Supplement VI.D: Java and XML

For Introduction to Java Programming
By Y. Daniel Liang

This supplement introduces how to use Java to

- extract XML contents
- create XML documents
- validate XML documents

NOTE: Basic knowledge of XML is required for this supplement. To learn XML, please read Supplement VI.C, "XML."

0 Introduction

There are two popular APIs for processing XML documents: *SAX* (Simple API for XML) and *DOM* (Document Object Model). Both are designed to allow programmers to process XML documents. You can use either. This supplement introduces how to use the DOM to process XML documents.

1 Extracting XML Documents

Before introducing the entire API for processing, modifying, and creating XML using DOM, it is helpful to see how to extract XML documents using a short and simple example. Suppose an XML named test.xml is created in Listing 1.

Listing 1: test.xml

PD: Please add line numbers in the following code

```
<?xml version = "1.0"?>
<!-- XML document for students -->
<students>
  <student num = "1">
    <ssn>444111110</ssn>
    <firstname>Jacob</firstname>
    <mi>R</mi>
    <lastname>Smith</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>99 Kingston Street</street>
    <zipcode>31435</zipcode>
  </student>
  <student num = "2">
    <ssn>444111111</ssn>
    <firstname>John</firstname>
    <mi>K</mi>
    <lastname>Stevenson</lastname>
```

```
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>100 Main Street</street>
    <zipcode>31411</zipcode>
  </student>
</students>
```

Listing 2 gives a program to extract data from test.xml. The output of the program is shown in Figure 1.

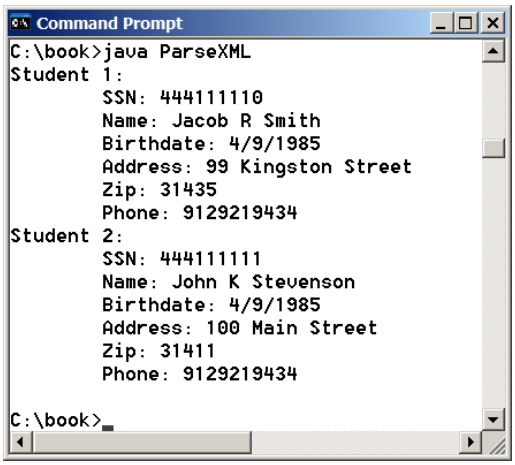

Figure 1
A Java program extracts contents from an XML file.

Listing 2: ParseXML.java (Extract contents from XML)
******PD: Please add line numbers in the following code******
<side remark Line 16: create a Document>
<side remark Line 19: create an XPath>
<side remark Line 21: evaluate XPath expression>

```
import javax.xml.parsers.*;
import javax.xml.xpath.*;
import java.io.*;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class ParseXML {
  public static void main(String[] args)
      throws ParserConfigurationException, IOException, SAXException,
             XPathExpressionException {
    DocumentBuilderFactory dbFactory =
      DocumentBuilderFactory.newInstance();
    DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();

    File file = new File("test.xml");
    Document document = docBuilder.parse(file);

    XPathFactory xpFactory = XPathFactory.newInstance();
```

```
    XPath xPath = xpFactory.newXPath();

    int numberOfStudents = Integer.parseInt(xPath.evaluate(
      "count(/students/student)", document));

    for (int i = 1; i <= numberOfStudents; i++) {
      System.out.println("Student " + i + ": ");
      System.out.println("\tSSN: " + xPath.evaluate(
        "/students/student[" + i + "]/ssn", document));
      System.out.println("\tName: " + xPath.evaluate(
        "/students/student[" + i + "]/firstname", document) + " " +
        xPath.evaluate("/students/student[" + i + "]/mi", document) +
        " " + xPath.evaluate("/students/student[" + i + "]/lastname",
        document));
      System.out.println("\tBirthdate: " + xPath.evaluate(
        "/students/student[" + i + "]/birthdate", document));
      System.out.println("\tAddress: " + xPath.evaluate(
        "/students/student[" + i + "]/street", document));
      System.out.println("\tZip: " + xPath.evaluate(
        "/students/student[" + i + "]/zipcode", document));
      System.out.println("\tPhone: " + xPath.evaluate(
        "/students/student[" + i + "]/phone", document));
    }
  }
}
```

Line 16 obtains an instance of Document. Document is an interface that describes the tree structure of an XML document. To obtain an instance of Document, you have to first create an object of DocumentBuilderFactory by invoking DocumentBuilderFactory.newInstance() (Lines 11-12) and then invoke newDocumentBuilder() on this object to create an object of DocmentBuilder (Line 13). Once you have a DocumentBuilder object, you can create an instance of Document from a file using the parse(File) method on a DocumentBuilder object (Line 16). Alternatively, you can create a Document from a URL or any InputStream object using the method parse(URL) or parse(InputStream).

A Document maintains the XML contents in a tree structure. Once you have created a Document, you can read its contents using XPath. XPath is introduced in Supplement VI.C, “XML.” To create an XPath object, you need to first create an XPathFactory by invoking XPathFactory.newInstance() (Line 18) and then invoke newPath() on this object to create an XPath. Once you have an XPath object, you can invoke its evaluate(String xPathExpression, Document doc) method to evaluate an XPath expression for the document. For example, in Lines 21-22

```
xPath.evaluate("count(/students/student)", document)
```

returns the number of the student nodes in the XML document.

In Lines 26-27,

```
xPath.evaluate("/students/student[" + i + "]/ssn", document);
```

returns the ssn of the ith student in the XML document. Note i is 1 for the first student.

2 Java XML API

The preceding section gave a simple example to demonstrate how to extract contents from an XML document. You got a glance of how to use the Document and XPath interfaces to extract data from XML. To add, delete, modify, and create XML documents, you need to know more about the Java XML API. The essential interfaces in the Java XML API are Node, Document, Element, CharacterData, Text, CDATASection, and Comment interfaces. Their relationship is shown in Figure 2.

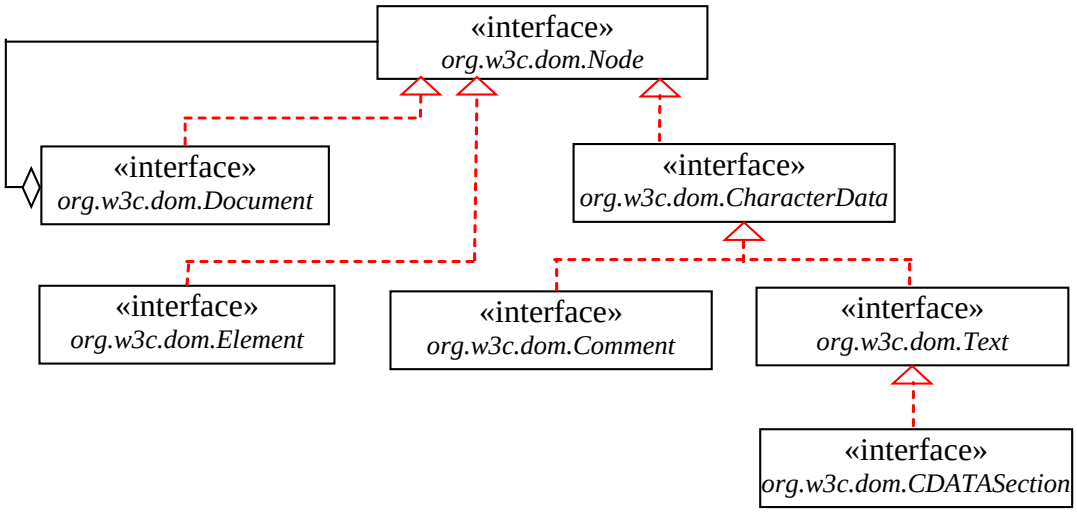

Figure 2

The Node, Document, Element, CharacterData, Text, CDATASection, and Comment interfaces are used to process nodes, elements, and text in an XML document.

The Node interface is the primary data type for the JAXP API. It represents a single node in the document tree and provides the operations for processing nodes, as shown in Figure 3. The getChildNodes() method returns a NodeList. For a NodeList, you can use its getLength() method to return the number of nodes in the list and use item(int index) to

return the item at the specified index.

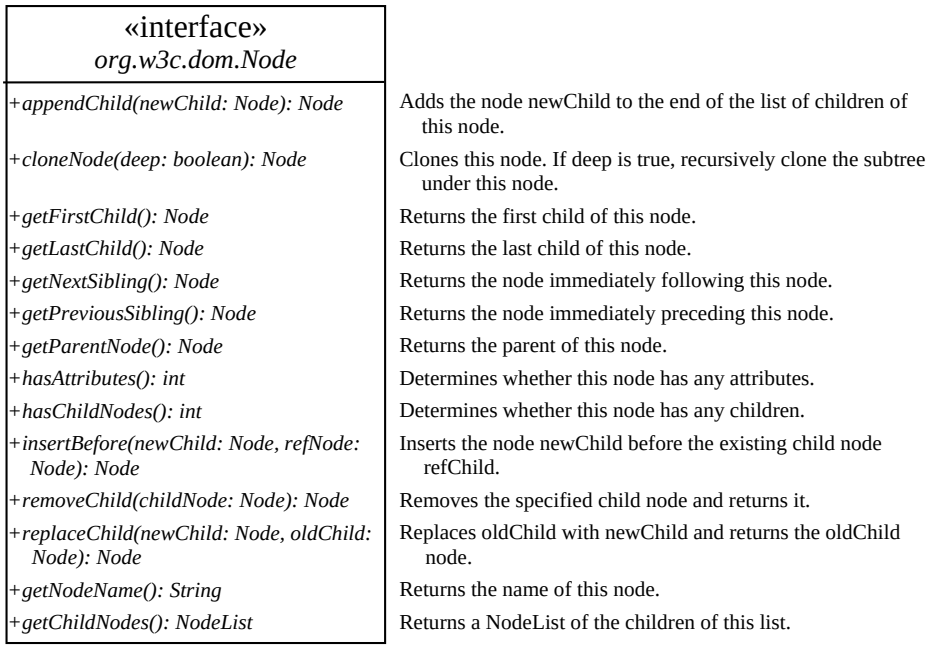

Figure 3

The Node interface defines the operations for manipulating nodes.

The Document interface extends Node and can contain nodes. The design pattern between Node and Document resembles the pattern between Component and Container. Container extends Component and contains Component. The Document interface provides the primary access to the document's data, as shown in Figure 4. You can use the methods createAttribute, createCDATASection, createComment, createElement, and createTextNode to create an attribute node, CDATA node, comment node, element node, and text node.

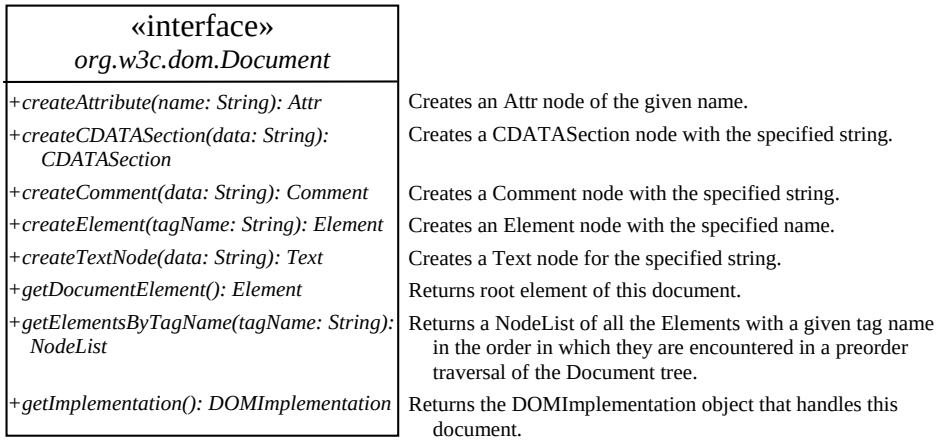

Figure 4

The Document interface describes the structure of the whole document.

The Element, Attr, Comment, and Text interfaces represent an XML element, attribute, comment, and text, respectively. All these interfaces are subinterfaces of Node. Their class diagrams are shown in Figures 5, 6, 7 and 8.

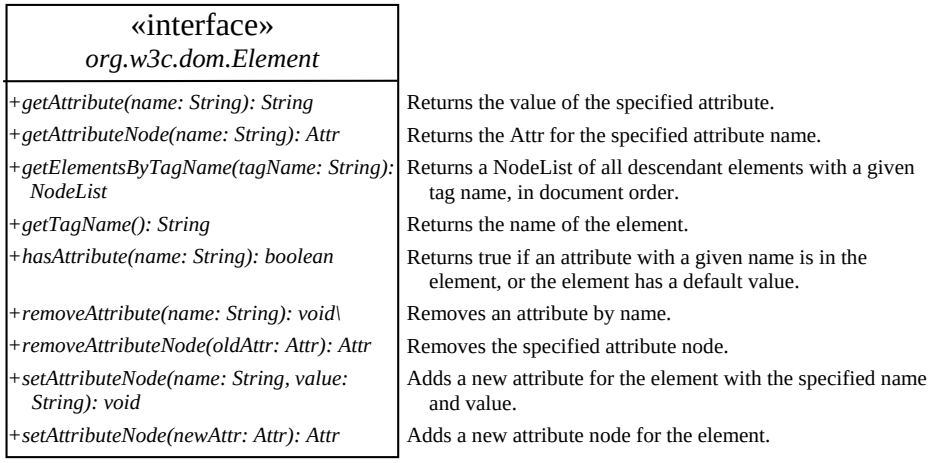

Figure 5

The Element interface represents an XML element.

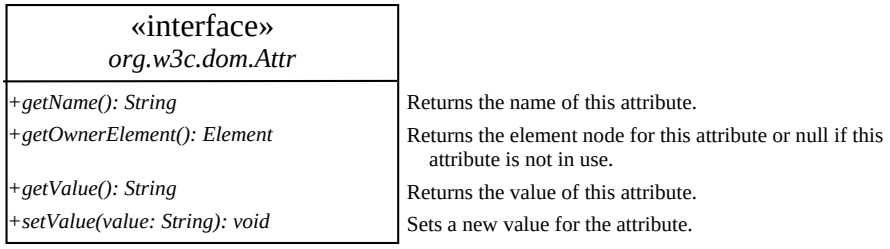

Figure 6

The Attr interface represents an attribute element.

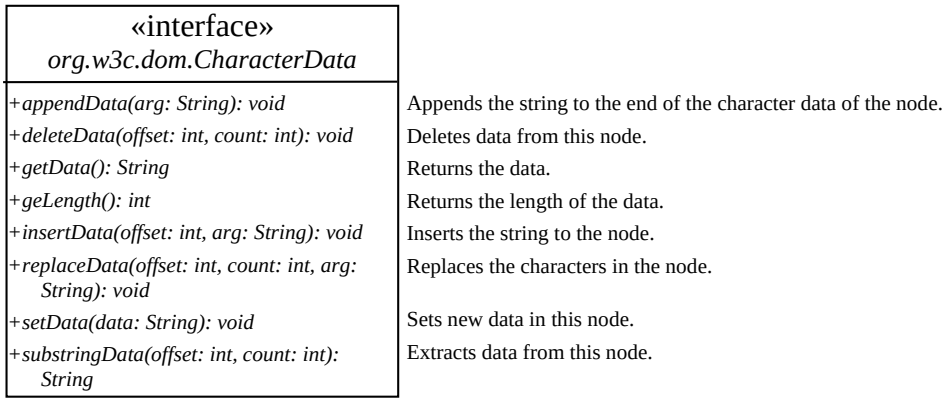

Figure 7

The CharacterData interface is the superinterface for Comment, Text, and CDATASection.

3 Creating XML Documents

Listing 1 showed how to extract data from an XML file. This section introduces how to create a new XML document. To extract data from an XML file, you have to create a Document object for the XML file and uses an XPath object's evaluate method to obtain XML contents. To create a new XML document, you first create DocumentBuilder in the same way as Listing 1 and then invoke the newDocument() method to create a Document, as follows:

```
DocumentBuilderFactory dbFactory =
  DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();
Document document = docBuilder.newDocument();
```

You can now use the Node, Element, Comment, and Text interfaces to add data into the document. Listing 3 gives an example to create the following simple XML document:

```
<?xml version = "1.0"?>
<!-- XML document for books -->
<books>
  <book num = "1">
    <title>Java Programming</title>
    <publisher>Prentice Hall</publisher>
  </book>
  <book num = "2">
    <ssn>XML Programming</ssn>
    <publisher>Prentice Hall</publisher>
  </book>
</books>
```

Listing 3: CreateXML.java (Create an XML file)

PD: Please add line numbers in the following code

<side remark Line 18: create a new Document>
<side remark Line 21: Comment node>
<side remark Line 22: add comment>
<side remark Line 25: create books element>
<side remark Line 26: add books element>
<side remark Line 29: create book element>
<side remark Line 30: set attribute>
<side remark Line 31: create title element>
<side remark Line 32: create text node>
<side remark Line 37: add child node>
<side remark Line 63: write XML as string>
<side remark Line 67: output to file>

```
import javax.xml.parsers.*;
import java.io.*;
import java.util.*;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import org.w3c.dom.Document;
import org.w3c.dom.Comment;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.ls.LSSerializer;

public class CreateXML {
  public static void main(String[] args) throws
      ParserConfigurationException, IOException {
    DocumentBuilderFactory dbFactory =
      DocumentBuilderFactory.newInstance();
    DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();
    Document document = docBuilder.newDocument();

    // Create a comment
    Comment comment = document.createComment("XML document for books");
    document.appendChild(comment);

    // Create the books element as the root
    Element booksElement = document.createElement("books");
    document.appendChild(booksElement);

    // Create elements for the first book
    Element bookElement = document.createElement("book");
    bookElement.setAttribute("num", "1");
    Element titleElement = document.createElement("title");
    Text titleText = document.createTextNode("Java Programming");
    Element publisherElement = document.createElement("publisher");
    Text publisherText = document.createTextNode("Prentice Hall");
```

```
    // Add elements for the first book
    booksElement.appendChild(bookElement);
    bookElement.appendChild(titleElement);
    titleElement.appendChild(titleText);
    bookElement.appendChild(publisherElement);
    publisherElement.appendChild(publisherText);

    // Create elements for the second book
    Element bookElement2 = document.createElement("book");
    bookElement2.setAttribute("num", "2");
    Element titleElement2 = document.createElement("title");
    Text titleText2 = document.createTextNode("Java Programming");
    Element publisherElement2 = document.createElement("publisher");
    Text publisherText2 = document.createTextNode("Prentice Hall");

    // Add elements for the second book
    booksElement.appendChild(bookElement2);
    bookElement2.appendChild(titleElement2);
    titleElement2.appendChild(titleText2);
    bookElement2.appendChild(publisherElement2);
    publisherElement2.appendChild(publisherText2);

    // Serialize a DOM document to an XML as a string
    DOMImplementation impl = document.getImplementation();
    DOMImplementationLS implLS =
      (DOMImplementationLS)impl.getFeature("LS", "3.0");
    LSSerializer serializer = implLS.createLSSerializer();
    String out = serializer.writeToString(document);
    System.out.println(out);

    // Write XML to a text file
    Formatter output = new Formatter("temp.xml");
    output.format("%s", out);
    output.close();
  }
}
```

Line 18 creates a new empty document from a DocumentBuilder object. Lines 21-22 create a Comment and add it to the document. Line 25 creates the <books> element and it is added to the document in Line 26.

Line 29 creates the <book> element. Line 30 adds an attribute for the <book> element. Line 31 creates the <title> element. Line 32 creates a text node for title. Line 33 creates the <publisher> element. Line 34 creates a text node for publisher. Lines 37-41 add the elements and text nodes to the document.

Lines 43-56 create the elements and texts for the second

book. Once you have created the document, you can write it to a file. There are several ways to accomplish it. You may use the LSSerializer interface, which can be created as follows (Lines 59-62):

```
DOMImplementation impl = document.getImplementation();
DOMImplementationLS implLS =
  (DOMImplementationLS)impl.getFeature("LS", "3.0");
LSSerializer serializer = implLS.createLSSerializer();
```

You can now use the writeToString(Document) method to write the XML document into an XML as a string (Line 63). This string is written to a text file using the Formatter class (Lines 67-69).

4 Validating XML Documents

When a DTD is included in an XML document, the program can use it to validate the XML document. To turn on validation, simply set the validating property to true in the DocumentBuilderFactory object. Listing 4 shows a simple Java program that validates the XML document in Listing 5.

Listing 4: ValidateXML.java (Validate an XML file)

PD: Please add line numbers in the following code

<side remark Line 11: validating true>

<side remark Line 14: parse XML with DTD>

```
import javax.xml.parsers.*;
import java.io.*;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;

public class ValidateXML {
  public static void main(String[] args)
      throws ParserConfigurationException, IOException, SAXException {
    DocumentBuilderFactory dbFactory =
      DocumentBuilderFactory.newInstance();
    dbFactory.setValidating(true);
    DocumentBuilder docBuilder = dbFactory.newDocumentBuilder();
    File file = new File("testdtd.xml");
    Document document = docBuilder.parse(file);
  }
}
```

Listing 5: testdtd.xml (XML file with DTD)

PD: Please add line numbers in the following code

<side remark Line 11: validating true>

```
<?xml version = '1.0'?>
<!DOCTYPE students [
  <!ELEMENT student (ssn, firstname, mi, lastname, birthdate,
```

```
    phone, street, zipcode)>
  <!ELEMENT ssn (#PCDATA)>
  <!ELEMENT firstname (#PCDATA)>
  <!ELEMENT mi (#PCDATA)>
  <!ELEMENT lastname (#PCDATA)>
  <!ELEMENT birthdate (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT street (#PCDATA)>
  <!ELEMENT zipcode (#PCDATA)>
  <!ELEMENT students (student+)>
]>
<students>
  <student>
    <ssn>444111110</ssn>
    <firstname>Jacob</firstname>
    <mi>R</mi>
    <lastname>Smith</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>99 Kingston Street</street>
    <zipcode>31435</zipcode>
  </student>
  <student>
    <ssn>444111111</ssn>
    <firstname>John</firstname>
    <mi>K</mi>
    <lastname>Stevenson</lastname>
    <birthdate>4/9/1985</birthdate>
    <phone>9129219434</phone>
    <street>100 Main Street</street>
    <zipcode>31411</zipcode>
  </student>
</students>
```

If the XML document does not conform to the DTD, errors will be reported. For example, if you mistyped firstname by firstName, the program will report this error.

Validation has another useful application. You can tell the parser to ignore the white space. By default, the parser converts all white spaces to text. This is confusing and wasteful. Often you have to write extra code to process the white space. You can invoke the setIgnoringElementContentWhitespace(true) method on a DocumentBuilderFactory object to tell the parser to ignore the white space, as follows:

```
dbFactory.setIgnoringElementContentWhitespace(true);
```