Practicing Safer SAX

James W. Cooper

It's amazing what you can find in he back of your cupboard. Just last weekend, I found a half-bottle of an inexpensive1991 Merlot that had gotten mixed in with the salad oil and vinegar bottles. How appropriate: now I have some new red wine vinegar I didn't know I had! Well, we sometimes find old code lying around in the same way, and while it doesn't really spoil, it can get out of date.

For example, consider XML parsers. All of the common methods for parsing XML documents, including both a SAX and a DOM parser are built into Java 1.4, and this prompted me to rewrite some code I have lying around to use these classes.

Let's suppose we have a passel of documents that we want to do some computations on. Now, these documents could just be separate files, but if they are short documents, perhaps just document abstracts, you will get better performance if you just put all of them in one big file.

Now, what sort of document analysis might we be doing where we would scan through a bunch of abstracts. Depending on your computational bent, you might try to analyze each document for readability, for occurrence of specific domain terms, or sentence complexity. In this example, we'll just count the number of words in each document.

In order to write a program to parse XML documents, we need to agree on a format. There are only a few rules for valid XML. First, the top line of a file must declare its membership in XML-dom with text like that below:

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

Then, the entire file must be enclosed in some beginning and ending tag. Here we use the <coll> tag to indicate the beginning and end of a set of document abstracts"

```
<coll>
      text of many documents
</coll>
```

We will then divide the file into segments, each representing a single document, each separated with a tag:

```
<coll>
      <seqment>
            text of document 1
      </segment>

      <seqment>
            text of document 2
      </segment>
</coll>
```

Finally, we might want to mark the title of each document separately.

```
<coll>
      <seqment>
```

```
            <title>title of doc 1</title>
            text of document 1
      </segment>

      <seqment>
            <title>title of doc 2</title>
            text of document 2
      </segment>
</coll>
```

So the code we want to write should recognize the beginning of a new document, save the title, save the text and store away the title and size until it runs out of documents. Recall that there are two kinds of XML parsers: DOM parsers and SAX parsers. A DOM parser builds a complete tree of the all the XML tags in a document and keeps them all in memory at once, and a SAX parser calls the methods of a predefined class each time a tag begins or ends. Since we have no idea how many documents might be in this file, the DOM parser is risky: we might run out of memory. And, the SAX parser is actually easier to use in any case.

Using the SAX Parser

In Java 1.4, the SAX parser class is part of the javax.xml.parsers package. However, the SAXParser class is abstract and can't be instantiated. We get an actual parser from the SAXParserFactory class.

```
try {
      SAXParserFactory sFact = SAXParserFactory.newInstance();
      parser = sFact.newSAXParser();
} catch (SAXException e) {
      System.out.println(e.getMessage() );
}
```

Then all we have to do is call the parsers *parse* method to parse the document.

```
      String fName = dataPath + fileName;
      parser.parse(new File(fName), handler);
```

But wait a minute! What happens when we ask the SAX parser to begin parsing. Who gets told about the results? You do, because you have to provide the class that the parser calls when it finds these tags. Essentially you have to provide a class that extends the DefaultHandler class. This class is part of the org.xml packages, and so you specifically import these packages, which are included in the Java 1.4 distribution:

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

The DefaultHandler class contains about 20 empty methods. You must override any that you want to actually use.

For me, the minimum set of these methods is *startElement*, *endElement*, *characters*, and *endDocument*. Every time a new tag is encountered, the parser calls your handler class's *startElement* method and every time a tag ends, it calls your class's *endElement* method. However, the text that lies between the tags is not passed into these calls. Instead, all the text between tags is passed into calls to the *characters* method. This means that you have

to initialize the character buffer to receive characters each time a new startElement call is made. Further, since the number of characters between a start and an end element can be any size, the number of times the character method is called is also variable and unpredictable. You need to use the character method to add more characters to a buffer each time it is called. This can be done most efficiently using a StringBuffer class instead of a String class:

```
public void characters(char[] ch, int start, int length)
                  throws SAXException {
      buffer.append( ch);
}
```

So the overall strategy is

1. Initialize buffer on a startElement call.
2. Fill buffer on character calls.
3. copy text from buffer on endElement call.

Finally, on the endDocument call, you close the application. In this case we'll print out the documents we have been accumulating.

Using the SaxParser

The easiest way to write a simple example is to derive your main program from the DefaultHandler class:

```
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.*;
import javax.xml.parsers.*;
import java.util.*;
import java.io.*;

public ParseDocs() {
      docs = new Vector();
      try {
            SAXParserFactory sFact = SAXParserFactory.newInstance();
            parser = sFact.newSAXParser();
            String fName = dataPath + fileName;
            buffer = new StringBuffer();
            parser.parse(new File(fName), this);
      } catch (SAXException e) {
            System.out.println(e.getMessage() );
      } catch (ParserConfigurationException e) {
                              System.out.println(e.getMessage() );
      } catch (IOException e) {
                              System.out.println(e.getMessage() );
      }
}
```

Then, you can just override the 4 methods in this main program class. For the startElement call, we always need to initialize the buffer:

```
public void startElement(String uri, String localName,
      String qName, Attributes attrib) throws SAXException {
      buffer = new StringBuffer();
      if(qName.equals( "segment")) {
```

```
	//create new document each time
		doc = new Document("");
	}
}
```

In this simple program, we simply are creating instances of a Document object where we keep the title and the document length:

```
public class Document {
	private String title;   //document title
	private String id;      //document ID
	private int size;       //document size
	//-----
	public Document(String id) {
		this.id = id;
	}
	//-----
	public void setSize(int size) {
		this.size = size;
	}
	public int getSize() {
		return size;
	}
	//-----
	public void setTitle(String title) {
		this.title = title;
	}
	public String getTitle() {
		if (title.length() > 30) {
			return title.substring(0, 30);
		} else
			return title;
	}
	//-----
	public void setID(String id) {
		this.id = id;
	}
	public String getID() {
		return id;
	}
}
```

Now with that class defined, we just need to get the title and the complete document text whenever we get to the end of a document. We carry out both of these operations in the endElement method call:

```
public void endElement(String uri, String localName,
		String qName) throws SAXException {
	if(qName.equals( "segment")) {
		String buf = buffer.toString() ;
		StringTokenizer tok = new StringTokenizer(buf);
		doc.setSize( tok.countTokens() );
		docs.addElement( doc);
	}
	if(qName.equals( "title")) {
		title = buffer.toString().trim() ;
		doc.setTitle(title);
	}
```

```
}
```

As you see, we test for the value of the tag in the qName variable and execute the code for that variable. In both of the above cases, we use the characters that have accumulated in the buffer. In the case of the end of the <title> tag, we copy the text into the title variable and into the current document. If tag is the tag, we now have the entire text of the document in the character buffer and can compute the document's length in words. We store that value in the document and add that document to the docs Vector.

When all of the documents in the file have been processed, the parser calls the endDocument method, where we print out the document title and size:

```
public void endDocument() throws SAXException {
      Iterator iter = docs.iterator() ;
      int i = 1;
      while(iter.hasNext() ) {
            Document doc = (Document)iter.next() ;
            System.out.println(i++ +" "+doc.getSize() +
                              " " +doc.getTitle() );
            }
      }
}
```

Handling Attributes

An XML tag can have any number of attributes in the form of

```
name="value"
```

For example, we could identify each document within the file as having an ID and make this part of the segment tag:

```
<segment PMID="11960384">
```

To process these attributes, we look for a possible array of attributes in the startElement method:

```
public void startElement(String uri, String localName,
            String qName, Attributes attrib)
            throws SAXException {
      buffer = new StringBuffer();  //create a new buffer
      if(qName.equals( "segment")) {
            //see if there are any attributes
            int length = attrib.getLength() ;
            if(length > 0) {  //if there are save the first one
                  name =attrib.getQName( 0);
                  value =attrib.getValue( 0);   //this will be the PMID
                  doc = new Document(value);    //create a document
            }
      }
}
```

For a series of 4 Medline abstracts, the output of our program looks like this:

```
1 225 11960384 Activation of caspase-3 and cl
2 427 11960380 Bloom's syndrome protein respo
```

```
3 429 11960378 Constitutive activation of Sta
4 433 11960377 Overexpression of B-type cycli
```

Returning to the Nest

In this fairly straightforward example, we have not used any nested tags, although XML supports tags nested to any level. When tags are nested your processing of the accumulating character buffer may be more complicated. You might push the current character buffer onto a stack each time a new tag occurs and pop the old one each time a tag ends. You might also have to decide whether to combine the characters from the inner tag with those from the outer tags depending on their meaning.

Unspoken in this initial example is the requirement that each of these sub-documents must be valid XML. Not only must all the tags be terminated and nested properly, the documents must not contain and ampersands or greater-then or less-than symbols. These must be escaped as &amp. &lt. and &gt. If these requirements are not met you will get and must be able to handle XML parsing errors. We'll deal with that next time.

Conclusions

Parsing XML is easier than ever now that the SAX parser is built into Java 1.4. It makes the whole system easier to write. And if you find some old code somewhere, you may only have to change a couple of import statements and derive your handler class from DefaultHandler. At least your old code won't turn sour.