Embedding in a Java application

by Jeremias Märki

Table of contents

1 Introduction

This page describes how to integrate **Barcode4J** in a Java application.

2 Basic steps

There are several basic steps involved for generating barcodes:

1. Provide an Avalon Configuration object to configure the barcode engine
2. Create a BarcodeGenerator
3. Create a CanvasProvider (depending on the output format)
4. Finally generate the barcode

3 Providing an Avalon Configuration object

Barcode4J uses Avalon Framework for configure the barcode symbology used. This is done in the form of a Configuration object. Please refer to the Avalon documentation for further examples. We simply provide an example for the most frequently used approach to get such a Configuration object. We load the Configuration from an XML file:

Here's an example:

```
import org.apache.avalon.framework.configuration.Configuration;
import org.apache.avalon.framework.configuration.DefaultConfigurationBuilder;

[..]

DefaultConfigurationBuilder builder = new DefaultConfigurationBuilder();
Configuration cfg = builder.buildFromFile(new File("C:\Temp\barcode-cfg.xml"));
```

An XML loaded in the above way would have to look like the following example. You will notice that this is basically the Barcode XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<barcode>
  <ean13>
    <module-width>0.4mm</module-width>
  </ean13>
</barcode>
```

For other ways to build the Configuration object please check the Avalon documentation or browse the source code of **Barcode4J** for more examples.

4 Creating a BarcodeGenerator

The BarcodeGenerator is used to generate the logical parts of a barcode (narrow bars, wide bars etc.).

To make it easy to create the BarcodeGenerator use the BarcodeUtil singleton class. This class will choose the correct barcode implementation depending on the configuration you've built above.

```
BarcodeGenerator gen = BarcodeUtil.getInstance().createBarcodeGenerator( cfg );
```

5 Creating the CanvasProvider and generating the barcode

We combine the last two steps in one section because the last step depends heavily on the other.

The CanvasProvider takes the painting instructions and transforms them to the target format. Depending on the desired output format you need to instantiate a different class.

All CanvasProvider classes support an orientation value as the last parameter. Normally, you will just specify "0", but you can rotate the barcode in steps of 90 degrees. Valid values are 0, 90, -90, 180, -180, 270 and -270.

5.1 SVG

Here you have two classes to choose from, depending on the preferred DOM implementation (W3C DOM or JDOM).

```
SVGCanvasProvider provider = new SVGCanvasProvider(false, 0);
gen.generateBarcode(provider, msg);
org.w3c.dom.DocumentFragment frag = provider.getDOMFragment();
```

..or..

```
JDOMSVGCanvasProvider provider = new JDOMSVGCanvasProvider(false, 0);
gen.generateBarcode(provider, msg);
org.jdom.Document doc = provider.getDocument();
```

The boolean parameter on the two constructors instruct the implementation not to use namespaces for the output (in this example). There are also constructors where you can freely choose the prefix to use. Please refer to the javadocs for more information.

If you need the barcode as an XML file, it is quite simple to use basic JAXP code to serialize the DOM to a file.

5.2 EPS

The EPS output simply takes an OutputStream to write to. One speciality is the need to call the finish() method after the generation because the EPS needs to be finished with a trailer part.

```
OutputStream out = new java.io.FileOutputStream(new File("output.eps"));
EPSCanvasProvider provider = new EPSCanvasProvider(out, 0);
gen.generateBarcode(provider, msg);
provider.finish();
```

5.3 Bitmaps

The bitmap output, like the EPS output, takes an OutputStream to write to. But in addition to that it needs the MIME type (or short format type) of the desired output format. Furthermore, it needs the resolution of the bitmap in dpi (dots per inch), the image type (which is one of BufferedImage's constants defining the color depth) and a boolean to enable or disable anti-aliasing. The

BitmapCanvasProvider also needs a call to the finish() method to encode the internally built image to the desired image format.

```
OutputStream out = new java.io.FileOutputStream(new File("output.png"));
BitmapCanvasProvider provider = new BitmapCanvasProvider(
    out, "image/x-png", 300, BufferedImage.TYPE_BYTE_GRAY, true, 0);
gen.generateBarcode(provider, msg);
provider.finish();
```

The example here produces a grayscale PNG with 300 dpi and anti-aliasing.

> **Note:**
> Although JPEG output is supported, it is NOT recommended as it is a lossy compression format and contrast will suffer and thus degrading the readability of the generated barcode.

Instead of writing the bitmap to a file you can fetch the BufferedImage instead. That would look like this:

```
BitmapCanvasProvider provider = new BitmapCanvasProvider(
    300, BufferedImage.TYPE_BYTE_GRAY, true, 0);
gen.generateBarcode(provider, msg);
provider.finish();
BufferedImage barcodeImage = provider.getBufferedImage();
```

5.4 Java2D (AWT)

Finally, we'll want to show you how to paint a barcode on a surface of your choosing. In principle, you just need to provide the Java2DCanvasProvider with a Graphics2D interface that it can paint on. However, since the Java2D output internally works with millimeters (mm) you need to make sure the Graphics2D is prepared for that. This means you may need to apply a transformation before painting the barcode. See the BitmapBuilder class for an example. BitmapBuilder is responsible to set up the Graphics2D object for painting barcodes on bitmaps for the bitmap output support.

It's a good idea, if possible, to enable fractional metrics for higher quality:
`g2d.setRenderingHint( RenderingHints.KEY_FRACTIONALMETRICS, RenderingHints.VALUE_FRACTIONALMETRICS_ON )`

```
Graphics2D g2d = <something>
Java2DCanvasProvider provider = new Java2DCanvasProvider(g2d, 0);
gen.generateBarcode(provider, msg);
```