Java + XML = JDOM

by Jason Hunter

and Brett McLaughlin

XML DevCon

November, 2000

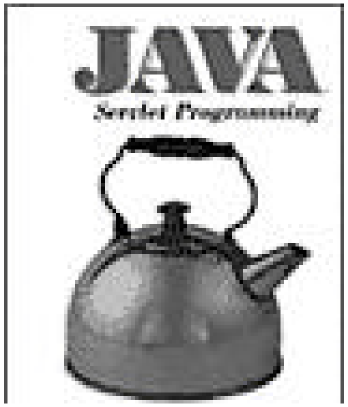
Introductions

Jason Hunter

jhunter@collab.net

CollabNet

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



Author of

"Java Servlet Programming"

(O'Reilly)

Introductions

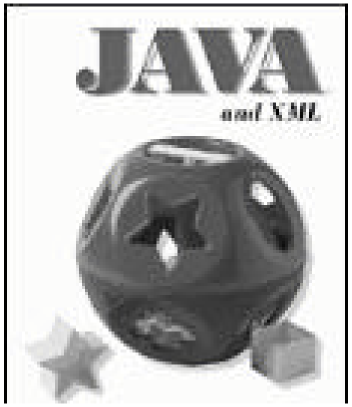
Brett McLaughlin

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

The JDOM Philosophy

- 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%

The Historical Alternatives: DOM

- DOM is a large API designed for complex environments
  - A W3C standard, developed by W3C working groups
  - Implemented by products like Xerces
  - Represents a document tree fully held in memory
  - Has to have the same API on multiple languages
  - Reading and changing the document is non-intuitive
  - Fairly heavyweight to load and store in memory
  - http://www.w3.org/DOM

The Historical Alternatives: SAX

- SAX is a lightweight API designed for fast reading
  - Public domain API from David Megginson and XML-DEV mailing list
  - Implemented by products like Xerces
  - Callback mechanism reports when document elements are encountered
  - Lightweight since the document is never entirely in memory
  - Does not support modifying the document
  - Does not support random access to the document
  - 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
  - Not all must be in memory at once

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

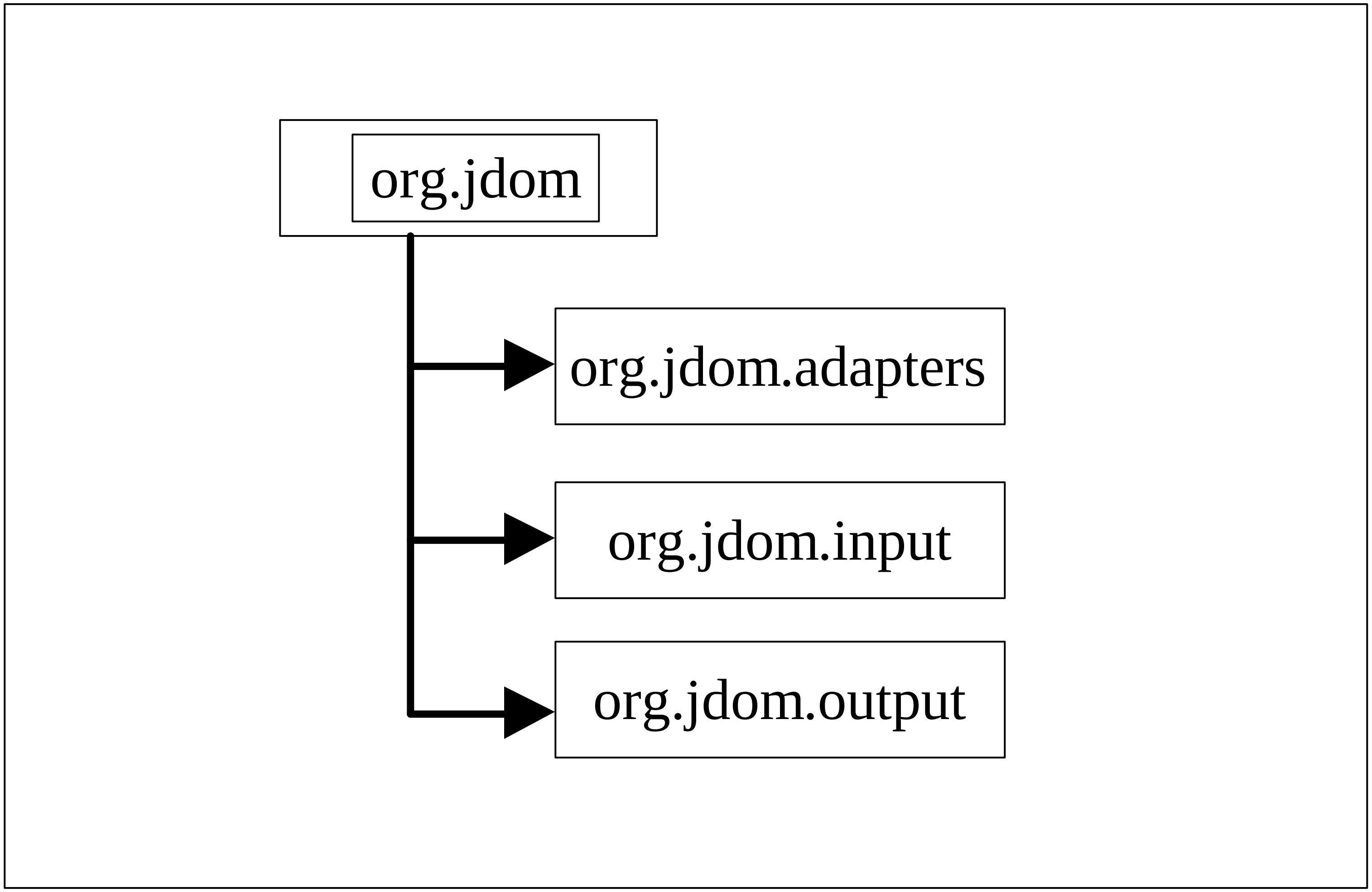
- JDOM is easy to learn
  - Optimized for Java programmers
  - Doesn't require in-depth XML knowledge
  - Allows easing into SAX and DOM, if needed
  - Easy to use namespaces, validation

JDOM Reading and Writing

(No Arithmetic)

Package Structure

- 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` (coming soon)
- Rarely accessed directly (used in `DOMBuilder`)

The org.jdom.input Package

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

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

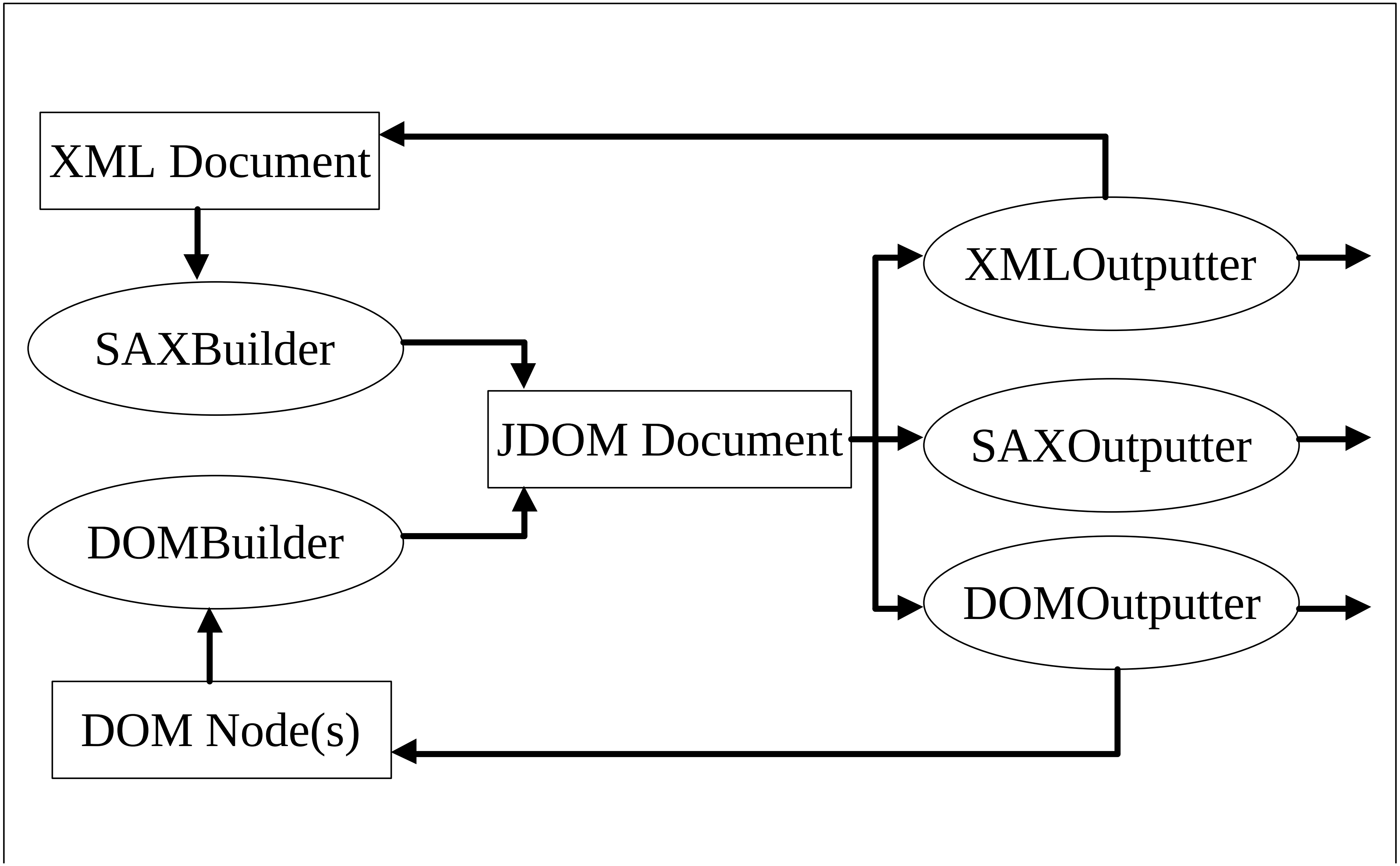
The org.jdom.output Package

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

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

General Program Flow

- Normally XML Document -> SAXBuilder -> XMLOutputter



The Document class

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

- 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);
```

JDOM vs DOM

- 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);
```

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`**

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

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:

```
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);
outputter.output(doc, System.out);
```

In and Out

```
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 Xerces, 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://..."));
```

The Element class

- 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>**

Playing with Children

- 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

Playing with Grandchildren

```
<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`**!

Managing the Population

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

```
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

```
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

Making Kids

- 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());
```

Ensuring Well-Formedness

- 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

Making the <linux-config>

- 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"))
    )
);
```

Custom Elements

- Imagine every document has a footer

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

- You could write a **FooterElement**:

```
public class FooterElement extends Element {
  public FooterElement(int year) {
    super("footer");
    addContent(new Element("copyright")
      .setText("JavaWorld " + year));
  }
}
```

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

Other Custom Elements

- 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

Getting Element Attributes

- Elements often contain attributes:

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

- Attributes can be retrieved several ways:

```
String value =
   table.getAttribute("width").getValue();

// 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

```
// 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();
```

Reading Element Content

- 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.getTextTrim();
```

Writing Element Content

- 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

Mixed Content

- 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

Reading Mixed Content

- 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());
  }
```

Manipulating Mixed Content

- 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();
```

The ProcessingInstruction class

- Some elements have **ProcessingInstruction**s

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

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

```
if (o instanceof ProcessingInstruction) {
  ProcessingInstruction pi =
    (ProcessingInstruction) o;
  out.println(pi.getTarget());
  out.println(pi.getValue("type"));
}
```

The ProcessingInstruction class

- When in their common place at the document level outside the root element, PIs can be retrieved by name:

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

XML Namespaces

- Namespaces are a DOM Level 2 addition
  - JDOM always supports even with DOM Level 1 parsers and even with validation on!

- 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>
```

JDOM Namespaces

- 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);
```

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`**

Exceptions

- **`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`**

The Push to 1.0

- Currently JDOM is at Beta 5

- 95% of XML vocabularies compliance
  - Some work to be done for IDs and IDREFs
  - Discussion about Namespace re-factoring
  - Inline DTDs still in progress
  - In-memory validation and well-formedness in progress

- Speed and memory optimizations yet to be done

Extending JDOM

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

API Standardization

- JDOM is likely to become a Java Specification Request (JSR)
  - Intended for inclusion in JAXP 1.2+ (or 2.0+)
  - By association, intended for inclusion in a future version of the JDK

- Possible standardization by OASIS or related technical ratification group

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!