# Docx4j - Getting Started

This guide is for docx4j **3.0.0**.

The latest version of this document can always be found in docx4j on GitHub in /docs.

The most up to date copy of this document is in English. There is also a Russian version. From time to time, it may be machine translated into other languages. Please let us know if you are interested in writing some basic documentation in your own language (either as a contribution, or for a fee).

## What is docx4j?

docx4j is a library for working with docx, pptx and xlsx files in Java. In essence, it can unzip a docx (or pptx/xlsx) "package", and parse the XML to create an in-memory representation in Java using developer friendly classes (as opposed to DOM or SAX).

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

A strength of docx4j is that its in-memory representation uses **JAXB**, the JCP standard for Java - XML binding. In this respect, Aspose is similar to it. In contrast, Apache POI uses XML Beans.

docx4j is open source, available under the Apache License (v2). As an open source project, docx4j has been substantially improved by a number of contributions (see the README or POM file for contributors), and further contributions are always welcome. Please see the docx4j forum at http://www.docx4java.org/forums/ for details.

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/pptx/xslsx on the .NET platform, you should consider Microsoft's OpenXML SDK. That said, docx4j can be used in a .NET environment via IKVM, and there are several reasons you might wish to do this:

- Where you need docx4j's capabilities, for example:

    - XHTML import
    - PDF or XHTML export
    - OpenDoPE processing
- Capabilities provided by docx4j commercial extensions, for example:
    - Merging documents or presentations
    - OLE embedding
    - TOC generation/updating
- Where you need to work in both Java and .NET, and want to use a single API in both environments
- Where you need the source code (Microsoft doesn't provide that)

An alternative to docx4j is Apache POI, which is likely to be an especially good fit 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)
- Save to various media zipped, or unzipped
- Do all this on Android (v3 or 4).

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

- Import XHTML
- Export as (X)HTML or PDF
- Template substitution; CustomXML binding
- Mail merge
- Produce/consume Word 2007's xmlPackage (pkg) format
- Apply transforms, including common filters
- 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.

If there is some feature you'd like to see added, http://docx4j.userecho.com/ is the place to raise it.

# What Word documents does it support?

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

The relevant parts of docx4j are generated from the ECMA schemas, with the addition of the key Microsoft proprietary extensions. For unsupported extensions, docx4j gracefully degrades to the specified 2007 substitutes.

It is not really intended read/write Word 2003 XML documents, although `package org.docx4j.convert.in.word2003xml` is a proof of concept of importing such documents.

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

# Handling legacy binary .doc files

Apache POI's HWPF can read .doc files, and docx4j's `org.docx4j.convert.in.Doc` does 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 LibreOffice or OpenOffice (via jodconverter) to convert the doc to docx, which docx4j can then process. If you need to return a binary .doc, LibreOffice or OpenOffice/jodconverter can convert the docx back to .doc.

# Getting Help: the docx4j forum

Free community support is available in the docx4j forum, at http://www.docx4java.org/forums/ and on Stack Overflow.

Before posting, please:

- check this document doesn't answer your question
- try to help yourself: people are unlikely to help you if it looks like you are asking someone else to do lots of work you presumably are being paid to do!
- ensure your post says which version of docx4j you are using, and contains your Java code (between [java] .. and .. [/java]) and XML (between [xml] .. and .. [/xml]), and if appropriate a docx/pptx/xlsx attachment
- consider browsing relevant docx4j source code

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

# Using docx4j via Maven

docx4j is in Maven Central.  For Maven users, this makes it really easy to get going with docx4j.

With Eclipse and m2eclipse installed, you just add docx4j, and you're done.  No need to mess around with manually installing jars, setting class paths etc.

The blog entry hello-maven-central shows you what to do, starting with a fresh OS (Win 7 is used, but these steps would work equally well on OSX or Linux).

# Using docx4j binaries

If Maven is not for you, you can download the latest version of docx4j from http://www.docx4java.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 or zip version, or in the relevant subdirectory.

# Command Line Samples

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-1.2.17.jar org.docx4j.samples.OpenMainDocumentAndTraverse [input.docx]
```

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

xmlgraphics-commons-1.5.jar

commons-logging-1.1.1.jar

on your classpath.

# docx4j dependencies

## *slf4j*

To do anything with docx4j, you need **slf4j** on your classpath. As the slf4j website puts it:

> The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks (e.g. java.util.logging, logback, log4j) allowing the end user to plug in the desired logging framework at *deployment* time.

(In 2.8.1 and earlier, docx4j used log4j directly)

So you need the slf4j api jar on your classpath:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.5</version>
</dependency>
```

If you want to use log4j, then include it, and:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.5</version>
</dependency>
```

## *other dependencies*

Depending what you want to do, the other dependencies will be required.

| Dependency | Comment |
|---|---|
| +- org.slf4j:slf4j-api:jar:1.7.5 | Logging |
| +- org.plutext:jaxb-xmldsig-core:jar:1.0.0 | |
| +- commons-lang:commons-lang:jar:2.4 | diffx |
| +- commons-codec:commons-codec:jar:1.3 | ole introspection |
| +- commons-io:commons-io:jar:1.3.1 | |
| +- org.apache.xmlgraphics:xmlgraphics-commons:jar:1.5 | image support |
| \| \- commons-logging:commons-logging:jar:1.0.4 | |
| +- org.apache.xmlgraphics:fop:jar:1.1 | PDF output |
| +- org.plutext:jaxb-xslfo:jar:1.0.1 | PDF output |
| +- org.apache.avalon.framework:avalon-framework-api:jar:4.3.1 | Font support |
| +- org.apache.avalon.framework:avalon-framework-impl:jar:4.3.1 | |
| +- xalan:xalan:jar:2.7.1 | (X)HTML, PDF export |
| \| \- xalan:serializer:jar:2.7.1 | |
| +- org.plutext:jaxb-svg11:jar:1.0.2 | Pptx export |
| +- net.arnx:wmf2svg:jar:0.9.0 | |

| | |
|---|---|
| +- org.apache.poi:poi-scratchpad:jar:3.8<br>\|  \- org.apache.poi:poi:jar:3.8<br><br>+- org.antlr:antlr-runtime:jar:3.3<br>+- org.antlr:stringtemplate:jar:3.2.1<br>\|  \- antlr:antlr:jar:2.7.7<br><br>Optional:<br>+- org.eclipse.persistence:org.eclipse.persistence.moxy:jar:2.5.1<br>\|  +- org.eclipse.persistence:org.eclipse.persistence.core:jar:2.5.1<br>\|  \|  \- org.eclipse.persistence:org.eclipse.persistence.asm:jar:2.5.1<br>\|  \- org.eclipse.persistence:org.eclipse.persistence.antlr:jar:2.5.1 | OLE, binary import<br><br>OpenDoPE |

# 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 XML parts in a docx/pptx/xlsx.

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.

You can also use the JAXB reference implementation (eg v2.2.4). If you want to use that in preference to the version included in the JDK, do so using the endorsed directory mechanism.

With docx4j 3.0, you can choose to use MOXy instead. To do so, simply include docx4j-MOXy-JAXBContext-3.0.0.jar and the MOXy jars on your classpath. If you are using Maven, this means adding the following to your POM:

```
<dependency>
        <groupId>org.docx4j</groupId>
        <artifactId>docx4j-MOXy-JAXBContext</artifactId>
        <version>3.0.0</version>
</dependency>
<dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>org.eclipse.persistence.moxy</artifactId>
        <version>2.5.1</version>
</dependency>
```

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

# Docx4j source code

Docx4j source is on GitHub at https://github.com/plutext/docx4j .

We accept pull requests; pull requests are presumed to be contributions under ASLv2 per our contributor agreement.

See docx4j-from-github-in-eclipse for details.

Source code can also be downloaded from Maven Central (search for docx4j at search.maven.org).

Our old subversion repositoryat `http://www.docx4java.org/svn/docx4j/trunk/docx4j` is obsolete.

# Javadoc

Javadoc can be downloaded from Maven Central (search for docx4j at search.maven.org), but you'll find the source code much more useful! See above.

# Building docx4j from source

Get the source code from GitHub (see above), then… (you probably want to skip down to the next page, to get it working in Eclipse).

## *Command line -via Maven*

```
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.

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*

See docx4j-from-github-in-eclipse.

Not working?

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 (remove "Excluded: **" if present!):
    - src/main/java
    - src/pptx4j/java
    - src/xslx4j/java
    - src/diffx
    - src/glox4j

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

## *Using a different IDE?*

Please post setup instructions in the forum, or as a wiki page on GitHub. Thanks!

[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, **".

# Open an existing docx/pptx/xlsx document

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

To load a document or "Flat OPC" XML file, all you have to do is:

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

With docx4j 3.0, you can use the façade:

```
WordprocessingMLPackage wordMLPackage =
        Docx4J.load(new java.io.File(inputfilepath));
```

which does the same thing under the covers.

There are similar signatures to load from an input stream.

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

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

After that, you can manipulate its contents.

A similar approach works for pptx files:

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

And similarly for xlsx files.

# OpenXML concepts

To do anything much beyond this, you need to have an understanding of basic WordML concepts (or PresentationML or SpreadsheetML).

According to the Microsoft Open Packaging spec, each docx document is made up of a number of "Part" files, zipped up.

An easy way to get an understanding of this is to unzip a docx/pptx/xlsx using your favourite zip utility. Even easier is to visit http://webapp.docx4java.org and explore your file using "PartsList". You can also generate code that way.

A Part is usually XML, but might not be (an image part, for example, isn't).

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, upload it to the PartsList webapp, or run the "Parts List" sample (see 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).

See also the free "Open XML Explained" ebook by Wouter Van Vugt.

# 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 started with

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.

docx4j started with ECMA-376 1st Edition. Where appropriate later versions of the schemas are used. docx4j 3.0 uses MathML 2ed, PresentationML 2ed, and SpreadsheemML 4ed transitional.

[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

Docx4j can open documents which contain Word 2010, 2013 specific content. The key extensions are supported. For other stuff, for example, <w14:glow w14:rad="101600"> it will look for and try to use mc:AlternateContent contained in the document. If you use docx4j to save the document, the w14:glow won't be there any more (ie the docx will effectively be a Word 2007 docx).

# Architecture

Docx4j has 3 layers:

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

   OpenPackaging handles things at the Open Packaging Conventions level.

   It includes objects corresponding to each Office file type:

| | |
|---|---|
| docx | org.docx4j.openpackaging.packages.**WordprocessingMLPackage** |
| pptx | org.docx4j.openpackaging.packages.**PresentationMLPackage** |
| xlsx | org.docx4j.openpackaging.packages.**SpreadsheetMLPackage** |

   and is responsible for unzipping the file into a set of objects inheriting from Part;

   **`openpackaging`** also includes functionalitiy allowing parts to be added/deleted; saving the docx/pptx/xlsx etc

   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. To explore these first two layers for a given document, upload it to the PartsList webapp.

   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.

   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 by using the **`get|setContents`** method.

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). Actually, docx4j 3.0 is lazy; it only does this when first needed.

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. Here is representative (non-exhaustive) content:

| | |
|---|---|
| `Jc` | `org.docx4j.wml`<br>`org.docx4j.dml`<br>`org.docx4j.dml.picture`<br>`org.docx4j.dml.wordprocessingDrawing`<br>`org.docx4j.vml`<br>`org.docx4j.vml.officedrawing`<br>`org.docx4j.math` |
| `jcThemePart` | `org.docx4j.dml` |
| `jcDocPropsCore` | `org.docx4j.docProps.core`<br>`org.docx4j.docProps.core.dc.elements`<br>`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<br>org.docx4j.dml<br>org.docx4j.dml.picture |

You'll find XmlUtils.marshalToString very useful as you put your code together. With this, you can easily output the content of a JAXB object, to see what XML it represents.

# 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.

So it's a bit like unzipping the docx/pptx/xlsx file, but it tells you what Java objects are being used for each part.

A more fully featured tool is the PartsList online webapp. With this, you can browse through the package, look up what elements mean in the spec, and generate code.

You can run PartsList locally from a command line:

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

though I always find it easier to run it from my IDE. Example output:

```
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. `RelationshipsPart` contains various useful utility methods, for example:

```
/** Gets a loaded Part by its id */
public Part getPart(String id)

public Part getPart(Relationship r ) {
```

The `RelationshipsPart` is the key player when it comes to adding/removing images and other parts from your document.

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, but you generally shouldn't add/remove things here directly!

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.

Its XML will look something like:

```
<w:document xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" >
  <w:body>
    <w:p >
      <w:pPr>
        <w:pStyle w:val="Heading1"/>
      </w:pPr>
      <w:r>
        <w:t>Hello World</w:t>
      </w:r>
    </w:p>
    :
    <w:sectPr >
      <w:pgSz w:w="12240" w:h="15840"/>
      <w:pgMar w:top="1440" w:right="1440" w:bottom="1440" w:left="1440" w:header="708"
w:footer="708" w:gutter="0"/>
    </w:sectPr>
  </w:body>
</w:document>
```

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 you can skip some of that with:

```
/**
 * Convenience method to getJaxbElement().getBody().getContent()
 */
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:

| | | |
|---|---|---|
| Body | w:body | document body |
| P | w:p | paragraph |
| R | w:r | run |
| Tbl<br>Tr<br>Tc | w:tbl<br>w:tr<br>w:tc | table<br>table row<br>table cell |
| SdtBlock<br>SdtRun<br>CTSdtRow<br>CTSdtCell | w:sdt<br>w:sdt<br>w:sdt<br>w:sdt | content controls; see the method `getSdtContent()` |

As well as

- Hdr, Ftr

Content is generally stored in a plain old Java List. So there are familiar methods for inserting content at the end of the list, or other location in it.

Read on for how to add text etc.

# Samples

The package org.docx4j.samples contains examples of how to do things with docx4j.

The docx4j samples include:

Basics

- CreateWordprocessingMLDocument
- DisplayMainDocumentPartXml
- OpenAndSaveRoundTripTest
- PartsList

Navigating the document body

- OpenMainDocumentAndTraverse

- XPathQuery

Output/Transformation

- ConvertOutHtml
- ConvertOutPDF

Import (X)HTML

- AltChunkXHTMLRoundTrip
- AltChunkAddOfTypeHtml
- ConvertInXHTMLDocument
- ConvertInXHTMLFragment

Image handling

- ImageAdd
- ImageConvertEmbeddedToLinked

Part Handling

- PartCopy
- PartLoadFromFileSystem
- PartsList
- PartsStrip

Document generation/document assembly using content controls

- ContentControlsAddCustomXmlDataStoragePart
- ContentControlsXmlEdit
- ContentControlsApplyBindings
- ContentControlBindingExtensions
- ContentControlsPartsInfo
- AltChunkAddOfTypeDocx
- VariableReplace (not recommended)

Specific docx features

- BookmarkAdd
- CommentsSample
- HeaderFooterCreate
- HeaderFooterList
- HyperlinkTest
- NumberingRestart
- SubDocument
- TableOfContentsAdd
- TemplateAttach (attach your.dotx)

Miscellaneous

- CompareDocuments
- DocProps
- Filter (remove proof errors, w:rsid)
- MergeDocx
- UnmarshallFromTemplate

Flat OPC XML

- ConvertOutFlatOpenPackage
- ConvertInFlatOpenPackage

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

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

https://github.com/plutext/docx4j/tree/master/src/samples (note new location for docx4j 3.0)

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

# 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.

It is based on:

```
public org.docx4j.wml.P createParagraphOfText(String simpleText) {

      org.docx4j.wml.ObjectFactory  factory  =  Context.getWmlObjectFactory();
       org.docx4j.wml.P   para = factory.createP();

       if (simpleText!=null) {
             org.docx4j.wml.Text   t = factory.createText();
             t.setValue(simpleText);

             org.docx4j.wml.R   run = factory.createR();
             run.getContent().add(t); // ContentAccessor
```

```
                para.getContent().add(run); // ContentAccessor
        }

        return para;
    }
```

Notice that the paragraph, the run, and indeed the Body, all implement the `ContentAccessor` interface:

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

    public List<Object> getContent();

}
```

The add method adds the content at the end of the document. If you want to insert it somewhere else, you could use something like:

```
    public org.docx4j.wml.P addParaAtIndex(MainDocumentPart mdp,
                String simpleText, int index) {

        org.docx4j.wml.ObjectFactory factory = Context.getWmlObjectFactory();
        org.docx4j.wml.P para = factory.createP();

        if (simpleText != null) {
            org.docx4j.wml.Text t = factory.createText();
            t.setValue(simpleText);

            org.docx4j.wml.R run = factory.createR();
            run.getContent().add(t);

            para.getContent().add(run);
        }

        mdp.getContent().add(index, para);

        return para;
    }
```

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
- upload it to the PartsList online webapp
- 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
- if you use Google's Chrome web browser, try **OOXML Viewer for Chrome**.

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, the easiest way by far is to use the PartsList online webapp. Alternatively, you could run `OpenMainDocumentAndTraverse` on your document, or use Eclipse to search the relevant schema (in /xsd) or source code.

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 to create stuff 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);
```

The PartsList online webapp can generate appropriate code for you, using both of these approaches. It also links to the Open XML spec documentation for the element.

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

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

# Formatting Properties

Usually you format the appearance of things via an object's properties element:

| Object | Method |
|---|---|
| Paragraph | P.getPPr() |
| Run | R.getRPr() |
| Table | Tbl.getTblPr() |
| Table row | Tr.getTrPr() |
| Table cell | Tc.getTcPr() |

In a docx, the appearance of text is basically determined by the style in the styles part which applies to it (styles can inherit from other styles), plus any direct formatting.

Docx4j contains code for working out the effective formatting, which is used in its PDF output.

In XHTML import, docx4j converts CSS into formatting properties.

# 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 add content, see ***General strategy/approach for creating stuff*** above. If you want format your table (make it prettier), see Formatting Properties immediately above.

Or you can use the PartsList online webapp to generate the code.

If you are looking to fill table rows with data, consider OpenDoPE content control data binding (in which you "repeat" a table row).

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

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

You can 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 are a few limitations however in the JAXB reference implementation:

- 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.
- For some objects,JAXB can't get parent (with getParent)
- For some document, JAXB can't set up the XPath

If these limitations are causing you problems, try using MOXy as your JAXB implementation, or see Traversing immediately below for a different approach.

# Traversing a document

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

This is an alternative to XSLT, which doesn't require marshalling to a DOM document and unmarshalling again.

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 earlier, many objects (eg the document body, a paragraph, a run), have a List containing their content. Traversal works by iterating over these lists.

Traversing is a very useful approach for finding and altering parts of the document.

For example, it is used in docx4j 2.8.0, to provide a way of producing HTML output without using XSLT/Xalan.

The org.docx4j.finders package contains classes which make it convenient to find various objects.

It is often superior to using XPath (owing to the limitations in the JAXB reference implementation noted above).

Note also, in **package** org.docx4j.utils:

```
/**
 * Use this if there is only a single object type (eg just P's)
 * you are interested in doing something with.
public class SingleTraversalUtilVisitorCallback
```

ImageConvertEmbeddedToLinked sample contains an example of the use of the above.

```
/**
 * Use this if there is more than one object type (eg Tables and Paragraphs)
 * you are interested in doing something with during the traversal.
public class CompoundTraversalUtilVisitorCallback
```

# 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).

# Importing XHTML

From docx4j 2.8.0, docx4j can convert XHTML content (paragraphs, tables, images) into native WordML, reproducing much of the formatting. If you are using this, v3 is highly recommended.

From v3, the XHTML Import functionality is now a separate project on GitHub.

The reason being that its main dependency – Flying Saucer - is licensed under LGPL v2.1 (as opposed to ASL v2, which docx4j's other dependencies use).

If you want this functionality, you have to add these jars to your classpath.

See the samples at https://github.com/plutext/docx4j-ImportXHTML/tree/master/src/samples

# docx to (X)HTML

docx4j can convert a docx to HTML or XHTML. You will find the generated HTML is clean (in comparison to the HTML Word produces).

Docx4j's HTML output is suitable for documents which contain paragraphs, tables and images. It can't handle more exotic features, such as equations, SmartArt, or WordArt (DrawingML or VML).

Elsewhere on the web, you'll find XSLT which can convert docx to HTML. That XSLT is very complex, since it has to derive effective formatting from the hierarchy.

In contrast, in docx4j, that logic is implemented in Java. Because of this, docx4j's XSLT is simple (Java XSLT extension functions do the heavy lifting).

In docx4j, you can create output using XSLT, or by traversing the document in Java. The façade lets you specify which:

```
                //Prefer the exporter, that uses a xsl transformation
                Docx4J.toHTML(htmlSettings, os, Docx4J.FLAG_EXPORT_PREFER_XSL);
                //Prefer the exporter, that doesn't use a xsl transformation (= uses a visitor)
//              Docx4J.toHTML(htmlSettings, os, Docx4J.FLAG_EXPORT_PREFER_NONXSL);
```

See the sample on GitHub at src/samples/docx4j/org/docx4j/samples/ConvertOutHtml.java

If you have output logging enabled, 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.***

# docx to PDF

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

You can try it with the online demo, at
http://webapp.docx4java.org/OnlineDemo/docx_to_pdf_fop.html

Generally speaking, docx4j's PDF output is suitable for documents which contain paragraphs, tables and images. It can't handle more exotic features, such as equations, SmartArt, or WordArt (DrawingML or VML).

At present, Apache FOP is integrated into docx4j for creating the PDF.

See the sample on GitHub at src/samples/docx4j/org/docx4j/samples/ConvertOutPDF.java.

If you have output logging enabled, 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.***

# 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 ConvertOutPDF 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 - DOCX

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.

# Image Handling – PPTX

See the pptx4j InsertPicture sample.

# Adding Headers/Footers

See the HeaderFooter sample for how to do this.

# Merging Documents and Presentations

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.

The commercial edition of docx4j includes "MergeDocx" code 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.

To try it, visit http://webapp.docx4java.org/

The commercial edition of docx4j includes MergePptx, which you can use to concatenate presentations.

The MergeDocx 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>
```

Generating page numbers is a challenge without Word.

The commercial edition of docx4j includes code to generate/update a TOC, including page numbers, based on a basic page layout model.

# 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, which could even be an image (or with docx4j, arbitrary XHTML). (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 (v2.8.0) can also take encoded XHTML and convert this to docx content. See further OpenDoPE_XHTML.docx in the docx4j docs directory.

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

# 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);
    }
```

The easiest way is to use the PartsList online webapp to generate the relevant code.

# 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 or priority support from Plutext

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 or StackOverflow. 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