Java and XML Parsing Using Standard APIs

Edwin Goei
Software Engineer edwingo@sun.com
Sun Microsystems September 11, 2000

Introduction

XML stands for eXtensible Markup Language. XML gives applications a portable way to represent structured data across many types of computer systems. Java is the natural complement to XML. Java enables code to be executed across different computer platforms. This paper will introduce common Java APIs used to parse XML documents along with example code. But first we will give an overview of XML. Note that each of the topics described here can be covered in a separate paper by itself so we will only provide a general flavor of each topic and refer readers to other documents for more information.

XML Overview

Before describing what XML is, let's first look at a sample XML document.

```
 1 <?xml version="1.0"?>
 2 <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
 3     "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
 4
 5 <!-- The XML "document element" or "root" here is 'svg' -->
 6 <svg width="800" height="600">
 7     <!-- A green rectangle -->
 8     <rect x="200" y="50" width="200" height="150" style="fill: green"/>
 9
10     <!-- Some text -->
11     <text x="230" y="220" style="fill: red">
12           A Green Rectangle
13     </text>
14
15     <!-- An example of nested elements -->
16     <g transform="translate(50, -10)">
17         <circle cx="250" cy="300" r="30" style="fill: blue"/>
18         <text x="210" y="350" style="fill: red">A Blue Circle</text>
19     </g>
```

```
20 </svg>
```

Figure 1 Sample XML document

On the surface, XML looks like HTML but you can make up your own tags. It also has stricter rules. Each start-tag must have a matching end-tag and they must also nest properly. (In XML, <foo/> is equivalent to <foo></foo>.) Attribute values must also be quoted. In addition, element tag names and attribute names are case-sensitive.

One fundamental difference, however, is that unlike HTML, XML is a framework used to define other markup languages. XML is really a simplified version of **SGML** (Standardized General Markup Language), which is another older framework used to describe other markup languages such as HTML itself. Each application of XML may specify a particular set of element tags and attributes in a **DTD** (Document Type Definition, described later). One reason why HTML is not an application of XML is that HTML has tags like `<hr>`, which do not have an end-tag required in XML. Such tags are allowed in SGML but not in XML.

So what are some applications of XML? One example is the document above, which is an **SVG** (Scalable Vector Graphics) document. SVG is used to describe vector graphics. The example above describes a rectangle and a circle along with two captions. When rendered, the image appears as shown in Figure 2.

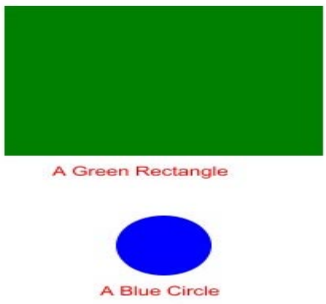

Figure 2 Sample document rendered as an image (colors may not appear properly on monochrome media)

Other example XML applications are WML or Wireless Markup Language, which is used to web-enable cellular phones. Also, returning to the <hr> element in HTML missing an end tag, there is a reformulation of HTML 4.0 in XML called XHTML. In XHTML, there is a `</hr>` end tag defined to make it a proper application of XML, among other modifications.

What are some of the reasons to use XML? As we mentioned before, it is cross platform. This means, for example, there is a uniform way to handle end-of-line terminators whether it's a newline on Unix, a CRLF on DOS, or a CR on an Apple. Also, XML defines a default character set and encoding. ISO 10646 also known as **UCS** (Universal Character Set) is used of which Unicode is a proper subset. Thus, another advantage is that documents can include script

from multiple languages simultaneously. Finally, it is a standard way of representing hierarchical structured data.

Well-formedness

Before explaining well-formedness, we should make a note about terminology. Standards documents use the term **element type** to refer to what many people call a "tag". It is useful to note equivalence of these two terms to avoid confusion. In this paper, we will use both terms interchangeably.

Earlier we mentioned some rules for XML such as each start tag must have a matching end tag. A document that conforms to these rules is said to be **well-formed.** It appears that in the sample document above, not all start-tags have end-tags. Instead there are tags of the form <foo/>. This is equivalent to <foo></foo> in XML and is called an **empty element**. Other **well-formedness** constraints are that tags must be properly nested, attributes must be quoted by a double or single quote, and exactly a single root document element is allowed.

Going back to the example in Figure 1, line 1 is an **XML Declaration** and it identifies this as an XML file. It is optional in this particular case, but other attributes may be used to specify alternate encodings, for example. Line 2 is optional for the non-validating case. Validation will be covered later. Line 5 is a comment and is similar to HTML. Line 6 is the **document element** or **root.** Exactly one document element is allowed. Its attributes are **width** and **height** specifying the dimensions of the overall canvas. The tag on line 8 describes a rectangle at (200, 50) with dimensions 200x150. The style attribute specifies a green fill color. The element on line 8 describes a text caption to be drawn in red. The `<g>` tag on line 16 groups several sub-elements together.

SAX

SAX is the Simple API for XML. As of this writing, the current version is 2.0. SAX is an event based API where the parser calls user-defined callbacks whenever it sees certain constructs in the input document during parsing. An example is in order. The following SAX program counts the number of times a particular tag occurs.

```
 1 import java.util.*;
 2 import java.io.*;
 3 import javax.xml.parsers.*;
 4 import org.xml.sax.*;
 5 import org.xml.sax.helpers.*;
 6
 7 public class SAXTagCount extends DefaultHandler {
 8     // A Hashtable with tag names as keys and Integers as values
 9     private Hashtable tags;
10
11     // Parser calls this once at the beginning of a document
12     public void startDocument() throws SAXException {
13         tags = new Hashtable();
14     }
15
16     // Parser calls this for each element in a document
17     public void startElement(String namespaceURI, String localName,
18                              String rawName, Attributes atts)
19         throws SAXException
20     {
21         String key = localName;
```

```
22          Object value = tags.get(key);
23          if (value == null) {
24              // Add a new entry
25              tags.put(key, new Integer(1));
26          } else {
27              // Get the current count and increment it
28              int count = ((Integer)value).intValue();
29              count++;
30              tags.put(key, new Integer(count));
31          }
32      }
33
34      // Parser calls this once after parsing a document
35      public void endDocument() throws SAXException {
36          Enumeration e = tags.keys();
37          while (e.hasMoreElements()) {
38              String tag = (String)e.nextElement();
39              int count = ((Integer)tags.get(tag)).intValue();
40              System.out.println("Tag <" + tag + "> occurs " + count
41                                 + " times");
42          }
43      }
44
45      static public void main(String[] args) {
46          if (args.length == 0) {
47              System.out.println("Usage: SAXTagCount <filename>");
48              System.exit(1);
49          }
50          String filename = args[0];
51
52          try {
53              // Create a SAX XMLReader instance
54              XMLReader xmlReader = XMLReaderFactory.createXMLReader(
55                      "org.apache.crimson.parser.XMLReaderImpl");
56
57              // Set the ContentHandler of the XMLReader
58              xmlReader.setContentHandler(new SAXTagCount());
59
60              // Tell the XMLReader to parse the XML document
61              xmlReader.parse(new File(filename).toURL().toString());
62          } catch (Exception e) {
63              e.printStackTrace();
64          }
65      }
66  }
```

Figure 3 Example SAX 2.0 program to count tag frequency in an XML document

The method `main` starting at line 53 performs three basic steps. First, it creates a new SAX `XMLReader` instance. An `XMLReader` is supplied by the parser implementation. It is an object that knows how to parse an XML document. SAX requires that the parser implementation class be specified either as an argument to `XMLReaderFactory.createXMLReader` or with a system property `org.xml.sax.driver`.

In line 58, method `main` then gives the new instance a `ContentHandler` instance. A `ContentHandler` is a SAX interface, which the developer provides. The `ContentHandler` interface defines callback methods that get invoked as the parser parses the XML document. Example callbacks are the methods `startDocument`, `startElement`, and `characters`. Note

that the sample code does not implement `ContentHandler` directly, but instead extends the class `org.xml.sax.helpers.DefaultHandler`, which implements `ContentHandler`. `DefaultHandler` implements all of the `ContentHandler` callback methods with empty do-nothing methods. Our SAX application class overrides a few of these empty methods with application code.

The third step in method `main`, on line 61, is to tell the `XMLReader` instance to parse a document. This is the method that causes the parser to invoke the callbacks in the `ContentHandler` that was set in the second step. After calling the `XMLReader.parse` method, the application exits.

Let's look at the callbacks in our application since that is where all of the action occurs. The parser invokes the `startDocument` method when it starts to parse a new XML document. Here we use this method to create a `Hashtable` of tags. The keys of the `Hashtable` will be the tag names, the values will be `Integers` holding a count of occurrences.

The parser calls the `startElement` method whenever it encounters a start-tag. The parameters of this method specify the particular start-tag encountered as well as a list of attributes. To fully describe the parameters of this method requires understanding `Namespaces`, which is beyond the scope of this paper. Since our sample document does not use `Namespaces`, we are only interested in the `localName` parameter, which specifies the name of the start-tag. The rest of the method looks up the start-tag in the `Hashtable` and either increments the count or creates a new entry initialized to 1.

After reaching the end of the document, the parser calls the `endDocument` method. In this method, we dump the contents of the `Hashtable`. The following figure shows the output of the program when run with our sample XML file in Figure 1.

```
Tag <circle> occurs 1 times
Tag <svg> occurs 1 times
Tag <text> occurs 2 times
Tag <g> occurs 1 times
Tag <rect> occurs 1 times
```

Figure 4 Output of SAXTagCount with XML of Figure 1 as input

DOM

DOM stands for Document Object Model. Many applications want a tree representation of an XML document instead of a series of callbacks from the parser. One API that provides this is DOM. DOM is a standard produced by the W3C. As of this writing, the current version is DOM Level 1, but Level 2 is a Proposed REC and should soon be a full REC.

A DOM tree representation of the sample document from Figure 1 follows. To simplify this example, we remove the `doctypedecl`, the statement beginning with `<!DOCTYPE` on lines 2 and 3. We will cover this statement later when we discuss validation.

```
 1 DOCUMENT:
 2   COMMENT: " The XML "document element" or "root" here is 'svg' "
 3   ELEMENT: "svg" ["width"="800", "height"="600"]
 4     TEXTNODE: WHITESPACE
 5     COMMENT: " A green rectangle "
 6     TEXTNODE: WHITESPACE
 7     ELEMENT: "rect" ["x"="200", "y"="50", "width"="200", "height"="150",
 8                      "style"="fill: green"]
 9     TEXTNODE: WHITESPACE
10     COMMENT: " Some text "
11     TEXTNODE: WHITESPACE
12     ELEMENT: "text" ["x"="230", "y"="220", "style"="fill: red"]
13       TEXTNODE: "
14         A Green Rectangle
15     "
16     TEXTNODE: WHITESPACE
17     COMMENT: " An example of nested elements "
18     TEXTNODE: WHITESPACE
19     ELEMENT: "g" ["transform"="translate(50, -10)"]
20       TEXTNODE: WHITESPACE
21       ELEMENT: "circle" ["cx"="250", "cy"="300", "r"="30",
22                          "style"="fill: blue"]
23       TEXTNODE: WHITESPACE
24       ELEMENT: "text" ["x"="210", "y"="350", "style"="fill: red"]
25         TEXTNODE: "A Blue Circle"
26       TEXTNODE: WHITESPACE
27     TEXTNODE: WHITESPACE
```

Figure 5 DOM tree representation of document in Figure 1

In DOM, each node type is represented by a Java interface in the package `org.w3c.dom`. Examples are `Element`, `Attribute`, `Comment`, and `Text`. In Figure 5, each node in the tree begins on a separate line and has a node type written in all upper case. For example, the tree shows DOM `Element` nodes with the label "ELEMENT" to represent elements in the document. Indentation signifies that a particular node is a child of a preceding non-indented node.

Note that the attributes of an element are represented by DOM `Attribute` nodes, but do not appear directly in the tree structure. Instead an `Attribute` node is associated with the element itself and is accessed via its corresponding `Element` node. Therefore the tree shows attributes with their associated elements between brackets.

One possible point of confusion in our SVG example is that SVG has an element type called "text", represented by a DOM `Element` with the name "text", which should not be confused with the DOM `Text` node type. In the tree, the DOM `Text` node type is labeled as "`TEXTNODE`" to minimize confusion. Please be aware of this difference.

Notice that the DOM tree contains `Text` nodes containing only white space. This is because XML requires all character data to be reported to the application. Thus if the input document contains elements within an element and there is white space used for formatting between child elements, such as for the <circle> and <text> child elements of the <g> element on line 16 of Figure 1, white space `Text` nodes will appear in the DOM tree. Currently, the only standard way of handling this is for the application to navigate around or filter out any unwanted white space `Text` nodes.

The following example program shows how to access data from the DOM tree representing the document from Figure 1.

```
 1 import java.io.*;
 2 import javax.xml.parsers.*;
 3 import org.w3c.dom.*;
 4
 5 public class DOMGetCharacters {
 6     static public void main(String[] args) {
 7         if (args.length == 0) {
 8             System.out.println("Usage: DOMGetCharacters <filename>");
 9             System.exit(1);
10         }
11         String filename = args[0];
12
13         try {
14             // Create a DOM Document object
15             DocumentBuilderFactory dbf =
16                 DocumentBuilderFactory.newInstance();
17             DocumentBuilder db = dbf.newDocumentBuilder();
18             Document doc = db.parse(new File(filename));
19
20             // Get the contents of the second <text> node in our example
21             Element svgElement = doc.getDocumentElement();
22             NodeList nodeList = svgElement.getElementsByTagName("g");
23             // There is only a single "<g>" element in the document
24             Element gElement = (Element)nodeList.item(0);
25             // Get the "<text>" element in a similar fashion
26             Element textElement =
27                 (Element)gElement.getElementsByTagName("text").item(0);
28             // The character data of the node is in a child TEXTNODE
29             Text contentNode = (Text)textElement.getFirstChild();
30             String content = contentNode.getData();
31
32             // Print out the character data content
33             System.out.println("Content of the second <text> node is '"
34                                + content + "'");
35
36             // Get the value of the 'style' attribute
37             String attValue = textElement.getAttribute("y");
38             System.out.println("Its 'y' attribue is '" + attValue + "'");
39         } catch (Exception e) {
40             e.printStackTrace();
41         }
42     }
43 }
```

Figure 6 Example DOM program to access data from document in Figure 1

Suppose we want to access the character data content “A Blue Circle” shown on line 25 of Figure 5. The first step would be to obtain a handle to the DOM `Document` object of line 1 in Figure 5. Unfortunately, the current DOM level 2 specification does not provide a way to load an existing XML document and return a DOM `Document` object. This feature is currently being developed for DOM level 3. As a result, each parser implementation provides a different implementation-dependent API to do this. Another alternative is to use JAXP, which we will describe later and is used in our example program.

Once we obtain a handle to the DOM `Document`, we can traverse the tree to obtain our data. This begins on line 20. Line 21 obtains the document element. Line 22 requests a list of all `<g>` elements below the document element of which there is only one. Line 24 gets the `<g> Element` node. Line 26 obtains the `<text> Element`. Line 29 obtains the single `Text` node child, which contains the character data contain within the `<text> Element`. Finally, line 36 shows an example of accessing an attribute of the `<text> Element`. Note that the code avoids the white space `Text` nodes in the tree by calling the `getElementsByTagName` method rather than obtaining a list of child nodes and then filtering out unwanted nodes.

The output of the program is shown below.

```
Content of the second <text> node is 'A Blue Circle'
Its 'y' attribue is '350'
```

Figure 7 Output of DOMGetCharacters when supplied with document from Figure 1

JAXP

JAXP stands for Java API for XML Parsing and its current version is 1.0 with a version 1.1 in development and to be released soon. JAXP 1.0 is composed of a few classes along with standard Java APIs such as SAX and DOM and was designed to be parser independent and to simplify XML application development in Java.

In JAXP, a DOM document can be loaded with the following code repeated from Figure 6.

```
14              // Create a DOM Document object
15              DocumentBuilderFactory dbf =
16                  DocumentBuilderFactory.newInstance();
17              DocumentBuilder db = dbf.newDocumentBuilder();
18              Document doc = db.parse(new File(filename));
```

Figure 8 Code fragment to load an XML document and return a DOM Document

In a similar manner, JAXP 1.0 supports SAX 1.0 (abbreviated SAX1) parsers, however, the current version of SAX is 2.0, which will not be supported until JAXP 1.1. JAXP 1.1 should be released soon. However, SAX2 contains a class that adapts a SAX1 parser into a SAX2 `XMLReader`. It is possible to use JAXP 1.0 to create a SAX1 `Parser` and then wrap it in a SAX2 `ParserAdapter` object to yield an `XMLReader`, although it does have some limitations. See the SAX documentation on `org.xml.sax.helpers.ParserAdapter` for more information.

Another feature that JAXP provides is a method to control validation. We cover validation in the next section.

Constraining XML

As mentioned before, XML is a framework for defining specific markup languages such as SVG, WML, and XHTML. Each of these specific markup languages has specific element types and attribute names. In addition, each also has constraints as to what elements can contain other elements and what attributes can be used with each element. These constraints are specified traditionally in a **DTD** or Document Type Definition.

DTDs originally came from SGML and use a different syntax, which is not XML. In addition, the constraints that can be expressed with DTDs are limited. As a result, new proposals for constraining XML have emerged. One such proposal is **XML Schema**, which is being developed by the W3C. One note of caution about confusing terminology, many people refer to "**schema**" in a more general sense to mean a method of constraining XML rather than a specific constraint scheme. Using this terminology, both XML Schema and DTDs are schemas. There are other schema frameworks as well such as **Relax, SOX,** and **XDR** (XML Data Reduced). The specification for the DTD syntax is contained in the XML specification itself. A separate specification covers XML Schema.

In this paper, we will only discuss DTDs as a means for constraining XML. The `doctypedecl` on lines 2 and 3 of Figure 1 refers to an external DTD subset located at the URL on line 3. Describing DTD syntax is beyond the scope of this paper. Instead we will assume that a DTD already exists for a document as in this example.

In XML, the term **validation** refers to checking that an XML document conforms to a DTD. SAX1 did not provide a standard way to control whether a parser validated a document. JAXP 1.0 provided such a mechanism. However, the newer SAX2 specification does provide a method to control validation. Thus, there are two standard ways to specify validation for SAX. For DOM, the only common method to specify validation is via JAXP.

To validate a document, three things must happen:

1. The XML document must have a doctypedecl.
2. The application must set a SAX ErrorHandler.
3. The application must turn on validation.

Line 2 in Figure 1 shows a `doctypedecl`, simplified versions look like

```
<!DOCTYPE root-element SYSTEM system-id>

or

<!DOCTYPE root-element PUBLIC public-id system-id>
```

where `root-element`, `public-id`, and `system-id` depend on the specific markup language being used. The example in Figure 1 shows these values for a particular version of SVG.

A SAX `ErrorHandler` is an interface with three methods. The parser calls the `fatalError` method when there is a well-formedness error. The default implementation for this method is to throw an exception. The parser calls the `error` method when there is a validation error. The default implementation does nothing so the application must implement this method to get notified of validation errors. For example, the application may print out an error message indicating the problem and return, allowing further errors to be reported. Finally, the parser calls the `warning` method for less severe problems. The default implementation for this method does nothing so an application may instead want to print out a warning message.

The third step is to tell the parser to validate. SAX 2.0 provides a standard way to do this as well as JAXP 1.0.

```
1 XMLReader xmlReader = XMLReaderFactory.createXMLReader(driver);
2 String validation = "http://xml.org/sax/features/validation";
3 xmlReader.setFeature(validation, true);
```

```
4 // Set the new error handler
5 xmlReader.setErrorHandler(new MyErrorHandler());
6 // Parse the input
7 xmlReader.parse(url);
```

Figure 9 Code excerpt to turn on validation using SAX2

Figure 9 shows some code to do this using SAX2. Line 3 turns on validation and line 5 sets the error handler. The code for JAXP is similar to the DOM case so we will just provide an example of the DOM case. The following code excerpt shows example code for DOM using JAXP.

```
1 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
2 dbf.setValidating(true);
3 DocumentBuilder db = dbf.newDocumentBuilder();
4 db.setErrorHandler(new MyErrorHandler());
5 doc = db.parse(new File(args[0]));
```

Figure 10 Code excerpt to turn on validation for DOM

In Figure 10, line 2 turns on validation and line 4 sets the error handler.

JDOM

JDOM stands for Java Document Object Model. It is a newer API that was recently introduced within the past year in spring of 2000. Unlike SAX it provides a tree representation of a document but it is also different from DOM. JDOM is a Java-only API and seeks to provide a simpler tree model to Java applications than DOM. Currently, no parsers directly support JDOM by packaging it with their parsers, however, a JDOM tree can be built from a parser that produces SAX events or from a DOM tree. See http://www.jdom.org for more information about this API and how to download an implementation that can be used with other parsers. JDOM is also open sourced.

API Comparison

How does one choose which API to use between the three main Java APIs mentioned: SAX, DOM, and JDOM? The choice between SAX and the two other APIs is easiest. Many applications will probably require a tree representation of their document so this makes SAX more difficult to use because it will require the application to create appropriate data structures while listening to events. Also, applications that need to write out XML will not use SAX since writing XML output is not covered. However, SAX is also a lightweight, non-memory intensive API, which is also fast, so some applications will prefer this API. In fact, some DOM implementations are layered on top of the SAX API.

The virtues of DOM versus JDOM are currently under much debate so developers should investigate each API in more detail before making a decision. For example, if the application wants to write out XML text, DOM does not yet define a standard way to do this (load/save is slated for DOM3) so you must rely on parser specific APIs. JDOM does define an output API, which can be used with at least three different parsers. Also, since JDOM only has a Java API, it uses familiar Java2 classes to represent objects such as lists instead of special cross-language classes defined by DOM. Because of this some developers claim that JDOM is easier to use

with Java. JDOM also can make white space handling easier in some cases. However, other developers claim that it trades off usability at the cost of being less XML standards compliant.

Conclusions

In this paper we presented an overview of XML and skimmed the surface of several Java XML parsing APIs. XML is a large topic and there are many areas to cover. Hopefully this introductory paper has given you a flavor for what the common APIs look like as well as given you pointers to new areas to explore.

References

The following is a list of references to specifications and to related topics not all of which are covered in this paper. There are also some books that may explain some of these topics better than the specifications do. Also, many areas in the XML space change frequently so details contained in books may not be current depending on the topic. Refer to the specifications, which are usually available on the web for detailed information.

- XML spec and DTDs, see http://www.w3.org/TR/REC-xml
- SAX, see http://www.megginson.com/SAX/index.html
- DOM, see http://www.w3.org/TR/DOM-Level-2/
- JAXP, see http://java.sun.com/xml/docs/api/
- JDOM, see http://www.jdom.org/
- Namespaces, see http://www.w3.org/TR/REC-xml-names/
- XML Schema, see http://www.w3.org/TR/xmlschema-0/
- XSLT, see http://www.w3.org/TR/xslt
- XML data binding, see http://java.sun.com/aboutJava/communityprocess/jsr/jsr_031_xmld.html

The following is a list of XML parsers.

- Xerces, see http://xml.apache.org/
- Crimson, currently at http://xml.apache.org/cvs in /xml-contrib repository
- Aelfred2, see http://xmlconf.sourceforge.net/?selected=java
- Oracle (not open sourced), see http://www.oracle.com/xml/
- XP, see http://www.jclark.com/xml/xp/