XML and Lobster Traps

James W. Cooper

Sitting here looking out over the Atlantic from Nantucket, it is sometimes hard to focus on important technical issues. For example, even though I write this on a gorgeous late summer day, it is certainly not the case that this weather will persist to the time that this column appears. But sometimes these remote climes can lead to interesting insights.

Last month, I showed you how you could write a set of classes to manipulate Windows-style ini-files to keep parameters that your program might need at execution time. While I didn't necessarily intend that column as flame-bait, I could hear some of you saying, "Idiot! Why doesn't he use XML?"

Well, XML is certainly the format of the millennium and it is much more powerful and flexible than ini-files. On the other hand, at the outset, it is harder to read and harder to use. So, let's recollect that ini-file of yore. It might look something like this:

```
[database]
connection=COM.ibm.db2.jdbc.app.DB2Driver
password=snarch
name=Customers
username=Jim

[files]
directory=d:\temp\
name=mydata.txt
```

We'd like to write an XML file containing this information. For one thing we wouldn't be limited to two levels of data, and we could expand the tree as deeply as we wanted. What we'd like to have, then, is a program that would manage an analogous XML file like this one:

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

  <paragraph name="database">
    <connection>COM.ibm.db2.jdbc.app.DB2Driver</connection>
    <user>jim</user>
    <name>Mydatabase</name>
    <password>snarch</password>
  </paragraph>

  <paragraph name="files">
    <directory>d:\temp</directory>
    <file>mydata.txt</file>
  </paragraph>

</profile>
```

How can we do this?

I started by downloading the xerces xml library from apache.org. This contains instances of both the SAX and DOM parsers and some useful example code. The packages it contains include:

org.w3c.dom, the Java implementation of the W3C's recommendation for a standard programmatic Document Object Model for XML

org.xml.sax, the event-driven Simple API for XML parsing

javax.xml.parsers, a factory implementation that allows you to configure and obtain a particular parser implementation

You may recall that the two parsers differ in that a SAX parser calls a method you provide for each XML element it reads. It is up to you which ones to keep and how you might want to store them internally. It is by definition a read-only parser. There is no way to modify an XML file using the SAX approach.

A DOM parser builds a Document Object Model tree in memory from the XML file it reads, and can keep the entire tree structure around for you to query or modify. You might think that the DOM Document object would then have some convenient write methods to allow you to add or change elements and put them back into the file. But it doesn't. It has a bunch of methods that seem like you could use them for them if only you could get your arms around them. The Apache example code even provides a DOM writer sample that maybe you could use to write out an XML document, but it really only writes on the one you just parsed. It is Really Hard to create a new XML document this way.

In fact it is like a lobster trap. Lobsters can one by one enter a lobster trap and try to eat the bait in the bait bag. They can even go back out again. But if a second lobster tries to enter, the first one backs up defensively and backs into the woven rope area that makes up the trap. I felt just as trapped by this system as that succulent crustacean. You could parse it, you could write it, but if you tried to create your own and write the file, something always seemed to go wrong. The Apache's org.ws3.dom.document class just doesn't do this XML creation very well.

So what to do? Well, as you may know, in addition to the Apache project and the Sun XML Java packages, there is another project called JDOM. This system was developed by Brett McLaughlin and Jason Hunter and has been developed in an opne-source fashion. The jdom.org website provides of XML Java class that are much easier to use, and allow you to parse XML into a DOM tree, and create and write out XML files with impunity. We aren't trapped by the funky older coding conventions that grew up when parsing existing files was much of the focus of the class design. As I write this, the JDOM code is in its seventh beta and slated for imminent release. I used that beta in writing this article without any difficulty at all.

Using JDOM

The JDOM classes make parsing XML a cinch. You can do it in just 3 or 4 lines of code:

```
DOMBuilder  builder = new DOMBuilder();
Document document = builder.build(new File(fullpath));
List dlist = document.getContent();
```

This gives us a classic List object made up of Element objects:

```
            Iterator iter = dlist.iterator ();
            Element el = (Element)iter.next ();
```

These can have children to any depth and every Element has a getName and getText method to fetch out the data:

```
            List ilist = el.getChildren();
            iter = ilist.iterator ();
            while (iter.hasNext ()) {
                el = (Element)iter.next ();
                System.out.println(el.getName()+ " " +el.getText());

}
```

Building XML files is just as easy. You just create a Document tree and write it out:

```
        buildDocument();
        FileWriter writer = new FileWriter(fullpath);
        XMLOutputter outputter = new XMLOutputter("  ", true);
        outputter.output(document, writer);
```

We'll get to how we build the document in a minute.

Classes for Managing Ini-Files

Since we want to emulate the ini-file format, we'll simply create a document tree with a series of "paragraph" tags, each with a variable number of elements, and we'll distinguish these upper level tags with attribute tags.

```
  <paragraph name="database">

  <paragraph name="files">
```

Then we simply scan through the top level saving these attribute names and build a tree of the subsidiary tags in a hash table. Our basic class element is an iniElement:

```
public class iniElement {
    private String tag, val;
    //saves one element of an XML ini-file
    //into a tag and a value
    public iniElement(Element element)  {
        tag = element.getName();
        val = element.getText();
    }
    public iniElement(String name, String value) {
        tag = name;
        val = value;
    }
    public String getName(){
        return tag;
    }
    public String getValue() {
        return val;
    }
}
```

We create a series of these elements in each paragraph and store them in a hash table:

```
public iniParagraph(Element element) {
        hash = new Hashtable();
        List lattr = element.getAttributes();
        //there will only be one attribute
        Iterator aiter = lattr.iterator ();
        while (aiter.hasNext ()) {
            Attribute attr = (Attribute)aiter.next ();
            name =attr.getValue();
            name =name.toLowerCase ();
        }
        //now get the children
        List ilist =   element.getChildren();
        Iterator iter =ilist.iterator ();
        while (iter.hasNext ()) {
            Element el = (Element)iter.next ();
            addElement(el.getName(), el.getText());
        }
    }
    //-------------
    public void addElement(String name, String value) {
        iniElement el = new iniElement(name, value);
        hash.put(name, el);
    }
```

and finally the ini-file itself consists of a hash table of named paragraph entries:

```
public IniFile(String fname) throws IOException, JDOMException {
        filename = fname;
        path ="";
        fullpath = filename;
        getParagraphs(fullpath);
    }
    //-------------------------------------------------------
    private synchronized void getParagraphs(String fullpath)
                  throws JDOMException{
        paragraphs = new Hashtable();
        File fl = new File(fullpath);
        if (fl.exists () ) {
            DOMBuilder  builder = new DOMBuilder();
            Document document = builder.build(new File(fullpath));
            List dlist = document.getContent();
            Iterator iter = dlist.iterator ();
            Element el = (Element)iter.next ();     //profile
            List ilist = el.getChildren();
            iter = ilist.iterator ();
            while (iter.hasNext ()) {
                el = (Element)iter.next ();
                iniParagraph para = new iniParagraph(el);
                paragraphs.put (para.getName(), para);
            }
        }
    }
```

Since we can add any number of elements to these hash tables before we write them back out: we have full flexibility in creating or adding to our XML file. We use the same iniFile method to add a new paragraph, tag and value as we did before.:

```
  /**Puts a profile entry into the ini file
    @param para -paragraph name
    @param tag - name of entry
    @param value - value of entry
    */
    public void putProfile(String pname,
      String tag, String value) /
      throws IOException {
            //get a paragraph if it already exists
        iniParagraph para = (iniParagraph)paragraphs.get (pname);

        if (para == null) {
            //create a new paragraph if needed
            para = new iniParagraph(pname);
            paragraphs.put (pname, para);
        }
        para.addElement (tag, value);
        //and rewrite entire ini-file
        putParagraphs(fullpath);
    }
```

Each time we create a new paragraph, we simply save the name and create a new hash table:

```
      public iniParagraph(String pname) {
        name = pname;
        hash = new Hashtable();
      }
      //-------------
    public void addElement(String name, String value) {
        iniElement el = new iniElement(name, value);
        hash.put(name, el);
    }
```

The addElement method just adds to that paragraphs hash table.

But the most significant part of the iniParagraph class is that it can easily construct the element tree to put back into the XML Document object :

```
//constructs the element tree for this paragraph
public Element getElement() {
        Element paragraph = new Element("paragraph");
        paragraph.setAttribute(new Attribute("name", name));
        Enumeration enum = hash.keys ();
        while (enum.hasMoreElements () ) {
            String key = (String)enum.nextElement ();
            iniElement iel = (iniElement)hash.get(key);
            String val = iel.getValue ();
            Element el = new Element(key);
            el.addContent(val);
            paragraph.addContent(el);
        }
        return paragraph;
    }
```

Then, all we have to do to save the XML file is to build the document up from these paragraph elements:

```
//builds the document tree from the paragraph elements
    private void buildDocument() {
        Element element =  new Element("profile");
        document = new Document (element);
        Enumeration enum = paragraphs.keys ();
        while (enum.hasMoreElements ()) {
            String key = (String)enum.nextElement ();
            iniParagraph para = (iniParagraph)paragraphs.get (key);
            element.addContent(para.getElement ());
        }
    }
```

Creating the Ini File

With these methods in place, we can now create the XML file in just a few lines of Java:

```
try {
    IniFile ini = new IniFile("test.ini");
    ini.putProfile ("Database", "name", "Mydatabase");
    ini.putProfile ("Database", "connection",
                  "COM.ibm.db2.jdbc.app.DB2Driver");
    ini.putProfile ("Database", "user", "jim");
    ini.putProfile ("Database", "password", "snarch");
    ini.putProfile ("Files", "directory", "d:\\temp");
    ini.putProfile ("Files", "file", "mydata.txt");
  } catch (IOException e) {
  } catch (JDOMException e) {
      System.out.println(e.getMessage());
  }
```

This makes using XML to store program parameters very easy indeed.

Writing XML for the Halibut

This simple example shows you how easy the JDOM classes make both reading and writing XML files: something that was head-poundingly hard to do using the previous org.w3c.dom classes. The JDOM classes have been developed to make all this a piece of hake. So, even if you spend some time scratching your head, at least you won't flounder.

References

1. You will find articles and documentation on this project at http://jdom.org
2. There is also an IBM DeveloperWorks article at http://www-106.ibm.com/developerworks/java/library/j-jdom/