JDOM

JDOM is an open source, tree-based, pure Java API for parsing, creating, manipulating, and serializing XML documents.

JDOM, which is not an acronym, was written in and for Java. Interfaces, classes, and methods from *java.lang* and *java.util* are integrated with the JDOM package.

JDOM has no parser of its own; it depends on a SAX parser with a custom ContentHandler to parse a document and build a JDOM tree that models its content.

Installing JDOM

JDOM is not yet part of the Java distribution.

Follow these steps to make JDOM available on your CS linux account.

1. Download the file *jdom.jar* from the course ftp site.
2. Create a top-level directory, called *bin*, if you do not have one.
3. Move *jdom.jar* into the *bin* directory.
4. In your *.cshrc* or *.tcshrc* file, add a path specification at the end of the "setenv CLASSPATH" declaration.

   :/space/userId/bin/jdom.jar
5. Execute *source .cshrc* or *source .tcshrc* to have the change recognized in the current session.

Be careful not to use an older version of *jdom.jar*.

You can find more information about the JDOM distribution at

www.jdom.org

Creating a JDOM Tree

To parse an XML document and create a tree to be inspected and modified by JDOM, follow these steps.

1. Create an org.jdom.input.SAXBuilder object.

```
SAXBuilder builder = new SAXBuilder();
```

2. Create a File object that encapsulates the file containing the XML document.

```
File file = new File("xmlFile");
```

3. Create an org.jdom.Document object using the SAXBuilder method *build* that takes a File object as its parameter.

```
Document doc = builder.build(file);
```

SAXBuilder has *build* methods that allow other kinds of parameters. Check the JDOM documentation.

Investigating the Document

The class org.jdom.Document has a number of methods for exploring the JDOM tree that it represents.

To get the nodes at the top level, the children of the "document root", use the methods *getContent* and *getContentSize*.

```
List content = doc.getContent();
```

This method produces a java.util.List (actually an ArrayList) of objects of the class org.jdom.Content, which has six subclasses corresponding to the kinds of top-level nodes in a tree.

org.jdom.Element	org.jdom.ProcessingInstruction
org.jdom.Text	org.jdom.DocType
org.jdom.Comment	org.jdom.EntityRef

To get information about a possible DOCTYPE declaration in the XML document use the method *getDocType*.

```
DocType docType = doc.getDocType();
```

From this object we can extract information about the DOCTYPE declaration that was found.

Finally, the most important item, the root element of the document, can be located using the method *getRootElement*.

```
Element root = doc.getRootElement();
```

Processing the Content

To investigate the content of the JDOM tree we need to iterate through the List of Content objects, determining the type of each.

The simplest way to identify the various kinds of nodes is to utilize the seldom-used operation **instanceof**.

If we have a Comment, we can get its content using the method *getText*.

If we have a ProcessingInstruction, we can get its two parts using *getTarget* and *getData*.

Processing an Element

Since elements embody the structure and data of an XML document, JDOM has a large collection of methods for investigating elements.

General instance methods in Element

```
String getName()       // local name of the element
List getContent()      // all children nodes of element
String getText()       // all textual content of element
List getAttributes()   // a List of Attribute objects
List getChildren()     // all children nodes that are elements
```

Specific instance methods in Element

Content getContent(**int** k) // child in position *k*

String getAttributeValue(String n)
  // value of attribute with name *n*

String getAttributeValue(String n, String default)
  // return *default* if attribute is missing

Element getChild(String n)
  // child element with name *n*

List getChildren(String n)
  // all child element nodes with name *n*

String getChildText(String n)
  // textual content of child with name *n*

Instance Methods in Attribute

String getName() // name of attribute

String getValue() // value of attribute

Displaying a JDOM Tree

In the next program we display the information uncovered in a XML document by using the JDOM methods to perform a depth-first traversal of the tree.

The first document to process has no DOCTYPE statement and therefore no validation is possible.

This XML document is a variation of one that we saw in the DOM chapter.

It has three nodes at the top level: a comment, a processing instruction, and the root element.

Another comment is a child of an element.

File: rt.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- rt.xml -->
<?JDomParse usage="java JDomParse rt.xml"?>
<root>
  <child position="first">
     <name>Eileen Dover</name>
  </child>
  <child position="second">
     <name>Amanda Reckonwith</name>
  </child>
<!-- Could be more children later. -->
</root>
```

At each node, pertinent information about its values and content is printed.

Since white space can be overlooked easily in an XML tree, textual spaces in values and content are converted to plus signs (+). In addition new line characters become "[nl]". These modifications are handled in a method named *reveal*.

In this first version, the parse is executed with default settings. Afterward we consider changing some of the parser settings.

A parameter to the method *traverse* is used to provide spaces so that the structure of the tree is shown by indenting.

The two exceptions that may be thrown by the File object creation or by the call of the *build* method are declared in the header of the main method.

File: JDomParse.java

```
import org.jdom.input.SAXBuilder;
import org.jdom.JDOMException;
import org.jdom.Document;
import org.jdom.DocType;
import org.jdom.Element;
import org.jdom.Comment;
import org.jdom.ProcessingInstruction;
import org.jdom.Attribute;
import java.util.*;
import java.io.*;

public class JDomParse
{
   public static void main(String [] args)
                              throws JDOMException, IOException
   {
      SAXBuilder builder = new SAXBuilder();

      if (args.length > 0)
      {
         Document doc = builder.build(new File(args[0]));
         visit(doc);
      }
      else
         System.out.println("Usage: java JDomParse file.xml ");
   }
```

```
static void visit(Document doc)
{
   DocType docType = doc.getDocType();
   if (docType != null)
       System.out.println("DOCTYPE: " +
                                   docType.getSystemID());
   List content = doc.getContent();
   traverse("", content);
}

static void traverse(String indent, List content)
{
   for (Iterator it = content.iterator(); it.hasNext(); )
   {
      Object obj = it.next();

      if (obj instanceof Comment)
      {
         Comment comment = (Comment)obj;
         System.out.println(indent + "Comment: " +
                                   reveal(comment.getText()));
      }

      if (obj instanceof ProcessingInstruction)
      {
         ProcessingInstruction pi =
                                   (ProcessingInstruction)obj;
         System.out.print(indent + "Processing instruction: "
                                   + pi.getTarget());
         System.out.println("=>" + pi.getData());
      }
```

```
            if (obj instanceof Element)
            {
               Element elt = (Element)obj;
               System.out.println(indent + "Element: " +
                                                      elt.getName());
               System.out.print(indent + "Attributes: ");
               List attribs = elt.getAttributes();
               if (attribs.size() == 0)
                  System.out.print("None");
               for (Iterator at = attribs.iterator(); at.hasNext(); )
               {
                  Attribute attrib = (Attribute)at.next();
                  System.out.print(attrib.getName());
                  System.out.print("=");
                  System.out.print(attrib.getValue() + ";  ");
               }
               System.out.println()
               System.out.println(indent + "Text: "
                                                 + reveal(elt.getText()));
               traverse(indent+"    ", elt.getContent());
            }
         }
      }

      static String reveal(String string)
      {
         return string.replaceAll("\n", "[nl]").replaceAll(" ","+");
      }
   }
```

In the execution of JDomParse on the next page, notice that JDOM reports no node corresponding to the entire document.

Also the white space is handled differently than with DOM.

% **java JDomParse rt.xml**

```
Comment: +rt.xml+
Processing instruction:
               JDomParse=>usage="java JDomParser rt.xml"
Element: root
Attributes: None
Text: [nl]+++[nl]+++[nl][nl]
  Element: child
  Attributes: position = first;
  Text: [nl]+++++++[nl]+++
    Element: name
    Attributes: None
    Text: Eileen+Dover
  Element: child
  Attributes: position = second;
  Text: [nl]+++++++[nl]+++
    Element: name
    Attributes: None
    Text: Amanda+Reckonwith
  Comment: +Could+be+more+children+later.+
```

The text that is contained in the content of the *root* element comes in four pieces.

Between <root> and <child>:	[nl]+++
Between <child> and <child>:	[nl]+++
Between <child> and comment:	[nl]
Between comment and </root>:	[nl]

We explore two ways of eliminating the ignorable white space that was found and reported in this JDOM program.

First Method: Validation

In the first version, we turn to an XML document that has a DTD to support validation.

Look at the files *root.xml* and *root.dtd* found originally in the DOM chapter.

File: root.dtd

```
<!ELEMENT root (child*)>
<!ELEMENT child (name)>
<!ELEMENT name (#PCDATA)>
<!ATTLIST child position NMTOKEN #REQUIRED>
<!ENTITY last1 "Dover">
<!ENTITY last2 "Reckonwith">
```

File: root.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root SYSTEM "root.dtd">
<!-- root.xml -->
<?JDomParse usage="java JDomParse root.xml"?>
<root>
  <child position="first">
    <name>Eileen &last1;</name>
  </child>
  <child position="second">
    <name><![CDATA[<<<Amanda>>>]]> &last2;</name>
  </child>
<!-- Could be more children later. -->
</root>
```

To eliminate the ignorable white space we need to set two conditions for the SAXBuilder object.

To maintain flexibility with the program, it will have two execution modes determined by a second command line argument.

Without validation

```
        % java JDomParse rt.xml
```

With validation

```
        % java JDomParse root.xml true
```

Add the following code immediately after the line where the SAXBuider object is created.

```
    if (args.length > 1 && args[1].equals("true"))
    {
        builder.setValidation(true);
        builder.setIgnoringElementContentWhitespace(true);
    }
```

```
% java JDomParse root.xml true
DOCTYPE: root.dtd
Comment: +root.xml+
Processing instruction:
                  JDomParse=>usage="java JDomParser root.xml"
Element: root
Attributes: None
Text:
  Element: child
  Attributes: position = first
  Text:
    Element: name
    Attributes: None
    Text: Eileen+Dover
  Element: child
  Attributes: position = second
  Text:
    Element: name
    Attributes: None
    Text: <<<Amanda>>>+Reckonwith
  Comment: +Could+be+more+children+later.+
```

Note that the two entities in the DTD are resolved automatically and that the CDATA section is integrated with the rest of the textual content.

Second Method: Trim

The Element class has variations of the *getText* method that handle white space differently.

```
String getTextTrim()          // trim leading & trailing white space

String getTextNormalize()     // trim and reduce interior white
                              // space strings to a single space
```

When we recover the text content of an Element, we now use the method *getTextTrim* instead of *getText*.

If the resulting string is empty, just ignore it.

New Code

```
            String text = elt.getTextTrim();
            if (!text.equals(""))
               System.out.println(indent + "Text: " + reveal(text));

            traverse(elt.getContent());
         }
```

Now all ignorable white space disappears.

In addition, if the XML document has an empty element, it will have no text specification.

To test this second version of the JDOM parser, an empty element *empty* was inserted between the two *child* elements in *rt.xml*.

Resulting Output

```
Comment: +rt.xml+
Processing instruction:
            JDomParse=>usage="java JDomParser rt.xml"
Element: root
Attributes: None
  Element: child
  Attributes: position = first
    Element: name
    Attributes: None
    Text: Eileen+Dover
  Element: empty
  Attributes: None
  Element: child
  Attributes: position = second
    Element: name
    Attributes: None
    Text: Amanda+Reckonwith
Comment: +Could+be+more+children+later.+
```

Extracting Information from an XML Document

The next problem repeats the work done in the PhoneParser program in the DOM chapter.

Again we start with the files *phoneA.dtd* and *phoneA.xml* that describe a small number of phone information entries.

The problem uses the same two Java classes to collect the information extricated from the XML document.

class Entry

class Name

The JDOM version of the phone parser program has a similar structure to the one that use DOM, including the following three methods for building the Java objects.

```
private static List<Entry> getEntries(Element root)

private static Entry getEntry(Element e)

private static Name getName(Element e)
```

Observe in the following Java program how JDOM allows us to move directly through the elements of the tree using the methods shown below.

```
Element root = doc.getRootElement();

Element entriesE = root.getChild("entries");

List entryChildren = entriesE.getChildren("entry");

for (Iterator it = entryChildren.iterator(); it.hasNext(); )
{
   Element entryChild = (Element)it.next();
   Entry e = getEntry(entryChild);
   entries.add(e);
}
```

Then the *getEntry* method continues to navigate through the *entry* and *name* elements.

File: JPhoneParse.java

```
import org.jdom.input.SAXBuilder;
import org.jdom.JDOMException;

import org.jdom.Document;
import org.jdom.Element;

import java.util.*;
import java.io.*;
```

```
public class JPhoneParser
{
   public static void main(String [] args)
                           throws IOException, JDOMException
   {
      if (args.length != 1)
      {
         System.out.println(
                    "Usage: java JPhoneParser filename.xml");
         return;
      }

      SAXBuilder builder = new SAXBuilder();

      Document doc = builder.build(new File(args[0]));

      Element root = doc.getRootElement();

      List<Entry> entries = getEntries(root);

      for (Entry anEntry : entries)
         System.out.println(anEntry);
   }

   private static List<Entry> getEntries(Element root)
   {
      List<Entry> entries = new ArrayList<Entry>();
      Element entriesE = root.getChild("entries");
      List entryChildren = entriesE.getChildren("entry");
      for (Iterator it = entryChildren.iterator(); it.hasNext(); )
      {
         Element entryChild = (Element)it.next();
         Entry e = getEntry(entryChild);
         entries.add(e);
      }
      return entries;
   }
   private static Entry getEntry(Element e)
   {
      String gender, phone, city = null;

      Element element = e.getChild("name");
      Name name = getName(element);
      gender = element.getAttributeValue("gender", "");

      element = e.getChild("phone");
      phone = element.getText();

      element = e.getChild("city");     // null if element missing
      if (element != null)
         city = element.getText();

      return new Entry(name, gender, phone, city);
   }

   private static Name getName(Element e)
   {
      String first, middle = null, last;

      Element element = e.getChild("first");
      first = element.getText();

      element = e.getChild("middle");   // null if element missing
      if (element != null)
         middle = element.getText();

      element = e.getChild("last");
      last = element.getText();

      return new Name(first, middle, last);
   }
}
```

The output produced by this program is identical to that generated by the DOM version.

```
% java JPhoneParser phoneA.xml
Rusty Nail
335-0055
Iowa City

Justin Case
gender = male
354-9876
Coralville

Pearl E. Gates
gender = female
335-4582
North Liberty

Helen Back
gender = female
337-5967
```

Alternate *name* Method

Using the method *getChildText*, we can shorten the *name* method significantly.

```
private static Name getName(Element e)
{
   String first = e.getChildText("first");
   String middle = e.getChildText("middle");
   String last = e.getChildText("last");
   return new Name(first, middle, last);
}
```

The method *getChildText* returns **null** if the element is missing.

Another Approach

JDOM has instance methods in the classes Document and Element that retrieve collections of nodes at one time.

Iterator getDescendants()

This method creates an iterator that produces all of the descendants of the document or element in document order.

Iterator getDescendants(Filter filter)

This method creates an iterator that produces all of the descendants of the document or element after applying the Filter to return only those items that match the filter rule.

Filter is an interface, in the package *org.jdom.filter*, whose objects can be designed to match:

- only Elements
- only Comments
- Elements or Comments
- only Elements with a given name
- and other conditions.

Filter has two implementation in the package *org.jdom.filter*.

class ContentFilter

Can specify the acceptance of different JDOM node types such as Comment and Text.

class ElementFilter

Can select elements with certain names or namespaces.

Constructor

ElementFilter(String elementName)

Creates a Filter object that recognizes only elements with the given name.

File: JFilter.java

In this version we use a filter to retrieve all of the *entry* elements at one time.

The collection of *entry* elements is inspected using the iterator produced by the *getDescendants* method given a filter that restricts the search to those elements.

The methods *getEntry* and *getName* are written to extract the needed information as directly as possible.

```
import org.jdom.input.SAXBuilder;
import org.jdom.JDOMException;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.filter.Filter;
import org.jdom.filter.ElementFilter;
import java.util.*;
import java.io.*;

public class JFilter
{
   public static void main(String [] args)
                  throws IOException, JDOMException
   {
      if (args.length != 1)
      {
         System.out.println("Usage: java JFilter file.xml");
         return;
      }
      SAXBuilder builder = new SAXBuilder();

      Document doc = builder.build(new File(args[0]));

      List<Entry> entries = new ArrayList<Entry>();
```

```
        Filter filter = new ElementFilter("entry");
        Iterator dit = doc.getDescendants(filter);
        while (dit.hasNext())
        {
            Element element = (Element)dit.next();
            Entry e = getEntry(element);
            entries.add(e);
        }

        for (Entry anEntry : entries)
            System.out.println(anEntry);
    }

    private static Entry getEntry(Element e)
    {
        Element element = e.getChild("name");
        Name name = getName(element);
        String gender = element.getAttributeValue("gender", "");
        String phone = e.getChildText("phone");
        String city = e.getChildText("city");
        return new Entry(name, gender, phone, city);
    }

    private static Name getName(Element e)
    {
        String first = e.getChildText("first");
        String middle = e.getChildText("middle");
        String last = e.getChildText("last");
        return new Name(first, middle, last);
    }
}
```

The method *getChildText* returns **null** if the element is missing, which is a possibility with the *city* and *middle* element*s*.

The output is exactly the same as with the first version of this program.

Creating XML Documents Dynamically

As with DOM, JDOM has tools for building XML documents using information that has been created dynamically or obtained from a database.

Since most of the components of JDOM are classes instead of interfaces, we build the nodes of the tree using the constructors for the various kinds of Content.

Methods for Creating a JDOM Tree

The constructors for Document and the Content classes allow one to build empty nodes to be completed later or nodes that have content as they come into existence.

The methods shown here can be used to add content and attributes to elements. Other methods permit a programmer to alter and delete nodes and content.

Constructor in Document

    Document(Element root)

Constructor in Element

    Element(String name)

Instance Methods in Element

```
Element addContent(String text)
Element addContent(Content child)
Element setText(String text)
Element setAttribute(String name, String value)
```

These methods return the Element object that they are working on so that the methods can be "chained" together.

For example, in the following program the method *build* can be written as two commands.

```
Document build(List<Entry> entries)
{
  Element entriesE = createEntries(entries);

  return new Document(
    new Element("phoneNumbers").
            addContent(new Element("title").
                         addContent("Phone Numbers")).
            addContent(entriesE));
}
```

To convert the JDOM tree into an XML document, we need a few more methods that are found in classes from the package *org.jdom.output*.

The process of producing a stream of bytes (a file) from the tree is referred to as *serialization*.

Constructor In XMLOutputter

```
XMLOutputter()
```

Instance Methods in XMLOutputter

```
void setFormat(Format format)
void output(Document doc, OutputStream out)
void output(Document doc, Writer out)
```

Methods in Format

```
static Format getPrettyFormat()
static Format getCompactFormat()
Format setIndent(String spaces)
```

The class method *getPrettyFormat* returns a Format object that pretty-prints the XML document using 2-space indents.

The class method *getCompactFormat* returns a Format object that performs white space normalization.

File: JXMLBuilder.java

```
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.XMLOutputter;
import org.jdom.output.Format;
import java.io.*;
import java.util.*;

public class JXMLBuilder
{
  Document build(List<Entry> entries)
  {
    Element root = new Element("phoneNumbers");
    Element title = new Element("title");
    title.addContent("Phone Numbers");
    root.addContent(title);
```

```
   Element entriesEle = createEntries(entries);
   root.addContent(entriesEle);
   return new Document(root);
}

private Element createEntries(List<Entry> entries)
{
   Element e = new Element("entries");
   for (Entry anEntry : entries)
      e.addContent(createEntry(anEntry));
   return e;
}

private Element createEntry(Entry anEntry)
{
   Element e = new Element("entry");
   e.addContent(createName(anEntry.getName(),
                                anEntry.getGender()));
   e.addContent(new Element("phone").
                          setText(anEntry.getPhone()));
   String city = anEntry.getCity();
   if (city != null && !city.equals(""))
      e.addContent(new Element("city").setText(city));
   return e;
}

private Element createName(Name n, String gender)
{
   Element e = new Element("name");
   if (gender != null && !gender.equals(""))
      e.setAttribute("gender", gender);
   e.addContent(new Element("first").setText(n.getFirst()));
```

```
      String middle = n.getMiddle();
      if (middle != null && !middle.equals(""))
         e.addContent(new Element("middle").
                                          setText(middle));
      e.addContent(new Element("last").setText(n.getLast()));
      return e;
   }

   public static void main(String [] args) throws IOException
   {
      List<Entry> entries = new ArrayList<Entry>();

      entries.add(new Entry(new Name("Robin", "Banks"),
                                    "354-4455"));

      entries.add(new Entry(new Name("Forrest", "Murmers"),
                                    "male", "341-6152", "Solon"));

      entries.add(new Entry(new Name("Barb", "A", "Wire"),
                                    "337-8182", "Hills"));

      entries.add(new Entry(new Name("Isabel", "Ringing"),
                                    "female", "335-5985", null));

      JXMLBuilder xmlBuilder = new JXMLBuilder();
      Document doc = xmlBuilder.build(entries);

      FileWriter result = new FileWriter("newPhone.xml");
      XMLOutputter out = new XMLOutputter();
      Format f = Format.getPrettyFormat();
      out.setFormat(f);

      out.output(doc, result);
      out.output(doc, System.out);
   }
}
```

Output

```
% java JXMLBuilder
<?xml version="1.0" encoding="UTF-8"?>
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
    <entry>
      <name>
        <first>Robin</first>
        <last>Banks</last>
      </name>
      <phone>354-4455</phone>
    </entry>
    <entry>
      <name gender="male">
        <first>Forrest</first>
        <last>Murmers</last>
      </name>
      <phone>341-6152</phone>
      <city>Solon</city>
    </entry>
    <entry>
      <name>
        <first>Barb</first>
        <middle>A</middle>
        <last>Wire</last>
      </name>
      <phone>337-8182</phone>
      <city>Hills</city>
    </entry>
    <entry>
      <name gender="female">
        <first>Isabel</first>
        <last>Ringing</last>
      </name>
      <phone>335-5985</phone>
    </entry>
  </entries>
</phoneNumbers>
```