JAVAWORLD gration with JDOM, Part 1

e API for working with XML

By Jason Hunter and Brett Mclaughlin

JavaWorld | May 18, 2000 1:00 AM PT

JDOM is an open source API designed to represent an XML document and its contents to the typical Java developer in an intuitive and straightforward way. As the name indicates, JDOM is Java optimized. It behaves like Java, it uses Java collections, and it provides a low-cost entry point for using XML. JDOM users don't need to have tremendous expertise in XML to be productive and get their jobs done.

TEXTBOX: TEXTBOX_HEAD: Easy Java/XML integration with JDOM: Read the whole series!

- Part 1. Learn about a new API for working with XML
- Part 2. Use JDOM to create and mutate XML

:END_TEXTBOX

JDOM interoperates well with existing standards such as the Simple API for XML (SAX) and the Document Object Model (DOM). However, it's more than a simple abstraction above those APIs. JDOM takes the best concepts from existing APIs and creates a new set of classes and interfaces that provide, in the words of one JDOM user, "the interface I expected when I first looked at org.w3c.dom." JDOM can read from existing DOM and SAX sources, and can output to DOM- and SAX-receiving components. That ability enables JDOM to interoperate seamlessly with existing program components built against SAX or DOM.

JDOM has been made available under an Apache-style, open source license. That license is among the least restrictive software licenses available, enabling developers to use JDOM in creating products without requiring them to release their own products as open source. It is the license model used by the Apache Project, which created the Apache server. In addition to making the software free, being open source enables the API to take contributions from some of the best Java and XML minds in the industry and to adapt quickly to new standards as they evolve.

The JDOM philosophy

First and foremost, the JDOM API has been developed to be straightforward for Java programmers. While other XML APIs were created to be cross-language (supporting the same API for Java, C++, and even JavaScript), JDOM takes advantage of Java's abilities by using features such as method overloading, the Collections APIs, and (behind the scenes) reflection.

To be straightforward, the API has to represent the document in a way programmers would expect. For example, how would a Java programmer expect to get the text content of an element?

```
<element>This is my text content</element>
```

In some APIs, an element's text content is available only as a child `Node` of the `Element`. While technically correct, that design requires the following code to access an element's content:

```
String content = element.getFirstChild()
                        .getValue();
```

However, JDOM makes the text content available in a more straightforward way:

```
String text = element.getText();
```

Wherever possible, JDOM makes the programmer's job easier. The rule of thumb is that JDOM should help solve 80 percent or more of Java/XML problems with 20 percent or less of the traditional effort. That does not mean that JDOM conforms to only 80 percent of the XML specification. (In fact, we expect that JDOM will be fully compliant before the 1.0 final release.) What that rule of thumb does mean is that just because something could be added to the API doesn't mean it will. The API should remain sleek.

JDOM's second philosophy is that it should be fast and lightweight. Loading and manipulating documents should be quick, and memory requirements should be low. JDOM's design definitely allows for that. For example, even the early, untuned implementation has operated more quickly than DOM and roughly on par with SAX, even though it has many more features than SAX.

Do you need JDOM?

So, do you need JDOM? It's a good question. There are existing standards already, so why invent another one? The answer is that JDOM solves a problem that the existing standards do not.

DOM represents a document tree fully held in memory. It is a large API designed to perform almost every conceivable XML task. It also must have the same API across multiple languages. Because of those constraints, DOM does not always come naturally to Java developers who

expect typical Java capabilities such as method overloading, the use of standard Java object types, and simple set and get methods. DOM also requires lots of processing power and memory, making it untractable for many lightweight Web applications and programs.

SAX does not hold a document tree in memory. Instead, it presents a view of the document as a sequence of events. For example, it reports every time it encounters a begin tag and an end tag. That approach makes it a lightweight API that is good for fast reading. However, the event-view of a document is not intuitive to many of today's server-side, object oriented Java developers. SAX also does not support modifying the document, nor does it allow random access to the document.

JDOM attempts to incorporate the best of DOM and SAX. It's a lightweight API designed to perform quickly in a small-memory footprint. JDOM also provides a full document view with random access but, surprisingly, it does not require the entire document to be in memory. The API allows for future flyweight implementations that load information only when needed. Additionally, JDOM supports easy document modification through standard constructors and normal set methods.

Getting a document

JDOM represents an XML document as an instance of the `org.jdom.Document` class. The `Document` class is a lightweight class that can hold a `DocType`, multiple `ProcessingInstruction` objects, a root `Element`, and `Comment` objects. You can construct a `Document` from scratch without needing a factory:

```
Document doc = new Document(new Element("rootElement"));
```

In the next article, we'll discuss how easy it is to create an XML structure from scratch using JDOM. For now we'll construct our documents from a preexisting file, stream, or URL:

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(url);
```

You can build documents from any data source using builder classes found in the `org.jdom.input` package. Currently there are two builders, `SAXBuilder` and `DOMBuilder`. `SAXBuilder` uses a SAX parser behind the scenes to build the `Document` from the file; the `SAXBuilder` listens for the SAX events and builds a corresponding `Document` in memory. That approach is very fast (basically as fast as SAX), and it is the approach we recommend. `DOMBuilder` is another alternative that builds a JDOM `Document` from an existing `org.w3c.dom.Document` object. It allows JDOM to interface easily with tools that construct DOM trees.

JDOM's speed has the potential to improve significantly upon completion of a deferred builder that scans the XML data source but doesn't fully parse it until the information is requested. For example, element attributes don't need to be parsed until their value is requested.

Builders are also being developed that construct JDOM `Document` objects from SQL queries, LDAP queries, and other data formats. So, once in memory, documents are not tied to their build tool.

The `SAXBuilder` and `DOMBuilder` constructors let the user specify if validation should be turned on, as well as which parser class should perform the actual parsing duties.

```
public SAXBuilder(String parserClass, boolean validation);
public DOMBuilder(String adapterClass, boolean validation);
```

The defaults are to use Apache's open source Xerces parser and to turn off validation. Notice that the `DOMBuilder` doesn't take a *`parserClass`* but rather an *`adapterClass`*. That is because not all DOM parsers have the same API. To still allow user-pluggable parsers, JDOM uses an adapter class that has a common API for all DOM parsers. Adapters have been written for all the popular DOM parsers, including Apache's Xerces, Crimson, IBM's XML4J, Sun's Project X, and Oracle's parsers V1 and V2. Each one implements that standard interface by making the right method calls on the backend parser. That works somewhat similarly to JAXP (Resources) except it supports newer parsers that JAXP does not yet support.

Outputting a document

You can output a `Document` using an output tool, of which there are several standard ones available. The `org.jdom.output.XMLOutputter` tool is probably the most commonly used. It writes the document as XML to a specified `OutputStream`.

The `SAXOutputter` tool is another alternative. It generates SAX events based on the JDOM document, which you can then send to an application component that expects SAX events. In a similar manner, `DOMOutputter` creates a DOM document, which you can then supply to a DOM-receiving application component. The code to output a `Document` as XML looks like this:

```
XMLOutputter outputter = new XMLOutputter();
outputter.output(doc, System.out);
```

`XMLOutputter` takes parameters to customize the output. The first parameter is the indentation string; the second parameter indicates whether you should write new lines. For machine-to-machine communication, you can ignore the niceties of indentation and new lines for the sake of speed:

```
XMLOutputter outputter = new XMLOutputter("", false);
outputter.output(doc, System.out);
```

Here's a class that reads an XML document and prints it in a nice, readable form:

```
import java.io.*;
import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;
public class PrettyPrinter {
   public static void main(String[] args) {
        // Assume filename argument
        String filename = args[0];
        try {
            // Build the document with SAX and Xerces, no
validation
            SAXBuilder builder = new SAXBuilder();
            // Create the document
            Document doc = builder.build(new File(filename));
            // Output the document, use standard formatter
            XMLOutputter fmt = new XMLOutputter();
            fmt.output(doc, System.out);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Reading the DocType

Now let's look at how to read the details of a `Document`. One of the things that many XML documents have is a document type, represented in JDOM by the `DocType` class. In case you're not an XML guru (hey, don't feel bad, you're our target audience), a document type declaration looks like this.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

The first word after `DOCTYPE` indicates the name of the element being constrained, the word after `PUBLIC` is the document type's public identifier, and the last word is the document type's system identifier. The `DocType` is available by calling `getDocType()` on a `Document`, and the

DocType class has methods to get the individual pieces of the DOCTYPE declaration.

```
DocType docType = doc.getDocType();
System.out.println("Element: " + docType.getElementName());
System.out.println("Public ID: " + docType.getPublicID());
System.out.println("System ID: " + docType.getSystemID());
```

Reading the element data

Every XML document must have a root element. That element is the starting point for accessing all the information within the document. For example, that snippet of a document has <web-app> as the root:

```
<web-app id="demo">
 <description>Gotta fit servlets in somewhere!</description>
 <distributable/>
</web-app>
```

The root Element instance is available on a Document directly:

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

You can then access that Element's attributes (for example, the id above), content, and child Elements.

Playing with children

XML documents are tree structures, and any Element may contain any number of child Elements. For example, the <web-app> element has <description> and <distributable> tags as children. You can obtain an Element's children with various methods. getChild() returns null if no child by that name exists.

```
List getChildren(); // return all children
List getChildren(String name); // return all children by name
Element getChild(String name); // return first child by name
```

To demonstrate:

```
  // Get a List of all direct children as Element objects
  List allChildren = element.getChildren();
  out.println("First kid: " +
((Element)allChildren.get(0)).getName());
  // Get a list of all direct children with a given name
  List namedChildren = element.getChildren("name");
  // Get a list of the first kid with a given name
  Element kid = element.getChild("name");
```

Using getChild() makes it easy to quickly access nested elements when the structure of the XML document is known in advance. Given that XML:

```
<?xml version="1.0"?>
<linux:config>
  <gui>
    <window-manager>
      <name>Enlightenment</name>
      <version>0.16.2</version>
    </window-manager>
    <!-- etc -->
  </gui>
</linux:config>
```

That code directly retrieves the current window manager name:

```
String windowManager = rootElement.getChild("gui")
                                  .getChild("window-manager")
                                  .getChild("name")
                                  .getText();
```

Just be careful about NullPointerExceptions if the document has not been validated. For simpler document navigation, future JDOM versions are likely to support XPath references. Children can get their parent using getParent().

Getting element attributes

Attributes are another piece of information that elements hold. They're familiar to any HTML programmer. The following <table> element has width and border attributes.

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

View 9 Comments