15. XML

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

> ***I played with an idea, and grew willful; tossed it into the air; transformed it; let it escape and recaptured it; made it irides-cent with fancy, and winged it with paradox.***
>
> **—Oscar Wilde**

Objectives

In this chapter you'll:

- Mark up data using XML.
- Learn how XML namespaces help provide unique XML element and attribute names.
- Create DTDs and schemas for specifying and validating the structure of an XML document.
- Create and use simple XSL style sheets to render XML document data.
- Retrieve and manipulate XML data programmatically using JavaScript.

Outline

15.1 Introduction

15.2 XML Basics

15.3 Structuring Data

15.4 XML Namespaces

15.5 Document Type Definitions (DTDs)

15.6 W3C XML Schema Documents

15.7 XML Vocabularies

15.7.1 MathML™

15.7.2 Other Markup Languages

15.8 Extensible Stylesheet Language and XSL Transformations

15.9 Document Object Model (DOM)

15.10 Web Resources

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

15.1. Introduction

The **Extensible Markup Language (XML)** was developed in 1996 by the **World Wide Web Consortium's (W3C's)** XML Working Group. XML is a widely supported **open technology** (i.e., nonproprietary technology) for describing data that has become the standard format for data exchanged between applications over the Internet.

Web applications use XML extensively, and web browsers provide many XML-related capabilities. Sections 15.2–15.7 introduce XML and XML-related technologies—XML namespaces for providing unique XML element and attribute names, and Document Type Definitions (DTDs) and XML Schemas for validating XML documents. These sections support the use of XML in many subsequent chapters. Sections 15.8–15.9 present additional XML technologies and key JavaScript capabilities for loading and manipulating XML documents programmatically—this material is optional but is recommended if you plan to use XML in your own applications.

15.2. XML Basics

XML permits document authors to create **markup** (i.e., a text-based notation for describing data) for virtually any type of information, enabling 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 human beings can understand and computers can process.

Figure 15.1 is a simple XML document that describes information for a baseball player. We focus on lines 5–9 to introduce basic XML syntax. You'll learn about the other elements of this document in Section 15.3.

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 15.1: player.xml -->
4 <!-- Baseball player structured with XML -->
5 <player>
6    <firstName>John</firstName>
7    <lastName>Doe</lastName>
8    <battingAverage>0.375</battingAverage>
9 </player>
```

Fig. 15.1. XML that describes a baseball player's information.

XML Elements

XML documents contain text that represents content (i.e., data), such as `John` (line 6 of Fig. 15.1), and **elements** that specify the document's structure, such as `firstName` (line 6 of Fig. 15.1). XML documents delimit elements with **start tags** and **end tags**. A start tag consists of the element name in **angle brackets** (e.g., `<player>` and `<firstName>` in lines 5 and 6, respectively). An end tag consists of the element name preceded by a **forward slash (/)** in angle brackets (e.g., `</firstName>` and `</player>`

in lines 6 and 9, respectively). An element's start and end tags enclose text that represents a piece of data (e.g., the `player`'s `firstName` — `John` —in line 6, which is enclosed by the `<firstName>` start tag and `</firstName>` end tag). Every XML document must have exactly one **root element** that contains all the other elements. In Fig. 15.1, the root element is `player` (lines 5–9).

XML Vocabularies

XML-based markup languages—called XML **vocabularies**—provide a means for describing particular types of data in standardized, structured ways. Some XML vocabularies include XHTML (Extensible HyperText Markup Language), MathML™ (for mathematics), VoiceXML™ (for speech), CML (Chemical Markup Language—for chemistry), XBRL (Extensible Business Reporting Language—for financial data exchange) and others that we discuss in Section 15.7.

Massive amounts of data are currently stored on the Internet in many formats (e.g., databases, web pages, text files). 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 business applications to communicate in common languages. For example, many web services allow web-based applications to exchange data seamlessly through standard protocols based on XML.

The next generation of the web is being built on an XML foundation, enabling you to develop 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 in a simple web page as a string of characters without any real meaning. In an XML document, however, this data can be clearly identified (i.e., 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 ap-

plication can recognize names of people, dates, ISBN numbers and any other type of XML-encoded data. The application can then present users with other related information, providing a richer, more meaningful user experience.

Viewing and Modifying XML Documents

XML documents are highly portable. Viewing or modifying an XML document—which is a text file that usually ends with the `.xml` filename 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. Any text editor that supports ASCII/Unicode characters can open XML documents for viewing and editing. Also, most web browsers can display XML documents in a formatted manner that shows the XML's structure (as we show in Section 15.3). An important characteristic of XML is that it's both human 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 an XML document's contents, 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 (i.e., 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 15.3. 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. XML parsers are often built into browsers and other software.

Validating XML Documents

An XML document can reference a **Document Type Definition (DTD)** or a **schema** that defines the document's proper structure. When an XML document references a DTD or a schema, some parsers (called **validating parsers**) can read it and check that the XML document follows the structure it defines. If the XML document conforms to the DTD/schema (i.e., has the appropriate structure), the document is **valid**. For example, if in Fig. 15.1 we were referencing a DTD that specified that a `player` element must have `firstName`, `lastName` and `battingAverage` elements, then omitting the `lastName` element (line 7 in Fig. 15.1) would invalidate the XML document `player.xml`. However, it would still be well-formed, because it follows proper XML syntax (i.e., it has one root element, each element has a start tag and an end tag, and the elements are nested properly). 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, not whether it's valid.

We discuss validation, DTDs and schemas, as well as the key differences between these two types of structural specifications, in Sections 15.5–15.6. For now, note that schemas are XML documents themselves, whereas DTDs are not. As you'll learn in Section 15.6, this difference presents several advantages in using schemas over DTDs.

SOFTWARE ENGINEERING OBSERVATION 15.1

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

Most XML documents contain only data, so applications that process XML documents must decide how to manipulate or display the data. For exam-

ple, 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 15.8.

XML-processing programs can also search, sort and manipulate XML data using 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 15.8.

15.3. Structuring Data

In this section and throughout this chapter, we create our own XML markup. XML allows you to describe data precisely in a well-structured format.

XML Markup for an Article

In Fig. 15.2, 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
3  <!-- Fig. 15.2: article.xml -->
4  <!-- Article structured with XML -->
5  <article>
6     <title>Simple XML</title>
7     <date>July 4, 2007</date>
8     <author>
9        <firstName>John</firstName>
10       <lastName>Doe</lastName>
11    </author>
```

```
12    <summary>XML is pretty easy.</summary>
13    <content>This chapter presents examples that use XML.</content>
14 </article>
```

Fig. 15.2. XMLusedto mark up an article.

XML Declaration

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. The current XML standard is version 1.0 . Though the W3C released a version 1.1 specification in February 2004, this newer version is not yet widely supported. The W3C may continue to release new versions as XML evolves to meet the requirements of different fields.

PORTABILITY TIP 15.1

Documents should include the XML declaration to identify the version of XML used. A document that lacks an XML declaration might be assumed to conform to the latest version of XML —when it does not, errors could result.

Blank Space and Comments

As in most markup languages, blank lines (line 2), white spaces and indentation help improve readability. Blank lines are normally ignored by XML parsers. XML comments (lines 3–4), which begin with `<!--` and end with `-->`, can be placed almost anywhere in an XML document and can span multiple lines. There must be one end marker (`-->`) for each begin marker (`<!--`).

COMMON PROGRAMMING ERROR 15.1

Placing any characters, including white space, before the XML declaration is an error.

COMMON PROGRAMMING ERROR 15.2

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 15.3

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

Root Node and XML Prolog

In Fig. 15.2, `article` (lines 5–14) 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.

XML Element Names

The elements we use 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 elements to mark up your data. For example, element `title` (line 6) contains text that describes the article's title (e.g., `Simple XML`). Similarly, `date` (line 7), `author` (lines 8–11), `firstName` (line 9), `lastName` (line 10), `summary`

(line 12) and `content` (line 13) 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 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 "`xml`" in any combination of uppercase and lowercase letters (e.g., `XML`, `Xml`, `xMl`), as this is reserved for use in the XML standards.

COMMON PROGRAMMING ERROR 15.4

Using a white-space character in an XML element name is an error.

GOOD PROGRAMMING PRACTICE 15.1

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

Nesting XML Elements

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 items. For example, elements `title`, `date`, `author`, `summary` and `content` are children of `article`. Elements `firstName` and `lastName` are children of `author`. We discuss the hierarchy of Fig. 15.2 later in this chapter (Fig. 15.23).

COMMON PROGRAMMING ERROR 15.5

Nesting XML tags improperly is a syntax error. 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 (e.g., `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. If those child elements are at the same nesting level, they're **siblings** of one another.

Viewing an XML Document in a Web Browser

The XML document in Fig. 15.2 is simply a text file named `article.xml`. It does not contain formatting information for the article. This is because XML is a technology for describing the structure of data. The formatting and displaying of data from an XML document are application-specific issues. For example, when the user loads `article.xml` in a web browser, the browser parses and displays the document's data. Each browser has a built-in **style sheet** to format the data. The resulting format of the data (Fig. 15.3) is similar to the format of the listing in Fig. 15.2. In Section 15.8, we show how you can create your own style sheets to transform XML data into formats suitable for display.

The down arrow (▼) and right arrow (▶) in the screen shots of Fig. 15.3 are not part of the XML document. Google Chrome places them next to every container element. A down arrow indicates that the browser is displaying the container element's child elements. Clicking the down arrow next to an element collapses that element (i.e., causes the browser to hide the container element's children and replace the down arrow with a right arrow). Conversely, clicking the right arrow next to an element expands that element (i.e., causes the browser to display the container element's children and replace the right arrow with a down arrow). This behavior is similar to viewing the directory structure in a file-manager window (like Windows Explorer on Windows or Finder on Mac OS X) or another similar directory viewer. In fact, a directory structure often is modeled as a series of tree structures, in which the **root** of a tree represents a disk drive (e.g., `C:`), and **nodes** in the tree represent directories. Parsers often store XML data as tree structures to facilitate efficient manipulation, as discussed in Section 15.9. [*Note:* Some browsers display minus and plus

signs, rather than down and right arrows.]

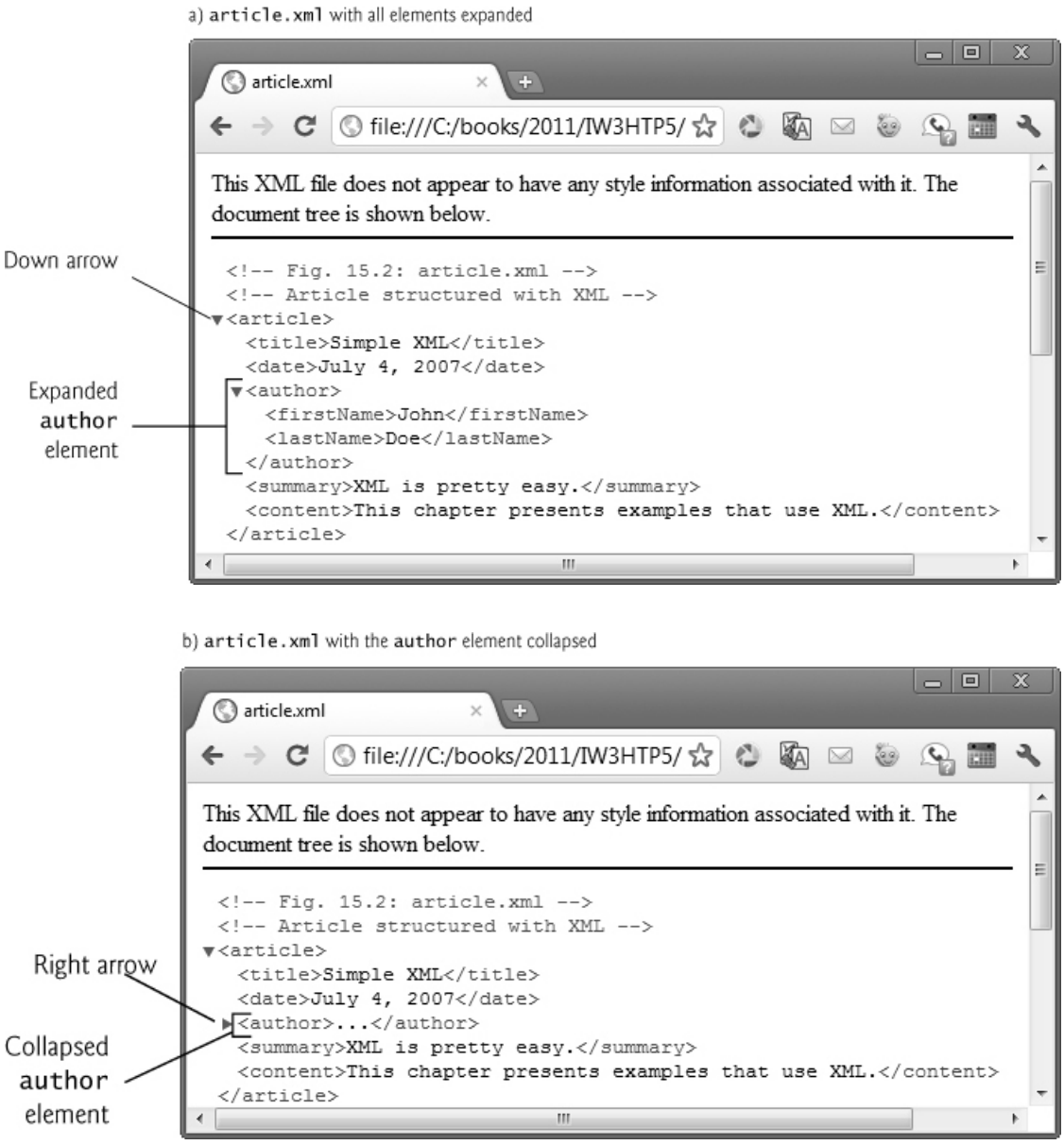


Fig. 15.3. `article.xml` displayed in the Google Chrome browser.

XML Markup for a Business Letter

Now that you've seen a simple XML document, let's examine a more complex one that marks up a business letter (Fig. 15.4). 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
3 <!-- Fig. 15.4: letter.xml -->
```

```
<!-- Business letter marked up with XML -->
<!DOCTYPE letter SYSTEM "letter.dtd">

<letter>
   <contact type = "sender">
      <name>Jane Doe</name>
      <address1>Box 12345</address1>
      <address2>15 Any Ave.</address2>
      <city>Othertown</city>
      <state>Otherstate</state>
      <zip>67890</zip>
      <phone>555-4321</phone>
      <flag gender = "F" />
   </contact>

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

   <salutation>Dear Sir:</salutation>

   <paragraph>It is our privilege to inform you about our new database
      managed with XML. This new system allows you to reduce the
      load on your inventory list server by having the client machine
      perform the work of sorting and filtering the data.
   </paragraph>

   <paragraph>Please visit our website for availability and pricing.
   </paragraph>
```

```
40
41    <closing>Sincerely,</closing>
42    <signature>Ms. Jane Doe</signature>
43 </letter>
```

Fig. 15.4. Business letter marked up with XML.

Line 5 specifies that this XML document references a DTD. Recall from Section 15.2 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 15.1

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 15.2

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

`DOCTYPE`

The `DOCTYPE` 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 (i.e., `letter.dtd` in the current directory; this could also be a fully quali-

fied URL). DTD document filenames typically end with the `.dtd` extension. We discuss DTDs and `letter.dtd` in detail in Section 15.5.

Validating an XML Document Against a DTD

Many online tools can validate your XML documents against DTDs (Section 15.5) or schemas (Section 15.6). The validator at

http://www.xmlvalidation.com/

can validate XML documents against either DTDs or schemas. To use it, you can either paste your XML document's code into a text area on the page or upload the file. If the XML document references a DTD, the site asks you to paste in the DTD or upload the DTD file. You can also select a checkbox for validating against a schema instead. You can then click a button to validate your XML document.

The XML Document's Contents

Root element `letter` (lines 7–43 of Fig. 15.4) contains the child elements `contact`, `contact`, `salutation`, `paragraph`, `paragraph`, `closing` and `signature`. Data can be placed between an element's tags or as **attributes**—name/value pairs that appear within the angle brackets of an element's start tag. Elements can have any number of attributes (separated by spaces) in their start tags. 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 recipient of the letter. Like element names, attribute names are case sensitive, can be any length, may contain letters, digits, underscores, hyphens and periods, and must begin with either a letter or 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 32–39 mark up the letter's body us-

ing two `paragraph` elements. Elements `closing` (line 41) and `signature` (line 42) mark up the closing sentence and the author's "signature," respectively.

COMMON PROGRAMMING ERROR 15.6

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

Line 16 introduces the **empty element** `flag` . An empty element is one that does not have any content. Instead, it sometimes places data in attributes. Empty element `flag` has one 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 right angle bracket, as shown in line 16, or by explicitly writing an end tag, as in line 22:

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

The `address2` element in line 22 is empty because there's 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 15.5). This DTD specifies that each `contact` element must have an `address2` child element (even if it's empty). In Section 15.5, you'll learn how DTDs indicate required and optional elements.

15.4. XML Namespaces

XML allows document authors to create custom elements. This extensibility can result in **naming collisions** among elements in an XML document that have the same name. 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. XML namespaces provide a means for document authors to unambiguously refer to elements with the same name (i.e., prevent collisions). For example,

<**subject**>Geometry</**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:

<**highschool:subject**>Geometry</**highschool:subject**>

and

<**medicalschool:subject**>Cardiology</**medicalschool:subject**>

Both `highschool` and `medicalschool` 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 prefix `xml` . In the subsections that follow, we demonstrate how document authors ensure that namespaces are unique.

COMMON PROGRAMMING ERROR 15.7

Attempting to create a namespace prefix named `xml` in any mixture of uppercase and lowercase letters is a syntax error—

the `xml` namespace prefix is reserved for internal use by XML itself.

Differentiating Elements with Namespaces

Figure 15.5 demonstrates namespaces. In this document, namespaces differentiate two distinct elements—the `file` element related to a text file and the `file` document related to an image file.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.5: namespace.xml -->
4  <!-- Demonstrating namespaces -->
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. 15.5. XML namespaces demonstration.

The `xmlns` Attribute

Lines 6–7 use the XML-namespace reserved attribute **`xmlns`** to create two namespace prefixes— `text` and `image` . Each namespace prefix is bound to a series of characters called a **Uniform Resource Identifier (URI)** that uniquely identifies the namespace. Document authors create their own

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

Unique URIs

To ensure that namespaces are unique, document authors must provide unique URIs. In this example, we use `urn:deitel:textInfo` and `urn:deitel:imageInfo` as URIs. These URIs employ the URN scheme that is often used to identify namespaces. Under this naming scheme, a URI begins with `"urn:"`, followed by a unique series of additional names separated by colons.

Another common practice is to use URLs, which specify the location of a file or a resource on the Internet. For example, www.deitel.com is the URL that identifies the home page of the Deitel & Associates website. Using URLs guarantees that the namespaces are unique because the domain names (e.g., 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.com` 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. They each simply represent 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 `hgjfkdlsa4556`, in which case our prefix assignment would be

```
xmlns:image = "hgjfkdlsa4556"
```

Namespace Prefix

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 (although they can have them), because each attribute is already part of an element that specifies the namespace prefix. For example, attribute `filename` (line 9) is implicitly part of namespace `text` because its element (i.e., `file`) specifies the `text` namespace prefix.

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 15.6 demonstrates using a default namespace (`urn:deitel:textInfo`) for element `directory`.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.6: defaultnamespace.xml -->
4  <!-- Using default namespaces -->
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. 15.6. Default namespace demonstration.

Line 5 defines a default namespace using attribute `xmlns` with no prefix specified but with a URI as its value. Once we define this, child elements

belonging to the namespace need not be qualified by a namespace prefix. Thus, element `file` (lines 8–10) is in the default namespace `urn:deitel:textInfo`. Compare this to lines 9–10 of Fig. 15.5, where we had to prefix the `file` and `description` element names with the namespace prefix `text`.

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 a particular element. For example, the `file` element in lines 12–15 of Fig. 15.16 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 (Section 15.6) and Extensible Stylesheet Language (XSL) (Section 15.8), 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 elements and user-defined elements.

15.5. Document Type Definitions (DTDs)

Document Type Definitions (DTDs) are one of two main types of documents you can use to specify XML document structure. Section 15.6 presents W3C XML Schema documents, which provide an improved method of specifying XML document structure.

SOFTWARE ENGINEERING OBSERVATION 15.2

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 15.6) solve this problem by providing an extensible way to describe XML docu-

ment structure. Applications should use DTDs or schemas to confirm whether XML documents are valid.

SOFTWARE ENGINEERING OBSERVATION 15.3

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 15.4 presented a simple business letter marked up with XML. Recall that line 5 of `letter.xml` references a DTD— `letter.dtd` (Fig. 15.7). This DTD specifies the business letter's element types and attributes and their relationships to one another.

```
1  <!-- Fig. 15.7: 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 )>
```

```
<!ELEMENT phone ( #PCDATA )>
<!ELEMENT flag EMPTY>
<!ATTLIST flag gender (M | F) "M">

<!ELEMENT salutation ( #PCDATA )>
<!ELEMENT closing ( #PCDATA )>
<!ELEMENT paragraph ( #PCDATA )>
<!ELEMENT signature ( #PCDATA )>
```

Fig. 15.7. Document Type Definition (DTD) for a business letter.

A DTD describes the structure of an XML document and enables an XML parser to verify whether an XML 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 using an EBNF (Extended Backus-Naur Form) grammar. DTDs are not themselves XML documents. [*Note:* EBNF grammars are commonly used to define programming languages. To learn more about EBNF grammars, visit en.wikipedia.org/wiki/EBNF or www.garshol.priv.no/download/text/bnf.html.]

COMMON PROGRAMMING ERROR 15.8

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

Defining Elements in a DTD

The **`ELEMENT` element type declaration** in 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 requires 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 requires 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 if the parser finds a `contact` element without a `type` attribute, the parser can choose an arbitrary value for the attribute or can ignore the attribute. Either way the document will still be valid (if the rest of the document is valid)—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.

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 15.4

DTD syntax cannot describe an element's or attribute's data 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's 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. See Appendix A, HTML Special Characters, for a list of other character entity references.

COMMON PROGRAMMING ERROR 15.9

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 exam-

ple, `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." Note that line 19 also indicates that `gender` has a default value of `M`.

Well-Formed Documents vs. Valid Documents

In Section 15.3, we demonstrated how to use the Microsoft XML Validator to validate an XML document against its specified DTD. The validation revealed that the XML document `letter.xml` (Fig. 15.4) is well-formed and valid—it conforms to `letter.dtd` (Fig. 15.7). 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, etc.), 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.

When a document fails to conform to a DTD or a schema, an XML validator displays an error message. For example, the DTD in Fig. 15.7 indicates that a `contact` element must contain the child element `name`. A document that omits this child element is still well-formed but is not valid. Figure 15.8 shows the error message displayed by the validator at www.xmlvalidation.com for a version of the `letter.xml` file that's missing the first `contact` element's `name` element.

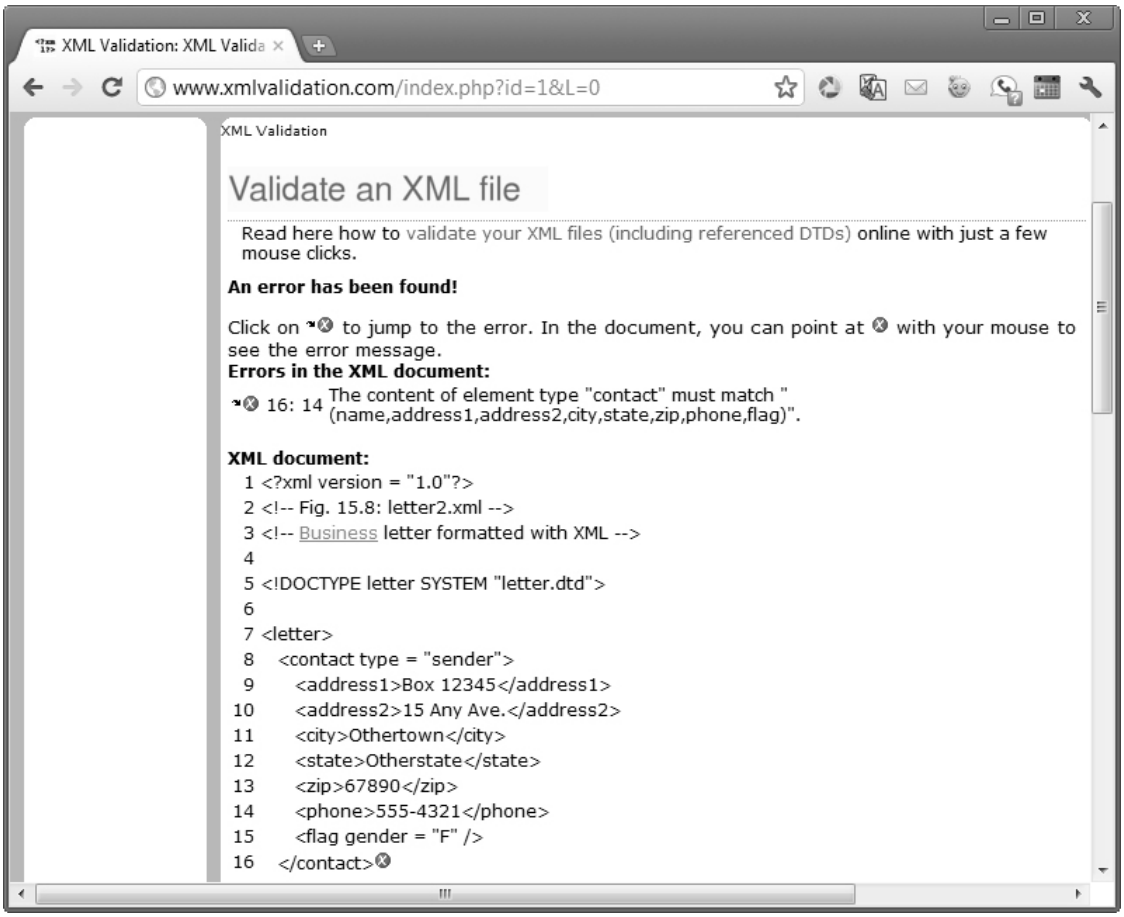


Fig. 15.8. Error message when validating `letter.xml` with a missing contact name.

15.6. 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, forcing developers to learn multiple grammars and developers to create multiple types of parsers. These and other limitations have led to the development of schemas.

Unlike DTDs, schemas do not use EBNF grammar. Instead, they 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 (note the capital "S" in "Schema"). To refer to it, we use the term XML Schema in

the rest of the chapter. 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.

Recall that 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 therefore must themselves be valid.

Validating Against an XML Schema Document

Figure 15.9 shows a schema-valid XML document named `book.xml`, and Fig. 15.10 shows the pertinent XML Schema document (`book.xsd`) that defines the structure for `book.xml`. By convention, schemas use the `.xsd` extension. We used an online XSD schema validator provided at

www.xmlforasp.net/SchemaValidator.aspx

to ensure that the XML document in Fig. 15.9 conforms to the schema in Fig. 15.10. To validate the schema document itself (i.e., `book.xsd`) and produce the output shown in Fig. 15.10, we used an online XSV (XML Schema Validator) provided by the W3C at

www.w3.org/2001/03/webdata/xsv

These tools are free and enforce the W3C's specifications regarding XML Schemas and schema validation.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.9: book.xml -->
4  <!-- Book list marked up as XML -->
5  <deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
6     <book>
7        <title>Visual Basic 2010 How to Program</title>
8     </book>
9     <book>
10       <title>Visual C# 2010 How to Program, 4/e</title>
11    </book>
12    <book>
13       <title>Java How to Program, 9/e</title>
14    </book>
15    <book>
16       <title>C++ How to Program, 8/e</title>
17    </book>
18    <book>
19       <title>Internet and World Wide Web How to Program, 5/e</title>
20    </book>
21 </deitel:books>
```

Fig. 15.9. Schema-valid XML document describing a list of books.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.10: book.xsd          -->
4  <!-- Simple W3C XML Schema document -->
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>
```

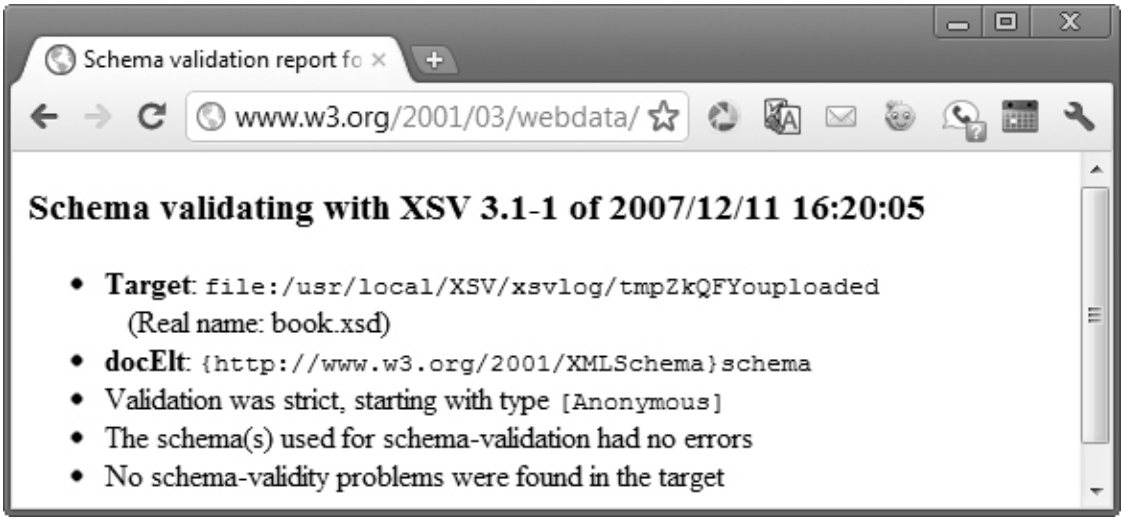


Fig. 15.10. XML Schema document for `book.xml`.

Figure 15.9 contains markup describing several Deitel books. The `books` element (line 5) has the namespace prefix `deitel`, indicating that the `books` element is a part of the http://www.deitel.com/booklist namespace.

Creating an XML Schema Document

Figure 15.10 presents the XML Schema document that specifies the structure of `book.xml` (Fig. 15.9). 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.

Root element `schema` (Fig. 15.10, 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 15.3

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. Note that the `targetNamespace` of `book.xsd` is the same as the namespace referenced in line 5 of `book.xml` (Fig. 15.9). This is what "connects" the XML document with the schema that defines its structure. When a 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 namespace 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 (line 9 of Fig. 15.10) 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. 15.9). Attributes **name** and **type** specify the `element`'s name and type, respectively. An element's type indicates the data 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 of Fig. 15.10). Figure 15.11 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.

Type	Description	Range or structure	Examples
`string`	A character string		`"hello"`
`boolean`	True or false	`true`, `false`	`true`
`decimal`	A decimal numeral	`i * (10^n)`, where `i` is an integer and `n` is an integer that's less than or equal to zero.	`5`, `-12`, `-45.78`
`float`	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, negative infinity and not-a-number (`NaN`).	`0`, `12`, `-109.375`, `NaN`
`double`	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 (`NaN`).	`0`, `12`, `-109.375`, `NaN`
`long`	A whole number	`-9223372036854775808` to `9223372036854775807`, inclusive.	`1234567890`, `-1234567890`
`int`	A whole number	`-2147483648` to `2147483647`, inclusive.	`1234567890`, `-1234567890`
`short`	A whole number	`-32768` to `32767`, inclusive.	`12`, `-345`
`date`	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 long and `mm` and `dd` are two digits long.	`2005-05-10`
`time`	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. 15.11. Some XML Schema types.

In this example, `books` is defined as an element of type `deitel:BooksType` (line 9). `BooksType` is a user-defined type (lines 11–16 of Fig. 15.10) in the namespace http://www.deitel.com/booklist and therefore must have the namespace prefix `deitel`. It's not an existing XML Schema type.

Two categories of type 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.

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. 15.11) or user-defined types (e.g., `SingleBookType` in Fig. 15.10).

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.

The schema document in Fig. 15.12 creates both simple types and complex types. The XML document in Fig. 15.13 (`laptop.xml`) follows the structure defined in Fig. 15.12 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. 15.12: 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. 15.12. XML Schema document defining simple and complex types.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.13: laptop.xml          -->
4  <!-- Laptop components marked up as XML -->
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. 15.13. XML document using the `laptop` element defined in `computer.xsd`.

Line 5 of Fig. 15.12 declares the default namespace to be the standard XML Schema namespace—any elements without a prefix are assumed to be in that 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.

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.

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's also of type `string`.

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`.

We've now created an element named `laptop` that contains child elements `processor`, `monitor`, `CPUSpeed` and `RAM`, and an attribute `manufacturer`. Figure 15.13 uses the `laptop` element defined in the `computer.xsd` schema. Once again, we used an online XSD schema validator (www.xmlforasp.net/SchemaValidator.aspx) to ensure that this XML instance document adheres to the schema's structural rules.

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.

This section introduced W3C XML Schema documents for defining the structure of XML documents, and we validated XML instance documents against schemas using an online XSD schema validator. Section 15.7 discusses several XML vocabularies and demonstrates the MathML vocabulary.

15.7. XML Vocabularies

XML allows authors to create their own tags to describe data precisely. People and organizations in various fields of study have created many different kinds of XML for structuring data. Some of these markup languages are: **MathML (Mathematical Markup Language)**, **Scalable Vector Graphics (SVG)**, **Wireless Markup Language (WML)**, **Extensible Business Reporting Language (XBRL)**, **Extensible User Interface Language (XUL)** and **Product Data Markup Language (PDML)**. Two other examples of XML vocabularies are W3C XML Schema and the Extensible Stylesheet Language (XSL), which we discuss in Section 15.6 and Section 15.8, respectively. The following subsections describe MathML and other custom markup languages.

15.7.1. MathML™

Until recently, computers typically required specialized software packages such as TeX and LaTeX for displaying complex mathematical expressions. This section introduces MathML, which the W3C developed for describing mathematical notations and expressions. The Firefox and Opera browsers can render MathML. There are also plug-ins or extensions available that enable you to render MathML in other browsers.

MathML markup describes mathematical expressions for display. MathML is divided into two types of markup—**content** markup and **presentation** markup. Content markup provides tags that embody mathe-

matical concepts. Content MathML allows programmers to write mathematical notation specific to different areas of mathematics. For instance, the multiplication symbol has one meaning in set theory and another in linear algebra. Content MathML distinguishes between different uses of the same symbol. Programmers can take content MathML markup, discern mathematical context and evaluate the marked-up mathematical operations. Presentation MathML is directed toward formatting and displaying mathematical notation. We focus on Presentation MathML in the MathML examples.

Simple Equation in MathML

Figure 15.14 uses MathML to mark up a simple expression. For this example, we show the expression rendered in Firefox.

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3     "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5  <!-- Fig. 15.14: mathml1.mml -->
6  <!-- MathML equation. -->
7  <math xmlns="http://www.w3.org/1998/Math/MathML">
8     <mn>2</mn>
9     <mo>+</mo>
10    <mn>3</mn>
11    <mo>=</mo>
12    <mn>5</mn>
13 </math>
```

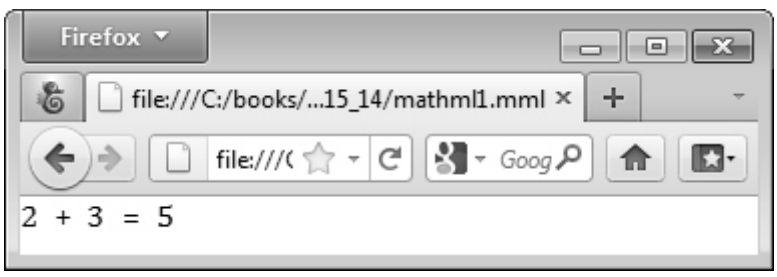


Fig. 15.14. Expression marked up with MathML and displayed in the Firefox browser.

By convention, MathML files end with the `.mml` filename extension. A MathML document's root node is the `math` element, and its default namespace is http://www.w3.org/1998/Math/MathML (line 7). The **`mn` element** (line 8) marks up a number. The **`mo` element** (line 9) marks up an operator (e.g., +). Using this markup, we define the expression 2 + 3 = 5, which any MathML capable browser can display.

Algebraic Equation in MathML

Let's consider using MathML to mark up an algebraic equation containing exponents and arithmetic operators (Fig. 15.15). For this example, we again show the expression rendered in Firefox.

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3    "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5  <!-- Fig. 15.15: mathml2.html -->
6  <!-- MathML algebraic equation. -->
7  <math xmlns="http://www.w3.org/1998/Math/MathML">
8    <mn>3</mn>
9    <mo>&InvisibleTimes;</mo>
10   <msup>
11     <mi>x</mi>
12     <mn>2</mn>
13   </msup>
14   <mo>+</mo>
15   <mn>x</mn>
16   <mo>&minus;</mo>
17   <mfrac>
18     <mn>2</mn>
19     <mi>x</mi>
20   </mfrac>
```

```
21    <mo>=</mo>
22    <mn>0</mn>
23 </math>
```

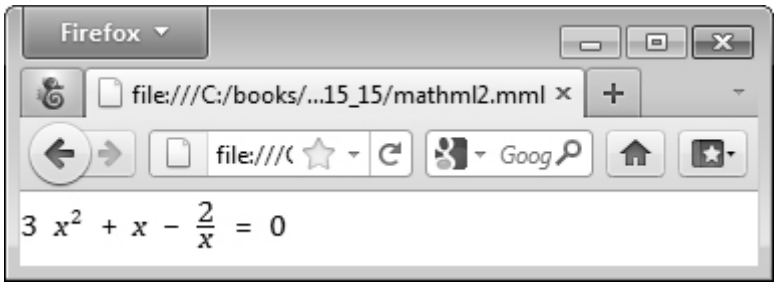


Fig. 15.15. Algebraic equation marked up with MathML and displayed in the Firefox browser.

Line 9 uses **entity reference** `&InvisibleTimes;` to indicate a multiplication operation without explicit **symbolic representation** (i.e., the multiplication symbol does not appear between the `3` and `x`). For exponentiation, lines 10–13 use the **`msup` element**, which represents a superscript. This `msup` element has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Correspondingly, the `msub` element represents a subscript. To display variables such as `x`, line 11 uses **identifier element** `mi`.

To display a fraction, lines 17–20 use the **`mfrac` element**. Lines 18–19 specify the numerator and the denominator for the fraction. If either the numerator or the denominator contains more than one element, it must appear in an `mrow` element.

Calculus Expression in MathML

Figure 15.16 marks up a calculus expression that contains an integral symbol and a square-root symbol.

```
1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3    "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
```

```
<!-- Fig. 15.16 mathml3.html -->
<!-- Calculus example using MathML -->
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mrow>
    <msubsup>
      <mo>&int;</mo>
      <mn>0</mn>
      <mrow>
        <mn>1</mn>
        <mo>&minus;</mo>
        <mi>y</mi>
      </mrow>
    </msubsup>
    <msqrt>
      <mn>4</mn>
      <mo>&InvisibleTimes;</mo>
      <msup>
        <mi>x</mi>
        <mn>2</mn>
      </msup>
      <mo>+</mo>
      <mi>y</mi>
    </msqrt>
    <mo>&delta;</mo>
    <mi>x</mi>
  </mrow>
</math>
```

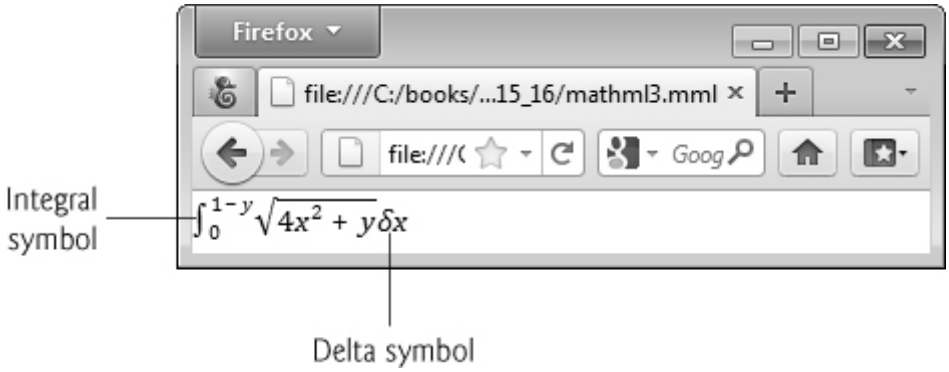


Fig. 15.16. Calculus expression marked up with MathML and displayed in the Firefox browser.

Lines 8–30 group the entire expression in an **`mrow` element**, which is used to group elements that are positioned horizontally in an expression. The entity reference **`&int;`** (line 10) represents the integral symbol, while the **`msubsup` element** (lines 9–17) specifies the subscript and superscript for a base expression (e.g., the integral symbol). Element `mo` marks up the integral operator. The `msubsup` element requires three child elements—an operator (e.g., the integral entity, line 10), the subscript expression (line 11) and the superscript expression (lines 12–16). Element `mn` (line 11) marks up the number (i.e., `0`) that represents the subscript. Element `mrow` (lines 12–16) marks up the superscript expression (i.e., `1-` *y*).

Element **`msqrt`** (lines 18–27) represents a square-root expression. Line 28 introduces entity reference `&delta;` for representing a lowercase delta symbol. Delta is an operator, so line 28 places this entity in element `mo`. To see other operations and symbols in MathML, visit www.w3.org/Math.

15.7.2. Other Markup Languages

Literally hundreds of markup languages derive from XML. Every day developers find new uses for XML. Figure 15.18 summarizes a few of these markup languages. The website

www.service-architecture.com/xml/articles/index.html

provides a nice list of common XML vocabularies and descriptions.

Markup language	Description
Chemical Markup Language (CML)	Chemical Markup Language (CML) is an XML vocabulary for representing molecular and chemical information. Many previous methods for storing this type of information (e.g., special file types) inhibited document reuse. CML takes advantage of XML's portability to enable document authors to use and reuse molecular information without corrupting important data in the process.
VoiceXML™	The VoiceXML Forum founded by AT&T, IBM, Lucent and Motorola developed VoiceXML. It provides interactive voice communication between humans and computers through a telephone, PDA (personal digital assistant) or desktop computer. IBM's VoiceXML SDK can process VoiceXML documents. Visit `www.voicexml.org` for more information on VoiceXML.
Synchronous Multimedia Integration Language (SMIL™)	SMIL is an XML vocabulary for multimedia presentations. The W3C was the primary developer of SMIL, with contributions from some companies. Visit `www.w3.org/AudioVideo` for more on SMIL.
Research Information Exchange Markup Language (RIXML)	RIXML, developed by a consortium of brokerage firms, marks up investment data. Visit `www.rixml.org` for more information on RIXML.
Geography Markup Language (GML)	OpenGIS developed the Geography Markup Language to describe geographic information. Visit `www.opengis.org` for more information on GML.
Extensible User Interface Language (XUL)	The Mozilla Project created the Extensible User Interface Language for describing graphical user interfaces in a platform-independent way.

Fig. 15.17. Various markup languages derived from XML.

15.8. 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 HTML5 document that presents the XML document's data (or a subset of the data) formatted for display in a web browser.

Transforming an XML document using XSLT involves two tree structures—the **source tree** (i.e., the XML document to transform) and the **result tree** (i.e., the XML document to create). XPath locates parts of the source-tree document that match **templates** defined in an **XSL style sheet**. When a match occurs, 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 XPath'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 15.18 lists an XML document that describes various sports. The output shows the result of the transformation (specified in the XSLT template of Fig. 15.19). Some web browsers will perform transformations on XML files only if they are accessed from a web server. For this reason, we've posted the example online at

http://test.deitel.com/iw3htp5/ch15/Fig15_18-19/sports.xml

Also, to save space, we do not show the contents of the example's CSS file here.

```
1 <?xml version = "1.0"?>
2 <?xml-stylesheet type = "text/xsl" href = "sports.xsl"?>
3
4 <!-- Fig. 15.18: sports.xml -->
5 <!-- Sports Database -->
6
```

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

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

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

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

  <game id = "418">
    <name>Soccer (Futbol)</name>

    <paragraph>
      Most popular sport in the world.
    </paragraph>
  </game>
</sports>
```

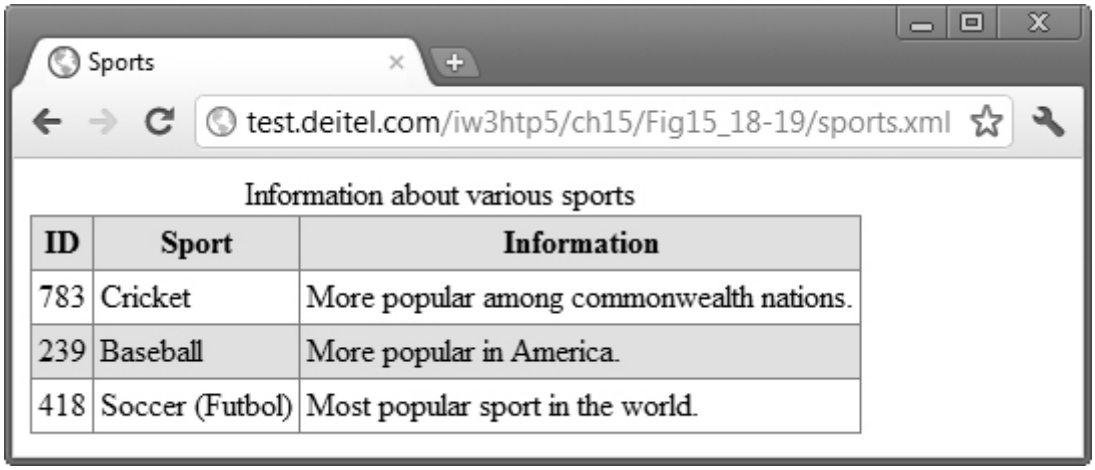


Fig. 15.18. XML document that describes various sports.

To perform transformations, an XSLT processor is required. Popular XSLT processors include Microsoft's MSXML and the Apache Software Foundation's **Xalan 2** (`xml.apache.org`). The XML document in Fig. 15.18 is transformed into an XHTML document when it's loaded into the web browser.

Line 2 (Fig. 15.18) is a **processing instruction (PI)** that references the XSL style sheet `sports.xsl` (Fig. 15.19). 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 `<?` and `?>` (line 2, Fig. 15.18) 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 15.5

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

COMMON PROGRAMMING ERROR 15.10

You'll sometimes see the XML processing instruction `<?xml-stylesheet?>` written as `<?xml:stylesheet?>` with a colon rather than a dash. The version with a colon results in an XML parsing error in Firefox.

Figure 15.19 shows the XSL document for transforming the structured data of the XML document of Fig. 15.18 into an XHTML document for presentation. By convention, XSL documents have the filename extension `.xsl`.

```
1  <?xml version = "1.0"?>
2  <!-- Fig. 15.19: sports.xsl -->
3  <!-- A simple XSLT transformation -->
4
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 = "html" doctype-system = "about:legacy-compat" />
10    <xsl:template match = "/"> <!-- match root element -->
11
12    <html xmlns = "http://www.w3.org/1999/xhtml">
13       <head>
14          <meta charset = "utf-8"/>
15          <link rel = "stylesheet" type = "text/css" href = "style.css"/>
16          <title>Sports</title>
17       </head>
18
19       <body>
20          <table>
21          <caption>Information about various sports</caption>
22             <thead>
23                <tr>
24                   <th>ID</th>
25                   <th>Sport</th>
26                   <th>Information</th>
27                </tr>
28             </thead>
29
```

```
30          <!-- insert each name and paragraph element value -->
31          <!-- into a table row. -->
32          <xsl:for-each select = "/sports/game">
33             <tr>
34                <td><xsl:value-of select = "@id"/></td>
35                <td><xsl:value-of select = "name"/></td>
36                <td><xsl:value-of select = "paragraph"/></td>
37             </tr>
38          </xsl:for-each>
39       </table>
40    </body>
41 </html>
42
43 </xsl:template>
44 </xsl:stylesheet>
```

Fig. 15.19. XSLT that creates elements and attributes in an HTML5 document.

Lines 6–7 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).

Outputting the `DOCTYPE`

Line 9 uses element **`xsl:output`** to write an HTML5 document type declaration (`DOCTYPE`) to the result tree (i.e., the XML document to be created). At the time of this writing, the W3C has not yet updated the XSLT recommendation (standard) to support the HTML5 `DOCTYPE` —in the meantime, they recommend setting the attribute `doctype-system` to the value `about:legacy-compat` to produce an HTML5 compatible `DOCTYPE` using XSLT.

Templates

XSLT uses **templates** (i.e., `xsl: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 required `match` attribute. Line 10 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) always selects the document root. 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're using **absolute addressing** (i.e., we're starting from the root and defining paths down the source tree). In the XML document of Fig. 15.18, 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. 15.19, line 14, matches a node (i.e., the root node), so the contents of the template are now added to the result tree.

Repetition in XSL

The browser's XML processor writes the HTML5 in lines 13–28 (Fig. 15.19) to the result tree exactly as it appears in the XSL document. Now the result tree consists of the `DOCTYPE` definition and the HTML5 code from lines 13–28. Lines 32–38 use element `xsl:for-each` to iterate through the source XML document, searching for `game` elements. Attribute `select` is an XPath expression that specifies the nodes (called the **node set**) on which the `xsl:for-each` operates. Again, the first forward slash means that we're using absolute addressing. The forward slash between `sports` and `game` indicates that `game` is a child node of `sports`. Thus, the `xsl: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 node. After finding the elements that match the selection criteria, the `xsl:for-each` processes each element with the code in lines 33–37 (these lines produce one row in a table each time they execute) and places the result in the result tree.

Line 34 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 context node `game`. Lines 35–36 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 `xsl: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 15.20 presents an XML document (`sorting.xml`) that marks up information about a book. Note that 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 2 (`sorting.xsl`) can sort the XML file's data for presentation purposes.

```
1  <?xml version = "1.0"?>
2  <?xml-stylesheet type = "text/xsl" href = "sorting.xsl"?>
3
4  <!-- Fig. 15.20: sorting.xml -->
5  <!-- XML document containing book information -->
6  <book isbn = "999-99999-9-X">
7     <title>Deitel&apos;s XML Primer</title>
8
9     <author>
10       <firstName>Jane</firstName>
11       <lastName>Blue</lastName>
12    </author>
13
14    <chapters>
15       <frontMatter>
16          <preface pages = "2" />
```

```
17         <contents pages = "5" />
18         <illustrations pages = "4" />
19       </frontMatter>
20
21       <chapter number = "3" pages = "44">Advanced XML</chapter>
22       <chapter number = "2" pages = "35">Intermediate XML</chapter>
23       <appendix number = "B" pages = "26">Parsers and Tools</appendix>
24       <appendix number = "A" pages = "7">Entities</appendix>
25       <chapter number = "1" pages = "28">XML Fundamentals</chapter>
26     </chapters>
27
28     <media type = "CD" />
29   </book>
```

Fig. 15.20. XML document containing book information.

Figure 15.21 presents an XSL document (`sorting.xsl`) for transforming the XML document `sorting.xml` (Fig. 15.20) to HTML5. (To save space, we do not show the contents of the example's CSS file here.) 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 HTML5. Line 12 of Fig. 15.21 matches the root element of the document in Fig. 15.20. Line 13 outputs an `html` start tag to the result tree. In line 14, the `<xsl:apply-templates/>` element specifies that the XSLT processor is to apply the `xsl:template` s 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 15. You can view the results of the transformation at:

http://test.deitel.com/iw3htp5/ch15/Fig15_20-21/sorting.xml

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.21: sorting.xsl -->
4  <!-- Transformation of book information into HTML5 -->
```

```
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <!-- write XML declaration and DOCTYPE DTD information -->
   <xsl:output method = "html" doctype-system = "about:legacy-compat" />

    <!-- match document root -->
    <xsl:template match = "/">
      <html>
        <xsl:apply-templates/>
      </html>
    </xsl:template>

    <!-- match book -->
    <xsl:template match = "book">
      <head>
        <meta charset = "utf-8"/>
        <link rel = "stylesheet" type = "text/css" href = "style.css"/>
        <title>ISBN <xsl:value-of select = "@isbn"/> -
          <xsl:value-of select = "title"/></title>
      </head>

      <body>
        <h1><xsl:value-of select = "title"/></h1>
        <h2>by
          <xsl:value-of select = "author/lastName"/>,
          <xsl:value-of select = "author/firstName"/></h2>

        <table>

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

```

```
41              <td>
42                ( <xsl:value-of select = "@pages"/> pages )
43              </td>
44            </tr>
45          </xsl:for-each>
46
47          <xsl:for-each select = "chapters/chapter">
48            <xsl:sort select = "@number" data-type = "number"
49               order = "ascending"/>
50            <tr>
51              <td>
52                Chapter <xsl:value-of select = "@number"/>
53              </td>
54
55              <td>
56                <xsl:value-of select = "text()"/>
57                ( <xsl:value-of select = "@pages"/> pages )
58              </td>
59            </tr>
60          </xsl:for-each>
61
62          <xsl:for-each select = "chapters/appendix">
63            <xsl:sort select = "@number" data-type = "text"
64              order = "ascending"/>
65            <tr>
66              <td>
67                Appendix <xsl:value-of select = "@number"/>
68              </td>
69
70              <td>
71                <xsl:value-of select = "text()"/>
72                ( <xsl:value-of select = "@pages"/> pages )
73              </td>
74            </tr>
75          </xsl:for-each>
76        </table>
```

```
77
78          <p>Pages:
79             <xsl:variable name = "pagecount"
80                select = "sum(chapters//*/@pages)"/>
81             <xsl:value-of select = "$pagecount"/>
82          <p>Media Type: <xsl:value-of select = "media/@type"/></p>
83       </body>
84    </xsl:template>
85 </xsl:stylesheet>
```

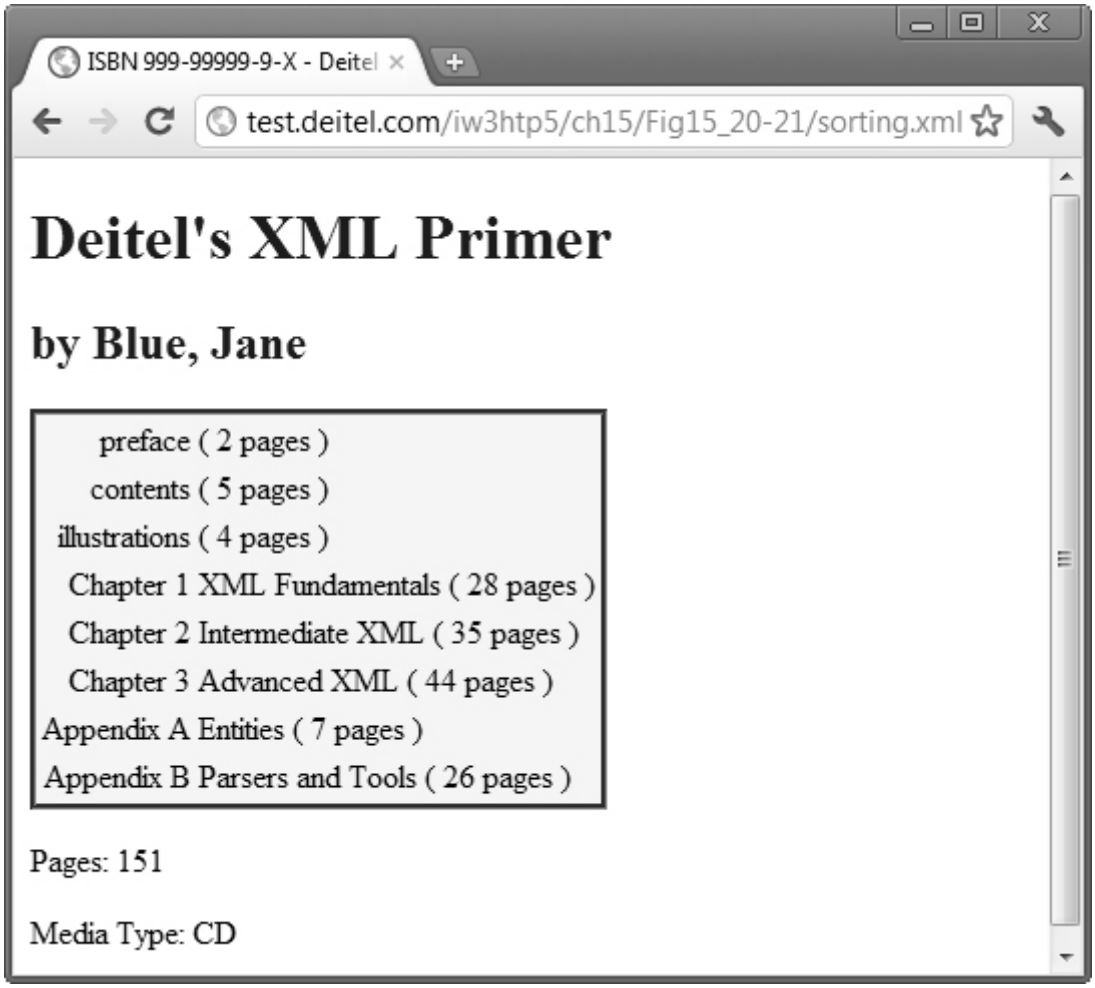


Fig. 15.21. XSL document that transforms `sorting.xml` into HTML5.

Lines 19–84 specify a template that matches element `book`. The template indicates how to format the information contained in `book` elements of `sorting.xml` (Fig. 15.20) as HTML5.

Lines 23–24 create the title for the HTML5 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 29–31 create a header element that contains the book's author. Because the context node (i.e., the current node being processed) is `book`, the XPath expression `author/lastName` selects the author's last name, and the expression `author/firstName` selects the author's first name.

Line 35 selects each element (indicated by an asterisk) that's a child of element `front-Matter`. Line 38 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 35). Line 42 retrieves the value of the `pages` attribute of the current node.

Line 47 selects each `chapter` element. Lines 48–49 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 63), and attribute `order` also accepts the value `"descending"`. Line 56 uses **node-set function** **`text`** to obtain the text between the `chapter` start and end tags (i.e., the name of the chapter). Line 57 retrieves the value of the `pages` attribute of the current node. Lines 62–75 perform similar tasks for each appendix.

Lines 79–80 use an **XSL variable** to store the value of the book's total page count and output the page count to the result tree. Attribute **`name`** specifies the variable's name (i.e., `pagecount`), and attribute `select` assigns a value to the variable. Function **`sum`** (line 80) totals the values for all `page` attribute values. The two slashes between `chapters` and `*` indicate a **recursive descent**—the MSXML 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 81 retrieves the value of the newly created XSL variable `pagecount` by placing a dollar sign in front of its name.

Summary of XSL Stylesheet Elements

This section's examples used several predefined XSL elements to perform various operations. Figure 15.22 lists these and several other commonly used XSL elements. For more information on these elements and XSL in general, see www.w3.org/Style/XSL.

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

Fig. 15.22. XSL style-sheet elements.

15.9. Document Object Model (DOM)

Although an XML document is a text file, retrieving data from the document using traditional sequential file-processing techniques is neither practical nor efficient, especially for adding and removing elements dynamically.

Upon successfully parsing a document, some XML parsers store document data as tree structures in memory. Figure 15.23 illustrates the tree structure for the root element of the document `article.xml` (Fig. 15.2). This hierarchical tree structure is called a **Document Object Model (DOM) tree**, and an XML parser that creates this type of structure is known as a **DOM parser**. 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. Many of the XML DOM capabilities you'll see in this section are similar or identical to those of the HTML5 DOM you learned in Chapter 12.

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. 15.23) contains a node for the XML declaration (line 1), two nodes for the comments (lines 3–4) and a node for the XML document's root element `article` (line 5).

To introduce document manipulation with the XML Document Object Model, we provide a scripting example (Figs. 15.24–15.25) that uses JavaScript and XML. This example loads the XML document `article.xml` (Fig. 15.2) and uses the XML DOM API to display the document's element names and values. The example also provides buttons that enable you to navigate the DOM structure. As you click each button, an appropriate part of the document is highlighted.

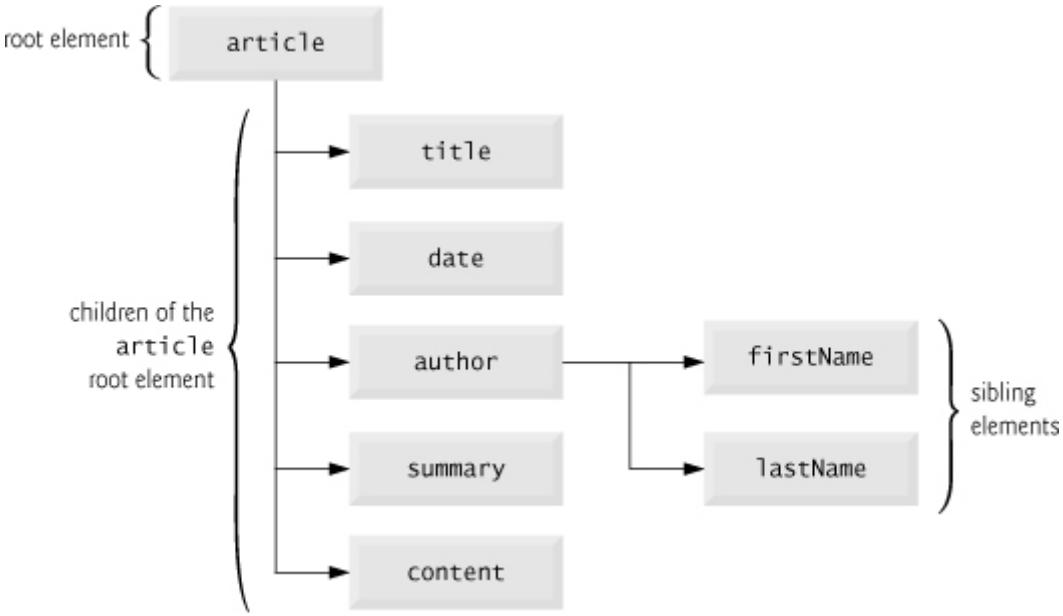


Fig. 15.23. Tree structure for the document `article.xml` of Fig. 15.2.

HTML5 Document

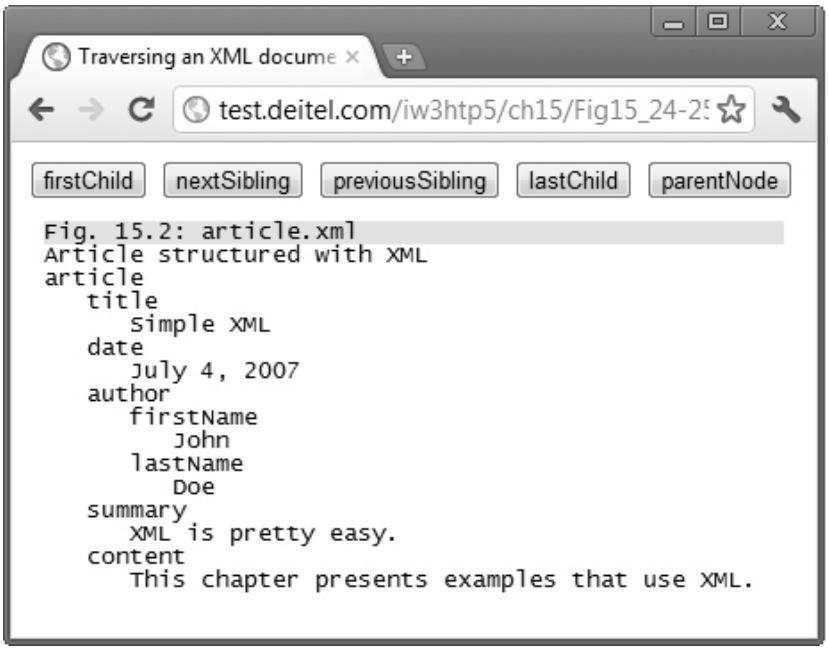
Figure 15.24 contains the HTML5 document. When this document loads, the `load` event calls our JavaScript function `start` (Fig. 15.25) to register event handlers for the buttons in the document and to load and display the contents of `article.xml` in the `div` at line 21 (`outputDiv`). Lines 13–20 define a form consisting of five buttons. When each button is pressed, it invokes one of our JavaScript functions to navigate `article.xml` 's DOM structure. (To save space, we do not show the contents of the example's CSS file here.) Some browsers allow you to load XML documents dynamically only when accessing the files from a web server. For this reason, you can test this example at:

http://test.deitel.com/iw3htp5/ch15/Fig15_24-25/XMLDOMTraversal.xml

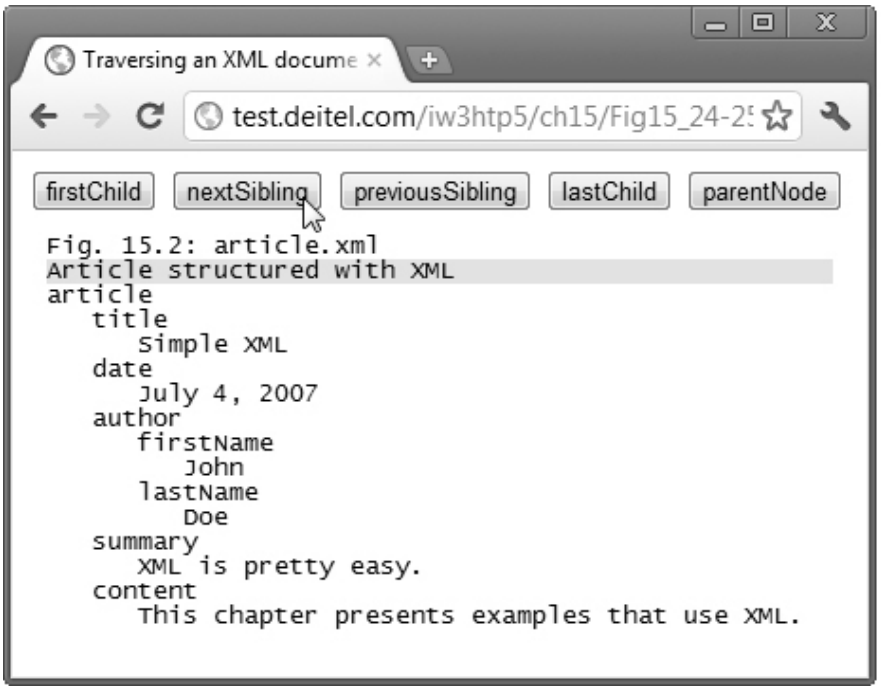
---

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 15.24: XMLDOMTraversal.html -->
4  <!-- Traversing an XML document using the XML DOM. -->
5  <html>
6  <head>
7     <meta charset = "utf-8">
8     <link rel = "stylesheet" type = "text/css" href = "style.css">
```

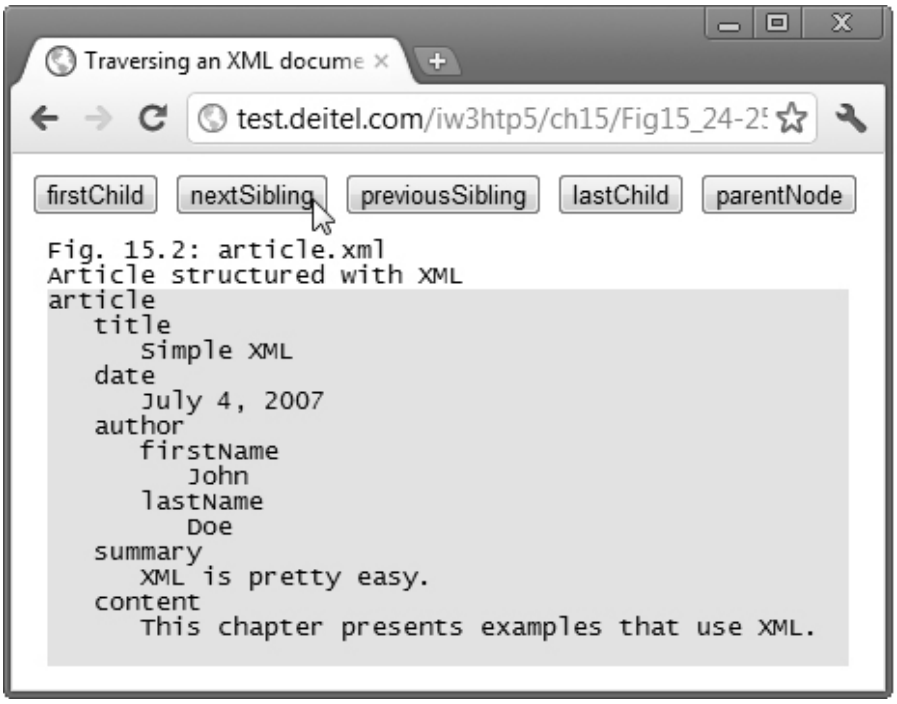
```
 9    <script src = "XMLDOMTraversal.js"></script>
10    <title>Traversing an XML document using the XML DOM</title>
11 </head>
12 <body id = "body">
13    <form action = "#">
14       <input id = "firstChild" type = "button" value = "firstChild">
15       <input id = "nextSibling" type = "button" value = "nextSibling">
16       <input id = "previousSibling" type = "button"
17          value = "previousSibling">
18       <input id = "lastChild" type = "button" value = "lastChild">
19       <input id = "parentNode" type = "button" value = "parentNode">
20    </form>
21    <div id = "outputDiv"></div>
22 </body>
23 </html>
```



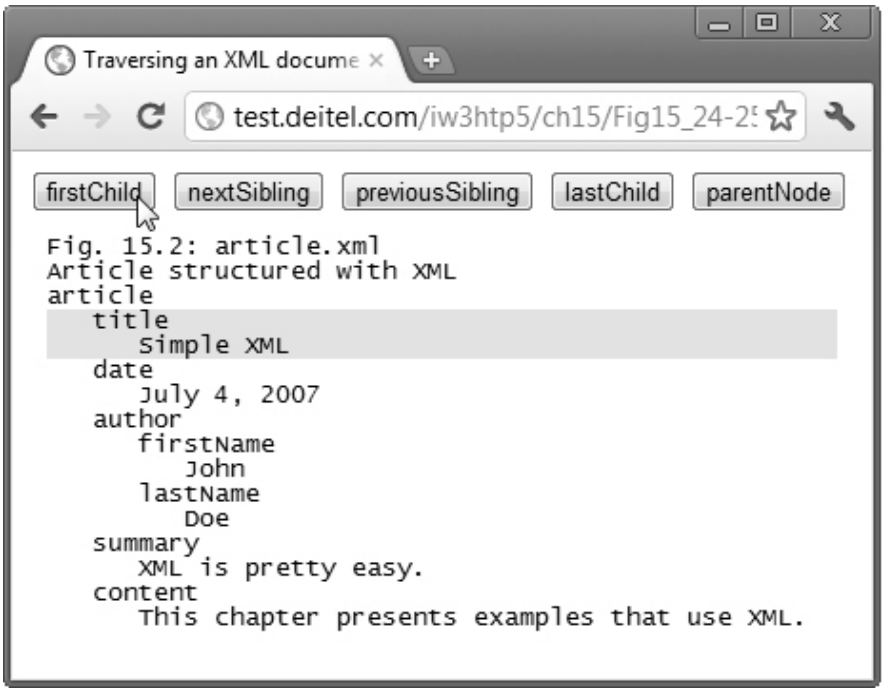
a) Comment node at the beginning of `article.xml` is highlighted when the XML document first loads



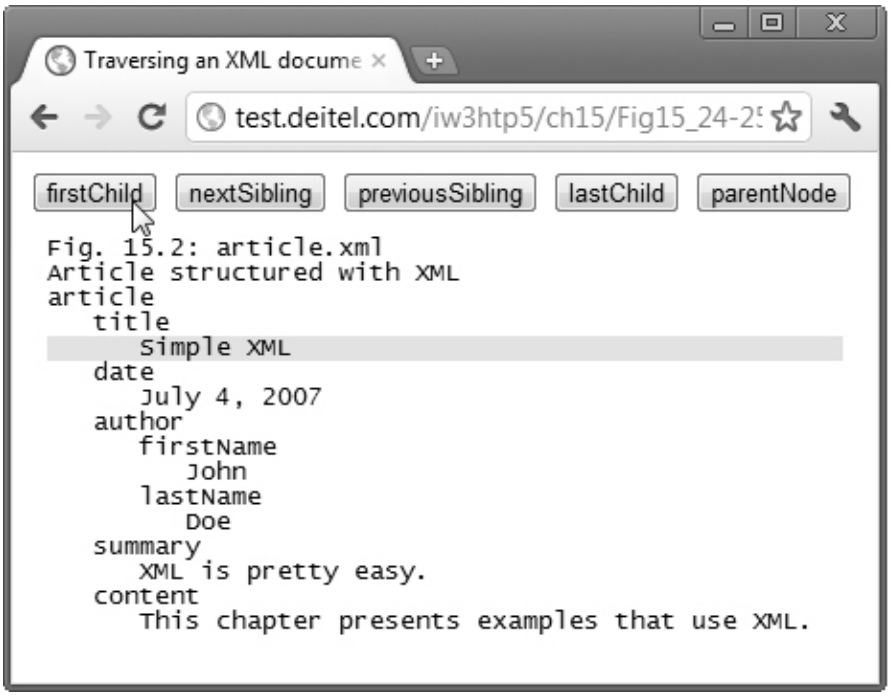
b) User clicked the **nextSibling** button to highlight the second comment node



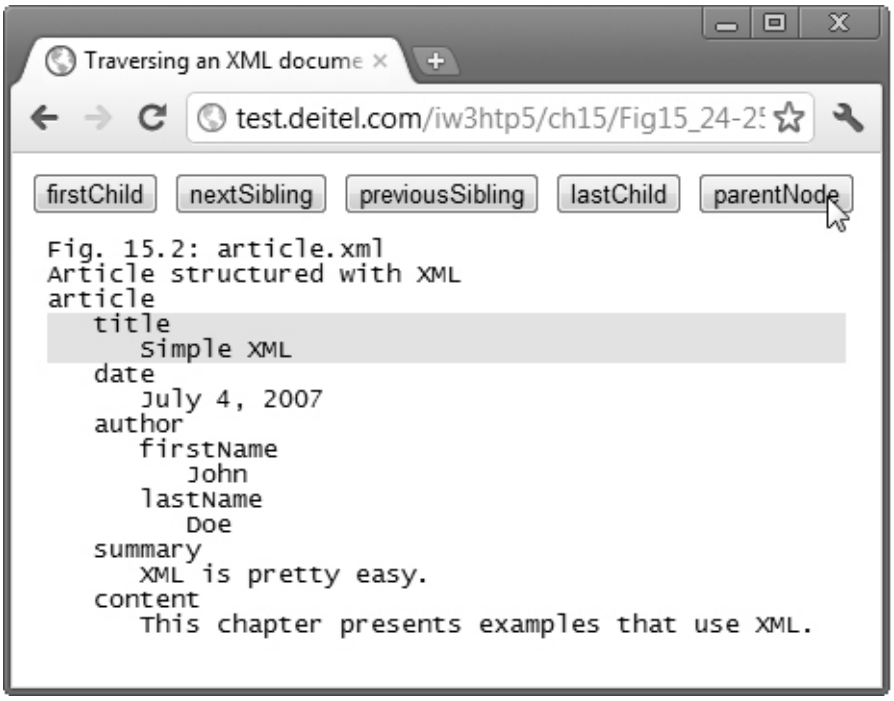
c) User clicked the **nextSibling** button again to highlight the `article` node



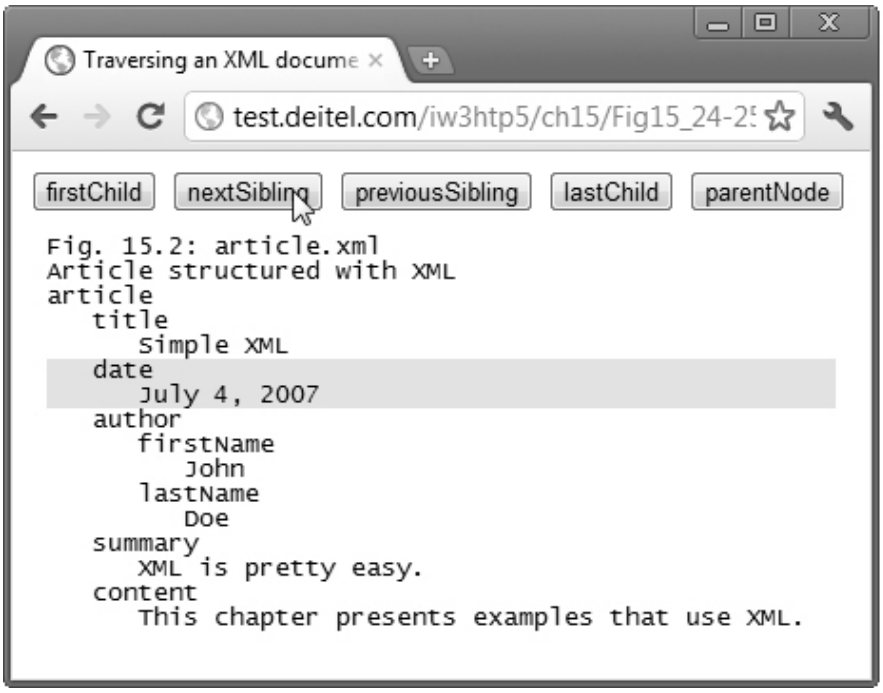
d) User clicked the **firstChild** button to highlight the `article` node's `title` child node



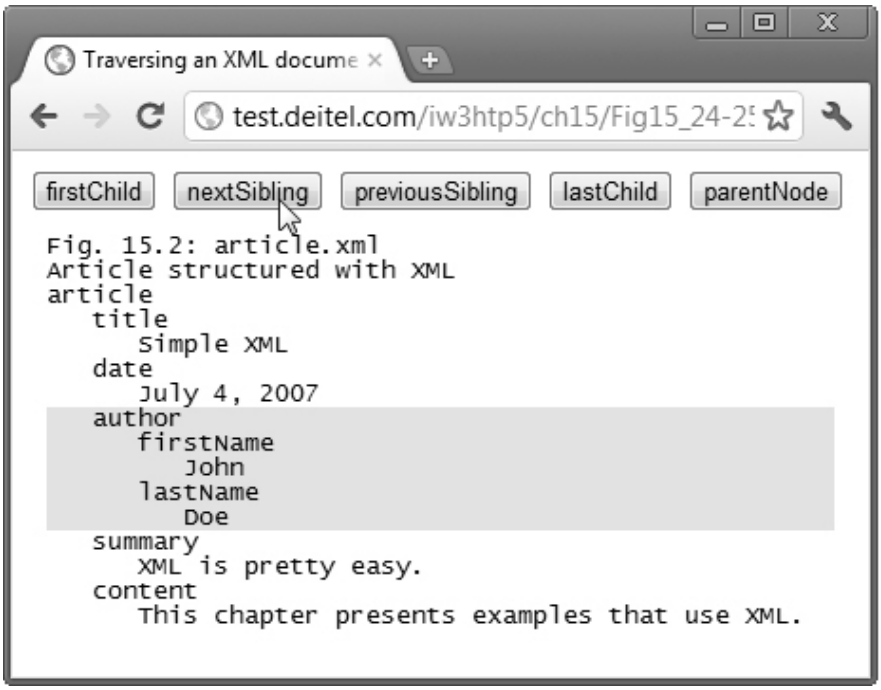
e) User clicked the **firstChild** button again to highlight the `title` node's text child node



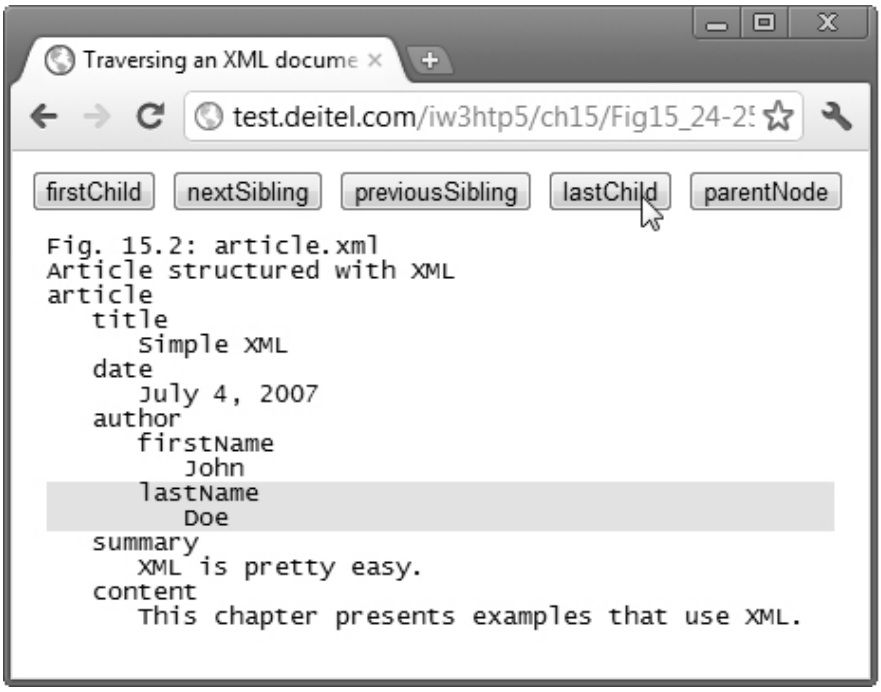
f) User clicked the **parentNode** button to highlight the text node's parent `title` node



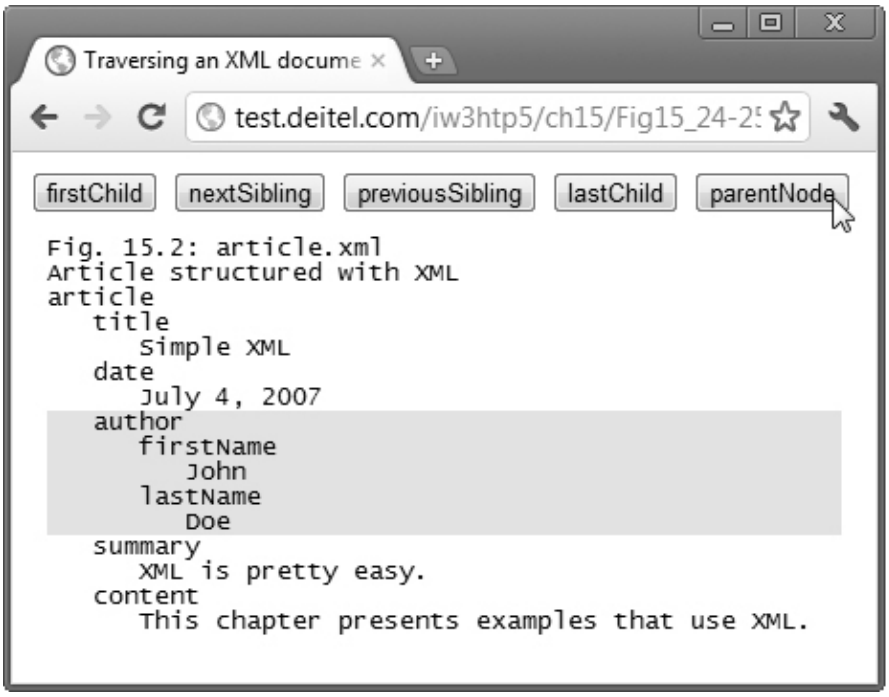
g) User clicked the **nextSibling** button to highlight the `title` node's `date` sibling node



h) User clicked the **nextSibling** button to highlight the `date` node's `author` sibling node



i) User clicked the **lastChild** button to highlight the `author` node's last child node (`lastName`)



j) User clicked the **parentNode** button to highlight the `lastName` node's `author` parent node

Fig. 15.24. Traversing an XML document using the XML DOM.

JavaScript Code

Figure 15.24 lists the JavaScript code that manipulates this XML document and displays its content in an HTML5 page. Line 187 indicates that the document's `load` event handler should call the script's `start` function.

```
1  <!-- Fig. 15.25: XMLDOMTraversal.html -->
2  <!-- JavaScript for traversing an XML document using the XML DOM. -->
3  var outputHTML = ""; // stores text to output in outputDiv
4  var idCounter = 1; // used to create div IDs
5  var depth = -1; // tree depth is -1 to start
6  var current = null; // represents the current node for traversals
7  var previous = null; // represents prior node in traversals
8
9  // register event handlers for buttons and load XML document
10 function start()
11 {
12    document.getElementById( "firstChild" ).addEventListener(
```

```
13      "click", processFirstChild, false );
14    document.getElementById( "nextSibling" ).addEventListener(
15      "click", processNextSibling, false );
16    document.getElementById( "previousSibling" ).addEventListener(
17      "click", processPreviousSibling, false );
18    document.getElementById( "lastChild" ).addEventListener(
19      "click", processLastChild, false );
20    document.getElementById( "parentNode" ).addEventListener(
21      "click", processParentNode, false );
22    loadXMLDocument( 'article.xml' )
23  } // end function start
24
25  // load XML document based on whether the browser is IE7 or Firefox
2
26  function loadXMLDocument( url )
27  {
28    var xmlHttpRequest = new XMLHttpRequest();
29    xmlHttpRequest.open( "get", url, false );
30    xmlHttpRequest.send( null );
31    var doc = xmlHttpRequest.responseXML;
32    buildHTML( doc.childNodes ); // display the nodes
33    displayDoc(); // display the document and highlight current node
34  } // end function loadXMLDocument
35
36  // traverse xmlDocument and build HTML5 representation of its con-
tent
37  function buildHTML( childList )
38  {
39    ++depth; // increase tab depth
40
41    // display each node's content
42    for ( var i = 0; i < childList.length; i++ )
43    {
44      switch ( childList[ i ].nodeType )
45      {
46        case 1: // Node.ELEMENT_NODE; value used for portability
```

```
47          outputHTML += "<div id=\"id" + idCounter + "\">";
48          spaceOutput( depth ); // insert spaces
49          outputHTML += childList[ i ].nodeName; // show node's name
50          ++idCounter; // increment the id counter
51
52          // if current node has children, call buildHTML recursively
53          if ( childList[ i ].childNodes.length != 0 )
54             buildHTML( childList[ i ].childNodes );
55
56          outputHTML += "</div>";
57          break;
58       case 3: // Node.TEXT_NODE; value used for portability
59       case 8: // Node.COMMENT_NODE; value used for portability
60          // if nodeValue is not 3 or 6 spaces (Firefox issue),
61          // include nodeValue in HTML
62          if ( childList[ i ].nodeValue.indexOf( "   " ) == -1 &&
63             childList[ i ].nodeValue.indexOf( "      " ) == -1 )
64          {
65             outputHTML += "<div id=\"id" + idCounter + "\">";
66             spaceOutput( depth ); // insert spaces
67             outputHTML += childList[ i ].nodeValue + "</div>";
68             ++idCounter; // increment the id counter
69          } // end if
70       } // end switch
71    } // end for
72
73    --depth; // decrease tab depth
74 } // end function buildHTML
75
76 // display the XML document and highlight the first child
77 function displayDoc()
78 {
79    document.getElementById( "outputDiv" ).innerHTML = out-
putHTML;
80    current = document.getElementById( 'id1' );
81    setCurrentNodeStyle( current.getAttribute( "id" ), true );
```

```
82  } // end function displayDoc
83
84  // insert nonbreaking spaces for indentation
85  function spaceOutput( number )
86  {
87     for ( var i = 0; i < number; i++ )
88     {
89        outputHTML += "   ";
90     } // end for
91  } // end function spaceOutput
92
93  // highlight first child of current node
94  function processFirstChild()
95  {
96     if ( current.childNodes.length == 1 && // only one child
97        current.firstChild.nodeType == 3 )  // and it's a text node
98     {
99        alert( "There is no child node" );
100    } // end if
101    else if ( current.childNodes.length > 1 )
102    {
103       previous = current; // save currently highlighted node
104
105       if ( current.firstChild.nodeType != 3 ) // if not text node
106          current = current.firstChild; // get new current node
107       else // if text node, use firstChild's nextSibling instead
108          current = current.firstChild.nextSibling; // get first sibling
109
110       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
111       setCurrentNodeStyle( current.getAttribute( "id" ), true );
112    } // end if
113    else
114       alert( "There is no child node" );
115 } // end function processFirstChild
116
117 // highlight next sibling of current node
```

```
118 function processNextSibling()
119 {
120    if ( current.getAttribute( "id" ) != "outputDiv" &&
121       current.nextSibling )
122    {
123       previous = current; // save currently highlighted node
124       current = current.nextSibling; // get new current node
125       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
126       setCurrentNodeStyle( current.getAttribute( "id" ), true );
127    } // end if
128    else
129       alert( "There is no next sibling" );
130 } // end function processNextSibling
131
132 // highlight previous sibling of current node if it is not a text node
133 function processPreviousSibling()
134 {
135    if ( current.getAttribute( "id" ) != "outputDiv" &&
136       current.previousSibling && current.previousSibling.nodeType !=
3 )
137    {
138       previous = current; // save currently highlighted node
139       current = current.previousSibling; // get new current node
140       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
141       setCurrentNodeStyle( current.getAttribute( "id" ), true );
142    } // end if
143    else
144       alert( "There is no previous sibling" );
145 } // end function processPreviousSibling
146
147 // highlight last child of current node
148 function processLastChild()
149 {
150    if ( current.childNodes.length == 1 &&
151       current.lastChild.nodeType == 3 )
152    {
```

```
153       alert( "There is no child node" );
154    } // end if
155    else if ( current.childNodes.length != 0 )
156    {
157       previous = current; // save currently highlighted node
158       current = current.lastChild; // get new current node
159       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
160       setCurrentNodeStyle( current.getAttribute( "id" ), true );
161    } // end if
162    else
163       alert( "There is no child node" );
164 } // end function processLastChild
165
166 // highlight parent of current node
167 function processParentNode()
168 {
169    if ( current.parentNode.getAttribute( "id" ) != "body" )
170    {
171       previous = current; // save currently highlighted node
172       current = current.parentNode; // get new current node
173       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
174       setCurrentNodeStyle( current.getAttribute( "id" ), true );
175    } // end if
176    else
177       alert( "There is no parent node" );
178 } // end function processParentNode
179
180 // set style of node with specified id
181 function setCurrentNodeStyle( id, highlight )
182 {
183    document.getElementById( id ).className =
184       ( highlight ? "highlighted" : "" );
185 } // end function setCurrentNodeStyle
186
187 window.addEventListener( "load", start, false );
```

Fig. 15.25. JavaScript for traversing an XML document using the XML DOM.

Global Script Variables

Lines 3–7 declare several variables used throughout the script. Variable `outputHTML` stores the markup that will be placed in `outputDiv`. Variable `idCounter` is used to track the unique `id` attributes that we assign to each element in the `outputHTML` markup. These `id`s will be used to dynamically highlight parts of the document when the user clicks the buttons in the form. Variable `depth` determines the indentation level for the content in `article.xml`. We use this to structure the output using the nesting of the elements in `article.xml`. Variables `current` and `previous` track the current and previous nodes in `article.xml`'s DOM structure as the user navigates it.

Function `start`

Function `start` (lines 10–23) registers event handlers for each of the buttons in Fig. 15.24, then calls function `loadXMLDocument`.

Function `loadXMLDocument`

Function `loadXMLDocument` (lines 26–35) loads the XML document at the specified URL. Line 28 creates an **`XMLHttpRequest` object**, which can be used to load an XML document. Typically, such an object is used with Ajax to make asynchronous requests to a server—the topic of the next chapter. Here, we need to load an XML document immediately for use in this example. Line 29 uses the `XMLHttpRequest` object's **`open` method** to create a `get` request for an XML document at a specified URL. When the last argument's value is `false`, the request will be made synchronously—that is, the script will not continue until the document is received. Next, line 30 executes the `XMLHttpRequest`, which actually loads the XML document. The argument `null` to the **`send` method** indicates that no data is being sent to the server as part of this request. When the request completes, the resulting XML document is stored in the `XMLHttpRequest` object's `responseXML` property, which we assign to local variable `doc`.

When this completes, we call our `buildHTML` method (defined in lines 37–74) to construct an HTML5 representation of the XML document. The expression `doc.childNodes` is a list of the XML document's top-level nodes. Line 33 calls our `displayDoc` function (lines 77–82) to display the contents of `article.xml` in `outputDiv`.

Function `buildHTML`

Function `buildHTML` (lines 37–74) is a recursive function that receives a list of nodes as an argument. Line 39 increments the `depth` for indentation purposes. Lines 42–71 iterate through the nodes in the list. The `switch` statement (lines 44–70) uses the current node's **`nodeType` property** to determine whether the current node is an element (line 46), a text node (i.e., the text content of an element; line 58) or a comment node (line 59). If it's an element, then we begin a new `div` element in our HTML5 (line 47) and give it a unique `id`. Then function `spaceOutput` (defined in lines 85–91) appends **nonbreaking spaces (` `)**—i.e., spaces that the browser is not allowed to collapse or that can be used to keep words together—to indent the current element to the correct level. Line 49 appends the name of the current element using the node's **`nodeName` property**. If the current element has children, the length of the current node's `childNodes` list is nonzero and line 54 recursively calls `buildHTML` to append the current element's child nodes to the markup. When that recursive call completes, line 56 completes the `div` element that we started at line 47.

If the current element is a text node, lines 62–63 obtain the node's value with the **`nodeValue` property** and use the string method `indexOf` to determine whether the node's value starts with three or six spaces. Some XML parsers do not ignore the white space used for indentation in XML documents. Instead they create text nodes containing just the space characters. The condition in lines 62–63 enables us to ignore these nodes in such browsers. If the node contains text, lines 65–67 append a new `div` to the markup and use the `node`'s `nodeValue` property to insert that text in the `div`. Line 73 in `buildHTML` decrements the `depth` counter.

PORTABILITY TIP 15.4

Firefox's XML parser does not ignore white space used for indentation in XML documents. Instead, it creates text nodes containing the white-space characters.

Function `displayDoc`

In function `displayDoc` (lines 77–82), line 79 uses the DOM's `getElementById` method to obtain the `outputDiv` element and set its `innerHTML` property to the new markup generated by `buildHTML` . Then, line 80 sets variable `current` to refer to the `div` with `id 'id1'` in the new markup, and line 81 uses our `setCurrentNodeStyle` method (defined at lines 181–185) to highlight that `div` .

Functions `processFirstChild` **and** `processLastChild`

Function `processFirstChild` (lines 94–115) is invoked by the `onclick` event of the `firstChild` button. If the `current` node has only one child and it's a text node (lines 96–97), line 99 displays an alert dialog indicating that there's no child node—we navigate only to nested XML elements in this example. If there are two or more children, line 103 stores the value of `current` in `previous` , and lines 105–108 set `current` to refer to its **`firstChild`** (if this child is not a text node) or its `firstChild` 's **`nextSibling`** (if the `firstChild` is a text node)—again, this is to ensure that we navigate only to nodes that represent XML elements. Then lines 110–111 unhighlight the `previous` node and highlight the new `current` node. Function `processLastChild` (lines 148–164) works similarly, using the `current` node's **`lastChild`** property.

Functions `processNextSibling` **and** `processPreviousSibling`

Function `processNextSibling` (lines 118–130) first ensures that the current node is not the `outputDiv` and that `nextSibling` exists. If so, lines 123–124 adjust the `previous` and `current` nodes accordingly and up-

date their highlighting. Function `processPreviousSibling` (lines 133–145) works similarly, ensuring first that the current node is not the `outputDiv`, that `previousSibling` exists and that `previousSibling` is not a text node.

Function `processParentNode`

Function `processParentNode` (lines 167–178) first checks whether the current node's **`parentNode`** is the HTML5 page's `body`. If not, lines 171–174 adjust the `previous` and `current` nodes accordingly and update their highlighting.

Common DOM Properties

The tables in Figs. 15.26–15.31 describe many common DOM properties and methods. Some of the key DOM objects are **`Node`** (a node in the tree), **`NodeList`** (an ordered set of `Node`s), **`Document`** (the document), **`Element`** (an element node), **`Attr`** (an attribute node) and **`Text`** (a text node). There are many more objects, properties and methods than we can possibly list here. Our XML Resource Center (www.deitel.com/XML/) includes links to various DOM reference websites.

Property/Method	Description
`nodeType`	An integer representing the node type.
`nodeName`	The name of the node.
`nodeValue`	A string or null depending on the node type.
`parentNode`	The parent node.
`childNodes`	A `NodeList` (Fig. 15.27) with all the children of the node.
`firstChild`	The first child in the `Node`'s `NodeList`.
`lastChild`	The last child in the `Node`'s `NodeList`.
`previousSibling`	The node preceding this node; `null` if there's no such node.
`nextSibling`	The node following this node; `null` if there's no such node.
`attributes`	A collection of `Attr` objects (Fig. 15.30) containing the attributes for this node.
`insertBefore`	Inserts the node (passed as the first argument) before the existing node (passed as the second argument). If the new node is already in the tree, it's removed before insertion. The same behavior is true for other methods that add nodes.
`replaceChild`	Replaces the second argument node with the first argument node.
`removeChild`	Removes the child node passed to it.
`appendChild`	Appends the node it receives to the list of child nodes.

Fig. 15.26. Common `Node` properties and methods.

Property/Method	Description
`item`	Method that receives an index number and returns the element node at that index. Indices range from 0 to *length* – 1. You can also access the nodes in a `NodeList` via array indexing.
`length`	The total number of nodes in the list.

Fig. 15.27. `NodeList` property and method.

Property/Method	Description
`documentElement`	The root node of the document.
`createElement`	Creates and returns an element node with the specified tag name.
`createAttribute`	Creates and returns an `Attr` node (Fig. 15.30) with the specified name and value.
`createTextNode`	Creates and returns a text node that contains the specified text.
`getElementsByTagName`	Returns a `NodeList` of all the nodes in the subtree with the name specified as the first argument, ordered as they would be encountered in a preorder traversal. An optional second argument specifies either the direct child nodes (0) or any descendant (1).

Fig. 15.28. `Document` property and methods.

Property/Method	Description
`tagName`	The name of the element.
`getAttribute`	Returns the value of the specified attribute.
`setAttribute`	Change

Fig. 15.29. `Element` property and methods.

Property	Description
`value`	The specified attribute's value.
`name`	The name of the attribute.

Fig. 15.30. `Attr` properties.

Property	Description
`data`	The text contained in the node.
`length`	The number of characters contained in the node.

Fig. 15.31. `Text` properties.

Locating Data in XML Documents with XPath

Although you can use XML DOM capabilities to navigate through and manipulate nodes, this is not the most efficient means of locating data in an XML document's DOM tree. A simpler way to locate nodes is to search for lists of nodes matching search criteria that are written as XPath expressions. Recall that XPath (XML Path Language) provides a syntax for locating specific nodes in XML documents effectively and efficiently. XPath is a string-based language of expressions used by XML and many of its related technologies (such as XSLT, discussed in Section 15.8).

The example of Figs. 15.32–15.34 enables the user to enter XPath expressions in an HTML5 form. (To save space, we do not show the contents of the example's CSS file here.) When the user clicks the **Get Matches** button, the script applies the XPath expression to the XML DOM and displays the matching nodes.

HTML5 Document

When the HTML5 document (Fig. 15.32) loads, its `load` event calls `loadDocument` (as specified in Fig. 15.33, line 61) to load the `sports.xml` file (Fig. 15.34). The user specifies the XPath expression in the `input` element at line 14 (of Fig. 15.32). When the user clicks the **Get Matches** button (line 15), its `click` event handler invokes our `processXPathExpression` function (Fig. 15.33) to locate any matches and display the results in `outputDiv` (Fig. 15.32, line 17). Some browsers allow you to load XML documents dynamically only when accessing the files from a web server. For this reason, you can test this example at:

http://test.deitel.com/iw3htp5/ch15/Fig15_24-25/XMLDOMTraversal.xml

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 15.32: xpath.html -->
4  <!-- Using XPath to locate nodes in an XML document. -->
5  <html>
6  <head>
7     <meta charset = "utf-8">
8     <link rel = "stylesheet" type = "text/css" href = "style.css">
9     <script src = "xpath.js"></script>
10     <title>Using XPath</title>
11  </head>
12  <body id = "body">
13     <form id = "myForm" action = "#">
14        <input id = "inputField" type = "text">
15        <input id = "matchesButton" type = "button" value = "Get Matches">
16     </form>
17     <div id = "outputDiv"></div>
18  </body>
19  </html>
```

Fig. 15.32. Using XPath to locate nodes in an XML document.

JavaScript

The script of Fig. 15.33 loads the XML document `sports.xml` (Fig. 15.34) using the same techniques we presented in Fig. 15.25, so we focus on only the new features in this example.

```
1  // Fig. 15.33: xpath.html
2  // JavaScript that uses XPath to locate nodes in an XML document.
3  var doc; // variable to reference the XML document
4  var outputHTML = ""; // stores text to output in outputDiv
5
6  // register event handler for button and load XML document
7  function start()
8  {
9     document.getElementById( "matchesButton" ).addEventListener(
10       "click", processXPathExpression, false );
11    loadXMLDocument( "sports.xml" );
12 } // end function start
13
14 // load XML document programmatically
15 function loadXMLDocument( url )
16 {
17    var xmlHttpRequest = new XMLHttpRequest();
18    xmlHttpRequest.open( "get", url, false );
```

```
19    xmlHttpRequest.send( null );
20    doc = xmlHttpRequest.responseXML;
21  } // end function loadXMLDocument
22
23  // display the XML document
24  function displayHTML()
25  {
26    document.getElementById( "outputDiv" ).innerHTML = out-
putHTML;
27  } // end function displayDoc
28
29  // obtain and apply XPath expression
30  function processXPathExpression()
31  {
32    var xpathExpression = document.getElementById( "inputField"
).value;
33    var result;
34    outputHTML = "";
35
36    if ( !doc.evaluate ) // Internet Explorer
37    {
38      result = doc.selectNodes( xpathExpression );
39
40      for ( var i = 0; i < result.length; i++ )
41      {
42        outputHTML += "<p>" + result.item( i ).text + "</p>";
43      } // end for
44    } // end if
45    else // other browsers
46    {
47      result = doc.evaluate( xpathExpression, doc, null,
48        XPathResult.ORDERED_NODE_ITERATOR_TYPE, null );
49      var current = result.iterateNext();
50
51      while ( current )
52      {
```

```
53          outputHTML += "<p>" + current.textContent + "</p>";
54          current = result.iterateNext();
55       } // end while
56    } // end else
57
58    displayHTML();
59 } // end function processXPathExpression
60
61 window.addEventListener( "load", start, false );
```

Fig. 15.33. Using XPath to locate nodes in an XML document.

Function `processXPathExpression`

Function `processXPathExpression` (Fig. 15.33, lines 30–59) obtains the XPath expression (line 32) from the `inputField`. Internet Explorer and other browsers handle XPath processing differently, so this function contains an `if`...`else` statement to handle the differences.

Lines 36–44 apply the XPath expression in Internet Explorer (or any other browser that does not support to `evaluate` method on an XML document object), and lines 45–56 apply the XPath expression in all other browsers. In IE, the XML document object's **`selectNodes` method** (line 38) receives an XPath expression as an argument and returns a collection of elements that match the expression. Lines 40–43 iterate through the results and mark up each one in a separate `p` element. After this loop completes, line 58 displays the generated markup in `outputDiv`.

For other browsers, lines 47–48 invoke the XML document object's **`evaluate` method**, which receives five arguments—the XPath expression, the document to apply the expression to, a namespace resolver, a result type and an `XPathResult` object into which to place the results. The result type `XPathResult.ORDERED_NODE_ITERATOR_TYPE` indicates that the method should return an object that can be used to iterate through the results in the order they appeared in the XML document. If the last argument is `null`, the function simply returns a new **`XPathResult` object** containing the matches. The namespace resolver argument can be `null`

if you're not using XML namespace prefixes in the XPath processing. Lines 47–55 iterate through the `XPathResult` and mark up the results. Line 49 invokes the `XPathResult`'s `iterateNext` method to position to the first result. If there's a result, the condition in line 51 will be true, and line 53 creates a `p` element for that result. Line 54 then positions to the next result. After this loop completes, line 58 displays the generated markup in `outputDiv`.

`sports.xml`

Figure 15.34 shows the XML document `sports.xml` that we use in this example. [*Note:* The versions of `sports.xml` presented in Fig. 15.34 and Fig. 15.18 are nearly identical. In the current example, we do not want to apply an XSLT, so we omit the processing instruction found in line 2 of Fig. 15.18. We also removed extra blank lines to save space.]

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.34: sports.xml -->
4  <!-- Sports Database       -->
5  <sports>
6    <game id = "783">
7      <name>Cricket</name>
8      <paragraph>
9        More popular among commonwealth nations.
10     </paragraph>
11   </game>
12   <game id = "239">
13     <name>Baseball</name>
14     <paragraph>
15       More popular in America.
16     </paragraph>
17   </game>
18   <game id = "418">
19     <name>Soccer (Futbol)</name>
20     <paragraph>
```

```
21         Most popular sport in the world.
22       </paragraph>
23     </game>
24   </sports>
```

Fig. 15.34. XML document that describes various sports.

Function `processXPathExpression`

Figure 15.35 summarizes the XPath expressions that we demonstrated in Fig. 15.32's sample outputs.

Fig. 15.35. `XPath` expressions and descriptions.

15.10. Web Resources

www.deitel.com/XML/

The Deitel XML Resource Center focuses on the vast amount of free XML content available online, plus some for-sale items. Start your search here for tools, downloads, tutorials, podcasts, wikis, documentation, conferences, FAQs, books, e-books, sample chapters, articles, newsgroups, fo-

rums, downloads from CNET's download.com, jobs and contract opportunities, and more that will help you develop XML applications.

Summary

Section 15.1 Introduction

- The eXtensible Markup Language (XML; p. 512) is a portable, widely supported, open (i.e., nonproprietary) technology for data storage and exchange.

Section 15.2 XML Basics

- XML documents are readable by both humans and machines.
- XML permits document authors to create custom markup for any type of information. This enables document authors to create entirely new markup languages (p. 512) that describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes.
- An XML parser (p. 514) is responsible for identifying components of XML documents (typically files with the `.xml` extension) and then storing those components in a data structure for manipulation.
- An XML document can optionally reference a Document Type Definition (DTD, p. 514) or schema that defines the XML document's structure.
- An XML document that conforms to a DTD/schema (i.e., has the appropriate structure) is valid.
- If an XML parser (validating or non-validating; p. 514) can process an XML document successfully, that XML document is well-formed (p. 514).

Section 15.3 Structuring Data

- An XML document begins with an XML declaration (p. 515), which identifies the document as an XML document. The `version` attribute (p. 515) specifies the version of XML syntax used in the document.

- XML comments begin with `<!--` and end with `-->`.

- An XML document contains text that represents its content (i.e., data) and elements that specify its structure. XML documents delimit an element with start and end tags.

- The root element (p. 518) of an XML document encompasses all its other elements.

- XML element names can be of any length and can contain letters, digits, underscores, hyphens and periods. However, they must begin with either a letter or an underscore, and they should not begin with " `xml` " in any combination of uppercase and lowercase letters, as this is reserved for use in the XML standards.

- When a user loads an XML document in a browser, a parser parses the document, and the browser uses a style sheet to format the data for display.

- Data can be placed between tags or in attributes (name/value pairs that appear within the angle brackets of start tags, p. 520). Elements can have any number of attributes.

Section 15.4 XML Namespaces

- XML allows document authors to create their own markup, and as a result, naming collisions (i.e., two different elements that have the same name, p. 521) can occur. XML namespaces (p. 521) provide a means for document authors to prevent collisions.

- Each namespace prefix (p. 521) is bound to a Uniform Resource Identifier (URI, p. 522) that uniquely identifies the namespace. A URI is a series of characters that differentiate names. Document authors create their own namespace prefixes. Any name can be used as a namespace prefix, but the namespace prefix `xml` is reserved for use in XML standards.

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

declare a default namespace using keyword `xmlns` (p. 522) with a URI as its value.

- Document authors commonly use URLs (Uniform Resource Locators, p. 522) for URIs, because domain names (e.g., `deitel.com`) in URLs must be unique.

Section 15.5 Document Type Definitions (DTDs)

- DTDs and schemas specify documents' element types and attributes and their relationships to one another.
- DTDs and schemas enable an XML parser to verify whether an XML document is valid (i.e., its elements contain the proper attributes and appear in the proper sequence).
- A DTD expresses the set of rules for document structure using an EBNF (Extended Backus-Naur Form) grammar.
- In a DTD, an `ELEMENT` element type declaration (p. 525) defines the rules for an element. An `ATTLIST` attribute-list declaration (p. 525) defines attributes for a particular element.

Section 15.6 W3C XML Schema Documents

- XML schemas use XML syntax and are themselves XML documents.
- Unlike DTDs, XML Schema (p. 527) documents can specify what type of data (e.g., numeric, text) an element can contain.
- An XML document that conforms to a schema document is schema valid (p. 528).
- Two categories of types exist in XML Schema: simple types and complex types (p. 531). Simple types cannot contain attributes or child elements; complex types can.
- Every simple type defines a restriction on an XML Schema-defined schema type or on a user-defined type.

- Complex types can have either simple content or complex content. Both can contain attributes, but only complex content can contain child elements.

- Whereas complex types with simple content must extend or restrict some other existing type, complex types with complex content do not have this limitation.

Section 15.7 XML Vocabularies

- XML allows authors to create their own tags to describe data precisely.

- Some of these XML vocabularies include MathML (Mathematical Markup Language, p. 534), Scalable Vector Graphics (SVG, p. 534), Wireless Markup Language (WML, p. 534), Extensible Business Reporting Language (XBRL, p. 534), Extensible User Interface Language (XUL, p. 534), Product Data Markup Language (PDML, p. 534), W3C XML Schema and Extensible Stylesheet Language (XSL).

- MathML markup describes mathematical expressions for display. MathML is divided into two types of markup—content markup (p. 534) and presentation markup (p. 534).

- Content markup provides tags that embody mathematical concepts. Content MathML allows programmers to write mathematical notation specific to different areas of mathematics.

- Presentation MathML is directed toward formatting and displaying mathematical notation.

- By convention, MathML files end with the `.mml` filename extension.

- A MathML document's root node is the `math` element and its default namespace is http://www.w3.org/1998/Math/MathML.

- The `mn` element (p. 535) marks up a number. The `mo` element (p. 535) marks up an operator.

- Entity reference `&InvisibleTimes;` (p. 536) indicates a multiplication operation without explicit symbolic representation (p. 536).

- The `msup` element (p. 536) represents a superscript. It has two children—the expression to be superscripted (i.e., the base) and the superscript (i.e., the exponent). Correspondingly, the `msub` element (p. 536) represents a subscript.

- To display variables, use identifier element `mi` (p. 536).

- The `mfrac` element (p. 536) displays a fraction. If either the numerator or the denominator contains more than one element, it must appear in an `mrow` element (p. 537).

- An `mrow` element is used to group elements that are positioned horizontally in an expression.

- The entity reference `&int;` (p. 537) represents the integral symbol.

- The `msubsup` element (p. 537) specifies the subscript and superscript of a symbol. It requires three child elements—an operator, the subscript expression and the superscript expression.

- Element `msqrt` (p. 537) represents a square-root expression.

- Entity reference `&delta;` represents a lowercase delta symbol.

Section 15.8 Extensible Stylesheet Language and XSL Transformations

- eXtensible Stylesheet Language (XSL; p. 538) can convert XML into any text-based document. XSL documents have the extension `.xsl`.

- XPath (p. 538) 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.

- 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 (p. 539). When there are no more matches, XSLT has transformed the source tree (p. 539) into the result tree.

- The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using XPath'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.

- XSL style sheets (p. 539) can be connected directly to an XML document by adding an `xml:stylesheet` processing instruction to the XML document.

- Two tree structures are involved in transforming an XML document using XSLT—the source tree (the document being transformed) and the result tree (the result of the transformation).

- The XPath character `/` (a forward slash) always selects the document root. In XPath, a leading forward slash specifies that we're using absolute addressing.

- An XPath expression with no beginning forward slash uses relative addressing (p. 542).

- XSL element `value-of` retrieves an attribute's value. The `@` symbol specifies an attribute node.

- XSL node-set function `name` (p. 546) retrieves the current node's element name.

- XSL node-set function `text` (p. 546) retrieves the text between an element's start and end tags.

- The XPath expression `//*` selects all the nodes in an XML document.

Section 15.9 Document Object Model (DOM)

- Although an XML document is a text file, retrieving data from the document using traditional sequential file-processing techniques is neither practical nor efficient, especially for adding and removing elements dynamically.

- Upon successfully parsing a document, some XML parsers store document data as tree structures in memory. This hierarchical tree structure is called a Document Object Model (DOM) tree (p. 547), and an XML parser that creates this type of structure is known as a DOM parser (p. 547).

- Each element name is represented by a node. A node that contains other nodes is called a parent node. A parent node (p. 547) can have many children, but a child node (p. 547) can have only one parent node.

- Nodes that are peers are called sibling nodes (p. 547).

- A node's descendant nodes (p. 547) include its children, its children's children and so on. A node's ancestor nodes (p. 547) include its parent, its parent's parent and so on.

- Many of the XML DOM capabilities are similar or identical to those of the HTML5 DOM.

- The DOM tree has a single root node (p. 547), which contains all the other nodes in the document.

- An `XMLHttpRequest` object can be used to load an XML document.

- The `XMLHttpRequest` object's `open` method (p. 556) can be used to create a `get` request for an XML document at a specified URL. When the last argument's value is `false` , the request will be made synchronously.

- `XMLHttpRequest` method `send` (p. 556) executes the request to load the XML document. When the request completes, the resulting XML document is stored in the `XMLHttpRequest` object's `responseXML` property.

- A document's `childNodes` property contains a list of the XML document's top-level nodes.

- A node's `nodeType` property (p. 557) contains the type of the node.

- Nonbreaking spaces (` ` , p. 557) are spaces that the browser is not allowed to collapse or that can be used to keep words together.

- The name of an element can be obtained by the node's `nodeName` property (p. 557).

- If the currrent node has children, the length of the node's `childNodes` list is nonzero.

- The `nodeValue` property (p. 557) returns the value of an element.

- Node property `firstChild` (p. 557) refers to the first child of a given node. Similarly, `lastChild` (p. 558) refers to the last child of a given node.

- Node property `nextSibling` (p. 557) refers to the next sibling in a list of children of a particular node. Similarly, `previousSibling` refers to the current node's previous sibling.

- Property `parentNode` (p. 558) refers to the current node's parent node.

- A simpler way to locate nodes is to search for lists of node-matching search criteria that are written as XPath expressions.

- In IE, the XML document object's `selectNodes` method (p. 563) receives an XPath expression as an argument and returns a collection of elements that match the expression.

- Other browsers search for XPath matches using the XML document object's `evaluate` method (p. 563), which receives five arguments—the XPath expression, the document to apply the expression to, a namespace resolver, a result type and an `XPathResult` object (p. 563) into which to place the results. If the last argument is `null` , the function simply re-

turns a new `XPathResult` object containing the matches. The namespace resolver argument can be `null` if you're not using XML namespace prefixes in the XPath processing.

Self-Review Exercises

15.1 Which of the following are valid XML element names? (Select all that apply.)

a. `yearBorn`

b. `year.Born`

c. `year Born`

d. `year-Born1`

e. `2_year_born`

f. `_year_born_`

15.2 State which of the following statements are *true* and which are *false*. If *false*, explain why.

a. XML is a technology for creating markup languages.

b. XML markup is delimited by forward and backward slashes (`/` and `\`).

c. All XML start tags must have corresponding end tags.

d. Parsers check an XML document's syntax.

e. XML does not support namespaces.

f. When creating XML elements, document authors must use the set of XML tags provided by the W3C.

g. The pound character (`#`), dollar sign (`$`), ampersand (`&`) and angle brackets (`<` and `>`) are examples of XML reserved characters.

h. XML is not case sensitive.

i. XML Schemas are better than DTDs, because 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.

j. DTDs are written using an XML vocabulary.

k. Schema is a technology for locating information in an XML document.

15.3 Fill in the blanks for each of the following:

a. _____ help prevent naming collisions.

b. _____ embed application-specific information into an XML document.

c. _____ is Microsoft's XML parser.

d. XSL element _____ writes a `DOCTYPE` to the result tree.

e. XML Schema documents have root element _____.

f. XSL element _____ is the root element in an XSL document.

g. XSL element _____ selects specific XML elements using repetition.

h. Nodes that contain other nodes are called _____ nodes.

i. Nodes that are peers are called _____ nodes.

15.4 In Fig. 15.2, we subdivided the `author` element into more detailed pieces. How might you subdivide the `date` element? Use the date May 5, 2005, as an example.

15.5 Write a processing instruction that includes style sheet `wap.xsl`.

15.6 Write an XPath expression that locates `contact` nodes in `letter.xml` (Fig. 15.4).

Answers to Self-Review Exercises

15.1 a, b, d, f. [Choice c is incorrect because it contains a space. Choice e is incorrect because the first character is a number.]

15.2

a. True.

b. False. In an XML document, markup text is delimited by tags enclosed in angle brackets (`<` and `>`) with a forward slash just after the `<` in the end tag.

c. True.

d. True.

e. False. XML does support namespaces.

f. False. When creating tags, document authors can use any valid name but should avoid ones that begin with the reserved word `xml` (also `XML` , `Xml` , etc.).

g. False. XML reserved characters include the ampersand (`&`), the left angle bracket (`<`) and the right angle bracket (`>`), but not `#` and `$` .

h. False. XML is case sensitive.

i. True.

j. False. DTDs use EBNF grammar, which is not XML syntax.

k. False. XPath is a technology for locating information in an XML document. XML Schema provides a means for type checking XML documents and verifying their validity.

15.3

a. Namespaces.

b. Processing instructions.

c. MSXML.

d. `xsl:output`.

e. `schema`.

f. `xsl:stylesheet`.

g. `xsl:for-each`.

h. parent.

i. sibling.

15.4

```
<date>
   <month>May</month>
   <day>5</day>
   <year>2005</year>
</date>
```

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

15.6 `/letter/contact`.

Exercises

15.7 ***(Nutrition Information XML Document)*** Create an XML document that marks up the nutrition facts for a package of Grandma White's cookies. A package of cookies has a serving size of 1 package and the following nutritional value per serving: 260 calories, 100 fat calories, 11 grams of fat, 2 grams of saturated fat, 5 milligrams of cholesterol, 210 milligrams of sodium, 36 grams of total carbohydrates, 2 grams of fiber, 15 grams of sugars and 5 grams of protein. Name this document `nutrition.xml`. Load the XML document into your web browser. [*Hint:* Your markup should contain elements describing the product name, serving

size/amount, calories, sodium, cholesterol, proteins, etc. Mark up each nutrition fact/ingredient listed above.]

15.8 ***(Nutrition Information XML Schema)*** Write an XML Schema document (`nutrition.xsd`) specifying the structure of the XML document created in Exercise 15.7.

15.9 ***(Nutrition Information XSL Style Sheet)*** Write an XSL style sheet for your solution to Exercise 15.7 that displays the nutritional facts in an HTML5 table.

15.10 ***(Sorting XSLT Modification)*** Modify Fig. 15.21 (`sorting.xsl`) to sort by the number of pages rather than by chapter number. Save the modified document as `sorting_byPage.xsl` .