Download it now!
PDF (43KB)
Free Acrobat[tm] Reader

XML tutorials for programmers

Ralf I. Pfeiffer
IBM XML Technology Group

See the abstract for this tutorial and Ralf's other XML tutorials in this series:

- Tutorial 1: Overview of XML
- Tutorial 2: Writing XML Documents

Tutorial 3: Parsing XML Using Java

Before you begin

In this tutorial, you'll learn how to process XML documents using Java and the IBM XML Parser for Java (XML4J) Version 2 (V2.0.9 and greater). We'll code the building blocks of a functional XML viewer included here (simple.zip). Although all of the example code is included or referenced from the tutorial itself, you may find it helpful to download, modify, and run this code while reading this tutorial. We will start with a simple program, and add functions and classes as we learn about the parser and the Document Object Model (DOM) specification.

The tutorial examples are graphical and make use of the Java JFC or Swing components. These are included in JDK Version 1.2, but can be obtained for previous JDK versions from Sun.

XML parsing options for applications

For parsing XML and accessing the results, there are two established APIs. First, the SAX (Simple API for XML) interfaces define *handlers* containing methods that are called as the various parts of an XML document are parsed. Second, the DOM (Document Object Model) interfaces define a logical *tree* representing the XML document after parsing.

Applications that do not require complex manipulations of the XML structure will find the SAX interfaces very useful. For example, a search-and-replace or element-name-transformation application using SAX code must only implement those callback methods in which they are interested.

For structural manipulations involving possibly all or most XML tokens, the DOM tree interfaces are useful. For example, an XSL processor can transform an input XML document to an XML or HTML output document based on rules in a second XML document. A DOM implementation such as the LotusXSL processor is useful in this case.

This tutorial’s SimpleTreeView examples could have been developed using either SAX or DOM. We have chosen DOM in this case somewhat arbitrarily. However, DOM does lay the groundwork for the next logical step, a tree editor allowing cutting and pasting of XML structure.

In addition, the SAX interfaces have well defined interfaces for common parsing tasks, such as parsing from any input source, error handling, document location, and more. Rather than reinvent this commonality for DOM, our DOM parser actually uses the same implementation of these SAX interfaces as our SAX parser. This reuse will expose you to some SAX interfaces as well.

The IBM XML Parser for Java (XML4J)

The IBM XML Parser for Java (XML4J) is written entirely in Java and comes with jar files for runtime, Java source, and documentation.

The XML4J parser has capabilities beyond the scope of this tutorial. In addition to the DOM *tree* interfaces, the parser also implements the SAX *callback* interfaces. Either SAX or DOM implementation can be validating or nonvalidating. In fact, two DOM implementations are provided. First, a new high-performance and DOM Level 1 compliant implementation is provided. The tutorial examples will use the DOM Parser, which uses this new DOM. And second, a backward-compatible DOM is provided that implements DOM Level 1 plus preliminary extensions. The extensions to the XML specification include the XPointer and Namespaces specifications.

The XML4J parser also contains online and packaged documentation of all of the APIs, including the SAX and DOM standards and parser-framework interfaces. The parser has been designed to be modular and observes a *pluggable* or *registration* design pattern for the major components such as validation. Applications or servers with special XML needs can build custom XML parsers using interfaces such as handlers. All of these capabilities are beyond the scope of this tutorial, but you can refer to the XML4J parser documentation for more information.

These powerful capabilities make XML4J a good base for building applications that ensure the creation of valid XML documents. For details on these capabilities and more, browse the extensive ReadMe included with the parser.

Parsing an XML document

The following sample program shows the simplest use of the parser. In this sample, we create a new DOMParser() instance with the simplest constructor of the DOMParser, and invoke parse(filename) and getDocument() to return a DOM Document reference.

In this sample, and in those that follow, the lines in red illustrate the main points covered in the text.

```
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;

public class GetParseTree {

 public static void main (String args[]) {

  // the first argument should be a filename.
  if (args.length > 0) {
   String filename = args[0];
   try {

    DOMParser parser = new DOMParser();
    parser.parse(filename);
    //*** The document is the root of the DOM tree.
    Document document = parser.getDocument();

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

Above, the lines in red create a new parser instance, parse an XML document, and return a document reference (the root of the DOM tree). We will learn more about the DOM interfaces soon.

The program above acts like a command line XML parser that you can invoke on the address book from the XML section like this:

java -cp [path-to-xml4j]\xml4jVERSION\xml4j.jar GetParseTree ab.xml

where VERSION is the version of the XML4J parser you have downloaded.

Handling errors while parsing an XML document

You can implement error handling consistently in any of the parser configurations in the com.ibm.xml.parsers package by implementing the org/xml/sax/ErrorHandler interface. For example, you may want to format the errors or redirect where they appear. You may even want to store the errors and provide a way for application programs to access them.

Custom error handling is accomplished by implementing the ErrorHandler interface with your own class and providing an instance of your class to the DOMParser#setErrorHandler(org/xml/sax/ErrorHandler) function.

If you would like the former GetParseTree program to output errors to System.err, we could implement it as follows:

```
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class GetParseTreeErrors {

 public static void main (String args[]) {

  // the first argument should be a filename.
   if (args.length > 0) {
    String filename = args[0];
    //*** The document is the root of the DOM tree.

    try {
     new GetParseTreeErrors(filename);
        } catch (Exception e) {
         e.printStackTrace(System.err);
        }
       }
      }

   public GetParseTreeErrors (String filename) {
    try {
     DOMParser parser = new DOMParser();
      Errors errors = new Errors();
      parser.setErrorHandler(errors);
      parser.parse(filename);
      Document document = parser.getDocument();
   } catch (Exception e) {
   e.printStackTrace(System.err);
  }
 }

 class Errors implements ErrorHandler {

  public void warning(SAXParseException ex) {
   System.err.println(ex);
  }

  public void error(SAXParseException ex) {
   System.err.println(ex);
  }

  public void fatalError(SAXParseException ex) throws SAXException {
   System.err.println(ex);
  }
 }
}
```

Note how we implemented three functions (warning, error and fatalError, highlighted in red above), and provided the implementing instance to the parser.setErrorHandler method.

Soon, after we've learned more about DOM, we'll learn how to modify our implementation of ErrorHandler, mapping errors to the appropriate DOM node. Our goal is to write a simple DOM viewer that can map JTree TreeNode nodes to DOM Nodes for our address book or any XML document.

Understanding the DOM and how to use it

The Document Object Model (DOM) is a set of language-independent interfaces for programmatic access to the logical structure of an XML document. We will use the latest Java DOM Interfaces. These correspond to the latest version of the language-independent DOM Level 1 interface as specified by the W3C, which is always accessible through this link. IBM's XML4J parser always tracks the latest version, pretty much as soon as it is available.

As we have learned, the structure of a well formed XML document can be expressed logically as a tree with a single root. The Interface that encapsulates the *structural* connections between the XML constructs is called the Node. The Node contains functions that express structural connections such as Node#getChildNodes(), Node#getNextSibling(), Node#getParentNode(), and so on.

The DOM Interfaces also contain separate interfaces for XML's high-level constructs such as Element. Each of these interfaces also extends Node. For example, there are interfaces for Element, Attribute, Comment, Text, and so on. Each of these specify the getter and setter functions for their own specific data. For example, the Attribute interface has Attribute#getName(), and Attribute#getValue() member functions. The Element interface has the means to get and set attributes via functions like Element#getAttributeNode(java.lang.String), and Element#setAttributeNode(Attribute).

Always remember that various high-level interfaces such as Element, Attribute, Text, Comment, and so on, all *extend* Node. This means the structural member functions of Node (such as getNodeName()) are available to Element, Attribute, and all these types. An illustration of this is that any node such as Element or Text knows what it is, by re-implementing getNodeType(). This allows the programmer to query the type using Node#getNodeType() instead of Java's more expensive run-time type instanceof function.

So, in Java you can write a simple recursive function to traverse a DOM tree:

```
import org.w3c.dom.*;
//*** Only refer to DOM Interfaces...
public static void traverseDOMBranch(Node node) {

// do what you want with this node here...

System.out.println(node.getNodeName()+":"+node.getNodeValue());

 if (node.hasChildNodes()) {
  NodeList nl = node.getChildNodes();
  int size = nl.getLength();
  for (int i = 0; i < size; i++) {
      traverseDOMBranch(nl.item(i));
  }
 }
}
....
//*** Note how we refer only to DOM Interface references.
Document doc = parser.getDocument();

Element root = (Element)doc.getDocumentElement();
traverseDOMBranch(root);
```

You can immediately use this traverseDOMBranch function on any Node from any DOM implementation. For larger documents, you may want to write a non-recursive function to enhance performance.

In this way, you can write classes or utility functions (such as traverseDOMBranch) that refer only to DOM interfaces and are independent of the DOM implementation. *The practice of coding against the DOM interfaces (rather than the specific implementation classes) is recommended unless your application actually requires or benefits from the extra functionality of a particular DOM implementation*. Even if parts of your code require the specification of a particular parser or settings, the code referring to DOM interfaces should be reusable across all DOM implementations.

The root of the DOM tree, Document

The root of the DOM tree is the Document interface. We have waited until now to introduce it because it serves multiple purposes. First, it represents the whole document and contains the methods by which you can get to the *global* document information, the DocumentType and the root Element.

Second, it serves as a general constructor or *factory* for all XML types, providing methods to create the various constructs allowed in an XML document. If an XML parser gives you a DOM Document reference, you may still invoke the create methods within the document to build more DOM nodes and use appendChild and other functions to add them to the document node or other nodes within it. Note that if the client programmer changes, adds, or removes nodes from the DOM tree, there is no DOM requirement to check or maintain validity. This burden is left to the programmer (with possible help from the specific DOM or parser implementation).

The SimpleTreeView application

Using our last example, GetParseTreeErrors as a starting point, the following example creates a SimpleTreeView by adding a few lines of code:

```
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import javax.swing.*;
import ui.DOMTree;

/** SimpleTreeView builds on GetParseTreeErrors
 *  to show how to build a JTree viewer from the DOM Document reference.
 */
public class SimpleTreeView extends JFrame {

 DOMParser parser;
 Errors errors;

 public static void main (String args[]) {

  if (args.length > 0) {
   String filename = args[0];

    try {

     SimpleTreeView frame = new SimpleTreeView(filename) ;
     frame.addWindowListener(new java.awt.event.WindowAdapter() {
     public void windowClosing(java.awt.event.WindowEvent e) {
     System.exit(0);
       }
      });
     frame.setSize(300, 400);
```

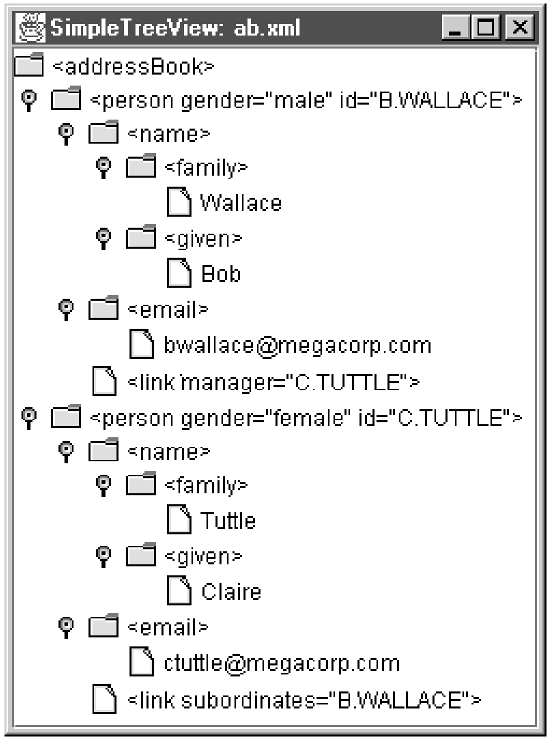
```
      frame.setVisible(true);

      } catch (Exception e) {
      e.printStackTrace(System.err);
      }
     }
    }

public SimpleTreeView (String filename) {
 super("SimpleTreeView: "+filename);
 try {
      parser = new DOMParser();
      errors = new Errors();
      parser.setErrorHandler(errors);
      parser.parse(filename);
      Document document = parser.getDocument();

    getContentPane().add(new JScrollPane(new DOMTree(document)));
    } catch (Exception e) {
     e.printStackTrace(System.err);
    }
   }

class Errors implements ErrorHandler {
 public void warning(SAXParseException ex) {
  System.err.println(ex);
}

public void error(SAXParseException ex) {
 System.err.println(ex);
}

public void fatalError(SAXParseException ex) throws SAXException {
 System.err.println(ex);
}
 }
}
```

We have added just a few lines to the GetParseTreeErrors class, and renamed the class to be SimpleTreeView. The new lines are highlighted in red above.

The work of creating a visual JTree from a DOM Document reference is done by a sample we are reusing called ui.DOMTree. The ui.DOMTree class is included as a sample in the XML4J (2.0.9 and later) download and can found in the simple.zip file of this tutorial. We have shown examples of traversing the DOM and recursively printing the results. You can envision building JTree TreeNodes instead of printing, and this is essentially what is done in the ui.DOMTree example. Refer to the code to see the exact details.

We will now use the SimpleTreeView above as a basis for building an application that shows off other practical tasks using DOM.

Here is an image of the SimpleTreeView to show the application displaying the address book example (ab.xml):



Storing errors

Our SimpleTreeView prints errors to System.err now, but it would be nice to see those errors in a JTextArea within the application. It would also be useful to visually highlight TreeNodes with errors and associate any JTree TreeNode with its DOM Node.

To accomplish these tasks, it's necessary to store errors. In the code below, note how we've implemented a store function. We simply call store from our ErrorHandler function callbacks instead of printing them out.

Here's the implementation:

```
import java.util.Hashtable;
import java.util.Enumeration;
import com.ibm.xml.parsers.DOMParser;
import org.w3c.dom.Document;
```

```
import org.w3c.dom.Node;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import javax.swing.*;
import ui.DOMTree;

/** SimpleTreeView2 builds on SimpleTreeView
 *  to show how store the errors and print them after parsing.
 */
public class SimpleTreeView2 extends JFrame {

DOMParser parser;
Errors errors;

public static void main (String args[]) {

 if (args.length > 0) {
 String filename = args[0];

 try {
      SimpleTreeView2 frame = new SimpleTreeView2(filename) ;
      frame.addWindowListener(new java.awt.event.WindowAdapter() {
      public void windowClosing(java.awt.event.WindowEvent e) {
       System.exit(0);
     }
     });
     frame.setSize(300, 400);
     frame.setVisible(true);
     } catch (Exception e) {
     e.printStackTrace(System.err);
     }
    }
   }

public SimpleTreeView2 (String filename) {
 super("SimpleTreeView2: "+filename);
   try {
        parser = new DOMParser();
        errors = new Errors();
        parser.setNodeExpansion(DOMParser.FULL);
        parser.setErrorHandler(errors);
        parser.parse(filename);
        Document document = parser.getDocument();

        Hashtable errorNodes = errors.getErrorNodes();
        Enumeration elements = errorNodes.elements();
         while (elements.hasMoreElements()) {
          System.out.println( (String)elements.nextElement());
       }
         getContentPane().add(new JScrollPane(new DOMTree(document)));
      } catch (Exception e) {
         e.printStackTrace(System.err);
     }
    }

 class Errors implements ErrorHandler {

     Hashtable errorNodes = new Hashtable();

     public void warning(SAXParseException ex) {
         store(ex, "[Warning]");
     }

     public void error(SAXParseException ex) {
         store(ex, "[Error]");
     }

     public void fatalError(SAXParseException ex) throws SAXException {
         store(ex, "[Fatal Error]");
```

```
    }

    public Hashtable getErrorNodes() {
        return errorNodes;
    }

    public Object getError(Node node) {
        return errorNodes.get(node);
    }

    public void resetErrors() {
        errorNodes.clear();
    }

    void store(SAXParseException ex, String type) {

        // build error text
        String errorString= type+" at line number, "+ex.getLineNumber()
            +": "+ex.getMessage()+"\n";
         Node currentNode = parser.getCurrentNode();
        if (currentNode == null) return;

         // accumulate any multiple errors per node in the Hashtable.
        String previous = (String) errorNodes.get(currentNode);
        if (previous != null)
            errorNodes.put(currentNode, previous +errorString);
        else
            errorNodes.put(currentNode, errorString);
    }

 }
}
```

Our inner class Errors now provides the client with member functions to access the error information given a DOM node. We also allow the client to ask for the whole errorNodes Hashtable. Note that nodes are the keys of our table, and the values are the String error information. More sophisticated applications (like the ui.TreeViewer sample) must wrap the error in another class (instead of a simple string) to enable access for the component parts such as source line and column location.

Note that the XML4J Version 2 DOMParser is performance-tuned to create DOM tree nodes on a DEFERRED basis for speed. This is the default. However, it can fully create all DOM nodes during parsing with the call setNodeExpansion(DOMParser.FULL). The FULL expansion parser setting is required for DOMParser.getCurrentNode() to always give an accurate non-null result. In red above, we have used the DOMParser.FULL expansion so we could arbitrarily ask the parser for any node.

Viewing errors in the application

From here, it’s a small step to view errors in the application. We can create a JTextArea and append the errors to it. Most of the code is new UI code to create a JTextArea and add it to the application. For reference, here is the complete SimpleTreeView3.java application.

The relevant change is highlighted below in red:

```
...

Hashtable errorNodes = errors.getErrorNodes();
Enumeration elements = errorNodes.elements();
while (elements.hasMoreElements()) {
//*** append errors to messageText
messageText.append( (String)elements.nextElement());
            }
```

The real motivation for the new storage mechanism is to show how to highlight errors and *map* a JTree selection to the DOM nodes. This is shown in the next section.

Highlighting the error TreeNodes in the JTree

You may refer to SimpleTreeView4.java, the full application, which includes the additional functionality of this section. This is the final step. Due to Swing design, we must subclass DefaultTreeCellRenderer to change the appearance of a Swing JTree's TreeNode. Also, we want to echo additional information when a node is selected.

Starting from the previous section's SimpleTreeView3, the following DOMTree construction:

```
  JPanel treePanel = new JPanel(new BorderLayout());
  treePanel.add(new JScrollPane(new DOMTree(document)) {
```

is expanded to:

```
       // jtree  UI setup
       jtree = new DOMTree(document)
       jtree.setCellRenderer(new XMLTreeCellRenderer());
       jtree.getSelectionModel().setSelectionMode
           (TreeSelectionModel.SINGLE_TREE_SELECTION);

       // Listen for when the selection changes, call nodeSelected(node)
       jtree.addTreeSelectionListener(
       new TreeSelectionListener() {
            public void valueChanged(TreeSelectionEvent e) {
                   TreeNode node = (TreeNode)
                       (e.getPath().getLastPathComponent());

                   nodeSelected(node);
               }
           }
       );
       JPanel treePanel = new JPanel(new BorderLayout());
       treePanel.add(new JScrollPane(jtree) {
```

The XMLTreeCellRenderer subclass simply sets the foreground color if the DOM Node is in the Error storage:

```
/*
 * The XMLTreeCellRenderer is an inner class which enables the
 * highlighting of errors in the tree.
 */
 class XMLTreeCellRenderer extends DefaultTreeCellRenderer
 {
  public Component getTreeCellRendererComponent(JTree tree, Object value,
         boolean selected, boolean expanded,
         boolean leaf, int row,
         boolean hasFocus)
 {
  Node node = ((DOMTree)tree).getNode(value);
         Component comp = super.getTreeCellRendererComponent(tree, value,
         selected,  expanded, leaf,  row, hasFocus);
         if (errors != null && errors.getErrorNodes() != null && ¬
         value != null && node != null
         && errors.getErrorNodes().containsKey( node ))  {
               comp.setForeground(Color.red);
 }
  return comp;
 }
}
```

The other interesting code from the DOM perspective is in the nodeSelected method. For nodes with error text, the error message foreground is set to red. But more interestingly, the attribute values for an element are traversed and printed.

```
/** called when our JTree's nodes are selected.
 */
void nodeSelected(TreeNode treeNode) {

    Node node = jtree.getNode(treeNode);
    StringBuffer sb = new StringBuffer();

    messageText.selectAll();
    messageText.cut();

    Object errorObject = errors == null ? null : errors.getError(node);
    if (errorObject != null) {
        // There *is* an error in this node.
        messageText.setForeground(Color.red);
        sb.append(errorObject);
    } else {
        messageText.setForeground(Color.black);
    }

    if (node.getNodeType() == Node.ELEMENT_NODE) {
        sb.append("Element= ");
        sb.append(node.getNodeName());
        sb.append("\n");
        NamedNodeMap attrs = node.getAttributes();
        if (attrs != null) {
            int length = attrs.getLength();
            for (int i = 0; i < length; i++) {
                Attr attr = (Attr)attrs.item(i);
                sb.append("  attribute name=");
                sb.append(attr.getName());
                sb.append(" attribute value=");
                sb.append(attr.getValue());
                sb.append("\n");
            }
        }
    } else
    if (node.getNodeType() == Node.TEXT_NODE) {
        sb.append("Text= ");
        sb.append(node.getNodeValue());
    }

    messageText.append(sb.toString());
}
```

In the nodeSelected function, you probably noticed that retrieving an element's attribute values was different from simply traversing the children of the tree. By DOM design, the Attr nodes belong to an element but are are not children of the element.

Summary

Throughout this tutorial, as in the first two tutorials in this series, we've come a long way with XML. Here, we've implemented a simple treeviewer which displays the DOM tree in a visual JTree and stores and highlights errors. We've also learned how to:

- Use the XML Parser for Java to parse an XML document
- Use the SAX ErrorHandler interfaces to customize error output
- Use the DOM interfaces

- Check the node's type and cast the node to its various subinterfaces and walk the DOMTree
- Set DOMParser options

For more information

Besides the XML Parser for Java, you can also check out IBM's alphaWorks site for XML tools.

To learn more about XML parsing with Java, you'll also find it useful to review the other samples included with the parser. There are both SAX and DOM examples for simple parsing and counting (SAXCount, DOMCount) and contrasting examples for writing XML from both SAX and DOM APIs (SAXWriter, DOMWriter).