JDOM: How It Works, and How It Opened the Java Process

by Jason Hunter

O'Reilly Open Source Convention 2001

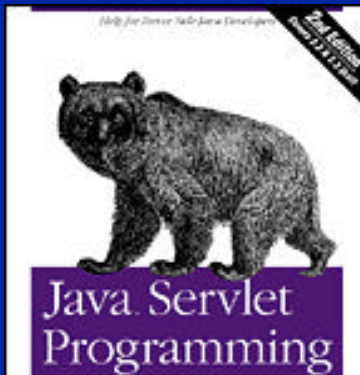July, 2001

Jason Hunter

jhunter@collab.net

CollabNet

http://collab.net
http://servlets.com

Author of
"Java Servlet Programming,
2nd Edition" (O'Reilly)

What is JDOM?

- JDOM is a way to represent an XML document for easy and efficient reading, manipulation, and writing
  - Straightforward API
  - Lightweight and fast
  - Java-optimized

- Despite the name similarity, it's not build on DOM or modeled after DOM
  - Although it integrates well with DOM and SAX

- An open source project with an Apache-style license
  - 1200 developers on jdom-interest (high traffic)
  - 1050 lurkers on jdom-announce (low traffic)

- JDOM should be straightforward for Java programmers
  - Use the power of the language (Java 2)
  - Take advantage of method overloading, the Collections APIs, reflection, weak references
  - Provide conveniences like type conversions

- JDOM should hide the complexities of XML wherever possible
  - An Element has content, not a child Text node with content
  - Exceptions should contain useful error messages
  - Give line numbers and specifics, use no SAX or DOM specifics

More JDOM Philosophy

- JDOM should integrate with DOM and SAX
  - Support reading and writing DOM documents and SAX events
  - Support runtime plug-in of *any* DOM or SAX parser
  - Easy conversion from DOM/SAX to JDOM
  - Easy conversion from JDOM to DOM/SAX

- JDOM should stay current with the latest XML standards
  - DOM Level 2, SAX 2.0, XML Schema

- JDOM does not need to solve every problem
  - It should solve 80% of the problems with 20% of the effort
 - We think we got the ratios to 90% / 10%

- JAXP wasn't around
  - Needed parser independence in DOM and SAX
  - Had user base using variety of parsers
  - Now integrates with JAXP 1.1
  - Expected to be part of JAXP version.next

- Why not use DOM:
  - Same API on multiple languages, *defined using IDL*
  - Foreign to the Java environment, Java programmer
  - Fairly heavyweight in memory

- Why not use SAX:
  - No document modification, random access, or output
  - Fairly steep learning curve to use correctly

JDOM Reading and Writing

(No Arithmetic)

Package Structure

- JDOM consists of five packages
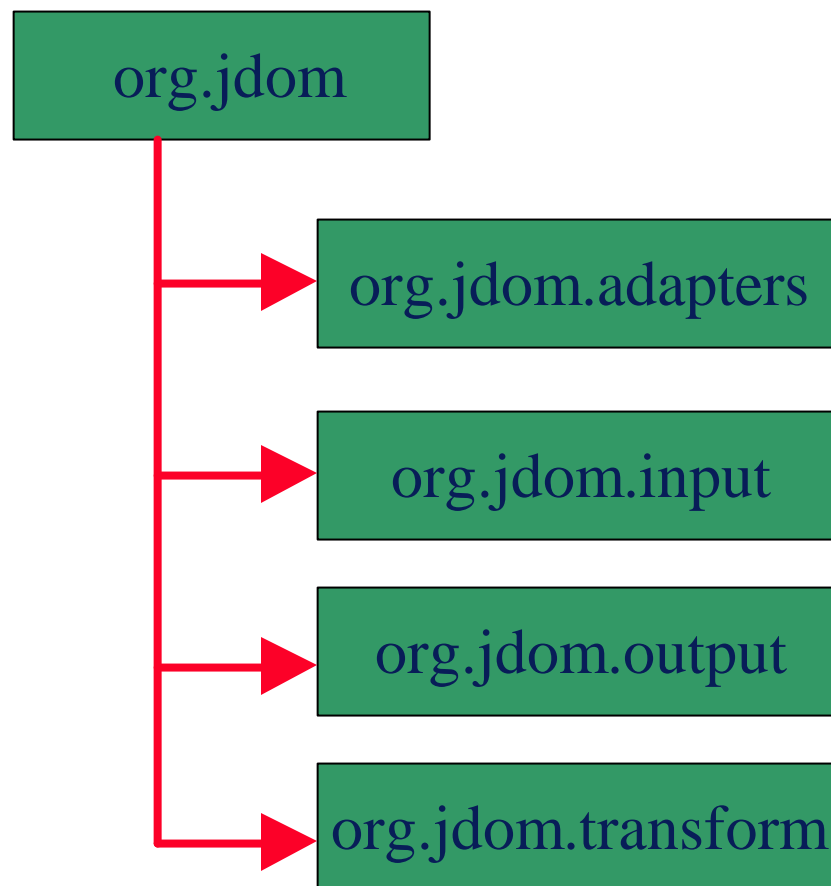
- These classes represent an XML document and XML constructs:
  - `Attribute`
  - `CDATA`
  - `Comment`
  - `DocType`
  - `Document`
  - `Element`
  - `EntityRef`
  - `Namespace`
  - `ProcessingInstruction`
  - (`PartialList`)
  - (`Verifier`)
  - (Assorted Exceptions)

- Classes for reading XML from existing sources:
  - `DOMBuilder`
  - `SAXBuilder`

- Also, outside contributions in jdom-contrib:
  - `ResultSetBuilder`
  - `SpitfireBuilder`

- New support for JAXP-based input
  - Allows consistency across applications
  - Builders pick up JAXP information and user automatically
 - Sets stage for JAXP version.next

- Classes for writing XML to various forms of output:
  - `DOMOutputter`
  - `SAXOutputter`
  - `XMLOutputter`

- Also, outside contributions in jdom-contrib:
  - `JTreeOutputter`

org.jdom.transform

- TRaX is now supported in **org.jdom.transform**
  - Supports XSLT transformations
  - Defines `Source` and `Result` interfaces
  - `JDOMSource`
  - `JDOMResult`

General Program Flow

- Normally XML Document -> SAXBuilder -> XMLOutputter
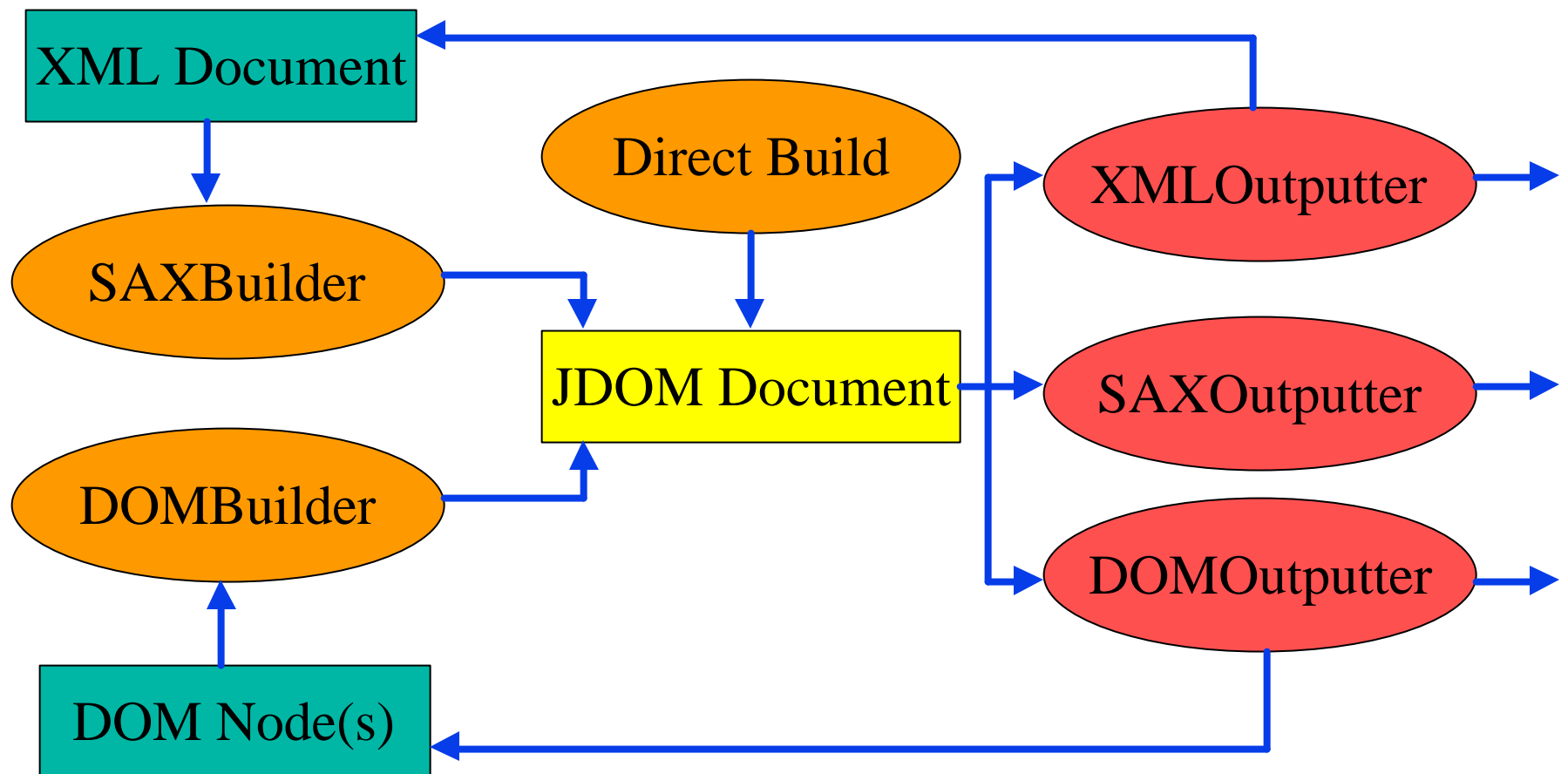
- Documents are represented by the **org.jdom.Document** class
  - A lightweight object holding a **DocType**, **ProcessingInstructions**, a root **Element**, and **Comments**

- It can be constructed from scratch:

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

- Or it can be constructed from a file, stream, or URL:

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

- Here's two ways to create a simple new document:

```
Document doc = new Document(
   new Element("rootElement")
   .setText("This is a root element"));
```

```
Document myDocument =
  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);
```

- A Document can be constructed using any build tool
  - The SAX build tool uses a SAX parser to create a JDOM document

- Current builders are SAXBuilder and DOMBuilder
  - `org.jdom.input.SAXBuilder` is fast and recommended
  - `org.jdom.input.DOMBuilder` is useful for reading an existing DOM tree
  - A builder can be written that lazily constructs the Document as needed
  - Other contributed builder: `ResultSetBuilder`

Builder Classes

- Builders have optional parameters to specify implementation classes and whether document validation should occur.

```
SAXBuilder(String parserClass, boolean validate);
DOMBuilder(String adapterClass, boolean validate);
```

- Not all DOM parsers have the same API
  - Xerces, XML4J, Project X, Oracle
  - The DOMBuilder `adapterClass` implements `org.jdom.adapters.DOMAdapter`
  - Implements standard methods by passing through to an underlying parser
  - Adapters for all popular parsers are provided
  - Future parsers require just a small adapter class

- Once built, documents are not tied to their build tool

- A Document can be written using any output tool
  - **`org.jdom.output.XMLOutputter`** tool writes the document as XML
  - **`org.jdom.output.SAXOutputter`** tool generates SAX events
  - **`org.jdom.output.DOMOutputter`** tool creates a DOM document
  - Any custom output tool can be used

- To output a **`Document`** as XML:

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

- For pretty-output, pass optional parameters
  - Two-space indent, add new lines

```
outputter = new XMLOutputter("  ", true);
```

```
import java.io.*; import org.jdom.*;
import org.jdom.input.*; import org.jdom.output.*;

public class InAndOut {
  public static void main(String[] args) {
    // Assume filename argument
    String filename = args[0];
    try {
      // Build w/ SAX and JAXP, no validation
      SAXBuilder b = new SAXBuilder();
      // Create the document
      Document doc = b.build(new File(filename));

      // Output as XML to screen
      XMLOutputter outputter = new XMLOutputter();
      outputter.output(doc, System.out);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

JDOM Core Functionality

The DocType class

- A **Document** may have a **DocType**

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

- This specifies the DTD of the document
  - It's easy to read and write

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

doc.setDocType(
  new DocType("html", "-//W3C...", "http://..."));
```

- A **Document** has a root **Element**:

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

- Get the root as an **Element** object:

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

- An **Element** represents something like **<web-app>**
  - Has access to everything from the open **<web-app>** to the closing **</web-app>**

- An element may contain child elements

```
// Get a List of direct children as Elements
List allChildren = element.getChildren();
out.println("First kid: " +
        ((Element)allChildren.get(0)).getName());

// Get all direct children with a given name
List namedChildren = element.getChildren("name");

// Get the first kid with a given name
Element kid = element.getChild("name");

// Namespaces are supported as we'll see later
```

- **getChild()** may return null if no child exists
- **getChildren()** returns an empty list if no children

```
<linux-config>
  <gui>
    <window-manager>
      <name>Enlightenment</name>
      <version>0.16.2</version>
    </window-manager>
    <!-- etc -->
  </gui>
</linux-config>
```

- Grandkids can be retrieved easily:

```
String manager =
    root.getChild("gui")
        .getChild("window-manager")
        .getChild("name")
        .getTextTrim();
```

- Just watch out for a `NullPointerException`!

- Children can be added and removed through **List** manipulation or convenience methods:

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

// Remove the fourth child
allChildren.remove(3);

// Remove all children named "jack"
allChildren.removeAll(
                element.getChildren("jack"));
element.removeChildren("jack");

// Add a new child
allChildren.add(new Element("jane"));
element.addContent(new Element("jane"));

// Add a new child in the second position
allChildren.add(1, new Element("second"));
```

- Moving elements is easy in JDOM but tricky in DOM

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

```
Element movable =
  doc1.createElement("movable");
parent1.appendChild(movable);  // place
parent1.removeChild(movable);  // remove
parent2.appendChild(movable);  // add
// This causes an error! Incorrect document!
```

- You need to call `importNode()` when moving between different documents

- Elements are constructed directly, no factory method needed

```
Element element = new Element("kid");
```

- Some prefer a nesting shortcut, possible since **addContent()** returns the **Element** on which the child was added:

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

- A subclass of **Element** can be made, already containing child elements

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

- The **Element** constructor (and all other object constructors) check to make sure the element is legal
  - i.e. the name doesn't contain inappropriate characters

- The add and remove methods also check document structure
  - An element may only exist at one point in the tree
  - Only one value can be returned by **getParent()**
  - No loops in the graph are allowed
  - Exactly one root element must exist

- This code constructs the `<linux-config>` seen previously:

```
Document doc = new Document(
  new Element("linux-config")
    .addContent(new Element("gui")
      .addContent(new Element("window-manager")
        .addContent(new Element("name")
            .setText("Enlightenment"))
      .addContent(new Element("version")
            .setText("0.16.2"))
    )
);
```

- Elements often contain attributes:

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

- Attributes can be retrieved several ways:

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

// Get "border" as an int
try {
  value =
    table.getAttribute("border").getIntValue();
}
catch (DataConversionException e) { }

// Passing default values was removed
// Good idea or not?
```

Setting Element Attributes

- Element attributes can easily be added or removed

```java
// Add an attribute
table.addAttribute("vspace", "0");

// Add an attribute more formally
table.addAttribute(
    new Attribute("name", "value"))

// Remove an attribute
table.removeAttribute("border");

// Remove all attributes
table.getAttributes().clear();
```

- Elements can contain text content:

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

- The text content is directly available:

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

- Whitespace must be preserved but often isn't needed, so we have a shortcut for removing extra whitespace:

```
// Remove surrounding whitespace
// Trim internal whitespace to one space
element.getTextNormalize();
```

- Element text can easily be changed:

```
// This blows away all current content
element.setText("A new description");
```

- Special characters are interpreted correctly:

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

- But you can also create CDATA:

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

- CDATA reads the same as normal, but outputs as CDATA.

JDOM Advanced Topics

- Sometimes an element may contain comments, text content, and children

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

- Text and children can be retrieved as always:

```
String text = table.getTextTrim();
Element tr = table.getChild("tr");
```

- This keeps the standard uses simple

- To get all content within an **Element**, use **getMixedContent()**
  - Returns a **List** containing **Comment**, **String**, **ProcessingInstruction**, **CDATA**, and **Element** objects

```
List mixedContent = table.getMixedContent();
Iterator i = mixedContent.iterator();
while (i.hasNext()) {
  Object o = i.next();
  if (o instanceof Comment) {
    // Comment has a toString()
    out.println("Comment: " + o);
  }
  else if (o instanceof String) {
    out.println("String: " + o);
  }
  else if (o instanceof Element) {
    out.println("Element: " +
                ((Element)o).getName());
  }
}
```

- The list of mixed content provides direct control over all the element's content.

```
List mixedContent = table.getMixedContent();

// Add a comment at the beginning
mixedContent.add(
   0, new Comment("Another comment"))

// Remove the comment
mixedContent.remove(0);

// Remove everything
mixedContent.clear();
```

XML Namespaces

- Namespaces are a DOM Level 2 addition

- Namespaces allow elements with the same local name to be treated differently
  - It works similarly to Java packages and helps avoid name collisions.

- Namespaces are used in XML like this:

```
<html xmlns:xhtml="http://www.w3.org/1999/xhtml">
   <!-- ... -->
   <xhtml:title>Home Page</xhtml:title>
</html>
```

- Namespace prefix to URI mappings are held statically in the `Namespace` class

- They're declared in JDOM like this:

```
Namespace xhtml = Namespace.getNamespace(
    "xhtml", "http://www.w3.org/1999/xhtml");
```

- They're passed as optional parameters to most element and attribute manipulation methods:

```
List kids = element.getChildren("p", xhtml);
Element kid = element.getChild("title", xhtml);
Attribute height = element.getAttribute(
    "height", xhtml);
```

- The current implementation uses `ArrayList` for speed
  - Will be migrating to a `FilterList`
  - Note that viewing a subset slows the relatively rare index-based access

- `List` objects are mutable
  - Modifications affect the backing document
  - Other existing list views do not currently see the change, but will with FilterList

- Because of its use of collections, JDOM requires JDK 1.2+ support, or JDK 1.1 with `collections.jar`

- Currently JDOM is at Beta 7

- Pending work:
  - Preserve internal DTD subsets
  - Polish the high-end features of the outputter
  - Discussion about Namespace re-factoring
  - Some well-formedness checking work to be done
  - Formal specification

- Speed and memory optimizations yet to be done!

- Some possible extensions to JDOM:
  - XPath (already quite far along, and usable)
  - XLink/XPointer (follows XPath)
  - XSLT (natively, now uses Xalan)
  - In-memory validation

JDOM as JSR-102

- In late February, JDOM was accepted by the Java Community Process (JCP) as a Java Specification Request (JSR-102)

- Sun's comment with their YES vote:
  - In general we tend to prefer to avoid adding new APIs to the Java platform which replicate the functionality of existing APIs. However JDOM does appear to be significantly easier to use than the earlier APIs, so we believe it will be a useful addition to the platform.

- What exactly does this mean?
  - Facilitates JDOM's corporate adoption
  - Opens the door for JDOM to be incorporated into the core Java Platform
  - JDOM will still be released as open source software
  - Technical discussion will continue to take place on public mailing lists

- For more information:
  - http://java.sun.com/aboutJava/communityprocess/jsr/jsr_102_jdom.html

- Jason Hunter is the "Specification Lead"

- The initial "Expert Group" (in order of acceptance):
    - Brett McLaughlin (individual, from Lutris)
    - Jools Enticknap (individual, software consultant)
    - James Davidson (individual, from Sun Microsystems and an Apache member)
    - Joe Bowbeer (individual, from 360.com)
    - Philip Nelson (individual, from Omni Resources)
    - Sun Microsystems (Rajiv Mordani)
    - CAPS (Bob McWhirter)

- Many other individuals and corporations have responded to the call for experts, none are yet official

- The JCP follows a benevolent dictator model
  - Strong spec lead making decisions based on input
  - Leaders may be deposed by a 2/3 vote of experts
  - But the replacement is from the same company!
  - What happens if you depose an individual?

- Open source RIs and TCKs are legit
  - Although the PMO is still learning about this
  - See JSR-053 (Servlets/JSPs), JSR-052 (Taglibs)
  - See JSR-080 (USB) which hit resistance

- Open source independent implementations?
  - Not technically allowed!!
  - Must enforce compatibility requirements, which violates open source; must pass costly TCK

- Unlike all other JSRs, JDOM discussion is public
  - We see no reason to work behind NDAs
  - On design issues the list keeps us in touch with people's needs, and people often step up to solve issues (i.e. long term serialization)
  - We use [eg] in the subject line for EG topics

- Unlike most other JSRs, the JDOM implementation leads the JDOM specification
  - Words on paper don't show all the issues
  - Witness JSR-047 (Logging)

- What's the role of an expert?
  - Similar to that of an Apache Member
  - Long-term commitment to help as needed

- Download the software
  - **http://jdom.org**

- Read the docs
  - **http://jdom.org**

- Sign up for the mailing lists (see **jdom.org**)
  - **jdom-announce**
  - **jdom-interest**

- *Java and XML*, by Brett McLaughlin
  - **http://www.oreilly.com/catalog/javaxml**

- Help improve the software!