Poul Klausen

JAVA 10

Java2D, Drawing of the window

Software Development

bookboon
The eBook company

POUL KLAUSEN

JAVA 10: JAVA2D, DRAWING OF THE WINDOW

SOFTWARE DEVELOPMENT

Java 10: Java2D, Drawing of the window: Software Development
1st edition

ISBN 978-87-403-1947-7
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the tenth in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. In the previous book, I have relatively detailed treated Swing, and the subject of this book is *Java2D*, which is the other half of what Java is making available for developing applications with a graphical user interface. One can also think of *Java2D* as the graphical tools that Swing uses to draw the components in a window. The book is relatively detailed and addresses issues that are not used so often in everyday programming, but the examples are, of course, and also the issues are important, to understand how the GUI works. It is similar to a few other books in this series a book where the focus is on language Java over the process, and only the final example focuses on system development with the development of a Java class library. The book assumes knowledge of Java corresponding to the books Java 3 and Java 4 and to some extent knowledge of Swing corresponding to the book Java 2.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

- NetBeans as IDE for application development
- MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
- GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

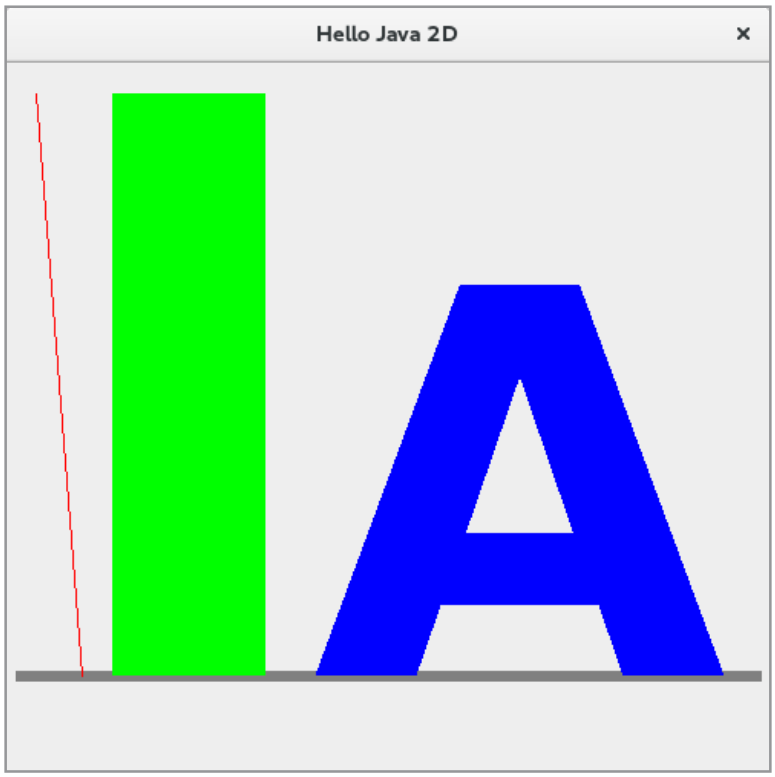
When you write a program with a graphical user interface, the programming is based to a large extent on the use of finished classes for windows, buttons, fonts, etc. From the start, these classes was gathered in AWT and its subpackages, and the APIs could be seen as two layers of classes that could be called a *user interface toolkit* and a *drawing toolkit*, the first layer comprised of the classes for components and properties of the components, while the second layer was consisted of classes for basic geometric tools that could draw lines and squares and manipulate the individual pixels. Later (starting with Java 2) AWT was split into two APIs in the form of *Swing* and *Java2D*. This book deals with Java2D and thus how you in Java can program 2-dimensional graphics and manipulate images.

Like other Java APIs includes *Java2D* a very large number of classes, but overall these are classes for:

1. *Shapes*, where you can create arbitrary geometric shapes as combinations of straight lines, curves, rectangles, ellipses and arcs.
2. *Stroking* that defines how lines should be drawn in terms of width, dashed or as a solid stroke. Furthermore, you can specify how the corners will be drawn.
3. *Filling* that specifies how shapes are filled. You can specify a color, a pattern or a gradient fill, and in fact you can also use an image.
4. *Transformations*, where everything that is drawn (shapes, images, text) can be stretched, scaled and rotated.
5. *Alpha compositing*, is the process of combining a drawing with a background to create figures with partial or full transparency.
6. *Clipping* is an option for limiting the part of a shape to be drawn so as not to draw on an unwanted area of the window.
7. *Anti-aliasing* is a technique to avoid uneven lines, where you can see the individual pixels.
8. *Text* where you can manipulate the text in the same way as other geometric shapes.
9. *Color* comprising classes and methods to represent colors, so they are hardware independent.
10. *Images* that provide the ability to manipulate images in the same way as other geometric shapes. It also includes classes to read images from and save images in files.
11. *Image processing*, which includes classes to manipulate the individual pixels in images.
12. *Animation*, which are classes that make services available, so you can animate geometric shapes, text and images.

The rest of this book, through a number of examples treats the above 12 items, but first I want to show a program. The program is called *HelloJava2D* and if you runs the program, it opens a window, as shown below. The window draws two straight lines, a rectangle and a large A. The window's code is as follows:

```
package hellojava2d;

import java.awt.*;
import java.awt.geom.Line2D;
import java.awt.geom.Rectangle2D;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Hello Java 2D");
  setSize(500, 500);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}
```



and the only thing that happens in the code is the addition of a single component of the type *Drawing()*. As a *JFrame* as default has a *BorderLayout*, this component automatically fills the whole window. The component is defined as follows:

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setStroke(new BasicStroke(7));
  g2d.setColor(Color.gray);
  g2d.draw(new Line2D.Double(10, 400, 490, 400));
  Line2D line = new Line2D.Double(20, 20, 50, 400);
  Rectangle2D rect = new Rectangle2D.Double(70, 20, 100, 380);
  g2d.setStroke(new BasicStroke(1));
  g2d.setColor(Color.red);
  g2d.draw(line);
  g2d.setColor(Color.green);
  g2d.fill(rect);
  g2d.setColor(Color.blue);
  g2d.setFont(new Font("Liberation Sherif", Font.BOLD, 350));
  g2d.drawString("A", 200, 400);
 }
}
```

It is thus a custom Swing component. A component has a method, called *paintComponent()*, and it is the method that draws the component. The method's parameter has the type *Graphics*, and it is an object that provides drawing tools available and represents the program's graphics. The type has been around all the time, Java has existed, but with the introduction of Java2D is defined a derived class called *Graphics2D*, which expands with new graphics methods. Therefore starts the method *paintComponent()* to type cast the parameter to a *Graphics2D* object. Then the method can draw on the graphic object, and basically the following happens.

The following statements defines to be drawn with a gray pen with a width of 7 pixels, and the third statement draws a straight line between two points:

```
g2d.setStroke(new BasicStroke(7));
g2d.setColor(Color.gray);
g2d.draw(new Line2D.Double(10, 400, 490, 400));
```

The next two statements defines two geometric shapes, which are, respectively, a line and a rectangle:

```
Line2D line = new Line2D.Double(20, 20, 50, 400);
Rectangle2D rect = new Rectangle2D.Double(70, 20, 100, 380);
```

The next three statements defines to draw with a red pen of 1 pixel, and the third statement draws the above straight line:

```
g2d.setStroke(new BasicStroke(1));
g2d.setColor(Color.red);
g2d.draw(line);
```

The next two statements again tells that there should be painted with a green color, and the above rectangle is filled with the color green:

```
g2d.setColor(Color.green);
g2d.fill(rect);
```

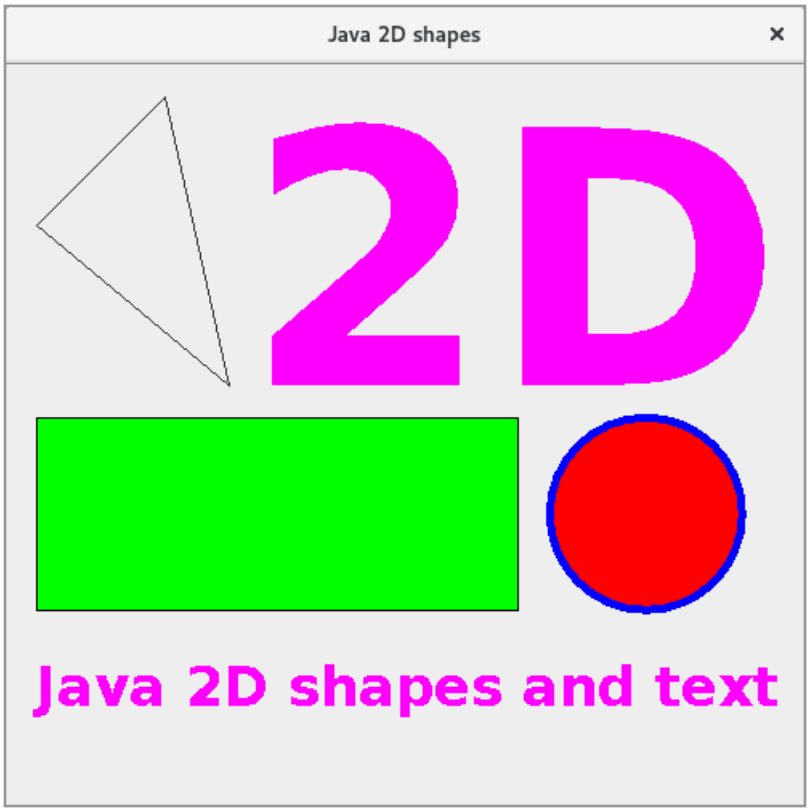
Final the last three statements says that the color should be blue, the text should be drawn with a bold font at 350 points, and finally drawn a large A:

```
g2d.setColor(Color.blue);
g2d.setFont(new Font("Liberation Sherif", Font.BOLD, 350));
g2d.drawString("A", 200, 400);
```

Of course one can not know that the statements should be written that way, but when you see the individual statements, they are easy enough to understand.

EXERCISE 1

Write a program that you can call *Java2DShapes*. The program should open a window, as shown below:



The program can be written in the same manner as above. The triangle is drawn as three straight lines. Try it yourself (using the Java documentation) to find out how to draw a circle. The type is *Ellipse2D*.

1.1 THE CLASS GRAPHICS2D

Before addressing the examples I will say a few words about the class *Graphics2D* and thus slightly concerning the technique.

The process of drawing in a window is based on a number of objects as shapes, images, text, etc. that able you to determine which colors the individual pixels on a screen or a printer should have, and this process is called *rendering*, and thus rendering is the process to show the results of graphic primitives on an output device. The class *Graphics2D* can be perceived as Java2D's rendering engine. In addition a Graphics2D object represents a graphics surface in the form of a collection of pixels each having a color. How the object converts graphics primitives to the pixels and which the colors they get are determined by the state of the object, which includes seven concepts:

1. *paint,* that determines which colors to use for drawing and filling a shape and also including text
2. *stroke*, indicating the thickness of the line that shapes are drawn with by the method *draw()*
3. *font* to indicate how text should be rendered
4. *transformation* that specifies how primitive graphical objects should be transformed (moved, rotated, stretched) before being rendered, including how the logical coordinates must be converted to physical coordinates (by default 72 logical coordinates are transformed to an inch on the physical device)
5. *compositing* used to determine how colors are to be combined with the existing color on the drawing surface
6. *clipping*, which definnes restrictions on the area to be rendered (default is null, which means the entire drawing surface)
7. *rendering hints*, there are techniques that tells how objects should be rendered

There are four general methods used for rendering geometric primitive:

1. *fill()*, which fills the interior of a shape
2. *draw()*, that draws the perimeter of a shape
3. *drawString()*, that draws a text
4. *drawImage()*, which is used to show an image

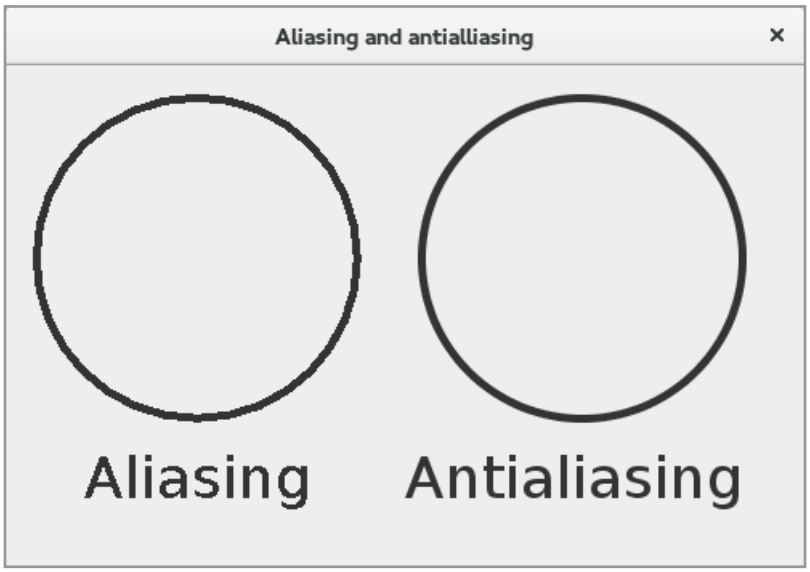
The rendering process consists of several steps which are in principle independent of the above four methods and performed in the following order:

1. *transformation*
2. *rasterizing*
3. *clipping*
4. *paint*
5. *compositing*

When a given figure needs to be rendered, for example on a screen, it is an approximation, and the shape may not necessarily be presented exactly. The screen represents a shape using dots (pixels), and you can for example not produce an exact circle with points (one can see that the periphery consists of points and not a bow). For determining the points that must be part of a specific shape, the computer applies one of two ways. Default is *aliasing*, where the points whose center is within the figure are colored. That is, the individual pixels are either part of the figure, or else they are not. The result is a shape in which edges occur pixilated. The advantage of aliasing is that it is an effective and quick way to draw a shape and in most cases the result is fully satisfactory. The second method is called *antialiasing*, and here the computer determines for each pixel the intersection of the figure and the current pixel. The pixel is colored with a saturation determined by the degree to which it is a part of the figure. The result is a figure where lines and arches not to the same degree occurs pixilatted. In the figure below the circle to the left is rendered with aliasing, while the circle to the right is rendered with antialiasing.

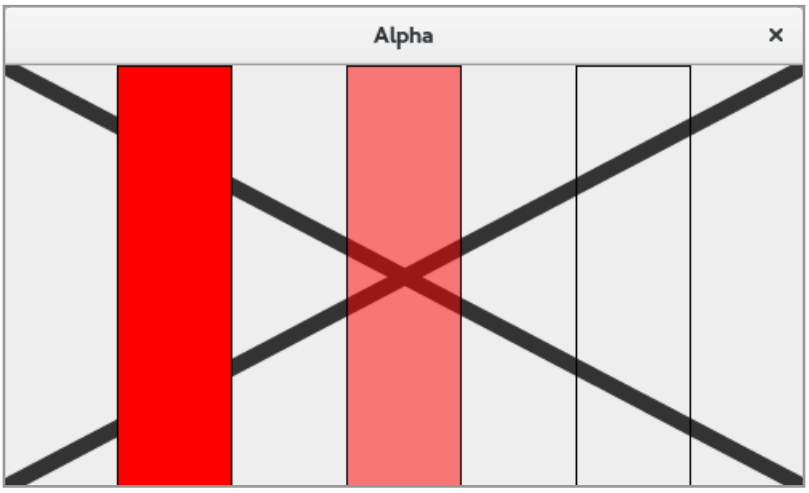
Also note the difference between text. The disadvantage of antialiasing is, of course, that it is a comprehensive rendering process, as to be carried out many calculations.

The code for the below window is the following where you primarily need to note how to specify that the *Graphics2D* object must use antialiasing:



```
package aliasing;

import java.awt.*;
import java.awt.geom.Ellipse2D;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Aliasing og antialliasing");
  setSize(500, 350);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}

class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Ellipse2D cirkel1 = new Ellipse2D.Double(20, 20, 200, 200);
  Ellipse2D cirkel2 = new Ellipse2D.Double(260, 20, 200, 200);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setStroke(new BasicStroke(5));
  g2d.draw(cirkel1);
  g2d.setFont(new Font("Liberation Sherif", Font.PLAIN, 36));
  g2d.drawString("Aliasing", 50, 270);
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  g2d.draw(cirkel2);
  g2d.drawString("Antialiasing", 250, 270);
 }
}
```

Above, I mentioned that the rendering process includes rasterizing and compositing, and it indicates the degree to which a figure's pixels have to cover the drawing surface's pixels. This value is called the *alpha value* and is a number between 0.0 (not visible or transparent) and 1.0 (fully covered or opaque). As an example, shows the following window two straight lines and three rectangles where the two lines are drawn first:



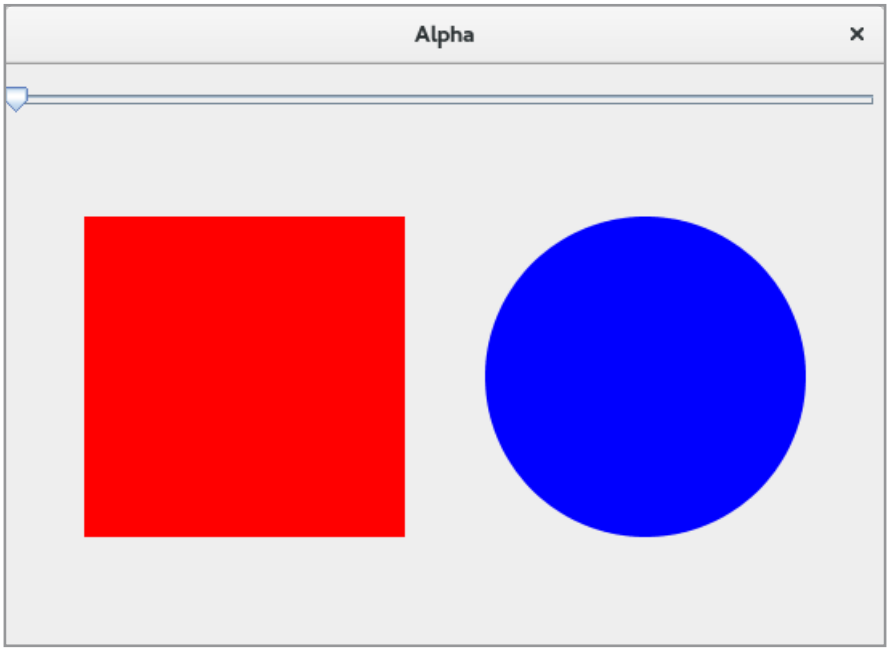
The first rectangle is drawn with an alpha value of 1 and would therefore total cover the underlying pixels. The result is that the two lines are covered of the rectangle. The middle rectangle has an alpha value of ½, and thus it does not cover the lines completely. The result is that the red color is combined partly with the black lines and and the gray background. The last rectangle have an alpha value of 0 and is thus completely transparent and therefore not visible. The code is as follows:

```
package alphaprogram;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Alpha");
  setSize(500, 300);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}

class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Dimension size = getSize();
  Graphics2D g2d = (Graphics2D)g;
  g2d.setStroke(new BasicStroke(10));
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
  Line2D line1 = new Line2D.Double(0, 0, size.width, size.height);
  Line2D line2 = new Line2D.Double(0, size.height, size.width, 0);
  double width = size.width / 7.0;
  Rectangle2D rect1 = new Rectangle2D.Double(width, 0, width, size.height);
  Rectangle2D rect2 = new Rectangle2D.Double(3 * width, 0, width, size.height);
  Rectangle2D rect3 = new Rectangle2D.Double(5 * width, 0, width, size.height);
  g2d.draw(line1);
  g2d.draw(line2);
  g2d.setColor(Color.red);
  g2d.fill(rect1);
  g2d.setColor(new Color(1.0F, 0F, 0F, 0.5F));
  g2d.fill(rect2);
  g2d.setColor(new Color(1.0F, 0F, 0F, 0F));
  g2d.fill(rect3);
  g2d.setStroke(new BasicStroke(1));
  g2d.setColor(Color.black);
  g2d.draw(rect1);
  g2d.draw(rect2);
  g2d.draw(rect3);
 }
}
```

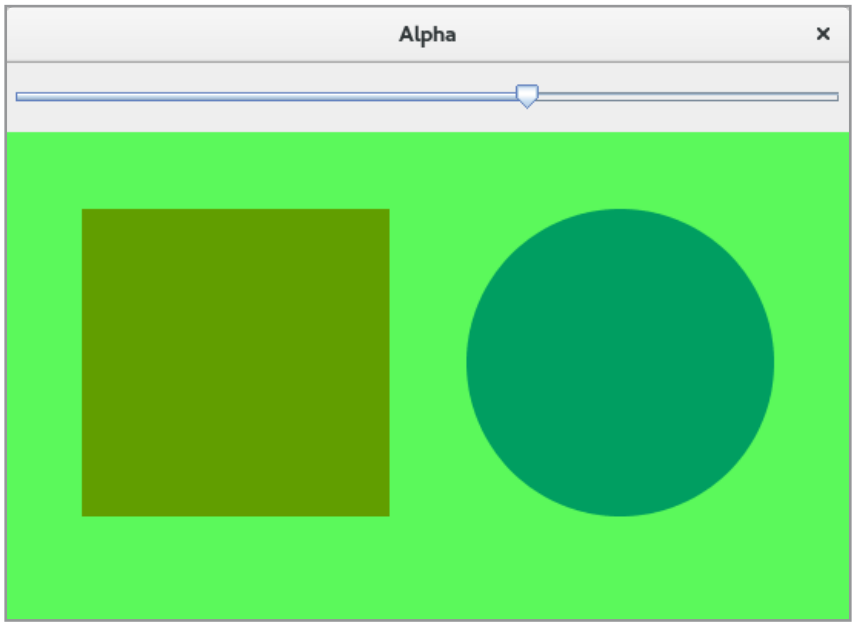
If you test the program, noting that both the lines and the rectangles follows the window size.

When working with Java 2D and in general with computer graphics you have a coordinate system. The window's upper left corner is (0,0), and the x axis is oriented to the right, while the y-axis is oriented downwards. It is a logical coordinate system called *user space*, and when the objects to be drawn on a physical display or printer, you have a different coordinate system, called *device space*. The rendering therefore requires a conversion of coordinates from user space to device coordinates, and for the screen this conversion usually is performed directly where 1 unit in user space corresponds to one pixel, but generally aim for conversion where 72 units (in user space) are converted into 1 inch.

EXERCISE 2



You must write a program (my solution is called *AlphaValue*), which opens a window, as shown above, where is painted a square and a circle. At the top is a slider, and if you move the slider, the figure should be covered by a green rectangle that eventually covers the figure completely:



You must associate an event handler to the slider, that redraw the component when the slider is moved. If *Drawing* refers to the component with the figures, the event handler can be defined as an object of the following inner class:

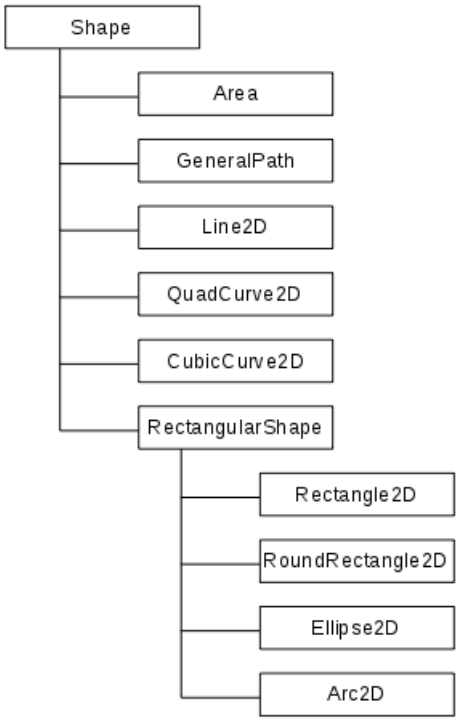
```
private class ChangeHandler implements ChangeListener
{
 public void stateChanged(ChangeEvent e)
 {
  drawing.repaint();
 }
}
```

2 SHAPES

In this chapter I will show how to define and draw geometric shapes, what I already did with the examples above, for example a rectangle. Its type is *Rectangle2D*, but it is an abstract class, and there are two concrete derived classes, which are called respectively *Rectangle2D.Float* and *Rectangle2D.Double*. The difference is about the figure's attributes and where they are defined by the type *float* or type *double*. This pattern is used for all primitive *Shape* objects.

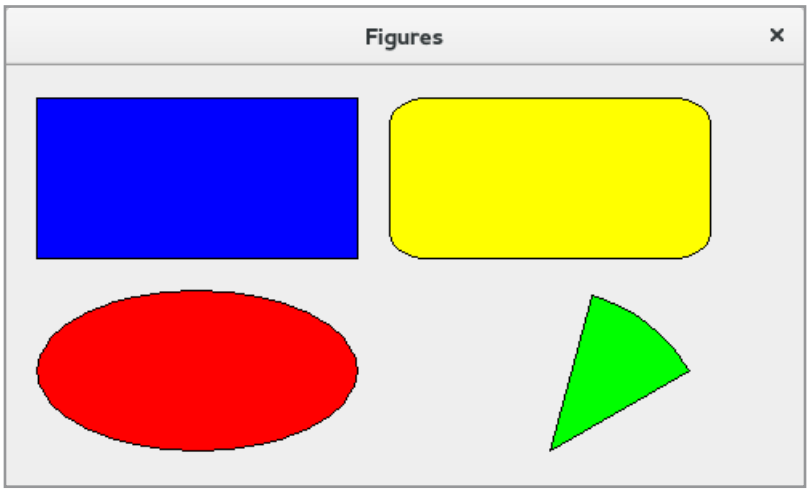
There is a helper class, called *Point2D* (and *Point2D.Float* and *Point2D.Double*) that represents a point. A *Point2D* object has no size and can not be rendered, and an object is used only to define attributes for other objects. The basic geometric classes are shown in the following figure, and they are all defined by a common interface called *Shape*. This interface defines a few important methods where I will mention

- *getBounds2D()*, which returns a circumscribing rectangle
- *contains()*, that tests where a point or rectangle is included in the shape
- *intersects()*, that tests where a rectangel overlaps the current shape



2.1 FILLED SHAPES

The last four are simple to use, and the difference is mainly the parameters you have to enter to create an object. Generally a figure is defined by a rectangle, that is the circumscribing rectangle which indicate the upper left corner and the width and height. The program *Figures* opens the following window:



As regards the types *Rectangle2D* and *Ellipse2D* there is not much more to tell, but for a *RoundRectangle2D* you must beyond the circumscribing rektangle specify how the corners are rounded – in fact, in terms of a width and a height of a rectangle. Then there is the green shape, which is an *Arc2D*. First you have to define the circumscribing rectangle, and here one must be aware that the rectangle is the rectangle that circumscribes the ellipse, that the shape is a part of. In addition to the rectangle you must specify the angle (in degrees), where the range should start and how large an angle the range should span. Finally, you must specify how the slice should appear where there are the following options:

- *Arc2D.PIE*, that shows the shape in this example
- *Arc2D.OPEN*, which only shows the arc
- *Arc2D.CHORD*, that shows the arc and the chord of the arc's end points

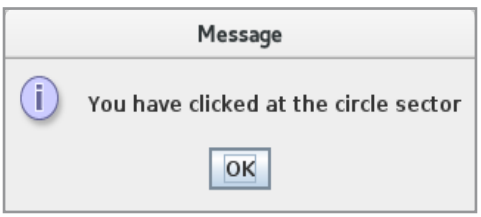
The code for the above window is:

```
package figures;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
```

```
public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Figures");
  setSize(500, 300);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}

class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  draw(g2d, new Rectangle2D.Double(20, 20, 200, 100), Color.blue);
  draw(g2d, new RoundRectangle2D.Double(240, 20, 200, 100, 50, 30),
    Color.yellow);
  draw(g2d, new Ellipse2D.Double(20, 140, 200, 100), Color.red);
  draw(g2d, new Arc2D.Double(240, 140, 200, 200, 30, 45, Arc2D.PIE),
    Color.green);
 }

 private void draw(Graphics2D g, Shape shape, Color color)
 {
  g.setColor(color);
  g.fill(shape);
  g.setColor(Color.black);
  g.draw(shape);
 }
}
```

EXERCISE 3

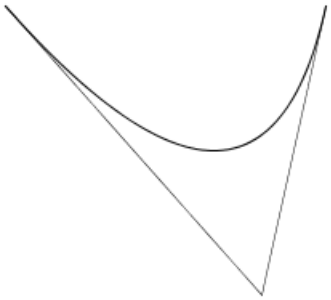
Make a copy of the above project, which opens the window with the four figures. You must change the program so that if you click on a shape, you should get a message box as shown below telling you which figure is clicked. Note that the only thing you need is to add an event handler for mouse clicks, which determines whether there is clicked on a figure and if so, what.



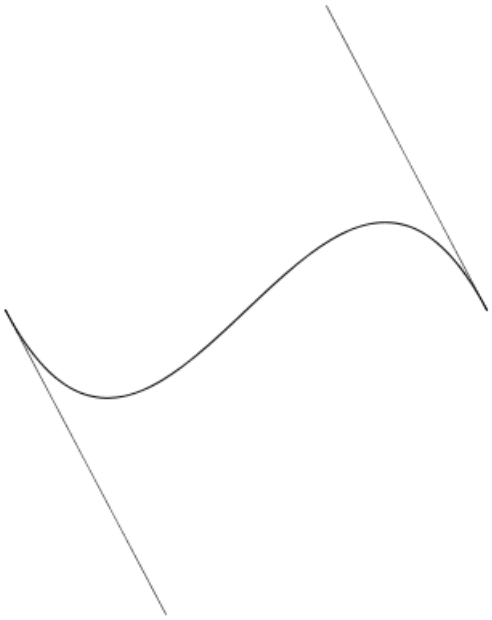
2.2 LINES

I will then look at the three types *Line2D*, *QuadCurve2D* and *CubicCurve2D*, each representing a line segment. *Line2D* represents a straight line between two points, and there is not much to explain. To create a *Line2D* object, you must only indicate the endpoints of the segment.

A *QuadCurve2D* represents a path between two points, which has an associated control point so that the lines from the control point of the curve's endpoints are tangents to the curve:



If you need to create a *QuadCurve2D*, one must therefore indicate three points. A *CubicCurve2D* is in principle the same, but here there are two control points, one for each of the two end points:



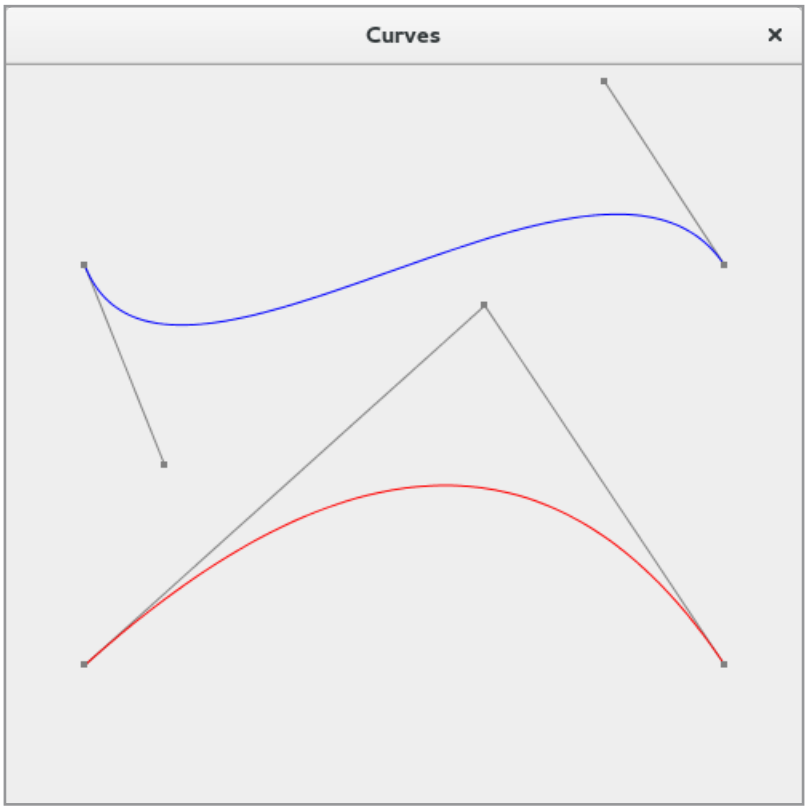
One must therefore specify four points to create a *CubicCurve2D*. It can be a little difficult to find out the result of these curves. The following program opens a window, as shown below. The blue curve is a *CubicCurve2D*, while the red is a *QuadCurve2D*. In addition to the endpoints, the drawing shows the control points, as well as the tangents and the purpose of the program is, that you can point on one of the points with the mouse and dragging it, and in that way to get an idea of what happens to the shapes, if you change the end points or control points.

The code for the window is shown below, and you must mainly observe how to create respectively a *QuadCurve2D* and a *CubicCurve2D*:

```
package curves;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
```

```
public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Curves");
  setSize(500, 500);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}

class Drawing extends JComponent
{
 private Point2D[] points = {
  new Point2D.Double(50, 125), new Point2D.Double(100, 250),
  new Point2D.Double(375, 10), new Point2D.Double(450, 125),
  new Point2D.Double(50, 375), new Point2D.Double(300, 150),
  new Point2D.Double(450, 375) };

 private Point2D current = null;

 public Drawing()
 {
  addMouseListener(new ClickHandler());
  addMouseMotionListener(new MoveHandler());
 }

 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  g2d.setColor(Color.gray);
  g2d.draw(new Line2D.Double(points[0], points[1]));
  g2d.draw(new Line2D.Double(points[2], points[3]));
  g2d.setColor(Color.blue);
  g2d.draw(new CubicCurve2D.Double(points[0].getX(), points[0].getY(),
   points[1].getX(), points[1].getY(), points[2].getX(),
   points[2].getY(), points[3].getX(), points[3].getY()));
  g2d.setColor(Color.gray);
  g2d.draw(new Line2D.Double(points[4], points[5]));
  g2d.draw(new Line2D.Double(points[5], points[6]));
  g2d.setColor(Color.red);
  g2d.draw(new QuadCurve2D.Double(points[4].getX(), points[4].getY(),
   points[5].getX(), points[5].getY(), points[6].getX(), points[6].getY()));
  for (int i = 0; i < points.length; ++i)
  {
```

```
   g2d.setColor(points[i].equals(current) ? Color.magenta : Color.gray);
   g2d.fill(getPoint(points[i]));
  }
 }

 private Shape getPoint(Point2D p)
 {
  int side = 4;
  return new Rectangle2D.Double(
   p.getX() - side / 2, p.getY() - side / 2, side, side);
 }

 private class ClickHandler extends MouseAdapter
 {
  public void mousePressed(MouseEvent e)
  {
   current = null;
   for (int i = 0; i < points.length; ++i)
   {
    Shape shape = getPoint(points[i]);
    if (shape.contains(e.getPoint()))
    {
     current = points[i];
     break;
    }
   }
   repaint();
  }
 }

 private class MoveHandler extends MouseMotionAdapter
 {
  public void mouseDragged(MouseEvent e)
  {
   if (current != null)
   {
    current.setLocation(e.getPoint());
    repaint();
   }
  }
 }
}
```

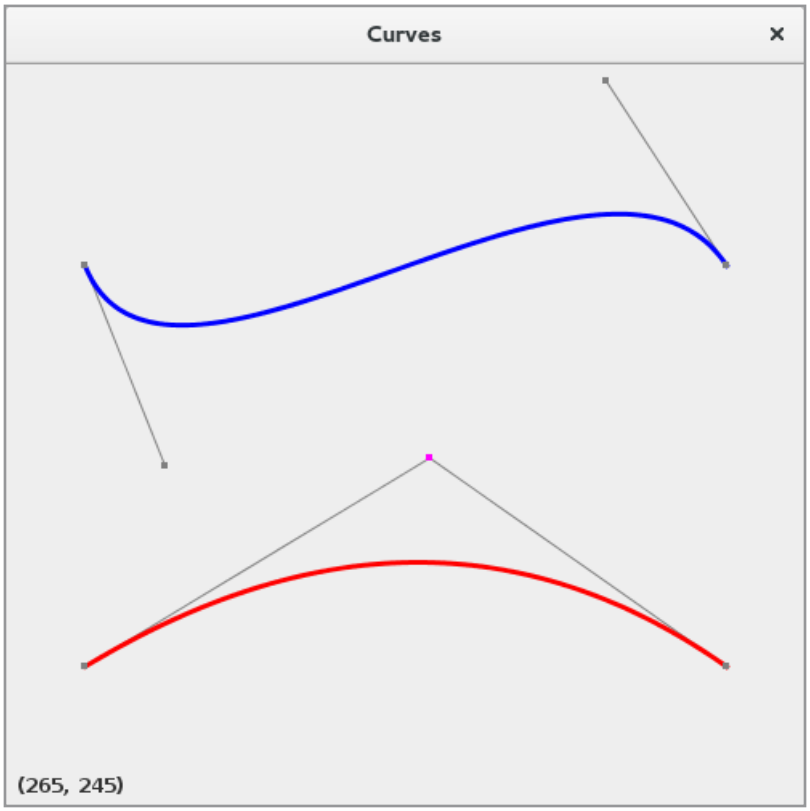

Curves

EXERCISE 4

Create a copy of the project *Curves* and modify the program so that the bottom of the window shows the mouse coordinates when you drag the points, and the two curves must be drawn with a thicker line. This means that you must use a *Stroke* object when the curves are drawn, for example

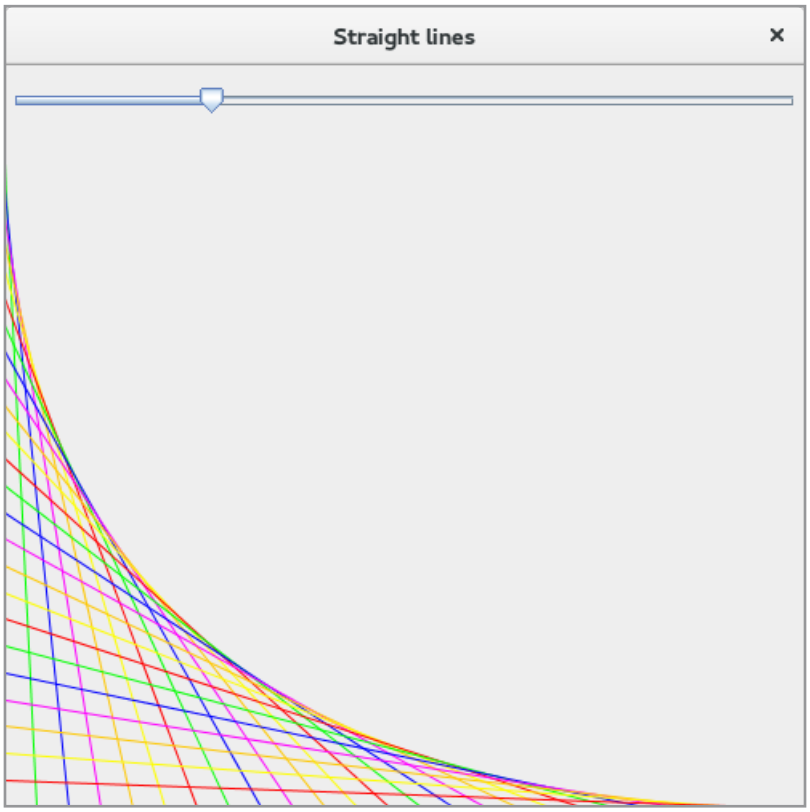
```
private Stroke stroke3 = new BasicStroke(3);
```

To display the mouse coordinates you can add a *JLabel* component to the bottom of the window and update it every time you drag one of the points.



EXERCISE 5

Write a program that you can call *StraightLines*, which opens the following window shown below. The slider should represent the values from 0 to 100 and start at 25. The window must draw the number of straight lines that the slider indicates, and alternates between six colors. A line start at the upper left corner to a point at the bottom of the window, where the bottom is divided in a number of points determinded be the slider. Similarly, a line starts in the lower right corner to a point on the left edge which is divided into a number of points corresponding to the value of the slider.



2.3 GENERALPATH

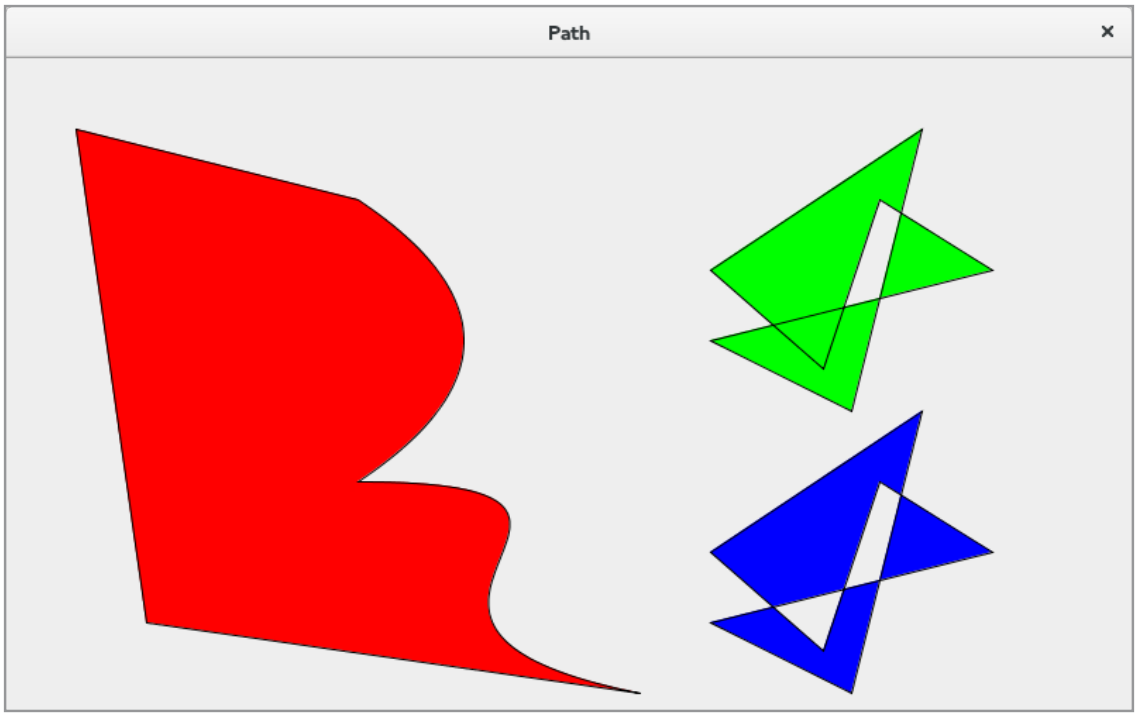
Line2D, *QuadCurve2D*, *CubicCurve2D*, *Rectangle2D*, *RoundRectangle2D*, *Ellipse2D* and *Arc2D* are the basic geometric primitives in Java2D, but from the above class diagram it appears that there are two other classes, and they may be used to define more complex shapes.

The class *GeneralPath* defines a shape by means of a number of points, where between each of these points may be a line either of the type *Line2D*, *QuadCurve2D* or *CubicCurve2D*. A *GeneralPath* defines an arbitrary shape, which does not necessarily have to be closed or coherent. You can think of the shape on that way, that you draw it with a pen, and you can do two things:

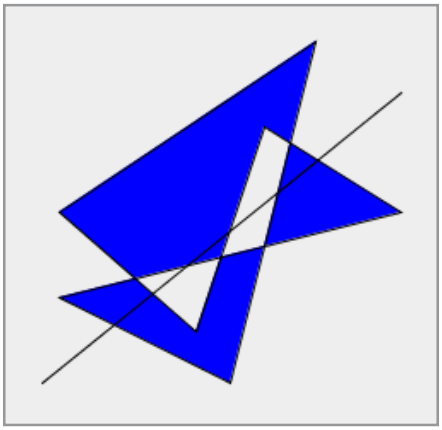
1. You can lift the pen and move it to a point.
2. You can draw a line from the point where the pen is, to another point.

For that you uses the methods: *moveTo(), lineTo(), quadTo(), cubicTo()* and *closePath().*

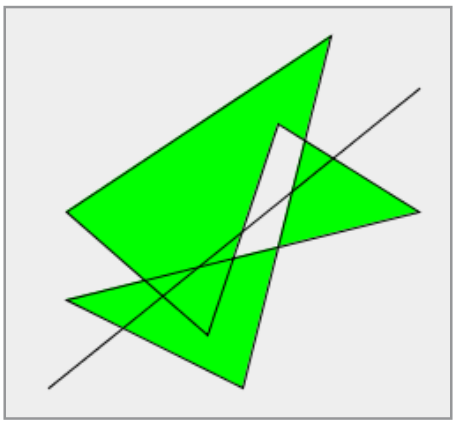
The window below shows three shapes of the type *GeneralPath* where the red is formed by three *line* segments, a *quad* segment and a *cubic* segment. The two other shapes are solely formed by the *line* segments:


Path

Generally, it is simple enough to create a *GeneralPath*, and there is only one thing that is complex, namely to determine the interior of shapes. If the shape is bounded by a simple closed curve (as the red), there is not much to be in doubt, but if the shape is formed by lines that intersects, it is more difficult, and it can happen in two ways, which you specify with a parameter to the constructor. If you considers the blue figure, it is defined as *GeneralPath.WIND_EVEN_ODD*. The straight lines divide the shape into a number of closed regions (five in this case). If you have to decide which of these areas belong to the figure, you can draw a straight line that cuts through the figures areas:



If you starts outside the shape and follows the line and counts a counter up by 1 each time you pass a segment, so the rule is that an area belongs to the figure, if the counter is odd and does not belong, if the counter is even. The green figure is defined as *GeneralPath.WIND_NON_ZERO* (which incidentally is the default). If you here must determine if a part of the figure belongs, one can similarly draw a line that intersects the figure's areas. When the line crosses one of the figure's segments, the rule is that the counter is incremented by 1 if the segment is drawn from left to right, and decremented by 1 if the segment is drawn from right to left. The areas that is part of the figure, are the areas in which the counter is not 0.

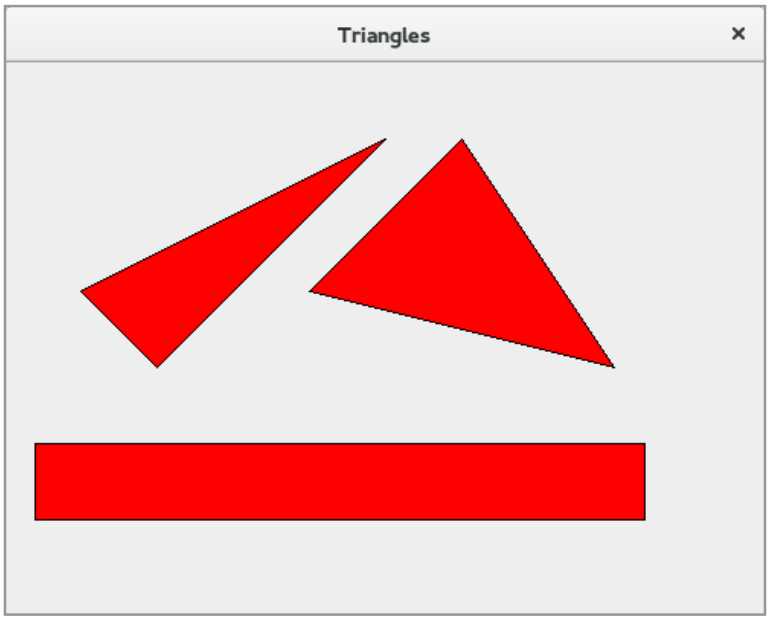


The program's code is:

```
package pathprogram;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Path");
  setSize(800, 500);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}

class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  GeneralPath path1 = new GeneralPath();
  path1.moveTo(50, 50);
  path1.lineTo(250, 100);
  path1.quadTo(400, 200, 250, 300);
  path1.curveTo(500, 300, 200, 400, 450, 450);
  path1.lineTo(100, 400);
  path1.closePath();
  g2d.setColor(Color.red);
  g2d.fill(path1);
  g2d.setColor(Color.black);
  g2d.draw(path1);

  GeneralPath path2 = new GeneralPath(GeneralPath.WIND_NON_ZERO);
  path2.moveTo(500, 150);
  path2.lineTo(650, 50);
  path2.lineTo(600, 250);
  path2.lineTo(500, 200);
  path2.lineTo(700, 150);
```

```
        path2.lineTo(620, 100);
        path2.lineTo(580, 220);
        path2.closePath();
        g2d.setColor(Color.green);
        g2d.fill(path2);
        g2d.setColor(Color.black);
        g2d.draw(path2);

        GeneralPath path3 = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
        path3.moveTo(500, 350);
        path3.lineTo(650, 250);
        path3.lineTo(600, 450);
        path3.lineTo(500, 400);
        path3.lineTo(700, 350);
        path3.lineTo(620, 300);
        path3.lineTo(580, 420);
        path3.closePath();
        g2d.setColor(Color.blue);
        g2d.fill(path3);
        g2d.setColor(Color.black);
        g2d.draw(path3);
    }
}
```

Besides creating a *GeneralPath* using segments, you can also create a *GeneralPath* as composed of other shapes. The window below shows a *GeneralPath* that draw a triangle, but also consists of another *GeneralPath* (which also represents a triangle) and a *Rectangle2D*. If you click on one of the three shapes, you get a message box, and the goal is to show that the three figures are all part of the same *Shape* object.



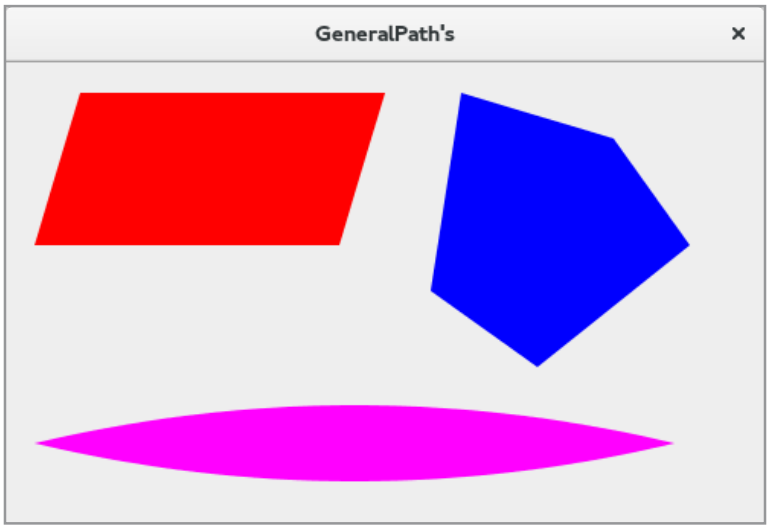
The code is as follows:

```
package triangles;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;
import java.awt.event.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Triangles");
  setSize(500, 400);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}
```

```
class Drawing extends JComponent
{
 private Shape shape;

 public Drawing()
 {
  addMouseListener(new ClickHandler());
 }

 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  GeneralPath path1 = new GeneralPath();
  path1.moveTo(50, 150);
  path1.lineTo(250, 50);
  path1.lineTo(100, 200);
  path1.closePath();

  GeneralPath path = new GeneralPath();
  path.moveTo(200, 150);
  path.lineTo(300, 50);
  path.lineTo(400, 200);
  path.closePath();
  path.append(path1, false);
  path.append(new Rectangle2D.Double(20, 250, 400, 50), false);
  g2d.setColor(Color.red);
  g2d.fill(path);
  g2d.setColor(Color.black);
  g2d.draw(path);

  shape = path;
 }

 private class ClickHandler extends MouseAdapter
 {
  public void mouseClicked(MouseEvent e)
  {
   Point2D point = new Point2D.Double(e.getX(), e.getY());
   if (shape.contains(point)) JOptionPane.showMessageDialog(Drawing.this,
    "You have clicked on the figure");
  }
 }
}
```

EXERSICE 6

Write a program that you can call *Polygons* where it must open a window, as shown below (where the lower shape is not a polygon). The three shapes, should all be defined as a *GeneralPath* in which the bottom is defined by two *quad* segments. Finally, if you click on one of the three figures you should get a message box telling you which of the shapes you are clicking on:



2.4 AREA

The last class for geometric shapes is called *Area*. It is a type that combines one or more shapes, but where you can specify how they should overlap each other.

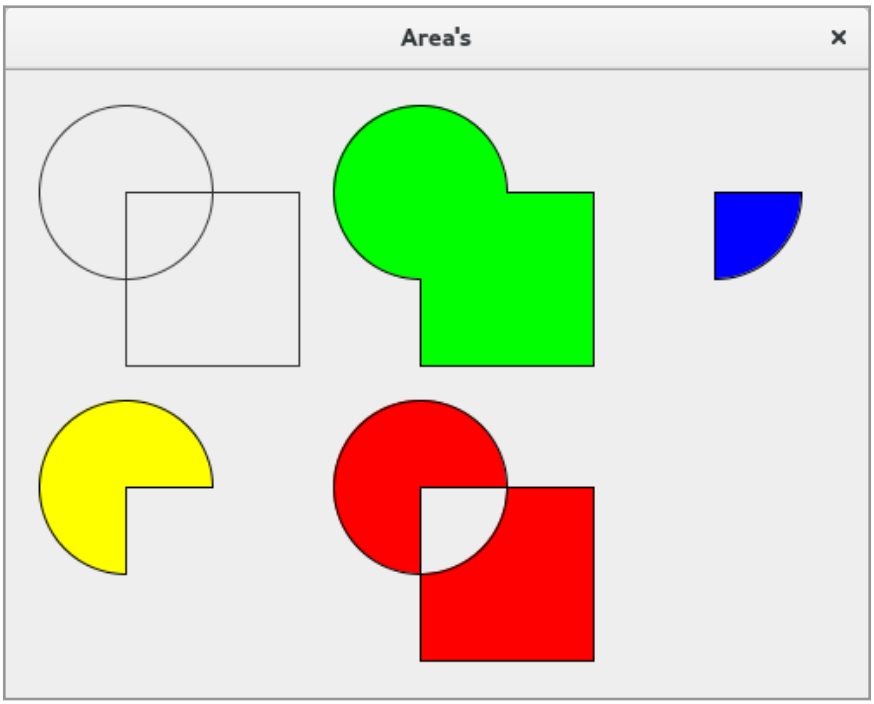
The following window draws a circle and a square, which overlap. These figures are used to define the four other figures by means of an *Area* object. Here you should notice the four methods that creates the *Area* objects, but also how they are placed in the window by translating the graphic object. This takes place with a transformation, which are discussed later, but in this case the figures is placed by a simple translation.

```
package areaprogram;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("Area's");
  setSize(500, 400);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}
```

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  Ellipse2D cirk = new Ellipse2D.Double(20, 20, 100, 100);
  Rectangle2D rect = new Rectangle2D.Double(70, 70, 100, 100);
  g2d.draw(cirk);
  g2d.draw(rect);
  union(g2d, cirk, rect, 170, 0);
  intersection(g2d, cirk, rect, 170, 0);
  subtraction(g2d, cirk, rect, -340, 170);
  exclusive(g2d, cirk, rect, 170, 0);
 }

 private void union(Graphics2D g, Shape shape1,
Shape shape2, int x, int y)
 {
  Area area1 = new Area(shape1);
  Area area2 = new Area(shape2);
  area1.add(area2);
  g.translate(x, y);
  g.setColor(Color.green);
  g.fill(area1);
  g.setColor(Color.black);
  g.draw(area1);
 }

 private void intersection(Graphics2D g, Shape shape1, Shape shape2, int x, int y)
 {
  Area area1 = new Area(shape1);
  Area area2 = new Area(shape2);
  area1.intersect(area2);
  g.translate(x, y);
  g.setColor(Color.blue);
  g.fill(area1);
  g.setColor(Color.black);
  g.draw(area1);
 }
```

```
private void subtraction(Graphics2D g, Shape shape1, Shape shape2, int x, int y)
{
 Area area1 = new Area(shape1);
 Area area2 = new Area(shape2);
 area1.subtract(area2);
 g.translate(x, y);
 g.setColor(Color.yellow);
 g.fill(area1);
 g.setColor(Color.black);
 g.draw(area1);
}
```

```
 private void exclusive(Graphics2D g, Shape shape1, Shape shape2, int x, int y)
 {
  Area area1 = new Area(shape1);
  Area area2 = new Area(shape2);
  area1.exclusiveOr(area2);
  g.translate(x, y);
  g.setColor(Color.red);
  g.fill(area1);
  g.setColor(Color.black);
  g.draw(area1);
 }
}
```



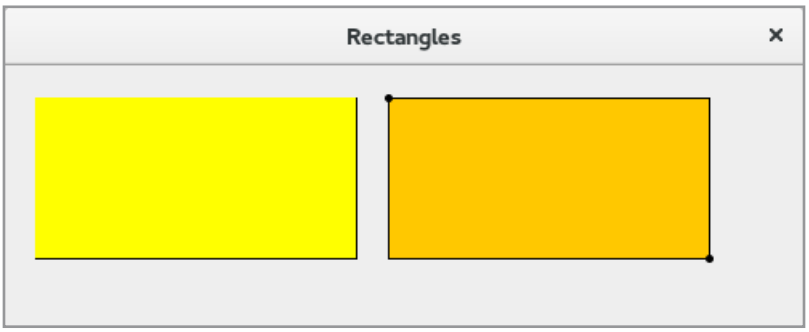
3 FILLING AND STROKING

In the previous chapter, I described how Java2D defines geometric shapes, and how to draw them in a window. To draw a shape includes

1. how should the figure's circumference be drawn
2. how should the interior of the figure be filled (if there is an interior)

and both are the subject of this chapter.

If you has a closed shape, such as a rectangle you can draw its perimeter with *draw()*, and you can fill the shape with *fill()*. In the window below is drawn two rectangles



The goal is to show how the perimeter is drawn. The first rectangle is drawn as follows:

```
Rectangle2D rect1 = new Rectangle2D.Double(20, 20, 200, 100);
g2d.setPaint(Color.black);
g2d.draw(rect1);
g2d.setPaint(Color.yellow);
g2d.fill(rect1);
```

You should note that I first draw the perimeter and then fills the rectangle with a yellow color. The result is that the perimeter is overwritten at the top and to the left while it is visible on the right and at the bottom. When the frame is drawn, 4 straight lines are drawn:

- a line from (20, 20) to (220, 20), that is the top edge
- a line from (20, 20) to (20, 120), that is the left edge
- a line from (20, 120) to (220, 120), that is the bottom edge
- a line from (220, 20) to (220, 120), that is the right edge

When the rectangle is filled, the area where the upper left corner is (20, 20) while the lower right corner is (219, 119) is colored. It is important to be aware of this fact, and the general rule is that you have to fill a figure before to draw the perimeter.

You should note that I in the program above have defined the color with *setPaint()* instead of *setColor()*. In this case there is no particular reason for it, but the class *Graphics2D* have a variable of type *Paint*, representing how a shape or line should be colored. There are three options:

1. a *Color*
2. a *GradientPaint*
3. a *TexturePaint*

The three options are illustrated below, but *Paint* is an interface that the class *Color* implements, and therefore you can used *setPaint()* instead of *setColor()*.

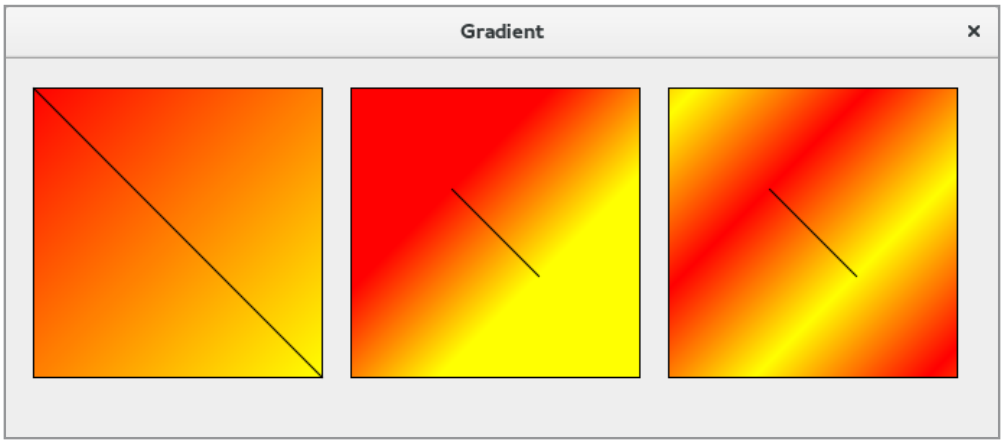
The second shape in the window below are drawn as follows:

```
Rectangle2D rect2 = new Rectangle2D.Double(240, 20, 200, 100);
Ellipse2D ellip1 = new Ellipse2D.Double(238, 18, 5, 5);
Ellipse2D ellip2 = new Ellipse2D.Double(438, 118, 5, 5);
g2d.setPaint(Color.orange);
g2d.fill(rect2);
g2d.setPaint(Color.black);
g2d.draw(rect2);
g2d.setPaint(Color.black);
g2d.fill(ellip1);
g2d.fill(ellip2);
```

and should again illustrate where the edge is drawn. In order to clarify the two corners of the rectangle, I have drawn two circles, both of which have a diameter of 5. The first has in its center (240, 20) and thus in the upper left corner of the rectangle. The second circle has its center at (440, 120) and thus in the lower right corner of the rectangle defining the shape.

3.1 GRADIENTPAINT

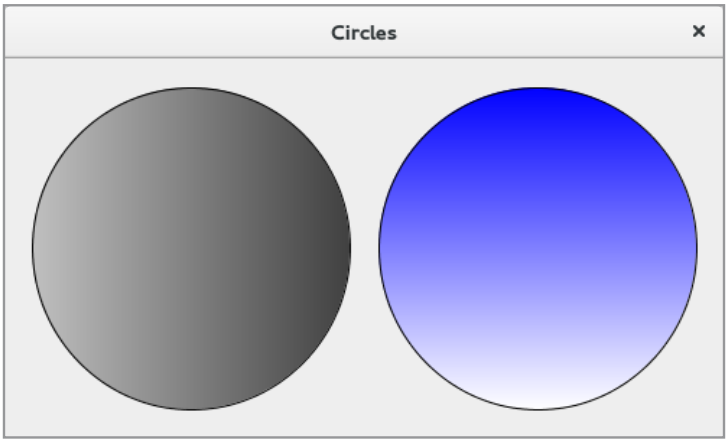
You can also fill a shape (and even draw a line with) a *GradientPaint*. Here you mix two colors along a straight line. Consider the following window:



showing three squares with a straight line. The straight lines are only drawn to explain the principle. The first figure mixes the colors red and yellow along the displayed line such that you start with red in the upper left corner and ending with yellow in the bottom right corner. The middle figure blends similar colors red and yellow, but this time the line does not span across the hole square. The result is that all before the starting of the line is red, while all after the line is yellow. The final shape is similar in principle, but this time the pattern is repeated. The component representing the three squares are:

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  Rectangle2D rect1 = new Rectangle2D.Double(20, 20, 200, 200);
  Rectangle2D rect2 = new Rectangle2D.Double(240, 20, 200, 200);
  Rectangle2D rect3 = new Rectangle2D.Double(460, 20, 200, 200);
  g2d.setPaint(new GradientPaint(20, 20, Color.red, 220, 220, Color.yellow));
  g2d.fill(rect1);
  g2d.setPaint(Color.black);
  g2d.draw(rect1);
  g2d.draw(new Line2D.Double(20, 20, 220, 220));
  g2d.setPaint(new GradientPaint(310, 90, Color.red, 370, 150, Color.yellow));
  g2d.fill(rect2);
  g2d.setPaint(Color.black);
  g2d.draw(rect2);
  g2d.draw(new Line2D.Double(310, 90, 370, 150));
  g2d.setPaint(new GradientPaint(
   530, 90, Color.red, 590, 150, Color.yellow, true));
  g2d.fill(rect3);
  g2d.setPaint(Color.black);
  g2d.draw(rect3);
  g2d.draw(new Line2D.Double(530, 90, 590, 150));
 }
}
```

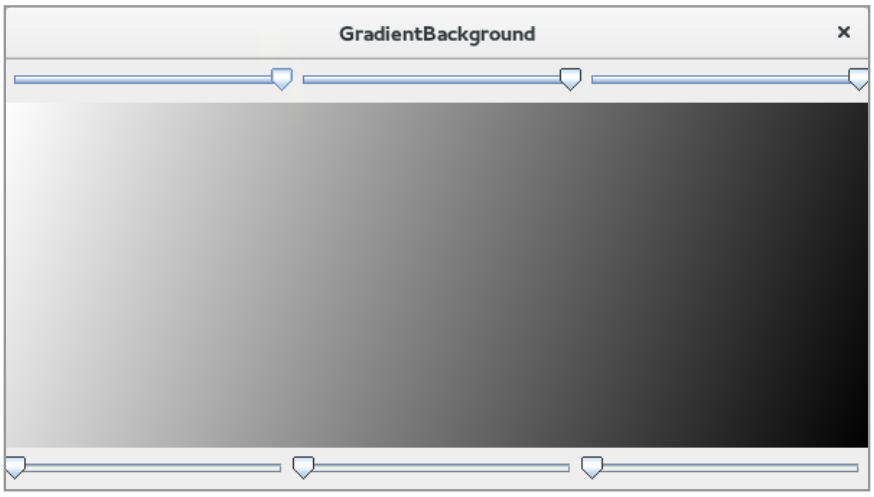
EXERCISE 7



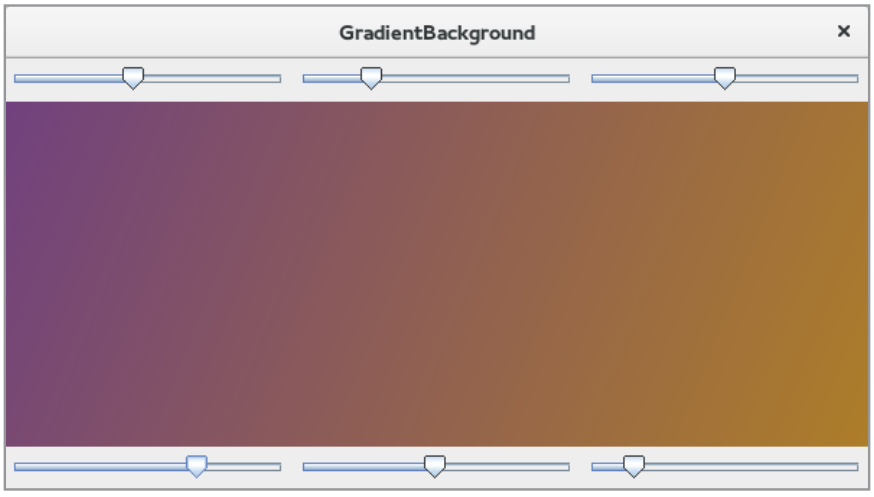
Write a program that you can call *Circles*, which paints the window as shown above. Here is the left ellipse is filled with a *GradientPaint* that mixes the colors light gray and dark gray along the horizontal diameter, while the right ellipse mixes the colors blue and white along the vertical diameter. The two ellipses must follow the window size and change when the size of the window changes.

EXERCISE 8

In this exercise, you must write a program, you can call *GradientBackground* which opens the following window:

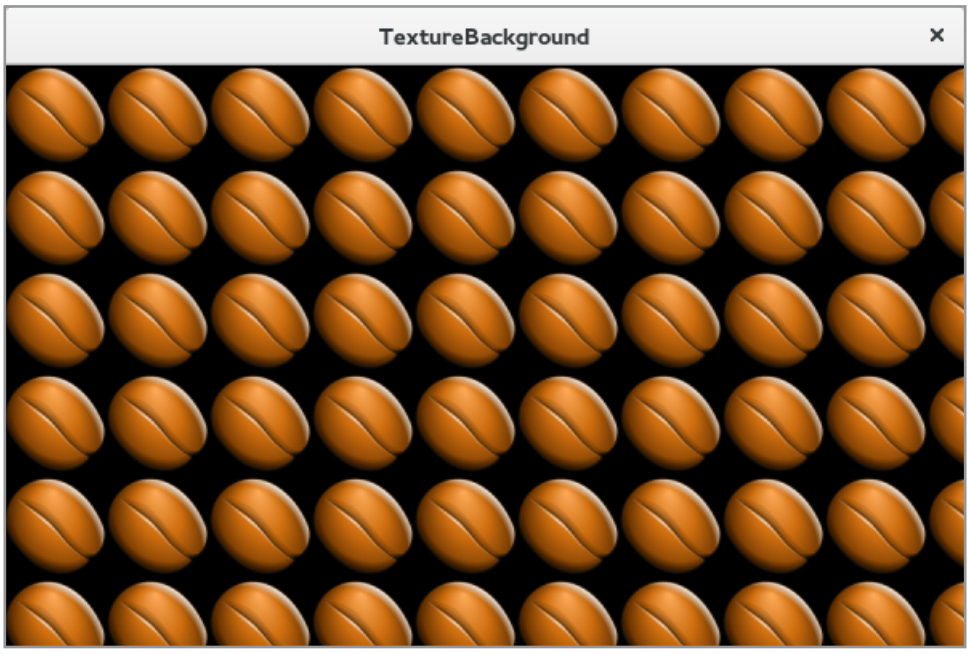


The window shows a rectangle (which follows the window size), which is filled with a *GradientPaint* that mixes white and black along the diagonal from the upper left corner. The window has three *JSlider* components respectively at the top and the bottom. They are used to define respectively the "white" and the "black" color. When you move the sliders, the result could, for instance be as shown below. Note that when the component representing the rectangle must refer to the six *JSlider* components, you can facilitate the work by defining the class *Drawing* as an inner class in the *MainWindow*.



3.2 TEXTUREPAINT

It is also possible to fill a figure with an image. The following program draws a rektangle and fills the entire window and the rectangle is filled with repetition of an icon of 64x64 pixels:



The project has a sub-package called images, and it contains the icon (a png file that is copied here, so the picture is a resource and packaged with the program's classes). The code for the window is:

```
package texturebackground;

import java.awt.*;
import java.awt.image.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("TextureBackground");
  setSize(600, 400);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}
```

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Dimension d = getSize();
  Graphics2D g2d = (Graphics2D)g;
  Rectangle2D rect = new Rectangle2D.Double(0, 0, d.width, d.height);
  BufferedImage image = createImage();
  g2d.setPaint(
   new TexturePaint(image,
   new Rectangle2D.Double(0, 0, image.getWidth(), image.getHeight())));
  g2d.fill(rect);
 }

 public BufferedImage createImage()
 {
  java.net.URL imgURL =
   Drawing.class.getResource("/texturebackground/images/Bean.png");
  ImageIcon icon = new ImageIcon(new ImageIcon(imgURL, "").
   getImage().getScaledInstance(64, 64, Image.SCALE_SMOOTH), "");
  BufferedImage image = new BufferedImage(icon.getIconWidth(),
   icon.getIconHeight(), BufferedImage.TYPE_INT_RGB);
  Graphics g = image.createGraphics();
```

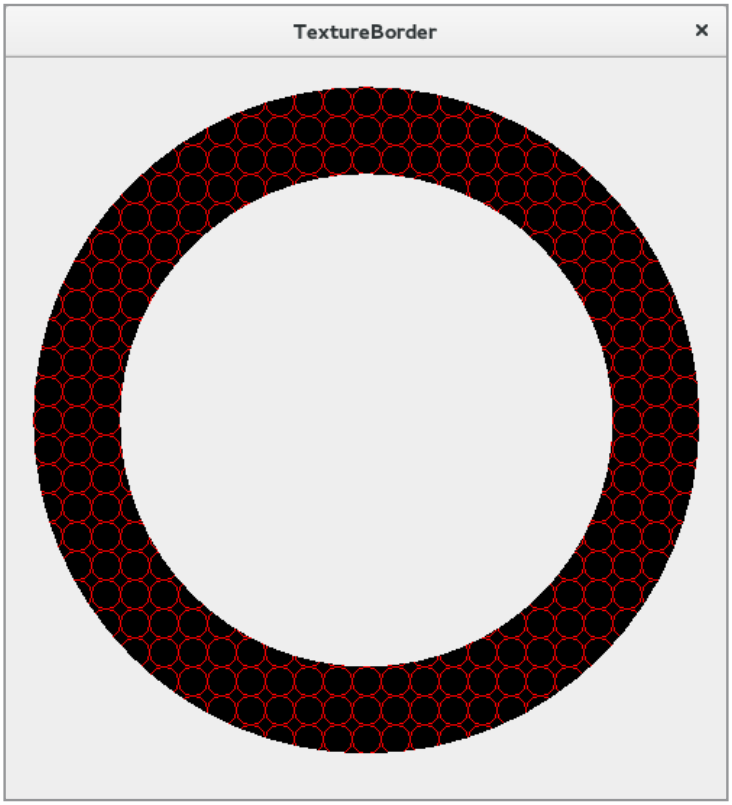
```
  icon.paintIcon(null, g, 0,0);
  g.dispose();
  return image;
 }
}
```

The method *createImage()* creates a *BufferedImage*. This class is explained later, but you can think of it as a class, that allows you to manimulere the pixels in an image. In this case, the image is in the application's jar file and is loaded as an *ImageIcon* object. Next, the metode creates a *BufferedImage* of the same size as the object, and the object's *Graphics* object is used to draw the object. With this object available you in *paintComponent()* can define a *TexturePaint* to fill the rectangle.

As another example of using a *TexturePaint* is shown below a window with a circle. The code window is:

```
package textureborder;

import java.awt.*;
import java.awt.image.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("TextureBorder");
  setSize(500, 550);
  setLocationRelativeTo(null);
  add(new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}

class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Dimension d = getSize();
  Graphics2D g2d = (Graphics2D)g;
  Ellipse2D ellip = new Ellipse2D.Double(50, 50, 400, 400);
  BufferedImage image = createImage();
```

```
  g2d.setPaint(new TexturePaint(image,
    new Rectangle2D.Double(0, 0, image.getWidth(), image.getHeight())));
  g2d.setStroke(new BasicStroke(60));
  g2d.draw(ellip);
 }

 public BufferedImage createImage()
 {
  BufferedImage image = new BufferedImage(20, 20, BufferedImage.TYPE_INT_RGB);
  Graphics2D g = (Graphics2D)image.createGraphics();
  g.setColor(Color.red);
  g.draw(new Ellipse2D.Double(0, 0, 20, 20));
  g.dispose();
  return image;
 }
}
```



The method *createImage()* creates a *BufferedImage* of 20×20 pixels, where is drawen a red circle. You can create and manipulate a *BufferedeImage* without loading an image from a file. The *BufferedImage* is created in *paintComponent()* to create a *TexturePaint*, and then draw a circle with a stroke on 60 pixels.

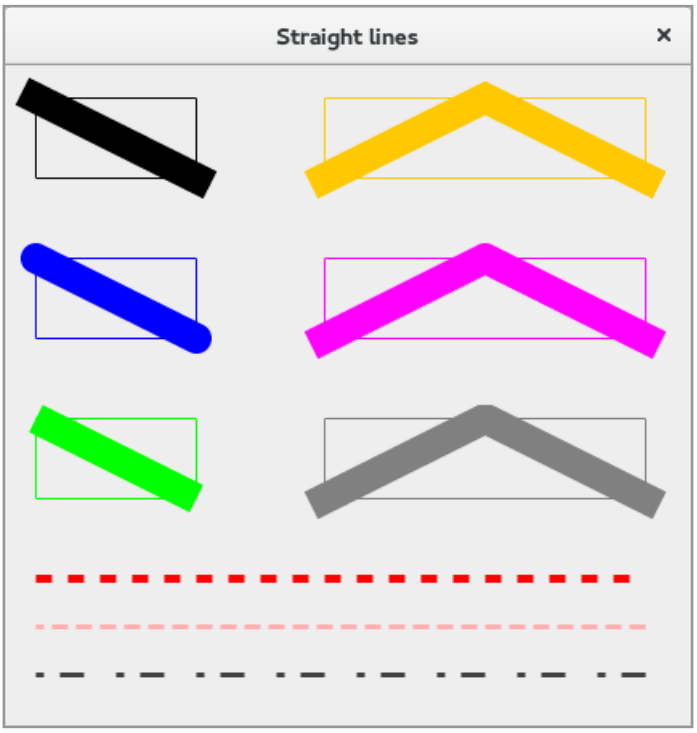
In addition to show how to create a *BufferedImage* from scratch, the example also shows that the perimeter of a figure in itself is a *Shape* that can be filled (by the method *draw()*) with a *Paint* object.

3.3 STROKES

In the above examples, I have shown how to define a *BasicStroke* object and thus indicate how thick a line has to be drawn. Default is 1. However, there are two things concerning strokes, you must have knowledge of:

- how the line's endpoints must be
- how line segments are put together and thus how corners must be

where these concepts only have interest if you has to draw thick lines. Finally, one must know how to draw a dotted line. The following window shows most of the options available:



As regards the line's endpoints are three possibilities:

1. *BasicStroke.CAP_SQUARE* (that is default)
2. *BasicStroke.CAP_ROUND*
3. *BasicStroke.CAP_BUTT*

The first line (the black) is drawn with default values and otherwise a stroke at 19. Furthermore, there is drawn a rectangle that is the line's circumscribing rectangle. Concerning the endpoints, the style is *CAP_SQUARE* that expands the line with a rectangle whose length is half the line thickness. The blue line has corresponding the endpoint *CAP_ROUND* that expands the line with a half-circle whose radius is half the line thickness. Finally, there is *CAP_BUTT* that is used by the green line and who simply do not expand the line.

The three figures on the right have all type *GeneralPath* and is drawn with a stroke on 19 and the option *CAP_SQUARE* for the endpoints, and also is defined how the corners will be drawn. Again, there are three options:

1. *JOIN_MITER* (that is default)
2. *JOIN_ROUND*
3. *JOIN_BEVEL*

The orange path is drawn with default values and thus the style *JOIN_MITER*. It mixes the segments by extending the outer edges until they intersect. The magenta line uses *JOIN_ROUND* which simply means that the line segments endpoints are using CAP_ROUND. Finally, *JOIN_BEVEL* is used by the gray path. Here the line segments endpoints uses *CAP_BUTT* and the outer edges are connected.

Then there are the three dashed lines. The red is drawn with a stroke that is defined as follows:

```
new BasicStroke(5,
 BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL, 0, new float[] { 10, 10 }, 0)
```

A dotted line is drawn by switching between a line segment and a blank segment. The last array defines that the two pieces should each be at length of 10. The last parameter is explained by the next example (the line pink):

```
new BasicStroke(3,
 BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL, 0, new float[] { 10, 5 }, 5)
```

Here the last parameter indicates, how long you have to start inside the first segment. Therefore, the first segment only has the length 5. Finally, the last example show that you can specify multiple segment lengths:

```
new BasicStroke(3, BasicStroke.CAP_BUTT,
 BasicStroke.JOIN_BEVEL, 0, new float[] { 5, 10, 15, 20 }, 0)
```

That is, first draw a line segment at 5, then a gap of 10, a segment of 15 and finally a gap of 20. Then repeat the pattern until the entire line is drawn. The code for the component is shown below:

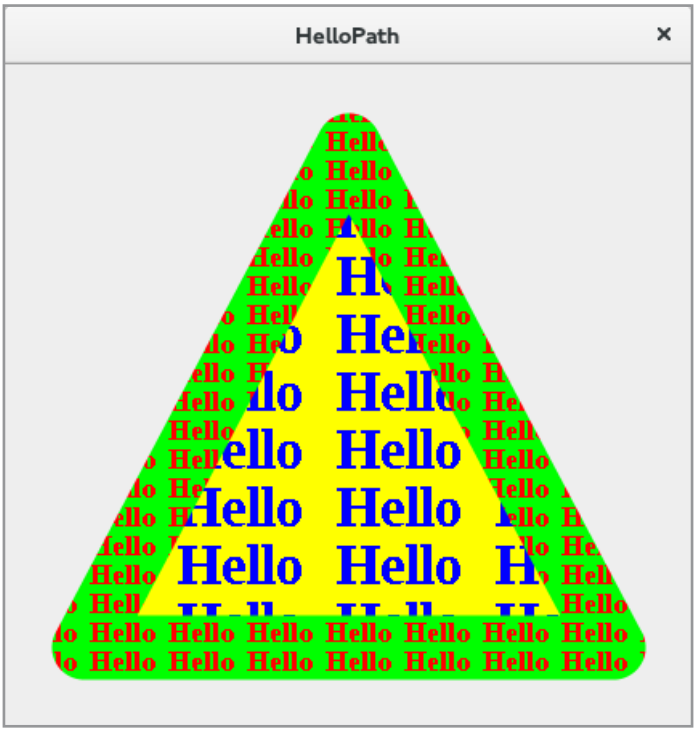
```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  draw(g2d, 20, 20, 120, 70, new BasicStroke(19), Color.black, true);
  draw(g2d, 20, 120, 120, 170, new BasicStroke(19, BasicStroke.CAP_ROUND,
   BasicStroke.JOIN_BEVEL), Color.blue, true);
  draw(g2d, 20, 220, 120, 270, new BasicStroke(19, BasicStroke.CAP_BUTT,
   BasicStroke.JOIN_BEVEL), Color.green, true);
  draw(g2d, 200, 70, 300, 20, 400, 70, new BasicStroke(19), Color.orange);
```

```
 draw(g2d, 200, 170, 300, 120, 400, 170, new BasicStroke(19,
  BasicStroke.CAP_SQUARE,BasicStroke.JOIN_ROUND), Color.magenta);
 draw(g2d, 200, 270, 300, 220, 400, 270,
  new BasicStroke(19, BasicStroke.CAP_SQUARE,
  BasicStroke.JOIN_BEVEL), Color.gray);
 draw(g2d, 20, 320, 400, 320, new BasicStroke(5, BasicStroke.CAP_BUTT,
  BasicStroke.JOIN_BEVEL, 0, new float[] { 10, 10 }, 0), Color.red, false);
 draw(g2d, 20, 350, 400, 350, new BasicStroke(3, BasicStroke.CAP_BUTT,
  BasicStroke.JOIN_BEVEL, 0, new float[] { 10, 5 }, 5), Color.pink, false);
 draw(g2d, 20, 380, 400, 380, new BasicStroke(3, BasicStroke.CAP_BUTT,
  BasicStroke.JOIN_BEVEL, 0,
  new float[] { 5, 10, 15, 20 }, 0), Color.darkGray, false);
}

private void draw(Graphics2D g, double x1, double y1, double x2, double y2,
 Stroke stroke, Color color, boolean showBound)
{
 g.setStroke(stroke);
 g.setPaint(color);
 Line2D line = new Line2D.Double(x1, y1, x2, y2);
 g.draw(line);
 if (showBound)
 {
```

```
    g.setStroke(new BasicStroke(0.5F));
    g.draw(line.getBounds2D());
   }
  }

 private void draw(Graphics2D g, double x1,
double y1, double x2, double y2,
  double x3, double y3, Stroke stroke, Color color)
 {
  g.setStroke(stroke);
  g.setPaint(color);
  GeneralPath path = new GeneralPath();
  path.moveTo(x1, y1);
  path.lineTo(x2, y2);
  path.lineTo(x3, y3);
  g.draw(path);
  g.setStroke(new BasicStroke(0.5F));
  g.draw(path.getBounds2D());
 }
}
```

EXERCISE 9

Write a program, that opens the folowing window:



4 RENDERING

In the previous chapter, I have discussed how to define geometric shapes. In this chapter I will look at how Java2D renders figures. That is how the follewing is:

1. transformation
2. compositing
3. clipping
4. rendering hints

4.1 TRANSFORMATIONS

A transformation is defined by the class called *AffineTransform*, and an affine transformation is a transformation that preserves parallel lines. The class *Graphics2D* has an instance of an *AffineTransform* applicable to all geometric shapes, when rendered. This transfomation can be changed in two ways. One can partly replace it with another one:

```
void setTransform(AffineTransform tr)
```

and you can change the existing transformation

```
void transform(AffineTransform Tx)
```

It is generally recommended that you use the latter approach, and examples shows how. The class *Graphics2D* also has methods that directly perform simple transformations. There are four types of transformations

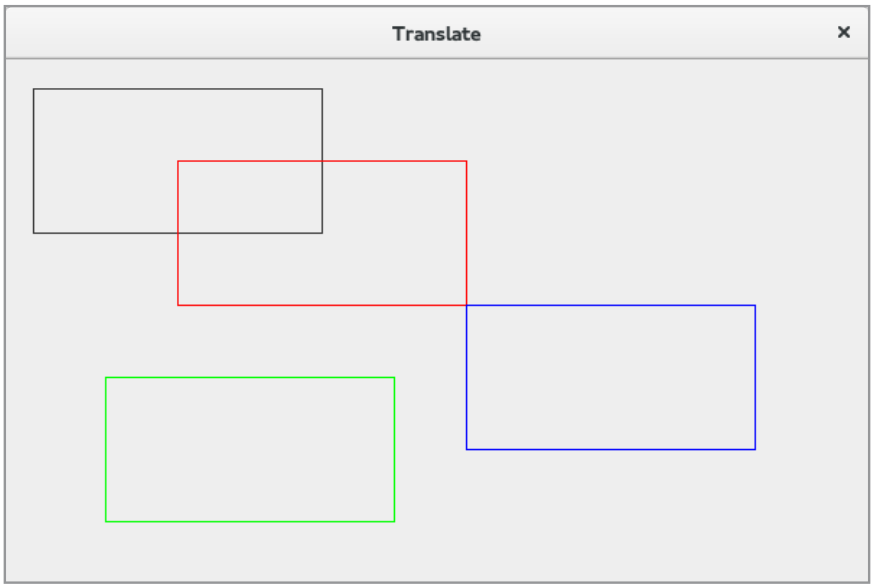
- *translation* (there is a parallel displacement)
- *rotation* (that rotates the figure around a point)
- *scaling* (that scales a figure)
- *shearing* (that shears a figure)

Translation

The program *TranslateDemo* opens a window as shown below. The window draws 4 rectangles and draw first the black rectangle. The other three are the same rectangle, but are drawn after a parallel shift of the graphics (drawing surface). Immediately it is simple, but if you made several transformations (as in this case), the result can be difficult to grasp. The component's code is as follows and should also show that a parallel shift can be defined in several ways:

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  Rectangle2D rect = new Rectangle2D.Double(20, 20, 200, 100);
  g2d.draw(rect);
  AffineTransform start = g2d.getTransform();
  g2d.translate(100, 50);
  g2d.setPaint(Color.red);
  g2d.draw(rect);
  AffineTransform trans = g2d.getTransform();
  trans.translate(100, 50);
  g2d.transform(trans);
  g2d.setPaint(Color.blue);
  g2d.draw(rect);
  g2d.setTransform(start);
  AffineTransform translate = AffineTransform.getTranslateInstance(50, 200);
```

```
    g2d.transform(translate);
    g2d.setPaint(Color.green);
    g2d.draw(rect);
  }
}
```



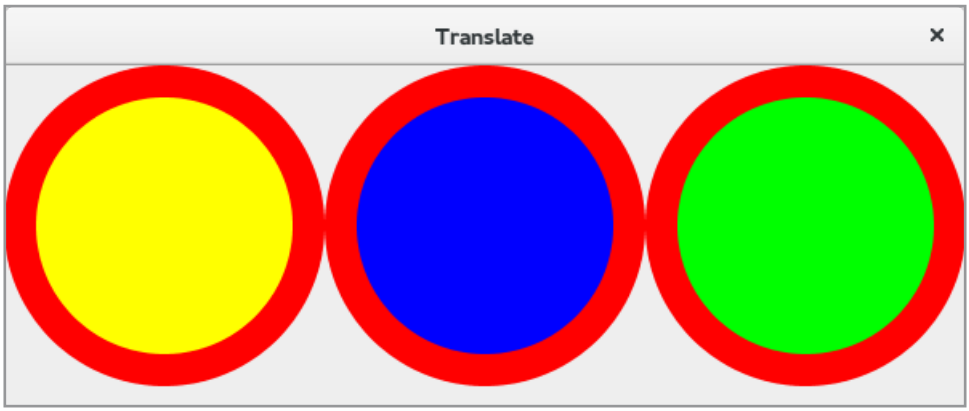
First, is defined a rectangle *rect*, that is drawn with default settings, and the result is the black rectangle. Next, a reference *start* is added for the current transformation. The goal is to enable the transformation of the drawing surface to be reset to default, after performing other transformations. Next the method performs the statement

```
g2d.translate(100, 50);
```

and the rectangle is drawn with a red color. It is a parallel shift of 100 in the x- axis direction and 50 in the y-axis direction. This means that the coordinate system's start point (the upper left corner) is shifted to (100, 50). You should note that after the method is performed, the *Graphics2D* object's *AffineTransform* is changed. As the next step is defined a reference *trans* to the current *AffineTransform* and then a parallel shift to (100, 50), that is used to modify *g2d* object's transformation, and the rectangle is drawn again, but this time with a blue color. Here, the result is not entirely obvious since it is a composite transformation. First performs the parallel (100, 50), which moves the current start point (100, 50) to (200, 100). Next, the first transformation, which is also a parallel shift (100, 50), that moves the start point (200,100) to (300, 150). The result is that the blue rectangle has the upper left corner in (320,170). After the blue rectangle is drawn, the transformation is reset to default, and a new parallel shift using a static method in the class *AffineTransform* is defined. The results in the green rectangle.

EXERCISE 10

Write a program that creates the following window:

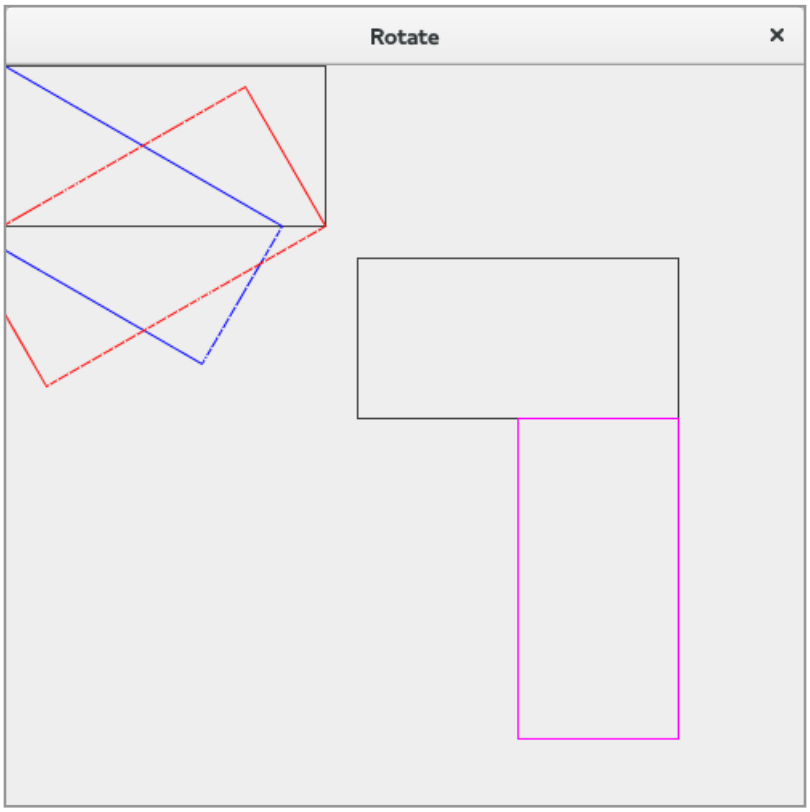


when the blue and the green circle must be transformations (parallel shifts) of the yellow.

Rotation

Rotations works in principle in the same way as the parallel shift and can likewise be defined in three different ways. In the following example, I've only used one way, but the two other methods works just like with parallel shifts. A rotation rotates the coordinate system, and thereby the drawing surface about a point. By default it is the point (0, 0), but you can also rotate about any other point, and the result is a parallel shift followed by a rotation. The rotation is specified by an angle measured in radians.

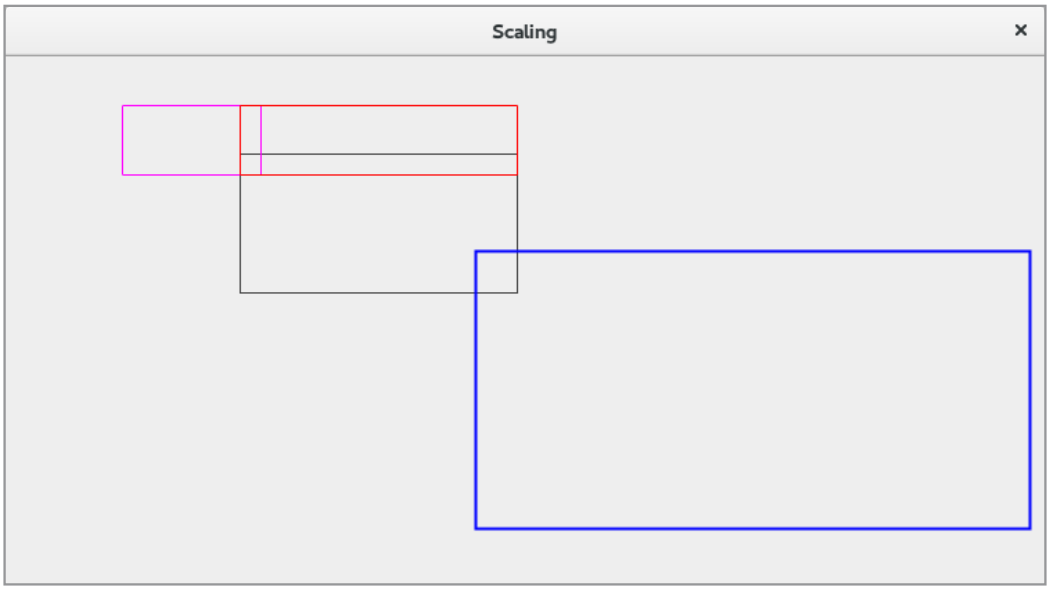
Below is a window that rotates two rectangles (the black rectangles). One rectangle (the blue) is rotated $\pi/6$ around the zero point (upper left corner of the rectangle), whereas the second (the red rectangle) is rotated the angle $-\pi/6$ around the rectangle's lower right corner. The magenta rectangle is a rotation of another black rectangle with $-\pi/2$ angle around the lower right corner.


Rotate

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  Rectangle2D rect1 = new Rectangle2D.Double(0, 0, 200, 100);
  Rectangle2D rect2 = new Rectangle2D.Double(220, 120, 200, 100);
  g2d.setPaint(Color.darkGray);
  g2d.draw(rect1);
  g2d.draw(rect2);
  AffineTransform start = g2d.getTransform();
  g2d.rotate(Math.PI / 6);
  g2d.setPaint(Color.blue);
  g2d.draw(rect1);
  g2d.setTransform(start);
  g2d.rotate(-Math.PI / 6, 200, 100);
  g2d.setPaint(Color.red);
  g2d.draw(rect1);
  g2d.setTransform(start);
  g2d.rotate(-Math.PI / 2, 420, 220);
  g2d.setPaint(Color.magenta);
  g2d.draw(rect2);
 }
}
```

Scaling

It is a transformation in which a figure is resized – both horizontally and vertically. The following example scales a rectangle:



```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  Rectangle2D rect = new Rectangle2D.Double(170, 70, 200, 100);
  g2d.setPaint(Color.darkGray);
  g2d.draw(rect);
  AffineTransform start = g2d.getTransform();
  g2d.scale(2, 2);
  g2d.setPaint(Color.blue);
  g2d.draw(rect);
  g2d.setTransform(start);
  g2d.scale(0.5, 0.5);
  g2d.setPaint(Color.magenta);
  g2d.draw(rect);
  g2d.setTransform(start);
  g2d.scale(1, 0.5);
  g2d.setPaint(Color.red);
  g2d.draw(rect);
 }
}
```

The rectangle that is scaled, is the black rectangle. You should note that when you scale a figure, it is both the figures shape and location that are changed. When the location is changed, it is because it is all distances from the coordinate system start point that is scaled. The black rectangle's upper left corner is (170, 70), and the blue rectangle has its left upper corner at (340, 140) and the width of 400 and the height of 200. Similarly, the upper left corner of the magenta rectangle's coordinates is (85, 35) and the width 100 and the height 50, and the red rectangle is the only scale vertically, and its top left corner is (170, 35). The width is unchanged 200 while the height is 50.

EXERCISE 11

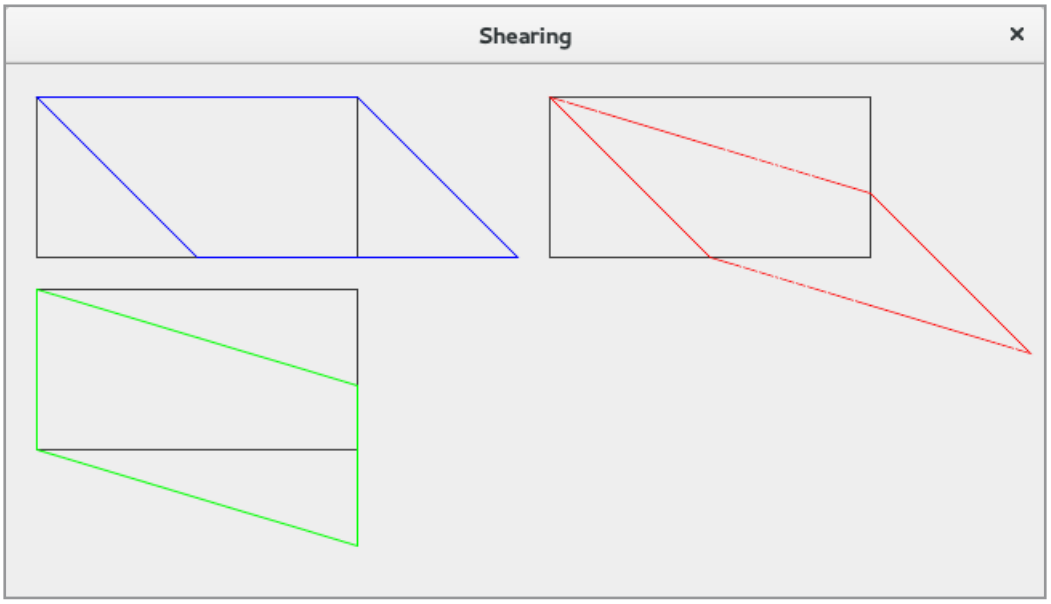
Create a copy of the project *ScaleDemo* (the program above). You must modify the program so that all scaling is based on the black rectangle's upper left corner.

Stretching

The last of the basic transformations is called shearing, where the x-axis, y-axis, or both axes are stretched. The window below shows how to stretch a rectangle (the black rectangle). In all three cases the transformation is a parallel shift of a rectangle with the upper left corner in (0, 0) followed by a shear transformation. This means that all three transformations are a composite transformation. In principle, it is simple enough to compose transformations, but the result can be difficult to understand, and the order is of significance.

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  Rectangle2D rect = new Rectangle2D.Double(0, 0, 200, 100);
  AffineTransform start = g2d.getTransform();
  g2d.translate(20, 20);
  g2d.setPaint(Color.darkGray);
  g2d.draw(rect);
  g2d.shear(1, 0);
```

```
    g2d.setPaint(Color.blue);
    g2d.draw(rect);
    g2d.setTransform(start);
    g2d.translate(20, 140);
    g2d.setPaint(Color.darkGray);
    g2d.draw(rect);
    g2d.setPaint(Color.green);
    g2d.shear(0, 0.3);
    g2d.draw(rect);
    g2d.setTransform(start);
    g2d.translate(340, 20);
    g2d.setPaint(Color.darkGray);
    g2d.draw(rect);
    g2d.setPaint(Color.red);
    g2d.shear(1, 0.3);
    g2d.draw(rect);
  }
}
```



4.2 COMPOSITING

Compositing refers to the process to put two images together and appears each time a shape or image is drawn. *Graphics2D* representing as mentioned a drawing surface, and every time you on this surface draw a shape, some text or an image, this adds a new element to the drawing surface. This takes place in principle using three operations:

1. *Rasterizing* produce from the current figure *alpha values* that you can think of as a grid of pixels, each having an alpha value. In principle, this grid corresponds to the entire drawing surface and are referred to below for the source. In contrast, the drawing surface is called the destination.
2. For each pixel in the source is determined the color based either on the current *Paint* object used by the *Graphics2D* object or an image.
3. Source and destination (color and alpha value) are combined pixel by pixel to form the image on the drawing surface. The combination of source and destination is done from a compositing rule that exactly specify how color and alpha values has to be used.

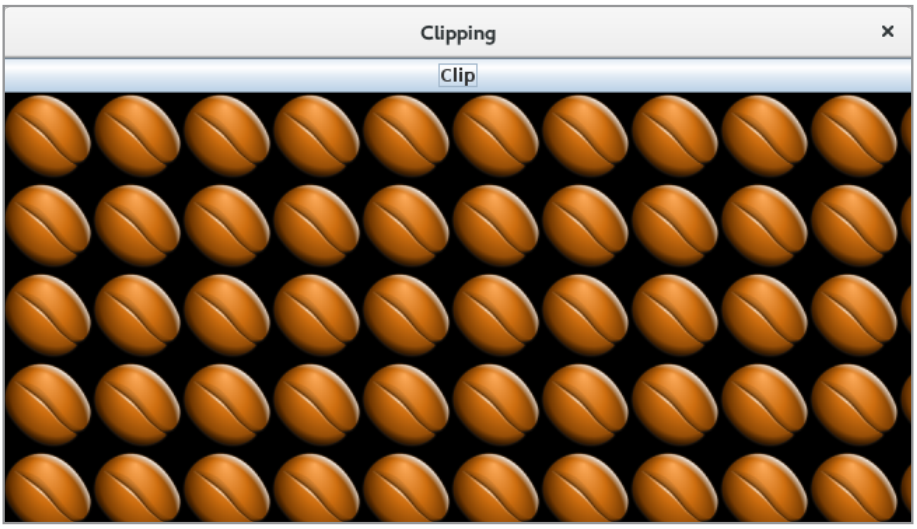
If, for example you look at what happens when a black figure is rendered on a white drawing surface and using the default compositing rule, you can describe the process as follows. Rasterizing convert the figure to an array of alpha values where pixels completely out of the shape are given the value 0.0 (interpreted as transparent), and the corresponding pixels in the destination will remain unchanged. Similarly, pixels in the interior of the figure have the alpha value 1.0, and the corresponding pixels in the destination will get the same color as sourcens pixels, and the result will in this case be a black color.

Then there are the pixels, which is located on the edge, that get an alpha value between 0.0 and 1.0 (assuming the use antialiasing). They have a color that is a combination of the colors of pixels in the source and destination, and in this case it will be a combination of black and white and thus a shade of gray. The exact color is determined by the alpha value in the source. A pixel with an alpha value of 0.9, for example give a very dark gray color.

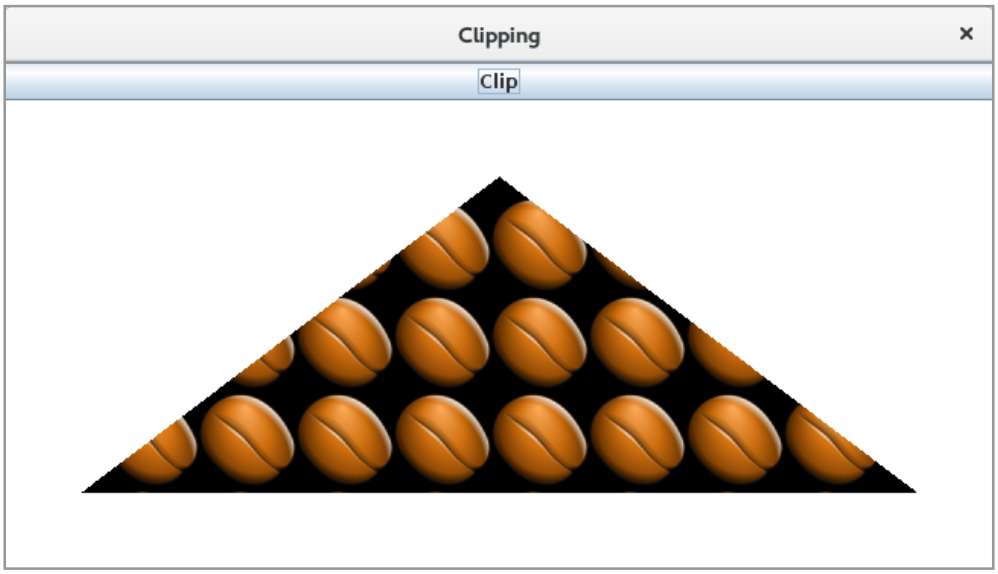
The above is referred to as the compositing default rule, but actually others may be used in special cases. They are called *Porter-Duff rules*, but are not discussed here.

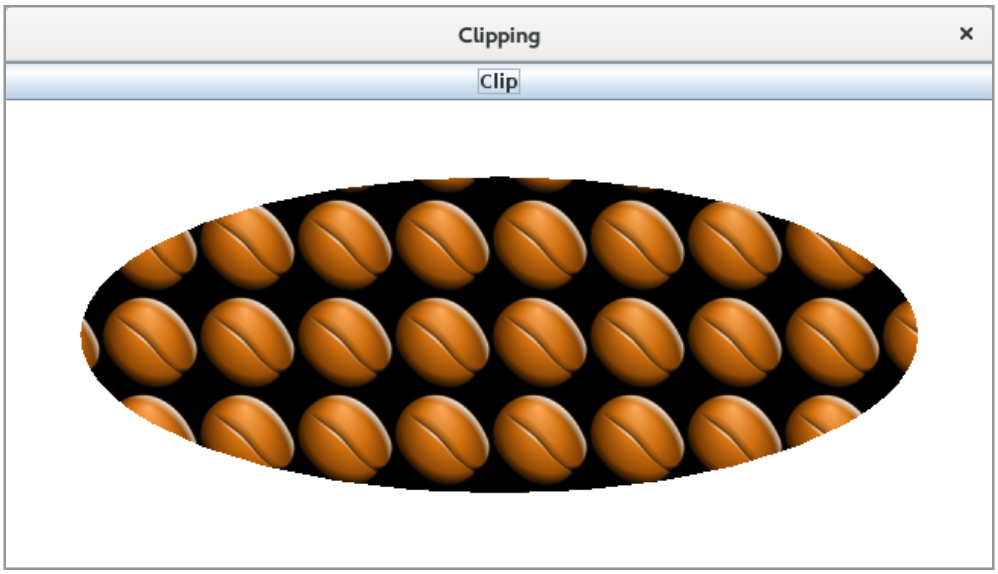
4.3 CLIPPING

Sometimes one is not interested that you can draw on the entire drawing area. When you build complex shapes, there is often a need to ensure that you does not draw on other figures, and it can be ensured by specifying a clip area. Consider the following window, which is taken from an earlier example:



The difference is only that there has been added a button at the top of the window. If you click the button, the window is redrawn, and the result is as shown below, which draw a triangle. This is done by defined a clip area as a triangle, and it is only that part of the component that is redrawn. Click the button once, the result is an ellipse, as there is now defined a clip area is an ellipse. Clicking a third time, you return to the start. It is simple to define a clip area as the code below shows, and you should note that the area can be any Shape:






```
package clipdemo;

import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 private JButton cmdClip = new JButton("Klip");
 private int state = 0;
```

```
public MainWindow()
{
 super("Clipping");
 setSize(650, 370);
 setLocationRelativeTo(null);
 add(cmdClip, BorderLayout.NORTH);
 add(new Drawing());
 cmdClip.addActionListener(new ClickHandler());
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
}

class ClickHandler implements ActionListener
{
 public void actionPerformed(ActionEvent e)
 {
  state = (state + 1) % 3;
  repaint();
 }
}

class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Dimension d = getSize();
  Graphics2D g2d = (Graphics2D)g;
  Rectangle2D rect = new Rectangle2D.Double(0, 0, d.width, d.height);
  g2d.setPaint(Color.white);
  g2d.fill(rect);
  BufferedImage image = createImage();
  g2d.setPaint(new TexturePaint(image,
   new Rectangle2D.Double(0, 0, image.getWidth(), image.getHeight())));
  if (state == 1)
  {
   GeneralPath path = new GeneralPath();
   path.moveTo(d.width / 2, 50);
   path.lineTo(d.width - 50, d.height - 50);
   path.lineTo(50, d.height - 50);
   path.closePath();
   g2d.setClip(path);
  }
  else if (state == 2)
  {
```

```
    Ellipse2D ellip =
      new Ellipse2D.Double(50, 50, d.width - 100, d.height - 100);
    g2d.setClip(ellip);
   }
   g2d.fill(rect);
  }

  public BufferedImage createImage()
  {
   java.net.URL imgURL = Drawing.class.getResource("/clipdemo/images/Bean.png");
   ImageIcon icon = new ImageIcon(new ImageIcon(imgURL, "").
    getImage().getScaledInstance(64, 64, Image.SCALE_SMOOTH), "");
   BufferedImage image = new BufferedImage(icon.getIconWidth(),
    icon.getIconHeight(), BufferedImage.TYPE_INT_RGB);
   Graphics g = image.createGraphics();
   icon.paintIcon(null, g, 0,0);
   g.dispose();
   return image;
  }
 }
}
```

4.4 RENDERING HINTS

As the last thing in this chapter I mention rendering hints, as I have already mentioned with antialiasing. It is an opportunity to check the quality of the figure being drawn. The renderings mechanism of *Graphics2D* can do most things in more than one way, and rendering hints are the programmer's wants respect to quality, but it is not certain that the mechanism can deliver the quality you wishes. You can define rendering hints as key / value pairs, and the possibilities is shown in the following table:

Key	Value
KEY_ANTIALIASING	VALUE_ANTIALIAS_ON VALUE_ANTIALIAS_OFF VALUE_ANTIALIAS_DEFAULT
KEY_RENDERING	VALUE_RENDER_QUALITY VALUE_RENDER_SPEED VALUE_RENDER_DEFAULT
KEY_DITHERING	VALUE_DITHER_DISABLE VALUE_DITHER_ENABLE VALUE_DITHER_DEFAULT
KEY_COLOR_RENDERING	VALUE_COLOR_RENDER_QUALITY VALUE_COLOR_RENDER_SPEED VALUE_COLOR_RENDER_DEFAULT
KEY_FRACTIONALMETRICS	VALUE_FRACTIONALMETRICS_ON VALUE_FRACTIONALMETRICS_OFF VALUE_FRACTIONALMETRICS_DEFAULT
KEY_TEXT_ANTIALIASING	VALUE_TEXT_ANTIALIAS_ON VALUE_TEXT_ANTIALIAS_OFF VALUE_TEXT_ANTIALIAS_DEFAULT
KEY_INTERPOLATION	VALUE_INTERPOLATION_BICUBIC VALUE_INTERPOLATION_BILINEAR VALUE_INTERPOLATION_NEAREST_NEIGHBOR
KEY_ALPHA_INTERPOLATION	VALUE_ALPHA_INTERPOLATION_QUALITY VALUE_ALPHA_INTERPOLATION_SPEED VALUE_ALPHA_INTERPOLATION_DEFAULT

So far I have only used

```
g2d.setRenderingHint(
RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
```

but I will later use other.

5 TEXT

Also text must be drawn, and even if one usually works with text in relation to Swing components, where you do not have to do anything, then text also somewhere must be drawn, a task which Java2D has to take care of. Actually, there are attached many details to text, which is the subject of this chapter, but basically text graphical objects can be manipulated in the same way as other graphic objects and shapes.

Text contains characters encoded as unicodes, and to draw a text, therefore, for each character the figure that represents the character must be found and rendered. Such a figure is called a *glyph*, and a font contains glyphs for all the characters that the font supports. Attached to a font you have font metrics indicating the sizes of each character. For the height there is three sizes. Characters are drawn on a baseline (the characters are standing on the baseline). *Descent* indicates how much the characters must fill below the baseline and *ascent* how much they can fill over the baseline. Finally specify *leading* how much space that should be between lines, and the three sizes together is called the font height. As an example of some of all that you can consider the following window:

Here is the black line is the baseline, while the distance between the black and the red line is the descent, the distance between the black and the magenta line is the ascent and the distance between the magenta line and the blue is the *leading*.

The component code is as shown below.

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  String text = "Frode Fredegod";
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
```

```
  Font font = new Font("Times New Roman", Font.PLAIN, 64);
  g2d.setFont(font);
  g2d.drawString(text, 50, 100);
  g2d.draw(new Line2D.Double(25, 100, 500, 100));
  FontMetrics metrics = g2d.getFontMetrics();
  g2d.setPaint(Color.blue);
  g2d.draw(new Line2D.Double(25, 100 - metrics.getAscent() -
   metrics.getLeading(), 500, 100 - metrics.getAscent() -
   metrics.getLeading()));
  g2d.setPaint(Color.magenta);
  g2d.draw(new Line2D.Double(25, 100 - metrics.getAscent(), 500, 100 -
   metrics.getAscent()));
  g2d.setPaint(Color.red);
  g2d.draw(new Line2D.Double(25, 100 + metrics.getDescent(), 500, 100 +
   metrics.getDescent()));
  int width = metrics.stringWidth(text);
  g2d.fill(new Rectangle2D.Double(width + 60, 90, 10, 10));
 }
}
```

Here you should specifically note the class *FontMetrics*, which contains information about a font, and you should note the method *stringWidth()*, which is used to measure how much a string fills in the current font.

One can in fact more with text, and for example you can associate an iterator to a text that can be used to manipulate the individual characters or a portion of a string. The program *AttributedText* opens a window as shown below:


```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Font serifFont = new Font("Serif", Font.PLAIN, 48);
  Font monoFont = new Font("Monospaced", Font.PLAIN, 48);
  String text = "Frode Fredegod, konge,,";
  AttributedString str = new AttributedString(text);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  str.addAttribute(TextAttribute.FONT, serifFont);
  str.addAttribute(TextAttribute.FONT, monoFont, 6, 14);
  str.addAttribute(TextAttribute.FOREGROUND, Color.red, 6, 14);
  str.addAttribute(TextAttribute.UNDERLINE, TextAttribute.UNDERLINE_ON, 6, 14);
  GeneralPath shape = new GeneralPath();
  shape.moveTo(0, -21);
  shape.lineTo(10, -40);
  shape.lineTo(20, -21);
  shape.closePath();
  shape.moveTo(0, -19);
  shape.lineTo(10, 0);
  shape.lineTo(20, -19);
  shape.closePath();
  ShapeGraphicAttribute shapeAttribute = new ShapeGraphicAttribute(shape,
   GraphicAttribute.ROMAN_BASELINE, ShapeGraphicAttribute.FILL);
  str.addAttribute(TextAttribute.CHAR_REPLACEMENT, shapeAttribute, 14, 15);
  str.addAttribute(TextAttribute.FOREGROUND, Color.gray, 14, 15);
  Shape space = new Rectangle2D.Double(0, 0, 60, 0);
  ShapeGraphicAttribute spaceAttribute = new ShapeGraphicAttribute(space,
   GraphicAttribute.ROMAN_BASELINE, ShapeGraphicAttribute.FILL);
  str.addAttribute(TextAttribute.CHAR_REPLACEMENT, spaceAttribute,
```

```
    text.length() - 2, text.length() - 1);
   Image image = createImage();
   ImageGraphicAttribute imageAttribute =
    new ImageGraphicAttribute(image, GraphicAttribute.TOP_ALIGNMENT);
   str.addAttribute(TextAttribute.CHAR_REPLACEMENT, imageAttribute,
    text.length() - 1, text.length());
   g2d.drawString(str.getIterator(), 40, 80);
  }

  public Image createImage()
  {
   java.net.URL imgURL =
    Drawing.class.getResource("/attributedtext/images/Bean.png");
   ImageIcon icon = new ImageIcon(new ImageIcon(imgURL, "").
    getImage().getScaledInstance(48, 48, Image.SCALE_SMOOTH), "");
   return icon.getImage();
  }
}
```

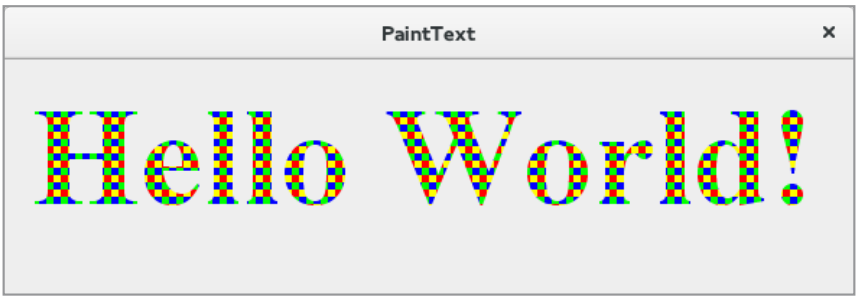
The method *createImage()* is a method that loads the file *Bean.png* from the jar file and the result is an icon of 48×48 pixels. Then there is *paintComponent()*, which this time is complex. First the method defines two fonts and a string. You must note that the string ends with two commas. It is not essential that it is a comma, but the two characters are replaced subsequently to something else. As a next step is defined an *AttributedString* for that string, and it is an object which allows for manipulating a string at the wide variety of ways. First is attached a serif font for the text, and the text will then generally be drawn on the basis of this font, but then is assigned a mono font, but only for the characters from position 6 and up to and including position 13, that is the word *Fredegod*. Also the method defines that this substring should be red and underlined. Next is defined a *Shape* as a *GeneralPath* which is a figure of two triangles. This figure is represented as a *ShapeGraphicAttribute* which indicate its location in the text (here at the baseline) and that it should be drawn as a filled figure. It is now inserted in the text at position 14, to replace the first comma (the comma after the word *Fredegod*). As a next step is defined a rectangle and it is inserted in the same way on the penultimate place. The rectangle does not fill anything, and it should only show how to add space in a text. Finnaly the icon is inserted in the last place. It happens in the same way, just is this time used another object that has the type *ImageGraphicAttribute*. The example shows how to build a very complex string, and there are many other options than what is shown above.

EXERCISE 12

Text is a graphical object in the same way as all other shapes, and text can thus be transformed in the same way as other *Shape* objects. Write a program that will open the following window:

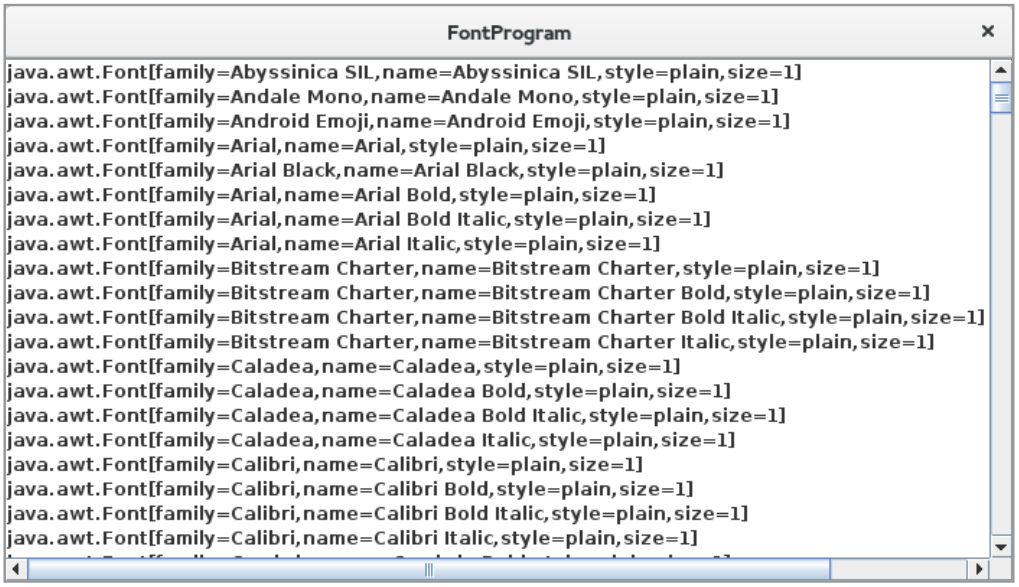
EXERCISE 13

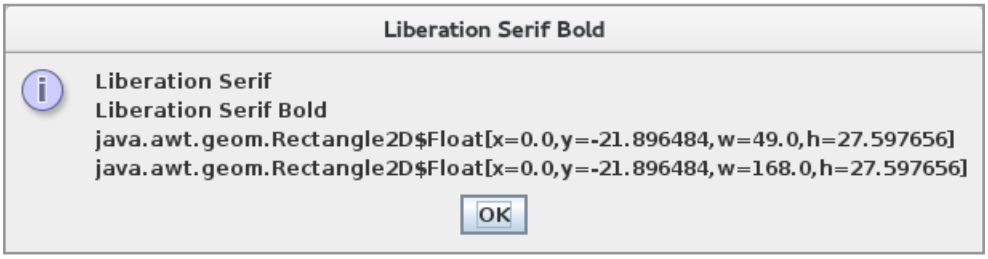
Write a program that displays a window, as shown below, where the text is drawn with a *TexturePaint* and where is drawn with images consisting of 4 small squares:



5.1 FONTS

Java2D contains several classes concerning fonts, and I have already mentioned the classes *Font* and *FontMetrics*. In this section I will show a program that opens a window, as shown below. The application displays a list box with a list of all the fonts that are available. If you double click on a line, you get a message box with information about the font.





The code is as follows, and there is not much to explain, but you should specifically note how to determine which fonts are available. Also note the class *FontRenderContext*, which may be useful in order to measure how much a string fills when drawing with a particular font:

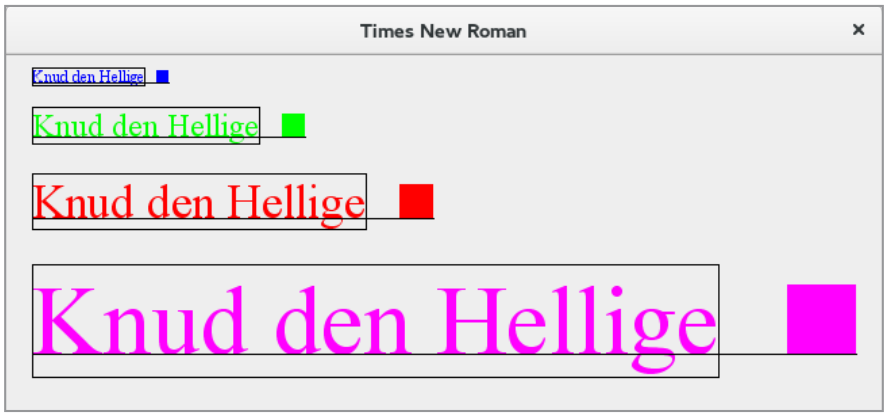
```
package fontprogram;

import java.awt.*;
import java.awt.font.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

public class MainWindow extends JFrame
{
 private DefaultListModel model = new DefaultListModel();

 public MainWindow()
 {
  super("FontProgram");
  setSize(700, 400);
  setLocationRelativeTo(null);
  createWindow();
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createWindow()
 {
  Font[] fonts = GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts();
  for (Font font : fonts) model.addElement(font);
  JList list = new JList(model);
  add(new JScrollPane(list));
  list.addMouseListener(new MouseHandler());
 }

 class MouseHandler extends MouseAdapter
 {
  public void mouseClicked(MouseEvent e)
  {
   if (e.getClickCount() == 2)
   {
    try
    {
     JList list = (JList)e.getSource();
     int n = list.locationToIndex(e.getPoint());
     Font font = (Font)model.get(n);
```

```
      StringBuilder builder = new StringBuilder(font.getFamily());
      builder.append("\n");
      builder.append(font.getName());
      Graphics2D g2d = (Graphics2D)list.getGraphics();
      g2d.setFont(new Font(font.getFamily(), font.getStyle(), 24));
      FontRenderContext frc = g2d.getFontRenderContext();
      Rectangle2D rect1 = g2d.getFont().getStringBounds("ABC", frc);
      Rectangle2D rect2 =
       g2d.getFont().getStringBounds("Knud den Hellige", frc);
      builder.append("\n");
      builder.append(rect1);
      builder.append("\n");
      builder.append(rect2);
      JOptionPane.showMessageDialog(MainWindow.this, builder.toString(),
       font.getFontName(), JOptionPane.INFORMATION_MESSAGE);
     }
     catch (Exception ex)
     {
     }
    }
   }
  }
}
```

EXERCISE 14

Write a program that opens the window below when

1. the filled squares side length is the width of a capital A in the current font
2. the line coincides with the font's baseline
3. the squares "is" on the baseline for the font
4. the rectangles are text circumscribing rectangles
5. the distance between the rectangle and the square is the width of a large A



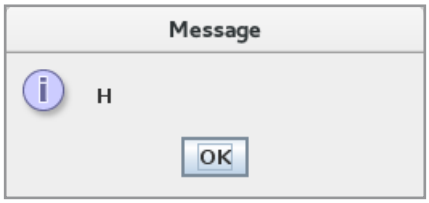
As font sizes are used 12, 24, 36 and 72 points.

5.2 TEXTLAYOUT

I will conclude this chapter about text with the class *TextLayout*, which allows you completely to control how text should be rendered. In practice, a program may display text using a *JTextField*, a *JTextArea* or a *JEditorPane*, but if you have very specific needs, where these components are not adequate, providing a *TextLayout* additional flexibility. In the following I will with four examples outline some of the options. A *TextLayout* is used as an alternative to *drawString()*, and you can think of an object of the class as an encapsulation of a string with expanded opportunities to manipulate the string.

The program *TextProgram* opens the following window:

Here is the first text (the black text) drawn with a *TextLayout*, while the other one is a *Shape* created from the black text, and thus a shape, which is composed of all of the text's glyphs. If you click somewhere in the top text (for example on the big H), you get a message box, as shown below, which shows the character that is clicked on:



The component's codes is as follows:

```
class Drawing extends JComponent
{
 private String str = "Knud den Hellige";
 private Font font = new Font("Liberation Serif", Font.BOLD, 36);
 private TextLayout text;
 private int xpos = 20;
 private int ypos = 50;

 public Drawing()
 {
  FontRenderContext context =
   new FontRenderContext(new AffineTransform(), false, false);
  text = new TextLayout(str, font, context);
  addMouseListener(new HitTester());
 }

 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  text.draw(g2d, xpos, ypos);
  Shape shape = text.getOutline(AffineTransform.getTranslateInstance(20, 100));
  g2d.setPaint(Color.yellow);
  g2d.fill(shape);
  g2d.setPaint(Color.red);
  g2d.draw(shape);
 }
```

```
class HitTester extends MouseAdapter
{
 public void mouseClicked(MouseEvent e)
 {
  Rectangle2D rect = text.getBounds();
  if (rect.contains(e.getX() - xpos, e.getY() - ypos))
  {
   TextHitInfo hit = text.hitTestChar(e.getX() - xpos, e.getY() - ypos);
   JOptionPane.showMessageDialog(
     Drawing.this, str.charAt(hit.getCharIndex()));
  }
 }
}
}
```

The class defines an instance variable of type text *TextLayout*. It is created in the constructor as an encapsulation of the string *str*. The constructor for a *TextLayout* object requires a *Font* and a *FontRenderContext*. You should note, how to create a *FontRenderContext*. The first parameter is an *AffineTransform* that here is just the identity, while the last specifies whether to use antialiasing and how sizes are to be calculated. Finally, assign the constructor a listener for mouse clicks, as I'll explain in a moment.

If you has a *TextLayout*, one can draw the text in the following manner:

```
text.draw(g2d, xpos, ypos)
```

The syntax is a little different than how you else draw a shape or a text. After the figure is drawn you can with the method *getOutline()* get a *Shape* that consists of text's glyphs. The method has a transformation which tells how objects are transformed, and the *Shape* object can then be treated in the same way as any other *Shape* objects. In this case, the figure is filled with a yellow color and drawn with a red.

The component is as specified in the constructor listening for mouse clicks, and the handler test if the mouse coordinates fall within the *TextLayout* object. Is that the case is referenced the object's *TextHitInfo* object that returns some important information where the most important is to convert the mouse's coordinates to the character that is clicked on. It is not trivial to determine it, and it is one of the services as a *TextLayout* provides.

The next example will open the following window, which again shows a text drawn with a *TextLayout*:

The program will show how to draw a caret (or cursor/marker), as well as how to move the cursor using the arrow keys. It sounds simple, but if you thinks about it, the cursor is (usually) a thin line that is drawn between two letters, and it is not simple to calculate how this string should be drawn. Fortunately the class *TextLayout* make the necessary available. The component's code is shown below:

```
class Drawing extends JComponent
{
 private String str = "Knud den Hellige";
 private Font font = new Font("Liberation Serif", Font.BOLD, 36);
 private TextLayout text;
 private int xpos = 20;
 private int ypos = 50;
 private TextHitInfo hit;
```

```
 public Drawing(MainWindow main)
 {
  FontRenderContext context =
   new FontRenderContext(new AffineTransform(), false, false);
  text = new TextLayout(str, font, context);
  hit = text.getNextLeftHit(1);
  main.addKeyListener(new KeyHandler());
 }

 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(
   RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
  g2d.setRenderingHint(RenderingHints.KEY_FRACTIONALMETRICS,
   RenderingHints.VALUE_FRACTIONALMETRICS_ON);
  text.draw(g2d, xpos, ypos);
  Shape[] carets = text.getCaretShapes(hit.getInsertionIndex());
  if (carets[0] != null)
  {
   Shape shape = AffineTransform.getTranslateInstance(xpos,
    ypos).createTransformedShape(carets[0]);
   g2d.setStroke(new BasicStroke());
   g2d.draw(shape);
  }
 }

 class KeyHandler extends KeyAdapter
 {
  public void keyPressed(KeyEvent e)
  {
   if (e.getKeyCode() == KeyEvent.VK_RIGHT)
   {
    hit = text.getNextRightHit(hit.getInsertionIndex());
    if (hit == null) hit = text.getNextLeftHit(1);
    repaint();
   }
   else if (e.getKeyCode() == KeyEvent.VK_LEFT)
   {
    hit = text.getNextLeftHit(hit.getInsertionIndex());
    if (hit == null) hit = text.getNextRightHit(text.getCharacterCount() - 1);
    repaint();
   }
  }
 }
}
```

The code is similar to the previous example, but the constructor has this time a reference to the main window, then to associate an event handler for the keyboard to the window. The event handler is defined by the class *KeyHandler*, which is an inner class in the class *Drawing*. The class has this time an instance variable of the type *TextHitInfo*. It will show where the cursor must be drawn, and is initialized in the constructor, and then the next hit is drawn to the left of the character with index 1 – and thus the first character.

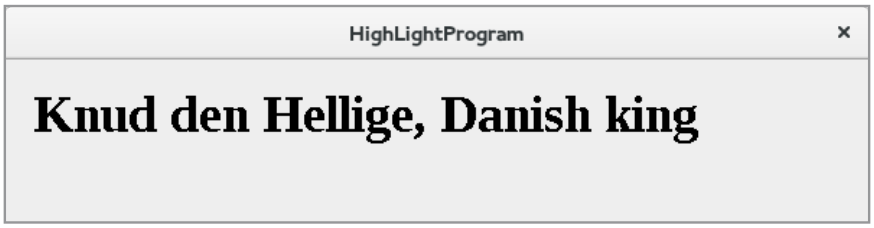
Then there is *paintComponent()*, that this time defines an extra rendering hint. It indicates how the determination of a character's limits should happen and with what accuracy. Once the text is drawn, one determines the position of the cursor:

```
Shape[] carets = text.getCaretShapes(hit.getInsertionIndex());
```

A *TextLayout* are not born with a caret, but it has a logical position that is attached to the *hit* object and which can be referred to the method *getInsertionIndex()*. The method *getCaretShapes()* returns a shape to a caret. In fact, it returns two, and the reason is that Java2D supports *bidirectional* text, and thus where the text is written from right to left. Therefore returns above method two *Shape* objects. With a *Shape* object to a caret available it must be drawn as any other figure after a transformation. Here you need to specifically note the method *createTransformedShape()* that transforms a figure without transforming the current *Graphics2D* object.

Then there is the event handler for the keyboard, which treat events concerning left and right arrows. When looking at the handler is easy enough to understand, and it works by using the two methods *getNextRightHit()* and *getNextLeftHit()*.

The next example is similar to the above and opens the following window:



The program will show how to select text using the mouse, for example.

If you select some text and release the mouse, you get a message box with the text that is selected.

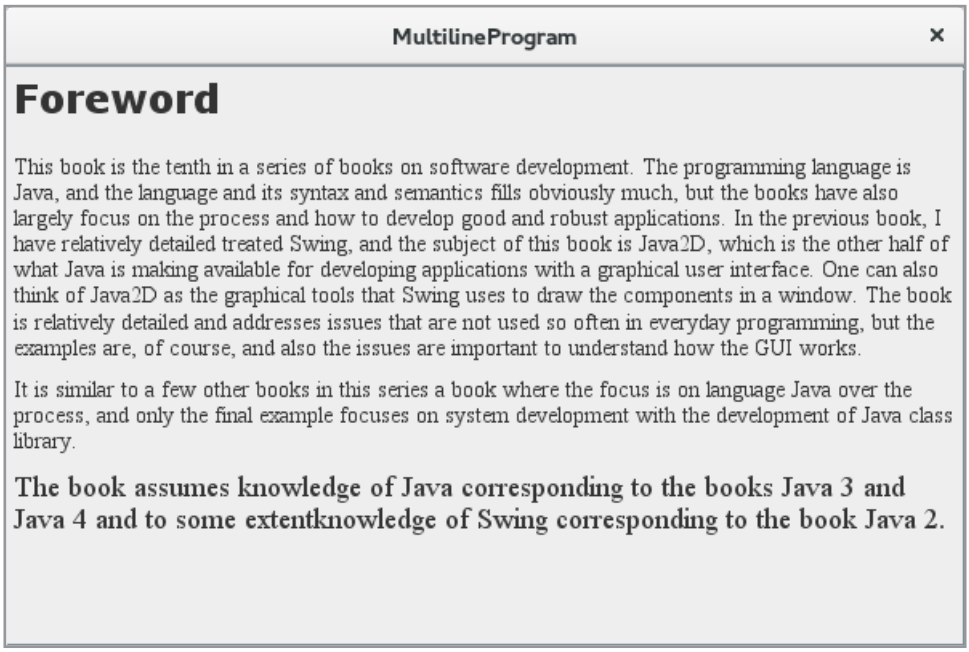
```
class Drawing extends JComponent
{
 private String str = "Knud den Hellige, Danisk king";
 private Font font = new Font("Liberation Serif", Font.BOLD, 36);
 private TextLayout text;
 private int xpos = 20;
 private int ypos = 50;
 private TextHitInfo hit1;
 private TextHitInfo hit2;
 private Rectangle2D rect;

 public Drawing()
 {
  FontRenderContext context =
   new FontRenderContext(new AffineTransform(), false, false);
  text = new TextLayout(str, font, context);
  rect = text.getBounds();
  addMouseListener(new HitTester());
  addMouseMotionListener(new MoveTester());
 }
```

```
public void paintComponent(Graphics g)
{
 Graphics2D g2d = (Graphics2D)g;
 g2d.setRenderingHint(
  RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
 g2d.setRenderingHint(RenderingHints.KEY_FRACTIONALMETRICS,
  RenderingHints.VALUE_FRACTIONALMETRICS_ON);
 if (hit1 != null && hit2 != null)
 {
  Shape base = text.getLogicalHighlightShape(hit1.getInsertionIndex(),
   hit2.getInsertionIndex());
  Shape rect = AffineTransform.
   getTranslateInstance(xpos, ypos).createTransformedShape(base);
  g2d.setPaint(Color.orange);
  g2d.fill(rect);
 }
 g2d.setPaint(Color.black);
 text.draw(g2d, xpos, ypos);
}

class HitTester extends MouseAdapter
{
 public void mousePressed(MouseEvent e)
 {
  hit1 = text.hitTestChar(e.getX() - xpos, e.getY() - ypos);
  hit2 = null;
  repaint();
 }

 public void mouseReleased(MouseEvent e)
 {
  if (hit1 != null && hit2 != null)
  {
   int n1 = hit1.getInsertionIndex();
   int n2 = hit2.getInsertionIndex();
   if (n1 > n2)
   {
    int n = n2;
    n2 = n1;
    n1 = n;
   }
   JOptionPane.showMessageDialog(Drawing.this, str.substring(n1, n2));
   hit2 = null;
  }
 }
}
```

```
 class MoveTester extends MouseMotionAdapter
 {
  public void mouseDragged(MouseEvent e)
  {
   if (rect.contains(e.getX() - xpos, e.getY() - ypos))
   {
    hit2 = text.hitTestChar(e.getX() - xpos, e.getY() - ypos);
    repaint();
   }
  }
 }
}
```

The code is basically simple enough and primarily consists of using two *TextHitInfo* objects to determine which part of the text that is selected and fill a corresponding rectangle. The most difficult thing is actually to control the logic in the event handlers in terms of when a new selection starts and when the check mark has to disappear again.

A *TextLayout* represents basically a single line of text, and if the text has to fill several lines you have to do better, but fortunately there are helper classes that can be used. Consider the following window. The only thing the program does is to display a text in a window (and of course it would be trivial to do with a JTextArea), but in this case the text is drawn with a *TextLayout*.




```
package multilineprogram;

import java.awt.*;
import java.awt.font.*;
import javax.swing.*;
import java.text.*;

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  super("MultilineProgram");
  setSize(600, 400);
  setLocationRelativeTo(null);
  add(new JScrollPane(new Drawing()));
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
}
```

```
class Drawing extends JComponent
{
 private String[] str = {
  "Foreword",
  "This book is the tenth in a series of books on software development. The " +
  " programming language is Java, and the language and its syntax and " +
  "semantics fills obviously much, but the books have also largely " +
  "focus on the process and how to develop good and robust applications. " +
  "In the previous book, I have relatively detailed treated Swing, and " +
  "the subject of this book is Java2D, which is the other half of what " +
  "Java is making available for developing applications with a graphical " +
  "user interface. One can also think of Java2D as the graphical tools that " +
  "Swing uses to draw the components in a window. The book is relatively " +
  "detailed and addresses issues that are not used so often in " +
  "everyday programming, but the examples are, of course, and also the issues " +
  "are important to understand how the GUI works.",
  "It is similar to a few other books in this series a book where the focus " +
  "is on language Java over the process, and only the final example " +
  "focuses on system development with the development of Java class library.",
  "The book assumes knowledge of Java corresponding to the books Java 3 and " +
  "Java 4 and to some extent knowledge of Swing corresponding to the book " +
  "Java 2."
 };
 private Font font1 = new Font("Verdana", Font.BOLD, 24);
 private Font font2 = new Font("Times New Roman", Font.PLAIN, 14);
 private Font font3 = new Font("Times New Roman", Font.PLAIN, 18);

 public void paintComponent(Graphics g)
 {
  float height = 0;
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
  height =
   draw(g2d, new Insets(5, 5, 20, 5), 0, new AttributedString(str[0]), font1);
  height = draw(g2d, new Insets(0, 5, 10, 5), height,
   new AttributedString(str[1]), font2);
  height = draw(g2d, new Insets(0, 5, 10, 5), height,
   new AttributedString(str[2]), font2);
  height = draw(g2d, new Insets(0, 5, 5, 5), height,
   new AttributedString(str[3]), font3);
  setPreferredSize(new Dimension(0, (int)height));
 }

 private float draw(Graphics2D g, Insets margin, float y, AttributedString str,
```

```
  Font font)
 {
  str.addAttribute(TextAttribute.FONT, font);
  AttributedCharacterIterator itr = str.getIterator();
  FontRenderContext context = g.getFontRenderContext();
  LineBreakMeasurer lbm = new LineBreakMeasurer(itr, context);
  float width = getSize().width - margin.left - margin.right;
  float x = margin.left;
  y += margin.top;
  while (lbm.getPosition() < itr.getEndIndex())
  {
   TextLayout textLayout = lbm.nextLayout(width);
   y += textLayout.getAscent();
   textLayout.draw(g, x, y);
   y += textLayout.getDescent() + textLayout.getLeading();
   x = margin.left;
  }
  y += margin.bottom;
  return y;
 }
}
```

The text to be drawn is defined as three strings in an array, and you can think of each string as a paragraph. The most important thing is the method *draw()* with the graphic object to be drawn on. The method has as parameter that is a paragraph in the form of an *AttributedString*, the font the text is to be drawn with, the y-position for where to draw, and an *Insets* object that indicates how much air there should be outside of the text. The primary task of the method is to divide the text into lines, and for that purpose the class *LineBreakMeasure* is used. In addition to this, the method should control the y position for each line.

The component is in the main window's constructor placed in a JScrollPane. Therefore, *paintComponent()* must define the component's size.

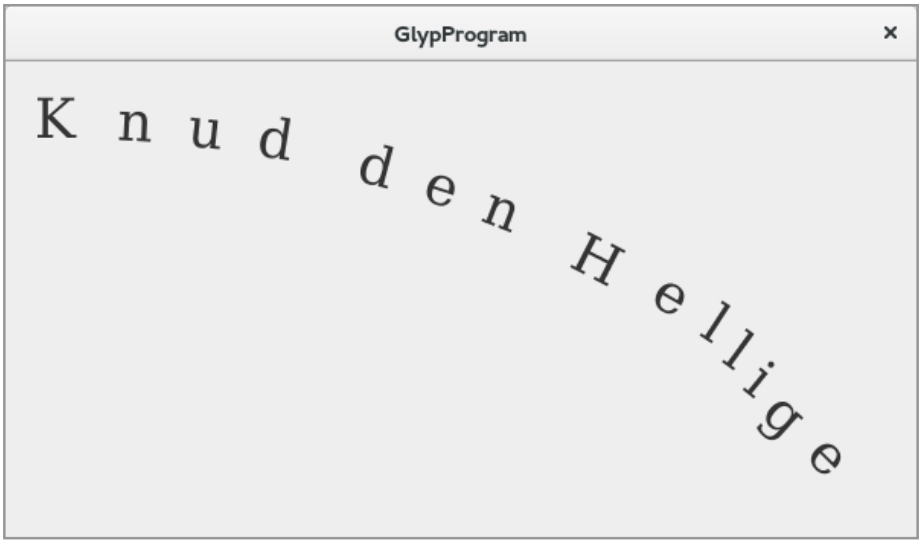
The principle of this example can be used if you need to write a custom component that supports text wrapping.

5.3 GLYPHS

In the previous section I have suggested the possibilities of a *TextLayout*, and although it is rare that you need the class's possibilities, there are examples, for example if you want to develop your own components. It is relatively complex to use a *TextLayout*, but you can actually go even further down and directly manipulate the individual glyps. Rarely – if ever – you will need it, but the following program should show that it is possible. The program opens the window shown on the next page. If you have a string, then the class *Font* has a method that returns a so-called *GlyphVector*, which is a collection consisting of the *Glyph* objects defined by that string. These objects, each of which are a *Shape* object, can then be manipulated in the same way as any other shape.

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
  String str = "Knud den Hellige";
  Font font = new Font("Serif", Font.PLAIN, 36);
  FontRenderContext context = g2d.getFontRenderContext();
  g2d.translate(20, 50);
  GlyphVector glyps = font.createGlyphVector(context, str);
  int length = glyps.getNumGlyphs();
  for (int i = 0; i < length; ++i)
  {
```

```
    Point2D p = glyps.getGlyphPosition(i);
    AffineTransform trans =
     AffineTransform.getTranslateInstance(p.getX(), p.getY());
    trans.rotate(i / (double)(length - 1) * Math.PI / 4);
    g2d.fill(trans.createTransformedShape(glyps.getGlyphOutline(i)));
   }
  }
}
```

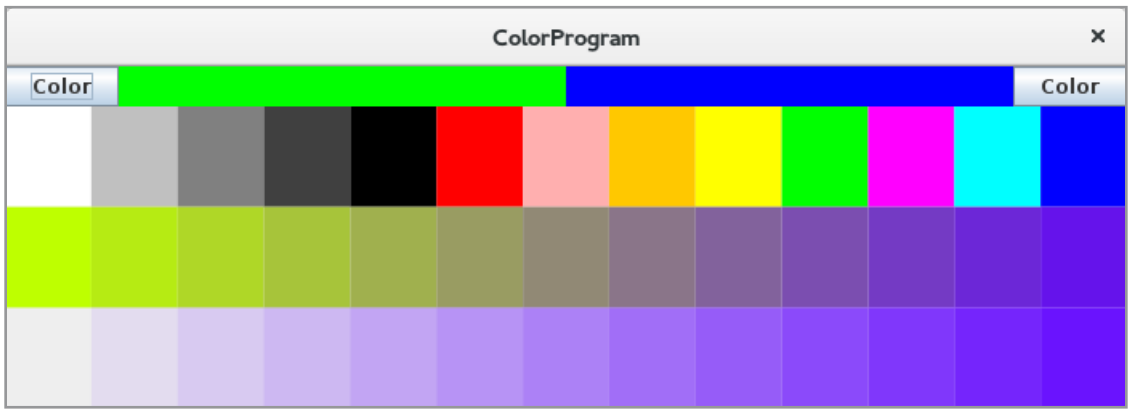


6 COLORS

Colors are treated several places in this series of books, and as a programmer, you usually do not need to know more than what has already been said. Colors are formed by combining the values of three colors red, green and blue, why wee are talking about RGB color encoding. In addition, an alpha value is associated, which tells the extent to which the color is transparent. Each of the four values is represented by a byte (and thus can have 256 different values), and internally a color is represented by a 32 bit int:

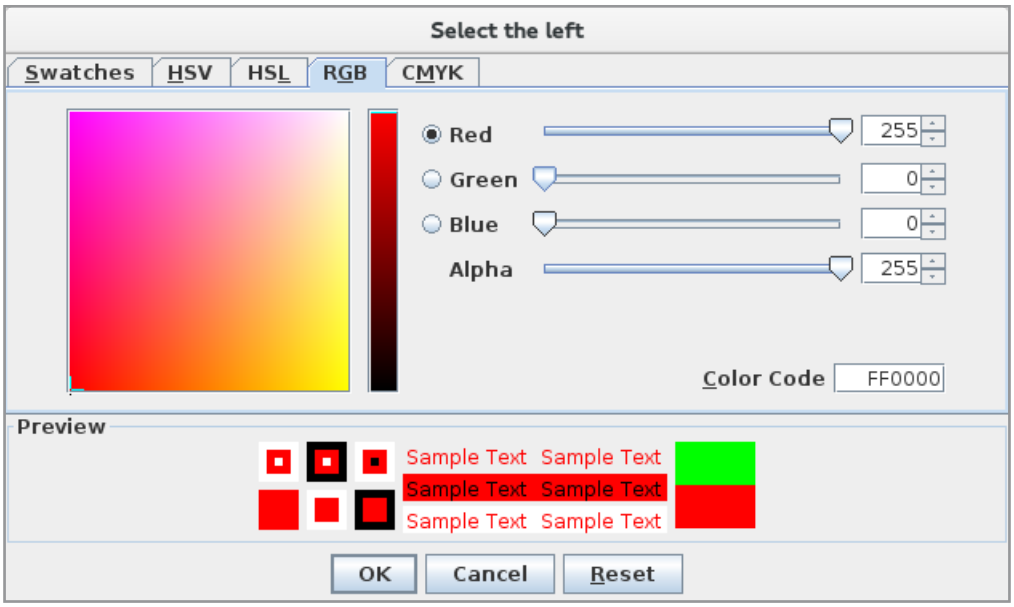
The basic class to represents a color is *Color*, and it has been used many times before. Looking at all the previous examples, the use of colors is generally simple, but the reality is that colors as the same as text are extremely complex, and therefore this section as a brief introduction to the problems regarding colors. In short, the problems can be explained by the fact that certain colors can look different (and often do) on different screens, and even worse, if printed on a color printer. The reasons are that the computer's color representation are relative (what does it actually mean a color is 40% red?) and that different hardware devices (screen and printer) do not display the same color in the same way. I will explain a little bit below, but first an example.

The *ColorProgram* application opens the following window:



showing 39 colored rectangles. The top row shows the color values that are defined as constants in the *Color* class, and above the window shows a green and a blue rectangle and the middle row of rectangles shows how these two colors can be mixed so that the rectangle on the left is green and as you move to the right, the green color is mixed with the blue color and finally the last is blue. The bottom row of the rectangles shows (from the right) the blue color more and more transparent to the far left to be completely transparent and thus not visible.

At the top there are two buttons, and they are used to open a standard color dialog selection box (see below), so you can change the two colors that the program manipulates – that is the colors of the two rectangles at the top of the window. This way you can experiment and see what happens when colors that are mixed and how a color's alpha value changes. The dialog box is relatively complex with many options, but you can create a *ColorChooser* object, which is a component that can be configured to show the wanted options.



The entire code is also shown below, and it does not contain so much new:

```
package colorprogram;

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
```

```
public class MainWindow extends JFrame
{
 private Color leftColor = Color.green;
 private Color rightColor = Color.blue;
 private JLabel lblLeft = new JLabel();
 private JLabel lblRight = new JLabel();
 private Drawing drawing;

 public MainWindow()
 {
  super("ColorProgram");
  setSize(700, 250);
  setLocationRelativeTo(null);
  add(createTop(), BorderLayout.NORTH);
  add(drawing = new Drawing());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private JPanel createTop()
 {
  JPanel panel = new JPanel(new GridLayout(1, 2));
  JPanel left = new JPanel(new BorderLayout());
  JPanel right = new JPanel(new BorderLayout());
  JButton cmdLeft = new JButton("Color");
  JButton cmdRight = new JButton("Color");
  cmdLeft.addActionListener(this::updateLeft);
  cmdRight.addActionListener(this::updateRight);
  lblLeft.setOpaque(true);
  lblLeft.setBackground(leftColor);
  lblRight.setOpaque(true);
  lblRight.setBackground(rightColor);
  left.add(cmdLeft, BorderLayout.WEST);
  left.add(lblLeft);
  right.add(cmdRight, BorderLayout.EAST);
  right.add(lblRight);
  panel.add(left);
  panel.add(right);
  return panel;
 }

 class Drawing extends JComponent
 {
  private Color[] colors = { Color.white, Color.lightGray, Color.gray,
    Color.darkGray, Color.black, Color.red, Color.pink, Color.orange,
    Color.yellow, Color.green, Color.magenta, Color.cyan, Color.blue };
```

```
public void paintComponent(Graphics gr)
{
 Dimension d = getSize();
 Graphics2D g2d = (Graphics2D)gr;
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
 float t = colors.length;
 float w = d.width / t;
 float h = d.height / 3.0F;
 for (int i = 0; i < t; ++i)
 {
  g2d.setPaint(colors[i]);
  g2d.fill(new Rectangle2D.Float(i * w, 0, w, h));
 }
 for (int i = 0; i < t; ++i)
 {
  float u = i / t;
  int r = (int)(rightColor.getRed() * u + leftColor.getRed() * (1 - u));
  int g = (int)(rightColor.getGreen() * u + leftColor.getGreen() * (1 - u));
  int b = (int)(rightColor.getBlue() * u + leftColor.getBlue() * (1 - u));
  g2d.setPaint(new Color(r, g, b));
  g2d.fill(new Rectangle2D.Float(i * w, h, w, h));
 }
```

```
   for (int i = 0; i < t; ++i)
   {
    int alpha = (int)(255 * i / t);
    g2d.setPaint(new Color(rightColor.getRed(), rightColor.getGreen(),
     rightColor.getBlue(), alpha));
    g2d.fill(new Rectangle2D.Float(i * w, 2 * h, w, h));
   }
  }
 }

 public void updateLeft(ActionEvent e)
 {
  Color color = JColorChooser.showDialog(this, "Select the left", leftColor);
  if (color != null)
  {
   leftColor = color;
   lblLeft.setBackground(color);
   drawing.repaint();
  }
 }

 public void updateRight(ActionEvent e)
 {
  Color color = JColorChooser.showDialog(this, "Select the right color",
   rightColor);
  if (color != null)
  {
   rightColor = color;
   lblRight.setBackground(color);
   drawing.repaint();
  }
 }
}
```

However, you should especially note how to mix two colors and how to create color objects. You should also note how to create a *JColorChooser* object, thus opening the color selection dialog, and you should investigate the documentation in what options are available to customize this dialog. You can choose to disable many of the options and the dialog should be more manageable.

6.1 ABOUT COLORS

In the world of physics, colors are light represented by electromagnetic waves, and the color of the light is determined by the wavelength of light. The human eye is using special cells capable of capturing these wavelengths and transforming these impressions into our brain, in fact, by capturing three intensities of the colors red, green and blue and thus a bit the same as a computer works with RGB colors. However, our eyes are a computer far superior. A screen must convert the three color values to a particular color, which is done by using special features, but these features can not form all colors, so colors on a computer screen or printer will always be approximated to the true color.

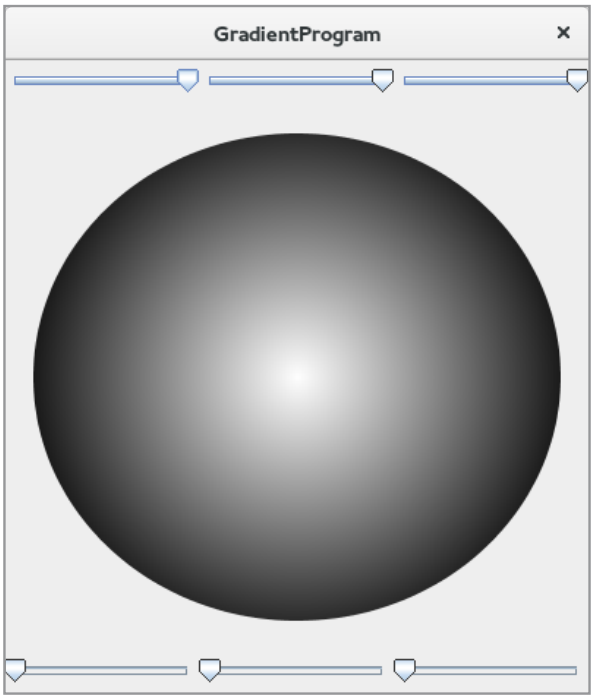
A color room is a family of colors that can be displayed on a particular device. For example has a screen a certain family of colors that it can display. These colors are formed from the intents of the colors red, green and blue, and in practice there are 256 values for each color. Two different screens do not have the same color space due to the variations of the red, green and blue lights and variations of the manufacturer's electronics. Although two screens use the same color encoding in the form of RGB, the colors will still vary, and the screens will each have their color room. It's all complicated by some printers using a different color encoding, combining the colors cyan, magenta, yellow and black. This encoding is called CMYK, and such a printer has a CMYK color room instead. Thus, RGB colors must be converted to CMYK colors to be displayed on a printer, thus saying differently that the color room for the current display should be converted to the color room of another device. The sum of all is that different color rooms show the colors differently.

To solve these problems, the industry has worked to define absolute color rooms that precisely define what is red, what is green and what is blue. One of them is called *sRGB* (for standard RGB), and it is used as the default by Java 2D, which converts colors into this color room.

It helps to solve the problems with colors, but it does not compensate for differences in the hardware and used color profiles. It is a table that converts the colors from a standard color room like *sRGB* to the color room for a particular physical device. Each physical device thus has its own color profile. This applies, for example for a screen, and to make the colors look right, you can calibrate the screen, which means adjusting the color profile so that the conversion from a specific color space to the profile is correct. It's not easy and requires practice and knowledge, but people who work with print do much to calibrate their equipment properly.

For many applications (most), everything with color rooms and color profiles is uninteresting. If a program has to draw a figure, it is seldom important to see if the figure appears with a little different color on another screen, but it is for example if you have to show photos, where it is very important. Here the colors should be right and if you are writing software for imaging, the above is important. Although I do not want to go into the subject further, Java 2D types are available for both color spaces and color profiles, types that have methods for manipulating details for colors.

To finish this section about colors, I will show a program that fills an ellipse with a gradient paint, but unlike what I have previously shown, it must be a gradient paint that blends two colors from the center to the periphery in an ellipse. The program opens a window as shown below:



In the same way as shown in a previous example (Exercise 8), you can change the two colors using *JSlider* components at the top and bottom of the window.

Java 2D does not directly have a *Paint* object for this purpose, and it is therefore necessary to write a class itself. The class should implement the *Paint* interface that defines two methods. The most important is called *createContext()* and will return a *PaintContext* object, which is also an interface where the most important method is called *getRaster()*, and return a *Raster* object to the figure to be drawn. It is the object with all the pixels been modified.

The code for the *Paint* object is shown below and I do not want to describe the code further, as the most important is explained by the comments:

```
package gradientprogram;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;

// Defines a Paint object, which is a gradient paint that blends two colors from
// a center point and along a radius. The color at the center is color1, whereas
// all points whose distance to the center is greater than or equal to the radius
// is color2. All points, whose distance to the center is less than the radius
// has a gradient color, so that all points with the same distance to the center
// have the same color.
```

```
public class RoundGradient implements Paint
{
 private Point2D point; // the center
 private Point2D radius; // the radius
 private Color color1; // the color at the center
 // color of points whose distance to center is greater than radius
 private Color color2;

 public RoundGradient(Point2D point, Color color1, Point2D radius, Color color2)
 {
  this.point = point;
  this.color1 = color1;
  this.radius = radius;
  this.color2 = color2;
 }

 // This method is defined by the interface Paint.
 public PaintContext createContext(ColorModel cm, Rectangle deviceBounds,
  Rectangle2D userBounds, AffineTransform trans, RenderingHints hints)
 {
  return new RoundContext(trans.transform(point, null), color1,
   trans.deltaTransform(radius, null), color2);
 }

 // This method is defined by the interface Paint.
 public int getTransparency()
 {
  int a1 = color1.getAlpha();
  int a2 = color2.getAlpha();
  return (((a1 & a2) == 0xff) ? OPAQUE : TRANSLUCENT);
 }
}

// PaintContext, that defines how the individual points should be colored
class RoundContext implements PaintContext
{
 private Point2D point; // the center
 private Point2D radius; // the radius
 private Color color1; // the color at the center
 // color of points whose distance to center is greater than radius
 private Color color2;
```

```
public RoundContext(Point2D point, Color color1, Point2D radius, Color color2)
{
 this.point = point;
 this.color1 = color1;
 this.radius = radius;
 this.color2 = color2;
}

// This method is defined by the interface PaintContext.
public void dispose()
{
}

// This method is defined by the interface PaintContext.
public ColorModel getColorModel()
{
 return ColorModel.getRGBdefault();
}
```

```
 // This method is defined by the interface PaintContext.
 // The method's parameters define the area to be colored.
 public Raster getRaster(int x, int y, int w, int h)
 {
  // defines a Raster with w*h pixels
  WritableRaster raster = getColorModel().createCompatibleWritableRaster(w, h);

  // array to color values,
  // there are used 4 places to each pixel (red, green, blue, alpha)
  int[] data = new int[w * h * 4];
  double rad = radius.distance(0, 0);

  // loop over all pixels
  for (int j = 0; j < h; ++j)
   for (int i = 0; i < w; i++)
   {
    // the distance between the center and the current point measured to raidus
    // must max be 1
    double r = point.distance(x + i, y + j) / rad;
    if (r > 1.0) r = 1.0;

    // pixel index
    int b = (j * w + i) * 4;

    // define pixel values
    // if r is close to 0, color1 is used
    // If r is close to 1, color2 is used
    // else the two colors are mixed
    data[b] = (int)(color1.getRed() + r * (color2.getRed() - color1.getRed()));
    data[b + 1] =
      (int)(color1.getGreen() + r * (color2.getGreen() - color1.getGreen()));
    data[b + 2] =
      (int)(color1.getBlue() + r * (color2.getBlue() - color1.getBlue()));
    data[b + 3] =
      (int)(color1.getAlpha() + r * (color2.getAlpha() - color1.getAlpha()));
   }

  // copy the pixel values to the Raster
  raster.setPixels(0, 0, w, h, data);
  return raster;
 }
}
```

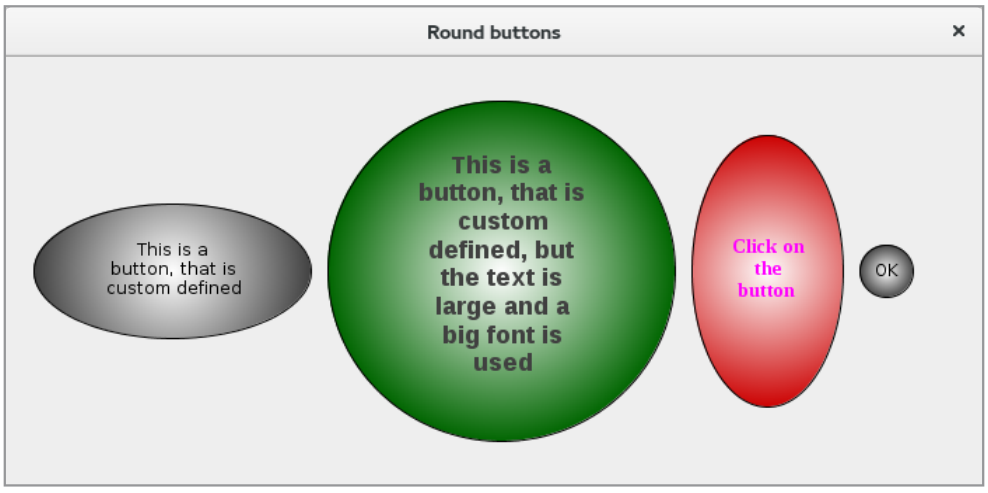
You should note that the class is relatively general and can be used to paint objects in other contexts. Note that the center point does not need to be the center. Also note that you can use the same color for both values, but with different alpha values.

Below is the component that draws the figure:

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Dimension d = getSize();
  Color color1 = new Color(sldFgR.getValue(), sldFgG.getValue(),
   sldFgB.getValue());
  Color color2 = new Color(sldBgR.getValue(), sldBgG.getValue(),
   sldBgB.getValue());
  Graphics2D g2d = (Graphics2D)g;
  g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
   RenderingHints.VALUE_ANTIALIAS_ON);
  Ellipse2D ellip = new Ellipse2D.Double(20, 20, d.width - 40, d.height - 40);
  Point2D rad = d.width < d.height ? new Point2D.Double(20, d.height / 2) :
   new Point2D.Double(d.width / 2, 20);
  g2d.setPaint(new RoundGradient(new Point2D.Double(d.width / 2, d.height / 2),
   color1, rad, color2));
  g2d.fill(ellip);
 }
}
```

PROBLEM 1

You must write a program that creates a round button. For example, a window with four buttons is shown below:



The buttons should be defined as custom components and they must support text wrapping. There must also be an effect when you click the button, and finally, the component will naturally fire an action event when clicked on it.

7 IMAGES

An image is a 2-dimensional array of colors, each element being called a pixel. In principle, it looks like any other figure, but there are two important differences

1. pixels do not necessarily respond directly to pixels on the drawing
2. an image has a width and height measured in pixels and a coordinate system that is independent of the drawing area

The basic class is *Image*, but there is as follows some others. The first step is to show how to load an image so that you can display it in a window. I have actually already shown how, for example by loading an icon or pictures to playing cards, etc. As an example, the following method reads an image from a file where the parameter is the file's name (full pathname):

```
public static ImageIcon loadImageIcon(String filename)
{
 return new ImageIcon(filename);
}
```

The image is represented as an *ImageIcon*, and if you want to refer to the image as an *Image*, the class *ImageIcon* has a method called *getImage()*. Simpler it can hardly be. The following method does the same:

```
public static BufferedImage loadImage(String filename)
{
 try
 {
  return ImageIO.read(new File(filename));
 }
 catch (IOException e)
 {
  return null;
 }
}
```

and the main difference is that the method this time returns the image as a *BufferedImage* such you can manipulate the image directly. Note, in particular, that a *BufferedImage* inherits the class *Image* and therefore can be used anywhere in which you can use an *Image*. Instead, if you want to read the image from the program's jar file (which is often the case with icons and smaller images), you can use the following method:

```
public static ImageIcon createImageIcon(String path)
{
 java.net.URL imgURL = PaGUI.class.getResource(path);
 if (imgURL != null) return new ImageIcon(imgURL, "");
 return null;
}
```

where the parameter should be the image's name relative to the package containing the image. A variation of this method that scales the image to a certain size (and which is especially useful when loading icons) is:

```
public static ImageIcon createImageIcon(String path, int width, int height)
{
 java.net.URL imgURL = PaGUI.class.getResource(path);
 if (imgURL != null) return new ImageIcon(new ImageIcon(imgURL, "").
  getImage().getScaledInstance(width, height, Image.SCALE_SMOOTH), "");
 return null;
}
```

Viewing an image in a window is, in principle, simple and the following class shows an image:

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  if (image != null)
  {
   Graphics2D g2d = (Graphics2D)g;
   AffineTransform trans = new AffineTransform();
   g2d.drawImage(image, trans, this);
  }
  else super.paintComponent(g);
 }
}
```

and there is not much to explain. *Graphics2D* has a *drawImage()* method that shows an image. The object *image* is the image, and *trans* is a transformation that transform the image before it is displayed. In this case, it's just the identity that does not matter and the result is that the image appears unscaled. This means that if the image is larger than the component, only a small portion of the image will appear, and the result may be as shown below (where the left button is clicked).

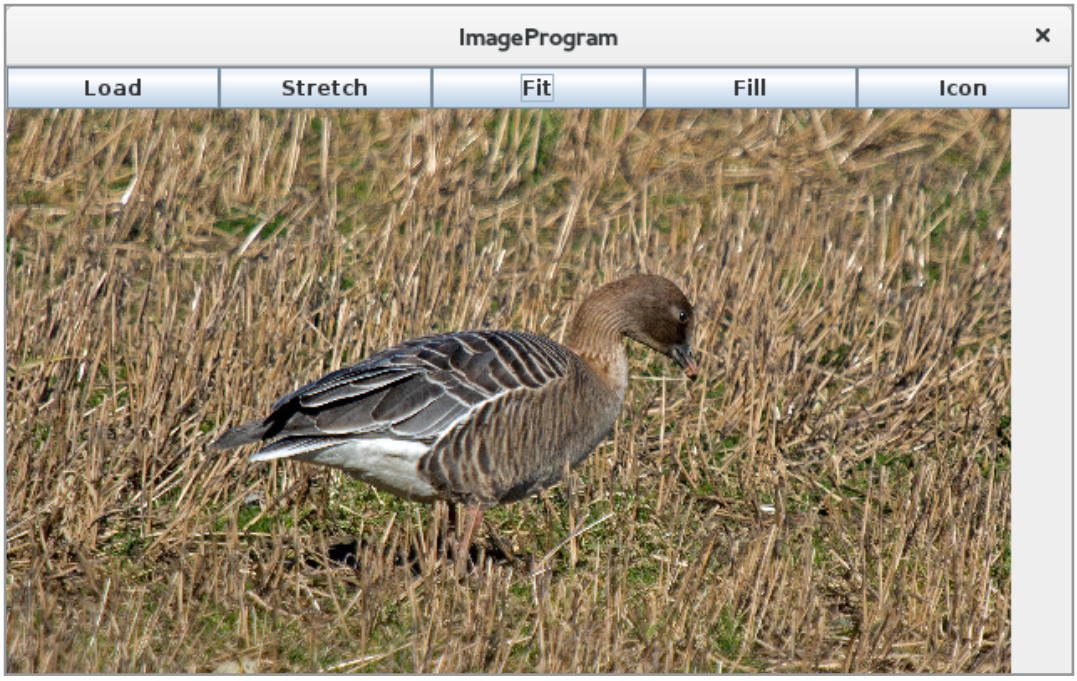
It is a large image (3306×1860 pixels), and you can see only the upper left corner of the image. You can use the *trans* parameter to scale the image and I want to show three ways. Similarly, the program defines the following type:

```
package imageprogram;

public enum ImageScaling { UNSCALED, STRETCHED, SCALEDTOFIT, SCALEDTOFILL }
```

as indicates

1. that the image should appear unscaled
2. that the image should be scaled in both directions so it exactly fills the window
3. that the image should be scaled in both directions, but such that the proportions are retained and the entire image is displayed
4. that the image should be scaled in both directions, but such that the proportions are retained and the entire window is filled

The problem with *STRETCHED* is that the image may be *deformed*, that is a big problem with photos. Below is the window where *SCALEDTOFIT* is selected:



Here is the image scaled, so all may be in the window, and then the whole picture is displayed. The result is that a part of the window may not be used. The method *paintComponent()* has been modified:

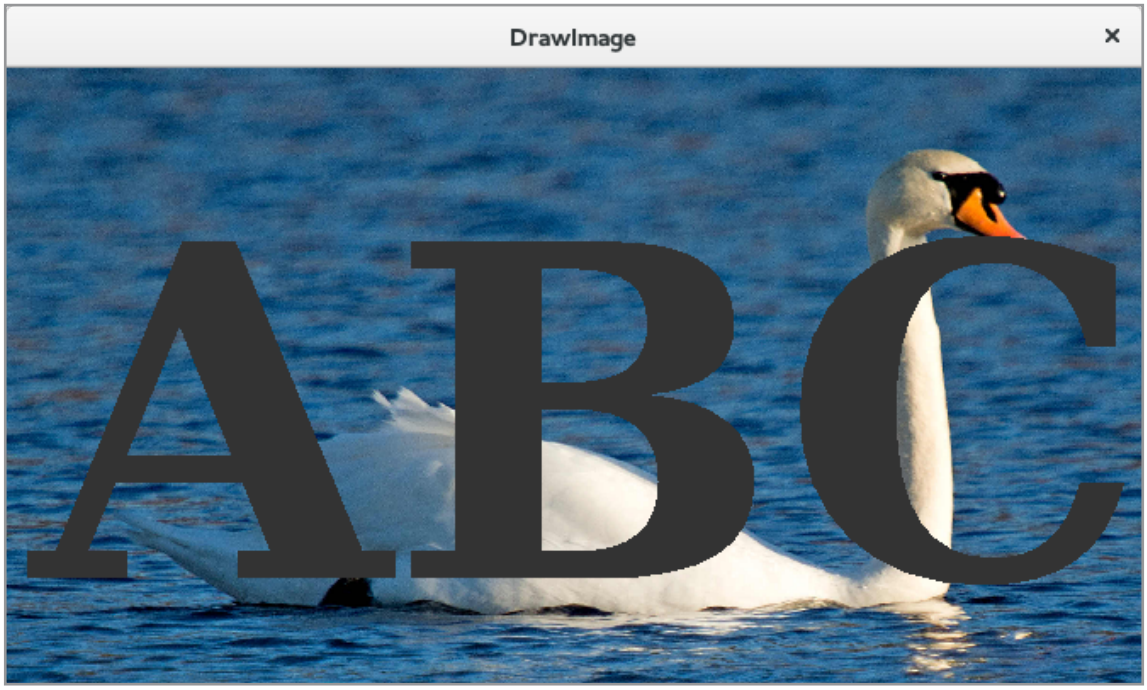
```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  if (image != null)
  {
   Dimension d = getSize();
   Graphics2D g2d = (Graphics2D)g;
   AffineTransform trans = new AffineTransform();
   PaGUI.scale(trans, image, d, action);
   g2d.drawImage(image, trans, this);
  }
  else super.paintComponent(g);
 }
}
```

and the important is the method *PaGUI.scale()* which changes the object *trans* corresponding to the value of *action*:

```
public static void scale(AffineTransform trans, Image image, Dimension dim,
 ImageScaling action)
{
 if (action == ImageScaling.UNSCALED) return;
 double scaleX = ((double)dim.width) / image.getWidth(null);
 double scaleY = ((double)dim.height) / image.getHeight(null);
 if (action == ImageScaling.SCALEDTOFIT)
  scaleX = scaleY = Math.min(scaleX, scaleY);
 else if (action == ImageScaling.SCALEDTOFILL)
  scaleX = scaleY = Math.max(scaleX, scaleY);
 trans.setToScale(scaleX, scaleY);
}
```

I do not want to show the rest of the program, which primarily concerns user interface and event handlers.

The method *drawImage()* draws an image that, in principle, can be manipulated in the same way as any other *Shape* object. Consider the following window where a text is written on top of an image:



The component's code is as follows:

```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Image image = PaGUI.createImageIcon("/drawimage/images/svane.jpg").getImage();
  if (image != null)
  {
   Dimension d = getSize();
   Graphics2D g2d = (Graphics2D)g;
   AffineTransform trans = new AffineTransform();
   PaGUI.scale(trans, image, d, ImageScaling.SCALEDTOFILL);
   g2d.drawImage(image, trans, this);
   g2d.setFont(new Font("Serif", Font.BOLD, 272));
   g2d.drawString("ABC", 15, 300);
  }
  else super.paintComponent(g);
 }
}
```

As another example, the following program draws the same image, but this time the clip area is defined using a text. The result is that you only see the water and a little of the bird:


```
class Drawing extends JComponent
{
 public void paintComponent(Graphics g)
 {
  Image image = PaGUI.createImageIcon("/clipimage/images/svane.jpg").getImage();
  if (image != null)
  {
   Dimension d = getSize();
   Graphics2D g2d = (Graphics2D)g;
   AffineTransform trans = new AffineTransform();
   PaGUI.scale(trans, image, d, ImageScaling.SCALEDTOFILL);
   g2d.setClip(getClipping(g2d));
   g2d.drawImage(image, trans, this);
  }
  else super.paintComponent(g);
 }

 private Shape getClipping(Graphics2D g2d)
 {
  Font font = new Font("Serif", Font.BOLD, 272);
  GlyphVector glyphs= font.createGlyphVector(g2d.getFontRenderContext(), "ABC");
  return glyphs.getOutline(15, 300);
 }
}
```

7.1 IMAGING

Image editing applications like GIMP and Photoshop are designed to manipulate digital photos, and Java2D actually contains image processing features, and these features are the subject of the following. Now no new competitor is suddenly developing for the above programs (it's extremely complex programs) and more than what Java2D directly makes available, but the following can, if nothing else, be used to explain how professional programs as Photoshop and GIMP works.

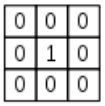
Image processing describes how digital images can be manipulated using a process, commonly called filtering. The idea is that an image called the source is sent through a filter, and the result is a new image, called the destination. Both source and destination are in Java2D represented by a *BufferedImage*. The filter is defined by an interface called *BufferedImageOp*, and the main method that this interface defines is *filter()*. A class that implements this interface is often called for an image operator. Java2D defines five such image operators:

1. *ConvolveOp*, used for bluring, sharpening and marking of edges
2. *AffineTransformOp*, used for geometric transformation
3. *LookupOp*, used for color reduction and color inversion
4. *RescaleOp*, used to make a picture brighter or darker
5. *ColorConvertOp*, used to convert color spaces

You can of course also define your own image operators, which is just a class that implements *BufferedImageOp*.

ConvolveOp

It is a filter that modifies each pixel in the source so the corresponding pixel is changed from the values of neighbor pixels. This is done by means of a so-called *Kernel*, which defines a matrix with the pixels to be used. An example might be the following:



where a pixel has 8 neighbor pixels. Each of the matrix elements has a value and 0 means that the pixel in question does not contribute to the result, whereas 1 means that the pixel is moved unchanged to the destination. The above kernel is thus the identifier and defines a filter that does not affect the destination. If you want to maintain the light conditions, the sum of the matrix elements must be 1. If the sum is less than 1, the image darkens and is the sum is larger than 1, the image becomes brighter. The syntax for creating a *ConvolveOp* object is

```
public ConvolveOp(Kernel kernel, int edgeHint)
```

Here is the last parameter a constant that indicates what should happen with the edges, where all neighbor pixels are not necessarily found. There are two options:

1. *public static final in EDGE_ZERO_FILL*, indicating that pixels on the edge are set to 0 and thus become a black pixel in the destination
2. *public static final int EDGE_NO_OP*, which indicates that pixels on the edge are left unchanged at the destination

These operations are typically used to make an image sharper or to the opposite, as is usually called blur. The result, of course, depends on the values of the matrix, but also of its size, and although in a professional photo editing program it is possible to determine these values, there are a number of standard examples, for example

```
0.11 0.11 0.11
0.11 0.11 0.11
0.11 0.11 0.11
```

that retains the light conditions and results in a blur operation, and as another example, the following can be used to make an image sharper:

```
0 -1 0
-1 5 -1
0 -1 0
```

AffineTransformOp

It is an operation that performs a geometric transformation of an image, but such that the operation works on each of the image's pixels. Depending on what transformation, it may mean that the image is slightly deformed (especially rotations) – a pixel is not necessarily the same in the source and the destination. Two different algorithms are used, which can be specified with a parameter:

1. *Nearest neighbor*, where the color of each pixel in the destination is based on its neighbor pixels in the source. It is the most effective of the two algorithms.
2. *Bilinear interpolation*, where the colors of each pixel are determined by combining the colors of pixels in the source that overlap each other in the destination. This algorithm is less efficient, but provides a better result.

LookupOp

It is an operation that can often be used as an alternative to *ConvolveOp*, which is simpler to define (it is simpler to predict the effect). Here is a so-called lookup table that, in the form of an array, contains the color of a pixel in the destination. The index of this array is the value of the corresponding pixel in the source, and the color of a pixel in the destination is thus determined by a lookup in the table. There is a lookup for each of the three values red, green and blue. Since there are 256 options for each color, the individual arrays must have room for 256 values.

When creating a lookup table, it is only necessary to specify a single array, and the values in the array are then used for all three colors, but you can also specify three arrays.

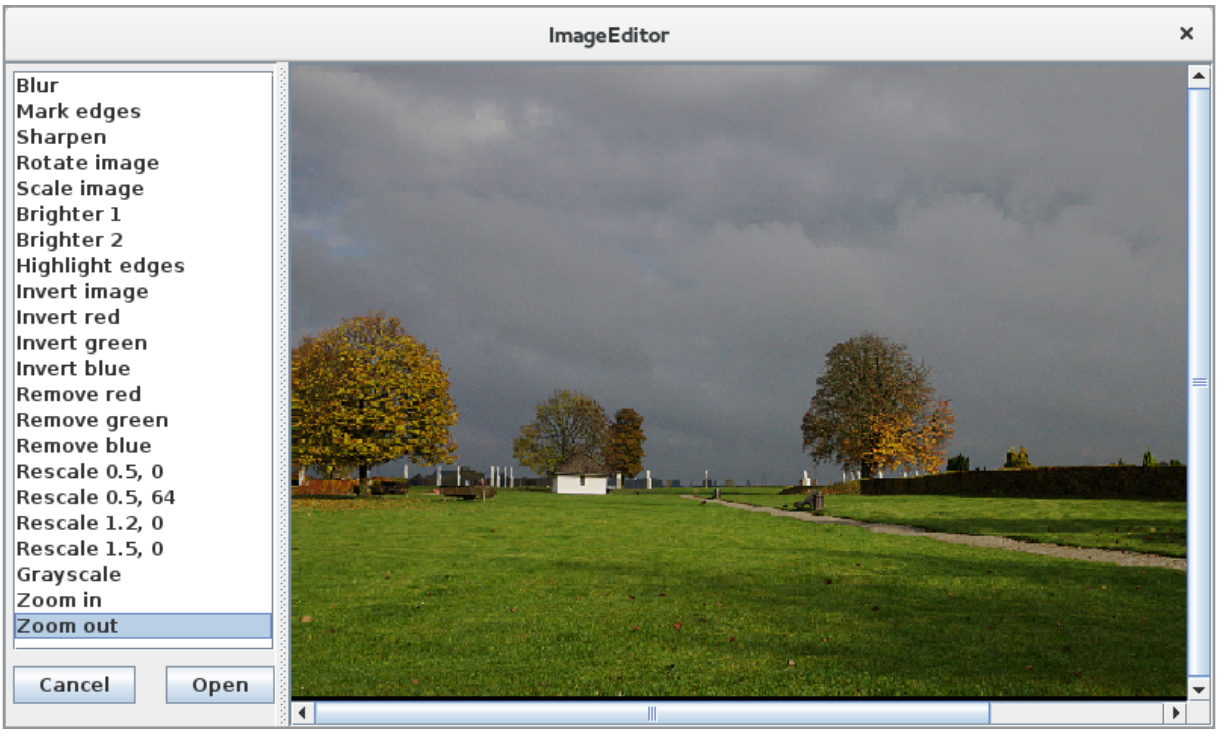
RescaleOp

It is a very simple operation that simply multiplies all color values by a factor. The applications may not be so many, but the operation may, for example, be used to make an image brighter or the opposite. It should be noted that this means that all values are multiplied by the same factor and that some values can then exceed 256 (or 0) so that the colors become white or black. It is also possible to specify an offset, which is a value added to the color value.

ColorConvertOp

The last operation is not demonstrated here, but it is used to convert the colors from one color space to another.

To test some of the above in practice, you can study the program *ImageEditor*. If you open the application, you can open an image, which you can then edit using the operators described above:



On the left side you have all the operations that you can perform while the right side has the image. The image is a component of the following type:

```
class Drawing extends JComponent
{
 private BufferedImage image = null;
```

```
public BufferedImage getImage()
{
 return image;
}

public void setImage(BufferedImage image)
{
 this.image = image;
}

public void paintComponent(Graphics g)
{
 if (image != null)
 {
  Graphics2D g2d = (Graphics2D)g;
  AffineTransform trans = new AffineTransform();
  g2d.drawImage(image, trans, this);
  setPreferredSize(new Dimension(image.getWidth(null), image.getHeight(null)));
 }
 else
 {
```

```
    super.paintComponent(g);
    Dimension d = getSize();
    setPreferredSize(new Dimension(d.width, d.height));
   }
  }
}
```

that represents the image as a *BufferedImage*. Otherwise, the most important of course is the method *paintComponent()*, which draws the image and calculates the component's preferred size.

The left side is a list box, and each element (near two) has the type *ImageOperation*, which is the following class:

```
private class ImageOperation implements Comparable<ImageOperation>
{
 private String name;
 private BufferedImageOp operation;

 public ImageOperation(String name, BufferedImageOp operation)
 {
  this.name = name;
  this.operation = operation;
 }

 public BufferedImageOp getOperation()
 {
  return operation;
 }

 public String toString()
 {
  return name;
 }

 public int compareTo(ImageOperation iop)
 {
  return name.compareTo(iop.toString());
 }
}
```

that is an inner class in *MainWindow*. The class has two variables, where the first is a name which is the name displayed in the list box, while the other is the actual image processing operation in the form of a *BufferedImageOp* object, and thus the operation performed when the user clicks on an object in listbox. The event handler regarding clicking on an item in the list box is as follows:

```
class MouseHandler extends MouseAdapter
{
 public void mouseClicked(MouseEvent e)
 {
  try
  {
   JList list = (JList)e.getSource();
   int n = list.locationToIndex(e.getPoint());
   ImageOperation operation = null;
   if (n == 3) operation = rotate();
   else if (n == 4) operation = scale();
   else operation = (ImageOperation)model.get(n);
   if (operation != null)
   {
    BufferedImageOp opr = operation.getOperation();
    if (opr instanceof LookupOp)
     opr.filter(drawing.getImage(), drawing.getImage());
    else drawing.setImage(opr.filter(drawing.getImage(), null));
    drawing.repaint();
   }
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
 }
}
```

that is also an inner class. The handler decides which item is clicked. If the index is 3 or 4, a standard dialog box opens, and the user must enter a value (the angle of a rotation, or a percent, for how much the image should be scaled). In either case, an *ImageOperation* object is returned for that image processing. If the index is not 3 or 4, a typecast of the item that is clicked is performed to an *ImageOperation* object. With an object available, the *BufferedImageOp* object is determined and the appropriate image processing is performed. Here you should notice two things: How to refer to the image itself and that image processing is performed by the method *filter()*, but in two ways depending on which operation it is.

Back then, there are the individual *BufferedImageOp* objects that all must be created. As an example, below is shown the method that creates the object to make the image sharper and thus a *ConvolveOp* object:

```
private ImageOperation createSharpen()
{
 float[] sharp = { 0f, -1f, 0f, -1f, 5f, -1f, 0f, -1f, 0f };
 return new ImageOperation("Skarpere", new ConvolveOp(new Kernel(3, 3, sharp)));
}
```

The only thing to note is how to create the *Kernel* object. As another example, below is shown how to add a *LookupOp* object to the list box:

```
model.addElement(new ImageOperation("Remove red",
  new LookupOp(new ShortLookupTable(0, new short[][] { zero, one, one }),
   null)));
```

Here is a filter that removes the red color. The two arrays one and zero (for lookup) are arrays with 256 places, where all places are respectively 1 and 0. As another example, the following is an example of a *RescaleOp* filter that scales all color values to half and thus darkens the image:

```
model.addElement(new ImageOperation(
 "Rescale 0.5, 0", new RescaleOp(.5f, 0, null)));
```

Finally, there is shown a method that creates an *AffineTransformOp* object that scales the image to 90 percent of the size:

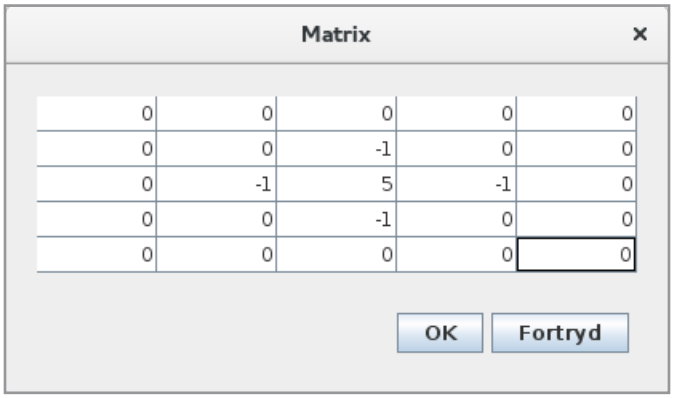
```
private ImageOperation zoomOut()
{
 AffineTransform trans = AffineTransform.getScaleInstance(0.9, 0.9);
 return new ImageOperation("Zoom out", new AffineTransformOp(trans, null));
}
```

You are encouraged to test the program and investigate the effect of the different operations. It's far from a new GIMP program, but it can explain a bit about how a program like GIMP works.

EXERCISE 15

Create a copy of the projet *ImageEditor*. The three top image operations (blur, sharpen and highlight edges) are all of the type *ConvolveOp* and a filter using a 3×3 matrix. Add another operation (for example below the *Scale image* operation), which is also a *ConvolveOp*, but an operation using a 5×5 matrix that you can enter in a dialog box shown below.

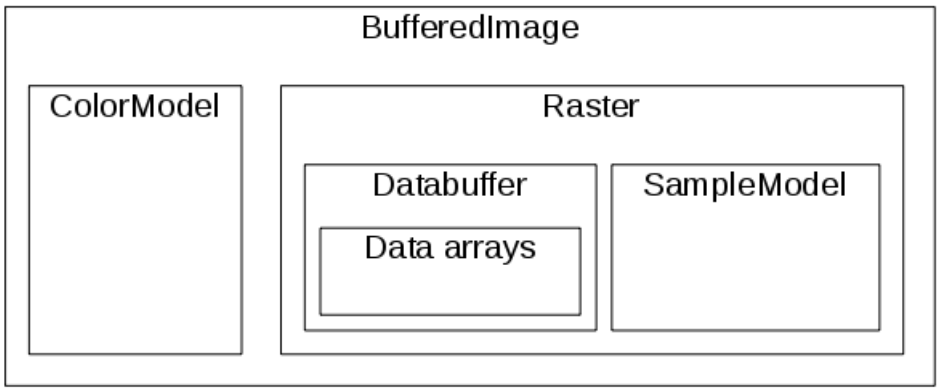
Once you've added the new operation, try different matrices and investigate the effect. Note that on the web you can find examples of matrices that others have had "luck" with.



7.2 BUFFEREDIMAGE

In this section, I would like to see a little more on the class *BufferedImage*, which is the class that provides the necessary services to manipulate the individual pixels in an image.

A BufferedImage is the central image processing class and is derived from the class *Image*, and an object of the type *BufferedImage* can therefore be used as a parameter for *drawImage()*. An image and thus an *Image* object is basically nothing but an array of pixels in different colors. A *BufferedImage* can be illustrated as follows:



and basically it consists of a *raster* and a color model. The raw data is saved as arrays in the raster portion, while the color model determines how data should be interpreted as colors. Each pixel in an image is defined by one or more values, called samples. For example has a black and white image one sample for each pixel, while an RGB image has 3 samples for each pixel. The buffer contains the current samples stored in arrays of byte or int arrays, and the raster part also has a *sample model* that defines how the buffer is organized and how to refer to a particular pixel in the buffer.

The color model interprets the individual pixels samples as colors. Is it a black and white image with one sample for each pixel it is interpreted as a grayscale between black and white, and in an RGB image, the color model uses all three samples for one pixel and interprets them as intensities of red, green and blue in an RGB color. All samples for a particular color are called a band or channel. Thus, an RGB image generally has 3 channels, but can also have a fourth for alpha values.

Consider, for example, a jpg image of 4920×2768 pixels. Since there is an RGB image, there are three samples for each pixels stored in three byte arrays each with space for 13618560 bytes. Each array is organized in the order of 2768 rows and each row uses 4920 places.

There are many details regarding a *BufferedImage*, and there are classes for all of the above components. It is rarely necessary to work directly with these classes, and I just want to show a small example that shows how to directly create a picture using a *BufferedImage*. The full code is as follows:

```
package createpicture;

import java.io.*;
import javax.imageio.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

public class MainWindow extends JFrame
{
 private Drawing drawing;
```

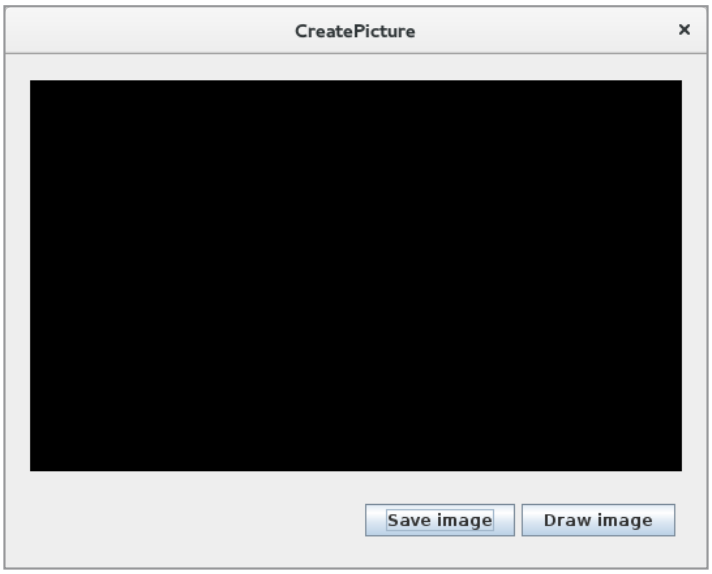
```
 public MainWindow()
 {
  super("CreatePicture");
  setLocationRelativeTo(null);
  createWindow();
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  pack();
  setVisible(true);
 }

 private void createWindow()
 {
  JPanel panel = new JPanel(new BorderLayout(0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(createBottom(), BorderLayout.SOUTH);
  panel.add(drawing = new Drawing());
  add(panel);
 }

 private JPanel createBottom()
 {
  JButton cmd1 = new JButton("Tegn billede");
  cmd1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
     drawing.draw();
    }
   });
  JButton cmd2 = new JButton("Gem billede");
  cmd2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
     try { File file = new File("billede.jpg");
     ImageIO.write(drawing.getImage(), "jpg", file); }
      catch (Exception ex) {};
    }
   });
  JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
  panel.add(cmd2);
  panel.add(cmd1);
  return panel;
 }
}

class Drawing extends JComponent
{
 private static final int WIDTH = 500;
 private static final int HEIGHT = 300;
 private BufferedImage image = new BufferedImage(WIDTH, HEIGHT,
  BufferedImage.TYPE_INT_RGB);
```

```
 public Dimension getPreferredSize()
 {
  return new Dimension(WIDTH, HEIGHT);
 }

 public BufferedImage getImage()
 {
  return image;
 }

 public void paintComponent(Graphics g)
 {
  super.paintComponent(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.drawImage(image, 0, 0, this);
 }

 public void draw()
 {
  int color = 0x00FF0000;
  for (int j = 0; j < 10; ++j)
   for (int i = 20; i < image.getWidth() - 20; ++i)
    image.setRGB(i, 20 + j, color);
  for (int j = image.getHeight() - 30; j < image.getHeight() - 20; ++j)
   for (int i = 20; i < image.getWidth() - 20; ++i) image.setRGB(i, j, color);
  for (int j = 30; j < image.getHeight() - 30; ++j)
   for (int i = 20; i < 30; ++i) image.setRGB(i, j, color);
  for (int j = 30; j < image.getHeight() - 30; ++j)
   for (int i = image.getWidth() - 30; i < image.getWidth() - 20; ++i)
    image.setRGB(i, j, color);
  int[] arr = new int[20000];
  Arrays.fill(arr, 0x000000FF);
  image.setRGB(150, 100, 200, 100, arr, 0, 200);
  repaint();
 }
}
```

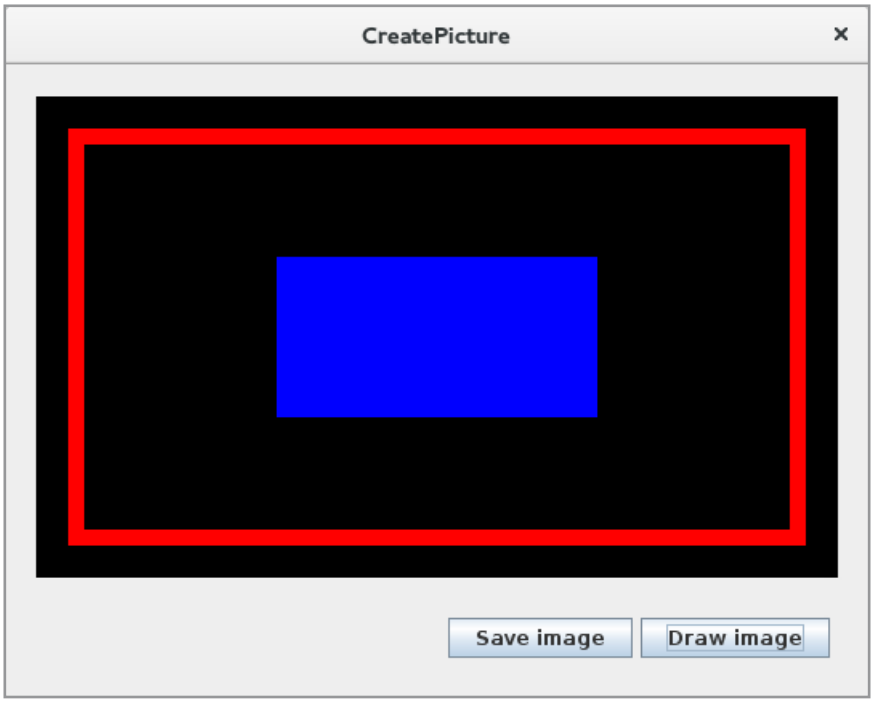
If you run the program, you will see the window on the next page. The window has a *BorderLayout*, where there is a panel with two buttons at the bottom, while the center is a *Drawing* component that draws an image. The image is a *BufferedImage*:

```
private BufferedImage image =
 new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
```

The class has more constructors, but in this case a constructor has been used to describe the size of the image and its color model. In this case, it is a three-channel RGB model, but the class *BufferedImage* defines constants for a number of other color models. As a result, in this case, an image of 500×300 pixels is created. The image is drawn in *paintComponent()* and when the result is a black image, it is because all sample data is 0 and hence the image buffer consists of three arrays (of length 150000) where all values are 0.



If you click the *Draw image* button, the *draw()* method is performed in the class *Drawing*. It modifies some pixels using the method *setPixel()*. It is available in two versions, one of which modifies a single pixel, while the other modifies a rectangular area of pixels in an array. The result is, as shown below:



Of course, it is not so often that there is a need to work with an image at this level, but the example shows that, using a *BufferedImage*, you can create an image from scratch and partly how to modify the individual pixels.

The last button is used to save the image as a jpg image, and the image can then be used as any other jpg image.

7.3 THE SCREEN

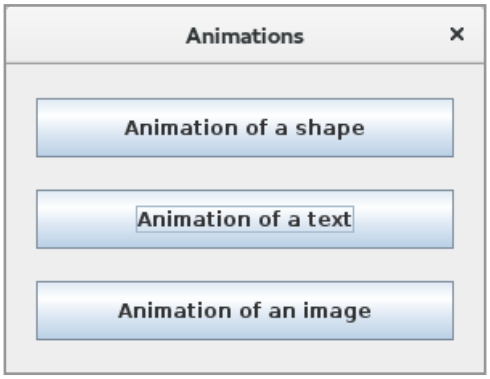
When working with graphics and images you may need to be able to determine the resolution of the monitor. This can be done in several ways, but the following program shows an example:

```
package screensize;

import java.awt.*;

public class ScreenSize
{
 public static void main(String[] args)
 {
  Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
  double width = screenSize.getWidth();
  double height = screenSize.getHeight();
  System.out.println(width + " x " + height);
 }
}
```

There is not much to explain besides taking note that you can get the screen resolution in this way.

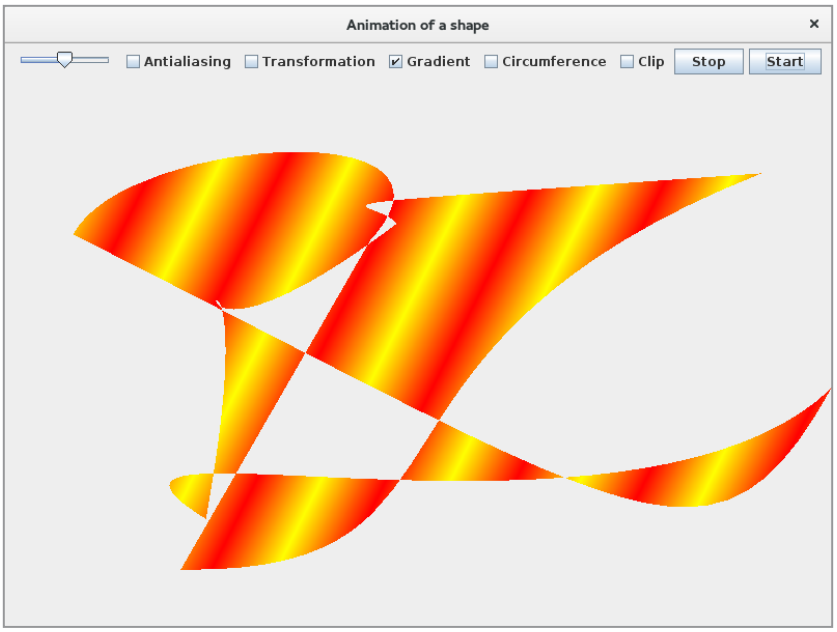
9 ANIMATIONS

As the last topic concerns Java2D, I will look at how to work with animations in a program. In fact, Java2D does not support animations, and all you can do is simulate animation of a figure by drawing the figure a little bit later and possibly in a new position. Doing it often (many times a second) gives you an effect that resembles an animation. The program *AnimationDemo* shows a little about what to do. If you open the program, you get the following window:



I will not mention the code for this window as it is only three buttons and each button opens a dialog box. The first dialog simulates an animation of a geometric figure, the next an animation of a text, while the last simulates an animation of an image.

Clicking at the top button gives you the window below which shows an animation of a relatively complex geometric figure:



At the top there is a toolbar, which includes five checkboxes indicating whether

1. antialiasing is used when drawing the figure
2. a transformation is performed, as a rotation about the center
3. a gradient paint is used (which is the case above)
4. a circumference is drawn around the figure
5. where to use a clip area (there are drawn as a text)

To the right there are two buttons used to start and stop the animation, and finally there is a slider that is used to adjust the animation speed.

In addition to this toolbar, the dial box contains a *JComponent*, which is the component that simulates the animation. I do not want to display the code for the dialog, but the component with the drawing is defined as an inner class as follows:

```
class Picture extends JComponent
{
 private float x;
 private float y;
 private float deltax = rand.nextFloat();
 private float deltay = rand.nextFloat();
 private float width;
 private float height;
 private float theta = 0;
 private BufferedImage image;
 private javax.swing.Timer timer;
```

```
public Picture(BufferedImage image, int delay)
{
 this.image = image;
 width = this.image.getWidth();
 height = this.image.getHeight();
 x = rand.nextFloat() * width;
 y = rand.nextFloat() * height;
 addComponentListener(new ResizeListener());
 timer = new javax.swing.Timer(delay, this::tick);
}

public void setTimer(int time)
{
 timer.setDelay(time);
}

public void start()
{
 timer.start();
}

public void stop()
{
 timer.stop();
}

private void tick(ActionEvent e)
{
 Dimension d = getSize();
 if (x + deltax < 0) deltax = -deltax;
 else if (x + width + deltax >= d.width) deltax = -deltax;
 if (y + deltay < 0) deltay = -deltay;
 else if (y + height + deltay >= d.height) deltay = -deltay;
 x += deltax;
 y += deltay;
 if (transform)
 {
  theta += Math.PI / 200;
  if (theta > 2 * Math.PI) theta -= (2 * Math.PI);
 }
 repaint();
}
```

```
public void paintComponent(Graphics g)
{
 Graphics2D g2d = (Graphics2D)g;
 setTransform(g2d);
 g2d.drawImage(image, AffineTransform.getTranslateInstance(x, y), null);
}

private void setTransform(Graphics2D g2)
{
 if (transform == false) return;
 Dimension d = getSize();
 g2.rotate(theta, d.width / 2, d.height / 2);
}
```

```
 class ResizeListener extends ComponentAdapter
 {
  @Override
  public void componentResized(ComponentEvent ce)
  {
   Dimension d = getSize();
   if (x < 0) x = 0;
   else if (x + width >= d.width) x = d.width - width - 1;
   if (y < 0) y = 0;
   else if (y + height >= d.height) y = d.height - height - 1;
  }
 }
}
```

Note first that the class inherits *JComponent* and thus is a component. The meaning of the class' variables should be self explanatory, but note the timer, which is the "engine" of the animation. Shapes are created by the method *createShape()*, and it creates a *GeneralPath* whit segments that are cubic curves defined by the points in the array *point*. The principle is that every time the timer is ticking, the figure is drawn, but first after the points have been modified slightly using the values in the delta array. Both the points and delta are initialized in the constructor, but here is not much to note besides the addition of an event handler for changing the window size. The handler should ensure that the shape's coordinates fall within the window after the size is changed.

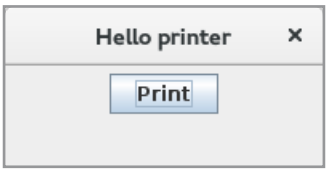
The event handler to the timer is called *tick()* and it modifies the points of the figure and ends with a *repaint()*. As a result, the window is repainted each time the timer is ticking. This means that *paintComponent()* is performed. Here you should note that it calls a number of auxiliary methods that set the graphics object for the checkboxes that are selected. Also note that the method creates the figure by calling *createShap()*, thus drawing the figure with the new coordinates.

Clicking in the main window of the other two buttons you get a result that animates a text and an image. I will not display the code for these components, as they are in principle are identical to the above, but you are encouraged to try the program, and especially when animating an image, you may find that it is difficult to the program to draw the image fast enough. This may be due to the current graphic card, but primarily because the method *drawImage()* does not have the desired performance.

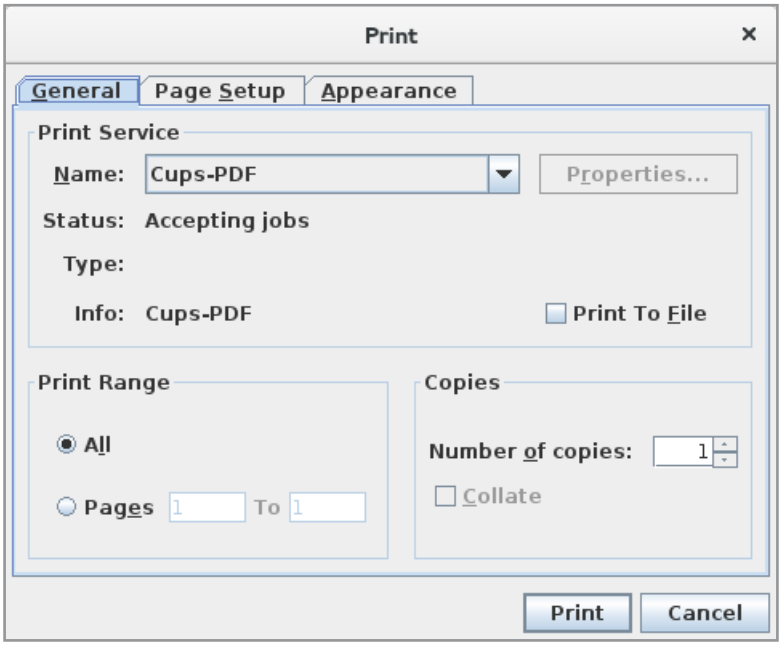
10 PRINT

In this chapter, I will describe how from a program to write to the printer. In our paperless society, it does not play the same role as before, but there are nevertheless some programs where you also want to print results on paper, so this chapter. With the development of Java, it has become easier to write to the physical printer, and among other things, several of Swing's components have built-in print facilities, but if you want to have full control over how data is being printed, some details are associated with the task. The chapter, like the rest of this book, consists of a number of examples of what you have to write.

I want to start simply with an application that opens the following window:



and clicking on the button you get the following dialog box:



which is Java's default dialog box for selecting a printer and settings. If you click *Print*, you print a single page with the text "*Hello world*". The code for the program's *MainView* is:

```
package helloprinter;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.print.*;

public class MainView extends JFrame implements Printable
{
 public MainView()
 {
  super("Hello printer");
  setSize(200, 100);
  setLayout(new FlowLayout());
  JButton cmdPrint = new JButton("Print");
  cmdPrint.addActionListener(this::print);
  add(cmdPrint);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
```

```
 public int print(Graphics g, PageFormat pf, int page) throws PrinterException
 {
  if (page > 0) return NO_SUCH_PAGE;
  Graphics2D g2d = (Graphics2D)g;
  g2d.translate(pf.getImageableX(), pf.getImageableY());
  g.drawString("Hello world!", 100, 100);
  return PAGE_EXISTS;
 }

 public void print(ActionEvent e)
 {
  PrinterJob job = PrinterJob.getPrinterJob();
  job.setPrintable(this);
  if (job.printDialog())
  {
   try
   {
    job.print();
   }
   catch (PrinterException ex)
   {
    JOptionPane.showMessageDialog(this, ex.toString());
   }
  }
 }
}
```

Regarding creating the window, there is nothing new, but you should note that the class implements the interface *Printable*, and you can generally print objects that implements this interface. The interface defines only a single method called *print()*, and this is the method that writes to the printer.

Looking at the event handler for the button *Print*, it creates a *PrinterJob* object, and as the name says, it is an object that represents a pint job. Next, the object to be printed – in this case *this* – is associated with the method *printDialog()*. It is the method that opens the above print dialog box. If it returns *true* corresponding to clicking the *Print* button, the method *print()* is performed on the job object, which in turn means that the method

```
public int print(Graphics g, PageFormat pf,
int page) throws PrinterException
```

is performed. The first parameter is the graphics to be drawn on, while the second is format of the page (defined in the dialog) and finally the last parameter is page number. The format defines, among other things, the clip area and the following statement translates the graphics to the upper left corner of the clip area:

```
g2d.translate(pf.getImageableX(), pf.getImageableY());
```

after which the text is drawn as any other text on a *Graphics* object.

The next example is called *PrintPages* and is an example of how to print multiple pages. The program has exactly the same user interface as the first example, so I'll just show the code that has to do with print, but first the following method:

```
private ArrayList<ArrayList<String>> createPages(int lines, int size)
{
 ArrayList<ArrayList<String>> pages = new ArrayList();
 ArrayList<String> page = new ArrayList();
 for (int n = 1; n <= lines; ++n)
 {
  page.add("This is line number " + n);
  if (page.size() == size)
  {
   pages.add(page);
   page = new ArrayList();
  }
 }
 if (page.size() > 0) pages.add(page);
 return pages;
}
```

which creates an *ArrayList* with objects of the type *ArrayList<String>* and simulates an *ArrayList* with text pages. The two parameters indicate respectively the total number of lines and number of lines for each page. The method is trivial and requires no special explanation.

The event handler for the *Print* button is the same as in the first example and is not shown here, so it's the *print()* method, where there is something to note:

```
public int print(Graphics g, PageFormat pf,
int page) throws PrinterException
{
 int height = g.getFontMetrics(font).getHeight();
 if (pages == null)
  pages = createPages(200, (int)(pf.getImageableHeight() / height));
```

```
 if (page >= pages.size()) return NO_SUCH_PAGE;
 Graphics2D g2d = (Graphics2D)g;
 g2d.translate(pf.getImageableX(), pf.getImageableY());
 for (int y = 0, i = 0; i < pages.get(page).size(); ++i)
 {
  y += height;
  g.drawString(pages.get(page).get(i), 0, y);
 }
 return PAGE_EXISTS;
}
```

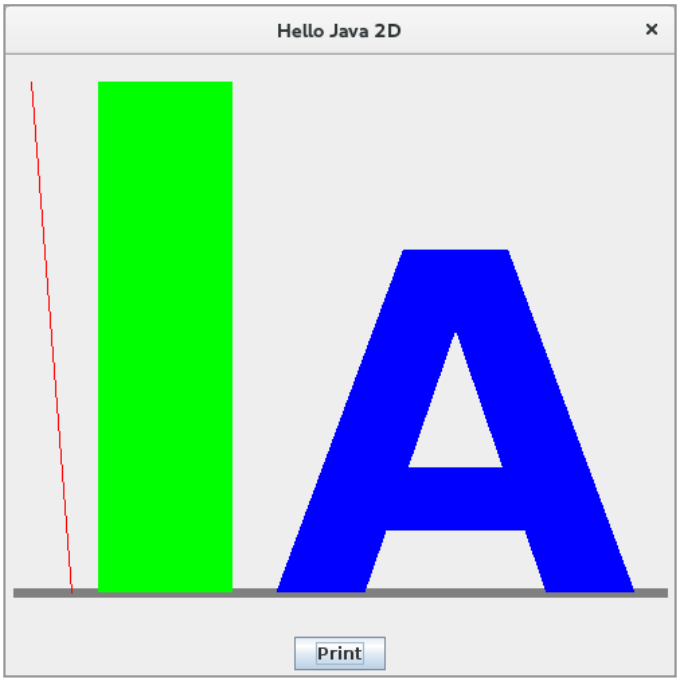
First, the height of the font used and thus the line height is determined, and it is used in the call of the method *createPages()* to determine the number of lines on the page. A *PageFormat* object defines the clip area, which is an area called *Imageable*, and to determine the number of lines the height of this area is used. Since multiple pages are printed this time, the method *print()* is called a corresponding number of times, and you should note that the method *createPages()* is called only once. The method *print()* should return *NO_SUCH_PAGE* if there are no more pages, and otherwise the method will return *PAGE_EXISTS*.

The rest of the method consists of printing the lines for a single page, which is referred to with the variable page, which specifies the page number, but otherwise the method only consists of a loop that loops over all lines and where a variable *y* keeps track of where to print.

Another example is *PrintOptions* and is almost identical to the first example and prints the text *"Hello world"*. The only difference is the event handler for the *Print* button, which shows that you can open a dialog box with the printer settings and print without the usual printer selection dialog.

```
public void print(ActionEvent e)
{
 PrinterJob job = PrinterJob.getPrinterJob();
 PrintRequestAttributeSet attr = new HashPrintRequestAttributeSet();
 PageFormat format = job.pageDialog(attr);
 job.setPrintable(this, format);
 try
 {
  job.print(attr);
 }
 catch (PrinterException ex)
 {
  JOptionPane.showMessageDialog(this, ex.getMessage());
 }
}
```

The next example is called *PrintWindow* and opens the following window:



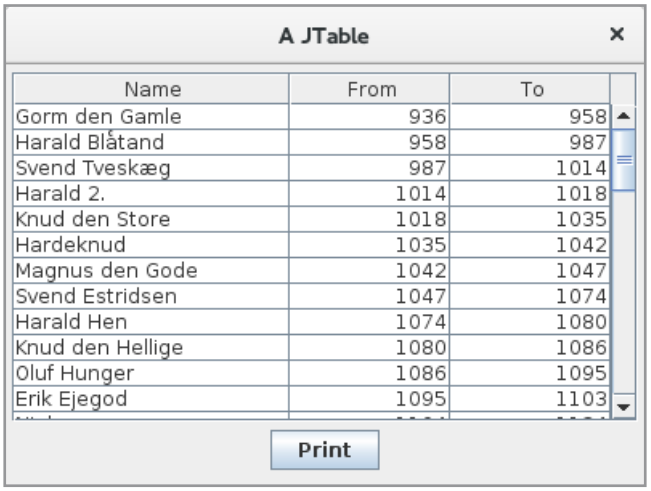
and is thus essentially the same program as the first example in this book, and the example should primarily show that you can print any graphic figure on the printer and that it is done in the same way as you print a figure on the screen. The code is essentially the same as the *HelloJava2D* program, except what is required for the button and the class implements *Printable*. The figure is similarly a *JComponent* called *Drawing*. The event handler of the button is identical to the event handler in the above example, so you should primarily note the method *print()* and how you can immediately print a *JComponent*:

```
public int print(Graphics g, PageFormat pf,
int page) throws PrinterException
{
 if (page > 0) return NO_SUCH_PAGE;
 Graphics2D g2d = (Graphics2D)g;
 g2d.translate(pf.getImageableX(), pf.getImageableY());
 drawing.paintAll(g);
 return PAGE_EXISTS;
}
```

10.1 SWING COMPONENTS

Looking at the above examples, it is generally simple to write to the printer, but in the end it is the responsibility of the programmer that what are printed can be on the paper and that it is properly divided into pages. It's what can make print of documents complex, but several of Swing's components have built-in facilities to print the content, and here are the two main examples *JTable* and *JTextComponent*.

The program *AJTable* opens the following window:



and clicking on the *Print* button will get a the print dialog and you can print the contents of the table. The code is the following, where I have not shown the data model for the table:

```
package ajtable;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.table.*;

public class MainView extends JFrame
{
 private JTable table = new JTable(new Kings());

 public MainView()
 {
  super("A JTable");
  setSize(400, 300);
  createView();
```

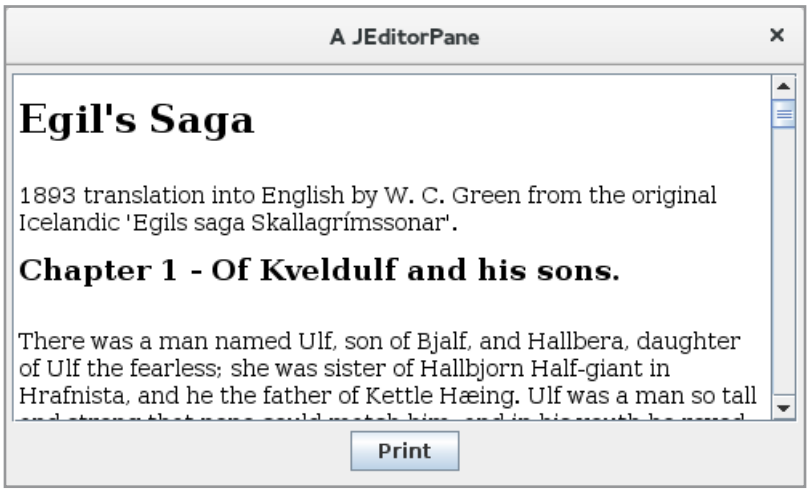
```
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(5, 5, 5, 5));
  setColumnWidth(table.getColumnModel().getColumn(1), 100);
  setColumnWidth(table.getColumnModel().getColumn(2), 100);
  panel.add(new JScrollPane(table));
  JButton cmdPrint = new JButton("Print");
  cmdPrint.addActionListener(this::print);
  JPanel command = new JPanel(new FlowLayout());
  command.add(cmdPrint);
  panel.add(command, BorderLayout.SOUTH);
  add(panel);
 }

 private void setColumnWidth(TableColumn col, int width)
 {
  col.setPreferredWidth(width);
  col.setMinWidth(width);
  col.setMaxWidth(width);
 }

 public void print(ActionEvent e)
 {
  try
  {
   table.print();
  }
  catch (PrinterException ex)
  {
   JOptionPane.showMessageDialog(this, ex.toString());
  }
 }
}
```

In fact, there is not much to explain and the design is quite simple. Note that the class does not implement *Printable*, and that the event handler of the button performs only the following statement:

```
table.print();
```

The statement opens the print dialog and prints the table contents. The *print()* method will automatically create the required number of pages. The method is found in several overrides, with parameters including, for example, a header and a footer, and it is quite simple to print a *JTabel*.

As another example, I will show a program *AEditorPane* that opens the following window:



The program shows the start of Egil's Saga in a *JEditorPane*. The document is a html document. Clicking on the *Print* button prints the document and thus the content of the *JEditorPane* component, and nicely formatted and divided into the required number of pages. It all happens quite automatically as in the previous program. I do not want to display the code here as there is nothing new about print.

10.2 PRINTSERVICES

Java's print API has evolved continuously, where the first versions of Java does not have support for print and so into an advanced API with many facilities for initiating print jobs. As the examples above shows, it is simple to write to a printer today, but when writing the program, one can not know which printer is located at the other end, and there may also be a variety of printers available with many different facilities. To handle these options, Java are expanded with *PrintServices*, and the following is a brief introduction to what it is.

The principle in a print job is with the new possibilities:

1. select the printer represented by a *PrintService* object
2. create the job represented by a *DocPrintJob* object
3. create a *Doc* object that represents the data to be printed
4. start the job by calling the *DocPrintJob* object's *print()* method

To select a printer, you uses static methods in the *PrintServiceLookup* class:

```
package printservicesprogram;

import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

public class PrintServicesProgram
{
 public static void main(String[] args)
 {
  PrintService[] services = { PrintServiceLookup.lookupDefaultPrintService() };
  print(services);
  print(PrintServiceLookup.lookupPrintServices(null, null));
  print(
   PrintServiceLookup.lookupPrintServices(DocFlavor.URL.TEXT_HTML_UTF_8, null));
  AttributeSet attrs = new HashAttributeSet();
  attrs.add(ColorSupported.SUPPORTED);
  attrs.add(OrientationRequested.LANDSCAPE);
  attrs.add(javax.print.attribute.standard.MediaSizeName.ISO_A3);
  print(PrintServiceLookup.lookupPrintServices(null, attrs));
  PrintRequestAttributeSet attr = new HashPrintRequestAttributeSet();
```

```
  PrintService service =
   ServiceUI.printDialog(null, 100, 100,
    PrintServiceLookup.lookupPrintServices(null, null),
    PrintServiceLookup.lookupDefaultPrintService(), null, attr);
  if (service != null) print(new PrintService[] { service });
  Attribute[] arr = attr.toArray();
  for (Attribute a : arr) System.out.println(a.getName() + ": " + a);
 }

 private static void print(PrintService[] services)
 {
  for (int i = 0; i < services.length; ++i)
   System.out.println(services[i].getName());
  System.out.println();
 }
}
```

The class *PrintServiceLookup* has a method *lookupPrintServices()* that returns an array of *PrintService* objects available, and it will be all the printers found. Using the method without parameters, you simply get all the printers the computer knows, but you can with two parameters specify which printers to return. The two parameters are respectively a *DocFlavor* and a *AttributeSet*, where the first indicates a MIME type, and thus what kind of documents the printer should be able to print, while the second indicates what other features the printer should be able to support. As an example from the above code, the following statement will return all printers that can print UTF8 encoded HTML:

```
PrintServiceLookup.lookupPrintServices(DocFlavor.URL.TEXT_HTML_UTF_8, null)
```

As another example, the following statement returns all printers that have the properties defined by the last parameter:

```
PrintServiceLookup.lookupPrintServices(null, attrs)
```

and in this case, it would be a color printer that should support LANDSCAPE and it should be an A3 printer.

To select a printer, you can also use *ServiceUI.printDialog()*, which opens the usual print dialog. The first three parameters indicate the parent window and the dialog box location relative to this window. Also note the last parameter, which is a *PrintRequestAttributeSet*, which returns the properties that the selected printer supports. You are encouraged to test the program on your machine. In addition, you are encouraged to investigate the help as to what constants and types are defined to specify *DocFlavor* and *AttributeSet* and there are many.

As an example, I will show a program where you can choose an image and print it. The program opens a custom browser to the file system where you can select an image and print it on the default printer. The program's code is as follows:

```
package printimage;

import java.io.*;
import java.awt.*;
import java.awt.print.*;
import javax.swing.*;
import javax.swing.filechooser.*;
import javax.imageio.*;
import javax.print.*;

public class PrintImage
{
 private ImageIcon icon = null;

 public static void main(String[] args)
 {
  (new PrintImage()).printImage();
 }

 private void printImage()
 {
  JFileChooser chooser =
   new JFileChooser(new File(System.getProperty("user.home")));
  chooser.setFileFilter(new FileNameExtensionFilter("Picture", "jpg"));
  if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
  {
   icon = loadImage(chooser.getSelectedFile());
   if (icon != null)
   {
    try
    {
     PrintService service = PrintServiceLookup.lookupDefaultPrintService();
     DocPrintJob job = service.createPrintJob();
     DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
     SimpleDoc doc = new SimpleDoc(new Picture(), flavor, null);
     job.print(doc, null);
    }
    catch (Exception ex)
    {
    }
   }
  }
 }
```

```
private ImageIcon loadImage(File file)
{
 try
 {
  return new ImageIcon(ImageIO.read(file));
 }
 catch (Exception ex)
 {
  return null;
 }
}

class Picture implements Printable
{
 public int print(Graphics g, PageFormat format, int page)
 {
  if (page > 0) return Printable.NO_SUCH_PAGE;
  Graphics2D g2d = (Graphics2D)g;
  g2d.translate((int)(format.getImageableX()), (int)(format.getImageableY()));
  scaleGraphics(g2d, format);
  g2d.drawImage(icon.getImage(), 0, 0, null);
  return Printable.PAGE_EXISTS;
 }
```

```
  private void scaleGraphics(Graphics2D g2d, PageFormat format)
  {
   double pageWidth = format.getImageableWidth();
   double pageHeight = format.getImageableHeight();
   double imageWidth = icon.getIconWidth();
   double imageHeight = icon.getIconHeight();
   double scaleX = pageWidth / imageWidth;
   double scaleY = pageHeight / imageHeight;
   double scale = Math.min(scaleX, scaleY);
   g2d.scale(scale, scale);
  }
 }
}
```

It all happens in the method *printImage()*, which opens a *JFileChooser* dialog box where you can select a jpg file. The file is opened with the method *loadImage(),* which returns the image as an *ImageIcon*. If an image has been opened, a *PrintService* object is determined for the default printer:

```
PrintService service = PrintServiceLookup.lookupDefaultPrintService();
```

and for this *PrintService*, a *DocPrintJob* is created:

```
DocPrintJob job = service.createPrintJob();
```

To print the image is defined an inner class *Picture*, which implements *Printable* and thus a page to the printer. The class looks like previous *Printable* classes and must implement the method *print()*, but as the entire image should be on the page, the graphics is scaled so that there is room for the entire image. You should note how the scaling is done, as it is a task that is often needed. With this class available, you can define the document to be printed:

```
SimpleDoc doc = new SimpleDoc(new Picture(), flavor, null);
```

and here you should especially note the flavor parameter, whose type is

```
DocFlavor.SERVICE_FORMATTED.PRINTABLE;
```

which indicates that a single page must be printed formatted as a *Printable* object.

I will then show a program called *PrintBook*, which prints three pages:

```
package printbook;

import java.awt.*;
import java.awt.font.*;
import java.awt.print.*;
import java.awt.geom.*;
import javax.print.*;

public class PrintBook
{
 public static void main(String[] args)
 {
  try
  {
   PrintService service = PrintServiceLookup.lookupDefaultPrintService();
   DocPrintJob job = service.createPrintJob();
   PageFormat format = new PageFormat();
   Book book = new Book();
   book.append(new Page0(), format);
   book.append(new Page1(), createFormat1(format));
   book.append(new Page2(), createFormat2(format));
   DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PAGEABLE;
   SimpleDoc doc = new SimpleDoc(book, flavor, null);
   job.print(doc, null);
  }
  catch (Exception ex)
  {
  }
 }

 private static PageFormat createFormat1(PageFormat df)
 {
  PageFormat pf = (PageFormat) df.clone();
  Paper paper = df.getPaper();
  paper.setImageableArea(20, 20, df.getWidth() - 40, df.getHeight() - 40);
  pf.setPaper(paper);
  return pf;
 }

 private static PageFormat createFormat2(PageFormat df)
 {
  PageFormat pf = (PageFormat) df.clone();
  pf.setOrientation(PageFormat.LANDSCAPE);
  return pf;
 }
}
```

```
class Page0 implements Printable
{
 public int print(Graphics g, PageFormat format, int page) throws PrinterException
 {
  Graphics2D g2d = (Graphics2D) g;
  double width = format.getImageableWidth();
  double height = format.getImageableHeight();
  double xpos = format.getImageableX();
  double ypos = format.getImageableY();
  Ellipse2D ellip = new Ellipse2D.Double(
   width / 2 - 50 + xpos, height / 2 - 50 + ypos, 100, 100);
  g2d.setColor(Color.red);
  g2d.fill(ellip);
  return Printable.PAGE_EXISTS;
 }
}

class Page1 implements Printable
{
 public int print(Graphics g, PageFormat format, int page) throws PrinterException
 {
  Graphics2D g2d = (Graphics2D) g;
  double width = format.getImageableWidth();
```

```
  double height = format.getImageableHeight();
  double xpos = format.getImageableX();
  double ypos = format.getImageableY();
  Rectangle2D rect = new Rectangle2D.Double(xpos, ypos, width, height);
  Line2D line1 = new Line2D.Double(xpos, ypos, xpos + width, ypos + height);
  Line2D line2 = new Line2D.Double(xpos, ypos + height, xpos + width, ypos);
  g2d.setColor(Color.blue);
  g2d.draw(rect);
  g2d.draw(line1);
  g2d.draw(line2);
  return Printable.PAGE_EXISTS;
 }
}

class Page2 implements Printable
{
 public int print(Graphics g, PageFormat format, int page) throws PrinterException
 {
  Graphics2D g2d = (Graphics2D) g;
  g2d.translate((int)(format.getImageableX()), (int)(format.getImageableY()));
  double width = format.getImageableWidth();
  double height = format.getImageableHeight();
  TextLayout text = new TextLayout("Hello", new Font("Serif", Font.BOLD, 144),
   g2d.getFontRenderContext());
  Rectangle2D rect = text.getBounds();
  text.draw(g2d, (int)(width - rect.getWidth()) / 2,
   (int)(height + rect.getHeight()) / 2);
  return Printable.PAGE_EXISTS;
 }
}
```

The three pages are defined as inner classes and are all *Printable* objects. The first page draws a filled circle with a radius of 50 centered at the page. The second page draws a frame beyond the clipping area and two diagonals. Finally, the last page draws a text centered at the page.

main() creates a *PrintSevice* for the default printer and for this printer creates a print job. Next, a default *PageFormat* is created, which is a *PORTRAIT* page format with a margin of 1 inch. As the next step is created a *Book* object, which really is just a collection of *Printable* objects, but where each object has a *PageFormat*. For this collection, the three pages are assigned, where the first uses the default *PageFormat* object, while the other page uses a *PageFormat* with a margin of 20 points, and finally the last one use a *LANDSCAPE PageFormat*. After the *Book* object is initialized with the three pages, a *Doc* object is created

```
SimpleDoc doc = new SimpleDoc(book, flavor, null);
```

that this time has a flavor of the type

```
DocFlavor.SERVICE_FORMATTED.PAGEABLE
```

and thus a multi-page document.

10.3 PRINT TEXT

I will finish this chapter on print with an example where it is the programmer that formats the individual pages from scratch. It's rarely needed, but vice versa, it gives you total flexibility with regard to printing on a printer. The program is called *HelloEgil*.

The task is to print a document with a cover page consisting of a name and a picture and a text consisting of a number of lines, each line being perceived as a paragraph. Each line starts with a digit (1, 2 or 3) and indicates the paragraph type that is used in the program as a code for the font to be used.

The program starts loading the image as an *ImageIcon* with the name *icon*, and the text as an *ArrayList<String>* named *list*, where each element is a line (and thus a paragraph). Next, the following method is performed:

```
private void print()
{
 PrinterJob job = PrinterJob.getPrinterJob();
 PageFormat format = job.defaultPage();
 Book book = new Book();
 book.append(new FrontPage(), format);
 pagination(book, format);
 job.setPageable(book);
 if (job.printDialog())
 {
  try
  {
   job.print();
  }
  catch (PrinterException ex)
  {
   System.out.println(ex);
  }
 }
}
```

which starts by creating a *PrinterJob* and a default *PageFormat* for this job. As a next step, a *Book* object is created and a *FrontPage* object is added, which is a page to the cover page. *FrontPage* is an inner class that represents a *Printable* object, an object that draw the image scaled and centered on the page. There is another inner class called *TextPage*, which represents a text consisting of a number of lines, each line being a *TextLayout* object. This class also implements *Printable*. The main task of the program is to divide the document into *TextPage* objects and thus text pages to the printer. This happens in the following method:

```
private void pagination(Book book, PageFormat format)
{
 try
 {
  TextPage page = new TextPage();
  int width = (int)format.getImageableWidth();
  int height = (int)format.getImageableHeight();
  int pos = 0;
  FontRenderContext frt = new FontRenderContext(null, false, false);
  for (String line : list)
  {
   Font font = font12;
   if (line.length() > 0)
   {
```

```
    if (line.charAt(0) == '1') font = font36;
    else if (line.charAt(0) == '2') font = font24;
    int lineHeight = font.getSize();
    AttributedString styledText = new AttributedString(line.substring(1));
    styledText.addAttribute(TextAttribute.FONT, font);
    AttributedCharacterIterator charIterator = styledText.getIterator();
    LineBreakMeasurer measurer = new LineBreakMeasurer(charIterator, frt);
    while (measurer.getPosition() < charIterator.getEndIndex())
    {
     if (pos + lineHeight < height) page.add(measurer.nextLayout(width));
     else
     {
      book.append(page, format);
      pos = 0;
      page = new TextPage();
      page.add(measurer.nextLayout(width));
     }
     pos += lineHeight;
    }
    if (pos + lineHeight < height)
    {
     page.add(new TextLayout(" ", font, frt));
     pos += lineHeight;
    }
    else
    {
     book.append(page, format);
     pos = 0;
     page = new TextPage();
    }
   }
   else
   {
    int lineHeight = font.getSize();
    if (pos + lineHeight < height)
    {
     page.add(new TextLayout(" ", font, frt));
     pos += lineHeight;
    }
    else
    {
     book.append(page, format);
     pos = 0;
     page = new TextPage();
    }
   }
  }
```

```
  if (page.getSize() > 0) book.append(page, format);
 }
 catch (Exception ex)
 {
  System.out.println(ex.toString());
 }
}
```

I do not have to review the code here, but it uses the same technique for text manipulation as I have mentioned in a previous chapter in this book, where it is the task of the program to measure how much text may be on the individual lines, as well as associates a font with the text. Then, the *Book* object is initialized with *Printable* objects, and the report can be sent to the printer.

11 MAINTENANCE OF PROGRAMS

Most programs must be maintained over time. This is one of the reasons why program design and program quality play a major role as programs with an incomprehensible design and code that are difficult to read are almost impossible to maintain and at best it may be time consuming to maintain a program with a bad design.

Many programs live for many years, and this period requires a program to be maintained at intervals. One reason may be that the program contains errors, which should of course be corrected. These are rarely the most difficult maintenance tasks, as errors are usually recognized shortly after the programs are launched, and the errors should typically be corrected by the same people who have developed the program, and thus by developers who have an overview of and understand the program code. Other reasons for maintenance is that the program needs to be expanded with new features or existing features should work in a different way, and for a large program that will be used over several or many years, there will certainly be requests for such changes over time. Here, the original design has a much greater significance for the maintenance task, as the task will typically be performed by anyone other than the developers who originally developed the program, and although they should be the same people, it is far from given that they can remember, how the program is written. In fact, one will be amazed at how fast one forgets how a program is written. Finally, a reason for maintaining a program may be that you simply want to modernize the program and market it as a new version, which is quite common for various standard programs like office suites and so forth.

When there is a maintenance task either because a program needs to be expanded or a new version of the program has to be developed, it is typically a new development project. In principle, such a project is like other software development projects, but if it is a good quality program with a good design, and if the program is to be expanded or part of the program to be modified, the project can be limited to include what needs to be expanded or changed while the rest of the program can live on unchanged. If, on the other hand, the program has been developed without plained for future maintenance, there is a great risk that changes may affect side effects to large portions of the overall code, and in the worst case even smaller changes may mean that large portions of the code must be rewritten.

Fortunately, with a good design, it is seldom so bad, but there is always reason to be aware that changes in a program can easily cause unintended changes to the code elsewhere with errors. Therefore, maintenance of programs must be planned to ensure that you have identified the consequences of the changes in question and fully uncovered what is to be tested subsequently and, if possible, documentation is required, so developers who are making changes can quickly gain an overview of the program's architecture and how it is made. The start of any maintenance task is therefore to read the program documentation and especially the design document and to get an overview of the full code so that you know the significance of the individual project files. The result of all that is, that one can not clearly emphasize the importance of design and program architecture. Otherwise, the program can not be maintained, and programs that can not be, it is in principle worthless.

Regardless what, the maintenance of programs it is not simple, and similar to the development of programs from startup, maintenance also starts with an analysis analyzing the changes to be made and the program to be maintained. Then modify/expand existing design before coding the changes. In particular, it is important that the documentation is also updated so it is clear what has been changed, why and when, and finally, the changes must be tested, with particular attention being paid to the fact that the changes do not cause errors or inconvenience elsewhere in the program.

As an example, the rest of this chapter will contain two tasks that will illustrate program maintenance. In one task, you need to implement some extensions of the calendar application from the book Java 8, and in the second task, you must make two changes to the slot machine from the book Java 9.

PROBLEM 2

Start by creating a copy of the *Calendar* project from the book Java 8. Take some time to test the program so you are sure that you can remember how the program works.

The task now is to add three print features to the program:

1. The toolbar must have two new icons, and if you click on the one the program should prints a calendar for the current month, while clicking on the other icon the program should prints a calendar for the current year.
2. The dialog box to maintenance notes must have an additional button and clicking on the button will allow you to print the note.

PROBLEM 3

Start by creating a copy of the *SlotMachine* project from the book Java 9. Take some time to test the program so you are sure that you can remember how the program works.

The task is to modify the program:

1. In the existing version, there is no animation of the wheels, but the animation is simulated by displaying the individual figures quite shortly before the next figure is selected. You now need to change the program, so instead simulating the rotation of the reels with an animation in the same way as in chapter 9.
2. The program has an unfortunate administration of the machine's administrator, where you can switch to administrator mode by clicking an icon in the toolbar. It should be changed so that a particular user (player) has administrator right. If this user has logged in, the icons for administrator tools appear automatically and otherwise not. It is a part of the task to ensure that there always is an administrator and that the administrator can not be deleted.

12 PACHART

By a chart library, you usually understand a family of classes that you can use in a program to represent numbers using graphs. Examples are histograms, bar charts, pie charts, and more. A good example of using such tools is a spreadsheet that offers different forms of graphical representation of numbers. The aim of the following project is to develop such a library as well as write a program that can illustrate how the library can be used. There are many such libraries, and they differ as to which graphs there are, how "nice" the graphs are, which options are possible, and also as the most important how easy the library is to use.

The purpose of the project is not so much the value of the library (there are so much of that kind), but it is a very good exercise in the use of Java2D, especially if the aim is to develop "nice" graphs (whatever it may be) and if you focus on developing a user-friendly library. Regarding the latter, keep in mind that user-friendliness is not the same as the greatest possible flexibility, but to a greater extent that the library is easy to understand and use. Finally, it is extremely important that such a library is robust and that the graphs behave sensibly if used on an incomplete or inappropriate data basis, and it is actually not so easy.

In the following, I would not focus on the process, but instead what is made and how the library is used.

12.1 THE LIBRARY

Basically, a graph should visualize numbers associated with categories (labels). A classic example is year numbers, where there are associated a number with each year. The current library should basically offer the following graphs:

- *Line*, where a graph consists of straight lines between points
- *Histogram* or a *bar chart*, which is probably the most commonly used graph
- *Circle* or pie chart

Each of these three basic graphs is available in several variants. A line graph has two variants

- *Curve*, where the graph is instead drawn as a soft curve
- *Plot* where the graph is drawn as points – a point for each category (label), which may be connected with thin straight lines

A bar chart is drawn as default with vertical bars, but there is a variant in which the bars instead are drawn horizontally, and finally there is a variation of both the vertical and horizontal bar charts with a simple 3D effect.

Finally, there is a variant of the pie chart, where the chart is drawn as a speedometer – just to point out that it is only the imagination (and the ability to draw nice graphs) that limit the graphs, that such a library should contain.

The library thus contains 9 basic (or simple) graphs, which are defined by the following type:

```
package pagraphs.charts;

public enum GraphType { LINE, CURVE, PLOT, VBAR,
HBAR, VBAR3D, HBAR3D, PIE, SPEED }
```

In addition, for a plot graph, you can indicate which figure is used to represent the graph's points, and these options are defined by the following type, where the names indicate which figure is used:

```
package pagraphs.charts;

public enum PlotType { CROSS, STAR, OPENCIRCLE, OPENSQUARE, OPENTRIANGLE,
 OPENRHOMBE, CLOSEDRHOMBE, CLOSEDCIRCLE, CLOSEDSQUARE, CLOSEDTRIANGLE }
```

A graph as above is called simple (or single-valued) corresponding to the fact that in principle it is defined by two arrays:

1. *labels* that is an array of strings and represent the categories and hence the independent variable or the x axis
2. v*alues* which is an array of numbers of the same length as *labels* and represent the dependent variable or the y axis

A simple graph is thus a graph for a mathematical function, and in addition to the two arrays, a graph has attached other parameters, which indicates how the graph is to be drawn.

A graph can also be multi-valued, which means that each label has several numbers and the difference is that the array *values* in principle are a 2-dimensional array. A multi-valued graph is as default drawn as several simple graphs after each other (that is, as a series of graphs). A multi-valued graph may in particular be merged, which means that the individual graphs are drawn in the same coordinate system – to the extent that it makes sense. This is exactly the part that makes the development of the following library both complex and comprehensive.

Basically, a graph is defined using two data structures. The first is called *SingleValues* and represents the y values of a simple graph:

```
package pagraphs.charts;

import java.awt.*;

public class SingleValues
{
 private String label;
 private String header;
 private GraphType type;
 private double begin;
```

```
 private double end;
 private int points;
 private int dec;
 private Paint color;
 private PlotType plot;
 private double param1;
 private double param2;
 private boolean onoff;
 private final double[] values;

 public SingleValues(GraphType type, String label, String header, double begin,
  double end, int points, int dec, Paint color,
PlotType plot, double param1,
  double param2, boolean onoff, double … values)
 {
  this.type = type;
  this.label = label;
  this.header = header;
  this.begin = begin;
  this.end = end;
  this.points = points;
  this.dec = dec;
  this.color = color;
  this.plot = plot;
  this.param1 = param1;
  this.param2 = param2;
  this.onoff = onoff;
  this.values = values;
 }
}
```

In short, the variables means the following:

- *label* defines a label for this function and is typical used as a label for the y axis, default is blank
- *header* if a name or description for this graph, default is blank
- *type* is the type of this graph, default is *VBAR*
- *begin* defines the minimum value on the y axis, default is 0
- *end* defines the maximum value on the y axis, default is 0 and if both *begin* and *end* are 0 the library automatically calculates til values
- *points* defines the number of intervals on the y axis, default is 0 and then the library automatically calculates the value
- *dec* defines the number of decimals to be used on the y axis, default is 0 and then the library automatically calculates the value

- *color* that defines the color for the graph – if not defined a default color is used
- *plot* which defines the figure used for a plot graph (the value is only used for a plot graph)
- *param1* that is a special parameter whose application depends on the graph type (is used for a *LINE*, *CURVE* and *SPEED* graph), default is 0
- *param2* that is a special parameter whose application depends on the graph type (is used for a *SPEED* graph), default is 0
- *onoff* that is a special parameter whose application depends on the graph type (is used for a *PIE* and a *PLOT* graph), default is *false*
- *values* the is values for this graph

The class has the necessary *get* and *set* methods, and to make it easier to define simple graphs, the class has more constructors than shown above.

The class *SimpleValues* only defines a single dependent variable. A graph that can be single-valued or multi-valued is defined as follows:

```
package pagraphs.charts;

import java.util.*;
import java.awt.Color;

public class MultiValues // defines a multi valued graph
{
 public static final Color colors[] =
 {
  new Color(0x7C, 0xAF, 0xDD),
  new Color(0xF1, 0x9B, 0x5A),
  …
 };

 public final List<SingleValues> list;
 public final String labels[];
 private String label;
 private boolean gitter;
 private boolean merge;
 private boolean same;
 private boolean stacked;
 private double scaleX;
 private double scaleY;
 private double minimum;
 private double maximum;
```

```
 public MultiValues(List<SingleValues> list, String label, boolean gitter,
  boolean merge, boolean same, boolean stacked, double scaleX, double scaleY,
  String … labels)
 {
  this.list = list;
  this.label = label;
  this.gitter = gitter;
  this.merge = merge;
  this.same = same;
  this.stacked = stacked;
  this.scaleX = scaleX;
  this.scaleY = scaleY;
  this.labels = labels;
 }
}
```

Initially, an array of default colors is defined, which is used if no colors are defined. Otherwise, the individual properties means the following:

- *list* is a list with objects of the type *SingleValues* which defines this graph
- *labels* is an array of the type *String* with texts for the independent axis (x axis or label axis)
- *label* that is a label to x axis, default is blank
- *gitter* specifies whether to draw grid lines (*true* that is default) or not (*false*)
- *merge* that indicates whether graphs should be merged into a single graph (*false* is default)
- same that specifies whether all graphs (in a merge graph) must use the same y axis (*true* is default) – the property is only used for a merge graph
- *stacked* that specifies whether *BAR* and *BAR3D* graphs should be stacked (*true*) or drawn side by side (*false* that is default) – the property is only used for a merge graph
- *scaleX* that defines the scaling in the x axis (horizontal) direction (1 is default and means no scaling)
- *scaleY* that defines the scaling in the y axis (vertical) direction (1 is default and means no scaling)
- *minimum*, that is used for a merge graph to calculate the minimum value for the y axix
- *maximum*, that is used for a merge graph to calculate the maximum value for the y axix

The class has the necessary *get* and *set* methods, and to make it easier to define graphs, the class has more constructors than shown above.

As mentioned, there are 9 different graphs, as well as the possibility to merge multiple graphs in the same coordinate system. The graphs are drawn using *Drawer* classes, which are tasked with drawing a particular graph as well as capturing clicks with the mouse. If you click on a graph with the mouse, it must raise an event:

1. if clicking on a label (on the x axis), the name is associated with the event
2. if clicking on the graph itself, the event has associated both the name on the x axis label and the corresponding function value

An event has the type *ChartEvent* that has an argument of the following type:

```
package pagraphs.charts;

public class FunctionValue
{
 private final String name;
 private final Double value;

 public FunctionValue(String name, Double value)
 {
  this.name = name;
  this.value = value;
 }

 public boolean hasValue()
 {
  return value != null;
 }

 public String getName()
 {
  return name;
 }

 public Double getValue()
 {
  return value;
 }
}
```

Here is *value null* if a label is clicked. The reason is that a graph may be multi-valued and thus there may be several values attached to a particular label. A listener object for events of this type will implement the interface *ChartListener*, which defines a single event handler.

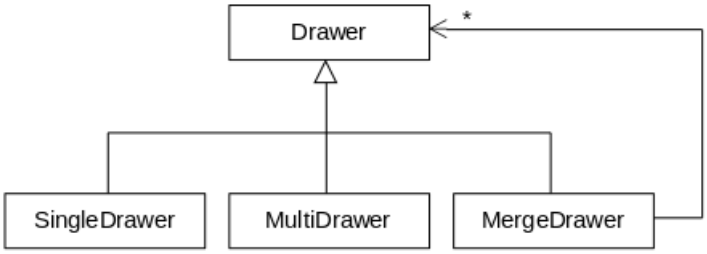
A graph is a custom component:

```
public class ChartComponent extends JComponent implements MouseListener
{
 private ArrayList<ChartListener> listeners = new ArrayList();
 private MultiValues data;
 private Color borderColor;
 private int gap;
 private java.util.List<Drawer> drawers;
```

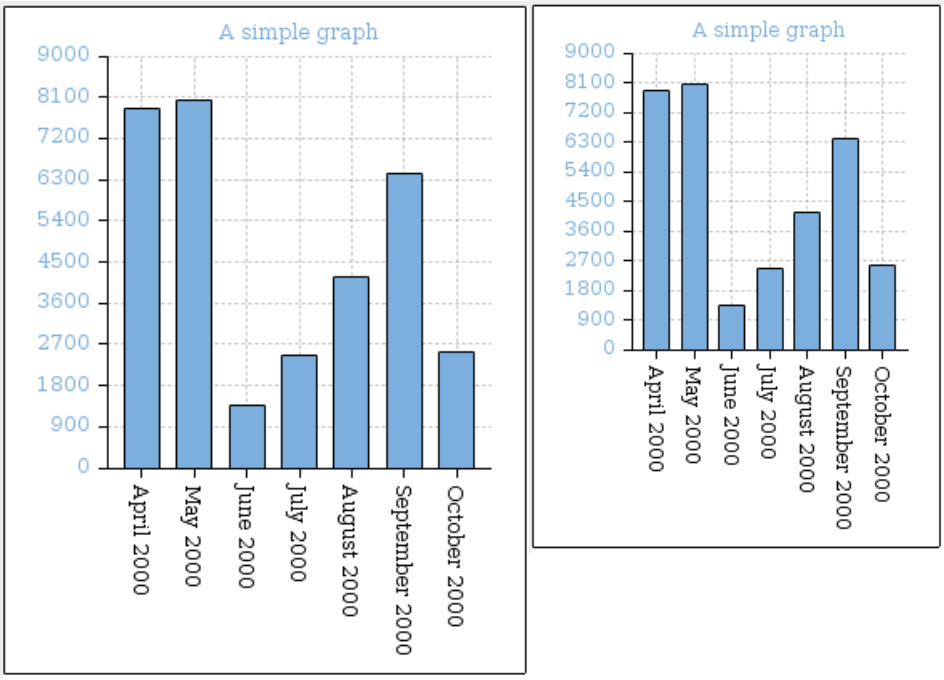
```
public ChartComponent(MultiValues data, Font font, int gap, Color borderColor)
{
 this.data = data;
 this.gap = gap;
 this.borderColor = borderColor;
 setFont(font);
 drawers = DrawerFactory.createDrawer(data, getFont());
 addMouseListener(this);
 setBackground(Color.white);
}

public Dimension getPreferredSize()
{
 double width = 0;
 double height = 0;
 for (int n = 0; n < drawers.size(); ++n)
 {
  width += drawers.get(n).getWidth(n) + gap;
  double h = drawers.get(n).getHeight(n);
  if (height < h) height = h;
 }
 return new Dimension((int)width, (int)height);
}
```

```
 public void paintComponent(Graphics g)
 {
  Graphics2D g2d = (Graphics2D)g;
  Dimension size = getSize();
  g2d.setPaint(getBackground());
  g2d.fill(new Rectangle2D.Double(0, 0, size.width, size.height));
  g2d.setFont(getFont());
  DrawingTools.smootGraphics(g2d);
  double offset = 0;
  if (data.isMerge()) offset =
    ((MergeDrawer)drawers.get(0)).draw(g2d, 0, 0, offset);
  else for (int n = 0; n < data.list.size(); ++n)
    offset += ((SingleDrawer)drawers.get(n)).draw(g2d, n, 0, 0, offset) + gap;
  if (borderColor != null)
  {
    g2d.setStroke(new BasicStroke(0.5f));
    g2d.setPaint(borderColor);
    Dimension dim = getPreferredSize();
    g2d.setClip(new Rectangle2D.Double(0, 0, dim.width + 1, dim.height + 1));
    g2d.draw(new Rectangle2D.Double(0, 0, dim.width, dim.height));
  }
 }
 public void addChartListener(ChartListener listener)
 {
  listeners.add(listener);
 }

 public void removeChartListener(ChartListener listener)
 {
  listeners.remove(listener);
 }

 public void mouseClicked(MouseEvent e)
 {
  for (int n = 0; n < drawers.size(); ++n)
  {
    FunctionValue functionValue =
     drawers.get(n).getClicked(n, e.getX(), e.getY());
    if (functionValue != null)
    {
     ChartEvent event = new ChartEvent(this, functionValue);
     for (ChartListener listener : listeners) listener.chartClicked(event);
     return;
    }
  }
 }
}
```

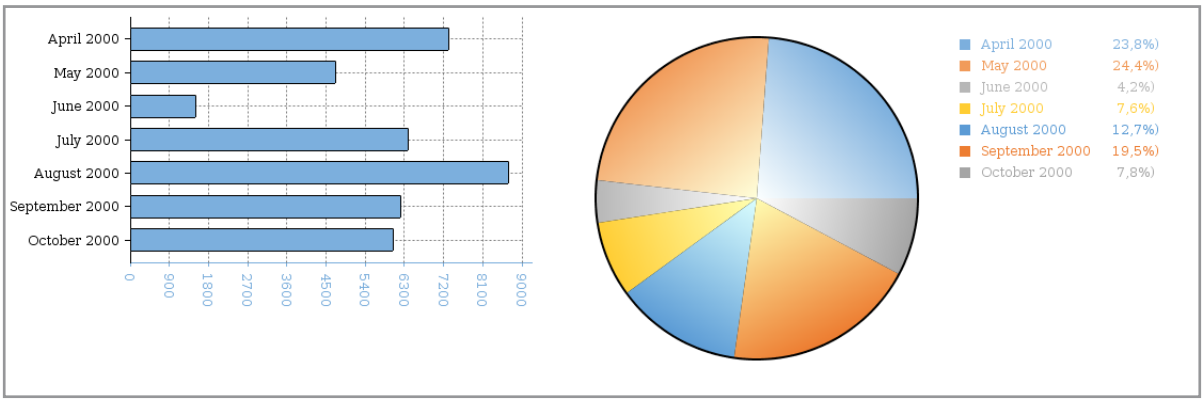
The component is created on the basis of a *MultiValues* object and hence the definition of a graph. In addition, you must specify the font to be used and you can specify a gap between the individual graphs (if it is a multi-valued graph) and if an edge must be drawn around the graph. The class has more constructors than shown above. Using the graph definition, the constructor creates a list of *Drawer* objects, where there is a *Drawer* for each graph. These *Drawer* objects are created using the class *DrawerFactory*, which is a simple factory class. The method *paintComponent()* uses these *Drawer* objects to draw the component, and it is also *paintComponent()* which draws an edge about the component. Should an edge be drawn, the preferred size of the component is used, which is also determined using the *Drawer* objects. The same applies to the mouse to determine whether a graph is clicked. That is, the *Drawer* classes are absolutely key classes:



Drawer is an interface that defines three methods that define the width and height of the nth graph in a *ChartComponent*, and also a method that returns a *FunctionValue* object if this *Drawer* is clicked. In general, a graph is drawn as a rectangular figure with a margin. The margin is used, for example, to the axes of the coordinate system, to a header text, and what the margin of the figure is used for and thus the size of the margin is so determined by the current graph (not all graphs have a coordinate system). *SingleDrawer* is an abstract class, which implements the interface's functions as well as methods that determine a graph's margin and draw the graph. The class also defines scaling functions, and here you should be aware that they only scale the graph, but not the margin. As an example, two graphs are shown below, where the difference only is that the second is scaled:

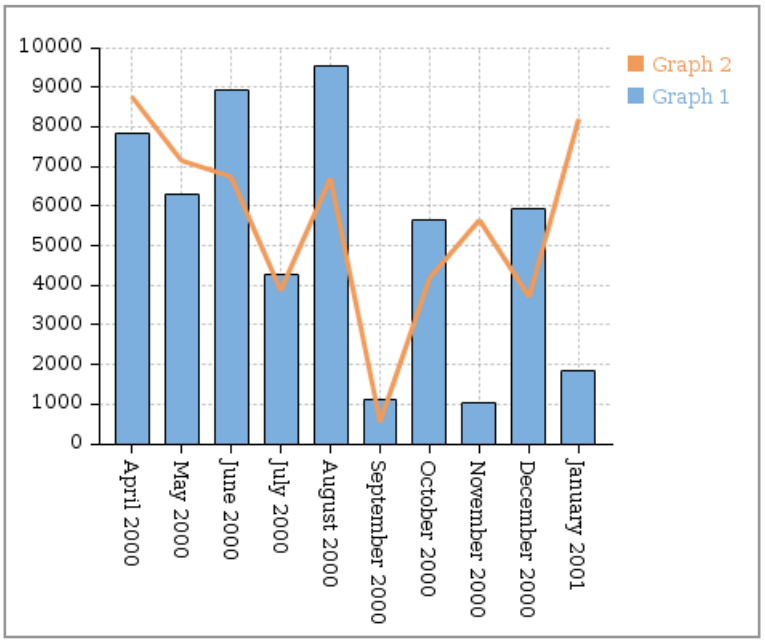


The type of the above graph is *VBAR*, and there is a specific class *VbarDrawer*, derived from *SingleDrawer*. It is this drawer that is used to draw the above graphs. There are correspondingly 8 other specific *Drawer* classes, which are derived from *SingleDrawer*. As another example, there is shown below a multi-valued graph, where the first has the type *HBAR* and is drawn with a *HbarDrawer*, while the other is of the type *PIE* and is drawn with a *PieDrawer*:



The code for these 9 concrete drawer classes is similar to each other and, in principle, they arre quite simple, although there may of course be many details attached to implementing the drawing functions. The classes use more auxiliary classes, including classes that draw the axes. Here, especially the classes that draw the y axes are complex, as they typically depend on the actual numbers to make a sensible division of the axis.

Then there is the class *MergeDrawer* which is also a concrete class, but it draws the graph using other *Drawer* objects. If you have a multi-valued function, you can draw the graphs in the same coordinate system, which is what I have called a merge graph. As an example, below is shown a merge of a *LINE* graph and a *VBAR* graph:



Now you can not generally merge all graphs. For example, it makes no sense to merge a histogram (a VBAR graph) and a pia graph, and it makes no sense to merge a VBAR and a HBAR graph. The *MergeDrawer* class starts by dividing the graphs into some categories similar to how they can be merged, and for each of these categories, a *Drawer* class is defined. It leads to no less than 14 new *Drawer* classes, all of which are in principle similar to the above, but instead, they are derived from the abstract class *MultiDrawer*, that as a single drawer this time always draws a multi-valued graph. When there are so many *MultiDrawer* classes due to

1. that you can specify whether merged graphs should use the same y axis or each use there own y axis
2. that you can specify whether bar charts should be drawn side by side or stacked

Specifically, it corresponds to the following *MultiDrawer* classes:

1. *MSLCPBDrawer*, which merges LINE, CURVE, PLOT and VBAR graphs with the same y axis
2. *MDLCPBDrawer*, which merges LINE, CURVE, PLOT and VBAR graphs with different y axis
3. *MSVBAR3DDrawer*, which merges VBAR3D graphs with the same y axis
4. *MDVBAR3DDrawer*, which merges VBAR3D graphs with different y axis
5. *MSHBARDrawer*, which merges HBAR graphs with the same y axis
6. *MDHBARDrawer*, which merges HBAR graphs with different y axis

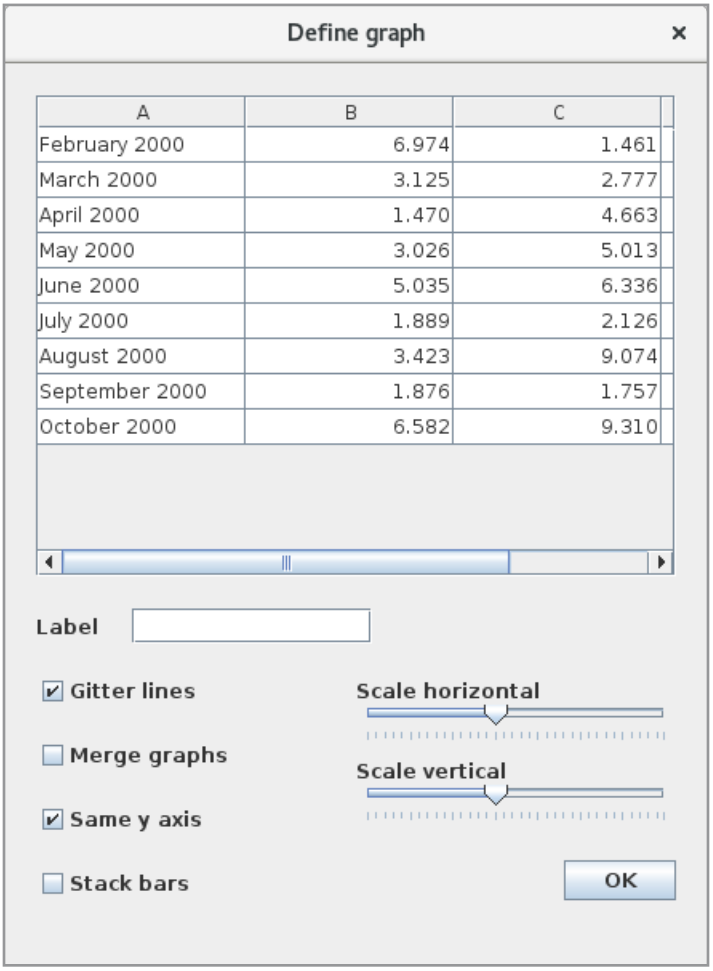
7. *MSHBAR3DDrawer*, which merges HBAR3D graphs with the same y axis
8. *MDHBAR3DDrawer*, which merges HBAR3D graphs with different y axis
9. *MSLCPDrawer*, which merges LINE, CURVE and PLOT graphs with the same y axis, when stack of bar charts are selected
10. *MSLCPDrawer*, which merges LINE, CURVE and PLOT graphs with the different y axis, when stack of bar charts are selected
11. *SVBARDrawer*, which stacks VBAR graphs
12. *SHBARDrawer*, which stacks HBAR graphs
13. *SVBAR3DDrawer*, which stacks VBAR3D graphs
14. *SHBAR3DDrawer*, which stacks HBAR3D graphs

If a multi-valued graph is merged and there is a graph that can not be merged with others, it is drawn as a *SingleValues* graph using the corresponding *Drawer*.

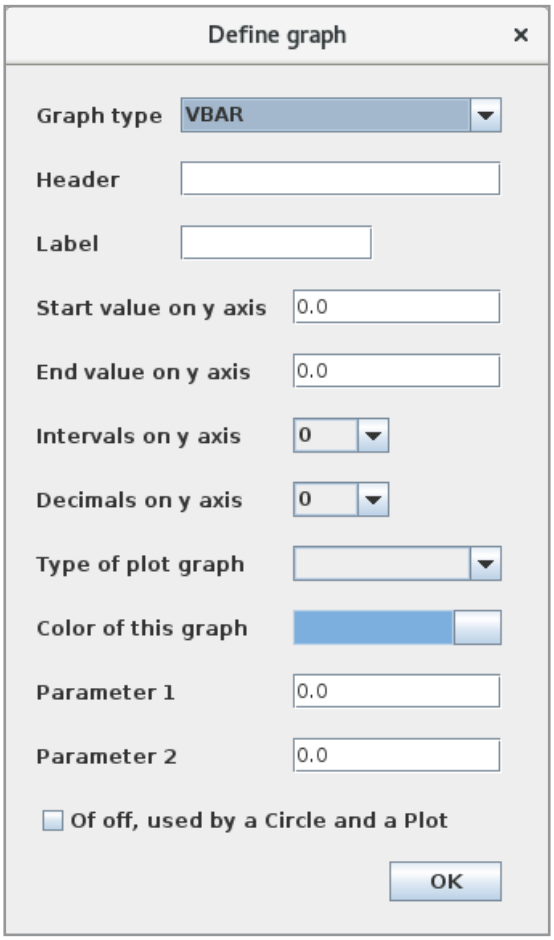
The result of all these is that there is a lot of code related to merge of graphs, but the work consists primarily of writing the respective *Drawer* classes, all of which look similar.

12.2 THE TEST PROGRAM

Then there is the test program that opens a window with a *JTable* containing numbers – near the first column, which contains texts to be used as labels for graphs. At the top there is a toolbar with buttons. Clicking on the first button allows you to specify an interval for the table's numbers, and the program will then fill the table with random numbers within this range. Next, 19 test buttons follows, which open windows I have used for testing in connection with the development of the library. The most interesting is the button to the far right. If you select a rectangular area in the table (which does not include the left column) and clicking the button the program creates a *MultiValues* object where you can edit the properties:



Here it has been selected three columns and defined a multi-graph for three graphs. Double-clicking the column header for one of the three graphs gives you the following dialog box where you can define settings for the individual graphs:



You can thus use the program to define different settings for the individual graphs and test the effect.

Note that for practical use, the last dialog box must be adapted to the individual graph types, so you can only enter and select values for the properties that make sense for the current graph.

12.3 LACKS AND THINGS THAT COULD BE IMPROVED

The graphs can not display all numbers with a reasonable result, and some of the graphs, for example, are mistaken for negative numbers, especially if you merges graphs. For example, it does not make sense to stack bar charts if the numbers can be negative. The conclusion is that several of the graphs have preconditions, which the drawing functions with advantage could validate and possibly raise an execption if the graph can not be drawn correctly. An alternative could be to let the *Drawer* classes return 0 if the graph can not be drawn with a sufficiently nice result – a strategy that is already partially implemented.

The actual component *ChartComponent* can raise a single event if the user clicks on a graph. The component is drawn based on an object of the type *MultiValues*, and the component could be extended as it fires a *PropertyChangeEvent* if a value for a property is changed in the *MultiValues* data structure. In practice, it would typically mean that the component should be redrawn, and it could also be of significance to the layout manager that contains the component.

Another question is how easy it is to expand the library with a new graph. If it is a whole new graph for a single-valued function, it's simple. In this case, the work consists primarily of writing a new *Drawer* derived from *SingleDrawer*, adding a new graph type to *GraphType* and expanding the class *DrawingFactory*. In addition, the *MergeDrawer* class must be expanded (the method *merge()*) to know the new type. If graphs of the new type can also be merged, a *Drawer* derived from *MultiDrawer* must also be written, and the class *MergeDrawer* has to know this *Drawer*. Even more extensive is the task, if the new graph type can be merged with existing graphs, which can both cause existing *Drawer* types to bc changed as well as possible new *Drawer* classes to be written. The conclusion is that if the library is expanded with new graphs, which can be merged with existing graphs, the work can be quite extensive.

It may be a good practice to improve the library corresponding to the above and optionally expand with a new graph type.