

\* big faceless organization

# The Big Faceless PDF Library

version 2.11

## User Guide

# Introduction

Thank you for your interest in the [Big Faceless PDF Library](#). This userguide will give you an overview of what the library is capable of, and start you off with some simple examples. For more detailed information, please see the API documentation supplied in HTML format with this package, and look at the examples supplied with the package.

## *What is it?*

The Big Faceless PDF Library is a collection of classes which allow easy creation of PDF® documents from Java™. When linked with your own application, it can be used to rapidly create, edit, view or print PDFs (like this userguide) from applications, Applets or Servlets.

The library is small, fast, easy to use and integrate into your projects, and is written in 100% pure Java (requires Java 1.4 or later). It's well documented and comes with many examples, and as it uses no native code or third-party packages it's easy install and run from applications, EJB's or Servlets on any Java 1.4 platform.

## *Features*

Here's a brief summary of the libraries features, for the impatient.

- Extract text and images from PDFs, display them or convert them to TIFF with the **Viewer Extension**
- Native Unicode™ support, even in the document body. No worrying about codepages and encodings, it *just works*.
- Full support for creating and editing AcroForms
- Edit existing PDF documents with the `PDFReader` class.
- 40 and 128-bit encryption using RC4 or AES, for eyes-only documents.
- Digitally sign documents for authenticity and non-repudiation
- Extend the library with custom Signature and Encryption handlers
- Full embedding of TrueType or Type 1 fonts, with subsetting for smaller fonts.
- Japanese, Chinese and Korean font support
- Right to left and bi-directional text is fully supported.
- Embed JPEG, PNG, GIF, PNM, TIFF or `java.awt.Image` images.
- Supports Adobes XMP™ specification, for embedding and extracting XML metadata from PDF documents
- Add hyperlinks to text or images
- Add Annotations and Audio samples to the document
- Insert many different types of barcode directly - no Barcode font required!
- Better text layout with track and pair kerning, ligatures and justification
- Paint text or graphics using "patterns"
- Simplify complex documents by defining and applying "*Styles*".
- Full support for PDF features like bookmarks, compression and document meta-information
- Non-linear creation of documents - pages can be created and edited in any order.
- Intelligent embedding. Fonts and images can be reused without increasing the file size.

The library *officially* only produces PDFs that are valid on **Acrobat 4.0 and up** (although most documents will work under Acrobat 3.0 as well, we don't support it). The library output has been verified against Acrobat 4.0 to 8.0 on Windows, along with numerous PDF viewers on other platforms (including Linux and OS/X).

# Installation

## ***Installation for use in an Application or Servlets***

Installing the library is as simple as unpacking it and adding the files `bfopdf.jar`, `bfopdf-cmap.jar` and `bfopdf-qrcode.jar` to your `CLASSPATH` or the `WEB-INF/lib` folder of your web application. You'll find several other files in the package, including `README.txt`, containing the overview of the package, `docs/LICENSE.txt` containing the licencing agreement, and the `docs` and `example` directories containing the API documentation and the examples respectively.

## ***Installation for use in a Java Web Start Application***

Deployment of the PDF Viewer as a Java Web Start application allows you to take advantage of the "pack200" compressed `bfopdf.jar.pack.gz` version of the Jar file, and the ability of JNLP to download components only when they're required. First you'll need to set up your web server to negotiate the pack200 version of the file - this involves the following steps:

1. Copy the `bfopdf.jar.pack.gz` file to the same location as your `bfopdf.jar` file.
2. Copy the `sample\jnlp\servlet\jnlp-servlet.jar` file that came with your Java distribution to the `WEB-INF\lib` folder of your web application. This Jar contains a Servlet that negotiates the pack200 version of the Jar with the client `ClassLoader`.
3. Modify the `WEB-INF\web.xml` file of your web application to map all requests for Jar files to this servlet by adding the following lines:

```
<servlet>
  <servlet-name>JnlpDownloadServlet</servlet-name>
  <servlet-class>jnlp.sample.servlet.JnlpDownloadServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>JnlpDownloadServlet</servlet-name>
  <url-pattern>*.jar</url-pattern>
</servlet-mapping>
```

The you'll need to create a `jnlp` file describing your application. A sample is supplied with the library package under the name `bfopdf.jnlp`. Be sure to update the codebase with the URL of your JNLP file, then place the file in the same directory as the application Jars.

## ***Installation for use as an Applet***

Installation of the PDF Viewer as an Applet is very similar to the Java Web Start installation described above. Set up the web server to serve the "pack200" compressed Jar as described above. You then have the option of setting up the applet tag to download the `bfopdf-cmap.jar` and `bfopdf-qrcode.jar` Jars completely:

```
<applet code="org.faceless.pdf2.viewer2.PDFViewerApplet.class" width="600" height="500"
  archive="bfopdf.jar, bfopdf-cmap.jar, bfopdf-qrcode.jar"></applet>
```

Alternatively, for a much smaller initial download modify the applet tag shown above to reference only the `bfopdf.jar` file then unpack the resources from the `bfopdf-cmap.jar` and `bfopdf-qrcode.jar` files into the same directory. This will allow the applet to download the individual resources as required.

```
C:\webapplication> dir
Directory of C:\webapplication
30/06/2007  01:29          1,820,091 bfopdf.jar
30/06/2007  01:29          755,371 bfopdf.jar.pack.gz
30/06/2007  01:29          1,386,008 bfopdf-cmap.jar
30/06/2007  01:29          1,029,332 bfopdf-qrcode.jar
30/06/2007  01:29    <DIR>          WEB-INF
                4 File(s)          4,990,802 bytes
C:\webapplication> jar xf bfopdf-cmap.jar org
C:\webapplication> jar xf bfopdf-qrcode.jar org
C:\webapplication> del bfopdf-cmap.jar bfopdf-qrcode.jar
C:\webapplication> dir
Directory of C:\webapplication
30/06/2007  01:29          1,820,091 bfopdf.jar
30/06/2007  01:29          755,371 bfopdf.jar.pack.gz
30/06/2007  01:29    <DIR>          org
30/06/2007  01:29    <DIR>          WEB-INF
                2 File(s)          2,575,462 bytes
```

# Section 1. Creating new PDF documents

## The Classic Example

```
1. import java.io.*;
2. import java.awt.Color;
3. import org.faceless.pdf2.*;
4.
5. public class HelloWorld
6. {
7.     public static void main(String[] args) throws IOException
8.     {
9.         PDF pdf = new PDF();
10.
11.         PDFPage page = pdf.newPage("A4");
12.
13.         PDFStyle mystyle = new PDFStyle();
14.         mystyle.setFont(new StandardFont(StandardFont.TIMES), 24);
15.         mystyle.setFill(Color.black);
16.
17.         page.setStyle(mystyle);
18.         page.drawText("Hello, World!", 100, page.getHeight()-100);
19.
20.         OutputStream out = new FileOutputStream("HelloWorld.pdf");
21.         pdf.render(out);
22.         out.close();
23.     }
24. }
```

Example 1 - A "Hello World" program

It doesn't get simpler than the classic HelloWorld example - a copy of which is included in the example sub-directory of the package, so you can try it yourself.

A quick glance at this example reveals several points to remember when using the library.

- The package is called `org.faceless.pdf2`, and the main class is the `PDF` class.
- Each PDF document is made of *pages*, which contain the visible contents of the PDF.
- Fonts and colors are set using a *style*, which is then applied to the page.
- The completed document is sent to an `OutputStream`, which can be a file, a servlet response, a `ByteArrayOutputStream` or anything else.

In this example, we've used one of the "standard" 14 fonts that are guaranteed to exist in all PDF viewers, **Helvetica**. If this isn't enough, the library can embed TrueType™ and Adobe Type 1 fonts, or for Chinese, Japanese and Korean other options exist. We'll cover more on fonts later.

Colors are defined with the standard `java.awt.Color` class. A PDF document can have two different colors - a *line color* for drawing the outlines of shapes, and a *fill color* for filling shapes. Read on for more on colors.

### ► Page Co-ordinates

By default, pages are measured from their bottom-left corner in **points** (1/72nd of an inch). So `(100, page.getHeight()-50)` is 100 points in from the left and 50 points down from the top of the page.

You can change this using the `setUnits` method of the `PDFPage` class, to measure in inches, centimeters etc. from any corner of the page. See the API documentation for more info.

# Defining and applying Styles

Document creators have found that defining the look of their document with a *style*, rather than setting the font, color and so on separately, makes life simpler. Everything relating to how the content of the page looks is controlled using the `PDFStyle` class.

Some methods in this class relate just to images, and others just to text. Here are some of the more common ones to get you started.

<code>setFillColor</code>	<i>Text, Graphics</i>	Sets the color of the interior of the shape. For example, this table has a Fill color of light gray, while this text has a Fill color of black.
<code>setLineColor</code>	<i>Text, Graphics</i>	Sets the color of the outline of the shape. This table has a Line color of black. Setting a Line color on text has no effect, unless the <code>setFontStyle</code> method is used to turn on text outlines.
<code>setFont</code>	<i>Text</i>	Sets the font and the font size
<code>setTextAlign</code>	<i>Text</i>	Sets the alignment of the text - left, centered, right or justified, and optionally the vertical alignment too.
<code>setTextUnderline</code>	<i>Text</i>	Whether to <u>underline</u> the text. See also <del><code>setTextStrikeout</code></del>
<code>setTextLineSpacing</code>	<i>Text</i>	Determines how far apart each line of text is - single, double spaced, line-and-a-half or any other value.
<code>setTextIndent</code>	<i>Text</i>	Sets how far to indent the first line of text, in points.
<code>setLineWeighting</code>	<i>Graphics, Text</i>	Set the thickness of any lines drawn using the Line color, in points (including outlined text).
<code>setLineDash</code>	<i>Graphics</i>	Set the <i>pattern</i> to draw lines with. Normally lines are solid, but you can draw dashed lines using this method.
<code>setLineJoin/ setLineCap</code>	<i>Graphics</i>	Determines how the ends of a line are drawn - squared off, rounded and so on. The line cap is the shape at the end of a line, and the line join is the shape where two lines meet at a corner.

*Table 1 - Common methods in the PDFStyle class*

Styles are more than just a useful way of grouping aspects of appearance together - they help you to *manage* the look and feel of your document:

- If many different items are meant to have the same look, give them all the same style. You only need to alter the style once to change their appearance.
- Styles can be *extended* - create a copy of a current style and change a single aspect, and everything else will be inherited.
- By using a relatively limited number of styles, a document has a more consistent "look-and-feel".
- Name your styles header, sourcecode and so on to get a clearer perspective of the *structure* of the document.

## Colors

As you can see from the table to the right, PDF documents have two active colors. The Line color (or "Stroke" color) is the color used to draw outlines of shapes or text, and the Fill color is the color used to fill those shapes. Text is normally drawn with just the fill color, although as you can see opposite you can call the `style.setFontStyle()` method in the `PDFStyle` class to set text to outlined, filled and outlined or even invisible (which we've seen used in an OCR application, so it's not *completely* useless).

## ► Outline or Solid?

- For outlined shapes, set the Line color only
- For solid shapes, set the Fill color only
- For both, set both colors!
- For text, call `style.setFontStyle()`

## Calibrated Colors

The library has full support for calibrated, or device-independent colors. The PDF specification allows colors to be calibrated against a *ColorSpace*, which is essentially a complex mathematical function which determines exactly which shade of red, white or black you get. For programmers who are used to working with computer monitors with a "brightness" and "contrast" knob, the concept of calibrated color may seem a little alien, but in the print world it's essential.

The Java language designers got it correct right from the start, and defined the excellent `java.awt.Color` class in such a way as to make defining colors simple. The default color space is a standard known as sRGB, which is a W3C standard and supported by a large number of computer companies. When you specify `Color.red` in your program, you're actually getting the color "red=100%" in the sRGB colorspace. Since 1.1.5, all PDF documents generated by the Big Faceless PDF Library are calibrated to use the sRGB `ColorSpace` as well.

So how do you use other colorspace? Here's an example:

```
1. import java.awt.color.*;
2. import java.awt.Color;
3. import org.faceless.pdf2.*;
4. import java.io.IOException;
5.
6. public void colorDemo(PDFPage page) throws IOException {
7.     Color red1 = Color.red;    // sRGB colorspace, red=100%
8.
9.     // Create the same red in an ICC colorspace loaded
10.    // from a file. Here we use the NTSC color profile.
11.    ICC_Profile prof = ICC_Profile.getInstance("NTSC-Profile.icc");
12.    ICC_ColorSpace ntsc = new ICC_ColorSpace(prof);
13.    float[] comp = { 1.0f, 0.0f, 0.0f };
14.    Color red2 = new Color(ntsc, comp, 1);
15.
16.    // Create a yellow color in the device-dependent CMYK
17.    // colorspace, supplied with the library.
18.    Color yellow = CMYKColorSpace.getInstance().getColor(0,0,1,0);
19.
20.    // Create a spot color from the PANTONE™ range
21.    SpotColorSpace spot = new SpotColorSpace("PANTONE Yellow CVC", yellow);
22.    Color spotyellow = spot.getColor();
23. }
```

*Example 2 - Specifying different colorspace*

This may seem a complicated example, but first it demonstrates using four different colorspace in 20 lines. If it doesn't make sense, then you're probably not going to need that bit anyway!

You may notice we've shown you how to create the colors, but haven't shown you how to apply the colors to the PDF document. That's because colors created using these different colorspaces are treated no differently than colors using the normal sRGB colorspace. Just use the `setFillColor` and `setLineColor` methods like you would do normally - the colorspace handling is all done behind the scenes.

For more information on Spot and CMYK colors, have a look at the Java API documentation for the `CMYKColorSpace` and `SpotColorSpace` classes that are part of the PDF library.

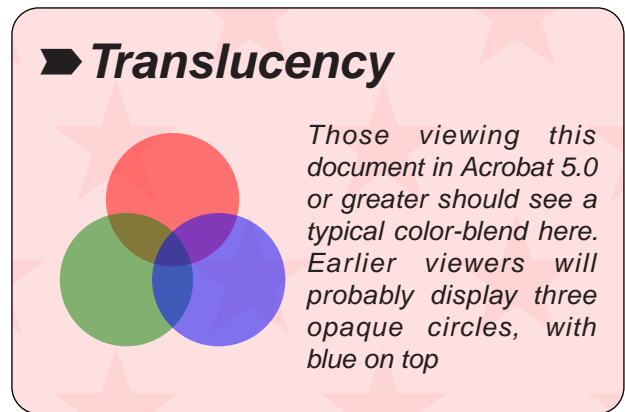
Some image formats can also use calibrated colors. PNG, TIFF and JPEG images may have an ICC color profile embedded, which the library will pick up and use automatically (GIF images do not, but can easily be converted to PNG if necessary). For all of these image formats and for `java.awt.Image` images as well, if a specific color profile is included as part of that image format, it will be used automatically.



## Transparency and Translucency

The release of Acrobat 5.0 introduced the concept of *translucent* colors to PDF, and since version 2.0 this is supported with the PDF library as well. The measure of transparency in a color is known as it's *alpha value* - a color with an alpha of 0.0 is completely transparent, while an alpha value of 1.0 is completely opaque. Specifying a color with an alpha value in Java is easily done - there are a number of constructors in the `java.awt.Color` class that take an alpha value.

For example, the image to the right shows three translucent circles. These are simply drawn by setting the fill color to `new Color(1, 0, 0, 0.5f)`, `new Color(0, 1, 0, 0.5f)` and `new Color(0, 0, 1, 0.5f)` respectively, applying the style and drawing the circles on the page.

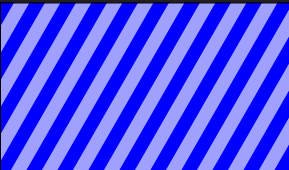
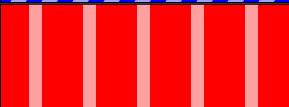
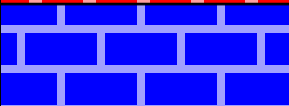


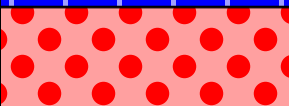
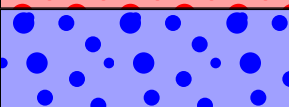



*Those viewing this document in Acrobat 5.0 or greater should see a typical color-blend here. Earlier viewers will probably display three opaque circles, with blue on top*

Translucency is also supported in the PNG and GIF image formats - basically, if you load an image with translucent colors, then assuming the PDF viewer is Acrobat 5 or later, then the image will be displayed correctly.

## Patterns

As well as solid colors, patterns and gradients can be used for special effects (for instance, the box on the top right of this page has a background pattern of stars). Several pre-defined patterns exist, or more advanced users can define their own. The `PDFPattern` class implements the `java.awt.Paint` interface, and so can be passed into the `setLineColor` and `setFillColor` methods of a `PDFStyle`. to fill **text or shapes with a pattern**.

	The "StripeXX" pattern creates a pattern of alternating stripes. "XX" may be any number of 0 to 360, and specifies the angle of the stripes in degrees. The box to the left, for example, was created using <code>new PDFPattern("Stripe30", 0, 0, 10, 10, color1, color2)</code> .
	Here is another example of a "Stripe", with vertical stripes and a different width for each stripe color. This is equivalent to <code>("Stripe0", 0, 0, 15, 5, color1, color2)</code>
	This is the pattern called "Brick". For budding brickies, this style of brick laying is known as a running pattern bond.
	This check pattern is created by passing "Check" to the <code>PDFPattern</code> constructor.
	This pattern is created by calling <code>new PDFPattern("Grid", 0, 0, 20, 20, color1, color2)</code>
	A spot pattern similar to the pattern used for half-toning in newspapers can be created by passing in "Spot" to the constructor.
	This pattern is created by calling <code>new PDFPattern("Polka", 0, 0, 20, 20, color1, color2)</code>
	Finally, a pattern of repeating 5-pointed stars (like those on the US flag) can be created with the "Star" pattern. be set

*Table 2 - Color patterns*

## Custom Patterns

Custom patterns can be created in one of two ways. First, a `PDFCanvas` can be created and a pattern drawn onto it. This can then be used as a tile to paint with. Here's a simple example showing how to paper the page with yellow smiley faces.

```
1. PDFCanvas tile = new PDFCanvas(40, 40);
2. PDFStyle style = new PDFStyle();
3. style.setFill(Color.yellow);
4. style.setLine(Color.black);
5. tile.setStyle(style);
6. tile.drawCircle(20, 20, 15);           // Draw the face
7. tile.drawCircleArc(20, 20, 11, 110, 250); // Add a smile...
8. tile.drawCircle(15, 25, 2);          // ...and two eyes
9. tile.drawCircle(25, 25, 2);
10. tile.flush(); // Optional but a good idea
11.
12. PDFPattern pattern = new PDFPattern(tile, 0, 0, 40, 40);
13.
14. PDFStyle pagestyle = new PDFStyle(); // Fill the page. Assumes PDFPage page
15. pagestyle.setFill(pattern);
16. page.setStyle(pagestyle);
17. page.drawRectangle(0, 0, page.getWidth(), page.getHeight());
```

*Example 3 - Creating your own patterns*

A second method, for those that know something about the internals of PDF, is to define a pattern as a sequence of PDF operators in a resource file. We'll cover this in more detail in a [later section](#).

# Text and Fonts

Document developers have a large choice when it comes to choosing a font for their document. First, every PDF document is guaranteed to have a set of 14 base fonts available to it, which the library calls "Standard" fonts. These are available via the `StandardFont` class, an example of which you've already seen in the "HelloWorld" example.

Secondly for Chinese, Japanese and Korean, Adobe has defined a set of standard fonts which are available in every PDF viewer, *providing the correct language pack is installed*. The language packs are part of the appropriate regional versions of Adobe Acrobat, or can be downloaded as separate packs for other versions of Acrobat. Other PDF viewers will have different requirements. These fonts are available via the `StandardCJKFont` package.

Finally, a document can contain both Adobe "Type 1" and OpenType™ fonts. Many users will have heard instead of *TrueType*™ fonts, which is a format developed jointly by Apple and Microsoft (many TrueType fonts are marked as "Apple" or "Windows" specific - the library works with either). A few years back Microsoft and Adobe finally made peace, and the two companies combined their competing formats into the "OpenType" format - so TrueType is a subset of OpenType. As of 2.3 we also support OpenType fonts with PostScript glyphs (which typically have a suffix of ".otf" rather than ".ttf")

Both Type 1 and OpenType fonts can be either be *embedded* or *referenced* in the document. Referencing the font requires the font to be available already on the target platform - because of this it's not really suitable unless your document is going to a limited audience, all of whom you know have the font installed.

Embedding a font guarantees it will be available to the PDF viewer, but increases the file size of the document. TrueType font glyphs can be subset (meaning only the characters that are used are embedded), which reduces the file size. This is generally a good idea, and TrueType fonts are subset by default when embedded.

When creating an embedded OpenType font, the user has to decide whether to embed it using 1 or 2 bytes per glyph. Using a single byte can save space, but limits the font to displaying only 255 different glyphs. This is more than enough for almost every language except Chinese, Japanese, Korean, although we generally recommend that when embedding an OpenType font, 2 bytes per glyph are used for all non-western European languages, as this is more compatible with older PDF viewers and also some text-extraction tools.

PDF does not have a native concept of **bold** or *italic*. Instead, each variation is treated as a completely separate font (most operating systems and word-processors work this way too, but shield this fact from the user). This means that to italicize a single word, you need to access two different fonts - which means even larger files when you're using embedded fonts.

So how do you use one of these fonts? In the HelloWorld example earlier, you saw how to access one of the built in fonts. Here's how you use a TrueType font in a document.

**Standard 14 fonts**

- Times
- Times-Italic*
- Times-Bold**
- Times-BoldItalic***
- Helvetica
- Helvetica-Oblique*
- Helvetica-Bold**
- Helvetica-BoldOblique***
- Courier
- Courier-Oblique*
- Courier-Bold**
- Courier-BoldOblique***
- ΑΒβΓγ (Symbol)
- ♠♣♥♦ (ZapfDingbats)

```
1. public void showText(PDFPage page)
2.     throws IOException
3. {
4.     PDFFont myfont = new OpenTypeFont(new FileInputStream("myfont.ttf"), 2);
5.     PDFStyle mystyle = new PDFStyle();
6.     mystyle.setFont(myfont, 11);
7.     page.setStyle(mystyle);
8.     page.drawText("This is a TrueType font", 100, 100);
9. }
```

Example 4 - Using an OpenType font

The only different between the earlier version and this one is line 4.

One of the unique features of this library is that the full range of characters from each font can be used. Traditionally in PDF documents, authors had to choose an *encoding*, which gave them access to a certain number of characters but no more. If your font had characters that weren't in this predefined set, tough. We work around that internally, so that the full range of characters from each font can be accessed easily - essential for languages that use characters outside the basic US-ASCII range. As you would expect for a Java library, we use the Unicode standard to access each character. To print the Euro character (€), you would use a line like `page.drawText("Hello, \u20AC world");` in your code. If the font has the correct symbol, the character will be displayed.

## Formatting Text

Up until version 1.2 of the library, the way to place text on the page was via the `beginText`, `drawText` and `endText` methods, the simplest variation of which you've already seen in the previous examples. While effective, these methods were limited - you couldn't mix text and graphics in a single paragraph, and the only way to calculate the height of a paragraph of text was to draw it first, find out how much space it took then discard it.

To remedy these problems, in version 1.2 the new `LayoutBox` class was added, which gives a lot more control at the expense of perhaps being a little more complicated. As they're quite different to use, we'll cover both techniques separately

### *Simple text formatting using the drawText method*

```
1. public void formatText(PDFPage page)
2. {
3.     PDFStyle plain = new PDFStyle();
4.     plain.setFill(Color.black);
5.     plain.setTextAlign(PDFStyle.TEXTALIGN_JUSTIFY);
6.     plain.setFont(new StandardFont(StandardFont.HELVETICA), 11);
7.
8.     PDFStyle bold = (PDFStyle)plain.clone();
9.     bold.setFont(new StandardFont(StandardFont.HELVETICABOLD), 11);
10.
11.     page.beginText(50,50, page.getWidth()-50, page.getHeight()-50);
12.     page.setStyle(plain);
13.     page.drawText("This text is in ");
14.     page.setStyle(bold);
15.     page.drawText("Helvetica Bold.");
16.     page.endText(false);
17. }
```

*Example 5 - Simple text using the drawText method*

For text layout, the library uses the concept of a *box* - a rectangle on the page containing a block of text. Text within this block can be written in different styles, and can be left or right aligned, centered or justified. You can even mix `TrueType`, `Type 1` and the `Standard` fonts on the same line.

The example above shows some of the method calls used to draw the text on the page. The `beginText` method is the most important, as it defines the rectangle in which to place the text. It takes the X and Y positions of two opposite corners of the rectangle - in this example the entire page is used, less a 50 point margin.

Once `beginText` is called, the `drawText` method can be called as many times as necessary to place the text on the page, interspersed with calls to `setStyle` as required.

To end the text block, call the `endText` method. This takes a single boolean parameter, which determines whether to justify the final line of text *if* the current text alignment is justified. Nine times out of ten, this will be set to false.

## *The End of the Line*

When a line of text hits the right margin, it's wrapped to the next line. Exactly where it is wrapped is open to debate. We've chosen to follow the guidelines set down by the Unicode consortium as closely as possible, which generally speaking means lines are wrapped at spaces or hyphens, but words themselves are not split. Japanese, Chinese and Korean follow a different set of rules, which means that words can be split just about anywhere (with a few exceptions to do with small kana and punctuation). For more control over this, we need to look at using some of the [control characters](#) defined in the Unicode specification.

## *The End of the Page*

When the text box defined by the `beginText` method is full, no further text will be displayed. Instead, it's held in an internal buffer waiting for you to tell the library what to do with it. You can tell when the box is full by checking the return value of the `drawText` method. Normally this returns the number of lines displayed (in points), but if it returns `-1` that indicates that the box is full. However, be warned - one of the major reasons we moved to the `LayoutBox` classes is because measuring text this way wasn't accurate enough for complicated layouts. If your text is going to wrap to the next column or page, we **strongly** recommend you use the `LayoutBox` class instead.

## *Advanced text formatting using the LayoutBox class*

The `LayoutBox` class was added in version 1.2 to remedy several deficiencies with the existing text-layout model. While able to do everything that the `drawText` method can do, it adds several new features:

- mix images and other blocks with the text
- determine the size and position of a phrase in the middle of a paragraph
- determine the height of the text before it's printed to the page
- better layout control when adjusting the font size over a line

First, a simple example, which is almost identical to the last one we demonstrated.

```
1. public void formatText(PDFPage page)
2. {
3.     Locale locale = Locale.getDefault();
4.
5.     PDFStyle plain = new PDFStyle();
6.     plain.setFill(Color.black);
7.     plain.setTextAlign(PDFStyle.TEXTALIGN_JUSTIFY);
8.     plain.setFont(new StandardFont(StandardFont.HELVETICA), 11);
9.
10.    PDFStyle bold = (PDFStyle)plain.clone();
11.    bold.setFont(new StandardFont(StandardFont.HELVETICABOLD), 11);
12.
13.    LayoutBox box = new LayoutBox(page.getWidth()-100);
14.    box.addText("This text is in ", plain, locale);
15.    box.addText("Helvetica Bold.", bold, locale);
16.    page.drawLayoutBox(box, 50, page.getHeight()-50);
17. }
```

*Example 6 - Text using the LayoutBox class*

We say "almost identical" to the previous example, because there's one subtle difference, and that's the *anchor point* for the two different text methods is different. With the `beginText`, the coordinates you give the method are the position of the **baseline** of the first line of text. The first line of text will mostly be displayed above this point. With the `drawLayoutBox` method, the coordinate you give it is the **top-left** corner of the box, equivalent to the top of the first line of text. This allows better control when mixing images or various different size of fonts on the same line.

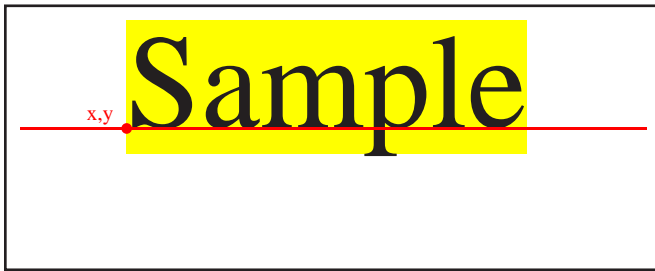


Figure 1a - Positioning using `beginText`



Figure 1b - Positioning using `drawLayoutBox`

## Mixing text and images

Now we've got the basics covered, we'll show you some of the more interesting tricks you can do with the `LayoutBox` class. First, and probably simplest, is placing an image in the middle of the text.

```

1. public void drawBoxImage(PDFPage page, PDFStyle style, PDFImage img, float x, float y) {
2.     LayoutBox box = new LayoutBox(page.getWidth()-100);
3.     float imgwidth = img.getWidth();
4.     float imgheight = img.getHeight();
5.     box.addText("This text is before the image ", style, Locale.getDefault());
6.     LayoutBox.Box ibox = box.addBoxInline(imgwidth, imgheight, PDFStyle.TEXTALIGN_BASELINE);
7.     box.addText(" and this text is after.", style, Locale.getDefault());
8.     page.drawLayoutBox(box, x, y);
9.     page.drawImage(img, x+ibox.getLeft(), y+ibox.getTop(), x+ibox.getRight(), y+ibox.getBottom());
10. }

```

Example 7 - Mixing Text and Images in a `LayoutBox`

As you can see here, we call the `LayoutBox.addBoxInline` method to place a "box" in the middle of the paragraph. This method, like all the other `add...` methods in the `LayoutBox` class, returns a `LayoutBox.Box` object which can be used to determine the position of the rectangle relative to the position of the `LayoutBox`. Here we use those coordinates to draw an image on the page, but it would be just as easy to, say, fill the rectangle with a color. Here's what the output of this example could look like.



Two further things to point out before we move on. First, the last parameter to the `addBoxInline` method controls the *vertical alignment* of the image, relative to the rest of the text. We'll cover more on vertical-alignment in a minute. The second thing is that there's an even easier way to draw an image than the method shown above - use the `LayoutBox.Box.setImage` method. This convenient shortcut saves you from having to call the `drawImage` method yourself, although it's limited to working with images.



## Text positions - using *LayoutBox.Text* class

The next example shows how to use the boxes returned from `addText` to draw a colored background to a piece of text.

```
1. PDFStyle background = new PDFStyle();
2. background.setFill(Color.blue);
3.
4. LayoutBox box = new LayoutBox(width);
5. box.addText("The following phrase will be drawn on ", style, Locale.getDefault());
6. LayoutBox.Text text = box.addText("a blue background", style, Locale.getDefault());
7. box.addText(", but now we're back to normal.", style, Locale.getDefault());
8. box.flush(); // Important, or the last line may not be positioned!
9.
10. page.setStyle(background);
11. do {
12.   page.drawRectangle(x+text.getLeft(), y+text.getTop(), x+text.getRight(), y+text.getBottom());
13.   text=text.getNextTwin();
14. } while (text != null);
15. page.drawLayoutBox(box, x, y);
```

Example 8 - Setting the background behind the text

If you take a close look at this example you'll see it's almost identical to the previous example, with the exception of the `do/while` loop. Why is it there? The answer lies in how text is positioned in the `LayoutBox`. When you add a piece of text to the box, unless you call the `addTextNoBreak` method or otherwise have good reason to assume the text won't be split over more than one line, you have to allow for this possibility. The `LayoutBox.Text.getNextTwin()` method allows you to cycle through the one or more `LayoutBox.Text` objects which represent the phrase of text on the page, until the method returns `NULL` - indicating all the boxes have been returned. If you're using the `LayoutBox` class and want to turn some of the text into hyperlinks, this is the way to do it.

## Vertical positioning in a *LayoutBox* line

Prior to version 1.2 there was only very basic support for mixing different sized text on a single line. The `LayoutBox` class adds full, Cascading Style-Sheet style support for vertical alignment. When mixing elements of different height on the page, you need to be aware of what options are available to help with positioning them.

Before we continue, there are two definitions we need to make. The **Text Box** is a box equivalent to the size of the text itself. This may be the same as or smaller than the **Line Box**, which is the box equivalent to the size of the entire line. A line box is always sized so that it fits the largest text box in the line. In the example below, the line box is in yellow, the larger text-box is in green and the smaller of the two text-boxes is shown in orange.

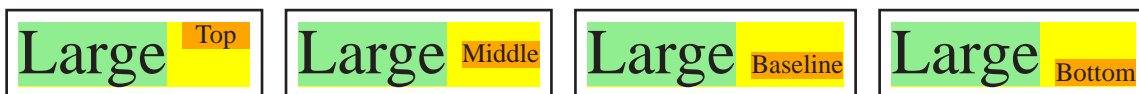


Figure 2 - Vertical alignment examples

This example shows the four different options for vertical alignment within a line box. **Top** places the top of the text box at the top of the line box. **Middle** places the middle of the text box at the middle of the line box. **Baseline**, the default, places the baseline of the text box at the baseline of the line box. Finally, **bottom** places the bottom of the text box at the bottom of the line box.

The same rule applies to boxes added using the `addBoxInline` method. In the [image example](#) above we use the `PDFStyle.TEXTALIGN_BASELINE` method to place the image, although as the image was the largest item on the line it had no effect. If the box representing the image is smaller than the line box however, the last parameter to `addBoxInline` has the same sort of effect as with the text demonstrations above.

The height of each text box depends on both the size of the font used, and it's *leading* - white space between lines. Each font has a preferred leading, which is set by the font author. This can be retrieved by the `PDFFont.getDefaultLeading` method. The line height can then be set as a multiple of this value by calling the `PDFStyle.setTextLineSpacing` method. Passing in a value of `1.0` results in each line of text using the default leading, whereas a value of `2.0` would double it, and so on. Leading is always evenly distributed, half above and half below the text.

## Floating boxes

You've seen how to add images in the middle of a paragraph, but there's one more common type of placement - known as *float positioning*. This allows a paragraph of text to wrap around a rectangle. We've already used this style of positioning a few times in this document - see the [Standard Fonts note](#) on page 11 for an example.

Boxes can be "floated" to the left or right of a paragraph - simply call the `addBoxLeft` or `addBoxRight` method to choose which. The top of the box will be placed either on the current line or the first clear line, depending on the clear flags (more in a minute), and any further text will wrap around it until the text grows beyond the height of the box. This is easier to describe with an example.

```
1. PDFStyle background = new PDFStyle();
2. background.setFill(Color.blue);
3.
4. LayoutBox box = new LayoutBox(width);
5. box.addText("The following text will be drawn around ", style, Locale.getDefault());
6. LayoutBox.Box ibox = box.addBoxRight(100, 50, LayoutBox.CLEAR_NONE);
7. box.addText("around the box to the right. When it grows beyond that box\
    it will automatically fill the width of the line", style, Locale.getDefault());
8.
9. page.drawLayoutBox(box, x, y);
10. page.drawRectangle(x+ibox.getLeft(), y+ibox.getTop(), x+ibox.getRight(), y+ibox.getBottom());
```

Example 9 - Text floating around a box to the right

“The following text will be drawn around the box to the right. When it grows beyond that box, it will automatically fill the width of the line.”




Figure 3 - Text flowing around a floating box

The `CLEAR_NONE` flag effectively says "it doesn't matter if another box is already floating to the right - in that case, place this box next to it" - although in this example there's only one floating box, so it has no measurable effect. The other alternative is to say that this box *must* be placed flush on the right margin - if another floating box is already there, it will position itself below that one. There are all sorts of variations on this theme, and we won't describe all of them. Instead, we'll leave you with an example showing the sort of layout that this class is capable of.

```
1. box.addBoxRight(50, 50, LayoutBox.CLEAR_RIGHT);
2. box.addBoxRight(50, 50, LayoutBox.CLEAR_RIGHT);
3. box.addBoxRight(50, 50, LayoutBox.CLEAR_NONE);
4. box.addText("Text text ...", mystyle, locale);
5. box.addBoxLeft(50, 50, LayoutBox.CLEAR_NONE);
6. box.addBoxLeft(50, 50, LayoutBox.CLEAR_NONE);
7. box.addBoxLeft(50, 50, LayoutBox.CLEAR_NONE);
```



```
8. box.addBoxLeft(50, 50, LayoutBox.CLEAR_LEFT);
9. box.addText("More more ...", mystyle, locale);
```

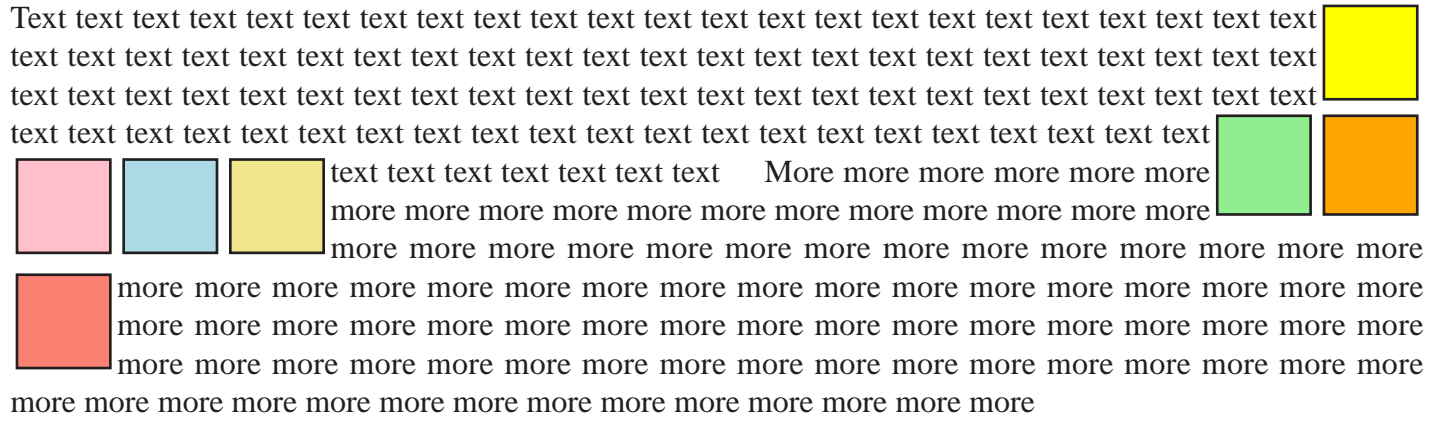


Figure 4 - Things you can do with the "clear" flag

# Advanced text layout features

## Quotes

The `requote` method in the `PDFFont` class can be used to substitute the normal single (') and double (") quote characters into nicer looking glyphs - turning "a test" into "a test", „ein test“, ”en prøve” or whatever is appropriate for the specified locale.



## Kerning, Spacing and Ligatures

Most good quality fonts have a *kerning table*, which allows for the spaces between letters to be adjusted for improved appearance - as you can see, a sequence of A's and W's without kerning look too far apart. Another way to improve the appearance of text is with ligatures - a single shape representing two or more joined characters. They're not too common in Latin scripts, but in languages like Arabic they're essential. Throughout the library, variable-width fonts have kerning and ligatures applied automatically, unless they're inhibited using the Unicode Control character "zero-width non-joiner".

For control over spacing between letters, there are several options. First is the method we'd generally recommend because we feel it gives the best results - setting the

*justification ratio*.

The `setJustificationRatio` method of the `PDFStyle` class allows you to choose where to add the extra whitespace required to justify a short line of text. A value of 0 means extend spaces between words, for a result like **THREE SHORT WORDS**, whereas a value of 1 means extend the spaces between letters like this: **THREE SHORT WORDS**. The default is 0.5, i.e. somewhere in the middle, and gives good results (this document is formatted using the default value). Kashida justification, required for justifying Arabic script, is not supported in this release.

Another option is track kerning, which allows you to manually squeeze or expand the space between letters. This is done using the `setTrackKerning` method in the `PDFStyle` class. The effect is equivalent to moving every letter together or apart by a fixed amount.

The final option is to use the Unicode spacing characters - U+2000 to U+200A. These only work with the proportional Standard Fonts, i.e. Helvetica and Times. These spacing glyphs are legacies of the pre-electronic typesetting days, when spacing was done manually by inserting small spacing blocks between each letter on the press. Although not as useful today, they are an option which some users may prefer. Simply insert them between letters in the same way as a normal space character.

## ***Text measurement and positioning***

In some situations it's required that the size of the text is known beforehand. This is easily done with the `getTextLength` method and friends from the `PDFStyle` class. They return the exact left, right, top and bottom of the specified line of text in points, so the size of the text is known beforehand. For even more precision, the methods like `getAscender` in the `PDFFont` class allow you to adjust the position of the text accordingly.

## **International support**

For many of the world's languages, provided an appropriate font is used the library will work out-of-the-box. The standard 14 fonts cover (to the best of our knowledge) English, French, German, Portuguese, Italian, Spanish, Dutch (no "ij" ligature"), Danish, Swedish, Norwegian, Icelandic, Finnish, Polish, Croatian, Czech, Hungarian, Romanian, Slovak, Slovenian, Latvian, Lithuanian, Estonian, Turkish, Catalan (although the "L with dot" character is missing), Basque, Albanian, Rhaeto-Romance, Sorbian, Faroese, Irish, Scottish, Afrikaans, Swahili, Frisian, Galician, Indonesian/Malay and Tagalog. With an appropriate embedded font, most other languages will work.

Arabic, Hebrew and Yiddish are fully supported as of version 1.1 of the library, which implements the full Unicode bidirectional algorithm version 3.1 - although the complete range of Arabic ligatures is not yet implemented, the number that are there should cover most cases. The Unicode directional override characters are supported as per that specification.

Japanese, Chinese (simplified and traditional) and Korean are also supported as of version 1.1, although in horizontal writing mode only. To display text in these languages, an OpenType font can be embedded with 2 bytes per glyph. This will probably result in a very large document however, as several hundred glyphs would typically be used. There is another option - Adobe have defined several "standard" fonts which are available with the localized versions of its viewers, or as separate language packs downloadable from their website.

These fonts can be used via the `StandardCJKFont` class, which is similar to the `StandardFont` class for latin-based scripts - although if the viewer doesn't have the appropriate font installed, all they'll see is a request to download it. Line breaking for these languages follows the recommendations laid out in the Unicode 3.1 specification, but the zero-width space characters can be used to override this. Since version 1.2.1 the Hong Kong Supplementary Character Set (HKSCS) is supported in the `MSung` font, although you'll need Acrobat 5.0 or later to display these.

Acrobat 6.0 and Java 1.5 both introduced the ability to go beyond the Basic Multilingual Plane (BMP). This is done using UTF-16, which uses *low and high surrogates* and so does not require the basic "character" to be expanded beyond 16 bits. This is fully supported in the PDF library since 2.2.6. Additionally, Java 1.5 is *not* required - using Java 1.3 it is quite possible to create a String containing the appropriate low and high surrogate characters, create a PDF from it using an appropriate font and have it display correctly in Acrobat 6.0. National standards using this range include HKSCS-2001 and JISX0213:2004, both of which are supported in Acrobat 6.0 with the "MSung" and "HeiSeiMin" fonts.

A word on GB18030-2000. This is the standard required by the Chinese Government, and covers latin, simplified and traditional Chinese, Japanese, Korean, cyrillic and Arabic scripts, as well as less well known ones like, Yi, Mongolian and Uygur. The PDF library *can* support this standard, although not with one font (as you might expect). Our suggestion for those requiring GB18030-2000 support is to use the `PDFStyle.addBackupFont` method to create a style with a primary font and a number of backups to be used as required. Typically you would use the `StandardFont` Times, the `StandardCJKFonts` STSong, MSung, HeiSeiMin and HYSMyeongJo, then would have one or more OpenType fonts covering Arabic, cyrillic and the lesser used chinese dialects.

The following languages have poor or non-existent support.

- Thai, Khmer and Myanmar should work fine, but the rules for line-breaking require analysis of the words being printed, which we can't do. This means that line-breaking will have to be done manually, by inserting spaces or zero-width spaces where appropriate.
- We haven't worked on the Devanagari scripts - Hindi, Bengali and friends - owing to the difficulty in finding a decent font with the correct Unicode encoding, and because we don't know anyone that writes it well enough to tell us whether we've got it right or not.
- Urdu, with it's right-to-left diagonal baseline, is unlikely to be supported in the near future, although if a horizontal baseline is acceptable it should work fine.
- Mongolian and Tibetan both have unusual line-breaking rules, which we haven't attempted to implement. This can be done manually like for Thai, above. Vertical display of traditional Mongolian text is not supported, nor is it likely to be.
- Any other languages requiring ligatures to display correctly (other than Arabic and Armenian) are not currently supported, probably because we don't know about them.

The `setLocale` method of the `PDF` class is important with multi-lingual documents, as it determines the primary text direction of a document, the default text alignment (right-aligned for Arabic and Hebrew locales), the style of quote-substitution to use and various other aspects of the display. It defaults to the system `Locale`, so generally it's already correct - but just in case it can be set and re-set as many times as needed. The locale being used when the `PDF` is written is considered to be the locale of the document as a whole, and the language flag in the `PDF` is set accordingly.

### ***Is this character available?***

How do you know if a certain character is available in the font you're using? One way is to create the document, and look for a line resembling `WARNING: Skipping unknown character '?' (0x530) printed to System.err.` Of course, by the time you've got this information it's too late, so a better way is to call the `isDefined` method of the `PDFFont` you're using, which returns true if the character is defined.

Something to remember if you're using non-embedded fonts (particularly the `StandardCJKFont` fonts) is that just because a character is defined on your local copy of the font, doesn't mean that the character is defined in the version on the viewers machine.

## **Unicode**

The Unicode specification is a universal encoding for every character in every language. It's a work in progress, which means that new characters are being added all the time. All Java strings are automatically in Unicode, and the library supports it too - which means it's easy to mix characters from many languages in a single phrase.

Any character from the entire Unicode code range can be added to the document, without having to worry about codepages, encodings and various other difficulties common when using Unicode in PDF - the library takes care of it all internally, and provided the font has the character defined, it will be displayed.

As well as defining a list of characters, the Unicode specification goes further and defines various rules for layout of text, such as the bi-directional algorithm (how to handle mixed left-to-right and right-to-left text on a single line), rules for when to add a line break, and so on. Some of the code-points in the specification are control characters which affect these algorithms, in the same way that the ASCII code 9 means "horizontal tab".

The library supports most of these control codes, which can be included in any text displayed on the page to control exactly how the page is laid out.

U+00A0	Non-Breaking Space	The same as a regular space character, but prevents the words on either side being separated by a line break
U+00AD	Soft Hyphen	Inserted into a word to indicate that the word <i>may</i> be split at that point. If the word is actually split, a hyphen is displayed, otherwise this character is invisible
U+200B	Zero-Width Space	Inserted into a word to indicate that the word <i>may</i> be split at that point. Regardless of whether the word is split or not, this character is invisible
U+2011	Non-Breaking Hyphen	Identical to a hyphen, but prevents the characters on either side from being split.
U+FEFF	Zero-Width Non-Breaking Space	Inserted into a word to prevent the word being split or hyphenated at that point, or around characters which would otherwise be a potential break-point.
U+200C	Zero-Width Non-Joiner	Inserted into a word to indicate that no ligature should be formed between these two letters. Mostly used in Arabic language scripts, we also use it to prevent the "f" and the "i" joining as a ligature in the example on the previous page, and to prevent kerning.
U+200D	Zero-Width Joiner	Inserted into a word to indicate that the characters <i>should</i> be replaced with a ligature. Currently has no effect.
U+200E, U+200F, U+202A - U+202E	Directional Indicators	Inserted into a phrase to control text direction for bidirectional text. These function as described in the Unicode bidirectional algorithm.
U+2044	Fraction Slash	Not strictly a control character, when placed between two numbers this slash results in the appropriate fraction being substituted - so <code>page.drawText("1\u20442")</code> displays as "1/2". In version 1.1 this will only work if the appropriate fraction is defined in the font.
U+2028, U+2029	Line and Paragraph Separators	Unicode's attempt at solving the age-old "CR+LF, CR or LF" problem was to add two new characters which are to be unambiguous in their meaning. There is no advantage to using these - the library regards both of these as a normal newline (whatever is produced by "\n" on your system).

*Table 3 - Unicode control characters*

# Graphics

## Bitmap Images

```
1. public void showImage(PDFPage page)
2.     throws IOException
3. {
4.     PDFImage img = new PDFImage(new FileInputStream("myimage.jpg"));
5.     float width = img.getWidth();
6.     float height = img.getHeight();
7.     page.drawImage(img, 100, 100, width, height);
8. }
```

*Example 10 - Displaying an image*

Adding bitmap images to the document can be done with a couple of lines - the first loads the image, the second places it on the page. Images can be read directly from a file (the library can parse JPEG, PNG, GIF, TIFF and PNM images) or loaded from a `java.awt.Image`. This gives enormous flexibility - bitmaps can easily be created using normal `java.awt` methods, or an extension library like [Sun's Jimi library](#) can be used to load PCX images, Adobe Photoshop™ files and many other formats.

When embedding a bitmap image, there are a few points you should remember.

1. Transparency is poorly supported in PDF prior to Acrobat 5.0 - earlier versions were limited to "masked" transparency (as used in GIF and 8-bit PNG images) where a single color can be flagged as transparent. Additionally, PostScript limitations in Acrobat 4.0 and earlier may cause large images to be rendered without transparency when printed in Acrobat 4.0 - see the API documentation for more info.
2. Most computer monitors have a resolution of 72 (Windows) or 96 (Macintosh) dots-per-inch, which means a 200x200 pixel bitmap will take between 2 and 2.8 inches on the screen. When printing an image to a high resolution printer however, the minimum you can get away with is probably around 200dpi for a color image and 300dpi for black and white, otherwise the image is going to start to get "blocky". Depending on the type of image you're embedding you should generally use bitmaps *at least* 3 times the size you would use for on-screen viewing if you want to print the PDF.

Here's an example of what we mean. The first logo is embedded at 200 dpi, the second is at 72dpi. Print the document out or zoom in for a closer look, and see the difference.



The DPI of the image in the document is the number of inches the bitmap takes up, divided by the number of pixels in the bitmap. So a 200 pixel wide bitmap sized to take up 72 points (1 inch) in the document has a resolution of 200dpi. All image formats except GIF can specify the DPI of the image (if it's not specified it defaults to 72dpi), so the example above displays the image at the point-size the artist intended. For more control, it's possible to extract the X and Y resolution of the image using the `getDPIX()` and `getDPIY()` methods.

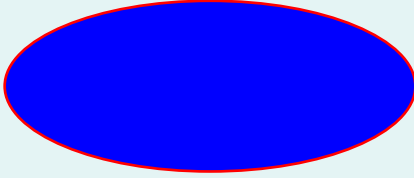
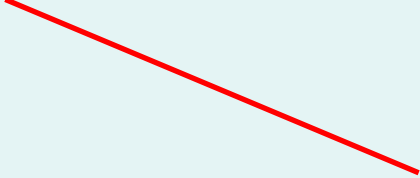



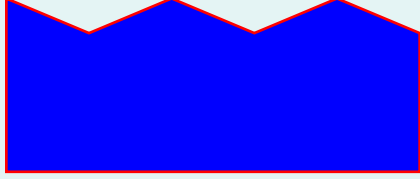
## Vector Graphics

It's also possible to draw "line-art" style images directly into the PDF. Here's an example that draws a circle of radius 100 in the center of the page, filled with blue stars on a red background and with a black border.

```
1. public void drawCircle(PDFPage page)
2. {
3.     PDFStyle style = new PDFStyle();
4.     style.setLineColor(Color.black);
5.     style.setFillColor(new PDFPattern("Star",0,0,50,50,Color.red,Color.blue));
6.     page.setStyle(style);
7.
8.     page.drawCircle(page.getWidth()/2, page.getHeight()/2, 100);
9. }
```

*Example 11 - Drawing a circle*

Not terribly difficult. As well as drawing circles, there are several other shapes that can be easily drawn.

drawCircle / drawEllipse	drawLine	drawArc
		
drawRectangle	drawRoundedRectangle	drawPolygon
		

Remember that every shape except the lines or the arcs can be drawn as outlines, solid or both - it depends on whether a fill, a line color or both is specified in the current style. If these shapes aren't enough, the more primitive `path` methods allow you to assemble complex shapes yourself using individual elements. This example draws a rectangle, with the top of the rectangle replaced by a wavy line.

```
1. public void drawPath(PDFPage page)
2. {
3.     PDFStyle style = new PDFStyle();
4.     style.setLineColor(Color.black);
5.     style.setFill(Color.blue);
6.     page.setStyle(style);
7.
8.     page.pathMove(100, 100);           // Always start with pathMove
9.     page.pathLine(100, 200);
10.    page.pathBezier(130,300, 160,100, 200, 200);
11.    page.pathLine(200, 100);
12.    page.pathClose();
13.    page.pathPaint();
14. }
```

*Example 12 - Drawing a complex shape with path operations*

The `pathLine`, `pathBezier` and `pathArc` methods are available to build up your shape. The only things to remember when drawing a shape with the `path` operators is to start with `pathMove` and end with `pathPaint`.

## **Graphics state: Transforming the page**

There are a few additional methods in the `PDFPage` class which can be used to alter the page itself. The `setUnits` method has already been mentioned, allowing you to redefine which corner of the page is (0,0) and which units you want to measure the page in. More interestingly, the `rotate` method can rotate the coordinates of a PDF page around a specified point. Take a look at the watermark on each page of this document for an example.

Other useful functions do with with the graphics state are `save` and `restore`. These allow you to save the current state of the page to a "stack" - it's good practice to save before doing a transformation and restore afterwards rather than apply a reverse transformation. For example, here's how to draw some text on the page at 45° without affecting all future operations as well.


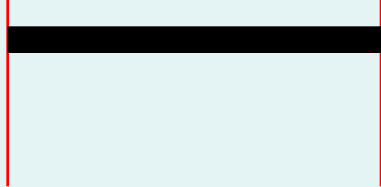
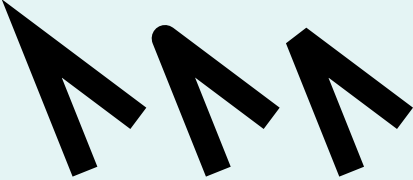
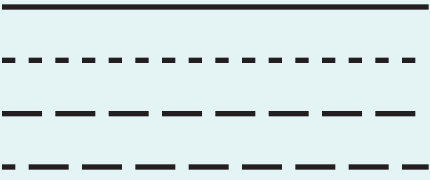
```
1. public void drawRotatedText(PDFPage page, String text, float x, float y)
2. {
3.     page.save();
4.     page.rotate(x, y, 45);
6.     page.drawText("Hello, World", x, y);
5.     page.restore();
7. }
```

*Example 13 - the Rotate, Save and Restore methods*

## **Graphics styles**

Various aspects of the `PDFStyle` class can be used to alter the look of the various graphics methods (although they also work on text, they're more likely to be used with graphics elements).



	<p>The <code>setLineWidth</code> method sets the width of any lines that are drawn, including outlined text and shapes. The width can be any positive number, and is specified in points. Setting a line width of zero is allowed and instructs the viewing device to create the <i>thinnest line possible on that device</i>. In Acrobat Viewer, this is 1 pixel wide, regardless of the zoom level of the document, while on a high resolution printer this may be nearly invisible. Because of its device dependent nature, this is not recommended by Adobe.</p>
	<p>The <code>setLineCap</code> method changes how the end of lines are drawn. The default option is <i>Butt cap</i>, where the stroke is squared off at the endpoint of the path and there is no projection beyond the end. Second is <i>Round cap</i>, causing a semicircle with a diameter equal to the line width to be drawn at the end of the line. Finally the <i>Square cap</i>, where the stroke continues beyond the endpoint of the path for a distance equal to half the line width, and is then squared off.</p>
	<p>The <code>setLineJoin</code> method changes how two line segments are joined. The default is a <i>Miter join</i>, where the outer edges of the two strokes are extended until they meet at an angle, as in a picture frame. For extremely sharp angles a bevel join is used instead. Second is a <i>Round join</i>, where a circle with a diameter equal to the line width is drawn around the point where the two segments meet and is filled in, producing a rounded corner. Finally, the <i>Bevel join</i>, where the two segments are finished with butt caps and the resulting notch is filled with a triangle.</p>
	<p>The <code>setLineDash</code> method allows you to change the line into a sequence of <i>on</i> and <i>off</i> line segments. The default is a solid line, the result of calling <code>setLineDash(0,0,0)</code>. The second example here is a sequence of 5 points on, 5 points off, from <code>setLineDash(5,5,0)</code>. The third example is 15 points on, 5 points off, from <code>setLineDash(15,5,0)</code>. The final example is the same as number 3, but the <i>phase</i> of the pattern has been changed by calling <code>setLineDash(15,5,10)</code>. The pattern of 15 on, 5 off, 15 on, 5 off starts ten pixels in, and the result is 5 on, 5 off, 15 on, 5 off, 15 on and so on.</p>

## Section 2. Importing and Editing existing documents

In version 1.1.12 we added the `PDFReader` class, which allows existing PDF documents to be read, modified and written. This framework has been completely overhauled for version 2.0, and is now much more stable and about twice as fast. The `PDFReader` class, which is required to read an existing document, is part of the *Extended Edition* of the library - you'll need to remember this when deciding which version to purchase.

One of the most common uses for the Extended PDF library is filling a blank form with data, possibly concatenating all the results together and then sending the document either back to the browser or to some other `OutputStream`. A read through the following section will be helpful to anyone planning to implement something like this.



So how do you import and edit a PDF? Let's take a fairly typical example first - the concatenation of several documents into one.

```
1. import java.io.*;
2. import org.faceless.pdf2.*;
3.
4.
5. public class Concatenate
6. {
7.     public static void main(String[] args) throws IOException
8.     {
9.         PDF out = new PDF();
10.        for (int i=0;i<args.length;i++) {
11.            FileInputStream in = new FileInputStream(args[i]);
12.            PDFReader reader = new PDFReader(in);
13.            in.close();
14.            PDF pdf = new PDF(reader);
15.
16.            out.getPages().addAll(pdf.getPages());
17.            out.getForm().getElements().putAll(pdf.getForm().getElements());
18.        }
19.        out.render(new FileOutputStream("Concatenated.pdf"));
20.    }
21. }
```

*Example 14 - Concatenating several documents*

Significant lines here include line 12 and 14, where the next PDF is read in from an `InputStream` (in this case a `FileInputStream`, but they could be read in from a database, a URL, a `ByteArrayInputStream` or similar - any `InputStream` will do).

The next interesting bit is line 16, where all the pages are **moved**, not copied, from the PDF we just read in to our output PDF. This is done by manipulating the list of pages returned by the `getPages` method in the `PDF` class. This returns a standard `java.util.List`, which can be manipulated in any of the usual ways. Line 17 moves the form fields across too - more on this [later](#).

## ***The trouble with page moves***

The ability to move pages from one PDF to another also doesn't come freely. Although the library takes care of a lot of the "dirty work", the programmer must also be aware of what he does when it comes to links within the document. Imagine a document containing two pages, with page 1 containing a hyperlink to page 2. If page 2 is deleted, a warning will be printed to `System.err` and the hyperlink action removed. The same applies to bookmarks linking to that page (which are *not* copied when a page is copied), and form fields, which may exist on more than one page at a time and therefore *are not* transferred when moving pages (although they can be moved separately). Here's a list of things to watch for when manipulating a document's pages.

- A page can only ever be in one PDF at a time, and can only exist in a PDF once. If you want to copy a page, rather than move it, you should clone it first, like so:

```
PDFPage copy = new PDFPage(original);
newpdf.getPages().add(copy);
```

- If you're cloning all the pages in the document, why not clone the entire document instead? This is almost certainly going to be faster, and gets around several nasty potential problems with form fields: Try something like this:

```
1. PDF template = new PDF(new PDFReader(new FileInputStream("Template.pdf")));
2. PDF out = new PDF();
3. for (int i=0;i<10;i++) {
4.     PDF copy = new PDF(template);
5.     out.getPages().addAll(copy.getPages()); // out now has 10 copies of "Template.pdf"
6. }
```

*Example 15 - Cloning a document*

- Manipulating documents containing Form Fields can cause confusion - so much so that we've given it its own section.

## **Stamping documents**

There are some other useful things you can do with a document that's been read in. First and most obvious, you can write to it's pages in the same way as with a newly created document. For instance, to add a message or watermark to every page:

```
1. PDFStyle style = new PDFStyle();
2. style.setFont(new StandardFont(StandardFont.HELVETICA), 12);
3. style.setFill(Color.black);
4.
5. PDF pdf = new PDF(new PDFReader(inputstream));
6.
7. for (int i=0;i<pdf.getPages().size();i++) {
8.     PDFPage page = pdf.getPage(i);
9.     page.setStyle(style);
10.    page.drawText("Received "+new Date(), 10, 10);
11. }
12.
13. pdf.render(outputstream);
```

*Example 16 - Stamping each page of a document*

Power users might like to try using translucent colors and the `rotate` method to stamp text on top of the page. Another alternative would be to create and attach a new Stamp Annotation for each page. All of these options are demonstrated in the `Stamp.java` example included in the examples supplied with this library.

These methods work fine with Acrobat 5.0, but earlier versions don't support transparency and so would result in text being obscured. One method we found quite effective is to write the stamp *under* the current contents of the page, rather than over top. This is done with the `seekStart` method, which moves the page "cursor" to the start of the pages contents. This method isn't bullet-proof - some documents clear the background first, which will overwrite the stamp - but it works in many cases. Here's the above example with two new lines (9 and 12) added to draw the stamp under any current content on the page.

```
1 PDFStyle style = new PDFStyle();
2 style.setFont(new StandardFont(StandardFont.HELVETICA), 12);
3 style.setFill(Color.black);
4.
5. PDF pdf = new PDF(new PDFReader(inputstream));
6.
7. for (int i=0;i<pdf.getPages().size();i++) {
8.     PDFPage page = pdf.getPage(i);
9.     page.seekStart();           // Move to start of page stream
10.    page.setStyle(style);
11.    page.drawText("Received "+new Date(), 10, 10);
12.    page.seekEnd();           // Move back to the end.
13. }
14.
15. pdf.render(outputstream);
```

*Example 17 - Stamping each page under the current content*

## **Stamping Whole Pages - a Page is a Canvas**

Another possibility is to use the contents of a page as an image or "Canvas", to be painted onto another page. We visited canvases briefly in the [Patterns](#) section, but essentially a Canvas is like it's equivalent in the art world; a sheet of virtual paper with markings on it. A page is simply a canvas with optional annotations on top.

This allows for some interesting ideas. Rather than cloning a page, you could simply turn it into a canvas and "draw" it onto another page, as you would an image. The effect is visually similar to cloning a page, although the concepts are quite different. Which method you choose depends on what your needs are.

On the one hand, when converting to a PDFCanvas you will lose any form fields and annotations on the page - only the fixed contents of the page will be taken. On the other hand, by treating a page as an image and repeatedly drawing it over and over, the image-page only needs to be saved in the PDF once. For documents that contain one or two pages repeated over and over, this can produce much smaller files. Here's an example of how you might do this - we write some information from an abstract set of records onto a sequence of pages:

```
1. PDFStyle style = new PDFStyle();
2. style.setFont(new StandardFont(StandardFont.HELVETICA), 12);
3. style.setFillColor(Color.black);
4.
5. PDF templatepdf = new PDF(new PDFReader(inputstream));
6. PDFCanvas template = new PDFCanvas(templatepdf.getPage(0));
7.
8. PDF pdf = new PDF();
9. while (hasMoreRecords()) {
10.     PDFPage page = pdf.newPage("A4");
11.     page.drawCanvas(template, 0, 0, page.getWidth(), page.getHeight());
12.     page.setStyle(style);
13.     page.drawText(record.getName(), 100, 100);
14.     page.drawText(record.getAddress(), 200, 100);
15. }
14.
15. pdf.render(outputstream);
```

*Example 18 - Using a page as a template canvas*

One thing to note here is that the canvas and text are both being drawn onto the page, rather than form fields being filled out, which is obviously a lot easier. We can't use a form in this case, simply because a canvas doesn't have annotations. Of course, there's nothing to stop you going through the original form in `templatepdf`, and finding out where the fields are positioned by checking their annotations, and writing the text to the same position in the new PDF... but we'll leave that as an exercise for the reader.

## **Completing Forms**

We'll cover forms in more detail [elsewhere](#), but we want to add a little bit here on batch completion of forms - the focus here is not so much on creating form fields and how they work, but on various "Gotchas" that can occur when trying to complete large numbers of forms in a batch.

First, some things you should know. Each PDF can have only one form (unlike HTML which can have many forms). Each field has a name, which must be unique across the form (and therefore, the document). Somewhat confusingly, a field may have more than one visible appearance in the document - each represented by a special type of Annotation called a "Widget". Although each may have a different style and be on a different page, they must all have the same value. Finally, forms take up a lot of space in the document.

With that in mind, let's look at some typical situations and the things that can go wrong. Lets say you have a template PDF with a form. This needs to be completed for each customer and the results concatenated together into one large document. Here's a way to do it that's clear, simple and *wrong*.

```
1. PDF template = new PDF(new PDFReader(templatestream));
2. PDF out = new PDF();
3.
4. while ((record=nextRecord())!=NULL) {
5.     PDF clone = new PDF(template);
6.     Form form = clone.getForm();
6.     ((FormText)form.getElement("Name")).setValue(record.getName());
7.     ((FormText)form.getElement("Phone")).setValue(record.getPhone());
8.
9.     out.getPages().addAll(clone.getPages());
10.    out.getForm().getElements().putAll(form.getElements());
11. }
```

*Example 19 - The incorrect way to concatenate documents with forms*

The idea here is fairly simple. For each record we clone the template (line 5), complete the form (line 6 and 7), then move all it's pages (on line 9) and all it's form elements (on line 10) to the output PDF. The problem is that after the first record has been added, the out PDF will already contain a FormElement called "Name". Trying to add another one will cause an error.

There are two ways around this. The first method is to flatten the form. Flattening a form permanently stamps all it's elements onto the page, preventing them from being altered, but it does have the pleasing side effect of drastically reducing the resulting file size. This would be done in the example above by adding the line `form.flatten()` at line 8. We could also remove line 10 altogether, as once a form is flattened there are no fields left to move.

This solution is the one we'd generally recommend. However, if you need the combined form to be editable, your only option is to rename the form elements so that each one has a unique name. The easiest way to do this is to call the `Form.renameAll` method - perhaps adding at line 8 something like `form.renameAll("", "_" + (recordnum++))`; This would add a unique suffix to each element - "\_1" for the first record, "\_2" for the second and so on.

## Interaction: Actions, Hyperlinks and Annotations

### **Actions**

The `PDFAction` class allows the user to interact with the document. Actions are used in several places throughout a PDF.

- Each bookmark uses an action to determine what happens when it's clicked on
- An action can optionally be run when a document is opened, closed, printed or saved, by calling the `PDF.setAction` method
- An action can optionally be run when a page is opened or closed, by calling the `PDFPage.setAction` methods.
- `AnnotationLink` annotations have an action associated with them, which runs when they're clicked on.
- A form element and it's Widgets can have actions associated with them, to do things when a field receives focus, when a key is pressed, it's value changed, and so on.

A number of different types of action are provided for in the `PDFAction` class. The various `goTo` actions allow the user to navigate to a specific page in the document, and different variations exist to bring the page up zoomed to fit, or to have a

specific rectangle visible. New in version 1.1 were the `goToURL` action (which is used in the hyperlink examples below), the `playSound` action which should be fairly obvious, and one other which isn't - the named action.

This last one will probably only work under Adobes own Acrobat viewer, but allows a measure of interaction with the viewer itself. A named action corresponds to selecting an action from the drop-down menus in Acrobat, and allow the user to print the document, quit the browser, search for text and so on. This is poorly documented in the PDF specification, but what information we have is available in the API documentation for this method. If you're trying to recreate a specific action that you've seen in Acrobat, what we suggest is create a document in Acrobat with that action, then run it through the `Dump.java` example to see what it's called.

Six other action types are mostly used with forms - the `formSubmit`, `formReset`, `formImportData`, `formJavaScript`, `showWidget` and `hideWidget`. These are usually applied to a form elements annotation via the `WidgetAnnotation.setAction` and `FormElement.setAction` methods, and are covered separately in the section on forms.

## Hyperlinks

A special subclass of annotation is the `Hyperlink`, represented by the `AnnotationLink` class. Like all annotations, these sit *above* the page content, rather than being a part of it.

An `AnnotationLink` annotation takes a `PDFAction`, which can be any of the actions available with the library - a link to a URL or another part of the document, play a sound, print the document and so on.

Positioning a link to match some text placed on the page means knowing where the exactly text is. There are two ways to do this. The first, for those using the `beginText` and `endText`, is to call the `beginTextLink` and `endTextLink` methods in the `PDFPage` class. The second is to get the coordinates of the text or object you're trying to make a hyperlink, and use those to set the rectangle of the `Annotation` by calling it's `setRectangle` method. The coordinates are easily obtained by getting the rectangle of the `LayoutBox.Box` used to position the text. For example:

```
1. LayoutBox box = new LayoutBox(100);
2. box.addText("some text here", style, Locale.getDefault());
3. LayoutBox.Text linktext = box.addText("hyperlink goes here", style, Locale.getDefault());
4. box.addText("some more text here", style, Locale.getDefault());
5. box.flush();
6.
7. page.drawLayoutBox(box, 200, 300);
8.
9. do {
10.     AnnotationLink link = new AnnotationLink();
11.     link.setRectangle(200+linktext.getLeft(), 300+linktext.getBottom(),
10.                      200+linktext.getRight(), 300+linktext.getTop());
11.     page.getAnnotations().add(link);
12.     linktext = linktext.getNextTwin();
13. } while (linktext!=NULL);
```

*Example 20 - Creating Hyperlinks over a `LayoutBox.Box`*

Key things to note here are that we have to loop around the text boxes returned by `getNextTwin`, because it's possible that the text we want to make a hyperlink was split over two or more lines. Also we need to add the coordinates of the `LayoutBox` to the hyperlink, as the coordinates returned by `getLeft()` and friends are relative to the `LayoutBox`. For creating hyperlinks over images and so on, it's a lot easier - just specify the coordinates directly in the `AnnotationLink.setRectangle` method call.

The second method is to use the `beginTextLink` and `endTextLink` methods of the `PDFPage` class to add hyperlinks in the middle of a line of text. Here's how:

```
1. public void showTextLink(PDFPage page)
2. {
3.     PDFAction action = PDFAction.goToURL(new URL("http://bfo.com"));
4.
5.     page.beginText(50,50, page.getWidth()-50, page.getHeight()-50);
6.     page.drawText("Thank you for choosing ");
7.     page.beginTextLink(action, PDFStyle.LINKSTYLE);
8.     page.drawText("the Big Faceless PDF Library");
9.     page.endTextLink();
10.    page.endText(false);
11. }
```

*Example 21 - Creating Hyperlinks using beginTextLink/endTextLink*

The `beginTextLink` method takes two parameters, the first a `PDFAction`, as described above, and the second a `PDFStyle`. The style is optional (it can be left `NULL` for no effect), but is a convenient way of marking the hyperlinked region of text. Here we use the predefined style `PDFStyle.LINKSTYLE` which underlines the text in the same way as an HTML hyperlink (as we've done in this document), but a user-defined style can be used instead for a different effect. This method handles the case where a hyperlink wraps at the end of the line, or even at the end of the page.

## Other Annotations

A good number of annotations are defined in the PDF specification - the ability to attach files and other useful features are all there. Many of these require the full version of Adobe Acrobat, and aren't supported by this library at the moment.


Two types of annotation that we do support are the *note* and *rubber-stamp* annotations. The *note* annotation is the electronic equivalent of adding a Post-It® note to your document. This can be clicked on by the viewer to view its contents or dragged about the page to a different location.

Notes are represented by the `AnnotationNote` class, and stamps by the `AnnotationStamp` class we've already mentioned above. Both of these can be drawn in a number of different pre-defined styles, and custom Stamps can be created by using a `PDFCanvas`, in the same way as a custom pattern. This is useful for applying "Received on DD-  
MMM-YYYY" type stamps. Alternatively, advanced users may pre-define and save custom patterns. How? Read on...

## Forms

Users of the *Extended Edition* of the library can read and create PDF forms, also called "AcroForms". Users who just want to complete a pre-existing form will probably find this section useful, but may get more mileage from the Completing Forms section.

These are part of a PDF document, in the same way a `<form>` tag in HTML creates a form on a web page. Like HTML forms, PDF forms can contain text-boxes, radio buttons, checkboxes, drop-down lists and push buttons, can reference JavaScript functions or submit the form to a website.



- Text Box*
- Choice Box*
- Check Boxes*
- Radio Buttons*
- Button*

*Forms are not the simplest area of a PDF document to understand, and if you're just starting with forms this document is the wrong place to start. Forms can be created from within Acrobat, and there are several guides with that product and on the net which explain about form fields, JavaScript, form submission and so on.*



Each PDF document has a single form (unlike HTML, where one page may have several forms), and the elements of this form may be spread across several pages. The `PDF.getForm()` method is used to return the documents form, and from there the various methods in the `Form` class can be used to set and retrieve elements. It's important to note that because each document can only have a single form, each field in the form must have a distinct name.

Each form element is a subclass of `FormElement`, and has a value which can be set or retrieved via the `setValue` and `getValue` methods. For many applications of forms, where the values on an existing form are read or written to, this is as much of the API that will be required.

For creating new forms, we need to get a little deeper. New elements can be created and added to the form using the `Form.addElement` method - don't forget this last step, or you'll be wondering why your fields aren't showing up!. All form elements can have a style set with the `setStyle` method, although not every feature of a style can be used in a form field.

Most form elements have a visual representation on the page - some, like radio buttons, have several, whereas digital signatures (covered later) often have none. This representation is a special class of `PDFAnnotation` called a `WidgetAnnotation`. The list of annotations associated with an element are returned by the `FormElement.getAnnotations` method.

Finally, one of the more interested aspects of form elements is the triggers or "Events" which can occur. Just like HTML, it's possible to call a JavaScript function (or other action) when the value of a field is changed, when the mouse enters a field or when a key is pressed inside one. The `setAction` method can be called on the elements' annotations or on the element itself (depending on the action you want to set), to submit a form, run some JavaScript, jump to another page or any other action you can think of. The example on this page calls a JavaScript function when the submit button is clicked.

Let's start with a simple example. Reading and writing values to an existing form is very easy, and probably for most `Form` users, this is as far as you'll need to go:

```
1. PDF pdf = new PDF(new PDFReader(new FileInputStream("template.pdf")));
2. Form form = pdf.getForm();
3.
4. FormText name = (FormText)form.getElement("name");
5. System.out.println("The name field was set to "+name.getValue());
6.
7. name.setValue("J. Quentin Public");
```

*Example 22 - Setting a form field*

As you can see, you first call the `getForm()` method to get the documents `AcroForm`, then the `getElement` method on that form to return a specific element (there are other ways to do this - the `Form` class has methods that return a `Map` of all the elements, for example). Once you've got the element, you can get or set the value using the `getValue` and `setValue` methods as required. When the document is eventually written out using the `render` method, the form is written out with the last values that were set.



Creating your own form fields isn't that much harder. All the elements have a consistent interface. Starting with the basics, here's how to add a text field to a new PDF.

```
1. PDF pdf = new PDF();
2. PDFPage page = pdf.newPage(PDF.PAGESIZE_A4);
3. Form form = pdf.getForm();
4.
5. FormText text = new FormText(page, 100, 100, 300, 130);
6. form.addElement("mytextfield", text);
7.
8. pdf.render(outputstream);
```

*Creating a new Form element*

The text field is placed in the rectangle 100,100 - 300,130 on the specified page. The field uses the default style for form elements - you can change this by calling the `setTextStyle` and `setBackgroundStyle` methods on the `Form` object, or you can customize the style for a single `WidgetAnnotation` by calling the same methods - following on from the example above, something like `text.getAnnotation(0).setTextStyle(style)` would do it.

## Form Actions

It's possible to set actions on a form elements annotations, which can range from simply submitting the form to calling complex JavaScript functions. The action can be any of the actions created by the `PDFAction` class, some of which we've already seen. Here's a quick summary.

Action	Description
goTo	Jump to a specific page in the current document
goToURL	Jump to a specific hyperlink. As you would expect this requires a web browser to be installed.
playSound	Play an <a href="#">audio sample</a>
named	Run a named action
showElement	Display an annotation that was previously hidden
hideElement	Hide an annotation that was previously visible
formSubmit	Submit the form to a specified URL on a server
formReset	Rest the form to it's default values
formImportData	Import an FDF file into the form
formJavaScript	Run a JavaScript action

*Table 5 - Form Actions*

The `goTo`, `playSound` and `named` actions are covered elsewhere in the document, so we'll briefly cover the form-specific actions `formSubmit` and `formJavaScript` - `formReset` is fairly obvious, and `formImportData` is for advanced use, and is better described in the PDF specification.

First, `formSubmit`. As you might have guessed, this allows the form to be submitted to a server. Like HTML forms, which can be submitted via GET or POST, there are several options for how to submit the form, including HTTP POST, FDF (to submit the form in Adobes own FDF format), and users of Acrobat 5.0 or later can submit the form as an XML document or can actually submit the entire PDF document - wordy, but good for digitally signed documents.

Next, the `formJavaScript` option, as demonstrated in the sample form above. Acrobat comes with a version of JavaScript similar, but not identical to the JavaScript supplied with most web browsers. Although the syntax is identical, the object model is very different - browsers use the Document Object Model, or *DOM*, whereas Acrobats object model is documented in it's own JavaScript guide in a file called `AcroJS.pdf` supplied with Acrobat 5.0. It's also currently available for download from <http://www.planetpdf.com/codecuts/pdfs/tutorial/AcroJS.pdf>. JavaScript actions can define JavaScript code directly, or call a function in the "document wide" JavaScript, which can be set via the `PDF.setJavaScript` method. This is recommended for anything but the simplest JavaScript.

Finally, the `showWidget` and `hideWidget` actions. These can be used to show or hide an existing widget annotation. This can be used to interesting effects, as you can see by taking a look at the `example/FormVoodoo.java` example supplied with the PDF library.

## Events

So when and how can you use these actions in a form? The most obvious time an action is required is when using a `FormButton`, for example to submit a document. You can call the `setAction` method on the buttons' widget to determine what to do when the the user clicks on it. Some events apply to the field and some to the fields annotations (see the API documentation for detail), and most of them will be familiar to most JavaScript programmers, and include `onMouseOver`, `onFocus` and `onChange`. Example uses could include verifying keyboard input in a text box by setting the `onKeyPress` handler to ensure only digits are entered, or maybe the `onOtherChange` handler, which is called when *other* fields in the document are changed - useful for updating a read-only field with the total of other fields, for example. In fact, we do just this in the `examples/FormVoodoo.java` example which we mentioned earlier.

## Digital Signatures

Since version 1.1.13, PDF documents may be digitally signed by those running the *Extended Edition* of the library. These are useful for two main purposes - one, to identify the author of the document, and two, to provide notice if the document has been altered after it was signed. This is done by calculating a checksum of the document, and then encrypting that checksum with the "private key" of the author, which can later be verified by a user with the full version of Adobe Acrobat or Acrobat Approval™, although *not* the free Acrobat Reader, by comparing it with the corresponding public key.

*Note: Applying Digital Signatures to a document requires some basic knowledge of public/private key cryptography, which is a weighty topic. We provide a brief description here, but some knowledge of public key cryptography is assumed.*

Digital Signatures are implemented in Acrobat via a plug-in or "handler". Over the years the available handlers have changed slightly - Acrobat 4.0 came with a fairly limited self-sign handler, and VeriSign produced a plug-in which could be used to certify PDF's signed by their keys. Acrobat 6.0 was the first to be delivered with an out-of-the-box solution for signing and verifying any PKCS#7 signature. The PDF library can sign and verify Digital Signatures created with the old VeriSign handler, the original self-sign and the new public-key Acrobat handlers, and the plugin used with the nCipher DSE-200 Timestamping engine. As the `SignatureHandler` class can be extended, other handlers can be developed by the user if required.

So, how do you sign a document? As this is a userguide rather than a reference, we'll step through how to do it without going into too much detail. See the API class documentation for the `FormSignature` class for more depth.

### Signing documents with the Adobe "Self-Sign" Handler

First, we'll cover the Adobe Self-Sign handler, which is supplied with every version of Adobe Acrobat. This handler requires a self-signed key, which you can generate using the `keytool` application that comes with Java. To generate a key, run the following command:

```
keytool -genkey -keyalg RSA -sigalg MD5withRSA -keystore testkeystore -alias mykey
```

This will ask a number of questions and will eventually save the key to the file "testkeystore". If you're going to try this, it's important to enter a two-letter country code (rather than leaving it set to "Unknown"), otherwise Acrobat will be unable to verify the signature.

Once you have the private key and its accompanying certificates stored in the keystore, the next trick is to sign the document. One method is just to use the `Sign.java` example, supplied in the examples directory. If you want to write your own code however, there's not much to it.

```
1. import java.security.KeyStore;
2.
3. PDF pdf = makeMyPDF(); // Create your PDF document somehow
4.
5. KeyStore keystore = KeyStore.getInstance("JKS");
6. keystore.load(new FileInputStream("testkeystore"), storepassword);
7.
8. FormSignature sig;
9. sig = new FormSignature(keystore, "mykey", secret, FormSignature.HANDLER_SELFSIGN);
10.
11. pdf.getForm().addElement("Test Signature", sig);
12.
13. pdf.render(new FileOutputStream("signed.pdf"));
```

*Example 23 - Signing a document using the Self-Sign handler*

First, we create and load the `KeyStore` on lines 5 and 6 - `storepassword` is a `char[]` array containing the password to decrypt the keystore. Then we actually create the signature on line 9, by specifying the keystore, the key alias ("mykey"), the password for that key (also a `char[]`), and the type of handler we want to verify this signature. Finally, on line 11 we add that signature to the PDF documents' form.

## **Signing documents with the VeriSign Handler**

*Note. This information is largely out of date, as the VeriSign handler is no longer required for Acrobat 6 or later. We're leaving it for reference.*

To use VeriSign signatures you'll need the VeriSign "Document Signer" handler, freely available for download from <http://www.verisign.com/products/acrobat/>. The signing procedure is the same except you change `HANDLER_SELFSIGN` to `HANDLER_VERISIGN`. The difference comes in how you acquire the key, as (unlike the Self-Sign handler) the key must be certified by VeriSign. These steps will almost certainly apply to signing documents for verification with the Acrobat 6.0 handler as well - most of the work here is in getting the key.

Luckily you can get one for free by visiting <http://www.verisign.com/client/enrollment>. Just follow the "trial certificate" instructions and you'll be issue with a private key and signed certificate by VeriSign which is good for 60 days, and is installed into your browser. Getting it out of your browser into a form we can use from Java is the next trick. For Internet Explorer:

- Go to the "Tools" menu, select "Internet Options"
- Select "certificates" (figure 1)
- Select the certificate you want to export and click "Export"
- Select "Yes", as you do want to export the private key
- Include the entire certification path (figure 2)
- The file is saved as a PKCS#12 keystore

For Netscape, go to the "Communicator" menu, select Tools -> Security Info". Then select the certificate off the list and choose "Export".

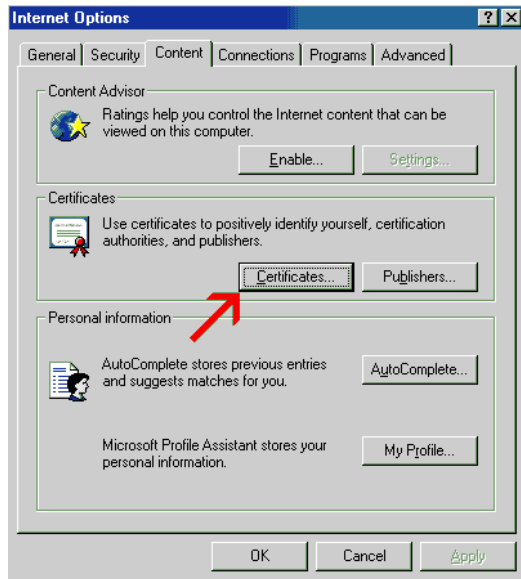


Figure 5a

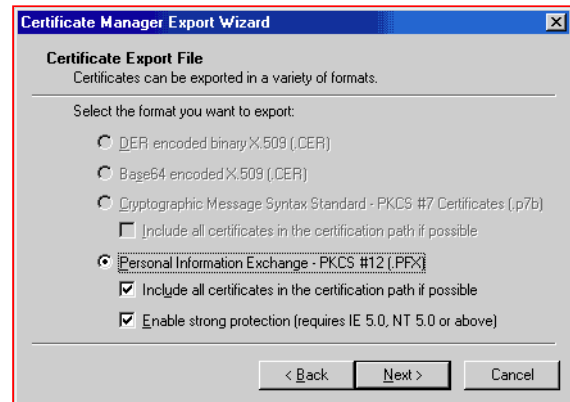


Figure 5b

To acquire a permanent key rather than a temporary one, you need to purchase one from a CA like VeriSign (other CA's can be used, although it's unlikely that VeriSigns plugin for Acrobat 5 will verify these out of the box. Acrobat 6 has a general PKI plugin which should theoretically work with any CA in the Windows trusted CA list, although what happens on non-windows boxes is anyone's guess).

VeriSign sell a bewildering number of digital products, but in order to sign documents we recommend the "Code Signing" key available at <http://www.verisign.com/products/signing/code>. They sell digital IDs specifically for signing Jar files, which is the one you want. The process for this is a little different. You need to follow exactly the same process as you would do to acquire a key for signing Jars, which is to say the following steps:

1. Use the `keytool` program to generate an RSA/MD5 key, in exactly the same way as described above for the Self Sign signatures.
2. Run `keytool -certreq -alias mykey` to generate a certificate signing request for your new key. This will output a block of Base64 encoded data, which you need to cut and paste into the appropriate section on VeriSigns website when prompted.
3. Once you're approved, the CA will send you a certificate reply chain. You need to copy this chain into your keystore by running `keytool -import`, e.g. `keytool -import -alias mykey -file verisign.cer`. The import must use the same alias as the original key - here we're using "mykey".

So now you have your private key and accompanying certificates in a Java keystore. If you created a test certificate via IE or Netscape, the keystore is a PKCS#12 keystore, an industry standard format which works out of the box in Java 1.4, but with earlier versions will require the installation of an appropriate Java Cryptography Extension, or JCE. The homepage for the JCE range is at <http://java.sun.com/products/jce>, which includes a list of providers. We developed with, and recommend the free JCE provided by "The Legion of the Bouncy Castle" (<http://www.bouncycastle.org>). Just download the package, add the Jar to your classpath and finally register the provider by adding a new line to the `JAVA_HOME/jre/lib/security/java.security` file as follows:

```
security.provider.2=org.bouncycastle.jce.provider.BouncyCastleProvider
```

Once that's done, you can verify things are working by loading the PKCS#12 keystore with the "keytool" program - use a command similar to the following to list the key details and find out the alias of your private key:

```
keytool -list -v -keystore mystore.pfx -storetype pkcs12 -storepass secret
```

Our alias (sometimes called "friendly name") was a decidedly unfriendly string of about 30 hex digits. No matter - modify the code example above to use this alias instead of "mykey", and change the "JKS" to a "pkcs12". You should now be able to sign a PDF document using this key - something you can confirm by loading it into a copy of Adobe Acrobat with the VeriSign handler installed, going to the "Window" menu, selecting "Show Signatures" and verifying the signature.

Annotations can be added to digital signatures to give them a visible appearance on the page. For more detail on this see the API, but the short version is that you can add a "standard" appearance just by adding a regular annotation, like this (taken from the example above, with line 10 inserted):

```
8. FormSignature sig;  
9. sig = new FormSignature(keystore, "mykey", secret, FormSignature.HANDLER_SELFSIGN);  
10. sig.addAnnotation(pdf.getPage(0), 100, 100, 200, 200);  
11. pdf.getForm().addElement("Test Signature", sig);  
12.  
13. pdf.render(new FileOutputStream("signed.pdf"));
```

*Example 24 - Adding an Annotation to a signature*

If you want to add a custom appearance, e.g. a photo or linedraw signature, you can define one on a canvas and pass it in as a custom appearance. You'd do this by inserting a new line between 9 and 10 in the example above that looked like this:

```
((PKCS7SignatureHandler) sig.getSignatureHandler()).setCustomAppearance(canvas, 0,0,0,0)
```

See the JavaDoc API for more detail on this.

## Verifying Signed Documents

As well as signing new documents, previously signed documents can be verified by comparing their signing signature to a list of "trusted" certificates. For example, verifying a document created by the VeriSign plugin is easy:

```
1. import java.security.KeyStore;
2. import java.security.cert.Certificate;
3.
4. PDF pdf = new PDF(new PDFReader(input));           // Load the PDF
5.
6. KeyStore trusted = FormSignature.loadDefaultKeyStore(); // Trusted Certificates
7.
8. Map form = pdf.getForm().getElements();
9. for (Iterator i = form.keySet().iterator();i.hasNext();) {
10.     String key = (String)i.next();
11.     FormElement value = (FormElement)form.get(key);
12.     if (value instanceof FormSignature)
13.     {
14.         // Verify the signature integrity
15.         boolean integrity = ((FormSignature)value).verify();
16.
17.         // Verify the signature covers the whole document
18.         int pdfrevision = pdf.getNumberOfRevisions();
18.         int sigrevision = ((FormSignature)value).getNumberOfRevisionsCovered();
19.         boolean allcovered = (pdfrevision==sigrevision);
20.
21.         // Find the first unverifiable certificate. Null means they're all OK.
22.         Certificate badcert = ((FormSignature)value).verifyCertificates(keystore);
23.     }
24. }
```

*Example 25 - Verifying a signature*

This example excerpt shows how to cycle through a documents digital signatures and verify them:

- First checking the document *integrity*, which confirms the section of the document that the signature covers hasn't been altered since it was signed.
- Second checking the signature *scope*, to confirm that the digital signature covers the entire file - that nothing has been appended to the file after signing.
- Finally, checking the certificate *authenticity* by ensuring that the certificate chain has been signed by a trusted certificate. Here we've loaded our trusted certificates on line 6, using a method which loads the same keystore that's used to verify signed Jar files.

This keystore contains the VeriSign Root certificates, so we can easily verify VeriSign-signed documents this way. But what about Adobe's Self-Sign handler? This handler doesn't have the concept of a "trusted root certificate", and all it's keys are self-signed, so verifying against a standard keystore is impossible - the above example would always return a certificate on line 18. However, Adobe Acrobat 4.0 can export the public key that was used to sign a document, which we can load as the "trusted" keystore. To export the public key from the Acrobat "Self-Sign" handler, open Acrobat and go to the "Self-Sign" menu. From there, select "User Settings", click the "Personal Address Book" tab and select "Export Key File". The file is exported as an "Adobe Key File", or .AKF file - another proprietary keystore format - which can be loaded using the `FormSignature.loadAKFKeyStore` method, which would replace line 6 above. Users of Acrobat 5.0 can export their key as the industry standard PKCS#12 keystore format, or as an .FDF file which can be loaded using the `FormSignature.loadFDFKeyStore` method.



## **Things to be aware of with Digital Signatures**

- Signing a file causes all future changes to it - including the addition of new signatures - to be appended to the file as a new "revision". When verifying a signature it's important to confirm that the signature covers the whole file. Remember that only the latest signature will cover the whole file, as every new signature adds a new revision. See the PDFReader API documentation for more about PDF revisions.
- The Adobe "Self-Sign" handler required each certificate to be personally verified, as there is no concept of signing or root certificates - something to be think about if you're thinking of using this handler for wide-scale deployment. These days the "Acrobat 6" handler is usually the one to go for.
- The `Sign.java` and `Dump.java` examples supplied with the library provide working code examples of signing and verifying signatures respectively.
- The concept of "trust" is a complicated one with PKI. Although we've blindly decided to trust the keystore supplied with Java and the certificates it contains, you should make this decision yourself before simply loading the default keystore and verifying using the `verifyCertificates` method (the list of certificates can be extracted from the `FormSignature` object for manual verification). Remember when it comes to cryptography it's not whether you're paranoid, but whether you're paranoid enough.
- Acrobat Reader prior to version 5.1 can not verify signatures at all - the full version of Acrobat or Acrobat Approval was required.

## **Special Features**

There are a number of features in the library that we haven't covered in the previous sections. There's no way to classify most of these so we'll just list them all.

### **Sound**

One of the less useful additions to the PDF format was the ability to play sounds. This feature has patchy support amongst viewers, even Adobes own Acrobat viewers. Still, in the best tradition of "feature creep" we thought we'd add it anyway. The `PDFSound` class in the library handles all the formats listed as acceptable by the PDF specification - Microsoft .WAV, Sun .AU and Macintosh .AIFF audio files (no MP3s, sorry). However, we've found only the .WAV format works on the Windows Acrobat viewers (and even that with some problems), and we can't get a peep out of our Linux machine. Still, for the curious, click [here](#) to hear this marvelous feature in action (bonus points if you can name the tune).

### **Bookmarks**

The bookmarks (also called "outlines") feature of PDF documents is probably one of the reasons they're so popular - allowing a true "table of contents" in larger documents. The libraries support for bookmarks is based around the `PDFBookmark` and the familiar `java.util.List` classes, allowing easy manipulation of complex chains of bookmarks and simple creation of bookmark trees.

## Encryption and Access Levels

Documents can be encrypted to prevent unauthorized access, either by limiting what the user can do with a document (i.e. they're not allowed to print it) or by adding a password. The library supports both the 40-bit RC4 encryption used in Acrobat 3 and later, the 128-bit RC4 encryption added in Acrobat 5 and the 128-bit AES encryption added in Acrobat 7. This is best demonstrated by a couple of simple examples. First, here's how to encrypt a PDF with 40-bit encryption so that it can be opened as normal in all versions of Acrobat, but cannot be printed.

```
1. PDF pdf = new PDF();
2. ... create contents of PDF here ...
3.
4. StandardEncryptionHandler enc = new StandardEncryptionHandler();
5. enc.setAcrobat3Level(false, true, true, true);
6. pdf.setEncryptionHandler(enc);
```

*Example 26a - Turning of Printing with 40-bit encryption*

Here's how to do the same thing, but using 128-bit encryption so the document can only be opened in Acrobat 5 or later.

```
1. PDF pdf = new PDF();
2. ... create contents of PDF here ...
3.
4. StandardEncryptionHandler enc = new StandardEncryptionHandler();
5. enc.setAcrobat5Level(enc.PRINT_NONE, enc.EXTRACT_ALL, enc.CHANGE_ALL);
6. pdf.setEncryptionHandler(enc);
```

*Example 26b - Turning of Printing with 128-bit encryption*

And here's how to encrypt with 40-bit again, but to require a password to open the document.

```
1. PDF pdf = new PDF();
2. ... create contents of PDF here ...
3.
4. StandardEncryptionHandler enc = new StandardEncryptionHandler();
5. enc.setUserPassword("secret");
6. pdf.setEncryptionHandler(enc);
```

*Example 26c - Setting encryption and a password*

## Creating your own predefined canvases

For advanced users who know about PDF internals, it's possible to create your own pre-defined canvases as `ResourceBundles`, which can be loaded in from a `properties` file. These can be used as stamps, as logos for signature handlers, or just as an easier way to define complex canvases. The `AnnotationStamp`, `PDFCanvas` and `PDFPattern` can all take the name of java resource bundle as an parameter.

Each `ResourceBundle` must have at least a `width` and `height` attribute to size the canvas, and an `action` attribute which contains a sequence of PDF stream operations. If the canvas needs to reference any resources, these may also be stored as *direct objects only* by giving them an attribute name of `resource.X.Y`, where X is the type of resource (Font, Pattern etc). and Y is the resource name.



Confused yet? Remember, this is for advanced users, and although a useful time-saving device is not necessary. What we'd recommend to get started is to extract some of the predefined canvases from the `bfopdf.jar` file (they're all in the `org.faceless.pdf2.resources.canvases` and `org.faceless.pdf2.resources.patterns` packages) and take a look. To get you started, here's a couple of quick examples - the first showing how to create a pattern of alternating up and down triangles, and the second showing how to create a stamp that simply says "Hello".

```
# alternating up/down triangle pattern
#
width      = 20
height     = 20
action     = 0 2 m 8 2 1 0 18 1 f   10 4 m 18 20 1 2 20 1 f   12 2 m 20 2 1 20 18 1 f
```

```
# "Hello" stamp - draw a rectangle and write "Hello" in black 24pt Helvetica
#
name       = Hello
width      = 80
height     = 40
resource.Font.F0 = <</Type /Font /Subtype /Type1 /BaseFont /Helvetica>>
action     = 0 G 0 0 80 40 re S   0 g 1 0 0 1 10 30 Tm /F0 24 Tf (Hello) Tj
```

*Example 27 - Resource files for a custom Pattern and Stamp*

To the first one, you could save the file as, say, "yourcompany/Triangle.properties", then create a new pattern by calling `new PDFPattern("yourcompany.Triangle", 0, 0, 20, 20, Color.red, NULL)`. Note we need to specify the colors, as they're not set in the pattern itself. The second one is a stamp, so you would save it as something like "yourcompany/HelloStamp.properties", then create a stamp by calling `new AnnotationStamp("yourcompany.Hello");`.

(Where this really comes into it's own is that with a bit of effort, you can create stamps in Illustrator and the like, distill them as uncompressed PDFs, then extract the PDF stream and save it out. This isn't as simple as it sounds, and you may wind up having to prefix the extracted stream with some page coordinate transformations, but it can be done - it's just not easy. Remember, this is for advanced users!)

## Bar Codes

The `drawBarCode` method of the `PDFPage` class allows a bar code to be added at a specific position on the page, in one of several encoding systems. The text of the code can optionally be written underneath (as we've done here) and a checksum can be added if the algorithm supports it.



CODE 39

The Code 39 barcode is a simple system which can represent the digits, 26 upper-case letters, the space and a few punctuation characters. We also support Extended Code 39, which supports more characters at the expense of even longer codes.



20010908

The Interleaved code 2 of 5 barcode is a simple system which can represent only the ten digits, although it's fairly compact.



Code 128

The Code 128 barcode is a newer barcode which can represent almost all the US-ASCII range of characters. It's also fairly compact. The algorithm chooses the appropriate CODEB or CODEC variant depending on the data to be encoded, although this can be overridden (see the `BarCode` class API documentation). For EAN128 codes, the newline character (`'\n'`) can be used to embed an FCN1 control character into the code.



0 012345 678905

The EAN-13 barcode is extremely common, and is generally used for product labeling (it's the barcode on all your groceries and books). It must contain 13 digits, the last of which is a checkdigit.








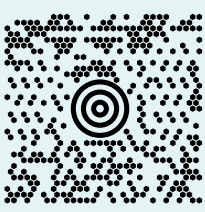
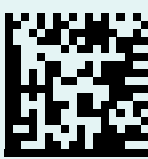


 <p>0 012345 678905</p>	<p>The UPC-A barcode is the US-only subset of EAN-13. Although all scanning equipment in the US should be updated to recognize EAN-13 codes by 2004, in the meantime the traditional UPC-A codes can also be printed.</p>
 <p>0123 4565</p>	<p>The EAN-8 barcode is variation on the EAN-13 code, but with only 8 digits.</p>
 <p>A12345B</p>	<p>The CodaBar barcode can represent the digits as well as the characters + - / \$ : and the decimal point (.). Special "start/stop digits" must be used to start and stop the barcode - these are one of A, B, C or D.</p>
	<p>The PostNet barcode is used by the US Postal Service to encode ZIP codes. It uses digits only.</p>
	<p>The Royal Mail 4-state Customer Code is used by the Royal Mail in the UK to encode PostCodes. It can represent digits and the upper case letters A to Z.</p>
	<p>PDF417 is a "stacked" 2D barcode - probably the most common one. It's used for a wide variety of purposes (for instance, paper archives of electronic invoices in Spain must use PDF417).</p>
	<p>QR-Code is a 2D barcode, invented and commonly used in Japan but making headway elsewhere too, due to it's ability to store Kanji and it's incredible density - the largest version can store over 6000 digits.</p>
	<p>"MaxiCode" is a 2D barcode invented by UPS to code destinations on it's packages, and is now in the public domain. It can encode up to 93 alphanumeric or 137 numeric characters.</p>
	<p>"Data Matrix" is another popular 2D barcode, which is also in the public domain.</p>
	<p>The IntelligentMail® code was introduced by the USPS in 2008 to replace Postnet. It takes a 20, 25, 29 or 31 digit value.</p>
	<p>"Aztec Code" is a modern 2D barcode that is visually similar to QR Code.</p>

Table 7 - Bar Codes

**Document Layout and Meta-Information**

The library can add meta-information about the document to the PDF with the `setInfo` method in the `PDF` class, allowing you to set the author, title, subject and so on. This can be viewed in Acrobat by going to the "Document Information" or "Document Properties" option under the File menu.

You can also specify various options in the document, instructing the PDF viewer how to display it (for instance, this document opens in Acrobat with the bookmarks pane already open). The `setOption` method in the `PDF` class control various settings to do with what to do with the document when it opens.

## XML Metadata (Adobe XMP™ support)

PDF documents supporting the PDF 1.4 specification can include XML meta-information on almost any object in the document - pages, fonts, images or the document itself. This XML is formatted using the Resource Description Framework (RDF), and the best place to start with all this is <http://www.adobe.com/products/xmp>, because a discussion of this is well beyond the scope of this document.

This potentially useful extension to the PDF format is supported by the library via the `setMetaData` and `getMetaData` methods which are part of several classes in the library. The XML data can be added or extracted from these objects as a String, which can then be passed to an XML API like SAX or DOM, and from there to an RDF-specific framework like Jena, available from <http://www.hpl.hp.com/semweb/arp.html>

## Preflighting and Output Profiles - controlling how the document is written

PDF is quite a large specification with several useful subsets. An obvious example of a subset is "PDFs that can be read by Acrobat 4" (which would rule out anything encrypted with 128-bit encryption), and there are many other possibilities. One of the more interesting subsets is "PDF/X", an ISO standard subset of PDF specially designed for use in the prepress industry. The PDF library supports two of the most common variations of PDF/X, known as PDF/X-1a and PDF/X-3.

Ensuring a PDF meets the requirements of a particular feature set is done with the `OutputProfile` class. A PDF can have an Output Profile assigned to it with the `PDF.setOutputProfile` method - prior to 2.6.1 this had to be done when it was created, but with the arrival of the Viewer Extensions it's possible to parse an existing PDF, determine it's feature set to ensure it matches the specified profile and (if possible) adjust it to match. This process is generally known as *preflighting*.

Applying an output profile to a PDF that doesn't match, or trying to use a feature that is disallowed in a PDF's output profile, will cause an `IllegalStateException` to be thrown with information on what the restriction is. For instance, this will cause an exception on line 6, because 128-bit encryption is not allowed in the Acrobat 4 profile:

```
1. PDF pdf = new PDF();
2. pdf.setOutputProfile(OutputProfile.Acrobat4Compatible);
3.
4. StandardEncryptionHandler enc = new StandardEncryptionHandler();
5. enc.setAcrobat5Level(enc.PRINT_NONE, enc.EXTRACT_ALL, enc.CHANGE_ALL);
6. pdf.setEncryptionHandler(enc);
```

*Example 28 - Using an OutputProfile to control the type of PDF written*

It's *sometimes* possible to load an existing PDF and adjust it to match an Output Profile. Why only sometimes? As an example, PDF/X-1a:2001 doesn't allow device-dependent RGB colors to be used in the PDF. Many PDFs do use RGB however - creating a PDF with this library and using a `java.awt.Color` for text will create such a PDF. The library doesn't convert RGB to CMYK, and so attempting to apply a PDF/X profile to this PDF would throw an Exception on the `setOutputProfile()` method.

The `OutputProfile` API documentation has a lot more information on this, and the `Preflight.java` example shows how to pre-flight an existing document against the PDF/X-1a:2001 standard.

# The PDF Viewer Extensions

## *About the Extension*

The PDF Viewer extensions are a collection of classes that allow PDF documents to be *parsed* - ie. the contents of the PDF are read, understood and converted to some other form. Currently available conversions are *rasterization* - the conversion of the PDF to a bitmap for viewing on the screen, printing or saving as an image, and *plain text*, which allows the text and images of the PDF to be extracted. More specifically, features include:

- Conversion of PDF to multi-page TIFF (Black and White, GrayScale, RGB or CMYK)
- Conversion of PDF page to a `java.awt.image.BufferedImage`, for other image formats such as PNG or JPEG.
- Printing PDFs using the standard Java `Printable` and `Pageable` interfaces.
- Extraction of text from a PDF
- Indexing of text in a PDF using the Apache Lucene API
- Extraction of bitmap images from a PDF as `java.awt.image.BufferedImage` objects
- Embed PDF viewer in your own application using a `JPanel`
- Use the supplied PDF viewer sample application as a standalone viewer

## *Installation and getting started*

No additional installation is required to use these classes, as they are included in the `bfopdf.jar` package - this includes the `org.faceless.pdf2.viewer2` package containing the Swing components for an interactive PDF viewer application. The exception to this is indexing the PDF using Apache Lucene - you will need the Lucene Jars in your classpath.

Unlike the rest of the library the rasterization code in the `PDFParser` makes heavy use of the `java.awt` package, so UNIX users running Java 1.3 or earlier will require X11 running (this does not apply for text extraction). Mac users will find that the rasterization process tends to crash Java 1.4 on OS X - we recommend Mac users run at least Java 1.5. Finally the `PDFViewer` class (the sample Swing application we supply to view PDFs) requires Java 1.4 or later to run, although the actual PDF to bitmap conversion will run under Java 1.3.

If you're trying to rasterize a PDF that does not have all it's fonts embedded, the viewer will use the fonts available to Java to try and pick an appropriate one to substitute. For Chinese, Japanese and Korean text in particular it is important to have a suitable font available, or no text will be displayed.

The top level class for all of this is `PDFParser`, which can either be used directly or as a way to instantiate `PageParser` or `PageExtractor` objects for parsing each individual page object. A very simple example here shows how to convert a multi-page PDF to a multi-page TIFF.

```
import java.io.*;
import org.faceless.pdf2.*;

public class PDFtoTIFF {
    public void main(String[] args) throws IOException {
        PDFReader reader = new PDFReader(new File(args[0]));
        PDF pdf = new PDF(reader);

        FileOutputStream out = new FileOutputStream("PDFtoTIFF.tiff");
        PDFParser parser = new PDFParser(pdf);
        parser.writeAsTIFF(out, 72, PDFParser.RGB);
        out.close();
    }
}
```

*Example 29 - Converting a PDF to TIFF*

The two bold lines show the `PDFParser` class in use - first, it's created from the `PDF` object, then the PDF is converted to a 72dpi RGB TIFF and written to a file. Simple.

## Printing a PDF

There are two printing models in Java: Printable jobs (which are single page print jobs) and Pageable jobs, for multi-page jobs. The PDFParser class implements Pageable, which means the entire document can be printed easily. Alternatively the PageParser object implements Printable so individual pages can be printed as necessary. Here's a comprehensive example demonstrating both approaches - an individual page can be printed using the Printable interface, or a range of pages can be printed using the Pageable approach.

```
import java.io.*;
import java.awt.print.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;
import org.faceless.pdf2.*;

// Usage is either
//
// java PrintPDF filename.pdf           To print the entire document
// java PrintPDF filename.pdf num       To print the page "num"
// java PrintPDF filename.pdf num1 num2 To print the pages "num1" to "num2"
//
public class PrintPDF
{
    public void main(String[] args) throws IOException, PrinterException
    {
        PDFReader reader = new PDFReader(new File(args[0]));
        PDF pdf = new PDF(reader);
        PDFParser parser = new PDFParser(pdf);

        PrintRequestAttributeSet printAttributes = new HashPrintRequestAttributeSet();
        printAttributes.add(OrientationRequested.PORTRAIT);
        MediaPrintableArea area = new MediaPrintableArea(0,0,210,296, MediaPrintableArea.MM);
        printAttributes.add(area);

        PrinterJob job = PrinterJob.getPrinterJob();

        if (args.length==1) { // Print the whole document
            job.setPageable(parser);
        } else if (args.length==2) { // Print a single page
            int pageno = Integer.parseInt(args[1]);
            PagePainter painter = parser.getPagePainter(pageno);
            job.setPrintable(painter);
        } else if (args.length==3) { // Print a range of pages
            int firstpage = Integer.parseInt(args[1]);
            int lastpage = Integer.parseInt(args[2]);
            PageRanges range = new PageRanges(firstpage, lastpage);
            printAttributes.add(range);
            job.setPageable(parser);
        }
        job.print(printAttributes);
    }
}
```

*Example 30 - Printing a PDF*

This example can be compiled and run, and will print a PDF specified on the command line to the default printer. It uses the Java 1.1 printing API, which works but is slightly less flexible than the new `javax.print` API added in Java 1.4. The `PrintPDF.java` example supplied in the `example` directory shows how to use the 1.4 API. Note that **not all**

**PDF's allow printing** - we honor the usage flags set on the document, and will throw an Exception if a PDF cannot be printed.

## Text Extraction

Text extraction from a PDF was added in version 2.6.2, and is done via the `PageExtractor` class. You can get one of these by calling `PDFParser.getPageExtractor` for the appropriate page, and once you have one you can extract text, images and so on. Here's a simple example:

```
PDFParser parser = new PDFParser(pdf);
for (int i=0;i<pdf.getNumberOfPages();i++) {
    PageExtractor extractor = parser.getPageExtractor(i);
    System.out.println(extractor.getTextAsStringBuffer());
}
```

*Example 31 - Extracting text from a PDF*

As you can see there's not much to it. The `getTextAsStringBuffer` method attempts to extract and format the text in an *approximation* of the layout used on the page. Text Extraction from PDF is not an exact science for a number of reasons, including:

- Fonts may have no Unicode information or be encoded incorrectly, making it impossible to convert back to text
- Text may be rendered as an image, making extraction impossible
- Layout may rely on visual features of the glyphs. Determining how much space is between the end of one glyph and how much is before the start of the next (and whether it's a layout feature or an inter-word space) is not always obvious.
- When determining layout, features like superscript or subscript, overlaid or rotated text cannot be accurately represented in plain text.

For this reason results cannot be guaranteed as 100% accurate. Having said that most modern PDFs can have their content extracted 100% reliably, with layout approximating the PDF as closely as possible in plain text. Other methods in the `PageExtractor` class allow the list of text objects to be returned as a `Collection`, which not only provides information on exact page position, color and font, but can be modified to improve the text extraction process (for example, if you know you don't want any rotated text to be extracted, you can delete all the rotated text objects before running `getTextAsStringBuffer`).



## Image Extraction

As well as extracting text from the PDF, if the PDF allows it it's possible to extract bitmap images as well. This is done via the `PageExtractor.getImages` method, which returns a collection of `Image` objects used in the PDF, including their position on the page, resolution and a `java.awt.image.BufferedImage`. This `BufferedImage` can be saved as a PNG, JPEG or similar using the `javax.imageio` classes. Here's a simple example which saves all the images out to a file. Notice how we check for duplicates - images may be used more than once.

```
1. Set all = new HashSet();
2. PDFParser parser = new PDFParser(pdf);
3.
4. for (int i=0; i<pdf.getNumberOfPages(); i++) {
5.     PageExtractor extractor = parser.getPageExtractor(i);
6.     Collection images = extractor.getImages();
7.     for (Iterator j=images.iterator(); j.hasNext();) {
8.         PageExtractor.Image pageimage = (PageExtractor.Image)j.next();
9.         BufferedImage image = pageimage.getImage();
10.        if (!all.contains(image)) {
11.            all.add(image);
12.            File file = new File("image"+(all.size())+".png");
13.            ImageIO.write(image, "png", file);
14.        }
15.    }
16. }
```

*Example 32 - Extracting images from a PDF*

# Appendix A - Warning messages

A number of warnings are written to `System.err` by default if unexpected (but not fatal) events occur. These can be turned off individually by setting the System property `org.faceless.pdf2.warning.XXX` to any not NULL value, where XXX is the warning code, or all the warnings can be disabled by setting the `org.faceless.pdf2.warning.*` property. Be sure you know what you're doing though - they're printed for a reason.

AC1	No named action. A reference to a named action is stored in the document, but an action with that name doesn't exist. The action is replaced with a "dummy" action that does nothing.
AN1	No rectangle for annotation. An annotation has been created but not positioned on the page. It will remain but will have a rectangle of (0,0,0,0), making it effectively invisible in Acrobat. Generally speaking you'll want to set a rectangle before saving the PDF.
AR1	No such object. The same as RF1.
BC1	The BarCode has been created with parameters outside the acceptable range. Your barcode may still scan with some equipment, but it's not guaranteed to with all. Generally this is because the code is too small, so you'll need to enlarge it if it's not scanning correctly.
CS1	JRE has buggy ColorSpace handling. A number of older Java runtimes (in our tests, mainly those by IBM) have very, very poor implementations of the standard <code>java.awt.color</code> package, with tendencies to throw everything from Exceptions to JNI errors. We work around these as best as we can, but this is to let you know that you've got one of those JREs.
CS2	The embedded ICC profile cannot be parsed for some reason, and the alternate space will be used. Color Reproduction for the affect object won't be quite as accurate as it should be but otherwise, theoretically, no difference should be noticed.
E11	No unicode character for glyph 0xNN. You'll see this occasionally with Type 1 fonts that define characters that aren't in the mapping we have from glyph name to Unicode value. Often these are symbol characters, or occasionally misnamed regular or punctuation characters. We assign these characters an arbitrary position in the private range (U+EF00 to U+EFFF).
E12	No name for byte 0xNN. Caused by trying to read a form field containing a corrupt value. The result is that the value of the field will contain some 0x0000 characters instead of the correct ones.
E13	No Unicode character for name NNNN. This would be caused by trying to read a form field containing a corrupt value. The result is that the value of the field will contain some 0x0000 characters instead of the correct ones.
E21	No Unicode character for glyph 0xNN. This would be caused by trying to read a form field containing a corrupt value. The result is that the value of the field will contain some 0x0000 characters instead of the correct ones.
F1	Field has no name. Several problems can and do occur with disturbing frequency in Acrobat, often when a form is deleted - sometimes the job isn't finished properly. This particular warning indicates that a field has no name and is going to be deleted. As described in FE3, dangling fields are common, and if it has no name you can't interact with it in Acrobat anyway, so it's not doing you any good keeping it around.
F2	Widget has no name. A variation on the same theme as F1.
F3	Widget has no parent. Another variation on the same theme as F1.
F4	Fields point to orphan field. Yet another variation of the same.
F5	We can't imagine how this particular one could occur, but this warning is here just in case. If it happens, we'd like to see your document!
FE1	Annotation for field has not been assigned to a page. Every form element may have annotations, but each one of those has to be on a page. If you create an annotation but don't set it's page, it's no use to anyone, and this warning is thrown when the document is written to show that it's being removed.

- FE2 Annotation for field is on another PDF's page. Like the error above, but this normally occurs when you've been moving pages around between documents. If document A has a page and one form field with an annotation on that page, and then you move the page to document B, when you try and render document A, your form element has an annotation on page no longer in the document. Like FE1, this warning is thrown and the annotation removed.
- FE3 Document still contains reference to deleted form field. This isn't too uncommon. It usually occurs after flattening a documents form or after deleting a field from the form. What it means is that even though the field isn't included in the form anymore, somewhere else in the document there is a pointer to it, maybe in an action or in the "logical structure" information that is added by Acrobat for it's own purposes. These kind of "dangling fields" are left around by Acrobat all the time - we repair them quietly when a document is loaded - and although depending on context they could in theory cause a problem, in practice we've never seen it happen.
- FN1 Skipping unknown character 0xNN. This is the most likely one you'll see. It means you're trying to display a character which isn't in the font. With most of the emails we get about this one, the character is 0x09 - the tab character. A tab cannot be displayed simply by including it in a line of text! For a start, with a variable width font we've no idea how wide it should be. Please see the `addTab` method in the `LayoutBox` class for a better way.
- IM1 Unable to extract ICC profile from PNG. Could be because the PNG has some corruption, or because you've got the problem described for the CS1 warning.
- IM2 Unable to extract ICC profile from JPEG. Same as IM1 but for JPEGs
- IM3 CMYK JPEGs have trouble printing under Java 1.3. Due to bugs in the Java AWT code, CMYK JPEGs are not handled well at all. In Java 1.4 we're able to recode them with values that work using the `javax.imageio` package, but under Java 1.3 this isn't available and you'll see CMYK JPEGs come out inverted. Install `javax.imageio` or upgrade to Java 1.4. Or stick with RGB.
- LB1 Can't display 0xNN. This is the most likely one you'll see. It means you're trying to display a character which isn't in the font. With most of the emails we get about this one, the character is 0x09 - the tab character. A tab cannot be displayed simply by including it in the text! For a start, with a variable width font we've no idea how wide it should be. Please see the `addTab` method in the `LayoutBox` class for a better way.
- LB2 General text error. This could be caused by a number of reasons, but probably relating to some issue in the font or encoding. One possible reason for this is trying to display more than 255 different characters in a single byte font. Take a good look at the font you've picked for the job and what you're doing with it. Either way, if you see this message, there will be some text missing from your document when it's written.
- MP1 No such object. The same as AR1.
- OT1 OpenType font has restrictions on embedding. The font author has explicitly stated that the font is not allowed to be embedded without prior permission from them. We will embed the font anyway, but be aware that fonts are copyrighted material and that you will need to obtain permission from the font author and publisher legally
- OT2 No POST name for 0xNN. For single byte OpenType fonts, each character is required to have a PostScript name. Some (arguably) corrupt fonts are missing some, which means that even though the glyph is in the font you've chosen, you can't access it. This message is letting you know which ones. You could always try embedding the font using 2 bytes per glyph to get around it.
- PD1 Removing invalid annotation. You're loading a PDF with some sort of corrupt annotation or an invalid annotation list on one of the pages.
- PD2 Removing invalid widget annotation. You're loading a PDF with some sort of corrupt annotation or an invalid annotation list on one of the form elements.
- PD3 Widget referenced from page but not form. Unfortunately reasonably common, this is another case where Acrobat has almost certainly deleted a form field but not it's annotation. This warning is to let you know we're finishing the job.
- PD4 No Form Field. You're importing an FDF that is trying to set a value in a form field that doesn't exist in this document. This could indicate you're importing the wrong FDF or that you're importing it into the wrong form, or maybe a later/earlier version of the same form.
- PD5 Invalid date field. The `CreationDate` or `ModDate` stored in the "Info" dictionary is invalid. As you would expect this prevents the end user from identifying when the document was created or modified, but other than that there are no side effects.

- PD6 Invalid XMP Metadata. The XMP Metadata is either invalid XML or doesn't match the XMP specification. You can still read and write the metadata manually, but no automatic updates will be performed on the data - it will not be kept in sync with the Info dictionary as controlled by `PDF.setInfo()`. This warning is new in 2.6.2 - prior to that the XMP metadata wasn't parsed at all.
- PG1 Annotation is part of another PDF's form. The opposite of FE2, this occurs when in the situation described for FE2, you try and render document B. It's saying that a widget on this page is actually the widget for a field in another document. Since a Widget has to have a field, we're not going to render it and it gets deleted.
- PG2 Page has no size specified. We see this rarely in poorly constructed PDFs, and the only reason we're not flagging it as an outright error is because Acrobat can display them. We have to pick an arbitrary page size, which is typically Letter or A4 depending on the locale.
- RD1 Can't find "startxref". This is caused either by a truncated PDF, or by a PDF with loads of junk at the end. A PDF should always end with the "%EOF" tag, and just before that we need to find the word "startxref". If we can't find it we ignore the xref table and scan the document - exactly what Acrobat does when it says it's "Repairing" a PDF that's being read in. This isn't a problem unless the resulting PDF isn't what you expected, which could happen when documents with multiple versions are loaded.
- RD2 Can't find "startxref" again. The same as RD1.
- RD3 General xref table error. The most common version of the same error described in RD1, this is caused by a corrupt cross-reference table. As for RD1/2, the PDF is then scanned to recreate the table.
- RD4 Can't find N/N, got NNN. An indirect object is not in the PDF where it should be. This is fairly serious, but like RF1, you might get lucky and find your document is usable none-the-less. It all depends on what is missing.
- RD5 Stream is NNN bytes too long. This is more common than we'd like, mainly due to the incorrect length calculations in some inferior PDF generation routines. If NNN is small that's probably it, if it's big you've probably got a badly corrupt PDF and should consider yourself lucky to have got this far without an exception!
- RD6 Stream is NNN bytes too short. The other side of RD5, but generally less serious. Usually not a problem at all, just an indicator of poor PDF generation by someone.
- RF1 No such object. Something in your PDF is pointing to a missing object. This is the most commonly seen warning (even Distiller frequently produces these documents), but whether it's something to worry about depends on what's missing. PDFs, like people and engines, have lots of parts you can remove without any noticeable difference. If the document looks correct then the missing piece probably wasn't important.
- SG1 Certificate expired or not by VeriSign. When signing a digital signature destined for the VeriSign plugin, this warning is to let you know that your signature is most likely going to be invalid, for one of the specified reasons.
- ST1 Specified font not in form. As it says, a PDF has been loaded that contains a form field that is to be displayed in a certain font - but that font isn't specified in the form. How Acrobat managed to display it in the first place, we don't know, so we're substituting Helvetica. If this is a poor choice, you can set the style by calling the `setTextStyle` method on the appropriate `WidgetAnnotation`
- UP1 Corrupt Bookmark. A Bookmarks parent/child/next/previous marker points to something that isn't a bookmark, or that just isn't there at all. You may lose a few bookmarks, but the corruption will be repaired as best as we can.
- UP5 Missing named action. A "named" action is missing in the document. At worst this will prevent a hyperlink from working, as the links action is converted to a no-op
- UP6 Identical to UP5

## Appendix B - System properties

A number of system properties can be set to extract debug information or modify the operation of the library. All of these are prefixed with `org.faceless.pdf2`, and can be set either directly in Java (eg. `System.setProperty("org.faceless.pdf2.Threads", "1")`) or from the command line, eg `java -Dorg.faceless.pdf2.NoInternKeys com.yourcompany.YourClass`. This list is by no means definitive, may change without notice, and is provided here as information only.

<code>Threads</code>	How many parallel threads to use when loading the PDF in the PDFReader class. The default is the number of CPU's available to Java. Loading is not memory intensive so this can be set to the optimal level for speed if necessary.
<code>Threads.TIFF</code>	How many parallel threads to use when writing a TIFF image using the <code>PDFParser.writeAsTIFF</code> method. Writing TIFF's can be parallelized for speed, but it is a memory intensive operation. The default is one thread, but if you have a lot of memory available you can increase this if necessary, at the risk of causing an <code>OutOfMemoryError</code> .
<code>DeviceColor</code>	Convert all calibrated colors (including ICC profiles) to uncalibrated. When rendering documents containing calibrated colors (and particularly images in calibrated colors), this setting will give much faster results, possibly at the expense of color accuracy.
<code>ExperimentalDCT</code>	Use an experimental pure java DCT decoder to decode JPEG images. This appears to be working correctly for 1 and 3 color JPEG images, is considerably slower than the native code decoder supplied with Java, and uses less memory for large images.
<code>RecompressImage</code>	Cause DCT (JPEG) images to be recompressed, even if the library feels it can insert them directly into the PDF.
<code>JBIG2.jbig2dec</code>	Specifies the path for the "jbig2dec" program, the third-party software required to render PDF's containing JBIG2 images. See <a href="http://bfo.com/products/pdf/jbig2">http://bfo.com/products/pdf/jbig2</a> for more details.
<code>JBIG2.jbig2enc</code>	Specifies the path for the "jbig2enc" program, the third-party software required to create PDF's with JBIG2 compressed images. See <a href="http://bfo.com/products/pdf/jbig2">http://bfo.com/products/pdf/jbig2</a> for more details.
<code>debug.MDP</code>	Dump information about the MDP calculations when verifying MDP signatures
<code>debug.FontSubstitution</code>	When rendering PDFs, display information on Font substitution to stderr
<code>debug.Token</code>	Dump the Tokens being parsed when rendering a PDF.
<code>debug.Glyph</code>	Dump a lot of information about glyph selection when rendering a PDF.
<code>debug.JavaScript</code>	Display information about the JavaScript methods being run by the Library.
<code>debug.OpenType</code>	When loading OpenType fonts, display a lot of technical information about them (mainly relating to OpenType substitution tables)
<code>debug.PKCS7</code>	If set to a path, the raw PKCS#7 object from a digital signature or public key encrypted document will be saved to a numbered file beginning with this prefix.
<code>debug.XMP</code>	If set to a path, any XMP data in the PDF will be saved to this file.

## Acknowledgments

Acrobat, AcroForm, Acrobat Forms, PDF, Portable Document Format, Type 1 Font, PostScript and XMP are trademarks of Adobe Corporation, Inc. Java is a trademark of Sun Microsystems, Inc. Unicode is a trademark of Unicode, Inc. TrueType is a trademark of Apple Computer, Inc. PANTONE is a trademark of the Pantone corporation, Inc.

Copyright (C) 2009 Big Faceless Organization. Created with the Big Faceless Report Generator.