CHAPTER 6

XML PROCESSING

XML is an important technology by anyone's standards. It powers many of the modern technologies and you have to know how to process XML if you want to work with software of any value in modern industry. In this chapter, we are going to talk about three different methods of parsing XML and turning it into a format that our programs can reason with. We begin with the most rudimentary approach.

6.1 Simple API for XML

Simple API for XML (or *SAX*) is the most basic (and oftentimes, most efficient) way of processing XML. It is an *event-based parser* – it starts processing the XML from top-down and generates events as it reads different portions of the file. Examples of events are `START_TAG` and `END_TAG`. Based on the different events (and their sequences), we can decide exactly what data is coming in to us. Let's take a look at an example to understand SAX.

SAX

event-based parser

6.1.1 A SAX Example

Let's consider an XML file that describes a mathematical expression. A basic version of this file (which we will save as `data/expression.xml`) looks like this:

```xml
<?xml version="1.0"?>
<expression>
  <expr func="+">
    <expr func="*">
      <const val="2" />
      <const val="3" />
    </expr>
    <expr func="-">
      <const val="4" />
      <const val="5" />
    </expr>
    <const val="6" />
  </expr>
</expression>
```

This file corresponds to the expression $(2*3)+(4-5)+6$. It's very verbose and if we want to do something useful with this file (such as calculate the result), we need to be able to represent it in some meaningful Java object format.

The first step in doing that would be to define a class `Expresion` in our project.

```java
package org.csrdu.java.ex.xml.expression.model;

import java.util.Vector;

public class Expression {
  public String function = "";
  public Vector operands;

  public Expression() {
    operands = new Vector();
  }

  public void printExpression(int level) {
      System.out.println(nSpaces(level * 2) + "Function:  [" + function + "]");

      for (Object i : operands) {
        if (i instanceof Integer)
          System.out.println(nSpaces(level * 2 + 2) + i);
        else
          ((Expression)i).printExpression(level+1);
      }
  }

  public String nSpaces(int length) {
    StringBuffer outputBuffer = new StringBuffer(length);
    for (int i = 0; i < length; i++) {
      outputBuffer.append(" ");
    }
    return outputBuffer.toString();
  }
}
```

The class itself is very simple. It has a field called `function` to store the function name of the expression and a `Vector` of `Expression` objects as the operands. This is needed as the operands can be both constants and other expressions. In case one of the operands is a constant, the `function` field is ignored and the only `operand` will be an `Integer`.

The `printExpression()` function simply outputs the expression in a properly indented format (the indentation coming from the `nSpaces()` function.

model *controller* Now that we have a *model*, we can define a *controller* for it that reads the data from the XML file and creates an instance of the expression in Java object code.

```java
package org.csrdu.java.ex.xml.expression.controller;

import org.csrdu.java.ex.xml.expression.model.Expression;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import java.io.*;
import java.util.*;

public class ExpressionParser extends DefaultHandler {

  Stack<Expression> st;

  Expression curExpr;

  public ExpressionParser() {
    st = new Stack<Expression>();
  }

  // Override methods of the DefaultHandler class
  // to gain notification of SAX Events.
  //
  // See org.xml.sax.ContentHandler for all available events.
  //
  public void startElement(String namespaceURI, String localName,
  String qName, Attributes attr) throws SAXException {
    if (localName.equals("expr")) {
      // we have a new expression.
      // push the old one on the stack (if any)
      if (curExpr != null)
      st.push(curExpr);

      // We need to create a new expression.
      curExpr = new Expression();

      // and set the function for this expression
      curExpr.function = attr.getValue("func");

    } else if (localName.equals("const")) {
      // we have a constant. Let's add it to the expression
      // first get the value of the constant
      Integer val = Integer.parseInt(attr.getValue("val"));
      curExpr.operands.add(val);
    }
  }

  public void endElement(String namespaceURI, String localName, String qName)
    throws SAXException {
    if (localName.equals("expr")) {
      // End of current expression
      // save the expression just ended
      Expression endedExpr = curExpr;
      // pop the current expression from the stack
      if (!st.empty()) {
        curExpr = st.pop();
        // add ended expression to parent
        curExpr.operands.add(endedExpr);
      }
    }
    // we can ignore the end of const tag
  }

  public static void main(String[] argv) {
    try {
      // Create SAX parser...
      XMLReader xr = XMLReaderFactory.createXMLReader();

      // Set the ContentHandler...
      ExpressionParser exprs = new ExpressionParser();
      xr.setContentHandler(exprs);

      // Parse the file...
      xr.parse(new InputSource(new FileReader("data/expression.xml")));
      exprs.curExpr.printExpression(0);

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Let's begin the explanation by describing the `main()` function. It sets up the `XMLReader` which is simply a SAX *parser*. We then create an instance of our `ExpressionParser` class and parse it to the XMLReader as the *handler*. This means that whenever the parser finds an "important piece" of the XMl, it will generate an event which will be handled by our handler. This allows our code to be notified whenever an XML event (such as start of tag) occurs.

parser

handler

Algorithm 1 The `startElement` event

```
1: if event source = expr then
2:    push old expression on stack (if any)
3:    create a new expression object
4:    set function of new expression to attribute 'func'
5: else if event source = const then
6:    add attribute 'val' as operand to current expression
7: end if
```

Algorithm 2 The `endElement` event

```
1: if event source = expr then
2:    save the expression just ended as endedExpr
3:    if stack is not empty then
4:       pop the expression at the top of stack as curExpr
5:       add endedExpr as an operand of curExpr
6:    end if
7: end if
```

Plain Old Java Object (POJO)

We then parse the XML file (`data/expression.xml`). This will convert our XML file into a *Plain Old Java Object (POJO)*. We can then print it out or do whatever we need to do with it.

All we have to understand now is how our `ExpressionHandler` handles the events. This is covered in two functions:

startElement()

This function defines the majority of our logic. The algorithm for handling the *startElement* event is given in Algorithm 1. We first check if the event was raised for `expr` tag. If so, we know that we are starting a new expression. So, we push the old one on the stack (if one exists) and create a new expression tag. Any constant we see from now on will be an operand of this new expression. We also get the `func` attribute of the expression and set the function name equal to it.

If, on the other hand, we have this event raised for `const`, we know that we got a constant and we can take it's `val` attribute and set it as an operand of the current expression.

The other half of this logic comes into place when we look at what happens when a tag ends.

endElement()

When the `const` tag ends, we don't care because it's can't be a parent tag. What we are interested in is when the `expr` tag ends (cf. Algorithm 2). When that happens, we know that the current expression has ended. We need to pop the (parent) expression that we pushed on the stack and then add the expression that just ended to the parent. And that takes care of the whole logic!

If you have trouble understanding, Figure 6.1 might be of help. Refer to this in conjunction with the algorithm and the code.

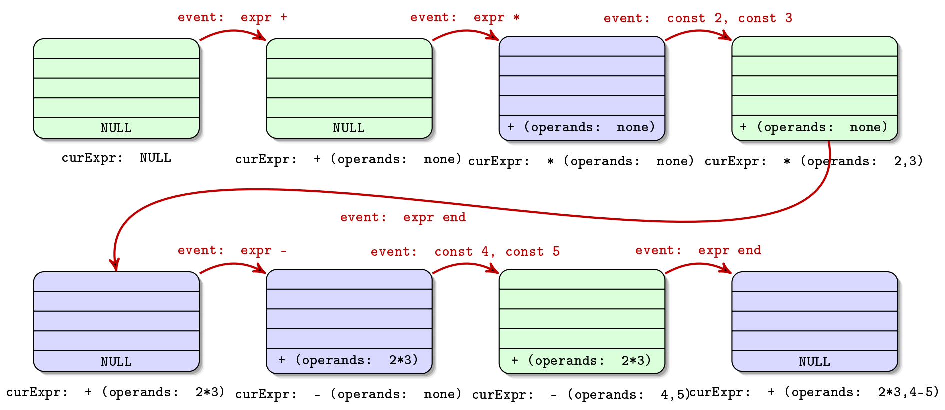

Figure 6.1: SAX Events Dry Run

6.1.2 Limitations of SAX

SAX is often very efficient but it has a major limitation. It does not allow random access. You have to parse the whole file to get to a particular piece of information. This becomes very problematic if you have a complicated file and you want to retrieve a particular piece of information from it. One way of solving this problem is through the use of a tree model of XML parsing. We discuss one of these below.

6.2 Document Object Model

Document Object Model (more commonly known as *DOM*) is a *tree-based parser* for XML. It allows you to traverse the XML document as if it were a tree (which it really is). The best way to understand DOM is by looking at a simple example. We are going to parse the same expression file that we parsed using SAX. Let's first take a look at the code.

DOM
tree-based parser

```java
package org.csrdu.java.ex.xml.expression.controller;

import javax.xml.parsers.DocumentBuilderFactory;
/* some imports hidden */

public class ExpressionParserDom {

  public static void main(String argv[]) {
    try {
      File file = new File("data/expression.xml");
      DocumentBuilderFactory dbFactory = DocumentBuilderFactory
      .newInstance();
      DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
      Document doc = dBuilder.parse(file);
      doc.getDocumentElement().normalize();

      Element rootElement = doc.getDocumentElement();
      NodeList nodes = rootElement.getChildNodes().item(1).getChildNodes();

      for (int i = 0; i < nodes.getLength(); i++) {
        Node node = nodes.item(i);

        if (node instanceof Element) {
          // a child element to process
          Element child = (Element) node;
          if (child.getTagName() == "expr") {
            String attribute = child.getAttribute("func");
            System.out.println("Expression with function ID "
                + attribute + " found.");
          } else if (child.getTagName() == "const"){
            String attribute = child.getAttribute("val");
            System.out.println("Constant with value "
                + attribute + " found.");
          }
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

document builder factory

First, we create what is called a *document builder factory*. This object can build a document by parsing a file (or string). We then normalize the document to get it into a standardized format. After that, it's fairly simple. We can get the root element of the XML file (using the `getDocumentElement()` function). This returns an object of type `Node`. All tags and text in DOM are *nodes*. When we have a node object, we can get its child nodes using the `getChildNodes()` function. This returns an *iterable* collection. We can loop over this collection and use the different functions.

nodes

iterable

6.3 XML Pull Parser

XML Pull Parser API

One of the more recent parsers has been proposed in the form of *XML Pull Parser API* [16]. It's a fast implementation that works (at least on the face), similar to SAX. Instead of relying on the push parsing of SAX, the XML Pull Parsers actively poll the document for events. This leads to a faster parsing performance. The XML Pull API is used extensively in the Android framework. However, an implementation of this API is not available in vanilla JREs. In this example, we will use the reference implementation[1] of this API provided by the kXML project [8].

Below, we provide code for parsing our expression XML through this pull parser.

```
package org.csrdu.java.ex.xml.expression.controller;

import java.io.FileReader;
import java.io.IOException;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;
import org.xmlpull.v1.XmlPullParserFactory;

/**
* An example of an application that uses XMLPULL V1 API.
*
* @author <a href="http://www.extreme.indiana.edu/~aslom/">Aleksander
*         Slominski</a>
```

[1]You can download the required JAR file from the lecture server or from the kXML site given in the references.

```java
*/
public class ExpressionParserXMLPull {
  public static void main(String args[])
    throws XmlPullParserException, IOException {
    XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
    factory.setNamespaceAware(true);
    XmlPullParser xpp = factory.newPullParser();
    System.out.println("Parser implementation class is "
        + xpp.getClass());

    ExpressionParserXMLPull app = new ExpressionParserXMLPull();

    System.out.println("Parsing file. ");
    xpp.setInput(new FileReader("data/expression.xml"));
    app.processDocument(xpp);
  }

  public void processDocument(XmlPullParser xpp)
    throws XmlPullParserException, IOException {
    int eventType = xpp.getEventType();
    do {
      if (eventType == xpp.START_DOCUMENT) {
        System.out.println("Start document");
      } else if (eventType == xpp.END_DOCUMENT) {
        System.out.println("End document");
      } else if (eventType == xpp.START_TAG) {
        processStartElement(xpp);
      } else if (eventType == xpp.END_TAG) {
        processEndElement(xpp);
      } else if (eventType == xpp.TEXT) {
        processText(xpp);
      }
      eventType = xpp.next();
    } while (eventType != xpp.END_DOCUMENT);
  }

  public void processStartElement(XmlPullParser xpp) {
    String name = xpp.getName();
    System.out.println("Start element: " + name);
  }

  public void processEndElement(XmlPullParser xpp) {
    String name = xpp.getName();
    System.out.println("End element: " + name);
  }

  int holderForStartAndLength[] = new int[2];

  public void processText(XmlPullParser xpp) throws XmlPullParserException {
    char ch[] = xpp.getTextCharacters(holderForStartAndLength);
    int start = holderForStartAndLength[0];
    int length = holderForStartAndLength[1];
    System.out.print("Characters:    \"");
    for (int i = start; i < start + length; i++) {
      if (ch[i] != '\n' && ch[i] != '\t')
      System.out.println(ch[i]);
    }
    System.out.print("\"\n");
  }
}
```

The developer-facing semantics of XML Pull Parser are approximately the same *factory design pattern*

as SAX. We first create an instance of the parser. This is done using the *factory design pattern*. We can then use our handler to parse this file. The most important line in our handler is the line `xpp.next()` (Line 46). It pulls the next event (hence the name *pull* parser). We can then decide which even it is and call the relevant function based on that. The only difference here from the SAX parser is that we're testing for basic text as well. This is the portion of characters that fall within the element starting and ending tags. This code can be modified to use the stack-based semantics we used in our SAX example.