# 12

# XML and RSS

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

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

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

## OBJECTIVES

In this chapter you'll learn:

- To mark up data using XML.
- How XML namespaces help provide unique XML element and attribute names.
- To create DTDs and schemas for specifying and validating the structure of an XML document.
- To create and use simple XSL style sheets to render XML document data.
- To retrieve and manipulate XML data programmatically using JavaScript.
- RSS and how to programmatically apply an XSL transformation to an RSS document using JavaScript.

Outline



## 12.1 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 12.2–12.7 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. Sections 12.8–12.9 present additional XML technologies and key JavaScript capabilities for loading and manipulating XML programmatically—this material is optional but is recommended if you plan to use XML in your own applications. Finally, Section 12.10 introduces RSS—an XML format used to syndicate simple website content—and shows how to format RSS elements using JavaScript and other technologies presented in this chapter.

## 12.2 XML Basics

XML permits document authors to create ***markup*** (i.e., a text-based notation for describing data) for virtually any type of information. This enables document authors to create entirely new markup languages for describing any type of data, such as mathematical formulas, software-configuration instructions, chemical molecular structures, music, news, recipes and financial reports. XML describes data in a way that both human beings and computers can understand.

Figure 12.1 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 learnyou'll learn about the other elements of this document in Section 12.3.

XML documents contain text that represents content (i.e., data), such as `John` (line 6 of Fig. 12.1), and ***elements*** that specify the document's structure, such as `firstName` (line 6 of Fig. 12.1). 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

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.1: 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. 12.1** | XML that describes a baseball player's information.

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. 12.1, the root element is `player` (lines 5–9).

XML-based markup languages—called XML ***vocabularies***—provide a means for describing 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 12.7.

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 application 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. Section 12.3 demonstrates this in Internet Explorer and Firefox. An important characteristic of XML is that it is 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 12.3. 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.

Often, XML parsers are built into software or available for download over the Internet. Some popular parsers include ***Microsoft XML Core Services (MSXML)***—which is included with Internet Explorer, the Apache Software Foundation's ***Xerces*** (`xml.apache.org`) and the open-source ***Expat XML Parser*** (`expat.sourceforge.net`).

### *Validating XML Documents*

An XML document can reference a ***Document Type Definition (DTD)*** or a ***schema*** that defines the proper structure of the XML document. When an XML document references a DTD or a schema, some parsers (called ***validating parsers***) can read the DTD/schema and check that the XML document follows the structure defined by the DTD/schema. If the XML document conforms to the DTD/schema (i.e., the document has the appropriate structure), the XML document is ***valid***. For example, if in Fig. 12.1 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. 12.1) would invalidate the XML document `player.xml`. However, the XML document 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 is valid.

We discuss validation, DTDs and schemas, as well as the key differences between these two types of structural specifications, in Sections 12.5–12.6. For now, note that schemas are XML documents themselves, whereas DTDs are not. As you'll learn in Section 12.6, this difference presents several advantages in using schemas over DTDs.

**Software Engineering Observation 12.1**

*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, not formatting instructions, so applications that process XML documents must decide how to manipulate or display the data. For example, a PDA (personal digital assistant) may render an XML document differently than a wireless phone or a desktop computer. You can use ***Extensible Stylesheet Language (XSL)*** to specify rendering instructions for different platforms. We discuss XSL in Section 12.8.

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 12.8.

## 12.3 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. 12.2, 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.

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 12.1**

*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.*

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.2: 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. 12.2** | XML used to mark up an article.

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 are delimited by `<!--` and `-->`, can be placed almost anywhere in an XML document and can span multiple lines. There must be exactly one end marker (`-->`) for each begin marker (`<!--`).

### Common Programming Error 12.1

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

### Common Programming Error 12.2

*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 12.3

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

In Fig. 12.2, `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.

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 12.4

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

### Good Programming Practice 12.1

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

XML elements are ***nested*** to form hierarchies—with the root element at the top of the hierarchy. This allows document authors to create parent/child relationships between data. For example, elements `title`, `date`, `author`, `summary` and `content` are nested within `article`. Elements `firstName` and `lastName` are nested within `author`. We discuss the hierarchy of Fig. 12.2 later in this chapter (Fig. 12.25).

### Common Programming Error 12.5

*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 are ***siblings*** of one another.

### *Viewing an XML Document in Internet Explorer and Firefox*

The XML document in Fig. 12.2 is simply a text file named `article.xml`. This document does not contain formatting information for the article. This is because XML is a technology for describing the structure of data. Formatting and displaying data from an XML document are application-specific issues. For example, when the user loads `article.xml` in Internet Explorer, MSXML (Microsoft XML Core Services) parses and displays the document's data. Firefox has a similar capability. Each browser has a built-in ***style sheet*** to format the data. Note that the resulting format of the data (Fig. 12.3) is similar to the format of the listing in Fig. 12.2. In Section 12.8, we show how to create style sheets to transform your XML data into various formats suitable for display.

Note the minus sign (**–**) and plus sign (**+**) in the screen shots of Fig. 12.3. Although these symbols are not part of the XML document, both browsers place them next to every container element. A minus sign indicates that the browser is displaying the container element's child elements. Clicking the minus sign next to an element collapses that element (i.e., causes the browser to hide the container element's children and replace the minus sign with a plus sign). Conversely, clicking the plus sign next to an element expands that element (i.e., causes the browser to display the container element's children and replace the plus sign with a minus sign). This behavior is similar to viewing the directory structure on your system in Windows Explorer 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 12.9.

a) `article.xml` in Internet Explorer.

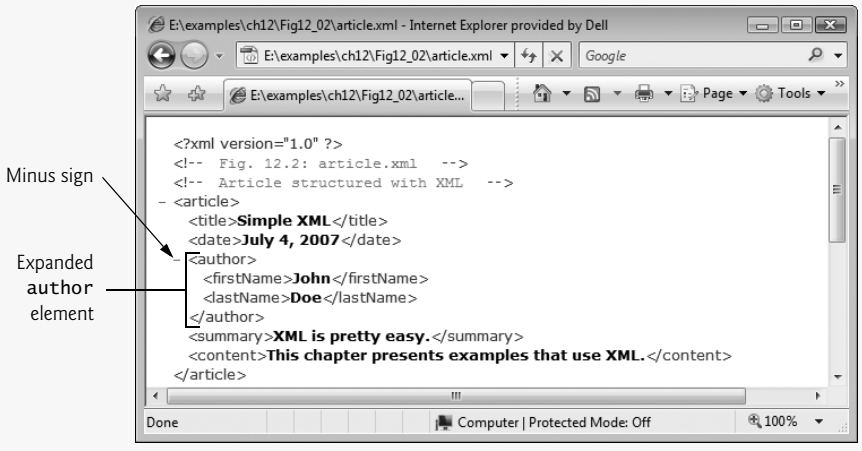



**Fig. 12.3** | `article.xml` displayed by Internet Explorer 7 and Firefox 3. (Part 1 of 2.)

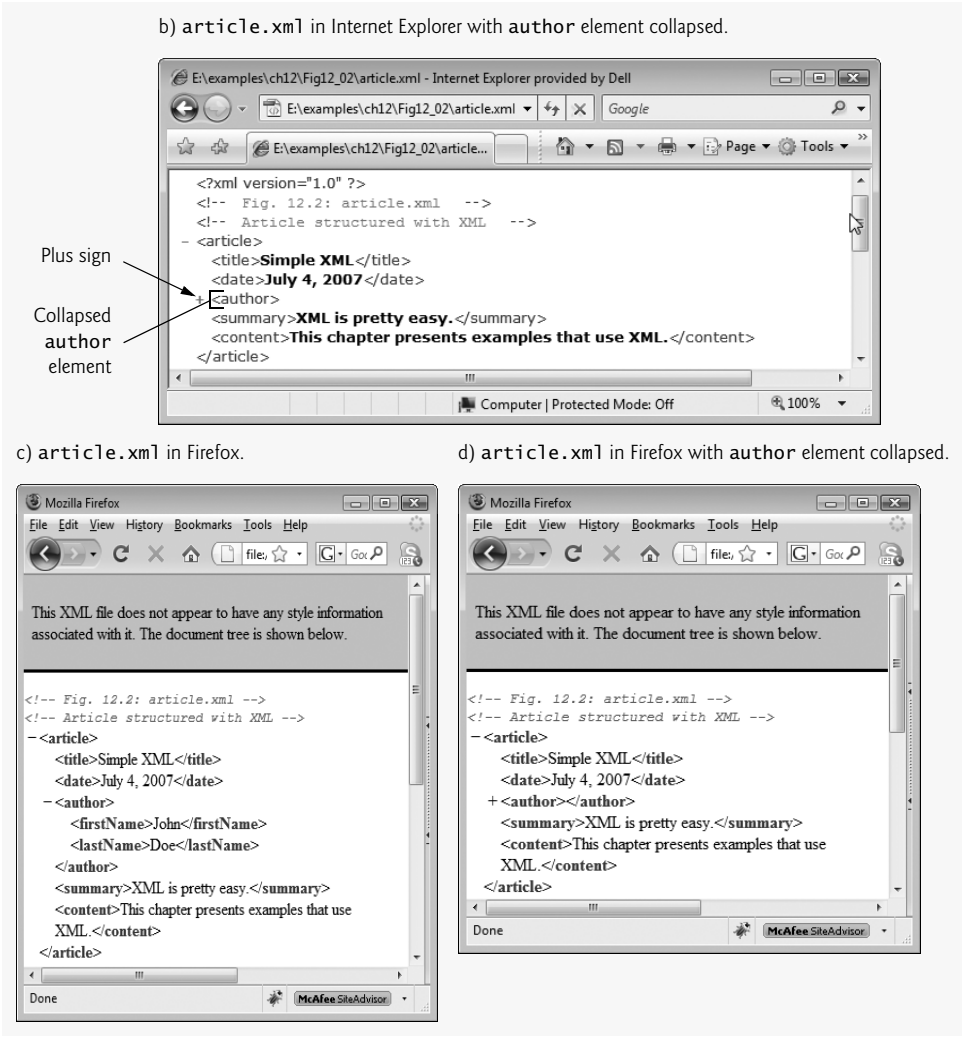



**Fig. 12.3** | `article.xml` displayed by Internet Explorer 7 and Firefox 3. (Part 2 of 2.)

[*Note:* In Windows XP and Windows Vista, by default Internet Explorer displays all the XML elements in expanded view, and clicking the minus sign (Fig. 12.3(a)) does not do anything. To enable collapsing and expanding, right click the *Information Bar* that appears just below the **Address** field and select **Allow Blocked Content...**. Then click **Yes** in the pop-up window that appears.]

### *XML Markup for a Business Letter*

Now that you've seen a simple XML document, let's examine a more complex XML document that marks up a business letter (Fig. 12.4). Again, we begin the document with the XML declaration (line 1) that states the XML version to which the document conforms.

Line 5 specifies that this XML document references a DTD. Recall from Section 12.2 that DTDs define the structure of the data for an XML document. For example, a DTD

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

<!-- Fig. 12.4: letter.xml -->
<!-- Business letter marked up as 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>

   <closing>Sincerely,</closing>
   <signature>Ms. Jane Doe</signature>
</letter>
```

**Fig. 12.4** | Business letter marked up as XML.

specifies the elements and parent/child relationships between elements permitted in an XML document.

### Error-Prevention Tip 12.1

*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 12.2

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

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 qualified URL). DTD document filenames typically end with the ***`.dtd`*** extension. We discuss DTDs and `letter.dtd` in detail in Section 12.5.

Several tools (many of which are free) validate documents against DTDs (discussed in Section 12.5) and schemas (discussed in Section 12.6). A free XML validator can be found at `www.xmlvalidation.com`. This validator can validate XML documents against both DTDs and schemas. You can paste your XML code into the provided text area, or upload the XML document (Fig. 12.5(a)). If you wish to validate the document against a DTD, simply click the **validate** button after pasting in your code or uploading the document. The next screen will prompt you to paste in your DTD code or upload the DTD file (Fig. 12.5(b)). The output (Fig. 12.6) shows the results of validating the document using this online validator—in this case, no errors were found so the XML document is valid. Visit `www.w3.org/XML/Schema` for a list of additional validation tools.

Root element `letter` (lines 7–43 of Fig. 12.4) contains the child elements `contact`, `contact`, `salutation`, `paragraph`, `paragraph`, `closing` and `signature`. Data can be placed between an elements' tags or as ***attributes***—name/value pairs that appear within

(a)

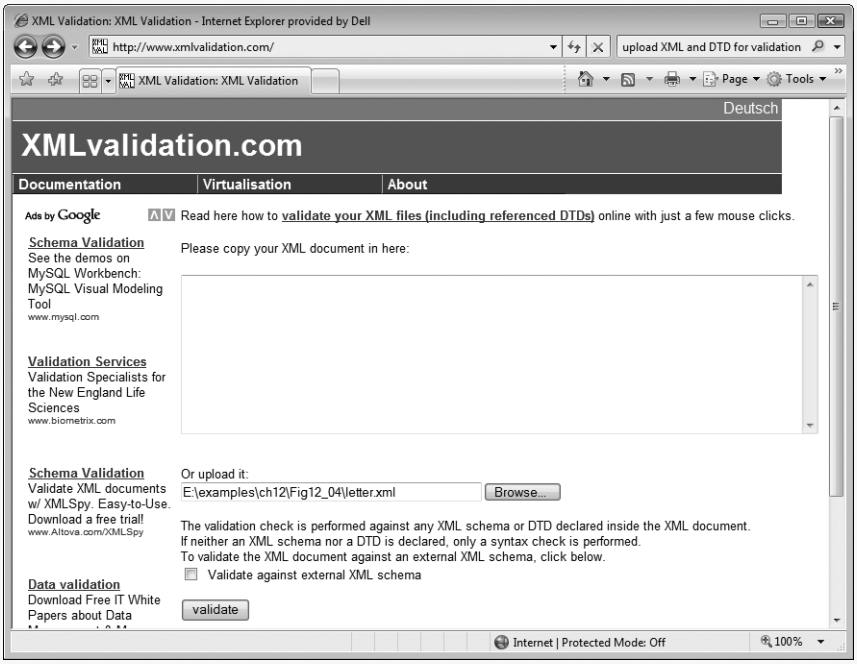



**Fig. 12.5** | Validating an XML document with Microsoft's XML Validator. (Part 1 of 2.)

(b)

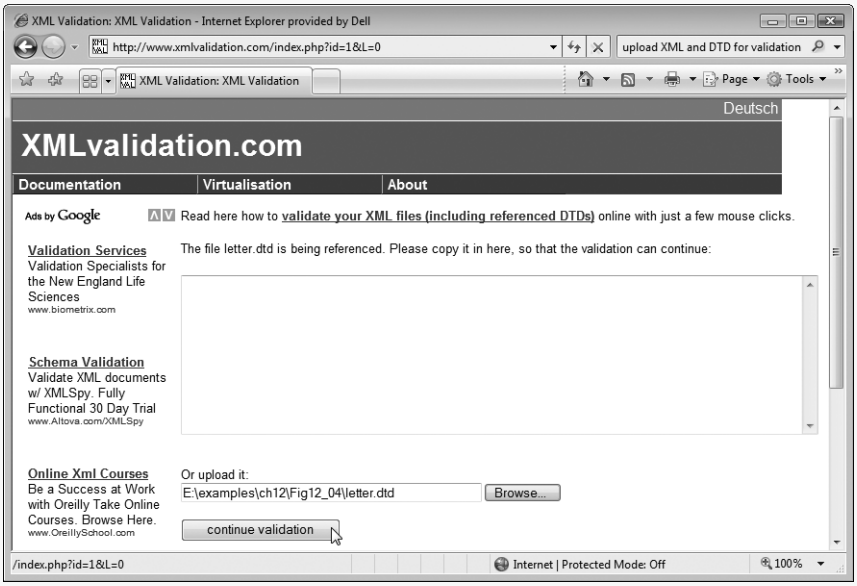



**Fig. 12.5** | Validating an XML document with Microsoft's XML Validator. (Part 2 of 2.)

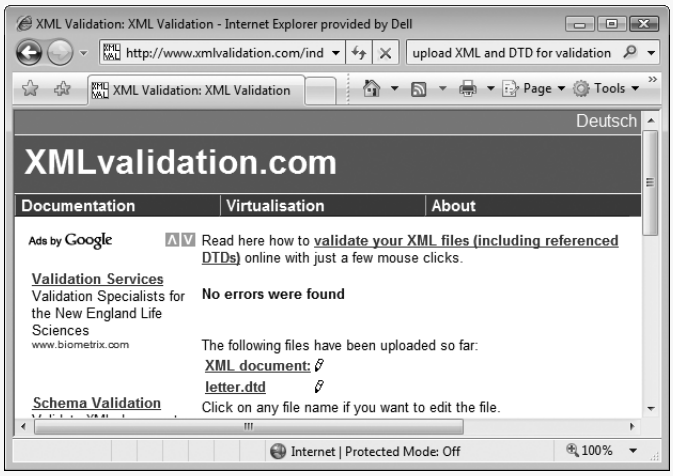



**Fig. 12.6** | Validation result using Microsoft's XML Validator.

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 letter's recipient. Like element names, attribute names are case sensitive, can be any length, may contain letters, digits, underscores, hyphens and periods, and must begin with either

a letter or 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 using 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 12.6**

*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, an empty element 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>
```

Note that the `address2` element in line 22 is empty because there is no second part to this contact's address. However, we must include this element to conform to the structural rules specified in the XML document's DTD—`letter.dtd` (which we present in Section 12.5). This DTD specifies that each `contact` element must have an `address2` child element (even if it is empty). In Section 12.5, you'll learn how DTDs indicate required and optional elements.

## 12.4 XML Namespaces

XML allows document authors to create custom elements. This extensibility can result in ***naming collisions*** among elements in an XML document that each 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 next subsections, we demonstrate how document authors ensure that namespaces are unique.

### Common Programming Error 12.7

*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 12.7 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. 12.7: 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>
```

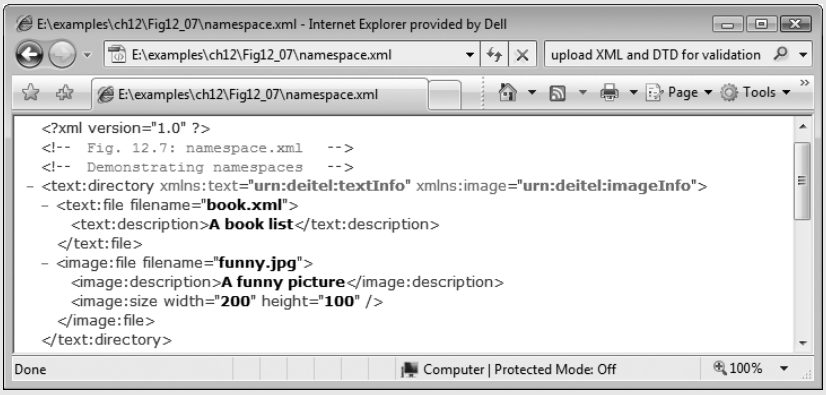



**Fig. 12.7** | XML namespaces demonstration.

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)***.

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

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"
```

Lines 9–11 use the `text` namespace prefix for elements `file` and `description`. Note that the end tags must also specify the namespace prefix `text`. Lines 13–16 apply namespace prefix `image` to the elements `file`, `description` and `size`. Note that 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 12.8 demonstrates using a default namespace (`urn:deitel:textInfo`) for element `directory`.

Line 5 defines a default namespace using attribute `xmlns` with no prefex specified, but with a URI as its value. Once we define this default namespace, child elements belonging to the namespace need not be qualified by a namespace prefix. Thus, element `file` (lines 8–10) is in the default namespace `urn:deitel:textInfo`. Compare this to lines 9–10 of Fig. 12.7, where we had to prefix the `file` and `description` element names with the namespace prefix `text`.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.8: 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>
```

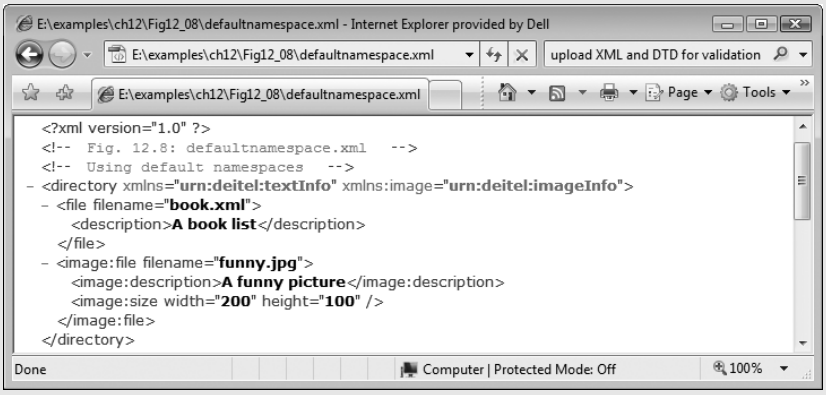



**Fig. 12.8** | Default namespace demonstration.

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 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 12.6) and Extensible Stylesheet Language (XSL) (Section 12.8), often use namespaces to identify their elements. Each of these vocabularies defines special-purpose elements that are grouped in namespaces. These namespaces help prevent naming collisions between predefined elements and user-defined elements.

## 12.5 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 12.6 presents W3C XML Schema documents, which provide an improved method of specifying XML document structure.

### Software Engineering Observation 12.2

*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 12.6) solve this problem by providing an extensible way to describe XML document structure. Applications should use DTDs or schemas to confirm whether XML documents are valid.*

### Software Engineering Observation 12.3

*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 12.4 presented a simple business letter marked up with XML. Recall that line 5 of `letter.xml` references a DTD—`letter.dtd` (Fig. 12.9). This DTD specifies the business letter's element types and attributes, and their relationships to one another.

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`.]

```
1  <!-- Fig. 12.9: letter.dtd       -->
2  <!-- DTD document for letter.xml -->
3
4  <!ELEMENT letter ( contact+, salutation, paragraph+,
5     closing, signature )>
6
7  <!ELEMENT contact ( name, address1, address2, city, state,
8     zip, phone, flag )>
9  <!ATTLIST contact type CDATA #IMPLIED>
10
11 <!ELEMENT name ( #PCDATA )>
12 <!ELEMENT address1 ( #PCDATA )>
13 <!ELEMENT address2 ( #PCDATA )>
14 <!ELEMENT city ( #PCDATA )>
15 <!ELEMENT state ( #PCDATA )>
16 <!ELEMENT zip ( #PCDATA )>
17 <!ELEMENT phone ( #PCDATA )>
18 <!ELEMENT flag EMPTY>
19 <!ATTLIST flag gender (M | F) "M">
20
21 <!ELEMENT salutation ( #PCDATA )>
22 <!ELEMENT closing ( #PCDATA )>
23 <!ELEMENT paragraph ( #PCDATA )>
24 <!ELEMENT signature ( #PCDATA )>
```

**Fig. 12.9** | Document Type Definition (DTD) for a business letter.

### Common Programming Error 12.8

*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 occurence 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 a `type` attribute 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***. `#REQUIRED` specifies that the attribute must be present in the element, and `#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 12.4

*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 is processed by an XML parser). Elements with parsed character data cannot contain markup characters, such as less than (`<`), greater than (`>`) or ampersand (`&`). The document author should replace any markup character in a `#PCDATA` element with the character's corresponding ***character entity reference***. For example, the character entity reference `<` should be used in place of the less-than symbol (`<`), and the character entity reference `>` should be used in place of the greater-than symbol (`>`). A

document author who wishes to use a literal ampersand should use the entity reference `&` instead—parsed character data can contain ampersands (&) only for inserting entities.

### Common Programming Error 12.9

*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 example, `flag`'s data appears in its `gender` attribute (line 19). Line 19 specifies that the `gender` attribute's value must be one of the enumerated values (`M` or `F`) enclosed in parentheses and delimited by a vertical bar (`|`) meaning "or." Note that line 19 also indicates that `gender` has a default value of `M`.

### *Well-Formed Documents vs. Valid Documents*

In Section 12.3, we demonstrated how to use an online XML validator to validate an XML document against its specified DTD. The validation revealed that the XML document `letter.xml` (Fig. 12.4) is well-formed and valid—it conforms to `letter.dtd` (Fig. 12.9). 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 is well-formed.

When a document fails to conform to a DTD or a schema, the XML validator we demonstrated in Section 12.3 displays an error message. For example, the DTD in Fig. 12.9 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. In such a scenario, the XML validator displays an error message like the one in Fig. 12.10.

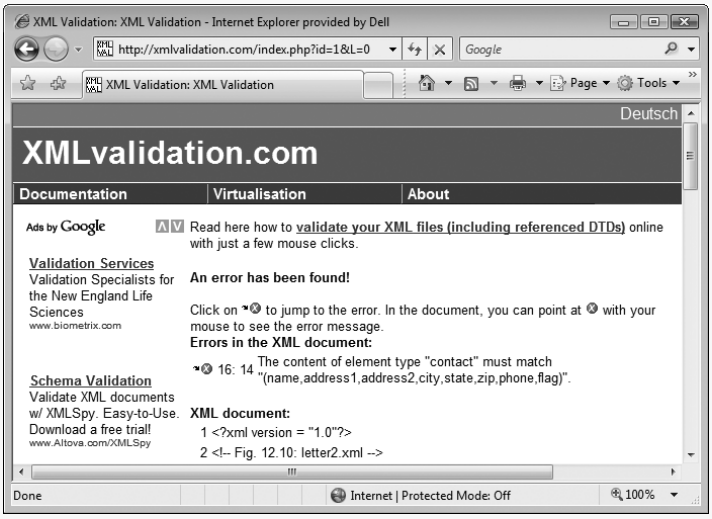



**Fig. 12.10** | XML Validator displaying an error message.

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

In this section, we focus on the W3C's ***XML Schema*** vocabulary (note the capital "S" in "Schema"). We use the term XML Schema in the rest of the chapter whenever we refer to W3C's XML Schema vocabulary. For the latest information on XML Schema, visit `www.w3.org/XML/Schema`. For tutorials on XML Schema concepts beyond what we present here, visit `www.w3schools.com/schema/default.asp`.

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 is 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 12.11 shows a schema-valid XML document named `book.xml`, and Fig. 12.12 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. 12.11 conforms to the schema in Fig. 12.12. To validate the schema document itself (i.e., `book.xsd`) and produce the output shown in Fig. 12.12, we used an online XSV (XML Schema Validator) provided by the W3C at

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

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

Figure 12.11 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.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.11: 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 2005 How to Program, 3/e</title>
8     </book>
9     <book>
10       <title>Visual C# 2005 How to Program, 2/e</title>
11    </book>
12    <book>
13       <title>Java How to Program, 7/e</title>
14    </book>
15    <book>
16       <title>C++ How to Program, 6/e</title>
17    </book>
18    <book>
19       <title>Internet and World Wide Web How to Program, 4/e</title>
20    </book>
21 </deitel:books>
```

**Fig. 12.11** | Schema-valid XML document describing a list of books.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.12: 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. 12.12** | XML Schema document for `book.xml`. (Part 1 of 2.)

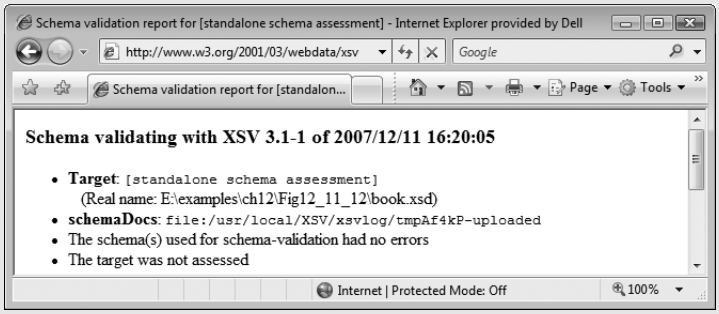



**Fig. 12.12** | XML Schema document for `book.xml`. (Part 2 of 2.)

### *Creating an XML Schema Document*

Figure 12.12 presents the XML Schema document that specifies the structure of `book.xml` (Fig. 12.11). 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. 12.12, 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 12.3

*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. 12.11). This is what "connects" the XML document with the schema that defines its structure. When an XML schema validator examines `book.xml` and `book.xsd`, it will recognize that `book.xml` uses elements and attributes from the `http://www.deitel.com/booklist` namespace. The validator also will recognize that this namespace is the 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) defines an element to be included in an XML document that conforms to the schema. In other words, `element` specifies the actual *ele-*

*ments* 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. 12.11). 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). Figure 12.13 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`.

| XML Schema type | Description | Ranges or structures | 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 is 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. 12.13** | 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) in the `http://www.deitel.com/booklist` namespace and therefore must have the namespace prefix `deitel`. It is not an existing XML Schema type.

Two categories of type exist in XML Schema—***simple types*** and ***complex types***. Simple and complex types 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. 12.13) or user-defined types (e.g., `SingleBookType` in Fig. 12.12).

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. 12.14 creates both simple types and complex types. The XML document in Fig. 12.15 (`laptop.xml`) follows the structure defined in Fig. 12.14 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. 12.14: 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">
```

**Fig. 12.14** | XML Schema document defining simple and complex types. (Part 1 of 2.)

```
 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. 12.14** | XML Schema document defining simple and complex types. (Part 2 of 2.)

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

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 is 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 are 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 have now created an element named `laptop` that contains child elements `processor`, `monitor`, `CPUSpeed` and `RAM`, and an attribute `manufacturer`. Figure 12.15 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 is 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 12.7 discusses several XML vocabularies and demonstrates the MathML vocabulary. Section 12.10 demonstrates the RSS vocabulary.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.15: 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. 12.15** | XML document using the `laptop` element defined in `computer.xsd`.

## 12.7 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 12.8. The following subsections describe MathML and other custom markup languages.

### 12.7.1 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. One application that can parse, render and edit MathML is the W3C's *Amaya*™ browser/editor, which can be downloaded from

```
www.w3.org/Amaya/User/BinDist.html
```

This page contains download links for several platforms. Amaya documentation and installation notes also are available at the W3C website. Firefox also can render MathML, but it requires additional fonts. Instructions for downloading and installing these fonts are available at `www.mozilla.org/projects/mathml/fonts/`. You can download a plug-in (`www.dessci.com/en/products/mathplayer/`) to render MathML in Internet Explorer .

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 mathematical 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 meaning 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 12.16 uses MathML to mark up a simple expression. For this example, we show the expression rendered in Firefox.

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.

```
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. 12.16** | Expression marked up with MathML and displayed in Firefox. (Part 1 of 2.)

```
<!-- Fig. 12.16: mathml1.mml -->
<!-- MathML equation. -->
<math xmlns="http://www.w3.org/1998/Math/MathML">
   <mn>2</mn>
   <mo>+</mo>
   <mn>3</mn>
   <mo>=</mo>
   <mn>5</mn>
</math>
```

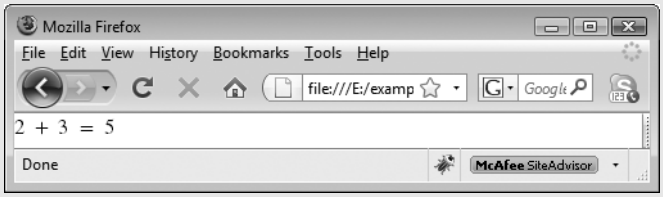



**Fig. 12.16** | Expression marked up with MathML and displayed in Firefox. (Part 2 of 2.)

### *Algebraic Equation in MathML*

Let's consider using MathML to mark up an algebraic equation containing exponents and arithmetic operators (Fig. 12.17). For this example, we again show the expression rendered in Firefox.

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

<!-- Fig. 12.17: mathml2.html -->
<!-- MathML algebraic equation. -->
<math xmlns="http://www.w3.org/1998/Math/MathML">
   <mn>3</mn>
   <mo>&InvisibleTimes;</mo>
   <msup>
      <mi>x</mi>
      <mn>2</mn>
   </msup>
   <mo>+</mo>
   <mn>x</mn>
   <mo>&minus;</mo>
   <mfrac>
      <mn>2</mn>
      <mi>x</mi>
   </mfrac>
   <mo>=</mo>
   <mn>0</mn>
</math>
```

**Fig. 12.17** | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 1 of 2.)

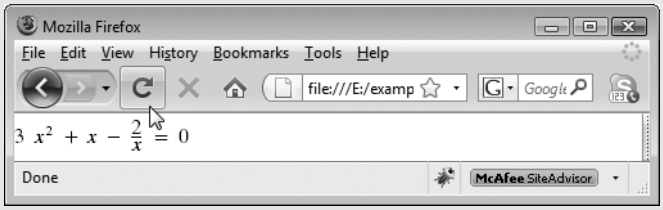



**Fig. 12.17** | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 2 of 2.)

Line 9 uses *entity reference* ***⁢*** 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 uses 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 12.18 marks up a calculus expression that contains an integral symbol and a square-root symbol.

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 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`.

```
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. 12.18 mathml3.html -->
6  <!-- Calculus example using MathML -->
```

**Fig. 12.18** | Calculus expression marked up with MathML and displayed in the Amaya browser. [Courtesy of World Wide Web Consortium (W3C).] (Part 1 of 2.)

```
<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>
```

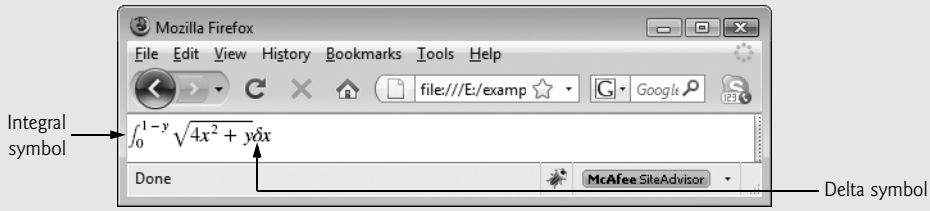



**Fig. 12.18** | Calculus expression marked up with MathML and displayed in the Amaya browser. [Courtesy of World Wide Web Consortium (W3C).] (Part 2 of 2.)

### 12.7.2 Other Markup Languages

Literally hundreds of markup languages derive from XML. Every day developers find new uses for XML. Figure 12.20 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.

## 12.8 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

| 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. 12.19** | Various markup languages derived from XML.

used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents.

The third portion of XSL—XSL Transformations (XSLT)—is a technology for transforming XML documents into other documents—i.e., transforming the structure of the XML document data to another structure. XSLT provides elements that define rules for transforming one XML document to produce a different XML document. This is useful when you want to use data in multiple applications or on multiple platforms, each of which may be designed to work with documents written in a particular vocabulary. For example, XSLT allows you to convert a simple XML document to an XHTML 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 be transformed) and the ***result tree*** (i.e., the XML document to be created). XPath is used to locate parts of the source-tree document that

match *templates* defined in an *XSL style sheet*. When a match occurs (i.e., a node matches a template), the matching template executes and adds its result to the result tree. When there are no more matches, XSLT has transformed the source tree into the result tree. The XSLT does not analyze every node of the source tree; it selectively navigates the source tree using 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 12.20 lists an XML document that describes various sports. The output shows the result of the transformation (specified in the XSLT template of Fig. 12.21) rendered by Internet Explorer.

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. 12.20 is transformed into an XHTML document by MSXML when the document is loaded in Internet Explorer. MSXML is both an XML parser and an XSLT processor. Firefox also includes an XSLT processor.

```
1  <?xml version = "1.0"?>
2  <?xml-stylesheet type = "text/xsl" href = "sports.xsl"?>
3
4  <!-- Fig. 12.20: sports.xml -->
5  <!-- Sports Database -->
6
7  <sports>
8     <game id = "783">
9        <name>Cricket</name>
10
11       <paragraph>
12          More popular among commonwealth nations.
13       </paragraph>
14    </game>
15
16    <game id = "239">
17       <name>Baseball</name>
18
19       <paragraph>
20          More popular in America.
21       </paragraph>
22    </game>
23
24    <game id = "418">
25       <name>Soccer (Futbol)</name>
26
27       <paragraph>
28          Most popular sport in the world.
29       </paragraph>
30    </game>
31 </sports>
```

**Fig. 12.20** | XML document that describes various sports. (Part 1 of 2.)

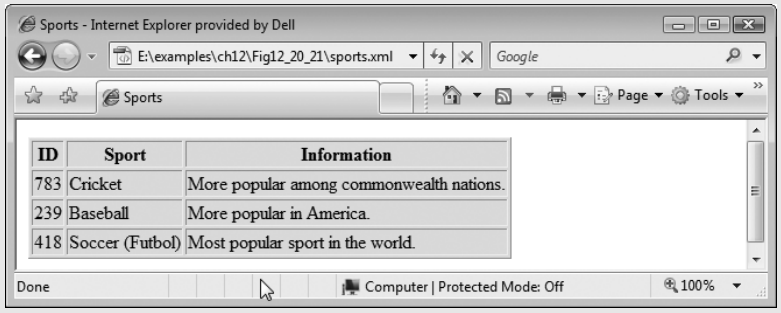



**Fig. 12.20** | XML document that describes various sports. (Part 2 of 2.)

Line 2 (Fig. 12.20) is a ***processing instruction*** (***PI***) that references the XSL style sheet `sports.xsl` (Fig. 12.21). 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. 12.20) 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 12.5

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

### Common Programming Error 12.10

*You will 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 12.21 shows the XSL document for transforming the structured data of the XML document of Fig. 12.20 into an XHTML document for presentation. By convention, XSL documents have the filename extension `.xsl`.

```
1  <?xml version = "1.0"?>
2  <!-- Fig. 12.21: 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
```

**Fig. 12.21** | XSLT that creates elements and attributes in an XHTML document. (Part 1 of 2.)

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

**Fig. 12.21** | XSLT that creates elements and attributes in an XHTML document. (Part 2 of 2.)

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`).

Lines 9–12 use element ***xsl:output*** to write an XHTML document type declaration (`DOCTYPE`) to the result tree (i.e., the XML document to be created). The `DOCTYPE` identifies XHTML as the type of the resulting document. Attribute `method` is assigned `"html"`, which indicates that HTML is being output to the result tree. Attribute ***omit-xml-declaration*** specifies whether the transformation should write the XML declaration to the result tree. In this case, we do not want to omit the XML declaration, so we assign to this attribute the value `"no"`. Attributes `doctype-system` and `doctype-public` write the `DOCTYPE` DTD information to the result tree.

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 14 uses the `match` attribute to select the *document root* (i.e., the conceptual part of the document that contains the root element and everything below it) of the XML source document (i.e., `sports.xml`). The XPath character `/` (a forward slash) 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 are using *absolute addressing* (i.e., we are starting from the root and defining paths down the source tree). In the XML document of Fig. 12.20, 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. 12.21, line 14, matches a node (i.e., the root node), so the contents of the template are now added to the result tree.

The MSXML processor writes the XHTML in lines 16–29 (Fig. 12.21) to the result tree exactly as it appears in the XSL document. Now the result tree consists of the `DOCTYPE` definition and the XHTML code from lines 16–29. Lines 33–39 use element `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 are 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 34–38 (these lines produce one row in a table each time they execute) and places the result of lines 34–38 in the result tree.

Line 35 uses element `value-of` to retrieve attribute `id`'s value and place it in a `td` element in the result tree. The XPath symbol `@` specifies that `id` is an attribute node of the context node `game`. Lines 36–37 place the `name` and `paragraph` element values in `td` elements and insert them in the result tree. When an XPath expression has no beginning forward slash, the expression uses *relative addressing*. Omitting the beginning forward slash tells the `xsl:value-of select` statements to search for `name` and `paragraph` elements that are children of the context node, not the root node. Due 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 12.22 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
```

**Fig. 12.22** | XML document containing book information. (Part 1 of 2.)

```
<!-- Fig. 12.22: sorting.xml -->
<!-- XML document containing book information -->
<book isbn = "999-99999-9-X">
   <title>Deitel&apos;s XML Primer</title>

   <author>
      <firstName>Jane</firstName>
      <lastName>Blue</lastName>
   </author>

   <chapters>
      <frontMatter>
         <preface pages = "2" />
         <contents pages = "5" />
         <illustrations pages = "4" />
      </frontMatter>

      <chapter number = "3" pages = "44">Advanced XML</chapter>
      <chapter number = "2" pages = "35">Intermediate XML</chapter>
      <appendix number = "B" pages = "26">Parsers and Tools</appendix>
      <appendix number = "A" pages = "7">Entities</appendix>
      <chapter number = "1" pages = "28">XML Fundamentals</chapter>
   </chapters>

   <media type = "CD" />
</book>
```

**Fig. 12.22** | XML document containing book information. (Part 2 of 2.)

Figure 12.23 presents an XSL document (`sorting.xsl`) for transforming `sorting.xml` (Fig. 12.22) to XHTML. Recall that an XSL document navigates a source tree and builds a result tree. In this example, the source tree is XML, and the output tree is XHTML. Line 14 of Fig. 12.23 matches the root element of the document in Fig. 12.22. Line 15 outputs an `html` start tag to the result tree. In line 16, the `<xsl:apply-templates/>` element specifies that the XSLT processor is to apply the `xsl:templates` defined in this XSL document to the current node's (i.e., the document root's) children. The content from the applied templates is output in the `html` element that ends at line 17.

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

<!-- Fig. 12.23: sorting.xsl -->
<!-- Transformation of book information into XHTML -->
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

   <!-- write XML declaration and DOCTYPE DTD information -->
   <xsl:output method = "html" omit-xml-declaration = "no"
      doctype-system = "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"
      doctype-public = "-//W3C//DTD XHTML 1.1//EN"/>

```

**Fig. 12.23** | XSL document that transforms `sorting.xml` into XHTML. (Part 1 of 3.)

```
13    <!-- match document root -->
14    <xsl:template match = "/">
15       <html xmlns = "http://www.w3.org/1999/xhtml">
16          <xsl:apply-templates/>
17       </html>
18    </xsl:template>
19
20    <!-- match book -->
21    <xsl:template match = "book">
22       <head>
23          <title>ISBN <xsl:value-of select = "@isbn"/> -
24             <xsl:value-of select = "title"/></title>
25       </head>
26
27       <body>
28          <h1 style = "color: blue"><xsl:value-of select = "title"/></h1>
29          <h2 style = "color: blue">by
30             <xsl:value-of select = "author/lastName"/>,
31             <xsl:value-of select = "author/firstName"/></h2>
32
33          <table style = "border-style: groove; background-color: wheat">
34
35             <xsl:for-each select = "chapters/frontMatter/*">
36                <tr>
37                   <td style = "text-align: right">
38                      <xsl:value-of select = "name()"/>
39                   </td>
40
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 style = "text-align: right">
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"/>
```

**Fig. 12.23** | XSL document that transforms `sorting.xml` into XHTML. (Part 2 of 3.)

```
65                <tr>
66                   <td style = "text-align: right">
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          <br /><p style = "color: blue">Pages:
79             <xsl:variable name = "pagecount"
80                select = "sum(chapters//*/@pages)"/>
81             <xsl:value-of select = "$pagecount"/>
82          <br />Media Type: <xsl:value-of select = "media/@type"/></p>
83       </body>
84    </xsl:template>
85 </xsl:stylesheet>
```

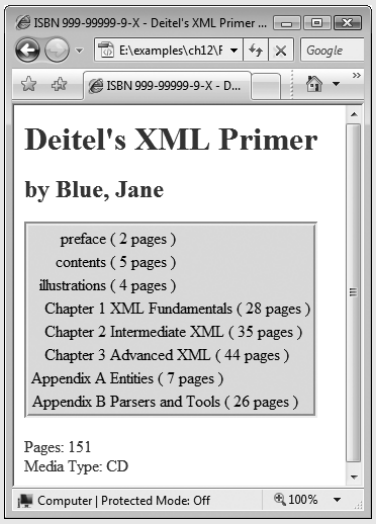



**Fig. 12.23** | XSL document that transforms `sorting.xml` into XHTML. (Part 3 of 3.)

Lines 21–84 specify a template that matches element `book`. The template indicates how to format the information contained in `book` elements of `sorting.xml` (Fig. 12.22) as XHTML.

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

Line 28 creates a header element that contains the book's title. Lines 29–31 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 is a child of element `frontMatter`. 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 Style-Sheet Elements*

This section's examples used several predefined XSL elements to perform various operations. Figure 12.24 lists these elements 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. |

**Fig. 12.24** | XSL style-sheet elements. (Part 1 of 2.)

| Element | Description |
|---|---|
| `<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, XHTML), 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. 12.24** | XSL style-sheet elements. (Part 2 of 2.)

This section introduced Extensible Stylesheet Language (XSL) and showed how to create XSL transformations to convert XML documents from one format to another. We showed how to transform XML documents to XHTML documents for display in a web browser. Recall that these transformations are performed by MSXML, Internet Explorer's built-in XML parser and XSLT processor. In most business applications, XML documents are transferred between business partners and are transformed to other XML vocabularies programmatically. Section 12.9 discusses the XML Document Object Model (DOM) and demonstrates how to manupulate the DOM of an XML document using JavaScript.

## 12.9 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 12.25 illustrates the tree structure for the root element of the document `article.xml` (Fig. 12.2). 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 XHTML DOM you learned in Chapter 10.

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` 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).

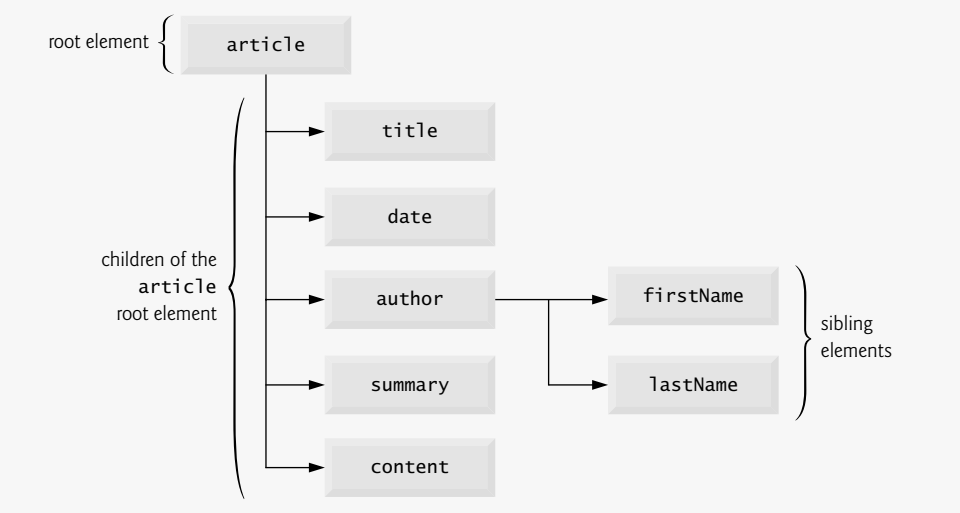



**Fig. 12.25** | Tree structure for the document `article.xml` of Fig. 12.2.

To introduce document manipulation with the XML Document Object Model, we provide a scripting example (Fig. 12.26) that uses JavaScript and XML. This example loads the XML document `article.xml` (Fig. 12.2) 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. All of this is done in a manner that enables the example to execute in both Internet Explorer 7 and Firefox 2 (and higher). Figure 12.26 lists the JavaScript code that manipulates this XML document and displays its content in an XHTML page.

### *Overview of the* `body` *Element*

Lines 203–217 create the XHTML document's `body`. When the body loads, its `onload` event calls our JavaScript function `loadXMLDocument` to load and display the contents of `article.xml` in the `div` at line 216 (`outputDiv`). Lines 204–215 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.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 12.26: XMLDOMTraversal.html -->
6  <!-- Traversing an XML document using the XML DOM. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Traversing an XML document using the XML DOM</title>
10    <style type = "text/css">
```

**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 1 of 9.)

```
11          .highlighted { background-color: yellow }
12          #outputDiv { font: 10pt "Lucida Console", monospace; }
13       </style>
14       <script type="text/javascript">
15       <!--
16       var doc; // variable to reference the XML document
17       var outputHTML = ""; // stores text to output in outputDiv
18       var idCounter = 1; // used to create div IDs
19       var depth = -1; // tree depth is -1 to start
20       var current = null; // represents the current node for traversals
21       var previous = null; // represent prior node in traversals
22
23       // load XML document based on whether the browser is IE7 or Firefox
24       function loadXMLDocument( url )
25       {
26          if ( window.ActiveXObject ) // IE7
27          {
28             // create IE7-specific XML document object
29             doc = new ActiveXObject( "Msxml2.DOMDocument.6.0" );
30             doc.async = false; // specifies synchronous loading of XML doc
31             doc.load( url ); // load the XML document specified by url
32             buildHTML( doc.childNodes ); // display the nodes
33             displayDoc();
34          } // end if
35          else if ( document.implementation &&
36             document.implementation.createDocument ) // other browsers
37          {
38             // create XML document object
39             doc = document.implementation.createDocument( "", "", null );
40             doc.load( url ); // load the XML document specified by url
41             doc.onload = function() // function to execute when doc loads
42             {
43                buildHTML( doc.childNodes ); // called by XML doc onload event
44                displayDoc(); // display the HTML
45             } // end XML document's onload event handler
46          } // end else
47          else // not supported
48             alert( 'This script is not supported by your browser' );
49       } // end function loadXMLDocument
50
51       // traverse xmlDocument and build XHTML representation of its content
52       function buildHTML( childList )
53       {
54          ++depth; // increase tab depth
55
56          // display each node's content
57          for ( var i = 0; i < childList.length; i++ )
58          {
59             switch ( childList[ i ].nodeType )
60             {
61                case 1: // Node.ELEMENT_NODE; value used for portability
62                   outputHTML += "<div id=\"id" + idCounter + "\">";
63                   spaceOutput( depth ); // insert spaces
```

**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 2 of 9.)

```
64                outputHTML += childList[ i ].nodeName; // show node's name
65                ++idCounter; // increment the id counter
66
67                // if current node has children, call buildHTML recursively
68                if ( childList[ i ].childNodes.length != 0 )
69                   buildHTML( childList[ i ].childNodes );
70
71                outputHTML += "</div>";
72                break;
73             case 3: // Node.TEXT_NODE; value used for portability
74             case 8: // Node.COMMENT_NODE; value used for portability
75                // if nodeValue is not 3 or 6 spaces (Firefox issue),
76                // include nodeValue in HTML
77                if ( childList[ i ].nodeValue.indexOf( "   " ) == -1 &&
78                   childList[ i ].nodeValue.indexOf( "      " ) == -1 )
79                {
80                   outputHTML += "<div id=\"id" + idCounter + "\">";
81                   spaceOutput( depth ); // insert spaces
82                   outputHTML += childList[ i ].nodeValue + "</div>";
83                   ++idCounter; // increment the id counter
84                } // end if
85             } // end switch
86          } // end for
87
88          --depth; // decrease tab depth
89       } // end function buildHTML
90
91       // display the XML document and highlight the first child
92       function displayDoc()
93       {
94          document.getElementById( "outputDiv" ).innerHTML = outputHTML;
95          current = document.getElementById( 'id1' );
96          setCurrentNodeStyle( current.id, true );
97       } // end function displayDoc
98
99       // insert non-breaking spaces for indentation
100      function spaceOutput( number )
101      {
102         for ( var i = 0; i < number; i++ )
103         {
104            outputHTML += "   ";
105         } // end for
106      } // end function spaceOutput
107
108      // highlight first child of current node
109      function processFirstChild()
110      {
111         if ( current.childNodes.length == 1 && // only one child
112            current.firstChild.nodeType == 3 )  // and it's a text node
113         {
114            alert( "There is no child node" );
115         } // end if
```

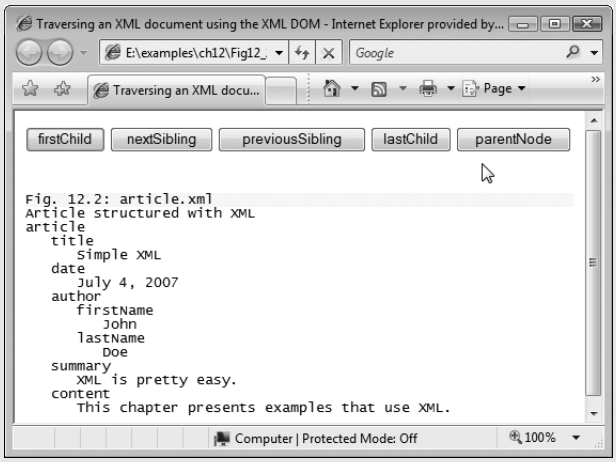
**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 3 of 9.)

```
116        else if ( current.childNodes.length > 1 )
117        {
118           previous = current; // save currently highlighted node
119
120           if ( current.firstChild.nodeType != 3 ) // if not text node
121              current = current.firstChild; // get new current node
122           else // if text node, use firstChild's nextSibling instead
123              current = current.firstChild.nextSibling; // get first sibling
124
125           setCurrentNodeStyle( previous.id, false ); // remove highlight
126           setCurrentNodeStyle( current.id, true ); // add highlight
127        } // end if
128        else
129           alert( "There is no child node" );
130     } // end function processFirstChild
131
132     // highlight next sibling of current node
133     function processNextSibling()
134     {
135        if ( current.id != "outputDiv" && current.nextSibling )
136        {
137           previous = current; // save currently highlighted node
138           current = current.nextSibling; // get new current node
139           setCurrentNodeStyle( previous.id, false ); // remove highlight
140           setCurrentNodeStyle( current.id, true ); // add highlight
141        } // end if
142        else
143           alert( "There is no next sibling" );
144     } // end function processNextSibling
145
146     // highlight previous sibling of current node if it is not a text node
147     function processPreviousSibling()
148     {
149        if ( current.id != "outputDiv" && current.previousSibling &&
150           current.previousSibling.nodeType != 3 )
151        {
152           previous = current; // save currently highlighted node
153           current = current.previousSibling; // get new current node
154           setCurrentNodeStyle( previous.id, false ); // remove highlight
155           setCurrentNodeStyle( current.id, true ); // add highlight
156        } // end if
157        else
158           alert( "There is no previous sibling" );
159     } // end function processPreviousSibling
160
161     // highlight last child of current node
162     function processLastChild()
163     {
164        if ( current.childNodes.length == 1 &&
165           current.lastChild.nodeType == 3 )
166        {
167           alert( "There is no child node" );
168        } // end if
```

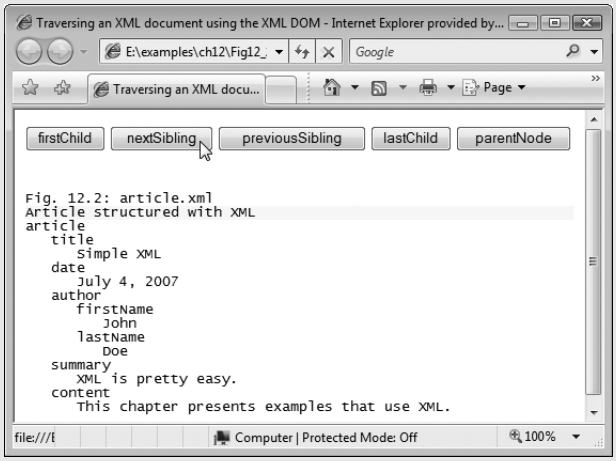
**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 4 of 9.)

```
169         else if ( current.childNodes.length != 0 )
170         {
171            previous = current; // save currently highlighted node
172            current = current.lastChild; // get new current node
173            setCurrentNodeStyle( previous.id, false ); // remove highlight
174            setCurrentNodeStyle( current.id, true ); // add highlight
175         } // end if
176         else
177            alert( "There is no child node" );
178      } // end function processLastChild
179
180      // highlight parent of current node
181      function processParentNode()
182      {
183         if ( current.parentNode.id != "body" )
184         {
185            previous = current; // save currently highlighted node
186            current = current.parentNode; // get new current node
187            setCurrentNodeStyle( previous.id, false ); // remove highlight
188            setCurrentNodeStyle( current.id, true ); // add highlight
189         } // end if
190         else
191            alert( "There is no parent node" );
192      } // end function processParentNode
193
194      // set style of node with specified id
195      function setCurrentNodeStyle( id, highlight )
196      {
197         document.getElementById( id ).className =
198            ( highlight ? "highlighted" : "" );
199      } // end function setCurrentNodeStyle
200      // -->
201      </script>
202   </head>
203   <body id = "body" onload = "loadXMLDocument( 'article.xml' );">
204      <form action = "" onsubmit = "return false;">
205         <input type = "submit" value = "firstChild"
206            onclick = "processFirstChild()"/>
207         <input type = "submit" value = "nextSibling"
208            onclick = "processNextSibling()"/>
209         <input type = "submit" value = "previousSibling"
210            onclick = "processPreviousSibling()"/>
211         <input type = "submit" value = "lastChild"
212            onclick = "processLastChild()"/>
213         <input type = "submit" value = "parentNode"
214            onclick = "processParentNode()"/>
215      </form><br/>
216      <div id = "outputDiv"></div>
217   </body>
218   </html>
```

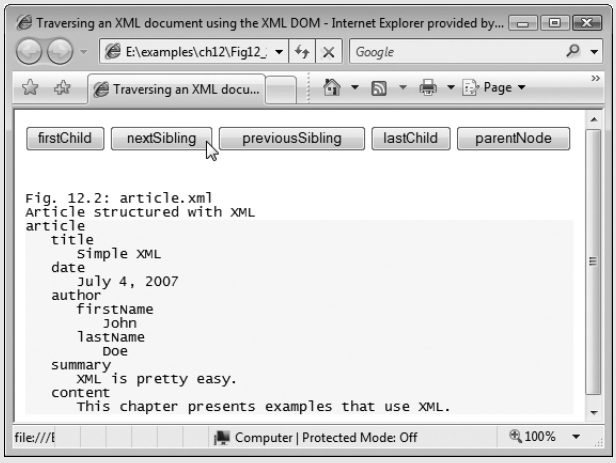
**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 5 of 9.)




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

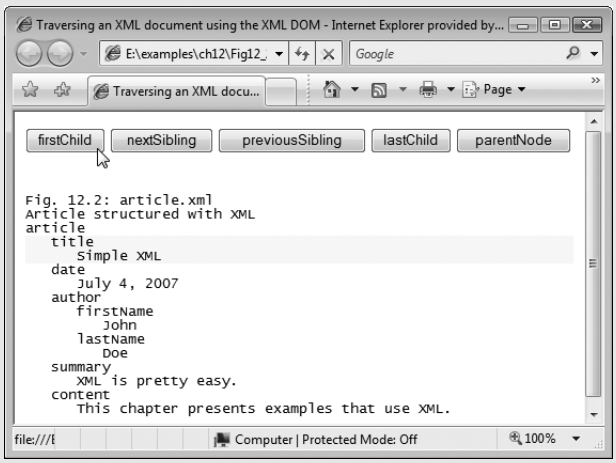



b) User clicked the **nextSibling** button to highlight the second comment node.

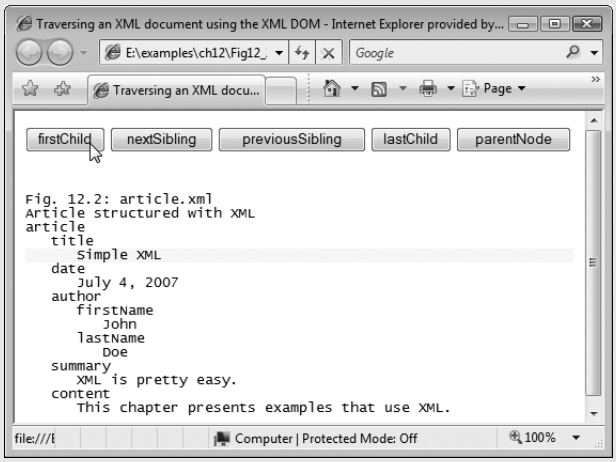



c) User clicked the **nextSibling** button again to highlight the `article` node.

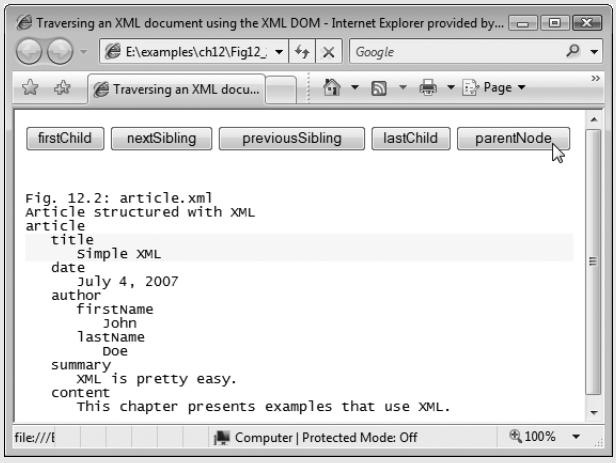
**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 6 of 9.)




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

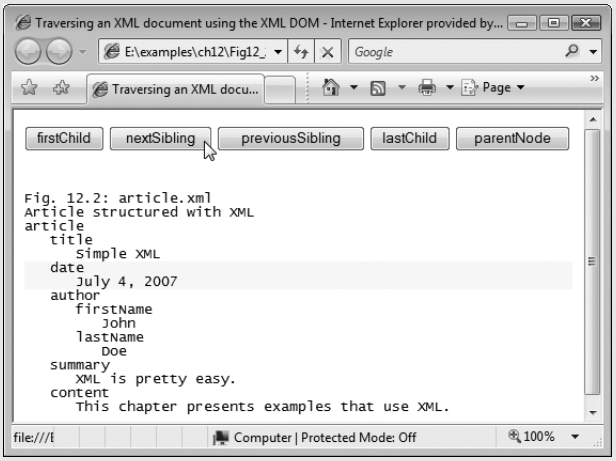



e) User clicked the **firstChild** button again to highlight the `title` node's text child node.

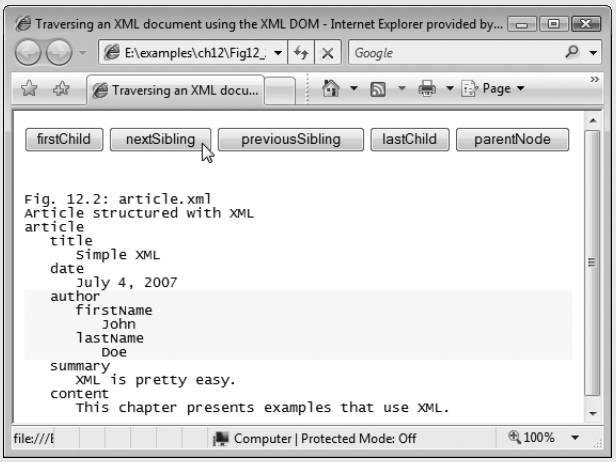



f) User clicked the **parentNode** button to highlight the text node's parent `title` node.

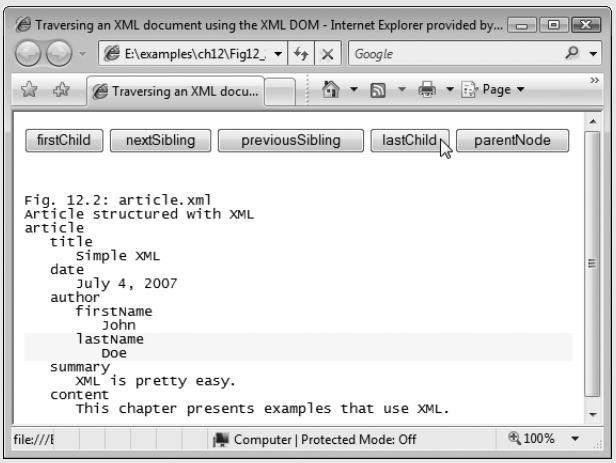
**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 7 of 9.)




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

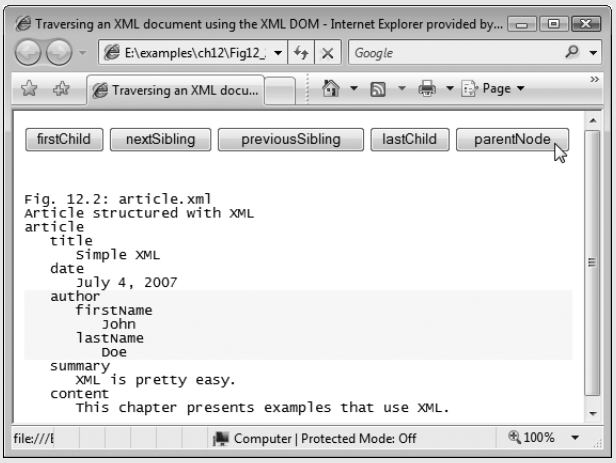



h) User clicked the **nextSibling** button to highlight the `date` node's `author` sibling node.




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

**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 8 of 9.)




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

**Fig. 12.26** | Traversing an XML document using the XML DOM. (Part 9 of 9.)

### *Global Script Variables*

Lines 16–21 in the `script` element (lines 14–201) declare several variables used throughout the script. Variable `doc` references a DOM object representation of `article.xml`. 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 `loadXMLDocument`*

Function `loadXMLDocument` (lines 24–49) receives the URL of an XML document to load, then loads the document based on whether the browser is Internet Explorer 7 (26–34) or Firefox (lines 35–46)—the code for Firefox works in several other browsers as well. Line 26 determines whether `window.ActiveXObject` exists. If so, this indicates that the browser is Internet Explorer. Line 29 creates a Microsoft `ActiveXObject` that loads Microsoft's *MSXML parser*, which provides capabilities for manipulating XML documents. Line 30 indicates that we'd like the XML document to be loaded synchronously, then line 31 uses the `ActiveXObject`'s ***`load` method*** to load `article.xml`. When this completes, we call our `buildHTML` method (defined in lines 52–89) to construct an XHTML 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 92–97) to display the contents of `article.xml` in `outputDiv`.

If the browser is Firefox, then the `document` object's ***`implementation`*** property and the `implementation` property's ***`createDocument`*** method will exist (lines 35–36). In this case, line 39 uses the `createDocument` method to create an empty XML document object. If necessary, you can specify the XML document's namespace as the first argument and its root element as the second argument. We used empty strings for both in this example.

According to the site `www.w3schools.com/xml/xml_parser.asp`, the third argument is not implemented yet, so it should always be `null`. Line 40 calls its `load` method to load `article.xml`. Firefox loads the XML document asynchronously, so you must use the XML document's `onload` property to specify a function to call (an anonymous function in this example) when the document finishes loading. When this event occurs, lines 43–44 call `buildHTML` and `displayDoc` just as we did in lines 32–33.

### Common Programming Error 12.11

*Attempting to process the contents of a dynamically loaded XML document in Firefox before the document's `onload` event fires is a logic error. The document's contents are not available until the `onload` event fires.*

#### *Function `buildHTML`*

Function `buildHTML` (lines 52–89) is a recursive function that receives a list of nodes as an argument. Line 54 increments the `depth` for indentation purposes. Lines 57–86 iterate through the nodes in the list. The `switch` statement (lines 59–85) uses the current node's ***`nodeType` property*** to determine whether the current node is an element (line 61), a text node (i.e., the text content of an element; line 73) or a comment node (line 74). If it is an element, then we begin a new `div` element in our XHTML (line 62) and give it a unique `id`. Then function `spaceOutput` (defined in lines 100–106) 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 64 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 69 recursively calls `buildHTML` to append the current element's child nodes to the markup. When that recursive call completes, line 71 completes the `div` element that we started at line 62.

If the current element is a text node, lines 77–78 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. Unfortunately, unlike MSMXL, Firefox's XML parser does not ignore the white space used for indentation in XML documents. Instead it creates text nodes containing just the space characters. The condition in lines 77–78 enables us to ignore these nodes in Firefox. If the node contains text, lines 80–82 append a new `div` to the markup and use the `node`'s `nodeValue` property to insert that text in the `div`. Line 88 in `buildHTML` decrements the `depth` counter.

### Portability Tip 12.4

*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 92–97), line 94 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 95 sets variable `current` to refer to the `div` with `id 'id1'` in the new markup, and line 96 uses our `setCurrentNodeStyle` method (defined at lines 195–199) to highlight that `div`.

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

Function `processFirstChild` (lines 109–130) is invoked by the `onclick` event of the button at lines 205–206. If the `current` node has only one child and it's a text node (lines 111–112), line 114 displays an alert dialog indicating that there is no child node—we navigate only to nested XML elements in this example. If there are two or more children, line 118 stores the value of `current` in `previous`, and lines 120–123 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 125–126 unhighlight the `previous` node and highlight the new `current` node. Function `processLastChild` (lines 162–178) works similarly, using the `current` node's ***`lastChild`*** property.

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

Function `processNextSibling` (lines 133–144) first ensures that the current node is not the `outputDiv` and that `nextSibling` exists. If so, lines 137–140 adjust the `previous` and `current` nodes accordingly and update their highlighting. Function `processPreviousSibling` (lines 147–159) 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 181–192) first checks whether the current node's ***`parentNode`*** is the XHTML page's `body`. If not, lines 185–188 adjust the `previous` and `current` nodes accordingly and update their highlighting.

### *Common DOM Properties*

The tables in Figs. 12.27–12.32 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. 12.28) 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 is no such node. |
| `nextSibling` | The node following this node; `null` if there is no such node. |

**Fig. 12.27** | Common `Node` properties and methods. (Part 1 of 2.)

| Property/Method | Description |
|---|---|
| `attributes` | A collection of `Attr` objects (Fig. 12.31) 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 is 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. 12.27** | Common `Node` properties and methods. (Part 2 of 2.)

| 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. 12.28** | `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. 12.31) 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. 12.29** | `Document` properties and methods.

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

**Fig. 12.30** | `Element` property and methods. (Part 1 of 2.)

| Property/Method | Description |
|---|---|
| `setAttribute` | Changes the value of the attribute passed as the first argument to the value passed as the second argument. |
| `removeAttribute` | Removes the specified attribute. |
| `getAttributeNode` | Returns the specified attribute node. |
| `setAttributeNode` | Adds a new attribute node with the specified name. |

**Fig. 12.30** | `Element` property and methods. (Part 2 of 2.)

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

**Fig. 12.31** | `Attr` properties.

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

**Fig. 12.32** | `Text` methods.

### *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 12.8).

Figure 12.33 enables the user to enter XPath expressions in an XHTML form. When the user clicks the **Get Matches** button, the script applies the XPath expression to the XML DOM and displays the matching nodes. Figure 12.34 shows the XML document `sports.xml` that we use in this example. [*Note:* The versions of `sports.xml` presented in Fig. 12.34 and Fig. 12.20 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. 12.20. We also removed extra blank lines to save space.]

The program of Fig. 12.33 loads the XML document `sports.xml` (Fig. 12.34) using the same techniques we presented in Fig. 12.26, so we focus on only the new features in this example. Internet Explorer 7 (MSXML) and Firefox handle XPath processing differently, so this example declares the variable `browser` (line 17) to store the browser that

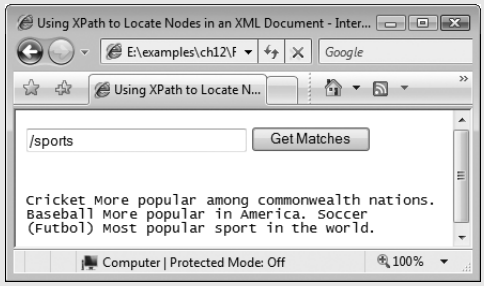
loaded the page. In function `loadDocument` (lines 20–40), lines 28 and 36 assign a string to variable `browser` indicating the appropriate browser.

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 12.33: xpath.html -->
6  <!-- Using XPath to locate nodes in an XML document. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
9     <title>Using XPath to Locate Nodes in an XML Document</title>
10    <style type = "text/css">
11       #outputDiv { font: 10pt "Lucida Console", monospace; }
12    </style>
13    <script type = "text/javascript">
14    <!--
15    var doc; // variable to reference the XML document
16    var outputHTML = ""; // stores text to output in outputDiv
17    var browser = ""; // used to determine which browser is being used
18
19    // load XML document based on whether the browser is IE7 or Firefox
20    function loadXMLDocument( url )
21    {
22       if ( window.ActiveXObject ) // IE7
23       {
24          // create IE7-specific XML document object
25          doc = new ActiveXObject( "Msxml2.DOMDocument.6.0" );
26          doc.async = false; // specifies synchronous loading of XML doc
27          doc.load( url ); // load the XML document specified by url
28          browser = "IE7"; // set browser
29       } // end if
30       else if ( document.implementation &&
31          document.implementation.createDocument ) // other browsers
32       {
33          // create XML document object
34          doc = document.implementation.createDocument( "", "", null );
35          doc.load( url ); // load the XML document specified by url
36          browser = "FF2"; // set browser
37       } // end else
38       else // not supported
39          alert( 'This script is not supported by your browser' );
40    } // end function loadXMLDocument
41
42    // display the XML document
43    function displayDoc()
44    {
45       document.getElementById( "outputDiv" ).innerHTML = outputHTML;
46    } // end function displayDoc
47
48    // obtain and apply XPath expression
49    function processXPathExpression()
50    {
```

**Fig. 12.33** | Using XPath to locate nodes in an XML document. (Part 1 of 3.)

```
51          var xpathExpression = document.getElementById( "inputField" ).value;
52          outputHTML = "";
53
54          if ( browser == "IE7" )
55          {
56             var result = doc.selectNodes( xpathExpression );
57
58             for ( var i = 0; i < result.length; i++ )
59                outputHTML += "<div style='clear: both'>" +
60                   result.item( i ).text + "</div>";
61          } // end if
62          else // browser == "FF2"
63          {
64             var result = document.evaluate( xpathExpression, doc, null,
65                XPathResult.ANY_TYPE, null );
66             var current = result.iterateNext();
67
68             while ( current )
69             {
70                outputHTML += "<div style='clear: both'>" +
71                   current.textContent + "</div>";
72                current = result.iterateNext();
73             } // end while
74          } // end else
75
76          displayDoc();
77       } // end function processXPathExpression
78       // -->
79       </script>
80    </head>
81    <body id = "body" onload = "loadXMLDocument( 'sports.xml' );">
82       <form action = "" onsubmit = "return false;">
83          <input id = "inputField" type = "text" style = "width: 200px"/>
84          <input type = "submit" value = "Get Matches"
85             onclick = "processXPathExpression()"/>
86       </form><br/>
87       <div id = "outputDiv"></div>
88    </body>
89    </html>
```

(a)




(b)

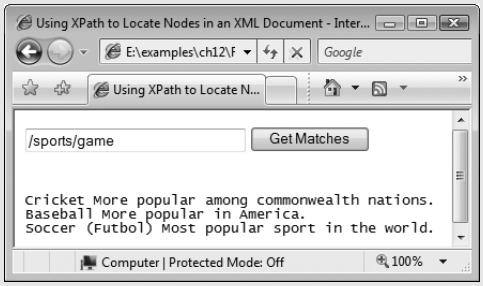



**Fig. 12.33** | Using XPath to locate nodes in an XML document. (Part 2 of 3.)


**Fig. 12.33** | Using XPath to locate nodes in an XML document. (Part 3 of 3.)

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 12.34: 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. 12.34** | XML document that describes various sports.

When the `body` of this XHTML document loads, its `onload` event calls `loadDocument` (line 81) to load the `sports.xml` file. The user specifies the XPath expression in the `input` element at line 83. When the user clicks the **Get Matches** button (lines 84–85), its `onclick` event handler invokes our `processXPathExpression` function to locate any matches and display the results in `outputDiv` (line 87).

Function `processXPathExpression` (lines 49–77) first obtains the XPath expression (line 51). The `document` object's `getElementById` method returns the element with the `id` `"inputField"`; then we use its value property to get the XPath expression. Lines 54–61 apply the XPath expression in Internet Explorer 7, and lines 62–74 apply the XPath expression in Firefox. In IE7, the XML document object's ***`selectNodes` method*** receives an XPath expression as an argument and returns a collection of elements that match the expression. Lines 58–60 iterate through the results and mark up each one in a separate `div` element. After this loop completes, line 76 displays the generated markup in `outputDiv`.

For Firefox, lines 64–65 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. 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 are not using XML namespace prefixes in the XPath processing. Lines 66–73 iterate through the `XPathResult` and mark up the results. Line 66 invokes the `XPathResult`'s `iterateNext` method to position to the first result. If there is a result, the condition in line 68 will be true, and lines 70–71 create a `div` for that result. Line 72 then positions to the next result. After this loop completes, line 76 displays the generated markup in `outputDiv`.

Figure 12.35 summarizes the XPath expressions that we demonstrate in Fig. 12.33's sample outputs. For more information on using XPath in Firefox, visit the site `developer.mozilla.org/en/docs/XPath`. For more information on using XPath in Internet Explorer, visit `msdn.microsoft.com/msdnmag/issues/0900/xml/`.

| Expression | Description |
|---|---|
| `/sports` | Matches all `sports` nodes that are child nodes of the document root node. |
| `/sports/game` | Matches all `game` nodes that are child nodes of `sports`, which is a child of the document root. |
| `/sports/game/name` | Matches all `name` nodes that are children of `game`. The `game` is a child of `sports`, which is a child of the document root. |
| `/sports/game/paragraph` | Matches all `paragraphs` that are children of `game`. The `game` is a child of `sports`, which is a child of the document root. |
| `/sports/game [@id='239']` | Matches the `game` node with the `id` number 239. The `game` is a child of `sports`, which is a child of the document root. |
| `/sports/game [name='Cricket']` | Matches all `game` nodes that contain a child element whose name is `Cricket`. The `game` is a child of `sports`, which is a child of the document root. |

**Fig. 12.35** | `XPath` expressions and descriptions.

## 12.10 RSS

*RSS* stands for ***RDF (Resource Description Framework) Site Summary*** and is also known as ***Rich Site Summary*** and ***Really Simple Syndication***. RSS is an XML format used to syndicate website content, such as news articles, blog entries, product reviews, podcasts, vodcasts and more for inclusion on other websites. An RSS feed contains an ***`rss` root element*** with a `version` attribute and a ***`channel` child element*** with ***`item` subelements***. Depending on the RSS version, the `channel` and `item` elements have certain required and optional child elements. The `item` elements provide the feed subscriber with a link to a web page or file, a title and description of the page or file. The most commonly used RSS feed versions are 0.91, 1.0, and 2.0, with RSS 2.0 being the most popular version. We discuss only RSS version 2.0 in this section.

RSS version 2.0, introduced in 2002, builds upon the RSS 0.9x versions. Version 2.0 does not contain length limitations or `item` element limitations of earlier versions, makes some formerly required elements optional, and adds new `channel` and `item` subelements. Removing length limitations on `item` descriptions allows RSS feeds to contain entire articles, blog entries and other web content. You can also have partial feeds that provide only a summary of the syndicated content. Partial feeds require the RSS subscriber to visit a website to view the complete content. RSS 2.0 allows `item` elements to contain an `enclosure` element providing the location of a media file that is related to the `item`. Such `enclosure`s enable syndication of audio and video (such as podcasts and vodcasts) via RSS feeds.

By providing up-to-date, linkable content for anyone to use, RSS enables website developers to draw more traffic. It also allows users to get news and information from many sources easily and reduces content development time. RSS simplifies importing information from portals, weblogs and news sites. Any piece of information can be syndicated via RSS, not just news. After putting information in RSS format, an RSS program, such as a feed reader or aggregator, can check the feed for changes and react to them. For more details on RSS and for links to many RSS sites, visit our RSS Resource Center at `www.deitel.com/RSS`.

### *RSS 2.0 `channel` and `item` Elements*

In RSS 2.0, the required child elements of `channel` are `description`, `link` and `title`, and the required child element of an `item` is either `title` or `description`. Figures 12.36–12.37 overview the child elements of `channel`s and `item`s, respectively.

| Element | Description |
|---|---|
| `title` | The name of the `channel` or feed. |
| `link` | The URL to the website of the `channel` or feed the RSS is coming from. |
| `description` | A description of the `channel` or feed. |
| `language` | The language the `channel` is in, using W3C language values. |
| `copyright` | The copyright material of the `channel` or feed. |
| `managingEditor` | The e-mail address of the editor of the `channel` or feed. |

**Fig. 12.36** | `channel` elements and descriptions. (Part 1 of 2.)

| Element | Description |
|---|---|
| `webMaster` | The e-mail address for the webmaster of the `channel` or feed. |
| `pubDate` | The date of the `channel` or feed release, using the RFC 822 Date and Time Specification—e.g., Sun, 14 Jan 2007 8:00:00 EST. |
| `lastBuildDate` | The last date the `channel` or feed was changed, using the RFC 822 Date and Time Specification. |
| `category` | The category (or several categories) of the `channel` or feed. This element has an optional `attribute` tag. |
| `generator` | Indicates the program that was used to generate the `channel` or feed. |
| `docs` | The URL of the documentation for the format used in the RSS file. |
| `cloud` | Specifies a SOAP web service that supports the `rssCloud` interface (`cyber.law.harvard.edu/rss/soapMeetsRss.html#rsscloudInterface`). |
| `ttl` | (Time To Live) A number of minutes for how long the `channel` or feed can be cached before refreshing from the source. |
| `image` | The GIF, JPEG or PNG image that can be displayed with the `channel` or feed. This element contains the required children `title`, `link` and `url`, and the optional children `description`, `height` and `width`. |
| `rating` | The PICS (Platform for Internet Content Selection) rating for the `channel` or feed. |
| `textInput` | Specifies a text input box to display with the `channel` or feed. This element contains the required children `title`, `name`, `link` and `description`. |
| `skipHours` | Tells aggregators which hours they can skip checking for new content. |
| `skipDays` | Tells aggregators which days they can skip checking for new content. |

**Fig. 12.36** | `channel` elements and descriptions. (Part 2 of 2.)

| Element | Description |
|---|---|
| `title` | The title of the `item`. |
| `link` | The URL of the `item`. |
| `description` | The description of the `item`. |
| `author` | The e-mail address of the author of the `item`. |
| `category` | The category (or several categories) of the `item`. This element has an optional `attribute` tag. |
| `comments` | The URL of a page for comments related to the `item`. |
| `enclosure` | The location of a media object attached to the `item`. This element has the required attributes `type`, `url` and `length`. |

**Fig. 12.37** | `item` elements and descriptions. (Part 1 of 2.)

| Element | Description |
|---|---|
| `guid` | (Globally Unique Identifier) A string that uniquely identifies the `item`. |
| `pubDate` | The date the item was published, using the RFC 822 Date and Time Specification—e.g., Sun, 14 Jan 2007 8:00:00 EST. |
| `source` | The RSS channel the `item` came from. This element has a required attribute `url`. |

**Fig. 12.37** | `item` elements and descriptions. (Part 2 of 2.)

### *Browsers and RSS Feeds*

Many of the latest web browsers can now view RSS feeds, determine whether a website offers feeds, allow you to subscribe to feeds and create feed lists. An ***RSS aggregator*** keeps tracks of many RSS feeds and brings together information from the separate feeds. There are many RSS aggregators available, including Bloglines, BottomFeeder, FeedDemon, Microsoft Internet Explorer 7, Mozilla Firefox, My Yahoo, NewsGator and Opera 9.

To allow browsers and search engines to determine whether a web page contains an RSS feed, a `link` element can be added to the `head` of a page as follows:

```
<link rel = "alternate" type = "application/rss+xml" title = "RSS"
   href = "file">
```

Many sites provide RSS feed validators. Some examples of RSS feed validators are `validator.w3.org/feed`, `feedvalidator.org`, and `www.validome.org/rss-atom/`.

### *Creating a Feed Aggregator*

The DOM and XSL can be used to create RSS aggregators. A simple RSS aggregator uses an XSL stylesheet to format RSS feeds as XHTML. Figure 12.38 loads two XML documents—an RSS feed (a small portion of which is shown in Fig. 12.39) and an XSL style sheet—then uses JavaScript to apply an XSL transformation to the RSS content and render it on the page. You'll notice as we discuss this program that there is little commonality between Internet Explorer 7 and Firefox with regard to programmatically applying XSL transformations. This is one of the reasons that JavaScript libraries have become popular in web development—they tend to hide such browser-specific issues from you. We discuss the Dojo toolkit—one of many popular JavaScript libraries—in Section 13.8. For more information on JavaScript libraries, see our JavaScript and Ajax Resource Centers (`www.deitel.com/JavaScript/` and `www.deitel.com/Ajax/`, respectively).

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 12.38: RssViewer.html -->
6  <!-- Simple RSS viewer. -->
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8  <head>
```

**Fig. 12.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 1 of 4.)

```
 9      <title>Simple RSS Viewer</title>
10      <style type = "text/css">
11         #outputDiv { font: 12px Verdana, Geneva, Arial,
12                      Helvetica, sans-serif; }
13      </style>
14      <script type = "text/javascript">
15      <!--
16      var browser = ""; // used to determine which browser is being used
17
18      // is the browser Internet Explorer 7 or Firefox?
19      if ( window.ActiveXObject ) // IE7
20         browser = "IE7";
21      else if ( document.implementation &&
22         document.implementation.createDocument ) // FF2 and other browsers
23         browser = "FF2";
24
25      // load both the RSS feed and the XSL file to process it
26      function start()
27      {
28         if ( browser == "IE7" )
29         {
30            var xsl = loadXMLDocument( 'rss.xsl' ); // load XSL file
31            var rss = loadXMLDocument( 'deitel-20.xml' ); // load RSS feed
32            var result = applyTransform( rss, xsl ); // apply transform
33            displayTransformedRss( result ); // display feed info
34         } // end if
35         else if ( browser == "FF2" )
36         {
37            var xsl = loadXMLDocument( 'rss.xsl' ); // load XSL file
38            xsl.onload = function() // function to execute when xsl loads
39            {
40               var rss = loadXMLDocument( 'deitel-20.xml' ); // load RSS feed
41               rss.onload = function() // function to execute when rss loads
42               {
43                  var result = applyTransform( rss, xsl ); // apply transform
44                  displayTransformedRss( result ); // display feed info
45               } // end onload event handler for rss
46            } // end onload event handler for xsl
47         } // end else
48      } // end function start
49
50      // load XML document based on whether the browser is IE7 or Firefox
51      function loadXMLDocument( url )
52      {
53         var doc = ""; // variable to manage loading file
54
55         if ( browser == "IE7" ) // IE7
56         {
57            // create IE7-specific XML document object
58            doc = new ActiveXObject( "Msxml2.DOMDocument.6.0" );
59            doc.async = false; // specifies synchronous loading of XML doc
60            doc.load( url ); // load the XML document specified by url
61         } // end if
```

**Fig. 12.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 2 of 4.)

```
62          else if ( browser == "FF2" ) // other browsers
63          {
64             // create XML document object
65             doc = document.implementation.createDocument( "", "", null );
66             doc.load( url ); // load the XML document specified by url
67          } // end else
68          else // not supported
69             alert( 'This script is not supported by your browser' );
70
71          return doc; // return the loaded document
72       } // end function loadXMLDocument
73
74       // apply XSL transformation and show results
75       function applyTransform( rssDocument, xslDocument )
76       {
77          var result; // stores transformed RSS
78
79          // transform the RSS feed to XHTML
80          if ( browser == "IE7" )
81             result = rssDocument.transformNode( xslDocument );
82          else // browser == "FF2"
83          {
84             // create Firefox object to perform transformation
85             var xsltProcessor = new XSLTProcessor();
86
87             // specify XSL stylesheet to use in transformation
88             xsltProcessor.importStylesheet( xslDocument );
89
90             // apply the transformation
91             result =
92                xsltProcessor.transformToFragment( rssDocument, document );
93          } // end else
94
95          return result; // return the transformed RSS
96       } // end function applyTransform
97
98       // display the XML document and highlight the first child
99       function displayTransformedRss( resultXHTML )
100      {
101         if ( browser == "IE7" )
102            document.getElementById( "outputDiv" ).innerHTML = resultXHTML;
103         else // browser == "FF2"
104            document.getElementById( "outputDiv" ).appendChild(
105               resultXHTML );
106      } // end function displayTransformedRss
107      // -->
108      </script>
109   </head>
110   <body id = "body" onload = "start();">
111      <div id = "outputDiv"></div>
112   </body>
113   </html>
```

**Fig. 12.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 3 of 4.)

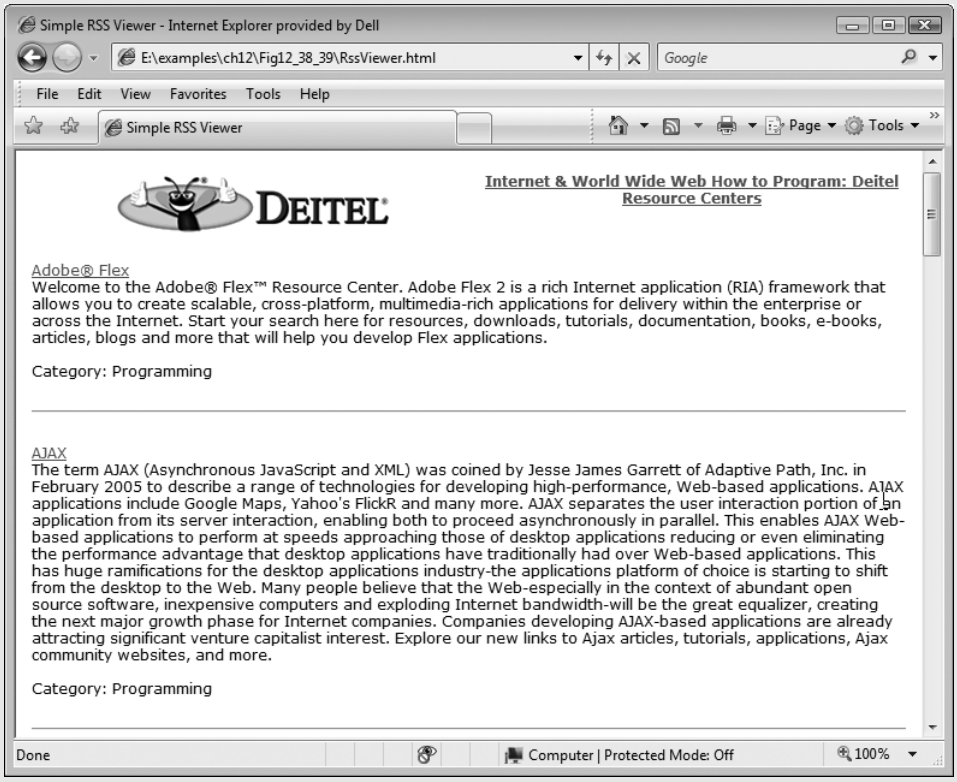



**Fig. 12.38** | Rendering an RSS feed in a web page using XSLT and JavaScript. (Part 4 of 4.)

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <!-- Fig. 12.39: deitel-20.xml -->
4  <!-- RSS 2.0 feed of Deitel Resource Centers -->
5  <rss version="2.0">
6     <channel>
7        <title>
8           Internet &amp;#38; World Wide Web How to Program:
9           Deitel Resource Centers
10       </title>
11       <link>http://www.deitel.com/ResourceCenters.html</link>
12       <description>
13          Check out our growing network of Resource Centers that focus on
14          many of today's hottest programming, Web 2.0 and technology
15          topics. Start your search here for downloads, tutorials,
16          documentation, books, e-books, blogs, RSS feeds, journals,
17          articles, training, webcasts, podcasts, videos and more.
18       </description>
```

**Fig. 12.39** | RSS 2.0 sample feed. (Part 1 of 2.)

```
19         <languague>en-us</languague>
20         <image>
21            <url>
22               http://www.deitel.com/Portals/0/deitel_transparent_smaller.png
23            </url>
24            <title>Deitel.com</title>
25            <link>http://www.deitel.com/</link>
26         </image>
27
28         <item>
29            <title>Adobe® Flex</title>
30            <link>http://www.deitel.com/Flex/</link>
31            <description>
32               <p>
33               Welcome to the Adobe® Flex™ Resource Center. Adobe Flex 2 is a
34               rich Internet application (RIA) framework that allows you to
35               create scalable, cross-platform, multimedia-rich applications
36               for delivery within the enterprise or across the Internet.
37               Start your search here for resources, downloads, tutorials,
38               documentation, books, e-books, articles, blogs and more that
39               will help you develop Flex applications.
40               </p>
41            </description>
42            <category>Programming</category>
43         </item>
44      </channel>
45   </rss>
```

**Fig. 12.39** | RSS 2.0 sample feed. (Part 2 of 2.)

### *Determining the Browser Type and Loading the Documents*

When this page first loads, lines 19–23 (Fig. 12.38) determine whether the browser is Internet Explorer 7 or Firefox and store the result in variable `browser` for use throughout the script. After the body of this XHTML document loads, its `onload` event calls function `start` (lines 26–48) to load RSS and XSL files as XML documents, and to transform the RSS. Since Internet Explorer 7 can download the files synchronously, lines 30–33 perform the loading, transformation and display steps sequentially. As mentioned previously, Firefox loads the files asynchronously. For this reason, line 37 starts loading the `rss.xsl` document (included with this example's code), and lines 38–46 register an `onload` event handler for that document. When the document finishes loading, line 40 begins loading the `deitel-20.xml` RSS document. Lines 41–45 register an `onload` event handler for this second document. When it finishes loading, lines 43–44 perform the transformation and display the results.

### *Transforming the RSS to XHTML*

Function `applyTransform` (Fig. 12.38, lines 75–96) performs the browser-specific XSL transformations using the RSS document and XSL document it receives as arguments. Line 81 uses the MSXML object's built-in XSLT capabilities to apply the transformations. Method ***`transformNode`*** is invoked on the `rssDocument` object and receives the `xslDocument` object as an argument.

Firefox provides built-in XSLT processing in the form of the **`XSLTProcessor`** object (created at line 85). After creating this object, you use its **`importStylesheet`** method to specify the XSL stylesheet you'd like to apply (line 88). Finally, lines 91–92 apply the transformation by invoking the `XSLTProcessor` object's **`transformToFragment`** method, which returns a document fragment—i.e., a piece of a document. In our case, the `rss.xsl` document transforms the RSS into an XHTML `table` element that we'll append to the `outputDiv` element in our XHTML page. The arguments to `transformToFragment` are the document to transform and the document object to which the transformed fragment will belong. To learn more about `XSLTProcessor`, visit `developer.mozilla.org/en/docs/The_XSLT/JavaScript_Interface_in_Gecko`.

In each browser's case, after the transformation, the resulting XHTML markup is assigned to variable `result` and returned from function `applyTransform`. Then function `displayTransformedRss` is called.

### *Displaying the XHTML Markup*

Function `displayTransformedRss` (lines 99–106) displays the transformed RSS in the `outputDiv` element (line 111 in the `body`). In both Internet Explorer 7 and Firefox, we use the DOM method `getElementById` to obtain the `outputDiv` element. In Internet Explorer 7, the node's `innerHTML` property is used to add the table as a child of the `outputDiv` element (line 102). In Firefox, the node's `appendChild` method must be used to append the table (a document fragment) to the `outputDiv` element.

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