Docx4j - Getting Started

This guide is for docx4j **2.7.1**.

The latest version of this document can always be found in docx4j svn in /docs (in Flat OPC XML format for Word 2007, HTML, and PDF).

The most up to date copy of this document is in English. From time to time, it is machine translated into other languages.

What is docx4j?

docx4j is a library for unzipping a docx "package", and parsing the WordprocessingML XML to create an in-memory representation in **Java**. Recent versions of docx4j also support Powerpoint pptx files and Excel xlsx files.

It is similar in concept to Microsoft's OpenXML SDK, which is for .NET.

docx4j is open source, available under the Apache License (v2). As an open source project, contributions are welcome. Please see the docx4j forum at http://dev.plutext.org/forums/ for details.

The Docx4j social contract

docx4j is currently available under the Apache Software license. This gives you freedom to do pretty much anything you like with it. It also means you don't have to pay for it (there is no incentive to take up a commercial license, so we don't offer one).

The ***quid pro quo*** is that if docx4j helps you out, you should ***please*** "give something back", by way of code, community support, by "spreading the word" (promotion), or by buying commerical support. Your choice. docx4j needs you help to make it easier for people to find it.

If you choose promotion, your options include:

- emailing to jharrop@plutext.com a testimonial which we can put on our website (preferably with your organization name, but without is worthwhile as well),
- a blog post, a tweet, or a helpful (non-spammy) comment in an online forum,
- sharing the content on our blog, following jasonharrop on Twitter, or connecting on LinkedIn.

Your promotion/support will help grow the docx4j community and thus its strength, to the benefit of all.

Please complete our very short new user survey at http://www.plutext.com/limesurvey/index.php?sid=78372. It includes a question on the above. Thanks.

Docx4j relies heavily on **JAXB**, the JCP standard for Java - XML binding. You can think of docx4j as a JAXB implementation of (amongst others):

- Open Packaging Conventions
- WordProcessingML (docx) part of Open XML
- Presentation ML (pptx) part of OpenXML
- SpreadsheetML (xlsx) part of Open XML.

The library is designed to round trip docx files with 100% fidelity, and supports all 2007 WordML. Support for new Word 2010 features will be added soon.

The docx4j project is sponsored by Plutext (www.plutext.com).

Is docx4j for you?

Docx4j is for processing docx documents (and pptx presentations and xlsx spreadsheets) in Java.

It isn't for old binary (.doc) files. If you wish to invest your effort around docx (as is wise), but you also need to be able to handle old doc files, see further below for your options.

Nor is it for RTF files.

If you want to process docx documents on the .NET platform, you should look at Microsoft's OpenXML SDK instead.

An alternative to docx4j is Apache POI. I'd particularly recommend that if you are only processing Excel documents, and need support for the old binary xls format. Since POI uses XmlBeans (not JAXB) it may be a better choice if you want to use XmlBeans.

What sorts of things can you do with docx4j?

- Open existing docx (from filesystem, SMB/CIFS, WebDAV using VFS), pptx, xlsx
- Create new docx, pptx, xlsx
- Programmatically manipulate the above (of course)

Specific to docx4j (as opposed to pptx4j, xlsx4j):

- Template substitution; CustomXML binding
- Produce/consume Word 2007's xmlPackage (pkg) format
- Save docx to filesystem as a docx (ie zipped), or to JCR (unzipped)

- Apply transforms, including common filters
- Export as HTML or PDF
- Diff/compare documents, paragraphs or sdt (content controls)
- Font support (font substitution, and use of any fonts embedded in the document)

This document focuses primarily on docx4j, but the general principles are equally applicable to pptx4j and xlsx4j.

docx4all is an example of an application based on docx4j; its a Swing-based wordprocessor for docx documents. You can try it or download its source code at **dev.plutext.org**

What Word documents does it support?

Docx4j can read/write docx documents created by or for Word 2007, or earlier versions which have the compatibility pack installed.

The relevant parts of docx4j are generated from the ECMA schemas.

It can't read/write Word 2003 XML documents. The main problem with those is that the XML namespace is different.

Docx4j 2.7.1 handles Word 2010 specific features, by gracefully degrading to the specified 2007

For more information, please see ***Specification versions*** below.

Handling legacy binary .doc files

Apache POI's HWPF can read .doc files, and docx4j could use this for basic conversion of .doc to .docx. The problem with this approach is that POI's HWPF code fails on many .doc files.

An effective approach is to use OpenOffice (via jodconverter) to convert the doc to docx, which docx4j can then process. If you need to return a binary .doc, OpenOffice/jodconverter can convert the docx back to .doc.

There is also http://b2xtranslator.sourceforge.net/ . If a pure Java approach were required, this could be converted.

Using docx4j binaries

You can download the latest version of docx4j from http://dev.plutext.org/docx4j/

In general, we suggest you develop against a currently nightly build, since the latest formal release can often be several months old.

Supporting jars can be found in the .tar.gz version, or in the relevant subdirectory.

Command Line Samples

With docx4j version 2.6.0, there are several samples you can run right away from the command line.

The two to try (both discussed in detail further below) are:

- OpenMainDocumentAndTraverse
- PartsList

Invoke with a command like:

```
java -cp docx4j.jar:log4j-1.2.15.jar org.docx4j.samples.OpenMainDocumentAndTraverse [input.docx]
```

If there are any images in the docx, you'd also need:

xmlgraphics-commons-1.4.jar

commons-logging-1.1.1.jar

on your classpath.

docx4j dependencies

log4j

To do anything with docx4j, you need **log4j** on your classpath.

To actually enable logging, you need a log4.properties or log4j.xml on your class path. See for example http://dev.plutext.org/trac/docx4j/browser/trunk/docx4j/src/main/resources/log4j.xml If you don't configure log4j like that, docx4j will auto configure logging at INFO level.

If you are using Eclipse to run things, in the run configuration:

- add VM argument

  -Dlog4j.configuration=log4j.xml

- to the classpath, add a user entry (click "advanced..") for

  src/main/resources

images

If there are any images in the docx, you'll also need:

xmlgraphics-commons-1.4.jar

which in turn requires commons-logging-1.1.1.jar

other dependencies

Depending what you want to do, the other dependencies will be required. The following table explains the other dependencies:

Functionality	Jar	which also requires
HTML export	Xalan	
PDF export	Xalan, FOP	commons-io avalon-framework api & impl
OLE, binary import	POI, commons-codec	
Differencing	commons-lang, stax (for Java 1.5 only)	
	wmf2svg	
Saving/loading via WebDAV etc External graphics	commons-vfs	jdom

As noted above, docx4j dependencies (with the exception of stax) can be found in the .tar.gz version, or in the relevant subdirectory of http://dev.plutext.org/docx4j/

You can also get them via Maven (see next section).

Using docx4j via Maven

Maven POM can be found at
http://dev.plutext.org/trac/docx4j/browser/trunk/docx4j/m2/org/docx4j/docx4j

As from version 2.7.1, docx4j is in Maven Central.

JDK versions

You need to be using Java 1.5+.

This is because of JAXB[1]. If you must use 1.4, retrotranslator can reportedly make it work.

If you are using 1.5 only, and want to do differencing, you will need stax (uncomment it in pom.xml).

A word about Jaxb

docx4j uses JAXB to marshall and unmarshall the key parts in a WordprocessingML document, including the main document part, the styles part, the theme part, and the properties parts.

JAXB is included in Sun's Java 6 distributions, but not 1.5. So if you are using the 1.5 JDK, you will need JAXB 2.1.x on your class path.

Bits of docx4j, such as org.docx4j.wml and org.docx4j.dml were generated using JAXB's XJC. We modified the wml.xsd schema in particular, so that the key resulting classes are a bit more human friendly (ie don't all start with CT_ and ST_).

Javadoc

Javadoc for browsing online or download, can be found in the directory http://dev.plutext.org/docx4j/

Docx4j source code

To obtain a copy of the current source code:

```
svn co http://dev.plutext.org/svn/docx4j/trunk/docx4j docx4j
```

Alternatively, you can browse it online, at:

http://dev.plutext.org/trac/docx4j/browser/trunk/docx4j/

Building docx4j from source

To obtain a copy of the current source code:

```
svn co http://www.docx4java.org/svn/docx4j/trunk/docx4j docx4j
```

Alternatively, you can browse it online, at:

http:// www.docx4java.org/trac/docx4j/browser/trunk/docx4j/

[1] http://forums.java.net/jive/thread.jspa?threadID=411

Building docx4j from source

Command line -via Maven

Create a directory called workspace, and cd into it.

```
svn co http:// www.docx4java.org/svn/docx4j/trunk/docx4j docx4j

export MAVEN_OPTS=-Xmx512m
mvn install
```

Command line - via Ant

Before you can build via ant, you need to obtain docx4j's dependencies. You can get them from the binary distribution, or via maven.

Once you have done that, create a directory called workspace, and cd into it.

```
svn co http:// www.docx4java.org/svn/docx4j/trunk/docx4j docx4j
```

Edit build.xml, so the pathelements point to where you placed the dependencies.

Then

```
ant dist
```

or on Linux

```
ANT_OPTS="-Xmx512m -XX:MaxPermSize=256m" ant dist
```

That ant command will create the docx4j.jar and place it and all its dependencies in the dist dir.

Eclipse

Prerequisites

- Eclipse installed
- Install an Eclipse subversion plugin eg http://subclipse.tigris.org/update_1.2.x
- Install Maven and the Eclipse plugin

And, as discussed above:

- Java 1.5 or 6
- JAXB: **either** the JAXB implementation included in Java 6, **or** the 2.x reference implementation.

Instructions

- File > New "Project .." > SVN > Checkout Projects from SVN
- Create a new repository location; Url is http://dev.plutext.org/svn/docx4j
- Click folder "trunk", and select docx4j; click next
- You want to check it out as a project configured using the New Project Wizard
- Then select Java > Java Project; click Next
- Choose a project name (eg docx4j) then click Next
- Click Finish (we'll define the Java build settings in a later step)

After a couple of other dialog boxes, you should have the new project in your workspace.

Enable Maven (make sure you have Maven and its plugin installed - see Prerequisites above):

- with Eclipse Indigo
    - Right click on the project
    - Click "Configure > Convert to Maven Project"
- with earlier versions of Eclipse
    - Run mvn install in the docx4j dir from a command prompt (just in case)
    - Right click on project > Maven 2 > EnableDependency Management

Set compiler version & system library:

- Right click on the project (or Alt-Enter)
- Choose "Java Compiler", then set JDK compliance to 1.6
- Choose "Java Build Path", and check you are using 1.6 "JRE System Library". If not, remove, then click "Add Library"

Now, we need to check the **class path** etc within Eclipse so that it can build.

- Build Path > Configure Build Path > Java Build Path > Source tab
- Verify it contains:
    - src/diffx
    - src/glox4j
    - src/main/java
    - src/pptx4j/java (remove "Excluded: **" if present!)
    - src/svg
    - src/xslx4j/java
    - src/xslfo

The project should now be working in Eclipse without errors[2].

Open an existing docx/pptx/xlsx document

`org.docx4j.openpackaging.packages.WordprocessingMLPackage` represents a docx document.

To load a document, all you have to do is:

```
WordprocessingMLPackage wordMLPackage =
      WordprocessingMLPackage.load(new java.io.File(inputfilepath));
```

That method can also load "Flat OPC" XML files.

You can then get the main document part (word/document.xml):

```
MainDocumentPart documentPart = wordMLPackage.getMainDocumentPart();
```

After that, you can manipulate its contents.

WordprocessingMLPackage.load uses

```
LoadFromZipNG loader = new LoadFromZipNG();
```

If you need to load a docx from an input stream, you can do something like:

```
WordprocessingMLPackage pkg = (WordprocessingMLPackage)loader.get(stream);
```

A similar approach works for pptx files:

```
PresentationMLPackage presentationMLPackage =
       (PresentationMLPackage)OpcPackage.load(new java.io.File(inputfilepath));
```

And similarly for xlsx files.

WordML concepts

To do anything much beyond this, you need to have an understanding of basic WordML concepts.

According to the Microsoft Open Packaging spec, each docx document is made up of a number of "Part" files, zipped up. A Part is usually XML, but might not be (an image part, for example, isn't).

[2] If you get the error 'Access restriction: The type is not accessible due to restriction on required library rt.jar' (perhaps using some combination of Eclipse 3.4 and/or JDK 6 update 10?), you need to go into the Build Path for the project, Libraries tab, select the JRE System Library, and add an access rule, "Accessible, **".

The parts form a tree. If a part has child parts, it must have a relationships part which identifies these.

The part which contains the main text of the document is the Main Document Part. Each Part has a name. The name of the Main Document Part is usually "/word/document.xml".

If the document has a header, then the main document part woud have a header child part, and this would be described in the main document part's relationships (part).

Similarly for any images. To see the structure of any given document, see "Parts List" further below.

An introduction to WordML is beyond the scope of this document. You can find a very readable introduction in 1st edition Part 3 (Primer) at http://www.ecma-international.org/publications/standards/Ecma-376.htm or http://www.ecma-international.org/news/TC45_current_work/TC45_available_docs.htm (a better link for the 1st edition (Dec 2006), since its not zipped up).

Specification versions

From Wikipedia:

> The Office Open XML file formats were standardised between December 2006 and November 2008,
>
> first by the Ecma International consortium (where they became **ECMA-376**),
>
> and subsequently .. by the ISO/IEC's Joint Technical Committee 1 (where they became **ISO/IEC 29500:2008**).

The Ecma-376.htm link also contains the 2nd edition documents (of Dec 2008), which are "technically aligned with ISO/IEC 29500".

Office 2007 SP2 implements ECMA-376 1st Edition[3]; this is what docx4j implements.

ISO/IEC 29500 (ECMA-376 2nd Edition) has *Strict* and *Transitional* conformance classes. Office 2010 supports[4] transitional, and also has read only support for strict.

Architecture

Docx4j has 3 layers:

1. **`org.docx4j.openpackaging`**

   OpenPackaging handles things at the Open Packaging Conventions level: unzipping a docx into **`WordprocessingMLPackage`** and a set of objects inheriting from Part; allowing parts to be added/deleted; saving the docx

[3] http://blogs.msdn.com/b/dmahugh/archive/2009/01/16/ecma-376-implementation-notes-for-office-2007-sp2.aspx
[4] http://blogs.msdn.com/b/dmahugh/archive/2010/04/06/office-s-support-for-iso-iec-29500-strict.aspx

This layer is based originally on OpenXML4J (which is also used by Apache POI).

2. Parts are generally subclasses of **`org.docx4j.openpackaging.parts.JaxbXmlPart`**

   This (the **jaxb content tree**) is the second level of the three layered model.

   Parts are arranged in a tree. If a part has descendants, it will have a **`org.docx4j.openpackaging.parts.relationships.RelationshipsPart`** which identifies those descendant parts. The sample PartsList (see next section) shows you how this works.

   A JaxbXmlPart has a content tree:

```
public Object getJaxbElement() {
	return jaxbElement;
}

public void setJaxbElement(Object jaxbElement) {
	this.jaxbElement = jaxbElement;
}
```

   Most parts (including MainDocumentPart, styles, headers/footers, comments, endnotes/footnotes) use **`org.docx4j.wml`** (WordprocessingML); wml references **`org.docx4j.dml`** (DrawingML) as necessary.

   These classes were generated from the Open XML schemas

3. **`org.docx4j.model`**

   This package builds on the lower two layers to provide extra functionality, and is being progressively further developed.

Jaxb: marshalling and unmarshalling

Docx4j contains a class representing each part. For example, there is a `MainDocumentPart` class. XML parts inherit from `JaxbXmlPart`, which contains a member called **`jaxbElement`**. When you want to work with the contents of a part, you work with its jaxbElement.

When you open a docx document using docx4j, docx4j automatically ***unmarshals*** the contents of each XML part to a strongly-type Java object tree (the jaxbElement).

Similarly, if/when you tell docx4j to save these Java objects as a docx, docx4j automatically ***marshals*** the jaxbElement in each Part.

Sometimes you will want to marshal or unmarshal things yourself. The class **`org.docx4j.jaxb.Context`** defines all the JAXBContexts used in docx4j:

Jc	org.docx4j.wml org.docx4j.dml org.docx4j.dml.picture org.docx4j.dml.wordprocessingDrawing org.docx4j.vml org.docx4j.vml.officedrawing org.docx4j.math
jcThemePart	org.docx4j.dml
jcDocPropsCore	org.docx4j.docProps.core org.docx4j.docProps.core.dc.elements org.docx4j.docProps.core.dc.terms
jcDocPropsCustom	org.docx4j.docProps.custom
jcDocPropsExtended	org.docx4j.docProps.extended
jcXmlPackage	org.docx4j.xmlPackage
jcRelationships	org.docx4j.relationships
jcCustomXmlProperties	org.docx4j.customXmlProperties
jcContentTypes	org.docx4j.openpackaging.contenttype
jcPML	org.docx4j.pml org.docx4j.dml org.docx4j.dml.picture

Parts List

To get a better understanding of how docx4j works – and the structure of a docx document – you can run the PartsList sample on a docx (or a pptx or xlsx). If you do, it will list the hierarchy of parts used in that package. It will tell you which class is used to represent each part, and where that part is a JaxbXmlPart, it will also tell you what class the `jaxbElement` is.

You can run it from a command line:

```
java -cp docx4j.jar:log4j-1.2.15.jar org.docx4j.samples.PartsList [input.docx]
```

If there are any images in the docx, you'd also need to add to your classpath: xmlgraphics-commons-1.4.jar and commons-logging-1.1.1.jar

For example:

```
Part /_rels/.rels [org.docx4j.openpackaging.parts.relationships.RelationshipsPart]
  containing JaxbElement:org.docx4j.relationships.Relationships

        Part /docProps/app.xml [org.docx4j.openpackaging.parts.DocPropsExtendedPart]
          containing JaxbElement:org.docx4j.docProps.extended.Properties

        Part /docProps/core.xml [org.docx4j.openpackaging.parts.DocPropsCorePart]
          containing JaxbElement:org.docx4j.docProps.core.CoreProperties
```

```
Part /word/document.xml [org.docx4j.openpackaging.parts.WordprocessingML.MainDocumentPart]
  containing JaxbElement:org.docx4j.wml.Document

        Part /word/settings.xml [org.docx4j.openpackaging.parts.WordprocessingML.DocumentSettingsPart]
          containing JaxbElement:org.docx4j.wml.CTSettings

        Part /word/styles.xml [org.docx4j.openpackaging.parts.WordprocessingML.StyleDefinitionsPart]
          containing JaxbElement:org.docx4j.wml.Styles

        Part /word/media/image1.jpeg [org.docx4j.openpackaging.parts.WordprocessingML.ImageJpegPart]
```

docx4j includes convenience methods to make it easy to access commonly used parts. These include,

on the package:

```
public MainDocumentPart getMainDocumentPart()

public DocPropsCorePart getDocPropsCorePart()
public DocPropsExtendedPart getDocPropsExtendedPart()
public DocPropsCustomPart getDocPropsCustomPart()
```

on the document part:

```
public StyleDefinitionsPart getStyleDefinitionsPart()
public NumberingDefinitionsPart getNumberingDefinitionsPart()
public ThemePart getThemePart()
public FontTablePart getFontTablePart()

public CommentsPart getCommentsPart()

public EndnotesPart getEndNotesPart()
public FootnotesPart getFootnotesPart()

public DocumentSettingsPart getDocumentSettingsPart()
public WebSettingsPart getWebSettingsPart()
```

If a part points to any other parts, it will have a relationships part listing these other parts.

```
RelationshipsPart rp = part.getRelationshipsPart();
```

You can access those, and from there, get the part you want:

```
for ( Relationship r : rp.getRelationships().getRelationship() ) {

        log.info("\nFor Relationship Id=" + r.getId()
                        + " Source is " + rp.getSourceP().getPartName()
                        + ", Target is " + r.getTarget()
                        + " type " + r.getType() + "\n");

        Part part = rp.getPart(r);
}
```

That gives access to just the parts this part points to.

There is also a list of **all** parts, in the package object:

```
Parts parts = wordMLPackage.getParts();
```

The Parts object encapsulates a map of parts, keyed by PartName.

To add a part, see the section Adding a Part below.

MainDocumentPart

The text of the document is to be found in the main document part.

Given:

```
WordprocessingMLPackage wordMLPackage
```

you can access:

```
MainDocumentPart documentPart = wordMLPackage.getMainDocumentPart();
```

Classically, you'd then do:

```
org.docx4j.wml.Document wmlDocumentEl
      = (org.docx4j.wml.Document) documentPart.getJaxbElement();
Body body = wmlDocumentEl.getBody();
```

But as from 2.7.0, there is:

```
/**
 * Convenience method to getJaxbElement().getBody().getContent()
 * @since 2.7
 */
public List<Object> getContent()
```

A paragraph is org.docx4j.wml.P; a paragraph is basically made up of runs of text.

```
@XmlRootElement(name = "p")
public class P implements Child, ContentAccessor
```

The `ContentAccessor` interface is simply:

```
/**
 * @since 2.7
 */
public interface ContentAccessor {

    public List<Object> getContent();

}
```

it is implemented by a number of objects, including:

- P, R (R is for run, which is where the document text lives)
- Hdr, Ftr

- table related objects (Tbl, Tc, Tr)
- content control objects

Read on for how to add text etc.

Samples

The package org.docx4j.samples contains examples of how to do things with docx4j. There are pptx and xlsx samples in packages org.pptx4j.samples and org.xlsx4j.samples respectively.

The docx4j samples include:

Basics

- CreateWordprocessingMLDocument
- DisplayMainDocumentPartXml
- OpenAndSaveRoundTripTest
- OpenMainDocumentAndTraverse
- XPathQuery

Output/Transformation

- CreateHtml
- CreatePdf

Flat OPC XML

- ExportInPackageFormat
- ImportFromPackageFormat

Image handling

- AddImage
- ConvertEmbeddedImageToLinked

Part Handling

- CopyPart
- ImportForeignPart
- PartsList
- StripParts

Document generation/document assembly using content controls

- AltChunk

- CreateDocxWithCustomXml
- CustomXmlBinding
- ContentControlBindingExtensions

Miscellaneous

- CompareDocuments
- DocProps
- Filter
- HyperlinkTest
- NumberingRestart
- UnmarshallFromTemplate

If you installed the source code, you'll have this package already.

If you didn't, you can browse it online, at

http://dev.plutext.org/trac/docx4j/browser/trunk/docx4j/src/main/java/org/docx4j/samples

There are also various **sample documents** in the /sample-docs directory; these are most easily accessed by checking out docx4j svn.

Creating a new docx

To create a new docx:

```
// Create the package
WordprocessingMLPackage wordMLPackage = WordprocessingMLPackage.createPackage();

// Save it
wordMLPackage.save(new java.io.File("helloworld.docx") );
```

That's it.

`createPackage()` is a convenience method, which does:

```
// Create the package
WordprocessingMLPackage wordMLPackage = new WordprocessingMLPackage();

// Create the main document part (word/document.xml)
MainDocumentPart wordDocumentPart = new MainDocumentPart();

// Create main document part content
ObjectFactory factory = Context.getWmlObjectFactory();
org.docx4j.wml.Body body = factory .createBody();
org.docx4j.wml.Document wmlDocumentEl = factory .createDocument();
```

```
wmlDocumentEl.setBody(body);

// Put the content in the part
wordDocumentPart.setJaxbElement(wmlDocumentEl);

// Add the main document part to the package relationships
// (creating it if necessary)
wmlPack.addTargetPart(wordDocumentPart);
```

docx4j.properties

Here is a sample docx4j.properties file:

```
# Page size: use a value from org.docx4j.model.structure.PageSizePaper enum
# eg A4, LETTER
docx4j.PageSize=LETTER
# Page size: use a value from org.docx4j.model.structure.MarginsWellKnown enum
docx4j.PageMargins=NORMAL
docx4j.PageOrientationLandscape=false

# Page size: use a value from org.pptx4j.model.SlideSizesWellKnown enum
# eg A4, LETTER
pptx4j.PageSize=LETTER
pptx4j.PageOrientationLandscape=false

# These will be injected into docProps/app.xml
# if App.Write=true
docx4j.App.write=true
docx4j.Application=docx4j
docx4j.AppVersion=2.7
# of the form XX.YYYY where X and Y represent numerical values

# These will be injected into docProps/core.xml
docx4j.dc.write=true
docx4j.dc.creator.value=docx4j
docx4j.dc.lastModifiedBy.value=docx4j

#
#docx4j.McPreprocessor=true

# If you haven't configured log4j yourself
# docx4j will autoconfigure it.  Set this to true to disable that
docx4j.Log4j.Configurator.disabled=false
```

The page size, margin & orientation values are used when new documents are created; naturally they don't affect an existing document you open with docx4j.

If no docx4j.properties file is found on your class path, docx4j has hard coded defaults.

Adding a paragraph of text

`MainDocumentPart` contains a method:

```
public org.docx4j.wml.P addStyledParagraphOfText(String styleId, String text)
```

You can use that method to add a paragraph using the specified style.

The XML we are looking to create will be something like:

```
<w:p  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
    <w:r>
        <w:t>Hello world</w:t>
    </w:r>
</w:p>
```

`addStyledParagraphOfText` builds the object structure "the JAXB way", and adds it to the document.

```
    ObjectFactory factory = Context.getWmlObjectFactory();

    // Create the paragraph
    org.docx4j.wml.P  para = factory.createP();

    // Create the text element
    org.docx4j.wml.Text  t = factory.createText();
    t.setValue(simpleText);

    // Create the run
    org.docx4j.wml.R  run = factory.createR();
    run.getRunContent().add(t);

    para.getParagraphContent().add(run);

    // Now add our paragraph to the document body
    Body body = this.jaxbElement.getBody();
    Body.getEGBlockLevelElts().add(para)
```

Notice that adding a paragraph involves:

```
    Body.getEGBlockLevelElts().add(para)
```

Similarly, the paragraph object P has:

```
    public List<Object> getParagraphContent()
```

and the run object:

```
    public List<Object> getRunContent()
```

Alternatively, you can create the paragraph by marshalling XML:

```
    // Assuming String xml contains the XML above
    org.docx4j.wml.P  para = XmlUtils.unmarshalString(xml);
```

For this to work, you need to ensure that all namespaces are declared properly in the string.

See further below for adding images, and tables.

General strategy/approach for creating stuff

The first thing you need to know is what the XML you are trying to create looks like.

To figure this out, start with a docx that contains the construct (create it in Word if necessary).

Now look at its XML. Choices:

- You can unzip it to do this
- easiest may be to save it as Flat OPC XML from Word (or use the `ExportInPackageFormat` sample), so you have just a single XML file which you don't need to unzip
- you can use the `DisplayMainDocumentPartXml` to get it
- you can open it with docx4all, and look at the source view
- on Windows, if you have Visual Studio 2010, you can drag the docx onto it
- on Windows, get PackageExplorer from codeplex.

Now you are ready to create this XML using JAXB. There are 2 basic ways.

The classic JAXB way is to use the ObjectFactory's .createX methods. For example:

```
ObjectFactory factory = Context.getWmlObjectFactory();
P p = factory.createP();
```

The challenge with this is to know what object it is you are trying to create. To find this out, run `OpenMainDocumentAndTraverse` on your document.

Here are the names for some common objects:

Object	XML element	docx4j class	Factory method
Document body	w:body	org.docx4j.wml.Body	factory.createBody();
Paragraph	w:p	org.docx4j.wml.P	factory.createP()
Paragraph props	w:pPr	org.docx4j.wml.PPr	factory.createPPr()
Run	w:r	org.docx4j.wml.R	factory.createR()
Run props	w:rPr	org.docx4j.wml.RPr	factory.createRPr()
Text	w:t	org.docx4j.wml.Text	factory.createText()
Table	w:tbl	org.docx4j.wml.Tbl	factory.createTbl()
Table row	w:tr	org.docx4j.wml.Tr	factory.createTr()
Table cell	w:tc	org.docx4j.wml.Tc	factory.createTc()
Drawing	w:drawing	org.docx4j.wml.Drawing	factory.createDrawing()
Page break	w:br	org.docx4j.wml.Br	factory.createBr()
Footnote or endnote ref	?	org.docx4j.wml.CTFtnEdnRef	factory.createCTFtnEdnRef()

An easier way may be to just unmarshal the XML (eg a String representing a paragraph to be inserted into the document).

For example, given:

```
<w:p  xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
    <w:r>
        <w:t>Hello world</w:t>
    </w:r>
</w:p>
```

you can simply:

```
    // Assuming String xml contains the XML above
    org.docx4j.wml.P  para = XmlUtils.unmarshalString(xml);
```

Problems? See ***javax.***xml.bind.JAXBElement

One annoying thing about JAXB, is that an object – say a table – could be represented as `org.docx4j.wml.Tbl` (as you would expect). Or it might be wrapped in a **`javax.xml.bind.JAXBElement`**, in which case to get the real table, you have to do something like:

```
if ( ((JAXBElement)o).getDeclaredType().getName().equals("org.docx4j.wml.Tbl") )
     org.docx4j.wml.Tbl tbl = (org.docx4j.wml.Tbl)((JAXBElement)o).getValue();
```

XmlUtils.**unwrap** can do this for you.

Be careful, though. If you are intend to copy an unwrapped object into your document (rather than just read it), you'll probably want the object to remain wrapped (JAXB usually wraps them for a reason; without the wrapper, you might find you need an @XmlRootElement annotation in order to be able to marshall ie save your document).

@XmlRootElement below.

If you need to be explicit about the type, you can use:

```
public static Object unmarshalString(String str, JAXBContext jc, Class declaredType)
```

The ContentAccessor interface

docx4j 2.7.0 introduced a content accessor interface.

This interface contains a single method:

```
public List<Object> getContent();
```

It is implemented for a number of objects, including the following:

Body	w:body	document body
P	w:p	paragraph
R	w:r	run

Tbl Tr Tc	w:tbl w:tr w:tc	table table row table cell
SdtBlock SdtRun CTSdtRow CTSdtCell	w:sdt w:sdt w:sdt w:sdt	content controls; see the method `getSdtContent()`

In earlier versions of docx4j, each of these classes had a different name for the method which returns the list of content objects:

Body	`getEGBlockLevelElts()`
P	`getParagraphContent()`
R	`getRunContent()`

Tbl
Tr
Tc
	`getEGContentRowContent()`
	`getEGContentCellContent()`
	`getEGBlockLevelElts()`

SdtBlock
SdtRun
CTSdtRow
CTSdtCell
	`getSdtContent().getEGContentBlockContent()`
	`getSdtContent().getParagraphContent()`
	`getSdtContent().getEGContentRowContent()`
	`getSdtContent().getEGContentCellContent()`

Creating and adding a table

org.docx4j.model.table.TblFactory provides an easy way to create a simple table. For an example of its use, see the CreateWordprocessingMLDocument sample.

If you want to know what you need to add to format your table (make it prettier), see ***General strategy/approach for creating stuff*** above.

Traversing a document

OpenMainDocumentAndTraverse.java in the samples directory shows you how to traverse the JAXB representation of a docx.

You can run it from a command line:

```
java -cp docx4j.jar:log4j-1.2.15.jar org.docx4j.samples.OpenMainDocumentAndTraverse [input.docx]
```

If there are any images in the docx, you'd also need to add to your classpath: xmlgraphics-commons-1.4.jar and commons-logging-1.1.1.jar

This sample is useful if you want to see what objects are used in your document.xml.

This is an alternative to XSLT, which doesn't require marshalling/unmarshalling.

The sample uses TraversalUtil, which is a general approach for traversing the JAXB object tree in the main document part. It can also be applied to headers, footers etc. TraversalUtil has an `interface Callback`, which you use to specify how you want to traverse the nodes, and what you want to do to them.

As noted in "docx4j.properties

Here is a sample docx4j.properties file:

```
# Page size: use a value from org.docx4j.model.structure.PageSizePaper enum
# eg A4, LETTER
docx4j.PageSize=LETTER
# Page size: use a value from org.docx4j.model.structure.MarginsWellKnown enum
docx4j.PageMargins=NORMAL
docx4j.PageOrientationLandscape=false

# Page size: use a value from org.pptx4j.model.SlideSizesWellKnown enum
# eg A4, LETTER
pptx4j.PageSize=LETTER
pptx4j.PageOrientationLandscape=false

# These will be injected into docProps/app.xml
# if App.Write=true
docx4j.App.write=true
docx4j.Application=docx4j
docx4j.AppVersion=2.7
# of the form XX.YYYY where X and Y represent numerical values

# These will be injected into docProps/core.xml
docx4j.dc.write=true
docx4j.dc.creator.value=docx4j
docx4j.dc.lastModifiedBy.value=docx4j

#
#docx4j.McPreprocessor=true

# If you haven't configured log4j yourself
# docx4j will autoconfigure it.  Set this to true to disable that
docx4j.Log4j.Configurator.disabled=false
```

The page size, margin & orientation values are used when new documents are created; naturally they don't affect an existing document you open with docx4j.

If no docx4j.properties file is found on your class path, docx4j has hard coded defaults.

Adding a paragraph of text" above, many objects (eg the document body, a paragraph, a run), have a List containing their content (see The ContentAccessor interface further below). Traversal works by iterating over these lists.

Selecting your insertion/editing point; accessing JAXB nodes via XPath

Sometimes, XPath is a succinct way to select the things you need to change.

Happily, from docx4j 2.**5**.0, you can do use XPath to select JAXB nodes:

```
MainDocumentPart documentPart = wordMLPackage.getMainDocumentPart();
String xpath = "//w:p";
List<Object> list = documentPart.getJAXBNodesViaXPath(xpath, false);
```

These JAXB nodes are live, in the sense that if you change them, your document changes.

There is a limitation however: the xpath expressions are evaluated against the XML document as it was when first opened in docx4j. You can update the associated XML document once only, by passing true into `getJAXBNodesViaXPath`. Updating it again (with current JAXB 2.1.x or 2.2.x) will cause an error.

Adding a Part

What if you wanted to add a new styles part? Here's how:

```
// Create a styles part
StyleDefinitionsPart stylesPart = new StyleDefinitionsPart();

// Populate it with default styles
stylesPart.unmarshalDefaultStyles();

// Add the styles part to the main document part relationships
wordDocumentPart.addTargetPart(stylesPart);
```

You'd take the same approach to add a header or footer.

When you add a part this way, it is automatically added to the source part's relationships part.

Generally, you'll also need to add a reference to the part (using its relationship id) to the Main Document Part. This applies to images, headers and footers. (Comments, footnotes and endnotes are a bit different, in that what you add to the main document part are references to individual comments/footnotes/endnotes).

docx to (X)HTML

docx4j uses XSLT to transform a docx to XHTML:

```
AbstractHtmlExporter exporter = new HtmlExporterNG2();
  // note the *2* there
```

```
// Write to StreamResult (in this case, an output stream)
OutputStream os = new java.io.FileOutputStream(inputfilepath + ".html");

javax.xml.transform.stream.StreamResult result
     = new javax.xml.transform.stream.StreamResult(os);

exporter.html(wordMLPackage, result,
       inputfilepath + "_files");
```

You will find the generated HTML is clean.

Docx4j uses Java XSLT extension functions to do the heavy lifting, so the XSLT itself is kept simple.

If you have log4j debug level logging enabled for **`org.docx4j.convert.out.html.HtmlExporterNG2`**, anything which is not implemented will be obvious in the output document. ***If debug level logging is not switched on, unsupported elements will be silently dropped.***

The XSLT can be found at src/main/java/org/docx4j/convert/out/html/docx2xhtmlNG2.xslt

There are several ways to customise the HTML output.

- one of course is to alter the xslt itself. This should be avoided, unless your objective is to improve the fidelity of the output (in which case, please contribute a patch!)

  To substitute your own XSLT, you can use the `HtmlExporterNG2` method:

  ```
  public static void setXslt(Templates xslt)
  ```

- another possibility (currently in svn trunk only) is to register an **SdtTagHandler**.

  An SdtTagHandler allows you to wrap SdtContent (Content Control content) in arbitrary HTML (for example, a with a particular class attribute, or style attribute, or associated javascript).

  The design envisages different tag handlers being applied depending on the value of w:sdt/w:sdtPr/w:tag (hence the name tag handler). The content of a tag should be name/value pairs delimited like a URL query string.

  For further details, please see the SdtWriter class.

docx to PDF

docx4j produces XSL FO, which can in turn be used to create a PDF.

At present, Apache FOP is integrated into docx4j for creating the PDF. (Soon, we will be changing things so that docx4j generates FO, for use by your preferred FO renderer, whether that's FOP, or a commercial tool such as XEP).

To create a PDF:

```
// Fonts identity mapping – best on Microsoft Windows
wordMLPackage.setFontMapper(new IdentityPlusMapper());

// Set up converter
org.docx4j.convert.out.pdf.PdfConversion c
    = new org.docx4j.convert.out.pdf.viaXSLFO.Conversion(wordMLPackage);

// Write to output stream
OutputStream os = new java.io.FileOutputStream(inputfilepath + ".pdf");
c.output(os);
```

See the CreatePdf sample.

If you have log4j debug level logging enabled for **`org.docx4j.convert.out.pdf.viaXSLFO`**, anything which is not implemented will be obvious in the output document. In addition, the logs will contain the intermediate XSL FO for inspection. ***If debug level logging is not switched on, unsupported elements will be silently dropped.***

The XSLT can be found at src/main/java/org/docx4j/convert/out/pdf/viaXSLFO/docx2fo.xslt

Fonts

When docx4j is used to create a PDF, it can only use fonts which are available to it.

These fonts come from 2 sources:

- those installed on the computer
- those embedded in the document

Note that Word silently performs ***font substitution***. When you open an existing document in Word, and select text in a particular font, the actual font you see on the screen won't be the font reported in the ribbon if it is not installed on your computer or embedded in the document. To see whether Word 2007 is substituting a font, go into Word Options > Advanced > Show Document Content and press the "Font Substitution" button.

Word's font substitution information is not available to docx4j. As a developer, you 3 options:

- ensure the font is installed or embedded
- tell docx4j which font to use instead, or
- allow docx4j to fallback to a default font

To embed a font in a document, open it in Word on a computer which has the font installed (check no substitution is occuring), and go to Word Options > Save > Embed Fonts in File.

If you want to tell docx4j to use a different font, you need to add a font mapping. The FontMapper interface is used to do this.

On a Windows computer, font names for installed fonts are mapped 1:1 to the corresponding physical fonts via the IdentityPlusMapper.

A font mapper contains Map<String, PhysicalFont>; to add a font mapping, as per the example in the CreatePdf sample:

```
// Set up font mapper
Mapper fontMapper = new IdentityPlusMapper();
wordMLPackage.setFontMapper(fontMapper);

// Example of mapping missing font Algerian to installed font Comic Sans MS
PhysicalFont font = PhysicalFonts.getPhysicalFonts().get("Comic Sans MS");
fontMapper.getFontMappings().put("Algerian", font);
```

You'll see the font names if you configure log4j debug level logging for **`org.docx4j.fonts.PhysicalFonts`**

Image Handling

When you add an image to a document in Word 2007, it is generally added as a new Part (ie you'll find a part in the resulting docx, containing the image in base 64 format).

When you open the document in docx4j, docx4j will create an image part representing it.

It is also possible to create a "linked" image. In this case, the image is not embedded in the docx package, but rather, is referenced at its external location.

Docx4j's **`BinaryPartAbstractImage`** class contains methods to allow you to create both embedded and linked images (along with appropriate relationships).

```
/**
 * Create an image part from the provided byte array, attach it to the
 * main document part, and return it.*/
public static BinaryPartAbstractImage createImagePart(WordprocessingMLPackage wordMLPackage,
    byte[] bytes)

/**
 * Create an image part from the provided byte array, attach it to the source part
 * (eg the main document part, a header part etc), and return it.*/
public static BinaryPartAbstractImage createImagePart(WordprocessingMLPackage wordMLPackage,
    Part sourcePart, byte[] bytes)

/**
 * Create a linked image part, and attach it as a rel of the specified source part
 * (eg a header part) */
public static BinaryPartAbstractImage createLinkedImagePart(
    WordprocessingMLPackage wordMLPackage, Part sourcePart, String fileurl)
```

For an image to appear in the document, there also needs to be appropriate XML in the main document part. This XML can take 2 basic forms:

- the Word 2007 **w:drawing** form

```
<w:p>
	<w:r>
		<w:drawing>
			<wp:inline distT="0" distB="0" distL="0" distR="0">
				<wp:extent cx="3238500" cy="2362200" />
				<wp:effectExtent l="19050" t="0" r="0" b="0" />
				:
				<a:graphic >
					<a:graphicData  ..>
						<pic:pic >
							:
							<pic:blipFill>
								<a:blip r:embed="rId5" />
								:
							</pic:blipFill>
							:
						</pic:pic>
					</a:graphicData>
				</a:graphic>
			</wp:inline>
		</w:drawing>
	</w:r>
</w:p>
```

- the Word 2003 VML-based **w:pict** form

```
<w:p>
	<w:r>
		<w:pict>
			<v:shapetype id="_x0000_t75" coordsize="21600,21600"  .. >
				<v:stroke joinstyle="miter" />
				<v:formulas>
					:
				</v:formulas>
				:
			</v:shapetype>
			<v:shape .. style="width:428.25pt;height:321pt">
				<v:imagedata r:id="rId4" o:title="" />
			</v:shape>
		</w:pict>
	</w:r>
</w:p>
```

Docx4j can create the Word 2007 **w:drawing/wp:inline** form for you:

```
/**
 * Create a <wp:inline> element suitable for this image,
 * which can be linked or embedded in w:p/w:r/w:drawing.
 * If the image is wider than the page, it will be scaled
 * automatically.  See Javadoc for other signatures.
 * @param filenameHint Any text, for example the original filename
 * @param altText  Like HTML's alt text
```

```
   * @param id1   An id unique in the document
   * @param id2   Another id unique in the document
   * @param link  true if this is to be linked not embedded */
  public Inline createImageInline(String filenameHint, String altText,
      int id1, int id2, boolean link)
```

which you can then add to a **w:r/w:drawing.**

Finally, with docx4j, you can convert images from formats unsupported by Word (eg PDF), to PNG, which is a supported format. For this, docx4j uses **ImageMagick**. So if you want to use this feature, you need to install ImageMagick. Docx4j invokes ImageMagick using:

```
Process p = Runtime.getRuntime().exec("imconvert -density " + density + " -units PixelsPerInch - png:-");
```

Note the name **imconvert**, which is used so that we don't have to supply a full path to exec. You'll need to accommodate that.

Manual Image Manipulation

Images involve three things:

- the image part itself
- a relationship, in the relationships part of the main document part (or header part etc). This relationship includes:
  - the name of the image part (for example, `/word/media/image1.jpeg`)
  - the relationship ID
- some XML in the main document part (or header part etc), referencing the relationship ID (see **w:drawing** and **w:pict** examples above)

This means that if you are moving images around, you need to take care to ensure that the relationships remain valid.

You can manually manipulate the relationship, and you can manually manipulate the XML referencing the relationship IDs.

Given an image part, you can get the relationship pointing to it

```
Relationship rel = copiedImagePart.getSourceRelationship();
String id = rel.getId();
```

You can then ensure the reference matches.

Adding Headers/Footers

See the HeaderFooter sample for how to do this.

Merging Documents

As Eric White's blog explained, combining multiple documents can be complicated:

> This programming task is complicated by the need to keep other parts of the document in sync with the data stored in paragraphs. For example, a paragraph can contain a reference to a comment in the comments part, and if there is a problem with this reference, the document is invalid. You must take care when moving / inserting / deleting paragraphs to maintain '***referential integrity***' within the document.

There is a paid extension for docx4j, called MergeDocx, which makes merging documents as easy as invoking the method:

```
public  WordprocessingMLPackage merge(List<WordprocessingMLPackage> wmlPkgs)
```

In other words, you pass a list of docx, and get a single new docx back.

The extension can also be used to process a **docx** which is embedded as an **altChunk**. (Without the extension, you have to rely on Word to convert the altChunk to normal content, which means if your docx contains w:altChunk, you have to round trip it through Word, before docx4j can create a PDF or HTML out of it.)

To process the w:altChunk elements in a docx, you invoke:

```
public WordprocessingMLPackage process(WordprocessingMLPackage srcPackage)
```

You pass in a docx containg altChunks, and get a new docx back which doesn't.

Table of Contents

The minimal XML docx4j needs to insert into the document for **Microsoft Word** to then generate a TOC (including hyperlinks and associated bookmarks), is:

```
<w:p>
  <w:r>
    <w:fldChar w:fldCharType="begin" w:dirty="true"/>
  </w:r>
  <w:r>
    <w:instrText xml:space="preserve"> TOC \o "1-3" \h \z \u </w:instrText>
  </w:r>
  <w:r>
    <w:fldChar w:fldCharType="end"/>
```

```
            </w:r>
          </w:p>
```

Note the **w:dirty="true"**. The actual field code in instrText could be altered to meet your requirements.

Note that simply including this is currently not enough for you to get a table of contents in your HTML or PDF output. Currently, you'd need to open/save in Word, and the HTML/PDF output would need to support the result (page numbering will be a problem).

Text extraction

A quick way to extract the text from a docx, is to use `TextUtils`'

```
public static void extractText(Object o, Writer w)
```

which marshals the object it is passed via a SAX ContentHandler, in order to output the text to the Writer.

Text substitution

Text substitution is easy enough, provided the string you are searching for is represented in a **`org.docx4j.wml.Text`** object in the form you expect.

However, that won't necessarily be the case. The string could be broken across text runs for any of the following reasons:

- part of the word is formatted differently (eg in bold)
- spelling/grammar
- editing order (rsid)

This is one reason that using data bound content controls is often a better approach (see next section).

Subject to that, you can do text substitution in a variety of ways, for example:

- traversing the main document part, and looking at the **`org.docx4j.wml.Text`** objects
- marshal to a string, search/replace in that, then unmarshall again

docx4j's XmlUtils also contains:

```
    /**
     * Give a string of wml containing ${key1}, ${key2}, return a suitable
     * object.*/
    public static Object unmarshallFromTemplate(String wmlTemplateString,
```

```
java.util.HashMap<String, String> mappings)
```

See the UnmarshallFromTemplate example, which operates on a string containing:

```
<w:p>
	<w:r>
		<w:t>My favourite colour is ${colour}.</w:t>
	</w:r>
</w:p>
<w:p />
<w:p>
	<w:r>
		<w:t>My favourite ice cream is ${icecream}.</w:t>
	</w:r>
</w:p>
```

Text substitution via data bound content controls

If you have an XML file containing your own data, WordML has a mechansim for associating entries in that XML with content controls in the document.

Then, when you open the document in Word 2007, Word automatically populates the content controls with the relevant XML data. (This approach supersedes Word's legacy mail merge fields. Simple VBA for migrating a document is available at http://blogs.msdn.com/b/microsoft_office_word/archive/2007/03/28/migrating-mail-merge-fields-to-content-controls.aspx)

This works using XPath. A data-bound content control looks something like:

```
<w:sdt>
  <w:sdtPr>
    <w:dataBinding w:xpath="/root[1]/customer[1]" w:storeItemID="{428C88D8-C0E3-44F0-B5D7-F65D8B9F7EC9}" />
  </w:sdtPr>
  <w:sdtContent>
    <w:r>
      <w:rPr>
        <w:rStyle w:val="PlaceholderText" />
      </w:rPr>
      <w:t>Click here to enter text.</w:t>
    </w:r>
  </w:sdtContent>
</w:sdt>
```

You XML file is stored as a part in the docx, typically with a path which is something like customXml/item1.xml. Note: despite the word "customXml" in the path, this functionality is not affected by the 2009 i4i patent saga.

If you have a Word document which contains data-bound content controls and your data, docx4j can fetch the data, and place it in the relevant content controls.

This is useful if you don't want to leave it to Word to do that (for example, you are creating PDFs with docx4j).

Your XML is represented using 2 parts:

```
CustomXmlDataStoragePart customXmlDataStoragePart
        = wordMLPackage.getCustomXmlDataStorageParts().get(itemId);

CustomXmlDataStorage customXmlDataStorage
        = customXmlDataStoragePart.getData();
```

To apply the bindings:

```
customXmlDataStoragePart.applyBindings(wordMLPackage.getMainDocumentPart());
```

See further the CustomXmlBinding sample.

If you want to create the same document 5 times, each populated with different data, obviously you'd need to insert new XML data first.

Binding extensions for repeats and conditionals

A content control is *conditional* if it (and its contents) are included/excluded from the document based on whether some condition is true or false.

A content control is a *repeat* if it designates that its contents are to be included more than once. For example, a row of a table for each invoice/order item, or person.

docx4j (from 2.5.0) contains a mechanism for processing conditional content controls and repeats. See http://www.opendope.org/opendope_conventions_v2.3.html for an explanation.

docx4j v2.5.0 implemented v1 of the conventions. docx4j v2.6.0 implements v2 of the conventions. You are advised to use a v2 implementation. See forum post for further details.

To set up the bindings, you can use the Word Add-In from http://www.opendope.org/implementations.html Please note that you will need to install .NET Framework 4.0 ("full" - the "client profile" is not enough).

See also the docx4j sample ContentControlBindingExtensions.

SmartArt

docx4j supports reading docx and pptx files which contain SmartArt.

From docx4j 2.7.0, you can also generate SmartArt.

To do this, you need:

- the layout definition for the SmartArt, either in the docx already, or from a glox file
- an XML file specifying the list of text items you want to render graphically
- an XSLT which can convert a transformed version of that XML file into a SmartArt data file.

Docx4j can be used to insert the SmartArt parts into a docx; Word or Powerpoint will then render it when the document is opened.

The code can be found in:

- org.opendope.SmartArt.dataHierarchy
- org.docx4j.openpackaging.parts.DrawingML, and
- src/glox4j/java

Work with the "Flat OPC XML Format"

When you want to look inside a docx document, it's a bit of a pain to have to unzip it to look at the relevant part.

There are 2 ways around this.

One is http://www.codeplex.com/PackageExplorer, which can unzip the docx, and pretty print a part.

Another is to save the docx as "Word XML document (*.xml)". This produces a single XML file, which you can open in an XML editor such as XPontus.

Docx4j can open Flat OPC XML files, and save to them.

To open a Flat OPC XML file:

```
WordprocessingMLPackage wordMLPackage =
      WordprocessingMLPackage.load(new java.io.File(inputfilepath));
```

To save as Flat OPC XML:

```
wordMLPackage.save(new java.io.File(outputfilepath));
```

In both cases, the Flat OPC code will be used if and only if the file extension is ".xml".

Converting to/from Flat OPC can be done at the command line, with:

```
java -cp docx4j.jar:log4j-1.2.15.jar org.docx4j.samples. ExportInPackageFormat [input.docx]
```

(and similar for ImportFromPackageFormat).

If there are any images in the docx, you'd also need to add to your classpath: xmlgraphics-commons-1.4.jar and commons-logging-1.1.1.jar

JAXB stuff

Cloning

To clone a JAXB object, use one of the following methods in XmlUtils:

```
/** Clone this JAXB object, using default JAXBContext. */
public static <T> T deepCopy(T value)

/** Clone this JAXB object */
public static <T> T deepCopy(T value, JAXBContext jc)
```

javax.xml.bind.JAXBElement

One annoying thing about JAXB, is that an object – say a table – could be represented as `org.docx4j.wml.Tbl` (as you would expect). Or it might be wrapped in a **`javax.xml.bind.JAXBElement`**, in which case to get the real table, you have to do something like:

```
if ( ((JAXBElement)o).getDeclaredType().getName().equals("org.docx4j.wml.Tbl") )
     org.docx4j.wml.Tbl tbl = (org.docx4j.wml.Tbl)((JAXBElement)o).getValue();
```

XmlUtils.**unwrap** can do this for you.

Be careful, though. If you are intend to copy an unwrapped object into your document (rather than just read it), you'll probably want the object to remain wrapped (JAXB usually wraps them for a reason; without the wrapper, you might find you need an @XmlRootElement annotation in order to be able to marshall ie save your document).

@XmlRootElement

Most commonly used objects have an **`@XmlRootElement`** annotation, so they can be marshalled and unmarshalled.

In some cases, you might find this annotation is missing.

If you can't add the annotation to the jaxb source code, an alternative is to marshall it using code which is explicit about the resulting QName. For example, XmlUtils contains:

```
/** Marshal to a W3C document, for object
 *  missing an @XmlRootElement annotation.  */
public static org.w3c.dom.Document marshaltoW3CDomDocument(Object o, JAXBContext jc,
    String uri, String local, Class declaredType)
```

You could use this like so:

```
CTFootnotes footnotes =
    wmlPackage.getMainDocumentPart().getFootnotesPart().getJaxbElement().getValue();
CTFtnEdn ftn = footnotes.getFootnote().get(1);

// No @XmlRootElement on CTFtnEdn, so ..
Document d = XmlUtils.marshaltoW3CDomDocument( ftn,
    Context.jc, Namespaces.NS_WORD12, "footnote",  CTFtnEdn.class );
```

Where the problematic object is something you're adding which isn't at the top of the tree, you should add it wrapped in a JAXBElement. For example, suppose you wanted to add FldChar fldchar. You'd create it in the ordinary way:

```
FldChar fldchar = factory.createFldChar();
```

but then what you'd actually add to r.getRunContent() is:

```
new JAXBElement( new QName(Namespaces.NS_WORD12, "fldChar"), FldChar.class, fldchar);
```

An easier way to do this is to find the appropriate method in the object factory (ie the method for creating it wrapped as a JAXBElement). Use that method signature. In this example:

```
    @XmlElementDecl(namespace = "http://schemas.openxmlformats.org/wordprocessingml/2006/main", name =
"fldChar", scope = R.class)
    public JAXBElement<FldChar> createRFldChar(FldChar value) {
        return new JAXBElement<FldChar>(_RFldChar_QNAME, FldChar.class, R.class, value);
    }
```

docx4j-extras

src/docx4j-extras contains functionality which is not part of the standard docx4j build:

- load/save via JCR
- PDF conversion via HTML or iText

The docx4j forum

Free community support is available in the docx4j forum, at http://dev.plutext.org/forums/

Before posting, please use http://dev.plutext.org/search.html to check for relevant material.

This discussion is generally in English. If you can volunteer to moderate a forum in another language (for example, French, Chinese, Spanish…), please let us know.

Other Support Options

If the free community support available in the docx4j forum does not meet your needs, or you simply want to outsource some coding, you are welcome to purchase programming, consulting and priority support from http://www.plutext.com/m/index.php/docx4j-support.html

By purchasing services from Plutext, you support the continued development of docx4j.

Colophon

This document was written in Word 2007, using:

- XML pretty printed using http://www.softlion.com/webTools/XmlPrettyPrint/default.aspx or Package Explorer
- Java source code formatted using http://www.java2html.de (or cut/pasted from Eclipse)

The PDF and HTML versions were generated using docx4j (PDF via XSL FO and FOP).

Contacting Plutext

Unless you have paid for support, general "How do I" type questions should be posted directly to the docx4j forum. Plutext may post to the forum any questions it receives by email which should have been directed to the forum.

Plutext can be contacted at either jason@plutext.org, or jharrop@plutext.com