Chapter 15
XML
Internet & World Wide Web
How to Program, 5/e

OBJECTIVES
In this chapter you will:
• Markup data using XML
• Learn how XML namespaces help provide unique XML element and attribute names.
• Create DTDs and schemas for specifying and validating the structure of an XML document.
• Create and use simple XML stylesheets to render XML document data.
• Review and manipulate XML data programmatically using JavaScript.

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
• 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.

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

```
xhtml version = "1.0"
</xhtml>
<player>
  <firstnames>John</firstnames>
  <surname>Babbage</surname>
  <height>5'10"</height>
</player>
```

1. <i>Fig. 15.1</i> player.xml
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.

```xml
<xml version="1.0">
  <!-- Fig. 15.2: article.xml -->
  <!-- Article structured with XML -->
  <article>
    <title>Article structured with XML</title>
    <date>July 4, 2017</date>
    <author>John Doe</author>
  </article>
</xml>
```

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

Portability Tip 15.1

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

Common Programming Error 15.1

Placing any characters, including white space, before the XML declaration is an error.
Common Programming Error 15.2

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

Common Programming Error 15.3

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

Common Programming Error 15.4

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

Good Programming Practice 15.1

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

Common Programming Error 15.5

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

Fig. 11-3 | `article.xml` displayed in the Google Chrome browser. (Part 1 of 2.)
Portability Tip 15.2

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

Common Programming Error 15.6

Failure to enclose attribute values in double (""") or single (‘’) quotes is a syntax error.
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 <html version = "1.0">
2 <head>
3   <title>Fig. 15.5: namespace.xml</title>
4 </head>
5 <html><head><html>
6   <meta http-equiv = "Content-Type" content = "text/html"/>
7   <base directory = ""/>
8   <title>Fig. 15.5: namespace.xml</title>
9   <html file = "book.xml">
10   <book description = "A book">
11   </book>
12 </head>
13 <body>
14   <image file = "funny.jpg">
15     <image description = "Funny pictures">
16     </image>
17   </image>
18 </body>
19 </html>
```

Fig. 15.5 | XML namespaces demonstration.

```
1 <html version = "1.0">
2 <head>
3   <title>Fig. 15.6: defaultnamespace.xml</title>
4 </head>
5 <html><head><html>
6   <meta http-equiv = "Content-Type" content = "text/html"/>
7   <base directory = ""/>
8   <title>Fig. 15.6: defaultnamespace.xml</title>
9   <html file = "book.xml">
10   <book description = "A book">
11   </book>
12 </head>
13 <body>
14   <image file = "funny.jpg">
15     <image description = "Funny pictures">
16     </image>
17     <image width = "200" height = "300"/>
18   </image>
19 </body>
20 </html>
```

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.

```
Software Engineering Observation 15.2

XML documents can have many different structures, and for this reason an application cannot be certain whether a particular document it receives is complete, ordered properly, and not missing data. DTDs and schemas (Section 15.6) solve this problem by providing an extensible way to describe XML document structure.

Applications should use DTDs or schemas to confirm whether XML documents are valid.
```
Software Engineering Observation 15.3

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

Unlike DTDs
- Schemas 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 can contain attributes or child elements; complex types cannot
- 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

Common Programming Error 15.8

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

Common Programming Error 15.9

Using markup characters (e.g., "", , and ) in parsed character data is an error. Use character entity references (e.g., &lt;, &gt; and &amp;) instead.

15.6 W3C XML Schema Documents

- Unlike DTDs
  - Schemas 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 can contain attributes or child elements; complex types cannot
  - 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 `books.xml` with a missing contact name.

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

**Portability Tip 15.3**
W3C XML Schema authors specify URI `http://www.w3.org/2001/XMLSchema` when referring to the XML Schema namespace. This namespace contains predefined elements that comprise the XML Schema vocabulary. Specifying this URI ensures that validation tools correctly identify XML Schema elements and do not confuse them with those defined by document authors.

**Fig. 15.11** Some XML Schema types (Part 1 of 2).
### 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
  - `no 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 result 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 \texttt{mrow} element

- **mrow element**
  - groups elements that are positioned horizontally in an expression
  - Entity reference \texttt{&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 \texttt{&delta;} represents a lowercase delta symbol

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

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

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

Fig. 15.17 | Calculus expression marked up with MathML and displayed in the Firefox browser. (Part 2 of 3.)
15.8 Extensible Styleheet Language and XSL Transformations

- Convert XML into any text-based document
- XSLT (Extensible Stylesheet Language Transformations) extends the XML specification
- XSLT is used to transform XML documents into other formats
- An XSLT processor reads a source XML document, applies transformations defined in an XSLT style sheet, and produces a result document
- XSLT uses a stylesheet to define how the XML document should be transformed
- XSLT is a powerful tool for generating dynamic content and customizing output
- XSLT is widely used in web applications, document publishing, and data integration

15.8 Extensible Styleheet 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 transformed document)
- XPath (XML Path Language) is used to navigate the source tree
- XPath is a language for specifying paths to nodes in an XML document
- XPath expressions can be used to select nodes, retrieve information, and manipulate the document structure
- XSLT (Extensible Stylesheet Language Transformations) is a language for defining transformations on XML documents
- XSLT uses templates and rules to transform XML documents
- XSLT is used to generate HTML, PDF, and other output formats from XML input

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

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

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

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.

15.9 Document Object Model (Cont.)

- nodeType property of a node
- contains the type of the node
- Non-breaking spaces (\&nbsp;)
  - spaces that the browser is not allowed to collapse or that can be used to keep words together.

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. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 1 of 16.)

```javascript
// load XML document based on whether the browser is IE or Firefox
function loadXMLDocument(url) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send(null);
    var xmlDoc = xhr.responseXML;
    return xmlDoc;
}

// traverse element and build HTML representation of its content
function buildHTMLChildNodes(n) {
    var html = ";";
    for (var i = 0; i < n.childNodes.length; i++)
        html += "<" + n.childNodes[i].nodeName + ""; // display each node's name
    return html;
}

// display XML document and highlight first child
function displayDoc(n) {
    var html = buildHTMLChildNodes(n);
    // display the XML document and highlight the first child
    document.getElementById("bodyContent").innerHTML = html;
}

// insert nonbreaking spaces for indentation
function spaceOutput(n) {
    var space = " 
    for (var i = 0; i < n.length; i++)
        spaceOutput(n[i]);
    outputHTML += "\n";
}

// function to traverse an XML document using the XML DOM
function traverseXML(doc, context) {
    if (doc.nodeType == 1) // if doc is element
        for (var i = 0; i < doc.childNodes.length; i++)
            traverseXML(doc.childNodes[i], context);
    else if (doc.nodeType == 3) // if doc is text
        outputHTML += doc.data;
}
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 2 of 16.)

```javascript
// load XML document based on whether the browser is IE or Firefox
function loadXMLDocument(url) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send(null);
    var xmlDoc = xhr.responseXML;
    return xmlDoc;
}

// traverse element and build HTML representation of its content
function buildHTMLChildNodes(n) {
    var html = ";";
    for (var i = 0; i < n.childNodes.length; i++)
        html += "<" + n.childNodes[i].nodeName + ""; // display each node's name
    return html;
}

// display XML document and highlight first child
function displayDoc(n) {
    var html = buildHTMLChildNodes(n);
    // display the XML document and highlight the first child
    document.getElementById("bodyContent").innerHTML = html;
}

// insert nonbreaking spaces for indentation
function spaceOutput(n) {
    var space = " 
    for (var i = 0; i < n.length; i++)
        spaceOutput(n[i]);
    outputHTML += "\n";
}

// function to traverse an XML document using the XML DOM
function traverseXML(doc, context) {
    if (doc.nodeType == 1) // if doc is element
        for (var i = 0; i < doc.childNodes.length; i++)
            traverseXML(doc.childNodes[i], context);
    else if (doc.nodeType == 3) // if doc is text
        outputHTML += doc.data;
}
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 3 of 16.)

```javascript
// load XML document based on whether the browser is IE or Firefox
function loadXMLDocument(url) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send(null);
    var xmlDoc = xhr.responseXML;
    return xmlDoc;
}

// traverse element and build HTML representation of its content
function buildHTMLChildNodes(n) {
    var html = ";";
    for (var i = 0; i < n.childNodes.length; i++)
        html += "<" + n.childNodes[i].nodeName + ""; // display each node's name
    return html;
}

// display XML document and highlight first child
function displayDoc(n) {
    var html = buildHTMLChildNodes(n);
    // display the XML document and highlight the first child
    document.getElementById("bodyContent").innerHTML = html;
}

// insert nonbreaking spaces for indentation
function spaceOutput(n) {
    var space = " 
    for (var i = 0; i < n.length; i++)
        spaceOutput(n[i]);
    outputHTML += "\n";
}

// function to traverse an XML document using the XML DOM
function traverseXML(doc, context) {
    if (doc.nodeType == 1) // if doc is element
        for (var i = 0; i < doc.childNodes.length; i++)
            traverseXML(doc.childNodes[i], context);
    else if (doc.nodeType == 3) // if doc is text
        outputHTML += doc.data;
}
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 4 of 16.)

```javascript
// load XML document based on whether the browser is IE or Firefox
function loadXMLDocument(url) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send(null);
    var xmlDoc = xhr.responseXML;
    return xmlDoc;
}

// traverse element and build HTML representation of its content
function buildHTMLChildNodes(n) {
    var html = ";";
    for (var i = 0; i < n.childNodes.length; i++)
        html += "<" + n.childNodes[i].nodeName + ""; // display each node's name
    return html;
}

// display XML document and highlight first child
function displayDoc(n) {
    var html = buildHTMLChildNodes(n);
    // display the XML document and highlight the first child
    document.getElementById("bodyContent").innerHTML = html;
}

// insert nonbreaking spaces for indentation
function spaceOutput(n) {
    var space = " 
    for (var i = 0; i < n.length; i++)
        spaceOutput(n[i]);
    outputHTML += "\n";
}

// function to traverse an XML document using the XML DOM
function traverseXML(doc, context) {
    if (doc.nodeType == 1) // if doc is element
        for (var i = 0; i < doc.childNodes.length; i++)
            traverseXML(doc.childNodes[i], context);
    else if (doc.nodeType == 3) // if doc is text
        outputHTML += doc.data;
}
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 5 of 16.)

```javascript
// load XML document based on whether the browser is IE or Firefox
function loadXMLDocument(url) {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", url, false);
    xhr.send(null);
    var xmlDoc = xhr.responseXML;
    return xmlDoc;
}

// traverse element and build HTML representation of its content
function buildHTMLChildNodes(n) {
    var html = ";";
    for (var i = 0; i < n.childNodes.length; i++)
        html += "<" + n.childNodes[i].nodeName + ""; // display each node's name
    return html;
}

// display XML document and highlight first child
function displayDoc(n) {
    var html = buildHTMLChildNodes(n);
    // display the XML document and highlight the first child
    document.getElementById("bodyContent").innerHTML = html;
}

// insert nonbreaking spaces for indentation
function spaceOutput(n) {
    var space = " 
    for (var i = 0; i < n.length; i++)
        spaceOutput(n[i]);
    outputHTML += "\n";
}

// function to traverse an XML document using the XML DOM
function traverseXML(doc, context) {
    if (doc.nodeType == 1) // if doc is element
        for (var i = 0; i < doc.childNodes.length; i++)
            traverseXML(doc.childNodes[i], context);
    else if (doc.nodeType == 3) // if doc is text
        outputHTML += doc.data;
}
```

Fig. 15.25 | JavaScript for traversing an XML document using the XML DOM. (Part 6 of 16.)
```javascript
// highlight next sibling of current node
function processNextSibling()
{
  if (current.getAttribute("id").indexOf("outputDiv") == -1)
    current.setAttribute("class", "outputDiv");
  previous = current;
  current = current.nextSibling;
  if (current)
    setCurrentNodeStyle(current, true);
  else
    alert("There is no next sibling");
}

// highlight previous sibling of current node if it is not a text node
function processPreviousSibling()
{
  if (current.getAttribute("id").indexOf("outputDiv") == -1)
    current.setAttribute("class", "outputDiv");
  previous = current;
  current = current.previousSibling;
  if (current)
    setCurrentNodeStyle(current, true);
  else
    alert("There is no previous sibling");
}

// highlight last child of current node
function processLastChild()
{
  if (current.childNodes.length == 0)
    current.setAttribute("class", "outputNull");
  else
    alert("There is no child node");
}

// highlight first child of current node
function processFirstChild()
{
  if (current.firstElementChild)
    setCurrentNodeStyle(current.firstElementChild, true);
  else
    alert("There is no child node");
}
```

**Fig. 15.23** | JavaScript for traversing an XML document using the XML DOM. (Part 7 of 10.)

**Portability Tip 15.4**
Firefox’s XML parser does not ignore white space used for indentation in XML documents. Instead, it creates text nodes containing the white-space characters.

**Fig. 15.26** | Common node properties and methods. (Part 1 of 2.)
### Fig. 13.26 | Common node properties and methods. (Part 2 of 2.)

<table>
<thead>
<tr>
<th>Property/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeBefore</td>
<td>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 inserted before insertion. The same behavior is seen for other methods that add nodes.</td>
</tr>
<tr>
<td>replaceChild</td>
<td>Replaces the second argument node with the first argument node.</td>
</tr>
<tr>
<td>removeChild</td>
<td>Removes the child node passed to it.</td>
</tr>
<tr>
<td>appendChild</td>
<td>Appends the node it receives to the list of child nodes.</td>
</tr>
</tbody>
</table>

### Fig. 13.27 | Node, list, property and method.

<table>
<thead>
<tr>
<th>Property/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>The length of the array.</td>
</tr>
</tbody>
</table>

### Fig. 13.28 | Document property and methods.

<table>
<thead>
<tr>
<th>Property/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentElement</td>
<td>The root node of the document.</td>
</tr>
<tr>
<td>createElement</td>
<td>Creates and returns a new node with the specified tag name.</td>
</tr>
<tr>
<td>createTextNode</td>
<td>Creates and returns a new text node with the specified name and value.</td>
</tr>
<tr>
<td>createElementNS</td>
<td>Creates and returns a new node with the specified tag name and namespace URI.</td>
</tr>
<tr>
<td>getElementsByTagName</td>
<td>Returns a list of all the nodes in the sub-tree with the name specified as the first argument, ordered as they would be enumerated in a preorder traversal. An optional second argument specifies either the direct child nodes (false) or any descendant (true).</td>
</tr>
</tbody>
</table>

### Fig. 13.29 | Element property and methods.

<table>
<thead>
<tr>
<th>Property/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The id attribute.</td>
</tr>
<tr>
<td>className</td>
<td>The class name.</td>
</tr>
<tr>
<td>style</td>
<td>The style attribute.</td>
</tr>
<tr>
<td>innerHTML</td>
<td>The inner HTML of the element.</td>
</tr>
</tbody>
</table>

### Fig. 13.30 | Attr properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The specified attribute's value.</td>
</tr>
<tr>
<td>name</td>
<td>The name of the attribute.</td>
</tr>
</tbody>
</table>

### Fig. 13.31 | Text properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The text contained in the node.</td>
</tr>
<tr>
<td>length</td>
<td>The number of characters contained in the node.</td>
</tr>
</tbody>
</table>
1. `<DOCTYPE html>`
2. `<html>`
3. `<head>`
4. `meta charset = "utf-8"`
5. `<link rel = "stylesheet" type = "text/css" href = "style.css">`
6. `<script src = "event.js" type = "text/javascript" defer>`
7. `</head>`
8. `body>`
9. `<div id = "body">`
10. `<h1 id = "header" style = "display: none;">`
11. `<input id = "inputField" type = "text" value = "Get Notches">`
12. `</header>`
13. `</body>`
14. `</html>`

---

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

---

---

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

---

---

Fig. 15.34 | Using XPath to locate nodes in an XML document. (Part 2 of 3.)

---

---

Fig. 15.35 | Using XPath to locate nodes in an XML document. (Part 3 of 3.)

---

---

Fig. 15.36 | Using XPath to locate nodes in an XML document. (Part 3 of 3.)
Fig. 15.34 | XML document that describes various sports.

```xml
<!-- Fig. 15.34: sports.xml -->
<-- Sports Database
<sports>
  <game id="757">
    <name>Cricket</name>
    <paragraph>More popular among Commonwealth nations.</paragraph>
  </game>
  <game id="719">
    <name>Football</name>
    <paragraph>More popular in America.</paragraph>
  </game>
  <game id="123">
    <name> outbreaks</name>
    <paragraph>More popular sport in the world.</paragraph>
  </game>
</sports>
```

Fig. 15.35 | XPath expressions and descriptions.

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="250"] Matches the game node with the id attribute 250. 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 the child element whose name is Cricket. The game is a child of sports, which is a child of the document root.