JAVAWORLD

gration with JDOM, Part 2

e XML

By Jason Hunter and Brett Mclaughlin
JavaWorld | Jul 28, 2000 1:00 AM PT

In Part 1 of this series, we introduced you to JDOM, and discussed how you can use it to extract information from an existing XML data source such as a file, input stream, or URL. Additionally, we covered the core JDOM classes and explained how those classes work together to represent an XML document.

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

In this article, we tell the rest of the story and explain how you can modify XML documents or even create them from scratch. JDOM accomplishes those tasks, using standard conventions as much as possible, and lets you create, change, or move nearly every document component at runtime.

One word of warning before we begin: The JDOM API is in beta and subject to change until the 1.0 release. That release is expected within the next few months, but it may happen sooner or later, depending on when the JDOM community approves on the solidity of the design and implementation. Keep an eye on the JDOM Website for changes.

Creating a Document

In Part 1 you learned how to construct a JDOM `Document` from an external source:

```
SAXBuilder builder = new SAXBuilder();  // parameters control
validation, etc
Document doc = builder.build(url);
```

It's also possible and even easier to construct a JDOM Document from scratch with no existing XML data store:

```
Element root = new Element("myRootElement");
Document doc = new Document(root);
```

As simple as that, you can construct an Element and a Document using that Element as the document root. At that point, normal document manipulation can occur as well:

```
root.setText("This is a root element");
```

The above code sets the root element's text content to the given string. If the Document were to output now, using XMLOutputter as discussed in Part 1, the result would be the following XML:

```
<?xml version="1.0"?>
<root>This is a root element</root>
```

One major advantage of JDOM is that you don't need to use factories or other advanced programming models to create either the JDOM Document or constructs within the Document. You can compare that to DOM, which requires the following code to accomplish the same task: (We cannot compare it to SAX because SAX is a read-only API.)

```
// DOM code:
// Creating the document is vendor-specific, or requires the use
of JAXP
Document doc = new org.apache.xerces.dom.DocumentImpl();
// Create the root node and its text node, using the document as a
factory
Element root = myDocument.createElement("myRootElement");
Text text = myDocument.createText("This is a root element");
// Put the nodes into the document tree
root.appendChild(text);
myDocument.appendChild(root);
```

One thing you'll notice is that with DOM you don't have a standard way to create a Document. You have to either use implementation-specific commands or use Sun Microsystems' new JAXP API, which doesn't work with all parsers and doesn't yet support DOM2 and SAX 2.0. With JDOM, that is not an issue because JDOM documents are created through normal construction calls. You'll also notice that a DOM Node is always constructed through a factory method on its parent Document. That poses the strange chicken-and-egg problem that a DOM Element can only be

created using a DOM Document object, but an XML Document should never exist without a root element! Again, with JDOM that is not an issue because elements can be created independently of documents.

Additionally, it is obvious how you move JDOM elements between documents: you simply move the children. Consider this example, where parent1 and parent2 are Elements in different JDOM documents:

```
// Add to first document's element
Element movable = new Element("movableRootElement");
parent1.addContent(movable);
// Remove and add to another document's element
parent1.removeContent(movable);
parent2.addContent(movable);
```

That straightforward move operation is possible because the JDOM Element is not created by or tied to a specific Document. In DOM, to accomplish that you must import the Element from the first DOM Document to the second, and then perform the append; a direct move fails to execute.

```
// DOM code:
Element movable = doc1.createElement("movable");
parent1.appendNode(movable);
// Try to append to another document's element
parent2.appendNode(movable);
// This causes an error! Incorrect document!
```

As you can see here, one of JDOM's core features is simplifying the way you create and modify documents.

Creating an Element

Every JDOM Document should contain, at a minimum, a root Element. Each Element, in turn, can contain as few or as many child elements as needed to represent the data in the XML document. The root Element can be passed to a Document at the time it's created, as we showed earlier:

```
Element root = new Element("myRootElement");
Document doc = new Document(root);
```

To change a Document's root Element, you call setRootElement(Element element) on the Document:

```
Element newRoot = new Element("myNewRootElement");
doc.setRootElement(newRoot);
```

The constructor for `org.jdom.Element` takes the `Element`'s string name and can optionally take information about the `Element`'s namespace (which we will discuss later). This name is checked against XML 1.0 specification requirements for element naming; if the name is illegal, an `IllegalNameException` is thrown. To avoid having to look out for exceptions every time an `Element` is created, `IllegalNameException` extends `IllegalArgumentException` -- a runtime exception. Thus, you don't have to take any special actions when creating a new `Element` since JDOM will ensure only legal XML names are used. For example:

```
// This code will throw an IllegalNameException at runtime.
Element badElement = new Element("*(foo)");
```

That validation is handled by the `org.jdom.Verifier` class, which contains rules for all aspects of XML syntax. Applications can also use that class directly to ensure that a supplied name, such as in an XML editor, is correct before usage.

`Elements` without children most typically contain textual content. That content looks like this in XML:

```
<elementName>Here is some textual content</elementName>
```

In Part 1, we discussed that you can retrieve that text content with `getText()`:

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

That content can also be set through `setText(String text)`:

```
element.setText("My textual content");
```

JDOM simplifies the handling of special characters such as &, <, >, ', and " by dealing with them automatically. Simply set the content, using the raw string value and, upon output, the `XMLOutputter` class will automatically convert them to the proper XML entities that represent those characters. Of course, all builders will automatically convert them back to their traditional string values. To demonstrate, the following is perfectly legal in JDOM:

```
element.setText("Save cocoon.properties in <TOMCAT_HOME>/conf");
```

The brackets will be automatically converted to entity references when outputted through `XMLOutputter`:

```
Save cocoon.properties in &lt;TOMCAT_HOME&gt;/conf
```

Making kids

`Elements` often have child elements. To add a child to an element, you can use the `addContent(Element element)` method:

```
// Create a parent and two kids
Element parent = new Element("parent");
Element child1 = new Element("firstChild").setText("I'm number
one");
Element child2 = new Element("secondChild").setText("I'm number
two");
// Add the kids
parent.addContent(child1);
parent.addContent(child2);
```

The resulting XML would look like:

```
<parent>
  <firstChild>I'm number one</firstChild>
  <secondChild>I'm number two</secondChild>
</parent>
```

JDOM also supports a nesting shortcut, based on a design that may be familiar to users of other APIs such as BEA/WebLogic's htmlKona package and the Java Apache Element Construction Set (ECS). The idea is that operations on an `Element` return that `Element` for further manipulation:

```
Document doc = new Document(
  new Element("family")
    .addContent(new Element("mom"))
    .addContent(new Element("dad")
       .addContent("kidOfDad")));
```

Just be careful: It's the parenthesis location, not the indention, that determines the family tree. While that is handy in some cases, it may be confusing and error prone when constructing large JDOM Documents. If you don't want to use that feature, just ignore the returned value.

Because elements are constructed without factories, you can easily subclass the `Element` class, allowing for customizable elements as well as template elements. For example, if every XML document should have a common footer, you can construct that footer as a `FooterElement`:

```
root.addContent(new FooterElement());
```

Its contents might be something like this:

```
<footer>
  <copyright>
    JavaWorld 2000
  </copyright>
</footer>
```

You could write the `FooterElement` class like this:

```
public class FooterElement extends Element {
  public FooterElement() {
    super("footer");
    addContent("copyright").setText("JavaWorld 2000");
  }
}
```

When the `Element` is created, it automatically adds its child elements and all their content. You can expand that idea to construct a general template for copyrights using an additional constructor:

```
  public FooterElement(int copyrightYear) {
    super("footer");
    addContent("copyright").setText("JavaWorld " + copyrightYear);
  }
```

Next year, you could then use it as follows:

```
root.addContent(new FooterElement(2001));
```

Managing the population

In Part 1, we showed you how to get an `Element`'s children as a Java `List` object:

```
List children = element.getChildren();
```

What's exciting is that the returned list of children is live and can be used to directly manage the children -- any changes to the list affect the actual children of the element. By leveraging the Java Collections API, JDOM lets the programmer change the document, using an API that is well understood by developers, making manipulations easy to understand and learn. The following code demonstrates a few things you can do with the returned `List`:

```
List children = element.getChildren();
// Remove the third child
children.remove(3);
// Remove all children named "jack"
children.removeAll(element.getChildren("jack"));
// Add a new child
children.add(new Element("jane"));
// Add a new child in the second position
children.add(1, new Element("second"));
```

Of course, for common tasks there are convenience methods that don't require `List` manipulations:

```
// A non-List way to remove children named "jack"
element.removeChildren("jack");
// A non-List way to add a new child
element.addContent(new Element("jane"));
```

Setting element attributes

Manipulating element attributes is even simpler than manipulating children because attributes always have a single textual value. JDOM provides basic methods to add and remove `Attributes` to and from a JDOM `Element`:

```
table.addAttribute("vspace", "0");
table.removeAttribute("border");
```

Additionally, you can construct an `Attribute` directly and add the `Attribute` object to the `Element` through an overloaded *add*`Attribute()` method:

```
element.addAttribute(new Attribute("align", "right"));
```

Constructing an `Attribute` such as that makes most sense when the `Attribute` needs to be placed in a namespace, which we cover in the next section.

Similar to handling child elements, you can obtain the attributes of an `Element` in a Java `List` through the `getAttributes()` method.

```
// Get all attributes
List attributes = element.getAttributes();
// Remove all attributes
table.getAttributes().clear();
```

Also similar to `Elements`, the `Attribute` constructors will ensure that correct naming is used and throw `IllegalNameException` when an invalid name is supplied:

```
// This will throw a runtime exception, IllegalNameException
Attribute illegalAttribute = new Attribute("@lutris.com");
```

Working with Namespaces

XML Namespaces are a way of giving an XML name two dimensions. For example, consider an ambiguous element name such as table -- it could mean a table on which you eat, an HTML table, or a data table as in a spreadsheet. That confusion is caused by the one dimension of information. Additionally, you couldn't use two elements named table in the same document to mean different things -- you would have a *namespace collision*. However, the XML Namespaces recommendation aims to solve that by providing an additional dimension. If you knew that the second dimension of an element was furniture, it would be easy to understand what table means. That is represented in XML by prefacing the element name with another name (a namespace prefix) and separating the two dimensions with a colon:

```
<furniture:table />
```

That prefix then maps to a URI, which is the actual unique identifier, usually identified in the root element of an XML document, using an `xmlns` attribute:

```
<?xml version="1.0"?>
<root xmlns:furniture="http://www.havertys.com"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <furniture:table>
    <furniture:numChairs>4</furniture:numChairs>
    <furniture:cushion available="yes"
xlink:href="available_yes.jpg"/>
  </furniture:table>
</root>
```

In JDOM, the `org.jdom.Namespace` class represents an XML Namespace. You can obtain a namespace through that class by supplying the URI and a prefix to map to that URI:

```
Namespace furniture =
  Namespace.getNamespace("furniture", "http://www.havertys.com");
Namespace xlink =
  Namespace.getNamespace("xlink", "http://www.w3.org/1999/xlink");
```