Handling SAX Errors

James W. Cooper

You're charging away using some great piece of code you wrote (or someone else wrote) that is making your life easier, when suddenly *plotz! boom!* The whole thing collapses in some useless Java error you don't understand and don't want to track down. Why do people write stuff like that? Well, because handling errors is a lot of trouble and it is much easier to leave them "as an exercise for the reader."

In his book on Extreme Programming, Kent Beck emphasizes that you should write the tests for each method before you write the method. Nice work if you can get it, but no one thought of that when they wrote the SAX parser and after the fact this is kind of tricky. But you do have to handle the inevitable errors that arise from incorrect XML. It's just that error recovery is necessarily limited.

SAX Redux

Last month, we discussed writing a simple SAX parser handler to find small documents in a large file of concatenated documents. To review, we decided that it would be more efficient to process a stream of short documents in a single file than to do all the I/O necessary to open and close each of the files in a large collection. So, we created a simple XML file format where we used XML tags to represent the documents and their titles:

```
<coll>
      <seqment>
            <title PMID="xxxx">title of doc 1</title>
            text of document 1
      </segment>

      <seqment>
            <title PMID="yyyy">title of doc 2</title>
            text of document 2
      </segment>
</coll>
```

Now, the SAX parser that is built into Java 1.4 is much the same as previous external parsers. You tell it where the routines are that extend the DefaultHandler class and start it to work. When it finds a new XML tag, it calls the *startElement()* method, and when it finds an end tag, it calls the *endElement()* method. In between, it makes one or more calls to the *characters()* method to pass on all the text it finds.

Now, expanding on the concepts of last month, we really want to create a class that extends the SAX DefaultHandler, and put the three methods named above in it. In addition, we can keep the document list in that class and ask for it when the parsing is done. The complete class is shown in Listing 1.

Listing 1 – A DocParser Class the extends the SAX DefaultHandler

```
public class DocParser extends DefaultHandler {
      private StringBuffer buffer;
      private String title;
      private Vector docs;
```

```
	private String name, value;
	private Document doc;

	/**
	 * Constructor for DocParser.
	 */
	public DocParser() {
		super();
		docs = new Vector();
	}
	//------
	//all characters between tags accumulate here
	//may be called any number of times between tags.
	public void characters(char[] ch,
		int start, int length)
				throws SAXException {
		buffer.append(ch);
	}
	//------
	//a tag has been started
	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 is the PMID
				doc = new Document(value); //create a document
			}
		}
	}
	//------
	//a tag has ended
	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);
			doc = null; //means there is not one in progress
		}
		if (qName.equals("title")) {
			title = buffer.toString().trim();
			doc.setTitle(title);
		}
	}
	//---------
```

```
	//return the list of documents
	public Vector getDocs() {
		if (doc != null) {
			docs.addElement(doc);
		}
		return docs;
	}
}
```

Using the DocParser Class

If everything worked without error, we would just use this class by creating an instance of a SAXParser and giving it a file to parse, and an instance of the DocParser to call when it finds each tag.

```
	SAXParserFactory sFact = SAXParserFactory.newInstance();
	parser = sFact.newSAXParser();
	String fName = dataPath + fileName;
	parser.parse(new File(fName), docParser);
	docs = docParser.getDocs() ;
```

The DocParser collects a Vector of Document objects, each containing a title and URL, and when the parser completes we just call the printDocumentList method and print out the list of documents.

```
private void printDocumentList() {
	//get the Vector of Documents
	if (docParser != null) {
		docs = docParser.getDocs();
		Iterator iter = docs.iterator();
		int i = 1;
		//print out the size, ID and title of each one
		while (iter.hasNext()) {
			Document doc = (Document) iter.next();
			System.out.println(
				i++ +" "
				+ doc.getSize() + " "
				+ doc.getID() + " "
				+ doc.getTitle());
		}
	}
}
```

The output of this method for our test data of 4 short document is:

```
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
```

Handling Errors

There are really only a few errors that the SAX parser can throw, and they are thrown when it encounters one kind of illegal XML or another. These exceptions are

- An IOException – if it cannot find or read the file.
- A SAXConfigurartionException – if the parser is configured wrong in some way.

- A SAXException, for any kind of illegal XML. You can get more information from the error messages, but can't recover from this.

But, other than exiting gracefully, what can we do with this information?

Well, in this particular case, we are accumulating a list of documents from the parser, and we really don't want to throw away all the prior documents because a later one is incorrect. So, we catch these exceptions and then go on to dump whatever data has been accumulated:

```
docParser = new DocParser();
try {
      SAXParserFactory sFact =
            SAXParserFactory.newInstance();
      parser = sFact.newSAXParser();
      String fName = dataPath + fileName;
      parser.parse(new File(fName), docParser);
} catch (SAXException e) {
      System.out.println("SAX error:"+e.getMessage());
} catch (ParserConfigurationException e) {
      System.out.println("Config error:"+e.getMessage());
} catch (IOException e) {
      System.out.println("IO error:"+ e.getMessage());
}
printDocumentList();
```

Note that we call the printDocumentList() method outside the exception handling block, so it is called regardless of whether an exception has been caught or not. We could just easily have written this inside a *finally* clause.

Just as critical to our success is this simple addition to our getDocs method in the DocParser class:

```
//return the list of documents
      public Vector getDocs() {
            if (doc != null) {
                  docs.addElement(doc);
            }
            return docs;
      }
```

It adds a partially accumulated document to the list if one has been started, but the parser has failed partway through the document. Since we carefully have initialized all the fields of the Document object to non-null in the constructor,

```
public Document(String id) {
            this.id = id;
            title = "";
            size = 0;
      }
```

we will always get back at least the id, even if parsing fails on the title.

Building Some Tests

Now that we have more or less bullet-proofed the code, let's see how we can arrange to test it. We'll start by putting the document parsing into a method by itself:

```
public void parseDocuments(String path) {
	docParser = new DocParser();
	try {
		SAXParserFactory sFact =
			SAXParserFactory.newInstance();
		parser = sFact.newSAXParser();
		parser.parse(path, docParser);
	} catch (SAXException e) {
		System.out.println("SAX error:" + e.getMessage());
	} catch (ParserConfigurationException e) {
		System.out.println("Config error:" + e.getMessage());
	} catch (IOException e) {
		System.out.println("IO error:" + e.getMessage());
	}
	printDocumentList();

}
```

My strategy for testing is to start with a legal XML file and make a few illegal changes and test it. To do this, we have to write a file each time with these changes, because the SAXParser expects a file, not a buffer. So we just create and write a temp file each time.

```
public void parse_a_doc(String buffer) {
	//write a temp file
	File tempFile = null;
	try {
		tempFile = File.createTempFile("sax", "tmp");
		FileWriter fw = new FileWriter(tempFile);
		fw.write(buffer);
		fw.close();
	} catch (IOException e) {
		System.out.println(e.getMessage());
	}
	//parse the documents and print the results
	parseDocuments(tempFile.getAbsolutePath());
	tempFile.delete();
}
```

For actual testing, I thought of 3 invalid XML cases to test.

1. An incorrect <?xml declaration.
2. An incomplete <title> tag.
3. An illegal character such as “&”.

So we put the entire file in a StringBuffer and make these modifications:

```
	String buffer = "";
//parse the original data file
	try {
		InputDocument doc =
			new InputDocument(dataPath + fileName);
		buffer = doc.getBuffer();
	} catch (IOException e) {
		System.out.println(e.getMessage());
	}
	parse_a_doc(buffer);
```

```
        //now try the same thing with an invalid XML header
              StringBuffer sbuf = new StringBuffer(buffer);
              int i = buffer.indexOf("?xml");
              sbuf.replace(i, i + 1, " ");
              parse_a_doc(sbuf.toString());

        //remove part of a <title> tag
              i = buffer.indexOf( "<title>");
              i = buffer.indexOf( "<title>", i+1);
              sbuf = new StringBuffer(buffer);
              sbuf.replace(i, i+7, "<title ");
              parse_a_doc(sbuf.toString() );

        //add an illegal & character
              i = buffer.indexOf( "<title>");
              i = buffer.indexOf( "<title>", i+1);
              sbuf = new StringBuffer(buffer);
              sbuf.insert(i+50," & ");

              parse_a_doc(sbuf.toString() );
```

Here are the results of these test cases, starting with the legal one:

```
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
SAX error:The markup in the document preceding the root element must be
well-formed.
SAX error:Attribute name "title" must be followed by the '=' character.
1 225 11960384 Activation of caspase-3 and cl
2 0 11960380
SAX error:The entity name must immediately follow the '&' in the entity
reference.
1 225 11960384 Activation of caspase-3 and cl
2 0 11960380
```

SAX is Even Safer

So just like children's toys and American cars, code still breaks a lot, but putting some tests into your code can be a great help in making sure you are really handling unexpected errors in an expected way. I actually made a number of modifications to my assumed-correct code before it worked as well as the final version in this article. So the work is really worth doing. You'll write better code if you write some tests to make sure you are handling errors thoroughly. You don't have to be utterly exhaustive in your testing, but hitting the main high points will definitely improve your code. Let's leave *plotz* and *boom* to Marvel comics!