JSR 172: XML Parsing Example

Input Data

In this example, we will parse the following XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person givenname="Bob" familyname="Fish">
    <phone type="mobile">+447234567890</phone>
  </person>
  <person givenname="Jim" familyname="Squid">
    <phone type="home">+441234567890</phone>
    <phone type="mobile">+447234567890</phone>
  </person>
  <person givenname="Dave" familyname="Octopus"/>
</people>
```

For the sake of the example, this document will be in the application's JAR, in the file JSR172Demo.xml, but it could equally come from some other source, such as an HttpConnection. The process is the same.

Note that I've kept all the tag and attribute names as lower case. This is not a requirement, but bear in mind that names in XML documents are case-sensitive. You must match the case in your code to the case in the document.

Data Classes

First, some classes to hold the data from the XML. The XML describes two kinds of object: Person and PhoneNumber.

```
/** A Person */
class Person {
    private String givenName;
    private String familyName;
    private Vector phoneNumbers;

    public Person(String givenName, String familyName) {
        this.givenName = givenName;
        this.familyName = familyName;
        phoneNumbers = new Vector();
    }

    public String getFullName() {
        return familyName + ", " + givenName;
    }

    public void addPhoneNumber(PhoneNumber pn) {
        phoneNumbers.addElement(pn);
    }

    public PhoneNumber[] getPhoneNumbers() {
        PhoneNumber[] numbers = new PhoneNumber[phoneNumbers.size()];
        phoneNumbers.copyInto(numbers);
        return numbers;
    }
}

/** A Phone Number */
class PhoneNumber {
    public static final int HOME   = 0;
    public static final int WORK   = 1;
    public static final int MOBILE = 2;

    private int type;
    private String number;

    public void setType(String type) {
        if (type.equals("home")) {
            this.type = HOME;
        } else if (type.equals("work")) {
            this.type = WORK;
        } else if (type.equals("mobile")) {
            this.type = MOBILE;
        } else {
            throw new IllegalArgumentException("unknown PhoneNumber type: " + type);
        }
    }

    public int getType() {
        return type;
    }

    public void setNumber(String number) {
        if (number.startsWith("+")) {
            this.number = number;
        } else {
            throw new IllegalArgumentException("PhoneNumber not in international format");
        }
    }

    public String getNumber() {
        return number;
    }
}
```

Writing an XML Handler

To process the XML, we must create a *Handler*. The Handler class must extend DefaultHandler, from the org.xml.sax.helpers package.

As the XML is processed, the parser will send events to the handler, to allow it to process different parts of the document. Our Handler subclass must override the appropriate event methods, in order to receive the information it wants. We're going to use three of these.

1. **startElement()**: this event occurs when the parser encounters an "opening" tag (one that starts with "<tagname...")
2. **endElement()**: this event occurs when the parser encounters an "ending" tag (one that looks like "</tagname>", or "<tagname.../>") - note that in the second case, both startElement() and endElement() events occur (in order) for the same tag
3. **characters()**: this event occurs for all text outside of "<>" tags, and includes all space characters

There are also other events (see the documentation), but these are the main three you will want to use.

Here's our Handler class:

```
/** Parser to process &lt;people...&gt; XML */
class PeopleHandler extends DefaultHandler {
    // this will hold all the data we read
    private Vector people;

    public PeopleHandler() {
        people = new Vector();
    }

    public Person[] getPeople() {
        Person[] persons = new Person[people.size()];
        people.copyInto(persons);
        return persons;
    }

    // VARIABLES TO MAINTAIN THE PARSER'S STATE DURING PROCESSING

    private Person currentPerson;
    private PhoneNumber currentPhoneNumber;

    // XML EVENT PROCESSING METHODS (DEFINED BY DefaultHandler)

    // startElement is the opening part of the tag "<tagname...>"
    public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {
        if (qName.equals("person")) {
            if (currentPerson != null) {
                // <person.../> cannot next inside itself
                throw new IllegalStateException("already processing a Person");
            }
            // "attributes" holds name and value pairs from inside the tag
            String givenName = attributes.getValue("givenname");
            String familyName = attributes.getValue("familyname");
            if (givenName == null || familyName == null) {
                throw new IllegalArgumentException("Person requires both givenname and familyname");
            }
            // create new Person object
            currentPerson = new Person(givenName, familyName);
        } else if (qName.equals("phone")) {
            if (currentPerson == null) {
                // <phonenumber.../> must appear inside a <person.../>
                throw new IllegalStateException("not processing a Person");
            }
            if (currentPhoneNumber != null) {
                // <phonenumber.../> cannot nest inside itself
                throw new IllegalStateException("already processing a PhoneNumber");
            }
            currentPhoneNumber = new PhoneNumber();
            String type = attributes.getValue("type");
            if (type == null) {
                throw new IllegalArgumentException("phone number is missing type");
            }
            currentPhoneNumber.setType(type);
        }
    }

    // endElement is the closing part ("</tagname>"), or the opening part if it ends with "/>"
    // so, a tag in the form "<tagname/>" generates both startElement() and endElement()
    public void endElement(String uri, String localName, String qName) throws SAXException {
        if (qName.equals("person")) {
            // add completed Person object to collection
            people.addElement(currentPerson);
            // we are no longer processing a <person.../> tag
            currentPerson = null;
        } else if (qName.equals("phone")) {
            // add completed PhoneNumber object to current Person
            currentPerson.addPhoneNumber(currentPhoneNumber);
            // we are no longer processing a <phone.../> tag
            currentPhoneNumber = null;
        }
    }

    // "characters" are the text inbetween tags
    public void characters(char[] ch, int start, int length) throws SAXException {
        // we're only interested in this inside a <phone.../> tag
        if (currentPhoneNumber != null) {
            // don't forget to trim excess spaces from the ends of the string
            String number = new String(ch, start, length).trim();
            currentPhoneNumber.setNumber(number);
        }
    }
}
```

Running the Parsing Process

Finally, a MIDlet class to make all this work. Note that the parsing is handled from a separate thread, because it may take some time and we don't want the event thread to be blocked. So, all the interesting stuff is in the run() method.

```
import java.io.*;
import java.util.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
```

```
import org.xml.sax.helpers.*;

public class JSR172Demo extends MIDlet implements CommandListener, Runnable {

    private Form form;

    public void startApp() {
        if (form == null) {
            form = new Form("JSR172Demo");
            form.addCommand(new Command("Exit", Command.EXIT, 0));
            form.setCommandListener(this);

            // start parsing
            (new Thread(this)).start();
        }
        Display.getDisplay(this).setCurrent(form);
    }

    public void pauseApp() {
        // empty
    }

    public void destroyApp(boolean must) {
        // empty
    }

    public void commandAction(Command c, Displayable d) {
        if (c.getCommandType() == Command.EXIT) {
            notifyDestroyed();
        }
    }

    // read XML and parse
    public void run() {
        try {
            // this will handle our XML
            PeopleHandler peopleHandler = new PeopleHandler();

            // get a parser object
            SAXParser parser = SAXParserFactory.newInstance().newSAXParser();

            // get an InputStream from somewhere (could be HttpConnection, for example)
            InputStream in = getClass().getResourceAsStream("/JSR172Demo.xml");
            // parse the XML data stream
            parser.parse(in, peopleHandler);

            // display the result
            Person[] people = peopleHandler.getPeople();
            if (people.length > 0) {
                for (int i = 0; i < people.length; i++) {
                    form.append(people[i].getFullName() + "\n");
                    PhoneNumber[] numbers = people[i].getPhoneNumbers();
                    if (numbers.length > 0) {
                        for (int j = 0; j < numbers.length; j++) {
                            form.append("* " + numbers[j].getNumber() + "\n");
                        }
                    } else {
                        form.append("* (no phone numbers)\n");
                    }
                }
            } else {
                form.append("(no people)\n");
            }
        } catch (Exception e) {
            form.append(e.toString());
        }
    }
}
```

I've put all the "import" statements in this class, so you can put all the classes in the same source file (JSR172Demo.java). If you want to split the classes into separate files, you'll need to add the appropriate imports to the other classes.

Another Technique

In the above example, each tag has represented an object. If you're handling XML in which there are may object attributes encoded as tags, with the values as text in between, you may need a slightly different approach.

For example, for XML like this:

```
<StockQuotes>
 <Stock>
  <Symbol>NOK</Symbol>
  <Last>12.76</Last>
  <Date>4/23/2010</Date>
  <Time>4:00pm</Time>
  <Change>-0.23</Change>
  <Open>12.70</Open>
  <High>12.76</High>
  <Low>12.57</Low>
  <Volume>50259424</Volume>
  <MktCap>47.317B</MktCap>
  <PreviousClose>12.99</PreviousClose>
  <PercentageChange>-1.77%</PercentageChange>
  <AnnRange>12.10 - 16.58</AnnRange>
  <Earns>0.00</Earns>
  <P-E>N/A</P-E>
  <Name>Nokia Corporation</Name>
 </Stock>
</StockQuotes>
```

We're just creating instances of one class (Stock). The Handler code will look something like:

```
// this is the object we're constructing
private Stock currentStock;
// this is the last text we read in characters()
private String lastCharacters;

// create any necessary objects on the opening tag
public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {
    if (qName.equals("Stock")) {
```

```
            currentStock = new Stock();
        }
    }

    // keep a record of any characters we see along the way
    public void characters(char[] ch, int start, int length) throws SAXException {
        lastCharacters = new String(ch, start, length).trim();
    }

    // on the closing tag, pass the last characters we saw to the object we're constructing
    public void endElement(String uri, String localName, String qName) throws SAXException {
        if (qName.equals("Symbol")) {
            currentStock.setSymbol(lastCharacters);
        }

        // ... and handle other tags
    }
```