

XML and LINQ to XML

Objectives

In this chapter you'll:

- Mark up data using XML.
- Specify and validate an XML document's structure.
- Create and use simple XSL style sheets to render XML document data.
- Use the Document Object Model (DOM) to manipulate XML in C# programs.
- Use LINQ to XML to extract and manipulate data from XML documents.
- Create new XML documents using the classes provided by the .NET Framework.
- Work with XML namespaces in your C# code.
- Transform XML documents into XHTML using class `XsltCompiledTransform`.

22.1 Introduction	22.8 LINQ to XML: Document Object Model (DOM)
22.2 XML Basics	22.9 LINQ to XML Class Hierarchy
22.3 Structuring Data	22.10 LINQ to XML: Namespaces and Creating Documents
22.4 XML Namespaces	22.11 XSLT with Class <code>XsltCompiledTransform</code>
22.5 Document Type Definitions (DTDs)	22.12 Wrap-Up
22.6 W3C XML Schema Documents	
22.7 Extensible Stylesheet Language and XSL Transformations	

22.1 Introduction

In this chapter, we begin our introduction to XML. The .NET Framework uses XML extensively. Many of the configuration files that Visual Studio creates—such as those that represent project settings—use XML format. XML is also used heavily in serialization, like the Flickr web service you saw in Chapter 23. It’s also used in XAML—an XML *vocabulary* used for creating user interfaces built with Windows Presentation Foundation (WPF) and the Universal Windows Platform (UWP).

In this chapter, you’ll learn XML’s syntax and how to use XML namespaces. You’ll also learn how to create DTDs and schemas to validate your XML documents. Sections 22.7–22.11 demonstrate techniques for working with XML documents in C# applications. Visual C# provides language features and .NET Framework classes for working with XML. **LINQ to XML** provides a convenient way to manipulate data in XML documents using the same LINQ syntax you used on arrays and collections in Chapter 9. LINQ to XML also provides a set of classes for easily navigating and creating XML documents in your code.

22.2 XML Basics

The Extensible Markup Language was developed in 1996 by the **World Wide Web Consortium’s (W3C’s)** XML Working Group. XML is a widely supported standard for describing data that is commonly used to exchange that data between applications over the Internet. It permits document authors to create markup for virtually any type of information. This enables them to create entirely new markup languages for describing any type of data, such as mathematical formulas, software-configuration instructions, chemical molecular structures, music, news, recipes and financial reports. XML describes data in a way that both human beings and computers can understand.

Figure 22.1 is a simple XML document that describes information for a baseball player. We focus on lines 5–11 to introduce basic XML syntax. You’ll learn about the other elements of this document in Section 22.3.

XML documents contain text that represents content (that is, data), such as John (line 6), and **elements** that specify the document’s structure, such as `firstName` (line 6). XML documents delimit elements with **start tags** and **end tags**. A start tag consists of the element name in **angle brackets** (for example, `<player>` and `<firstName>` in lines 5 and 6,

```
1 <?xml version = "1.0"?>
2 <!-- Fig. 22.1: player.xml -->
3 <!-- Baseball player structured with XML -->
4
5 <player>
6   <firstName>John</firstName>
7
8   <lastName>Doe</lastName>
9
10  <battingAverage>0.375</battingAverage>
11 </player>
```

Fig. 22.1 | XML that describes a baseball player's information.

respectively). An end tag consists of the element name preceded by a **forward slash** (/) in angle brackets (for example, </firstName> and </player> in lines 6 and 11, respectively). An element's start and end tags enclose text that represents a piece of data (for example, the firstName of the player—John—in line 6, which is enclosed by the <firstName> start tag and </firstName> end tag) or other elements (for example, the firstName, lastName, and battingAverage elements in the player element). Every XML document must have exactly one **root element** that contains all the other elements. In Fig. 22.1, player (lines 5–11) is the root element.

Some XML-based markup languages include MathML (for mathematics), VoiceXML™ (for speech), CML (Chemical Markup Language—for chemistry) and XBRL (Extensible Business Reporting Language—for financial data exchange). ODF (Open Document Format—developed by Sun Microsystems) and OOXML (Office Open XML—developed by Microsoft as a replacement for the old proprietary Microsoft Office formats) are two competing standards for electronic office documents such as spreadsheets, presentations, and word processing documents. These markup languages are called XML **vocabularies** and provide a means for describing particular types of data in standardized, structured ways.

Massive amounts of data are currently stored on the Internet in a variety of formats (for example, databases, web pages, text files). Based on current trends, it's likely that much of this data, especially that which is passed between systems, will soon take the form of XML. Organizations see XML as the future of data encoding. Information-technology groups are planning ways to integrate XML into their systems. Industry groups are developing custom XML vocabularies for most major industries that will allow computer-based business applications to communicate in common languages. For example, web services allow web-based applications to exchange data seamlessly through standard protocols based on XML. Also, web services are described by an XML vocabulary called WSDL (Web Services Description Language).

The next generation of the Internet and World Wide Web is being built on a foundation of XML, which enables the development of more sophisticated web-based applications. XML allows you to assign meaning to what would otherwise be random pieces of data. As a result, programs can “understand” the data they manipulate. For example, a web browser might view a street address listed on a simple web page as a string of characters without any real meaning. In an XML document, however, this data can be clearly identified (that is, marked up) as an address. A program that uses the document can recognize

this data as an address and provide links to a map of that location, driving directions from that location or other location-specific information. Likewise, an application can recognize names of people, dates, ISBN numbers and any other type of XML-encoded data. Based on this data, the application can present users with other related information, providing a richer, more meaningful user experience.

Viewing and Modifying XML Documents

XML documents are portable. Viewing or modifying an XML document—a text file, usually with the `.xml` file-name extension—does not require special software, although many software tools exist, and new ones are frequently released that make it more convenient to develop XML-based applications. Most text editors can open XML documents for viewing and editing. Visual Studio includes an XML editor that provides *IntelliSense*. The editor also checks that the document is well formed and is valid if a schema (discussed shortly) is present. Also, most web browsers can display an XML document in a formatted manner that shows its structure. We demonstrate this using Microsoft's Edge web browser in Section 22.3. One important characteristic of XML is that it's both human readable and machine readable.

Processing XML Documents

Processing an XML document requires software called an **XML parser** (or **XML processor**). A parser makes the document's data available to applications. While reading the contents of an XML document, a parser checks that the document follows the syntax rules specified by the W3C's XML Recommendation (www.w3.org/XML). XML syntax requires a single root element, a start tag and end tag for each element and properly nested tags (that is, the end tag for a nested element must appear before the end tag of the enclosing element). Furthermore, XML is case sensitive, so the proper capitalization must be used in elements. A document that conforms to this syntax is a **well-formed XML document**, and is syntactically correct. We present fundamental XML syntax in Section 22.3. If an XML parser can process an XML document successfully, that XML document is well formed. Parsers can provide access to XML-encoded data in well-formed documents only—if a document is not well-formed, the parser will report an error to the user or calling application.

Often, XML parsers are built into software such as Visual Studio or available for download over the Internet. You can find a list of many popular XML parsers at

http://en.wikipedia.org/wiki/Category:XML_parsers

Validating XML Documents

An XML document can optionally reference a **Document Type Definition (DTD)** or a **W3C XML Schema** (referred to simply as a “schema” for the rest of this book) that defines the XML document's proper structure. When an XML document references a DTD or a schema, some parsers (called **validating parsers**) can use the DTD/schema to check that it has the appropriate structure. If the XML document conforms to the DTD/schema (that is, the document has the appropriate structure), the XML document is **valid**. For example, if in Fig. 22.1 we were referencing a DTD that specifies that a `player` element must have `firstName`, `lastName` and `battlingAverage` elements, then omitting the `lastName` element (line 8) would cause the XML document `player.xml` to be invalid. The

XML document would still be well formed, however, because it follows proper XML syntax (that is, it has one root element, and each element has a start and an end tag). By definition, a valid XML document is well formed. Parsers that cannot check for document conformity against DTDs/schemas are **nonvalidating parsers**—they determine only whether an XML document is well formed.

You'll learn more about validation, DTDs and schemas, as well as the key differences between these two types of structural specifications, later in this chapter. For now, schemas are XML documents themselves, whereas DTDs are not. As you'll learn, this difference presents several advantages in using schemas over DTDs.



Software Engineering Observation 22.1

DTDs and schemas are essential for business-to-business (B2B) transactions and mission-critical systems. Validating XML documents ensures that disparate systems can manipulate data structured in standardized ways and prevents errors caused by missing or malformed data.

Formatting and Manipulating XML Documents

XML documents contain only data, not formatting instructions, so applications that process XML documents must decide how to manipulate or display each document's data. For example, a PDA (personal digital assistant) may render an XML document differently than a wireless phone or a desktop computer. You can use **Extensible Stylesheet Language (XSL)** to specify rendering instructions for different platforms. We discuss XSL in Section 22.7.

XML-processing programs also can search, sort and manipulate XML data using technologies such as XSL. Some other XML-related technologies are XPath (XML Path Language—a language for accessing parts of an XML document), XSL-FO (XSL Formatting Objects—an XML vocabulary used to describe document formatting) and XSLT (XSL Transformations—a language for transforming XML documents into other documents). We present XSLT and XPath in Section 22.7. We'll also present new C# features that greatly simplify working with XML in your code. With these features, XSLT and similar technologies are not needed while coding in C#, but they remain relevant on platforms where C# and .NET are not available.

22.3 Structuring Data

In Fig. 22.2, we present an XML document that marks up a simple article using XML. The line numbers shown are for reference only and are not part of the XML document.

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.2: article.xml -->
3 <!-- Article structured with XML -->
4
5 <article>
6     <title>Simple XML</title>
7

```

Fig. 22.2 | XML used to mark up an article. (Part 1 of 2.)

```

8      <date>July 24, 2013</date>
9
10     <author>
11         <firstName>John</firstName>
12         <lastName>Doe</lastName>
13     </author>
14
15     <summary>XML is pretty easy.</summary>
16
17     <content>
18         In this chapter, we present a wide variety of examples that use XML.
19     </content>
20 </article>

```

Fig. 22.2 | XML used to mark up an article. (Part 2 of 2.)

This document begins with an **XML declaration** (line 1), which identifies the document as an XML document. The **version attribute** specifies the XML version to which the document conforms.

Some XML documents also specify an **encoding attribute** in the XML declaration. An encoding specifies how characters are stored in memory and on disk—historically, the way an uppercase "A" was stored on one computer architecture was different than the way it was stored on a different computer architecture. Unicode specifies encodings that can describe characters in any written language. An introduction to different encodings in XML can be found at the website bit.ly/EncodeXMLData.



Portability Tip 22.1

Documents should include the XML declaration to identify the version of XML used. A document that lacks an XML declaration might be assumed erroneously to conform to the latest version of XML—in which case, errors could result.



Common Programming Error 22.1

Placing whitespace characters before the XML declaration is an error.

XML comments (lines 2–3), which begin with `<!--` and end with `-->`, can be placed almost anywhere in an XML document. XML comments can span to multiple lines—an end marker on each line is not needed; the end marker can appear on a subsequent line, as long as there is exactly one end marker (`-->`) for each begin marker (`<!--`). Comments are used in XML for documentation purposes. Line 4 is a blank line. As in a C# program, blank lines, whitespaces and indentation are used in XML to improve readability. Later you'll see that the blank lines are normally ignored by XML parsers.



Common Programming Error 22.2

In an XML document, each start tag must have a matching end tag; omitting either tag is an error. Soon, you'll learn how such errors are detected.



Common Programming Error 22.3

XML is case sensitive. Using different cases for the start-tag and end-tag names for the same element is a syntax error.

In Fig. 22.2, `article` (lines 5–20) is the root element. The lines that precede the root element (lines 1–4) are the XML **prolog**. In an XML prolog, the XML declaration must appear before the comments and any other markup.

The elements we used in the example do not come from any specific markup language. Instead, we chose the element names and markup structure that best describe our particular data. You can invent whatever elements make sense for the particular data you’re dealing with. For example, element `title` (line 6) contains text that describes the article’s title (for example, `Simple XML`). Similarly, `date` (line 8), `author` (lines 10–13), `firstName` (line 11), `lastName` (line 12), `summary` (line 15) and `content` (lines 17–19) contain text that describes the date, author, the author’s first name, the author’s last name, a summary and the content of the document, respectively. XML element and attribute names can be of any length and may contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore, and they should not begin with “`xm1`” in any combination of uppercase and lowercase letters (for example, `XML`, `Xm1`, `xM1`), as this is reserved for use in the XML standards.



Common Programming Error 22.4

Using a whitespace character in an XML element name is an error.



Good Programming Practice 22.1

XML element names should be meaningful to humans and should not use abbreviations.

XML elements are **nested** to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create parent/child 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 22.5

Nesting XML tags improperly is a syntax error—it causes an XML document to not be well-formed. For example, `<x><y>hello</x></y>` is an error, because the `</y>` tag must precede the `</x>` tag.

Any element that contains other elements (for example, `article` or `author`) is a **container element**. Container elements also are called **parent elements**. Elements nested inside a container element are **child elements** (or children) of that container element.

Viewing an XML Document in Microsoft Edge

The XML document in Fig. 22.2 is simply a text file named `article.xml`. This document does not contain formatting information for the article. The reason is that XML is a technology for describing the structure of data. Formatting and displaying data from an XML document are application-specific issues. For example, when you load `article.xml` in Edge, it parses and displays the document’s data using a built-in **style sheet** to format the XML. The resulting format (Fig. 22.3) is similar to the code listing in Fig. 22.2. In Section 22.7, we show how to create style sheets to transform your XML data into various formats suitable for display.

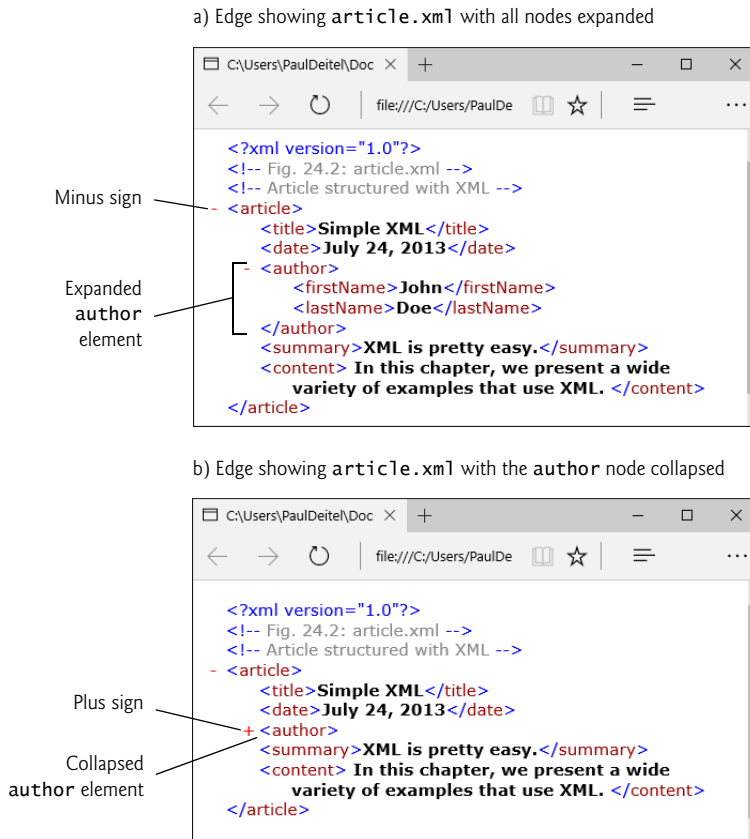


Fig. 22.3 | `article.xml` displayed by Microsoft Edge on Windows 10.

Note the minus sign (–) and plus sign (+) in the screenshots of Fig. 22.3. Although these symbols are not part of the XML document, Edge places them next to every container element. A minus sign indicates that a container element is expanded and its child elements are displayed. Clicking the minus sign next to an element collapses that element (that is, causes Edge to hide the container element’s children and replace the minus sign with a plus sign). Conversely, clicking the plus sign next to an element expands that element (that is, causes Edge to display the container element’s children and replace the plus sign with a minus sign). This behavior is similar to viewing the directory structure using File Explorer (Windows 8 and higher) or Windows Explorer (Windows 7). In fact, a directory structure often is modeled as a series of tree structures, in which the **root** of a tree represents a drive letter (for example, C:), and **nodes** in the tree represent directories. Parsers often store XML data as tree structures to facilitate efficient manipulation.

XML Markup for a Business Letter

Now that we have seen a simple XML document, let’s examine a more complex one that marks up a business letter (Fig. 22.4). Again, we begin the document with the XML declaration (line 1) that states the XML version to which the document conforms.

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.4: letter.xml -->
3 <!-- Business letter marked up as XML -->
4
5 <!DOCTYPE letter SYSTEM "letter.dtd">
6
7 <letter>
8   <contact type = "sender">
9     <name>Jane Doe</name>
10    <address1>Box 12345</address1>
11    <address2>15 Any Ave.</address2>
12    <city>Othertown</city>
13    <state>Otherstate</state>
14    <zip>67890</zip>
15    <phone>555-4321</phone>
16    <flag gender = "F" />
17  </contact>
18
19  <contact type = "receiver">
20    <name>John Doe</name>
21    <address1>123 Main St.</address1>
22    <address2></address2>
23    <city>Anytown</city>
24    <state>Anystate</state>
25    <zip>12345</zip>
26    <phone>555-1234</phone>
27    <flag gender = "M" />
28  </contact>
29
30  <salutation>Dear Sir:</salutation>
31
32  <paragraph>It is our privilege to inform you about our new database
33    managed with XML. This new system allows you to reduce the
34    load on 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 website for availability
39    and pricing.
40  </paragraph>
41
42  <closing>Sincerely,</closing>
43  <signature>Ms. Jane Doe</signature>
44 </letter>

```

Fig. 22.4 | Business letter marked up as XML.

Line 5 specifies that this XML document references a DTD. Recall from Section 22.2 that DTDs define the structure of the data for an XML document. For example, a DTD specifies the elements and parent/child relationships between elements permitted in an XML document.



Error-Prevention Tip 22.1

An XML document is not required to reference a DTD, but validating XML parsers can use a DTD to ensure that the document has the proper structure.



Portability Tip 22.2

Validating an XML document helps guarantee that independent developers will exchange data in a standardized form that conforms to the DTD.

The DTD reference (line 5) contains three items: the name of the root element that the DTD specifies (`letter`); the keyword **SYSTEM** (which denotes an **external DTD**—a DTD declared in a separate file, as opposed to a DTD declared locally in the same file); and the DTD’s name and location (that is, `letter.dtd` in the same directory as the XML document). DTD document file names typically end with the `.dtd` extension. We discuss DTDs and `letter.dtd` in detail in Section 22.5.

Root element `letter` (lines 7–44 of Fig. 22.4) contains the child elements `contact`, `contact`, `salutation`, `paragraph`, `paragraph`, `closing` and `signature`. Besides being placed between tags, data also can be placed in **attributes**—name/value pairs that appear within the angle brackets of start tags. Elements can have any number of attributes (separated by spaces) in their start tags, provided all the attribute names are unique. The first `contact` element (lines 8–17) has an attribute named `type` with **attribute value** `"sender"`, which indicates that this `contact` element identifies the letter’s sender. The second `contact` element (lines 19–28) has attribute `type` with value `"receiver"`, which indicates that this `contact` element identifies the letter’s recipient. Like element names, attribute names are case sensitive, can be of any length, may contain letters, digits, underscores, hyphens and periods, and must begin with either a letter or an underscore character. A `contact` element stores various items of information about a contact, such as the contact’s name (represented by element `name`), address (represented by elements `address1`, `address2`, `city`, `state` and `zip`), phone number (represented by element `phone`) and gender (represented by attribute `gender` of element `flag`). Element `salutation` (line 30) marks up the letter’s salutation. Lines 32–40 mark up the letter’s body using two `paragraph` elements. Elements `closing` (line 42) and `signature` (line 43) mark up the closing sentence and the author’s “signature,” respectively.



Common Programming Error 22.6

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

Line 16 introduces the **empty element** `flag`. An empty element contains no content. However, it may sometimes contain data in the form of attributes. Empty element `flag` contains an attribute that indicates the gender of the contact (represented by the parent `contact` element). Document authors can close an empty element either by placing a slash immediately preceding the rightmost angle bracket, as shown in line 16, or by explicitly writing an end tag, as in line 22:

```
<address2></address2>
```

Line 22 also can be written as:

```
<address2/>
```

The `address2` element in line 22 is empty, because there is no second part to this contact's address. However, we must include this element to conform to the structural rules specified in the XML document's DTD—`letter.dtd` (which we present in Section 22.5). This DTD specifies that each `contact` element must have an `address2` child element (even if it's empty). In Section 22.5, you'll learn how DTDs indicate that certain elements are required while others are optional.

22.4 XML Namespaces

XML allows you to create custom elements. This extensibility can result in **naming collisions**—elements with identical names that represent different things—when combining content from multiple sources. For example, we may use the element `book` to mark up data about a Deitel publication. A stamp collector may use the element `book` to mark up data about a book of stamps. Using both of these elements in the same document could create a naming collision, making it difficult to determine which kind of data each element contains.

An XML **namespace** is a collection of element and attribute names. Like C# namespaces, XML namespaces provide a means for document authors to unambiguously refer to elements that have the same name (that is, prevent collisions). For example,

```
<subject>Math</subject>
```

and

```
<subject>Cardiology</subject>
```

use element `subject` to mark up data. In the first case, the subject is something one studies in school, whereas in the second case, the subject is a field of medicine. Namespaces can differentiate these two `subject` elements. For example,

```
<school:subject>Math</school:subject>
```

and

```
<medical:subject>Cardiology</medical:subject>
```

Both `school` and `medical` are **namespace prefixes**. A document author places a namespace prefix and colon (`:`) before an element name to specify the namespace to which that element belongs. Document authors can create their own namespace prefixes using virtually any name except the reserved namespace prefixes `xml` and `xmlns`. In the subsections that follow, we demonstrate how document authors ensure that namespaces are unique.



Common Programming Error 22.7

Attempting to create a namespace prefix named `xml` in any mixture of uppercase and lowercase letters is a syntax error—the `xml` namespace is reserved for internal use by XML itself.

Differentiating Elements with Namespaces

Figure 22.5 uses namespaces to differentiate two distinct elements—the `file` element related to a text file and the `file` document related to an image file.

Lines 6–7 use the XML-namespace reserved attribute `xmlns` to create two namespace prefixes—`text` and `image`. Creating a namespace prefix is similar to using a `using` statement in C#—it allows you to access XML elements from a given namespace. Each name-

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.5: namespace.xml -->
3 <!-- Demonstrating namespaces -->
4
5 <text:directory
6     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 </text:directory>

```

Fig. 22.5 | XML namespaces demonstration.

space 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. A URI is a way to identify a resource, typically on the Internet. Two popular types of URI are **Uniform Resource Name (URN)** and **Uniform Resource Locator (URL)**.

To ensure that namespaces are unique, document authors must provide unique URIs. We chose `urn:deitel:textInfo` and `urn:deitel:imageInfo` as URIs. These URIs employ the URN scheme frequently used to identify namespaces. Under this naming scheme, a URI begins with "urn:", followed by a unique series of additional names separated by colons. These URIs are not guaranteed to be unique—the idea is simply that creating a long URI in this way makes it unlikely that two authors will use the same namespace.

Another common practice is to use URLs, which specify the location of a file or a resource on the Internet. For example, `http://www.deitel.com` is the URL that identifies the home page of the Deitel & Associates website. Using URLs for domains that you own guarantees that the namespaces are unique, because the domain names (for example, `www.deitel.com`) are guaranteed to be unique. For example, lines 5–7 could be rewritten as

```

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

```

where URLs related to the Deitel & Associates, Inc. domain name serve as URIs to identify the `text` and `image` namespaces. The parser does not visit these URLs, nor do these URLs need to refer to actual web pages. Each simply represents a unique series of characters used to differentiate URI names. In fact, any string can represent a namespace. For example, our `image` namespace URI could be `hgjfkdl1sa4556`, in which case our prefix assignment would be

```

xmlns:image = "hgjfkdl1sa4556"

```

Lines 9–11 use the text namespace prefix for elements `file` and `description`. The end tags must also specify the namespace prefix `text`. Lines 13–16 apply namespace prefix `image` to the elements `file`, `description` and `size`. Attributes do not require namespace prefixes, because each attribute is already part of an element that specifies the namespace prefix. For example, attribute `filename` (line 9) is already uniquely identified by being in the context of the `file` start tag, which is prefixed with `text`.

Specifying a Default Namespace

To eliminate the need to place namespace prefixes in each element, document authors may specify a **default namespace** for an element and its children. Figure 22.6 demonstrates using a default namespace (`urn:deitel:textInfo`) for element `directory`.

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.6: defaultnamespace.xml -->
3 <!-- Using default namespaces -->
4
5 <directory xmlns = "urn:deitel:textInfo"
6   xmlns:image = "urn:deitel:imageInfo">
7
8   <file filename = "book.xml">
9     <description>A book list</description>
10  </file>
11
12  <image:file filename = "funny.jpg">
13    <image:description>A funny picture</image:description>
14    <image:size width = "200" height = "100" />
15  </image:file>
16 </directory>

```

Fig. 22.6 | Default namespace demonstration.

Line 5 defines a default namespace using attribute `xmlns` with a URI as its value. Once we define this default namespace, child elements which do not specify a prefix belong to the default namespace. Thus, element `file` (lines 8–10) is in the default namespace `urn:deitel:textInfo`. Compare this to lines 9–11 of Fig. 22.5, where we had to prefix the `file` and `description` element names with the namespace prefix `text`.



Common Programming Error 22.8

The default namespace can be overridden at any point in the document with another `xmlns` attribute. All direct and indirect children of the element with the `xmlns` attribute use the new default namespace.

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 lines 12–15 includes the `image` namespace prefix, indicating that this element is in the `urn:deitel:imageInfo` namespace, not the default namespace.

Namespaces in XML Vocabularies

XML-based languages, such as XML Schema and Extensible Stylesheet Language (XSL), often use namespaces to identify their elements. Each vocabulary defines special-purpose elements that are grouped in namespaces. These namespaces help prevent naming collisions between predefined and user-defined elements.

22.5 Document Type Definitions (DTDs)

Document Type Definitions (DTDs) are one of two technologies you can use to specify XML document structure. Section 22.6 presents W3C XML Schema documents, which provide an improved method of specifying XML document structure.

**Software Engineering Observation 22.2**

XML documents can have many different structures, and for this reason an application cannot be certain whether a particular document it receives is complete, ordered properly, and not missing data. DTDs and schemas (Section 22.6) solve this problem by providing an extensible way to describe XML document structure. Applications should use DTDs or schemas to confirm whether XML documents are valid.

**Software Engineering Observation 22.3**

*Many organizations and individuals are creating DTDs and schemas for a broad range of applications. These collections—called **repositories**—are available free for download from the web (e.g., www.xml.org, www.oasis-open.org).*

Creating a Document Type Definition

Figure 22.4 presented a simple business letter marked up with XML. Recall that line 5 of `letter.xml` references a DTD—`letter.dtd` (Fig. 22.7). This DTD specifies the business letter's element types and attributes and their relationships to one another.

```

1  <!-- Fig. 22.7: 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">

```

Fig. 22.7 | Document Type Definition (DTD) for a business letter. (Part I of 2.)

```

20
21 <!ELEMENT salutation (#PCDATA)>
22 <!ELEMENT closing (#PCDATA)>
23 <!ELEMENT paragraph (#PCDATA)>
24 <!ELEMENT signature (#PCDATA)>

```

Fig. 22.7 | Document Type Definition (DTD) for a business letter. (Part 2 of 2.)

A DTD describes an XML document's structure and enables an XML parser to verify whether a document is valid (i.e., whether its elements contain the proper attributes and appear in the proper sequence). DTDs allow users to check document structure and to exchange data in a standardized format. A DTD expresses the set of rules for document structure by specifying what attributes and other elements may appear inside a given element.



Common Programming Error 22.9

For documents validated with DTDs, any document that uses elements, attributes or relationships not explicitly defined by a DTD is an invalid document.

Defining Elements in a DTD

The **ELEMENT element type declaration** in Fig. 22.7, lines 4–5 defines the 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 the DTD allows one or more occurrences of an element. Other occurrence indicators include the **asterisk (*)**, which indicates an *optional* element that can occur *zero or more* times, and the **question mark (?)**, which indicates an optional element that can occur at most once (i.e., zero or one occurrence). If an element does not have an occurrence indicator, the DTD allows exactly one occurrence.

The contact element type declaration (lines 7–8) specifies that a contact element contains child elements name, address1, address2, city, state, zip, phone and flag—in that order. The DTD requires exactly one occurrence of each of these elements.

Defining Attributes in a DTD

Line 9 uses the **ATTLIST attribute-list declaration** to define an attribute named type for the contact element. Keyword **#IMPLIED** specifies that the type attribute of the contact element is *optional*—a missing type attribute will not invalidate the document. Other keywords that can be used in place of **#IMPLIED** in an ATTLIST declaration include **#REQUIRED** and **#FIXED**. Keyword **#REQUIRED** specifies that the attribute must be present in the element, and keyword **#FIXED** specifies that the attribute (if present) must have the given fixed value. For example,

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

indicates that attribute zip (if present in element address) must have the value 01757 for the document to be valid. If the attribute is not present, then the parser, by default, uses the fixed value that the ATTLIST declaration specifies. You can supply a default value in-

stead of one of these keywords. Doing so makes the attribute optional, but the default value will be used if the attribute's value is not specified.

Character Data vs. Parsed Character Data

Keyword **CDATA** (line 9) specifies that attribute type contains **character data** (i.e., a string). A parser will pass such data to an application without modification.



Software Engineering Observation 22.4

DTD syntax cannot describe an element's (or attribute's) type. For example, a DTD cannot specify that a particular element or attribute can contain only integer data.

Keyword **#PCDATA** (line 11) specifies that an element (e.g., `name`) may contain **parsed character data** (i.e., data that is processed by an XML parser). Elements with parsed character data cannot contain markup characters, such as less than (`<`), greater than (`>`) or ampersand (`&`). The document author should replace any markup character in a **#PCDATA** element with the character's corresponding **character entity reference**. For example, the character entity reference `<` should be used in place of the less-than symbol (`<`), and the character entity reference `>` should be used in place of the greater-than symbol (`>`). A document author who wishes to use a literal ampersand should use the entity reference `&`; instead—parsed character data can contain ampersands (`&`) only for inserting entities. The final two entities defined by XML are `'` and `"`, representing the single (`'`) and double (`"`) quote characters, respectively.



Common Programming Error 22.10

Using markup characters (e.g., `<`, `>` and `&`) in parsed character data is an error. Use character entity references (e.g., `<`, `>` and `&` instead).

Defining Empty Elements in a DTD

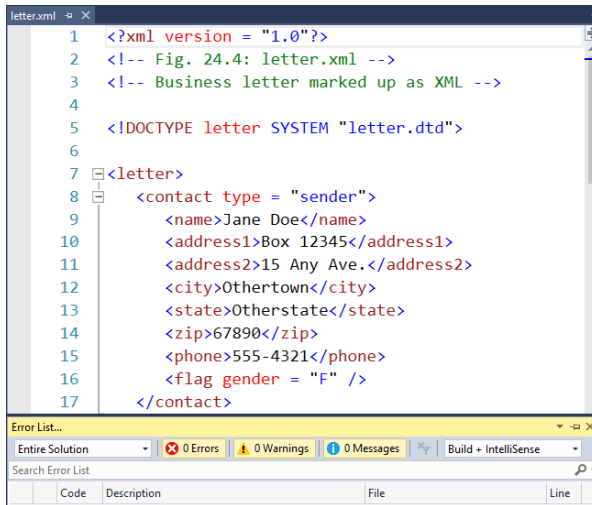
Line 18 defines an empty element named `flag`. Keyword **EMPTY** specifies that the element does not contain any data between its start and end tags. Empty elements commonly describe data via attributes. For example, `flag`'s data appears in its `gender` attribute (line 19). Line 19 specifies that the `gender` attribute's value must be one of the enumerated values (`M` or `F`) enclosed in parentheses and delimited by a vertical bar (`|`) meaning "or." Line 19 also indicates that `gender` has a default value of `M`.

Well-Formed Documents vs. Valid Documents

Recall that a well-formed document is *syntactically correct* (i.e., each start tag has a corresponding end tag, the document contains only one root element, and so on), and a valid document contains the proper elements with the proper attributes in the proper sequence. An XML document cannot be valid unless it's well formed.

Visual Studio can validate XML documents against both DTDs and schemas. You do not have to create a project to use this facility—simply open the XML file in Visual Studio as in Fig. 22.8. If the DTD or schema referenced in the XML document can be retrieved, Visual Studio will automatically validate the XML. If the XML file does not validate, Visual Studio will display a warning just as it does with errors in your C# code. Visit www.w3.org/XML/Schema for a list of additional validation tools.

a) No errors or warnings in the Error List window indicate that this XML document validated properly



b) The XML document did not validate properly because the value for the `gender` attribute (line 16) as invalid

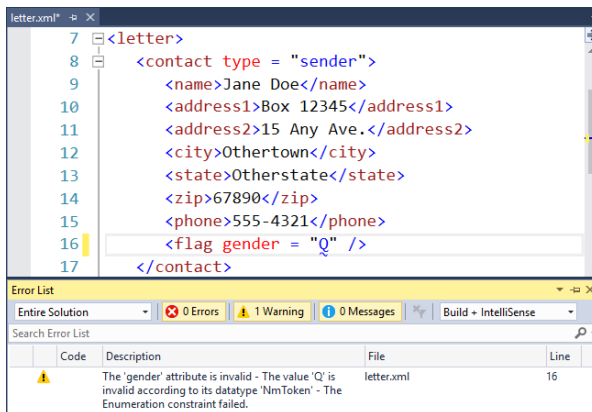


Fig. 22.8 | XML file open in Visual Studio.

22.6 W3C XML Schema Documents

In this section, we introduce schemas for specifying XML document structure and validating XML documents. Many developers in the XML community believe that DTDs are not flexible enough to meet today's programming needs. For example, DTDs lack a way of indicating what specific type of data (e.g., numeric, text) an element can contain, and DTDs are not themselves XML documents, which makes it difficult to manipulate them programmatically. These and other limitations have led to the development of schemas.

Unlike DTDs, schemas use XML syntax and are actually XML documents that programs can manipulate. Like DTDs, schemas are used by validating parsers to validate documents.

In this section, we focus on the W3C's **XML Schema** vocabulary. For the latest information on XML Schema, visit www.w3.org/XML/Schema. For tutorials on XML Schema concepts beyond what we present here, visit www.w3schools.com/schema/default.asp.

A DTD describes an XML document's structure, not the content of its elements. For example,

```
<quantity>5</quantity>
```

contains character data. If the document that contains element `quantity` references a DTD, an XML parser can validate the document to confirm that this element indeed does contain PCDATA content. However, the parser cannot validate that the content is numeric; DTDs do not provide this capability. So, unfortunately, the parser also considers

```
<quantity>hello</quantity>
```

to be valid. An application that uses the XML document containing this markup should test that the data in element `quantity` is numeric and take appropriate action if it's not.

XML Schema enables schema authors to specify that element `quantity`'s data must be numeric or, even more specifically, an integer. A parser validating the XML document against this schema can determine that `5` conforms and `hello` does not. An XML document that conforms to a schema document is **schema valid**, and one that does not conform is **schema invalid**. Schemas are XML documents and also can be validated.

Validating Against an XML Schema Document

Figure 22.9 shows a schema-valid XML document named `book.xml`, and Fig. 22.10 shows the pertinent XML Schema document (`book.xsd`) that defines the structure for `book.xml`. By convention, schemas use the `.xsd` extension. Recall that Visual Studio can perform schema validation if it can locate the schema document. Visual Studio can locate a schema if it's specified in the XML document, is in the same solution or is simply open in Visual Studio at the same time as the XML document. To validate the schema document itself (i.e., `book.xsd`) and produce the output shown in Fig. 22.10, we used an online XML Schema validator at

```
http://www.validome.com/grammar/
```

Figure 22.9 contains markup describing several books. The `books` element (line 5) has the namespace prefix `deitel` (declared in line 5), indicating that the `books` element is a part of the namespace `http://www.deitel.com/booklist`.

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.9: book.xml -->
3 <!-- Book list marked up as XML -->
4
5 <deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
6   <book>
7     <title>Visual Basic How to Program</title>
8   </book>
9

```

Fig. 22.9 | Schema-valid XML document describing a list of books. (Part I of 2.)

```

10     <book>
11       <title>Visual C# How to Program</title>
12     </book>
13
14     <book>
15       <title>Java How to Program</title>
16     </book>
17
18     <book>
19       <title>C++ How to Program</title>
20     </book>
21
22     <book>
23       <title>Internet and World Wide Web How to Program</title>
24     </book>
25 </deitel:books>

```

Fig. 22.9 | Schema-valid XML document describing a list of books. (Part 2 of 2.)

Creating an XML Schema Document

Figure 22.10 presents the XML Schema document that specifies the structure of `book.xml` (Fig. 22.9). This document defines an XML-based language (i.e., a vocabulary) for writing XML documents about collections of books. The schema defines the elements, attributes and parent-child relationships that such a document can (or must) include. The schema also specifies the type of data that these elements and attributes may contain.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 22.10: book.xsd          -->
3  <!-- Simple W3C XML Schema document -->
4
5  <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6         xmlns:deitel = "http://www.deitel.com/booklist"
7         targetNamespace = "http://www.deitel.com/booklist">
8
9     <element name = "books" type = "deitel:BooksType"/>
10
11     <complexType name = "BooksType">
12       <sequence>
13         <element name = "book" type = "deitel:SingleBookType"
14             minOccurs = "1" maxOccurs = "unbounded"/>
15       </sequence>
16     </complexType>
17
18     <complexType name = "SingleBookType">
19       <sequence>
20         <element name = "title" type = "string"/>
21       </sequence>
22     </complexType>
23 </schema>

```

Fig. 22.10 | XML Schema document for `book.xml`.

Root element **schema** (Fig. 22.10, lines 5–23) contains elements that define the structure of an XML document such as `book.xml`. Line 5 specifies as the default namespace the standard W3C XML Schema namespace URI—<http://www.w3.org/2001/XMLSchema>. This namespace contains predefined elements (e.g., root element `schema`) that comprise the XML Schema vocabulary—the language used to write an XML Schema document.



Portability Tip 22.3

W3C XML Schema authors specify URI <http://www.w3.org/2001/XMLSchema> when referring to the XML Schema namespace. This namespace contains predefined elements that comprise the XML Schema vocabulary. Specifying this URI ensures that validation tools correctly identify XML Schema elements and do not confuse them with those defined by document authors.

Line 6 binds the URI <http://www.deitel.com/booklist> to namespace prefix `deitel`. As we discuss momentarily, the schema uses this namespace to differentiate names created by us from names that are part of the XML Schema namespace. Line 7 also specifies <http://www.deitel.com/booklist> as the **targetNamespace** of the schema. This attribute identifies the namespace of the XML vocabulary that this schema defines. The `targetNamespace` of `book.xsd` is the same as the namespace referenced in line 5 of `book.xml` (Fig. 22.9). This is what “connects” the XML document with the schema that defines its structure. When an XML schema validator examines `book.xml` and `book.xsd`, it will recognize that `book.xml` uses elements and attributes from the <http://www.deitel.com/booklist> namespace. The validator also will recognize that this namespace is the one defined in `book.xsd` (i.e., the schema’s `targetNamespace`). Thus the validator knows where to look for the structural rules for the elements and attributes used in `book.xml`.

Defining an Element in XML Schema

In XML Schema, the **element** tag (Fig. 22.10, line 9) defines an element to be included in an XML document that conforms to the schema. In other words, `element` specifies the actual *elements* that can be used to mark up data. Line 9 defines the `books` element, which we use as the root element in `book.xml` (Fig. 22.9). Attributes **name** and **type** specify the `element`’s name and type, respectively. An element’s `type` attribute indicates the data type that the element may contain. Possible types include XML Schema–defined types (e.g., `string`, `double`) and user-defined types (e.g., `BooksType`, which is defined in lines 11–16). Figure 22.11 lists several of XML Schema’s many built-in types. For a complete list of built-in types, see Section 3 of the specification found at www.w3.org/TR/xmlschema-2.

In this example, `books` is defined as an element of type `deitel:BooksType` (line 9). `BooksType` is a user-defined type (Fig. 22.10, lines 11–16) in the <http://www.deitel.com/booklist> namespace and therefore must have the namespace prefix `deitel`. It’s not an existing XML Schema type. Two categories of types exist in XML Schema—**simple types** and **complex types**. They differ only in that simple types cannot contain attributes or child elements and complex types can.

A user-defined type that contains attributes or child elements must be defined as a complex type. Lines 11–16 use element **complexType** to define `BooksType` as a complex type that has a child element named `book`. The sequence element (lines 12–15) allows you to specify the sequential order in which child elements must appear. The `element` (lines 13–14) nested within the `complexType` element indicates that a `BooksType` element (e.g.,

books) can contain child elements named `book` of type `deitel:SingleBookType` (defined in lines 18–22). Attribute `minOccurs` (line 14), with value 1, specifies that elements of type `BooksType` must contain a minimum of one book element. Attribute `maxOccurs` (line 14), with value `unbounded`, specifies that elements of type `BooksType` may have any number of book child elements. Both of these attributes have default values of 1.

Lines 18–22 define the complex type `SingleBookType`. An element of this type contains a child element named `title`. Line 20 defines element `title` to be of simple type `string`. Recall that elements of a simple type cannot contain attributes or child elements. The schema end tag (`</schema>`, line 23) declares the end of the XML Schema document.

A Closer Look at Types in XML Schema

Every element in XML Schema has a type. Types include the *built-in types* provided by XML Schema (Fig. 22.11) or *user-defined types* (e.g., `SingleBookType` in Fig. 22.10).

Type	Description	Ranges or structures	Examples
<code>string</code>	A character string		hello
<code>boolean</code>	True or false	<code>true</code> , <code>false</code>	<code>true</code>
<code>decimal</code>	A decimal numeral	$i * (10^n)$, where i is an integer and n is an integer that is less than or equal to zero	5, -12, -45.78
<code>float</code>	A floating-point number	$m * (2^e)$, where m is an integer whose absolute value is less than 2^{24} and e is an integer in the range -149 to 104. Plus three additional numbers: positive infinity (INF), negative infinity (-INF) and not-a-number (NaN)	0, 12, -109.375, NaN
<code>double</code>	A floating-point number	$m * (2^e)$, where m is an integer whose absolute value is less than 2^{53} and e is an integer in the range -1075 to 970. Plus three additional numbers: positive infinity, negative infinity and not-a-number	0, 12, -109.375, NaN
<code>long</code>	A whole number	-9223372036854775808 to 9223372036854775807, inclusive	1234567890, -1234567890
<code>int</code>	A whole number	-2147483648 to 2147483647, inclusive	1234567890, -1234567890
<code>short</code>	A whole number	-32768 to 32767, inclusive	12, -345
<code>date</code>	A date consisting of a year, month and day	yyyy-mm with an optional dd and an optional time zone, where yyyy is four digits and mm and dd are two digits. The time zone is specified as +hh:mm or -hh:mm, giving an offset in hours and minutes	2013-07-25+01:00
<code>time</code>	A time consisting of hours, minutes and seconds	hh:mm:ss with an optional time zone, where hh, mm and ss are two digits long	16:30:25-05:00

Fig. 22.11 | Some XML Schema types.

Every *simple type* defines a **restriction** on an XML Schema-defined type or a restriction on a user-defined type. Restrictions limit the possible values that an element can hold.

Complex types are divided into two groups—those with **simple content** and those with **complex content**. Both can contain attributes, but only complex content can contain child elements. Complex types with simple content must extend or restrict some other existing type. Complex types with complex content do not have this limitation. We demonstrate complex types with each kind of content in the next example.

XML Schema with Simple and Complex Types

The schema in Fig. 22.12 creates simple types and complex types. The XML document in Fig. 22.13 (laptop.xml) follows the structure defined in Fig. 22.12 to describe parts of a laptop computer. A document such as laptop.xml that conforms to a schema is known as an **XML instance document**—the document is an instance (i.e., example) of the schema.

```

1  <?xml version = "1.0"?>
2  <!-- Fig. 22.12: computer.xsd -->
3  <!-- W3C XML Schema document -->
4
5  <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6    xmlns:computer = "http://www.deitel.com/computer"
7    targetNamespace = "http://www.deitel.com/computer">
8
9    <simpleType name = "gigahertz">
10     <restriction base = "decimal">
11       <minInclusive value = "2.1"/>
12     </restriction>
13   </simpleType>
14
15   <complexType name = "CPU">
16     <simpleContent>
17       <extension base = "string">
18         <attribute name = "model" type = "string"/>
19       </extension>
20     </simpleContent>
21   </complexType>
22
23   <complexType name = "portable">
24     <all>
25       <element name = "processor" type = "computer:CPU"/>
26       <element name = "monitor" type = "int"/>
27       <element name = "CPUSpeed" type = "computer:gigahertz"/>
28       <element name = "RAM" type = "int"/>
29     </all>
30     <attribute name = "manufacturer" type = "string"/>
31   </complexType>
32
33   <element name = "laptop" type = "computer:portable"/>
34 </schema>

```

Fig. 22.12 | XML Schema document defining simple and complex types.

Opening schema Tag

Line 5 (Fig. 22.12) declares the default namespace as the standard XML Schema namespace—any elements without a prefix are assumed to be in the XML Schema namespace. Line 6 binds the namespace prefix `computer` to the namespace `http://www.deitel.com/computer`. Line 7 identifies this namespace as the `targetNamespace`—the namespace being defined by the current XML Schema document.

simpleType Element gigahertz

To design the XML elements for describing laptop computers, we first create a simple type in lines 9–13 using the `simpleType` element. We name this `simpleType` `gigahertz` because it will be used to describe the clock speed of the processor in gigahertz. Simple types are restrictions of a type typically called a **base type**. For this `simpleType`, line 10 declares the base type as `decimal`, and we restrict the value to be at least 2.1 by using the `minInclusive` element in line 11.

complexType Element CPU

Next, we declare a `complexType` named `CPU` that has `simpleContent` (lines 16–20). Remember that a complex type with simple content can have attributes but not child elements. Also recall that complex types with simple content must extend or restrict some XML Schema type or user-defined type. The `extension` element with attribute `base` (line 17) sets the base type to `string`. In this `complexType`, we extend the base type `string` with an attribute. The `attribute` element (line 18) gives the `complexType` an attribute of type `string` named `model`. Thus an element of type `CPU` must contain `string` text (because the base type is `string`) and may contain a `model` attribute that is also of type `string`.

complexType Element portable

Last, we define type `portable`, which is a `complexType` with complex content (lines 23–31). Such types are allowed to have child elements and attributes. The element `all` (lines 24–29) encloses elements that must each be included once in the corresponding XML instance document. These elements can be included in any order. This complex type holds four elements—`processor`, `monitor`, `CPUSpeed` and `RAM`. They're given types `CPU`, `int`, `gigahertz` and `int`, respectively. When using types `CPU` and `gigahertz`, we must include the namespace prefix `computer`, because these user-defined types are part of the `computer` namespace (`http://www.deitel.com/computer`)—the namespace defined in the current document (line 7). Also, `portable` contains an attribute defined in line 30. The `attribute` element indicates that elements of type `portable` contain an attribute of type `string` named `manufacturer`.

Line 33 declares the actual element that uses the three types defined in the schema. The element is called `laptop` and is of type `portable`. We must use the namespace prefix `computer` in front of `portable`.

Using the laptop Element

We have now created an element named `laptop` that contains child elements `processor`, `monitor`, `CPUSpeed` and `RAM`, and an attribute `manufacturer`. Figure 22.13 uses the `laptop` element defined in the `computer.xsd` schema. We used Visual Studio's built-in schema validation to ensure that this XML instance document adheres to the schema's structural rules.

```
1 <?xml version = "1.0"?>
2 <!-- Fig. 22.13: laptop.xml -->
3 <!-- Laptop components marked up as XML -->
4
5 <computer:laptop xmlns:computer = "http://www.deitel.com/computer"
6     manufacturer = "IBM">
7
8     <processor model = "Centrino">Intel</processor>
9     <monitor>17</monitor>
10    <CPUSpeed>2.4</CPUSpeed>
11    <RAM>256</RAM>
12 </computer:laptop>
```

Fig. 22.13 | XML document using the `Laptop` element defined in `computer.xsd`.

Line 5 declares namespace prefix `computer`. The `laptop` element requires this prefix because it's part of the `http://www.deitel.com/computer` namespace. Line 6 sets the laptop's manufacturer attribute, and lines 8–11 use the elements defined in the schema to describe the laptop's characteristics.

Automatically Creating Schemas using Visual Studio

Visual Studio can create a schema from an existing XML document, using the document as a template. With an XML document open, select **XML > Create Schema**. A new schema file opens that conforms to the standards of the XML document. You can then save it and add it to the project. The schema generated by Visual Studio is a good starting point, but you should refine the restrictions and types it specifies so they're appropriate for your XML documents.

22.7 Extensible Stylesheet Language and XSL Transformations

Extensible Stylesheet Language (XSL) documents specify how programs are to render XML document data. XSL is a group of three technologies—**XSL-FO (XSL Formatting Objects)**, **XPath (XML Path Language)** and **XSLT (XSL Transformations)**. XSL-FO is a vocabulary for specifying formatting, and XPath is a string-based language of expressions used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents.

The third portion of XSL—XSL Transformations (XSLT)—is a technology for transforming XML documents into other documents—i.e., transforming the structure of the XML document data to another structure. XSLT provides elements that define rules for transforming one XML document to produce a different XML document. This is useful when you want to use data in multiple applications or on multiple platforms, each of which may be designed to work with documents written in a particular vocabulary. For example, XSLT allows you to convert a simple XML document to an **XHTML (Extensible HyperText Markup Language)** document that presents the XML document's data (or a subset of the data) formatted for display in a web browser. (See Fig. 22.14 for a sample “before” and “after” view of such a transformation.)

Transforming an XML document using XSLT involves two tree structures—the **source tree** (i.e., the XML document to be transformed) and the **result tree** (i.e., the XML document to be created). XPath is used to locate parts of the source-tree document that match **templates** defined in an **XSL style sheet**. When a match occurs (i.e., a node matches a template), the matching template executes and adds its result to the result tree. When there are no more matches, XSLT has transformed the source tree into the result tree. The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using XSLT’s `select` and `match` attributes. For XSLT to function, the source tree must be properly structured. Schemas, DTDs and validating parsers can validate document structure before using XPath and XSLTs.

A Simple XSL Example

Figure 22.14 lists an XML document that describes various sports. The output shows the result of the transformation (specified in the XSLT template of Fig. 22.15) rendered by Internet Explorer. Right click with the page open in Edge and select **View Source** to view the generated XHTML.

```

1  <?xml version = "1.0"?>
2  <?xml-stylesheet type = "text/xsl" href = "sports.xsl"?>
3
4  <!-- Fig. 22.14: sports.xml -->
5  <!-- Sports Database -->
6
7  <sports>
8      <game id = "783">
9          <name>Cricket</name>
10
11         <paragraph>
12             More popular among Commonwealth nations.
13         </paragraph>
14     </game>
15
16     <game id = "239">
17         <name>Baseball</name>
18
19         <paragraph>
20             More popular in America.
21         </paragraph>
22     </game>
23
24     <game id = "418">
25         <name>Soccer (Futbol)</name>
26
27         <paragraph>
28             Most popular sport in the world.
29         </paragraph>
30     </game>
31 </sports>

```

Fig. 22.14 | XML document that describes various sports. (Part 1 of 2.)

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. 22.14 | XML document that describes various sports. (Part 2 of 2.)

To perform transformations, an XSLT processor is required. For a list of some XSLT processors, visit en.wikipedia.org/wiki/XSL_Transformations. The XML document shown in Fig. 22.14 is transformed into an XHTML document by Edge when the document is loaded into the browser—most popular web browsers include an XSLT processor. For security reasons, some web browsers—such as Microsoft Edge and Google Chrome—will not apply an XSL transformation to a file loaded from the local file system. If you load this example in to your web browser and do not see the formatting, use the following URL to test the XSL transformation:

```
http://deitel.com/test/csharp6fp/fig22_14-15/sports.xml
```

Line 2 (Fig. 22.14) is a **processing instruction (PI)** that references the XSL style sheet `sports.xsl` (Fig. 22.15). A processing instruction is embedded in an XML document and provides application-specific information to whichever XML processor the application uses. In this particular case, the processing instruction specifies the location of an XSLT document with which to transform the XML document. The characters `<?>` and `?>` (Fig. 22.14, line 2) delimit a processing instruction, which consists of a **PI target** (e.g., `xml-stylesheet`) and a **PI value** (e.g., `type = "text/xsl" href = "sports.xsl"`). The PI value's `type` attribute specifies that `sports.xsl` is a `text/xsl` file (i.e., a text file containing XSL content). The `href` attribute specifies the name and location of the style sheet to apply—in this case, `sports.xsl` in the current directory.



Software Engineering Observation 22.5

XSL enables document authors to separate data presentation (specified in XSL documents) from data description (specified in XML documents).

XSL for Transforming `sports.xml`

Figure 22.15 shows the XSL document for transforming the structured data of the XML document of Fig. 22.14 into an XHTML document for presentation. By convention, XSL documents have the file-name extension `.xsl`.

```
1 <?xml version = "1.0"?>
2 <!-- Fig. 22.15: sports.xsl -->
3 <!-- A simple XSLT transformation -->
4
```

Fig. 22.15 | XSLT that creates elements and attributes in an XHTML document. (Part 1 of 2.)

```

5 <!-- reference XSL style sheet URI -->
6 <xsl:stylesheet version = "1.0"
7   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9   <xsl:output method = "xml" omit-xml-declaration = "no"
10     doctype-system =
11       "http://www.w3c.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
12     doctype-public = "-//W3C//DTD XHTML 1.0 Strict//EN"/>
13
14   <xsl:template match = "/"> <!-- match root element -->
15
16     <html xmlns = "http://www.w3.org/1999/xhtml">
17       <head>
18         <title>Sports</title>
19       </head>
20
21       <body>
22         <table border = "1" style = "background-color: turquoise">
23           <thead>
24             <tr>
25               <th>ID</th>
26               <th>Sport</th>
27               <th>Information</th>
28             </tr>
29           </thead>
30
31           <!-- insert each name and paragraph element value -->
32           <!-- into a table row. -->
33           <xsl:for-each select = "/sports/game">
34             <tr>
35               <td><xsl:value-of select = "@id"/></td>
36               <td><xsl:value-of select = "name"/></td>
37               <td><xsl:value-of select = "paragraph"/></td>
38             </tr>
39           </xsl:for-each>
40         </table>
41       </body>
42     </html>
43
44   </xsl:template>
45 </xsl:stylesheet>

```

Fig. 22.15 | XSLT that creates elements and attributes in an XHTML document. (Part 2 of 2.)

stylesheet Start Tag

Lines 6–7 (Fig. 22.15) begin the XSL style sheet with the **stylesheet** start tag. Attribute **version** specifies the XSLT version to which this document conforms. Line 7 binds namespace prefix **xsl** to the W3C’s XSLT URI (i.e., <http://www.w3.org/1999/XSL/Transform>).

xsl:output Element

Lines 9–12 use element **xsl:output** to write an XHTML document type declaration (DOCTYPE) to the result tree (i.e., the XML document to be created). The DOCTYPE identi-

fies XHTML as the type of the resulting document. Attribute `method` is assigned "xml", which indicates that XML is being output to the result tree. (Recall that XHTML is a type of XML.) Attribute `omit-xml-declaration` specifies whether the transformation should write the XML declaration to the result tree. In this case, we do not want to omit the XML declaration, so we assign to this attribute the value "no". Attributes `doctype-system` and `doctype-public` write the DOCTYPE DTD information to the result tree.

xs1:template Element

XSLT uses **templates** (i.e., **xs1:template** elements) to describe how to transform particular nodes from the source tree to the result tree. A template is applied to nodes that are specified in the `match` attribute. Line 14 uses the `match` attribute to select the **document root** (i.e., the conceptual part of the document that contains the root element and everything below it) of the XML source document (i.e., `sports.xml`). The **XPath character /** (a forward slash) is used as a separator between element names. Recall that XPath is a string-based language used to locate parts of an XML document easily. In XPath, a leading forward slash specifies that we are using **absolute addressing** (i.e., we are starting from the root and defining paths down the source tree). In the XML document of Fig. 22.14, the child nodes of the document root are the two processing-instruction nodes (lines 1–2), the two comment nodes (lines 4–5) and the `sports` element node (lines 7–31). The template in Fig. 22.15, line 14, matches a node (i.e., the document root), so the contents of the template are now added to the result tree.

xs1:for-each Element

The XSLT processor writes the XHTML in lines 16–29 (Fig. 22.15) to the result tree exactly as it appears in the XSL document. Now the result tree consists of the DOCTYPE definition and the XHTML code from lines 16–29. Lines 33–39 use element **xs1:for-each** to iterate through the source XML document, searching for `game` elements. The `xs1:for-each` element is similar to C#'s `foreach` statement. Attribute `select` is an XPath expression that specifies the nodes (called the **node set**) on which the `xs1:for-each` operates. Again, the first forward slash means that we are using absolute addressing. The forward slash between `sports` and `game` indicates that `game` is a child node of `sports`. Thus, the `xs1:for-each` finds `game` nodes that are children of the `sports` node. The XML document `sports.xml` contains only one `sports` node, which is also the document root element. After finding the elements that match the selection criteria, the `xs1:for-each` processes each element with the code in lines 34–38 (these lines produce one row in an XHTML table each time they execute) and places the result of lines 34–38 in the result tree.

Line 35 uses element **value-of** to retrieve attribute `id`'s value and place it in a `td` element in the result tree. The XPath symbol `@` specifies that `id` is an attribute node of the **game context node** (i.e., the current node being processed). Lines 36–37 place the `name` and `paragraph` element values in `td` elements and insert them in the result tree. When an XPath expression has no beginning forward slash, the expression uses **relative addressing**. Omitting the beginning forward slash tells the `xs1:value-of select` statements to search for `name` and `paragraph` elements that are children of the context node, not the root node. Owing to the last XPath expression selection, the current context node is `game`, which indeed has an `id` attribute, a `name` child element and a `paragraph` child element.

Using XSLT to Sort and Format Data

Figure 22.16 presents an XML document (`sorting.xml`) that marks up information about a book. Several elements of the markup describing the book appear out of order (e.g., the element describing Chapter 3 appears before the element describing Chapter 2). We arranged them this way purposely to demonstrate that the XSL style sheet referenced in line 5 (`sorting.xsl`) can sort the XML file's data for presentation purposes.

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.16: sorting.xml -->
3 <!-- XML document containing book information -->
4
5 <?xml-stylesheet type = "text/xsl" href = "sorting.xsl"?>
6
7 <book isbn = "999-99999-9-X">
8   <title>Deitel&apos;s XML Primer</title>
9
10  <author>
11    <firstName>Jane</firstName>
12    <lastName>Blue</lastName>
13  </author>
14
15  <chapters>
16    <frontMatter>
17      <preface pages = "2" />
18      <contents pages = "5" />
19      <illustrations pages = "4" />
20    </frontMatter>
21
22    <chapter number = "3" pages = "44">Advanced XML</chapter>
23    <chapter number = "2" pages = "35">Intermediate XML</chapter>
24    <appendix number = "B" pages = "26">Parsers and Tools</appendix>
25    <appendix number = "A" pages = "7">Entities</appendix>
26    <chapter number = "1" pages = "28">XML Fundamentals</chapter>
27  </chapters>
28
29  <media type = "CD" />
30 </book>

```

Fig. 22.16 | XML document containing book information.

Figure 22.17 presents an XSL document (`sorting.xsl`) for transforming `sorting.xml` (Fig. 22.16) to XHTML. Recall that an XSL document navigates a source tree and builds a result tree. In this example, the source tree is XML, and the output tree is XHTML.

```

1 <?xml version = "1.0"?>
2 <!-- Fig. 22.17: sorting.xsl -->
3 <!-- Transformation of book information into XHTML -->
4

```

Fig. 22.17 | XSL document that transforms `sorting.xml` into XHTML. (Part I of 3.)

```

5 <xsl:stylesheet version = "1.0" xmlns = "http://www.w3.org/1999/xhtml"
6   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
7
8   <!-- write XML declaration and DOCTYPE DTD information -->
9   <xsl:output method = "xml" omit-xml-declaration = "no"
10     doctype-system = "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
11     doctype-public = "-//W3C//DTD XHTML 1.1//EN"/>
12
13   <!-- match document root -->
14   <xsl:template match = "/">
15     <html>
16       <xsl:apply-templates/>
17     </html>
18   </xsl:template>
19
20   <!-- match book -->
21   <xsl:template match = "book">
22     <head>
23       <title>ISBN <xsl:value-of select = "@isbn"/> -
24       <xsl:value-of select = "title"/></title>
25     </head>
26
27     <body>
28       <h1 style = "color: blue"><xsl:value-of select = "title"/></h1>
29       <h2 style = "color: blue">by
30         <xsl:value-of select = "author/firstName"/>
31         <xsl:text> </xsl:text>
32         <xsl:value-of select = "author/lastName"/>
33       </h2>
34
35       <table style = "border-style: groove; background-color: gold">
36
37         <xsl:for-each select = "chapters/frontMatter/*">
38           <tr>
39             <td style = "text-align: right">
40               <xsl:value-of select = "name()"/>
41             </td>
42
43             <td>
44               ( <xsl:value-of select = "@pages"/> pages )
45             </td>
46           </tr>
47         </xsl:for-each>
48
49         <xsl:for-each select = "chapters/chapter">
50           <xsl:sort select = "@number" data-type = "number"
51             order = "ascending"/>
52           <tr>
53             <td style = "text-align: right">
54               Chapter <xsl:value-of select = "@number"/>
55             </td>
56

```

Fig. 22.17 | XSL document that transforms `sorting.xml` into XHTML. (Part 2 of 3.)

```

57         <td>
58             <xsl:value-of select = "text()"/>
59             ( <xsl:value-of select = "@pages"/> pages )
60         </td>
61     </tr>
62 </xsl:for-each>
63
64     <xsl:for-each select = "chapters/appendix">
65         <xsl:sort select = "@number" data-type = "text"
66             order = "ascending"/>
67         <tr>
68             <td style = "text-align: right">
69                 Appendix <xsl:value-of select = "@number"/>
70             </td>
71
72             <td>
73                 <xsl:value-of select = "text()"/>
74                 ( <xsl:value-of select = "@pages"/> pages )
75             </td>
76         </tr>
77     </xsl:for-each>
78 </table>
79
80 <p style = "color: blue">Pages:
81     <xsl:variable name = "pagecount"
82         select = "sum(chapters/**/@pages)"/>
83     <xsl:value-of select = "$pagecount"/>
84 <br />Media Type: <xsl:value-of select = "media/@type"/></p>
85 </body>
86 </xsl:template>
87 </xsl:stylesheet>

```

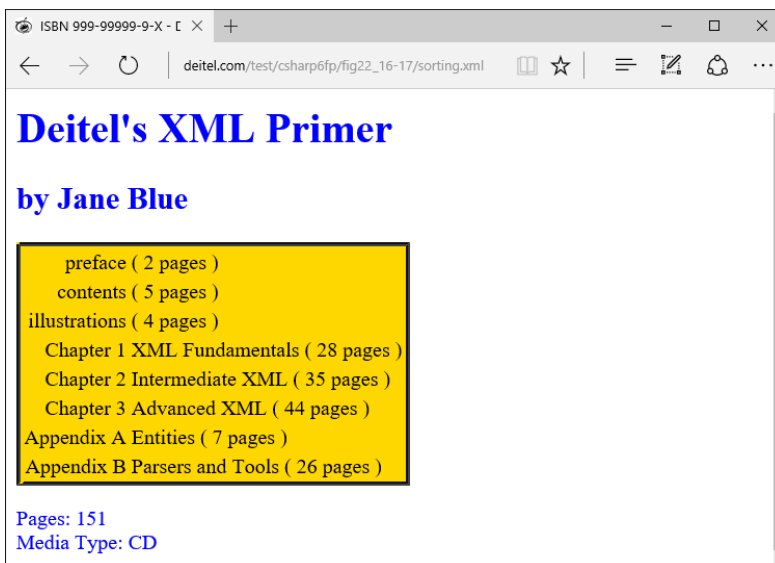


Fig. 22.17 | XSL document that transforms `sorting.xml` into XHTML. (Part 3 of 3.)

Line 14 of Fig. 22.17 matches the root element of the document in Fig. 22.16. Line 15 (Fig. 22.17) outputs an `html` start tag to the result tree. The `<xsl:apply-templates/>` element (line 16) specifies that the XSLT processor is to apply the `xsl:templates` defined in this XSL document to the current node's (i.e., the document root's) children. The content from the applied templates is output in the `html` element that ends at line 17. Lines 21–86 specify a template that matches element `book`. The template indicates how to format the information contained in `book` elements of `sorting.xml` (Fig. 22.16) as XHTML.

Lines 23–24 create the title for the XHTML document. We use the book's ISBN (from attribute `isbn`) and the contents of element `title` to create the string that appears in the browser window's title bar (`ISBN 999-99999-9-X - Deitel's XML Primer`).

Line 28 creates a header element that contains the book's title. Lines 29–33 create a header element that contains the book's author. Because the *context node* (i.e., the 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. The `xsl:text` element (line 31) is used to insert literal text. Because XML (and therefore XSLT) ignores whitespace, the author's name would appear as `JaneBlue` without inserting the explicit space.

Line 37 selects each element (indicated by an asterisk) that is a child of element `frontMatter`. Line 40 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 37). Line 44 retrieves the value of the `pages` attribute of the current node.

Line 49 selects each `chapter` element. Lines 50–51 use element `xsl:sort` to sort chapters by number in *ascending* order. Attribute `select` selects the value of attribute `number` in context node `chapter`. Attribute `data-type`, with value "number", specifies a numeric sort, and attribute `order`, with value "ascending", specifies ascending order. Attribute `data-type` also accepts the value "text" (line 65), and attribute `order` also accepts the value "descending". Line 58 uses **node-set function text** to obtain the text between the `chapter` start and end tags (i.e., the name of the chapter). Line 59 retrieves the value of the `pages` attribute of the current node. Lines 64–77 perform similar tasks for each appendix.

Lines 81–82 use an **XSL variable** to store the value of the book's total page count and output the page count to the result tree. Such variables cannot be modified after they're initialized. Attribute `name` specifies the variable's name (i.e., `pagecount`), and attribute `select` assigns a value to the variable. Function `sum` (line 82) totals the values for all page attribute values. The two slashes between `chapters` and `*` indicate a **recursive descent**—the XSLT processor will search for elements that contain an attribute named `pages` in all descendant nodes of `chapters`. The XPath expression

```
//*
```

selects all the nodes in an XML document. Line 83 retrieves the value of the newly created XSL variable `pagecount` by placing a dollar sign in front of its name.



Performance Tip 22.1

Selecting all nodes in a document when it's not necessary slows XSLT processing.

Summary of XSL Style-Sheet Elements

This section's examples used several predefined XSL elements to perform various operations. Figure 22.18 lists commonly used XSL elements. For more information on these elements and XSL in general, see www.w3.org/Style/XSL.

Element	Description
<code><xsl:apply-templates></code>	Applies the templates of the XSL document to the children of the current node.
<code><xsl:apply-templates match = "expression"></code>	Applies the templates of the XSL document to the children of the nodes matching <i>expression</i> . The value of the attribute <i>match</i> (i.e., <i>expression</i>) must be an XPath expression that specifies elements.
<code><xsl:template></code>	Contains rules to apply when a specified node is matched.
<code><xsl:value-of select = "expression"></code>	Selects the value of an XML element or attribute and adds it to the output tree of the transformation. The required <i>select</i> attribute contains an XPath expression.
<code><xsl:for-each select = "expression"></code>	Applies a template to every node selected by the XPath specified by the <i>select</i> attribute.
<code><xsl:sort select = "expression"></code>	Used as a child element of an <code><xsl:apply-templates></code> or <code><xsl:for-each></code> element. Sorts the nodes selected by the <code><xsl:apply-templates></code> or <code><xsl:for-each></code> element so that the nodes are processed in sorted order.
<code><xsl:output></code>	Has various attributes to define the format (e.g., XML, XHTML), version (e.g., 1.0, 2.0), document type and MIME type of the output document. This tag is a top-level element—it can be used only as a child element of an <code>xsl:stylesheet</code> .
<code><xsl:copy></code>	Adds the current node to the output tree.

Fig. 22.18 | XSL style-sheet elements.

This section introduced Extensible Stylesheet Language (XSL) and showed how to create XSL transformations to convert XML documents from one format to another. We showed how to transform XML documents to XHTML documents for display in a web browser. In most business applications, XML documents are transferred between business partners and are transformed to other XML vocabularies programmatically. In Section 22.11, we demonstrate how to perform XSL transformations using the `XsltCompiledTransform` class provided by the .NET Framework.

22.8 LINQ to XML: Document Object Model (DOM)

Although an XML document is a text file, retrieving data from the document using traditional sequential file-processing techniques is not practical, especially for adding and removing elements dynamically.

On successfully parsing a document, some XML parsers store document data as *trees* in memory. Figure 22.19 illustrates the tree structure for the document `article.xml` dis-

cussed in Fig. 22.2. This hierarchical tree structure is called a **Document Object Model (DOM) tree**, and an XML parser that creates such a tree is known as a **DOM parser**. DOM gets its name from the conversion of an XML document's tree structure into a tree of objects that are then manipulated using an object-oriented programming language such as C#. Each element name (e.g., `article`, `date`, `firstName`) is represented by a node. A node that contains other nodes (called **child nodes** or children) is called a **parent node** (e.g., `author`). A parent node can have *many* children, but a child node can have *only one* parent node. Nodes that are *peers* (e.g., `firstName` and `lastName`) are called **sibling nodes**. A node's **descendant nodes** include its children, its children's children and so on. A node's **ancestor nodes** include its parent, its parent's parent and so on.

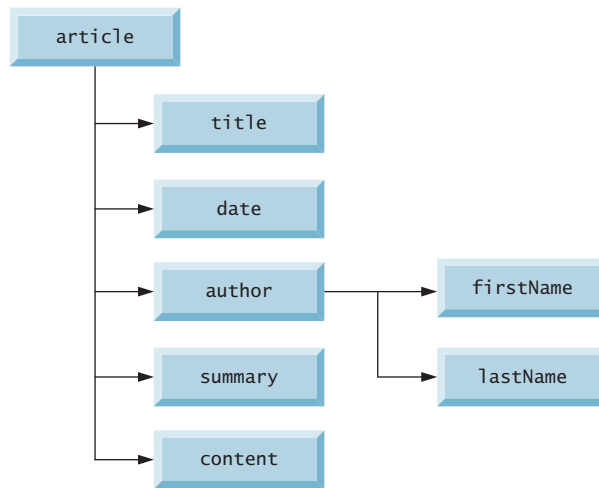


Fig. 22.19 | Tree structure for the document `article.xml`.

The DOM tree has a single **root node**, which contains all the other nodes in the document. For example, the root node of the DOM tree that represents `article.xml` (Fig. 22.2) contains a node for the XML declaration (line 1), two nodes for the comments (lines 2–3) and a node for the XML document's root element `article` (line 5).

Classes for creating, reading and manipulating XML documents are located in the **System.Xml** namespace, which also contains additional namespaces that provide other XML-related operations.

Reading an XML Document with an XDocument

Namespace **System.Xml.Linq** contains the classes used to manipulate a DOM in .NET. Though LINQ query expressions are not required to use them, the technologies used are collectively referred to as **LINQ to XML**. Previous versions of the .NET Framework used a different DOM implementation in the `System.Xml` namespace. These classes (such as `XmlDocument`) should generally be avoided in favor of LINQ to XML. In LINQ to XML, the **XElement** class represents a DOM element node—an XML document is represented by a tree of `XElement` objects. The **XDocument** class represents an entire XML document. Unlike `XElements`, `XDocuments` cannot be nested. Figure 22.20 uses these classes to load

the `article.xml` document (Fig. 22.2) and display its data in a `TextBox`. The program displays a formatted version of its input XML file. If `article.xml` were poorly formatted, such as being all on one line, this application would allow you to convert it into a form that is much easier to understand.

```

1 // Fig. 22.20: XDocumentTestForm.cs
2 // Reading an XML document and displaying it in a TextBox.
3 using System;
4 using System.Xml.Linq;
5 using System.Windows.Forms;
6
7 namespace XDocumentTest
8 {
9     public partial class XDocumentTestForm : Form
10    {
11        public XDocumentTestForm()
12        {
13            InitializeComponent();
14        }
15
16        // read XML document and display its content
17        private void XDocumentTestForm_Load(object sender, EventArgs e)
18        {
19            // load the XML file into an XDocument
20            XDocument xmlFile = XDocument.Load("article.xml");
21            int indentLevel = 0; // no indentation for root element
22
23            // print elements recursively
24            PrintElement(xmlFile.Root, indentLevel);
25        }
26
27        // display an element (and its children, if any) in the TextBox
28        private void PrintElement(XElement element, int indentLevel)
29        {
30            // get element name without namespace
31            string name = element.Name.LocalName;
32
33            // display the element's name within its tag
34            IndentOutput(indentLevel); // indent correct amount
35            outputTextBox.AppendText('<' + name + ">\n");
36
37            // check for child elements and print value if none contained
38            if (element.HasElements)
39            {
40                // print all child elements at the next indentation level
41                foreach (var child in element.Elements())
42                {
43                    PrintElement(child, indentLevel + 1);
44                }
45            }
46            else
47            {

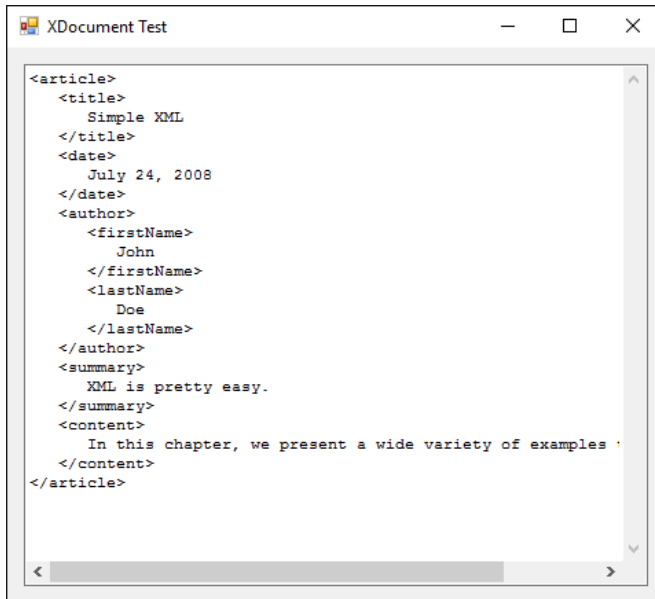
```

Fig. 22.20 | Reading an XML document and displaying it in a `TextBox`. (Part I of 2.)

```

48         // increase the indentation amount for text elements
49         IndentOutput(indentLevel + 1);
50
51         // display the text inside this element
52         outputTextBox.AppendText(element.Value.Trim() + '\n');
53     }
54
55     // display end tag
56     IndentOutput(indentLevel);
57     outputTextBox.AppendText("</" + name + ">\n");
58 }
59
60 // add the specified amount of indentation to the current line
61 private void IndentOutput(int number)
62 {
63     for (int i = 0; i < number; ++i)
64     {
65         outputTextBox.AppendText("    ");
66     }
67 }
68 }
69 }

```



```

<article>
  <title>
    Simple XML
  </title>
  <date>
    July 24, 2008
  </date>
  <author>
    <firstName>
      John
    </firstName>
    <lastName>
      Doe
    </lastName>
  </author>
  <summary>
    XML is pretty easy.
  </summary>
  <content>
    In this chapter, we present a wide variety of examples
  </content>
</article>

```

Fig. 22.20 | Reading an XML document and displaying it in a TextBox. (Part 2 of 2.)

To create an XDocument from an existing XML document, we use XDocument's static **Load method**, giving the location of the document as an argument (Fig. 22.20, line 20). The returned XDocument contains a tree representation of the loaded XML file, which is used to navigate the file's contents. The XDocument's **Root property** (line 24) returns an XElement representing the root element of the XML file.

Method `PrintElement` (lines 28–58) displays an `XElement` in `outputTextBox`. Because nested elements should be at different indentation levels, `PrintElement` takes an `int` specifying the amount of indentation to use in addition to the `XElement` it's displaying. Variable `indentLevel` is passed as an argument to the `IndentOutput` method (lines 61–67) to add the correct amount of spacing before the begin (line 35) and end (line 57) tags are displayed.

As you've seen in previous sections, tag and attribute names often have a namespace prefix. Because the full names consist of two parts (the prefix and name), tag and attribute names are stored not simply as strings, but as objects of class `XName`. The `Name` property of an `XElement` (line 31) returns an `XName` object containing the tag name and namespace—we are not interested in the namespace for this example, so we retrieve the unqualified name using the `XName`'s `LocalName` property.

`XElements` with and without children are treated differently in the program—this test is performed using the `HasElements` property (line 38). For `XElements` with children, we use the `Elements` method (line 41) to obtain the children, then iterate through them and recursively print their children by calling `PrintElement` (line 43). For `XElements` that do not have children, the text they contain is displayed using the `Value` property (line 52). If used on an element with children, the `Value` property returns all of the text contained within its descendants, with the tags removed. For simplicity, elements with attributes and those with both elements and text as children are not handled. The indentation is increased by one in both cases to allow for proper formatting.

22.9 LINQ to XML Class Hierarchy

As you saw in the previous section, `XElement` objects provide several methods for quickly traversing the DOM tree they represent. LINQ to XML provides many other classes for representing different parts of the tree. Figure 22.21 demonstrates the use of these additional classes to navigate the structure of an XML document and display it in a `TreeView` control. It also shows how to use these classes to get functionality equivalent to the `XPath` strings introduced in Section 22.7. The file used as a data source (`sports.xml`) is shown in Fig. 22.14.

```

1 // Fig. 22.21: PathNavigatorForm.cs
2 // Document navigation using XNode.
3 using System;
4 using System.Collections.Generic;
5 using System.Linq;
6 using System.Xml; // for XmlNodeType enumeration
7 using System.Xml.Linq; // for XNode and others
8 using System.Xml.XPath; // for XPathSelectElements
9 using System.Windows.Forms;
10
11 namespace PathNavigator
12 {
13     public partial class PathNavigatorForm : Form
14     {

```

Fig. 22.21 | Document navigation using `XNode`. (Part I of 7.)

```
15     private XNode current; // currently selected node
16     private XDocument document; // the document to navigate
17     private TreeNode tree; // TreeNode used by TreeView control
18
19     public PathNavigatorForm()
20     {
21         InitializeComponent();
22     }
23
24     // initialize variables and TreeView control
25     private void PathNavigatorForm_Load(object sender, EventArgs e)
26     {
27         document = XDocument.Load("sports.xml"); // load sports.xml
28
29         // current node is the entire document
30         current = document;
31
32         // create root TreeNode and add to TreeView
33         tree = new TreeNode(NodeText(current));
34         pathTreeView.Nodes.Add(tree); // add TreeNode to TreeView
35         TreeRefresh(); // reset the tree display
36     }
37
38     // print the elements of the selected path
39     private void locateComboBox_SelectedIndexChanged(
40         object sender, EventArgs e)
41     {
42         // retrieve the set of elements to output
43         switch (locateComboBox.SelectedIndex)
44         {
45             case 0: // print all sports elements
46                 PrintElements(document.Elements("sports"));
47                 break;
48             case 1: // print all game elements
49                 PrintElements(document.Descendants("game"));
50                 break;
51             case 2: // print all name elements
52                 PrintElements(document.XPathSelectElements("//name"));
53                 break;
54             case 3: // print all paragraph elements
55                 PrintElements(document.Descendants("game")
56                     .Elements("paragraph"));
57                 break;
58             case 4: // print game elements with name element of "Cricket"
59                 // use LINQ to XML to retrieve the correct node
60                 var cricket =
61                     from game in document.Descendants("game")
62                     where game.Element("name").Value == "Cricket"
63                     select game;
64                 PrintElements(cricket);
65                 break;
```

Fig. 22.21 | Document navigation using XNode. (Part 2 of 7.)

```

66         case 5: // print all id attributes of game
67             PrintIDs(document.Descendants("game"));
68             break;
69     }
70 }
71
72 // traverse to first child
73 private void firstChildButton_Click(object sender, EventArgs e)
74 {
75     // try to convert to an XContainer
76     var container = current as XContainer;
77
78     // if container has children, move to first child
79     if (container != null && container.Nodes().Any())
80     {
81         current = container.Nodes().First(); // first child
82
83         // create new TreeNode for this node with correct label
84         var newNode = new TreeNode(NodeText(current));
85         tree.Nodes.Add(newNode); // add node to TreeNode Nodes list
86         tree = newNode; // move current selection to newNode
87         TreeRefresh(); // reset the tree display
88     }
89     else
90     {
91         // current node is not a container or has no children
92         MessageBox.Show("Current node has no children.", "Warning",
93             MessageBoxButtons.OK, MessageBoxIcon.Information);
94     }
95 }
96
97 // traverse to node's parent
98 private void parentButton_Click(object sender, EventArgs e)
99 {
100     // if current node is not the root, move to parent
101     if (current.Parent != null)
102         current = current.Parent; // get parent node
103     else // node is at top level: move to document itself
104         current = current.Document;
105
106     // move TreeView if it is not already at the root
107     if (tree.Parent != null)
108     {
109         tree = tree.Parent; // get parent in tree structure
110         tree.Nodes.Clear(); // remove all children
111         TreeRefresh(); // reset the tree display
112     }
113 }
114

```

Fig. 22.21 | Document navigation using XNode. (Part 3 of 7.)

```

115 // traverse to previous node
116 private void previousButton_Click(object sender, EventArgs e)
117 {
118     // if current node is not first, move to previous node
119     if (current.PreviousNode != null)
120     {
121         current = current.PreviousNode; // move to previous node
122         var treeParent = tree.Parent; // get parent node
123         treeParent.Nodes.Remove(tree); // delete current node
124         tree = treeParent.LastNode; // set current display position
125         TreeRefresh(); // reset the tree display
126     }
127     else // current element is first among its siblings
128     {
129         MessageBox.Show("Current node is first sibling.", "Warning",
130             MessageBoxButtons.OK, MessageBoxIcon.Information);
131     }
132 }
133
134 // traverse to next node
135 private void nextButton_Click(object sender, EventArgs e)
136 {
137     // if current node is not last, move to next node
138     if (current.NextNode != null)
139     {
140         current = current.NextNode; // move to next node
141
142         // create new TreeNode to display next node
143         var newNode = new TreeNode(NodeText(current));
144         var treeParent = tree.Parent; // get parent TreeNode
145         treeParent.Nodes.Add(newNode); // add to parent node
146         tree = newNode; // set current position for display
147         TreeRefresh(); // reset the tree display
148     }
149     else // current node is last among its siblings
150     {
151         MessageBox.Show("Current node is last sibling.", "Warning",
152             MessageBoxButtons.OK, MessageBoxIcon.Information);
153     }
154 }
155
156 // update TreeView control
157 private void TreeRefresh()
158 {
159     pathTreeView.ExpandAll(); // expand tree node in TreeView
160     pathTreeView.Refresh(); // force TreeView update
161     pathTreeView.SelectedNode = tree; // highlight current node
162 }
163
164 // print values in the given collection
165 private void PrintElements(IEnumerable< XElement > elements)
166 {

```

Fig. 22.21 | Document navigation using XElement. (Part 4 of 7.)

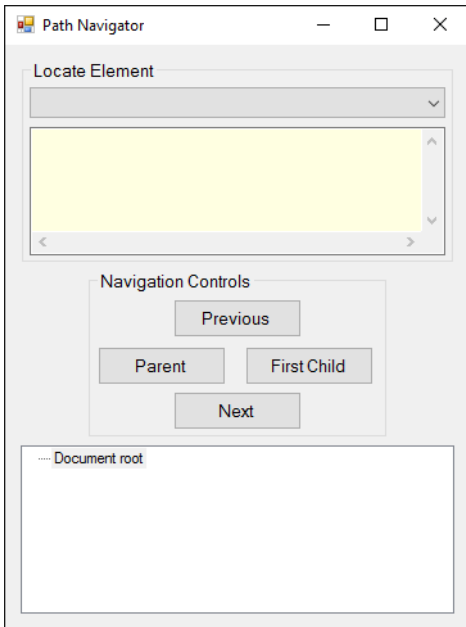
```

167         locateTextBox.Clear(); // clear the text area
168
169         // display text inside all elements
170         foreach (var element in elements)
171         {
172             locateTextBox.AppendText(element.Value.Trim() + '\n');
173         }
174     }
175
176     // print the ID numbers of all games in elements
177     private void PrintIDs(IEnumerable< XElement > elements)
178     {
179         locateTextBox.Clear(); // clear the text area
180
181         // display "id" attribute of all elements
182         foreach (var element in elements)
183         {
184             locateTextBox.AppendText(
185                 element.Attribute("id").Value.Trim() + '\n');
186         }
187     }
188
189     // returns text used to represent an element in the tree
190     private string NodeText(XNode node)
191     {
192         // different node types are displayed in different ways
193         switch (node.NodeType)
194         {
195             case XmlNodeType.Document:
196                 // display the document root
197                 return "Document root";
198             case XmlNodeType.Element:
199                 // represent node by tag name
200                 return '<' + (node as XElement).Name.LocalName + '>';
201             case XmlNodeType.Text:
202                 // represent node by text stored in Value property
203                 return (node as XText).Value;
204             case XmlNodeType.Comment:
205                 // represent node by comment text
206                 return (node as XComment).ToString();
207             case XmlNodeType.ProcessingInstruction:
208                 // represent node by processing-instruction text
209                 return (node as XProcessingInstruction).ToString();
210             default:
211                 // all nodes in this example are already covered;
212                 // return a reasonable default value for other nodes
213                 return node.NodeType.ToString();
214         }
215     }
216 }
217 }

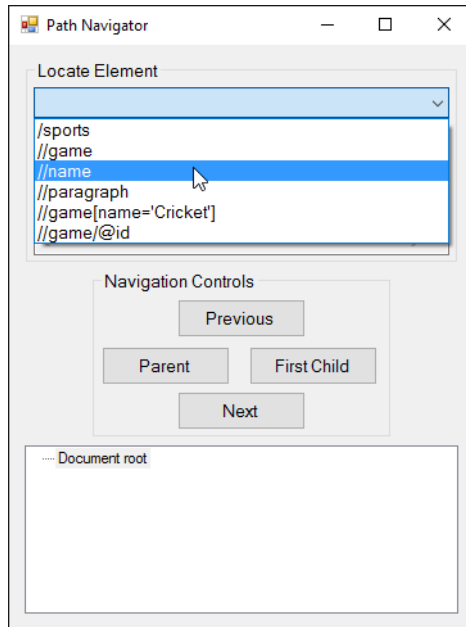
```

Fig. 22.21 | Document navigation using XNode. (Part 5 of 7.)

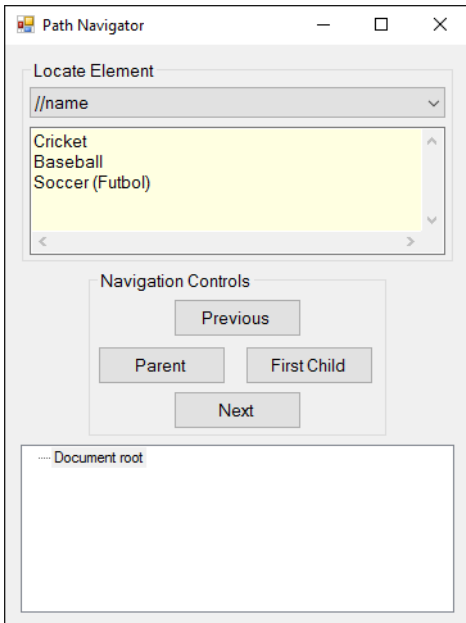
a) Path Navigator form upon loading



b) The //name path is selected



c) The //name path displays all name elements in the document



d) The //game[name='Cricket'] path displays game elements with a name element containing "Cricket"

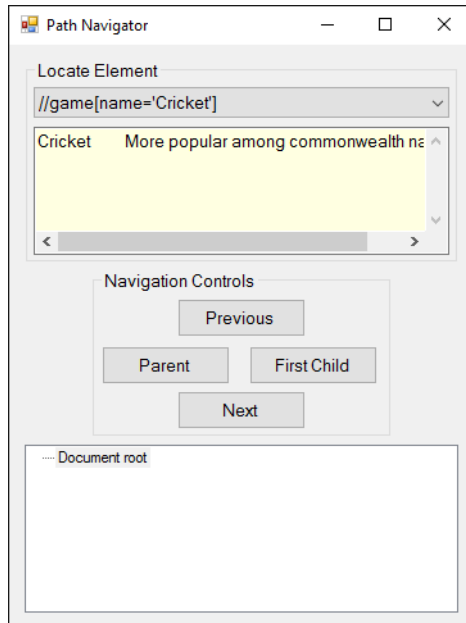
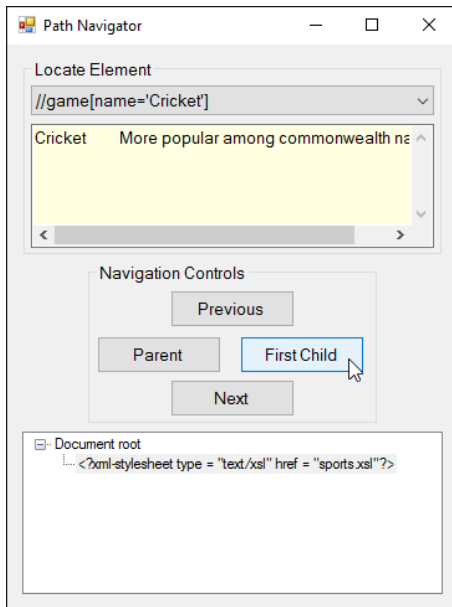


Fig. 22.21 | Document navigation using XNode. (Part 6 of 7.)

e) The **First Child** button expands the tree to show the first element in that group



f) The **Next** button lets you view siblings of the current element

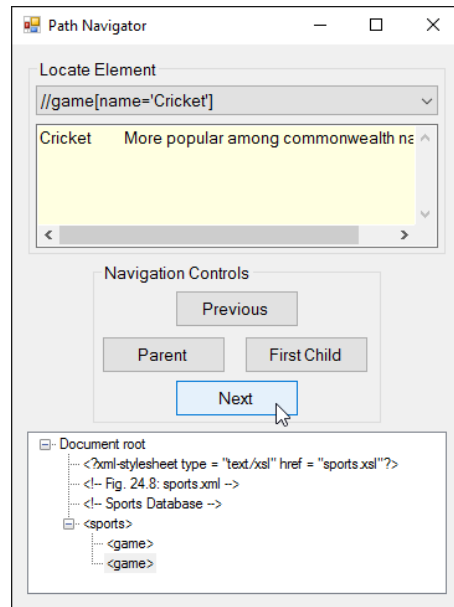


Fig. 22.21 | Document navigation using XElement. (Part 7 of 7.)

The interface for this example allows the user to display selected elements in the TextBox, or to navigate through the DOM tree in the lower TreeView. Initially, the TextBox is blank, and the TreeView is initialized to show the root of the tree. The ComboBox at the top of the Form contains XPath expressions. These are not used directly—instead, the example uses the LINQ to XML DOM classes and a LINQ query to retrieve the same results. As in the previous example, the XElement's Load method (line 27) is used to load the contents of the XML file into memory. Instance variable current, which points to the current position in the DOM, is initialized to the document itself (line 30). Line 33 creates a TreeNode for the XElement with the correct text, which is then inserted into the TreeView (lines 34–35). The TreeRefresh method (lines 157–162) refreshes the pathTreeView control so that the user interface updates correctly.

locateComboBox_SelectedIndexChanged Event Handler

The SelectedIndexChanged event handler of locateComboBox (lines 39–70) fills the TextBox with the elements corresponding to the path the user selected. The first case (lines 45–47) uses the Elements method of the XElement object document. The Elements method is overloaded—one version has no parameter and returns all child elements. The second version returns only elements with the given tag name. Recall from the previous example that XElement also has an Elements method. This is because the method is actually defined in the XContainer class, the base class of XElement and XElement. XContainer represents nodes in the DOM tree that can contain other nodes. The results of the call to the method Elements are passed to the PrintElements method (lines 165–174). The

PrintElements method uses the XElement's Value property (line 172) introduced in the previous example. The Value property returns all text in the current node and its descendants. The text is displayed in locateTextBox.

The second case (lines 48–50) uses the **Descendants method**—another XElement method common to XElement and XDocument—to get the same results as the XPath double slash (//) operator. In other words, the Descendants method returns all descendant elements with the given tag name, not just direct children. Like Elements, it's overloaded and has a version with no parameter that returns all descendants.

The third case (lines 51–53) uses extension method **XPathSelectElements** from namespace **System.Xml.XPath** (imported at line 8). This method allows you to use an XPath expression to navigate XDocument and XElement objects. It returns an IEnumerable<XElement>. There's also an XPathSelectElement method that returns a single XElement.

The fourth case (lines 54–57) also uses the Descendants method to retrieve all game elements, but it then calls the Elements method to retrieve the child paragraph elements. Because the Descendants method returns an IEnumerable<XElement>, the Elements method is not being called on the XElement class that we previously stated contains the Elements method. Calling the Elements method in this way is allowed because there's an extension method in the System.Xml.Linq namespace that returns an IEnumerable<XElement> containing the children of all elements in the original collection. To match the interface of the XElement class, there's also a Descendants extension method, and both have versions that do not take an argument.

In a document where a specific element appears at multiple nesting levels, you may need to use chained calls of the Elements method explicitly to return only the elements in which you are interested. Using the Descendants method in these cases can be a source of subtle bugs—if the XML document's structure changes, your code could silently accept input that the program should not treat as valid. The Descendants method is best used for tags that can appear at any nesting level within the document, such as formatting tags in XHTML, which can occur in many distinct parts of the text.

The fifth case (lines 58–65) retrieves only the game elements with a name element containing "Cricket". To do this, we use a LINQ query (lines 61–63). The Descendants and Element methods return an IEnumerable<XElement>, so they can be used as the subject of a LINQ query. The where clause (line 62) uses the Element method to retrieve all name elements that are children of the game element the range variable represents. The **Element method**, a member of the XElement class, returns the first child element with the given tag name or null if no such element exists. The where clause uses the Value property to retrieve the text contained in the element. We do not check for Element returning null because we know that all game elements in sports.xml contain name elements.

PrintIDs Method

The PrintIDs method (lines 177–187) displays the id attributes of the XElement objects passed to it—specifically, the game elements in the document (line 67). To do this, it uses the **Attribute method** of the XElement class (line 185). The Attribute method returns an XAttribute object matching the given attribute name or null if no such object exists. The **XAttribute class** represents an XML attribute—it holds the attribute's name and value. Here, we access its Value property to get a string that contains the attribute's value—it also can be used to set the value.

LINQ to XML Class Hierarchy

The `Click` event handlers for the Buttons in the example are used to update the data displayed in the `TreeView`. These methods introduce many other classes from the namespace `System.Xml.Linq`. The entire LINQ to XML class hierarchy is shown in the UML class diagram of Fig. 22.22. `XNamespace` will be covered in the next section, and `XDocumentType` holds a DTD, which may be defined directly in an XML file rather than externally referenced (as we did in Fig. 22.4, `letter.xml`).

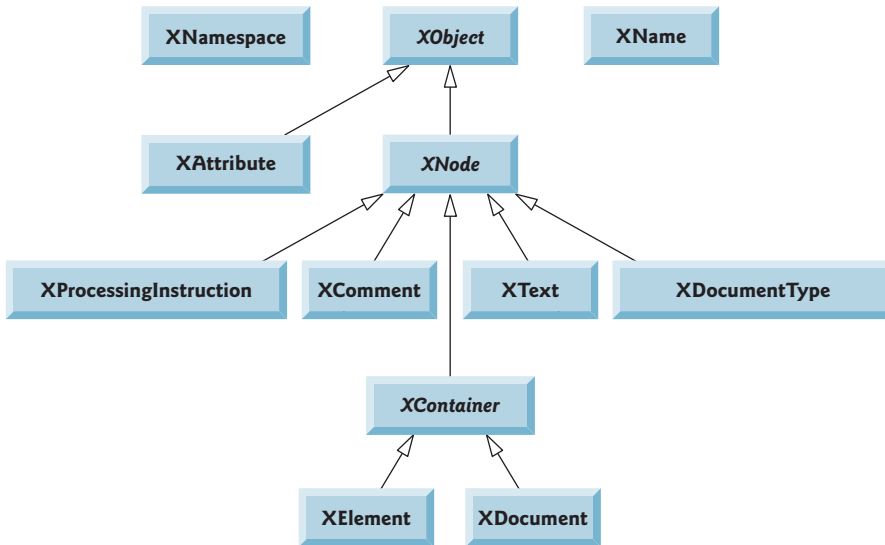


Fig. 22.22 | LINQ to XML class hierarchy diagram.

As you can see from the diagram, the `XNode` class is a common abstract base class of all the node types in an XML document—including elements, text and processing instructions. Because all DOM node classes inherit from `XNode`, an `XNode` object can be used to keep track of our current location as we navigate the DOM tree.

FirstChildButton_Click Event Handler

The `FirstChildButton_Click` event handler (Fig. 22.21, lines 73–95) uses the `as` operator to determine whether the current node is an `XContainer` (line 76). Recall that the `as` operator attempts to cast the reference to another type, and returns `null` if it cannot. If `current` is an `XContainer` and has children (line 79), we move `current` to its first child (line 81). These operations use the `Nodes` method of class `XContainer`, which returns a reference to an object of type `IEnumerable<XNode>` containing all children of the given `XContainer`. Line 79 uses the `Any` extension method introduced in Chapter 9—all of the standard LINQ to Objects methods may be used with the LINQ to XML classes. The event handler then inserts a `TreeNode` into the `TreeView` to display the child element that `current` now references (lines 84–87).

NodeText Method

Line 84 uses the `NodeText` method (lines 190–215) to determine what text to display in the `TreeNode`. It uses the **`NodeType` property**, which returns a value of the **`XmlNodeType` enumeration** from the `System.Xml` namespace (imported at line 6) indicating the object's node type. Although we call it on an `XNode`, the `NodeType` property is actually defined in the **`XObject` class**. `XObject` is an abstract base class for all nodes and attributes. The `NodeType` property is overridden in the concrete subclasses to return the correct value.

After the node's type has been determined, it's converted to the appropriate type using the `as` operator, then the correct text is retrieved. For the entire document, it returns the text `Document root` (line 197). For elements, `NodeText` returns the tag name enclosed in angle brackets (line 200). For text nodes, it uses the contained text. It retrieves this by converting the `XNode` to an `XText` object—the **`XText` class** holds the contents of a text node. `XText`'s **`Value` property** returns the contained text (line 203)—we could also have used its `ToString` method. Comments, represented by the **`XComment` class**, are displayed just as they're written in the XML file using the `ToString` method of `XComment` (line 206). The `ToString` methods of all subclasses of `XNode` return the XML they and their children (if any) represent with proper indentation. The last type handled is processing instructions, stored in the **`XProcessingInstruction` class** (line 209)—in this example, the only processing instruction is the XML declaration at the beginning of the file. A default case returning the name of the node type is included for other node types that do not appear in `sports.xml` (line 213).

Other Button Event Handlers

The event handlers for the other Buttons are structured similarly to `FirstChildButton_Click`—each moves current and updates the `TreeView` accordingly. The `parentButton_Click` method (lines 98–113) ensures that the current node has a parent—that is, it's not at the root of the `XDocument`—before it tries to move current to the parent (line 102). It uses the **`Parent` property** of `XObject`, which returns the parent of the given `XObject` or `null` if the parent does not exist. For nodes at the root of the document, including the root element, XML declaration, header comments and the document itself, `Parent` with return `null`. We want to move up to the document root in this case, so we use the **`Document` property** (also defined in `XObject`) to retrieve the `XDocument` representing the document root (line 104). The `Document` property of an `XDocument` returns itself. This is consistent with most file systems—attempting to move up a directory from the root will succeed, but not move.

The event handlers for the **`Previous`** (lines 116–132) and **`Next`** (lines 135–154) Buttons use the **`PreviousNode`** (lines 119 and 121) and **`NextNode`** (lines 138 and 140) properties of `XNode`, respectively. As their names imply, they return the previous or next sibling node in the tree. If there's no previous or next node, the properties return `null`.

22.10 LINQ to XML: Namespaces and Creating Documents

As you know, XML namespaces provide a technique for preventing collisions between tag names used for different purposes. LINQ to XML provides the **`XNamespace` class** to enable creation and manipulation of XML namespaces.

Using LINQ to XML to navigate data already stored in an XML document is a common operation, but sometimes it's necessary to create an XML document from scratch. Figure 22.23 uses these features to update an XML document to a new format and combine the data in it with data from a document already in the new format. Figures 22.24 and 22.25 contain the XML files in the old and new formats, respectively. Figure 22.26 displays the file output by the program.

```
1 // Fig. 22.23: XMLCombine.cs
2 // Transforming an XML document and splicing its contents with another.
3 using System;
4 using System.Linq;
5 using System.Xml.Linq;
6
7 class XMLCombine
8 {
9     // namespaces used in XML files
10    private static readonly XNamespace employeesOld =
11        "http://www.deitel.com/employeesold";
12    private static readonly XNamespace employeesNew =
13        "http://www.deitel.com/employeesnew";
14
15    static void Main(string[] args)
16    {
17        // load files from disk
18        XDocument newDocument = XDocument.Load("employeesNew.xml");
19        XDocument oldDocument = XDocument.Load("employeesOld.xml");
20
21        // convert from old to new format
22        oldDocument = TransformDocument(oldDocument);
23
24        // combine documents and write to output file
25        SaveFinalDocument(newDocument, oldDocument);
26
27        // tell user we have finished
28        Console.WriteLine("Documents successfully combined.");
29    }
30
31    // convert the given XDocument in the old format to the new format
32    private static XDocument TransformDocument(XDocument document)
33    {
34        // use a LINQ query to fill the new XML root with the correct data
35        var newDocumentRoot = new XElement(employeesNew + "employeelist",
36            from employee in document.Root.Elements()
37            select TransformEmployee(employee));
38
39        return new XDocument(newDocumentRoot); // return new document
40    }
41
42    // transform a single employee's data from old to new format
43    private static XElement TransformEmployee(XElement employee)
44    {
```

Fig. 22.23 | Transforming an XML document and splicing its contents with another. (Part 1 of 2.)

```

45 // retrieve values from old-format XML document
46 XNamespace old = employeesOld; // shorter name
47 string firstName = employee.Element(old + "firstname").Value;
48 string lastName = employee.Element(old + "lastname").Value;
49 string salary = employee.Element(old + "salary").Value;
50
51 // return new-format element with the correct data
52 return new XElement(employeesNew + "employee",
53     new XAttribute("name", firstName + " " + lastName),
54     new XAttribute("salary", salary));
55 }
56
57 // take two new-format XDocuments and combine
58 // them into one, then save to output.xml
59 private static void SaveFinalDocument(XDocument document1,
60     XDocument document2)
61 {
62     // create new root element
63     var root = new XElement(employeesNew + "employeelist");
64
65     // fill with the elements contained in the roots of both documents
66     root.Add(document1.Root.Elements());
67     root.Add(document2.Root.Elements());
68
69     root.Save("output.xml"); // save document to file
70 }
71 }

```

Fig. 22.23 | Transforming an XML document and splicing its contents with another. (Part 2 of 2.)

```

1 <?xml version="1.0"?>
2 <!-- Fig. 22.24: employeesOld.xml -->
3 <!-- Sample old-format input for the XMLCombine application. -->
4 <employees xmlns="http://www.deitel.com/employeesold">
5     <employeelist>
6         <firstname>Christopher</firstname>
7         <lastname>Green</lastname>
8         <salary>1460</salary>
9     </employeelist>
10    <employeelist>
11        <firstname>Michael</firstname>
12        <lastname>Red</lastname>
13        <salary>1420</salary>
14    </employeelist>
15 </employees>

```

Fig. 22.24 | Sample old-format input for the XMLCombine application.

```
1 <?xml version="1.0"?>
2 <!-- Fig. 22.25: employeesNew.xml -->
3 <!-- Sample new-format input for the XMLCombine application. -->
4 <employeeelist xmlns="http://www.deitel.com/employeesnew">
5     <employee name="Jenn Brown" salary="2300"/>
6     <employee name="Percy Indigo" salary="1415"/>
7 </employeeelist>
```

Fig. 22.25 | Sample new-format input for the XMLCombine application.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <employeeelist xmlns="http://www.deitel.com/employeesnew">
3     <employee name="Jenn Brown" salary="2300" />
4     <employee name="Percy Indigo" salary="1415" />
5     <employee name="Christopher Green" salary="1460" />
6     <employee name="Michael Red" salary="1420" />
7 </employeeelist>
```

Fig. 22.26 | XML file generated by XMLCombine (Fig. 22.23).

Lines 10–13 of Fig. 22.23 define `XNamespace` objects for the two namespaces used in the input XML documents. There’s an implicit conversion from `string` to `XNamespace`.

The `TransformDocument` method (lines 32–40) converts an XML document from the old format to the new format. It creates a new `XElement` `newDocumentRoot`, passing the desired name and child elements as arguments. It then creates and returns a new `XDocument`, with `newDocumentRoot` as its root element.

The first argument (line 35) creates an `XName` object for the tag name using the `XNamespace`’s overloaded `+` operator—the `XName` contains the `XNamespace` from the left operand and the local name given by the `string` in the right operand. Recall that you can use `XName`’s `LocalName` property to access the element’s unqualified name. The **Namespace property** gives you access to the contained `XNamespace` object. The second argument is the result of a LINQ query (lines 36–37), which uses the `TransformEmployee` method to transform each `employeeisting` entry in the old format (returned by calling the `Elements` method on the root of the old document) into an `employee` entry in the new format. When passed a collection of `XElements`, the `XElement` constructor adds all members of the collection as children.

The `TransformEmployee` method (lines 43–55) reformats the data for one employee. It does this by retrieving the text contained in the child elements of each of the `employeeisting` entries, then creating a new `employee` element and returning it. The expressions passed to the `Element` method use `XNamespaces`—this is necessary because the elements they’re retrieving are in the old namespace. Passing just the tag’s local name would cause the `Element` method to return `null`, creating a `NullReferenceException` when the `Value` property was accessed.

Once we’ve retrieved the values from the original XML document, we add them as attributes to an `employee` element. This is done by creating new `XAttribute` objects with the attribute’s name and value, and passing these to the `XElement` constructor (lines 52–54).

The `SaveFinalDocument` method (lines 59–70) merges the two documents and saves them to disk. It first creates a new root element in the correct namespace (line 63). Then it adds the `employee` elements from both documents as children using the `Add` method defined in the `XContainer` class (lines 66–67). The `Add` method, like `XElement`'s constructor, will add all elements if passed a collection. After creating and filling the new root, we save it to disk (line 69).

22.11 XSLT with Class `XsltCompiledTransform`

Recall from Section 22.7 that XSL elements define rules for transforming one type of XML document to another type of XML document. We showed how to transform XML documents into XHTML documents and displayed the results in Edge. The XSLT processor built into Edge performed the transformations. We now perform a similar task in a C# program.

Performing an XSL Transformation in C# Using the .NET Framework

Figure 22.27 applies the style sheet `sports.xsl` (Fig. 22.15) to the XML document `sports.xml` (Fig. 22.14) programmatically. The result of the transformation is written to an XHTML file on disk and displayed in a text box. Figure 22.27(c) shows the resulting XHTML document (`sports.html`) when you view it in Edge.

```

1 // Fig. 22.27: TransformTestForm.cs
2 // Applying an XSLT style sheet to an XML Document.
3 using System;
4 using System.IO;
5 using System.Windows.Forms;
6 using System.Xml.Xsl; // contains class XsltCompiledTransform
7
8 namespace TransformTest
9 {
10     public partial class TransformTestForm : Form
11     {
12         public TransformTestForm()
13         {
14             InitializeComponent();
15         }
16
17         // applies the transformation
18         private XsltCompiledTransform transformer;
19
20         // initialize variables
21         private void TransformTestForm_Load(object sender, EventArgs e)
22         {
23             transformer = new XsltCompiledTransform(); // create transformer
24
25             // load and compile the style sheet
26             transformer.Load("sports.xsl");
27         }
28

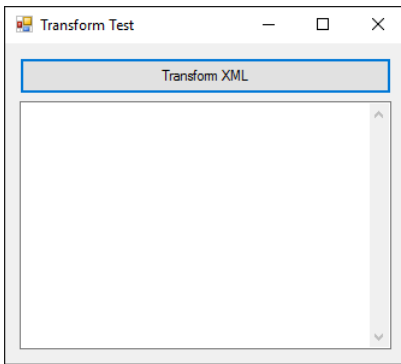
```

Fig. 22.27 | Applying an XSLT style sheet to an XML document. (Part 1 of 2.)

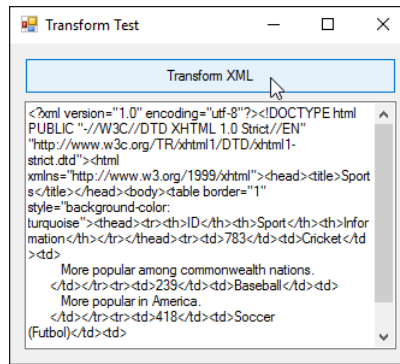
```

29     // transform data on transformButton_Click event
30     private void transformButton_Click(object sender, EventArgs e)
31     {
32         // perform the transformation and store the result in new file
33         transformer.Transform("sports.xml", "sports.html");
34
35         // read and display the XHTML document's text in a TextBox
36         consoleTextBox.Text = File.ReadAllText("sports.html");
37     }
38 }
39 }
    
```

a) Initial GUI



b) GUI showing transformed raw XHTML



c) Transformed XHTML rendered in Edge

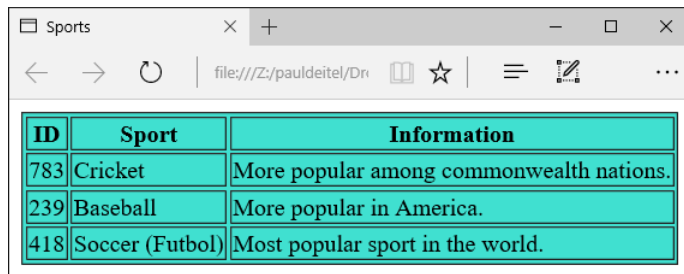


Fig. 22.27 | Applying an XSLT style sheet to an XML document. (Part 2 of 2.)

Line 6 imports the `System.Xml.Xsl` namespace, which contains class `XsltCompiledTransform` for applying XSL style sheets to XML documents. Line 18 declares `XsltCompiledTransform` object `transformer`, which serves as an XSLT processor to transform XML data from one format to another.

In event handler `TransformTestForm_Load` (lines 21–27), line 23 creates and initializes `transformer`. Line 26 calls the `XsltCompiledTransform` object’s `Load` method, which loads and parses the style sheet that this application uses. This method takes an argument specifying the name and location of the style sheet—`sports.xsl` (Fig. 22.15) located in the current directory.

The event handler `transformButton_Click` (lines 30–37) calls the **Transform** method of class `XsltCompiledTransform` to apply the style sheet (`sports.xsl`) to `sports.xml` (line 33). This method takes two string arguments—the first specifies the XML file to which the style sheet should be applied, and the second specifies the file in which the result of the transformation should be stored on disk. Thus the `Transform` method call in line 33 transforms `sports.xml` to XHTML and writes the result to disk as the file `sports.html`. Figure 22.27(c) shows the new XHTML document rendered in Edge. The output is identical to that of Fig. 22.14—in the current example, though, the XHTML is stored on disk rather than generated dynamically.

After applying the transformation, the program displays the content of the new file `sports.html` in `consoleTextBox`, as shown in Fig. 22.27(b). Line 36 obtains the text of the file by passing its name to method `ReadAllText` of the `System.IO.File` class, which simplifies file-processing tasks on the local system.

22.12 Wrap-Up

In this chapter, we introduced XML and several related technologies. We discussed how to create DTDs and schemas for specifying and validating the structure of an XML document. We showed how to use various tools to confirm whether XML documents are valid (i.e., conform to a DTD or schema).

You learned how to create and use XSL documents to specify rules for converting XML documents between formats. Specifically, you learned how to format and sort XML data and output it as XHTML for display in a web browser.

The final sections of the chapter presented more advanced uses of XML in C# applications. We demonstrated how to retrieve and display data from an XML document using various .NET classes. We illustrated how a DOM tree represents each element of an XML document as a node in the tree. The chapter also demonstrated loading data from an XML document using the `Load` method of the `XDocument` class. We demonstrated the tools LINQ to XML provides for working with namespaces. Finally, we showed how to use the `XsltCompiledTransform` class to perform XSL transformations.