15 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 **`XslTransform`**.
- To become familiar with BizTalk™.

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

Outline

15.1 Introduction

The *Extensible Markup Language* (XML) was developed in 1996 by the *World Wide Web Consortium's (W3C's) XML Working Group*. XML is a portable, widely supported, *open technology* (i.e., non-proprietary technology) for describing data. XML is becoming the standard for storing data that is exchanged between applications. Using XML, document authors can describe any type of data, including mathematical formulas, software-configuration instructions, music, recipes and financial reports. XML documents are readable by both humans and machines.

The .NET Framework uses XML extensively. The Framework Class Library (FCL) provides an extensive set of XML-related classes. Much of Visual Studio's internal implementation also employs XML. In this chapter, we introduce XML, XML-related technologies and key classes for creating and manipulating XML documents.

15.2 XML Documents

In this section, we present our first XML document, which describes an article (Fig. 15.1). [*Note:* The line numbers shown are not part of the XML document.]

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

<!-- Fig. 15.1: article.xml      -->
<!-- Article structured with XML -->

<article>

   <title>Simple XML</title>
```

Fig. 15.1 XML used to mark up an article. (Part 1 of 2.)

```
 9
10     <date>December 6, 2001</date>
11
12     <author>
13        <firstName>John</firstName>
14        <lastName>Doe</lastName>
15     </author>
16
17     <summary>XML is pretty easy.</summary>
18
19     <content>In this chapter, we present a wide variety of examples
20        that use XML.
21     </content>
22
23  </article>
```

Fig. 15.1 XML used to mark up an article. (Part 2 of 2.)

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 a C# program, comments are used in XML for documentation purposes.

Common Programming Error 15.1

The placement of any characters, including whitespace, before the XML declaration is an error.

Portability Tip 15.1

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 using *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*, whereas 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 15.2

Failure to provide a corresponding end tag for a start tag is an error.

Individual units of markup (i.e., everything included between a start tag and its corresponding end tag) are called *elements*. An XML document includes one element (called a *root element*) that contains every other element. The root element must be the first element after the XML declaration. In Fig. 15.1, **`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 15.3

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 15.4

XML is case sensitive. The use of the wrong case for an XML element name 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`***. The processing of XML documents requires a program called an *XML parser* also called *XML processors*. Parsers are responsible for checking an XML document's syntax and making the XML document's data available to applications. Often, XML parsers are built into applications such as Visual Studio or available for download over the Internet. Popular parsers include Microsoft's *msxml*, the Apache Software Foundation's *Xerces* and IBM's *XML4J*. In this chapter, we use msxml.

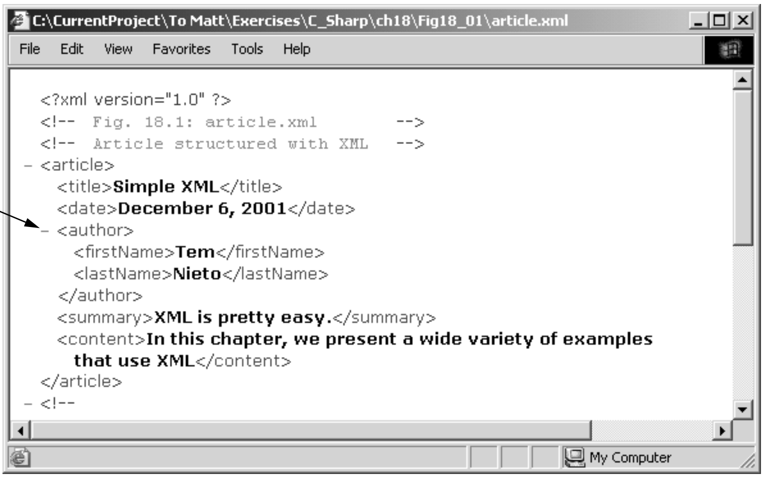
When the user loads **`article.xml`** into Internet Explorer (IE),[1] msxml parses the document and passes the parsed data to IE. IE then uses a built-in *style sheet* to format the data. Notice that the resulting format of the data (Fig. 15.2) is similar to the format of the XML document shown in Fig. 15.1. 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. 15.2. 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, a minus sign becomes a plus sign (which collapses the container element and hides all children). 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 15.4.

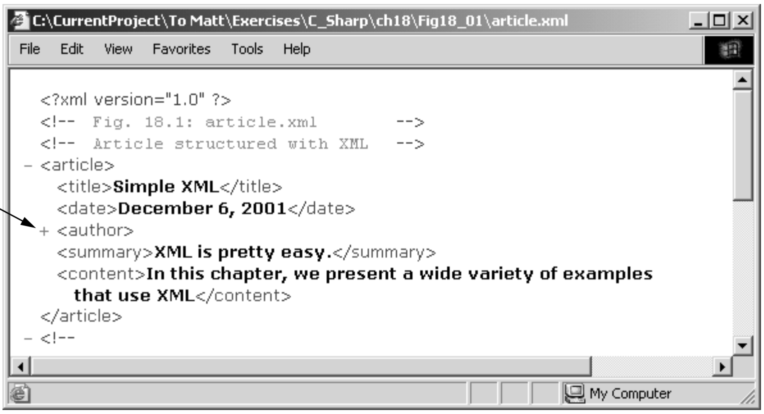
Common Programming Error 15.5

*Nesting XML tags improperly is a syntax error. For example, **`<x><y>hello</x></y>`** is a error, because the **`</y>`** tag must precede the **`</x>`** tag.*

1. IE 5 and higher.


C:\CurrentProject\To Matt\Exercises\C_Sharp\ch18\Fig18_01\article.xml
File Edit View Favorites Tools Help
Minus sign
My Computer

```
  <?xml version="1.0" ?>
  <!--  Fig. 18.1: article.xml        -->
  <!--  Article structured with XML   -->
- <article>
    <title>Simple XML</title>
    <date>December 6, 2001</date>
  - <author>
      <firstName>Tem</firstName>
      <lastName>Nieto</lastName>
    </author>
    <summary>XML is pretty easy.</summary>
    <content>In this chapter, we present a wide variety of examples
      that use XML</content>
  </article>
- <!--
```


C:\CurrentProject\To Matt\Exercises\C_Sharp\ch18\Fig18_01\article.xml
File Edit View Favorites Tools Help
Plus sign
My Computer

```
  <?xml version="1.0" ?>
  <!--  Fig. 18.1: article.xml        -->
  <!--  Article structured with XML   -->
- <article>
    <title>Simple XML</title>
    <date>December 6, 2001</date>
  + <author>
    <summary>XML is pretty easy.</summary>
    <content>In this chapter, we present a wide variety of examples
      that use XML</content>
  </article>
- <!--
```

Fig. 15.2 `article.xml` displayed by Internet Explorer.

We now present a second XML document (Fig. 15.3), which marks up a business letter. This document contains significantly more data than did the previous XML document.

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

<!-- Fig. 15.3: 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>
```

Fig. 15.3 XML to mark up a business letter. (Part 1 of 2.)

```
13          <zip>67890</zip>
14          <phone>555-4321</phone>
15          <flag gender = "F" />
16       </contact>
17
18       <contact type = "to">
19          <name>John Doe</name>
20          <address1>123 Main St.</address1>
21          <address2></address2>
22          <city>Anytown</city>
23          <state>Anystate</state>
24          <zip>12345</zip>
25          <phone>555-1234</phone>
26          <flag gender = "M" />
27       </contact>
28
29       <salutation>Dear Sir:</salutation>
30
31          <paragraph>It is our privilege to inform you about our new
32          database managed with <technology>XML</technology>. This
33          new system allows you to reduce the load on
34          your inventory list server by having the client machine
35          perform the work of sorting and filtering the data.
36          </paragraph>
37
38          <paragraph>Please visit our Web site for availability
39          and pricing.
40          </paragraph>
41
42       <closing>Sincerely</closing>
43
44       <signature>Ms. Doe</signature>
45    </letter>
```

Fig. 15.3 XML to mark up a business letter. (Part 2 of 2.)

Root element **letter** (lines 6–45) contains the child elements **contact** (lines 7–16 and 18–27), **salutation**, **paragraph** (lines 31–36 and 38–40), **closing** and **signature**. 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 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 are case sensitive, can be any length; may contain letters, digits, underscores, hyphens and periods; and must begin with either a letter or underscore character. A **contact** element stores a contact's name, address and phone number. 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 15.6

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

Common Programming Error 15.7

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>
```

15.3 XML Namespaces

Object-oriented programming languages, such as C# and Visual Basic .NET, provide massive class libraries that group their features 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 C#, 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 15.5), Extensible Stylesheet Language (Section 15.6) and BizTalk (Section 15.7)—often use namespaces to identify their elements.

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

```
<deitel:book>C# For Experienced Programmers</deitel:book>
```

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 15.8

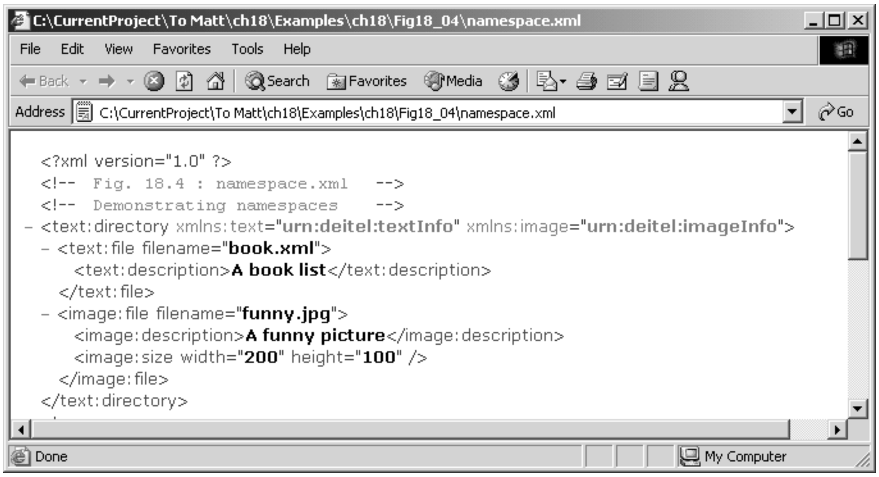
*Attempting to create a namespace prefix named **`xml`** in any mixture of case is a syntax error.*

The mark up in Fig. 15.4 demonstrates the use of namespaces. This XML document contains two **`file`** elements that are differentiated using namespaces.

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

<!-- Fig. 15.4: namespace.xml -->
<!-- Demonstrating namespaces -->
```

Fig. 15.4 XML namespaces demonstration. (Part 1 of 2.)

```
5
6  <text:directory xmlns:text = "urn:deitel:textInfo"
7     xmlns:image = "urn:deitel:imageInfo">
8
9     <text:file filename = "book.xml">
10       <text:description>A book list</text:description>
11    </text:file>
12
13    <image:file filename = "funny.jpg">
14       <image:description>A funny picture</image:description>
15       <image:size width = "200" height = "100" />
16    </image:file>
17
18 </text:directory>
```


C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_04\namespace.xml
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_04\namespace.xml
Go
Done
My Computer

```
  <?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. 15.4 XML namespaces demonstration. (Part 2 of 2.)

Software Engineering Observation 15.1

A programmer has the option of qualifying an attribute with a namespace prefix. However, it is not required, because attributes always are associated with elements.

Lines 6–7 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.

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 *Universal 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

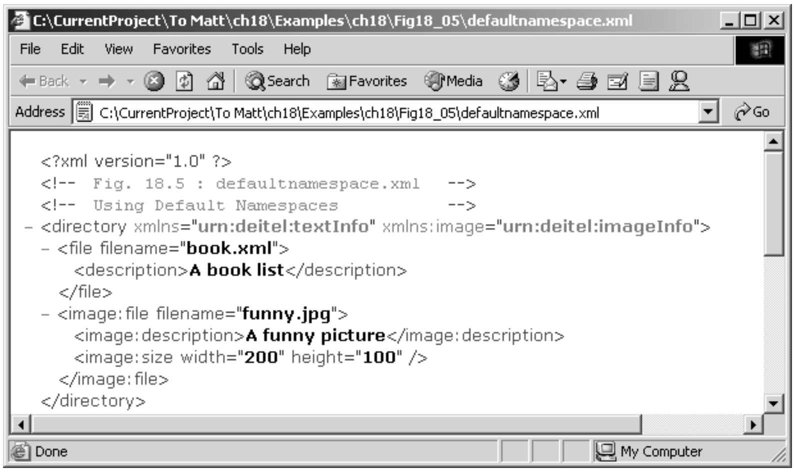
```
<text:directory xmlns:text =
   "http://www.deitel.com/xmlns-text"
   xmlns:image = "http://www.deitel.com/xmlns-image">
```

In this example, we use URLs related to the Deitel & Associates, Inc, domain name to identify namespaces. The 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 formed properly.

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 15.5 demonstrates the creation and use of default namespaces.

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

<!-- Fig. 15.5: defaultnamespace.xml -->
<!-- Using default namespaces         -->

<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>
```


C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_05\defaultnamespace.xml
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_05\defaultnamespace.xml Go
Done
My Computer

```
  <?xml version="1.0" ?>
  <!--  Fig. 18.5 : defaultnamespace.xml   -->
  <!--  Using Default Namespaces           -->
- <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>
```

Fig. 15.5 Default namespace demonstration.

Line 6 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. 15.4, 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.

15.4 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, some XML parsers store document data as tree structures in memory. Figure 15.6 illustrates the tree structure for the document **`article.xml`** discussed in Fig. 15.1. 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, elements, etc.

Classes for creating, reading and manipulating XML documents are located in the C# namespace ***`System.Xml`***. This namespace also contains additional namespaces that contain other XML-related operations.

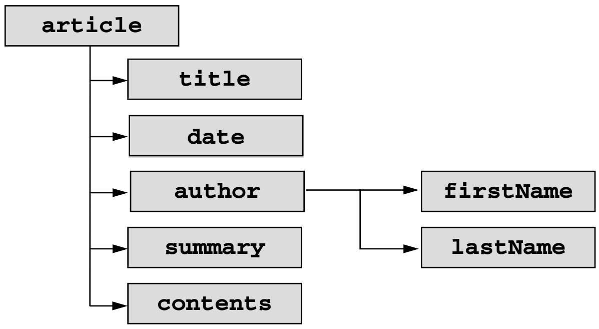


Fig. 15.6 Tree structure for Fig. 15.1.

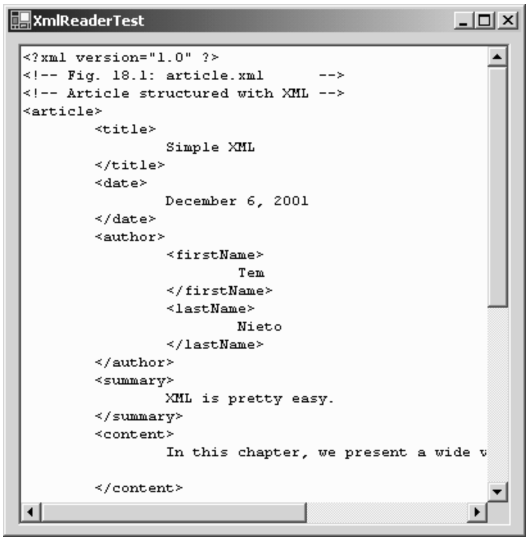
In this section, we present several examples that use DOM trees. Our first example, the program in Fig. 15.7, loads the XML document presented in Fig. 15.1 and displays its data in a text box. This example uses class ***XmlNodeReader*** which is derived from ***XmlReader***, which iterates through each node in the XML document. Class **XmlReader** is an **abstract** class that defines the interface for reading XML documents.

```
// Fig. 15.7: XmlReaderTest.cs
// Reading an XML document.

using System;
using System.Windows.Forms;
using System.Xml;

public class XmlReaderTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox outputTextBox;
   private System.ComponentModel.Container components = null;

   public XmlReaderTest()
   {
      InitializeComponent();

      // reference to "XML document"
      XmlDocument document = new XmlDocument();
      document.Load( "..\\..\\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.Text += "<" + reader.Name + ">" +
                  "\r\n";

```

Fig. 15.7 **XmlNodeReader** used to iterate through an XML document. (Part 1 of 3.)

```
                  // if empty element, decrease depth
                  if ( reader.IsEmptyElement )
                     depth--;

                  break;

               // if Comment, display it
               case XmlNodeType.Comment:
                  TabOutput( depth );
                  outputTextBox.Text +=
                     "<!--" + reader.Value + "-->\r\n";
                  break;

               // if Text, display it
               case XmlNodeType.Text:
                  TabOutput( depth );
                  outputTextBox.Text += "\t" + reader.Value +
                     "\r\n";
                  break;

               // if XML declaration, display it
               case XmlNodeType.XmlDeclaration:
                  TabOutput( depth );
                  outputTextBox.Text += "<?" + reader.Name + " "
                     + reader.Value + " ?>\r\n";
                  break;

               // if EndElement, display it and decrement depth
               case XmlNodeType.EndElement:
                  TabOutput( depth );
                  outputTextBox.Text += "</" + reader.Name
                     + ">\r\n";
                  depth--;
                  break;
            } // end switch statement
         } // end while loop
      } // End XmlReaderTest constructor

      // insert tabs
      private void TabOutput( int number )
      {
         for ( int i = 0; i < number; i++ )
            outputTextBox.Text += "\t";
      } // end TabOutput

      // Windows Form Designer generated code

      [STAThread]
      static void Main()
      {
         Application.Run( new XmlReaderTest() );
      } // end Main
   } // end XmlReaderTest
```

Fig. 15.7 **XmlNodeReader** used to iterate through an XML document. (Part 2 of 3.)



XmlReaderTest

```
<?xml version="1.0" ?>
<!-- Fig. 18.1: article.xml      -->
<!-- Article structured with XML -->
<article>
        <title>
                Simple XML
        </title>
        <date>
                December 6, 2001
        </date>
        <author>
                <firstName>
                        Tem
                </firstName>
                <lastName>
                        Nieto
                </lastName>
        </author>
        <summary>
                XML is pretty easy.
        </summary>
        <content>
                In this chapter, we present a wide v

        </content>
```

Fig. 15.7 **`XmlNodeReader`** used to iterate through an XML document. (Part 3 of 3.)

Line 6 includes the **`System.Xml`** namespace, which contains the XML classes used in this example. Line 18 creates a reference 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 19. Once an XML document is loaded into an **`XmlDocument`**, its data can be read and manipulated programmatically. In this example, we read each node in the **`XmlDocument`**, which is the DOM tree. In successive examples, we demonstrate how to manipulate node values.

In line 22, we create an **`XmlNodeReader`** and assign it to reference **`reader`**, which enables us to read one node at a time from the **`XmlDocument`**. Method ***`Read`*** of **`Xml-Reader`** reads one node from the DOM tree. Placing this statement in the **`while`** loop (lines 31–78) makes **`reader Read`** all the document nodes. The **`switch`** statement (lines 33–77) processes each node. Either the ***`Name`*** property (line 41), which contains the node's name, or the ***`Value`*** property (line 53), which contains the node's data, is formatted and concatenated to the **`string`** assigned to the text box **`Text`** property. 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.

Notice that the displayed output emphasizes the structure of the XML document. Variable **`depth`** (line 28) sets the number of tab characters used to indent each element. The depth is incremented each time an **`Element`** type is encountered and is decremented each time an **`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.

The C# program in Fig. 15.8 demonstrates how to manipulate DOM trees programmatically. This program loads **`letter.xml`** (Fig. 15.3) into the DOM tree and then creates a second DOM tree that duplicates the DOM tree containing **`letter.xml`**'s contents. The GUI for this application contains a text box, a **`TreeView`** control and three buttons—**Build**, **Print** and **Reset**. When clicked, **Build** copies **`letter.xml`** and displays the document's tree structure in the **`TreeView`** control, **Print** displays the XML element values and names in a text box and **Reset** clears the **`TreeView`** control and text box content.

Lines 20 and 23 create references to **`XmlDocument`**s **`source`** and **`copy`**. Line 32 assigns a new **`XmlDocument`** object to reference **`source`**. Line 33 then invokes method **`Load`** to parse and load **`letter.xml`**. We discuss reference **`copy`** shortly.

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

```
1   // Fig. 15.8: XmlDom.cs
2   // Demonstrates DOM tree manipulation.
3
4   using System;
5   using System.Windows.Forms;
6   using System.Xml;
7   using System.IO;
8   using System.CodeDom.Compiler;  // contains TempFileCollection
9
10  // Class XmlDom demonstrates the DOM
11  public class XmlDom : System.Windows.Forms.Form
12  {
13     private System.Windows.Forms.Button buildButton;
14     private System.Windows.Forms.Button printButton;
15     private System.Windows.Forms.TreeView xmlTreeView;
16     private System.Windows.Forms.TextBox consoleTextBox;
17     private System.Windows.Forms.Button resetButton;
18     private System.ComponentModel.Container components = null;
19
20     private XmlDocument source; // reference to "XML document"
21
22     // reference copy of source's "XML document"
23     private XmlDocument copy;
24
25     private TreeNode tree; // TreeNode reference
26
27     public XmlDom()
28     {
29        InitializeComponent();
30
31        // create XmlDocument and load letter.xml
32        source = new XmlDocument();
33        source.Load( "..\\..\\letter.xml" );
34
35        // initialize references to null
36        copy = null;
```

Fig. 15.8 DOM structure of an XML document illustrated by a class. (Part 1 of 6.)

```
         tree = null;
      } // end XmlDom

      [STAThread]
      static void Main()
      {
         Application.Run( new XmlDom() );
      }

      // event handler for buildButton click event
      private void buildButton_Click( object sender,
         System.EventArgs e )
      {
         // determine if copy has been built already
         if ( copy != null )
            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 buildButton_Click

      // event handler for printButton click event
      private void printButton_Click( object sender,
         System.EventArgs e )
      {
         // exit if copy does not reference an XmlDocument
         if ( copy == null )
            return;

         // create temporary XML file
         TempFileCollection file = new TempFileCollection();

         // create file that is deleted at program termination
         file.AddExtension( "xml", false );
         string[] filename = new string[ 1 ];
         file.CopyTo( filename, 0 );

         // write XML data to disk
         XmlTextWriter writer = new XmlTextWriter( filename[ 0 ],
            System.Text.Encoding.UTF8 );
         copy.WriteTo( writer );
```

Fig. 15.8 DOM structure of an XML document illustrated by a class. (Part 2 of 6.)

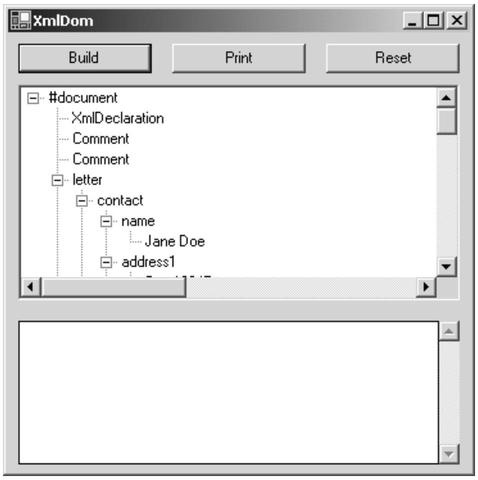
```
         writer.Close();

         // parse and load temporary XML document
         XmlTextReader reader = new XmlTextReader( filename[ 0 ] );

         // read, format and display data
         while( reader.Read() )
         {
            if ( reader.NodeType == XmlNodeType.EndElement )
               consoleTextBox.Text += "/";

            if ( reader.Name != String.Empty )
               consoleTextBox.Text += reader.Name + "\r\n";

            if ( reader.Value != String.Empty )
               consoleTextBox.Text += "\t" + reader.Value +
                  "\r\n";
         } // end while

         reader.Close();
      } // end printButton_Click

      // handle resetButton click event
      private void resetButton_Click( object sender,
         System.EventArgs e )
      {
         // remove TreeView nodes
         if ( tree != null )
            xmlTreeView.Nodes.Remove( tree );

         xmlTreeView.Refresh(); // force TreeView update

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

         consoleTextBox.Text = "";  // clear text box

         printButton.Enabled = false;
         resetButton.Enabled = false;

      } // end resetButton_Click

      // construct DOM tree
      private void BuildTree( XmlNode xmlSourceNode,
         XmlNode document, TreeNode treeNode )
      {
         // create XmlNodeReader to access XML document
         XmlNodeReader nodeReader = new XmlNodeReader(
            xmlSourceNode );

         // represents current node in DOM tree
         XmlNode currentNode = null;
```

Fig. 15.8 DOM structure of an XML document illustrated by a class. (Part 3 of 6.)

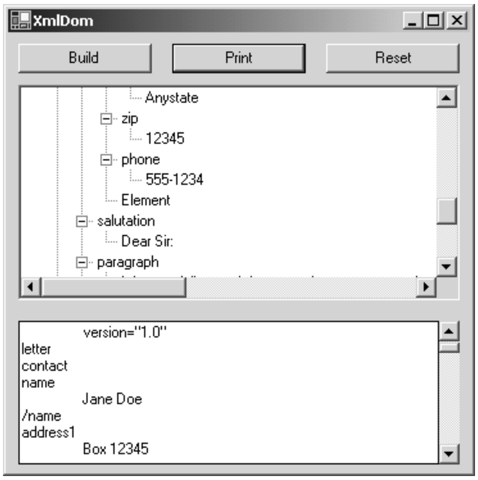
```

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

         // references modified node type for CreateNode
         XmlNodeType modifiedNodeType;

         while ( nodeReader.Read() )
         {
            // get current node type
            modifiedNodeType = nodeReader.NodeType;

            // check for EndElement, store as Element
            if ( modifiedNodeType == XmlNodeType.EndElement )
               modifiedNodeType = XmlNodeType.Element;

            // create node copy
            currentNode = copy.CreateNode( modifiedNodeType,
               nodeReader.Name, nodeReader.NamespaceURI );

            // build tree based on node type
            switch ( nodeReader.NodeType )
            {
               // if Text node, add its value to tree
               case XmlNodeType.Text:
                  newNode.Text = nodeReader.Value;
                  treeNode.Nodes.Add( newNode );

                  // append Text node value to currentNode data
                  ( ( XmlText ) currentNode ).AppendData(
                     nodeReader.Value );
                  document.AppendChild( currentNode );
                  break;

               // if EndElement, move up tree
               case XmlNodeType.EndElement:
                  document = document.ParentNode;
                  treeNode = treeNode.Parent;
                  break;

               // if new element, add name and traverse tree
               case XmlNodeType.Element:

                  // determine if element contains content
                  if ( !nodeReader.IsEmptyElement )
                  {
                     // assign node text, add newNode as child
                     newNode.Text = nodeReader.Name;
                     treeNode.Nodes.Add( newNode );

                     // set treeNode to last child
                     treeNode = newNode;

```

Fig. 15.8 DOM structure of an XML document illustrated by a class. (Part 4 of 6.)

```
                  document.AppendChild( currentNode );
                  document = document.LastChild;
               }
               else // do not traverse empty elements
               {
                  // assign NodeType string to newNode
                  newNode.Text =
                     nodeReader.NodeType.ToString();

                  treeNode.Nodes.Add( newNode );
                  document.AppendChild( currentNode );
               }

               break;

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

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

      // update the TreeView control
      xmlTreeView.ExpandAll();
      xmlTreeView.Refresh();

   } // end BuildTree
} // end XmlDom
```



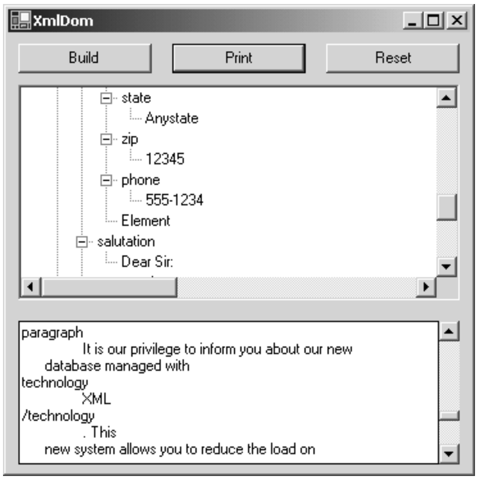
```
#document
   XmlDeclaration
   Comment
   Comment
   letter
      contact
         name
            Jane Doe
         address1
```



```
            Anystate
         zip
            12345
         phone
            555-1234
         Element
      salutation
         Dear Sir:
      paragraph
```

```
        version="1.0"
letter
contact
name
        Jane Doe
/name
address1
        Box 12345
```

Fig. 15.8 DOM structure of an XML document illustrated by a class. (Part 5 of 6.)



```
state
  Anystate
zip
  12345
phone
  555-1234
Element
salutation
  Dear Sir:
```

```
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
```

Fig. 15.8 DOM structure of an XML document illustrated by a class. (Part 6 of 6.)

When clicked, button **Build** triggers event handler **`buildButton_Click`** (lines 47–68), 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 55–56. Line 60 retrieves the **`Name`** of the node referenced by **`source`** (i.e., **`#root`**, 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 64 calls method **`BuildTree`** to copy the **`XMLDocument`** referenced by **`source`** and to update the **`TreeView`**.

Method **`BuildTree`** (line 134–226) receives an **`XmlNode`** representing the source node, an empty **`XmlNode`** and a **`treeNode`** to place in the DOM tree. Parameter **`treeNode`** references the current location in the tree (i.e., the **`TreeNode`** most recently added to the **`TreeView`** control). Lines 138–139 instantiate a new **`XmlNodeReader`** for iterating through the DOM tree. Lines 142–145 declare **`XmlNode`** and **`TreeNode`** references that indicate the next nodes added to **`document`** (i.e., the DOM tree referenced by **`copy`**) and **`treeNode`**. Lines 150–220 iterate through each node in the tree.

Lines 153–161 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`**. If the **`NodeType`** is of an **`EndElement`** type, lines 156–157 assign **`modifiedNodeType`** type **`Element`**.

The **`switch`** statement in lines 164–217 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. In lines 172–174, we downcast **`currentNode`** to **`XmlText`** and append the node's value. The **`currentNode`** then is appended to the **`document`**. Lines 178–181 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 184 matches **Element** node types. Each nonempty **Element NodeType** (line 187) increases the depth of the tree; thus, we assign the current **nodeReader Name** to the **newNode**'s **Text** property and add the **newNode** to the **treeNode** node list. Lines 194–197 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. If it is an empty element (line 199), 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 206 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** (line 71). Lines 79–84 create a temporary file for storing the XML. Line 87 creates an **XmlTextWriter** for streaming the XML data to disk. Method ***WriteTo*** is called to write the XML representation to the ***XmlTextWriter*** stream (line 89). Line 93 creates an ***XmlTextReader*** to read from the file. The **while** loop (line 96–107) reads each node in the DOM tree and writes tag names and character data to the text box. 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**, deletes both dynamically generated trees and updates the **TreeView** control's display. Reference **copy** is assigned **null** (to allow its tree to be garbage collected in line 123), and the **TreeNode** node list reference **tree** is assigned **null**.

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 the ***System.Xml.XPath*** namespace for iterating through node lists that match search criteria, which are 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 (such as, XSLT, discussed in Section 15.6).

Figure 15.9 demonstrates how to navigate through an XML document with an **XPathNavigator**. Like Fig. 15.8, 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 **XPath-Navigator** 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 Figure 15.10.

This program loads XML document **sports.xml** into an ***XPathDocument*** object by passing the document's file name to the **XPathDocument** constructor (line 36). Method ***CreateNavigator*** (line 39) creates and returns an **XPathNavigator** reference to the **XPathDocument**'s tree structure.

The navigation methods of **XPathNavigator** used in Fig. 15.9 are ***MoveTo-FirstChild*** (line 66), ***MoveToParent*** (line 94), ***MoveToNext*** (line 122) and ***MoveToPrevious*** (line 151). Each method performs the action that its name implies. Method **MoveToFirstChild** moves to the first child of the node referenced by the

XPathNavigator, **MoveToParent** moves to the parent node of the node referenced by the **XPathNavigator**, **MoveToNext** moves to the next sibling of the node referenced by the **XPathNavigator** and **MoveToPrevious** moves to the previous sibling of the node referenced by the **XPathNavigator**. Each method returns a **bool** indicating whether the move was successful. In this example, we display a warning in a **MessageBox** whenever a move operation fails. Furthermore, each of these methods is called in the event handler of the button that matches its name (e.g., button **First Child** triggers **firstChildButton_Click**, 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 208–229) that determines whether to assign the **Node**'s ***Name*** property or ***Value*** property to the **TreeNode** (lines 218 and 225). 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.

The other event handler corresponds to button **Select** (line 173–174). Method **Select** (line 182) 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. 15.11.

Method **DisplayIterator** (defined in lines 195–204) appends the node values from the given **XPathNodeIterator** to the **selectTreeViewer** text box. Note that we call the **string** method **Trim** to remove unnecessary whitespace. Method ***MoveNext*** (line 200) advances to the next node, which can be accessed via property ***Current*** (line 202).

```
 1  // Fig. 15.9: PathNavigator.cs
 2  // Demonstrates Class XPathNavigator.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml.XPath; // contains XPathNavigator
 7
 8  public class PathNavigator : System.Windows.Forms.Form
 9  {
10     private System.Windows.Forms.Button firstChildButton;
11     private System.Windows.Forms.Button parentButton;
12     private System.Windows.Forms.Button nextButton;
13     private System.Windows.Forms.Button previousButton;
14     private System.Windows.Forms.Button selectButton;
15     private System.Windows.Forms.TreeView pathTreeViewer;
16     private System.Windows.Forms.ComboBox selectComboBox;
17     private System.ComponentModel.Container components = null;
18     private System.Windows.Forms.TextBox selectTreeViewer;
19     private System.Windows.Forms.GroupBox navigateBox;
20     private System.Windows.Forms.GroupBox locateBox;
21
```

Fig. 15.9 **XPathNavigator** class used to navigate selected nodes. (Part 1 of 7.)

```
      // navigator to traverse document
      private XPathNavigator xpath;

      // references document for use by XPathNavigator
      private XPathDocument document;

      // references TreeNode list used by TreeView control
      private TreeNode tree;

      public PathNavigator()
      {
         InitializeComponent();

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

         // create navigator
         xpath = document.CreateNavigator();

         // create root node for TreeNodes
         tree = new TreeNode();

         tree.Text = xpath.NodeType.ToString(); // #root
         pathTreeViewer.Nodes.Add( tree );      // add tree

         // update TreeView control
         pathTreeViewer.ExpandAll();
         pathTreeViewer.Refresh();
         pathTreeViewer.SelectedNode = tree;    // highlight root
      } // end constructor

      [STAThread]
      static void Main()
      {
         Application.Run( new PathNavigator() );
      }

      // traverse to first child
      private void firstChildButton_Click( 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. 15.9 `XPathNavigator` class used to navigate selected nodes. (Part 2 of 7.)

```
            // 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;
         }
         else // node has no children
            MessageBox.Show( "Current Node has no children.",
               "", MessageBoxButtons.OK,
               MessageBoxIcon.Information );
      }

      // traverse to node's parent on parentButton click event
      private void parentButton_Click( 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;
         }
         else // if node has no parent (root node)
            MessageBox.Show( "Current node has no parent.", "",
               MessageBoxButtons.OK,
               MessageBoxIcon.Information );
      }

      // find next sibling on nextButton click event
      private void nextButton_Click( object sender,
         System.EventArgs e )
      {
         TreeNode newTreeNode = null, newNode = null;

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

            newNode = new TreeNode(); // create new node
```

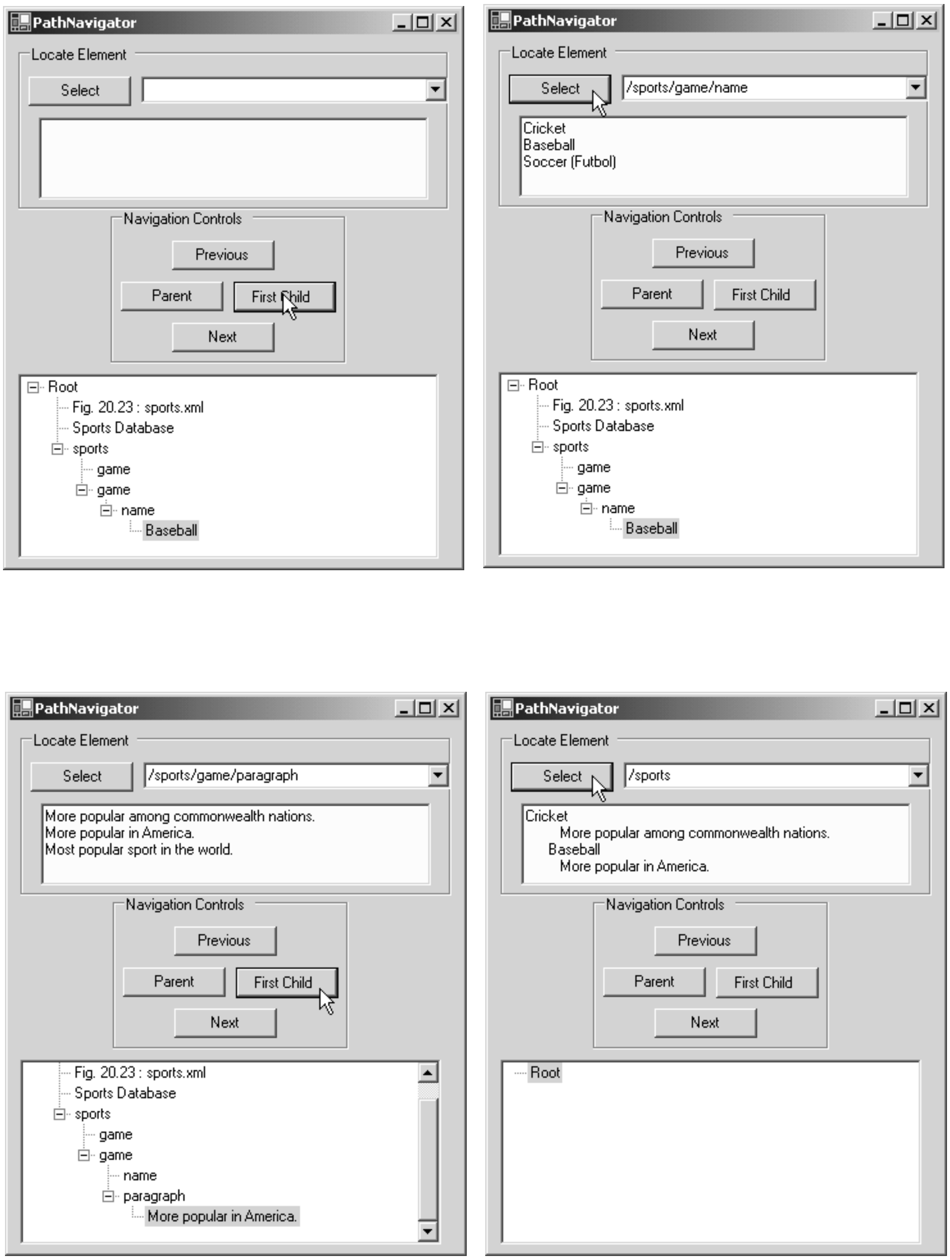
Fig. 15.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 3 of 7.)

```
            DetermineType( newNode, xpath );
            newTreeNode.Nodes.Add( newNode );

            // set current position for display
            tree = newNode;

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

      // get previous sibling on previousButton click
      private void previousButton_Click( object sender,
         System.EventArgs e )
      {
         TreeNode parentTreeNode = null;

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

      // process selectButton click event
      private void selectButton_Click( object sender,
         System.EventArgs e )
      {
         XPathNodeIterator iterator; // enables node iteration

```

Fig. 15.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 4 of 7.)

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

         // catch invalid expressions
         catch ( System.ArgumentException argumentException )
         {
            MessageBox.Show( argumentException.Message,
               "Error", MessageBoxButtons.OK,
               MessageBoxIcon.Error );
         }
      } // end selectButton_Click

      // print values for XPathNodeIterator
      private void DisplayIterator( XPathNodeIterator iterator )
      {
         selectTreeViewer.Text = "";

         // prints selected node's values
         while ( iterator.MoveNext() )
            selectTreeViewer.Text +=
               iterator.Current.Value.Trim()
               + "\r\n";
      } // end DisplayIterator

      // determine if TreeNode should display current node
      // name or value
      private void DetermineType( TreeNode node,
         XPathNavigator xPath )
      {
         // determine NodeType
         switch ( xPath.NodeType )
         {
            // if Element, get its name
            case XPathNodeType.Element:

               // get current node name, and remove whitespace
               node.Text = xPath.Name.Trim();
               break;

            // obtain node values
            default:

               // get current node value and remove whitespace
               node.Text = xPath.Value.Trim();
               break;

         } // end switch
      } // end DetermineType
   } // end PathNavigator
```

Fig. 15.9 `XPathNavigator` class used to navigate selected nodes. (Part 5 of 7.)



```
Root
  Fig. 20.23 : sports.xml
  Sports Database
  sports
    game
    game
      name
        Baseball
```

/sports/game/name

```
Cricket
Baseball
Soccer (Futbol)
```

/sports/game/paragraph

```
More popular among commonwealth nations.
More popular in America.
Most popular sport in the world.
```

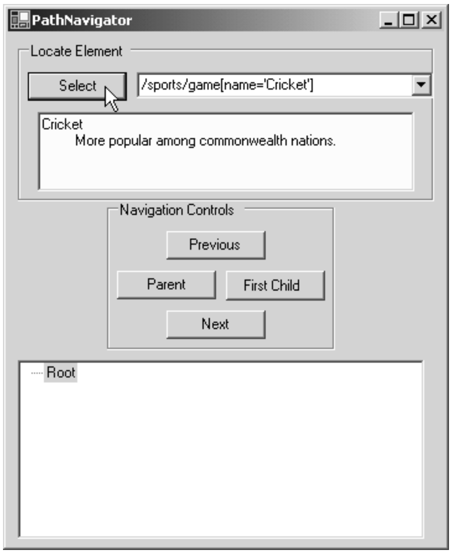
```
  Fig. 20.23 : sports.xml
  Sports Database
  sports
    game
    game
      name
      paragraph
        More popular in America.
```

/sports

```
Cricket
      More popular among commonwealth nations.
    Baseball
      More popular in America.
```

```
Root
```

Fig. 15.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 6 of 7.)



/sports/game[name='Cricket']

```
Cricket
	More popular among commonwealth nations.
```

Fig. 15.9 **`XPathNavigator`** class used to navigate selected nodes. (Part 7 of 7.)

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

<!-- Fig. 15.10: sports.xml -->
<!-- Sports Database        -->

<sports>

  <game id = "783">
      <name>Cricket</name>

      <paragraph>
         More popular among commonwealth nations.
      </paragraph>
   </game>

   <game id = "239">
      <name>Baseball</name>

      <paragraph>
         More popular in America.
      </paragraph>
   </game>

   <game id = "418">
      <name>Soccer(Futbol)</name>
      <paragraph>Most popular sport in the world</paragraph>
   </game>
</sports>
```

Fig. 15.10 XML document that describes various sports.

15.5 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 15.2

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 15.3

Because XML document content can be structured in many different ways, an application cannot determine whether the document data it receives is complete, missing data or 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.

Expression	Description
`/sports`	Matches the **`sports`** node that is child node of the document root node. This node contains the root element.
`/sports/game/name`	Matches all **`name`** nodes that are child nodes of **`game`**. The **`game`** node must be a child of **`sports`** and **`sports`** must be a root element node.
`/sports/game/paragraph`	Matches all **`paragraph`** nodes that are child nodes of **`game`**. The **`game`** node must be a child of **`sports`**, and **`sports`** must be a root element node.
`/sports/game[name='Cricket']`	Matches all **`game`** nodes that contain a child element **`name`** whose value is **`Cricket`**. The **`game`** node must be a child of **`sports`**, and **`sports`** must be a root element node.

Fig. 15.11 XPath expressions and descriptions.

15.5.1 Document Type Definitions

Document type definitions (DTDs) provide a means for type checking XML documents and thus 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 are recommended to ensure document conformity. The DTD in Fig. 15.12 defines the set of rules (i.e., the grammar) for structuring the business letter document contained in Fig. 15.13.

Portability Tip 15.2

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

Line 4 uses 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 definition (line 7) 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.

```
 1  <!-- Fig. 15.12: letter.dtd      -->
 2  <!-- DTD document for letter.xml -->
 3
 4  <!ELEMENT letter ( contact+, salutation, paragraph+,
 5     closing, signature )>
 6
 7  <!ELEMENT contact ( name, address1, address2, city, state,
 8     zip, phone, flag )>
 9  <!ATTLIST contact type CDATA #IMPLIED>
10
11  <!ELEMENT name ( #PCDATA )>
12  <!ELEMENT address1 ( #PCDATA )>
13  <!ELEMENT address2 ( #PCDATA )>
14  <!ELEMENT city ( #PCDATA )>
15  <!ELEMENT state ( #PCDATA )>
16  <!ELEMENT zip ( #PCDATA )>
17  <!ELEMENT phone ( #PCDATA )>
18  <!ELEMENT flag EMPTY>
19  <!ATTLIST flag gender (M | F) "M">
20
21  <!ELEMENT salutation ( #PCDATA )>
22  <!ELEMENT closing ( #PCDATA )>
23  <!ELEMENT paragraph ( #PCDATA )>
24  <!ELEMENT signature ( #PCDATA )>
```

Fig. 15.12 Document Type Definition (DTD) for a business letter.

Line 9 uses the ***ATTLIST*** *element type 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 and the 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 15.4

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 ampersand (**&**) must be replaced by their *entities* (i.e., **&lt;** and **&amp;**). However, the ampersand character can be inserted when used with entities. See Appendix M, HTML/XHTML Special Characters, for a list of pre-defined entities.

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

Common Programming Error 15.9

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

Many XML documents explicitly reference a DTD. Figure 15.13 is an XML document that conforms to **letter.dtd** (Fig. 15.12).

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

<!-- Fig. 15.13: 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>
```

Fig. 15.13 XML document referencing its associated DTD. (Part 1 of 2.)

```
17          <flag gender = "F" />
18       </contact>
19
20       <contact type = "to">
21          <name>John Doe</name>
22          <address1>123 Main St.</address1>
23          <address2></address2>
24          <city>Anytown</city>
25          <state>Anystate</state>
26          <zip>12345</zip>
27          <phone>555-1234</phone>
28          <flag gender = "M" />
29       </contact>
30
31       <salutation>Dear Sir:</salutation>
32
33       <paragraph>It is our privilege to inform you about our new
34          database managed with XML. This new system
35          allows you to reduce the load on your inventory list
36          server by having the client machine perform the work of
37          sorting and filtering the data.
38       </paragraph>
39
40       <paragraph>Please visit our Web site for availability
41          and pricing.
42       </paragraph>
43       <closing>Sincerely</closing>
44       <signature>Ms. Doe</signature>
45    </letter>
```

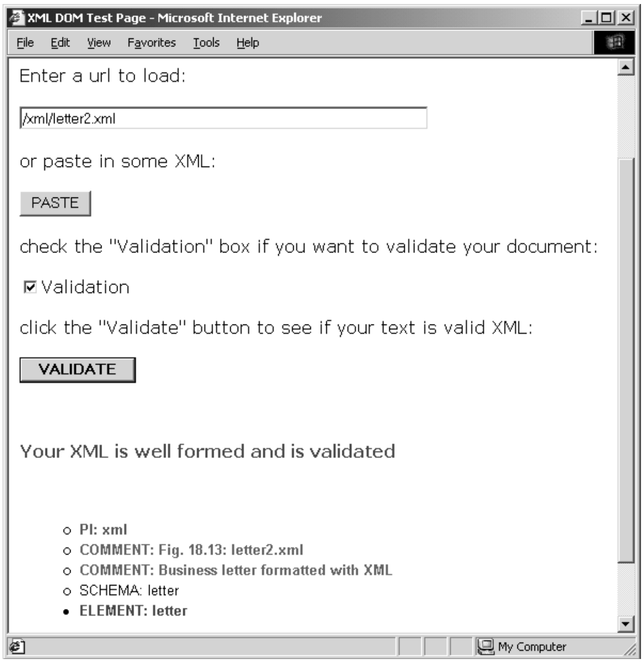
Fig. 15.13 XML document referencing its associated DTD. (Part 2 of 2.)

This XML document is similar to that in Fig. 15.3. 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). The output in Fig. 15.14 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 XML Validator is available free for download from

`msdn.microsoft.com/downloads/samples/Internet/xml/`
`xml_validator/sample.asp`

Microsoft 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 DOM Test Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
PASTE
VALIDATE
My Computer

Enter a url to load:

/xml/letter2.xml

or paste in some XML:

check the "Validation" box if you want to validate your document:

☑ Validation

click the "Validate" button to see if your text is valid XML:

Your XML is well formed and is validated

- PI: xml
- COMMENT: Fig. 18.13: letter2.xml
- COMMENT: Business letter formatted with XML
- SCHEMA: letter
- ELEMENT: letter

Fig. 15.14 XML Validator validates an XML document against a DTD.

XML documents that fail validation are still 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. 15.12 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. 15.15.

C# programs can use msxml to validate XML documents against DTDs. For information on how to accomplish this, visit:

`msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguidnf/html/cpconvalidationagainstdtdwithxmlvalidatingreader.asp`

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

15.5.2 Microsoft XML Schemas[2]

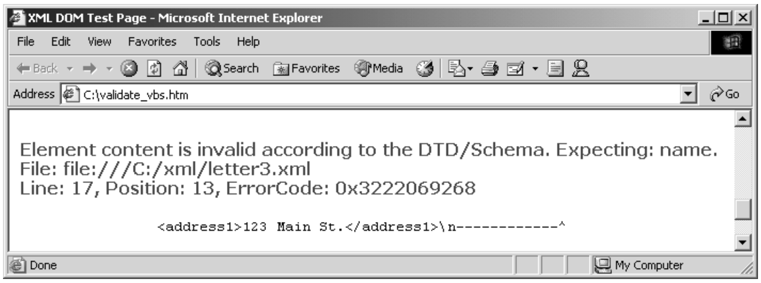
In this section, we introduce an alternative to DTDs—called Schemas—for defining an XML document's structure. Many developers in the XML community feel that DTDs are

2. W3C Schema, which we discuss in Section 15.5.3, 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.

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 that can be manipulated (e.g., elements can be added or removed, etc.) like any other XML document. As with DTDs, Schemas require validating parsers.

In this section, we focus on Microsoft's *XML Schema* vocabulary. Figure 15.16 presents an XML document that conforms to the Microsoft Schema document shown in Fig. 15.17. By convention, Microsoft XML Schema documents use the file extension ***.xdr***, which is short for *XML-Data Reduced*. Line 6 (Fig. 15.16) references the Schema document **book.xdr**.


XML DOM Test Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\validate_vbs.htm
Go
Done
My Computer

```
Element content is invalid according to the DTD/Schema. Expecting: name.
File: file:///C:/xml/letter3.xml
Line: 17, Position: 13, ErrorCode: 0x3222069268

            <address1>123 Main St.</address1>\n------------^
```

Fig. 15.15 XML Validator displaying an error message.

```
1   <?xml version = "1.0"?>
2
3   <!-- Fig. 15.16: bookxdr.xml          -->
4   <!-- XML file that marks up book data -->
5
6   <books xmlns = "x-schema:book.xdr">
7      <book>
8         <title>C# How to Program</title>
9      </book>
10
11     <book>
12        <title>Java How to Program, 4/e</title>
13     </book>
14
15     <book>
16        <title>Visual Basic .NET How to Program</title>
17     </book>
18
19     <book>
20        <title>Advanced Java 2 Platform How to Program</title>
21     </book>
```

Fig. 15.16 XML document that conforms to a Microsoft Schema document. (Part 1 of 2.)

```

   <book>
      <title>Python How to Program</title>
   </book>
</books>
```

Fig. 15.16 XML document that conforms to a Microsoft Schema document. (Part 2 of 2.)

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 15.17: book.xdr                         -->
 4  <!-- Schema document to which book.xml conforms -->
 5
 6  <Schema xmlns = "urn:schemas-microsoft-com:xml-data">
 7     <ElementType name = "title" content = "textOnly"
 8        model = "closed" />
 9
10     <ElementType name = "book" content = "eltOnly" model = "closed">
11        <element type = "title" minOccurs = "1" maxOccurs = "1" />
12     </ElementType>
13
14     <ElementType name = "books" content = "eltOnly" model = "closed">
15        <element type = "book" minOccurs = "0" maxOccurs = "*" />
16     </ElementType>
17  </Schema>
```

Fig. 15.17 Microsoft Schema file that contains structure to which **bookxdr.xml** conforms.

Software Engineering Observation 15.5

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 15.6

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.*[3]

In line 6, root element ***Schema*** begins the Schema markup. Microsoft Schemas use the namespace URI ***"urn:schemas-microsoft-com:xml-data"***. Line 7 uses element ***ElementType*** to define element **title**. Attribute ***content*** specifies that this element contains parsed character data (i.e., text only). Element **title** is not permitted to contain child elements. Setting the ***model*** *attribute* to ***"closed"*** specifies that a conforming XML document can contain only elements defined in this 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** ele-

3. See, for example, **opengis.net/schema.htm**.

ment 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 15.5.4.

15.5.3 W3C XML Schema[4]

In this section, we focus on *W3C XML Schema*[5]—the schema that the W3C created. XML Schema is a *Recommendation* (i.e., a stable release suitable for use in industry). Figure 15.18 shows a Schema-valid XML document named **bookxsd.xml** and Fig. 15.19 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.

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

<!-- Fig. 15.18: bookxsd.xml                     -->
<!-- Document that conforms to W3C XML Schema -->

<deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
   <book>
      <title>e-Business and e-Commerce How to Program</title>
   </book>
   <book>
      <title>Python How to Program</title>
   </book>
</deitel:books>
```

Fig. 15.18 XML document that conforms to W3C XML Schema.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.19: book.xsd           -->
4  <!-- Simple W3C XML Schema document -->
5
6  <xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
7     xmlns:deitel = "http://www.deitel.com/booklist"
8     targetNamespace = "http://www.deitel.com/booklist">
9
10    <xsd:element name = "books" type = "deitel:BooksType"/>
11
12    <xsd:complexType name = "BooksType">
13       <xsd:sequence>
14          <xsd:element name = "book" type = "deitel:BookType"
15          minOccurs = "1" maxOccurs = "unbounded"/>
16       </xsd:sequence>
17    </xsd:complexType>
```

Fig. 15.19 XSD Schema document to which **bookxsd.xml** conforms. (Part 1 of 2.)

4. We provide a detailed treatment of W3C Schema in *XML for Experienced Programmers* (2003).
5. For the latest on W3C XML Schema, visit **www.w3.org/XML/Schema**.

```
18
19    <xsd:complexType name = "BookType">
20       <xsd:sequence>
21          <xsd:element name = "title" type = "xsd:string"/>
22       </xsd:sequence>
23    </xsd:complexType>
24
25 </xsd:schema>
```

Fig. 15.19 XSD Schema document to which **`bookxsd.xml`** conforms. (Part 2 of 2.)

W3C XML Schema use the namespace URI ***`http://www.w3.org/2001/XMLSchema`*** and often use *namespace prefix* ***`xsd`*** (line 6 in Fig. 15.19). Root element ***`schema`*** 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 element ***`complexType`*** to define an element type that has a child element named **`book`**. Because **`book`** contains a child element, its type must be a complex type (e.g., **`BookType`**). Attribute ***`minOccurs`*** specifies that **`books`** must contain a minimum of one **`book`** element. Attribute ***`maxOccurs`***, with value ***`unbounded`*** (line 14) 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 15.1

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

15.5.4 Schema Validation in C#

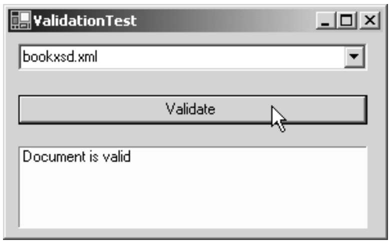
In this section, we present a C# application (Fig. 15.20) that uses classes from the .NET Framework Class Library to validate the XML documents presented in the last two sections against their respective Schemas. We use an instance of ***`XmlValidatingReader`*** to perform the validation.

Line 17 creates an ***`XmlSchemaCollection`*** reference named **`schemas`**. Line 28 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 29 calls method **Add** to add a W3C XML Schema. The first argument specifies the namespace URI (i.e., line 18 in Fig. 15.19) and the second argument identifies the schema file (i.e., **"book.xsd"**). This is the Schema that is used to validate **bookxsd.xml**.

```
 1  // Fig. 15.20: ValidationTest.cs
 2  // Validating XML documents against Schemas.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml;
 7  using System.Xml.Schema;          // contains Schema classes
 8
 9  // determines XML document Schema validity
10  public class ValidationTest : System.Windows.Forms.Form
11  {
12     private System.Windows.Forms.ComboBox filesComboBox;
13     private System.Windows.Forms.Button validateButton;
14     private System.Windows.Forms.Label consoleLabel;
15     private System.ComponentModel.Container components = null;
16
17     private XmlSchemaCollection schemas;   // Schemas
18     private bool valid;                    // validation result
19
20     public ValidationTest()
21     {
22        InitializeComponent();
23
24        valid = true;  // assume document is valid
25
26        // get Schema(s) for validation
27        schemas = new XmlSchemaCollection();
28        schemas.Add( "book", "book.xdr" );
29        schemas.Add( "http://www.deitel.com/booklist", "book.xsd" );
30     } // end constructor
31
32     // Visual Studio .NET generated code
33
34     [STAThread]
35     static void Main()
36     {
37        Application.Run( new ValidationTest() );
38     } // end Main
39
40     // handle validateButton click event
41     private void validateButton_Click( object sender,
42        System.EventArgs e )
43     {
44        // get XML document
45        XmlTextReader reader =
46           new XmlTextReader( filesComboBox.Text );
47
```

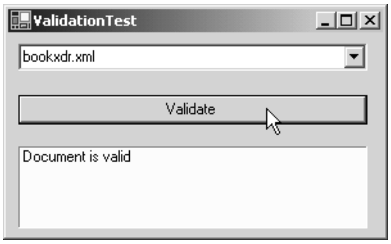
Fig. 15.20 Schema-validation example. (Part 1 of 2.)

```
48       // get validator
49       XmlValidatingReader validator =
50          new XmlValidatingReader( reader );
51
52       // assign Schema(s)
53       validator.Schemas.Add( schemas );
54
55       // set validation type
56       validator.ValidationType = ValidationType.Auto;
57
58       // register event handler for validation error(s)
59       validator.ValidationEventHandler +=
60          new ValidationEventHandler( ValidationError );
61
62       // validate document node-by-node
63       while ( validator.Read() ) ; // empty body
64
65       // check validation result
66       if ( valid )
67          consoleLabel.Text = "Document is valid";
68
69       valid = true; // reset variable
70
71       // close reader stream
72       validator.Close();
73    }  // end validateButton_Click
74
75    // event handler for validation error
76    private void ValidationError( object sender,
77       ValidationEventArgs arguments )
78    {
79       consoleLabel.Text = arguments.Message;
80       valid = false; // validation failed
81    } // end ValidationError
82 } // end ValidationTest
```



bookxsd.xml

Document is valid



bookxdr.xml

Document is valid

Fig. 15.20 Schema-validation example. (Part 2 of 2.)

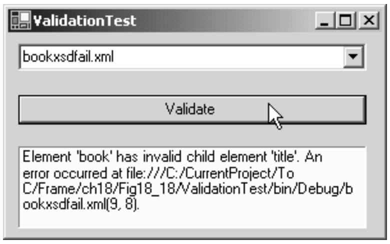
Lines 45–46 create an **XmlReader** for the file that the user selected from **filesComboBox**. The XML document to be validated against a Schema contained in the **XmlSchemaCollection** must be passed to the **XmlValidatingReader** constructor (lines 49–50).

Line 53 **Add**s the Schema collection referenced by **Schemas** to the ***Schemas*** *property*. This property sets the Schema used to validate the document. The ***ValidationType*** property (line 56) is set to the ***ValidationType*** *enumeration* constant for **Auto**matically identifying the Schema's type (i.e., XDR or XSD). Lines 59–60 register method **ValidationError** with ***ValidationEventHandler***. Method **ValidationError** (lines 76–81) is called if the document is invalid or 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 63). Each call to **Read** validates the next node in the document. The loop terminates either when all nodes have been validated successfully or a node fails validation. When validated against their respective Schemas, the XML documents in Fig. 15.16 and Fig. 15.18 validate successfully.

Figure 15.21 and Fig. 15.22 list two XML documents that fail to conform to **book.xdr** and **book.xsd**, respectively. In Fig. 15.21, the extra **title** element in **book** (lines 19–22) invalidate the document. In Fig. 15.22, the extra **title** element in **book** (lines 7–10) invalidates the document. Although both documents are invalid, they are well formed.

```
 1   <?xml version = "1.0"?>
 2
 3  <!-- Fig. 15.21: bookxsdfail.xml                  -->
 4  <!-- Document that does not conforms to W3C Schema -->
 5
 6  <deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
 7     <book>
 8        <title>e-Business and e-Commerce How to Program</title>
 9        <title>C# How to Program</title>
10     </book>
11     <book>
12        <title>Python How to Program</title>
13     </book>
14  </deitel:books>
```



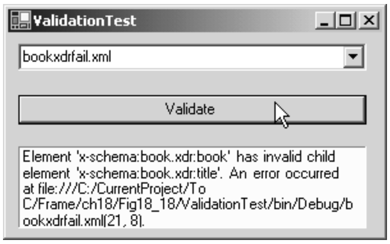
```
Element 'book' has invalid child element 'title'. An
error occurred at file:///C:/CurrentProject/To
C/Frame/ch18/Fig18_18/ValidationTest/bin/Debug/b
ookxsdfail.xml(9, 8).
```

Fig. 15.21 XML document that does not conform to the XSD schema of Fig. 15.19.

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 15.22: bookxdrfail.xml                       -->
 4  <!-- XML file that does not conform to Schema book.xdr -->
 5
```

Fig. 15.22 XML file that does not conform to the Schema in Fig. 15.17. (Part 1 of 2.)

```
<books xmlns = "x-schema:book.xdr">
   <book>
      <title>XML How to Program</title>
   </book>

   <book>
      <title>Java How to Program, 4/e</title>
   </book>

   <book>
      <title>Visual Basic .NET How to Program</title>
   </book>

   <book>
      <title>C++ How to Program, 3/e</title>
      <title>Python How to Program</title>
   </book>

   <book>
      <title>C# How to Program</title>
   </book>
</books>
```



```
Element 'x-schema:book.xdr:book' has invalid child
element 'x-schema:book.xdr:title'. An error occurred
at file:///C:/CurrentProject/To
C/Frame/ch18/Fig18_18/ValidationTest/bin/Debug/b
ookxdrfail.xml(21, 8).
```

Fig. 15.22 XML file that does not conform to the Schema in Fig. 15.17. (Part 2 of 2.)

15.6 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—called *XSL Transformations* (*XSLT*)—that creates formatted text-based 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) of the transformation.[6] 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*. The XML document, shown in Fig. 15.23, is transformed by msxml into an XHTML document (Fig. 15.24).

6. 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 K and L and visit **`www.w3.org`**.

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 15.23: sorting.xml                  -->
 4  <!-- XML document containing book information -->
 5
 6  <?xml:stylesheet type = "text/xsl" href = "sorting.xsl"?>
 7
 8  <book isbn = "999-99999-9-X">
 9     <title>Deitel&apos;s XML Primer</title>
10
11     <author>
12        <firstName>Paul</firstName>
13        <lastName>Deitel</lastName>
14     </author>
15
16     <chapters>
17        <frontMatter>
18           <preface pages = "2" />
19           <contents pages = "5" />
20           <illustrations pages = "4" />
21        </frontMatter>
22
23        <chapter number = "3" pages = "44">
24           Advanced XML</chapter>
25
26        <chapter number = "2" pages = "35">
27           Intermediate XML</chapter>
28
29        <appendix number = "B" pages = "26">
30           Parsers and Tools</appendix>
31
32        <appendix number = "A" pages = "7">
33           Entities</appendix>
34
35        <chapter number = "1" pages = "28">
36           XML Fundamentals</chapter>
37     </chapters>
38
39     <media type = "CD" />
40  </book>
```

Fig. 15.23 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. 15.24 presents the XSLT document (`sorting.xsl`) that transforms `sorting.xml` (Fig. 15.23) to XHTML.

Performance Tip 15.1

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. 15.23 contains the XML declaration. Recall that an XSL document is an XML document. Line 6 is 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 11–13 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.

XSLT documents contain one or more ***`xsl:template`*** elements that specify which information is output to the result tree. The template on line 16 ***`match`***es 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 referenced by **`xsl`** is output to the result tree. Line 18 calls for all the **`template`**s that match children of the document root to be applied. Line 23 specifies a **`template`** that **`match`**es element **`book`**.

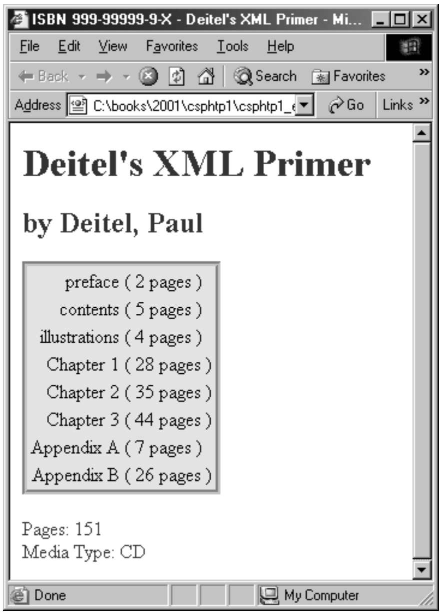
```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. 15.24: sorting.xsl                     -->
 4  <!-- Transformation of book information into XHTML -->
 5
 6  <xsl:stylesheet version = "1.0"
 7     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
 8
 9     <!-- write XML declaration and DOCTYPE DTD information -->
10     <xsl:output method = "xml" omit-xml-declaration = "no"
11        doctype-system =
12           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
13        doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
14
15     <!-- match document root -->
16     <xsl:template match = "/">
17        <html xmlns = "http://www.w3.org/1999/xhtml">
18           <xsl:apply-templates/>
19        </html>
20     </xsl:template>
21
22     <!-- match book -->
23     <xsl:template match = "book">
24        <head>
25           <title>ISBN <xsl:value-of select = "@isbn" /> -
26              <xsl:value-of select = "title" /></title>
27        </head>
```

Fig. 15.24 XSL document that transforms **`sorting.xml`** (Fig. 15.23) into XHTML. (Part 1 of 3.)

```

      <body>
         <h1 style = "color: blue">
            <xsl:value-of select = "title"/></h1>

         <h2 style = "color: blue">by <xsl:value-of
            select = "author/lastName" />,
            <xsl:value-of select = "author/firstName" /></h2>

         <table style =
            "border-style: groove; background-color: wheat">

            <xsl:for-each select = "chapters/frontMatter/*">
               <tr>
                  <td style = "text-align: right">
                     <xsl:value-of select = "name()" />
                  </td>

                  <td>
                    ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/chapter">
               <xsl:sort select = "@number" data-type = "number"
                    order = "ascending" />
               <tr>
                  <td style = "text-align: right">
                     Chapter <xsl:value-of select = "@number" />
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>

            <xsl:for-each select = "chapters/appendix">
               <xsl:sort select = "@number" data-type = "text"
                    order = "ascending" />
               <tr>
                  <td style = "text-align: right">
                     Appendix <xsl:value-of select = "@number" />
                  </td>

                  <td>
                     ( <xsl:value-of select = "@pages" /> pages )
                  </td>
               </tr>
            </xsl:for-each>
         </table>
```

Fig. 15.24 XSL document that transforms **`sorting.xml`** (Fig. 15.23) into XHTML. (Part 2 of 3.)

```
80
81             <br /><p style = "color: blue">Pages:
82                <xsl:variable name = "pagecount"
83                   select = "sum(chapters//*/@pages)" />
84                <xsl:value-of select = "$pagecount" />
85             <br />Media Type:
86                <xsl:value-of select = "media/@type" /></p>
87          </body>
88       </xsl:template>
89
90    </xsl:stylesheet>
```


ISBN 999-99999-9-X - Deitel's XML Primer - Mi...
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\books\2001\csphtp1\csphtp1_e
Go Links
Done
My Computer

Deitel's XML Primer

by Deitel, Paul

preface (2 pages)
contents (5 pages)
illustrations (4 pages)
Chapter 1 (28 pages)
Chapter 2 (35 pages)
Chapter 3 (44 pages)
Appendix A (7 pages)
Appendix B (26 pages)

Pages: 151
Media Type: CD

Fig. 15.24 XSL document that transforms **`sorting.xml`** (Fig. 15.23) into XHTML. (Part 3 of 3.)

Lines 25–26 create the title for the XHTML document. We use the ISBN of the book from attribute **`isbn`** and the contents of element **`title`** to create the title string **ISBN 999-99999-9-X - Deitel's XML Primer**. Element **`xsl:value-of`** selects the **`book`** element's **`isbn`** attribute.

Lines 33–35 create a header element that contains the book's author. Because the *context node* (i.e., the current node being processed) is **`book`**, 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 (indicated by an asterisk) that is a child of element **`frontMatter`**. 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).

Lines 53–54 sort **`chapter`**s by number in ascending order. Attribute **`select`** selects the value of context node **`chapter`**'s attribute **`number`**. Attribute ***`data-type`*** with 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 descendent 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*** performs the transformation.

Figure 15.25 applies a style sheet (**sports.xsl**) to **sports.xml** (Fig. 15.10). The transformation result is written to a text box and to a file. We also show the transformation results rendered in IE.

Line 20 declares **XslTransform** reference **transformer**. An object of this type is necessary to transform the XML data to another format. In line 29, the XML document is parsed and loaded into memory with a call to method **Load**. Method **CreateNavigator** is called in line 32 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 36) parses and loads the style sheet that this application uses. The argument that is passed contains the name and location of the style sheet.

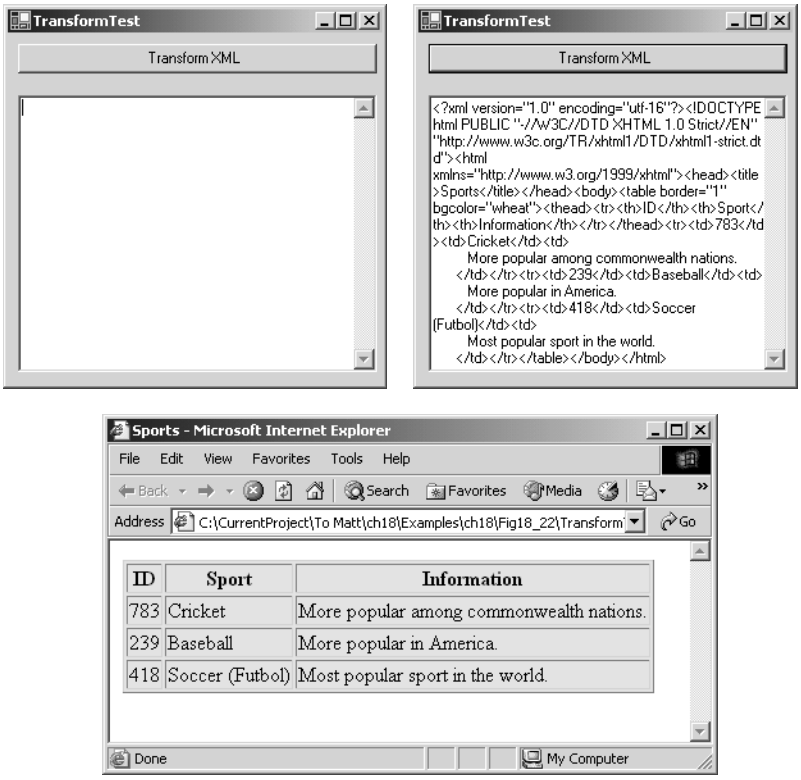
Event handler **transformButton_Click** calls method ***Transform*** of class **XslTransform** to apply the style sheet (**sports.xsl**) to **sports.xml** (line 53). This method takes three arguments: An **XPathNavigator** (created from **sports.xml**'s **XmlDocument**), an instance of class ***XsltArgumentList***, which is a list of **string** parameters that can be applied to a style sheet—**null**, 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 referenced by **output**. Lines 59–62 write the transformation results to disk. The third screen shot depicts the created XHTML document when it is rendered in IE.

```
 1  // Fig. 15.25: TransformTest.cs
 2  // Applying a style sheet to an XML document.
 3
 4  using System;
 5  using System.Windows.Forms;
 6  using System.Xml;
 7  using System.Xml.XPath;   // contains XPath classes
 8  using System.Xml.Xsl;     // contains style sheet classes
 9  using System.IO;          // contains stream classes
10
11  // transforms XML document to XHTML
12  public class TransformTest : System.Windows.Forms.Form
13  {
14     private System.Windows.Forms.TextBox consoleTextBox;
15     private System.Windows.Forms.Button transformButton;
16     private System.ComponentModel.Container components = null;
```

Fig. 15.25 XSL style sheet applied to an XML document. (Part 1 of 3.)

```

   private XmlDocument document;     // Xml document root
   private XPathNavigator navigator; // navigate document
   private XslTransform transformer; // transform document
   private StringWriter output;      // display document

   public TransformTest()
   {
      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" );
   } // end constructor

   // Windows Form Designer generated code

   [STAThread]
   static void Main()
   {
      Application.Run( new TransformTest() );
   } // end Main

   // transformButton click event
   private void transformButton_Click( object sender,
      System.EventArgs e )
   {
      // transform XML data
      output = new StringWriter();
      transformer.Transform( navigator, null, output );

      // display transformation in text box
      consoleTextBox.Text = output.ToString();

      // write transformation result to disk
      FileStream stream = new FileStream( "..\\..\\sports.html",
         FileMode.Create );
      StreamWriter writer = new StreamWriter( stream );
      writer.Write( output.ToString() );

      // close streams
      writer.Close();
      output.Close();
   } // end transformButton_Click
}    // end TransformTest
```

Fig. 15.25 XSL style sheet applied to an XML document. (Part 2 of 3.)


TransformTest
Transform XML
TransformTest
Transform XML
Sports - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address C:\CurrentProject\To Matt\ch18\Examples\ch18\Fig18_22\Transform
Go
Done
My Computer

```
<?xml version="1.0" encoding="utf-16"?><!DOCTYPE
html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3c.org/TR/xhtml1/DTD/xhtml1-strict.dt
d"><html
xmlns="http://www.w3.org/1999/xhtml"><head><title
>Sports</title></head><body><table border="1"
bgcolor="wheat"><thead><tr><th>ID</th><th>Sport</
th><th>Information</th></tr></thead><tr><td>783</td
><td>Cricket</td><td>
        More popular among commonwealth nations.
     </td></tr><tr><td>239</td><td>Baseball</td><td>
        More popular in America.
     </td></tr><tr><td>418</td><td>Soccer
(Futbol)</td><td>
        Most popular sport in the world.
     </td></tr></table></body></html>
```

ID	Sport	Information
783	Cricket	More popular among commonwealth nations.
239	Baseball	More popular in America.
418	Soccer (Futbol)	Most popular sport in the world.

Fig. 15.25 XSL style sheet applied to an XML document. (Part 3 of 3.)

15.7 Microsoft BizTalk™

Increasingly, organizations are using the Internet to exchange critical data between business partners and their own business divisions. However, transferring data between organizations can become difficult, because companies often use different platforms, applications and data specifications that complicate data transfer. For example, consider a business that supplies raw materials to a variety of industries. If the supplier cannot receive all orders electronically because their customers use different computing platforms, an employee must input order data manually. If the supplier receives hundreds of orders a day, typing mistakes are likely, resulting in incorrect inventories or wrong order fulfillments, thereby jeopardizing the business by losing customers.

The supplier has several options—either continue to have data entered manually, purchase the same software packages as the ones their customers use or encourage customers to adopt the applications used by the supply company. In a growing economy, a business would have to purchase and maintain disparate software packages, spend money for more employees to process data or force their business partners to standardize their own organizational software programs. To facilitate the flow of information between businesses, Microsoft developed *BizTalk* ("business talk"), an XML-based technology that helps to manage and facilitate business transactions.

BizTalk creates an environment in which data marked up as XML is used to exchange business-specific information, regardless of platform or programming applications. This section overviews BizTalk and presents a code example to illustrate the business-specific information included in the markup.

BizTalk consists of three parts: The BizTalk Server, the BizTalk Framework and the BizTalk Schema Library. The *BizTalk Server* (*BTS*) parses and translates all inbound and outbound messages (or documents) that are sent to and from a business, using Internet standards such as HTTP. The *BizTalk Framework* is a Schema for structuring those messages. The Framework offers a specific set of core tags. Businesses can download the Framework to use in their organizations and can submit new schemas to the BizTalk organization, at **`www.biztalk.org`**. Once the BizTalk organization verifies and validates the submissions, the Schemas become BizTalk Framework Schemas. The *BizTalk Schema Library* is a collection of Framework Schemas. Figure 15.26 summarizes BizTalk terminology.

Fig. 15.27 is an example BizTalk message for a product offer from a clothing company. The message Schema for this example was developed by Microsoft to facilitate online purchases by a retailer from a wholesaler. We use this Schema for a fictitious company, named ExComp.

BizTalk	Description
Framework	A specification that defines a format for messages.
Schema library	A repository of Framework XML Schemas.
Server	An application that assists vendors in converting their messages to BizTalk format. For more information, visit **`www.microsoft.com/biztalkserver`**
JumpStart Kit	A set of tools for developing BizTalk applications.

Fig. 15.26 BizTalk terminology.

```
<?xml version = "1.0"?>
<BizTalk xmlns =
   "urn:schemas-biztalk-org:BizTalk/biztalk-0.81.xml">

<!-- Fig. 15.27: biztalkmarkup.xml      -->
<!-- Example of standard BizTalk markup -->

   <Route>
      <From locationID = "8888888" locationType = "DUNS"
         handle = "23" />

      <To locationID = "454545445" locationType = "DUNS"
         handle = "45" />
   </Route>

```

Fig. 15.27 BizTalk markup using an offer Schema. (Part 1 of 3.)

```
   <Body>
      <Offers xmlns =
         "x-schema:http://schemas.biztalk.org/eshop_msn_com/
t7ntoqnq.xml">
         <Offer>
            <Model>12-a-3411d</Model>
            <Manufacturer>ExComp, Inc.</Manufacturer>
            <ManufacturerModel>DCS-48403</ManufacturerModel>

            <MerchantCategory>
               Clothes | Sports wear
            </MerchantCategory>

            <MSNClassId></MSNClassId>

            <StartDate>2001-06-05 T13:12:00</StartDate>
            <EndDate>2001-12-05T13:12:00</EndDate>

            <RegularPrice>89.99</RegularPrice>
            <CurrentPrice>25.99</CurrentPrice>
            <DisplayPrice value = "3" />
            <InStock value = "15" />

            <ReferenceImageURL>
               http://www.Example.com/clothes/index.jpg
            </ReferenceImageURL>

            <OfferName>Clearance sale</OfferName>

            <OfferDescription>
               This is a clearance sale
            </OfferDescription>

            <PromotionalText>Free Shipping</PromotionalText>

            <Comments>
               Clothes that you would love to wear.
            </Comments>

            <IconType value = "BuyNow" />

            <ActionURL>
               http://www.example.com/action.htm
            </ActionURL>

            <AgeGroup1 value = "Infant" />
            <AgeGroup2 value = "Adult" />

            <Occasion1 value = "Birthday" />
            <Occasion2 value = "Anniversary" />
            <Occasion3 value = "Christmas" />

         </Offer>
```

Fig. 15.27 BizTalk markup using an offer Schema. (Part 2 of 3.)

```
68          </Offers>
69       </Body>
70    </BizTalk>
```

Fig. 15.27 BizTalk markup using an offer Schema. (Part 3 of 3.)

All Biztalk documents have the root element ***BizTalk*** (line 2). Line 3 defines a default namespace for the **BizTalk** framework elements. Element ***Route*** (lines 8–14) contains the routing information, which is mandatory for all BizTalk documents. Element **Route** also contains elements ***To*** and ***From*** (lines 9–12), which indicate the document's destination and source, respectively. This makes it easier for the receiving application to communicate with the sender. Attribute ***locationType*** specifies the type of business that sends or receives the information, and attribute ***locationID*** specifies a business identity (the unique identifier for a business). These attributes facilitate source and destination organization. Attribute ***handle*** provides information to routing applications that handle the document.

Element ***Body*** (lines 16–69) contains the actual message, whose Schema is defined by the businesses themselves. Lines 17–18 specify the default namespace for element ***Offers*** (lines 17–68), which is contained in element **Body** (note that line 18 wraps—if we split this line, Internet Explorer cannot locate the namespace). Each offer is marked up using an ***Offer*** element (lines 19–67) that contains elements describing the offer. Note that the tags all are business-related elements, and easily understood. For additional information on BizTalk, visit **www.biztalk.com**.

In this chapter, we studied the Extensible Markup Language and several of its related technologies. In Chapter 16, we begin our discussion of databases, which are crucial to the development of multi-tier Web-based applications.

15.8 Summary

XML is a widely supported, open (i.e., nonproprietary) 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. Because XML elements describe the data they contain, they are readable by both humans and machines.

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 that describe specific types of data—i.e., mathematical formulas, chemical molecular structures, music and recipes.

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

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

Data are marked up with tags whose names are enclosed in angle brackets (**<>**). Tags are used in pairs to delimit markup. A tag that begins markup is called a start tag, and a tag that terminates markup is called an end tag. End tags differ from start tags in that end tags 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.

Because XML allows document authors to create their own tags, naming collisions (i.e., two different elements that have the same name) can occur. As in C#, XML namespaces provide a means for document authors to prevent collisions. Elements are qualified with namespace prefixes that specify the namespace to which they belong.

Each namespace prefix is bound to a uniform resource identifier (URI) that uniquely identifies the namespace. A URI is a series of characters that differentiates names. Document authors create their own namespace prefixes. Virtually any name can be used as a namespace prefix, except the reserved namespace prefix **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. Nodes that contain other nodes (called child nodes) are called parent nodes. Nodes that have the same parent are called sibling nodes. A node's descendant nodes include that node's children, its children's children and so on. A node's ancestor nodes include that node's parent, its parent's parent and so on. The DOM tree has a single root node that contains all other nodes in the document.

Namespace **System.Xml** contains classes for creating, reading and manipulating XML documents. **XmlReader**-derived class **XmlNodeReader** iterates through each node in the XML document. An **XmlDocument** object conceptually represents an empty XML document. XML documents are parsed and loaded into an **XmlDocument** object when method **Load** is invoked. Once an XML document is loaded into an **XmlDocument**, its data can be read and manipulated programmatically. An **XmlNodeReader** allows programmers to read one node at a time from an **XmlDocument**. An **XmlTextWriter** streams XML data to disk. An **XmlTextReader** reads XML data from a file.

XPath (XML Path Language) provides 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. Class **XPathNavigator** in the **System.Xml.XPath** namespace can iterate through node lists that match search criteria, written as an XPath expression.

XML documents contain only data; however, XSLT is capable of transforming XML documents into any text-based format. 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. XML documents can be transformed programmatically through C#. The **System.Xml.Xsl** namespace facilities the application of XSLT style sheets to XML documents.

15.9 Internet and World Wide 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.xml.org`
`xml.org` is a reference for XML, DTDs, schemas and namespaces.

`www.w3.org/style/XSL`
This W3C page provides information on XSL, including topics such 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 was derived.

`www.xml-zone.com`
The Development Exchange XML Zone is a complete resource for XML information. This site includes a 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, FAQs, the latest news, and numerous 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`
The microsoft XML validator, which can be downloaded from this site, can validate both online and off-line 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.gca.org/whats_xml/default.htm`
The GCA site offers an XML glossary, list of books on XML, brief descriptions of the draft standards for XML and links to online drafts.

`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/index.html`
The XML and Metadata Developer Central site has demos, technical notes and news articles related to XML.

`www.projectcool.com/developer/xmlz`
The Project Cool Developer Zone site includes several tutorials covering introductory through advanced XML topics.

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