You can also combine these operations to form more complex expressions:

```
<!ELEMENT section (title, (paragraph | (image, title?))+)
```

defines an element `section` whose children are:

1. A `title` element
2. A sequence of one or more of the following:
   - `paragraph` elements
   - `image` elements followed by optional `title` elements

Thus,

```
<section>
   <title/>
   <paragraph/>
   <image/>
   <title/>
   <paragraph/>
</section>
```

is valid, but

```
<section>
   <paragraph/>
   <paragraph/>
   <title/>
</section>
```

is not—there is no starting title, and the title at the end doesn't follow an image.

You already saw the `(#PCDATA)` rule. It means that the children can consist of any character data. For example, in our product list DTD, the `description` element can have any character data inside.

You can also allow *mixed content*—any sequence of character data and specified elements. However, in mixed content, you have no control over the order in which the elements appear. As explained in Programming Tip 23.2, you should avoid mixed content for DTDs that describe data sets. This feature is intended for documents that contain both text and markup instructions, such as HTML pages.

Finally, you can allow an element to have children of any type—you should avoid that for DTDs that describe data sets.

You now know how to specify what children an element may have. A DTD also gives you control over the allowed attributes of an element. An attribute description looks like this:

```
<!ATTLIST Element Attribute Type Default>
```

The most useful attribute type descriptions are listed in Table 5. The `CDATA` type describes any sequence of character data. As with `#PCDATA`, certain characters, such as `<` and `&`, need to be encoded (as `<`, `&` and so on). There is no practical difference between the `CDATA` and `#PCDATA` types. Simply use `CDATA` in attribute declarations and `#PCDATA` in element declarations.

Rather than allowing arbitrary attribute values, you can specify a finite number of choices. For example, you may want to restrict a `currency` attribute to U.S. dollar, euro, and Japanese yen. Then use the following declaration:

```
<!ATTLIST price currency (USD | EUR | JPY) #REQUIRED>
```

You can use letters, numbers, and the hyphen (-) and underscore (_) characters for the attribute values.

Table 5 Common Attribute Types

Type Description	Attribute Type
`CDATA`	Any character data
`(`V_1 `|` V_2 `| . . . )`	One of $V_1, V_2, \ldots$

Table 6 Attribute Defaults

Default Declaration	Explanation
`#REQUIRED`	Attribute is required
`#IMPLIED`	Attribute is optional
V	Default attribute, to be used if attribute is not specified
`#FIXED` *V*	Attribute must either be unspecified or contain this value

There are other type descriptions that are less common in practice. You can find them in the XML reference (`http://www.xml.com/axml/axml.html`).

The attribute type description is followed by a "default" declaration. The reserved words that can appear in a "default" declaration are listed in Table 6.

For example, this attribute declaration specifies that each `price` element must have a `currency` attribute whose value is any character data:

```
<!ATTLIST price currency CDATA #REQUIRED>
```

To fulfill this declaration, each `price` element must have a `currency` attribute, such as `<price currency="USD">`. A price without a currency would not be valid.

For an optional attribute, you use the `#IMPLIED` reserved word instead:

```
<!ATTLIST price currency CDATA #IMPLIED>
```

That means that you can supply a currency attribute in a `price` element, or you can omit it. If you omit it, then the application that processes the XML data implicitly assumes some default currency.

A better choice would be to supply the default value explicitly:

```
<!ATTLIST price currency CDATA "USD">
```

That means that the `currency` attribute is understood to mean `USD` if the attribute is not specified. An XML parser will then report the value of `currency` as `USD` if the attribute was not specified.

Finally, you can state that an attribute can only be identical to a particular value. For example, the rule

```
<!ATTLIST price currency CDATA #FIXED "USD">
```

means that a `price` element must either not have a `currency` attribute at all (in which case the XML parser will report its value as `USD`), or specify the `currency` attribute as `USD`. Naturally, this kind of rule is not very common.

You have now seen the most common constructs for DTDs. Using these constructs, you can define your own DTDs for XML documents that describe data sets. In the next section, you will see how to specify which DTD an XML document should use, and how to have the XML parser check that a document conforms to its DTD.

23.4.2 Specifying a DTD in an XML Document

When you reference a DTD with an XML document, you can instruct the parser to check that the document follows the rules of the DTD. That way, the parser can check errors in the document.

An XML document can contain its DTD or refer to a DTD that is stored elsewhere.

In the preceding section you saw how to develop a DTD for a class of XML documents. The DTD specifies the permitted elements and attributes in the document. An XML document has two ways of referencing a DTD:

1. The document may contain the DTD.
2. The document may refer to a DTD that is stored elsewhere.

A DTD is introduced with the DOCTYPE declaration. If the document contains its DTD, then the declaration looks like this:

```
<!DOCTYPE rootElement [ rules ]>
```

For example, an item list can include its DTD like this:

```
<?xml version="1.0"?>
<!DOCTYPE items [

<!ELEMENT items (item*)>
<!ELEMENT item (product, quantity)>
<!ELEMENT product (description, price)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT price (#PCDATA)>

]>
<items>
   <item>
      <product>
         <description>Ink Jet Refill Kit</description>
         <price>29.95</price>
      </product>
      <quantity>8</quantity>
   </item>
   <item>
      <product>
         <description>4-port Mini Hub</description>
         <price>19.95</price>
      </product>
      <quantity>4</quantity>
   </item>
</items>
```

However, if the DTD is more complex, then it is better to store it outside the XML document. In that case, you use the SYSTEM reserved word inside the DOCTYPE declaration to indicate that the system that hosts the XML processor must locate the DTD. The SYSTEM reserved word is followed by the location of the DTD. For example, a DOCTYPE declaration might point to a local file

When referencing an external DTD, you must supply a URL for locating the DTD.

```
<!DOCTYPE items SYSTEM "items.dtd">
```

Alternatively, the resource might be a URL anywhere on the Web:

```
<!DOCTYPE items SYSTEM "http://www.mycompany.com/dtds/items.dtd">
```

For commonly used DTDs, the DOCTYPE declaration can contain a PUBLIC reserved word. For example,

```
<!DOCTYPE faces-config PUBLIC
   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
```

A program parsing the DTD can look at the public identifier. If it is a familiar identifier, then it need not spend time retrieving the DTD from the URL.

23.4.3 Parsing and Validation

When your XML document has a DTD, you can request validation when parsing.

When you include a DTD with an XML document, then you can tell the parser to *validate* the document. That means that the parser will check that all child elements and attributes of an element conform to the `ELEMENT` and `ATTLIST` rules in the DTD. If a document is invalid, then the parser reports an error. To turn on validation, you use the `setValidating` method of the `DocumentBuilderFactory` class before calling the `newDocumentBuilder` method:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(. . .);
```

When you parse an XML file with a DTD, tell the parser to ignore white space.

Validation can simplify your code for processing XML documents. For example, if the DTD specifies that the child elements of each `item` element are `product` and `quantity` elements in that order, then you can rely on that fact and don't need to put tedious checks in your code.

If the parser has access to the DTD, it can make another useful improvement. By default, the parser converts all spaces in the input document to text, even if the spaces are only used to logically line up elements. As a result, the document contains text nodes that are wasteful and can be confusing when you analyze the document tree.

To make the parser ignore white space, call the `setIgnoringElementContentWhitespace` method of the `DocumentBuilderFactory` class.

```
factory.setValidating(true);
factory.setIgnoringElementContentWhitespace(true);
```

Finally, if the parser has access to the DTD, it can fill in default values for attributes. For example, suppose a DTD defines a `currency` attribute for a `price` element:

```
<!ATTLIST price currency CDATA "USD">
```

If a document contains a `price` element without a `currency` attribute, then the parser can supply the default:

```
String attributeValue = priceElement.getAttribute("currency");
   // Gets "USD" if no currency specified
```

This concludes our discussion of XML. You now know enough XML to put it to work for describing data formats. Whenever you are tempted to use a "quick and dirty" file format, you should consider using XML instead. By using XML for data interchange, your programs become more professional, robust, and flexible.

8. How can a DTD specify that the `quantity` element in an `item` is optional?
9. How can a DTD specify that a `product` element can contain a `description` and a `price` element, in any order?
10. How can a DTD specify that the `description` element has an optional attribute `language`?

Practice It Now you can try these exercises at the end of the chapter: R23.13, P23.3, P23.5.

HOW TO 23.3 Writing a DTD

You write a DTD to describe a set of XML documents of the same type. The DTD specifies which elements contain child elements (and the order in which they may appear) and which elements contain text. It also specifies which elements may have attributes, which attributes are required, and which defaults are used for missing attributes.

These rules are for DTDs that describe program data. DTDs that describe narrative text generally have a much more complex structure.

Step 1 Get or write a couple of sample XML documents.

For example, if you wanted to make a DTD for XML documents that describe an invoice, you could study samples such as the one in How To 23.1.

Step 2 Make a list of all elements that can occur in the XML document.

In the invoice example, they are

- `invoice`
- `address`
- `name`
- `street`
- `city`
- `state`
- `zip`
- `items`
- `item`
- `product`
- `description`
- `quantity`

Step 3 For each of the elements, decide whether its children are elements or text.

It is best to avoid elements whose children are a mixture of both.
In the invoice example, the following elements have element content:

- `invoice`
- `address`
- `items`
- `item`
- `product`

The remainder contain text.

Step 4 For elements that contain text, the DTD rule is

```
<!ELEMENT elementName (#PCDATA)>
```

Thus, we have the following simple rules for the invoice elements that contain text:

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Step 5 For each element that contains other elements, make a list of the possible child elements.

Here are the lists in the invoice example:

- `invoice`
  `address`
  `items`
- `address`
  `name`
  `street`
  `city`
  `state`
  `zip`
- `items`
  `item`
- `item`
  `product`
  `quantity`
- `product`
  `description`
  `price`

Step 6 For each of those elements, decide in which order the child elements should occur and how often they should occur.

Then form the rule

<!ELEMENT *elementName* $child_1$ $count_1$, $child_2$ $count_2$, . . .>

where each *count* is one of the following:

Quantity	Count
0 or 1	?
1	omit
0 or more	*
1 or more	+

In the invoice example, the `items` element can contain any number of items, so the rule is

```
<!ELEMENT items (item*)>
```

In the remaining cases, each child element occurs exactly once. That leads to the rules

```
<!ELEMENT invoice (address, items)>
<!ELEMENT address (name, street, city, state, zip)>
<!ELEMENT item (product, quantity)>
<!ELEMENT product (descripton, price)>
```

Step 7 Decide whether any elements should have attributes.

Following Programming Tip 23.1, it is best to avoid attributes altogether or to minimize the use of attributes. Because we have no good reason to add attributes in the invoice example, our invoice is complete without attributes.

Special Topic 23.2

Schema Languages

Several mechanisms have been developed to deal with the limitations of DTDs. DTDs cannot express certain details about the structure of an XML document. For example, you can't force an element to contain just a number or a date—any text string is allowed for a `(#PCDATA)` element.

The XML Schema specification is one mechanism for overcoming these limitations. An XML schema is like a DTD in that it is a set of rules that documents of a particular type need to follow, but a schema can contain far more precise rule descriptions.

Here is just a hint of how an XML schema is specified. For each element, you specify the element name and the type. For example, this definition restricts the contents of `quantity` to an integer.

```
<xsd:element name="quantity" type="xsd:integer"/>
```

Note that an XML schema is itself written in XML—unlike a DTD, which uses a completely different syntax. (The `xsd:` prefix is a *name space* prefix to denote that `xsd:element` and `xsd:integer` are part of the XML Schema Definition name space. See Special Topic 23.3 for more information about name spaces.)

In XML Schema, you can define complex types, much as you define classes in Java. Here is the definition of an `Address` type:

```
<xsd:complexType name="Address">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
```

Then you can specify that an invoice should have `shipto` and `billto` instance variables that are both of type `Address`:

```
<xsd:element name="shipto" type="Address"/>
<xsd:element name="billto" type="Address"/>
```

These examples show that an XML schema can be more precise than a DTD.

The XML Schema specification has many advanced features—see the W3C web site, `www.w3.org/xml`, for details. However, some programmers find that specification overly complex and instead use a competing standard called Relax NG—see `www.relaxng.org`. Relax NG is simpler than XML Schema, and it shares a feature with DTDs: a compact notation that is not XML.

For example, in Relax NG, you simply write

```
element quantity { xsd:integer }
```

to denote that `quantity` is an element containing an integer. The designers of Relax NG realized that XML, despite its many advantages, is not always the best notation for humans.

Special Topic 23.3

Other XML Technologies

This chapter covers the subset of the XML 1.0 specification that is most useful for common programming situations. Since version 1.0 of the XML specification was released, there has been a huge amount of interest in advanced XML technologies. A number of useful technologies have recently been standardized. Among them are:

- Schema Definitions
- Name Spaces
- XHTML
- XSL and Transformations

Special Topic 23.2 contains more information about schema definitions.

Name spaces were invented to ensure that many different people and organizations can develop XML documents without running into conflicts with element names. For example, if you look inside Special Topic 23.2, you will see that XML Schema definitions have element names that are prefixed with a tag `xsd:`, such as

```
<xsd:element name="city" type="xsd:string"/>
```

That way, the tag and attribute names, such as `element` and `string`, don't conflict with other names. In that regard, name spaces are similar to Java packages. However, a name space prefix such as `xsd:` is just a shortcut for the actual name space identifier, which is a much longer, unique string. For example, the full name space for XML Schema definitions is `http://www.w3.org/2000/08/XMLSchema`. Each schema definition starts out with the statement

```
<xsd:schema xmlns:xsd="http://www.w3.org/2000/08/XMLSchema">
```

which binds the `xsd` prefix to the full name space.

XHTML is the most recent recommendation of the W3C for formatting web pages. Unlike HTML, XHTML is fully XML-compliant. Once web-editing tools switch to XHTML, it will become much easier to write programs that parse web pages. The XHTML standard has been carefully designed to be backward compatible with existing browsers.

While XHTML documents are intended to be viewed by browsers, general XML documents are not designed to be viewed at all. Nevertheless, it is often desirable to *transform* an XML document into a viewable form. XSL (Extensible Stylesheet Language) was created for this purpose. A style sheet indicates how to change an XML document into an HTML document, or even a completely different format, such as PDF.

For more information on these and other emerging technologies, see the W3C web site, `http://www.w3.org/xml`.

CHAPTER SUMMARY

Describe the purpose of XML and the structure of an XML document.

- XML allows you to encode complex data, independent of any programming language, in a form that the recipient can easily parse.
- XML files are readable by computer programs and by humans.
- XML-formatted data files are resilient to change.
- XML describes the meaning of data, not how to display them.
- An XML document starts out with an XML declaration and contains elements and text.
- An element can contain text, child elements, or both (mixed content). For data descriptions, avoid mixed content.
- Elements can have attributes. Use attributes to describe how to interpret the element content.

Use a parser and the XPath language to process an XML document.

- A parser is a program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document.
- A streaming parser reports the building blocks of an XML document. A tree-based parser builds a document tree.
- A `DocumentBuilder` can read an XML document from a file, URL, or input stream. The result is a `Document` object, which contains a tree.
- An XPath describes a node or node set, using a notation similar to that for directory paths.

Write Java programs that create XML documents.

- The `Document` interface has methods to create elements and text nodes.
- Use an `LSSerializer` to write a DOM document.

Explain the use of DTDs for validating XML documents.

- A DTD is a sequence of rules that describes the valid child elements and attributes for each element type.

- An XML document can contain its DTD or refer to a DTD that is stored elsewhere.
- When referencing an external DTD, you must supply a URL for locating the DTD.
- When your XML document has a DTD, you can request validation when parsing.
- When you parse an XML file with a DTD, tell the parser to ignore white space.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
javax.xml.parsers.DocumentBuilder
   newDocument
   parse
javax.xml.parsers.DocumentBuilderFactory
   newDocumentBuilder
   newInstance
   setIgnoringElementContentWhitespace
   setValidating
javax.xml.xpath.XPath
   evaluate
javax.xml.xpath.XPathExpressionException
javax.xml.xpath.XPathFactory
   newInstance
   newXPath
org.w3c.dom.Document
   createElement
   createTextNode
   getImplementation
org.w3c.dom.DOMConfiguration
   setParameter
org.w3c.dom.DOMImplementation
   getFeature
org.w3c.dom.Element
   getAttribute
   setAttribute
org.w3c.dom.ls.DOMImplementationLS
   createLSSerializer
org.w3c.dom.ls.LSSerializer
   getDomConfig
   writeToString
org.xml.sax.SAXException
```

REVIEW QUESTIONS

- **R23.1** Give some examples to show the differences between XML and HTML.
- **R23.2** Design an XML document that describes a bank account.
- **R23.3** Draw a tree view for the XML document you created in Exercise R23.2.
- **R23.4** Write the XML document that corresponds to the parse tree in Figure 2.
- **R23.5** Make an XML document describing a book, with child elements for the author name, the title, and the publication year.
- **R23.6** Add a description of the book's language to the document of Exercise R23.5. Should you use an element or an attribute?
- **R23.7** What is mixed content? What problems does it cause?
- **R23.8** Design an XML document that describes a purse containing three quarters, a dime, and two nickels.
- **R23.9** Explain why a paint program, such as Microsoft Paint, is a WYSIWYG program that is also "what you see is all you've got".

■■ **R23.10** Consider the XML file

```
<purse>
   <coin>
      <value>0.5</value>
      <name lang="en">half dollar</name>
   </coin>
   <coin>
      <value>0.25</value>
      <name lang="en">quarter</name>
   </coin>
</purse>
```

What are the values of the following XPath expressions?

a. `/purse/coin[1]/value`

b. `/purse/coin[2]/name`

c. `/purse/coin[2]/name/@lang`

d. `name(/purse/coin[2]/*[1])`

e. `count(/purse/coin)`

f. `count(/purse/coin[2]/name)`

■■ **R23.11** With the XML file of Exercise R23.10, give XPath expressions that yield

a. the value of the first coin.

b. the number of coins.

c. the name of the first child element of the first `coin` element.

d. the name of the first attribute of the first coin's `name` element. (The expression `@*` selects the attributes of an element.)

e. the value of the `lang` attribute of the second coin's `name` element.

■■■ **R23.12** Harry Hopeless doesn't want to build a DOM tree to produce an XML document. Instead, he uses the following code:

```
System.out.println("<?xml version="1.0"?><items>");
for (LineItem anItem: items)
{
   Product p = anItem.getProduct();
   System.out.println("<item><product><description>" + p.getDescription()
      + "</description><price>" + p.getPrice()
      + "</price></product><quantity>" + anItem.getQuantity()
      + "<quantity></item>");
}
System.out.println("</items>");
```

What can go wrong? How can one fix the problems?

■■ **R23.13** Design a DTD that describes a bank with bank accounts.

■■ **R23.14** Design a DTD that describes a library patron who has checked out a set of books. Each book has an ID number, an author, and a title. The patron has a name and telephone number.

■■ **R23.15** Write the DTD file for the following XML document

```
<?xml version="1.0"?>
<productlist>
   <product>
      <name>Comtrade Tornado</name>
```

```
      <price currency="USD">2495</price>
      <score>60</score>
   </product>
   <product>
      <name>AMAX Powerstation 75</name>
      <price>2999</price>
      <score>62</score>
   </product>
</productlist>
```

■■ **R23.16** Design a DTD for invoices, as described in How To 23.3.

■■■ **R23.17** Design a DTD for simple English sentences, as described in Special Topic 23.1.

■■■ **R23.18** Design a DTD for arithmetic expressions, as described in Special Topic 23.1.

PROGRAMMING EXERCISES

■■ **P23.1** Write a program that can read XML files, such as

```
<purse>
   <coin>
      <value>0.5</value>
      <name>half dollar</name>
   </coin>
   . . .
</purse>
```

Your program should construct a `Purse` object and print the total value of the coins in the purse.

■■■ **P23.2** Building on Exercise P23.1, make the program read an XML file as described in that exercise. Then print an XML file of the form

```
<purse>
   <coins>
      <coin>
         <value>0.5</value>
         <name>half dollar</name>
      </coin>
      <quantity>3</quantity>
   </coins>
   <coins>
      <coin>
         <value>0.25</value>
         <name>quarter</name>
      </coin>
      <quantity>2</quantity>
   </coins>
</purse>
```

■■ **P23.3** Repeat Exercise P23.1, using a DTD for validation.

■■ **P23.4** Write a program that can read XML files, such as

```
<bank>
   <account>
      <number>3</number>
      <balance>1295.32</balance>
```

```
   </account>
   . . .
</bank>
```

Your program should construct a `Bank` object and print the total value of the balances in the accounts.

■■ **P23.5** Repeat Exercise P23.4, using a DTD for validation.

■■ **P23.6** Enhance Exercise P23.4 as follows: First read the XML file in, then add ten percent interest to all accounts, and write an XML file that contains the increased account balances.

■■■ **P23.7** Write a DTD file that describes documents that contain information about countries: name of the country, its population, and its area. Create an XML file that has five different countries. The DTD and XML should be in different files. Write a program that uses the XML file you wrote and prints:

- The country with the largest area.
- The country with the largest population.
- The country with the largest population density (people per square kilometer).

■■ **P23.8** Write a parser to parse invoices using the invoice structure described in How To 23.1. The parser should parse the XML file into an `Invoice` object and print out the invoice in the format used in Chapter 11.

■■ **P23.9** Modify Exercise P23.8 to support separate shipping and billing addresses. Supply a modified DTD with your solution.

■■ **P23.10** Write a document builder that turns an invoice object, as defined in Chapter 11, into an XML file of the format described in How To 23.2.

■■■ **P23.11** Modify Exercise P23.10 to support separate shipping and billing addresses.

■ **Graphics P23.12** Write a program that can read an XML document of the form

```
<rectangle>
   <x>5</x>
   <y>10</y>
   <width>20</width>
   <height>30</height>
</rectangle>
```

and draw the shape in a window.

■ **Graphics P23.13** Write a program that can read an XML document of the form

```
<ellipse>
   <x>5</x>
   <y>10</y>
   <width>20</width>
   <height>30</height>
</ellipse>
```

and draw the shape in a window.

■■ **Graphics P23.14** Write a program that can read an XML document of the form

```
<rectangularshape shape="ellipse">
   <x>5</x>
   <y>10</y>
```

```
   <width>20</width>
   <height>30</height>
</rectangularshape>
```

Support shape attributes `"rectangle"`, `"roundrectangle"`, and `"ellipse"`.

Draw the shape in a window.

■■ **Graphics P23.15** Write a program that can read an XML document of the form

```
<polygon>
   <point>
      <x>5</x>
      <y>10</y>
   </point>
   . . .
</polygon>
```

and draw the shape in a window.

■■■ **Graphics P23.16** Write a program that can read an XML document of the form

```
<drawing>
   <rectangle>
      <x>5</x>
      <y>10</y>
      <width>20</width>
      <height>30</height>
   </rectangle>
   <line>
      <x1>5</x1>
      <y1>10</y1>
      <x2>25</x2>
      <y2>40</y2>
   </line>
   <message>
      <text>Hello, World!</text>
      <x>20</x>
      <y>30</y>
   </message>
</drawing>
```

and show the drawing in a window.

■■■ **Graphics P23.17** Repeat Exercise P23.16, using a DTD for validation.

■■■ **P23.18** Following Exercise P11.8, design an XML format for the appointments in an appointment calendar. Write a program that first reads in a file with appointments, then another file of the format

```
<commands>
   <add>
      <appointment>
         . . .
      </appointment>
   </add>
   . . .
   <remove>
      <appointment>
         . . .
      </appointment>
   </remove>
</commands>
```

Your program should process the commands and then produce an XML file that consists of the updated appointments.

ANSWERS TO SELF-CHECK QUESTIONS

1. Your answer should look similar to this:

```
<student>
   <name>James Bond</name>
   <id>007</id>
</student>
```

2. Most browsers display a tree structure that indicates the nesting of the tags. Some browsers display nothing at all because they can't find any HTML tags.
3. The text `hamster.jpg` is never displayed, so it should not be a part of the document. Instead, the `src` attribute tells the browser where to find the image that should be displayed.
4. 29.95.
5. `name(/*[1])`.
6. The `createTextElement` method is useful for creating other documents.
7. First construct a string, as described, and then use a `PrintWriter` to save the string to a file.
8. `<!ELEMENT item (product, quantity?)>`
9. `<!ELEMENT product ((description, price) | (price, description))>`
10. `<!ATTLIST description language CDATA #IMPLIED>`