The Big Faceless PDF Library

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 PDF's (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 1.2 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 1.2 platform.

Features

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

- Native Unicode™ support, even in the document body. No worrying about codepages and encodings, it *just works*.
- Edit existing PDF documents with the `PDFReader` class.
- 40 and 128-bit encryption, for eyes-only documents.
- Digitally sign documents for authenticity and non-repudiation
- Full embeddding of TrueType or Type 1 fonts, with subsetting for smaller fonts.
- Full support for creating and editing AcroForms
- Japanese, Chinese and Korean font support
- Right to left and Bidirectional text is fully supported.
- Embed JPEG, PNG, GIF, 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 PDF's 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 has been verified against Acrobat 4.0 and 5.0 on Windows and Acrobat 4.05 on Linux. It's also been tested against GhostScript 5.5 and xpdf 0.91 on Linux, each of which have their limitations - we've documented these where we've found them.

Installation

Installing the library is as simple as unpacking it and adding the file `bfopdf.jar` to your `CLASSPATH`. You'll find several other files in the package, including `README.txt`, containing the overiew of the package, `docs/LICENSE.txt` containing the licencing agreement, and the `docs` and `example` directories containing the API documentation and the examples respectively.

Getting Started

The Classic Example

```
import java.io.*;
import java.awt.Color;
import org.faceless.pdf.*;

public class HelloWorld
{
  public static void main(String[] args) throws IOException
  {
    PDF p = new PDF();

    PDFPage page = p.newPage(PDF.PAGESIZE_A4);

    PDFStyle mystyle = new PDFStyle();
    mystyle.setFont(new StandardFont(StandardFont.HELVETICA), 24);
    mystyle.setFillColor(Color.black);

    page.setStyle(mystyle);
    page.drawText("Hello, World!", 100, page.getHeight()-50);

    OutputStream out = new FileOutputStream("HelloWorld.pdf");
    p.render(out);
    out.close();
  }
}
```

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.pdf`, 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 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 has 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 `setCanvas` 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.

`setFillColor`	*Text, Graphics*	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.
`setLineColor`	*Text, Graphics*	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 `setFontStyle` method is used to turn on text outlines.
`setFont`	*Text*	Sets the font and the font size
`setTextAlign`	*Text*	Sets the alignment of the text - left, centered, right or justified
`setTextUnderline`	*Text*	Whether to underline the text. See also ~~`setTextStrikeout`~~
`setTextLineSpacing`	*Text*	Determines how far apart each line of text is - single, double spaced, line-and-a-half or any other value.
`setTextIndent`	*Text*	Sets how far to indent the first line of text, in points.
`setLineWeighting`	*Graphics, Text*	Set the width of any lines drawn using the Line color, in points (including outlined text).
`setLineDash`	*Graphics*	Set the *pattern* to draw lines with. Normally lines are solid, but you can draw dashed lines using this method.
`setLineJoin/` `setLineCap`	*Graphics*	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 above, 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

New in version 1.1.5 of the library is support for 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 colorspaces? Here's an example:

```
import java.awt.color.*;
import java.awt.Color;

public void colorDemo(PDFPage page)
    throws IOException
{
    Color red1 = Color.red;    // sRGB colorspace, red=100%

    // Create the same red in an ICC colorspace loaded
    // from a file. Here we use the NTSC color profile.
    ICC_Profile prof = ICC_Profile.getInstance("NTSC-Profile.icc");
    ICC_ColorSpace ntsc = new ICC_ColorSpace(prof);
    float[] comp = { 1.0, 0.0, 0.0 };
    Color red2 = new Color(ntsc, comp, 0);

    // Create a color in the device-dependent CMYK
    // colorspace, supplied with the library.
    Color yellow = CMYKColorSpace.getInstance().getColor(0,0,1,0);

    // Create a spot color from the PANTONE™ range
    SpotColorSpace spot = new SpotColorSpace("PANTONE Yellow CVC", yellow);
    Color spotyellow = spot.getColor();
}
```

This may seem a complicated example, but first it demonstrates using four different colorspaces in 20 lines, and if part of 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 and TIFF images may have a colormap embedded, which the library will pick up and use automatically. GIF images don't and the JPEG format handles colors a little differently, but for all of these image formats and for `java.awt.Image` images as well, the colorspace can be specified by calling the `setColorSpace` on the image. Generally speaking though you're not going to need to do this - 99% of the time it's best to go with the default ColorSpace used by the image.

Patterns

A new feature in version 1.1 is the ability to fill a shape with a pattern. Currently you can choose from several pre-defined patterns, and we plan to add more (and the ability to define your own) in the future. The `ColorPattern` class implements `java.awt.Paint`, and can be used to fill text or shapes.

Patterns work with with text as well!

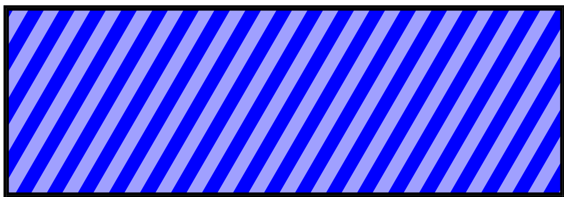
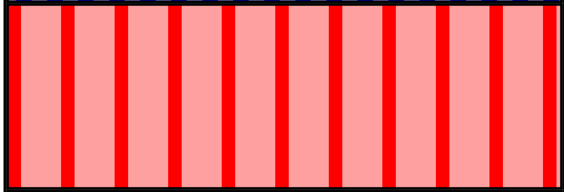
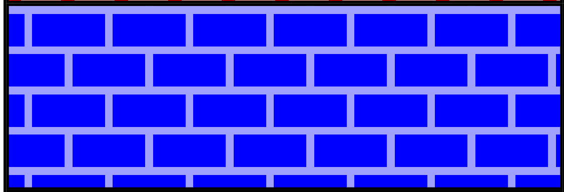
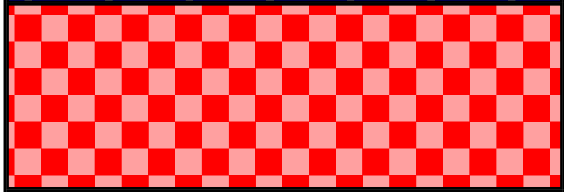
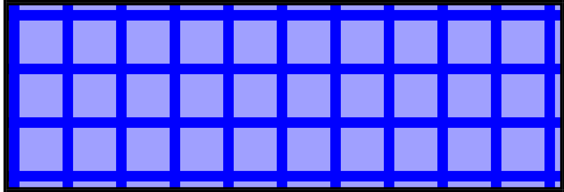
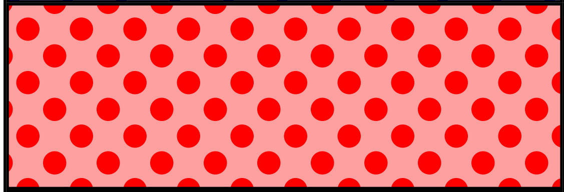
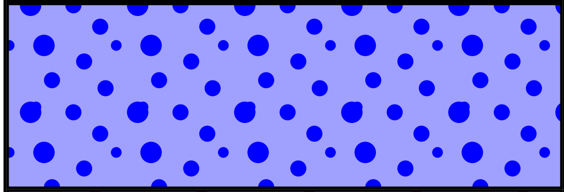
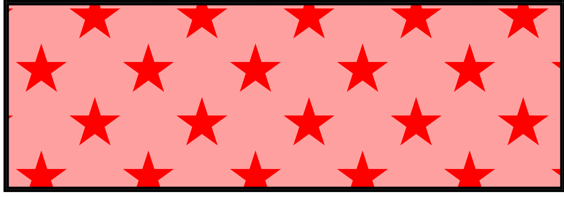
	The `ColorPattern.stripe` method returns a striped pattern. The angle and width of each stripe can be set separately - here we've set the angle to 30°, and the background and foreground stripes to 5 points wide each.
	Here is another example of a `ColorPattern.stripe`, with vertical stripes and a different width for each stripe color.
	This pattern is returned from the `ColorPattern.brick` method. The width and height of each brick can be set separately. For budding brickies, this style of brick laying is known as a running pattern bond.
	This check pattern is created by the `ColorPattern.check` method.
	This pattern is created by the `ColorPattern.grid` method. The width of the line and the width of the space between the lines can be set separately.
	A "spot" pattern similar to the pattern used for halftoning in newspapers can be created with the `ColorPattern.spot` method.
	A different kind of "spot" pattern, containing a number of different size random spots can be created with the `ColorPattern.polka` method. The average size of the spots can be set
	Finally, a pattern of repeating 5-pointed stars (like those on the US flag) can be created with the `ColorPattern.star` method. Again, the size of each star can be set

Table 2 - Color patterns

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, the document can contain TrueType™ or Adobe "Type 1" fonts, which can either be *embedded* or *referenced*. 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.

Standard 14 fonts

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

Embedding a font guarantees it will be available to the PDF viewer, but increases the filesize of the document. In version 1.1 of the library, the ability to *subset* TrueType fonts was added (we hope to extend this to Type 1 fonts in a future version). This means that when embedding a font, only the characters that are actually required are embedded, and the rest (along with other redundant information) is stripped from the font. This can result in huge space savings, particularly on large fonts. The `TrueTypeFont` and `Type1Font` classes cover these two types of font.

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 - this means even larger files when you're using embedded fonts.

So how do you use the 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.

```
1. public void showText(PDFPage page)
2.     throws IOException
3. {
4.     PDFFont myfont = new TrueTypeFont(new File("myfont.ttf"), true);
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. }
```

The only line that needs to be changed is line 4. TrueType™ is a format developed by Apple and Microsoft, and many TrueType fonts are marked as "Apple" or "Windows" specific - the library works with either.

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, \u20AD 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 lotmore 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

```
public void formatText(PDFPage page)
{
    PDFStyle plain = new PDFStyle();
    plain.setFillColor(Color.black);
    plain.setTextAlign(PDFStyle.TEXTALIGN_JUSTIFY);
    plain.setFont(new StandardFont(StandardFont.HELVETICA), 11);

    PDFStyle bold = (PDFStyle)plain.clone();
    bold.setFont(new StandardFont(StandardFont.HELVETICABOLD), 11);

    page.beginText(50,50, page.getWidth()-50, page.getHeight()-50);
    page.setStyle(plain);
    page.drawText("This text is in ");
    page.setStyle(bold);
    page.drawText("Helvetica Bold.");
    page.endText(false);
}
```

For complex 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, Type1 and the Standard fonts on the same line.

The example above shows the methods 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.

The `continueText` method allows you to continue the text somewhere else. It works exactly like the `beginText` method, but takes an additional parameter - a page. The method will display any text that overflowed from the last text box on that page before it calls the `beginText` method to allow you to continue adding text.

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.

```
public void formatText(PDFPage page)
{
    Locale locale = Locale.getDefault();

    PDFStyle plain = new PDFStyle();
    plain.setFillColor(Color.black);
    plain.setTextAlign(PDFStyle.TEXTALIGN_JUSTIFY);
    plain.setFont(new StandardFont(StandardFont.HELVETICA), 11);

    PDFStyle bold = (PDFStyle)plain.clone();
    bold.setFont(new StandardFont(StandardFont.HELVETICABOLD), 11);

    LayoutBox box = new LayoutBox(page.getWidth()-100);
    box.addText("This text is in ", plain, locale);
    box.addText("Helvetica Bold.", bold, locale);
    page.drawLayoutBox(50, page.getHeight()-50);
}
```

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 co-ordinates 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 co-ordinate 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.

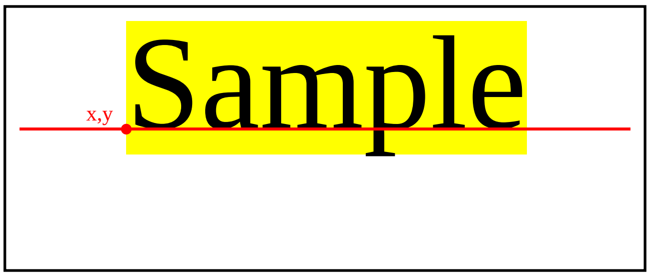


Figure 1a - Positioning using `beginText`

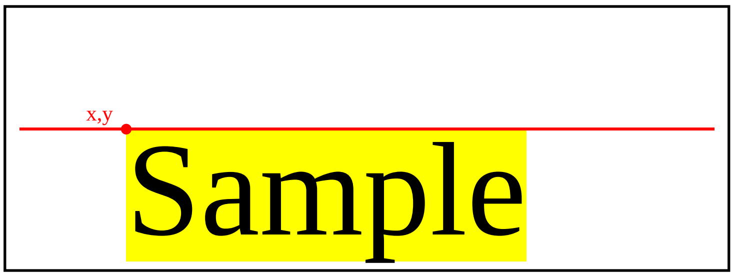


Figure 1b - Positioning using `drawLayoutBox`

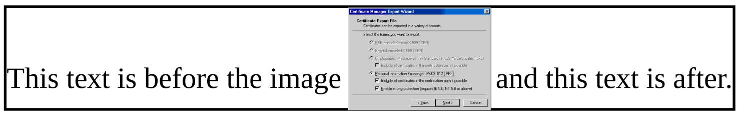
Mixing text and images

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.

```
LayoutBox box = new LayoutBox(width);
box.addText("This text is before the image ", mystyle, locale);
LayoutBox.Box image = box.addBoxInline(imgwidth, imgheight, PDFStyle.TEXTALIGN_BASELINE);
box.addText(" and this text is after.", mystyle, locale);

page.drawLayoutBox(x, y);
page.drawImage(image, x+box.getLeft(), y+box.getLeft(), x+box.getRight, y+box.getBottom());
```

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 co-ordinates 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.


This text is before the image and this text is after.

Two futher 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.

```
PDFStyle background = new PDFStyle();
background.setFillColor(Color.blue);

LayoutBox box = new LayoutBox(width);
box.addText("The following phrase will be drawn on ", mystyle, locale);
LayoutBox.Text text = box.addText("a blue background", mystyle, locale);
box.addText(", but now we're back to normal.", mystyle, locale);

page.setStyle(background);
do {
  page.drawRectangle(x+text.getLeft(), y+text.getLeft(), x+text.getRight, y+text.getBottom());
  text=text.getNextTwin();
} while (text==null);
page.drawLayoutBox(x, y);
```

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.

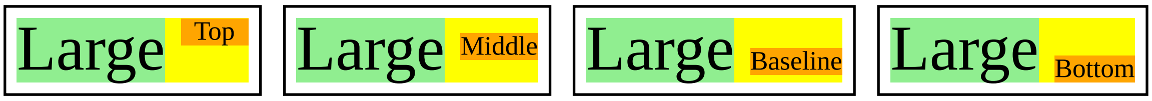


Figure 3 - Vertical alignment examples

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 7 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.

```
PDFStyle background = new PDFStyle();
background.setFillColor(Color.blue);

LayoutBox box = new LayoutBox(width);
box.addText("The following text will be drawn  ", mystyle, locale);
LayoutBox.Box box = box.addBoxRight(50, 50, LayoutBox.CLEAR_NONE);
box.addText("around the box to the right. When it grows beyond that box\
                 it will automatically fill the width of the line", mystyle, locale);

page.drawLayoutBox(x, y);
page.drawRectangle(x+text.getLeft(), y+text.getLeft(), x+text.getRight, y+text.getBottom());
```

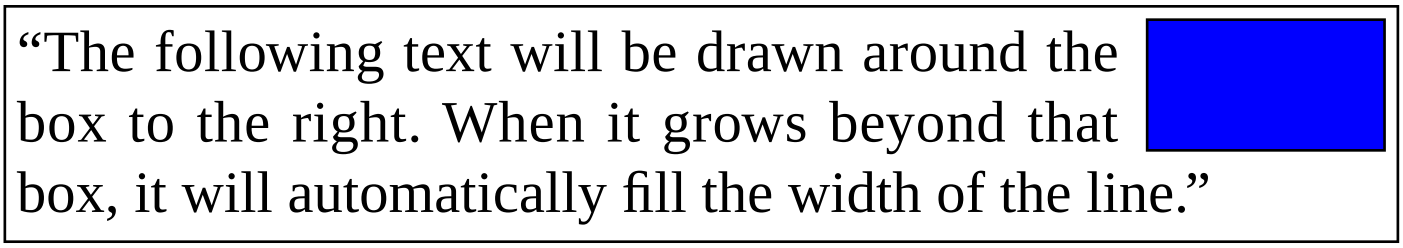

“The following text will be drawn around the box to the right. When it grows beyond that box, it will automatically fill the width of the line.”

Figure 4 - 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.

```
box.addBoxRight(50, 50, LayoutBox.CLEAR_RIGHT);
box.addBoxRight(50, 50, LayoutBox.CLEAR_RIGHT);
box.addBoxRight(50, 50, LayoutBox.CLEAR_NONE);
box.addText("Text text ...", mystyle, locale);
box.addBoxLeft(50, 50, LayoutBox.CLEAR_NONE);
box.addBoxLeft(50, 50, LayoutBox.CLEAR_NONE);
box.addBoxLeft(50, 50, LayoutBox.CLEAR_NONE);
box.addBoxLeft(50, 50, LayoutBox.CLEAR_LEFT);
box.addText("More more ...", mystyle, locale);
```

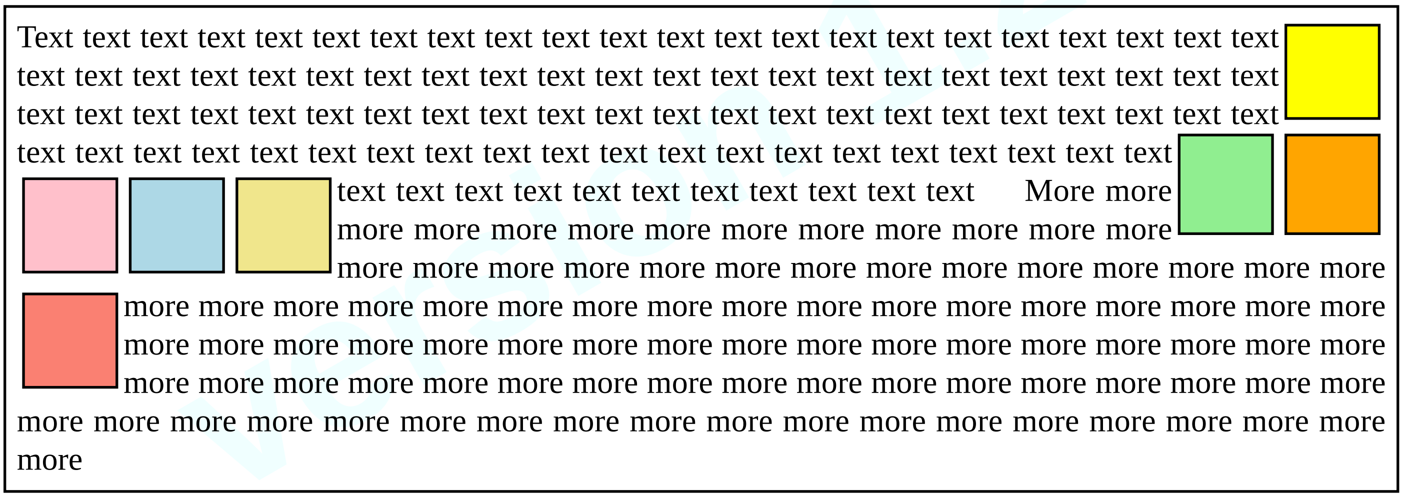

Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text More more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more more

Figure 5 - Things you can do with the "clear" flag

Advanced text layout features

Quotes

The `requote` method was added to the `PDFPage` class in version 1.1, to allow substitution of 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 current locale. To use it, simply call `page.drawText(page.requote(mystring))` instead of `page.drawText(mystring)`.

With kerning

AWAW

Without kerning

A W A W

Ligatures, with & without

ﬁ fi

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: **T H R E E   S H O R T   W O R D S**. 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, ie 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 worlds 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, German, French, Spanish, Norwegian, Swedish, Danish, Finnish, Italian, Portuguese, Catalan (although the "L with dot" character is missing), Basque, Dutch (no "ij" ligature), Albanian, Estonian, Rhaeto-Romance, Faroese, Icelandic, Irish, Scottish, Afrikaans, Swahili, Frisian, Galician, Indonesian/Malay and Tagalog. With an appropriate embedded font, most other languages will work, although remember that Java up to and including version 1.4 is limited to the Basic Multilingual Plane of Unicode - the first 65,536 characters.

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 overide characters are supported as per that specification.

Japanese, Chinese (simplified and traditional) and Korean are also newly supported as of version 1.1, although in horizontal writing mode only. Rather than forcing all it's Asian users to embed huge fonts in their documents, Adobe have thought ahead and defined several "standard" fonts (known variously as CJK or CID fonts) which are available with the localised versions of it's 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 is supported in the MSung font, although you'll need a recent language pack to view those characters (Acrobat 5.0 or later)

The following languages have poor or non-existant 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 Devengari, Bengali and friends yet, owing to the difficulty in finding a decent font with the correct Unicode encoding, and because we don't know anyone that speaks 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.
- 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 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.

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 Bidirectional 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 *may* 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 *may* 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 *should* 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 `page.drawText("1\u20442")` 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	Unicodes 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

```
public void showImage(PDFPage page)
    throws IOException
{
    PDFImage img = new PDFImage(new FileInputStream("myimage.jpg"));
    int width = img.getWidth();
    int height = img.getHeight();
    page.drawImage(img, 100, 100, width, height);
}
```

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 and TIFF 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. The library is limited to "masked" transparency, as used in GIF and some PNG images, where a single color can be flagged as transparent. Limitations in Acrobat and/or PostScript causes large images to be rendered without transparency when printed - 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.

big.faceless.org
big.faceless.org

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.

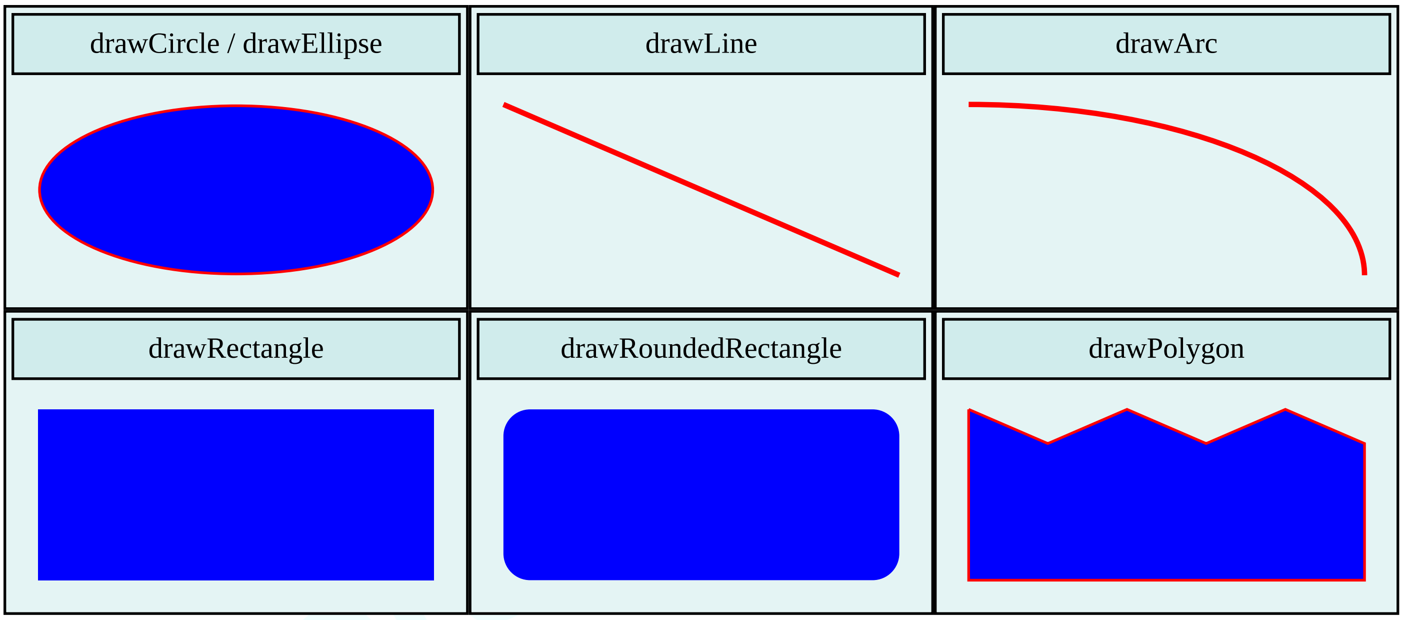
Vector Graphics

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.

```
public void drawCircle(PDFPage page)
{
    PDFStyle style = new PDFStyle();
    style.setLineColor(Color.black);
    style.setFillColor(ColorPattern.star(Color.red, Color.blue, 20));
    page.setStyle(style);

    page.drawCircle(page.getWidth()/2, page.getHeight()/2, 100);
}
```

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



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.

```
public void drawPath(PDFPage page)
{
    PDFStyle style = new PDFStyle();
    style.setLineColor(Color.black);
    style.setFillColor(ColorPattern.star(Color.red, Color.blue, 20));
    page.setStyle(style);

    page.pathMove(100, 100);         // Always start with pathMove
    page.pathLine(100, 200);
     page.pathBezier(130,300, 160,100, 200, 200);
     page.pathLine(200, 100);
     page.pathClose();
     page.pathPaint();
}
```

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 `setCanvas` 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. To go further than that, the `rotate`, `translate` and `scale` methods can also change the way co-ordinates map into the PDF document. The rotate method in particular is useful - notice the watermark on each page of this document.

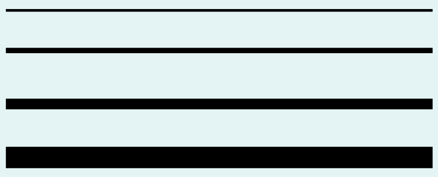
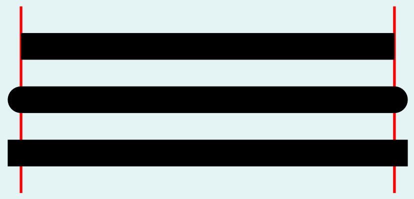
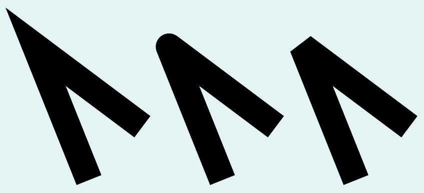
Because these routines change the way the page is represented in a very fundamental way, it's easy to get unusual results - use with care. Additionally, because of the way annotations are stored in a PDF document, if you rotate, scale or translate the page they won't continue to work as expected.

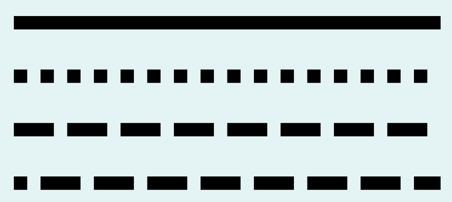
Other useful functions do with with the graphics state are `save`, `restore` and `undo`. These allow you to save the current state of the page to a "stack" - it's good practice to save before doing a transformation and restore afterwards rather than apply a reverse transformation. The `undo` feature is useful to test before painting. For example, here's how to center a barcode on the page (we don't know how big it is until it's drawn):

```
public void centerBarcode(PDFPage page)
{
    page.save();
    float size = drawBarcode(PDFPage.BARCODE128, "code here", 0, 0, true);
    page.undo();                     // X and Y don't matter, it's going to be undone
    drawBarCode(BARCODE128, "code here", (page.getWidth()-size)/2, 50, true, 1);
}
```

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).

	The `setLineWidth` 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 *thinnest line possible on that device*. 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 it's device-dependant nature, this is not recommended by Adobe.
	The `setLineCap` method changes how the end of lines are drawn. The default option is *Butt cap*, where the stroke is squared off at the endpoint of the path and there is no projection beyond the end. Second is *Round cap*, causing a semicircle with a diameter equal to the line width to be drawn at the end of the line. Finally the *Square cap*, where the stroke continues beyond the endpoint of the path for a distance equal to half the line width, and is then squared off.
	The `setLineJoin` method changes how two line segments are joined. The default is a *Miter join*, 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 *Round join*, 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 *Bevel join*, where the two segments are finished with butt caps and the resulting notch is filled with a triangle.



The `setLineDash` method allows you to change the line into a sequence of *on* and *off* line segments. The default is a solid line, the result of calling `setLineDash(0,0,0)`. The second example here is a sequence of 5 points on, 5 points off, from `setLineDash(5,5,0)`. The third example is 15 points on, 5 points off, from `setLineDash(15,5,0)`. The final example is the same as number 3, but the *phase* of the pattern has been changed by calling `setLineDash(15,5,10)`. 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.

Importing and Editing existing documents

In version 1.1.12 we added the `PDFReader` class, which allows existing PDF documents to be opened, parsed, and optionally edited and written out. Although this class is supplied with the package, it is part of the *Extended Edition* of the library.

So how do you import and edit a PDF? Here's the `HelloWorld` example we showed earlier, modified to add the text "Hello, World!" to the first page of an existing PDF document. Significant lines are in bold.

```java
import java.io.*;
import java.awt.Color;
import org.faceless.pdf.*;

public class HelloWorld
{
  public static void main(String[] args) throws IOException
  {
    PDFReader reader = new PDFReader("input.pdf");
    PDF pdf = new PDF(reader);

    PDFPage page = pdf.getPage(1);

    PDFStyle mystyle = new PDFStyle();
    mystyle.setFont(new StandardFont(StandardFont.HELVETICA), 24);
    mystyle.setFillColor(Color.black);

    page.setStyle(mystyle);
    page.drawText("Hello, World!", 100, page.getHeight()-50);

    OutputStream out = new FileOutputStream("output.pdf");
    p.render(out);
    out.close();
  }
}
```

As you can see, the only differences are the loading of the original document into a `PDFReader`, which is then passed to the PDF constructor. From there the PDF can be used in exactly the same way as if it had been created by the library - you can edit the pages, annotations, bookmarks, document information and so on.

Note that this access is limited in some respects. First, there is currently no way to extract the text, fonts or images from a page. We plan to add support for these features, although it's important to note that extracting text is not always an exact science, as often information is lost when the text is added to the PDF.

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 hyperlinks' action removed. The same applies to bookmarks linking to that page (which are *not* copied when a page is copied).

The following methods were added to the PDF library with the manipulation of existing PDF documents in mind.

`PDF.getPages`	This method returns all the pages from the PDF document as a standard Java List - this can then be altered using any of the normal `java.util.List` methods to add, delete or reorder pages as necessary - even pages from other documents can be added to the list, to append PDFs together. Using this method of joining pages is **considerably faster** than calling `drawPage`, so stick with this one if you can. The `example/Concatenate.java` example supplied with the package demonstrates this.
`PDFPage.drawPage`	In the same way that you can draw an Image onto a page, this method allows you to draw another Page onto a page! This allows for some interesting effects - the most obvious of which is printing a document in "2-up" format. The `example/PageStitch.java` example supplied with the package demonstrates this functionality.
`PDFPage.seekStart` `PDFPage.seekEnd`	Although the contents of the current page cannot be changed or removed, it can be added to. Typical uses would be adding watermarks, barcodes or text. These two methods allow you to determine whether to add these new items before or after the current content of the page is drawn - the `seekStart` method can be called to add items before the current page content, so it appears "under" the content. The `seekEnd` method is the reverse of this and adds the new content after, or on top of the existing content. Note that this is the default way to add content, so you don't need to call it unless you've previously called `seekStart`.

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 first loaded, by calling the `pdf.setOpenAction` method
- An action can optionally be run when a page is opened or closed, by calling the `page.setOpenAction` and `page.setCloseAction` methods.
- An action can be used with a `PDFAnnotation` to act as a hyperlink
- A form element can have actions specified for various *events*, such as the mouse entering the elements rectangle or the button being pressed.

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 roughly corresponds to selecting an action from the drop-down menus in the application, 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.

Six other action types are mostly used with forms - the `formSubmit`, `formReset`, `formImportData`, `formJavaScript`, `showElement` and `hideElement`. These are usually applied to a form elements annotation via the `PDFAnnotation.setEventAction` method, and are covered separately in the section on forms.

Hyperlinks

New in version 1.1 is the ability to add *annotations* to the document, a special class of which is the hyperlink. Annotations, defined by the `PDFAnnotation` class, are not really part of the page but sit apart from the page content. Because of this, they're potentially difficult to integrate into your document, particularly when you want to make a particular phrase in the middle of a sentence a hyperlink (the situation is usually worse when the phrase wraps at the end of the line or page).

We've got around this by creating two different ways to add hyperlinks to your document. The first is by specifying the position on the page that the viewers can click in. This method works well for making an image a hyperlink, as they're already positioned this way. All you need to do is create a new `PDFAnnotation`, then call it's `setRectangle` method to determine where on the page the user has to click.

```
public void showImageLink(PDFPage page, PDFImage image)
{
    PDFAction action = PDFAction.goToURL(new URL("http://big.faceless.org"));

    page.drawImage(200, 200, image.getWidth(), image.getHeight());
    PDFAnnotation link = PDFAnnotation.link(action, false);
    link.setRectangle(200, 200, image.getWidth(), image.getHeight());
    page.addAnnotation(link);
}
```

The `link` annotation always 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. The second parameter (`false` in this example) determines whether to draw a border around the annotation.

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:

```
public void showTextLink(PDFPage page)
{
    PDFAction action = PDFAction.goToURL(new URL("http://big.faceless.org"));

    page.beginText(50,50, page.getWidth()-50, page.getHeight()-50);
    page.drawText("Thank you for choosing ");
    page.beginTextLink(action, PDFStyle.LINKSTYLE);
    page.drawText("the Big Faceless PDF Library");
    page.endTextLink();
    page.endText(false);
}
```

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.

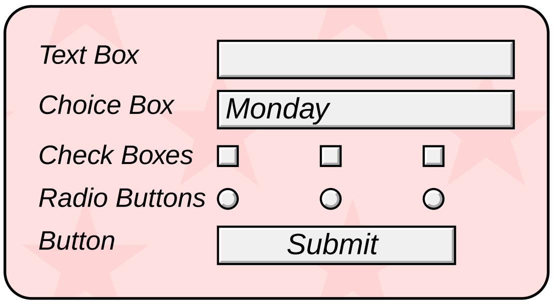
Other Annotations

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 *text* and *rubber-stamp* annotations. The *text* annotation is the electronic equivalent of adding a Post-It® note to your document. This can be clicked on by the viewer to view it's contents or dragged about the page to a different location.

These can be created by calling the `PDFAnnotation.text` method, which, given the title and the contents of the note, creates a new Text annotation which can be added to the page using the `addAnnotation` method in the same way as the first hyperlink example above. The rubber-stamp annotation can be created by calling the `PDFAnnotation.stamp` method, where you can select from one of 14 names stamps (like Draft, Confidential). It is added to the page in the same way.

Forms



Users of the *Extended-Edition* of the library can read and create PDF forms, also called "AcroForms". 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.

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 "Widget" - 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 `setEventAction` method can be called on the elements annotations, 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:

```
PDF pdf = new PDF(new PDFReader(new FileInputStream("template.pdf")));
Form form = pdf.getForm();

FormText name = (FormText)form.getElement("name");
System.out.println("The name field was set to "+name.getValue());

name.setValue("J. Quentin Public");
```

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 consistant interface, although the Radio Button is a little different as it's the only element in the library with more than once annotation. Starting with the basics though, here's how to add a text field to a new PDF.

```
PDF pdf = new PDF();
PDFPage page = pdf.newPage(PDF.PAGESIZE_A4);
Form form = pdf.getForm();

FormText text = new FormText(page, 100, 100, 300, 130);
form.addElement("mytextfield", text);

pdf.render(outputstream);
```

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 `Form.setDefaultStyle` method, or you can just update a specific field by calling the `setStyle` method for that element.

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 audio sample
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

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. GET and POST are available - although GET should be avoided, for reasons described in the API docuemenatation. Other options include 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 recomended for anything but the simplest JavaScript.

Finally, the `showElement` and `hideElement` actions. These can be used to show or hide an existing annotation, although our tests indicate it will probably only work with "widget" annotations (those created by form elements). 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 element to determine what to do when the the user clicks on it. However, there are several other possibilities. If you retrieve the list of annotations for a form element by calling the `FormElement.getAnnotations` method, each of these annotations can have several *events* attached to it by calling the `PDFAnnotation.setEvent` method. These events 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 purposed - 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". There are a number of handlers on the market - at the moment we are aware of handlers by Adobe, VeriSign, Entrust, Baltimore and CIC. Of these, all except the CIC handler revolve around a public/private key infrastructure or "PKI". We currently support signing and verifying documents intended for the Adobe and VeriSign® handlers.

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
```

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 it's 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"));
```

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.

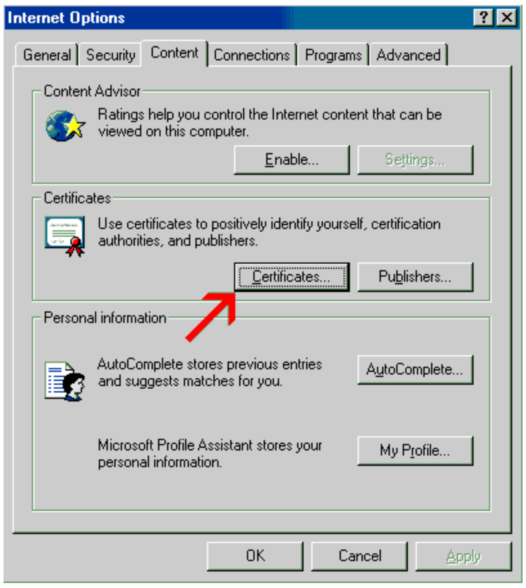
Signing documents with the VeriSign Handler

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.

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".


Internet Options

General | Security | Content | Connections | Programs | Advanced

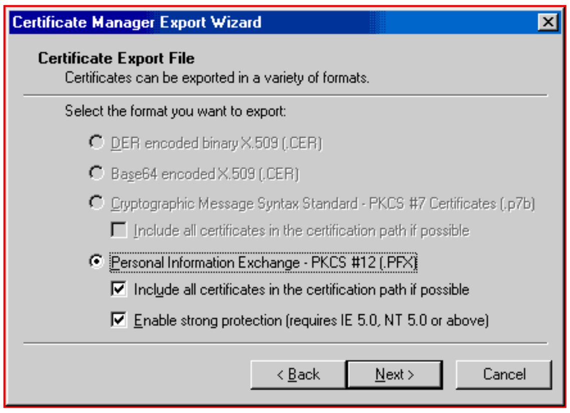
Content Advisor
Ratings help you control the Internet content that can be viewed on this computer.
Enable... | Settings...

Certificates
Use certificates to positively identify yourself, certification authorities, and publishers.
Certificates... | Publishers...

Personal information
AutoComplete stores previous entries and suggests matches for you. AutoComplete...
Microsoft Profile Assistant stores your personal information. My Profile...

OK | Cancel | Apply

Figure 1


Certificate Manager Export Wizard

Certificate Export File
Certificates can be exported in a variety of formats.

Select the format you want to export:
- DER encoded binary X.509 (.CER)
- Base64 encoded X.509 (.CER)
- Cryptographic Message Syntax Standard - PKCS #7 Certificates (.p7b)
  - Include all certificates in the certification path if possible
- [✓ Personal Information Exchange - PKCS #12 (.PFX)]
  - [✓ Include all certificates in the certification path if possible]
  - [✓ Enable strong protection (requires IE 5.0, NT 5.0 or above)]

< Back | Next > | Cancel

Figure 2

So now you have your private key and accompanying certificates as PKCS#12 keystore, an industry standard format which is not directly supported by Java - although support can be added by installing an appropriate Java Cryptography Extension (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). Version 1.12 or newer is required - 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.

Of course, if you've already got a digital signature (for signing JAR files, for example) then you can use this and skip most of the steps outlined above. Once you can load a key into a Java keystore, the actual signing of the document is just the three lines in bold in the above example.

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:

```
import java.security.KeyStore;
import java.security.cert.Certificate;

PDF pdf = new PDF(new PDFReader(input));                 // Load the PDF

KeyStore trusted = FormSignature.getDefaultKeyStore();  // Trusted Certificates

```

```
 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. }
```

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 proprietry 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#11 keystore format, or as an `.FDF` file which can be loaded using the `FormSignature.loadFDFKeyStore` method.

Things to be aware of with Digital Signatures

1. 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.
2. 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.
3. Although PDF documents may potentially have more than one digital signature, we currently do not support signing with more than one.
4. In the current release it's not possible for signatures to have an appearance on the page. This is planned for a later version of the library.
5. The `Sign.java` and `Dump.java` examples supplied with the library provide working code examples of signing and verifying signatures respectively.

6. 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.

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 marvellous 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

Since version 1.1 we've added the ability to encrypt the document to the library. The encryption method used by default is a 40-bit symmetric key cipher, compatible with that used by Acrobat 3.x and above. The 128-bit cipher added in Acrobat 5.0 can be used by calling the `PDF.setEncryptionAlgorithm` method, although obviously documents encryted with the 128-bit cipher can only be read by Acrobat 5 and later. Public/private key encryption (as supported by the Entrust PKI architecture and it's Acrobat plugin) is not supported.

Adding encryption to the document is easy - in fact, often you won't even realise you've done it! The downside is that it's a *relatively* slow procedure, worth remembering if raw speed is a priority.

There are four methods in the `PDF` class that enable encryption.

- The `setPassword` method adds a password to the document, which needs to be entered before the document can be opened.
- The `setAccessLevel` method determines what level of access is available when the document has been opened. This method actually encrypts the document, although it's not obvious because (unless you've also called `setPassword`) no password is required to open the document.

  For example, to allow a user to do everything to a document except print it, add the following piece of code:

  ```
  pdf.setAccessLevel(PDF.ACCESS_ALL - PDF.ACCESS_PRINT);
  ```

  Please remember that enforcing the restrictions set by the `setAccessLevel` method is up to the viewer application, and that these restrictions are fairly easy to remove. Consequently this method should be thought of as *advisory only*.

- The `setSecurityPassword` method can be used when setting a normal password, to provide a second password which gives "security" privileges to the viewer, allowing them to change the access levels or the user password. Often this is left blank.
- The `setEncryptionAlgorithm` method sets the encryption algorithm that is to be used. By default the document is not encrypted, unless one of the `setPassword` or `setAccessLevel` methods is called, in which case the 40-bit cipher is used by default. This can be changed to the 128-bit cipher by calling this method with the parameter `PDF.ENCRYPT_128BIT`.

Bar Codes

Also added in version 1.1 was the ability to add bar codes to the document. 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 varient, depending on the data to be encoded. 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 labelling (it's the barcode on all your groceries and books). It must contain 13 digits, the last of which is a checkdigit.

0 12345 67890 5

The UPC-A barcode is the US-only subset of EAN-13. Although all scanning equipment in the US should be updated to recognise EAN-13 codes by 2004, in the meantime the traditional UPC-A codes can also be printed.

A12345B

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.

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 `setLayout`, `setViewerPreferences` and `setOpenAction` methods in the `PDF` class control various settings to do with what to do with the document when it opens.

Document Compression

PDF Documents are automatically compressed using the "Flate" algorithm (implemented by the `java.util.zip` package). Most of the time document creators won't have to worry about this as it's done by default, but for debugging the compression can optionally be turned off by calling the `setFilter` method of the `PDFPage` class.

XML Metadata (Adobe XMP™ support)

PDF documents supporting the PDF 1.4 specification can include XML metainformation 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

Problems, limitations and future enhancements

There are a number of limitations with the library, many of which we hope to overcome in future versions, and some which we don't expect we'll ever cover.

- Features we plan to add in future versions
  - Subsetting of Type 1 fonts, and improved subsetting of TrueTypes for even smaller documents
  - Vertical writing for Chinese, Japanese and Korean (Mongolian is not planned).
  - Support for Adobes Compact Font Format (CFF), and by extension the full OpenType™ font format
  - User definable patterns
  - User definable hyphenation rules
  - Linearization (optimizing for web download) of PDF's. We've voted it "most unpleasant specification ever seen", so it may be a while off
  - Alpha transparency (new in Acrobat 5.0)
  - Embedded of CJK TrueType fonts
- Features we don't ever plan to support
  - Multiple Master and Type 0 fonts. Does anyone actually use them?

We're open to suggestions on the order these are attacked, so if you have a pressing need drop us a line and we'll see what we can do. Priority will naturally be given to those with support contracts.

Incompatibilities

Even Adobes own acrobat viewer doesn't adhere to the PDF specification in some places, so it's important to take the "portable" from "Portable Document Format" with a pinch of salt. Here are a list of problems we've found with other PDF viewers - we've tested with ghostscript 5.5, ghostscript 7 and xpdf 0.91.

- TrueType fonts don't work in Xpdf 0.91 and earlier if they've been rewritten. Because the rewriting process is transparent to the user, it's hard to tell when it's going to fail, but generally speaking if you use any characters outside the standard Windows codepage, you're probably going to see a lot of question marks. If Xpdf 0.91 or earlier support is required, we recommend using Type 1 fonts instead.
- Neither ghostview or Xpdf support Bookmarks
- Xpdf recognises annotations, but support seems incomplete. GhostView doesn't recognise them at all
- Sound support is fairly poor even in Acrobat. We mentioned this earlier, but try and stick with .WAV files, and don't expect them to work on UNIX. We haven't tested a Macintosh ourselves but have been told it doesn't work, at least under OS X.
- Patterns don't work in Xpdf 0.91 or ghostscript 5.5. They do work in ghostscript 7, but when converted to PostScript the file is enormous - bigger than it needs to be, we think (although we're no PostScript wizards). God only knows what will happen if you try and print it.

- The Standard Chinese, Japanese and Korean fonts don't work in ghostscript, although they do work in Xpdf (and out of the box too, which took us by surprise, as we didn't even think we had them installed!)

F.A.Q.

How can I add to or edit a previously created PDF document

As of version 1.1.12, yes. The `PDFReader` class allows existing PDF documents to be loaded and edited. This class is available under a separate license.

How can I create tables and lists?

Under the basic PDF library, you have to do this manually (i.e it's not easy). Chiefly this is because of the complexity required to cover all the bases - the API would need to cover borders, padding, margins, "rowcount" and "colcount" to make cells cover more than one row, images in cells and so on. The API would be horrendous. Instead, we're building this functionality into the Report generator (which this document was created in) - an XML front-end to the library. Beta versions are currently available from the website.

How can I create headers and footers on each page?

This is a pretty generic type of thing to do - add a standard item to every page. Because of that we haven't made a specific routine to add the pagenumber, a watermark and so on. However, it's easy to make your own method to do this - here's one that sets the background color and adds the page number, but you can add watermarks, images, whatever you need.

```
public PDFPage getNextPage(PDF pdf)
{
    PDFPage page = pdf.nextPage(PDF.PAGESIZE_A4);

    // Set the background color to yellow - just draw a rectangle over the whole page
    PDFStyle colorstyle = new PDFStyle();
    colorstyle.setBackgroundColor(Color.yellow);
    page.setStyle(colorstyle);
    page.drawRectangle(0, 0, page.getWidth(), page.getHeight());

    // Add the page number at the bottom of the page, centered
    PDFStyle footerstyle = new PDFStyle();
    footerstyle.setFont(new StandardFont(StandardFont.HELVETICA), 8);
    footerstyle.setTextAlign(PDFStyle.TEXTALIGN_CENTER);
    page.setStyle(footerstyle);

    page.drawText("Page "+page.getPageNumber(), page.getWidth()/2, 30);
    return page;
}
```

How can I put a watermark or background image on the page

See the previous question. For a background image (it's going to be a big document...) try `page.drawImage (image, 0, 0, page.getWidth(), page.getHeight())`. Watermarks are generally just text in a light gray.

I am getting errors with Acrobat 3.0

Although most documents created by this library are compatible with Acrobat 3.0, you shouldn't rely on this - officially, we only support Acrobat 4.0 and up. Having said that, if you want your documents to work with Acrobat 3, steer clear of using non-ASCII characters in TrueType fonts and you should have no problems.

How can I insert a PCX/BMP/Other format image?

You need to load the image as a `java.awt.Image` through another library before you can insert it. We recommend the **Jimi** package, available from http://java.sun.com/products/jimi, which can load BMP's, PCX's, and many other formats.

A character isn't rendered in the document properly

This is probably because it's not in the font. The `CharacterMap` example supplied with this package will dump a complete list of every character in the font with it's Unicode number, so you can see if the character exists. Or check the `isDefined` method in the font. It may also be a viewer problem - xpdf 0.91 on UNIX is known to have problems displaying non-standard characters from TrueType fonts.

How can I find out which characters are available in a font

The `CharacterMap` example supplied with this library builds a Unicode-style character map of all the characters defined in a font. The `map-xxx.pdf` files in the `docs` directory show the characters available in the standard built-in fonts.

I printed some text but only the first few lines came out

Did you check the return value from `drawText` and `endText`? If it's returning -1, you've run out of space in the text box.

Where can I find good Unicode fonts?

There are a few around. Many of the newer Microsoft TrueType fonts like **Times** or **Tahoma** have a large number of glyphs, and for some the license appears to state that they can be embedded in your documents (provided you own a copy of Windows). While testing the library we also used *CaslonRoman-Unicode*, available from http://bibliofile.mc.duke.edu/gww/fonts/Unicode.html. It covers Cyrillic, Greek, Thai, Hebrew, Armenian, Japanese (kanas only) and many other symbols.

I tried the Arial Unicode font that comes with Microsoft Office 2000 and it crashed

I'm not surprised - it contains over 60,000 glyphs and is 28 megabytes in size. To subset the font you'd need to allocate about 60 megabytes off the heap! You could try upping the heap size on your JVM, or even better try something a little smaller!

What's involved in upgrading from version 1.0 to 1.1

Full details of what to do are in the CHANGELOG file, but you won't have to worry unless you've been calculating the number of lines of text displayed (by adding up the return values from `drawText`), changing the linespacing by calling `setTextLineSpacing` or drawing text with outlines by setting the Line color.

What's involved in upgrading from version 1.1 to 1.2

Again, check the CHANGELOG for details, but for 99% of users, all that's required is to recompile your source code, as a couple of method signatures have changed slightly.

What's the story on PDF documents and the LZW patent?

Early PDF documents (those created with Adobe Distiller 1.0, for example) used a different algorithm for compressing PDF documents than newer versions. This algorithm, known as LZW, was patented by Unisys, who since 1996 have been demanding royalties for it's use. The patent applies in the US, Canada, the UK, France, Germany, Italy and Japan. Although it's never been tested in court, patent lawyers have been consulted about the validity of the patent and the general assessment is that the patent covers only systems that compress AND decompress - systems that decompress only are not covered. It's worth pointing out that LZW decoders are shipped with a number of products including Netscape Navigator and the Sun JDK itself, without any royalty fees being paid.

This library includes an LZW decompression algorithm for reading these older PDF documents. Unless you're parsing documents over 5 years old, you probably won't need it, and it can be deleted from the JAR without affecting anything else - the file is `org/faceless/pdf/LZW.class`.

Do any licensing fees have to be paid to Adobe for this product

No. Adobe have given the go ahead to anyone to create, parse and edit documents in the PDF format, provided they adhere to the PDF specification (which we do).

Acknowledgements

"Acrobat", "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.

This document was created with the Big Faceless Report Generator, version 1.0.15