Java + XML = JDOM

by Jason Hunter
and Brett McLaughlin

Enterprise Java O’Reilly Conference
March, 2001
Jason Hunter
jhunter@collab.net
CollabNet
http://collab.net
http://servlets.com

Author of
"Java Servlet Programming"
(O'Reilly)
Introductions

Brett McLaughlin
brett@jdom.org
Lutris Technologies
http://enhydra.org
http://www.newInstance.com

Author of
"Java and XML"
(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
  - 1050 developers on jdom-interest (high traffic)
  - 900 lurkers on jdom-announce (low traffic)
The JDOM Story

• XML Tutorial at Enterprise Java Conference, 2000
  – Brett details usage of DOM
  – Jason scratches head
  – Brett and Jason share DOM war stories
  – Decision to create JDOM is made

• Architecture
  – Initially based on interfaces
  – Reviewed by James Davidson (Sun) and Pier Fumagalli (Apache)
  – Moved to concrete classes
  – Released Alpha version to community
• 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%
Scratching an Itch

- 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
Do you need JDOM?

- JDOM is a lightweight API
  - Its design allows it to hold less in memory

- JDOM can represent a full document
  - Possible (but not implemented) to build a document where not all must be in memory at once

- JDOM supports document modification
  - And document creation from scratch, no "factory"

- JDOM is easy to learn
  - Optimized for Java programmers
  - Doesn't require in-depth XML knowledge
  - Allows easing into SAX and DOM, if needed
  - Easy to use namespaces, validation
JDOM Reading and Writing

(No Arithmetic)
JDOM consists of four packages:

- org.jdom
- org.jdom.adapters
- org.jdom.input
- org.jdom.output
The org.jdom Package

- These classes represent an XML document and XML constructs:
  - Attribute
  - CDATA
  - Comment
  - DocType
  - Document
  - Element
  - Entity
  - Namespace
  - ProcessingInstruction
  - (PartialList)
  - (Verifier)
  - (Assorted Exceptions)
The org.jdom.adapters Package

- Classes for hooking up JDOM to DOM implementations:
  - AbstractDOMAdapter
  - OracleV1DOMAdapter
  - OracleV2DOMAdapter
  - ProjectXDOMAdapter
  - XercesDOMAdapter
  - XML4JDOMAdapter
  - CrimsonDOMAdapter

- Rarely accessed directly (used in DOMBuilder and DOMOutputter)
The org.jdom.input Package

- 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

• TRAX integration in progress
  – TRAX is part of JAXP 1.1
  – Defines Source and Result interfaces
  – Can use JDOMSource, JDOMResult
  – Can use SAXSource, SAXResult subclasses
• Classes for writing XML to various forms of output:
  – DOMOutputter
  – SAXOutputter
  – XMLOutputter

• Also, outside contributions in jdom-contrib:
  – JTreeOutputter
• Normally XML Document -> SAXBuilder -> XMLOutputter
The Document class

- Documents are represented by the `org.jdom.Document` class
  - A lightweight object holding a `DocType`, `ProcessingInstruction`, a root `Element`, and `Comments`

- It can be constructed from scratch:

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

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

  ```java
  SAXBuilder builder = new SAXBuilder();
  Document doc = builder.build(url);
  ```
Here's two ways to create a simple new document:

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

```java
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.createTextNode("This is a root element");

// Put the nodes into the document tree
root.appendChild(text);
```
The Build Process

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

```java
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
The Output Process

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

- For pretty-output, pass optional parameters
  - Two-space indent, add new lines
  ```java
  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
• A Document may have a DocType

```xml
<!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

```java
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://..."));
```
The Element class

- A Document has a root Element:

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

- Get the root as an Element object:

```java
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>`
Playing with Children

- An element may contain child elements

```java
// 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
Playing with Grandchildren

• Grandkids can be retrieved easily:

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

• Just watch out for a `NullPointerException`!
Managing the Population

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

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

// Remove the third 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"));
```
JDOM vs DOM

- Moving elements is easy in JDOM but tricky in DOM

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

```java
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

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

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

```java
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

```java
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:

```java
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"))
        )
    );
```
• Imagine every document has a footer

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

• You could write a `FooterElement`:

```java
public class FooterElement extends Element {
    public FooterElement(int year) {
        super("footer");
        addContent(new Element("copyright")
                    .setText("JavaWorld " + year));
    }
}
```
• Other ideas for custom elements:
  – An element that uses the proxy pattern to defer parsing all document text until required
  – An element that stores application-specific information
  – An element that auto-conforms to a DTD

• Different builders can create different *Element* subclasses
Elements often contain attributes:

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

Attributes can be retrieved several ways:

```java
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?
```
• Element attributes can easily be added or removed

```javascript
// 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:

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

The text content is directly available:

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

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

```java
// Remove surrounding whitespace
// Trim internal whitespace to one space
element.getTextTrim();
```
• Element text can easily be changed:

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

• Special characters are interpreted correctly:

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

• But you can also create CDATA:

```java
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

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

• Text and children can be retrieved as always:

```java
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

```java
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.

```java
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();
```
The ProcessingInstruction class

- Some elements have `ProcessingInstructions`

```xml
<?cocoon-process type="xslt"?>
```

- PIs can be retrieved using `getMixedContent()` and their "attribute" values are directly available:

```java
if (o instanceof ProcessingInstruction) {
    ProcessingInstruction pi = (ProcessingInstruction) o;
    out.println(pi.getTarget());
    out.println(pi.getValue("type"));
    out.println(pi.getData()); // all data
}
```
• When in their common place at the document level outside the root element, PIs can be retrieved by name:

```java
ProcessingInstruction cp = 
    doc.getProcessingInstruction("cocoon-process");

    cp.getValue("type");
```
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:

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

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

- They're declared in JDOM like this:

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

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

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

- The current implementation uses `LinkedList` for speed
  - Speeds growing the `List`, modifying the `List`
  - Slows the relatively rare index-based access

- All `List` objects are mutable
  - Modifications affect the backing document
  - Other existing list views do not see the change
  - Same as SQL `ResultSet`s, etc.

- Because of its use of collections, JDOM requires JDK 1.2+ support, or JDK 1.1 with `collections.jar`
• **JDOMException** is the root exception
  – Thrown for build errors
  – Always includes a useful error message
  – May include a "root cause" exception

• Subclasses include:
  – **IllegalAddException**
  – **IllegalDataException**
  – **IllegalNameException**
  – **IllegalTargetException**
  – **DataConversionException**
• Currently JDOM is at Beta 6

• 95% of XML vocabularies compliance
  – Some work to be done for IDs and IDREFs
  – Discussion about Namespace re-factoring
  – Entities and inline DTDs still in progress
  – Some well-formedness checking work to be done

• 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
Case Studies: JDOM in Use
• Uses JDOM in all cases (no DOM or SAX)
  – Improved performance from DOM by 50% in sample documents
  – Reduced code size by 40% over DOM usage in original version
  – Greatly reduced development time

• Why not SAX?
  – Required extensive data structures to be developed for schema processing
  – Significantly complicated code base

• Why not DOM?
  – Code bloat
  – Performance
  – High barrier for community involvement
• Uses JDOM in all cases (no DOM or SAX)
  – Perfect for quick proof of concept
  – Allowed articles to focus on validation, not DOM or SAX tricks
  – Catered to all programmers, not just XML gurus

• Why not SAX?
  – Callback methodology not familiar to servlet programmers
  – Needed write capabilities

• Why not DOM?
  – Simplicity
  – Readability
• Provide JDOM alternative to DOM
  – Let developers choose the preferred tool
  – JDOM fits in with the Enhydra philosophy

• Why not SAX?
  – In-memory representation needed
  – Accessor/mutator methods desired

• Why is DOM not sufficient?
  – Requires HTML binding for DOM
  – Limits XMLC to a specific parser at distribution time
  – DOM not adapting to XMLC’s needs at any degree of speed
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 It Means

• 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:
• 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)

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

• The responsibilities of an EG member when discussion is public has yet to be determined
You Too Can Get Involved!

- 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!