C h a p t e r 29

XML

▶ To understand XML elements and attributes

▶ To understand the concept of an XML parser

▶ To write programs that load and save XML documents

▶ To design Document Type Definitions for XML documents

Y ou saw in Chapter 28 how to store data in a database. Databases are excellent for storing large quantities of of data and for retrieving the data quickly. In this chapter we are concerned with a different data storage issue, namely, how to transmit data from one program to another in a structured form. You will learn about the Extensible Markup Language (XML), a robust mechanism for encoding and decoding data.

XML allows you to encode complex data in a form that is easy to decode. XML is becoming very popular for data exchange. It is simple enough that a wide variety of programs can easily generate XML data. Because the XML format is standardized, it is easy to read and write XML documents in C++. Libraries for parsing the data are widely available and—as you will see in this chapter—easy to use for a programmer.

2   **CHAPTER 29**   XML

CHAPTER CONTENTS

29.1   XML Tags and Documents

29.1.1   Advantages of XML

Let's look at a typical example for encoding data. Suppose we want to save employee data in a file. Here is a naïve encoding of employee names and salaries:

```
Mary Miller 64500
Jim J. Jones Jr 42000
```

It is easy enough to write this data. Now consider the effort that is required to read it back. You read the name and then the salary. But the name contains a variable number of words. It is possible to overcome this issue, but not without some pain.

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

```
<employee>
    <name>Mary Miller</name>
    <salary>64500</salary>
</employee>
<employee>
    <name>Jim J. Jones Jr</name>
    <salary>42000</salary>
</employee>
```

As you will see in this chapter, you can process input of this form by using an XML library. The library code automatically handles the element names enclosed in `< >` brackets.

One 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 what a "salary" is. As a programmer, you still need to write code to extract the salary as the contents of the `salary` element. Nevertheless, the fact that an XML document is comprehensible by humans is a huge advantage for program development.

A second advantage of the XML version is that it is *resilient to change*. Suppose the employee data change, and an additional field is introduced to denote the year in which the employee was hired. In the naïve format, the year might be added at the end:

```
Mary Miller 64500 1982
```

Now, a program that can process the old format might choke when reading a sequence of employees in the new format. Thus, the program needs to be updated to work with the new data format. Then the updated program may not read the old data format any longer. (Of course, one can overcome these issues with some amount of ingenuity, but 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:

```
<employee>
    <name>Mary Miller</name>
    <salary>64500</salary>
    <year>1982</year>
</employee>
```

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

Quality Tip   29.1

XML Is Stricter Than HTML

You may have noticed that the XML format of the employee data looked somewhat like HTML code that is used for authoring web pages. In particular, the XML tag pairs `<salary>` and `</salary>` look just like HTML tag pairs such as `<h1>` and `</h1>`. Both in XML and in HTML, tags are enclosed in angle brackets `< >`, and an opening tag is paired with a closing tag that starts with a slash (/) character.

However, in order to accommodate Web page authors of varying skill levels, most browsers are very lenient when interpreting HTML code. They will try to display pages with missing elements or wrongly nested elements.

Superficially, finding a way to display the element sounds "nicer" than refusing to display an invalid document, but in practice it is a major problem. A Web designer who checks a particular page in one or two browsers may think that the page is correct because these browsers happened to display it correctly, but another browser may display the invalid portions quite differently. It would have been better if the page had been diagnosed as incorrect.

The designers of XML learned from this experience and created a few simple rules that any document must fulfill.

- In XML, you *must* pay attention to the letter case of the tags; for example, `<img>` and `<IMG>` 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 `</h1>`. However, if a tag has no end tag, it must end in `/>`, for example

    ```
    <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"/>
    ```

Common Error   29.1

XML Describes Structure, Not Appearance

Because XML looks somewhat like HTML encoding for Web pages, 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. Some browsers make an effort to display XML documents, and there is a *transformation* mechanism that can transform XML into HTML that is suitable for display in a browser (see Advanced Topic 29.2). However, most data that are encoded in XML have nothing to do with browsers. For example, displaying an XML document with nothing but employee data (such as the ones in the previous section) in a browser would have no visual appeal. You will learn in this chapter how to write programs that analyze XML data. The fundamental difference between XML and HTML is that XML tells you what the data *mean*, not how to display them.

Random Fact   29.1

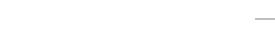### Word Processing and Typesetting Systems

You have almost certainly used a word processor for writing letters or reports. A word processor is a program used to write and edit documents made up of text and images. The text can contain characters in various fonts. It can be arranged in paragraphs, tables, and footnotes. Paragraphs can be formatted in various ways, such as ragged right (that is, the left ends of the lines of text are aligned under each other, but the right ends aren't), centered, and fully justified (that is, both the left and right ends of the lines are aligned). A useful characteristic of modern word processors is their what-you-see-is-what-you-get, or WYSIWYG, operation. You enter text and commands, using the keyboard and the mouse, and the computer screen instantly shows what the printed document will look like (see Figure 1).

However, there are disadvantages to the what-you-see-is-what-you-get nature of a word processor. You may labor to arrange various related images and tables to be together on the
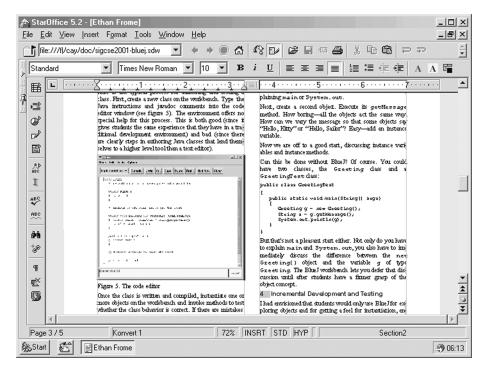
A What-You-See-Is-What-You-Get Word Processing Program

same page. Later, you find that you need to add a couple of paragraphs on the preceding page. Now half of the material moves to the next page, and you have do the arranging all over again. It would have been more useful if you could have told the word processor your intention, namely: "Always keep these images and tables together on the same page". In general, WYSIWYG programs allow you to arrange material, but they don't know *why* you arranged the material in a certain way. Thus, they can't keep the arrangement when your document changes. Some people call these programs "what you see is all you've got".

More fundamentally, WYSIWYG programs break down when you need to publish the same material in multiple ways. You may want to format product information as a product parts list and an advertising brochure. Or you may want to publish the information in printed form, on the Web, and in spoken form for telephone retrieval. Now you no longer want to "get" a single result, so it isn't as helpful to see what you get. Instead, it becomes much more important to visualize the *structure* of the information.

A program for editing structured text needs to capture three pieces of information:

- The text itself
- The structural element (paragraph, bulleted list, heading, and so on) to which each part of the text belongs
- The rules for formatting the structural elements

To make it easy to interchange structured documents between computer systems, the structural information is often encoded in *markup tags*. For example, XML and HTML use tags that are enclosed in angle brackets such as the familiar `<p>`, `<ul>`, and `<h1>` tags.

In the 1970s, when publishers began to move away from traditional manual typesetting to computer-based typesetting, the result at first was much inferior quality, particularly for mathematical formulas. Arranging the symbols in complex formulas in a way that makes mathematical sense is an art that requires practice and good judgment, and the first computer-based typesetting programs were definitely not up to the job. Frustrated by this situation, the famous computer scientist Donald Knuth of Stanford University decided to do something about it and invented a typesetting program that he called $\TeX$ (pronounced "tek" because the "X" is the capital Greek letter chi). Input to this program consists of text with markup tags that start with a backslash; curly braces {} for grouping; and other special markup symbols, such as _ and ∧ to indicate subscript and superscript. For example, to specify a summation, you type

```
\sum_{i=1}^n i^2
```

The $\TeX$ program typesets the summation as shown in Figure 2. Note that the expression is formatted one way when it occurs in a line of text and another way when it appears as part of a displayed formula.

Only the most hardened HTML or $\TeX$ authors produce markup such as `<h1>` or `\sum` by hand. For HTML in particular, many programs are available that display the structure of an HTML document and allow authors to edit both text and structure in a convenient way, combining the benefits of visual feedback and structure editing.

A sum in a line of text: $\sum_{i=1}^{n} i^2$.

Figure 2

A Formula Typeset in the $\TeX$
Typesetting System

The same sum as a displayed formula:

$$\sum_{i=1}^{n} i^2$$

29.1.2 — The Structure of an XML Document

In this section, you will see the rules for properly formatting XML data. For historical reasons, an XML data set is called a *document*, even though most XML data have little, if anything, to do with traditional documents.

The XML standard recommends that every XML document start out with a header

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

(Note that the header is delimited with `<?` and `?>`. The header is not considered an XML element.)

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

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

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

> *<elementTag optional attributes> content </elementTag>*

or

> *<elementTag optional attributes/>*

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 subelement
3. More text: " data formats."

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 employee data—it is better to stick with elements that contain *either* other elements or text. The XML specification calls content that consists only of elements *element content*. An example is the employee element, which contains only name and salary elements.

Each element can also contain *attributes*. For example, the img element of HTML has an src attribute that specifies the image location:

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

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

Programmers often wonder whether it is better to use attributes or subelements. For example, should an employee be described as

```
<employee name="Mary Miller" salary="64500"/>
```

or

```
<employee>
    <name>Mary Miller</name>
    <salary>64500</salary>
</employee>
```

The first example is shorter. However, it violates the spirit of attributes. Attributes are intended to provide information *about* the element content. For example, the salary element might have an attribute currency that helps interpret the element content. The content 64500 has a different interpretation in the element

```
<salary currency="USD">64500</salary>
```

than it does in the element

```
<salary currency="EUR">64500</salary>
```

An element can have multiple attributes, for example

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

Elements and attributes are the components of an XML document that are required for encoding data. There are other XML constructs for more specialized situations—see [1] for more information. In the next section, you will see how to use C++ to parse XML documents.

29.2

Quality Tip

Prefer XML Elements over Attributes

Attributes are shorter than elements. For example,

```
<employee name="Mary Miller" salary="64500"/>
```

seems simpler than

```
<employee>
    <name>Mary Miller</name>
    <salary>64500</salary>
</employee>
```

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:

```
<salary currency="USD">64500</salary>
```

or even

```
<salary>
    <currency>USD</currency>
    <amount>64500</amount>
</salary>
```

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

```
<employee name="Mary Miller" salary="USD 64500"/>
```

But then your program must manually take apart the string USD 64500. That's the kind of tedious and error-prone coding that XML is designed to avoid.

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

```
<a href="http://www.wiley.com">John Wiley & Sons</a>
```

The text inside the a element, John Wiley & Sons, is part of what the user sees on the Web page, but the href attribute string http://www.wiley.com is not displayed on the page.

Of course, HTML is a little different from the XML coding used in documents constructed to describe data such as employee 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 if an item is part of the data or tells something about the data, make the item an element, not an attribute.

▼ ## Quality Tip 29.3

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:

```
<salary>
    <currency>USD</currency>
    64500
</salary>
```

Instead, the children of an element should be either text

```
<salary>64500</salary>
```

or elements

```
<salary>
    <currency>USD</currency>
    <amount>64500</amount>
</salary>
```

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 text.

▼ ## Random Fact 29.2

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
```
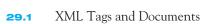
Here the symbol ::= means "can be replaced with". For example, `<article>` can be replaced with a or the.

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
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

Grammar symbols, such as `<noun>`, happen to be enclosed in angle brackets like XML tags, but they are different from tags. One purpose of a grammar is to produce strings that are *legal* 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, as shown in the table at left.

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. One way to show the derivation is to construct a parse tree (see Figure 3).

A *parser* is a program that reads strings, checks whether the input conforms to the rules of a certain grammar, and then builds a tree structure or carries out other functions as it processes the input.

The most important use for parsers is inside compilers for programming languages. Just as our grammar can describe (some) simple English language sentences, the legal "sentences" in a programming language can be described by a grammar. The actual grammar for the C++ programming language occupies 22 pages in the book *The C++ Programming Language* [3].
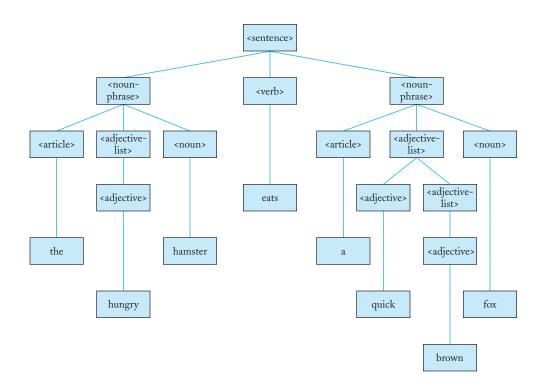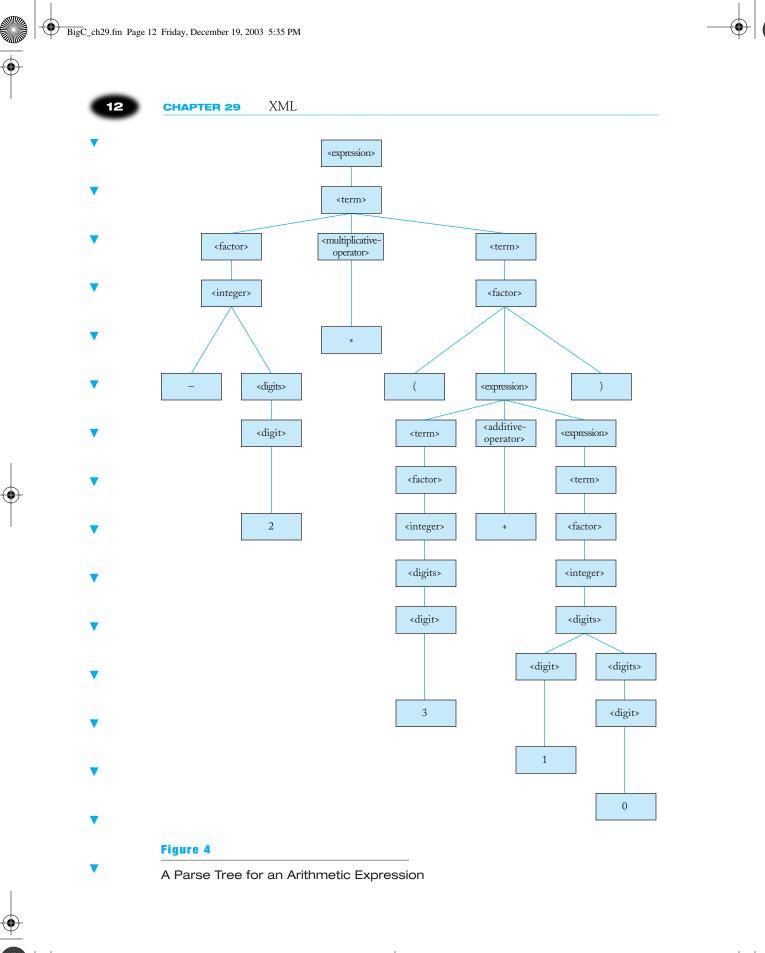

Figure 3

A Parse Tree for a Simple Sentence

Figure 4

A Parse Tree for an Arithmetic Expression

▼

To give a flavor of a small subset of such a grammar, here is a grammar that describes arithmetic expressions.

▼

```
<expression> ::=
    <term> | <term> <additive-operator> <expression>
<additive-operator> ::= + | -
<term> ::=
    <factor> | <factor> <multiplicative-operator> <term>
<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 legal expression in this grammar is

▼

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

Figure 4 shows the parse tree for this expression.

In a compiler, parsing the program source is the first step towards generating code that the target processor 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 tree.

29.2   Parsing XML Documents

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. XML parsers are available in C++, Java, and many other programming languages.

Two kinds of XML parsers are in common use. One of them follows a specification called SAX (Simple Access to XML), and the other follows a specification called DOM (Document Object Model). A SAX parser is *event-driven*. Whenever the parser encounters a particular construct (for example, an opening tag such as `<value>`), then it calls a function that you must provide. In contrast, a DOM parser builds up a tree that represents the parsed document. Once the parser is done, you can analyze the tree. SAX parsers have one advantage—they are more efficient for handling large XML documents because they don't have to build up a tree. DOM parsers, however, are easier to use for most applications—the parse tree gives you a complete overview of the data, whereas a SAX parser gives you the information in bits and pieces. In this section, you will learn how to use a DOM parser.

To run the programs in this chapter, you need to download and install the Xerces library that is distributed by the Apache organization. (This organization developed the famous Apache Web server that powers the majority of Web sites on the Internet.) Directions for installing and using the Xerces library differ between operating systems and compilers. You will find instructions on the Apache Web site [2] as well as the companion Web site for this book.

The Xerces library contains classes that process XML input, as well as classes that represent the tree structure of an XML document. In order to read a specific kind of XML document, you need to write a C++ program that translates the XML input into C++ objects. You use the Xerces library to parse the input. Then you examine the resulting tree structure and translate it into objects of classes that are meaningful to your application. Thus, you need to write a separate parsing program for each type of XML file. The parser takes care of the task that is common to all those programs: transforming the input into a tree structure. Your job is to translate the generic tree nodes into objects.

To read an XML document from a file, you need a `DOMBuilder` object. You get such an object through the following calls:

```
DOMImplementation* implementation
   = DOMImplementation::getImplementation();
DOMBuilder* parser = implementation->createDOMBuilder(
   DOMImplementationLS::MODE_SYNCHRONOUS, NULL);
```

The first call retrieves the default `DOMImplementation` object that is a factory for classes for reading and writing XML documents. The second call asks the implementation object to create a builder object.

Note that all DOM-related classes start with the prefix `DOM`. Also note that you generally need to manipulate pointers to DOM objects. For example, the `createDOMBuilder` function returns a pointer to the builder object.
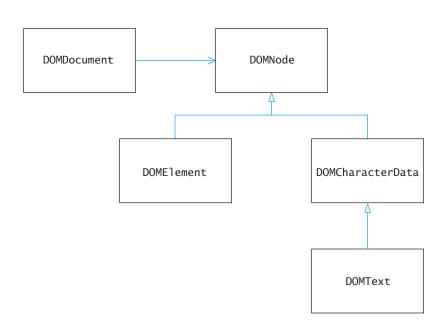

Figure 5

UML Diagram of DOM Classes for Document Trees

You never call `delete` on these pointers. Instead, you invoke the `release` function on the builder object when you are done with it. This function recycles the memory occupied by the builder object itself as well as the memory used by the documents that were built.

Once you have a builder object, you can use it to parse a document from a file.

```
DOMDocument* doc = parser->parseURI("items.xml");
```

The `DOMDocument` describes the tree structure of an XML document. Once the parser has processed the document, you can inspect and modify it. Start with the root element. The `getDocumentElement` function of the `DOMDocument` class returns the root element.

```
DOMNode* root = doc.getDocumentElement();
```

The `getDocumentElement` function actually returns a `DOMElement` object. The `DOMElement` class is a derived class of the more general `DOMNode` class. In addition to the `DOMElement` class there are several other node types that are derived classes of the `DOMNode` class. In this chapter, the only other derived class of interest to us is the `DOMText` class (see Figure 5).

If an element has only one or two children, you can use the `getFirstChild` and `getLastChild` function to retrieve just one child. These functions return `NULL` if the requested node doesn't exist.

Most elements have more than one or two children so it is necessary to use the `getChildNodes` function of the `DOMNode` class to get the child elements of an element. The child elements are placed in a `DOMNodeList` object. For example, consider the item list in Figure 6. You can consider the document as a tree whose root is the `items` element (see Figure 7). Calling the `getChildNodes` function on the root element yields a node list containing two `item` elements.

```
<?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>
```
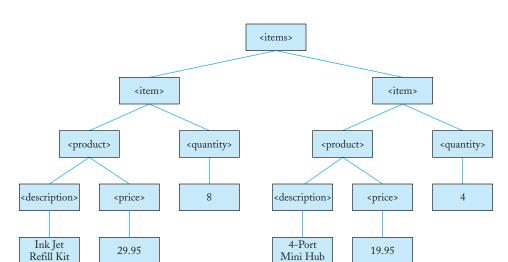
Figure 6

An XML Document

Figure 7

The Tree View of the Document

Sadly, a `DOMNodeList` is not the same as a `list<DOMNode*>`. To use a `DOMNodeList`, you have to learn yet another set of functions. Use the `getLength` function to obtain the number of nodes in the list. Then call the `item` function to get an item in the node list. Here is how you access the child element with index i of the root.
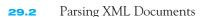
```
DOMNodeList* nodes = root->getChildNodes();
int i = . . .; // A value between 0 and nodes->getLength() - 1
DOMNode* child_node = nodes->item(i);
```

By default, the XML parser *keeps all white space* (that is, spaces, tabs, and newline characters) between elements, just in case you want to analyze it. As a result, the parser creates child nodes for that white space as well as for elements. When your XML document describes data records, you never care about that white space. If you have made sure not to mix meaningful text with elements, as described in Quality Tip 29.3, ignoring the white space is easy: If you have an element whose children are also elements (such as the item element in our example), just skip all child nodes that aren't element nodes. Use the following loop:

```
for (int i = 0; i < nodes->getLength(); i++)
{
   DOMNode* child_node = nodes->item(i);
   DOMElement* child_element
      = dynamic_cast<DOMElement*>(child_node);
   if (child_element != NULL)
   {
      // Do something with child_element
      . . .
   }
}
```

Recall that the `dynamic_cast` expression returns `NULL` if the `child_node` pointer does not point to a `DOMElement` object.

In Section 29.5, you will learn how to instruct the parser to ignore this white space. To do so, you will have to provide a document type definition that specifies the structure of your XML document. Right now, it is simpler to include a test to skip white space.

Once you have an element, you can get the element name (such as `price`) by calling the `getTagName` function.

```
DOMElement* price_element = . . .;
XMLCh* name = price_element->getTagName();
    // Returns a tag name, such as price
```

However, you get the name as an array of `XMLCh` values, not as a C++ `string` object. The designers of the Xerces library chose not to use the C++ `string` class because XML documents may contain arbitrary Unicode characters. As explained in Random Fact 9.2, Unicode is an encoding scheme for tens of thousands of characters that are in use around the world. But a C++ `string` can only hold single-byte characters, typically from the ASCII character set that encodes only 128 symbols. There is a `wstring` class in the standard C++ library that can be used to represent Unicode strings, but the Xerces designers didn't use it.

In our examples, the ASCII character set will suffice, and we will simply use the following function to convert `XMLCh` arrays to `string` objects.

```
string XMLCh_to_string(const XMLCh* in)
{
    char* s = XMLString::transcode(in);
    string r(s);
    XMLString::release(&s);
    return r;
}
```

The `XMLString::transcode` function converts an `XMLCh` array into an array of `char` values. The next line constructs a `string` object from that array. Finally, we recycle the memory for the `char` array by calling the `XMLString::release` function.

It may seem strange that one has to pay so much attention to string implementations and memory management when using a third-party class library. However, this is a common phenomenon with C++. Many C++ class libraries do not use the standard C++ classes, and they often have their own schemes for memory management. There are two reasons. Not all C++ compilers have complete support for the C++ standard. Library designers often cater to the lowest common denominator. Moreover, even though memory management can be automated through destructors and copy constructors, these mechanisms can be inefficient. Library designers often feel that safety is less important than speed. Generally, it is a good idea to localize library idiosyncrasies in helper functions, as we have done with the `XMLCh_to_string` function.

You just saw how to retrieve the name of an element. You can also look up attributes by name, with the `getAttribute` function. However, now we have the opposite problem: we need to supply the attribute name as an `XMLCh` array. We cannot simply write a helper function for the conversion from `string` objects to `XMLCh` arrays, because the `XMLCh` arrays need to be deleted after they have been used. Instead, we will wrap the `getAttribute` function into a helper function:

```
string get_attribute(DOMElement* element, string name)
{
    XMLCh* xname = new XMLCh[name.length() + 1];
    XMLString::transcode(name.c_str(), xname, name.length());
    XMLCh* xvalue = element->getAttribute(xname);
    delete xname;
    return XMLCh_to_string(xvalue);
}
```

For example, here is how you look up the currency attribute:

```
string attribute_value = get_attribute(price_element,
    "currency");
```

If you don't know an attribute's name, you can iterate through all attributes. For that purpose you use a DOMNamedNodeMap, which, like the DOMNodeList class, is a data structure that is proprietary to the DOM library. The DOMNamedNodeMap is conceptually similar to the standard C++ map template that you saw in Chapter 23, but the usage details are different. The getAttributes function returns a map of all pairs of attribute names and values:

```
DOMNamedNodeMap* attributes = price_element->getAttributes();
```

The getLength function tells you how many attributes this element has. To get a particular attribute, call the item function, just as you would for a DOMNodeList. Then call getNodeName and getNodeValue on the returned DOMNode object:

```
int i = . . .;
// A value between 0 and attributes->getLength() - 1
DOMNode* attribute_node = nodes->item(i);
XMLCh* attribute_name = attribute_node->getNodeName();
// For example, "currency"
XMLCh* attribute_value = attribute_node.getNodeValue();
// For example, "USD"
```

Now you know how to analyze elements and their attributes, and how to find their children if they themselves are elements. However, some elements have children that contain text. In our example, those are the description, price, and quantity elements. The document builder creates nodes of type DOMText for these children. If you follow the recommendation of Quality Tip 29.2, then any elements containing text have a *single* DOMText child node, which you can retrieve with the getFirstChild function. To read the text in such a node, use the getData function. This function returns a string, which you may need to parse to convert it to a number. For example, here is how you determine the price that is stored in a price element:

```
DOMNode* price_element = . . .;
DOMText* price_data = dynamic_cast<DOMText*>(
    price_element->getFirstChild());
string price_string = XMLCh_to_string(price_data->getData());
    // For example, "24.95"
double price = string_to_double(price_string);
    // For example, 24.95
```

Here, string_to_double is the helper function from Chapter 12 that converts a string to a floating-point number.

Now you know how to read and analyze an XML document. The example program at the end of this section puts these techniques to work. This program parses an XML file that describes a list of product items, and produces a vector of `Item` objects. The program uses several helper functions

```
Product get_product(DOMNode* e)
Item get_item(DOMNode* e)
vector<item> get_items(DOMNode* e)
```

However, there are no helper functions for parsing the `description`, `price`, and `quantity` elements. Parsing these elements is so simple that it is done inside the `get_product` and `get_item` functions. In general, it is a good idea to make one helper function for each XML element that contains subelements.

Finally, note that the `main` function contains a call to the `XMLPlatformUtils::` `Initialize` function. This initialization is required for the proper use of the Xerces library.

This program uses the `Item` and `Product` classes from Chapter 13.

File parser.cpp

```
 1 #include <string>
 2 #include <sstream>
 3 #include <vector>
 4 #include <xercesc/dom/DOM.hpp>
 5 #include <xercesc/util/XMLString.hpp>
 6 #include <xercesc/util/PlatformUtils.hpp>
 7
 8 #include "item.h"
 9
10 using namespace std;
11 using namespace xercesc;
12
13 /*
14    Converts a sequence of XMLCh characters to a string.
15    @param in the sequence of XMLCh characters
16    @return the transcoded string
17 */
18 string XMLCh_to_string(const XMLCh* in)
19 {
20    char* s = XMLString::transcode(in);
21    string r(s);
22    XMLString::release(&s);
23    return r;
24 }
25
26 /**
27    Converts a string to a floating-point value, e.g., "3.14" -> 3.14.
28    @param s a string representing a floating-point value
29    @return the equivalent floating-point value
30 */
31 double string_to_double(string s)
32 {
33    istringstream instr(s);
34    double x;
35    instr >> x;
```

```
36      return x;
37  }
38
39  /**
40      Converts a string to an integer, e.g., "314" -> 314.
41      @param s a string representing an integer
42      @return the equivalent integer
43  */
44  int string_to_int(string s)
45  {
46      istringstream instr(s);
47      int x;
48      instr >> x;
49      return x;
50  }
51
52  /**
53      Obtains a product from a DOM node.
54      @param e a <product> element
55      @return the product described by the given node
56  */
57  Product get_product(DOMNode* e)
58  {
59      DOMNodeList* children = e->getChildNodes();
60      string name = "";
61      double price = 0;
62      for (int i = 0; i < children->getLength(); i++)
63      {
64          DOMNode* child_node = children->item(i);
65          DOMElement* child_element
66              = dynamic_cast<DOMElement*>(child_node);
67          if (child_element != NULL)
68          {
69              string tagName = XMLCh_to_string(child_element->getTagName());
70              DOMText* textNode
71                  = dynamic_cast<DOMText*>(child_element->getFirstChild());
72
73              if (tagName == "description")
74                  name = XMLCh_to_string(textNode->getData());
75              else if (tagName == "price")
76              {
77                  string price_text = XMLCh_to_string(textNode->getData());
78                  price = string_to_double(price_text);
79              }
80          }
81      }
82      return Product(name, price);
83  }
84
85  /**
86      Obtains an item from a DOM node.
87      @param e an <item> element
88      @return the item described by the given node
89  */
90  Item get_item(DOMNode* e)
91  {
```

```
92      DOMNodeList* children = e->getChildNodes();
93      Product p;
94      int quantity = 0;
95      for (int i = 0; i < children->getLength(); i++)
96      {
97         DOMNode* child_node = children->item(i);
98         DOMElement* child_element
99            = dynamic_cast<DOMElement*>(child_node);
100        if (child_element != NULL)
101        {
102           string tagName = XMLCh_to_string(child_element->getTagName());
103           if (tagName == "product")
104              p = get_product(child_element);
105           else if (tagName == "quantity")
106           {
107              DOMText* textNode = dynamic_cast<DOMText*>(
108                 child_element->getFirstChild());
109              string quantity_text
110                 = XMLCh_to_string(textNode->getData());
111              quantity = string_to_int(quantity_text);
112           }
113        }
114     }
115     return Item(p, quantity);
116  }
117
118
119  /**
120     Obtains an array list of items from a DOM node.
121     @param e an <items> element
122     @return a vector of all <item> children of e
123  */
124  vector<Item> get_items(DOMNode* e)
125  {
126     vector<Item> items;
127
128     // Get the <item> children
129
130     DOMNodeList* children = e->getChildNodes();
131     for (int i = 0; i < children->getLength(); i++)
132     {
133        DOMNode* child_node = children->item(i);
134        DOMElement* child_element
135           = dynamic_cast<DOMElement*>(child_node);
136        if (child_element != NULL)
137        {
138           string tagName = XMLCh_to_string(child_element->getTagName());
139           if (tagName == "item")
140           {
141              Item c = get_item(child_element);
142              items.push_back(c);
143           }
144        }
145     }
146     return items;
147  }
```

```
148
149  int main()
150  {
151     XMLPlatformUtils::Initialize();
152
153     DOMImplementation* implementation
154        = DOMImplementation::getImplementation();
155     DOMBuilder* parser = implementation->createDOMBuilder(
156        DOMImplementationLS::MODE_SYNCHRONOUS, NULL);
157     DOMDocument* doc = parser->parseURI("items.xml");
158
159     DOMNode* root = doc->getDocumentElement();
160     vector<Item> items = get_items(root);
161
162     for (int i = 0; i < items.size(); i++)
163        items[i].print();
164     parser->release();
165
166     return 0;
167  }
```

Productivity Hint   29.1

Helper Functions in an XML Parser

Suppose you need to write a parser that reads an XML document of a particular structure. Consider all elements with element content (that is, elements that contain subelements). For each such element, ask yourself what *class* it should belong to. This may be an existing class; a class that you need to write; or, if an element simply contains a sequence of child elements of the same type, a vector. Then provide a helper function of this form:

```
ClassForElement getElementName(DOMNode* e)
{
    DOMNodeList* children = e->getChildNodes();
    for (int i = 0; i < children->getLength(); i++)
    {
        DOMNode* child_node = children->item(i);
           DOMElement* child_element
           = dynamic_cast<DOMElement*>(child_node);
        if (child_element != NULL)
        {
            Get value of child element
        }
    }
    Use the child element values to construct and
    return a ClassForElement object
}
```

There are two cases to consider when looking at the child elements. If the child element doesn't contain text, then call its get*ChildElementName* function. If the child element contains text, then read the element data:

```
▼                      DOMText* text_node = dynamic_cast<DOMText*>(
                           child_element->getFirstChild());
                       string data = XMLCh_to_string(text_node->getData());
```

▼   If necessary, convert that string to a number, using the `string_to_int` or `string_to_double` functions of Chapter 12.

      You may find it helpful to implement these helper functions from the bottom up, starting
▼   with the simplest function (such as `get_product`) and finishing with the function for the root element (`get_items`).

▼   ⊗ **Common Error**   29.2
                                             ▶

XML Elements Describe Data Fields, Not Classes

▼   Productivity Hint 29.1 explains how to convert XML elements to C++ objects. You need to determine a class for each element type. This can be confusing. For example, consider an XML file for an invoice, with separate shipping and billing addresses:

```
▼       <invoice>
           <number>11365</number>
           <shipto>
▼             <name>John Meyers</name>
              <company>ACME Computer Supplies Inc.</company>
              <street>1195 W. Fairfield Rd.</street>
              <city>Sunnyvale</city>
▼             <state>CA</state>
              <zip>94085</zip>
           </shipto>
▼          <billto>
              <name>Accounts Payable</name>
              <company>ACME Computer Supplies Inc.</company>
              <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? No, this 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 equivalent of a data field and then determine an appropriate class. For example, an invoice object has data fields

▼   ● `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 Advanced Topic 29.1 for more information.

Quality Tip   29.4

Stand on the Shoulders of Others

When looking at the parser program, you may find that using the Xerces library makes a simple task rather complex. Indeed, the Xerces library seems an inelegant creation. You may feel that you can do better by writing your own XML library. However, that sentiment has lead many programmers astray.

It is an easy matter for a competent programmer to implement a simple parser that takes care of 90 percent of the XML parsing process. However, the devil is in the details of the remaining 10 percent. XML files can contain Unicode characters or characters from international encoding schemes whose handling requires special expertise. The XML specification [1] has tedious rules for white space, text inside quotation marks, and so on. Is it really worth your time to learn about these details when your primary interest is to solve a business problem?

Here is an excerpt from a speech by the computer scientist Richard Hamming. "Newton said, 'If I have seen further than others, it is because I've stood on the shoulders of giants.' These days we stand on each other's feet! You should do your job in such a fashion that others can build on top of it, so they will indeed say, 'Yes, I've stood on so and so's shoulders and I saw further.' The essence of science is cumulative."

29.3   Creating XML Documents

In the preceding section, you saw how to read an XML document into a `DOMDocument` object and how to analyze the contents of that object. In this section, you will see how to do the opposite—build up a `DOMDocument` object and then save it as an XML document. Of course, you can also generate an XML document simply as a sequence of print statements. However, this is not a good idea—it is easy to build an illegal XML document in this way, as when strings contain special characters such as `<` or `&`. (These characters need to be written as "entities" `<` and `"` in most contexts—see the XML specification [1] for details).

Recall that you needed a `DOMImplementation` object to create an XML parser. You also need such an object to create a new, empty document. Thus, to create a new document, first get the implementation object and then create the empty document:

```
DOMImplementation* implementation
   = DOMImplementation::getImplementation();
DOMDocument* doc = implementation->createDocument();
doc->setStandalone(true);
// An empty document
```

We set the "standalone" flag of the document to indicate that the document does not have a document type definition. You will see in the next section how to supply such a definition that constrains the document structure.

Now you are ready to insert nodes into the document. The document object is a factory for nodes. You use the `createElement` and `createTextNode` functions of the `DOMDocument` class to create the element and text nodes that you need. Because the

element names and text node data need to be given as an XMLCh array, we wrap these functions into helper functions that have string parameters. Here is the wrapper function for createElement:

```
DOMElement* create_element(DOMDocument* doc, string name)
{
    XMLCh* xname = new XMLCh[name.length() + 1];
    XMLString::transcode(name.c_str(), xname, name.length());
    DOMElement* r = doc->createElement(xname);
    delete xname;
    return r;
}
```

As you can see, the function simply converts the string parameter to an XMLCh array, creates a DOMElement, and then deletes the array. The createElement function makes a copy of the array content that is owned by the document.

To create an element containing a text node, we first convert the text data to an XMLCh array. We then call the createTextNode function to construct a text node, and the create_element helper function to create an element node. Then the text node is set as the child of the element node. Here is the code:

```
DOMElement* create_text_element(DOMDocument* doc,
    string name, string text)
{
    XMLCh* xtext = new XMLCh[text.length() + 1];
    XMLString::transcode(text.c_str(), xtext, text.length());
    DOMText* textNode = doc->createTextNode(xtext);
    delete xtext;
    DOMElement* r = create_element(doc, name);
    r->appendChild(textNode);

    return r;
}
```

You can also set element attributes with the setAttribute function:

```
XMLCh xname = . . .;
XMLCh xvalue = . . .;
price_element->setAttribute(xname, xvalue);
```

To construct the tree structure of a document, start with the root, then add children with the appendChild function. For example, here we build an XML tree that describes an item (see Figure 8).

```
// Create elements
DOMElement* itemElement = create_element(doc, "item");
DOMElement* product_element = create_element(doc, "product");
DOMElement* description_element = create_text_element(doc,
    "description", "Ink Jet Refill Kit");
DOMElement* price_element = create_text_element(doc,
    "price", "29.95");
DOMElement* quantity_element
    = create_text_element(doc, "quantity", "8");

// Add elements to the document
doc->appendChild(item_element);
item_element->appendChild(product_element);
```

```
item_element->appendChild(quantity_element);
product_element->appendChild(description_element);
product_element->appendChild(price_element);
```

However, it is a good idea to organize these steps into a set of helper functions, one per element. For example, the sample program at the end of this section contains helper functions

```
DOMElement* create_product(DOMDocument* doc, const Product& p)
DOMElement* create_item(DOMDocument* doc, const Item& anItem)
DOMElement* create_item_list(DOMDocument* doc,
    const vector<Item>& items)
```

Note that these functions are the reverse of the helper functions of the program of the preceding section. The parser helper functions turned XML elements into C++ objects. These helper functions turn C++ objects into XML elements.

In most practical applications, you construct a DOM tree in order to write an XML document that is saved to disk or transmitted to another program through a network connection. To write the tree, obtain a `DOMWriter` from the `DOMImplementation` object:

```
DOMWriter* writer = implementation->createDOMWriter();
writer->setFeature(XMLUni::fgDOMWRTFormatPrettyPrint, true);
```

Here we set the "pretty print" feature of the writer in order to align the elements. By default, the document would be written without any line breaks or indentations. This default is actually more suitable for transmitting a document to another program because it is free from unnecessary white space.

To write the document, you need a "format target". The following instructions simply write to the standard output stream:

```
XMLFormatTarget* out = new StdOutFormatTarget();
writer->writeNode(out, *doc);
```

To write to a local file, you use these instructions:

```
XMLCh* filename = . . .;
XMLFormatTarget* out = new LocalFileFormatTarget(filename);
writer->writeNode(out, *doc);
```

Unlike the other Xerces functions, the `writeNode` function requires a reference to the document, not a pointer. Therefore, we pass the expression `*doc`.
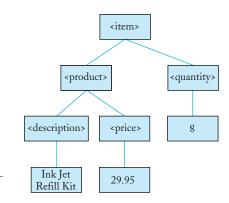
Figure 8

An XML Tree That Describes an Item

Here is the complete program. This program uses the Product and Item classes from Chapter 13. The Item class has been modified by adding get_product and get_quantity functions.
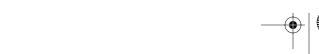
File builder.cpp

```
 1 #include <string>
 2 #include <sstream>
 3 #include <vector>
 4 #include <xercesc/dom/DOM.hpp>
 5 #include <xercesc/util/XMLString.hpp>
 6 #include <xercesc/util/PlatformUtils.hpp>
 7 #include <xercesc/framework/StdOutFormatTarget.hpp>
 8
 9 #include "item.h"
10
11 using namespace std;
12 using namespace xercesc;
13
14 /**
15     Creates a DOM element with a given name.
16     @param doc  the document that creates the element
17     @param name  the element name
18     @return  an element with the given name
19 */
20 DOMElement* create_element(DOMDocument* doc, string name)
21 {
22     XMLCh* xname = new XMLCh[name.length() + 1];
23     XMLString::transcode(name.c_str(), xname, name.length());
24     DOMElement* r = doc->createElement(xname);
25     delete xname;
26     return r;
27 }
28
29 /**
30     Creates a DOM element containing a text node.
31     @param doc  the document that creates the element
32     @param name  the element name
33     @param text  the text for the text node
34     @return  an element with the given name and a child that is a
35     text node with the given text
36 */
37 DOMElement* create_text_element(DOMDocument* doc,
38     string name, string text)
39 {
40     XMLCh* xtext = new XMLCh[text.length() + 1];
41     XMLString::transcode(text.c_str(), xtext, text.length());
42     DOMText* textNode = doc->createTextNode(xtext);
43     delete xtext;
44
45     DOMElement* r = create_element(doc, name);
46     r->appendChild(textNode);
47
48     return r;
49 }
```

```
50
51 /**
52       Converts an floating-point value to a string, e.g., 3.14 -> "3.14".
53       @param n  a floating-point value
54       @return  the equivalent string
55 */
56 string double_to_string(double n)
57 {
58     ostringstream outstr;
59     outstr << n;
60     return outstr.str();
61 }
62
63 /**
64       Builds a DOM element for a product.
65       @param p  the product
66       @return  a DOM element describing the product
67 */
68 DOMElement* create_product(DOMDocument* doc, const Product& p)
69 {
70     DOMElement* description_element = create_text_element(doc,
71         "description", p.get_description());
72     DOMElement* price_element = create_text_element(doc,
73         "price", double_to_string(p.get_price()));
74
75     DOMElement* product_element = create_element(doc, "product");
76     product_element->appendChild(description_element);
77     product_element->appendChild(price_element);
78
79     return product_element;
80 }
81
82 /**
83       Builds a DOM element for an item.
84       @param anItem  the item
85       @return  a DOM element describing the item
86 */
87 DOMElement* create_item(DOMDocument* doc, const Item& anItem)
88 {
89     DOMElement* product_element
90         = create_product(doc, anItem.get_product());
91     DOMElement* quantity_element = create_text_element(doc,
92         "quantity", double_to_string(anItem.get_quantity()));
93
94     DOMElement* item_element = create_element(doc, "item");
95     item_element->appendChild(product_element);
96     item_element->appendChild(quantity_element);
97
98     return item_element;
99 }
100
101 /**
102       Builds a DOM element for an array list of items.
103       @param items  the items
104       @return  a DOM element describing the items
105 */
```

```
106  DOMElement* create_item_list(DOMDocument* doc,
107     const vector<Item>& items)
108  {
109     DOMElement* items_element = create_element(doc, "items");
110     for (int i = 0; i < items.size(); i++)
111     {
112        DOMElement* item_element = create_item(doc, items[i]);
113        items_element->appendChild(item_element);
114     }
115     return items_element;
116  }
117
118  int main()
119  {
120     XMLPlatformUtils::Initialize();
121
122     // Populate vector of items
123     vector<Item> items;
124     items.push_back(Item(Product("Toaster", 29.95), 3));
125     items.push_back(Item(Product("Hair dryer", 24.95), 1));
126
127     // Build the DOM document
128     DOMImplementation* implementation
129        = DOMImplementation::getImplementation();
130     DOMDocument* doc = implementation->createDocument();
131     doc->setStandalone(true);
132
133     DOMElement* root = create_item_list(doc, items);
134     doc->appendChild(root);
135
136     // Print the DOM document
137
138     DOMWriter* writer = implementation->createDOMWriter();
139     writer->setFeature(XMLUni::fgDOMWRTFormatPrettyPrint, true);
140     XMLFormatTarget* out = new StdOutFormatTarget();
141     writer->writeNode(out, *doc);
142
143     writer->release();
144     doc->release();
145
146     return 0;
147  }
```

Productivity Hint

29.2

Writing an XML Document

You could of course write an XML document as a series of print statements such as the following:

```
cout << "<product><description>" // Don't do this!
   << x.get_description()
   << "</description><price>"
      << x.get_price()
      << "</price>
```

But that's tedious. It is easy to miss a few tags. Moreover, you can get yourself into trouble if some of the strings contain characters such as < or ". For example, simply printing `x.get_description()` leads to an invalid XML file if the description happens to contain a < character.

To avoid these complexities, you should build a `DOMDocument` object and write it with a `DOMWriter`.

As in Productivity Hint 29.1, consider all elements with element content and find their matching C++ classes. For each element, produce a helper function:

```
DOMElement* create_ElementName(DOMDocument* doc,
   const ClassForElement& x)
{
   DOMElement* r = create_element(doc, "element name");
   DOMElement* child1 = . . .;
   r->appendChild(child1);
   DOMElement* child2 = . . .;
   r->appendChild(child2);
   . . .
   return r;
}
```

If the child element contains text, then use the `create_text_element` helper function:

```
DOMElement* child = create_text_element(doc,
   "child element name", "text value");
```

If the child element doesn't contain text, then query the object x for the data that is to become part of the child element and call its create*ChildElementName* function:

```
ClassForChildElement y = x.getChildData();
DOMElement* child = create_ChildElementName(doc, y);
```

As with the parser functions, you may find it helpful to implement these functions from the bottom up, starting with the simplest function (e.g., `create_product`) and finishing with the function for the root element (e.g., `create_items`).

29.4   Document Type Definitions

In the previous sections, you saw how to read and write XML documents. In this section you will learn how to produce Document Type Definitions (DTDs), which are sets of rules for correctly formed documents of a particular type.

For example, 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 describe

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

The legal 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. (The *
character is often used to denote "0 or more repetitions".)

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.

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

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

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

The definition for a product is similar:

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

Finally, here are the definitions of the three remaining node types:

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

The symbol #PCDATA refers to text, called *parsable 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 encoded if they occur in character data. Table 1 shows the encodings for special characters.

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

Table 1

Encodings for Special Characters

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's take a closer look at the descriptions of the allowed children. Table 2 shows the expressions that you can use to describe the children of an element. The EMPTY keyword 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 2 and Figure 9. Also see Productivity Hint 5.3 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. For example,

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

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)
(*E1*, *E2*, . . .)	Element *E1*, followed by *E2*, followed by . . .
(*E1*|*E2*|. . .)	Element *E1* or *E2* or . . .
(#PCDATA)	Text only
(#PCDATA|*E1*|*E2*|. . .)*	Any sequence of text and elements *E1*, *E2*, . . ., in any order
ANY	Any children allowed

Table 2

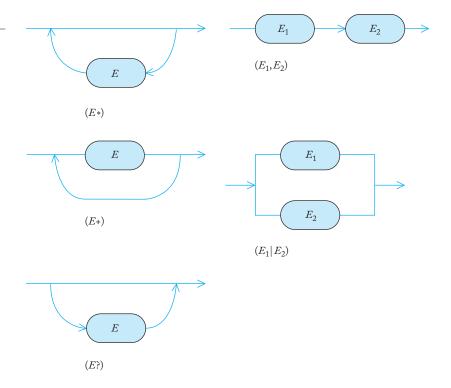Regular Expressions for Element Content

Figure 9

DTD Regular
Expression Operations

$(E*)$

(E_1, E_2)

$(E+)$

$(E_1 | E_2)$

$(E?)$

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 legal, but

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

is not—there is no starting title, and the title at the end doesn't follow an image. Fortunately, such complex rules are not very common.

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 `price` 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 lose control over the order in which the elements appear. As explained in Quality Tip 29.3, 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 a node to have children of any type—you should avoid that for DTDs that describe data sets.

You now know how to specify what children a node can 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 3. The `CDATA` type describes any sequence of character data. As with #PCDATA, certain characters such as < and & need to be encoded (<, &, 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>
```

Each of the choices must consist of letters, numbers, or the special characters

```
-, _, or .
```

There are other type descriptions that are less common in practice. You can find them in the XML reference [1].

The attribute type description is followed by a "default" declaration. The keywords that can appear in a default declaration are listed in Table 4.
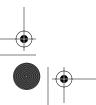
For example, this attribute declaration stipulates that each `price` element must have a `currency` attribute whose value is any character data:

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

Type Description	Attribute Type
CDATA	Any character data
$(N_1 \| N_2 \| . \quad . \quad .)$	One of N_1, N_2, ...

Table 3

Common Attribute Types

To fulfill this declaration, each `price` element must have a `currency` attribute, such as `<price currency="USD">`. It would be illegal to specify a price without a currency.

For an optional attribute, you use the `#IMPLIED` keyword 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 verify that a document conforms to its DTD.

Of course, you can't place text and elements anywhere you like. The rules in the DTD tell you which elements can go where. For example, the DTD for product lists, which we will develop later in this chapter, will tell you that the `price` element can be inside a `product` element but not inside an `item` element.

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

Table 4

Attribute Defaults

Advanced Topic   29.1

The XML Schema Specification

A limitation of DTDs is that you can't really specify what you want each XML element to contain. 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 Definition specification overcomes 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.

At the time of this writing, the XML Schema Definition specification was a recommendation and only experimental implementations were available. As you read this, XML schema support may be much more widespread.

Here is just a hint of how an XML schema is specified. For each element, you specify the element name and the type. The element names are the same as in a DTD, but the types are new. There are many basic types; Table 5 lists several.

Here is a typical definition:

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

This defines the quantity element as an element that contains text, where the text can't be arbitrary but must be an integer. 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 Advanced Topic 29.2 for more information about name spaces.

Furthermore, you can define complex types, much as you define classes in C++. Here is the definition of an Address type:

```
<xsd:complexType name="Address">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="company" 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 fields that are both of type Address:

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

To indicate that an element can be repeated, you use minOccurs and maxOccurs:

```
<xsd:complexType name="ItemList">
    <xsd:element name="item" type="Item"
        minOccurs="0" maxOccurs="unbounded"/>
</xsd:complexType>
```

These examples show that an XML schema can be both simpler and more precise than a DTD. For these reasons, it is likely that DTDs will be of lesser importance over the long run and that XML schema definitions will become the method of choice for describing XML document types.

▼   The XML Schema Definition specification has many advanced features—see the W3C
Web site [4] for details.

XML Schema Type	Meaning
string	Any string
integer	An integer
double	A floating-point number
boolean	False or true
date	A date such as 2001-01-23
time	A time of day such as 20:15

Table 5

A Subset of the XML Schema Basic Types

29.5   Parsing with Document Type Definitions

You have learned how to parse an XML document without a DTD. However, it is
always a good idea to reference a DTD with every XML document and to instruct the
parser to check that the document follows the rules of the DTD. That way, the parser
can check errors in the document and it can be more intelligent about parsing. For exam-
ple, if the parser knows that the children of an element are again elements, then it can
suppress white space, and you can save yourself the trouble of checking for white space.

In the preceding section you saw how to develop a DTD for a class of XML docu-
ments. 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 keyword inside the DOCTYPE declaration to indicate that the system that hosts the XML processor must locate the DTD. The SYSTEM keyword is followed by the location of the DTD. For example, a DOCTYPE declaration might point to a local file

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

Alternatively, the resource might be an 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 keyword. For example,

```
<!DOCTYPE web-app
    PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
```

A program parsing the DTD can look at the public identifier. If it is an identifier of a DTD that the program already knows, then it need not retrieve the DTD from the URL.

When you include a DTD with an XML document, then you should 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. You turn on validation with the following call:

```
parser->setFeature(XMLUni::fgDOMValidation, true);
```

When you use validation, you should install an error handler. An error handler is a class that inherits from the DOMErrorHandler class and implements the function

```
bool handleError(const DOMError& domError)
```

That function should log an error message and return `true` if the error was not fatal. The program at the end of this section contains a class `SimpleErrorHandler` that prints out error messages to `cout`.

You install the error handler into the parser:

```
DOMErrorHandler* handler = new SimpleErrorHandler();
parser->setErrorHandler(handler);
```

A great advantage of validation is that you know that the document is correct if it has been read in and validated. 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. Compare the program at the end of this section with the one from Section 29.2, and see how much simpler the code is.

If the parser has access to the DTD, it can make another useful improvement. Recall that in the preceding example program we had to skip the spaces that the parser reported. In that program, the parser didn't have access to the DTD, so it had no way of knowing whether spaces are significant or not. But if the parser knows the DTD, it can strip out white space in elements that don't have `#PCDATA` text as children (that is, elements that have element content). For example, in the product list DTD, the children of a `product` element are all elements, not text. Thus, any spaces between the elements can be ignored. To generate parsers with that behavior, you need to make the following call:

```
parser->setFeature(XMLUni::fgDOMWhitespaceInElementContent,
    false);
```

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 `getAttribute` function automatically supplies the default value.

```
XMLCh* attribute_value
    = price_element->getAttribute("currency");
    // Gets "USD" if no currency specified
```

As these arguments show, it is always a good idea to construct DTDs and to supply them with all XML documents.

This concludes our discussion of XML. You now know enough XML to put it to work for describing data formats in everyday programming. 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. For more advanced features that can be useful in specialized situations, please see [1]. Note that XML technology is still undergoing rapid change at the time of this writing; therefore, it is a good idea to check out the latest developments. Good Web sites are [4] and [5].

File parser2.cpp

```
1 #include <string>
2 #include <sstream>
3 #include <vector>
```

```
 4 #include <xercesc/dom/DOM.hpp>
 5 #include <xercesc/util/XMLString.hpp>
 6 #include <xercesc/util/PlatformUtils.hpp>
 7
 8 #include "item.h"
 9
10 using namespace std;
11 using namespace xercesc;
12
13 /*
14    Converts a sequence of XMLCh characters to a string.
15    @param in the sequence of XMLCh characters
16    @return the transcoded string
17 */
18 string XMLCh_to_string(const XMLCh* in)
19 {
20    char* s = XMLString::transcode(in);
21    string r(s);
22    XMLString::release(&s);
23    return r;
24 }
25
26 /**
27    Converts a string to a floating-point value, e.g. "3.14" -> 3.14.
28    @param s a string representing a floating-point value
29    @return the equivalent floating-point value
30 */
31 double string_to_double(string s)
32 {
33    istringstream instr(s);
34    double x;
35    instr >> x;
36    return x;
37 }
38
39 /**
40    Converts a string to an integer, e.g. "314" -> 314.
41    @param s a string representing an integer
42    @return the equivalent integer
43 */
44 int string_to_int(string s)
45 {
46    istringstream instr(s);
47    int x;
48    instr >> x;
49    return x;
50 }
51
52 /**
53    Obtains a product from a DOM element.
54    @param e a <product> element
55    @return the product described by the given element
56 */
57 Product get_product(DOMNode* e)
58 {
59    DOMNodeList* children = e->getChildNodes();
```

```
60
61     DOMText* textNode
62         = dynamic_cast<DOMText*>(children->item(0)->getFirstChild());
63     string name = XMLCh_to_string(textNode->getData());
64
65     textNode
66         = dynamic_cast<DOMText*>(children->item(1)->getFirstChild());
67     string price_text = XMLCh_to_string(textNode->getData());
68     double price = string_to_double(price_text);
69
70     return Product(name, price);
71 }
72
73 /**
74     Obtains an item from a DOM element.
75     @param e an <item> element
76     @return the item described by the given element
77 */
78 Item get_item(DOMNode* e)
79 {
80     DOMNodeList* children = e->getChildNodes();
81     Product p = get_product(children->item(0));
82
83     DOMText* textNode = dynamic_cast<DOMText*>(
84         children->item(1)->getFirstChild());
85     string quantity_text = XMLCh_to_string(textNode->getData());
86     int quantity = string_to_int(quantity_text);
87
88     return Item(p, quantity);
89 }
90
91 /**
92     Obtains an array list of items from a DOM element.
93     @param e an <items> element
94     @return a vector of all <item> children of e
95 */
96 vector<Item> get_items(DOMNode* e)
97 {
98     vector<Item> items;
99
100    // Get the <item> children
101
102    DOMNodeList* children = e->getChildNodes();
103    for (int i = 0; i < children->getLength(); i++)
104    {
105        Item c = get_item(children->item(i));
106        items.push_back(c);
107    }
108    return items;
109 }
110
111 class SimpleErrorHandler : public DOMErrorHandler
112 {
113 public:
114     bool handleError(const DOMError& domError);
115 };
```

```
116
117 bool SimpleErrorHandler::handleError(const DOMError& error)
118 {
119     cout << XMLCh_to_string(error.getLocation()->getURI())
120         << ", line " << error.getLocation()->getLineNumber()
121         << ", char " << error.getLocation()->getColumnNumber()
122         << ": " << XMLCh_to_string(error.getMessage()) << "\n";
123     return error.getSeverity() != DOMError::DOM_SEVERITY_FATAL_ERROR;
124 }
125
126 int main()
127 {
128     XMLPlatformUtils::Initialize();
129
130     DOMImplementation* implementation
131         = DOMImplementation::getImplementation();
132     DOMBuilder* parser = implementation->createDOMBuilder(
133         DOMImplementationLS::MODE_SYNCHRONOUS, NULL);
134     DOMErrorHandler* handler = new SimpleErrorHandler();
135     parser->setErrorHandler(handler);
136     parser->setFeature(XMLUni::fgDOMValidation, true);
137     parser->setFeature(XMLUni::fgDOMWhitespaceInElementContent, false);
138     DOMDocument* doc = parser->parseURI("items.xml");
139
140     DOMNode* root = doc->getDocumentElement();
141     vector<Item> items = get_items(root);
142
143     for (int i = 0; i < items.size(); i++)
144         items[i].print();
145
146     parser->release();
147     delete handler;
148
149     return 0;
150 }
```

Productivity Hint

29.3

Use DTDs with Your XML Files

It is an additional effort to develop a DTD for a set of XML files. However, that effort is repaid almost immediately, because the code that translates a document to a C++ object becomes much simpler. For example, contrast the get_item functions of the parser in Section 29.2 (which doesn't use a DTD) and that in Section 29.5 (which validates a DTD and skips white space). The first version doesn't know in which order the elements come, and it must manually skip white-space nodes. That makes the function somewhat tedious to implement.

The second version is much shorter. It simply follows the DTD rule

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

It knows that an item has two children: product and quantity.

```
Item get_item(DOMNode* e)
{
    DOMNodeList* children = e->getChildNodes();

    Product p = get_product(children->item(0));

    DOMText* textNode = dynamic_cast<DOMText*>(
        children->item(1)->getFirstChild());
    string quantity_text = XMLCh_to_string(textNode->getData());
    int quantity = string_to_int(quantity_text);

    return Item(p, quantity);
}
```

If there is an error in the XML file, then the parser will refuse to parse the file. Your code doesn't have to worry about bad inputs, which is a real time saver.

Advanced Topic

29.2

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 developments are currently emerging. Among them are:

- Schema definitions
- Name spaces
- XHTML
- XSL and transformations

Advanced Topic 29.1 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 Advanced Topic 29.1, 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. As with C++ name spaces, you use aliases (such as `xsd`) as shortcuts for much longer, unique strings. 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. As soon as 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 backwards 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. An XSL stylesheet can indicate 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 [4].

CHAPTER SUMMARY

1. XML allows you to encode complex data, independent from any programming language, in a form that a recipient can parse easily, and that is resilient to change.

2. An XML data set is called a document. It starts out with a header and contains elements and text. An element can contain text, subelements, or both (mixed content). For data descriptions, avoid mixed content. Elements can have attributes. Use attributes to describe how to interpret the element content.

3. A parser is a program that reads a document, checks whether it is syntactically correct, and takes some action as it processes the document. There are two kinds of XML parsers. SAX fires events as it analyzes a document. DOM builds a document tree.

4. A `DOMDocument` object consists of nodes that are represented by objects of the `DOMNode` class or one of its derived classes. The most important derived classes are `DOMElement` and `DOMText`.

5. A DTD is a sequence of rules that describe the legal 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 file name or URL.

6. It is best to use DTDs with XML files. When using DTDs, you should tell the parser to validate the document and to ignore white space. This greatly simplifies the implementation of the parser functions.

FURTHER READING

[1] `http://www.xml.com/axml/axml.html` Annotated XML specification.
[2] `http://xml.apache.org/xerces-c/` The Web site for the C++ version of the Xerces parser.
[3] Bjarne Stroustrup, *The C++ Programming Language*, Special Ed., Addison-Wesley, 2000.
[4] `http://www.w3.org/xml` The W3C XML Web site.
[5] `http://cafeconleche.org` A Web site for XML news.

REVIEW EXERCISES

Exercise R29.1. Give some examples to show the differences between XML and HTML.

Exercise R29.2. Design an XML document that describes a bank account.

Exercise R29.3. Draw a tree view for the XML document you created in Exercise R29.2.

Exercise R29.4. Write the XML document that corresponds to the parse tree in Figure 3.

Exercise R29.5. Write the XML document that corresponds to the parse tree in Figure 4.

Exercise R29.6. Make an XML document that describes a book, using child elements for the author name, title, and publication year.

Exercise R29.7. Add a description of the book's language to the document in Exercise R29.6. Should you use an element or an attribute?

Exercise R29.8. What is mixed content? What problems does it cause?

Exercise R29.9. Design an XML document that describes a purse containing three quarters, a dime, and two nickels.

Exercise R29.10. Explain why a paint program such as Microsoft Paint is a WYSIWYG program that is also "what you see is all you've got".

Exercise R29.11. How can you visit the children of a node without using a `NodeList`? (*Hint:* `getNextSibling`)

Exercise R29.12. What are the differences between a `DOMNodeList` and a `vector<DOMNode*>`?

Exercise R29.13. What are the differences between a `DOMNodeList` and a `list<DOMNode*>`?

Exercise R29.14. What are the differences between a `DOMNodeList` and a `DOMNamedNodeMap`?

Exercise R29.15. Design a DTD that describes a bank with bank accounts.

Exercise R29.16. 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.

Exercise R29.17. 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>
```

Exercise R29.18. Design a DTD for invoices as described in Chapter 13.

Exercise R29.19. Design a DTD for simple English sentences as described in Random Fact 29.2.

Exercise R29.20. Design a DTD for arithmetic expressions as described in Random Fact 29.2.

Programming Exercises

Exercise P29.1. Write a program that can read an XML file like this one:

```
<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.

Exercise P29.2. Building on Exercise P29.1, make the program read an XML file as described in that exercise. Then print out 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>
```
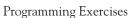
Exercise P29.3. Repeat Exercise P29.1, using a DTD for validation.

Exercise P29.4. Write a program that can read an XML file like this one:

```
<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.

Exercise P29.5. Repeat Exercise P29.4, using a DTD for validation.

Exercise P29.6. Enhance Exercise P29.4 as follows: First read the XML file in, add 10 percent interest to all accounts, and write an XML file that contains the increased account balances.

Exercise P29.7. Write a program that can read an XML document of the form

```
<rectangle>
    <x1>5</x>
    <y1>10</y>
    <x2>20</x2>
    <y2>30</y2>
</rectangle>
```

Draw the shape in a window.

Exercise P29.8. Write a program that can read an XML document of the form

```
<ellipse>
    <x>25</x>
    <y>30</y>
    <radius>20</radius>
</ellipse>
```

Draw the shape in a window.

Exercise P29.9. Write a program that can read an XML document of the form

```
<shape type="circle">
    <x>5</x>
    <y>10</y>
    <radius>20</radius>
</shape>
```

Support shape attributes "square", "circle", and "ellipse".

Draw the shape in a window.

Exercise P29.10. 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>
```

Show the drawing in a window.

Exercise P29.11. Repeat Exercise P29.10, using a DTD for validation.

Exercise P29.12. Write a program that can read an XML document of the form

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

Draw the shape in a window.

Exercise P29.13. Write a DTD file that describes documents that contain information about countries, such as the 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)

Exercise P29.14. Write a parser to parse invoices as described in Chapter 13. The parser should create a tree structure of the invoice first, then print it out in the format used in Chapter 13.

Exercise P29.15. Modify Exercise P29.14 to support separate shipping and billing addresses.

Exercise P29.16. Write a document builder that turns an invoice object, as defined in Chapter 13, into an XML file.

Exercise P29.17. Modify Exercise P29.16 to support separate shipping and billing addresses.

Exercise P29.18. Following Exercise P13.8, design an XML format for the appointments in an appointment calendar. Write a parser for the appointment file format that generates an `AppointmentCalendar` object containing `Appointment` objects.

Exercise P29.19. Enhance Exercise P29.18 so that the program 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.

Exercise P29.20. Write a program to simulate an airline seat reservation system, using XML documents. Reference Exercise P13.9 for the airplane seats information. The program reads a seating chart in an XML format of your choice, and a command file in an XML format of your choice—similar to the command file of Exercise P29.19. Then the program processes the commands and produces an updated seating chart.