Document Object Model (DOM™)

Objectives

- To understand what the Document Object Model (DOM) is.
- To understand and be able to use the major DOM features.
- To manipulate XML documents programmatically from a Java application.
- To use the DOM to create XML documents.

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

There was a child went forth every day,
And the first object he look'd upon, that object he became.
Walt Whitman

I think that I shall never see
A poem lovely as a tree.
Joyce Kilmer

Outline

4.1 Introduction

In previous chapters, we concentrated on basic XML markup and DTDs for validating XML documents. In this chapter, we focus on manipulating the contents of an XML document programmatically.

XML documents, when parsed, are represented as a hierarchical tree structure in memory. This tree structure contains the document's elements, attributes, character data, etc. The W3C provides a recommendation—called the *XML Document Object Model* (*DOM*)—for building these tree structures in memory. Any parser that adheres to this recommendation is called a *DOM-based parser*. Each element, attribute, **CDATA** section, etc., in an XML document is represented by a *node* in the DOM tree. For example, when the simple XML document

```
<?xml version = "1.0" encoding = "UTF-8"?>
<message from = "Ms. Quito" to = "Russ Tick">
   <body>Hi, Russ!</body>
   <attachment>Hawaii.jpg</attachment>
</message>
```

is parsed, a DOM tree with several nodes is created in memory. One node is created for the **message** element. This node has two *child nodes* (or *children*) that correspond to the **body** element and the **attachment** element. The **body** element has a child node (in this case, a text node) that corresponds to the text **Hi, Russ!**. The **from** and **to** attributes of the **message** element also have corresponding nodes in the DOM tree. A node that contains child nodes is called a *parent node* (e.g., **message**). A parent node can have many children, but a child node can have only one parent node. Nodes that are peers (e.g., **body** and **attachment**) are called *sibling nodes*. A node's *descendent nodes* include that node's children, its children's children and so on. A node's *ancestor nodes* include that node's parent, its parent's parent and so on. The DOM tree has a single *root node* that contains the root element (e.g., **message**), which contains all other nodes in the document.

Although an XML document is a text file, using traditional sequential-file access techniques to retrieve data from the document is neither practical nor efficient, especially for adding and removing elements dynamically. A DOM-based parser *exposes* (i.e., makes available) a programmatic library—called the *DOM Application Programming Interface* (*API*)—that allows data in an XML document to be accessed and modified by manipulating the nodes in a DOM tree. Each node is an object that has methods for accessing the node's names, values, child nodes, etc.

In the DOM API,[1] the primary DOM interfaces are ***Node*** (which represents any node in the tree), ***NodeList*** (which represents an ordered set of nodes), ***NamedNodeMap*** (which represents an unordered set of nodes), ***Document*** (which represents the document), ***Element*** (which represents an element node), ***Attr*** (which represents an attribute node), ***Text*** (which represents a text node) and ***Comment*** (which represents a comment node). In Section 4.3, we provide an overview of these interfaces and their key methods.

Portability Tip 4.1

The DOM interfaces for creating and manipulating XML documents are platform and language independent. DOM parsers exist for many different languages, including Java, C, C++, Python and Perl.

4.2 DOM with Java

To introduce document manipulation with the XML DOM, we begin with a simple Java example. This example takes an XML document (Fig. 4.1) and uses the JAXP API to display the document's element names and values. Figure 4.2 presents the Java code[2] that manipulates this XML document and displays its content.

```
1  <?xml version = "1.0" encoding = "UTF-8" standalone = "yes"?>
2
3  <!-- Fig. 4.1: article.xml       -->
4  <!-- Article formatted with XML  -->
5
6  <article>
7
8     <title>Simple XML</title>
9
10    <date>May 31, 2002</date>
11
12    <author>
13       <fname>Tarz</fname>
14       <lname>Ant</lname>
15    </author>
16
17    <summary>XML is easy.</summary>
18
19    <content>Once you have mastered XHTML, you can easily learn
20       XML. You must remember that XML is not for
21       displaying information but for managing information.
22    </content>
23
24 </article>
```

Fig. 4.1 XML document used in Fig. 4.2.

1. In this chapter, we use the reference implementation for the Java API for XML Processing 1.2 (JAXP), which is part of the Java Web Services Developer Pack 1.0.
2. Before running the examples in this chapter, execute the batch file (**jws1_xml.bat**) provided with the book's examples.

Lines 9, 12 and 13 **import** packages related to XML processing. Package ***javax.xml.parsers*** provides classes related to parsing an XML document. Package ***org.w3c.dom*** provides the DOM-API programmatic interface (i.e., classes, methods, etc.). Package ***org.xml.sax*** provides classes used by the example's exception handlers.

```
1  // Fig. 4.2 : XMLInfo.java
2  // Outputs node information
3  package com.deitel.jws1.xml;
4
5  // Java core libraries
6  import java.io.*;
7
8  // Java standard extensions
9  import javax.xml.parsers.*;
10
11 // third-party libraries
12 import org.w3c.dom.*;
13 import org.xml.sax.*;
14
15 public class XMLInfo {
16
17    public static void main( String args[] )
18    {
19
20       if ( args.length != 1 ) {
21          System.err.println( "Usage: java XMLInfo input.xml" );
22          System.exit( 1 );
23       }
24
25       try {
26
27          // create DocumentBuilderFactory
28          DocumentBuilderFactory factory =
29             DocumentBuilderFactory.newInstance();
30
31          // create DocumentBuilder
32          DocumentBuilder builder = factory.newDocumentBuilder();
33
34          // obtain document object from XML document
35          Document document = builder.parse(
36             new File( args[ 0 ] ) );
37
38          // get root node
39          Node root = document.getDocumentElement();
40
41          System.out.print( "Here is the document's root node:" );
42          System.out.println( " " + root.getNodeName() );
43
44          System.out.println( "Here are its child elements: " );
45          NodeList childNodes = root.getChildNodes();
46          Node currentNode;
```

Fig. 4.2 **XMLInfo** displays an XML document's element names and character data. (Part 1 of 3.)

```
47
48          for ( int i = 0; i < childNodes.getLength(); i++ ) {
49
50             currentNode = childNodes.item( i );
51
52             // print node name of each child element
53             System.out.println( currentNode.getNodeName() );
54          }
55
56          // get first child of root element
57          currentNode = root.getFirstChild();
58
59          System.out.print( "The first child of root node is: " );
60          System.out.println( currentNode.getNodeName() );
61
62          // get next sibling of first child
63          System.out.print( "whose next sibling is: " );
64          currentNode = currentNode.getNextSibling();
65          System.out.println( currentNode.getNodeName() );
66
67          // print value of first child's next sibling
68          System.out.println( "value of " +
69             currentNode.getNodeName() + " element is: " +
70             currentNode.getFirstChild().getNodeValue() );
71
72          // print name of next sibling's parent
73          System.out.print( "Parent node of " +
74             currentNode.getNodeName() + " is: " +
75             currentNode.getParentNode().getNodeName() );
76       }
77
78       // handle exception creating DocumentBuilder
79       catch ( ParserConfigurationException parserError ) {
80          System.err.println( "Parser Configuration Error" );
81          parserError.printStackTrace();
82       }
83
84       // handle exception reading data from file
85       catch ( IOException fileException ) {
86          System.err.println( "File IO Error" );
87          fileException.printStackTrace();
88       }
89
90       // handle exception parsing XML document
91       catch ( SAXException parseException ) {
92          System.err.println( "Error Parsing Document" );
93          parseException.printStackTrace();
94       }
95    }
96 }
```

Fig. 4.2 **XMLInfo** displays an XML document's element names and character data. (Part 2 of 3.)

```
c:\examples>java -classpath %CLASSPATH% com.deitel.jws1.xml.XMLInfo
chapter4/fig04_01_02/article.xml
Here is the document's root node: article
Here are its child elements:
#text
title
#text
date
#text
author
#text
summary
#text
content
#text
The first child of root node is: #text
whose next sibling is: title
value of title element is: Simple XML
Parent node of title is: article
```

Fig. 4.2 **XMLInfo** displays an XML document's element names and character data. (Part 3 of 3.)

Lines 28–29 create a new ***DocumentBuilderFactory***. The **DocumentBuilderFactory** is required to produce an appropriate **DocumentBuilder** object for the currently configured XML parser. JAXP can be configured to use many different XML parsers, such as the Apache Group's Xerces and IBM's XML4J. JAXP also has its own parser built in, which is used by default.

Line 32 uses the **DocumentBuilderFactory** class to create a ***DocumentBuilder*** object. Class **DocumentBuilder** provides an interface for loading and parsing XML documents. Lines 35–36 call the **DocumentBuilder** method ***parse*** to load and parse the XML document passed to the application as a command-line argument. The object returned contains the in-memory representation (i.e., the tree structure) of the XML document.

To access nodes in the tree structure, the root node must be retrieved. Line 39 retrieves the root node of the XML document by calling method ***getDocumentElement***. Line 42 retrieves the root node's name (i.e., **article**) by calling method ***getNodeName***. Line 45 calls method ***getChildNodes*** to obtain an ordered list (i.e., a **NodeList**) of **Element** nodes. The **NodeList** contains all the root node's child nodes. The first **Node** in the **NodeList** has an ordinal value of **0**, the next node has an ordinal value of **1** and so on. When passed as an argument to method ***item***, this ordinal value (or index) allows programmers to access any **Node** in the **NodeList**.

Line 48 calls **NodeList** method ***getLength*** to obtain the number of nodes in the list. Lines 48–54 retrieve each **Node**'s name in the **NodeList** by calling **NodeList** method **item**. This method is passed the index of the desired **Node** in the **NodeList**.

Line 57 calls method ***getFirstChild*** to obtain a **Node** reference to the first child **Node** of the **Node** referenced by **root**. Line 60 displays the name of the **Node** referenced by **currentNode**. Line 64 calls **Node** method ***getNextSibling*** to obtain a reference to the node's next sibling (e.g., the **Node** containing **title**). Line 65 displays the sibling

Node's name. Lines 68–70 print the name of the **Node** referenced by **currentNode** and the value of the first child **Node** referenced by **currentNode**. The **Node** method ***getNodeValue*** returns different values for different node types. In the tree structure created from the XML document in Fig. 4.1, the child node is of type **Text** (which represents character data), so **getNodeValue** returns the **String** contents of the **Text** node. We will explain **Node** types in greater detail in the next section.

4.3 DOM Components

In this section, we use Java, JAXP and some of the interfaces described in Fig. 4.3 to manipulate an XML document. Due to the large number of DOM interfaces and methods available, we provide only a partial list. For a complete list of DOM classes, interfaces and methods, browse the HTML documentation (**index.html** in the **api** folder) included with JAXP.

The **Document** interface represents the top-level node of an XML document in memory and provides methods for creating nodes and retrieving nodes. Figure 4.4 lists some **Document** methods. Interface **Node** represents any node type. Figure 4.5 lists some methods of interface **Node**.

Interface	Description
Document	Represents the XML document's top-level node (i.e., the *document root*), which provides access to all the document's nodes, including the root node.
Node	Represents a node in the XML document.
NodeList	Represents an ordered list of **Node**s.
Element	Represents an element node. Inherits from **Node**.
Attr	Represents an attribute node. Inherits from **Node**.
CharacterData	Represents character data. Inherits from **Node**.
Text	Represents a text node. Inherits from **CharacterData**.
Comment	Represents a comment node. Inherits from **CharacterData**.
ProcessingInstruction	Represents a processing-instruction node. Inherits from **Node**.
CDATASection	Represents a **CDATA** section. Inherits from **Text**.

Fig. 4.3 Some DOM interfaces.

Method Name	Description
createElement	Creates and returns an element node with the specified tag name.

Fig. 4.4 Some **Document** methods. (Part 1 of 2.)

Method Name	Description
`createAttribute`	Creates and returns an attribute node with the specified name and value.
`createTextNode`	Creates and returns a text node that contains the specified text.
`createComment`	Creates and returns a comment node that contains the specified text.
`createProcessingInstruction`	Creates and returns a processing-instruction node that contains the specified target and value.
`createCDATASection`	Creates and returns a **`CDATA`** section node that contains the specified text.
`getDocumentElement`	Returns the document's root element.
`appendChild`	Appends a child node.
`getChildNodes`	Returns a **`NodeList`** containing the node's child nodes.

Fig. 4.4 Some **`Document`** methods. (Part 2 of 2.)

Method Name	Description
`appendChild`	Appends a child node.
`cloneNode`	Duplicates the node.
`getAttributes`	Returns the node's attributes.
`getChildNodes`	Returns the node's child nodes.
`getNextSibling`	Returns the node's next sibling.
`getNodeName`	Returns the node's name.
`getNodeType`	Returns the node's type (e.g., element, attribute or text). Node types are described in greater detail in Fig. 4.6.
`getNodeValue`	Returns the node's value.
`getParentNode`	Returns the node's parent.
`hasChildNodes`	Returns **`true`** if the node has child nodes.
`removeChild`	Removes a child node from the node.
`replaceChild`	Replaces a child node with another node.
`setNodeValue`	Sets the node's value.
`insertBefore`	Appends a child node in front of a child node.

Fig. 4.5 **`Node`** methods.

Figure 4.6 lists some node types that may be returned by method **`getNodeType`**. Each type listed in Fig. 4.6 is a **`static final`** member of interface **`Node`**.

Node Type	Description
`Node.ELEMENT_NODE`	Represents an element node.
`Node.ATTRIBUTE_NODE`	Represents an attribute node.
`Node.TEXT_NODE`	Represents a text node.
`Node.COMMENT_NODE`	Represents a comment node.
`Node.PROCESSING_INSTRUCTION_NODE`	Represents a processing-instruction node.
`Node.CDATA_SECTION_NODE`	Represents a **`CDATA`** section node.

Fig. 4.6 Some **`Node`** types.

`Element` represents an element node. Figure 4.7 lists some **`Element`** methods. Figure 4.8 and Fig. 4.9 present a Java application that validates **`introduction.xml`** (Fig. 4.10) and replaces the text in its **`message`** element with the text **`New Changed Message!!`**.

Lines 27–28 create a new **`DocumentBuilderFactory`**. By passing the value **`true`** as an argument to method ***`setValidating`***, line 31 indicates that validating parser should be used. Line 34 creates a new **`DocumentBuilder`**. A **`CErrorHandler`** (Fig. 4.9) object is created in line 37 to define methods for handling exceptions related to parsing. Line 40 calls method **`parse`** to load and parse the XML document stored in the file **`introduction.xml`**. If parsing is successful, a tree structure representing **`introduction.xml`** is created. If parsing fails because the document is not valid, a ***`SAXException`*** is thrown. If the document is not well formed, a ***`SAXParseException`*** is thrown.

Method Name	Description
`getAttribute`	Returns the value of the specified attribute.
`getTagName`	Returns an element's name.
`removeAttribute`	Removes an element's attribute.
`setAttribute`	Changes the value of the attribute passed as the first argument to the value passed as the second argument.

Fig. 4.7 **`Element`** methods.

```
1  // Fig. 4.8 : ReplaceText.java
2  // Reads introduction.xml and replaces a text node.
3  package com.deitel.jws1.xml;
4
5  // Java core packages
6  import java.io.*;
```

Fig. 4.8 Simple example that replaces an existing text node. (Part 1 of 4.)

```
 7
 8  // Java extension packages
 9  import javax.xml.parsers.*;
10  import javax.xml.transform.*;
11  import javax.xml.transform.stream.*;
12  import javax.xml.transform.dom.*;
13
14  // third-party libraries
15  import org.xml.sax.*;
16  import org.w3c.dom.*;
17
18  public class ReplaceText {
19     private Document document;
20
21     public ReplaceText( String fileName )
22     {
23        // parse document, find/replace element, output result
24        try {
25
26           // obtain default parser
27           DocumentBuilderFactory factory =
28              DocumentBuilderFactory.newInstance();
29
30           // set parser as validating
31           factory.setValidating( true );
32
33           // obtain object that builds Documents
34           DocumentBuilder builder = factory.newDocumentBuilder();
35
36           // set error handler for validation errors
37           builder.setErrorHandler( new CErrorHandler() );
38
39           // obtain document object from XML document
40           document = builder.parse( new File( fileName ) );
41
42           // retrieve root node
43           Node root = document.getDocumentElement();
44
45           if ( root.getNodeType() == Node.ELEMENT_NODE ) {
46              Element myMessageNode = ( Element ) root;
47              NodeList messageNodes =
48                 myMessageNode.getElementsByTagName( "message" );
49
50              if ( messageNodes.getLength() != 0 ) {
51                 Node message = messageNodes.item( 0 );
52
53                 // create text node
54                 Text newText = document.createTextNode(
55                    "New Changed Message!!" );
56
57                 // get old text node
58                 Text oldText =
59                    ( Text ) message.getChildNodes().item( 0 );
```

Fig. 4.8 Simple example that replaces an existing text node. (Part 2 of 4.)

```
60
61                // replace text
62                message.replaceChild( newText, oldText );
63             }
64          }
65
66          // output Document object
67
68          // create DOMSource for source XML document
69          Source xmlSource = new DOMSource( document );
70
71          // create StreamResult for transformation result
72          Result result = new StreamResult( System.out );
73
74          // create TransformerFactory
75          TransformerFactory transformerFactory =
76             TransformerFactory.newInstance();
77
78          // create Transformer for transformation
79          Transformer transformer =
80             transformerFactory.newTransformer();
81
82          transformer.setOutputProperty( "indent", "yes" );
83
84          // transform and deliver content to client
85          transformer.transform( xmlSource, result );
86          System.out.println( "Output written to: " + fileName );
87       }
88
89       // handle exception creating DocumentBuilder
90       catch ( ParserConfigurationException parserException ) {
91          parserException.printStackTrace();
92       }
93
94       // handle exception parsing Document
95       catch ( SAXException saxException ) {
96          saxException.printStackTrace();
97       }
98
99       // handle exception reading/writing data
100      catch ( IOException ioException ) {
101         ioException.printStackTrace();
102         System.exit( 1 );
103      }
104
105      // handle exception creating TransformerFactory
106      catch (
107         TransformerFactoryConfigurationError factoryError ) {
108         System.err.println( "Error while creating " +
109            "TransformerFactory" );
110         factoryError.printStackTrace();
111      }
112
```

Fig. 4.8 Simple example that replaces an existing text node. (Part 3 of 4.)

```
113         // handle exception transforming document
114         catch ( TransformerException transformerError ) {
115            System.err.println( "Error transforming document" );
116            transformerError.printStackTrace();
117         }
118      }
119
120      public static void main( String args[] )
121      {
122         ReplaceText replace = new ReplaceText();
123      }
124   }
```

Fig. 4.8 Simple example that replaces an existing text node. (Part 4 of 4.)

Line 43 retrieves the **Document**'s root node. Line 45 calls method ***getNodeType*** to obtain the root node's type and determines whether the root node is of type **Element**.

Line 46 downcasts **root** from a superclass **Node** type to an **Element** derived type. As mentioned earlier, class **Element** inherits from class **Node**. By calling method ***getElementsByTagName***, lines 47–48 retrieve a **NodeList** containing all the **Node**s that contain the name **message** in the XML document. Line 50 determines whether the **NodeList** contains at least one item. The first **Node** in the **NodeList** is retrieved in line 51.

Lines 54–55 call **createTextNode** to create a **Text** node that contains the text **New Changed Message!!**. This node exists in memory independently of the XML document referenced by **document**—i.e., it has not been inserted into the document yet.

Lines 58–59 get the first child node of the **message** element (referenced by **Node message** in line 51), which is a **Text** node containing the text **Welcome to XML!**. Method **item** returns a **Node** that we downcast to **Text**. Line 62 calls method ***replaceChild*** to replace the **Node** referenced by the second argument with the **Node** referenced by the first argument. The XML document has now been modified—element **message** now contains the text **New Changed Message!!**.

Lines 66–86 output the modified **Document**. Line 69 creates a new **Source** object that wraps the modified **Document** object. Line 72 creates a new **StreamResult** object that passes an **OutputStream** as an argument, which, in this case, is **System.out**. Lines 75–76 create a new **TransformerFactory** by calling **static** method **newInstance**. Lines 79–80 create a new **Transformer** by calling method **newTransformer**. The object referenced by **transformer** writes the contents of the **document** to **System.out**. We discuss both **TransformerFactory** and **Transformer** in Chapter 5.

Line 82 sets the **"indent"** output property to **"yes"**, which causes the **Transformer** to add indentation when it produces the resulting document. Lines 90–91 handle a **ParserConfigurationException**, which can be thrown by **DocumentBuilderFactory** method **newDocumentBuilder**. Line 95 begins a **catch** block for a **SAXException**. This exception contains information about errors thrown by the parser.

Figure 4.9 presents **CErrorHandler.java**, which provides the implementation for handling errors thrown by the parser in **ReplaceText.java**. The programmer can provide an error handler, which is registered using method ***setErrorHandler*** (line 37 in Fig. 4.8).

```
1  // Fig. 4.9 : CErrorHandler.java
2  // Error Handler for validation errors.
3  package com.deitel.jws1.xml;
4
5  import org.xml.sax.ErrorHandler;
6  import org.xml.sax.SAXException;
7  import org.xml.sax.SAXParseException;
8
9  public class CErrorHandler implements ErrorHandler
10 {
11
12    // throw SAXException for fatal errors
13    public void fatalError( SAXParseException exception )
14       throws SAXException
15    {
16       throw exception;
17    }
18
19    // throw SAXParseException for errors
20    public void error( SAXParseException errorException )
21       throws SAXParseException
22    {
23       throw errorException;
24    }
25
26    // print any warnings
27    public void warning( SAXParseException warningError )
28       throws SAXParseException
29    {
30      System.err.println( "Warning: " + warningError.getMessage() );
31    }
32 }
```

Fig. 4.9 Class definition for **MyErrorHandler**.

Lines 5–7 of Fig. 4.9 **import** **ErrorHandler**, **SAXException** and **SAXParseException**. Interface ***ErrorHandler*** provides methods ***fatalError***, ***error*** and ***warning*** for *fatal errors* (i.e., errors that violate the XML 1.0 recommendation; parsing is halted), *errors* (i.e., invalid markup; parsing is not halted) and *warnings* (i.e., any problems other than fatal errors or errors; parsing is not halted), respectively. These methods are overridden in lines 13, 20 and 27, respectively. Fatal errors and errors are rethrown, and warnings are output to the standard error stream (**System.err**).

Figure 4.10 presents the XML document manipulated by the Java application in Fig. 4.8. Notice that the **message** element's text has been changed in the output.

```
1  <?xml version = "1.0" encoding = "UTF-8" standalone = "yes"?>
2
3  <!-- Fig. 4.10 : introduction.xml      -->
4  <!-- Simple introduction to XML markup -->
```

Fig. 4.10 Input document (**introduction.xml**) and output from **ReplaceText.java**. (Part 1 of 2.)

```
 5
 6  <!DOCTYPE myMessage [
 7     <!ELEMENT myMessage ( message )>
 8     <!ELEMENT message ( #PCDATA )>
 9  ]>
10
11  <myMessage>
12     <message>Welcome to XML!</message>
13  </myMessage>
```

```
c:\examples>java -classpath %CLASSPATH% com.deitel.jws1.xml.ReplaceT-
ext chapter4/fig04_08_09_10/introduction.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 4.10 : introduction.xml      -->
<!-- Simple introduction to XML markup -->
<myMessage>
   <message>New Changed Message!!</message>
</myMessage>
Output written to: chapter4/fig04_08_09_10/introduction.xml
```

Fig. 4.10 Input document (**introduction.xml**) and output from **ReplaceText.java**. (Part 2 of 2.)

4.4 Creating Nodes

The majority of the XML markup that has been presented up to this point has been "hand coded" (i.e., typed into an editor by a document author). By using the DOM API, XML documents can be created in an automated way through programming. Figure 4.11 lists a Java application that creates an XML document for a list of contacts.

```
 1  // Fig. 4.11 : BuildXml.java
 2  // Creates element node, attribute node, comment node,
 3  // processing instruction and a CDATA section.
 4  package com.deitel.jws1.xml;
 5
 6  // Java core packages
 7  import java.io.*;
 8
 9  // Java extension packages
10  import javax.xml.parsers.*;
11  import javax.xml.transform.*;
12  import javax.xml.transform.stream.*;
13  import javax.xml.transform.dom.*;
14
15  // third-party libraries
16  import org.xml.sax.*;
17  import org.w3c.dom.*;
18
19  public class BuildXml {
20     private Document document;
```

Fig. 4.11 Building an XML document with the DOM. (Part 1 of 4.)

```
21
22     public BuildXml( String fileName )
23     {
24        DocumentBuilderFactory factory =
25              DocumentBuilderFactory.newInstance();
26
27        // create new DOM tree
28        try {
29
30           // get DocumentBuilder
31           DocumentBuilder builder =
32              factory.newDocumentBuilder();
33
34           // create root node
35           document = builder.newDocument();
36        }
37
38        // handle exception thrown by DocumentBuilder
39        catch ( ParserConfigurationException parserException ) {
40           parserException.printStackTrace();
41        }
42
43        Element root = document.createElement( "root" );
44        document.appendChild( root );
45
46        // add comment to XML document
47        Comment simpleComment = document.createComment(
48           "This is a simple contact list" );
49        root.appendChild( simpleComment );
50
51        // add child element
52        Node contactNode = createContactNode( document );
53        root.appendChild( contactNode );
54
55        // add processing instruction
56        ProcessingInstruction pi =
57           document.createProcessingInstruction(
58               "myInstruction", "action silent" );
59        root.appendChild( pi );
60
61        // add CDATA section
62        CDATASection cdata = document.createCDATASection(
63           "I can add <, >, and ?" );
64        root.appendChild( cdata );
65
66        // write XML document to disk
67        try {
68
69           // create DOMSource for source XML document
70           Source xmlSource = new DOMSource( document );
71
```

Fig. 4.11 Building an XML document with the DOM. (Part 2 of 4.)

```
72            // create StreamResult for transformation result
73            Result result = new StreamResult(
74               new FileOutputStream( new File( fileName ) ) );
75
76            // create TransformerFactory
77            TransformerFactory transformerFactory =
78               TransformerFactory.newInstance();
79
80            // create Transformer for transformation
81            Transformer transformer =
82               transformerFactory.newTransformer();
83
84            transformer.setOutputProperty( "indent", "yes" );
85
86            // transform and deliver content to client
87            transformer.transform( xmlSource, result );
88         }
89
90         // handle exception creating TransformerFactory
91         catch (
92            TransformerFactoryConfigurationError factoryError ) {
93            System.err.println( "Error creating " +
94               "TransformerFactory" );
95            factoryError.printStackTrace();
96         }
97
98         // handle exception transforming document
99         catch ( TransformerException transformerError ) {
100           System.err.println( "Error transforming document" );
101           transformerError.printStackTrace();
102        }
103
104        // handle exception writing data to file
105        catch ( IOException ioException ) {
106           ioException.printStackTrace();
107        }
108     }
109
110     public Node createContactNode( Document document )
111     {
112
113        // create FirstName and LastName elements
114        Element firstName = document.createElement( "FirstName" );
115        Element lastName = document.createElement( "LastName" );
116
117        firstName.appendChild( document.createTextNode( "Sue" ) );
118        lastName.appendChild( document.createTextNode( "Green" ) );
119
120        // create contact element
121        Element contact = document.createElement( "contact" );
122
123        // create attribute
124       Attr genderAttribute = document.createAttribute( "gender" );
```

Fig. 4.11 Building an XML document with the DOM. (Part 3 of 4.)

```
125         genderAttribute.setValue( "F" );
126
127         // append attribute to contact element
128         contact.setAttributeNode( genderAttribute );
129         contact.appendChild( firstName );
130         contact.appendChild( lastName );
131
132         return contact;
133      }
134
135      public static void main( String args[] )
136      {
137        BuildXml buildXml = new BuildXml( args[ 0 ] );
138      }
139   }
```

```
c:\examples>java -classpath %CLASSPATH% com.deitel.jws1.xml.BuildXml
chapter4/fig04_11/mydocument.xml
```

Fig. 4.11 Building an XML document with the DOM. (Part 4 of 4.)

Lines 43–44 create an element named **root** (whose **Element** node is referenced by **root**) and append it to the document root (whose **Document** node is referenced by **document**). This node is the first node appended, so it is the root node of the document. Lines 47–48 create a comment node by calling method ***createComment*** and append the node as a child of the node referenced by **root**. Line 52 calls programmer-defined method **createContactNode** (line 110) to create a node for a **contact** element. We will discuss this method momentarily. Lines 56–58 create a processing-instruction node. The first argument passed to ***createProcessingInstruction*** is the target **myInstruction**, and the second argument passed is the value **action silent**. Line 59 appends the processing-instruction node to the root node. Lines 62–63 create a **CDATA** section node, which is appended to the **Node** referenced by **root** in line 64.

Line 110 defines method **createContactNode**, which returns a **Node**. This method creates a **Node** for the **contact** element. The new **Node** is returned and appended to the **Node** referenced by **root** in line 53. Lines 114–115 create **Element** nodes for elements **FirstName** and **LastName**, which have their respective child **Text** nodes created and appended on lines 117–118. Lines 124–125 create an **Attr** node for attribute **gender** by calling method ***createAttribute***. The value for this **Attr** node is set by calling method ***setValue***. Line 128 associates this **Attr** node with the **Element** node (i.e., **contact**) by calling method ***setAttributeNode***.

The XML document is written to disk in lines 67–88. Figure 4.12 lists the XML document (**myDocument.xml**) created by **BuildXml.java** (Fig. 4.11). [*Note*: We have modified this document for presentation purposes.]

```
1  <?xml version = "1.0" encoding = "UTF-8"?>
2
```

Fig. 4.12 Output (**myDocument.xml**) created by **buildXml.java**. (Part 1 of 2.)

```
<root>
   <!--This is a simple contact list-->
   <contact gender = "F">
      <FirstName>Sue</FirstName>
      <LastName>Green</LastName>
   </contact>
   <?myInstruction action silent?>
   <![CDATA[I can add <, >, and ?]]>
</root>
```

Fig. 4.12 Output (**myDocument.xml**) created by **buildXml.java**. (Part 2 of 2.)

4.5 Traversing the DOM

In this section, we demonstrate how to use the DOM to traverse an XML document. In Fig. 4.13, we present a Java application that outputs element nodes, attribute nodes and text nodes. This application takes the name of an XML document (e.g., **simpleContact.xml** in Fig. 4.14) as a command-line argument.

```
// Fig. 4.13 : TraverseDOM.java
// Traverses DOM and prints various nodes.
package com.deitel.jws1.xml;

// Java core packages
import java.io.*;

// Java extension packages
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;

// third-party libraries
import org.w3c.dom.*;
import org.xml.sax.*;

public class TraverseDOM {
   private Document document;

   public TraverseDOM( String file )
   {
      // parse XML, create DOM tree, call method processNode
      try {

         // obtain default parser
         DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
         factory.setValidating( true );
         DocumentBuilder builder = factory.newDocumentBuilder();

```

Fig. 4.13 Traversing the DOM. (Part 1 of 3.)

```
32            // set error handler for validation errors
33            builder.setErrorHandler( new CErrorHandler() );
34
35            // obtain document object from XML document
36            document = builder.parse( new File( file ) );
37            processNode( document );
38         }
39
40         // handle exception thrown by DocumentBuilder
41         catch ( ParserConfigurationException parserException ) {
42            parserException.printStackTrace();
43         }
44
45         // handle exception thrown by Parser
46         catch ( SAXException saxException ) {
47            saxException.printStackTrace();
48         }
49
50         // handle exception thrown when reading data from file
51         catch ( IOException ioException ) {
52            ioException.printStackTrace();
53            System.exit( 1 );
54         }
55      }
56
57      public void processNode( Node currentNode )
58      {
59         switch ( currentNode.getNodeType() ) {
60
61            // process Document root
62            case Node.DOCUMENT_NODE:
63               Document doc = ( Document ) currentNode;
64
65               System.out.println(
66                    "Document node: " + doc.getNodeName() +
67                    "\nRoot element: " +
68                    doc.getDocumentElement().getNodeName() );
69               processChildNodes( doc.getChildNodes() );
70               break;
71
72            // process Element node
73            case Node.ELEMENT_NODE:
74               System.out.println( "\nElement node: " +
75                                   currentNode.getNodeName() );
76               NamedNodeMap attributeNodes =
77                  currentNode.getAttributes();
78
79               for ( int i = 0; i < attributeNodes.getLength(); i++) {
80                  Attr attribute = ( Attr ) attributeNodes.item( i );
81
```

Fig. 4.13 Traversing the DOM. (Part 2 of 3.)

```
82                System.out.println( "\tAttribute: " +
83                   attribute.getNodeName() + " ; Value = " +
84                   attribute.getNodeValue() );
85             }
86
87             processChildNodes( currentNode.getChildNodes() );
88             break;
89
90          // process text node and CDATA section
91          case Node.CDATA_SECTION_NODE:
92          case Node.TEXT_NODE:
93             Text text = ( Text ) currentNode;
94
95             if ( !text.getNodeValue().trim().equals( "" ) )
96                System.out.println( "\tText: " +
97                   text.getNodeValue() );
98             break;
99       }
100   }
101
102   public void processChildNodes( NodeList children )
103   {
104      for ( int i = 0; i < children.getLength(); i++ )
105         processNode( children.item( i ) );
106   }
107
108   public static void main( String args[] )
109   {
110      if ( args.length < 1 ) {
111         System.err.println(
112            "Usage: java TraverseDOM <filename>" );
113         System.exit( 1 );
114      }
115
116      TraverseDOM traverseDOM = new TraverseDOM( args[ 0 ] );
117   }
118 }
```

Fig. 4.13 Traversing the DOM. (Part 3 of 3.)

Lines 21–55 define the class constructor for **`TraverseDOM`**, which takes the name of the file specified at the command line (i.e., **`simpleContact.xml`**) and loads and parses the XML document before passing it to programmer-defined method **`processNode`**.

Lines 57–100 define method **`processNode`**, which takes one **`Node`** parameter and outputs information about the **`Node`** and its child nodes. Line 59 begins a **`switch`** structure that determines the **`Node`**'s type. Line 62 matches the document's root node. This **`case`** outputs the document node (represented as **`#document`**) and processes its child nodes by calling method **`processChildNodes`** (lines 102–106). We will discuss method **`processchildNodes`** momentarily. Line 73 matches an **`Element`** node. This **`case`** outputs the element's attributes and then processes its child nodes by calling **`processChildNodes`**.

An element may contain any number of attributes. Lines 76–77 call method ***getAttributes*** to retrieve the list of **Attr** nodes for the **Element** node referenced by **currentNode**. The unordered[3] node list returned is assigned to the ***NamedNodeMap*** reference **attributeNodes**. If the **Element** node contains **Attr** nodes, the **for** loop (lines 79–85) outputs each **Attr** node's name and value.

Lines 91–92 match **CDATA** section nodes and **Text** nodes. These **case**s output the node's text content (lines 95–98).

Lines 102–106 define method **processChildNodes**, which takes one **NodeList** parameter and calls **processNode** on a node's child nodes. Each child node is retrieved by calling **NodeList** method **item** (line 105). Figure 4.14 lists the contents of **simpleContact.xml**, the XML document used by **TraverseDOM.java**.

```
1  <?xml version = "1.0" encoding = "UTF-8" standalone = "yes"?>
2
3  <!-- Fig. 4.14 : simpleContact.xml    -->
4  <!-- Input file for traverseDOM.java -->
5
6  <!DOCTYPE contacts [
7     <!ELEMENT contacts ( contact+ )>
8     <!ELEMENT contact ( FirstName, LastName )>
9     <!ATTLIST contact gender ( M | F ) "M">
10    <!ELEMENT FirstName ( #PCDATA )>
11    <!ELEMENT LastName ( #PCDATA )>
12 ]>
13
14 <contacts>
15    <contact gender = "M">
16       <FirstName>John</FirstName>
17       <LastName>Black</LastName>
18    </contact>
19 </contacts>
```

```
c:\examples>java -classpath %CLASSPATH% com.deitel.jws1.xml.Traverse-
DOM chapter4/fig04_13_14/simplecontact.xml
Document node: #document
Root element: contacts

Element node: contacts

Element node: contact
        Attribute: gender ; Value = M

Element node: FirstName
        Text: John

Element node: LastName
        Text: Black
```

Fig. 4.14 Sample XML document used by **TraverseDOM.java**.

3. By definition, an element's attributes are unordered.

In this chapter, we have introduced the DOM. In Chapter 5, we introduce the portion of the Extensible Stylesheet Language (XSL) called *XSL Transformations* (*XSLT*), which is used for converting XML data into other text-based formats, such as the Extensible HyperText Markup Language (XHTML).

4.6 Summary

XML documents, when parsed, are represented as a hierarchal tree structure in memory. This tree structure contains the document's elements, attributes, text, etc. The W3C provides a recommendation—called the XML Document Object Model (DOM)—for building tree structures in memory that represent XML documents. Any parser that adheres to this recommendation is called a DOM-based parser.

A DOM-based parser exposes (i.e., makes available) a programmatic library—called the DOM Application Programming Interface (API)—that allows data in an XML document to be accessed and manipulated.

In the DOM API, the primary DOM interfaces are **`Node`** (which represents any node in the tree), **`NodeList`** (which represents an ordered set of nodes), **`NamedNodeMap`** (which represents an unordered set of nodes), **`Document`** (which represents the document), **`Element`** (which represents an element node), **`Attr`** (which represents an attribute node), **`Text`** (which represents a text node) and **`Comment`** (which represents a comment node). Each of these interfaces provide methods for manipulating nodes.

4.7 Internet and World Wide Web Resources

`www.w3.org/DOM`
W3C DOM home page.

`www.w3schools.com/dom/default.asp`
The W3Schools DOM introduction, tutorial and links site.

`www.oasis-open.org/cover/dom.html`
The Oasis-Open DOM page provides an overview of the DOM.

`dmoz.org/Computers/Programming/Internet/W3C_DOM`
This site contains links to different locations and instructional matter on DOM.

`www.w3.org/DOM/faq.html`
Answers to frequently asked questions (FAQs) on DOM.