Chapter 15
XML

15.1 Introduction

- XML is a portable, widely supported, open (i.e., nonproprietary) technology for data storage and exchange

15.2 XML Basics

- XML permits document authors to create markup for virtually any type of information
  - Can create entirely new markup languages that describe specific types of data, including mathematical formulas, chemical molecular structures, music and recipes
- XML describes data in a way that human beings can understand and computers can process.
- An XML parser is responsible for identifying components of XML documents (typically files with the `.xml` extension) and then storing those components in a data structure for manipulation
- An XML document can reference a Document Type Definition (DTD) or schema that defines the document's proper structure
- An XML document that conforms to a DTD/schema (i.e., has the appropriate structure) is valid
- If an XML parser (validating or non-validating) can process an XML document successfully, that XML document is well-formed

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

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

15.3 Structuring Data

- An XML document begins with an optional XML declaration, which identifies the document as an XML document. The `version` attribute specifies the version of XML syntax used in the document.
- XML comments begin with `<!--` and end with `-->`
- An XML document contains text that represents its content (i.e., data) and elements that specify its structure. XML documents delimit an element with start and end tags
- The root element of an XML document encompasses all its other elements
- XML element names can be of any length and can contain letters, digits, underscores, hyphens and periods
  - Must begin with either a letter or an underscore, and they should not begin with "`xml`" in any combination of uppercase and lowercase letters, as this is reserved for use in the XML standards

15.3 Structuring Data (Cont.)

- When a user loads an XML document in a browser, a parser parses the document, and the browser uses a style sheet to format the data for display
- Google Chrome places a down arrow and right arrow next to every container element; they're not part of the XML document.
  - down arrow indicates that the browser is displaying the container element's child elements
  - clicking the right arrow next to an element expands that element

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.2: article.xml -->
4  <!-- Article structured with XML -->
5  <article>
6     <title>Simple XML</title>
7     <date>July 4, 2007</date>
8     <author>
9        <firstName>John</firstName>
10       <lastName>Doe</lastName>
11    </author>
12    <summary>XML is pretty easy.</summary>
13    <content>This chapter presents examples that use XML.</content>
14 </article>
```

Fig. 15.2 | XML used to mark up an article.

a) `article.xml` with all elements expanded

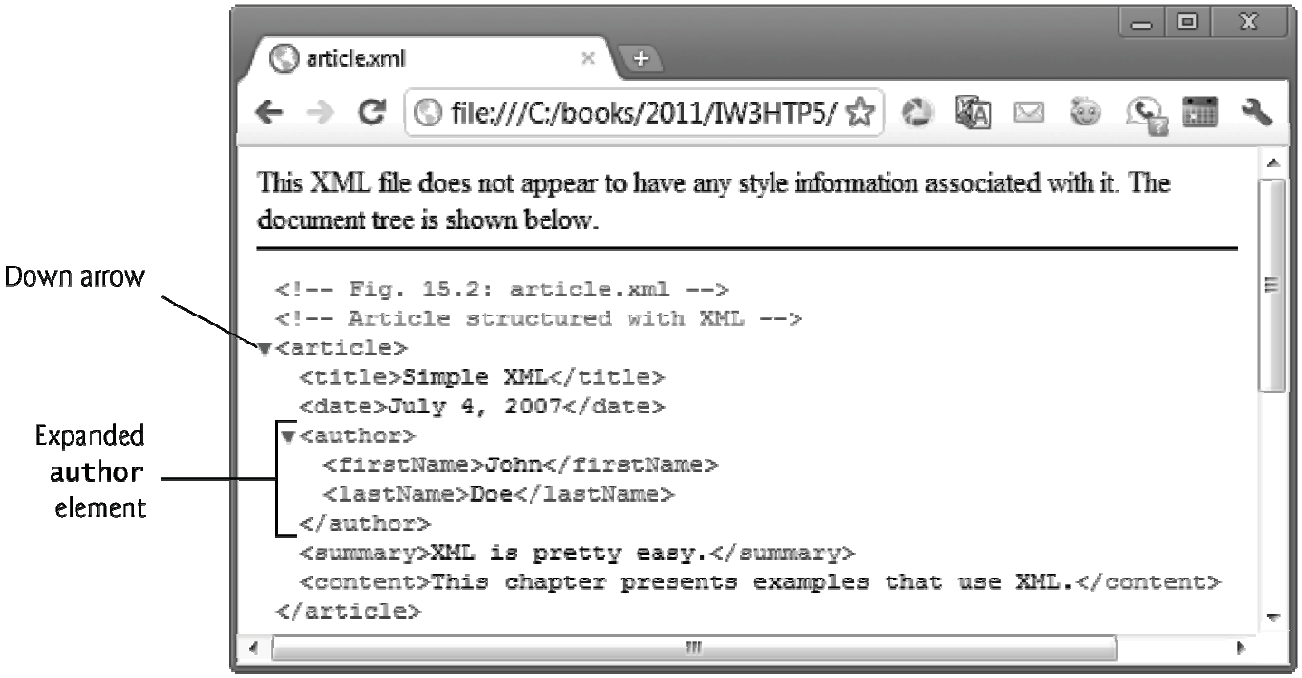


Fig. 15.3 | `article.xml` displayed in the Google Chrome browser. (Part 1 of 2.)

b) `article.xml` with the `author` element collapsed

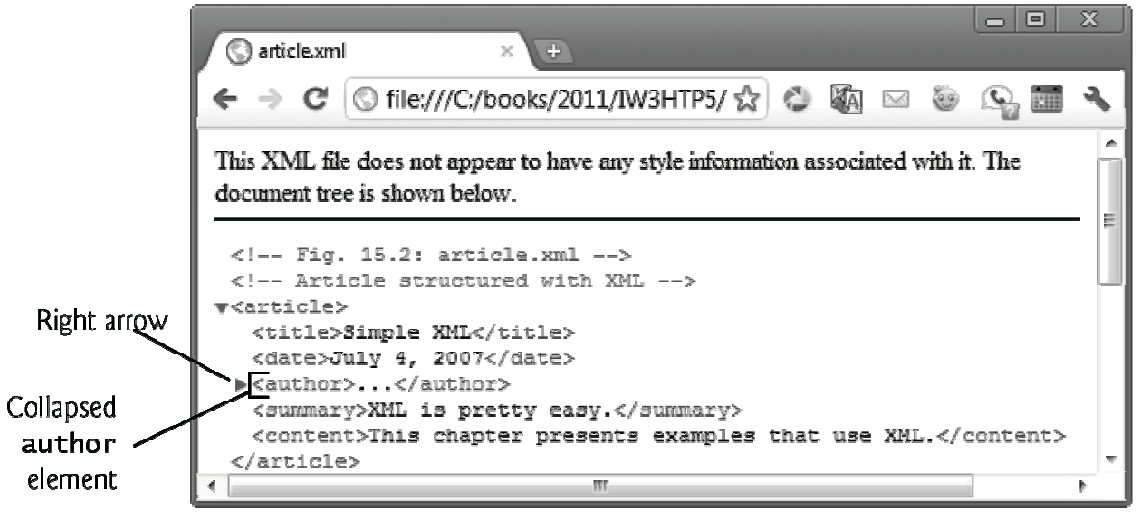


Fig. 15.3 | `article.xml` displayed in the Google Chrome browser. (Part 2 of 2.)

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

<!-- Fig. 15.4: letter.xml -->
<!-- Business letter marked up with XML -->
<!DOCTYPE letter SYSTEM "letter.dtd">

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

```

Fig. 15.4 | Business letter marked up with XML. (Part 1 of 2.)

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

Fig. 15.4 | Business letter marked up with XML. (Part 2 of 2.)

15.4 Namespaces

- XML namespaces provide a means for document authors to prevent naming collisions
- Each namespace prefix is bound to a uniform resource identifier (URI) that uniquely identifies the namespace
  - A URI is a series of characters that differentiate names
  - Document authors create their own namespace prefixes
  - Any name can be used as a namespace prefix, but the namespace prefix `xml` is reserved for use in XML standards
- To eliminate the need to place a namespace prefix in each element, authors can specify a default namespace for an element and its children
  - We declare a default namespace using keyword `xmlns` with a URI (Uniform Resource Identifier) as its value
- Document authors commonly use URLs (Uniform Resource Locators) for URIs, because domain names (e.g., `deitel.com`) in URLs must be unique

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.5: namespace.xml -->
4  <!-- Demonstrating namespaces -->
5  <text:directory
6     xmlns:text = "urn:deitel:textInfo"
7     xmlns:image = "urn:deitel:imageInfo">
8
9     <text:file filename = "book.xml">
10       <text:description>A book list</text:description>
11    </text:file>
12
13    <image:file filename = "funny.jpg">
14       <image:description>A funny picture</image:description>
15       <image:size width = "200" height = "100" />
16    </image:file>
17 </text:directory>
```

Fig. 15.5 | XML namespaces demonstration.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.6: defaultnamespace.xml -->
4  <!-- Using default namespaces -->
5  <directory xmlns = "urn:deitel:textInfo"
6     xmlns:image = "urn:deitel:imageInfo">
7
8     <file filename = "book.xml">
9        <description>A book list</description>
10    </file>
11
12    <image:file filename = "funny.jpg">
13       <image:description>A funny picture</image:description>
14       <image:size width = "200" height = "100" />
15    </image:file>
16 </directory>
```

Fig. 15.6 | Default namespace demonstration.

15.5 Document Type Definitions (DTDs)

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

```
<!-- Fig. 15.7: letter.dtd       -->
<!-- DTD document for letter.xml -->

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

<!ELEMENT contact ( name, address1, address2, city, state,
   zip, phone, flag )>
<!ATTLIST contact type CDATA #IMPLIED>

<!ELEMENT name ( #PCDATA )>
<!ELEMENT address1 ( #PCDATA )>
<!ELEMENT address2 ( #PCDATA )>
<!ELEMENT city ( #PCDATA )>
<!ELEMENT state ( #PCDATA )>
<!ELEMENT zip ( #PCDATA )>
<!ELEMENT phone ( #PCDATA )>
<!ELEMENT flag EMPTY>
<!ATTLIST flag gender (M | F) "M">

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

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

15.6 W3C XML Schema Documents

- Unlike DTDs
  - Schemas use use XML syntax not EBNF grammar
  - XML Schema documents can specify what type of data (e.g., numeric, text) an element can contain
- An XML document that conforms to a schema document is schema valid
- Two categories of types exist in XML Schema: simple types and complex types
  - Simple types cannot contain attributes or child elements; complex types can
- Every simple type defines a restriction on an XML Schema-defined schema type or on a user-defined type
- Complex types can have either simple content or complex content
  - Both can contain attributes, but only complex content can contain child elements
- Whereas complex types with simple content must extend or restrict some other existing type, complex types with complex content do not have this limitation

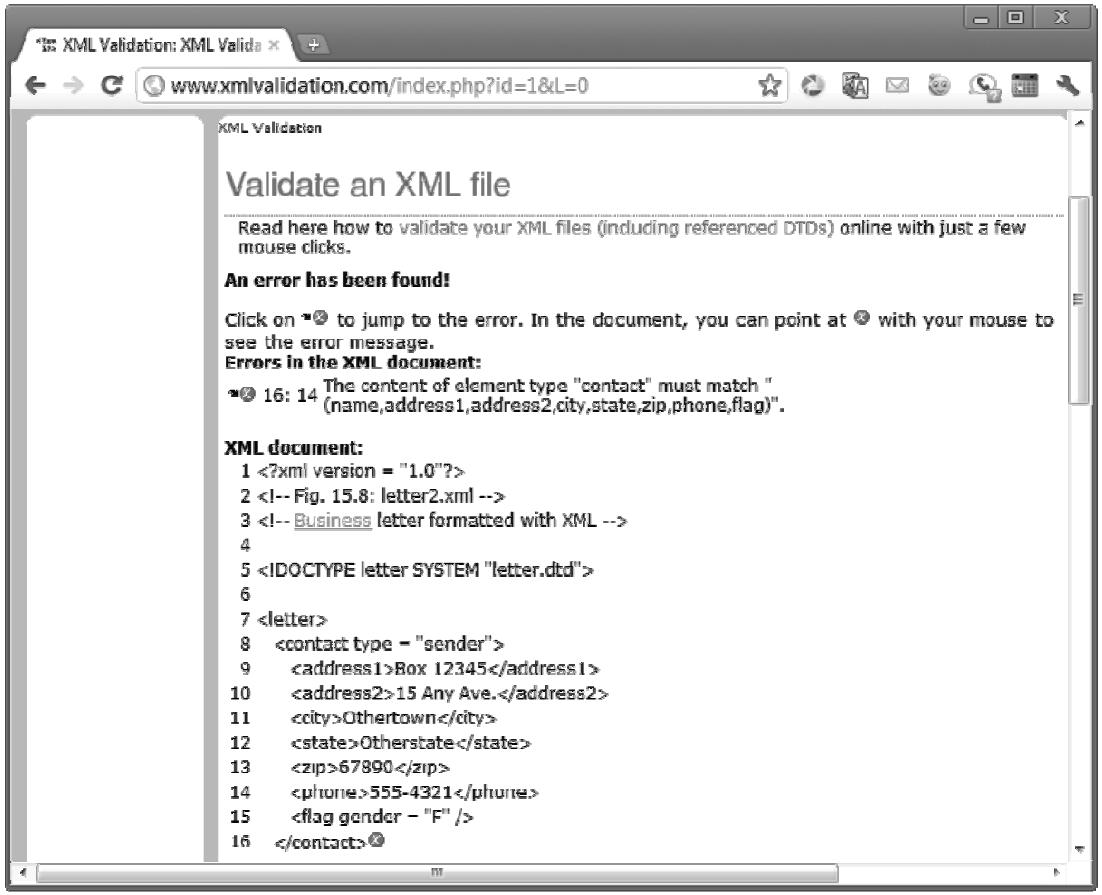


Fig. 15.8 | Error message when validating `letter.xml` with a missing contact name.

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

<!-- Fig. 15.9: book.xml -->
<!-- Book list marked up as XML -->
<deitel:books xmlns:deitel = "http://www.deitel.com/booklist">
   <book>
      <title>Visual Basic 2010 How to Program</title>
   </book>
   <book>
      <title>Visual C# 2010 How to Program, 4/e</title>
   </book>
   <book>
      <title>Java How to Program, 9/e</title>
   </book>
   <book>
      <title>C++ How to Program, 8/e</title>
   </book>
   <book>
      <title>Internet and World Wide Web How to Program, 5/e</title>
   </book>
</deitel:books>
```

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

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.10: book.xsd           -->
4  <!-- Simple W3C XML Schema document -->
5  <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6     xmlns:deitel = "http://www.deitel.com/booklist"
7     targetNamespace = "http://www.deitel.com/booklist">
8
9     <element name = "books" type = "deitel:BooksType"/>
10
11    <complexType name = "BooksType">
12       <sequence>
13          <element name = "book" type = "deitel:SingleBookType"
14             minOccurs = "1" maxOccurs = "unbounded"/>
15       </sequence>
16    </complexType>
17
18    <complexType name = "SingleBookType">
19       <sequence>
20          <element name = "title" type = "string"/>
21       </sequence>
22    </complexType
23 </schema>
```

Fig. 15.10 | XML Schema document for book.xml. (Part 1 of 2.)

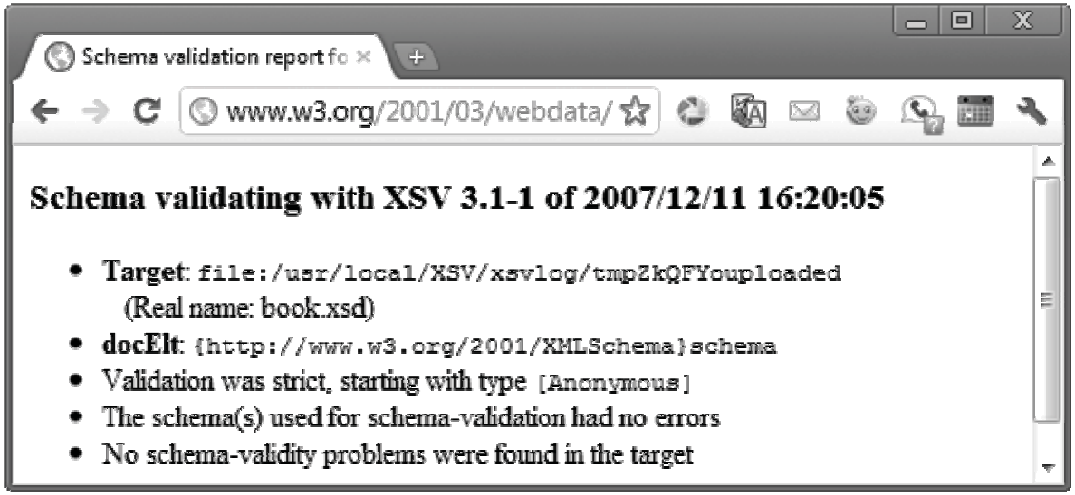


Fig. 15.10 | XML Schema document for book.xml. (Part 2 of 2.)

Type	Description	Range or structure	Examples
`string`	A character string		`"hello"`
`boolean`	True or false	`true`, `false`	`true`
`decimal`	A decimal numeral	`i * (10^n)`, where `i` is an integer and `n` is an integer that's less than or equal to zero.	`5`, `-12`, `-45.78`
`float`	A floating-point number	`m * (2^e)`, where `m` is an integer whose absolute value is less than 2^{24} and `e` is an integer in the range `-149` to `104`. Plus three additional numbers: positive infinity, negative infinity and not-a-number (`NaN`).	`0`, `12`, `-109.375`, `NaN`
`double`	A floating-point number	`m * (2^e)`, where `m` is an integer whose absolute value is less than 2^{53} and `e` is an integer in the range `-1075` to `970`. Plus three additional numbers: positive infinity, negative infinity and not-a-number (`NaN`).	`0`, `12`, `-109.375`, `NaN`

Fig. 15.11 | Some XML Schema types. (Part 1 of 2.)

Type	Description	Range or structure	Examples
`long`	A whole number	`-9223372036854775808` to `9223372036854775807`, inclusive.	`1234567890`, `-1234567890`
`int`	A whole number	`-2147483648` to `2147483647`, inclusive.	`1234567890`, `-1234567890`
`short`	A whole number	`-32768` to `32767`, inclusive.	`12`, `-345`
`date`	A date consisting of a year, month and day	`yyyy-mm` with an optional `dd` and an optional time zone, where `yyyy` is four digits long and `mm` and `dd` are two digits long.	`2005-05-10`
`time`	A time consisting of hours, minutes and seconds	`hh:mm:ss` with an optional time zone, where `hh`, `mm` and `ss` are two digits long.	`16:30:25-05:00`

Fig. 15.11 | Some XML Schema types. (Part 2 of 2.)

```
1  <?xml version = "1.0"?>
2  <!-- Fig. 15.12: computer.xsd -->
3  <!-- W3C XML Schema document  -->
4
5  <schema xmlns = "http://www.w3.org/2001/XMLSchema"
6     xmlns:computer = "http://www.deitel.com/computer"
7     targetNamespace = "http://www.deitel.com/computer">
8
9     <simpleType name = "gigahertz">
10       <restriction base = "decimal">
11          <minInclusive value = "2.1"/>
12       </restriction>
13    </simpleType>
14
15    <complexType name = "CPU">
16       <simpleContent>
17          <extension base = "string">
18             <attribute name = "model" type = "string"/>
19          </extension>
20       </simpleContent>
21    </complexType>
22
```

Fig. 15.12 | XML Schema document defining simple and complex types. (Part 1 of 2.)

```
23      <complexType name = "portable">
24         <all>
25            <element name = "processor" type = "computer:CPU"/>
26            <element name = "monitor" type = "int"/>
27            <element name = "CPUSpeed" type = "computer:gigahertz"/>
28            <element name = "RAM" type = "int"/>
29         </all>
30         <attribute name = "manufacturer" type = "string"/>
31      </complexType>
32
33      <element name = "laptop" type = "computer:portable"/>
34   </schema>
```

Fig. 15.12 | XML Schema document defining simple and complex types. (Part 2 of 2.)

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. 15.13: laptop.xml                -->
4  <!-- Laptop components marked up as XML -->
5  <computer:laptop xmlns:computer = "http://www.deitel.com/computer"
6     manufacturer = "IBM">
7
8     <processor model = "Centrino">Intel</processor>
9     <monitor>17</monitor>
10    <CPUSpeed>2.4</CPUSpeed>
11    <RAM>256</RAM>
12 </computer:laptop>
```

Fig. 15.13 | XML document using the `laptop` element defined in `computer.xsd`.

15.7 XML Vocabularies

- Some XML vocabularies
  - MathML (Mathematical Markup Language)
  - Scalable Vector Graphics (SVG)
  - Wireless Markup Language (WML)
  - Extensible Business Reporting Language (XBRL)
  - Extensible User Interface Language (XUL)
  - Product Data Markup Language (PDML)
  - W3C XML Schema
  - Extensible Stylesheet Language (XSL)
- MathML markup describes mathematical expressions for display
  - Divided into two types of markup—content markup and presentation markup
  - Content MathML allows programmers to write mathematical notation specific to different areas of mathematics
  - Presentation MathML is directed toward formatting and displaying mathematical notation
  - By convention, MathML files end with the `.mml` filename extension

15.7 XML Vocabularies (Cont.)

- MathML document root node is the `math` element
  - Default namespace is `http:// www.w3.org/1998/Math/MathML`
- `mn` element
  - marks up a number
- `mo` element
  - marks up an operator
- Entity reference `&InvisibleTimes;`
  - indicates a multiplication operation without explicit symbolic representation
- `msup` element
  - represents a superscript
  - 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, use identifier element `mi`

15.7 XML Vocabularies (Cont.)

- `mfrac` element
  - displays a fraction
  - If either the numerator or the denominator contains more than one element, it must appear in an `mrow` element
- `mrow` element
  - groups elements that are positioned horizontally in an expression
- Entity reference `&int;`
  - represents the integral symbol
- `msubsup` element
  - specifies the subscript and superscript of a symbol
  - Requires three child elements—an operator, the subscript expression and the superscript expression
- `msqrt` element
  - represents a square-root expression
- Entity reference `&delta;`
  - represents a lowercase delta symbol

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

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

```
1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
3     "http://www.w3.org/TR/MathML2/dtd/mathml2.dtd">
4
5  <!-- Fig. 15.15: mathml2.html -->
6  <!-- MathML algebraic equation. -->
7  <math xmlns="http://www.w3.org/1998/Math/MathML">
8     <mn>3</mn>
9     <mo>&InvisibleTimes;</mo>
10    <msup>
11       <mi>x</mi>
12       <mn>2</mn>
13    </msup>
14    <mo>+</mo>
15    <mn>x</mn>
16    <mo>&minus;</mo>
17    <mfrac>
18       <mn>2</mn>
19       <mi>x</mi>
20    </mfrac>
21    <mo>=</mo>
22    <mn>0</mn>
23 </math>
```

Fig. 15.15 | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 1 of 2.)

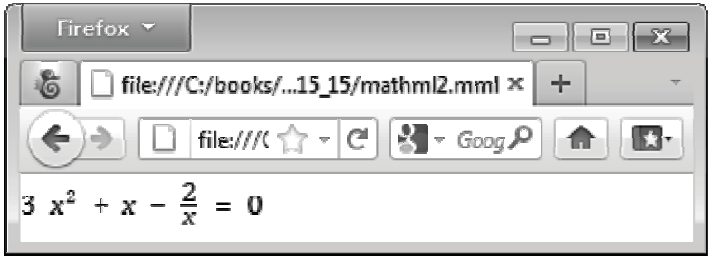


Fig. 15.15 | Algebraic equation marked up with MathML and displayed in the Firefox browser. (Part 2 of 2.)

```
<?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. 15.16 mathml3.html -->
<!-- Calculus example using MathML -->
<math xmlns="http://www.w3.org/1998/Math/MathML">
   <mrow>
      <msubsup>
         <mo>&int;</mo>
         <mn>0</mn>
         <mrow>
            <mn>1</mn>
            <mo>&minus;</mo>
            <mi>y</mi>
         </mrow>
      </msubsup>
```

Fig. 15.16 | Calculus expression marked up with MathML and displayed in the Firefox browser. (Part 1 of 3.)

```
18          <msqrt>
19             <mn>4</mn>
20             <mo>&InvisibleTimes;</mo>
21             <msup>
22                <mi>x</mi>
23                <mn>2</mn>
24             </msup>
25             <mo>+</mo>
26             <mi>y</mi>
27          </msqrt>
28          <mo>&delta;</mo>
29          <mi>x</mi>
30       </mrow>
31    </math>
```

Fig. 15.16 | Calculus expression marked up with MathML and displayed in the Firefox browser. (Part 2 of 3.)

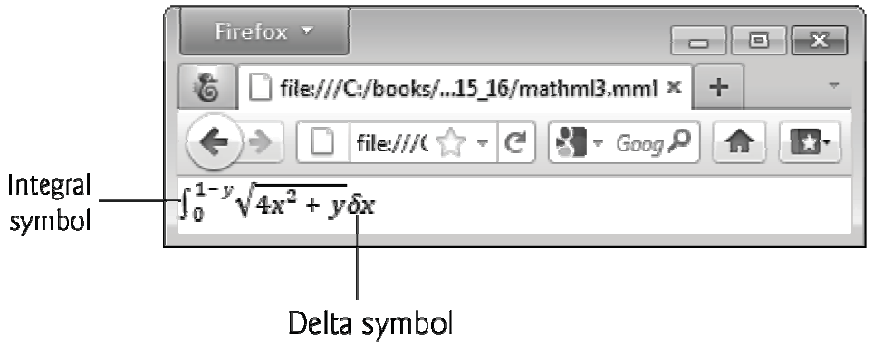


Fig. 15.16 | Calculus expression marked up with MathML and displayed in the Firefox browser. (Part 3 of 3.)

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.

Fig. 15.17 | Various markup languages derived from XML. (Part 1 of 2.)

Markup language	Description
Research Information Exchange Markup Language (RIXML)	RIXML, developed by a consortium of brokerage firms, marks up investment data. Visit `www.rixml.org` for more information on RIXML.
Geography Markup Language (GML)	OpenGIS developed the Geography Markup Language to describe geographic information. Visit `www.opengis.org` for more information on GML.
Extensible User Interface Language (XUL)	The Mozilla Project created the Extensible User Interface Language for describing graphical user interfaces in a platform-independent way.

Fig. 15.17 | Various markup languages derived from XML. (Part 2 of 2.)

15.8 Extensible Stylehsheet Language and XSL Transformations

- Convert XML into any text-based document
- XSL documents have the extension `.xsl`
- XPath
  - A string-based language of expressions used by XML and many of its related technologies for effectively and efficiently locating structures and data (such as specific elements and attributes) in XML documents
  - 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.
- XSLT does not analyze every node of the source tree
  - it selectively navigates the source tree using XPath's `select` and `match` attributes
- For XSLT to function, the source tree must be properly structured
  - Schemas, DTDs and validating parsers can validate document structure before using XPath and XSLTs
- XSL style sheets can be connected directly to an XML document by adding an `xml:stylesheet` processing instruction to the XML document

15.8 Extensible Stylehsheet Language and XSL Transformations (Cont.)

- Two tree structures are involved in transforming an XML document using XSLT
  - source tree (the document being transformed)
  - result tree (the result of the transformation)
- XPath character / (a forward slash)
  - Selects the document root
  - In XPath, a leading forward slash specifies that we are using absolute addressing
  - An XPath expression with no beginning forward slash uses relative addressing
- XSL element `value-of`
  - Retrieves an attribute's value
  - The @ symbol specifies an attribute node
- XSL node-set function `name`
  - Retrieves the current node's element name
- XSL node-set function `text`
  - Retrieves the text between an element's start and end tags
- The XPath expression `//*`
  - Selects all the nodes in an XML document

```
<?xml version = "1.0"?>
<?xml-stylesheet type = "text/xsl" href = "sports.xsl"?>

<!-- Fig. 15.18: sports.xml -->
<!-- Sports Database -->

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

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

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

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

```

Fig. 15.18 | XML document that describes various sports. (Part 1 of 2.)

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

```
1  <?xml version = "1.0"?>
2  <!-- Fig. 15.19: sports.xsl -->
3  <!-- A simple XSLT transformation -->
4
5  <!-- reference XSL style sheet URI -->
6  <xsl-stylesheet version = "1.0"
7     xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
8
9     <xsl:output method = "html" doctype-system = "about:legacy-compat" />
10    <xsl:template match = "/"> <!-- match root element -->
11
12    <html xmlns = "http://www.w3.org/1999/xhtml">
13       <head>
14          <meta charset = "utf-8"/>
15          <link rel = "stylesheet" type = "text/css" href = "style.css"/>
16          <title>Sports</title>
17       </head>
18
19       <body>
20          <table>
21          <caption>Information about various sports</caption>
22             <thead>
23                <tr>
```

Fig. 15.19 | XSLT that creates elements and attributes in an HTML5 document. (Part 1 of 2.)

```
                  <th>ID</th>
                  <th>Sport</th>
                  <th>Information</th>
               </tr>
            </thead>

            <!-- insert each name and paragraph element value -->
            <!-- into a table row. -->
            <xsl:for-each select = "/sports/game">
               <tr>
                  <td><xsl:value-of select = "@id"/></td>
                  <td><xsl:value-of select = "name"/></td>
                  <td><xsl:value-of select = "paragraph"/></td>
               </tr>
            </xsl:for-each>
         </table>
      </body>
   </html>

   </xsl:template>
</xsl:stylesheet>
```

Fig. 15.19 | XSLT that creates elements and attributes in an HTML5 document. (Part 2 of 2.)

```
<?xml version = "1.0"?>
<?xml-stylesheet type = "text/xsl" href = "sorting.xsl"?>

<!-- Fig. 15.20: 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>

```

Fig. 15.20 | XML document containing book information. (Part 1 of 2.)

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

Fig. 15.20 | XML document containing book information. (Part 2 of 2.)

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

<!-- Fig. 15.21: sorting.xsl -->
<!-- Transformation of book information into HTML5 -->
<xsl:stylesheet version = "1.0"
   xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

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

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

```

Fig. 15.21 | XSL document that transforms `sorting.xml` into HTML5. (Part 1 of 5.)

```
18      <!-- match book -->
19      <xsl:template match = "book">
20         <head>
21            <meta charset = "utf-8"/>
22            <link rel = "stylesheet" type = "text/css" href = "style.css"/>
23            <title>ISBN <xsl:value-of select = "@isbn"/> -
24               <xsl:value-of select = "title"/></title>
25         </head>
26
27         <body>
28            <h1><xsl:value-of select = "title"/></h1>
29            <h2>by
30               <xsl:value-of select = "author/lastName"/>,
31               <xsl:value-of select = "author/firstName"/></h2>
32
33            <table>
34
35               <xsl:for-each select = "chapters/frontMatter/*">
36                  <tr>
37                     <td>
38                        <xsl:value-of select = "name()"/>
39                     </td>
40
```

Fig. 15.21 | XSL document that transforms `sorting.xml` into HTML5. (Part 2 of 5.)

```
41                      <td>
42                         ( <xsl:value-of select = "@pages"/> pages )
43                      </td>
44                   </tr>
45                </xsl:for-each>
46
47                <xsl:for-each select = "chapters/chapter">
48                   <xsl:sort select = "@number" data-type = "number"
49                        order = "ascending"/>
50                   <tr>
51                      <td>
52                         Chapter <xsl:value-of select = "@number"/>
53                      </td>
54
55                      <td>
56                         <xsl:value-of select = "text()"/>
57                         ( <xsl:value-of select = "@pages"/> pages )
58                      </td>
59                   </tr>
60                </xsl:for-each>
61
```

Fig. 15.21 | XSL document that transforms `sorting.xml` into HTML5. (Part 3 of 5.)

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

Fig. 15.21 | XSL document that transforms `sorting.xml` into HTML5. (Part 4 of 5.)

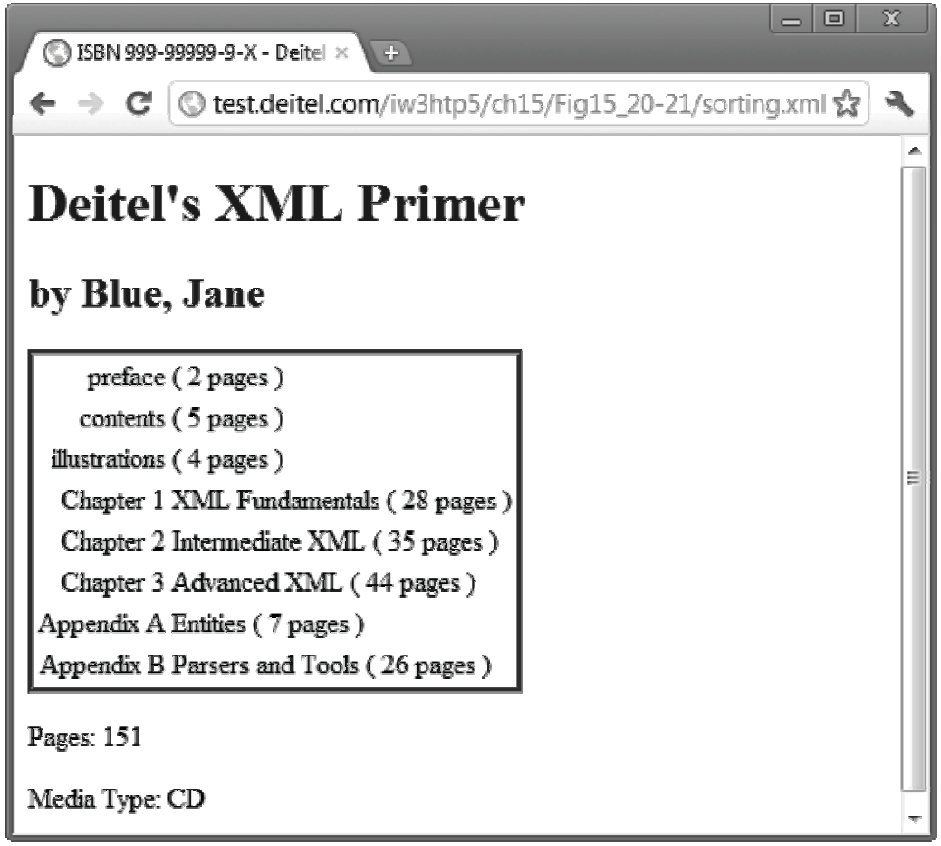


Fig. 15.21 | XSL document that transforms `sorting.xml` into HTML5. (Part 5 of 5.)

Element	Description
`<xsl:apply-templates>`	Applies the templates of the XSL document to the children of the current node.
`<xsl:apply-templates match = "`*expression*`">`	Applies the templates of the XSL document to the children of *expression*. The value of the attribute match (i.e., *expression*) must be an XPath expression that specifies elements.
`<xsl:template>`	Contains rules to apply when a specified node is matched.
`<xsl:value-of select = "`*expression*`">`	Selects the value of an XML element and adds it to the output tree of the transformation. The required `select` attribute contains an XPath expression.
`<xsl:for-each select = "`*expression*`">`	Applies a template to every node selected by the XPath specified by the `select` attribute.
`<xsl:sort select = "`*expression*`">`	Used as a child element of an `<xsl:apply-templates>` or `<xsl:for-each>` element. Sorts the nodes selected by the `<xsl:apply-template>` or `<xsl:for-each>` element so that the nodes are processed in sorted order.

Fig. 15.22 | XSL style-sheet elements. (Part 1 of 2.)

Element	Description
`<xsl:output>`	Has various attributes to define the format (e.g., XML), version (e.g., 1.0, 2.0), document type and media type of the output document. This tag is a top-level element—it can be used only as a child element of an `xml:stylesheet`.
`<xsl:copy>`	Adds the current node to the output tree.

Fig. 15.22 | XSL style-sheet elements. (Part 2 of 2.)

15.9 Document Object Model

- Retrieving data from an XML document using traditional sequential file processing techniques is neither practical nor efficient
- Some XML parsers store document data as tree structures in memory
  - 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 is represented by a node
  - A node that contains other nodes is called a parent node
  - A parent node can have many children, but a child node can have only one parent node
  - Nodes that are peers 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

15.9 Document Object Model (Cont.)

- Many of the XML DOM capabilities are similar or identical to those of the HTML DOM
- The DOM tree has a single root node, which contains all the other nodes in the document
- `XMLHttpRequest` object
  - can be used to load an XML document.
  - Typically, such an object is used with Ajax to make asynchronous requests to a server—the topic of the next chapter.
- `XMLHttpRequest` object's `open` method is used to create a `get` request for an XML document at a specified URL.
- The argument `null` to the `send` method indicates that no data is being sent to the server as part of this request.
- `nodeType` property of a node
  - contains the type of the node
- Nonbreaking spaces (` `)
  - spaces that the browser is not allowed to collapse or that can be used to keep words together.

15.9 Document Object Model (Cont.)

- `nodeName` property of a node
  - Obtain the name of an element
- `childNodes` list of a node
  - Nonzero if the currrent node has children
- `nodeValue` property
  - Returns the value of an element
- `firstChild` property of a node
  - Refers to the first child of a given node
- `lastChild` property of a node
  - refers to the last child of a given node
- `nextSibling` property of a node
  - refers to the next sibling in a list of children of a particular node.
- `previousSibling` property of a node
  - refers to the current node's previous sibling
- `parentNode` property of a node
  - refers to the current node's parent node

15.9 Document Object Model (Cont.)

- Use XPath expressions to specify search criteria
- When the user clicks the Get Matches button, the script applies the XPath expression to the XML DOM and displays the matching nodes.

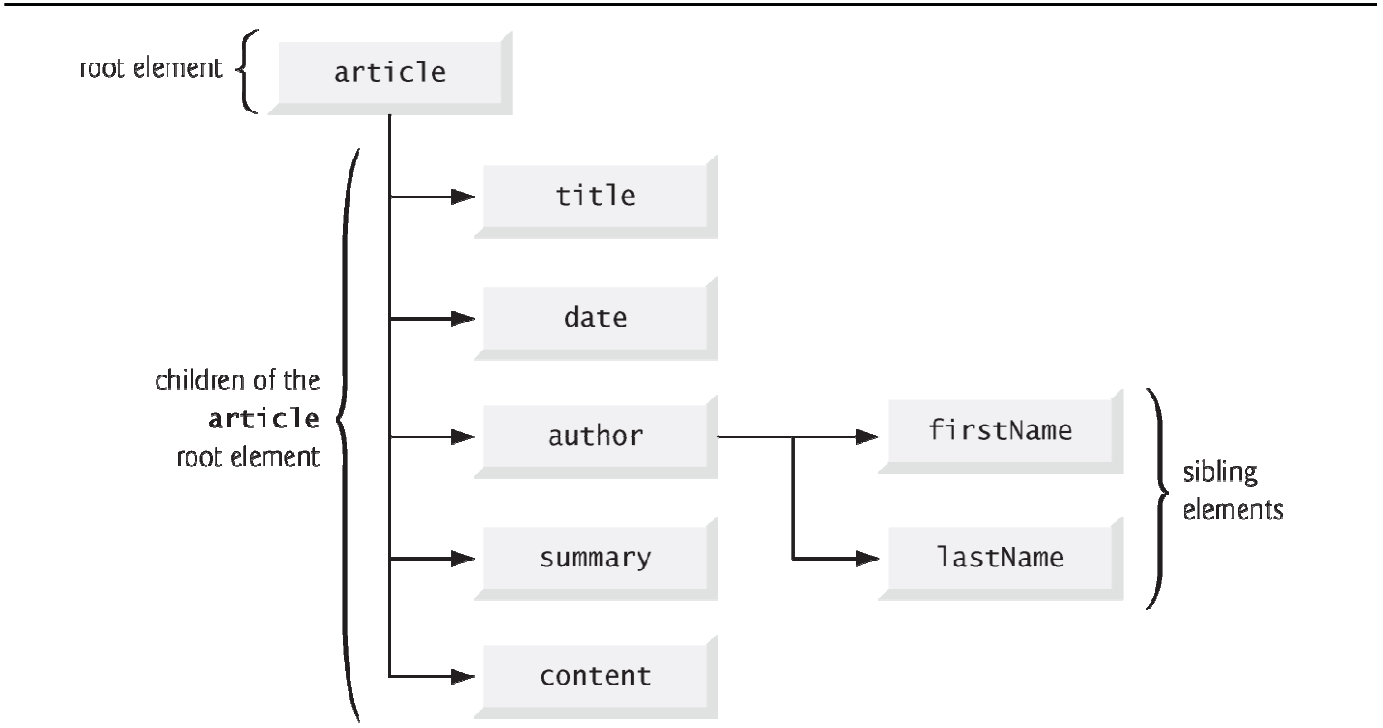
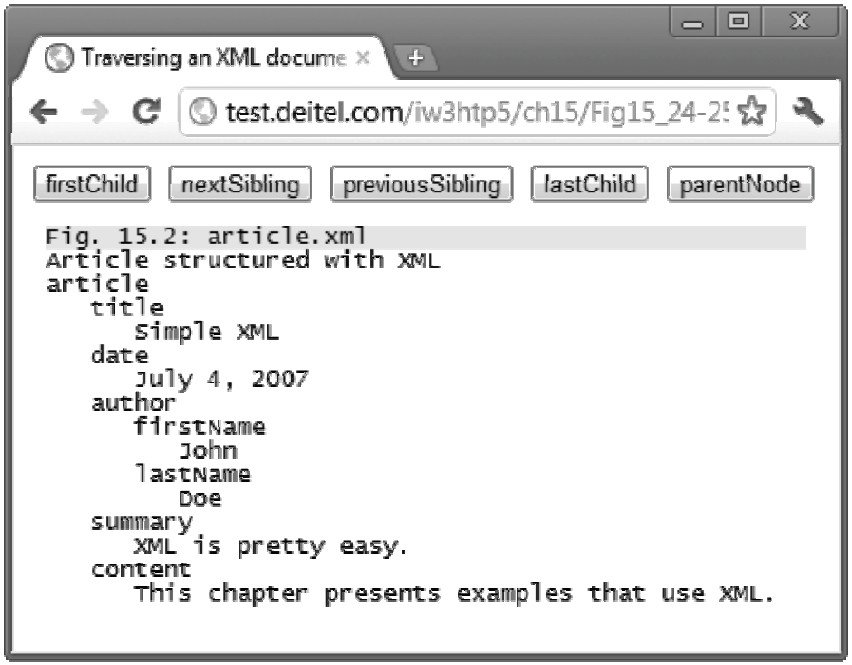


Fig. 15.23 | Tree structure for the document `article.xml` of Fig. 15.2.

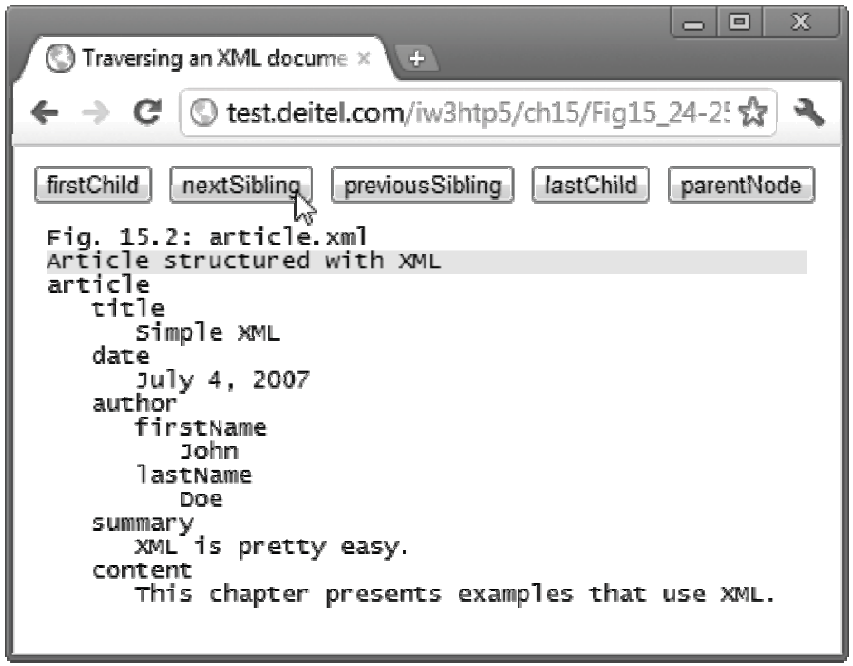
```
<!DOCTYPE html>

<!-- Fig. 15.24: XMLDOMTraversal.html -->
<!-- Traversing an XML document using the XML DOM. -->
<html>
<head>
   <meta charset = "utf-8">
   <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "XMLDOMTraversal.js"></script>
   <title>Traversing an XML document using the XML DOM</title>
</head>
<body id = "body">
   <form action = "#">
      <input id = "firstChild" type = "button" value = "firstChild">
      <input id = "nextSibling" type = "button" value = "nextSibling">
      <input id = "previousSibling" type = "button"
         value = "previousSibling">
      <input id = "lastChild" type = "button" value = "lastChild">
      <input id = "parentNode" type = "button" value = "parentNode">
   </form>
   <div id = "outputDiv"></div>
</body>
</html>
```

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 1 of 11.)



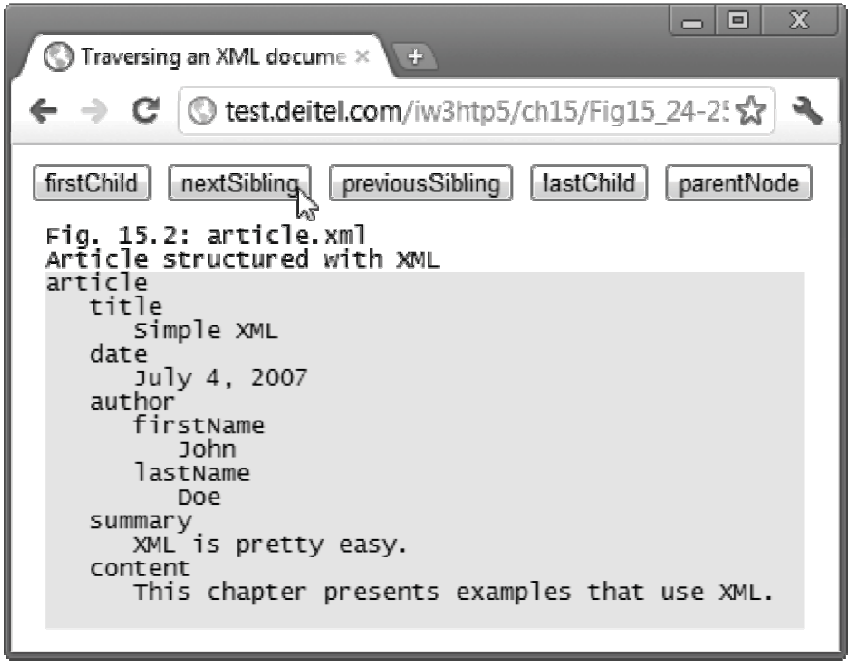
a) Comment node at the beginning of `article.xml` is highlighted when the XML document first loads

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 2 of 11.)



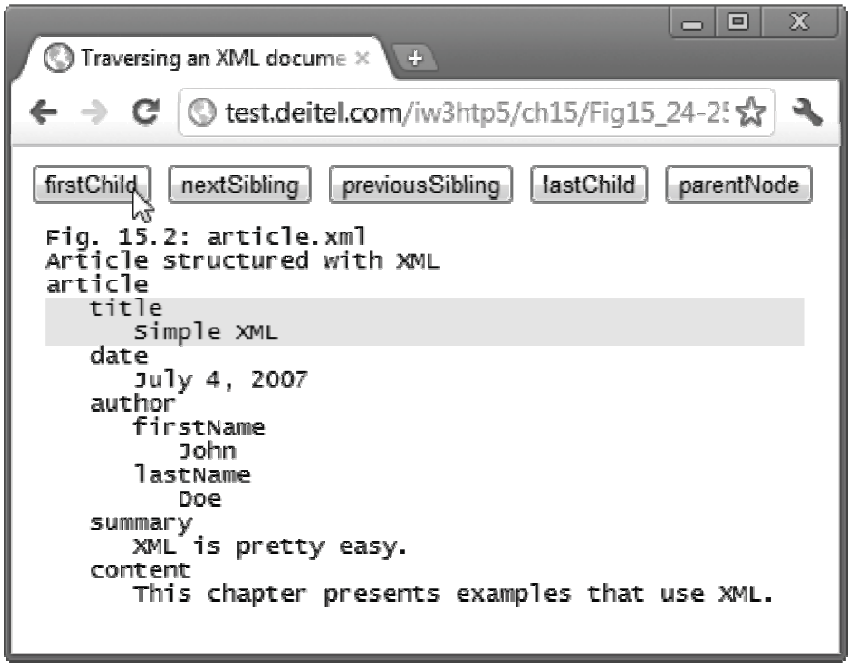
b) User clicked the **nextSibling** button to highlight the second comment node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 3 of 11.)



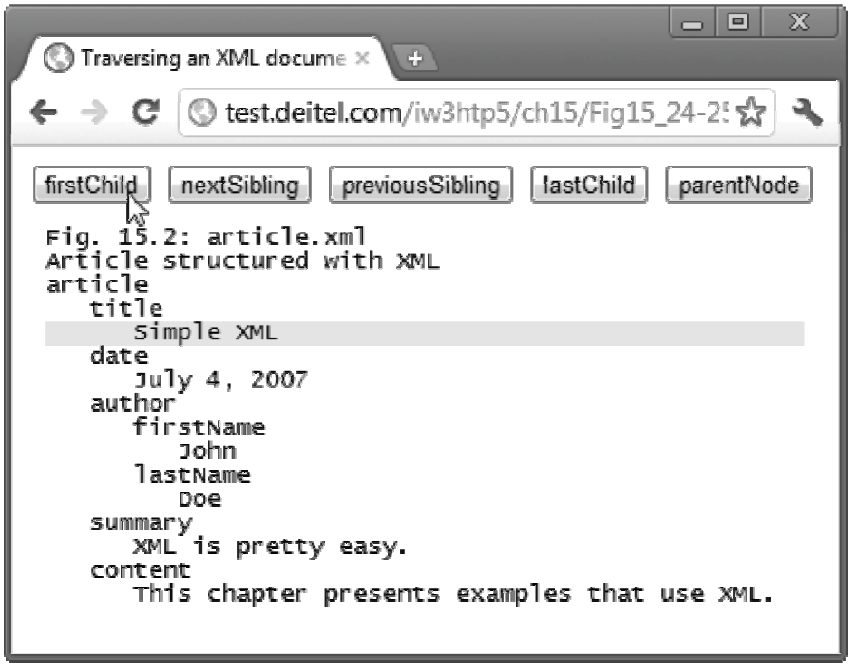
c) User clicked the **nextSibling** button again to highlight the `article` node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 4 of 11.)



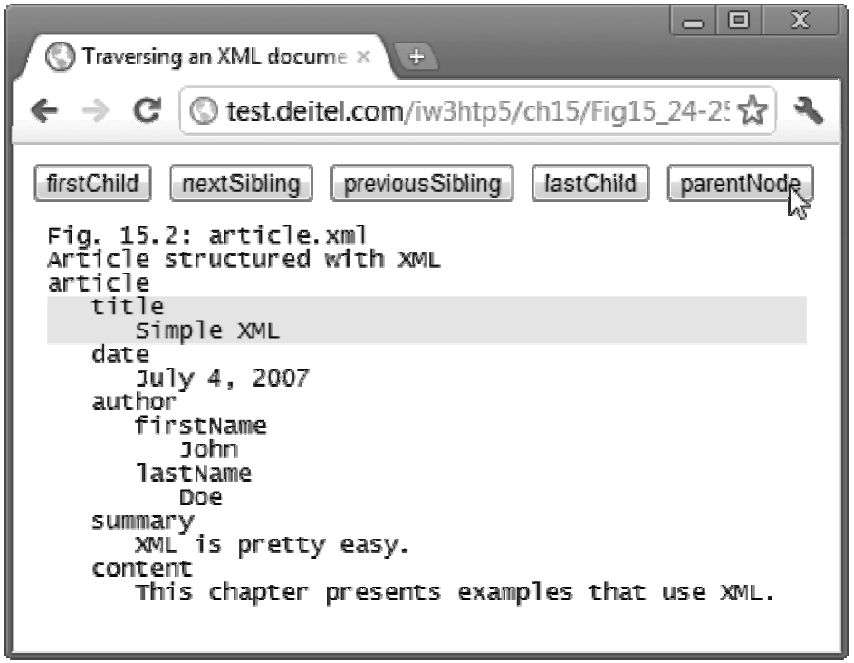
d) User clicked the **firstChild** button to highlight the `article` node's `title` child node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 5 of 11.)



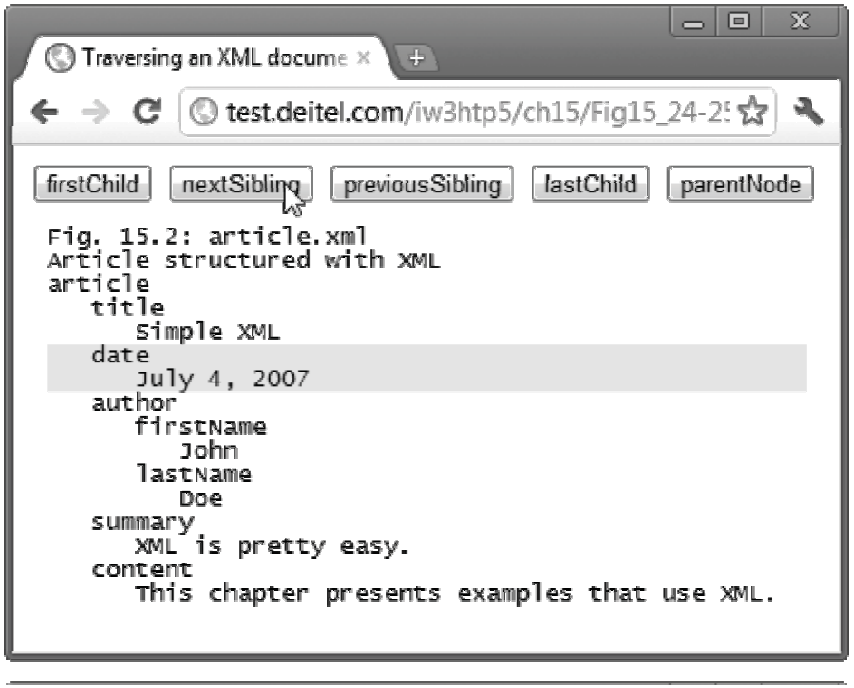
e) User clicked the **firstChild** button again to highlight the `title` node's text child node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 6 of 11.)



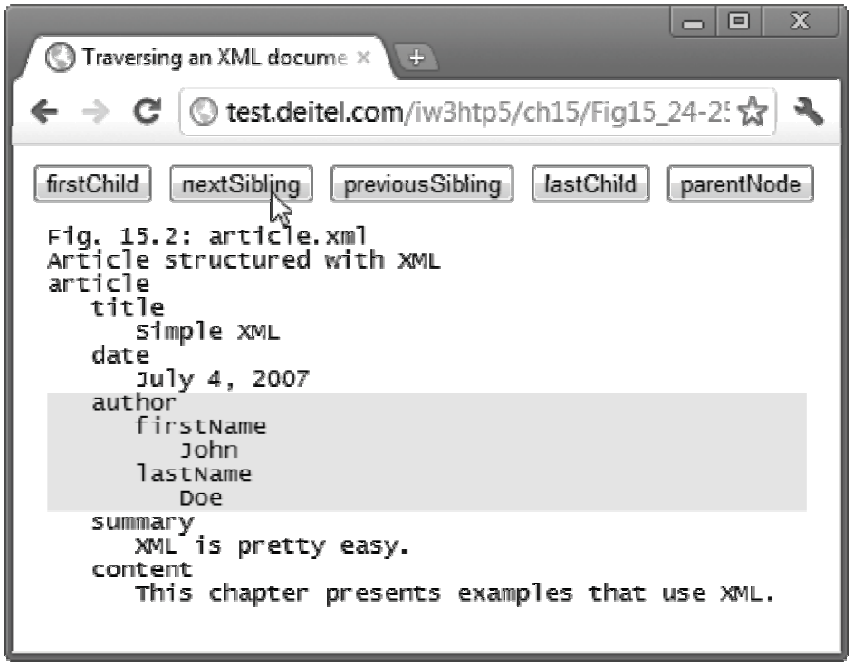
f) User clicked the **parentNode** button to highlight the text node's parent `title` node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 7 of 11.)



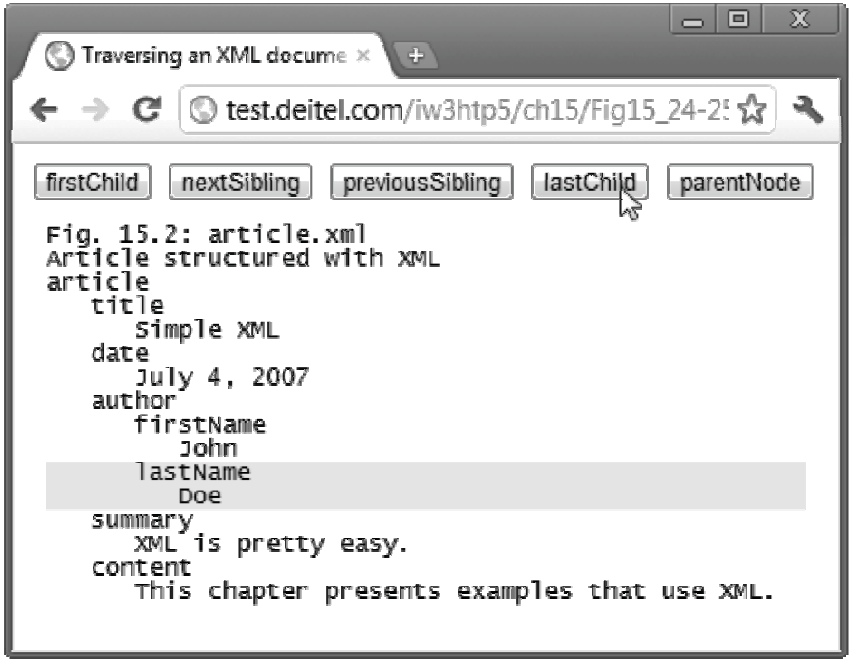
g) User clicked the **nextSibling** button to highlight the `title` node's `date` sibling node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 8 of 11.)



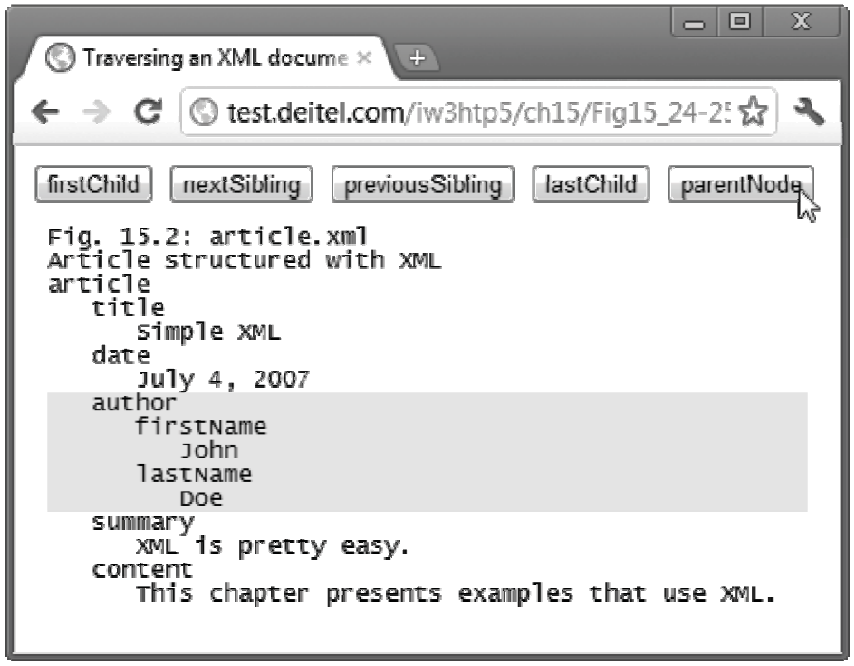
h) User clicked the **nextSibling** button to highlight the `date` node's `author` sibling node

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 9 of 11.)



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

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 10 of 11.)



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

Fig. 15.24 | Traversing an XML document using the XML DOM. (Part 11 of 11.)

```
1  <!-- Fig. 15.25: XMLDOMTraversal.html -->
2  <!-- JavaScript for traversing an XML document using the XML DOM. -->
3  var outputHTML = ""; // stores text to output in outputDiv
4  var idCounter = 1; // used to create div IDs
5  var depth = -1; // tree depth is -1 to start
6  var current = null; // represents the current node for traversals
7  var previous = null; // represents prior node in traversals
8
9  // register event handlers for buttons and load XML document
10 function start()
11 {
12    document.getElementById( "firstChild" ).addEventListener(
13       "click", processFirstChild, false );
14    document.getElementById( "nextSibling" ).addEventListener(
15       "click", processNextSibling, false );
16    document.getElementById( "previousSibling" ).addEventListener(
17       "click", processPreviousSibling, false );
18    document.getElementById( "lastChild" ).addEventListener(
19       "click", processLastChild, false );
20    document.getElementById( "parentNode" ).addEventListener(
21       "click", processParentNode, false );
22    loadXMLDocument( 'article.xml' )
23 } // end function start
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 1 of 10.)

```
24
25  // load XML document based on whether the browser is IE7 or Firefox 2
26  function loadXMLDocument( url )
27  {
28     var xmlHttpRequest = new XMLHttpRequest();
29     xmlHttpRequest.open( "get", url, false );
30     xmlHttpRequest.send( null );
31     var doc = xmlHttpRequest.responseXML;
32     buildHTML( doc.childNodes ); // display the nodes
33     displayDoc(); // display the document and highlight current node
34  } // end function loadXMLDocument
35
36  // traverse xmlDocument and build HTML5 representation of its content
37  function buildHTML( childList )
38  {
39     ++depth; // increase tab depth
40
41     // display each node's content
42     for ( var i = 0; i < childList.length; i++ )
43     {
44        switch ( childList[ i ].nodeType )
45        {
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 2 of 10.)

```
46            case 1: // Node.ELEMENT_NODE; value used for portability
47            outputHTML += "<div id=\"id" + idCounter + "\">";
48            spaceOutput( depth ); // insert spaces
49            outputHTML += childList[ i ].nodeName; // show node's name
50            ++idCounter; // increment the id counter
51
52            // if current node has children, call buildHTML recursively
53            if ( childList[ i ].childNodes.length != 0 )
54               buildHTML( childList[ i ].childNodes );
55
56            outputHTML += "</div>";
57            break;
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 3 of 10.)

```
58          case 3: // Node.TEXT_NODE; value used for portability
59          case 8: // Node.COMMENT_NODE; value used for portability
60             // if nodeValue is not 3 or 6 spaces (Firefox issue),
61             // include nodeValue in HTML
62             if ( childList[ i ].nodeValue.indexOf( "   " ) == -1 &&
63                childList[ i ].nodeValue.indexOf( "      " ) == -1 )
64             {
65                outputHTML += "<div id=\"id" + idCounter + "\">";
66                spaceOutput( depth ); // insert spaces
67                outputHTML += childList[ i ].nodeValue + "</div>";
68                ++idCounter; // increment the id counter
69             } // end if
70          } // end switch
71       } // end for
72
73       --depth; // decrease tab depth
74    } // end function buildHTML
75
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 4 of 10.)

```
76  // display the XML document and highlight the first child
77  function displayDoc()
78  {
79     document.getElementById( "outputDiv" ).innerHTML = outputHTML;
80     current = document.getElementById( 'id1' );
81     setCurrentNodeStyle( current.getAttribute( "id" ), true );
82  } // end function displayDoc
83
84  // insert nonbreaking spaces for indentation
85  function spaceOutput( number )
86  {
87     for ( var i = 0; i < number; i++ )
88     {
89        outputHTML += "   ";
90     } // end for
91  } // end function spaceOutput
92
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 5 of 10.)

```
93  // highlight first child of current node
94  function processFirstChild()
95  {
96     if ( current.childNodes.length == 1 && // only one child
97        current.firstChild.nodeType == 3 )  // and it's a text node
98     {
99        alert( "There is no child node" );
100    } // end if
101    else if ( current.childNodes.length > 1 )
102    {
103       previous = current; // save currently highlighted node
104
105       if ( current.firstChild.nodeType != 3 ) // if not text node
106          current = current.firstChild; // get new current node
107       else // if text node, use firstChild's nextSibling instead
108          current = current.firstChild.nextSibling; // get first sibling
109
110       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
111       setCurrentNodeStyle( current.getAttribute( "id" ), true );
112    } // end if
113    else
114       alert( "There is no child node" );
115 } // end function processFirstChild
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 6 of 10.)

```
116
117 // highlight next sibling of current node
118 function processNextSibling()
119 {
120    if ( current.getAttribute( "id" ) != "outputDiv" &&
121       current.nextSibling )
122    {
123       previous = current; // save currently highlighted node
124       current = current.nextSibling; // get new current node
125       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
126       setCurrentNodeStyle( current.getAttribute( "id" ), true );
127    } // end if
128    else
129       alert( "There is no next sibling" );
130 } // end function processNextSibling
131
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 7 of 10.)

```
132 // highlight previous sibling of current node if it is not a text node
133 function processPreviousSibling()
134 {
135    if ( current.getAttribute( "id" ) != "outputDiv" &&
136       current.previousSibling && current.previousSibling.nodeType != 3 )
137    {
138       previous = current; // save currently highlighted node
139       current = current.previousSibling; // get new current node
140       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
141       setCurrentNodeStyle( current.getAttribute( "id" ), true );
142    } // end if
143    else
144       alert( "There is no previous sibling" );
145 } // end function processPreviousSibling
146
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 8 of 10.)

```
147 // highlight last child of current node
148 function processLastChild()
149 {
150    if ( current.childNodes.length == 1 &&
151       current.lastChild.nodeType == 3 )
152    {
153       alert( "There is no child node" );
154    } // end if
155    else if ( current.childNodes.length != 0 )
156    {
157       previous = current; // save currently highlighted node
158       current = current.lastChild; // get new current node
159       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
160       setCurrentNodeStyle( current.getAttribute( "id" ), true );
161    } // end if
162    else
163       alert( "There is no child node" );
164 } // end function processLastChild
165
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 9 of 10.)

```
166 // highlight parent of current node
167 function processParentNode()
168 {
169    if ( current.parentNode.getAttribute( "id" ) != "body" )
170    {
171       previous = current; // save currently highlighted node
172       current = current.parentNode; // get new current node
173       setCurrentNodeStyle( previous.getAttribute( "id" ), false );
174       setCurrentNodeStyle( current.getAttribute( "id" ), true );
175    } // end if
176    else
177       alert( "There is no parent node" );
178 } // end function processParentNode
179
180 // set style of node with specified id
181 function setCurrentNodeStyle( id, highlight )
182 {
183    document.getElementById( id ).className =
184       ( highlight ? "highlighted" : "" );
185 } // end function setCurrentNodeStyle
186
187 window.addEventListener( "load", start, false );
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 10 of 10.)

Property/ Method	Description
`nodeType`	An integer representing the node type.
`nodeName`	The name of the node.
`nodeValue`	A string or null depending on the node type.
`parentNode`	The parent node.
`childNodes`	A `NodeList` (Fig. 15.27) with all the children of the node.
`firstChild`	The first child in the `Node`'s `NodeList`.
`lastChild`	The last child in the `Node`'s `NodeList`.
`previousSibling`	The node preceding this node; `null` if there's no such node.
`nextSibling`	The node following this node; `null` if there's no such node.
`attributes`	A collection of `Attr` objects (Fig. 15.30) containing the attributes for this node.

Fig. 15.26 | Common Node properties and methods. (Part 1 of 2.)

Property/ Method	Description
`insertBefore`	Inserts the node (passed as the first argument) before the existing node (passed as the second argument). If the new node is already in the tree, it's removed before insertion. The same behavior is true for other methods that add nodes.
`replaceChild`	Replaces the second argument node with the first argument node.
`removeChild`	Removes the child node passed to it.
`appendChild`	Appends the node it receives to the list of child nodes.

Fig. 15.26 | Common Node properties and methods. (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. 15.27 | `NodeList` property and method.

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

Fig. 15.28 | Document property and methods.

Property/ Method	Description
`tagName`	The name of the element.
`getAttribute`	Returns the value of the specified attribute.
`setAttribute`	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. 15.29 | `Element` property and methods.

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

Fig. 15.30 | `Attr` properties.

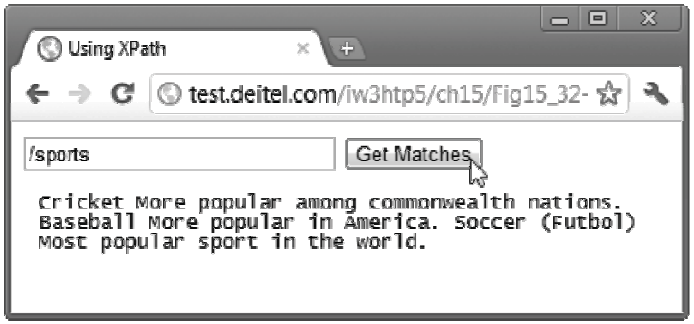
Property	Description
`data`	The text contained in the node.
`length`	The number of characters contained in the node.

Fig. 15.31 | `Text` properties.

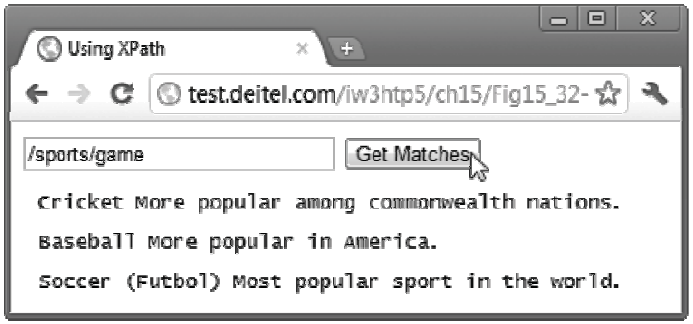
```
<!DOCTYPE html>

<!-- Fig. 15.32: xpath.html -->
<!-- Using XPath to locate nodes in an XML document. -->
<html>
<head>
   <meta charset = "utf-8">
   <link rel = "stylesheet" type = "text/css" href = "style.css">
   <script src = "xpath.js"></script>
   <title>Using XPath</title>
</head>
<body id = "body">
   <form id = "myForm" action = "#">
      <input id = "inputField" type = "text">
      <input id = "matchesButton" type = "button" value = "Get Matches">
   </form>
   <div id = "outputDiv"></div>
</body>
</html>
```

Fig. 15.32 | Using XPath to locate nodes in an XML document. (Part 1 of 3.)

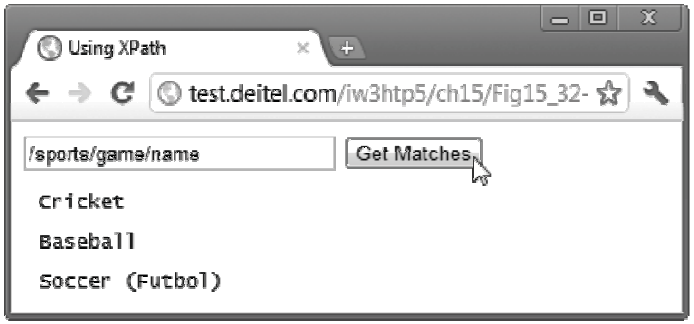
a) Selecting the `sports` node



b) Selecting the `game` nodes from the `sports` node



c) Selecting the `name` node from each `game` node



d) Selecting the `paragraph` node from each `game`

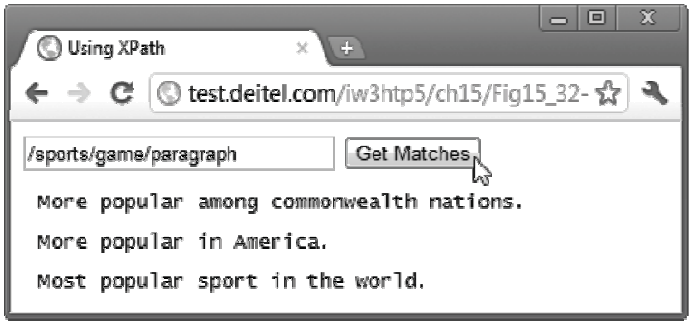
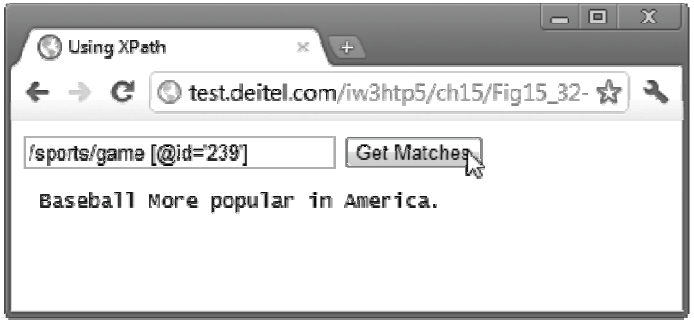


Fig. 15.32 | Using XPath to locate nodes in an XML document. (Part 2 of 3.)

e) Selecting the `game` with the `id` attribute value `239`



f) Selecting the `game` with `name` element value `Cricket`

Fig. 15.32 | Using XPath to locate nodes in an XML document. (Part 3 of 3.)

```
1  // Fig. 15.33: xpath.html
2  // JavaScript that uses XPath to locate nodes in an XML document.
3  var doc; // variable to reference the XML document
4  var outputHTML = ""; // stores text to output in outputDiv
5
6  // register event handler for button and load XML document
7  function start()
8  {
9     document.getElementById( "matchesButton" ).addEventListener(
10       "click", processXPathExpression, false );
11    loadXMLDocument( "sports.xml" );
12 } // end function start
13
14 // load XML document programmatically
15 function loadXMLDocument( url )
16 {
17    var xmlHttpRequest = new XMLHttpRequest();
18    xmlHttpRequest.open( "get", url, false );
19    xmlHttpRequest.send( null );
20    doc = xmlHttpRequest.responseXML;
21 } // end function loadXMLDocument
22
```

Fig. 15.33 | Using XPath to locate nodes in an XML document. (Part 1 of 3.)

```
23  // display the XML document
24  function displayHTML()
25  {
26     document.getElementById( "outputDiv" ).innerHTML = outputHTML;
27  } // end function displayDoc
28
29  // obtain and apply XPath expression
30  function processXPathExpression()
31  {
32     var xpathExpression = document.getElementById( "inputField" ).value;
33     var result;
34     outputHTML = "";
35
36     if ( !doc.evaluate ) // Internet Explorer
37     {
38        result = doc.selectNodes( xpathExpression );
39
40        for ( var i = 0; i < result.length; i++ )
41        {
42           outputHTML += "<p>" + result.item( i ).text + "</p>";
43        } // end for
44     } // end if
```

Fig. 15.33 | Using XPath to locate nodes in an XML document. (Part 2 of 3.)

```
45      else // other browsers
46      {
47         result = doc.evaluate( xpathExpression, doc, null,
48            XPathResult.ORDERED_NODE_ITERATOR_TYPE, null );
49         var current = result.iterateNext();
50
51         while ( current )
52         {
53            outputHTML += "<p>" + current.textContent + "</p>";
54            current = result.iterateNext();
55         } // end while
56      } // end else
57
58      displayHTML();
59   } // end function processXPathExpression
60
61   window.addEventListener( "load", start, false );
```

Fig. 15.33 | Using XPath to locate nodes in an XML document. (Part 3 of 3.)

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

Fig. 15.34 | XML document that describes various sports.

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 child nodes of `game`. The `game` is a child of `sports`, which is a child of the document root.
`/sports/game/paragraph`	Matches all `paragraph` nodes that are child nodes 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. 15.35 | XPath expressions and descriptions.