18

Extensible Markup Language (XML)

Objectives

• To mark up data using XML.
• To understand the concept of an XML namespace.
• To understand the relationship between DTDs, Schemas and XML.
• To create Schemas.
• To create and use simple XSLT documents.
• To transform XML documents into XHTML, using class XsltTransform.

Knowing trees, I understand the meaning of patience.
Knowing grass, I can appreciate persistence.
Hal Borland

Like everything metaphysical, the harmony between thought and reality is to be found in the grammar of the language.
Ludwig Wittgenstein

I played with an idea and grew willful, tossed it into the air; transformed it; let it escape and recaptured it; made it iridescent with fancy, and winged it with paradox.
Oscar Wilde
Chapter 18
Extensible Markup Language (XML) 851

Outline

18.1 Introduction
18.2 XML Documents
18.3 XML Namespaces
18.4 Document Object Model (DOM)
18.5 Document Type Definitions (DTDs), Schemas and Validation
   18.5.1 Document Type Definitions
   18.5.2 Microsoft XML Schema
   18.5.3 W3C XML Schema
   18.5.4 Schema Validation in Visual C++.NET
18.6 Extensible Stylesheet Language and XslTransform
18.7 Internet and Web Resources


©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
This document begins with an optional XML declaration (line 1), which identifies the
document as an XML document. The version information parameter
specifies the version of XML that is used in the document. XML comments (lines 3–4), which begin with
<!-- and end with -->, can be placed almost anywhere in an XML document. As in an
MC++ program, comments are used in XML for documentation purposes.

Common Programming Error 18.1
Placing any characters, including white space, before the XML declaration is a syntax error.

Portability Tip 18.1
Although the XML declaration is optional, documents should include the declaration to identify
the version of XML used. Otherwise, in the future, a document that lacks an XML declaration
might be assumed to conform to the latest version of XML, and errors could result.

In XML, data are marked up with tags, which are names enclosed in angle brackets (<>).
Tags are used in pairs to delimit character data (e.g., Simple XML in line 8). A tag that begins
markup (i.e., XML data) is called a start tag; a tag that terminates markup is called an end tag.
Examples of start tags are <article> and <title> (lines 6 and 8, respectively). End tags
differ from start tags in that they contain a forward slash (/) character immediately after the
< character. Examples of end tags are </title> and </article> (lines 8 and 23, respectively). XML documents can contain any number of tags.

Common Programming Error 18.2
Failure to provide a corresponding end tag for a start tag is a syntax error.

Individual units of markup (i.e., everything from a start tag through its corresponding
end tag) are called elements. An XML document includes one element (called a root element or document element) that contains all other elements. The root element must be the
first element after the XML declaration. In Fig. 18.1, article (line 6) is the root element. Elements are nested within each other to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create explicit relationships between data. For example, elements title, date, author, summary and content are nested within article. Elements firstName and lastName are nested within author.

Common Programming Error 18.3

Attempting to create more than one root element in an XML document is a syntax error.

Element title (line 8) contains the title of the article, Simple XML, as character data. Similarly, date (line 10), summary (line 17) and content (lines 19–21) contain as character data the date, summary and content, respectively. XML element names can be of any length and may contain letters, digits, underscores, hyphens and periods—they must begin with a letter or an underscore.

Common Programming Error 18.4

XML is case sensitive. The use of the wrong case for an XML element name (in a begin tag, end tag, etc.) is a syntax error.

By itself, this document is simply a text file named article.xml. Although it is not required, most XML documents end in the file extension .xml. Processing XML documents requires a program called an XML parser (also called an XML processor). Parsers are responsible for checking an XML document’s syntax and making the XML document’s data available to applications. XML parsers are built into such applications as Visual Studio .NET or are available for download over the Internet. Popular parsers include Microsoft’s MSXML (msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/htm/sdk_intro_6g53.asp), the Apache Software Foundation’s Xerces (xml.apache.org) and IBM’s XML4J (www-106.ibm.com/developerworks/xml/library/x-xm4j/). In this chapter, we use MSXML.

When the user loads article.xml into Internet Explorer (IE), MSXML parses the document and passes the parsed data to IE. IE then uses a built-in style sheet to format the data. Notice that the resulting format of the data (Fig. 18.2) is similar to the format of the XML document shown in Fig. 18.1. As we soon demonstrate, style sheets play an important and powerful role in the transformation of XML data into formats suitable for display.

Notice the minus (–) and plus (+) signs in Fig. 18.2. Although these are not part of the XML document, IE places them next to all container elements (i.e., elements that contain other elements). Container elements also are called parent elements. A minus sign indicates that the parent element’s child elements (i.e., nested elements) are being displayed. When clicked, IE collapses the container element and hides its children, and the minus sign becomes a plus sign. Conversely, clicking a plus sign expands the container element and changes the plus sign to a minus sign. This behavior is similar to the viewing of the directory structure on a Windows system using Windows Explorer. In fact, a directory structure often is modeled as a series of tree structures, in which each drive letter (e.g., C:, etc.) represents the root of a tree. Each folder is a node in the tree. Parsers often place XML data into trees to facilitate efficient manipulation, as discussed in Section 18.4.

2. IE 5 and higher.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Common Programming Error 18.5

Nesting XML tags improperly is a syntax error. For example, `<x><y>hello</x></y>` is an error, because the child element’s ending tag (`</y>`) must precede the parent element’s ending tag (`</x>`).

We now present a second XML document (Fig. 18.3), which marks up a business letter. This document contains significantly more data than did the previous XML document.

Root element `letter` (lines 6–45) contains the child elements `contact` (lines 7–16 and 18–27), `salutation` (line 29), `paragraph` (lines 31–36 and 38–40), `closing` (line 42) and `signature` (line 44). In addition to being placed between tags, data also can be placed in attributes, which are name-value pairs separated by `=` in start tags. Elements can
### Extensible Markup Language (XML)

XML is a markup language for encoding documents in a format that is both human-readable and machine-readable. It is a standard for data interchange on the Internet. XML is designed to be a general-purpose, syntactic framework that can be extended to capture data structured in any way, regardless of the computing environment.

### Chapter 18

XML used to mark up a business letter.

```xml
<?xml version = "1.0"?>
<!-- Fig. 18.3: letter.xml -->
<!-- Business letter formatted with XML. -->
<letter>
  <contact type = "from">
    <name>Jane Doe</name>
    <address1>Box 12345</address1>
    <address2>15 Any Ave.</address2>
    <city>Othertown</city>
    <state>Otherstate</state>
    <zip>67890</zip>
    <phone>555-4321</phone>
    <flag gender = "F"/>
  </contact>
  <contact type = "to">
    <name>John Doe</name>
    <address1>123 Main St.</address1>
    <address2></address2>
    <city>Anytown</city>
    <state>Anystate</state>
    <zip>12345</zip>
    <phone>555-1234</phone>
    <flag gender = "M"/>
  </contact>
  <salutation>Dear Sir:</salutation>
  <paragraph>It is our privilege to inform you about our new database managed with <technology>XML</technology>. This new system allows you to reduce the load on your inventory list server by having the client machine perform the work of sorting and filtering the data.</paragraph>
  <paragraph>Please visit our Web site for availability and pricing.</paragraph>
  <closing>Sincerely</closing>
  <signature>Ms. Doe</signature>
</letter>
```

*Fig. 18.3* XML used to mark up a business letter.

have any number of attributes in their start tags. The first `contact` element (lines 7–16) has attribute `type` with attribute value "from", which indicates that this `contact` element marks up information about the letter’s sender. The second `contact` element (lines 18–27) has attribute `type` with value "to", which indicates that this `contact` element marks up information about the letter’s recipient. Like element names, attribute names can be of any length; may contain letters, digits, underscores, hyphens and periods; and must begin with...
either a letter or underscore character. A contact element contains a contact’s name, address, phone number and gender. Element salutation (line 29) marks up the letter’s salutation. Lines 31–40 mark up the letter’s body with paragraph elements. Elements closing (line 42) and signature (line 44) mark up the closing sentence and the signature of the letter’s author, respectively.

**Common Programming Error 18.6**

Failure to enclose attribute values in either double (""") or single ('') quotes is a syntax error.

**Common Programming Error 18.7**

Attempting to provide two attributes with the same name for an element is a syntax error.

In line 15, we introduce empty element flag, which indicates the gender of the contact. Empty elements do not contain character data (i.e., they do not contain text between the start and end tags). Such elements are closed either by placing a slash at the end of the element (as shown in line 15) or by explicitly writing a closing tag, as in

```
<flag gender = "F"></flag>
```

Notice that element address2 in line 21 also contains no data. Thus, we could safely change line 21 to `<address2 />`. However, it probably would not be wise to omit the element entirely, as it may be required by some DTD (Document Type Definition) or Schema. We discuss these in detail in Section 18.5.

### 18.3 XML Namespaces

The .NET Framework provides groups class libraries into namespaces. These namespaces prevent naming collisions between programmer-defined identifiers and identifiers in class libraries. For example, we might use class Book to represent information on one of our publications; however, a stamp collector might use class Book to represent a book of stamps. A naming collision would occur if we use these two classes in the same assembly, without using namespaces to differentiate them.

Like the .NET Framework, XML also provides namespaces, which provide a means of uniquely identifying XML elements. In addition, XML-based languages—called vocabularies, such as XML Schema (Section 18.5) and Extensible Stylesheet Language (Section 18.6)—often use namespaces to identify their elements.

Elements are differentiated via namespace prefixes, which identify the namespace to which an element belongs. For example,

```
```

qualifies element book with namespace prefix deitel. This indicates that element book is part of namespace deitel. Document authors can use any name for a namespace prefix except the reserved namespace prefix xml.

**Common Programming Error 18.8**

Attempting to create a namespace prefix named xml in any mixture of case is a syntax error.
The markup in Fig. 18.4 demonstrates namespaces. This XML document contains two file elements that are differentiated via namespaces.

```
<?xml version = "1.0"?>
<!-- Fig. 18.4: namespace.xml -->
<!-- Demonstrating namespaces. -->
<text:directory xmlns:text = "urn:deitel:textInfo"
    xmlns:image = "urn:deitel:imageInfo">
    <text:file filename = "book.xml">
        <text:description>A book list</text:description>
    </text:file>
    <image:file filename = "funny.jpg">
        <image:description>A funny picture</image:description>
        <image:size width = "200" height = "100" />
    </image:file>
</text:directory>
```

Fig. 18.4 XML namespaces demonstration.

**Software Engineering Observation 18.1**

A programmer has the option of qualifying an attribute with a namespace prefix. However, doing so is not required, because attributes always are associated with elements that would already be qualified with a namespace.

Lines 6–7 (Fig. 18.4) use attribute `xmlns` to create two namespace prefixes: `text` and `image`. Each namespace prefix is bound to a series of characters called a **uniform resource identifier (URI)** that uniquely identifies the namespace. Document authors create their own namespace prefixes and URIs. Notice that we use the `text` namespace prefix on the same line that the prefix is created (line 6).

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
To ensure that namespaces are unique, document authors must provide unique URIs. Here, we use the text "urn:deitel:textInfo" and "urn:deitel:imageInfo" as URIs. A common practice is to use Uniform Resource Locators (URLs) for URIs, because the domain names (such as www.deitel.com) used in URLs are guaranteed to be unique. For example, lines 6–7 could have been written as:

```xml
<text:directory xmlns:text = "http://www.deitel.com/xmlns-text"
    xmlns:image = "http://www.deitel.com/xmlns-image">
```

In this example, we use URLs related to the Deitel & Associates, Inc, domain name to identify namespaces. The XML parser never visits these URLs—they simply represent a series of characters used to differentiate names. The URLs need not refer to actual Web pages or be properly formed.

Lines 9–11 use the namespace prefix `text` to qualify elements `file` and `description` as belonging to the namespace "urn:deitel:textInfo". Notice that the namespace prefix `text` is applied to the end tags as well. Lines 13–16 apply namespace prefix `image` to elements `file`, `description` and `size`.

To eliminate the need to precede each element with a namespace prefix, document authors can specify a default namespace. Figure 18.5 demonstrates default namespaces.

Line 6 (Fig. 18.5) declares a default namespace using attribute `xmlns` with a URI as its value. Once we define this default namespace, child elements belonging to the namespace need not be qualified by a namespace prefix. Element `file` (line 9–11) is in the namespace `urn:deitel:textInfo`. Compare this to Fig. 18.4, where we prefixed `file` and `description` with `text` (lines 9–11).

The default namespace applies to the `directory` element and all elements that are not qualified with a namespace prefix. However, we can use a namespace prefix to specify a different namespace for particular elements. For example, the `file` element in line 13 is prefixed with `image` to indicate that it is in the namespace `urn:deitel:imageInfo`, rather than the default namespace.

---

```xml
<?xml version = "1.0"?>
<directory xmlns = "urn:deitel:textInfo"
    xmlns:image = "urn:deitel:imageInfo">
    <file filename = "book.xml">
        <description>A book list</description>
    </file>
    <image:file filename = "funny.jpg">
        <image:description>A funny picture</image:description>
        <image:size width = "200" height = "100" />
    </image:file>
</directory>
```

---

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
18.4 Document Object Model (DOM)

Although XML documents are text files, retrieving data from them via sequential-file access techniques is neither practical nor efficient, especially in situations where data must be added or deleted dynamically.

Upon successful parsing of documents, some XML parsers store document data as tree structures in memory. Figure 18.6 illustrates the tree structure for the document article.xml discussed in Fig. 18.1. This hierarchical tree structure is called a Document Object Model (DOM) tree, and an XML parser that creates this type of structure is known as a DOM parser. The DOM tree represents each component of the XML document (e.g., article, date, firstName, etc.) as a node in the tree. Nodes (such as author) that contain other nodes (called child nodes) are called parent nodes. Nodes that have the same parent (such as firstName and lastName) are called sibling nodes. A node’s descendant nodes include that node’s children, its children’s children and so on. Similarly, a node’s ancestor nodes include that node’s parent, its parent’s parent and so on. Every DOM tree has a single root node that contains all other nodes in the document, such as comments and elements.
Classes for creating, reading and manipulating XML documents are located in namespace `System::Xml`. This namespace also contains additional namespaces that contain other XML-related classes.

Creating DOM Trees Programmatically

In this section, we present several examples that use DOM trees. Our first example, the program in Fig. 18.7–Fig. 18.8, loads the XML document presented in Fig. 18.1 and displays its data in a text box. This example uses class `XmlNodeReader` (derived from `XmlReader`), which iterates through each node in the XML document. Class `XmlReader` is an abstract class that defines the interface for reading XML documents.

```
1 // Fig. 18.7: Form1.h
2 // Reading an XML document.
3
4 #pragma once
5
6 namespace XmlReaderTest
7 {
8     using namespace System;
9     using namespace System::ComponentModel;
10    using namespace System::Collections;
11    using namespace System::Windows::Forms;
12    using namespace System::Data;
13    using namespace System::Drawing;
14    using namespace System::Xml;
15
16    /// <summary>
17    /// Summary for Form1
18    ///
19    /// WARNING: If you change the name of this class, you will need to
20    /// change the 'Resource File Name' property for the managed
21    /// resource compiler tool associated with all .resx files
22    /// this class depends on. Otherwise, the designers will not
23    /// be able to interact properly with localized resources
24    /// associated with this form.
25    /// </summary>
26    public __gc class Form1 : public System::Windows::Forms::Form
27    {
28        public:
29            Form1(void)
30            {
31                InitializeComponent();
32                PrintXml1();
33            }
34
35        protected:
36            void Dispose(Boolean disposing)
37            {
38                if (disposing && components)
39                {
40
Fig. 18.7  Iterating through an XML document. (Part 1 of 3.)
```

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
components->Dispose();
}
__super::Dispose(disposing);
}
private: System::Windows::Forms::TextBox * outputTextBox;
private:
/// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container * components;

// Visual Studio .NET generated GUI code
private:

void PrintXml()
{
    // create and load XMLDocument
    XmlDocument *document = new XmlDocument();
document->Load(S"article.xml");

    // create XmlNodeReader for document
    XmlNodeReader *reader = new XmlNodeReader( document );

    // show form before outputTextBox is populated
    this->Show();

    // tree depth is -1, no indentation
    int depth = -1;

    // display each node's content
    while ( reader->Read() ) {

        switch ( reader->NodeType ) {

        // if Element, display its name
        case XmlNodeType::Element:

            // increase tab depth
depth++;
            TabOutput( depth );
            outputTextBox->AppendText( String::Concat( S"<", reader->Name, S"\r\n" ) );

        // if empty element, decrease depth
        if ( reader->IsEmptyElement )
depth--;
break;

        // if Comment, display it
        case XmlNodeType::Comment:

        }
    }
}

Fig. 18.7 Iterating through an XML document. (Part 2 of 3.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
TabOutput( depth );
outputTextBox->AppendText( String::Concat(
S "<--", reader->Value, S "\r\n" ) );
break;

// if Text, display it
  case XmlNodeType::Text:
  TabOutput( depth );
  outputTextBox->AppendText( String::Concat(
S "\t", reader->Value, S "\r\n" ) );
  break;

// if XML declaration, display it
  case XmlNodeType::XmlDeclaration:
  TabOutput( depth );
  outputTextBox->AppendText( String::Concat( S "<!--",
reader->Name, S "", reader->Value, S "-->\r\n" ) );
  break;

// if EndElement, display it and decrement depth
  case XmlNodeType::EndElement:
  TabOutput( depth );
  outputTextBox->AppendText( String::Concat( S "</",
reader->Name, S "\r\n" ) );
depth--;
break;

} // end switch
} // end while
} // end method PrintXml

private:
  // insert tabs
  void TabOutput( int number )
  {
    for ( int i = 0; i < number; i++ )
      outputTextBox->AppendText( S "\t" );
  } // end method TabOutput

Fig. 18.7 Iterating through an XML document. (Part 3 of 3.)

Fig. 18.8 XmlReaderTest entry point. (Part 1 of 2.)
Chapter 18  Extensible Markup Language (XML)  863

Line 15 (Fig. 18.7) includes the System::Xml namespace, which contains the XML classes used in this example. Line 60 (Fig. 18.7) creates a pointer to an XmlDocument object that conceptually represents an empty XML document. The XML document article.xml is parsed and loaded into this XmlDocument object when method Load is invoked in line 61. Once an XML document is loaded into an XmlDocument object, its data can be read and manipulated programmatically. In this example, we read each node in the XmlDocument, which represents the DOM tree. In later examples, we demonstrate how to manipulate node values.

Line 64 creates an XmlNodeReader and assigns it to pointer reader, which enables us to read one node at a time from the XmlDocument. Method Read of XmlNodeReader reads one node from the DOM tree. Placing this statement in the while loop (lines 73–122) makes reader read all the document nodes. The switch statement (lines 75–121) processes each node. Either the Name property (line 84), which contains the node’s name, or the Value property (line 96), which contains the node’s data, is formatted and concatenated to the String displayed in outputTextBox. The NodeType property contains the node type (specifying whether the node is an element, comment, text, etc.). Notice that each case specifies a node type, using XmlNodeType enumeration constants.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Notice that the displayed output emphasizes the structure of the XML document. Variable \texttt{depth} (line 70) sets the number of tab characters used to indent each element. The depth is incremented each time an \texttt{Element} type is encountered and is decremented each time an \texttt{EndElement} or empty element is encountered. We use a similar technique in the next example to emphasize the tree structure of the XML document in the display. Notice that our line breaks use the character sequence "\\r\\n", which denotes a carriage return followed by a line feed. This is the standard line break for Windows-based applications and controls.

\textbf{Manipulating DOM Trees Programmatically}

The program in Fig. 18.9–Fig. 18.10 demonstrates how to manipulate DOM trees programmatically. This program loads \texttt{letter.xml} (Fig. 18.3) into the DOM tree, then creates a second DOM tree that duplicates the DOM tree containing \texttt{letter.xml}'s contents. The GUI for this application contains a text box, a \texttt{TreeView} control and three buttons—\texttt{Build}, \texttt{Print} and \texttt{Reset}. When clicked, \texttt{Build} copies \texttt{letter.xml} and displays the document's tree structure in the \texttt{TreeView} control, \texttt{Print} displays the XML element values and names in a text box and \texttt{Reset} clears the \texttt{TreeView} control and text box content.

Lines 62 and 65 (Fig. 18.9) create pointers to \texttt{XmlDocuments} \texttt{source} and \texttt{copy}. Line 39 in the constructor assigns a new \texttt{XmlNode} object to pointer \texttt{source}. Line 40 then invokes method \texttt{Load} to load and parse \texttt{letter.xml}. We discuss pointer \texttt{copy} shortly.

Unfortunately, \texttt{XmlDocuments} do not provide any features for displaying their content graphically. In this example, we display the document's contents via a \texttt{TreeView} control. We use objects of class \texttt{TreeNode} to represent each node in the tree. Class \texttt{TreeNode} and class \texttt{TreeView} are part of namespace \texttt{System::Windows::Forms}. \texttt{TreeNode}s are added to the \texttt{TreeView} to emphasize the structure of the XML document.

---

\begin{verbatim}
// Fig. 18.9: Form1.h
// Demonstrates DOM tree manipulation.

#pragma once

namespace XmlDom
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::IO;
    using namespace System::Xml;

    // contains TempFileCollection
    using namespace System::CodeDom::Compiler;

    /// <summary>
    /// Summary for Form1
    /// </summary>
} // namespace XmlDom

Fig. 18.9  DOM structure of an XML document illustrated by a class. (Part 1 of 6.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
\end{verbatim}
/// WARNING: If you change the name of this class, you will need to
/// change the 'Resource File Name' property for the managed
/// resource compiler tool associated with all .resx files
/// this class depends on. Otherwise, the designers will not
/// be able to interact properly with localized resources
/// associated with this form.
/// </summary>
public __gc class Form1 : public System::Windows::Forms::Form
{
public:
Form1(void)
{
InitializeComponent();

// create XmlDocument and load letter.xml
source = new XmlDocument();
source->Load( S"letter.xml" );

// initialize pointers to 0
copy = 0;
tree = 0;
}

protected:\nvoid Dispose(Boolean disposing)
{
if (disposing && components)
{
components->Dispose();

__super::Dispose(disposing);
}
private: System::Windows::Forms::Button * resetButton;
private: System::Windows::Forms::Button * buildButton;
private: System::Windows::Forms::TreeView * xmlTreeView;
private: System::Windows::Forms::TextBox * consoleTextBox;
private: System::Windows::Forms::Button * printButton;
private: XmlDocument *source; // pointer to "XML document"
private: XmlDocument *copy;
private: TreeNode *tree; // TreeNode pointer
private:
/// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container * components;

// Visual Studio .NET generated GUI code

Fig. 18.9   DOM structure of an XML document illustrated by a class. (Part 2 of 6.)
```csharp
private: System::Void buildButton_Click(System::Object * sender, System::EventArgs * e) {
    // determine if copy has been built already
    if ( copy != 0 ) return; // document already exists
    // instantiate XmlDocument and TreeNode
    copy = new XmlDocument();
    tree = new TreeNode();
    // add root node name to TreeNode and add
    // TreeNode to TreeView control
    tree->Text = source->Name; // assigns #root
    xmlTreeView->Nodes->Add( tree );
    // build node and tree hierarchy
    BuildTree( source, copy, tree );
    printButton->Enabled = true;
    resetButton->Enabled = true;
} // end method buildButton_Click

private: System::Void printButton_Click(System::Object * sender, System::EventArgs * e) {
    // exit if copy does not point to an XmlDocument
    if ( copy == 0 ) return;
    // create temporary XML file
    TempFileCollection *file = new TempFileCollection();
    // create file that is deleted at program termination
    String *filename = file->AddExtension( S"xml", false );
    // write XML data to disk
    XmlTextWriter *writer = new XmlTextWriter( filename, System::Text::Encoding::UTF8 );
    copy->WriteTo( writer );
    writer->Close();
    // parse and load temporary XML document
    XmlTextReader *reader = new XmlTextReader( filename );
    // read, format and display data
    while ( reader->Read() ) {
        if ( reader->NodeType == XmlNodeType::EndElement )
            consoleTextBox->AppendText( S""/"" );
```
```csharp
if ( reader->Name != String::Empty )
    consoleTextBox->AppendText(String::Concat( reader->Name, S"\r\n" ));
    if ( reader->Value != String::Empty )
        consoleTextBox->AppendText(String::Concat( S"\t", reader->Value, S"\r\n" ));

reader->Close();

// handle resetButton click event
private: System::Void resetButton_Click(System::Object * sender, System::EventArgs * e)
{
    // remove TreeView nodes
    if ( tree != 0 )
        xmlTreeView->Nodes->Remove( tree );

    xmlTreeView->Refresh(); // force TreeView update

    // delete XmlDocument and tree
    copy = 0;
    tree = 0;

    consoleTextBox->Text = S""
        // clear text box

    printButton->Enabled = false;
    resetButton->Enabled = false;
}

private:

// construct DOM tree
void BuildTree(XMLElement *xmlSourceNode,
               XMLElement *document, TreeNode *treeNode )
{
    XmlNodeReader *nodeReader = new XmlNodeReader( xmlSourceNode );

    XmlNode *currentNode = 0;

    while ( nodeReader->Read() )
    {
        // represents current node in DOM tree
        XMLElement *currentNode = 0;

        // treeNode to add to existing tree
        TreeNode *newNode = new TreeNode();

        // points to modified node type for CreateNode
        XMLElementType modifiedNodeType;

        while ( nodeReader->Read() )
        {

            if ( reader->Name != String::Empty )
                consoleTextBox->AppendText(String::Concat( reader->Name, S"\r\n" ));

            if ( reader->Value != String::Empty )
                consoleTextBox->AppendText(String::Concat( S"\t", reader->Value, S"\r\n" ));

            reader->Close();
        }
    }
```
182 // get current node type
183 modifiedNodeType = nodeReader->NodeType;
184
185 // check for EndElement, store as Element
186 if ( modifiedNodeType == XmlNodeType::EndElement )
187 modifiedNodeType = XmlNodeType::Element;
188
189 // create node copy
190 currentNode = copy->CreateNode( modifiedNodeType,
191 nodeReader->Name, nodeReader->NamespaceURI );
192
193 // build tree based on node type
194 switch ( nodeReader->NodeType ) {
195
196 // if Text node, add its value to tree
197 case XmlNodeType::Text:
198 newNode->Text = nodeReader->Value;
199 treeNode->Nodes->Add( newNode );
200
201 // append Text node value to currentNode data
202 dynamic_cast< XmlText * >( currentNode )->
203 AppendData( nodeReader->Value );
204 document->AppendChild( currentNode );
205 break;
206
207 // if EndElement, move up tree
208 case XmlNodeType::EndElement:
209 document = document->ParentNode;
210 treeNode = treeNode->Parent;
211 break;
212
213 // if new element, add name and traverse tree
214 case XmlNodeType::Element:
215
216 // determine if element contains content
217 if ( !nodeReader->IsEmptyElement ) {
218
219 // assign node text, add newNode as child
220 newNode->Text = nodeReader->Name;
221 treeNode->Nodes->Add( newNode );
222
223 // set treeNode to last child
224 treeNode = newNode;
225
226 document->AppendChild( currentNode );
227 document = document->LastChild;
228 } // end if
229
230 // do not traverse empty elements
231 else {
232
233 // assign NodeType string to newNode

Fig. 18.9  DOM structure of an XML document illustrated by a class. (Part 5 of 6.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
newNode->Text =
  __box( nodeReader->NodeType )->ToString();
treeNode->Nodes->Add( newNode );
document->Nodes->Add( newNode );
} // end else
break;

// all other types, display node type
default:
  newNode->Text =
    __box( nodeReader->NodeType )->ToString();
treeNode->Nodes->Add( newNode );
document->AppendChild( currentNode );
break;
} // end switch

newNode = new TreeNode();
} // end while

// update the TreeView control
xmlTreeView->ExpandAll();
xm1TreeView->Refresh();
} // end method BuildTree
};

Fig. 18.9  DOM structure of an XML document illustrated by a class. (Part 6 of 6.)

// Fig. 18.10: Form1.cpp
// Entry point for application.

#include "stdafx.h"
#include "Form1.h"
#include <windows.h>

using namespace XmlDom;

int APIENTRY _tWinMain(HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPTSTR    lpCmdLine,
  int       nCmdShow)
{
  System::Threading::Thread::CurrentThread->ApartmentState =
    System::Threading::ApartmentState::STA;
  Application::Run(new Form1());
  return 0;
} // end _tWinMain

Fig. 18.10  XmlDom entry point. (Part 1 of 2.)
When clicked, button **Build** triggers event handler **buildButton_Click** (lines 78–99), which copies **letter.xml** dynamically. The new **XmlDocument** and **TreeNode**s (i.e., the nodes used for graphical representation in the **TreeView**) are created in lines 86–87. Line 91 retrieves the **Name** of the node pointed to by **source** (i.e., #document, which represents the document root) and assigns it to **tree**’s **Text** property. This **TreeNode** then is inserted into the **TreeView** control’s node list. Method **Add** is called to add each new **TreeNode** to the **TreeView**’s **Nodes** collection (line 92). Line 95 calls method **BuildTree** to copy the XML-Document pointed to by **source** and to update the **TreeView**.

Method **BuildTree** (line 165–258) receives an **XmlNode** representing the source node, an empty **XmlNode** and a **treeNode** to place in the DOM tree. Parameter **treeNode** points to the current location in the tree (i.e., the **TreeNode** most recently added to the **TreeView** control). Line 169 instantiates a new **XmlNodeReader** for iterating through the
DOM tree. Lines 172 and 175 declare XmlNode and TreeNode pointers that indicate the next nodes added to document (i.e., the DOM tree pointed to by copy) and treeNode. Lines 180–253 iterate through each node in the tree.

Lines 190–191 create a node containing a copy of the current nodeReader node. Method CreateNode of XmlDocument takes a NodeType, a Name and a NamespaceURI as arguments. The NodeType cannot be an EndElement or method CreateNode throws an ArgumentOutOfRangeException. If the NodeType is of type EndElement, lines 186–187 assign modifiedNodeType type Element.

The switch statement in lines 194–250 determines the node type, creates and adds nodes to the TreeView and updates the DOM tree. When a text node is encountered, the new TreeNode's newNode's Text property is assigned the current node's value. This TreeNode is added to the TreeView control. Lines 202–204 downcast currentNode to XmlText and append the node's value. The currentNode then is appended to the document. Lines 208–211 match an EndElement node type. This case moves up the tree, because the end of an element has been encountered. The ParentNode and Parent properties retrieve the document's and treeNode's parents, respectively.

Line 214 matches Element node types. Each nonempty Element (line 217) increases the depth of the tree; thus, we assign the current nodeReader's Name to the newNode's Text property and add the newNode to the treeNode node list. Lines 220–224 reorder the nodes in the node list to ensure that newNode is the last TreeNode in the node list. XmlNode currentNode is appended to document as the last child, and document is set to its LastChild, which is the child we just added. For an empty element (line 231), we assign to the newNode's Text property the String representation of the NodeType. Next, the newNode is added to the treeNode node list. Line 238 appends the currentNode to the document. The default case assigns the string representation of the node type to the NewNode Text property, adds the newNode to the TreeNode node list and appends the currentNode to the document.

After building the DOM trees, the TreeNode node list displays in the TreeView control. Clicking the nodes (i.e., the + or - boxes) in the TreeView either expands or collapses them. Clicking Print invokes event handler printButton_Click (lines 102–140). Lines 110 and 113 create a temporary file for storing the XML. Line 110 creates an instance of class TempFileCollection (namespace System::CodeDom::Compiler). This class can be used to create and delete temporary files (i.e., files that store short-lived information). Line 113 calls TempFileCollection method AddExtension. We use a version of this method that accepts two arguments. The first argument is a String * that specifies the file extension of the file to create—in this case, "xml". The second argument is a bool that specifies whether temporary files of this type (xml) should be kept after being used (i.e., when the TempFileCollection object is destroyed). We pass the value false, indicating that the temporary files (of type xml) should be deleted when the TempFileCollection is destroyed (i.e., goes out of scope). Method AddExtension returns the filename that it has just created, which we store in String * filename.

Lines 116–117 then create an XmlTextWriter for streaming the XML data to disk. The first argument to the XmlTextWriter constructor is the filename that it will use to output the data (filename). The second argument passed to the XmlTextWriter constructor specifies the encoding to use. We specify UTF-8, an 8-bit encoding for Unicode characters. For more information about Unicode, refer to Appendix D.
Line 118 calls method WriteTo to write the XML representation to the XmlTextWriter stream. Line 122 creates an XmlTextReader to read from the file. The while loop (line 125–137) reads each node in the DOM tree and writes tag names and character data to the textbox. If it is an end element, a slash is concatenated. If the node has a Name or Value, that name or value is concatenated to the textbox text.

The Reset button’s event handler, resetButton_Click (lines 143–160), deletes both dynamically generated trees and updates the TreeView control’s display. Pointer copy is assigned 0 (to allow its tree to be garbage collected in line 153), and the TreeNode node list pointer tree is assigned 0.

**XPath Expressions**

Although XmlReader includes methods for reading and modifying node values, it is not the most efficient means of locating data in a DOM tree. The .NET framework provides class XPathNavigator in namespace System::Xml::XPath for iterating through node lists that match search criteria, which are written as XPath expressions. XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies (such as XSLT, discussed in Section 18.6).

Figure 18.11–Fig. 18.12 demonstrate how to navigate through an XML document with an XPathNavigator. Like Fig. 18.9–Fig. 18.10, this program uses a TreeView control and TreeNode objects to display the XML document’s structure. However, instead of displaying the entire DOM tree, the TreeNode node list is updated each time the XPathNavigator is positioned to a new node. Nodes are added to and deleted from the TreeView to reflect the XPathNavigator’s location in the DOM tree. The XML document sports.xml that we use in this example is presented in Fig. 18.13.

This program (Fig. 18.11–Fig. 18.12) loads XML document sports.xml (Fig. 18.13) into an XPathDocument object by passing the document’s file name to the XPathDocument constructor (line 36 of Fig. 18.11). Method CreateNavigator (line 39) creates and returns an XPathNavigator pointer to the XPathDocument’s tree structure.

The navigation methods of XPathNavigator used in Fig. 18.11 are MoveToFirstChild (line 96), MoveToParent (line 123), MoveToNext (line 150) and MoveToPrevious (line 178). Each method performs the action that its name implies. Method MoveToFirstChild moves to the first child of the node pointed to by the XPathNavigator. MoveToParent moves to the parent node of the node pointed to by the XPathNavigator. MoveToNext moves to the next sibling of the node pointed to by the XPathNavigator and MoveToPrevious moves to the previous sibling of the node pointed to by the XPathNavigator. Each method returns a bool indicating whether the move was successful. We display a warning in a MessageBox whenever a move operation fails. Each of the XPathNavigator methods is called in the event handler of the button that matches the method name (e.g., button First Child triggers firstChildButton_Click (line 90), which calls MoveToFirstChild).

Whenever we move forward via the XPathNavigator, as with MoveToFirstChild and MoveToNext, nodes are added to the TreeNode node list. Method DetermineType is a private method (defined in lines 243–256) that determines whether to assign the Node’s Name property or Value property to the TreeNode (lines 249 and 254). Whenever MoveToParent is called, all children of the parent node are removed from the display.
Similarly, a call to MoveToPrevious removes the current sibling node. Note that the nodes are removed only from the TreeView, not from the tree representation of the document.

```cpp
// Fig. 18.11: Form1.h
// Demonstrates Class XPathNavigator.

#pragma once

namespace PathNavigator
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::Xml;
    using namespace System::Xml::XPath; // contains XPathNavigator

    /// <summary>
    /// Summary for Form1
    ///
    /// WARNING: If you change the name of this class, you will need to change
    ///          the 'Resource File Name' property for the managed resource
    ///          compiler tool associated with all .resx files
    ///          this class depends on. Otherwise, the designers will not be able
    ///          to interact properly with localized resources associated with this form.
    /// </summary>
    public __gc class Form1 : public System::Windows::Forms::Form
    {
        public:
            Form1(void)
            {
                InitializeComponent();

                // load in XML document
                document = new XPathDocument( S"sports.xml" );

                // create navigator
                xpath = document->CreateNavigator();

                // create root node for TreeNodes
                tree = new TreeNode();
                tree->Text = __box( xpath->NodeType )->ToString(); // #root
                pathTreeViewer->Nodes->Add( tree ); // add tree

                // update TreeView control
                pathTreeViewer->ExpandAll();
                pathTreeViewer->Refresh();
                pathTreeViewer->SelectedNode = tree; // highlight root
            }
}
```

Fig. 18.11 XPathNavigator class used to navigate selected nodes. (Part 1 of 5.)
protected:
  void Dispose(Boolean disposing)
  {
    if (disposing && components)
    {
      components->Dispose();
    }
    __super::Dispose(disposing);
  }

private: System::Windows::Forms::Button *  firstChildButton;
private: System::Windows::Forms::Button *  parentButton;
private: System::Windows::Forms::Button *  nextButton;
private: System::Windows::Forms::Button *  previousButton;
private: System::Windows::Forms::Button *  selectButton;
private: System::Windows::Forms::TreeView *  pathTreeViewer;
private: System::Windows::Forms::ComboBox *  selectComboBox;
private: System::Windows::Forms::TextBox *  selectTreeViewer;
private: System::Windows::Forms::GroupBox *  navigateBox;
private: System::Windows::Forms::GroupBox *  locateBox;

// navigator to traverse document
private: XPathNavigator *xpath;

// points to document for use by XPathNavigator
private: XPathDocument *document;

// points to TreeNode list used by TreeView control
private: TreeNode *tree;

private:
  /// <summary>
  /// Required designer variable.
  /// </summary>
  System::ComponentModel::Container * components;

// traverse to first child
private: System::Void firstChildButton_Click(
    System::Object * sender, System::EventArgs * e)
  {
    TreeNode *newTreeNode;
    // move to first child
    if ( xpath->MoveToFirstChild() )  {
      newTreeNode = new TreeNode(); // create new node
      // set node's Text property to either
      // navigator's name or value
      DetermineType( newTreeNode, xpath );
  
Fig. 18.11 XPathNavigator class used to navigate selected nodes. (Part 2 of 5.)
// add node to TreeNode node list
tree->Nodes->Add( newTreeNode );
tree = newTreeNode; // assign tree newTreeNode

// update TreeView control
pathTreeViewer->ExpandAll();
pathTreeViewer->Refresh();
pathTreeViewer->SelectedNode = tree;
} // end if
else // node has no children
    MessageBox::Show( S"Current Node has no children.",
                      S"", MessageBoxButtons::OK,
                      MessageBoxIcon::Information );
} // end method firstChildButton_Click

// traverse to node's parent on parentButton click event
private: System::Void parentButton_Click(System::Object * sender, System::EventArgs * e)
{
    // move to parent
    if ( xpath->MoveToParent() ) {
        tree = tree->Parent;

        // get number of child nodes, not including subtrees
        int count = tree->GetNodeCount( false );

        // remove all children
        tree->Nodes->Clear();

        // update TreeView control
        pathTreeViewer->ExpandAll();
        pathTreeViewer->Refresh();
        pathTreeViewer->SelectedNode = tree;
    } // end if
    else // if node has no parent (root node)
        MessageBox::Show( S"Current node has no parent.", S"", MessageBoxButtons::OK,
                          MessageBoxIcon::Information );
} // end method parentButton_Click

// find next sibling on nextButton click event
private: System::Void nextButton_Click(System::Object * sender, System::EventArgs * e)
{
    TreeNode *newTreeNode = 0, *newNode = 0;

    // move to next sibling
    if ( xpath->MoveToNext() ) {
        newTreeNode = tree->Parent; // get parent node

        newNode = new TreeNode(); // create new node
        DetermineType( newNode, xpath );

        newTreeNode->Nodes->Add( newNode );
    } // end if

Fig. 18.11 XPathNavigator class used to navigate selected nodes. (Part 3 of 5.)
// set current position for display

// update TreeView control
pathTreeViewer->ExpandAll();
pathTreeViewer->Refresh();
pathTreeViewer->SelectedNode = tree;
} // end if
else // node has no additional siblings
    MessageBox::Show(S"Current node is last sibling.",
        S"", MessageBoxButtons::OK,
        MessageBoxIcon::Information );
} // end method nextButton_Click

// get previous sibling on previousButton click
private: System::Void previousButton_Click(
    System::Object * sender, System::EventArgs * e)
{
    TreeNode *parentTreeNode = 0;

    // move to previous sibling
    if (xpath->MoveToPrevious() ) {
        parentTreeNode = tree->Parent; // get parent node

        // delete current node
        parentTreeNode->Nodes->Remove( tree );

        // move to previous node
        tree = parentTreeNode->LastNode;

        // update TreeView control
        pathTreeViewer->ExpandAll();
        pathTreeViewer->Refresh();
        pathTreeViewer->SelectedNode = tree;
    } // end if

    else // if current node has no previous siblings
        MessageBox::Show(S"Current node is first sibling.",
            S"", MessageBoxButtons::OK,
            MessageBoxIcon::Information );
} // end method previousButton_Click

// process selectButton click event
private: System::Void selectButton_Click( 
    System::Object * sender, System::EventArgs * e)
{
    XPathNodeIterator *iterator; // enables node iteration

    // get specified node from ComboBox
    try {
        iterator = xpath->Select( selectComboBox->Text );
        DisplayIterator( iterator ); // print selection
    } // end try

Fig. 18.11 XPathNavigator class used to navigate selected nodes. (Part 4 of 5.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
// catch invalid expressions
catch (ArgumentException *argumentException) {
    MessageBox::Show(argumentException->Message,
                     S"Error", MessageBoxButtons::OK,
                     MessageBoxIcon::Error);
} // end catch

// catch empty expressions
catch (XPathException *) {
    MessageBox::Show(S"Please select an expression",
                     S"Error", MessageBoxButtons::OK,
                     MessageBoxIcon::Error);
} // end catch
} // end method selectButton_Click

private:

// print values for XPathNodeIterator
void DisplayIterator(XPathNodeIterator *iterator) {
    selectTreeViewer->Text = S""
    while (iterator->MoveNext()) {
        selectTreeViewer->Text =
            String::Concat(selectTreeViewer->Text,
                           iterator->Current->Value->Trim(), S"\r\n");
    } // end while
} // end method DisplayIterator

private:

// determine if TreeNode should display current node name or value
void DetermineType(TreeNode *node, XPathNavigator *xPath) {
    if (xPath->NodeType == XPathNodeType::Element) {
        // get current node name, remove white space
        node->Text = xPath->Name->Trim();
    } // end if
    else {
        // get current node value, remove white space
        node->Text = xPath->Value->Trim();
    } // end else
} // end method DetermineType

Fig. 18.11 XPathNavigator class used to navigate selected nodes. (Part 5 of 5.)
The other event handler corresponds to button **Select** (line 199–223). Method **Select** (line 206) takes search criteria in the form of either an **XPathExpression** or a **String** that represents an XPath expression and returns as an **XPathNodeIterator** object any nodes that match the search criteria. The XPath expressions provided by this program’s combo box are summarized in Fig. 18.14. The catch blocks in lines 211–222 catch any exceptions (**ArgumentException** or **XPathException**) that can occur if the user enters an invalid or empty expression.

```cpp
// Fig. 18.12: PathNavigatorTest.cpp
// Entry point for application.

#include "stdafx.h"
#include "Form1.h"
#include <windows.h>

using namespace PathNavigator;

int APIENTRY _tWinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPTSTR    lpCmdLine,
int       nCmdShow)
{
    System::Threading::Thread::CurrentThread->ApartmentState =
        System::Threading::ApartmentState::STA;
    Application::Run(new Form1());
    return 0;
} // end _tWinMain
```

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
<?xml version="1.0"?>

<!-- Fig. 18.13: sports.xml  -->
<!-- Sports Database         
Fig. 18.12 PathNavigator entry point. (Part 2 of 2.)

Fig. 18.13 XML document that describes various sports. (Part 1 of 2.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Method `DisplayIterator` (defined in lines 228–238 of Fig. 18.11) appends the node values from the given `XPathNodeIterator` to the `selectTreeViewer` text box. Note that we call the `String` method `Trim` to remove unnecessary white space. Method `MoveNext` (line 233) advances to the next node, which can be accessed via property `Current` (line 236).

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
18.5 Document Type Definitions (DTDs), Schemas and Validation

XML documents can reference optional documents that specify how the XML documents should be structured. These optional documents are called Document Type Definitions (DTDs) and Schemas. When a DTD or Schema document is provided, some parsers (called validating parsers) can read the DTD or Schema and check the XML document’s structure against it. If the XML document conforms to the DTD or Schema, then the XML document is valid. Parsers that cannot check for document conformity against the DTD or Schema are called non-validating parsers. If an XML parser (validating or non-validating) is able to process an XML document (that does not reference a DTD or Schema), the XML document is considered to be well formed (i.e., it is syntactically correct). By definition, a valid XML document is also a well-formed XML document. If a document is not well formed, parsing halts, and the parser issues an error.

Software Engineering Observation 18.2

DTD and Schema documents are essential components for XML documents used in business-to-business (B2B) transactions and mission-critical systems. These documents help ensure that XML documents are valid.

Software Engineering Observation 18.3

XML document content can be structured in many different ways, so an application cannot determine whether the document it receives is complete, is missing data or is ordered properly. DTDs and Schemas solve this problem by providing an extensible means of describing a document’s contents. An application can use a DTD or Schema document to perform a validity check on the document’s contents.

18.5.1 Document Type Definitions

Document type definitions (DTDs) provide a means for type-checking XML documents and thus for verifying their validity (confirming that elements contain the proper attributes, elements are in the proper sequence, etc.). DTDs use EBNF (Extended Backus-Naur Form) grammar to describe an XML document’s content. XML parsers need additional functionality to read EBNF grammar, because it is not XML syntax. Although DTDs are optional, they can be used to ensure document conformity. The DTD in Fig. 18.15 defines the set of rules (i.e., the grammar) for structuring the business letter document contained in Fig. 18.16.

Portability Tip 18.2

DTDs can ensure consistency among XML documents generated by different programs.

```xml
<!ELEMENT letter ( contact+, salutation, paragraph+, closing, signature )>

<!ELEMENT contact ( name, address1, address2, city, state, zip, phone, flag )>
```

Fig. 18.15 Document Type Definition (DTD) for a business letter. (Part 1 of 2.)
Lines 4–5 use the `ELEMENT` element type declaration to define rules for element `letter`. In this case, `letter` contains one or more `contact` elements, one `salutation` element, one or more `paragraph` elements, one `closing` element and one `signature` element, in that sequence. The `plus sign` (+) occurrence indicator specifies that an element must occur one or more times. Other indicators include the `asterisk` (*), which indicates an optional element that can occur any number of times, and the `question mark` (?), which indicates an optional element that can occur at most once. If an occurrence indicator is omitted, exactly one occurrence is expected.

The `contact` element declaration (lines 7–8) specifies that it contains the `name`, `address1`, `address2`, `city`, `state`, `zip`, `phone` and `flag` elements—in that order. Exactly one occurrence of each is expected.

Line 9 uses the `ATTLIST` attribute-list declaration to define an attribute (i.e., `type`) for the `contact` element. Keyword `#IMPLIED` specifies that, if the parser finds a `contact` element without a `type` attribute, the application can provide a value or ignore the missing attribute. The absence of a `type` attribute cannot invalidate the document. Other types of default values include `#REQUIRED` and `#FIXED`. Keyword `#REQUIRED` specifies that the attribute must be present in the document; keyword `#FIXED` specifies that the attribute (if present) must always be assigned a specific value. For example,

```
<!ATTLIST address zip #FIXED "01757”>
```

indicates that the value 01757 must be used for attribute `zip`; otherwise, the document is invalid. If the attribute is not present, then the parser, by default, uses the fixed value that is specified in the `ATTLIST` declaration. Flag `CDATA` specifies that attribute `type` contains a `String` that is not processed by the parser, but instead is passed to the application as is.

**Software Engineering Observation 18.4**

`DTD` syntax does not provide any mechanism for describing an element’s (or attribute’s) data type.

Flag `#PCDATA` (line 11) specifies that the element can store parsed character data (i.e., text). Parsed character data cannot contain markup. The characters `less than` (<) and `amper-`
sand (&) must be replaced by their entities (i.e., &lt; and &amp;). See Appendix G, XHTML Special Characters, for a list of predefined entities.

Line 18 declares an empty element named flag. Keyword EMPTY specifies that the element cannot contain character data. Empty elements commonly are used for their attributes.

Line 19 presents an enumerated attribute type, which declare a list of possible values an attribute can have. The attribute must be assigned a value from this list to conform to the DTD. Enumerated type values are separated by pipe characters (|). Line 19 contains an enumerated attribute type declaration that allows attribute gender to have either the value M or F. A default value of "M" is specified to the right of the element attribute type.

**Common Programming Error 18.9**

Any element, attribute or relationship not explicitly declared by a DTD results in an invalid document.

XML documents must explicitly reference a DTD against which they are going to be validated. Figure 18.16 is an XML document that conforms to letter.dtd (Fig. 18.15).

```xml
<?xml version = "1.0"?>
<!-- Fig. 18.16: letter2.xml -->
<!-- Business letter formatted with XML -->
<!DOCTYPE letter SYSTEM "letter.dtd">
<letter>
  <contact type = "from">
    <name>Jane Doe</name>
    <address1>Box 12345</address1>
    <address2>15 Any Ave.</address2>
    <city>Othertown</city>
    <state>Otherstate</state>
    <zip>67890</zip>
    <phone>555-4321</phone>
    <flag gender = "F" />
  </contact>

  <contact type = "to">
    <name>John Doe</name>
    <address1>123 Main St.</address1>
    <address2></address2>
    <city>Anytown</city>
    <state>Anystate</state>
    <zip>12345</zip>
    <phone>555-1234</phone>
    <flag gender = "M" />
  </contact>

  <salutation>Dear Sir:</salutation>
  <paragraph>It is our privilege to inform you about our new database managed with XML. This new system</paragraph>
</letter>
```

Fig. 18.16  XML document referencing its associated DTD. (Part 1 of 2.)
This XML document is similar to that in Fig. 18.3. Line 6 references a DTD file. This markup contains three pieces: the name of the root element (letter in line 8) to which the DTD is applied, the keyword SYSTEM (which in this case denotes an external DTD—a DTD defined in a separate file) and the DTD’s name and location (i.e., letter.dtd in the current directory). Though almost any file extension can be used, DTD documents typically end with the .dtd extension.

Various tools (many of which are free) check document conformity against DTDs and Schemas (discussed momentarily). Figure 18.17 shows the results of the validation of letter2.xml using Microsoft’s XML Validator. Visit www.w3.org/XML/Schema.html for a list of validating tools. Microsoft’s XML Validator is available free for download from msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp.

![XML Validator window](image)

**Fig. 18.17** XML Validator validates an XML document against a DTD.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Microsoft’s XML Validator can validate XML documents against DTDs locally or by uploading the documents to the XML Validator Web site. Here, letter2.xml and letter.dtd are placed in folder C:\XML\. This XML document (letter2.xml) is well formed and conforms to letter.dtd.

XML documents that fail validation might still be well-formed documents. When a document fails to conform to a DTD or Schema, Microsoft XML Validator displays an error message. For example, the DTD in Fig. 18.15 indicates that the contacts element must contain child element name. If the document omits this child element, the document is well formed, but not valid. In such a scenario, Microsoft XML Validator displays the error message shown in Fig. 18.18.

MC++ programs can use MSXML to validate XML documents against DTDs. For information on how to accomplish this, visit:


Schemas are the preferred means of defining structures for XML documents in .NET. Although several types of Schema exist, the two most popular are Microsoft Schema and W3C Schema. We begin our discussion of Schemas in the next section.

18.5.2 Microsoft XML Schema

In this section, we introduce an alternative to DTDs—called Schemas—for defining an XML document’s structure. Many developers in the XML community feel that DTDs are not flexible enough to meet today’s programming needs. For example, DTDs cannot be manipulated (e.g., searched, programmatically modified, etc.) in the same manner that XML documents can, because DTDs are not XML documents. Furthermore, DTDs do not provide features for describing an element’s (or attribute’s) data type.

Unlike DTDs, Schemas do not use Extended Backus-Naur Form (EBNF) grammar. Instead, Schemas are XML documents which can be manipulated (e.g., elements can be added or removed, etc.) like any other XML document. As with DTDs, Schemas require validating parsers.

3. W3C Schema, which we discuss in Section 18.5.3, is emerging as the industry standard for describing an XML document’s structure. Within the next two years, we expect most developers will be using W3C Schema.
In this section, we focus on Microsoft’s XML Schema vocabulary. Figure 18.19 presents an XML document that conforms to the Microsoft Schema document shown in Fig. 18.20. By convention, Microsoft XML Schema documents use the file extension .xdr, which is short for XML-Data Reduced. Line 6 (Fig. 18.19) references the Schema document book.xdr. A document using a Microsoft XML Schema uses attribute xmlns to reference its schema through a URI which begins with x-schema followed by a colon (:) and the name of the schema document.

```
<?xml version = "1.0"?>
<!-- Fig. 18.19: bookxdr.xml -->
<!-- XML file that marks up book data. -->
<books xmlns = "x-schema:book.xdr">
  <book>
  </book>
  <book>
    <title>C# for Experienced Programmers</title>
  </book>
  <book>
    <title>Visual Basic .NET for Experienced Programmers</title>
  </book>
  <book>
    <title>Java Web Services for Experienced Programmers</title>
  </book>
  <book>
    <title>Web Services: A Technical Introduction</title>
  </book>
</books>
```

Fig. 18.19 XML document that conforms to a Microsoft Schema document.

```
<?xml version = "1.0"?>
<!-- Fig. 18.20: book.xdr -->
<!-- Schema document to which book.xml conforms. -->
<Schema xmlns = "urn:schemas-microsoft-com:xml-data">
  <ElementType name = "title" content = "textOnly" model = "closed"/>
</Schema>
```

Fig. 18.20 Microsoft Schema file that contains the structure to which bookxdr.xml conforms. (Part 1 of 2.)
Chapter 18 Extensible Markup Language (XML) 887

10 <ElementType name = "book" content = "eltOnly" model = "closed">
11     <element type = "title" minOccurs = "1" maxOccurs = "1" />
12 </ElementType>
13 <ElementType name = "books" content = "eltOnly" model = "closed">
14     <element type = "book" minOccurs = "0" maxOccurs = "*" />
15 </ElementType>
16 </Schema>

Fig. 18.20 Microsoft Schema file that contains the structure to which bookxdr.xml conforms. (Part 2 of 2.)

Software Engineering Observation 18.5

Schemas are XML documents that conform to DTDs, which define the structure of a Schema. These DTDs, which are bundled with the parser, are used to validate the Schemas that authors create.

Software Engineering Observation 18.6

Many organizations and individuals are creating DTDs and Schemas for a broad range of categories (e.g., financial transactions, medical prescriptions, etc.). Often, these collections—called repositories—are available free for download from the Web.4

In line 6 of Fig. 18.20, root element Schema begins the Schema markup. Microsoft Schemas use the namespace URI "urn:schemas-microsoft-com:xml-data". Lines 7–8 use element ElementType to define element title. Attribute content specifies that this element contains parsed character data (i.e., text only). Setting the model attribute to "closed" specifies that the element can only contain elements defined in the specified Schema. Line 10 defines element book; this element’s content is “elements only” (i.e., eltOnly). This means that the element cannot contain mixed content (i.e., text and other elements). Within the ElementType element named book, the element element indicates that title is a child element of book. Attributes minOccurs and maxOccurs are set to "1", indicating that a book element must contain exactly one title element. The asterisk (*) in line 15 indicates that the Schema permits any number of book elements in element books. We discuss how to validate bookxdr.xml against book.xdr in Section 18.5.4.

18.5.3 W3C XML Schema5

In this section, we focus on W3C XML Schema6—the schema that the W3C created. W3C XML Schema is a Recommendation (i.e., a stable release suitable for use in industry). Figure 18.21 shows a Schema-valid XML document named bookxsd.xml and Fig. 18.22 shows the W3C XML Schema document (book.xsd) that defines the structure for bookxsd.xml. Although Schema authors can use virtually any filename extension, W3C XML Schemas typically use the .xsd extension. We discuss how to validate bookxsd.xml against book.xsd in the next section.

4. See, for example, opengis.net/schema.htm.
5. We provide a detailed treatment of W3C Schema in XML for Experienced Programmers (late 2003).

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
W3C XML Schemas use the namespace URI http://www.w3.org/2001/XMLSchema and often use namespace prefix xsd (line 6 in Fig. 18.22). Root element schema (line 6) contains elements that define the XML document’s structure. Line 7 binds the URI http://www.deitel.com/booklist to namespace prefix deitel. Line 8 specifies the targetNamespace, which is the namespace for elements and attributes that this schema defines.

In W3C XML Schema, element element (line 10) defines an element. Attributes name and type specify the element’s name and data type, respectively. In this case, the name of the element is books and the data type is deitel:BooksType. Any element (e.g.,
books) that contains attributes or child elements must define a complex type, which defines each attribute and child element. Type deitel:BooksType (lines 12–17) is an example of a complex type. We prefix BooksType with deitel, because this is a complex type that we have created, not an existing W3C XML Schema complex type.

Lines 12–17 use elementcomplexType to define type BooksType (used in line 10). Here, we define BooksType as an element type that has a child element named book. Because book also contains a child element, its type must be a complex type (e.g., BookType, which we define later). Attribute minOccurs specifies that books must contain a minimum of one book element. Attribute maxOccurs, having value unbounded (line 15), specifies that books may have any number of book child elements. Element sequence specifies the order of elements in the complex type.

Lines 19–23 define the complexType BookType. Line 21 defines element title with type xsd:string. When an element has a simple type such as xsd:string, it is prohibited from containing attributes and child elements. W3C XML Schema provides a large number of data types, such as xsd:date for dates, xsd:int for integers, xsd:double for floating-point numbers and xsd:time for time.

**Good Programming Practice 18.1**

By convention, W3C XML Schema authors use namespace prefixes xsd or xs when referring to the URI http://www.w3.org/2001/XMLSchema.

### 18.5.4 Schema Validation in Visual C++ .NET

In this section, we present an MC++ application (Fig. 18.23–Fig. 18.24) that uses classes from the .NET Framework Class Library to validate the XML documents presented in the last two sections against their respective Schemas. We use an instance of XmlValidatingReader to perform the validation.

```cpp
// Fig. 18.23: Form1.h
// Validating XML documents against Schemas.

#pragma once

namespace ValidationTest
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::Xml;
    using namespace System::Windows::Forms;
    using namespace System::Xml::Schema; // contains Schema classes

    /// <summary>
    /// Summary for Form1
    /// </summary>
```

**Fig. 18.23** Schema-validation example. (Part 1 of 4.)
/// WARNING: If you change the name of this class, you will need to
/// change the 'Resource File Name' property for the managed
/// resource compiler tool associated with all .resx files
/// this class depends on. Otherwise, the designers will not
/// be able to interact properly with localized resources
/// associated with this form.
/// </summary>
public __gc class Form1 : public System::Windows::Forms::Form
{
public:
Form1(void)
{
    InitializeComponent();
    valid = true;  // assume document is valid

    // get Schema(s) for validation
    schemas = new XmlSchemaCollection();
    schemas->Add( S"book", S"book.xdr" );
    schemas->Add( S"http://www.deitel.com/booklist", S"book.xsd" );
}

protected:
void Dispose(Boolean disposing)
{
    if (disposing && components)
    {
        components->Dispose();
    }
    __super::Dispose(disposing);
}
private: System::Windows::Forms::Button * validateButton;
private: System::Windows::Forms::Label * consoleLabel;
private: System::Windows::Forms::ComboBox * filesComboBox;
private: XmlSchemaCollection *schemas;  // Schemas collection
private: bool valid;                    // validation result
private:
/// <summary>
/// Required designer variable.
/// </summary>
System::ComponentModel::Container * components;

// Visual Studio .NET generated GUI code
// handle validateButton click event
private: System::Void validateButton_Click(System::Object * sender, System::EventArgs * e)
{
    try 
    {

Fig. 18.23  Schema-validation example. (Part 2 of 4.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
// get XML document
XmlTextReader *reader =
    new XmlTextReader( filesComboBox->Text );

// get validator
XmlValidatingReader *validator =
    new XmlValidatingReader( reader );

// assign Schema(s)
validator->Schemas->Add( schemas );

// set validation type
validator->ValidationType = ValidationType::Auto;

// register event handler for validation error(s)
validator->ValidationEventHandler +=
    new ValidationEventHandler( this, ValidationError );

// validate document node-by-node
while ( validator->Read() ) ; // empty body

// check validation result
if ( valid )
    consoleLabel->Text = S"Document is valid";

valid = true; // reset variable

// close reader stream
validator->Close();
}

private:

    // event handler for validation error
    void ValidationError( Object *sender,
        ValidationEventArgs *arguments )
    {
        consoleLabel->Text = arguments->Message;
    }

Fig. 18.23 Schema-validation example. (Part 3 of 4.)
valid = false; // validation failed
} // end method ValidationException
valid = true; // validation passed
} // end method ValidationError

Fig. 18.23 Schema-validation example. (Part 4 of 4.)

Fig. 18.24 ValidationTest entry point.

Line 39 (Fig. 18.23) creates an XmlSchemaCollection pointer named schemas. Line 40 calls method Add to add an XmlSchema object to the Schema collection. Method Add is passed a name that identifies the Schema (i.e., "book") and the name of the Schema file (i.e., "book.xdr"). Line 41 calls method Add to add a W3C XML Schema. The first argument specifies the namespace URI (i.e., line 8 of Fig. 18.22) and the second argument identifies the schema file (i.e., "book.xsd"). This is the Schema that is used to validate book.xsd.xml.

Lines 75–76 create an XmlTextReader for the file that the user selected from filesComboBox. The XML document to be validated against a Schema contained in the XmlSchemaCollection must be passed to the XmlValidatingReader constructor (79–80). The catch blocks in lines 106–117 handle any exceptions (ArgumentException or IOException) that can occur if the user enters no filename or an invalid filename in filesComboBox.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Line 83 adds the Schema collection pointed to by schemas to the Schemas property. This property sets the Schema used to validate the document. The ValidationType property (line 86) is set to the ValidationType enumeration constant for automatically identifying the Schema’s type (i.e., XDR or XSD). Lines 89–90 register method ValidationEvent with ValidationEventHandler. Method ValidationEvent (lines 123–128) is called if the document is invalid or if an error occurs, such as if the document cannot be found. Failure to register a method with ValidationEventHandler causes an exception to be thrown when the document is missing or invalid.

Validation is performed node-by-node, by calling the method Read (line 93). Each call to Read validates the next node in the document. The loop terminates either when all nodes have been validated successfully (and valid is still true) or a node fails validation (and valid has been set to false in line 127). When validated against their respective Schemas, the XML documents in Fig. 18.19 and Fig. 18.21 validate successfully.

Figure 18.25 and Fig. 18.26 list two XML documents that fail to conform to book.xdr and book.xsd, respectively. In both documents, an extra title element within a book element invalidates the documents. In Figure 18.25 and Figure 18.26, the extra elements are found in lines 9 and 21, respectively. Although both documents are invalid, they are well formed.

```xml
<?xml version = "1.0"?>
<!-- Fig. 18.25: bookxsdfail.xml -->
<!-- Document that does not conforms to W3C Schema. -->
<deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
  <book>
    <title>Java Web Services for Experienced Programmers</title>
    <title>C# for Experienced Programmers</title>
  </book>
  <book>
    <title>Visual C++ .NET: A Managed Code Approach</title>
  </book>
</deitel:books>
```

Fig. 18.25 XML document that does not conform to the XSD schema of Fig. 18.22.

```xml
<?xml version = "1.0"?>
<!-- Fig. 18.26: bookxdrfail.xml -->
<book>
  <title>C# for Experienced Programmers</title>
</book>
```

Fig. 18.26 XML file that does not conform to the Schema in Fig. 18.20. (Part 1 of 2.)
18.6 Extensible Stylesheet Language and XSLTransform

Extensible Stylesheet Language (XSL) is an XML vocabulary for formatting XML data. In this section, we discuss the portion of XSL that creates formatted text-based documents (including other XML documents) from XML documents. This process is called a transformation and involves two tree structures: the source tree, which is the XML document being transformed, and the result tree, which is the result (i.e., any text-based format such as XHTML or XML) of the transformation. The source tree is not modified when a transformation occurs.

To perform transformations, an XSLT processor is required. Popular XSLT processors include Microsoft’s MSXML and the Apache Software Foundation’s Xalan.

7. Extensible HyperText Markup Language (XHTML) is the W3C technical recommendation that replaces HTML for marking up content for the Web. For more information on XHTML, see the XHTML Appendices E and F, and visit www.w3.org.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
The XML document, shown in Fig. 18.27, is transformed by MSXML into an XHTML document (Fig. 18.28).

```xml
<?xml version = "1.0"?>
<!-- Fig. 18.27: sorting.xml -->
<!-- Usage of elements and attributes. -->
<?xml:stylesheet type = "text/xsl" href = "sorting.xsl"?>
<book isbn = "999-99999-9-X">
  <title>Deitel's XML Primer</title>
  <author>
    <firstName>Paul</firstName>
    <lastName>Deitel</lastName>
  </author>
  <chapters>
    <frontMatter>
      <preface pages = "2"/>
      <contents pages = "5"/>
      <illustrations pages = "4"/>
    </frontMatter>
    <chapter number = "3" pages = "44">
      Advanced XML</chapter>
    <chapter number = "2" pages = "35">
      Intermediate XML</chapter>
    <chapter number = "B" pages = "26">
      Parsers and Tools</chapter>
    <chapter number = "A" pages = "7">
      Entities</chapter>
    <chapter number = "1" pages = "28">
      XML Fundamentals</chapter>
  </chapters>
  <media type = "CD"/>
</book>
```

Fig. 18.27 XML document containing book information.

Line 6 is a processing instruction (PI), which contains application-specific information that is embedded into the XML document. In this particular case, the processing instruction is specific to IE and specifies the location of an XSLT document with which to transform the XML document. The characters <? and ?> delimit a processing instruction, which consists of a PI target (e.g., xml:stylesheet) and PI value (e.g., type=",text/xsl" href="sorting.xsl"). The portion of this particular PI value that follows href specifies the name and location of the style sheet to apply—in this case, sorting.xsl, which is located in the same directory as this XML document.

Fig. 18.28 presents the XSLT document (sorting.xsl) that transforms sorting.xml (Fig. 18.27) to XHTML.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Performance Tip 18.1

Using Internet Explorer on the client to process XSLT documents conserves server resources by using the client’s processing power (instead of having the server process XSLT documents for multiple clients).

Line 1 of Fig. 18.28 contains the XML declaration. Recall that an XSL document is an XML document. Lines 6–7 contain the xsl:stylesheet root element. Attribute version specifies the version of XSLT to which this document conforms. Namespace prefix xsl is defined and is bound to the XSLT URI defined by the W3C. When processed, lines 10–13 write the document type declaration to the result tree. Attribute method is assigned "xml", which indicates that XML is being output to the result tree. Attribute omit-xml-declaration is assigned "no", which outputs an XML declaration to the result tree. Attribute doctype-system and doctype-public write the Doctype DTD information to the result tree.

```xml
<?xml version = "1.0"?>
<!-- Fig. 18.28: sorting.xsl -->
<!-- Transformation of book information into XHTML. -->
<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

    <!-- write XML declaration and DOCTYPE DTD information -->
    <xsl:output method = "xml"
        omit-xml-declaration = "no"
        doctype-system = "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
        doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>

    <!-- match document root -->
    <xsl:template match = "/">
        <html xmlns = "http://www.w3.org/1999/xhtml">
            <xsl:apply-templates/>
        </html>
    </xsl:template>

    <!-- match book -->
    <xsl:template match = "book">
        <head>
            <title>ISBN <xsl:value-of select = "@isbn" /> -
                <xsl:value-of select = "title" /></title>
        </head>
        <body>
            <h1 style = "color: blue">
                <xsl:value-of select = "title"/>
            </h1>
    </xsl:template>
</xsl:stylesheet>
```

Fig. 18.28 XSL document that transforms sorting.xml (Fig. 18.27) into XHTML. (Part 1 of 3.)
Fig. 18.28 XSL document that transforms sorting.xml (Fig. 18.27) into XHTML.
(Part 2 of 3.)
XSLT documents contain one or more `xsl:template` elements that specify which information is output to the result tree. The template in line 16 matches the source tree’s document root. When the document root is encountered, this template is applied, and any text marked up by this element that is not in the namespace pointed to by `xsl` is output to the result tree. Line 18 calls for all the templates that match children of the document root to be applied. Line 23 specifies a template that matches element `book`.


Lines 33–35 create a header element that contains the book’s author. The context node (i.e., the current node being processed) is `book`, so the XPath expression `author/lastName` selects the author’s last name, and the expression `author/firstName` selects the author’s first name.

Line 40 selects each element that is a child of element `frontMatter` (indicated by an asterisk). Line 43 calls `node-set function name` to retrieve the current node’s element name (e.g., `preface`). The current node is the context node specified in the `xsl:for-each` (line 40).

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Lines 53–54 sort chapters by number in ascending order. Attribute select selects the value of context node chapter’s attribute number. Attribute data-type, having value "number", specifies a numeric sort, and attribute order specifies "ascending" order. Attribute data-type also can be assigned the value "text" (line 67), and attribute order also may be assigned the value "descending".

Lines 82–83 use an XSL variable to store the value of the book’s page count and output it to the result tree. Attribute name specifies the variable’s name, and attribute select assigns it a value. Function sum totals the values for all page attribute values. The two slashes between chapters and * indicate that all descendant nodes of chapters are searched for elements that contain an attribute named pages.

The System::Xml::Xsl namespace provides classes for applying XSLT style sheets to XML documents. Specifically, an object of class XslTransform can be used to perform the transformation.

Figure 18.29–Fig. 18.30 apply a style sheet (sports.xsl) to sports.xml (Fig. 18.13). The transformation result is written to a text box and to a file. We also load the results in IE to show the transformation results rendered as HTML.
Form1(void)
{
    InitializeComponent();
    // load XML data
    document = new XmlDocument();
document->Load("sports.xml");
    // create navigator
    navigator = document->CreateNavigator();
    // load style sheet
    transformer = new XslTransform();
    transformer->Load("sports.xsl");
}

protected:
    void Dispose(Boolean disposing)
    {
        if (disposing && components)
        {
            components->Dispose();
        }
        __super::Dispose(disposing);
    }

private: System::Windows::Forms::TextBox * consoleTextBox;
private: System::Windows::Forms::Button * transformButton;
private: XmlDocument *document;     // Xml document root
private: XPathNavigator *navigator; // navigate document
private: StringWriter *output;      // display document

private:
    /// <summary>
    /// Required designer variable.
    /// </summary>
    System::ComponentModel::Container * components;

    // Visual Studio .NET generated GUI code
    // transformButton click event
    private: System::Void transformButton_Click(System::Object * sender, System::EventArgs * e)
    {
        // transform XML data
        output = new StringWriter();
        transformer->Transform(navigator, 0, output);
        // display transformation in text box
        consoleTextBox->Text = output->ToString();
    }

Fig. 18.29 XSL style sheet applied to an XML document. (Part 2 of 3.)
Chapter 18  
Extensible Markup Language (XML)  

Fig. 18.30  Form1.cpp  
// Entry point for application.   

#include "stdafx.h"  
#include "Form1.h"  
#include <windows.h>  
using namespace TransformTest;  

int APIENTRY _tWinMain(HINSTANCE hInstance,  
HINSTANCE hPrevInstance,  
LPTSTR lpszCmdLine,  
int nCmdShow)  
{  
System::Threading::Thread::CurrentThread->ApartmentState =  
System::Threading::ApartmentState::STA;  
Application::Run(new Form1());  
return 0;  
}  // end _tWinMain

// write transformation result to disk  
FileStream *stream = new FileStream( L"sports.html", FileMode::Create );  
StreamWriter *writer = new StreamWriter( stream );  
writer->Write( output->ToString() );  
// close streams  
writer->Close();  
output->Close();  
} // end method transformButton_Click

Fig. 18.29  XSL style sheet applied to an XML document. (Part 3 of 3.)

Fig. 18.30  TransformTest entry point. (Part 1 of 2.)
Fig. 18.30 TransformTest entry point. (Part 2 of 2.)

Line 63 (Fig. 18.29) declares XslTransform pointer transformer. An object of this type is necessary to transform the XML data to another format. In line 39, the XML document is parsed and loaded into memory with a call to method Load. Method CreateNavigator is called in line 42 to create an XPathNavigator object, which is used to navigate the XML document during the transformation. A call to method Load of class XslTransform (line 46) parses and loads the style sheet that this application uses. The argument that is passed contains the name and location of the style sheet.

Event handler transformButton_Click (lines 75–94) calls method Transform of class XslTransform to apply the style sheet (sports.xsl) to sports.xml (line 80). This method takes three arguments: An XPathNavigator (created from sports.xml's XmlDocument), an instance of class XsltArgumentList, which is a list of String * parameters that can be applied to a style sheet—0, in this case—and an instance of a derived class of TextWriter (in this example, an instance of class StringWriter). The results of the transformation are stored in the StringWriter object pointed to by output. Lines 86–89 write the transformation results to disk. The third screen shot depicts the created XHTML document when it is rendered in IE.

In this chapter, we studied the Extensible Markup Language and several of its related technologies. In Chapter 19, Database, SQL and ADO .NET, we begin our discussion of databases, which are crucial to the development of multi-tier Web-based applications.

18.7 Internet and Web Resources

www.w3.org/xml
The W3C (World Wide Web Consortium) facilitates the development of common protocols to ensure interoperability on the Web. Their XML page includes information about upcoming events, publications, software and discussion groups. Visit this site to read about the latest developments in XML.

www.w3.org/TR/REC-xml
This W3C page contains a short introduction to XML as well as the most recent XML specification.

www.w3.org/XML/1999/XML-in-10-points
This W3C page describes the basics of XML in ten simple points. It is a useful page for those new to XML.
xml.org is a reference for XML, DTDs, schemas and namespaces.

www.w3.org/style/XSL
This W3C page provides information on XSL, including such topics as XSL development, learning XSL, XSL-enabled tools, XSL specification, FAQs and XSL history.

www.w3.org/TR
This is the W3C technical reports and publications page. It contains links to working drafts, proposed recommendations and other resources.

www.xmlbooks.com
This site provides a list of XML books recommended by Charles Goldfarb, one of the original designers of GML (General Markup Language), from which SGML is derived.

www.xml-zone.com
The Development Exchange XML Zone is a complete resource for XML information. This site includes an FAQ, news, articles and links to other XML sites and newsgroups.

wdvl.internet.com/Authoring/Languages/XML
Web Developer's Virtual Library XML site includes tutorials, an FAQ, the latest news and extensive links to XML sites and software downloads.

www.xml.com
XML.com provides the latest news and information about XML, conference listings, links to XML Web resources organized by topic, tools and other resources.

msdn.microsoft.com/xml/default.asp
The MSDN Online XML Development Center features articles on XML, Ask the Experts chat sessions, samples and demos, newsgroups and other helpful information.

msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp
Microsoft's XML validator, which can be downloaded from this site, can validate both online and offline documents.

www.oasis-open.org/cover/xml.html
The SGML/XML Web Page is an extensive resource that includes links to several FAQs, online resources, industry initiatives, demos, conferences and tutorials.

www-106.ibm.com/developerworks/xml
The IBM XML Zone site is a great resource for developers. It provides news, tools, a library, case studies and information about events and standards.

developer.netscape.com/tech/xml
The XML and Metadata Developer Central site has demos, technical notes and news articles related to XML.

www.ucc.ie/xml
This site is a detailed XML FAQ. Developers can check out responses to some popular questions or submit their own questions through the site.

**SUMMARY**

- XML is a widely supported, open (i.e., non-proprietary) technology for data exchange. XML is quickly becoming the standard by which applications maintain data.
- XML is highly portable. Any text editor that supports ASCII or Unicode characters can render or display XML documents. XML elements describe the data they contain, so they are both human and machine readable.
- XML permits document authors to create custom markup for virtually any type of information. This extensibility enables document authors to create entirely new markup languages, ones that
describe specific types of data, including mathematical formulas, chemical molecular structures, music, recipes, etc.

- The processing of XML documents—which programs typically store in files whose names end with the \texttt{.xml} extension—requires a program called an XML parser. A parser is responsible for identifying components of XML documents and for then storing those components in a data structure for manipulation.

- An XML document can reference another optional document that defines the XML document’s structure. Two types of optional structure-defining documents are Document Type Definitions (DTDs) and XML Schemas.

- An XML document begins with an optional XML declaration, which identifies the document as an XML document. The \texttt{version} information parameter specifies the version of XML syntax that is used in the document.

- XML comments begin with \texttt{<!--} and end with \texttt{-->}. Data is marked up with tags whose names are enclosed in angle brackets (\texttt{<>}). Tags are used in pairs to delimit markup. A tag that begins markup is called a start tag; a tag that terminates markup is called an end tag. End tags differ from start tags in that they contain a forward slash (/) character.

- Individual units of markup are called elements, which are the most fundamental XML building blocks. XML documents contain one element, called a root element, that contains every other element in the document. Elements are embedded or nested within each other to form hierarchies, with the root element at the top of the hierarchy.

- In addition to being placed between tags, data also can be placed in attributes, which are name–value pairs in start tags. Elements can have any number of attributes.

- XML allows document authors to create their own tags, so naming collisions can occur. As in the .NET Framework, XML namespaces provide a means for document authors to prevent collisions. Namespace prefixes are prepended to elements to specify the namespace to which the element belongs.

- Each namespace prefix is bound to a uniform resource identifier (URI) that uniquely identifies the namespace. Document authors create their own namespace prefixes. Virtually any name can be used as a namespace prefix except the reserved namespace prefix \texttt{xml}.

- To eliminate the need to place a namespace prefix in each element, document authors can specify a default namespace for an element and its children.

- When an XML parser successfully parses a document, the parser stores a tree structure containing the document’s data in memory. This hierarchical tree structure is called a Document Object Model (DOM) tree. The DOM tree represents each component of the XML document as a node in the tree. The DOM tree has a single root node that contains all other nodes in the document.

- Namespace \texttt{System::Xml} contains classes for creating, reading and manipulating XML documents.

- Class \texttt{XmlReader} is an abstract class that defines the interface for reading XML documents.

- \texttt{XmlReader}-derived class \texttt{XmlNodeReader} iterates through each node in the XML document.

- An \texttt{XmlDocument} object conceptually represents an empty XML document.

- Method \texttt{CreateNode} of \texttt{XmlDocument} takes a \texttt{NodeType}, a \texttt{Name} and a \texttt{NamespaceURI} as arguments.

- XML documents are parsed and loaded into an \texttt{XmlDocument} object when method \texttt{Load} is invoked. Once an XML document is loaded into an \texttt{XmlDocument}, its data can be read and manipulated programmatically.

- An \texttt{XmlNodeReader} allows us to read one node at a time from an \texttt{XmlDocument}.
Chapter 18  Extensible Markup Language (XML)  905

- Method Read of XmlReader reads one node from the DOM tree.
- The Name property contains the node’s name, the Value property contains the node’s data and the NodeType property contains the node type (i.e., element, comment, text, etc.).
- An XmlTextWriter streams XML data to a stream. Method WriteTo writes an XML representation to an XmlTextWriter stream.
- An XmlTextWriter reads XML data from a stream.
- Class XPathNavigator in the System::Xml::XPath namespace can iterate through node lists that match search criteria, written as an XPath expression.
- XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies.
- Navigation methods of XPathNavigator are MoveToFirstChild, MoveToParent, MoveToNext and MoveToPrevious.
- Whereas XML contains only data, XSLT is capable of converting XML into any text-based document (including another XML document). XSLT documents typically have the extension .xsl.
- When transforming an XML document via XSLT, two tree structures are involved: the source tree, which is the XML document being transformed, and the result tree, which is the result (e.g., XHTML) of the transformation.
- The node-set function name retrieves the current node’s element name.
- Attribute select selects the value of context node’s attribute.
- XML documents can be transformed programatically through MC++. The System::Xml::Xsl namespace facilitates the application of XSLT style sheets to XML documents.
- Class XsltArgumentList is a list of String parameters that can be applied to a style sheet.

**TERMINOLOGY**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ character</td>
<td>document root</td>
</tr>
<tr>
<td>Add method of class XmlSchemaCollection</td>
<td>Document Type Definition (DTD)</td>
</tr>
<tr>
<td>ancestor node</td>
<td>DOM (Document Object Model)</td>
</tr>
<tr>
<td>asterisk (*) occurrence indicator</td>
<td>EBNF (Extended Backus-Naur Form) grammar</td>
</tr>
<tr>
<td>ATTLIST</td>
<td>ELEMENT element type declaration</td>
</tr>
<tr>
<td>attribute</td>
<td>empty element</td>
</tr>
<tr>
<td>attribute node</td>
<td>EMPTY keyword</td>
</tr>
<tr>
<td>attribute value</td>
<td>end tag</td>
</tr>
<tr>
<td>CDATA character data</td>
<td>Extensible Stylesheet Language (XSL)</td>
</tr>
<tr>
<td>child element</td>
<td>external DTD</td>
</tr>
<tr>
<td>child node</td>
<td>forward slash</td>
</tr>
<tr>
<td>container element</td>
<td>#IMPLIED flag</td>
</tr>
<tr>
<td>context node</td>
<td>invalid document</td>
</tr>
<tr>
<td>CreateNavigator method of</td>
<td>IsEmptyElement property of</td>
</tr>
<tr>
<td>class XPathDocument</td>
<td>class XmlNodeReader</td>
</tr>
<tr>
<td>CreateNode method of class XmlDocument</td>
<td>LastChild property of class XmlNode</td>
</tr>
<tr>
<td>Current property of XPathNodeIterator</td>
<td>Load method of class XmlDocument</td>
</tr>
<tr>
<td>data-type attribute</td>
<td>markup</td>
</tr>
<tr>
<td>default namespace</td>
<td>match attribute</td>
</tr>
<tr>
<td>descendant node</td>
<td>maxOccurs attribute</td>
</tr>
<tr>
<td>doctype-public attribute</td>
<td>method attribute</td>
</tr>
<tr>
<td>doctype-system attribute</td>
<td>minOccurs attribute</td>
</tr>
</tbody>
</table>

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
MoveToFirstChild property of class XPathNavigator
MoveToNext property of XPathNavigator
MoveToParent property of class XPathNavigator
MoveToPrevious property of class XPathNavigator
MoveToRoot property of XPathNavigator
MSXML parser
name attribute
name node-set function
Name property of class XmlNodeReader
namespace prefix
class XmlNodeReader
node
Nodes collection
node-set function
NodeType property
occurrence indicator
omit-xml-declaration attribute
order attribute
parent node
Parent property of class TreeNode
#PCDATA flag
property
parsed character data
parser
 PI (processing instruction)
 PI target
 PI value
plus-sign (+) occurrence indicator
processing instruction
question-mark (?) occurrence indicator
Read method of XmlNodeReader
recursive descent
reserved namespace prefix xml
result tree
root element
root node
Schema element
schema property
Schemas property of XmlSchemaCollection
select attribute
Select method of class XPathNavigator
sibling node
single-quote character (')
source tree
style sheet
sum function
SYSTEM flag

System::Xml namespace
System::Xml::Schema namespace
text node
Transform method of class XslTransform
tree-based model
type attribute
validating XML parser
ValidatingReader class
ValidationEventHandler class
ValidationType property of class XmlValidatingReader
ValidationType::Auto constant
version property
version information parameter
W3C XML Schema
well-formed document
.XDR extension
XML (Extensible Markup Language)
XML declaration
.xml file extension
xml1 namespace
XML node
XML processor
XML Schema
XML Validator
Xml1Document class
Xml1NodeType enumeration
Comment constant
Element constant
EndElement constant
Text constant
Xml1NodeType::XmlDeclaration constant
xmlns attribute
XmlPathNavigator class
XPathNodeIterator class
class
xml namespace
XmlPathNavigator class
XmlNodeReader class
XmlNodeType::Comment
XmlNodeType::Element
XmlNodeType::EndElement
XmlNodeType::Text
XmlNodeType::XmlDeclaration
Xm1ns attribute
Xm1PathNavigator class
Xm1Reader class
Xm1Schema class
Xm1SchemaCollection collection
Xm1TextWriter class
XPathExpression class
.xsl extension
XSL Transformations (XSLT)
XSL variable
xsl:apply-templates element
xsl:for-each element
xsl:output element
xsl:sort element
xsl:stylesheet element
SELF-REVIEW EXERCISES

18.1 Which of the following are valid XML element names?
   a) yearBorn
   b) year.Born
   c) year Born
   d) year-Born1
   e) 2_year_born
   f) --year/born
   g) year*born
   h) .year_born
   i) _year_born_
   j) y_e-a_r-b_o-r_n

18.2 State whether the following are true or false. If false, explain why.
   a) XML is a technology for creating markup languages.
   b) XML markup is delimited by forward and backward slashes (/ and \\).
   c) All XML start tags must have corresponding end tags.
   d) Parsers check an XML document’s syntax.
   e) XML does not support namespaces.
   f) When creating new XML elements, document authors must use the set of XML tags provided by the W3C.
   g) The pound character (#), the dollar sign ($), ampersand (&), greater-than (>) and less-than (<) are examples of XML reserved characters.

18.3 Fill in the blanks for each of the following statements:
   a) ________ help prevent naming collisions.
   b) ________ embed application–specific information into an XML document.
   c) ________ is Microsoft’s XML parser.
   d) XSL element ________ writes a DOCTYPE to the result tree.
   e) Microsoft XML Schema documents have root element ________.
   f) To define an attribute in a DTD, ________ is used.
   g) XSL element ________ is the root element in an XSL document.
   h) XSL element ________ selects specific XML elements using repetition.

18.4 State which of the following statements are true and which are false. If false, explain why.
   a) XML is not case sensitive.
   b) Schemas are the preferred means of defining structures for XML documents in .NET.
   c) DTDs are a vocabulary of XML.
   d) Schema is a technology for locating information in an XML document.

18.5 In Fig. 18.1, we subdivided the author element into more detailed pieces. How might you subdivide the date element?

18.6 Write a processing instruction that includes the stylesheet wap.xsl for use in Internet Explorer.

18.7 Fill in the blanks in each of the following statements:
   a) Nodes that contain other nodes are called ________ nodes.
   b) Nodes that are peers are called ________ nodes.
   c) Class XmlDocument is analogous to the ________ of a tree.
   d) Method ________ adds an XmlNode to an XmlTree as a child of the current node.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
18.8 Write an XPath expression that locates contact nodes in letter.xml (Fig. 18.3).

18.9 Describe the Select method of XPathNavigator.

ANSWERS TO SELF-REVIEW EXERCISES

18.1 a, b, d, i, j. [Choice c is incorrect because it contains a space. Choice e is incorrect because the first character is a number. Choice f is incorrect because it contains a forward slash (/) and does not begin with a letter or underscore. Choice g is incorrect because it contains an asterisk (*); Choice h is incorrect because the first character is a period (.) and does not begin with a letter or underscore.]

18.2 a) True. b) False. In an XML document, markup text is delimited by angle brackets (< and >), with a forward slash in the end tag. c) True. d) True. e) False. XML does support namespaces. f) False. When creating new tags, document authors can use any valid name except the reserved word xml (also XML, Xm1 etc.). g) False. XML reserved characters include the ampersand (&), the left-angle bracket (<) and the right-angle bracket (>). but not # and $.

18.3 a) namespaces. b) processing instructions. c) MSXML. d) xsl:output. e) Schema. f) ATTLIST. g) xsl:stylesheet. h) xsl:for-each.

18.4 a) False. XML is case sensitive. b) True. c) False. DTDs use EBNF grammar which is not XML syntax. d) False. XPath is a technology for locating information in an XML document.

18.5 <date>
    <month>August</month>
    <day>6</day>
    <year>2003</year>
</date>

18.6 <?xsl:stylesheet type = "text/xsl" href = "wap.xsl"?>

18.7 a) parent. b) sibling. c) root. d) AppendChild.

18.8 /letter/contact.

18.9 Select takes either an XPathExpression or a String argument containing an XPathExpression to select nodes referenced by the navigator.

EXERCISES

18.10 Create an XML document that marks up the nutrition facts for a package of cookies. A package of cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 2 grams of saturated fat, 5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugars and 5 grams of protein. Name this document nutrition.xml. Load the XML document into Internet Explorer [Hint: Your markup should contain elements describing the product name, serving size/amount, calories, sodium, cholesterol, proteins, etc. Mark up each nutrition fact/ingredient listed above.

18.11 Write an XSLT style sheet for your solution to Exercise 18.10 that displays the nutritional facts in an XHTML table. Modify Fig. 18.29–Fig. 18.30 (application TransformTest) to output an XHTML file, nutrition.html. Render nutrition.html in a Web browser.

18.12 Write a Microsoft Schema for Fig. 18.27.

18.13 Alter Fig. 18.23–Fig. 18.24 (application ValidationTest) to include a list of Schemas in a drop-down box, along with the list of XML files. Allow the user to test for whether any XML file on the list satisfies a specific Schema. Use books.xml, books.xsd, nutrition.xml, nutrition.xsd and fail.xml.
Chapter 18  Extensible Markup Language (XML)  909

18.14 Modify XmlReaderTest (Fig. 18.7–Fig. 18.8) to display letter.xml (Fig. 18.3) in a TreeView, instead of in a text box.

18.15 Modify Fig. 18.28 (sorting.xsl) to sort each section (i.e., frontmatter, chapters and appendices) by page number rather than by chapter number. Save the modified document as sorting_byPage.xsl.

18.16 Modify TransformTest (Fig. 18.29–Fig. 18.30) to take in sorting.xml (Fig. 18.27), sorting.xsl (Fig. 18.28) and sorting_byPage.xsl, and print the XHTML document resulting from the transform of sorting.xml into two XHTML files, sorting_byPage.html and sorting_byPage.html.