

# Docx4j – Getting Started

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](#)).

## 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**. 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.

Docx4j relies heavily on **JAXB**, the JCP standard for Java – XML binding.

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](http://www.plutext.com)).

## Is docx4j for you?

Docx4j is for processing docx documents in Java.

It isn't for old binary (.doc) files. For those, look at Apache POI's HWPf. (In fact, docx4j uses HWPf for basic conversion of .doc to .docx)

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 for processing Excel documents. It can also be used to process Word documents, and since it 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)
- Create new docx
- Programmatically manipulate the docx document (of course)
- Template substitution; CustomXML binding
- Import a binary doc (uses Apache POI's HWPf)
- 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)

## Projects using docx4j

Docx4all is a Swing-based word processor.

Plutext collaboration for Word 2007 uses docx4j on the server, to shred Word documents into smaller pieces which can be independently versioned.

Please contact Plutext if you would like your project to be listed here.

## 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 will support Word 2010 docx files.

## 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.

## Using docx4j via Maven

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

## JDK versions

You need to be using Java 1.5+.

This is because of JAXB<sup>1</sup>. If you must use 1.4, retrotranslator can [reportedly make](#) it work.

---

1 <http://forums.java.net/jive/thread.jspa?threadID=411>

## 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\_).

### Log4j

Docx4j uses log4j for logging. To 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>

## 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

### ***Command line - Quick Instructions***

"Quick" that is, provided you have maven and ant installed. Note that we only use maven to grab the dependencies, not to do the actual build.

Create a directory called workspace, and cd into it.

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

open pom.xml, find the line which reads

```
<systemPath>/usr/lib/jvm/java-6-sun/jre/lib/rt.jar</systemPath>
```

and edit it to suit your system.

```
mvn install
```

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](http://subclipse.tigris.org/update_1.2.x)
- Install [Maven and the Eclipse plugin](#)

And, as discussed above:

- Java 1.5 or 6
- JAXB: **both** the JAXB implementation included in Java 6, **and** the 2.x reference implementation. (This is the price of supporting either at runtime)

### 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.

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

- Build Path > Configure Build Path > Java Build Path > Source tab
- Click on src, then press the remove button
- Then click "add folder" and navigate through to src/main/java and tick 'java'
- Then add src/diffx as well

The Maven bit:

- Make sure you have Maven and its plugin installed - see Prerequisites above.
- Run mvn install in the docx4j dir from a command prompt (just in case)
- Right click on project > Maven 2 > Enable

The project should now be working in Eclipse without errors<sup>2</sup>.

### Open an existing docx 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));
```

---

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, \*\*\*".

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.

## 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).

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>.

### 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	<a href="#">org.docx4j.wml</a> <a href="#">org.docx4j.dml</a> <a href="#">org.docx4j.vml</a> <a href="#">org.docx4j.vml.officedrawing</a> <a href="#">org.docx4j.math</a>
jcThemePart	<a href="#">org.docx4j.dml</a>
jcDocPropsCore	<a href="#">org.docx4j.docProps.core</a>  <a href="#">org.docx4j.docProps.core.dc.elements</a>  <a href="#">org.docx4j.docProps.core.dc.terms</a>
jcDocPropsCustom	<a href="#">org.docx4j.docProps.custom</a>
jcDocPropsExtended	<a href="#">org.docx4j.docProps.extended</a>
jcXmlPackage	<a href="#">org.docx4j.xmlPackage</a>
jcRelationships	<a href="#">org.docx4j.relationships</a>
jcCustomXmlProperties	<a href="#">org.docx4j.customXmlProperties</a>
jcContentTypes	<a href="#">org.docx4j.openpackaging.contenttype</a>

## Architecture

Docx4j has 3 layers:

1. **org.docx4j.openpackaging**  
OpenPackaging handles things at the Open Packaging Conventions level: unzipping a docx into a set of objects inheriting from Part; allowing parts to be added/deleted; saving the docx  
This layer is based originally on OpenXML4J (which is also used by Apache POI). Parts are generally subclasses of org.docx4j.openpackaging.parts.JaxbXmlPart  
A JaxbXmlPart has a content tree:

```
public Object getJaxbElement() {
    return jaxbElement;
}
public void setJaxbElement(Object jaxbElement) {
    this.jaxbElement = jaxbElement;
}
```
2. The **jaxb content tree** is the second level of the three layered model.  
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.

## Samples

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

### Basics

- CreateWordprocessingMLDocument
- DisplayMainDocumentPartXml
- OpenAndSaveRoundTripTest
- OpenMainDocumentAndTraverse

### Output/Transformation

- CreateHtml
- CreatePdf

### Flat OPC XML

- ExportInPackageFormat
- ImportFromPackageFormat

### Image handling

- AddImage
- ConvertEmbeddedImageToLinked

### Part Handling

- CopyPart
- ImportForeignPart

- PartsList
- StripParts

#### Miscellaneous

- CompareDocuments
- CreateDocxWithCustomXml
- CustomXmlBinding
- 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.

## Traversing a docx

[OpenMainDocumentAndTraverse.java](#) in the samples directory shows you how to traverse the JAXB representation of a docx.

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();
```

## 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);

```

## 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)

```

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.

## 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);
```

## 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.

## 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);
```

## 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.

## 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)

docx4j will eventually accommodate these cases, but doesn't right now.

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>
```

## Tips and Tricks

### ***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”.

### ***Remember UnmarshalFromString***

When you are manipulating docx documents, it is often useful to unmarshal snippets of 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);
```

## 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)
```

## @XmlRootElement

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

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 );
```

## The docx4j forum

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

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, you are welcome to purchase programming, consulting and priority support from <http://www.plutext.com/>

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

## Roadmap

**Word 2010 support.** Support for the new XML elements/schemas introduced with Word 2010, and for the compatibility mechanism. This is the main justification for the 3.0 label.

**HTML exporters:** get rid of old ones; standardise on NG2. The idea is to remove any 'which should I use' confusion, and focus effort/know-how.

**PDF exporters:** standardise on viaXSLFO, and get rid of viaText and viaHTML. As with HTML, the idea is to remove any 'which should I use' confusion, and focus effort/know-how. docx4j could produce XSL FO only, and rely on the user to have FOP or equivalent to actually produce the PDF. This will reduce dependencies, making docx4j lighter. The goal would be to remove the fop jar (2.8M), PDF renderer jar (1.6M), iText jar (1.1M), and core-renderer (1M).

**Font handling:** remove the panose stuff, so we don't need a customised FOP jar.

**Layout model/intermediate format:** docx4j contains a DocumentModel, which could be further developed to support:

- Search/replace
- Estimating page content
- XSLT, by enclosing sections, lists

**Inserting OLE objects:** so spreadsheets, PDFs etc can be embedded.

## 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>

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](mailto:jason@plutext.org), or [jharrop@plutext.com](mailto:jharrop@plutext.com)

## Endnotes

