JDOM and XML Parsing, Part 1

JDOM makes XML manipulation in Java easier than ever.

Chances are, you've probably used one of a number of Java libraries to manipulate XML data structures in the past. So what's the point of JDOM (Java Document Object Model), and why do developers need it?

JDOM is an open source library for Java-optimized XML data manipulations. Although it's similar to the World Wide Web Consortium's (W3C) DOM, it's an alternative document object model that was not built on DOM or modeled after DOM. The main difference is that while DOM was created to be language-neutral and initially used for JavaScript manipulation of HTML pages, JDOM was created to be Java-specific and thereby take advantage of Java's features, including method overloading, collections, reflection, and familiar programming idioms. For Java programmers, JDOM tends to feel more natural and "right." It's similar to how the Java-optimized remote method invocation library feels more natural than the language-neutral Common Object Request Broker Architecture.

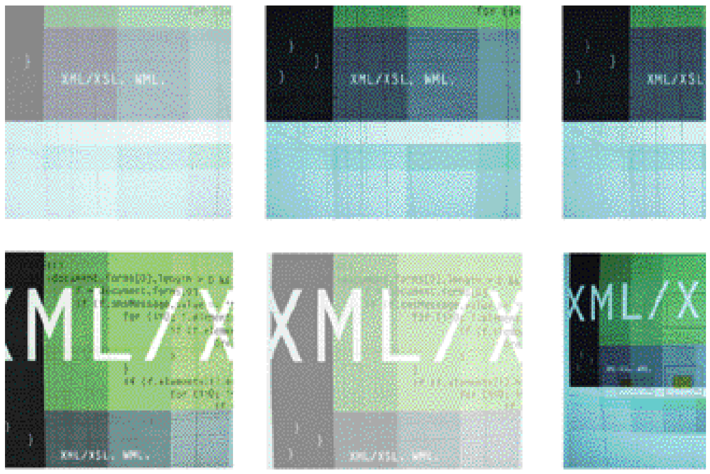
You can find JDOM at www.jdom.org under an open source Apache-style (commercial-friendly) license. It's collaboratively designed and developed and has mailing lists with more than 3,000 subscribers. The library has also been accepted by Sun's Java Community Process (JCP) as a Java Specification Request (JSR-102) and is on track to become a formal Java specification.

The articles in this series will provide a technical introduction to JDOM. This article provides information about important classes. The next article will give you a feel for how to use JDOM inside your own Java programs.

THE JDOM PACKAGE STRUCTURE

The JDOM library consists of six packages. First, the `org.jdom` package holds the classes representing an XML document and its components: `Attribute`, `CDATA`, `Comment`, `DocType`, `Document`, `Element`, `EntityRef`, `Namespace`, `ProcessingInstruction`, and `Text`. If you're familiar with XML, the class names should be self-explanatory.

Next is the `org.jdom.input` package. which holds classes that build XML documents. The main and most important class is `SAXBuilder`. `SAXBuilder` builds a document by listening to incoming SAX events and constructing a corresponding document. When you want to build from a file or other stream, you use `SAXBuilder`. It uses a SAX parser to read the stream and then builds



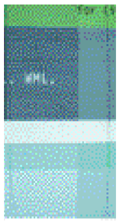
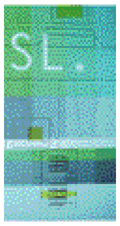
the document according to the SAX parser callbacks. The good part of this design is that as SAX parsers get faster, `SAXBuilder` gets faster. The other main input class is `DOMBuilder`. `DOMBuilder` builds from a DOM tree. This class comes in handy when you have a preexisting DOM tree and want a JDOM version instead.

There's no limit to the potential builders. For example, now that Xerces has the Xerces Native Interface (XNI) to operate at a lower level than SAX, it may make sense to write an `XNIBuilder` to support some parser knowledge not exposed via SAX. One popular builder that has been contributed to the project is the `ResultSetBuilder`. It takes a JDBC result set and creates an XML document representation of the SQL result, with various configurations regarding what should be an element and what should be an attribute.

The `org.jdom.output` package holds the classes that output XML documents. The most important class is `XMLOutputter`. It converts documents to a stream of bytes for output to files, streams, and sockets. The `XMLOutputter` has many special configuration options supporting raw output, pretty output, or compressed output, among others. It's a fairly complicated class. That's probably why this capability still doesn't exist in DOM Level 2.

Other outputters include the `SAXOutputter`, which generates SAX events based on the document content. Although seemingly arcane, this class proves extremely useful in XSLT transforms, because SAX events can be a more efficient way than bytes to transfer document data to an engine. There's also a `DOMOutputter`, which builds a DOM tree representation of the document. An

GETTY ONE/EYEWIRE

interesting contributed outputter is the `JTreeOutputter`, which—with just a few dozen lines of code—builds a JTree representation of the document. Combine that with the `ResultSetBuilder`, and you can go from a SQL query to a tree view of the result with just a couple of lines of code, thanks to JDOM.

Note that, unlike in DOM, documents are not tied to their builder. This produces an elegant model in which you have classes to hold data, various classes to construct data, and various other classes to consume the data. Mix and match as desired!

The `org.jdom.transform` and `org.jdom.xpath` packages have classes that support built-in XSLT transformations and XPath lookups.

Finally, the `org.jdom.adapters` package holds classes that assist the library in DOM interactions. Users of the library never need to call upon the classes in this package. They're there because each DOM implementation has different method names for certain bootstrapping tasks, so the adapter classes translate standard calls into parser-specific calls. The Java API for XML Processing (JAXP) provides another approach to this problem and actually reduces the need for these classes, but they've retained them because not all parsers support JAXP, nor is JAXP installed everywhere, due to license issues.

CREATING A DOCUMENT

Documents are represented by the `org.jdom.Document` class. You can construct a document from scratch:

```
// This builds: <root/>
Document doc = new Document(new Element("root"));
```

Or you can builkd a document from a file, stream, system ID, or URL:

```
// This builds a document of whatever's in the given resource
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(url);
```

Putting together a few calls makes it easy to create a simple document in JDOM:

```
// This builds: <root>This is the root</root>
Document doc = new Document();
Element e = new Element("root");
e.setText("This is the root");
doc.addContent(e);
```

If you're a power user, you may prefer to use "method chaining," in which multiple methods are called in sequence. This works because the set methods return the object on which they acted. Here's how that looks:

```
Document doc = new Document(
  new Element("root").setText("This is the root"));
```

For a little comparison, here's how you'd create the same document, using JAXP/DOM:

```
// JAXP/DOM
DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.newDocument();
Element root = doc.createElement("root");
Text text = doc.createText("This is the root");
root.appendChild(text);
doc.appendChild(root);
```

BUILDING WITH SAXBUILDER

As shown earlier, `SAXBuilder` presents a simple mechanism for building documents from any byte-oriented resource. The default no-argument `SAXBuilder()` constructor uses JAXP behind the scenes to select a SAX parser. If you want to change parsers, you can set the javax.xml.parsers.SAX ParserFactory system property to point at the SAXParser Factory implementation provided by your parser. For the Oracle9*i* Release 2 XML parser, you would do this:

```
java -
Djavax.xml.parsers.SAXParserFactory=oracle.xml.jaxp.JXSAXParser
Factory YourApp
```

For the Xerces parser, you would do this instead:

```
java -Djavax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.
SAXParserFactoryImpl YourApp
```

If JAXP isn't installed, `SAXBuilder` defaults to Apache Xerces. Once you've created a `SAXBuilder` instance, you can set several properties on the builder, including:

```
setValidation(boolean validate)
```

This method tells the parser whether to validate against a Document Type Definition (DTD) during the build. It's false (off) by default. The DTD used is the one referenced within the document's `DocType`. It isn't possible to validate against any other DTD, because no parsers support that capability yet.

```
setIgnoringElementContentWhitespace(boolean ignoring)
```

This method tells the parser whether to ignore what's called *ignorable whitespace* in element content. Per the XML 1.0 spec, whitespace in element content must be preserved by the parser, but when validating against a DTD it's possible for the parser to know that certain parts of a document don't declare to support whitespace, so any whitespace in that area is "ignorable." It's false (off) by default. It's generally good to turn this on for a little performance savings, unless you want to "round trip" a document and output the same content as was input. Note that this flag has an effect only if validation is on, and validation causes a performance slowdown, so this trick is useful only when validation is already in use.

```
setFeature(String name, String value)
```

This method sets a feature on the underlying SAX parser. This is a raw pass-through call, so be very careful when using this method, because setting the wrong feature (such as tweaking namespaces) could break JDOM behavior. Furthermore, relying on any parser-specific features could limit portability. This call is most useful for enabling schema validation.

```
setProperty(String name, Object value)
```

This method sets a property on the underlying SAX parser. It's also a raw pass-through call, with the same risks and the same usefulness to power users, especially for schema validation.

Putting together the methods, the following code uses the JAXP-selected parser to read a local file, with validation turned on and ignorable whitespace ignored.

```
SAXBuilder builder = new SAXBuilder();
builder.setValidation(true);
builder.setIgnoringElementContentWhitespace(true);
Document doc = builder.build(new File("/tmp/foo.xml"));
```

WRITING WITH XMLOUTPUTTER

A document can be output to many different formats, but the most common is a stream of bytes. In JDOM, the `XMLOutputter` class provides this capability. Its default no-argument constructor attempts to faithfully output a document exactly as stored in memory. The following code produces a raw representation of a document to a file.

WEBLOCATOR

Open source JDOM library
www.jdom.org

Java Servlet Programming by Jason Hunter (O'Reilly & Associates, 2001)
www.oreilly.com

```
// Raw output
XMLOutputter outp = new XMLOutputter();
outp.output(doc, fileStream);
```

If you don't care about whitespace, you can enable trimming of text blocks and save a little bandwidth:

```
// Compressed output
outp.setTextTrim(true);
outp.output(doc, socketStream);
```

If you'd like the document pretty-printed for human display, you can add some indent whitespace and turn on new lines:

```
outp.setTextTrim(true);
outp.setIndent("  ");
outp.setNewlines(true);
outp.output(doc, System.out);
```

When pretty-printing a document that already has formatting whitespace, be sure to enable trimming. Otherwise, you'll add formatting on top of formatting and make something ugly.

NAVIGATING THE ELEMENT TREE

JDOM makes navigating the element tree quite easy. To get the root element, call:

```
Element root = doc.getRootElement();
```

To get a list of all its child elements:

```
List allChildren = root.getChildren();
```

To get just the elements with a given name:

```
List namedChildren = root.getChildren("name");
```

And to get just the first element with a given name:

```
Element child = root.getChild("name");
```

The "List" returned by the `getChildren()` call is a java.util.List, an implementation of the List interface all Java programmers know. What's interesting about the List is that it's live. Any changes to the list are immediately reflected in the backing document.

```
// Remove the fourth child
allChildren.remove(3);
// Remove children named "jack"
allChildren.removeAll(root.getChildren("jack"));
// Add a new child, at the tail or at the head
allChildren.add(new Element("jane"));
allChildren.add(0, new Element("jill"));
```

ORACLE XML TOOLS

The XML Developer Kit (XDK) is a free library of XML tools Oracle provides for developers. It includes an XML parser and an XSLT transformation engine that can be used with JDOM. You can find lots of information about these tools at the Oracle XML home page, oracle.com/xml.

To download the parser, look for the XML Developer Kit with the name "XDK for Java." Click on "Software" in the left column for the download links. Once you unpack the distribution, the file `xmlparserv2.jar` contains the parser.

To configure JDOM and other software to use the Oracle parser by default, you need to set the JAXP javax.xml.parsers .SAXParserFactory system property to oracle.xml.jaxp.JXSAX ParserFactory. This tells JAXP that you prefer the Oracle parser. The easiest way is at the command line:

```
   java -
Djavax.xml.parsers.SAXParserFactory=oracle.xml.jaxp.JXSA
XParserFactory
```

You can also set this programmatically:

```
   System.setProperty("javax.xml.parsers.
   SAXParserFactory",
   "oracle.xml.jaxp.JXSAXParserFactory");
```

In addition to XDK, Oracle provides a native XML repository with Oracle9*i* Database Release 2. Oracle9*i* XML Database (XDB) is a high-performance, native XML storage and retrieval technology. It fully absorbs the W3C XML data model into Oracle9*i* Database and provides new standard access methods for navigating and querying XML. With XDB, you get all the advantages of relational database technology plus the advantages of XML technology.

Using the List metaphor makes possible many element manipulations without adding a plethora of methods. For convenience, however, the common tasks of adding elements at the end or removing named elements have methods on `Element` itself and don't require obtaining the List first:

```
root.removeChildren("jill");
root.addContent(new Element("jenny"));
```

One nice perk with JDOM is how easy it can be to move elements within a document or between documents. It's the same code in both cases:

```
Element movable = new Element("movable");
parent1.addContent(movable);    // place
parent1.removeContent(movable); // remove
parent2.addContent(movable);    // add
```

With DOM, moving elements is not as easy, because in DOM elements are tied to their build tool. Thus a DOM element must be "imported" when moving between documents.

With JDOM the only thing you need to remember is to remove an element before adding it somewhere else, so that you don't create loops in the tree. There's a `detach()` method that makes the detach/add a one-liner:

```
parent3.addContent(movable.detach());
```

If you forget to detach an element before adding it to another parent, the library will throw an exception (with a truly precise and helpful error message). The library also checks `Element` names and content to make sure they don't include inappropriate characters such as spaces. It also verifies other rules, such as having only one root element, consistent namespace declarations, lack of forbidden character sequences in comments and `CDATA` sections, and so on. This feature pushes "well-formedness" error checking as early in the process as possible.

HANDLING ELEMENT ATTRIBUTES

Element attributes look like this:

```
<table width="100%" border="0"> ... </table>
```

With a reference to an element, you can ask the element for any named attribute value:

```
String val = table.getAttributeValue("width");
```

You can also get the attribute as an object, for performing special manipulations such as type conversions:

```
Attribute border = table.getAttribute("border");
int size = border.getIntValue();
```

To set or change an attribute, use `setAttribute()`:

```
table.setAttribute("vspace", "0");
```

To remove an attribute, use `removeAttribute()`:

```
table.removeAttribute("vspace");
```

WORKING WITH ELEMENT TEXT CONTENT

An element with text content looks like this:

```
<description>
  A cool demo
</description>
```

In JDOM, the text is directly available by calling:

```
String desc = description.getText();
```

Just remember, because the XML 1.0 specification requires whitespace to be preserved, this returns "\n A cool demo\n". Of course, as a practical programmer you often don't need want to be so literal about formatting whitespace, so there's a convenient method for retrieving the text while ignoring surrounding whitespace:

```
String betterDesc = description.getTextTrim();
```

If you really want whitespace out of the picture, there's even a `getTextNormalize()` method that normalizes internal whitespace with a single space. It's handy for text content like this:

```
<description>
  Sometimes you have text content with formatting
  space within the string.
</description>
```

To change text content, there's a `setText()` method:

```
description.setText("A new description");
```

Any special characters within the text will be interpreted correctly as a character and escaped on output as needed to maintain the appropriate semantics. Let's say you make this call:

```
element.setText("<xml/> content");
```

The internal store will keep that literal string as characters. There will be no implicit parsing of the content. On output, you'll see this:

```
<elt>&lt;xml/&gt; content<elt>
```

This behavior preserves the semantic meaning of the earlier `setText()` call. If you want XML content held within an element, you must add the appropriate JDOM child element objects.

Handling `CDATA` sections is also possible within JDOM. A `CDATA` section indicates a block of text that shouldn't be parsed. It is essentially a "syntactic sugar" that allows the easy inclusion of HTML or XML content without so many &lt; and &gt; escapes. To build a `CDATA` section, just wrap the string with a `CDATA` object:

```
element.addContent(new CDATA("<xml/> content"));
```

What's terrific about JDOM is that a `getText()` call returns the string of characters without bothering the caller with whether or not it's represented by a `CDATA` section.

DEALING WITH MIXED CONTENT

Some elements contain many things such as whitespace, comments, text, child elements, and more:

```
<table>
  <!-- Some comment -->
  Some text
  <tr>Some child element</tr>
</table>
```

When an element contains both text and child elements, it's said to contain "mixed content." Handling mixed content can be potentially difficult, but JDOM makes it easy. The standard-use cases—retrieving text content and navigating child elements—are kept simple:

```
String text = table.getTextTrim();  // "Some text"
Element tr = table.getChild("tr");  // A straight reference
```

For more advanced uses needing the comment, whitespace blocks, processing instructions, and entity references, the raw mixed content is available as a List:

```
List mixedCo = table.getContent();
Iterator itr = mixedCo.iterator();
while (itr.hasNext()) {
  Object o = i.next();
  if (o instanceof Comment) {
    ...
  }
  // Types include Comment, Element, CDATA, DocType,
  // ProcessingInstruction, EntityRef, and Text
}
```

As with child element lists, changes to the raw content list affect the backing document:

```
// Remove the Comment.  It's "1" because "0" is a whitespace block.
mixedCo.remove(1);
```

If you have sharp eyes, you'll notice that there's a `Text` class here. Internally, JDOM uses a `Text` class to store string content in order to allow the string to have parentage and more easily support XPath access. As a programmer, you don't need to worry about the class when retrieving or setting text—only when accessing the raw content list.

For details on the `DocType`, `ProcessingInstruction`, and `EntityRef` classes, see the API documentation at www.jdom.org.

COMING IN PART 2

In this article we began examining how to use JDOM in your applications. In the next article, I examine XML Namespaces, `ResultSetBuilder`, XSLT, and XPath. You can find Part 2 of this series online now at otn.oracle.com/oraclemagazine. ■

Jason Hunter *(jasonhunter@servlets.com) is a consultant, publisher of Servlets.com, and vice president of the Apache Software Foundation. He holds a seat on the JCP Executive Committee.*