XML Parsers

There are two models for XML parsers, SAX (Simple API for XML) and DOM (Document Object Model). SAX is the simpler model. It uses callbacks to get the data from the parser. These are similar to listeners such as the MouseListener found in Java. DOM creates a complete parse tree and provides methods to find information in it. It is more complicated to write and to use.

If an XML document is well-formed, it defines a general ordered tree. A parser has to read through the document and store the tree information. And if the document has either a DTD or a schema, it must determine whether or not the document is valid. Parsers are not easy to construct, but several have been made available.

The parsers we will use are part of Xerces, from the Apache Software Foundation. The original versions were created in 1999 at IBM and subsequently donated to Apache. The open source community has since modified them. Using them requires several JAR files. These can be downloaded from Apache. There are also copies on my web site.

The SAX Parser

SAX (Simple API for XML) uses a call back model. As the parser reads through the document, it sends back information about what it finds. There are several built-in methods (similar to those that come with AWT listeners) that can be used to access this information. The interface that defines them follows:

```
public interface ContentHandler
{
    public void setDocumentLocator (Locator locator);
    public void startDocument () throws SAXException;
    public void endDocument () throws SAXException;
    public void startPrefixMapping (String prefix, String uri) throws SAXException;
    public void endPrefixMapping (String prefix) throws SAXException;
    public void startElement (String namespaceURI, String localName, String qualifiedName,
                              Attributes atts) throws SAXException;
    public void endElement (String namespaceURI, String localName, String qualifiedName)
                              throws SAXException;
    public void characters (char[] text, int start, int length) throws SAXException;
    public void ignorableWhitespace (char[] text, int start, int length) throws SAXException;
    public void processingInstruction (String target, String data) throws SAXException;
    public void skippedEntity (String name)throws SAXException;
}
```

The important methods here are startDocument, endDocument, startElement, endElement, and characters. The others are needed for documents with namespaces (prefixes).

A specific class called a content handler has to be added to the parser, similar to way we added listeners to buttons when using the AWT. The code for that follows, where ElementExtractor is the name of the content handler class.

```
    // Get a content handler and add it to the parser.
    ContentHandler handler = new ElementExtractor ();
    parser.setContentHandler (handler);
```

The parser we will use is called SAXParser, one of the Xerces parsers. It goes into its own folder, with the name, org.apache.xerces.parsers.SAXParser. The parser itself is an instance of the XMLReader class. So the declaration of the parser is

```
XMLReader parser =
    XMLReaderFactory.createXMLReader ("org.apache.xerces.parsers.SAXParser");
```

If we want the parser to validate the xml document using a DTD (document type definition), we have to add this feature to the parser. It is done by

```
parser.setFeature ("http://xml.org/sax/features/validation", true);
```

The boolean, true, means that the feature will be active. So the parser will look for a link to the DTD and use it to validate the xml document.

The entire method, doParse follows:

```
// doParse gets a parser, adds a content handler, and parses the document.
public void doParse (String document)
{
    final String VALIDATION_FEATURE_ID = "http://xml.org/sax/features/validation";
    try
    {
        // Set the parser to validate using a DTD.
        XMLReader parser = XMLReaderFactory.createXMLReader
            ("org.apache.xerces.parsers.SAXParser");
        parser.setFeature (VALIDATION_FEATURE_ID, true);

        // Get a content handler and add it to the parser.
        ContentHandler handler = new ElementExtractor ();
        parser.setContentHandler (handler);

        // Parse the document.
        parser.parse(document);
    } catch (SAXException e)
    {   System.err.println("Warning: Parser does not support this validation feature .");}
    catch (Exception e) {System.err.println(e);}
} // doParse
```

Element Extractor

The main work is done by an inner class called ElementExtractor. It uses methods from the ContentHandler interface to extract information from the document. One of the methods in it is used to get the text data from between the tags. This uses the characters method in the interface.

```
// characters receives all the text in the xml document.
public void characters(char[] text, int start, int length) throws SAXException
{
    String word = "";
    word = word.copyValueOf (text, start, length);
    Node textNode = new Node ("text", word);
    tokens.add (textNode);
} // characters
```

The character data is stored in an array of characters called text. The method provides the start index and the length of the tag data. For example, if the xml document has

```
<name>Alice Lee</name>
```

The method will include in the text array the characters in Alice Lee, tell where these start in the array, and how many (9) there are. These can then be copied into a string and stored in a node in a vector. The vector is called tokens, and the node is textNode. ElementExtractor has an inner class that can access data in the parser class.

SAX Parser Program.

```
package parsers;

import java.util.*;
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

/*	SAXParser uses a SAX parser from the Apache Software Foundation. It parses the document and stores the tags and text in a vector. When the end tag is read, it displays all the tags stored in the vector on the screen.
*/
public class SAXParser
{
	public static void main (String [] args)
	{
		try
		{
			BufferedReader stdin = new BufferedReader (new InputStreamReader (System.in));
			System.out.println ("Enter XML document to parse: ");
			String xmlDoc = stdin.readLine ();
			ParseDocument parseDoc = new ParseDocument ();
			parseDoc.doParse (xmlDoc);
		} catch (IOException e) {System.out.println ("IO Exception");}
	} // main
} // SAXParser

/* ParseDocument parses the document and stores elements and text in a vector. When the end tag is read, it displays the contents of the vector on the screen.
*/
class ParseDocument
{
	private Vector tokens = new Vector (50);

	// doParse gets a parser, adds a content handler, and parses the document.
	public void doParse (String document)
	{
		final String VALIDATION_FEATURE_ID = "http://xml.org/sax/features/validation";
		try
		{
			// Set the parser to validate using a DTD.
```

```
            XMLReader parser = XMLReaderFactory.createXMLReader
                ("org.apache.xerces.parsers.SAXParser");
            parser.setFeature (VALIDATION_FEATURE_ID, true);

            // Get a content handler and add it to the parser.
            ContentHandler handler = new ElementExtractor ();
            parser.setContentHandler (handler);

            // Parse the document.
            parser.parse(document);
        } catch (SAXException e)
        {   System.err.println ("Warning: Parser does not support validation feature .");}
        catch (Exception e) {System.err.println(e);}
    } // doParse

    // ElementExtractor is an inner class that receives call backs from the parser for text and elements.
    class ElementExtractor extends DefaultHandler
    {
        // characters receives all the text in the xml document.
        public void characters (char[] text, int start, int length) throws SAXException
        {

            String word = "";
            word = word.copyValueOf (text, start, length);
            Node textNode = new Node ("text", word);
            tokens.add (textNode);
        } // characters

        // startElement picks out each start element and stores it in the vector.
        public void startElement (String namespaceURI, String localName, String qName,
                                  Attributes atts) throws SAXException
        {
            Node startNode, attrNode;
            startNode = new Node ("startTag", localName);
            tokens.add (startNode);

            // Get the attributes in the tag and add them to the vector.
            for (int count = 0; count < atts.getLength (); count ++)
            {
                String value = atts.getLocalName (count) + " = " + atts.getValue (count);
                attrNode = new Node ("attr", value);
                tokens.add (attrNode);
            }
        } // startElement

        // endElement picks out each end element and stores it in the vector.
        public void endElement (String namespaceURI, String localName, String qName)
            throws SAXException
        {
            Node endNode = new Node ("endTag", localName);
            tokens.add (endNode);
```

```
            } // endElement

            /*    endDocument reads the last tag in the document and then displays all the tokens in the
            vector on the screen.
            */
            public void endDocument() throws SAXException
            {
                  for (int count = 0; count < tokens.size (); count ++)
                  {
                        Node token = (Node)tokens.elementAt (count);
                        token.displayToken ();
                  }
            } // endDocument
      } // ElementExtractor
} // ParseDocument

// The Node class is used to store each token returned by the parser.
class Node
{
      private String tagType, tagValue;

      Node (String type, String value)
      {
            tagType = type;
            tagValue = value;
      } // constructor

      protected String getType () {return tagType;}
      protected String getValue () {return tagValue;}
      protected void displayToken () {System.out.println (tagType + "  " + tagValue);}
} // Node
```

DOM Parsers

The Document Object Model works on a different principle from the SAX Parser. The DOM Parser builds a complete parse tree with nodes for everything that can be included in an XML file. There are a number of these, but the most important ones are the document node, the element node and the text node. There are also special nodes for processing instructions, document declarations, comments, CDATA sections, and entities. Here we will just consider the first three.

The document node is used to store information about the entire document. It contains a pointer to the root node of the XML tree, but there are a number of other pointers in it as well. These include those for entities and system and public IDs. In order to begin processing the parse tree, you have to start with this node. The root node of the tree is obtained by

```
      Node node = document.getDocumentElement ();
```

The root node of the tree is an element node. It contains a name and a value, pointers to its first child and next sibling, and a pointer to a collection of attributes. The child nodes are used when you are doing a traversal of the tree, usually recursive. The name is the name in the start tag and usually the value is null. The attributes of the node are stored in a collection called a NamedNodeMap. They can be accessed by using an item method. The order may not be the same as that in the original document.

The text node is the simplest. It just contains all the text between the start and end tags. So the text for

```
<name>Alice Lee</name>
```

is just 'Alice Lee'. Text nodes are always leaf nodes in the tree.

Each node contains a short value that indicates the type of the node. These all have constant values so that programs can refer to Node.DOCUMENT_NODE, Node.ELEMENT_NODE, and Node.TEXT_NODE, rather than having to remember the exact short value.

The first thing that a program that uses DOM has to do is to get a parser and parse the tree. The parser is usually configured to do either DTD or schema validation as well. This is handled by the following code. It first gets the type of validation and the parser, adds the appropriate validation feature to the parser and then parses the tree.

```
package dom;

import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

/**
 *	Adapted from code that comes with Xerces and written by Andy Clark, IBM.
 *	The DomParser sets up a parser with either DTD or Schema validation.  It then calls DisplayTree
 *	to display the document on the console screen.
*/
public class DomParser
{
	// Validation feature id (http://xml.org/sax/features/validation).
	protected static final String
		Validation_Feature_Id = "http://xml.org/sax/features/validation";

	// Schema validation feature id (http://apache.org/xml/features/validation/schema).
	protected static final String
		Schema_Validation_Feature_Id = "http://apache.org/xml/features/validation/schema";

	// Default parser name.
	protected static final String Default_Parser_Name = "dom.wrappers.Xerces";

	// Main program.
	public static void main(String [] args)
	{
		// Request the validation option.
		System.out.println ("DTD and Schema Parser and Validator");
		BufferedReader stdin = new BufferedReader (new InputStreamReader (System.in));
		String xmlfile = "", option = "";
		try
		{
			// Get the validation option and file name.
			System.out.print ("Option: v for DTD or s for schema. ");
			option = stdin.readLine ();
```

```
            System.out.print ("XML File Name: ");
            xmlfile = stdin.readLine ();
        } catch (IOException e) {System.err.println ("No xml file name or option");}

        ParserWrapper parser = null;
        boolean DTDvalidation = false;
        boolean schemaValidation = false;

        if (option.equalsIgnoreCase ("v")) DTDvalidation = option.equals ("v");
        if (option.equalsIgnoreCase ("s")) schemaValidation = option.equals ("s");

        // Create parser.
        try
        {
            parser = (ParserWrapper)Class.forName(Default_Parser_Name).newInstance();
        }
        catch (Exception e) {System.err.println("Unable to instantiate parser");}

        // Set parser features.
        try
        {
            parser.setFeature (Validation_Feature_Id, DTDvalidation);
            parser.setFeature (Schema_Validation_Feature_Id, schemaValidation);
        } catch (SAXException e)
            {System.err.println ("Parser does not support this validation feature.");}

        try
        {
            // Parse the file.
            Document document = parser.parse (xmlfile);

            // Display the file on the screen.
            DisplayTree treeDisplay = new DisplayTree ();
            treeDisplay.display (document);
            System.out.println ();
        } catch (SAXParseException e) {System.err.println ("SAXParse error.");}
        catch (Exception e)
        {
            System.err.println ("Parse error occurred - "+e.getMessage());
            e.printStackTrace (System.err);
        }
    } // main
} // class DomParser
```

The parser above can be used in a number of different ways to get data from the XML file. It can be used alone in order to validate a file with either a DTD or schema. It can also be used to find a specific node in the tree. The following is taken from the program Write.java that is included as an example with Xerces version 1.1. Write.java contains much more code and can be used with XML files that have processing instructions, CDATA sections, and comments.

The class, DisplayTree contains a number of methods, but the important one is the first one, display. It uses recursion to get the data from the tree. The recursion is a little unusual, since only start tags are members of the tree, not end tags. So to display an end tag in the correct place, it has to first display all the children of the node before tacking on the end tag. This is done by first visiting all the first child descendents of the node and then going back to their next siblings.

```
Node child = node.getFirstChild();
while (child != null)
{
    display (child); // Recursive call.
    child = child.getNextSibling();
}
```

The rest of the code is used to find the node type and then display the data in the node.

```
/**
 *    DisplayTree uses recursion to travel through the parse tree and display the document.
 *    The node types displayed are document, element and text.
 */
package dom;

import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import java.lang.reflect.Method;

public class DisplayTree
{
    // display writes the specified node and then uses recursion to visit the rest of the tree.
    public void display (Node node)
    {
        if (node == null) return; // End of recursion
        short type = node.getNodeType();
        displayNode (node, type);

        /* Get the first child of the node and then travel all the way to the left most leaf.  Pop up, go to
         the next sibling, and repeat.  This continues until the entire tree has been visited. */
        Node child = node.getFirstChild();
        while (child != null)
        {
            display (child); // Recursive call.
            child = child.getNextSibling();
        }

        // Add the end tag after returning from the recursion.
        if (type == Node.ELEMENT_NODE)
            System.out.print ("</" + node.getNodeName() + ">");
    } // display (Node)

    // displayNode checks for the node type and calls the method that displays that type.
    protected void displayNode (Node node, short type)
    {
```

```
            if (type == Node.DOCUMENT_NODE)       // xml declaration
            {
                displayDocNode (node);
                Document document = (Document) node;
                node = document.getDocumentElement ();
            }
            else
            if (type == Node.ELEMENT_NODE)         //start-tag
                displayElementNode (node);
            else
            if (type == Node.TEXT_NODE)                // text
                System.out.print (node.getNodeValue ());
        } // displayNode

        // displayDocNode is used to display some of the data in the document node.
        protected void displayDocNode (Node node)
        {
            Document document = (Document) node;
            String version = null;
            try
            {
                // Get the XML version
                Method getXMLVersion =
                    document.getClass ().getMethod ("getXmlVersion", new Class[]{});
                // If Document class implements DOM L3, this method will exist.
                if (getXMLVersion != null)
                    version = (String) getXMLVersion.invoke (document, null);
            } catch (Exception e) {}

            // Display the appropriate version declaration.
            if (version.equals("1.1"))
                {System.out.println ("<?xml version=\"1.1\" encoding=\"UTF-8\"?>");}
            else {System.out.println ("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");}
        } // displayDocNode

        // displayElementNode first displays the name of the node and then gets and displays the attributes.
        protected void displayElementNode (Node node)
        {
            System.out.print ("<" + node.getNodeName ());

            // Get the attributes from the node and display them in the tag.
            NamedNodeMap attrs = node.getAttributes ();
            for (int count = 0; count < attrs.getLength (); count++)
            {
                Attr attr = (Attr) attrs.item(count);
                System.out.print (" " + attr.getNodeName() + "=\"" + attr.getNodeValue () + "\"");
            }
            System.out.print ('>');
        } // displayElementNode
    } // TreeDisplayClass
```

References

1. Clark, Andy, *Write.java*, Xerces version 1.1, 2003.
2. Elliotte Rusty Harold, *Processing XML with Java,* chapter 17, Addison Wesley, 2002.
3. Elliotte Rusty Harold and Scott Means, *XML Programming*, O'Reilly & Associates, Inc., 2002.
4. W3Schools Online Web Tutorials, http://www.w3schools.com.