Poul Klausen

JAVA 9

Swing, Documents and printing

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 9: SWING, DOCUMENTS AND PRINTING

SOFTWARE DEVELOPMENT

Java 9: Swing, Documents and printing: Software Development
1st edition

ISBN 978-87-403-1933-0
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the ninth 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. This book deals with Swing and how to use Swing to develop applications with a graphical user interface. I have also looked at the development of programs with a graphical user interface in the book Java 2, and this book is a continuation of Java 2, but with greater emphasis on details and how Swing works internally. Several of the chapters of the book include topics that are not used so often in practical programming, but on the other hand, topics that it is necessary to know in part to solve specific problems, and partly to develop user interfaces with focus on quality. After reading the book, the reader should be able to develop and maintain complex user interfaces written using Swing. The book requires knowledge of programming and Java throughout with special introductory knowledge of Swing corresponding to the contents of the book Java 2. The book ends with the development of an application, with primary focus on the graphical user interface.

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

In the book Java 2, I have considered the development of programs with a graphical user interface and including Swing, which is the Java API for the development of GUI applications, and I have also in subsequent books considered more examples of GUI programs. Specifically, I have in Java 6 about databases treated *JTable*, which is the most complex of the Swing components. All what previously are said regarding GUI programs are good enough, but there's a lot of more to tell, and including several technical concepts of Swing. The following deals primarily with what not previously are said, but some will also be repetitions and elaborations of what I already have touched on concerning the development of GUI applications.

I will divide the book in the following chapters and the individual chapters are largely independent of each other and can be read in any order:

1. Swing details
2. Layout
3. Swing components
4. User defined components
5. The clipboard
6. Drag and drop
7. Edit text
8. Internationalization
9. Final example

Part of the following are perhaps beyond the needs encountered in everyday life, but as important issues, I would particularly highlight *drag and drop* as well as *the clipboard*, there are issues that necessarily must be mastered to work as a software developer in practice. Also I will mention the development of custom components.

1.1 A COMMENT ABOUT SWING

Before I address the subjects of this book I want to mentioning three classes, which I have already used, but may not have been referred to profit, and at least, it is worth knowing the classes, and I will use them as follows when needed.

First, I would like to mention *SwingUtilities* which is a class that contains many static utility methods. I have used the methods *invokeLater()* and *invokeAndWait()* to ensure that a method is performed in the event dispatcher thread. There are many other methods and you should study the class and the methods it provides. Most of the methods you rarely will ever need, but it is good to know their existence.

Another class is called *SwingConstants*, and as the name says, the class defines a number of constants. You are encouraged to investigate which ones. Generally, the different *Swing* components defines constants for example regarding alignment. That is, the same constants are defined in several places, and the meaning by *SwingConstants* is instead of using the constants defined in the individual components classes, then you should use the constants in *SwingConstants*.

Finally, there is the class *SwingWorker*, which is an abstract class that is used to perform time-consuming GUI operations in one or more background threads. The goal is that time-consuming tasks should not be performed in the event dispatcher thread, as a program then will be in a state where it is not responding. The goal of the class is to make it easier to perform time-consuming tasks in their own threads rather than yourself has to write the necessary code for threads.

Comparing what is said about Swing in this book and the book Java 2, you are familiar with most of what Swing makes available to write programs with a graphical user interface. Everything is not explained, and in practice it is necessary to constantly look up the online documentation, partly because Swing is so extensive that it is impossible to learn everything, and partly because Swing is also developing. Swing is a very successful API for developing GUI programs, but is also criticized for the need to writing a lot. It's also correct, but if you develop your own class libraries that fit the tasks that you typically solve, Swing can actually be done quite efficiently. However, there are alternatives and when the language is Java, it is JavaFX, which is a newer API for developing graphical user interfaces, but it is the subject for the book Java 16.

2 SWING DETAILS

The programs user interface is developed by means of components which the user can do something with, and examples are *JButton* and *JTextField*, and there are about 30 – or slightly more, depending on how you count – so there is a lot to work with. The basic base component is called *JComponent* and is an abstract class containing all that is common to Swing components.

All the Swing components today support the Java Beans specification, and it means that components consist of properties, where a property is an instance variable with accessor methods. If, for example a variable has the name *propertyName* and describes a property of the component, you can expect that the component has methods *setPropertyName()* and *getPropertyName()* or *isPropertyName()*. The first only if it is a property whose value must be changed, and the last only if its type is *boolean*.

Properties can be divided into

1. *Simple* properties, that are properties, not firing an event when the value is changed.
2. *Bound* properties, that are firing a *PropertyChangeEvent* when the value changes, and where an object of the type *PropertyChangeListener* can be registered as a listener for a *PropertyChangeEvent* with the method *addPropertyChangeListener()*.
3. *Constrained* properties, which also fired a *PropertyChangeEvent* but before the value is changed. An object of the type *VetoableChangeListeners* can be registered as a listener for these events with the method *addVetoableChangeListener()* (only a few components fires these events).

A *PropertyChangeEvent* has assigned three values: The name of the property, the old value and the new value.

A component can also fire a *ChangeEvent* that is fired when a property change state. Objects of the type *ChangeListener* can be registered as a listeners for a *ChangeEvent*. Such an event is only associated with a single value, which is the component that raises the event.

Another kind of properties are called *client properties* and are key/value pairs that are stored in a *Hashtable* attached to each Swing component. This means that, with the method *putClientProperty()* you can add or remove a client property. For example you can add a property in the following manner:

```
component.putClientProperty("propertyName", propertyValue);
```

and you can remove it with the statement

```
component.putClientProperty("propertyName", null);
```

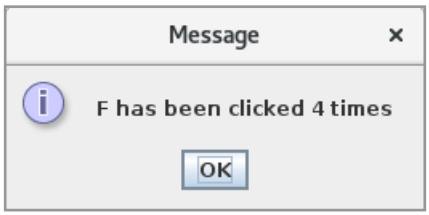
You can get the value with the statement:

```
value = component.getClientProperty("propertyName");
```

Client properties allows you to add properties at runtime, and you can think of them as an opportunity to associate additional data with a component. Client properties are bound properties and fired a *PropertyChangeEvent* when the value is changed. For performance reasons, you should only occasionally add client properties for a component, and if you need new features of a component, you should consider writing a derived class.

As an example, the program *PropertiesProgram* opens the following window:

If you click on one of the buttons, you get a message box that shows how many times the button has been pressed:



and when you click *OK* you get another message box that shows that the number of clicks is counted by one (here it is counted from 3 to 4):

Below is the completed program code:

```
package propertiesprogram;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;
import javax.swing.border.*;
```

```
public class MainView extends JFrame implements PropertyChangeListener
{
 public MainView()
 {
  super("PropertiesProgram");
  setResizable(false);
  createView();
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  pack();
  setLocationRelativeTo(null);
  setVisible(true);
 }

 public void propertyChange(PropertyChangeEvent e)
 {
  if (e.getOldValue() != null)
  {
   JButton cmd = (JButton)e.getSource();
   JOptionPane.showMessageDialog(this, String.format("%s: %d -> %d",
     cmd.getText(), e.getOldValue(), e.getNewValue()));
  }
 }

 private void createView()
 {
  JPanel panel = new JPanel(new GridLayout(3, 3, 5, 5));
  panel.setBorder(new EmptyBorder(5, 5, 5, 5));
  panel.add(createButton("A"));
  panel.add(createButton("B"));
  panel.add(createButton("C"));
  panel.add(createButton("D"));
  panel.add(createButton("E"));
  panel.add(createButton("F"));
  panel.add(createButton("G"));
  panel.add(createButton("H"));
  panel.add(createButton("I"));
  add(panel);
 }

 private JButton createButton(String text)
 {
  JButton cmd = new JButton(text);
  cmd.setPreferredSize(new Dimension(50, 50));
  cmd.setMargin(new Insets(0, 0, 0, 0));
  cmd.putClientProperty("clickCounter", 0);
  cmd.addPropertyChangeListener(this);
  cmd.addActionListener(new ClickHandler());
  return cmd;
 }
```

```
 class ClickHandler implements ActionListener
 {
  public void actionPerformed(ActionEvent e)
  {
   JButton cmd = (JButton)e.getSource();
   Integer value = (Integer)cmd.getClientProperty("clickCounter");
   value += 1;
   JOptionPane.showMessageDialog(MainView.this, String.format(
     "%s has been clicked %d times", cmd.getText(), value));
   cmd.putClientProperty("clickCounter", value);
  }
 }
}
```

Note that the *MainView* class implements the interface *PropertyChangeListener* and can thus be registered as listener for *PropertyChangeEvent*'s. Otherwise, the most important method is *createButton()*, which creates a button. Note how to define a client property for the button, which means that a button has a counter that from the start has the value 0. The button's event handler has the the type *ClickHandler* (an inner class) and it displays the first of the above message boxes. Here you should note how to refer to the client property *clickCounter*, and also that the event handler counts this property with 1. When this happens, the button fires an *PropertyChangeEvent*, which in this case means that the *MainView's propertyChange()* method is performed (*MainView* is registered as listener) that opens the second dialog box.

EXERCISE 1

Create a copy of the project *PropertiesProgram*. You must now change the program to perform exactly the same, but instead of associating a client property with the buttons, you should write a type *CounterButton* that extends *JButton* with a property *counter*, and the window should then contain 9 *CounterButton* components. Note that this means that the method *setCounter()* should fire a *PropertyChangeEvent*.

2.2 SIZE AND LOCATION

As described in the book Java 2, a component's size is defined by three properties of the type *Dimension*:

1. *preferredSize*
2. *minimumSize*
3. *maximumSize*

but to what extent these properties are used, is as explained in Java 2, completely determined by the current layout manager. In fact, it is allowed to write a layout manager, that completely ignores these properties. It is absolutely essential to mention that the exact size and location of a component in a window is determined by the current layout manager. You can also define the size and location of a component with *setBounds()*, but here you should be aware that a layout manager suppresses the values set with *setBounds()* (or *setSize()* and *setLocation()*). If you want to place the components in this way, remove the layout manager by setting it to *null*.

As regards the size and location of a component, note the following methods:

- *getWidth()*
- *getHeight()*
- *getSize()*
- *getBounds()*
- *getLocation()*

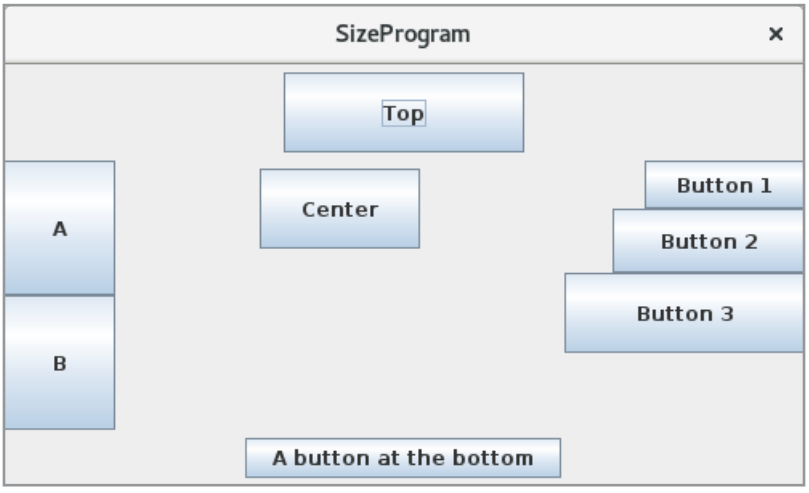
which returns the component's current size and location.

You can also define a component's alignment within the container where it is located and the basic methods are

- *setAlignmentX()*
- *setAlignmentY()*

and both methods have a *float* parameter, where 0 means left or top, 1 means right or bottom, and a value between these extremes indicates the degree to which the component is to be adjusted. For example means 0.5 centered. The only layout managers that use these properties are a *BoxLayout* or an *OverlayLayout*.

As an example of all that, I will show a program that opens the following window:



The window has 8 buttons and when you click on a button, you get a message box. The window has a *BorderLayout*, which lay out five panels:

- at the top a *FlowLayout* with a button
- to the left a *GridLayout* with two buttons
- at the bottom a *FlowLayout* with a button
- to the right a *BoxLayout* with three buttons
- center a *FlowLayout* with one button

The code for the top panel is:

```
private JPanel top()
{
 JPanel panel = new JPanel(new FlowLayout());
 JButton cmd = new JButton("Top");
 cmd.setPreferredSize(new Dimension(150, 50));
 cmd.addActionListener(e -> show(cmd));
 panel.add(cmd);
 panel.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e) { show(panel); }});
 return panel;
}
```

If you click on the top button, you get the result:

and that means that the current width and height of the button is determined by its preferred size, and it corresponds to, that a *FlowLayout* layout components so that the sizes is their preferred size. Clicking in the top panel outside the button you get the result:

The panel's preferred size is 160 × 60, which is determined by the button's preferred size, but a *FlowLayout* inserts as default a margin of 5 (a space on 5 pixels between the components). The panel's current height is 60, which is determined by the outer *BorderLayout*, since the height of NORTH is the height of the component, that is the panel. The width is, on the other hand, the width of the window, which is 500, and the current width of the top panel will therefore be 500.

The code for the left panel is:

```
 private JPanel left()
 {
  JPanel panel = new JPanel(new GridLayout(2, 1));
  JButton cmd1 = new JButton("A");
  JButton cmd2 = new JButton("B");
  cmd1.setPreferredSize(new Dimension(50, 50));
  cmd2.setPreferredSize(new Dimension(70, 50));
  cmd1.addActionListener(e -> show(cmd1));
  cmd2.addActionListener(e -> show(cmd2));
  panel.add(cmd1);
  panel.add(cmd2);
  return panel;
}
```

If you click on the button *A* you get the message box:

The current width is 70, which is actually determined by the preferred size of the bottom button, since the width of the WEST component in a *BorderLayout* is determined by the component's preferred size. Here it is a *GridLayout* component, and its width is determined by the largest of the two components. When the current height is 84, it is because the *GridLayout* component divides the total height into two equal parts. You should note that the button's preferred size is completely ignored. If you click on the B button, you get:

Then there is the code for the panel at the bottom:

```
private JPanel bund()
{
 JPanel panel = new JPanel(new FlowLayout());
 JButton cmd = new JButton("A button at the bottom");
 cmd.addActionListener(e -> show(cmd));
 panel.add(cmd);
 panel.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e) { show(panel); }});
 return panel;
}
```

In particular, note that the button this time has not defined any preferred size. Clicking on the button gives you the result:

As in the first case, the current size of the button is determined by its preferred size, but you should note that the button's preferred size is determined by the text and the current font. Clicking in the bottom panel, but outside the button you get:

and the explanation is the same as with the top panel. The right panel has the following code:

```
private JPanel right()
{
 JPanel panel = new JPanel();
 panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
 panel.add(createCmd("Button 1", 100, 30));
 panel.add(createCmd(" Button 2", 120, 40));
 panel.add(createCmd(" Button 3", 150, 50));
 panel.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e) { show(panel); }});
 return panel;
}

private JButton createCmd(String text, int width, int height)
{
 JButton cmd = new JButton(text);
 cmd.setPreferredSize(new Dimension(width, height));
 cmd.setMinimumSize(new Dimension(width, height));
 cmd.setMaximumSize(new Dimension(width, height));
 cmd.addActionListener(e -> show(cmd));
 cmd.setAlignmentX(Component.RIGHT_ALIGNMENT);
 return cmd;
}
```

and here you should primarily notice the preferred size for the three buttons. If you click on the buttons, you get:

That is, in all three cases, the current size of the button is the same as its preferred size and then a *BoxLayout* thus determines the size of components using their preferred sizes. If you click at the bottom of the panel, you get

and thus the size of the right panel is the largest width of the components as well as the height of the window. You should note that the panel's preferred height is 120, which is the sum of the heights of the three buttons, and a *BorderLayout* ignores this value (*EAST* uses the window height).

Back there is the middle panel:

```
private JPanel center()
{
 JPanel panel = new JPanel(new FlowLayout());
 JButton cmd = new JButton("Center");
 cmd.setPreferredSize(new Dimension(100, 50));
 cmd.addActionListener(e -> show(cmd));
 panel.add(cmd);
 panel.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e) { show(panel); }});
 return panel;
}
```

If you click on the button, you get:

which shows that the current size of the button is its preferred size as it is in a *FlowLayout*. Clicking outside the button, you get

and from this, you can see that CENTER in a *BorderLayout* ignores the preferred size of the panel.

The example thus shows that it is different how the preferred size is used.

2.3 EVENT HANDLING

Events appear generally when a key is pressed on the keyboard or when something happens to the mouse (clicking on a button or moving the mouse). Swing components can fire many different kinds of events, which are types defined in *java.awt.event* or *javax.swing.event*, and many of them are component specific. Each event type defines at least the object that has fired the event in question but will also often contain other information about the state of the object. However, there are also events that are not raised by a component.

In order for an object to receive notifications regarding an event, the object must be registered as listening to that event. To make it possible, the class of the object must implement a listener interface (for example *ActionListener*), and then the object can be registered (for example, with *addActionListener()*).

The abstract class *JComponent* defines a protected field of the type *EventListenerList*, called *listenerList*, which all components inherit. The type *EventListenerList* is an array of pairs of the form *XXEvent* / *XXListener*, where XX is an event type. This array contains the listeners of the component and which events is listened to, and the component can thus send notifications to listeners when an event occurs. When an array is used instead of a collection, it is to increase performance similar to a collection such as an *ArrayList* is a relatively complex class. In fact, it means that a new array must be created (and copied) every time a listener is added, but it is still cheaper than using a collection, because it is limited how many listeners are registered for a given component.

All events are handled by event handlers by the registered listener objects, and as described in the book Java 8, these handlers must be performed by the *event dispatching thread*, which is the thread that carries out all about location and drawing of the individual components. The thread is linked to a FIFO queue of events pending processing, and the individual events are processed in the order in which they are inserted in the queue and the next one will not be processed until the previous one is completed. Otherwise, one could risk the state of a component being changed while it, for example, became redrawn. For this reason, you must not perform an event handler outside the event-dispatching thread by, for example, to call the method *fireXX()* directly from another thread. It is also important that event handlers and code that draws the components are effective, as you may else risk blocking the system from events in the event queue waiting for treatment.

To ensure that all codes associated with event handlers are performed in the event dispatching thread, I have in Java 8 shown how the *SwingUtilities* class has two methods *invokeLater()* and *invokeAndWait()*, which adds a runnable object to the system event queue. A classic pattern to ensure that a code is performed in the event dispatching thread is to execute the code as follows:

```
public void doThreadSafeWork()
{
 if (SwingUtilities.isEventDispatchThread())
 {
  // do all work here…
 }
 else
 {
  Runnable callDoThreadSafeWork = new Runnable()
  {
   public void run()
   {
    doThreadSafeWork();
   }
  };
  SwingUtilities.invokeLater(callDoThreadSafeWork);
 }
}
```

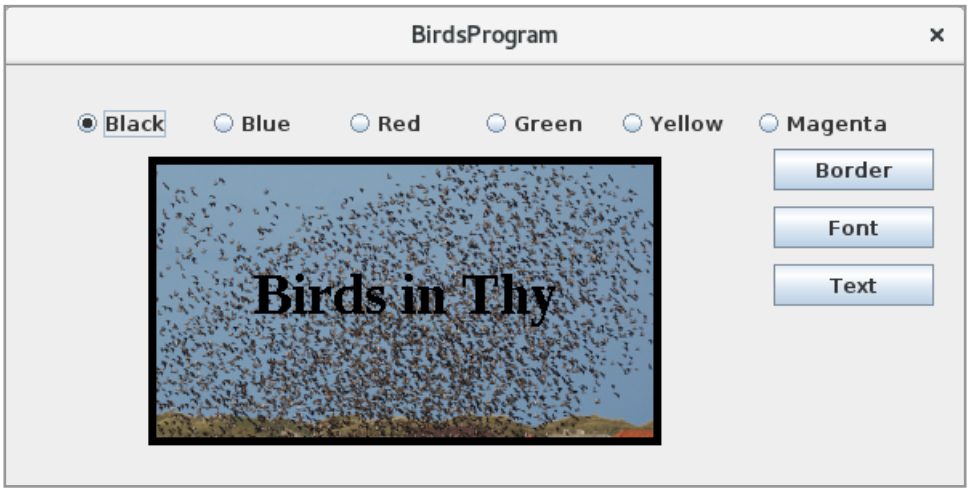
Here it is first tested whether the code is performed by the event dispatching thread and if not, is defined a runnable object that performs the method and it is then sent to the event dispatching thread.

When *SwingUtilities* receives a *Runnable* object by a call of *invokeLater()*, it is immediately forwarded to a method *postRunnable()* in the *SystemEventQueueUtilities* class. If instead, the object is received by calling *invokeAndWait()*, it is first tested that the current thread is not the event dispatching the thread and, if necessary, an exception is raised. Otherwise, an object is created that is used as a lock for a critical region that contains two statements, where it first sends the object to *postRunnable()* while the other is a *wait()* on the lock. The result is that the calling thread is waiting for a *notify()*.

2.4 RENDERING OF COMPONENTS

When the machine displays a component on the screen, it means that the component must be drawn and it is said that Swing is rendering the component. If the component changes at a later time – for example if a button is clicked – it must be drawn again. Generally a component is drawn by performing the method *paintComponent()*, and is it a custom component where it is the programmer that is in charge of drawing the component, you must override this method and the override must always start with the statement *super.paintComponent()*. In fact, it's not simple as much can happen with a component, and for example the component may be fully or partially covered by another component.

As an example, I will show a simple custom component. It should be immediately said that a custom component often is written in a different way than the following example shows, but I will return to it later. If you open the program *BirdsProgram*, it displays the following window:



Here the picture is a component as all the other components in the window. It's a very simple component that has a fixed size and shows an image with which has a frame been drawn and has a text been drawn on top of the picture. The component has properties so you can determine the width of the frame and change the text. You can also change the font in which the text is drawn, and finally you can change the color of the frame and the text. If you click on the image, the component raises an *ActionEvent* that registered listeners can capture. The code af the component is as follows:

```
package birdsprogram;

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

public class TheBirds extends JComponent
{
 private static ImageIcon birds =
  createImageIcon("/birdsprogram/birds.jpg", 320, 180);
 private float borderWidth = 5;
 private String text = "Birds in Thy";

 public TheBirds()
 {
  setBackground((Color.black));
  setFont(new Font("Liberation Serif", Font.BOLD, 36));
  addMouseListener(new ClickHandler());
 }
```

```
@Override
public void paintComponent(Graphics g)
{
 super.paintComponent(g);
 Graphics2D g2d = (Graphics2D)g;
 g2d.drawImage(birds.getImage(), 0, 0, this);
 float width = borderWidth / 2;
 Rectangle2D rect = new Rectangle2D.Double(width, width,
  320 - borderWidth, 180 - borderWidth);
 g2d.setPaint(getBackground());
 g2d.setStroke(new BasicStroke(borderWidth));
 g2d.draw(rect);
 FontMetrics fm = g2d.getFontMetrics();
 int w = fm.stringWidth(text);
 int h = fm.getAscent();
 g.drawString(text, 160 - (w / 2), 90 + (h / 4));
}

public float getBorderWidth()
{
 return borderWidth;
}

public void setBorderWidth(float borderWidth)
{
 this.borderWidth = borderWidth;
 repaint();
}

public String getText()
{
 return text;
}

public void setText(String text)
{
 this.text = text;
 repaint();
}

@Override
public void setFont(Font font)
{
 super.setFont(font);
}
```

```
@Override
public void setBackground(Color color)
{
 super.setBackground(color);
}

@Override
public Dimension getPreferredSize()
{
 return new Dimension(320, 180);
}

@Override
public Dimension getMinimumSize()
{
 return getPreferredSize();
}

@Override
public Dimension getMaximumSize()
{
 return getPreferredSize();
}
```

```
 @Override
 public void setPreferredSize(Dimension dimension)
 {
  throw new UnsupportedOperationException();
 }

 @Override
 public void setMinimumSize(Dimension dimension)
 {
  throw new UnsupportedOperationException();
 }

 @Override
 public void setMaximumSize(Dimension dimension)
 {
  throw new UnsupportedOperationException();
 }

 public void addActionListener(ActionListener listener)
 {
  listenerList.add(ActionListener.class, listener);
 }

 public void removeActionListener(ActionListener listener)
 {
  listenerList.remove(ActionListener.class, listener);
 }

 private class ClickHandler extends MouseAdapter
 {
  public void mouseClicked(MouseEvent e)
  {
   ActionListener[] listeners = listenerList.getListeners(ActionListener.class);
   for (ActionListener listener : listeners)
    listener.actionPerformed(
     new ActionEvent(TheBirds.this, ActionEvent.ACTION_PERFORMED, ""));
  }
 }

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

Firstly, it should be noted that the class inherits *JComponent* and is therefore a component. This means, among other things, that the component has all the functionality that a *JComponent* provides. The class defines a static variable that refers to the image. You should note that the class uses the method *createImageIcon()* to load the image and gives it the size 320 × 180, which is also the size of the component. The class also defines two instance variables, where the first defines the width of the frame while the other is the text to be drawn on the image.

The constructor defines the background color (to black). Note that it is possible because the class *TheBirds* is a *JComponent* and therefore has a *setBackground()* method. The background color is used to the frame and to the text. Similarly, the constructor defines the font, which is the font used to draw the text. Finally, the constructor associates an event handler to the component so that it catches click with the mouse.

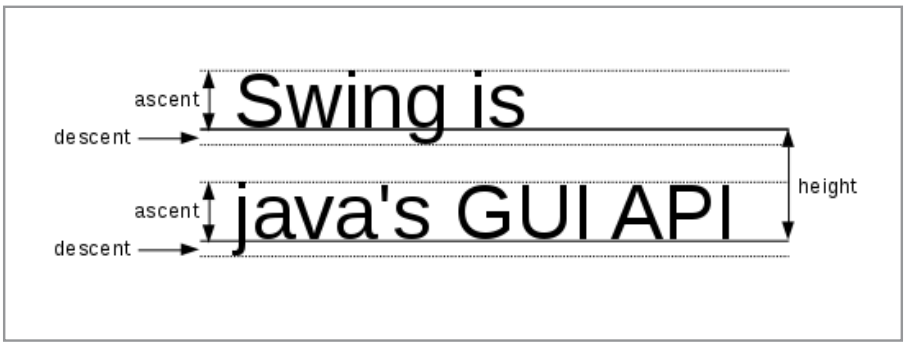
Then there is the method *paintComponent()*, which is the method that must be performed when it is necessary to draw the component, and that is when the window opens and the component is displayed. You should note that the method is never called explicit, but is called by the runtime system when it is necessary to draw the window. The method has a parameter of the type *Graphics*, which is an abstract base class that defines how components can be drawen on different devices such as a screen. One can think of a *Graphics* object as a canvas where a component can draw and the class provides a variety of drawing tools available. In addition, the class defines properties in the form of font and color, cutting area and a coordinate system. The upper left corner is (0,0) and positive × values are oriented to the right while positive y values are oriented downwards. The *paintComponent()* method starts by calling the base class' *paintComponent()*, and then performing a type of cast of the parameter to a *Graphics2D* object. The *Graphics2D* class is derived from *Graphics* (and has been introduced along with Swing after the introduction of Java 2). Improved drawing tools are available, and therefore, in practice, you always performs this type of cast. An example of a drawing tool is *drawImage()*, and the third statement uses this method to draw the image. The next five statements defines and draw the frame. It involves telling what color to draw with which pen (how thick a line) should be drawn with and finally defining the rectangle that defines the frame. Drawing geometric figures are treats detailed in the next book, but it requires some calculations to get the frame drawn within the component's physical boundary at the 320 × 180 pixels. The last statements in *paintComponent()* are used to draw the text, and the challenge is to get the text drawn in the middle of the component.

The rest of the code concerns primarily to define accessor methods for properties. First, there are the two instance variables *borderWidth* and *text*. Here you should note the two set methods end with *repaint()*. This statement forces the component to become redrawen. The method fires an event that is inserted into the event queue, and the event dispatch thread will then redraw the component when time is. There are also overrides of the two methods *setFont()* and *setBackground()* which do nothing but call the corresponding method in the base class. Note that no *repaint()* is required here, as the basic class takes care of it. Finally, I also override the methods *getPreferredSize()*, *getMinimumSize()* and *getMaximumSize()*, so they return a fixed size. It is for the sake of layout managers, as they often use these methods. Since the size is fixed, it does not make sense to use the corresponding *set* methods (they would not have any effect), and therefore they are overstyred so that they raise an exception. Alternatively, one could instead overriding methods as empty methods.

Back there is the event management, which must necessarily implement the methods *addActionListener()* and *removeActionListener()*. Here you should note that *listenerList* is defined protected in the base class and can thus can be referred to from the derivative class *TheBirds*. Also note how *listenerList* is used in the *ClickHandler* class to send notifications to any listeners.

As for the main application that opens the window above, I will not mention it here as it does not add anything new, but I can mention that it has a *BorderLayout* and shows *TheBirds* component encapsulated in a *FlowLayout*.

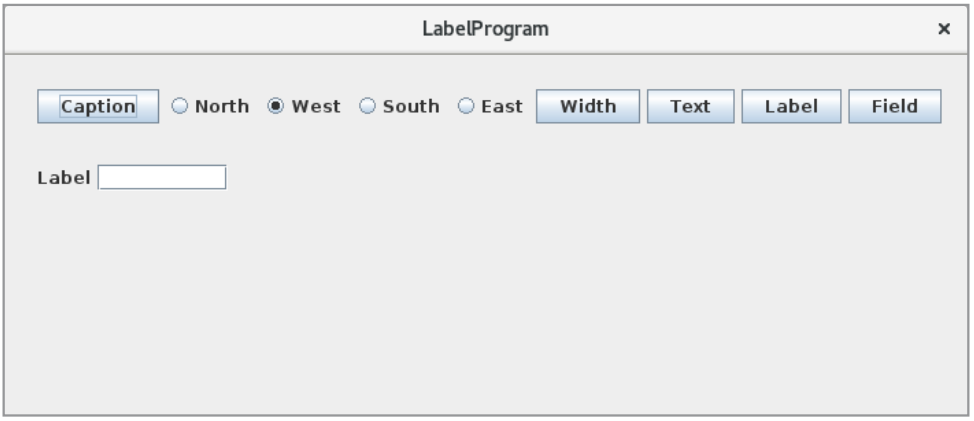
The above component draws a text using a particular font, and in fact fonts are quite complex. Considering the figure below, it should illustrate two lines drawn with a particular font. The fully drawn lines are the *baseline*, and when *drawString()* draws a text, the y-coordinates refer to the baseline. The part above the baseline is called *ascent*, while the part below the baseline is called *descent*. Between lines there is a distance and *height* is the distance between two baselines.



The above *paintComponent()* uses the class *FontMetrics*, which generally returns a number of information about the current font. The class also has a method called *stringWidth()*, which returns the length of a string measured relative to the current font. In the example above, these values are used to draw the text centered in the component.

The key to validating and rendering components is performed by a service called *RepaintManager*. It is responsible for sending events to the event dispatch queue, which is done using two methods *repaint()* and *revalidate()* that encapsulate events in runnable objects and send them to *invokeLater()*. The difference is that *repaint()* tells the component to be redrawn and thus that the *paintComponent()* needs to be executed again, while the *revalidate()* tells the layout manager that the components must be layout again, so that their location and size should be recalculated. Swing components use as default double-buffering, which is a technique in which, instead of drawing directly on the physical device, the drawing is performed in a memory buffer. One can think of it as a memory representation of a drawing like a picture, and after the image is drawen, it is sent to the physical device as a hole, which is very fast and for us people it means that we get a stable image. Otherwise, in complicated screens, one might find that the drawing process takes time.

I want to show another example of a custom component. If you open the program *LabelProgram*, you get the following window:



In the top row there are a number of components consisting of buttons and radio buttons. In the bottom row there is a label and an entry field, and it is a custom component called *InputField*. The components in the upper row are used to define different properties for the custom component:

- the text to the component's label
- where the text should be shown (north, west, south or east)
- the with of the text
- the entry field's preferred size
- the font to text
- the font to the entry field

The component is a *JPanel* with two components: A *JPanel* and a *JTextField*:

```
package labelprogram;

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

public class InputField extends JPanel
{
 public static final String NORTH = BorderLayout.NORTH;
 public static final String SOUTH = BorderLayout.SOUTH;
 public static final String EAST = BorderLayout.EAST;
 public static final String WEST = BorderLayout.WEST;
 private final JTextField field = new JTextField();
 private final JLabel caption = new JLabel();
 private String position;
```

```
public InputField(String text)
{
 this(text, 100, WEST);
}

public InputField(String text, int width)
{
 this(text, width, WEST);
}

public InputField(String text, int width, String position)
{
 setLayout(new BorderLayout(5, 0));
 caption.setHorizontalAlignment(JLabel.RIGHT);
 caption.setText(text);
 add(caption, position);
 add(field);
 this.position = position;
 field.setPreferredSize(new Dimension(width, field.getPreferredSize().height));
}
```

```
public String getCaption()
{
 return caption.getText();
}

public Font getCaptionFont()
{
 return caption.getFont();
}

public Font getTextFont()
{
 return field.getFont();
}

public int getCaptionWidth()
{
 return caption.getPreferredSize().width;
}

public Dimension getFieldSize()
{
 return field.getPreferredSize();
}

public String getText()
{
 return field.getText();
}

public String getPosition()
{
 return position;
}

public void setCaption(String text)
{
 String oldValue = caption.getText();
 caption.setText(text);
 firePropertyChange("Caption", oldValue, text);
}

public void setCaptionFont(Font font)
{
 Font oldValue = caption.getFont();
```

```
 caption.setFont(font);
 revalidate();
 firePropertyChange("CaptionFont", oldValue, font);
}

public void setTextFont(Font font)
{
 Font oldValue = caption.getFont();
 field.setFont(font);
 field.setPreferredSize(new Dimension(field.getPreferredSize().width,
  field.getGraphics().getFontMetrics().getHeight() + 3));
 revalidate();
 firePropertyChange("TextFont", oldValue, font);
}

public void setCaptionWidth(int width)
{
 Integer oldValue = getCaptionWidth();
 caption.setPreferredSize(
  new Dimension(width, caption.getPreferredSize().height));
 revalidate();
 firePropertyChange("CaptionWidth", oldValue, new Integer(width));
}

public void setPosition(String position)
{
 String oldValue = this.position;
 this.position = position;
 remove(caption);
 add(caption, position);
 if (position.equals(WEST)) caption.setHorizontalAlignment(JLabel.RIGHT);
 else caption.setHorizontalAlignment(JLabel.LEFT);
 revalidate();
 firePropertyChange("Position", oldValue, position);
}

public void setFieldSize(Dimension size)
{
 Dimension oldValue = field.getPreferredSize();
 field.setPreferredSize(size);
 revalidate();
 firePropertyChange("FieldSize", oldValue, size);
}

@Override
public Dimension getPreferredSize()
{
```

```
 if (position.equals(EAST) || position.equals(WEST))
 {
  int height = Math.max(caption.getPreferredSize().height,
   field.getPreferredSize().height);
  int width = caption.getPreferredSize().width +
   field.getPreferredSize().width;
  return new Dimension(width, height);
 }
 else
 {
  int height = caption.getPreferredSize().height +
   field.getPreferredSize().height;
  int width = Math.max(caption.getPreferredSize().width,
   field.getPreferredSize().width);
  return new Dimension(width, height);
 }
}

@Override
public Dimension getMinimumSize()
{
 return getPreferredSize();
}
```

```
 @Override
 public Dimension getMaximumSize()
 {
  return getPreferredSize();
 }

 @Override
 public void setPreferredSize(Dimension size)
 {
  throw new UnsupportedOperationException();
 }

 @Override
 public void setMinimumSize(Dimension size)
 {
  throw new UnsupportedOperationException();
 }

 @Override
 public void setMaximumSize(Dimension size)
 {
  throw new UnsupportedOperationException();
 }

 @Override
 public void addPropertyChangeListener(PropertyChangeListener listener)
 {
  listenerList.add(PropertyChangeListener.class, listener);
 }

 @Override
 public void removePropertyChangeListener(PropertyChangeListener listener)
 {
  listenerList.remove(PropertyChangeListener.class, listener);
 }

 @Override
 protected void firePropertyChange(String name, Object oldValue, Object newValue)
 {
  PropertyChangeEvent event =
   new PropertyChangeEvent(this, name, oldValue, newValue);
  for (PropertyChangeListener listener :
   listenerList.getListeners(PropertyChangeListener.class))
    listener.propertyChange(event);
 }
}
```

The code is comprehensive, but in principle simple, and you should notice that it requires some code to write a good custom component – and more than what this example shows. The class inherits a *JPanel* with a *BorderLayout*, where there is a *JLabel NORTH*, *WEST*, *SOUTH*, or *EAST*. *CENTER* is a *JTextField*. The component implements seven properties, but only one is defined as a variable in the class *InputField*. The others are represented as properties pertaining to the two components that the custom component consists of. You should note that the methods of the component's properties performs a *revalidate()* as changes in component's properties mean that the label or input field changes the size and possibly also the location. Also note the *setPosition()* method, which means that the label component must be placed elsewhere. It requires that it is first is removed from the container and then placed the right place. Also note that the set methods fire a *PropertyChangeEvent*. The class should therefore have methods so listeners can sign up for *PropertyChange* notifications and possibly calculates new locations.

One of the challenges with custom components is the size where *getPreferredSize()* returns a value, so layout managers treat the components properly. In this case, the size depends of the two internal components, and you must note how the *getPreferredSize()* method is implemented.

Below is the code of the window that uses the component:

```
package labelprogram;

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.beans.*;

public class MainView extends JFrame implements PropertyChangeListener
{
 private InputField component = new InputField("Label");

 public MainView()
 {
  super("LabelProgram");
  setSize(700, 300);
  setLocationRelativeTo(null);
  createView();
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
  component.addPropertyChangeListener(this);
 }
```

```
private void createView()
{
 JPanel panel = new JPanel(new BorderLayout(0, 20));
 panel.setBorder(new EmptyBorder(20, 20, 20, 20));
 panel.add(top(), BorderLayout.NORTH);
 panel.add(wrap(component));
 add(panel);
}

private JPanel top()
{
 JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
 panel.add(createButton("Caption", e -> { String text =
  JOptionPane.showInputDialog(MainView.this, component.getCaption());
    if (text != null) component.setCaption(text); }));
 ButtonGroup group = new ButtonGroup();
 panel.add(createRadio("North", group, false,
  e -> component.setPosition(InputField.NORTH)));
 panel.add(createRadio("West", group, true,
  e -> component.setPosition(InputField.WEST)));
 panel.add(createRadio("South", group, false,
  e -> component.setPosition(InputField.SOUTH)));
 panel.add(createRadio("East", group, false,
```

```
    e -> component.setPosition(InputField.EAST)));
   panel.add(createButton("Width", e -> { String text =
    JOptionPane.showInputDialog(MainView.this, "Caption height");
    if (text != null) { int width = Integer.parseInt(text);
    component.setCaptionWidth(width); })));
   panel.add(createButton("Text", e -> { String text1 =
    JOptionPane.showInputDialog(MainView.this, "Width"); String text2 =
    JOptionPane.showInputDialog(MainView.this, "Height");
    if (text1 != null && text2 != null) { int width = Integer.parseInt(text1);
    int height = Integer.parseInt(text2);
    component.setFieldSize(new Dimension(width, height)); })));
   panel.add(createButton("Label", e -> { new FontView(new FontListener()
{
    public void fontChanged(FontEvent e) {
    component.setCaptionFont(e.getFont());} }); }));
   panel.add(createButton("Field", e -> { new FontView(new FontListener()
{
    public void fontChanged(FontEvent e)
    { component.setTextFont(e.getFont());} }); }));
   return panel;
  }

  private JRadioButton createRadio(String text, ButtonGroup group,
   boolean checked, ActionListener listener)
  {
   JRadioButton cmd = new JRadioButton(text);
   cmd.setSelected(checked);
   cmd.addActionListener(listener);
   group.add(cmd);
   return cmd;
  }

  private JButton createButton(String text, ActionListener listener)
  {
   JButton cmd = new JButton(text);
   cmd.addActionListener(listener);
   return cmd;
  }

  private JPanel wrap(JComponent component)
  {
   JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
   panel.add(component);
   return panel;
  }
```

```
  @Override
  public void propertyChange(PropertyChangeEvent e)
  {
   JOptionPane.showMessageDialog(this, e.getPropertyName() + "\nFrom: " +
    e.getOldValue() + "\nTo: " + e.getNewValue());
  }
 }
```

Here's also not much to explain, but you should note that the class implements the interface *PropertyChangeListener*, as it should receive notifications regarding changes to properties on the object *component*. This means that the class must implement the method *propertyChange()*, which in this case opens a message box that shows the change. For practical purposes, of course, it does not matter much, and the goal is also to show you how to capture changes for a component's properties.

The class (the program) uses a simple dialog box to define a font. It's a very simple dialog box (and I do not want to display the code here), but problem 1 below is about improving the program at that point.

2.5 FOCUS AND THE KEYBOARD

Swing components are placed in a container and one of the components has focus, so that events related to the keyboard are sent to this component. The general focus cycle is from left to right and from top to bottom, and you switch focus with TAB and CTRL-TAB, and if you want to change focus in the opposite direction, use SHIFT-TAB and CTRL-SHIFT-TAB. Focus is controlled using an instance of the *FocusManager* class. Viewed from the programmer, there are not so many challenges in it, but a component sends an event when it gets focus and again when it loses focus:

1. *focusGained(FocusEvent e)* when the component gets focus
2. *focusLost(FocusEvent e)* when the component loses the focus

When a component has foccus and a key is pressed on the keyboard, the component (depending on the component in question) can send a *KeyEvent*. These events can be captured by associating a *KeyListener*. Unlike other events, a *KeyEvent* is sent, possibly before the corresponding operation is performed. When a key is pressed, three *KeyEvent*s are generally fired:

1. *KEY_PRESSED* that occurs when a key is pressed. The key is specified with a *keyCode* property, and *getKeyCode()* can get a virtual code representing the key, such as *KeyEvent.VK_ENTER*. You should be aware that combination keys such as *CTRL-C* fires two *KEY_PRESSED* events, and the virtual codes are *KeyEvent.VK_CTRL* and *KeyEvent.VK_C*. A *KeyEvent* also has a property *keyChar*, which contains the unicode code.
2. *KEY_RELEASED* are identical to *KEY_PRESSED* events and firing when the key is released, but they are not fired as often as *KEY_PRESSED* events.
3. *KEY_TYPED* are events that are firing between *KEY_PRESSED* and *KEY_RELEASED*, but no *keyCode* is attached. These events are not fired for keys that do not have a unicode representation such as *PAGE UP* and *PRINT SCREEN*.

Holding a key down (for keys with a unicode representation) briefly generates sequences of *KEY_PRESSED* and *KEY_TYPED* events.

Each KeyEvent has associated so-called modifiers, which indicate the state of *SHIFT*, *CTRL*, *ALT* and *META* keys at the time of pressing a key. It is an *int*, which is a bitwise *OR* of the following constants

- *InputEvent.SHIFT_MASK*
- *InputEvent.CTRL_MASK*
- *InputEvent.ALT_MASK*
- *InputEvent.ALT_GRAPH_MASK*
- *InputEvent.META_MASK*
- *InputEvent.BUTTON1_MASK*
- *InputEvent.BUTTON2_MASK*
- *InputEvent.BUTTON3_MASK*

The value can be determined by *getModifiers()* and you can test for the single keys with *isShiftDown()*, *isControlDown()*, *isAltDown()* and *isMetaDown()*.

It is thus possible to control components using of the keyboard by registering *KeyListener* objects. Since it can be very extensive (much to be written), Swing has defined an alternative way of using *KeyStroke* objects, as you also call keyboard accelerators. A *KeyStroke* encapsulates a *keyCode*, a modifier value, and a *boolean* value, which indicates whether it is a key press (*false*) or a key release (*true*). The *KeyStroke* class has 5 static methods that create *KeyStroke* objects:

- *getKeyStroke(char keyChar)*
- *getKeyStroke(int keyCode, int modifiers)*
- *getKeyStroke(int keyCode, int modifiers, boolean onKeyRelease)*
- *getKeyStroke(String representation)*
- *getKeyStroke(KeyEvent anEvent)*

To register a listener for the keyboard for a *JComponent* is used:

registerKeyBoardAction(ActionListener action, KeyStroke stroke, int condition)

The *ActionListener* parameter must define the method *actionPerformed()* to perform the necessary operations corresponding to the *KeyStroke* parameter. The last parameter specifies under what conditions that *KeyStroke* is valid:

1. *JComponent.WHEN_FOCUSED*, the *ActionListener* is called only if the component to which the *KeyStroke* object relates has focus.
2. *JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT*, the component's *ActionListeners* are called only if the component that the *KeyStroke* object concerns is on a higher level (ancestor) than the component that has focus.
3. *JComponent.WHEN_IN_FOCUSED_WINDOW*, the *ActionListener* is only called if the component to which the *KeyStroke* object relates is in a window that has focus (*JFrame*, *JDialog*, *JWindow*).

For example, if you want to define an *ActionListener* for ALT-H, regardless of which component in a *JFrame* with the name *frame* that has focus, you can write something like the following:

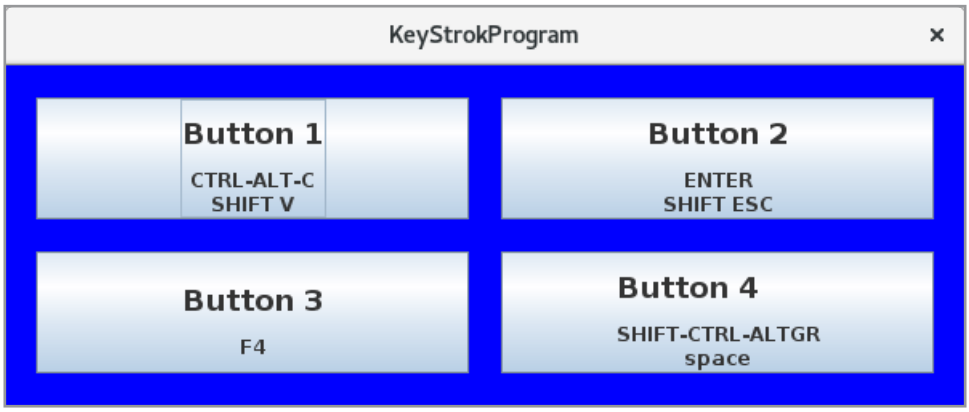
```
KeyStroke keyStroke =
KeyStroke.getKeyStroke(KeyEvent.VK_H, InputEvent.ALT_MASK, false);
frame.getRootPane().registerKeyBoardAction(listener, keyStroke,
 JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
```

where *listener* is the name of the actual *ActionListener*.

Each *JComponent* maintains a *Hashtable* with all *KeyStrokes* as client properties. When a *KeyStroke* is registered with *registerKeyboardAction()*, it is added to this data structure. Only one *ActionListener* for each *KeyStroke* can be registered and if there is already an *ActionListener* for a particular *KeyStroke*, a new one overwrites the previous one. You can determine the current *KeyStroke* bindings using the *getRegisteredKeyStrokes()* method and remove all bindings with *resetKeyboardActions()*.

By an *Action*, you basically understand an instance of an *ActionListener* with *Hashtable* bound properties, in fact, similarly to client properties in *JComponent*. Typically, you use *Action* objects to record keyboard actions.

As a small example, the program *KeyStrokeProgram* opens the following window:



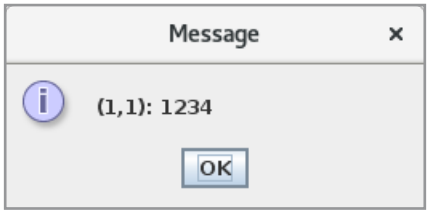
The window has four buttons. For each of the two top buttons, two shortcuts are attached, while each of the two lower ones has attached one shortcut. Clicking one of the buttons or pressing a shortcut key opens a message box. The difference is that for the top two buttons, shortcut keys are assigned as *WHEN_FOCUSED*, and the buttons must therefore have focus for the keys to work. For the two bottom buttons, the shortcut keys are assigned as *WHEN_IN_FOCUSED_WINDOW*, and they therefore work when only the window or parent component has focus. In order to illustrate it, clicking the mouse gives the panel that contains the buttons, focus, and if you do, the background of the panel the background becomes red. The window's code is as shown below:

```
package keystrokeprogram;

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

public class MainView extends JFrame
{
 private Action actionListener = new AbstractAction() {
  public void actionPerformed(ActionEvent e) {
   JButton source = (JButton)e.getSource();
   JOptionPane.showMessageDialog(MainView.this, source.getText());
  }
 };

 public MainView()
 {
  setSize(600, 250);
  setLocationRelativeTo(null);
  createView();
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new GridLayout(2, 2, 20, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20,20));
  panel.setBackground(Color.blue);
  JButton cmd1 = new JButton(
   "<html><h2> Buttom 1</h2><center>CTRL-ALT-C<br/>SHIFT V</center></html>");
  JButton cmd2 = new JButton(
   "<html><h2> Buttom 2</h2><center>ENTER<br/>SHIFT ESC</center></html>");
  JButton cmd3 =
   new JButton("<html><h2>Buttom 3</h2><center>F4</center></html>");
  JButton cmd4 = new JButton("<html><h2> Buttom 4</h2>
   <center>SHIFT-CTRL-ALTGR<br/>space</center></html>");
  assignKey(cmd1, JComponent.WHEN_FOCUSED, "CRTL-ALT-C-KEY",
   KeyStroke.getKeyStroke("control alt C"));
  assignKey(cmd1, JComponent.WHEN_FOCUSED, "SHIFT-V-KEY",
   KeyStroke.getKeyStroke("shift V"));
  assignKey(cmd2, JComponent.WHEN_FOCUSED, "ENTER-KEY",
   KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0, true));
  assignKey(cmd2, JComponent.WHEN_FOCUSED, "SHIFT-ESC-KEY",
   KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, InputEvent.SHIFT_MASK, false));
```

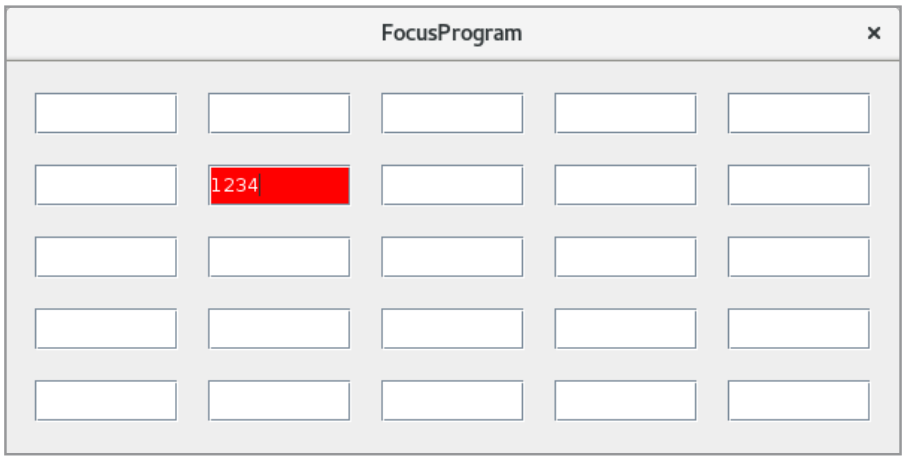
```
    assignKey(cmd3, JComponent.WHEN_IN_FOCUSED_WINDOW, "F4-KEY",
     KeyStroke.getKeyStroke(KeyEvent.VK_F4, 0));
    assignKey(cmd4, JComponent.WHEN_IN_FOCUSED_WINDOW, "SPACE-KEY",
     KeyStroke.getKeyStroke(KeyEvent.VK_SPACE, InputEvent.SHIFT_MASK |
     InputEvent.CTRL_MASK | InputEvent.ALT_GRAPH_MASK, true));
    panel.add(cmd1);
    panel.add(cmd2);
    panel.add(cmd3);
    panel.add(cmd4);
    panel.addMouseListener(new MouseAdapter() {
     public void mouseClicked(MouseEvent e) { panel.requestFocus(); }
    });
    panel.addFocusListener(new FocusListener() {
     public void focusGained(FocusEvent e) { panel.setBackground(Color.red); }
     public void focusLost(FocusEvent e) { panel.setBackground(Color.blue); }
    });
    add(panel);
  }

  private void assignKey(JComponent component, int condition,
   String name, KeyStroke keyStroke)
  {
   component.getInputMap(condition).put(keyStroke, name);
   component.getActionMap().put(name, actionListener);
  }
}
```

EXERCISE 2

You should write a program called *FocusProgram* that opens a Window as shown below. The window has 25 input fields, and the field with focus is shown with a red background and a white text. Fields without focus must have white background af black text. If must de possible to move from field to field by the arrow keys, and if a field has focus and you type *Enter* the program should opens a message box:



that show indexes for the row and column and the value entered in the field.



To solve the exercise the input fields shoud not be *JTextField* components, but a component extended from *JTextField*:

```
class InputField extends JTextField implements FocusListener, KeyListener
{
 private int row;
 private int col;

 public InputField(int row, int col)
 {
  …
 }

 public void keyReleased(KeyEvent e)
 {
 }

 public void keyPressed(KeyEvent e)
 {
  switch (e.getKeyCode())
  {
   case KeyEvent.VK_LEFT:
     moveLeft();
     e.consume();
     break;
  }
 }

 public void keyTyped(KeyEvent e)
 {
 }

 private void moveLeft()
 {
  ….
 }
}
```

that is an inner class.

PROBLEM 1

Start by creating a copy of the program *LabelProgram*. The program has a dialog box called *FontView* that can be used to select a font. It's a very simple dialog box and should be improved to increase user friendliness. You must write a component that you can call *FontChooser*, which can be used to select the parameters of a font. For example, you can write a class *FontChooser*, which has an *openDialog()* method, which opens a dialog box for selecting a font, and the class can then have a method that returns the selected font if the *OK* button is clicked.

Once you've written the class and tested it, delete the *FontView* class from the project and instead use your new class.

When you think your *FontChooser* works properly, take a copy of the latest version of the *PaLib* library (the version from the book Java 6) and move the *FontChooser* class to the library. The project *LabelProgram* may have a reference to *PaLib*. If necessary, remove this reference and reference the new version of *PaLib*. Remove the *FontChooser* class from the project and try the program again, so it now uses the class in *PaLib*.

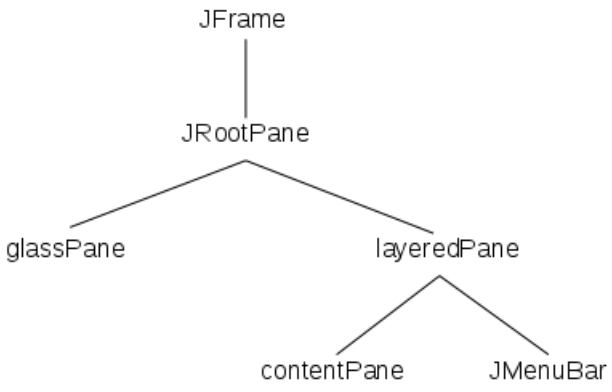
3 LAYOUT

A window consists of panels in which to place components. Viewed from the programmer, a window is a *JFrame*, *JDialog* or a *JWindow* (a window without a title bar), all of which have a single panel of components. There is a special panel called *rootPane*, which has the type of *JRootPane* and which rarely needs to be referred to. It contains two panels, called respectively *layeredPane* and *glassPane*. Here is the last a *JPanel* that general is invisible (transparent), but can be used to place something in a window that is located above all. The other has the type *JLayeredPane* and is also a panel that is not directly referenced, but it has two panels, where the first one is special and is called *JMenuBar* and is used for the window menu while the other is a *JPanel* and is called *contentPane* and is for the other contents of the window (see below). It is therefore for in the window's *contentPane* that you want to add components, but generally you do not have to think about it, because if you (for example, in the constructor of a *JFrame* or a method called from the constructor write something like the following:

```
setLayout(new FlowLayout);
add(component);
```

it automatically refers to the window's *contentPane* – at least today, but previously it was necessary to write

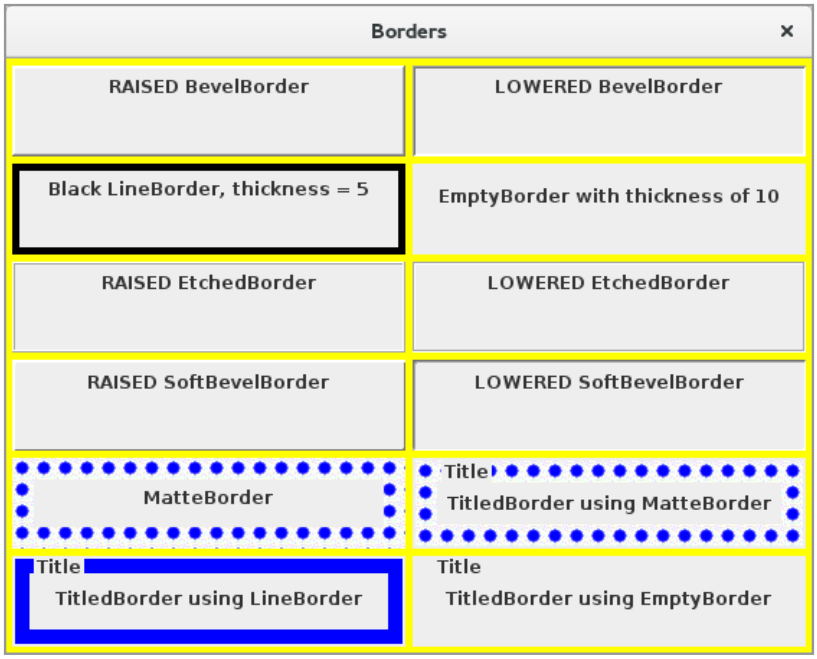
```
getContentPane().setLayout(new FlowLayout);
getContentPane().add(component);
```



A panel has a layout manager that determines how components are placed in the panel. By default, a *JFrame* and a *JDialog* have a *BorderLayout*, while a *JPanel* has a *FlowLayout*.

In addition to a layout manager, a *JPanel* may have attached a *Border*, which I have used many times in the form of an *EmtyBorder*. Swing defines many options for borders. Opening the program *BorderProgram* gives you a window with 12 *JLabel* components organized in a *GridLayout* with 6 rows and 2 columns.

Each label has a border, and they differ according to the type and the parameters with which the *Border* object is created. The code is shown below and there is not much to explain, but you should observe how the different *Border* objects are created:




```
package borderprogram;

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

public class MainView extends JFrame
{
 private static ImageIcon donnert =
  palib.gui.Tools.createImageIcon("/swing04/donnert.gif", 15, 15);

 public MainView()
 {
  setTitle("Borders");
  getContentPane().setBackground(Color.yellow);
  ((JPanel)getContentPane()).setBorder(new EmptyBorder(5, 5, 5, 5));
  setLayout(new GridLayout(6, 2, 5, 5));
```

```
 add(createPanel("RAISED BevelBorder", new BevelBorder(BevelBorder.RAISED)));
 add(createPanel("LOWERED BevelBorder", new BevelBorder(BevelBorder.LOWERED)));
 add(createPanel(
  "Black LineBorder, thickness = 5", new LineBorder(Color.black, 5)));
 add(createPanel("EmptyBorder with thickness of 10",
 new EmptyBorder(10,10,10,10)));
 add(createPanel("RAISED EtchedBorder", new EtchedBorder(EtchedBorder.RAISED)));
 add(createPanel("LOWERED EtchedBorder",
  new EtchedBorder(EtchedBorder.LOWERED)));
 add(createPanel("RAISED SoftBevelBorder",
  new SoftBevelBorder(SoftBevelBorder.RAISED)));
 add(createPanel("LOWERED SoftBevelBorder",
  new SoftBevelBorder(SoftBevelBorder.LOWERED)));
 add(createPanel("MatteBorder", new MatteBorder(donnert)));
 add(createPanel("TitledBorder using MatteBorder",
  new TitledBorder(new MatteBorder(donnert), "Title")));
 add(createPanel("TitledBorder using LineBorder",
  new TitledBorder(new LineBorder(Color.blue, 10), "Title")));
 add(createPanel("TitledBorder using EmptyBorder",
  new TitledBorder(new EmptyBorder(10,10, 10, 10), "Title")));
 pack();
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 setVisible(true);
}
```

```
 private JPanel createPanel(String text, Border border)
 {
  JPanel panel = new JPanel();
  panel.setBorder(border);
  panel.add(new JLabel(text));
  return panel;
 }
}
```

You should note that when writing

```
setLayout(new GridLayout(6, 2, 5, 5));
```

it refers to the window's *contentPane*, and the same applies when writing

```
add(createPanel("RAISED BevelBorder", new BevelBorder(BevelBorder.RAISED)));
```

On the other hand, it is in the statement (which sets the background color for the window's content pane):

```
getContentPane().setBackground(Color.yellow);
```

necessary to refer to the window's content pane, otherwise you set the background color of the window's *rootpane*, which has no effect. Something similar applies to the statement

```
((JPanel)getContentPane()).setBorder(new EmptyBorder(5, 5, 5, 5));
```

there define an empty border around the default layout manager in the window's content pane.

Note that the window is not assigned any size with *setSize()*. Instead, the constructor calls a method:

```
pack();
```

This means that the size of the window is adjusted to the preferred size of the current components, as determined by the 12 label components, and the *GridLayout* place the components into cells of the same size.

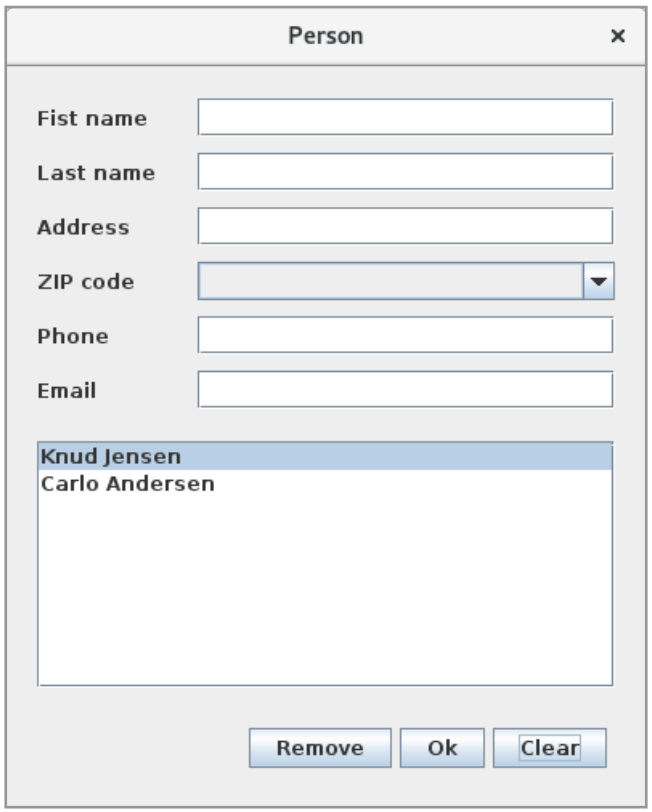
3.1 LAYOUT MANAGERS

Layout managers are treated in the book Java 2, where I have shown how most managers work, and while there are more, I will not associate additional comments with the classic layout managers in this place. With the layout managers discussed in Java 2, you can actually design any layout for a window – although it is not necessarily simple. For that reason or maybe because you have special needs, you may be interested in writing your own layout manager, and I will below show a two examples of how to do.

A layout manager is basically a class that implements the interface *LayoutManager*, or a derived interface called *LayoutManager2*. The difference is whether it is a layout manager where you call so-called constraints for the components' location, what you do in, for example, a *BorderLayout* and a *GridBagLayout*. The interface *LayoutManager* defines basically two methods:

- *preferredLayoutSize(Container parent)* that determines the preferred size for the actual container
- *layoutContainer(Container parent)* that layouts the components in the container

If you opens the program *LayoutProgram*, you get the following window:



This is an example of a typical dialog box where you can enter the name and address of a person, and click OK, the person will be added to the list box. In the example, two people have been created. The combo box contains the zip codes. If you double-click a name in the list box, the person's data is inserted into the fields where they can be edited and you can also delete the person. The example should show something about layout, and the data entered will not be saved. The goal is that the class should be simple. In addition to the above window, the program has two classes where *Person* represents a person with a property for each data item that can be entered. In addition, there is the class *Zipcodes*, which retrieve the zip codes from the database.

The main window does not contain anything new to what has been said about Swing. Below is the part of the code that creates the window:

```
private void createView()
{
 JPanel panel = new JPanel(new BorderLayout(0, 20));
 panel.setBorder(new EmptyBorder(20, 20, 20, 20));
 panel.add(createTop(), BorderLayout.NORTH);
 panel.add(createBottom(), BorderLayout.SOUTH);
 panel.add(createCenter());
 add(panel);
}
```

```
private JScrollPane createCenter()
{
 JList list = new JList(model);
 list.addMouseListener(new ClickHandler());
 return new JScrollPane(list);
}

private JPanel createTop()
{
 JPanel panel = new JPanel(new GridLayout(6, 1, 0, 10));
 panel.add(createLine("Fist name", txtFirstname));
 panel.add(createLine("Last name", txtLastname));
 panel.add(createLine("Address", txtAddress));
 panel.add(createLine("ZIP code", lstZipcode = createList()));
 panel.add(createLine("Phone.", txtPhone));
 panel.add(createLine("Email", txtEmail));
 return panel;
}

private JPanel createBottom()
{
 JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
 panel.add(createButton("Remove", new RemoveHandler()));
 panel.add(createButton("Ok", new OkHandler()));
 panel.add(createButton("Clear", new ClearHandler()));
 return panel;
}

private JButton createButton(String text, ActionListener listener)
{
 JButton cmd = new JButton(text);
 cmd.addActionListener(listener);
 return cmd;
}

private JPanel createLine(String text, JComponent component)
{
 JPanel panel = new JPanel(new BorderLayout());
 JLabel label = new JLabel(text);
 label.setPreferredSize(new Dimension(100, 22));
 panel.add(label, BorderLayout.WEST);
 panel.add(component);
 return panel;
}
```

You should note that the layout consists of a *BorderLayout*, that *CENTER* has a *JScrollPane* with a list box while *NORTH* and *SOUTH* have a *JPanel*. *SOUTH* is a *JPanel* with a *FlowLayout*, which lay out the three buttons, while *NORTH* is a *JPanel* with a *GridLayout* with 6 rows. Each row contains a *BorderLayout* with a label and component that is either a *JTextField* or a *JComboBox*.

If you look at the panel at the top, it's a typical layout for a dialog box where the user has to enter something. It is a layout that consists of lines that contain a *JLabel* and another component. The situation seems so often that one might consider writing a special layout manager, which could facilitate the design of such a dialog box. That is exactly the subject of the example *LayoutProgam1*.

The example opens the same window, and the code is almost the same. In the main window, the *createLine()* method is removed and the method *createTop()* has been changed to the following:

```
private JPanel createTop()
{
 JPanel panel = new JPanel(new DialogLayout(10, 10));
 panel.add(new JLabel("Firstname"));
 panel.add(txtFirstname);
 panel.add(new JLabel("Lastname"));
 panel.add(txtLastname);
 panel.add(new JLabel("Address"));
 panel.add(txtAddress);
 panel.add(new JLabel("ZIP code"));
 panel.add(lstZipcode = createList());
 panel.add(new JLabel("Phone"));
 panel.add(txtPhone);
 panel.add(new JLabel("Email"));
 panel.add(txtEmail);
 return panel;
}
```

Here, the 12 components are added directly to the panel that has a layout manager of the type *DialogLayout*. This is an example of a custom layout manager. It can be used in a container containing pairs of components of the form

label1, komponent1
label2, komponent2
…

and added to the container in that order, which is an essential condition for this layout manager. The layout manager will then put the components into two columns, where the first column (called the header column) contains all label components, while the other column contains the other fields. The width of the first column is by default the largest width of all label components, but you can specify a fixed size. The width of the second column is the largest width of the fields. The code is as follows:

```
public class DialogLayout implements LayoutManager
{
 private int divider = -1;     // indicats a possible fixed width of the headers
 private int hGap = 10;        // horizontal gap between components
 private int vGap = 5;         // vertical gap between components

 public DialogLayout()
 {
 }
 public DialogLayout(int hGap, int vGap)
 {
  this.hGap = hGap;
  this.vGap = vGap;
 }
```

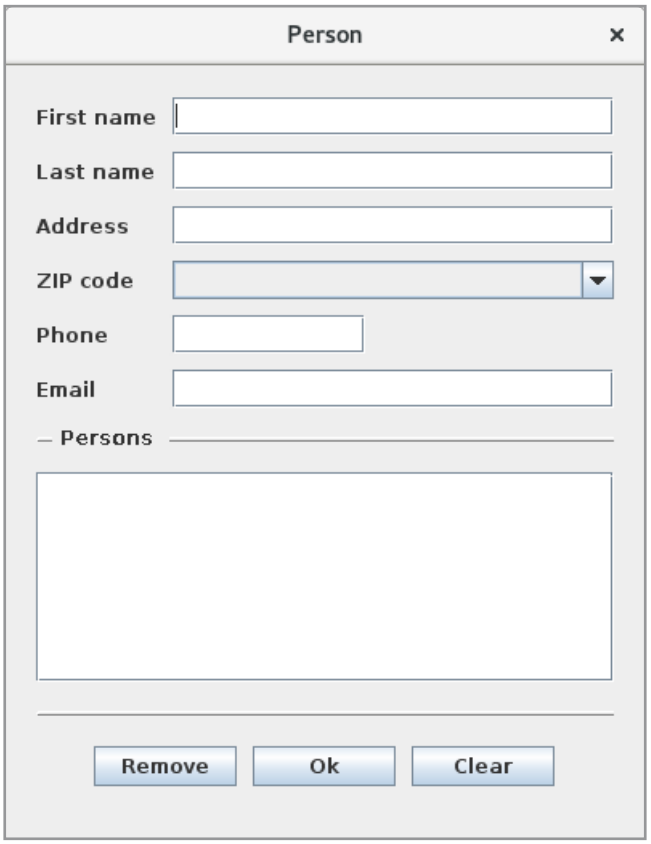
```
public void addLayoutComponent(String name, Component comp)
{
}

public void removeLayoutComponent(Component comp)
{
}

public Dimension preferredLayoutSize(Container parent)
{
 int left = getDivider(parent);
 int w = 0;
 int h = 0;
 for (int k = 1 ; k < parent.getComponentCount(); k += 2)
 {
  Dimension d = parent.getComponent(k).getPreferredSize();
  w = Math.max(w, d.width);
  h += d.height + vGap;
 }
 h -= vGap;
 Insets insets = parent.getInsets();
 return new Dimension(
  left + w + insets.left + insets.right, h + insets.top + insets.bottom);
}

public Dimension minimumLayoutSize(Container parent)
{
 return preferredLayoutSize(parent);
}

public void layoutContainer(Container parent)
{
 int left = getDivider(parent);
 Insets insets = parent.getInsets();
 int w = parent.getWidth() - insets.left - insets.right - left;
 int x = insets.left;
 int y = insets.top;
 for (int k = 1 ; k < parent.getComponentCount(); k += 2)
 {
  Component component = parent.getComponent(k);
  Dimension d = component.getPreferredSize();
  parent.getComponent(k - 1).setBounds(x, y, left - hGap, d.height);
  component.setBounds(x + left, y, w, d.height);
  y += d.height + vGap;
 }
}
```

```
  public int getHGap()
  {
   return hGap;
  }

  public int getVGap()
  {
   return vGap;
  }

  public void setDivider(int divider)
  {
   if (divider > 0) this.divider = divider;
  }

  public int getDivider()
  {
   return divider;
  }

  private int getDivider(Container parent)
  {
   if (divider > 0) return divider;
   int left = 0;
   for (int k = 0; k < parent.getComponentCount(); k += 2)
   {
    Dimension d = parent.getComponent(k).getPreferredSize();
    left = Math.max(left, d.width);
   }
   left += hGap;
   return left;
  }

  public String toString()
  {
   return getClass().getName() + " [hgap=" + hGap + ", vgap=" +
    vGap + ", divider=" + divider + "]";
  }
 }
```

The important thing is to note the methods *preferredLayoutSize()* and *layoutContainer()*. Both methods are simple enough, but it requires some calculations to determine the container's preferred size, and in fact, the same calculations are performed in *layoutContainer()* where the components are placed in the container. You should also note how the individual components are placed with *setBounds()*. Also note the helper method *getDivider()* used to calculate the width of the header column.

PROBLEM 2

Create a copy of the project *LayoutProgram1* as you can call *LayoutProgram2*. The program should opens the window shown belov, which is almost identical to the previous windows except for the lines on both sides of the list box. The task is to write an improved version of the layout manager *DialogLayout*. The idea behind this layout manager is, that there is a need for a dialog box (as shown below), and it can then be created by simply filling components into the container in the correct order. The layout manager should not properly support all components, but many dialogs actually consist only of components as shown in the below example and then it is simple to design the dialog using a *DialogLayout*.



All components should be added to a panel with a *DialogLayout* in a sequence (as in the above example), but the components are placed in groups in the order in which they are added to the container based on the following rules, where there is three grouls:

1. *COMP_LBL* that includes pairs consisting of a *JLabel* and another component – that is in the same way as with the previous version of *DialogLayout*
2. *COMP_PAN* that includes *JPanel*, *JScrollPane* and *DialogSeparator* (a custom component that is the horizontal line shown in then window).
3. *COMP_CMD* that includes *AbstractButton* (*JButton*, *JCheckBox*, *JRadioButton*)

When added components, a group span until the group type is changed. The components in the individual groups are laid out as follows:

1. *COMP_LBL*: The components are laid out in pairs in the samme way as the previous *DialogLayout*
2. *COMP_PAN*: The components are laid out vertically underneath each other
3. *COMP_CMD*: The components are laid out horizontal and ajusted left, centered or right

With respect to the size of the components, the following shall apply:

1. in COMP_LBL the with og the field is the rest of the container if the field's preferred width is less than 10, and else the preferred width is used
2. in COMP_PAN the width is the width of the container
3. in COMP_CMD the components preferred width is used

You must also write a simple custom component *DialogSeparator*, which draws a line across the panel. The purpose is only to have a visual effect between components in a *COMP_BIG* group.

Once the program is finished and works as expected, you can copy the layout manager to the class library *PaLib*. It is recommended as the class has some practical interest. I have called the class *FlexLayout*. Also remember to copy the component *DialogSeparator*.

VARIABLEGRIDLAYOUT

Before I leave this section on layout managers, I will mention a *VariableGridLayout*, that is an older layout manager from Sun, who has been deprecated for many years. However, I have had a lot of pleasure from this manager, so the following. A *VariableGridLayout* is simply a layout manager who inherits *GridLayout*, and it is thus a *GridLayout*, but with the extension that rows and columns do not necessarily have the same height/width. After creating a *VariableGridLayout*, you can specify how much of the total height is to be applied to a particular row, and in the same way you can specify how much of the total width to be used for a particular column.

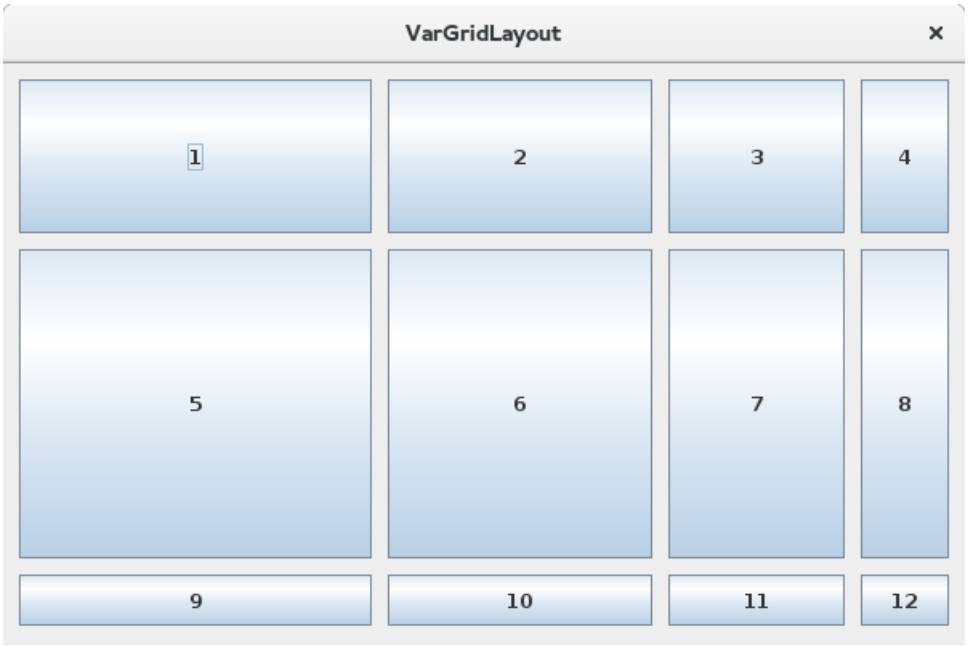
As mentioned, a *VariableGridLayout* is deprecated, and it's no longer part of the Java API. Fortunately, the code is publicly available and I have added it to my library under the name of *VarGridLayout*. The program *VarGridProgram* defines a window with 12 buttons that is laid out using a *VarGridLayout* (the buttons have no function):

```
package vargridprogram;

import javax.swing.*;
import javax.swing.border.*;

import palib.gui.*;
```

```
public class MainView extends JFrame
{
 public MainView()
 {
  super("VarGridLayout");
  setSize(600, 400);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  VarGridLayout layout = new VarGridLayout(3, 4, 10, 10);
  layout.setRowFraction(0, 0.3);
  layout.setRowFraction(1, 0.6);
  layout.setRowFraction(2, 0.1);
  layout.setColFraction(0, 0.4);
  layout.setColFraction(1, 0.3);
  layout.setColFraction(2, 0.2);
  layout.setColFraction(3, 0.1);
  JPanel panel = new JPanel(layout);
```

```
    panel.setBorder(new EmptyBorder(10, 10, 10, 10));
    for (int i = 0; i < 12; ++i) panel.add(new JButton("" + (i + 1)));
    add(panel);
  }
}
```

The only thing to note is how to specify how much a proportion is to be used for rows and columns. If you run the program, you get the following window:

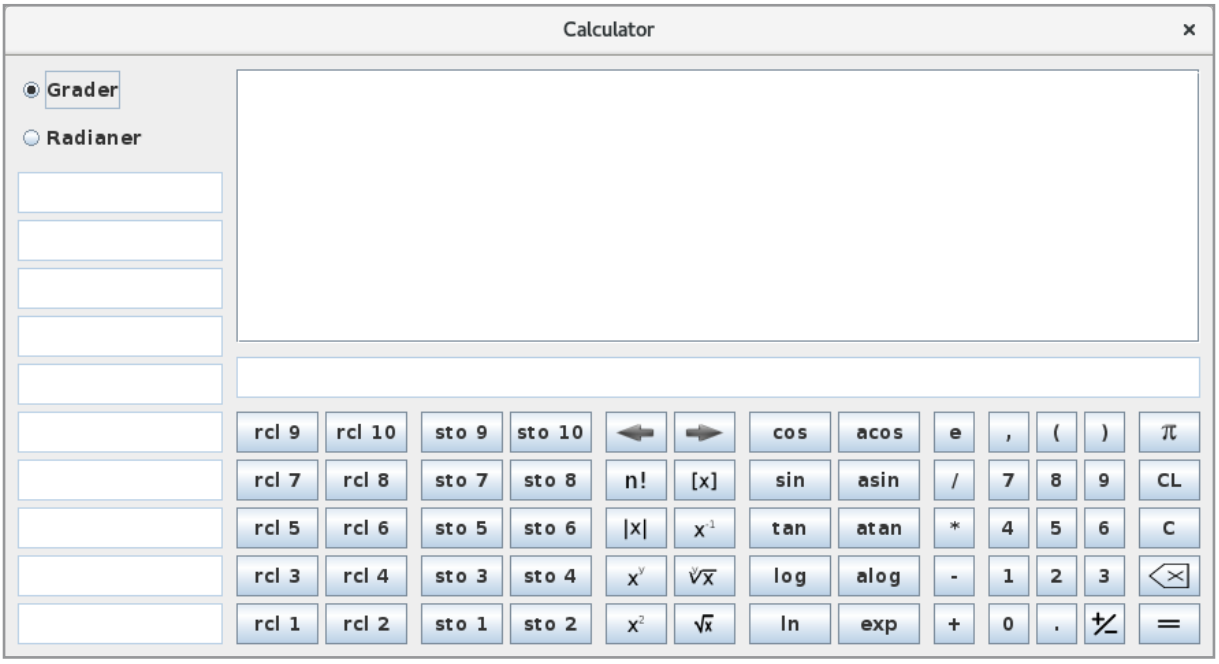


PROBLEM 3

You must write a classical program that simulates a mathematical calculator. The program must open the window shown below. The machine must have 10 registers for intermediate results and the left-hand fields must display the contents of these registers – if there is otherwise a content. The content of the display are stored in a register with a *sto* button and you can insert the content of a register into the display with a *rcl* button. The display is the field above the buttons, while the top field is for a history. The meaning of the buttons should be self explanatory, perhaps not the C and CL buttons, where the first deletes the display while the other resets the entire machine.

The registers, display and history fields are all *JTextField* and *JTextArea* fields, but none of the fields must be editable. That is, they should be defined as not *editable*. You can only insert in the fields using the buttons of the machine. On the other hand, each button must be assigned a shortkey (you decide which ones) so that the machine can be used solely using the keyboard. Some of the buttons show icons, and these icons can be found in the director *icons*.

In addition to the user interface, the biggest challenge is to be able to parse and evaluate a mathematical expression. Here you should be aware that this problem has been solved in the project *Calc* in the book Java 3. Therefore, you can reuse the classes from this project to the actual problem.



4 SWING COMPONENTS

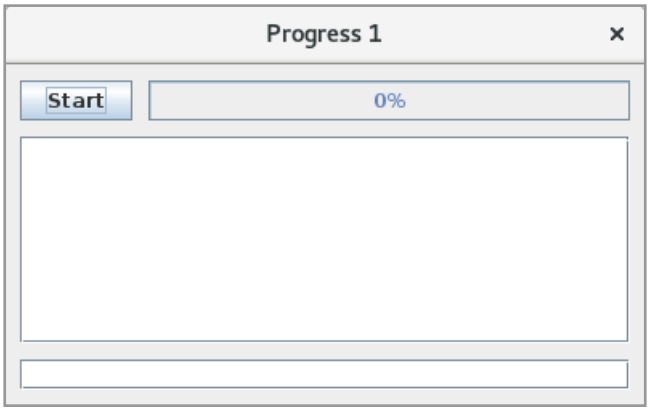
Swing provides many components available, and below are the most mentioned:

JButton
JRadioButton
JComponent
JMenuBar
JRadioButtonMenuItem
JScrollPane
JTextField
JEditorPane
JToogleButton
JTable
JToolBar
JInternalFrame
JSplitPane
JPanel
ButtonGroup
JLabel
JMenuItem
JCheckBoxMenuItem
JScrollBar
JPasswordField
JSpinner
JProgressBar
JTree
JSeparator
JOptionPane
JTabbedPane
JCheckBox
JComboBox
JList
JMenu
JPopupMenu
JTextArea
JTextPane
JSlider
JFormattedTextField
JToolTip
JDesktopPanel
JColorChooser

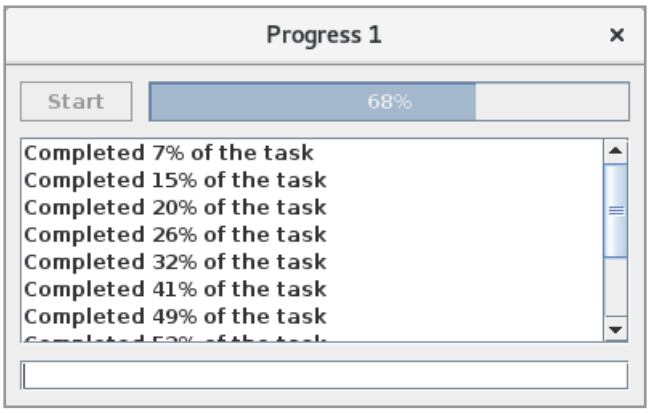
Many of these components have been used and discussed earlier and should not be further processed here, and generally the individual components are used in the same way. Specifically, I would like to remind *JTable*, which is dealt with in the book Java 6. Some of the components are used specifically for text entry, where *JTextField* and *JTextArea* are previously discussed, but others are treated in the chapter *Edit text* later in this book. In this chapter, I would like to look at two of the components: *JProgressBar* and *JTree*.

4.1 JPROGRESSBAR

A progress bar is a component that visually shows progress in a process, for example to download a file from a web server. A progress bar will show the user that something is happening and the program is stopped and waiting. As an example, the following window contains a progress bar



which sits at the top of the window to the right of the button. The window also contains a list box and an input field. If you click the button, it should simulate starting a job that takes time, and the progress bar will then show momentum with the job:



In the lower entry field you can enter text – and nothing else – and the intention is to show that you can work with the window while the job is performed. It is clear that the process must start the job in its own thread, and the thread must then periodically update both the progress bar and list box. In principle, it is simple enough, but when the job is performed in a different thread than the Swing dispathcer thread it is necessary with a few steps to ensure that the user interface is updated correctly.

```
package progressprogram1;

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.beans.*;
import javax.swing.border.*;
public class MainView extends JFrame
{
 private DefaultListModel model = new DefaultListModel();
 private JButton cmdStart;
 private JProgressBar progressBar;
 private JTextField txtField;

 public MainView()
 {
  super("Progress");
  setSize(400, 300);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new BorderLayout(0, 10));
  panel.setBorder(new EmptyBorder(10, 10, 10, 10));
  panel.add(createTop(), BorderLayout.NORTH);
  panel.add(txtField = new JTextField(), BorderLayout.SOUTH);
  panel.add(new JScrollPane(createList()));
  add(panel);
 }
```

```
private JPanel createTop()
{
 JPanel panel = new JPanel(new BorderLayout(10, 0));
 panel.add(cmdStart = createButton(), BorderLayout.WEST);
 panel.add(progressBar = createProgress());
 return panel;
}

private JButton createButton()
{
 JButton cmd = new JButton("Start");
 cmd.addActionListener(new ActionHandler());
 return cmd;
}

private JProgressBar createProgress()
{
 JProgressBar bar = new JProgressBar(0, 100);
 bar.setValue(0);
 bar.setStringPainted(true);
 return bar;
}
```

```
private JList createList()
{
 JList list = new JList(model);
 return list;
}

class ActionHandler implements ActionListener
{
 public void actionPerformed(ActionEvent e)
 {
  cmdStart.setEnabled(false);
  model.clear();
  txtField.setText("");
  setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
  Task task = new Task();
  task.addPropertyChangeListener(new PropertyChangeHandler());
  task.execute();
  txtField.requestFocus();
 }
}

class PropertyChangeHandler implements PropertyChangeListener
{
 public void propertyChange(PropertyChangeEvent e)
 {
  if ("progress" == e.getPropertyName())
  {
   progressBar.setValue((Integer) e.getNewValue());
   model.addElement((String.format(
    "Completed %d%% of the task", ((Task)e.getSource()).getProgress())));
  }
 }
}

class Task extends SwingWorker<Void, Void>
{
 public Void doInBackground()
 {
  Random random = new Random();
  int progress = 0;
  setProgress(0);
  while (progress < 100)
  {
   try
   {
    Thread.sleep(random.nextInt(1000));
   }
```

```
      catch (Exception ex)
      {
      }
      progress += random.nextInt(10);
      setProgress(Math.min(progress, 100));
     }
     return null;
    }

    public void done()
    {
     cmdStart.setEnabled(true);
     setCursor(null);
     model.addElement("Done!");
    }
   }
}
```

As for the window layout is nothing new, but the class creates a *JProgressBar* and adds it to the window. You should notice how the component is created, where you specify a minimum value and a maximum value, and the progress bar has all the time a value within this range. Note especially the statement

```
bar.setStringPainted(true);
```

which means that the progress bar will show the value (the text). The class has three inner classes, two event handlers and the class *Task* representing the job. The class is derived from *SwingWorker*. This class is described in the last chapter of the book, but the aim is that the class must make it easy to update the user interface from another thread. *SwingWorker* is an abstract class, and there are two abstract methods to be implemented. The method *doInBackground()* starts a thread, and in this case the method simulates a work that takes time. The class has a bound property called *progress*, and the class is therefore sending *PropertyChange* notifications to listeners when this property is changed. The method *done()* is another abstract method, which is performed when the thread terminates. You must specifically note that the class *Task* not know the progress bar, and the class job is only intermittently performing the event handler *propertyChange()*.

If you look at the event handler *actionPerformed()*, that is executed when the user clicks the button, so it disables the button, deletes the model (for the list box) and the input field, set a cursor, and then instantiated a *Task* object. Next, the event handler register a listener as a *PropertyChangeHandler* object, after which the job (the thread) is started with the method *execute()*.

The result is that every time the object *Task* performs *setProgress()*, then the *propertyChange()* event handler is performed, but in the dispatcher thread. The result is that the progress bar and the model of the list box is updated.

The program *ProgressProgram2* has the same user interface and works in principle in the same way, but there are two differences. The progress bar is displayed in a different way, and the job does something else. The progress bar shows this time not a text and a percentage indication of how far the job is reached, but it shows an animation with a rectangle that changes from left to right and back again. Sometimes it can be difficult (impossible) to calculate the percentage of a job that is done, and so this progress bar may be preferable. The job is this time is defined by the following class:

```
class Task extends SwingWorker<Void, Void>
{
 private static final int N = 5;

 public Void doInBackground()
 {
  setProgress(0);
  progressBar.setIndeterminate(true);
  long n = Long.MAX_VALUE;
```

```
   for (int i = 0; i < N; ++i, n -= 2)
   {
    while (n > 0 && !isPrime(n)) n -= 2;
    prime = n;
    setProgress(100 * (i + 1) / N);
   }
   return null;
  }

  public void done()
  {
   cmdStart.setEnabled(true);
   progressBar.setIndeterminate(false);
   model.addElement("Done!");
  }

  private boolean isPrime(long n)
  {
   if (n == 2 || n == 3 || n == 5 || n == 7) return true;
   if (n < 11 || n % 2 == 0) return false;
   for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
    if (n % t == 0) return false;
   return true;
  }
}
```

Note first the statement

```
progressBar.setIndeterminate(true);
```

which is the statement that starts the animation. The job is to determine 5 large prime numbers, and the goal is merely to show a job that takes time – a real job that uses the machine's CPU. Otherwise, there is not so much new and the application works in principle in exactly the same way as the first example.

Finally, the example *ProgressProgram3* that performs exactly the same as *ProgrssProgram2*, but the difference is that the *Task* class this time not inherit *SwingWorker*. Instead it implements the interface *Runnable* and the thread must be able to ensure that the updates of the user interface is done in the event dispatcher thread:

```
class Task implements Runnable
{
 private static final int N = 5;
```

```
public void run()
{
 try
 {
  updateState(false);
  updateBar(0);
  progressBar.setIndeterminate(true);
  long n = Long.MAX_VALUE;
  for (int i = 0; i < N; ++i, n -= 2)
  {
   while (n > 0 && !isPrime(n)) n -= 2;
   updatePrime(n);
   updateBar(100 * (i + 1) / N);
  }
  updateState(true);
 }
 catch (Exception ex)
 {
 }
 finally
 {
  progressBar.setIndeterminate(false);
 }
}

private void updateBar(int value)
{
 try
 {
  SwingUtilities.invokeAndWait(() -> progressBar.setValue(value));
 }
 catch (Exception ex)
 {
 }
}

private void updatePrime(long prime)
{
 try
 {
  SwingUtilities.invokeAndWait(() -> model.addElement(prime));
 }
 catch (Exception ex)
 {
 }
}
```

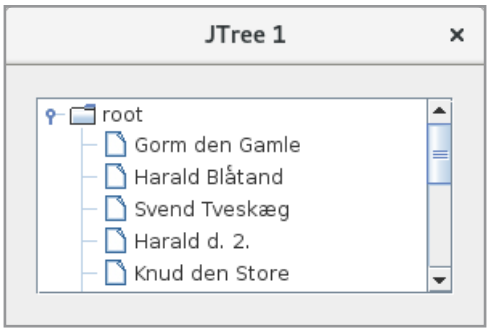
```
 private void updateState(boolean onoff)
 {
  try
  {
   SwingUtilities.invokeAndWait(() -> cmdStart.setEnabled(onoff));
  }
  catch (Exception ex)
  {
  }
 }

 private boolean isPrime(long n)
 {
  if (n == 2 || n == 3 || n == 5 || n == 7) return true;
  if (n < 11 || n % 2 == 0) return false;
  for (long t = 3, m = (long)Math.sqrt(n) + 1; t <= m; t += 2)
   if (n % t == 0) return false;
  return true;
 }
}
```

In principle, this program works in the same way, but it gives the programmer a greater freedom with regard to the use of a progress bar, but conversely, the example also means that there is a lot, that the programmer must take care of.

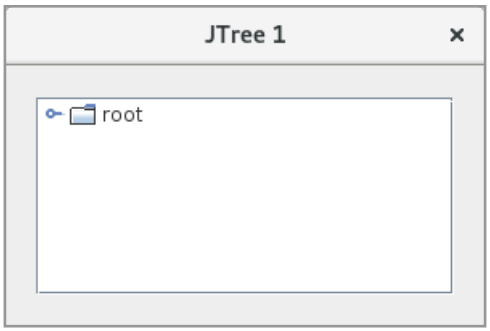
4.2 JTREE

JTree is a component that can visually display hierarchical data, and you actually know the component from a *JFileChooser*, where the component is used to browse the file system. The actual Swing component must visualize a hierarchy, and the data to be visualized comes from a data model, and similar to, for example, the component *JTable*, a *JTree* uses several helper classes, as in this case are found in *javax.swing.tree*. *JTree* is a complex class with many properties and settings, and in the following I will illustrate how the component can be used.

I would like to start with an application that opens the following window:



The window shows a *JTree* in a *JScrollPane*, and the hierarchy (tree) that in this case consists only of a list of names, but such that all names are child nodes under the *root*. If you click on the root of the tree, you can collapse the tree:

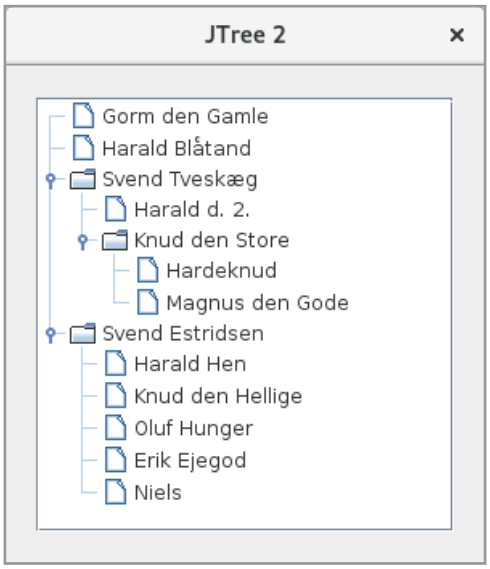


and of course you can expand the tree out again. The code is as shown below:

```
package treeprogram1;

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

public class MainView extends JFrame
{
 private Object[] model = {
  "Gorn denGamle", "Harald Blåtand", "Svend Tveskæg", "Harald d. 2.",
  "Knud den Store", "Hardeknud", "Magnus den Gode", "Svend Estridsen",
  "Harald Hen", "Knud den Hellige", "Oluf Hunger", "Erik Ejegod", "Niels" };

 public MainView()
 {
  super("JTree 1");
  setSize(300, 200);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  JTree tree = new JTree(model);
  tree.setRootVisible(true);
  panel.add(new JScrollPane(tree));
  add(panel);
 }
}
```

In *createView()*, a *JTree* component is created, which is initialized with a data model, which is an array of objects. In fact, the constructor in *JTree* creates a data model, which is a very flat data model, where all nodes are placed in a list below the root. Also note the statement that tells the root to appear. That is, you do not have to show the root if you don't wish. By default, a *JTree* displays a hierarchy of objects, and for each object, the result of the object's *toString()* is displayed.

In this case, the data model was defined as an array of *Object* objects, and the constructor in the class *JTree* will automatically create a model, which is a tree consisting of *TreeNode* objects. There are several constructors and, among other constructors that as parameters have respectively a *Hashtable* and a *Vector*. I will show an example of how to create a *JTree* based on a *Vector*. A *Vector* is basically the same as an *ArrayList*, but an older collection class that has existed since the birth of the language. The class has since been modernized in a generic version, but I will use the original version. It should be said that in general, it is not recommended to use the class *Vector*, but instead to use *ArrayList*, but the constructor in *JTree* only supports *Vector*, so therefore. The class *Vector* is thread safe, and *ArrayList* therefore has a better performance. The example is called *TreeProgram2* and opens the following window:



This time is displayed a hierarchy where the individual branches can be closed and opened by clicking the mouse. Note that the root is not displayed. The program is written as follows:

```
package treeprogram2;

import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class MainView extends JFrame
{
 public MainView()
 {
  super("JTree 2");
  setSize(300, 200);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  JTree tree = new JTree(createVector("", "Gorm den Gamle", "Harald Blåtand",
   createVector("Svend Tveskæg", "Harald d. 2.",
    createVector("Knud den Store", "Hardeknud", "Magnus den Gode")),
```

```
    createVector("Svend Estridsen", "Harald Hen", "Knud den Hellige",
      "Oluf Hunger", "Erik Ejegod", "Niels")));
   tree.setRootVisible(false);
   panel.add(new JScrollPane(tree));
   add(panel);
  }

  private Vector createVector(String name, Object … objects)
  {
   Vector v =
    new java.util.Vector() { public String toString() { return name; } };
   for (Object obj : objects) v.add(obj);
   return v;
  }
 }
```

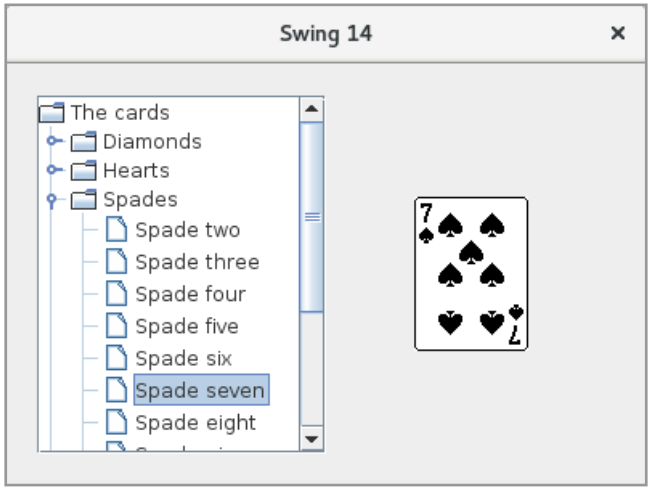
The method *createVector()* creates a *Vector*, which is a collection of objects that a *JTree* can use to create a data model. A *Vector* is itself an object and can therefore be an element in another *Vector*, thus constructing a hierarchy. The method creates a *Vector* based on an anonymous class (a class inheriting *Vector*) alone for the purpose of override *toString()*. The constructor has the value that *toString()* must return and otherwise the objects that the class should contain. The hierarchy is defined in the constructor of *JTree*, and the code is actually not very readable, so maybe it would be better to write something like the following:

```
 Vector v1 = createVector("Knud den Store", "Hardeknud", "Magnus den
Gode");
 Vector v2 = createVector("Svend Tveskæg", "Harald d. 2.");
 v2.add(v1);
 Vector v3 = createVector("Svend Estridsen", "Harald Hen", "Knud den Hellige",
                          "Oluf Hunger", "Erik Ejegod", "Niels");
 Vector v4 = createVector("", "Gorm den Gamle", "Harald Blåtand");
 v4.add(v2);
 v4.add(v3);
 JTree tree = new JTree(v4);
```

When the constructor in the *JTree* class builds a data model, it is an object of the type *TreeModel*, which consists of a hierarchy of *TreeNode* objects. Both of these types are interfaces, and of course, the intention is that you can write your own model, but in the vast majority of cases, you can use default classes that are part of the Swing API. The model is implemented by the class *DefaultTreeModel*, but seen from the programmer is the most important class the class *DefaultMutableTreeNode*, which implements the *TreeNode* interface (and its sub interface, called *MutableTreeNode*). A typical use of a JTree is to construct a hierarchy consisting of *DefaultMutableTreeNode* objects, and then use this hierarchy as a parameter for the constructor in the class *JTree*.

The program TreeProgram3 opens the window shown below. The window has a *JTree*, which showed an overview of the playing cards. Double-clicking a particular card (a leaf node) will display the card in a *JLabel* component. So to write the program, there are two problems to be solved:

1. to build a data model as a hierarchy showing the playing cards
2. to capture double click on an element in the tree



The project defines the following class, which represents a playing card with a name and an image respectively:

```
package treeprogram3;

import javax.swing.*;

public class Card
{
 private String name;
 private ImageIcon img;

 public Card(String name, ImageIcon img)
 {
  this.name = name;
  this.img = img;
 }

 public ImageIcon getImage()
 {
  return img;
 }

 public String toString()
 {
  return name;
 }
}
```

The name is the text that is returned by the class's *toString()* and hence the text displayed in the *JTree* component. With this class available, the data model can be structured as follows (where I have only shown a small part of the method):

```
private TreeModel createModel()
{
 MutableTreeNode root = new DefaultMutableTreeNode("The cards");
 MutableTreeNode diam = new DefaultMutableTreeNode("Diamonds");
 MutableTreeNode hear = new DefaultMutableTreeNode("Hearts");
 MutableTreeNode spad = new DefaultMutableTreeNode("Spades");
 MutableTreeNode club = new DefaultMutableTreeNode("Clubs");
 diam.insert(createNode("Diamond two", "/swing14/images/ru2.GIF"), 0);
 diam.insert(createNode("Diamond three", "/swing14/images/ru3.GIF"), 1);
 diam.insert(createNode("Diamond four", "/swing14/images/ru4.GIF"), 2);
 …
```

```
  root.insert(diam, 0);
  root.insert(hear, 1);
  root.insert(spad, 2);
  root.insert(club, 3);
  return new DefaultTreeModel(root);
 }
```

The class *DefaultMutableTreeNode* defines a node for the tree, and you can specify an object to be associated with that node. It is the *toString()* value of this object that is displayed in the tree. The method starts by creating 5 nodes, the first one being used for the root, while the four others must be used for each of the four colors. The *DefaultMutableTreeNode* class has a method *insert()*, and it is subsequently used to associate child nodes with the nodes representing the colors. One example is

```
diam.insert(createNode("Diamond two", "/treeprogram3/images/ru2.GIF"), 0);
```

which associates the card diamond 2 to the node of diamonds and the method thus has additional 51 corresponding statements. The *createNode()* method creates and returns a *DefaultMutableTreeNode* to which a *Card* object is attached. This means, among other things, that the image for that card must be loaded from the program's jar file, which occurs in the same way as shown in previous examples.

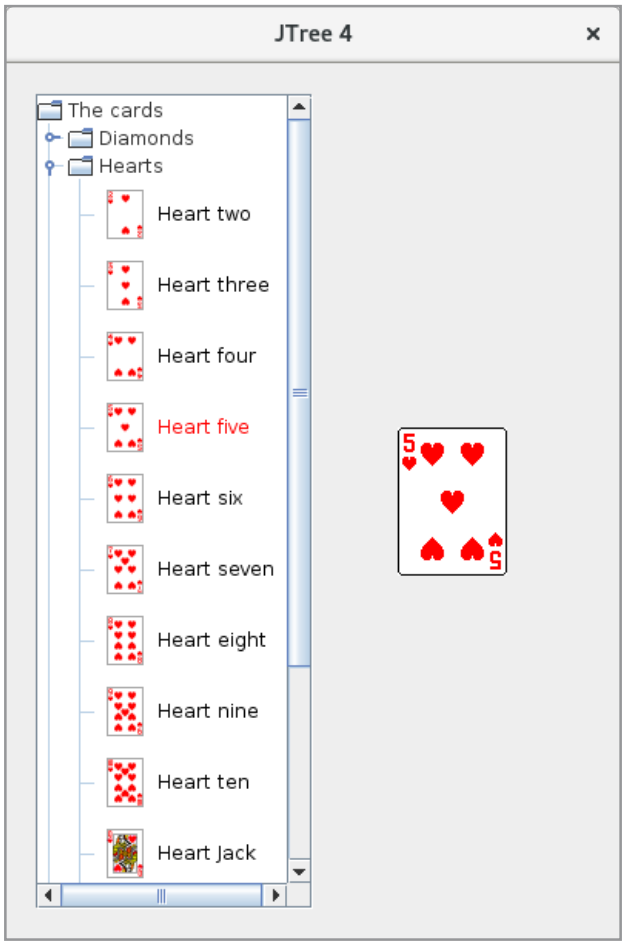
Finally, the four nodes to colors are added in the root node, and the method returns the model to a *JTree*.

Then there is the double click with the mouse, and for that purpose, a *MouseListener* is attached to the *JTree* component:

```
class MouseHandler extends MouseAdapter
{
 public void mousePressed(MouseEvent e)
 {
  int row = tree.getRowForLocation(e.getX(), e.getY());
  if(row != -1 && e.getClickCount() == 2)
  {
   TreePath path = tree.getPathForLocation(e.getX(), e.getY());
   MutableTreeNode node = (MutableTreeNode)path.getLastPathComponent();
   if (node.isLeaf())
   {
    Card card = (Card)((DefaultMutableTreeNode)node).getUserObject();
    lblCard.setIcon(card.getImage());
   }
  }
 }
}
```

The first statement in *mousePressed()* determines the index on the row that is selected. If a row is selected and if it is a double click, the path (in the hierarchy of the tree) is determined to the node that is clicked. Such a path is represented by a *TreePath*, which is a list of nodes from root to the current node. For example, if the click is on spade seven, the list – *TreePath* object – consists of the root, the node representing spades, and the node represented spade seven. With the *TreePath* object available, the last node in the path is determined, which is the double-clicked node. Since only something has to be done when is clicked on a node for a specific card, it will be tested whether the node clicked on is a leaf node. If that is the case, the *Card* object assigned to the node is determined, which is used to update a *JLabel* with that card.

A *JTree* is born with a *TreeCellRenderer*, which determines how the component should be drawn, and it will exactly say the icons used and the text. If you open the program *TreeProgram4*, you get the following window:



That is, the *JTree* component now displays a picture of that card and also shows that a node is selected. Apart from that it is the same program as the *TreeProgram3,* but the program defines its own *TreeCellRenderer*, which in this case is an inner class:

```
public class LeafCellRenderer extends DefaultTreeCellRenderer
{
 private JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
 private JLabel lblIcon = new JLabel();
 private JLabel lblText = new JLabel();

 public LeafCellRenderer()
 {
  lblIcon.setPreferredSize(new Dimension(28, 36));
  lblText.setHorizontalAlignment(JLabel.LEFT);
  lblText.setFont(tree.getFont());
  panel.setBackground(Color.white);
  panel.add(lblIcon);
  panel.add(lblText);
 }
```

```
@Override
public Component getTreeCellRendererComponent(JTree tree, Object value,
 boolean selected, boolean expanded, boolean leaf, int row, boolean hasFocus)
{
 DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
 if (!node.isLeaf())
  return super.getTreeCellRendererComponent(tree, value, selected,
   expanded, leaf, row, hasFocus);
 prepareRenderer((Card)node.getUserObject(), selected);
 return panel;
 }

 private void prepareRenderer(Card card, boolean selected)
 {
  lblIcon.setIcon(scale(card.getImage()));
  lblText.setText(card.toString());
  lblText.setForeground(selected ? Color.red : Color.black);
 }

 private ImageIcon scale(ImageIcon icon)
 {
  return new ImageIcon(
   icon.getImage().getScaledInstance(24, 32, java.awt.Image.SCALE_SMOOTH));
 }
}
```

Note that the class is derived from *DefaultTreeCellRenderer*, which is the default renderer a *JTree* uses. The class defines a panel with two *JLabel* components, which are used for an icon and a text, respectively. These components are initialized in the constructor. The class must override the method *getTreeCellRendererComponent()*, which is the method that renders an element in the tree. The method first tests whether it is a leaf node, and if not it calls the base class's method, and the tree is rendered as default. Otherwise, the method calls *prepareRenderer()*, which initializes the two label components.

For everything to work, the tree must use the *TreeCellRenderer*:

```
tree.setCellRenderer(new LeafCellRenderer());
```

and then the tree will apply the custom *TreeCellRenderer*.

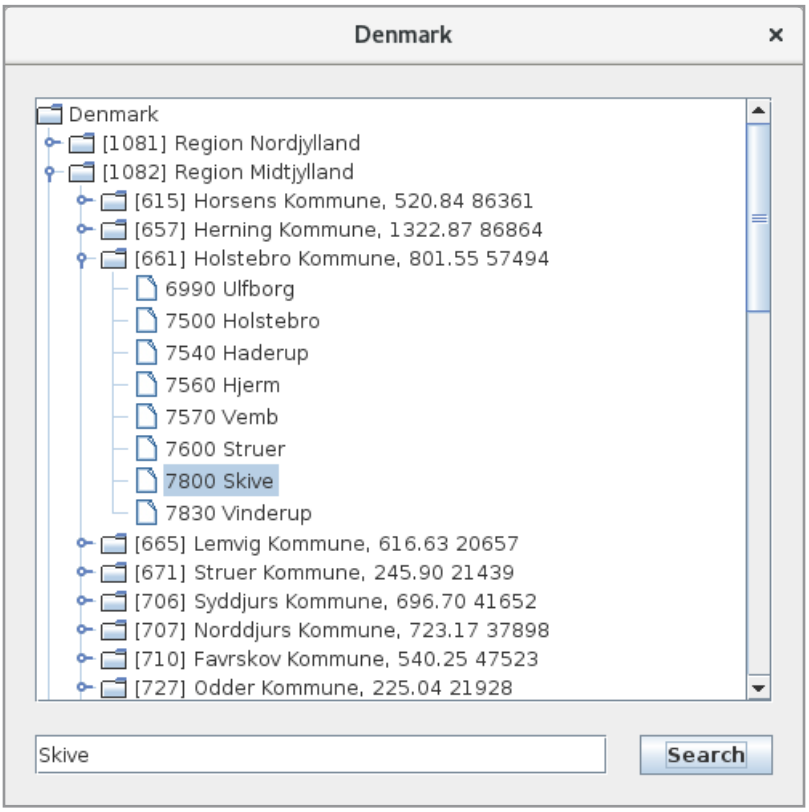
You can also edit the individual nodes in a *JTree*, and you can do this by using the command

```
tree.setEditable(true);
```

The component uses a *DefaultTreeCellEditor*, and if you double-click on a node, opens a *JTextField* component so that the text can be edited. It may not so often have interest, but of course, there are examples, and you can define your own *TreeCellEditor* exactly as with a *TreeCellRender*, and attach it to the tree.

DENMARK

The database *padata* has three tables *region*, *municipality* and *zipcode* containing information about Danish regions and municipalities as well as the zip codes used by the municipalities. The example *DenmarkProgram* shows these data in a *JTree*. In addition, the program has a search function where you can search for a text, and all nodes in the tree where the text contains the search string are selected. Below is an example of the application window where searched for *Skive*:



The program has three simple model classes

1. *Region*
2. *Municipality*
3. *Zipcode*

where the first has a list of all municipalities in this region, while the other has a list of all zip codes used by this municipality. I do not want to show these model classes here. There is also a model class called *Denmark*, which contains only a list of all regions.

Finally, there is a class *Model*, which in the constructor reads the database tables and builds a tree structure with an object of the type of *Denmark* as the root. The class contains *Region* objets, and each *Region* object contains *Municipality* objects which in turn contains *Zipcode* objects. Nor do I want to show the *Model* class here, and it consists solely of methods that read the appropriate database tables.

Back there is the *MainView* of the program, where there are several concepts about the *JTree* component that you should be aware of. The code is as follows:

```
package denmarkprogram;

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.border.*;

public class MainView extends JFrame
{
 private Model model = new Model();
 private JTree tree;
 private JTextField txtText = new JTextField();

 public MainView()
 {
  super("Denmark");
  setSize(500, 500);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new BorderLayout(0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  tree = new JTree(createTreeModel());
  tree.setRootVisible(true);
  panel.add(new JScrollPane(tree));
  panel.add(createBottom(), BorderLayout.SOUTH);
  add(panel);
 }

 private JPanel createBottom()
 {
  JButton cmd = new JButton("Search");
  cmd.addActionListener(this::select);
  JPanel panel = new JPanel(new BorderLayout(20, 0));
  panel.add(cmd, BorderLayout.EAST);
  panel.add(txtText);
  return panel;
 }
```

```
private void select(ActionEvent e)
{
 TreeNode root = (TreeNode)tree.getModel().getRoot();
 collapse(getPath(root));
 String text = txtText.getText().trim();
 tree.clearSelection();
 ArrayList<TreePath> paths = new ArrayList();
 for (int i = 0; i < root.getChildCount(); ++i)
  select(paths, (DefaultMutableTreeNode)root.getChildAt(i), text);
 TreePath[] array = new TreePath[paths.size()];
 tree.setSelectionPaths(paths.toArray(array));
}

private void select(ArrayList<TreePath> paths,
 DefaultMutableTreeNode node, String text)
{
 if (node.getUserObject().toString().contains(text)) paths.add(getPath(node));
 for (int i = 0; i < node.getChildCount(); ++i)
  select(paths, (DefaultMutableTreeNode)node.getChildAt(i), text);
}
```

```
 private void collapse(TreePath root)
 {
  TreeNode node = (TreeNode) root.getLastPathComponent();
  if (node.getChildCount() >= 0)
  {
   for (Enumeration e = node.children(); e.hasMoreElements(); )
   {
    TreePath path = root.pathByAddingChild((TreeNode) e.nextElement());
    collapse(path);
   }
  }
  tree.collapsePath(root);
 }

 private TreePath getPath(TreeNode node)
 {
  ArrayList<Object> nodes = new ArrayList();
  nodes.add(node);
  for (node = node.getParent(); node != null; node = node.getParent())
   nodes.add(0, node);
  return new TreePath(nodes.toArray());
 }

 private TreeNode createTreeModel()
 {
  DefaultMutableTreeNode root = new DefaultMutableTreeNode(model.getDenmark());
  int n1 = 0;
  for (Region r : model.getDenmark().getRegions())
  {
   MutableTreeNode node1 = new DefaultMutableTreeNode(r);
   int n2 = 0;
   for (Municipality m : r.getMunicipalities())
   {
    MutableTreeNode node2 = new DefaultMutableTreeNode(m);
    int n3 = 0;
    for (Zipcode z : m.getZipcodes())
     node2.insert(new DefaultMutableTreeNode(z), n3++);
    node1.insert(node2, n2++);
   }
   root.insert(node1, n1++);
  }
  return root;
 }
}
```

The program defines three variables, which is a *Model* object and thus data for the tree. In addition, there is a variable for the *JTree* component and the search text input field. With regard to the design of the window there is no new in it, so it should not be commented further here.

The last method builds the data model of the tree based on the model object and thus the data read in the database. The data model is constructed as a hierarchy consisting of nodes of the type *MutableTreeNode*. The first node represents the root of the tree and thus a *Denmark* object. To this node, is created a *Region* node for each region, and a node for the region's municipalities, and finally, foor each municipality nodes a node for each zip code used by that municipality. The method is used in *createView()* to create a *JTree* object, and you should note that the argument of the constructor thus is a *TreeNode* for the root of the tree.

The individual nodes in a *JTree* can be referenced in several ways, but the most important is with a *TreePath*, which is the list of nodes leading from the root to the particular node. Here you should note the method *getPath()*, which is a method that determines a *TreePath* object for a specific node. Here, it is assumed that all nodes that is not the root has a parent reference to the node from which that node is a child.

You must then note the method *collapse()* that collapses part of the tree. The method has a *TreePath* parameter and collapses that part of the tree that has path's node as root. You should especially note how to refer to the node that that *TreePath* represents. Also note that the method is recursive and how to collapse the node *root* with the statement

```
tree.collapsePath(root);
```

If you instead replace this statement with

```
tree.expandPath(root);
```

the method will expand the tree completely.

Back there is the event handler for the button. It starts by determining the tree's root as a node, and then collapses the whole tree. Next, an *ArrayList* is created for *TreePath* objects, and a loop is performed over all child nodes to the root. Each iteration calls a method *select()* that creates a *TreePath* object for all nodes that match the search string. This method is also recursive, but the result is that the *ArrayList* contains all nodes that are to be represented by a *TreePath*. The list is finally used to mark the individual nodes

```
tree.setSelectionPaths(paths.toArray(array));
```

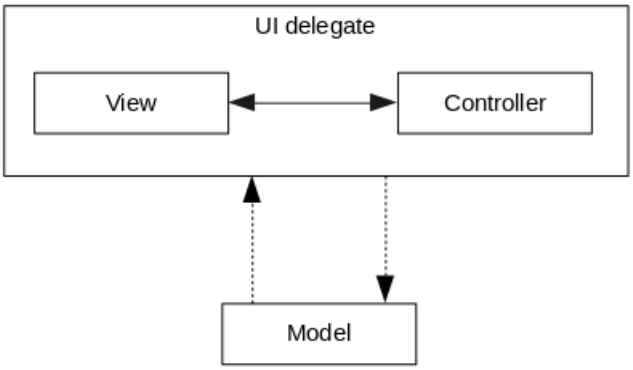
which means that the individual nodes are selected and that the tree is expanded so that all selected nodes are visible.

5 USER DEFINED COMPONENTS

In the first chapters of this book I have mentioned custom components and actually even given two examples, and in this chapter I will attach additional remarks to how to create good custom components. As described in the previous chapter, Swing is born with many components that offer most of what may be needed to design a graphical user interface, but on the other hand, there may be special needs where you want to develop your own components, and, on the other hand, a window may contain panels that will be used in multiple places, and therefore it may be worth developing your own custom components. In any case, the motivation for writing a custom component will always be that there is a need for a graphical element that may be used in multiple places and typically even in completely different programs than the current one.

In principle, it's easy to write custom components using Swing, but if you do it right, and thus write components that behave like any other Swing component, it's not quite simple and there are several requirements that you need ensure that a custom component complies. Therefore this chapter.

Swing components are designed according to a modified MVC pattern, which can be described as the view and controller being built into a common component called a *UI delegate*:



When I mentioned above that writing custom components is not simple, custom components should follow the same pattern as Swing's own components, so I'll start a little about the architecture of swing components and their design.

5.1 LOOK-AND-FEEL

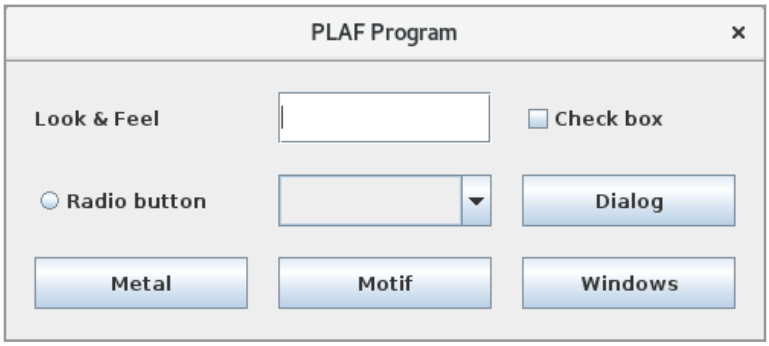
An UI delegate is derived from the class *ComponentUI*, which is an abstract class that basically describes how a UI delegate and a component should communicate, and without giving any explanation, the class (defines) has the following methods:

- *static ComponentUI CreateUI(JComponent c)*
- *installUI(JComponent c)*
- *uninstallUI(JComponent c)*
- *update(Graphics g, JComponent c)*
- *paint(Graphics g, JComponent c)*
- *getPreferredSize(JComponent c)*
- *getMinimumSize(JComponent c)*
- *getMaximumSize(JComponent c)*

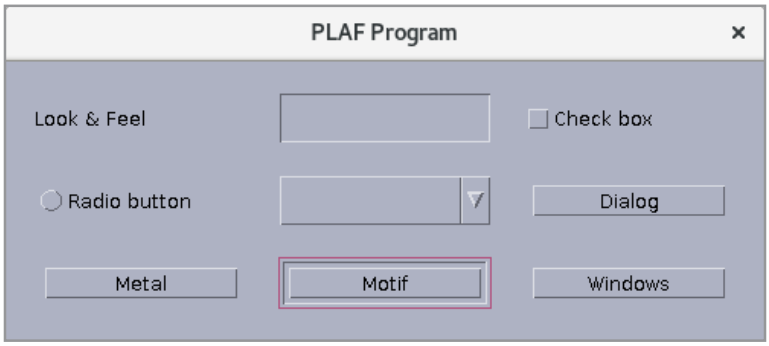
Most UI delegates are written to know a component's model (or models as a component may have multiple model objects) through a defined interface. Swing has multiple families of UI delegates, and each family contains a class (which implements *ComponentUI*) for most Swing components, and such a family is called a *look-and-feel* (or *PLAF* for *pluggable look-and-feel*). *javax.swing.plaf* contains abstract classes all derived from *ComponentUI* and *javax.swing.plaf.basic* consists of classes that expand the abstract classes from *javax.swing.plaf* with specific classes. These UI delegate classes are used as the building blocks of all *look-and-feel* families. There are the following families, called *Windows*, *Motif* and *Metal*, respectively:

1. *com.sun.java.swing.plaf.windows.WindowsLookAndFeel*
2. *com.sun.java.swing.plaf.motif.MotifLookAndFeel*
3. *javax.swing.plaf.metal.MetalLookAndFeel*

where the latter is default (and there is also a *MacLookAndFell*, but has to be installed). However, a given look-and-feel can only be used if it is on the right platform, and for example the first of the above can not be used on a Linux machine. A look-and-feel and its UI delegates define how Swing components are displayed on the screen and typically with support from the current platform, and it is actually possible to change look-and-feel at runtime. If you open the *PLAFProgram* program you get the following window:



that has 9 components arranged using a *GridLayout*. The window's look-and-feel is *Metal*, which is default, but if you click on the *Motif* button, the window changes to:



This is because the look-and-feel is changed from *Metal* to *Motif*. If I then click on the *Windows* button, I get an exception, but it's because my program is running on a Linux machine. Instead, clicking the *Dialog* button the program opens a message box:

where the appearance is due to that the look-and-feel is *Motif*. The example shows that how a program's windows and components are displayed is determined by the current look-and-feel. The program code is as follows:

```
package plafprogram;

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

public class MainView extends JFrame
{
 public MainView()
 {
  super("PLAF Program");
  setSize(500, 220);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
```

```
private void createView()
{
 JPanel panel = new JPanel(new GridLayout(3, 3, 20, 20));
 panel.setBorder(new EmptyBorder(20, 20, 20, 20));
 JLabel lbl = new JLabel("Look & Feel");
 panel.add(lbl);
 panel.add(new JTextField());
 panel.add(new JCheckBox("Check box"));
 panel.add(new JRadioButton("Radio button"));
 panel.add(new JComboBox());
 panel.add(createButton("Dialog",
  e -> JOptionPane.showMessageDialog(this, "Hello World")));
 panel.add(createButton("Metal", this::metal));
 panel.add(createButton("Motif", this::motif));
 panel.add(createButton("Windows", this::windows));
 add(panel);
}

private JButton createButton(String text, ActionListener listener)
{
 JButton cmd = new JButton(text);
 cmd.addActionListener(listener);
 return cmd;
}

private void metal(ActionEvent e)
{
 try
 {
  UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
  SwingUtilities.updateComponentTreeUI(this);
 }
 catch (Exception ex)
 {
  JOptionPane.showMessageDialog(this, ex.toString());
 }
}

private void motif(ActionEvent e)
{
 try
 {
  UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  SwingUtilities.updateComponentTreeUI(this);
 }
 catch (Exception ex)
 {
```

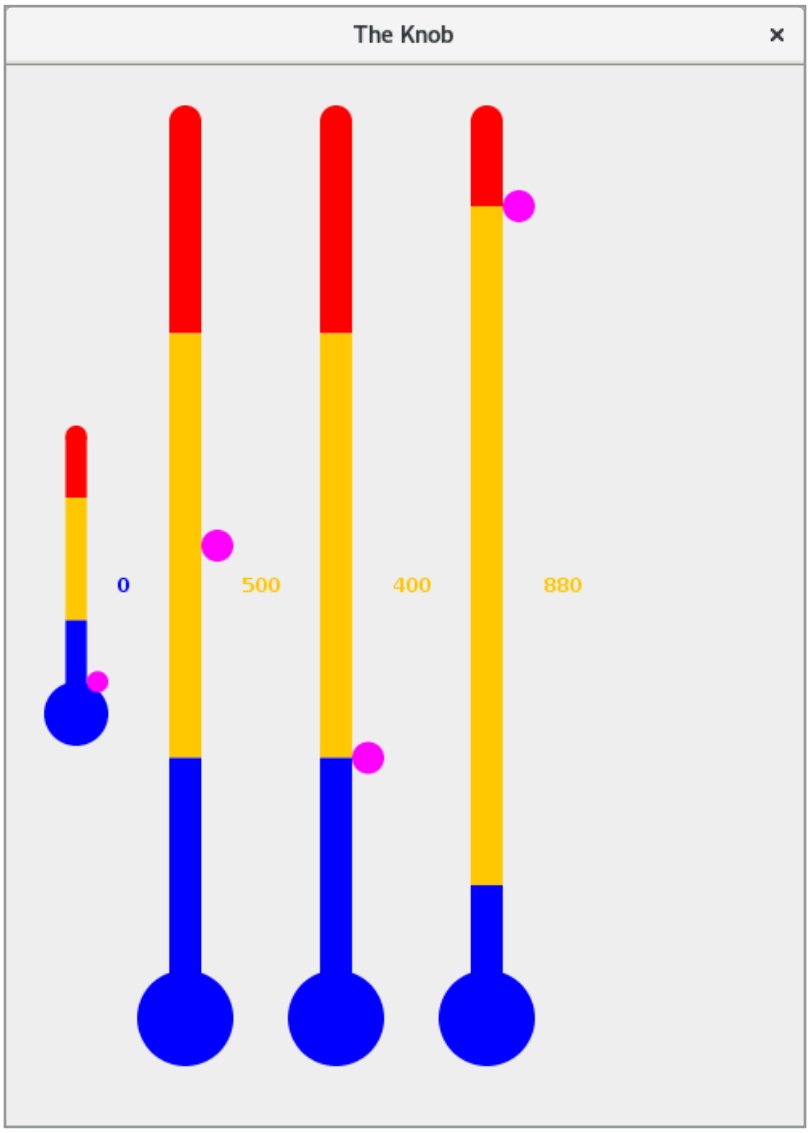
```
    JOptionPane.showMessageDialog(this, ex.toString());
   }
  }

  private void windows(ActionEvent e)
  {
   try
   {
    UIManager.setLookAndFeel(
      "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    SwingUtilities.updateComponentTreeUI(this);
   }
   catch (Exception ex)
   {
    JOptionPane.showMessageDialog(this, ex.toString());
   }
  }
}
```

and here you should of course primarily notice how to change the look-and-feel, and especially what the individual families of UI delegates are called.

The look-and-feel view is, of course, to ensure that programs with a graphical user interface look like other programs on a particular platform (Windows, Mac), and it may be important if you have to write your own custom components from scratch, because you may need to write UI delegates to the individual look-and-feels. Finally, it means that you in principle can write your own look-and-feel (which is not quite simple), thus determining how components should behave.

5.2 KNOB

The *KnobProgram* example opens a window with eight components placed in a Window using a *FlowLayout*:



There are four *JLabel* components, and four *Knob* components. It is a custom component that, like a *JScrollBar* and a *JSlider*, encapsulates a value within an interval. You can change the value by dragging the small circle (the *magenta* circle). The four label components show the value of the *Knob* component and are updated whenever the value is changed. The *Knob* component also has a lower and an upper value between 0 and 1, indicating the blue (cold) area, the red (the hot) area and the yellow (normal) area. If you change area by draging the knob, the component fires an event.

In the following I will show how the component is developed. In this section I will start with a simple version that is not fully compatible with Swing's own components, but I will improve that in the next section.

The starting point for a custom component is a class that inherits *JComponent* – or another component, and especially a container as a *JPanel*. Starting from the development of a component derived from *JComponent*, you can say that you are starting from scratch.

A component can usually raises events and will thus have associated listeners for which notifications can be sent. In this case, a notification must be sent when the component's properties (there are five) are changed, but since the class inherits *JComponent*, it already has the required logic for *PropertyChangeEvent* so *PropertyChangeListener* objects can be registered as listeners. The class must thus take care only of raising events for the new five properties.

As mentioned above, the class must also fire an event when the component's value changes between the boundaries of the three areas (cold, normal and hot) and for that I have defined the following event:

```
package knobprogram;

import java.util.*;

public class KnobEvent extends EventObject
{
 private int value;
 private int from;
 private int to;

 public KnobEvent(Knob source, int value, int from, int to)
 {
  super(source);
  this.value = value;
  this.from = from;
  this.to = to;
 }

 public int getValue()
 {
  return value;
 }

 public int getFrom()
 {
  return from;
 }

 public int getTo()
 {
  return to;
 }
}
```

where the parameters of the event is the current value of the component, the state that changed from and the state that is changed to. You should note that the class is derived from *EventObject*. It is the base class for all events, and its constructor requires a single parameter of the type *Object* that represents the object (source object) that has fired the event. The class is trivial and only has a *get* method for the source object, but it is part of the *Swing* component convention that all event objects must be directly or indirectly derived from *EventObject*.

Since the *Knob* class can fire events of the type *KnobEvent*, there must be other objects that can be registered as listeners, and therefore, a simple listener interface is also defined:

```
package knobprogram;

import java.util.*;

public interface KnobListener extends EventListener
{
 void knobChanged(KnobEvent e);
}
```

Here you should also note that this interface extends *EventListener*, which is nothing but a simple marker interface, and again it is a convention in Swing that all listeners should implement this interface. The reason is that a listener object can thus be added to a collection with listeners.

Then there is the component itself, which consists of five properties:

1. *min*, which is the smallest value of the component
2. *max*, which is the largest value of the component
3. *value*, that is the current value
4. *lower*, which is the proportion for the cold area and is a value between 0 and 1
5. *upper*, which is the proportion of the cold and normal area and is a value between 0 and 1 – it is thus the lower limit for the hot area.

The component's code is shown below in full:

```
package knobprogram;

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

public class Knob extends JComponent
{
 public static final int HOT = 3;
 public static final int NORMAL = 2;
 public static final int COLD = 1;
 private int min;
 private int max;
 private double upper;
 private double lower;
 private int value;
 private int state;
```

```
private boolean marked = false;
private Ellipse2D knob = null;
private double height;
private double width;
private double t;
private double m;
private double w;

public Knob()
{
 this(0, 100, 0);
}

public Knob(int min, int max, int value)
{
 this(min, max, 0.25, 0.75, value);
}

public Knob(int min, int max, double lower, double upper, int value)
{
 this.min = min;
 this.max = max;
 this.lower = lower;
```

```
 this.upper = upper;
 this.value = value;
 addMouseListener(new MouseHandler());
 addMouseMotionListener(new MouseMoveHandler());
 setState();
}

public void paintComponent(Graphics g)
{
 double pos = calculatePosition();
 Graphics2D g2d = (Graphics2D)g;
 g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
 g2d.setPaint(Color.red);
 g2d.fill(new Ellipse2D.Double(w, 0, w, w));
 g2d.fill(new Rectangle2D.Double(w, t, w, m * (1 - upper)));
 g2d.setPaint(Color.orange);
 g2d.fill(new Rectangle2D.Double(w, t + m * (1 - upper),
  w, m * (upper - lower)));
 g2d.setPaint(Color.blue);
 g2d.fill(new Ellipse2D.Double(0, height - width, width, width));
 g2d.fill(new Rectangle2D.Double(w, t + m * (1 - lower),
  w, m * lower + width / 2));
 g2d.setPaint(Color.magenta);
 g2d.fill(knob = new Ellipse2D.Double(width * 2 / 3, pos - t, w, w));
}

public void setValue(int value)
{
 int old = this.value;
 this.value = value;
 repaint();
 firePropertyChange("value", old, value);
 old = state;
 setState();
 if (old != state) fireEvent(old, state);
}

public int getValue()
{
 return value;
}

public int getMinimum()
{
 return min;
}
```

```
public void setMinimum(int min)
{
 firePropertyChange("minnimum", this.min, min);
 this.min = min;
 repaint();
}

public void setMaximum(int max)
{
 firePropertyChange("maximum", this.max, max);
 this.max = max;
 repaint();
}

public int getMaximum()
{
 return max;
}

public void setUpper(double upper)
{
 firePropertyChange("upper", this.upper, upper);
 this.upper = upper;
 repaint();
}

public double getUpper()
{
 return upper;
}

public void setLower(double lower)
{
 firePropertyChange("lower", this.lower, lower);
 this.lower = lower;
 repaint();
}

public double getLower()
{
 return lower;
}

public int getState()
{
 return state;
}
```

```
public void addKnobListener(KnobListener listener)
{
 listenerList.add(KnobListener.class, listener);
}

public void removeKnobListener(KnobListener listener)
{
 listenerList.remove( KnobListener.class, listener );
}

protected void fireEvent(int from, int to)
{
 Object[] listeners = listenerList.getListenerList();
 for (int i = 0; i < listeners.length; i += 2)
  if (listeners[i] == KnobListener.class)
    ((KnobListener)listeners[i + 1]).knobChanged(
     new KnobEvent(this, value, from, to));
}

private void calculate()
{
 Dimension dim = getSize();
 width = dim.getWidth();
```

```
 height = dim.getHeight();
 w = width / 3;
 t = w / 2;
 m = height - width - t;
}

private double calculatePosition()
{
 calculate();
 return (m * value + t * min - t * max - m * max) / (min - max);
}

private int calculateValue(double y)
{
 calculate();
 double z = ((y - t) * min + (t + m - y) * max) / m;
 return Math.min(Math.max((int)z, min), max);
}

private void setState()
{
 int state;
 double minValue = (max - min) * lower + min;
 double maxValue = (max - min) * upper + min;
 if (value < minValue) state = COLD;
 else if (value > maxValue) state = HOT;
 else state = NORMAL;
 if (this.state != state) this.state = state;
}

class MouseHandler extends MouseAdapter
{
 @Override
 public void mousePressed(MouseEvent e)
 {
  marked = knob.contains(e.getX(), e.getY());
 }

 @Override
 public void mouseReleased(MouseEvent e)
 {
  marked = false;
 }
```

```
    @Override
    public void mouseExited(MouseEvent e)
    {
     marked = false;
    }
  }

  class MouseMoveHandler extends MouseMotionAdapter
  {
    @Override
    public void mouseDragged(MouseEvent e)
    {
     if (marked)
     {
      setValue(calculateValue(e.getY()));
     }
    }
  }
}
```

There are many details, but the class is derived from *JComponent*, which actually says that it is a component. First, three constants are defined which indicates in which of the three areas the current value is. There are six variables for properties, where the first five are the five properties that can be changed with a *set* method, while the last *state* tells the current of the three areas. There are additional seven instance variables, which are help variables and are used for the following:

- *marked* indicates that the mouse button is holding down, so that you can drag the knob
- *knob* is the figure (ellipse) that currently represents the knob
- *height* is the component's current height
- *width* is the component's current width
- *t* is the height of a figure above the thermometer (a half circle)
- *m* is the area of the three thermometer (cold, normal and warm)
- *w* is one third of the current width of the component and indicates the width of the thermometer areas

These variables are initialized each time the component is drawn and each time the knob is moved.

The class has three constructors that initializes the six properties. The constructors are generally trivial, but you should note that they defines that the component is listening to the mouse. The event handlers are defined as inner classes at the end of the code. The two classes are trivial and there is not much to explain, but you should note what the classes are doing, and especially how they assign values to the variable *marked*.

Then there are *get* and *set* methods for class's properties. There are six *get* methods that, of course, are all trivial, and there are five *set* methods in which *setValue()* is the most important. The method can be called by the users, but is also called from the method of the event handler *MouseMoveHandler* and hence each time the knob is moved. First, the current value of the component is saved in a local variable, after which the component is updated. Next, the component must be redrawn, which occurs by calling *repaint()*. Then the method fires a *PropertyChangeEvent* for the property *value*. Next, it is tested whether there is a changed *state* (if the value represents another area) and, if it is the case, a *KnobEvent* is firing, which occurs in the method *fireEvent()*. The four other set methods work a bit the same, but are simpler. First, a *PropertyChangeEvent* is firing for that property, which is then updated and the component is redrawn. Note that the class has methods so *KnobListener* objects can register themselves as listeners for *KnobEvent*'s and possibly again be unregistered.

Back there is only drawing the component and how to do that is the subject in the next book, but briefly the following happens. When the component has to be drawn, the method *paintComponent()* is called. It is called when the runtime system sees that the component is invalid (must be redrawn) and it happens for example when the window is created or when the method *repaint()* is executed. Note that the *set* methods do so and ensure that the window becomes redrawn when the component's properties are changed. The *paintComponent()* method has a *Graphics* object that represents the drawing area and provides some drawing tools. In this case, the following happens:

1. the method *calculatePosition()* is called, that initialize the help variables and returns the y positionen for the knob
2. the *Graphics* object is type casted to a *Graphics2D* object, that has better drawing tools
3. the drawing quality is defined
4. the color is defined as red
5. a circle is drawn
6. a rectangle is drawn for the hot area
7. the color is defined as orange
8. a rectangle is drawn for the normal area
9. the color is defined as blue
10. a circle is drawn – the lower circle with the mercury
11. a rectangle is drawn for the cold area
12. the color is defined as magenta
13. the knob is drawn

Then there is the test application that uses the *Knob* component and I do not want to display the code here, but you should note that the component is used in the same way as other components.

5.3 A BETTER KNOB

A custom component is often developed as described above, and as components as part of one's own applications, it is also an excellent method, but the method is not entirely in line with the recommendations for how to develop a Swing component, partly because the component should support look-and-feel. For practical programming, the biggest problem with the *Knob* component, as described above, is that it does not separate user interface and model, which makes it more difficult to maintain the code. In this section I will look at another version of the *Knob* component, where the difference is that the code is written differently.

Basically, a Swing component consists of

1. The component's class, which specifies how to create the component, how the component can be changed and how to read the component's state.
2. The component's model that consists of a defining interface and a default implementation. It is the model that implements the component's logic and sends notifications to listeners.
3. A so-called UI delegate, who stands for the component layout, event management regarding mouse and keyboard, and finally, is the UI delegate who will draw the component.

The project *KnobProgram1* is the same program as the project *KnobProgram*, but the *Knob* component is written in accordance with the above pattern. Basically, it is the same code, but it is just organized in a different way.

The component can still raise a *KnobEvent*, and the two types of *KnobEvent* and *KnobListener* are completely unchanged relative to the first version of the component.

I will then start with the model, which is defined by the following interface:

```
package knobprogram;

import java.beans.*;

public interface KnobModel
{
 public int getMinimum();
 public int getMaximum();
 public int getValue();
 public int getState();
 public double getLower();
 public double getUpper();
 public void setMinimum(int minimum);
 public void setMaximum(int maximum);
 public void setValue(int value);
 public void setLower(double lower);
 public void setUpper(double upper);
 public void addPropertyChangeListener(PropertyChangeListener listener);
 public void removePropertyChangeListener(PropertyChangeListener listener);
 public void addKnobListener(KnobListener listener);
 public void removeKnobListener(KnobListener listener);
}
```

The interface thus defines the component's properties and the extent to which there are *get* and *set* methods for these properties. In addition, the interface defines which listeners can register as recipients of notifications from the component. Here are the *PropertyChangeListener* and *KnobListener* objects. The model is implemented by classes *DefaultKnobModel*. I do not want to show the overall class here, but the start is as follows:

```
public class DefaultKnobModel implements KnobModel
{
 public static final double epsilon = 0.0000000001;
 private EventListenerList listenerList = new EventListenerList();
 public static final int HOT = 3;
 public static final int NORMAL = 2;
 public static final int COLD = 1;
 private int min;
 private int max;
 private double upper;
 private double lower;
 private int value;
 private int state;
```

Here you can see that the variables have been moved from the *Knob* class, but only those variables that represents the component's state, but not the variables used to draw the component. Also note that the model has its own collection for listeners and note the type that is *EventListenerList*. The class has appropriate constructors – this case three in the same way as the original *Knob* class. The rest of the code consists in implementing the methods that the interface defines and here it is especially the *set* methods that you need to be aware of as they will fire the desired events.

Then there is the UI delegate, and according to the pattern for the development of a custom component, it starts with the following class:

```
package knobprogram;

import javax.swing.plaf.*;

public class KnobUI extends ComponentUI
{
 public static final String UI_CLASS_ID = "KnobID";
}
```

The class inherits *ComponentUI* and does not contain much, and all it does is define a constant so the class has a unique name in terms of look-and-feel. With this class in place, you can write the UI delegate, which is the class that will draw the component and handle the events for mouse and keyboard. Below I have shown the start of the class, but I have not shown anything about drawing the component as well as inner classes that defines event handlers for the mouse, as it is the same code as in the previous version.

```
public class BasicKnobUI extends KnobUI
{
 private MouseHandler mouseHandler = null;
 private MouseMoveHandler motionHandler = null;
 private KnobModel model = null;
 private boolean marked = false;
 private Ellipse2D knob = null;
 private double height;
 private double width;
 private double t;
 private double m;
 private double w;

 public BasicKnobUI(JComponent component)
 {
  model = ((Knob)component).getModel();
 }
```

```
 public static ComponentUI createUI(JComponent c)
 {
  return new BasicKnobUI(c);
 }

 public void installUI(JComponent component)
 {
  Knob knob = (Knob)component;
  knob.addMouseListener(mouseHandler = new MouseHandler());
  knob.addMouseMotionListener(motionHandler = new MouseMoveHandler());
 }

 public void uninstallUI(JComponent component)
 {
  Knob knob = (Knob)component;
  if (mouseHandler != null) knob.removeMouseListener(mouseHandler);
  if (motionHandler != null) knob.removeMouseMotionListener(motionHandler);
 }
```

You should note that the variables used to draw the user interface have been moved to this class. Otherwise, you should notice the methods that have been implemented and without trying to explain what they esact is doing, these are methods that mean that the component supports look-and-feel and is thus part of the pattern for developing a custom component. Back there is the component itself and hence the class *Knob*, which has now become a very thin class:

```
package knobprogram;

import javax.swing.*;
import java.beans.*;

public class Knob extends JComponent implements PropertyChangeListener
{
 private KnobModel model;

 public Knob()
 {
  init(new DefaultKnobModel());
 }

 public Knob(KnobModel model)
 {
  init(model);
 }
```

```
public Knob(int min, int max, int value)
{
 init(new DefaultKnobModel(min, max, value));
}

public Knob(int min, int max, double lower, double upper, int value)
{
 init(new DefaultKnobModel(min, max, lower, upper, value));
}

private void init(KnobModel model)
{
 setModel(model);
 updateUI();
}

public String getUIClassID()
{
 return KnobUI.UI_CLASS_ID;
}

public void setModel(KnobModel model)
{
```

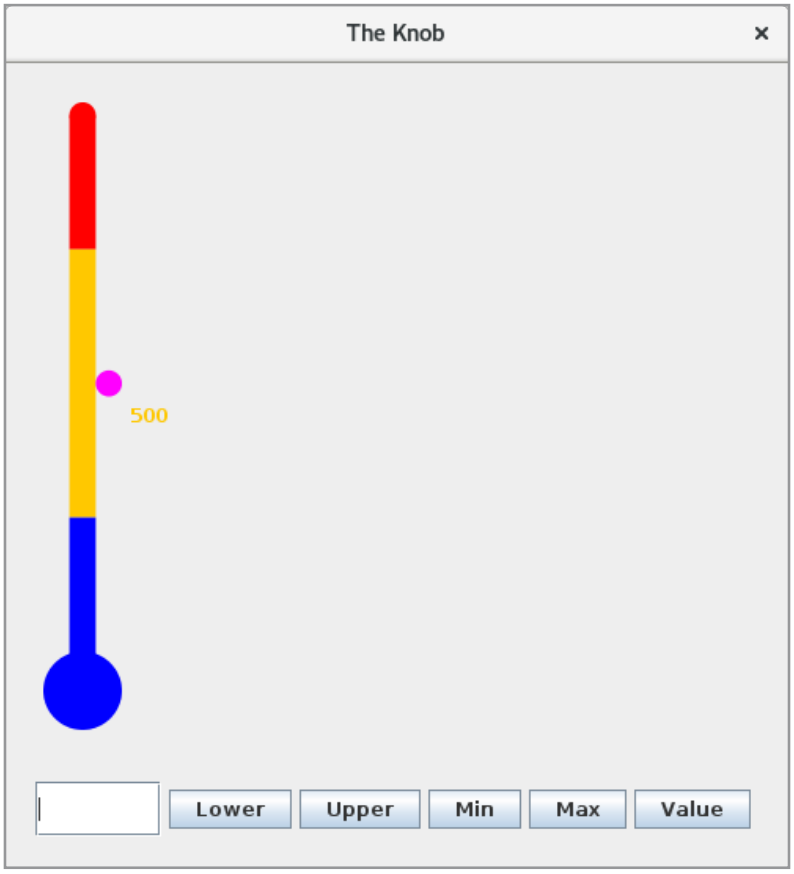
```
 KnobModel old = this.model;
 if (old != null) old.removePropertyChangeListener(this);
 if (model == null) this.model = new DefaultKnobModel();
 else this.model = model;
 this.model.addPropertyChangeListener(this);
 firePropertyChange("model", old, this.model);
}

public KnobModel getModel()
{
 return model;
}

public KnobUI getUI()
{
 return (KnobUI) ui;
}

public void setUI(KnobUI ui)
{
 super.setUI(ui);
}

public void updateUI()
{
 if (UIManager.get(getUIClassID()) != null)
  setUI((KnobUI)UIManager.getUI(this));
 else setUI(new BasicKnobUI(this));
}

public void propertyChange(PropertyChangeEvent e)
{
 firePropertyChange(e.getPropertyName(), e.getOldValue(), e.getNewValue());
 repaint();
}

public int getMinimum()
{
 return model.getMinimum( );
}

public void setMinimum(int min)
{
 int old = getMinimum();
 if (min != old)
 {
```

```
   model.setMinimum(min);
   firePropertyChange("minimum", old, min);
  }
 }

 public int getMaximum()
 {
  return model.getMaximum();
 }

 public void setMaximum(int max)
 {
  int old = getMaximum();
  if (max != old)
  {
   model.setMaximum(max);
   firePropertyChange("maximum", old, max);
  }
 }

 public int getValue()
 {
  return model.getValue();
 }

 public void setValue(int val)
 {
  int old = getValue();
  if (val != old)
  {
   model.setValue(val);
   firePropertyChange("value", old, val);
  }
 }

 public int getState()
 {
  return model.getState();
 }

 public double getLower()
 {
  return model.getLower();
 }
```

```
public void setLower(double lower)
{
 double old = getLower();
 if (Math.abs(lower - old) > DefaultKnobModel.epsilon)
 {
  model.setLower(lower);
  firePropertyChange("lower", old, lower);
 }
}

public double getUpper()
{
 return model.getUpper();
}

public void setUpper(double upper)
{
 double old = getUpper();
 if (Math.abs(upper - old) > DefaultKnobModel.epsilon)
 {
  model.setUpper(upper);
  firePropertyChange("upper", old, upper);
 }
}
```

```
  public void addKnobListener(KnobListener listener)
  {
   model.addKnobListener(listener);
  }

  public void removeKnobListener(KnobListener listener)
  {
   model.removeKnobListener(listener);
  }
 }
```

In fact, there is not much to explain and the class is in many ways just a wrapper for the model. Since it is the model that maintains the class's properties, the class *Knob* (which is the class known by the users) must delegate the maintenance to the model. The test program has been modified so that it now has only one component of the type *Knob*, but some buttons have been inserted so that you can modify the component's properties:



If entering a value in the input field and clicking on a button the value changes for the corresponding property. You should note that the program does not handle illegal values and raises an execption if the value can not be converted correctly.

EXERCISE 3

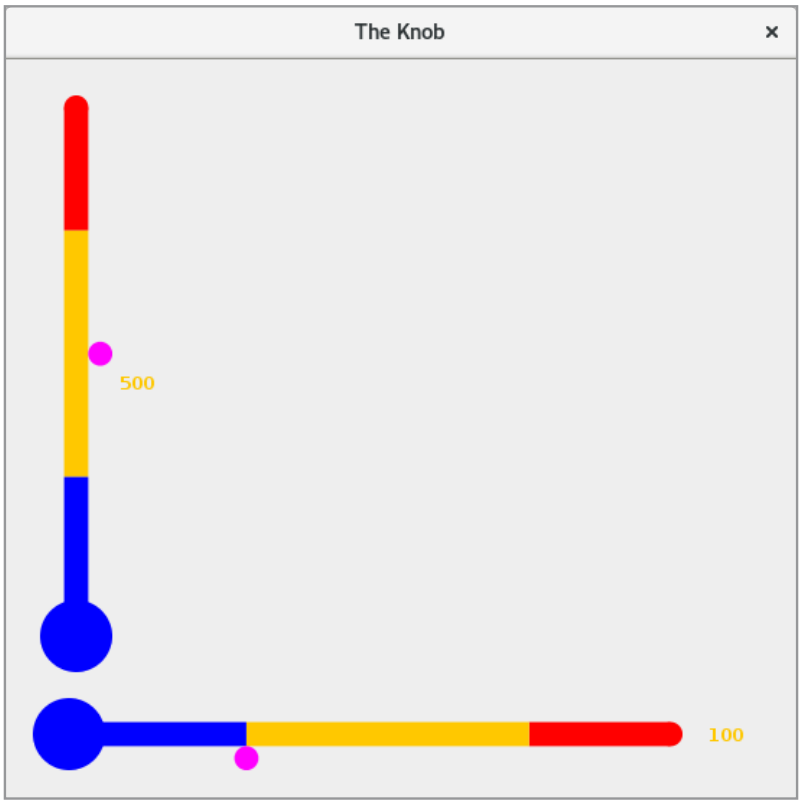
Create a copy of the project *KnobProgram1*. The *Knob* component has several shortcomings, and you can assign random values to its properties, which is unfortunate. It should be a requirement that

1. *minimum* is less than *maximum*
2. *lower* is less than or equal to *upper*
3. *lower* and *upper* must have a value between 0 and 1
4. *value* must be greater than or equal to *minimum* and less than or equal to *maximum*

You must change the component to validate changes to these properties. In the case of illegal values, the component should not raise exceptions, but instead handle the errors by automatically allocating properties a "reasonable" value. You must then expand the model with a property:

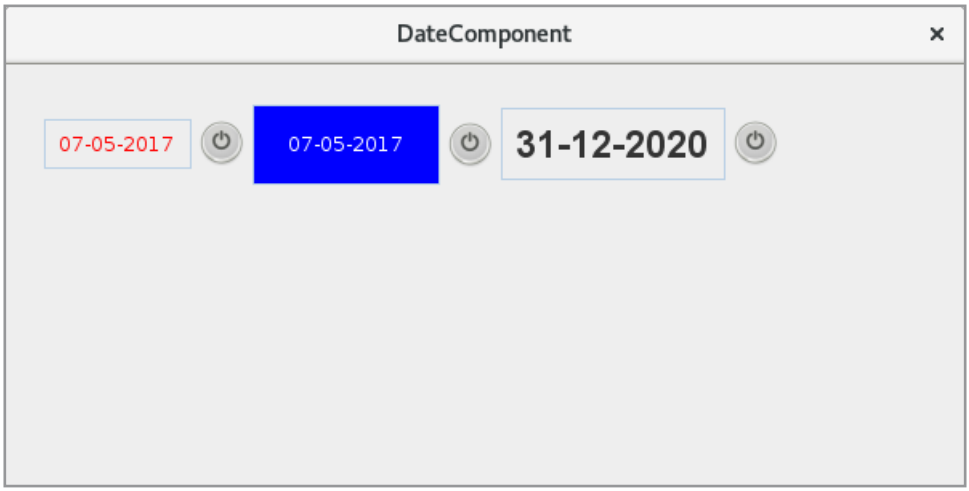
```
private boolean vertical = true;
```

If this property has the default value *true*, the component must be displayed as before, but if the value is changed to *false*, the component should be displayed horizontally. For example, you can change the test program to open the following window:



5.4 A DATEPICKER

I want to finish this chapter with another example of a custom component. The component is different this time, as it is a class that inherits *JPanel*, and instead of drawing the component's user interface, it is a container that contains other Swing components. The component is a so-called *DatePicker*, which is an example of a component where the user can select a date. There are many similar components on the *Internet*, and the following is another example. If you open the test program, you get the following window:



which shows three components of the type *DatePicker*. That is, a *DatePicker* object shows a readonly text box with a date and a button to the right. If you click on the button, you get the following popup:

where you can navigate a calendar and select a date by clicking on it. If you click the red icon in the toolbar, you close the popup without selecting a date.

To write the component, it is clear that a large part of the code is reused from the *Calendar* project in the book Java 8. The basis for the calendar is also the *Date* class from this project, and this class is not mentioned further. The component also has a model, which is largely the same model as in the *Calendar* project. Nor should it be explained further and is a very simple class.

Back there is the actual *DatePicker* component, which is an extensive class whose code I do not want to display here, and where there are many reuses from the *Calendar* project. You are encouraged to study the code and specifically examine which properties are defined. Here you should note that there is a property date used to read the value of the component and possibly change it as well as its type is *Calendar*.

Finally, there is the test program, which is shown below:

```
package datecomponent;

import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.beans.*;
```

```
public class MainView extends JFrame implements PropertyChangeListener
{
 private JLabel lblValue1 = new JLabel();
 private JLabel lblValue2 = new JLabel();

 public MainView()
 {
  super("DateComponent");
  setSize(600, 300);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  DatePicker dp1 = new DatePicker();
  dp1.setForeground(Color.red);
  dp1.addPropertyChangeListener(this);
  panel.add(dp1);
  DatePicker dp2 = new DatePicker();
  dp2.setPreferredSize(new Dimension(150, 50));
  dp2.setBackground(Color.blue);
  dp2.setForeground(Color.white);
  panel.add(dp2);
  DatePicker dp3 = new DatePicker();
  dp3.setFont(new Font("Arial", Font.BOLD, 24));
  dp3.setDate(new GregorianCalendar(2020, 11, 31));
  panel.add(dp3);
  add(panel);
 }

 @Override
 public void propertyChange(PropertyChangeEvent e)
 {
  if (e.getNewValue() instanceof Calendar)
   JOptionPane.showMessageDialog(this, getText((Calendar)e.getNewValue()));
 }
```

```
  private String getText(Calendar date)
  {
   return String.format("%02d-%02d-%04d", date.get(Calendar.DATE),
    date.get(Calendar.MONTH) + 1, date.get(Calendar.YEAR));
  }
 }
```

Here you should especially note how to create *DatePicker* components and add them to the window and that it is done in the same way as other components. Also note that the program is registered as *PropertyChangeListener* for the first component and receives notifications when that component changes date.

The DatePicker component can be of practical interest, and I want to add it to my class library *PaLib*. It takes the following steps:

1. I have created a package *palib.gui.images* and for this I have copied the icons that the component uses (there are 7).
2. The class *CalendarException* is copied th the package *palib.util*.
3. The class *Date* has been copied to the package *palib.util*. It gives a naming problem in the class *Tools*, as it refers to the class *java.util.Date* alone with the name *Date*. This issue must be solved in the *Tools* class by changing all references to *Date* to the full name *java.util.Date*.
4. I have created a package *palib.gui.models* and here I have created a class named *PickerModel*. The content of this class is the component's model (the class *Model* in the *DateComponent* project).
5. The *DatePicker* class is copied to *palib.gui*. This again gives rise to problems with names for *Date* and these issues must be solved by referencing *Date* with the full name *palib.util.Date*.
6. In the class *DatePicker* all names of icons (references) must be changed to the icons that are part of the library.

After these changes, the library is updated and the *DateComponent* project can be modified (to *DateComponent1*), so all classes relating to the component are deleted, and then the project instead uses *PaLib*.

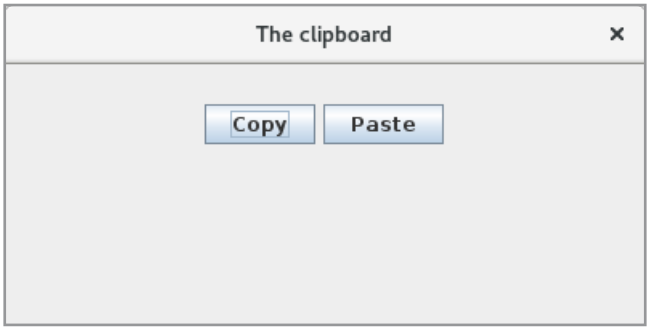
As a last comment. When naming classes, one should generally try not to use names that Java also uses. Not that it is allowed, but it easily leads to problems of the kind, as is the case above.

6 THE CLIPBOARD

Using the clipboard and thus copy and paste is an expected functionality in modern programs, at least as long that it are programs that process text. Basically, it is possible to select some object, such as some text, and save them to the clipboard, then retrieve it and perhaps even into a completely different program. In most programs, it is something that the programmer does not need to relate to, as many components (such as components for editing text) have the functionality built-in. There are also situations where it is necessary to program copy and paste (for example, if you write custom components), therefore, this chapter. Moreover, in the book Java 2, I have shown a little about how to do.

In principle, it is quite simple, but if you are investigating how it is implemented, it is by no means simple, and there are several reasons for that. The clipboard is of course some memory where you can save something, but the clipboard is a concept outside the program and something that is administered by the operating system and not by the virtual machine. Another problem is that data is not just data, and even text is not just text. For example, if writing text in a word processor contains the text many control characters and other formatting information that must also be saved and later re-inserted for paste. It is therefore necessary to also save information about what kind of data is stored on the clipboard. Finally, if you have to copy and paste between two different programs, they may not know each other's data formats. In general, one can immediately conclude that copying and pasting data between two different programs does not necessarily produce meaningful results.

I want to start with a simple program that opens the following window:



Clicking the *Copy* button some text are copied to the clipboard and clicking the button *Paste* opens a message box that displays the text copied on the clipboard (the text is taken from the clipboard). The program thus shows how to save data on the clipboard and retrieve this data. The code is as follows:

```
package theclipboard;

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.datatransfer.*;

public class MainView extends JFrame
{
 private static Random rand = new Random();
 private String[] names = { "Gorm den Gamle", "Harald Blåtand", "Svend Tveskæg" };
 private Clipboard cb = Toolkit.getDefaultToolkit().getSystemClipboard();

 public MainView()
 {
  super("The clipboard");
  setSize(400, 200);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
```

```
 private void createView()
 {
  JPanel panel = new JPanel(new FlowLayout());
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  JButton cmdCopy = new JButton("Copy");
  JButton cmdPaste = new JButton("Paste");
  cmdCopy.addActionListener(this::copy);
  cmdPaste.addActionListener(this::paste);
  panel.add(cmdCopy);
  panel.add(cmdPaste);
  add(panel);
 }

 private void copy(ActionEvent e)
 {
  cb.setContents(new StringSelection(names[rand.nextInt(names.length)]), null);
 }

 private void paste(ActionEvent e)
 {
  try
  {
   JOptionPane.showMessageDialog(this,
     cb.getContents(DataFlavor.stringFlavor).
     getTransferData(DataFlavor.stringFlavor));
   for (DataFlavor df : cb.getAvailableDataFlavors())
     JOptionPane.showMessageDialog(this, df.getMimeType());
  }
  catch (Exception ex)
  {
   JOptionPane.showMessageDialog(this, ex.toString());
  }
 }
}
```

The class defines an array with three strings, and the meaning is that clicking on Copy you saves a random one of these strings on the clipboard, and thus you can see that it is not the same string that is saved each time. You should especially note how to define a reference to the clipboard

```
Clipboard cb = Toolkit.getDefaultToolkit().getSystemClipboard();
```

as in Java is represented by a class *Clipboard*, but otherwise the most important of course are the two event handlers. You save data using the *setContents()* method, which has two parameters. The first is the data to be saved, while the other is the owner. It is not used so often, but has the type *ClipboardOwner* (that is an interface) and the meaning is that the owner can get a notification if the clipboard's content are overwritten (for example, from another application). In this example, the value of the parameter is *null*, which simply means that no notifications about the content are sent.

In order for data to be saved on the clipboard – and read again – they must be encapsulated in a data structure of the type *Transferable*. It is an interface and there are several concrete classes that implements this interface, and *StringSelection* is an example of a *Transferable* that is specially designed to transfer text. In this case, a string is encapsulated in a *StringSelection* object.

Then there is the event handler *paste()*. One retrieves data from the clipboard using the method *getContents()*, where the parameter is a *DataFlavor*, which tells what it is for a kind of data to be retrieved. The *DataFlavor* class has some predefined flavors, and *stringFlavor* is an example that tells that data is a *StringSelection*. It is a *Transferable* that has a method *getTranfereData()*, which returns the specific data, and this method also has a *DataFlavor* as parameter. In fact, the statement that retrieves data from the clipboard could be written as follows:

```
cb.getContents(null).getTransferData(DataFlavor.stringFlavor)
```

The class *Clipboard* has a method *getAvailableDataFlavors()* that returns an array of *DataFlavor* objects that tells how the contents of the clipboard can be interpreted. Sometimes (for example, if it is text), the content can be interpreted in several ways (styled text, html, plain text) and therefore there may be more *DataFlavor* objects where the object that defines most information will be first in the array. The last *for* loop in the event handler *paste()* iterates over all *DataFlavor* objects, and there are two. In particular, if you to *getContents()* indicates an incorrect *DataFlavor* or if the clipboard is empty, you get an exception.

6.1 MIME TYPES

When saving data on the clipboard, data will be defined by a class such as *Java.lang.String*, and as long as there is only a need to transfer data between Java applications, it is sufficient to tell a *DataFlavor* which type, but data should be transferred to other programs, which of course do not know Java's classes, and there is a need for another way to define what data formats are. Thus, there is a need for platform independent and language-neutral term for data formats. To this end, *MIME* is used, which is an Internet standard, which makes it possible to attach various documents to electronic mail.

A MIME type consists of a media type that can be considered as a parent category as well as a subtype that defines a more specific type within the category. The types are separated by the character / and an example could be

```
text/plain
```

which indicates plain text without formatting information. There are many other *MINE* types, for example

```
image/gif
image/jpg
text/html
…
```

You can also define your own MIME types, which must consist of a known media type as well as a subset that starts with "x-", which indicates that it is an unregistered type. Finally is

```
application/octet-stream
```

a generic type that specifies binary data in an unknown format. In addition to type and subtype, a *MIME* type may also have associated other information for example regarding encoding of text. The syntax is as shown in the following examples

- `text/plain;charset=unicode`
- `text/plain;charset=ascii`
- `text/plain;charset=iso-8859-1`

When creating a *DataFlavor*, you can specify the *MIME* type, for example:

```
DataFlavor df = new DataFlavor("text/htm");
```

As another example of a program that uses the clipboard, the program *ClipBytes* copies a byte array to the clipboard. The program has exactly the same user interface as the above program, and it also works in the same way, just copying it an array of 10 random bytes to the clipboard. Similarly, if you click the *Paste* button, the array is read from the clipboard and displayed in a message box. The code is as shown below, where I have only shown that part of the code, which is different:

```
public class MainView extends JFrame implements ClipboardOwner
{
 private static Random rand = new Random();
 private Clipboard cb = Toolkit.getDefaultToolkit().getSystemClipboard();

 public MainView()
 {
  …
 }

 private void createView()
 {
  …
 }

 @Override
 public void lostOwnership(Clipboard clipboard, Transferable contents)
 {
  JOptionPane.showMessageDialog(this, "The clipboard data is invalid");
 }

 private void copy(ActionEvent e)
 {
  cb.setContents(new DataHandler(create(10), "application/octet-stream"), this);
 }
```

```
  private void paste(ActionEvent e)
  {
   try
   {
    Transferable data = cb.getContents(null);
    DataFlavor flavor = data.getTransferDataFlavors()[0];
    show((byte[])data.getTransferData(flavor));
   }
   catch (Exception ex)
   {
    JOptionPane.showMessageDialog(this, ex.toString());
   }
  }

  private byte[] create(int n)
  {
   byte[] arr = new byte[n];
   rand.nextBytes(arr);
   return arr;
  }

  private void show(byte[] arr)
  {
   StringBuilder builder = new StringBuilder();
   for (byte b : arr)
   {
    builder.append(b);
    builder.append(' ');
   }
   JOptionPane.showMessageDialog(this, builder.toString());
  }
}
```

First, note that the class now implements the interface *ClipboardOwner*. This interface defines a single method called *lostOwnership()*, and this method is performed if you have saved something on the clipboardy and another program overwrites it. In this case nothing happens other than a message box appears. It is easy to test this feature by running the program and saving something on the clipboard (press the *Copy* button). If you then open another program and from here save something on the clipboard, you will notice that the message box is displayed.

To the end of the program there are two auxiliary methods in which the first creates a byte array with random byte values, while the other displays a message box containing the content of such an array.

Then there are the events handlers. The method *copy()* uses the type *DataHandler* that is a *Transferable*. In addition to encapsulating data, such an object will also have a *MIME* type and can therefore be used to transfer a variety of data types. In this case, the generic MIME type has been used, which states that raw data is being transmitted, which is not to be interpreted. Also note the last parameter for *setContents()*, where *this* is a *ClipboardOwner*. The method *paste()* has to retrieve data with *getContents()*, but you must specify a *DataFlavor*, and the result is a *Transferable* object. From this object, the first *DataFlavor* (there is only one) is used to retrieve data from the *Transferable* object as a byte array.

6.2 SERIALIZING OBJECTS

It is also possible to save objects on the clipboard by serialization, and I show how to do so in the next example. I want to save objects of the following type:

```
class King implements Serializable
{
 private String name;
 private int from;
 private int to;
```

```
 public King(String name, int from, int to)
 {
  this.name = name;
  this.from = from;
  this.to = to;
 }

 public String getName()
 {
  return name;
 }

 public int getFrom()
 {
  return from;
 }

 public int getTo()
 {
  return to;
 }

 public String toString()
 {
  return name + String.format(" [%d – %d]", from, to);
 }
}
```

The class requires no explanation, but you should note that it is *Serializable*, as well as that there are instance variables, both of the type *String* and simple types. To transfer data, you must use a *Transferable*, and here are several options, but as an example, I will show you how to implement your own *Transferable* type. *Transferable* is an interface that defines three methods, and the task is therefore to write a class that implements this interface:

```
class SerialTransferable implements Transferable
{
 private Serializable obj;

 SerialTransferable(Serializable obj)
 {
  this.obj = obj;
 }
```

```
 @Override
 public DataFlavor[] getTransferDataFlavors()
 {
  DataFlavor[] flavors = new DataFlavor[2];
  try
  {
   flavors[0] = new DataFlavor("application/x-java-serialized-object;
     class=" + obj.getClass().getName());
   flavors[1] = DataFlavor.stringFlavor;
   return flavors;
  }
  catch (ClassNotFoundException e)
  {
   return new DataFlavor[0];
  }
 }

 @Override
 public boolean isDataFlavorSupported(DataFlavor flavor)
 {
  return DataFlavor.stringFlavor.equals(flavor) ||
   "application".equals(flavor.getPrimaryType()) &&
   "x-java-serialized-object".equals(flavor.getSubType()) &&
   flavor.getRepresentationClass().isAssignableFrom(obj.getClass());
 }

 @Override
 public Object getTransferData(DataFlavor flavor)
  throws UnsupportedFlavorException
 {
  if (!isDataFlavorSupported(flavor))
   throw new UnsupportedFlavorException(flavor);
  if (DataFlavor.stringFlavor.equals(flavor)) return obj.toString();
  return obj;
 }
}
```

The class has a reference to the object to be serialized, which is initialized in the constructor. The first method must return the *DataFlavor* objects that may be used and defines two. The first is defined as the MIME type for serializing an object:

```
"application/x-java-serialized-object; class=" + obj.getClass().getName()
```

where the type is *application*, while the subtype is *x-java-serialized-object*. Finally, a parameter, which is the object type, is associated. The second *DataFlavor* is a *DataFlavor.stringFlavor*, which means that the contents of the clipboard can be interpreted as a string. You must note the sequence of the two *DataFlavor* objects, which means that the *DataFlavor* object for serialization is the primary.

The next method should test if a given *DataFlavor* can be used to retrieve this object, and it may be if it is a *DataFlavor.stringFlavor* or if the type and subtype are correct and that is the correct class type. Finally, there is the last method, which from a *DataFlavor* returns the encapsulated data object. If it is *DataFlavor.stringFlavor*, the result is returned as the object's *toString()* and if it is otherwise a legal *DataFlavor* it is the object itself.

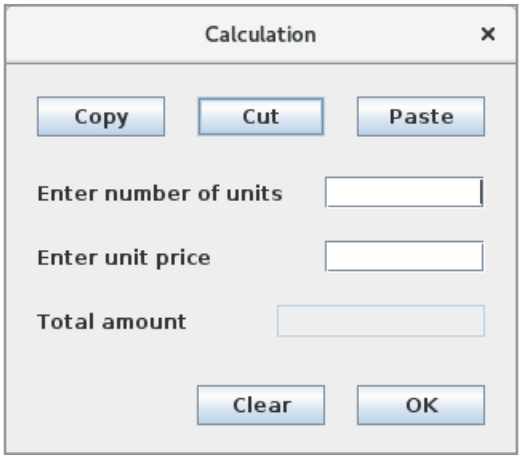
Then there is the program and it is in principle the same program as in the first two examples of this chapter:

```
public class MainView extends JFrame
{
 private static Random rand = new Random();
 private King[] kings = {
  new King("Gorm den Gamle", 936, 958),
  new King("Harald Blåtand", 958, 987),
  new King("Svend Tveskæg", 987, 1014) };
 private Clipboard cb = Toolkit.getDefaultToolkit().getSystemClipboard();

 public MainView()
 {
  …
 }

 private void createView()
 {
  …
 }

 private void copy(ActionEvent e)
 {
  cb.setContents(
   new SerialTransferable(kings[rand.nextInt(kings.length)]), null);
 }

 private void paste(ActionEvent e)
 {
  try
  {
   DataFlavor flavor = new DataFlavor("application/x-java-serialized-object;
    class=clipobject.King");
   if (cb.isDataFlavorAvailable(flavor)) show((King)cb.getData(flavor));
  }
  catch (Exception ex)
  {
   JOptionPane.showMessageDialog(this, ex.toString());
  }
 }

 private void show(King king)
 {
  JOptionPane.showMessageDialog(this, String.format("%s\n%d – %d",
   king.getName(), king.getFrom(), king.getTo()));
 }
}
```

When an object is saved on the clipboard, it is a random *King* object, and you must note that it is encapsulated in a *Transferable* object of the type *SerialTransferable*. When the object is retrieved in the method *paste()*, a *DataFlavor* of the correct type is defined. If the object can be converted with this *DataFlavor*, it is retrieved from the clipboard using the method *getData()*, which is a method that does not return the *Transferable* object on the clipboard, but the data that the object encapsulates.

EXERCISE 4

You must write an application that opens the window as shown below. You can just make a simple design. My window has a fixed size, and the components are laid out without the use of a layout manager and hence in fixed positions.



In the first entry field, enter a number of units (an *int*), and in the second entry field an unit price (a *double*). If you click the *OK* button, the program must calculate the total amount as the number of units multiplied by the unit price and inserts the result in the bottom field. Clicking the *Clear* button will clear all three fields. If you click the *Copy* button, the contents of the two top fields must be used to instantiate an object of the following type:

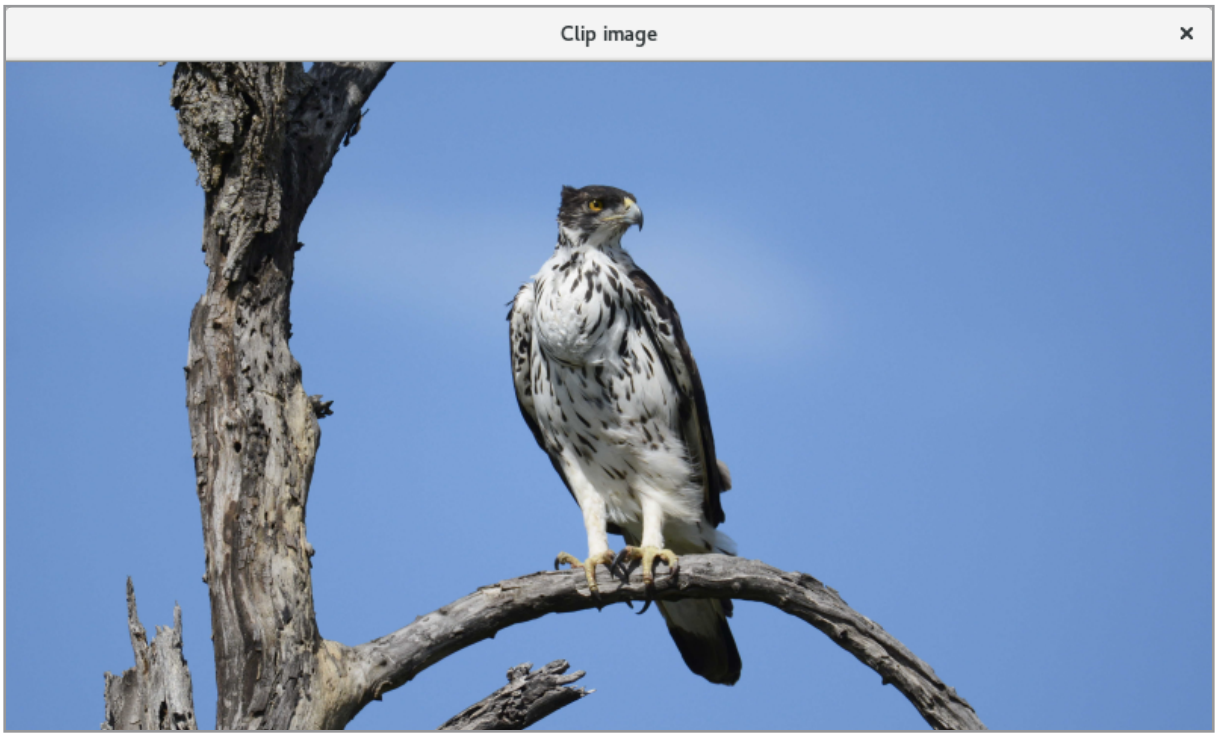
```
class Values implements Serializable
{
 private int units;
 private double price;

 public Values(int units, double price)
 {
  this.units = units;
  this.price = price;
 }
```

```
  public int getUnits()
  {
   return units;
  }

  public double getPrice()
  {
   return price;
  }
 }
```

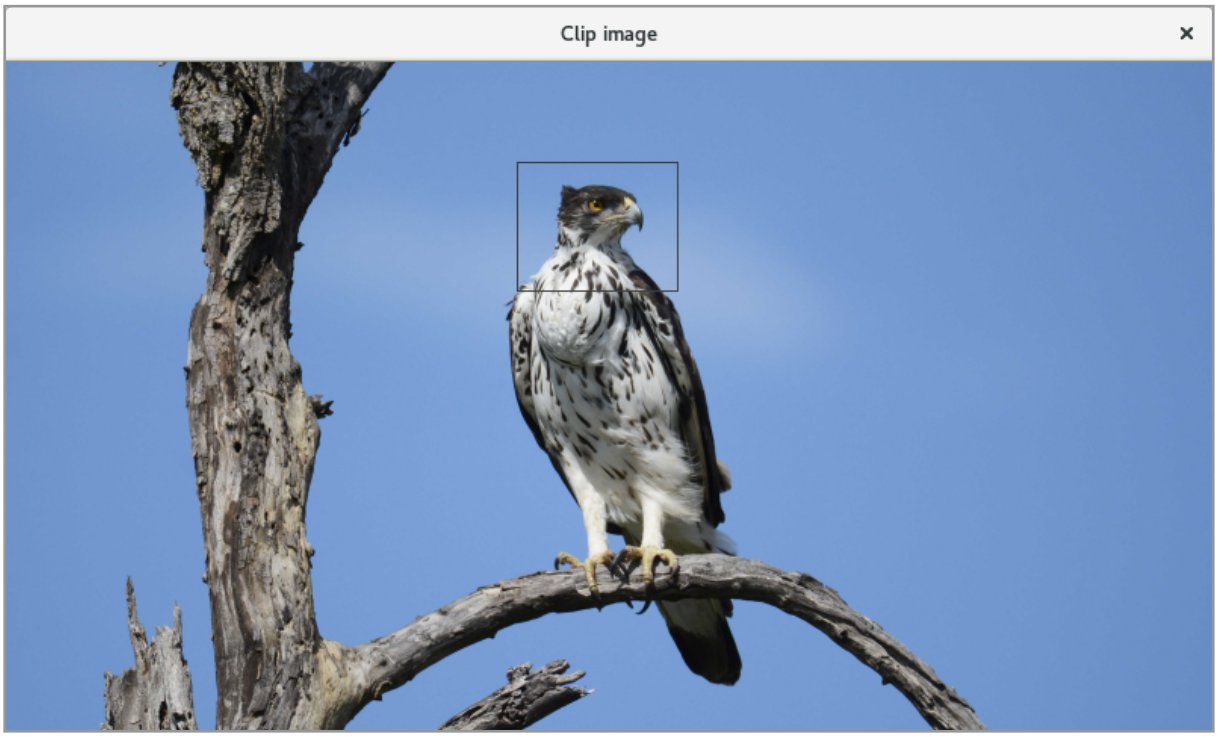
and the object must be saved on the clipboard. The *Cut* button should, in principle, work in the same way, but it should also delete the two top fields. Finally, the *Paste* button must initialize the two top fields with the values of a *Values* object was saved on the clipboard – if there is such an object.

6.3 IMAGES ON THE CLIPBOARD

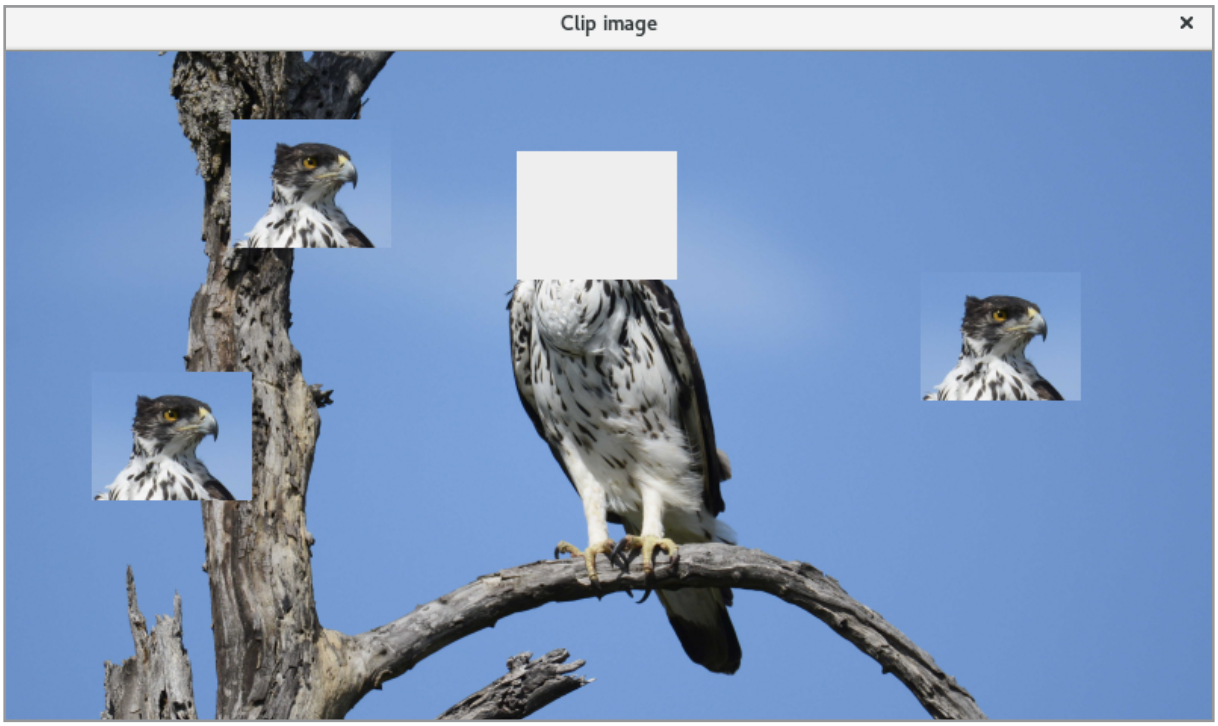
I want to close this chapter about the clipboard with an application that saves a portion of an image on the clipboard. It is a program where you can mark a rectangular area of a picture that you can save on the clipboard and paste it into the picture again, but it requires some knowledge about image processing that is dealt with in the next book so you may ignore the details of the processing of images. The important thing here is how to copy image data to and from the clipboard and not so much the program works. If you open the program you get a window as shown below:



where the image is copied to the project and is part of the program's jar file. If you point somewhere in the picture, hold down and move the mouse are drawn a rectangle showing what is marked:



If you right-click a location on the image, you get a popup menu with three menu items for copy, cut and paste. The following window shows the result where I have selected cut and subsequently pasted the head of the eagle in three places:



You should note that the image is not located in a *JScrollPane*, and the reason is that it complicates the program a bit, as it becomes harder to calculate where the rectangle is to be drawn.

The program looks a bit like the previous program corresponding to a data structure defined for the pixels to be saved on the clipboard:

```
class ImageData implements Serializable
{
 private int width;
 private int height;
 private int[] pixels;

 public ImageData(int width, int height, int[] pixels)
 {
  this.width = width;
  this.height = height;
  this.pixels = pixels;
 }
```

```
 public int getWidth()
 {
  return width;
 }

 public int getHeight()
 {
  return height;
 }

 public int[] getPixels()
 {
  return pixels;
 }
}
```

The data structure consists of an array for the rectangle of pixels and the width and height of the rectangle. In the same way as in the previous program I have to define a *Transferable*:

```
class ImageTransferable implements Transferable
{
 public final static DataFlavor IMAGE_FLAVOR =
  new DataFlavor (clipimage.ImageData.class, "Image Data");
 private final static DataFlavor [] flavors = { IMAGE_FLAVOR };
 private ImageData data;

 public ImageTransferable(ImageData data)
 {
  this.data = data;
 }

 public DataFlavor [] getTransferDataFlavors()
 {
  return flavors;
 }

 public boolean isDataFlavorSupported(DataFlavor flavor)
 {
  return flavor.equals(IMAGE_FLAVOR);
 }
```

```
 public Object getTransferData(DataFlavor flavor)
  throws UnsupportedFlavorException
 {
  if (!flavor.equals(IMAGE_FLAVOR)) throw new UnsupportedFlavorException(flavor);
  return data;
 }
}
```

First, a *DataFlavor* is defined, but this time it is not defined using a MIME type, but by specifying the class of the type of objects that can be transferred to the clipboard as well as a name. The *Transferable* object must this time only support this single *DataFlavor*, and the array returned by *getTransferDataFlavors()* is defined as a static array. The implementation of the class' methods are trivial.

Then there is the program itself and the code fills a lot. A good part of the code involves drawing the rectangle as well as creating an array of pixels that form a rectangular area of the image and how to copy stored pixels into the image. The latter happens with two methods called *getPixels()* and *setPixels()* respectively. I do not want to display the code for these methods here, but shortly the program works as follows: The image is displayed in a *JLabel* that is anchored in the upper left corner of the window. If you press the left mouse button, the mouse's coordinates are saved as the upper left corner of the rectangle, and when you drag the mouse, the mouse's coordinates are saved as the bottom right corner of the rectangle, after which the rectangle is drawn. Right-clicking anywhere in the window will opens a pop up menu with the three menu items. All that, requires some code, but I would like only to show the three event handlers to the popup menu. Below is the method *copy()*:

```
private Rectangle copy()
{
 Rectangle rect = getSelected();
 int[] pixels = getPixels(rect);
 ImageData data = new ImageData(rect.width, rect.height, pixels);
 Clipboard cb = getToolkit().getSystemClipboard();
 cb.setContents(new ImageTransferable(data), null);
 return rect;
}
```

The method starts by determining the rectangle that is selected, which is done using a *getSelected()* method. Next, the *getPixels()* method is used to form your array with all image pixels within this rectangle, and then creates the object to be saved on the clipboard. The object has the type *ImageData*, and it is now saved on the clipboard by embedding it in an *ImageTransferable* object. When the method returns the rectangle, it is because it is also used in the *cut()* method:

```
private void cut()
{
 Rectangle rect = copy();
 int[] pix = getPixels(rect);
 for (int i = 0; i < pix.length; ++i) pix[i] = 0;
 setPixels(pix, rect);
}
```

cut() must start with a *copy()* and then the area saved must be blanked by setting all pixels to 0. The actual image area is updated using the *setPixels()* method. Back there is the *paste()* method:

```
private void paste()
{
 Clipboard cb = getToolkit().getSystemClipboard();
 try
 {
  Transferable tf = cb.getContents(null);
  if (tf.isDataFlavorSupported(ImageTransferable.IMAGE_FLAVOR))
  {
   ImageData data =
    (ImageData)(tf.getTransferData(ImageTransferable.IMAGE_FLAVOR));
   Rectangle area =
    new Rectangle(point.x, point.y, data.getWidth(), data.getHeight());
   int[] pixels = data.getPixels();
   setPixels(pixels, area);
   end = start;
   repaint();
  }
 }
 catch (Exception ex)
 {
  JOptionPane.showMessageDialog(this, ex.toString());
 }
}
```

The method starts by referring to the contents of the clipboard as a *Transferable* object. If this object supports the correct *DataFlavor*, data is retrieved as an *ImageData* object and based on it, a rectangle is created for the pixels to be updated. The actual update is done with the method *setPixels()*. When the image is finally redrawn, the rectangle that is selected must be removed.

7 DRAG AND DROP

Drag and drop is an operation where data using the mouse are moved from one position to another position. In fact, most Swing components come with built-in features for drag and drop, and it is thus relatively simple to implement this feature in a program. The principle is a bit like the clipboard that the data to be moved must be encapsulated in a *Transferable* object, and in addition, three interfaces must be implemented, one of which defines the component on which to perform a move, the other the component where you must be able to drop and finally an object indicating the visual representation of the operation. It sounds like a lot, but it's because you have high degrees of liberty as to how the operation is to be performed, and in practice it is actually limited what is needed.

Drag and drop operation includes three functions

1. copy (hold the left button down and move the mouse)
2. move (hold the left button down and move the mouse + CTRL)
3. link (hold the left button down and move the mouse + CTRL + SHIFT)

where the last means that an object is not copied, but only a reference to the original object is defined. These operations are defined as constants in the class *DnDConstants*:

1. *DnDConstants.ACTION_MOVE*
2. *DnDConstants.ACTION_COPY*
3. *DnDConstants.ACTION_REFERENCE*
4. *DnDConstants.ACTION_LINK*
5. *DnDConstants.ACTION_COPY_OR_MOVE*

The principle of drag and drop is that, with the mouse, one can drag one component over another component and then drop the first one. When Swing components supports drag and drop, it means that all the necessary logic about the mouse is built-in, but what's going to happen when dropping, Swing can of course not know and that's why something needs to be programmed. I want to start with an application that opens the following window:

The window contains three labels and the meaning is that using a drag-and-drop operation you can change the background color of the top label. Below is an example where the background color has been changed three times (the number 3 is displayed in a *JLabel* and the meaning only is to show whar the interfaces are used for as part of the drag and drop API):

The full code is shown below:

```
package dropswing;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.datatransfer.*;
import java.awt.dnd.*;
import java.io.*;

public class MainView extends JFrame
{
 public final static DataFlavor LABEL_FLAVOR =
  new DataFlavor(JLabel.class, "Label Instances");
 private DragSourceListener dragSource;
 private JPanel panel = new JPanel(null);
 private JLabel dropLabel;
 private JLabel lblCount = new JLabel();
 private int count = 0;

 public MainView()
 {
  super("Drag and drop");
  setSize(260, 250);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
```

```
private void createView()
{
 addComponent(dropLabel = createLabel(), 20, 20, 220, 50);
 addComponent(createLabel(Color.red), 20, 100, 60, 30);
 addComponent(createLabel(Color.green), 100, 100, 60, 30);
 addComponent(createLabel(Color.blue), 180, 100, 60, 30);
 addComponent(lblCount, 100, 140, 60, 60);
 lblCount.setHorizontalAlignment(JLabel.CENTER);
 lblCount.setFont(new Font("Liberation Serif", Font.BOLD, 36));
 new DropTarget(dropLabel, DnDConstants.ACTION_COPY, new DnDDrop());
 dragSource = new DnDSource();
 add(panel);
}

private JLabel createLabel()
{
 JLabel label = new JLabel("Drop things here");
 label.setOpaque(true);
 label.setHorizontalAlignment(JLabel.CENTER);
 label.setFont(new Font("Liberation Serif", Font.BOLD, 24));
 return label;
}

private JLabel createLabel(Color color)
{
 JLabel label = new JLabel();
 label.setOpaque(true);
 label.setBackground(color);
 DragSource.getDefaultDragSource().createDefaultDragGestureRecognizer(label,
  DnDConstants.ACTION_COPY, new DnDGesture());
 return label;
}

private void addComponent(Component component, int xpos, int ypos,
 int width, int height)
{
 component.setBounds(xpos, ypos, width, height);
 panel.add(component);
}

class DnDGesture implements DragGestureListener
{
 public void dragGestureRecognized(DragGestureEvent e)
 {
  Cursor cursor = null;
  JLabel label = (JLabel)(e.getComponent());
  switch (e.getDragAction())
  {
```

```
    case DnDConstants.ACTION_MOVE: cursor = DragSource.DefaultMoveDrop; break;
    case DnDConstants.ACTION_COPY: cursor = DragSource.DefaultCopyDrop; break;
    case DnDConstants.ACTION_LINK: cursor = DragSource.DefaultLinkDrop; break;
   }
   e.startDrag(cursor, new LabelTransferable(label), dragSource);
  }
 }

 class DnDSource implements DragSourceListener
 {
  public void dragEnter(DragSourceDragEvent e)
  {
  }

  public void dragExit(DragSourceEvent e)
  {
  }

  public void dragOver(DragSourceDragEvent e)
  {
  }
```

```
 public void dropActionChanged(DragSourceDragEvent e)
 {
 }

 public void dragDropEnd(DragSourceDropEvent e)
 {
  if (e.getDropSuccess()) lblCount.setText(String.format("%d", ++count));
 }
}

class DnDDrop implements DropTargetListener
{
 public void dragEnter(DropTargetDragEvent e)
 {
  if (!e.isDataFlavorSupported(LABEL_FLAVOR)) e.rejectDrag();
  else dropLabel.setBorder(new LineBorder(Color.black));
 }

 public void dragExit(DropTargetEvent e)
 {
  dropLabel.setBorder(null);
 }

 public void dragOver(DropTargetDragEvent e)
 {
 }

 public void dropActionChanged(DropTargetDragEvent e)
 {
 }

 public void drop(DropTargetDropEvent e)
 {
  try
  {
   if (e.isDataFlavorSupported(LABEL_FLAVOR))
   {
    dropLabel.setBackground(((JLabel)e.getTransferable().getTransferData(
     LABEL_FLAVOR)).getBackground());
    dropLabel.setBorder(null);
    e.dropComplete(true);
  }
  }
```

```
    catch (Exception ex)
    {
     e.dropComplete(false);
    }
   }
  }

  class LabelTransferable implements Transferable
  {
   private DataFlavor[] flavors = { LABEL_FLAVOR };
   private JLabel label;

   public LabelTransferable(JLabel label)
   {
    this.label = label;
   }

   public DataFlavor[] getTransferDataFlavors()
   {
    return flavors;
   }

   public boolean isDataFlavorSupported(DataFlavor flavor)
   {
    return flavor.equals(LABEL_FLAVOR);
   }

   public Object getTransferData(DataFlavor flavor)
    throws UnsupportedFlavorException, IOException
   {
    if (flavor.equals(LABEL_FLAVOR)) return label;
    throw new UnsupportedFlavorException(flavor);
   }
  }
 }
```

If you start from the top, drag and drop require two packages:

```
import java.awt.datatransfer.*;
import java.awt.dnd.*;
```

where the first is the same package as I mentioned in the previous chapter about the clipboard. The other package contains types that are required to implement drag and drop in an application. Regarding the program's variables, a constant is first defined as a *DataFlaver* for a *JLabel*. This corresponds to the objects that this program should be able to drag are *JLabel* components. In addition, a variable of the type *DragSourceListener* defines where an object representing the component to be dragged and which during the operation fires multiple events. The top variables relate to user interface components and do not require any particular explanation, but you should note that the panel has no layout manager and that the components are therefore are placed in fixed positions in the window. It's just to get rid of the design of the user interface, as it's not the primary in this example.

Drag and drop requires that you define four classes. They are explained below, but in this example they are all defined as inner classes. Apart from these classes, the code includes the design of the user interface, and here are three things that you should notice. In the method *createView()*, the following method is performed:

```
new DropTarget(dropLabel, DnDConstants.ACTION_COPY, new DnDDrop());
```

dropLabel is the *JLabel* that I want to drop on, and the statement tells me that it should be possible by attaching a *DropTargetListener* to the component. *DnDDrop* is an inner class that implements the *DropTargetListener* interface. *createView()* also performs the statement

```
dragSource = new DnDSource();
```

that creates a *DragSourceListener* (*dragSource* is an instance variable) and *DnDSource* is an inner class that implements the *DragSourceListener* interface. Finally, note the last *createLabel()* method that creates the three *JLabel* components that show a color (the components that must be draggable), which perform the following statement:

```
DragSource.getDefaultDragSource().createDefaultDragGestureRecognizer(label,
 DnDConstants.ACTION_COPY, new DnDGesture());
```

The statement links a *DragGestureListener* to the component. *DndGesture* is an internal class that implements the *DragGestureListener* interface, and the class initiates a drag operation. The sum of the above is that to implement drag and drop in a program one must

1. assign a *DropTargetListener* to the components, where you must be able to drop
2. defines a *DragSourceListener* object
3. assign a *DragGestureListener* to the components that must be draggable

Back there is writing classes implementing the three interfaces and as mentioned above and a class implementing *Transferable*.

I want to start with the last called *LabelTransferable* that implement the *Transferable* interface. The class's constructor has a *JLabel* as parameter and is the label included in a drag operation. As shown in the previous chapter, the class must implement three methods, and since there is only one *DataFlavor*, these methods are all trivial.

Then there is the *DnDSource* class that implements *DragSourceListener*. This interface defines 5 event handlers that fire events associated with a drag operation. Often you do not implements these methods (leave them blank), but I have implemented a single method that is performed when the operation is completed. If the drop operation was completed correctly, the method counts a counter by 1 and displays the value of the counter in the bottom JLabel. The goal is to show an example of how these event handlers can be used, and you are encouraged to check when other events occurs, but it appears mostly from the name.

The class *DnDDrop* implements *DropTargetListener*, which in the same way defines 5 event traders, but this time, events are triggered by the drop component. I have implemented three of them. The first means that a frame appears around the top label when the mouse in a drag and drop operation is entered over the component. It shows the user that it is a place where you can drop. The second event handler removes the frame again if the mouse no longer points to the component. Then there is event the handler *drop()*, which is performed when dropping – and that is, releasing the mouse. The method tests whether the object to be dropped has the correct *DataFlaver* (is a *JLabel* component), and if so, the following statement is performed:

```
dropLabel.setBackground(((JLabel)e.getTransferable().
 getTransferData(LABEL_FLAVOR)).getBackground());
```

The method takes the object that was transferred and converted it to a *JLabel*. Next, the background color of the component *dropLabel* is changed to the background color of the transferred object. As the next step, the method removes the frame and finally the ollowing statement is executed

```
e.dropComplete(true);
```

which tells that the drop operation is performed correctly.

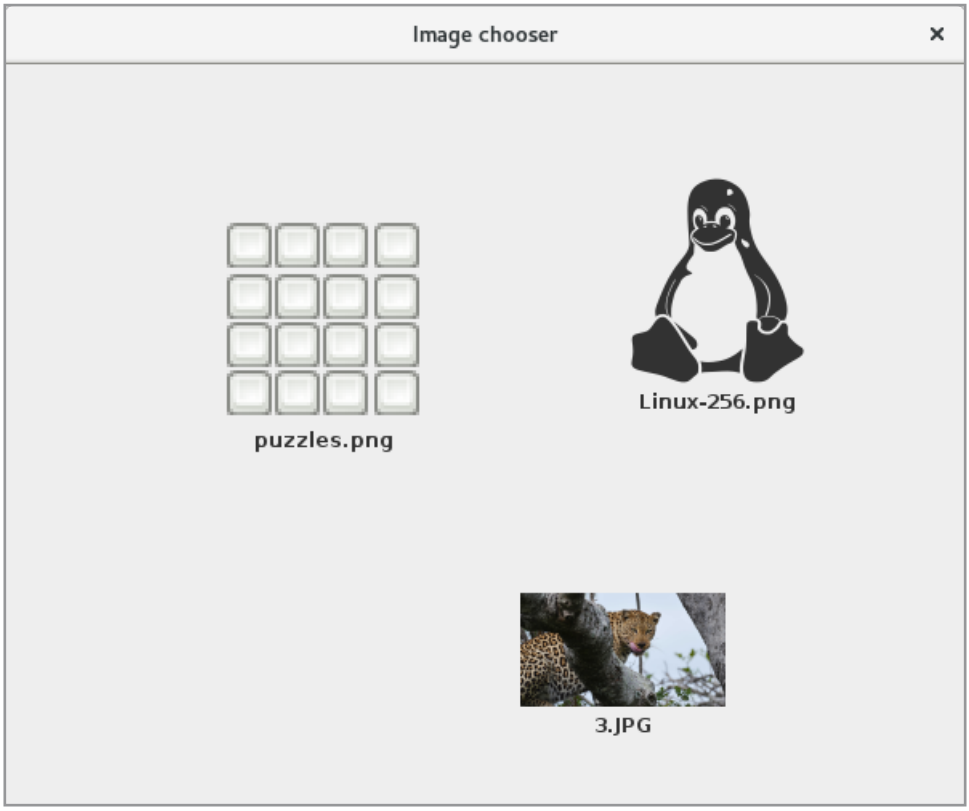
Finally, there is the class *DnDGesture* that implements the *DragGestureListener* interface. This interface defines only a single method (event handler), which is a method performed when the drag and drop operation starts. It is actually the method that binds it all together. The method starts to determine a reference to the label the drag operation concerns. Then, the method changes the cursor so there is a visual effect of the operation. Finally, the method starts the operation by specifying the selected cursor as well as encapsulating the current object in a *Transferable* object.

As shown above, drag and drop requires a part, but it is the same thing that should happen every time.

7.1 DRAG IMAGES

The above is not a typical drag and drop operation and therefore the following example, which shows better how to use drag and drop in a program. In fact, the program does not contain much new and drag and drop is in the same way as in the above example implemented using 4 inner classes. If you open the program, you get a blank window:

If you opens *Files*, you can drag pictures (JPG, GIF and PNG files) into the window, then they appear as 128×128 icons:



If the images are copied to the window, you can drag them with the mouse to another location.

The program is similar to the above, but this time there are two drag and drop operations. You can point to a file (an image) in *Files* and drag it over the application window. That is, drag operations are initiated in another program (the program *Files*), while the drop operation is performed in my program. The other operation, where you can move the images, relates solely to the Java program, where the drag operation is started based on an object in the program window, and the drop operation is performed on the same object as above.

The program's window is design very simple and consists only of a *JPanel* without a layout manger. The panel is called *panel*. In *createView()* this panel is defined as *DropTarget* so it receives events from the *DnDDrop* class. Here is the important method

```
public void drop(DropTargetDropEvent e)
{
 try
 {
  if (e.isDataFlavorSupported(DataFlavor.javaFileListFlavor))
  {
   e.acceptDrop(DnDConstants.ACTION_COPY);
   java.util.List list = (java.util.List)(e.getTransferable().
    getTransferData(DataFlavor.javaFileListFlavor));
   for (Object obj : list) addComponent(getImage((File)obj), e.getLocation());
   e.dropComplete(true);
  }
 }
 catch (Exception ex)
 {
  e.dropComplete(false);
 }
}
```

If you drag a file from the outside (that is from the *Files* program), the object type is defined by a *DataFlavor* named *javaFileListFlavor*. This means that the *Transferable* object encapsulates an *ArrayList*, and the methods refer to this list. The list's objects have the type of *File*, and for each object, the method *addComponent()* is called, which adds a *JLabel* to the panel where the mouse points. The method starts by calling a method *getImage()* with a *File* object as parameter. This method loads the image using the filename and, and on basis of this, creates an icon of *128×128* pixels. However, only if it is a jpg, png or gif file. Otherwise, a default icon will be used. The program thus supports dragging random files into the window.

```
private JLabel getImage(File file)
{
 String name = file.getAbsolutePath().toLowerCase();
 ImageIcon icon = name.endsWith("png") || name.endsWith("jpg") ||
  name.endsWith("gif") ? scaleImage(new ImageIcon(file.getAbsolutePath())) :
  defIcon;
 JLabel label = new JLabel(icon);
 label.setText(cut(file.getName()));
 label.setHorizontalTextPosition(JLabel.CENTER);
 label.setVerticalTextPosition(JLabel.BOTTOM);
 return label;
}
```

```
private static ImageIcon scaleImage(ImageIcon icon)
{
double w = icon.getIconWidth();
 double h = icon.getIconHeight();
 double z = Math.min(128 / w, 128 / h);
 return new ImageIcon(icon.getImage().getScaledInstance(
  (int)(w * z), (int)(h * z), java.awt.Image.SCALE_SMOOTH));
}
```

After the icon is determined (created), a *JLabel* is created for the icon and at the same time the file name appears below the icon, and the icon is then added to the panel:

```
private void addComponent(Component component, Point point)
{
 component.setLocation(point);
 component.setSize(component.getPreferredSize());
 panel.add(component);
 repaint();
}
```

Then you can actually drag files into the program and drop them in the window. That is, as long that none of the class's components should be draggable, it is not necessary to implement the three interfaces *DragSourceListener*, *DragGestureListener* and *Transferable*.

If you also need to be able to drag the components in the window, the three interfaces must be implemented. Since it must be possible to draw a *JLabel*, the *Transferable* object can be implemented in exactly the same way as in the previous program. Then there is the *DnDSource* class, which implements the *DragSourceListener* interface, where it is necessary to implement the *dragDropEnd()* method:

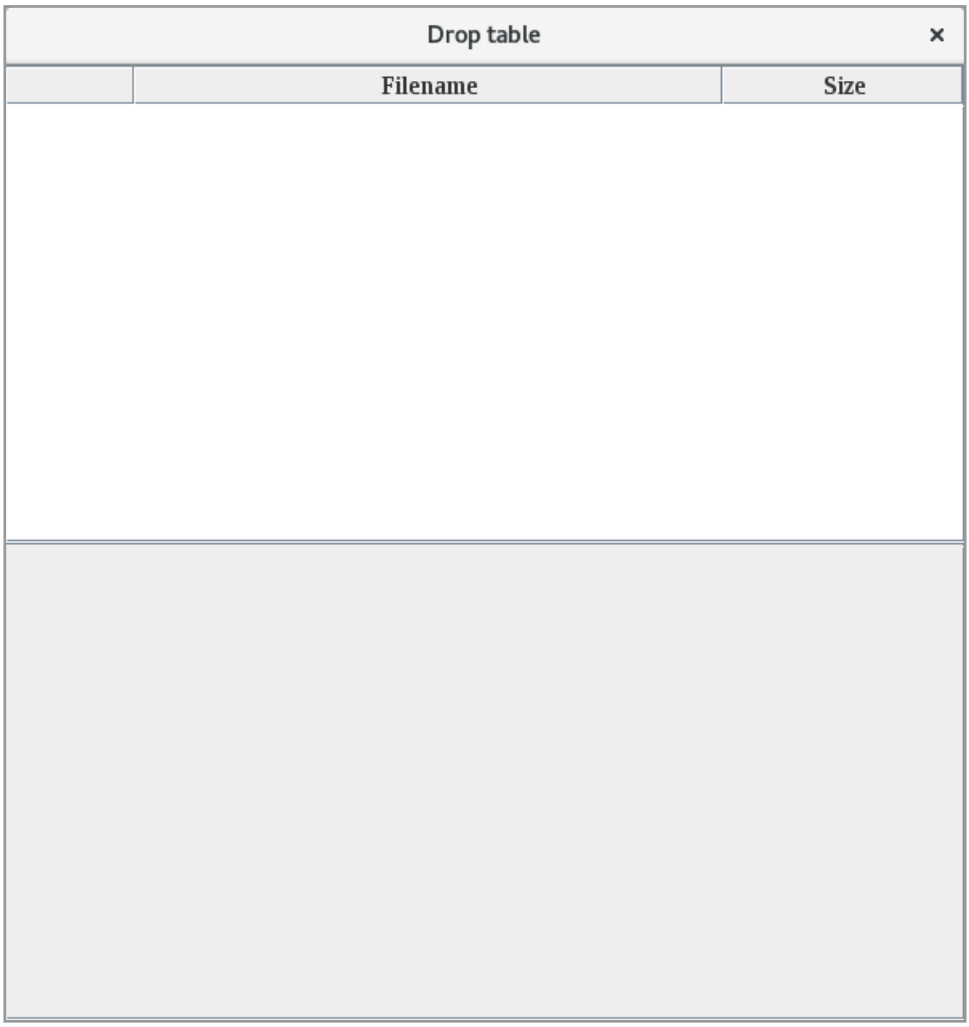
```
public void dragDropEnd(DragSourceDropEvent e)
{
 if ((e.getDropSuccess()) && (e.getDropAction() == DnDConstants.ACTION_MOVE))
 {
  panel.remove(dndLabel);
  panel.repaint();
 }
 dndLabel = null;
}
```

The reason is that once you have moved a component and dropped it, you have formed a copy and the original component must be removed.

Finally, there is the class *DnDGesture()* which is unchanged from the previous program.

PROBLEM 4

You should write a program, that you can call *DropTable*. The program must open the following Window:

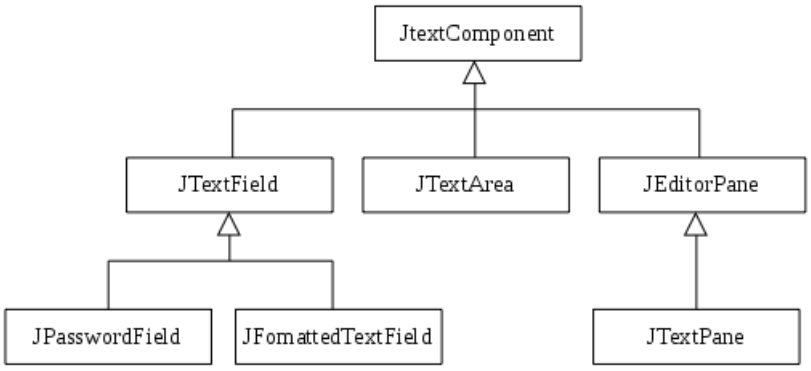


At the top there is a *JTable* in a *JScrollPane*, while at the bottom there is a *JPanel* without a layout manager. Just like in the previous program, you can drag files that represent images from the file system (the program *Files*) and drop them onto a *JTable*, and each file inserts a row in the table. You can also take a row in the table and drag it down over the bottom panel and drop it there, and the result will be a label with an icon. That is, the *JTable* component supports both drag and drop. Below is a window in which there are droped 5 icons (images) in the *JTable* component, and three of these are by drag dropped in the bottom panel:

8 EDIT TEXT

In practice programming, editing of texts plays an important role, and in most contexts it is without the big challenges, as Swing provides more components that can be used for text input. So far, I have looked at the components *JTextField* and *JTextArea*, which are used for entering single text lines and entering multiple lines. As the previous examples have shown, it is simple to use these components, but there are actually four additional components for text input. One of them is *JPasswordField*, which I have already used once, and as the name says is used for entering a password. Basically, it is a *JTextField*, but with the difference that the characters entered do not appear, but the field only shows a dot for each entered character (if you do not indicate that nothing should be displayed at all).

Another input field is a *JFormattedTextField*, which is also a *JTextField*, but where you specify how to format the entered data. For example, you can enter an integer only. As an alternative to a *JTextArea*, there is also a *JTextPane* where you can edit styled text and, for example, decide which colors and fonts to use for parts of the text. Finally, there is a *JEditorPane*, which uses a so-called *EditorKit* to define how text is displayed and edited. These components represents a hierarchy as follows:



Please note that *JTextComponent* is a common basic class for the 6 concrete components for text input.

In this chapter I will look at the components *JFomattedTextField*, *JTextPane* and *JEditorPane* as well as a number of classes used by these components. The components are by no means simple, but on the other hand they provide the programmer with very great flexibility in working with text. When editing text is complex, it is because components should handle keyboard and mouse events, and that components should be able to show styled text. However, before I start, I will show a program that opens a window, as shown below, where the program illustrates a little about how the components work.

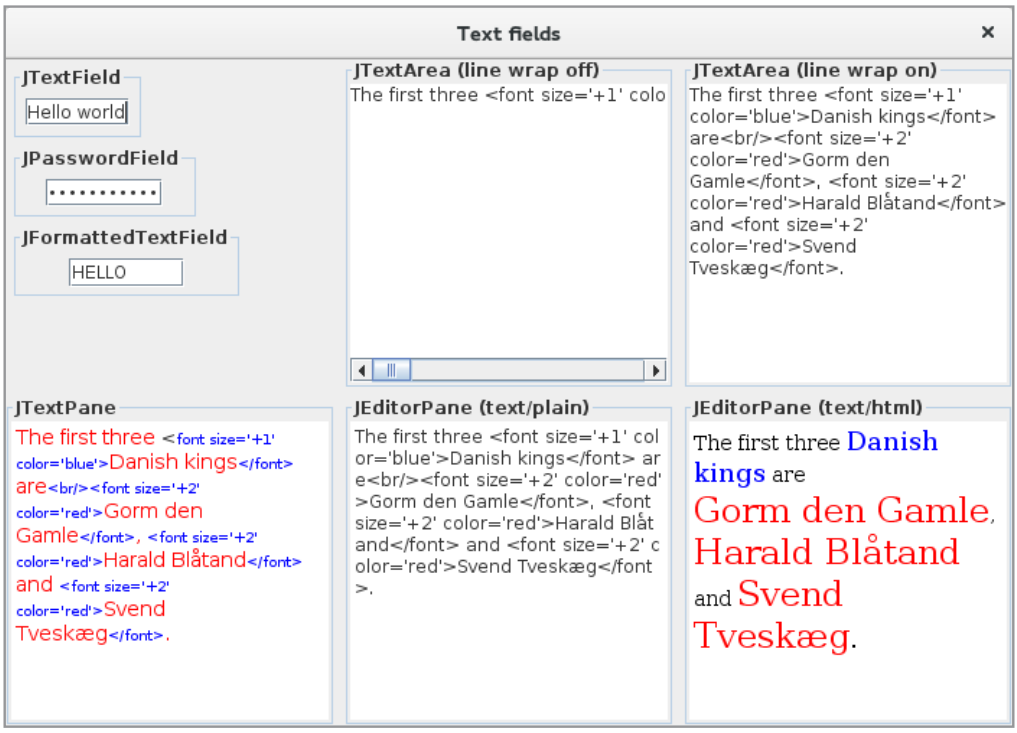
```
package textfields;

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

public class MainView extends JFrame
{
 public static final String text1 = "Hello world";
 public static final String text2 =
  "The first three <font size='+1' color='blue'>Danish kings</font> are
  <br/><font size='+2' color='red'>Gorm den Gamle</font>,
  <font size='+2' color='red'>Harald Blåtand</font> and
  <font size='+2' color='red'>Svend Tveskæg</font>.";

 public MainView()
 {
  super("Text fields");
  setSize(700, 500);
  setLayout(new FlowLayout());
```

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

private void createView()
{
 setLayout(new GridLayout(2, 3, 5, 5));
 add(createTextFields());
 add(createTextAre1());
 add(createTextAre2());
 add(createTextPane());
 add(createEditorPane1());
 add(createEditorPane2());
}

private JScrollPane createEditorPane1()
{
 JScrollPane scroll = new JScrollPane(new JEditorPane("text/plain", text2));
 scroll.setBorder(new TitledBorder("JEditorPane (text/plain)"));
 return scroll;
}

private JScrollPane createEditorPane2()
{
 JScrollPane scroll = new JScrollPane(new JEditorPane("text/html", text2));
 scroll.setBorder(new TitledBorder("JEditorPane (text/html)"));
 return scroll;
}

private JScrollPane createTextPane()
{
 JTextPane field = new JTextPane();
 field.setText(text2);
 SimpleAttributeSet attrs1 = new SimpleAttributeSet();
 StyleConstants.setForeground(attrs1, Color.red);
 StyleConstants.setFontSize(attrs1, 14);
 SimpleAttributeSet attrs2 = new SimpleAttributeSet();
 StyleConstants.setForeground(attrs2, Color.blue);
 StyleConstants.setFontSize(attrs2, 10);
 StyledDocument sdoc = field.getStyledDocument( );
 sdoc.setCharacterAttributes(0, 16, attrs1, false);
 sdoc.setCharacterAttributes(17, 28, attrs2, false);
 sdoc.setCharacterAttributes(45, 12, attrs1, false);
 sdoc.setCharacterAttributes(57, 7, attrs2, false);
```

```
 sdoc.setCharacterAttributes(64, 4, attrs1, false);
 sdoc.setCharacterAttributes(68, 33, attrs2, false);
 sdoc.setCharacterAttributes(101, 14, attrs1, false);
 sdoc.setCharacterAttributes(115, 7, attrs2, false);
 sdoc.setCharacterAttributes(122, 2, attrs1, false);
 sdoc.setCharacterAttributes(124, 28, attrs2, false);
 sdoc.setCharacterAttributes(152, 14, attrs1, false);
 sdoc.setCharacterAttributes(166, 7, attrs2, false);
 sdoc.setCharacterAttributes(173, 5, attrs1, false);
 sdoc.setCharacterAttributes(178, 28, attrs2, false);
 sdoc.setCharacterAttributes(206, 14, attrs1, false);
 sdoc.setCharacterAttributes(219, 7, attrs2, false);
 sdoc.setCharacterAttributes(226, 1, attrs1, false);
 JScrollPane scroll = new JScrollPane(field);
 scroll.setBorder(new TitledBorder("JTextPane"));
 return scroll;
}

private JScrollPane createTextAre1()
{
 JScrollPane scroll = new JScrollPane(new JTextArea(text2));
 scroll.setBorder(new TitledBorder("JTextArea (line wrap off)"));
 return scroll;
}
```

```
private JScrollPane createTextAre2()
{
 JTextArea field = new JTextArea(text2);
 field.setLineWrap(true);
 field.setWrapStyleWord(true);
 JScrollPane scroll = new JScrollPane(field);
 scroll.setBorder(new TitledBorder("JTextArea (line wrap on)"));
 return scroll;
}

private JPanel createTextFields()
{
 JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
 panel.add(createTextField());
 panel.add(createPasswordField());
 panel.add(createFormattedField());
 return panel;
}

private JPanel createTextField()
{
 JPanel panel = new JPanel(new FlowLayout());
 panel.setBorder(new TitledBorder("JTextField"));
 panel.add(new JTextField(text1));
 return panel;
}

private JPanel createPasswordField()
{
 JPanel panel = new JPanel(new FlowLayout());
 panel.setBorder(new TitledBorder("JPasswordField"));
 panel.add(new JPasswordField(text1));
 panel.setPreferredSize(new Dimension(130, 50));
 return panel;
}

private JPanel createFormattedField()
{
 JFormattedTextField field = new JFormattedTextField(createFormatter());
 field.setValue(text1);
 field.setPreferredSize(new Dimension(80, 20));
 JPanel panel = new JPanel(new FlowLayout());
 panel.setBorder(new TitledBorder("JFormattedTextField"));
 panel.add(field);
 panel.setPreferredSize(new Dimension(160, 50));
 return panel;
}
```

```
  private MaskFormatter createFormatter()
  {
   try
   {
    return new MaskFormatter("UUUUU");
   }
   catch (Exception ex)
   {
    return null;
   }
  }
}
```



The program divides the window into 6 areas using a *GridLayout*. In the upper left corner, a *JTextField*, a *JPasswordField* and a *JFormattedTextField* are displayed. The first two, I should not mention, but the third is created in the method *createFormattedField()*, and it is important that it is initialized with a *MaskFormatter* object created in the method *createFormatter()*. *MaskFormatter* is a class that allows to define a particular pattern that the content of a *JFormattedTextField* must adhere to. In this case, the pattern is "UUUUU", which means 5 uppercase letters. The field is initialized with the text "Hello world", and the result is that the field shows "HELLO" and thus the first 5 letters converted to uppercase letters. You should investigate what happens if you type in the field and note that only 5 characters can be entered, that lowercase letters are automatically converted to uppercase letters and all non-letter characters are ignored.

The *MaskFormatter* class has many options for formatting text, and you are encouraged to read the help to get an idea of the possibilities. There are also other classes for formatting text, and you should pay special attention to the class *NumberFormat*.

The window's top line shows two additional fields, that both are *JTextArea* fields. The first does not break the line and therefore the entire text is shown as one long line (unless you directly insert line breaks). The other *JTextArea* will automatically break lines that are to long. You should note that the text is defined as html, and since a *JTextArea* does not format the text, all characters are displayed without interpretation.

The field in the lower left corner shows a *JTextPane* created by the method *createTextPane()*. A *JTextPane* shows styled text, which means that you can insert attributes into the text that specify how the text is to be formatted. In this case, font size and color attributes are entered. You should study the method *createTextPane()* and here specifically how to insert attributes.

Finally, each of the last two fields in the bottom line shows a *JEditorPane*. Here you can specify how the text is to be rendered with a MIME type, and exactly that means which *EditorKit* to use. The first indicates *text / plain* and therefore shows the text as plain text, while the last uses *text / html* and thus interprets the html tags and renderes the text as an HTML document.

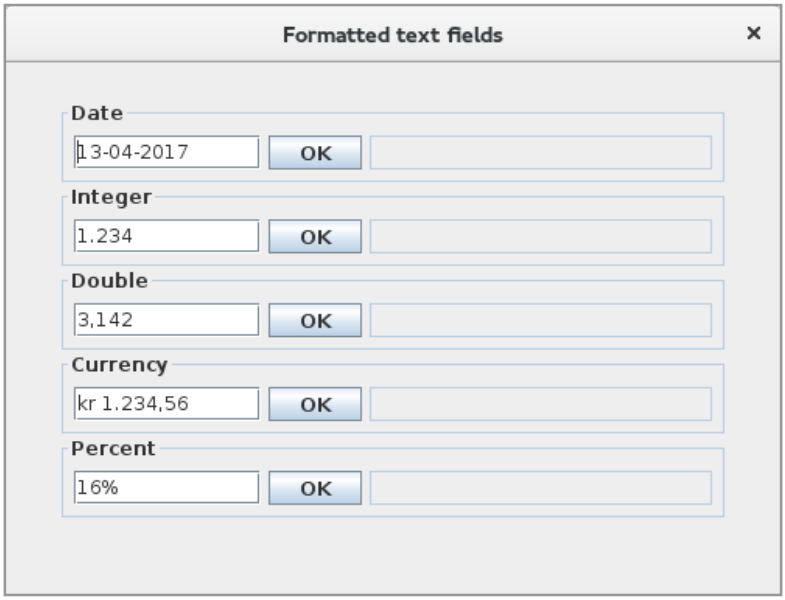
In principle, the 6 text components work in the same way that can be briefly described as: The component has a UI delegate, which, among other things, defines font and color properties. It also defines the component's caret and highlighter, as well as *InputMap* and *ActionMap* that relate to the keyboard and commands such as cut / copy / paste, select-all, caret-to-end-of-line, page-down, and so on. It is also the UI delegate that instantiates an *EditorKit*. For a *JTextArea*, it is a *DefaultEditorKit* that defines most of the component's *Actions*. A text component works on a *Document* (such as a *PlainDocument*) either created by the component or transferred to the constructor. The document object represents the text by dividing it into one or more *Elements.* An *Element* object represents a portion of the document's content as well as style information. For example, a *JTextArea* creates an *Element* object for each line, but ignores everything about styles. A text document registers itself as listening to the events it needs to keep track of, and it registers, among other things, as a *DocumentListener* so it can update itself if the *Document* object is changed. It is the UI delegate that is responsible for drawing the component using the component's *EditorKit* and a *ViewFactory*. This class creates a hierarchy of one or more *View* objects based on the *Element* objects of the document, and this hierarchy is drawn on the screen.

That story hardly tells much, but it mentions several classes, and it's all these classes that you can override and in that way get a text component to work as you like. These are the classes that makes Java's text API incredibly flexible and the subject of the rest of this chapter is to give som examples of what you can do.

8.1 JFORMATTEDTEXTFIELD

As shown in the introductory example, a *JFormattedTextField* is a *JTextField*, where a formatter is attached to the component and determines how to format the content of the field. In the introductory example, I used a *MaskFormatter*, which can be used to ensure that you can only enter certain characters, but there are many other options. The type of a formatter is *AbstractFormatter*, and the specific formatter determines the format and when it happens. Regarding the latter, it may happen during the entry, for example, for a *MaskFormatter*, but other formatter objects the formatting first takes place after the field loses focus. With an option for the component, you can specify what should be done if the input does not conform to the format, but the default setting is that the component formats the content as well as possible and ignores what that do not conforms the format.

The following example uses the default setting. The window shows 5 *JFormattedTextField* components, and when you click OK, the formatted object appears in the field to the right.



A *JFormattedTextField* component can format any object, and the result is determined by the formatter. Here are two options. If a formatter is not specified, the component will select a format based on the type of the object. Otherwise, the formatter associated with the component in the constructor is used. The code is shown below, and you are requested to study the code, as well as test the program, and especially what format objects are used.

```
package formatterfields;

import java.util.*;
import java.text.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class MainView extends JFrame
{
 public MainView()
 {
  super("Formatted text fields");
  setSize(500, 380);
  setLayout(new FlowLayout());
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }
```

```
private void createView()
{
 JPanel panel = new JPanel();
 panel.setBorder(new EmptyBorder(20, 20, 20, 20));
 panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
 panel.add(createField("Date", new Date()));
 panel.add(createField("Integer", new Integer(1234)));
 panel.add(createField("Double", NumberFormat.getInstance(),
  new Double(Math.PI)));
 panel.add(createField("Currency", NumberFormat.getCurrencyInstance(),
  new Double(1234.56)));
 panel.add(createField("Percent", NumberFormat.getPercentInstance(),
  new Double(0.155)));
 add(panel);
}

private JPanel createField(String text, Object value)
{
 JFormattedTextField field = new JFormattedTextField(value);
 return createField(text, field);
}
```

```
  private JPanel createField(String text, Format format, Object value)
  {
   JFormattedTextField field = new JFormattedTextField(format);
   field.setValue(value);
   return createField(text, field);
  }

  private JPanel createField(String text, JFormattedTextField field)
  {
   field.setPreferredSize(new Dimension(120, 22));
   JTextField result = new JTextField();
   result.setEditable(false);
   result.setPreferredSize(new Dimension(220, 22));
   JButton cmd = new JButton("OK");
   cmd.setPreferredSize(new Dimension(60, 22));
   cmd.addActionListener(e -> result.setText(field.getValue().toString()));
   JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
   panel.setBorder(new TitledBorder(text));
   panel.add(field);
   panel.add(cmd);
   panel.add(result);
   return panel;
  }
}
```

You are also encouraged to investigate the help and which other formatter types there are.

EXERCISE 5

Create a copy of the solution *Calculations* for exercise 4 and call the copy *Calculations1*. Open it in NetBeans. Delete all that has to do with the clipboard so the program alone can calculate the total amount. You must now change the three input fields to *JFormattedTextField* components, where the number of units must use a *NumberFomat* to format an integer, while the others must also use a *NumberFormat* to convert the value to a currency.

When the program works with these changes, change the format of the *txtPrice* field (to enter the unit price) so that it instead uses a *DecimalFormat*.

You should study the *DecimalFormat* class thoroughly and investigate what opportunities there are – and there are many.

8.2 THE CARET

For each *JTextComponent*, a caret is attached, which visually shows where the next operation is to be performed (where a character must be inserted, where to delete a character and so on). Typically, it's a thin line that blinks, but it's an object of the type *DefaultCaret*, and basically it's the task of the class to draw that particular caret in the right place. The *JTextArea* class (and similar to the other text components) has an object of the type *DefaultCaret*, so you can write a derived class that draws a caret differently and then associate an object of this class with the component. It is not very difficult (even the opportunity has only limited interest). As an example, below is shown a custom caret that draws a caret as follows:

and the class can be written as:

```
class ACaret extends DefaultCaret
{
 public ACaret()
 {
  setBlinkRate(500);
 }

 protected synchronized void damage(Rectangle rect)
 {
  if (rect == null) return;
  x = rect.x;
  y = rect.y;
  width = 6;
  height = rect.height + 1;
  repaint();
 }

 public void paint(Graphics g)
 {
  JTextComponent component = getComponent();
  if (component == null) return;
  Rectangle rect = null;
  try
  {
   rect = component.modelToView(getDot());
  }
```

```
   catch (BadLocationException e)
   {
    return;
   }
   if (rect == null) return;
   if (isVisible())
   {
    g.setColor(Color.red);
    g.drawLine(rect.x, rect.y, rect.x, rect.y + rect.height);
    g.drawLine(rect.x, rect.y, rect.x + 5, rect.y);
    g.drawLine(rect.x, rect.y + rect.height, rect.x + 5, rect.y + rect.height);
   }
  }
 }
```

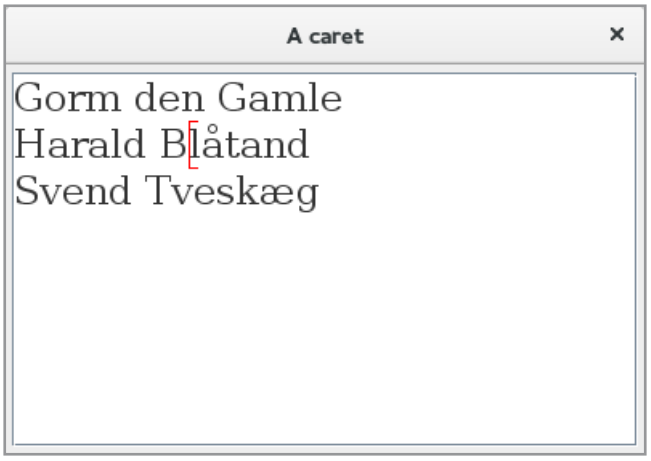
The constructor starts to define that the component's caret should blink every half second. Otherwise, most occurs in the *paint()* method, which is the method called by the runtime system whenever it is necessary for it to be redrawn (and it is often if you move it using the arrow keys, or if you enter a character). The method must draw the figure, which occurs in the last if statement, but before it is necessary to calculate where to draw. Here the class *DefaultCaret* has what is needed. The method *getDot()* returns the logical position in the text for the caret, and this position can be converted to coordinates relative to the component using method *modelToView()*. The result is a rectangle that encloses the components' caret. With this rectangle available, the figure can be drawn.

The class has another method called *damage()*, that is called by the rutime system when it is necessary to redraw. The class *DefaultCaret* has protected variables, which are used to specify the area that is required for redrawing, and these are initialized by the method *damage()*. After the variables are initialized, a *repaint()* will be performed, which ensures that *paint()* is performed at some point.

Then there is the following class that uses the above custom caret:

```
public class MainView extends JFrame
{
 public MainView()
 {
  super("Formatted text fields");
  setSize(500, 380);
  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));
  JTextArea txt = new JTextArea();
//   txt.setFont(new Font("Serif", Font.PLAIN, 24));
  txt.setCaret(new ACaret());
  txt.setText("Gorm den Gamle\nHarald Blåtand\nSvend Tveskæg");
  panel.add(new JScrollPane(txt));
  add(panel);
 }
}
```

There is not much to explain, but you should notice how to define that the *JTextArea* component must use another caret. The statement with the comment is to show that the new caret also respects another font. Below is an example of a run of the program where the comment has been removed and where the caret is moved:



8.3 HIGHLIGHTER

A text component also has a so-called highlighter, which is an object used to display text that is selected. By default, it displays the highlighted text with a different background color and is performed by a *Highlighter* object defined by the interface *Highlighter.HighlightPainter* and thus an internal interface in the *Highlighter* interface. It is possible to implement this interface yourself (although there are hardly any reasons for it), but the class below is an example where text is highlighted with a red underline:

```
class AHighlightPainter implements Highlighter.HighlightPainter
{
 private void paintLine(Graphics g, Rectangle rect, int x)
 {
  g.fillRect(rect.x, rect.y + rect.height - 3, x - rect.x, 3);
 }

 public void paint(Graphics g, int p0, int p1, Shape shape,
  JTextComponent component)
 {
  Rectangle rect0 = null;
  Rectangle rect1 = null;
  try
  {
   rect0 = component.modelToView(p0);
   rect1 = component.modelToView(p1);
  }
```

```
    catch (BadLocationException ex)
    {
     return;
    }
    if (rect0 == null || rect1 == null) return;
    Rectangle rect2 = shape.getBounds();
    int max = rect2.x + rect2.width;
    g.setColor(Color.red);
    if (rect0.y == rect1.y) paintLine(g, rect0, rect1.x);
    else
    {
     paintLine(g, rect0, max);
     rect0.y += rect0.height;
     rect0.x = rect2.x;
     while (rect0.y < rect1.y)
     {
      paintLine(g, rect0, max);
      rect0.y += rect0.height;
     }
     paintLine(g, rect0, rect1.x);
    }
   }
}
```

As it appears, it is only a matter of overriding a *paint()* method. The parameters are the graphic object to be drawn with, start and end position of the text to be highlighted, a rewriting rectangle for the text to be highlighted and then the text component. I do not want to go through the method's statements as it basically takes place in the same way as in the previous example with a caret, and the code is something nerd and belongs to what the next book describes, but in principle it's simple enough to figure out what happens. You should note that the method is complicated by the fact that the selected text can fill several lines. You should also note that the above highlighter can only be used by a component where all lines have the same height, that is, a *JTextArea* or a *JTextField*. The test program is identical to the above, where the difference is that you associate another caret:

```
private void createView()
{
 JPanel panel = new JPanel(new BorderLayout());
 panel.setBorder(new EmptyBorder(5, 5, 5, 5));
 JTextArea txt = new JTextArea();
 txt.setLineWrap(true);
 txt.setWrapStyleWord(true);
 txt.setCaret(new DefaultCaret() {
  private Highlighter.HighlightPainter hl = new AHighlightPainter();
  protected Highlighter.HighlightPainter getSelectionPainter()
   { return hl; } });
 txt.setText("Gorm den Gamle\nHarald Blåtand\nSvend Tveskæg");
 panel.add(new JScrollPane(txt));
 add(panel);
}
```

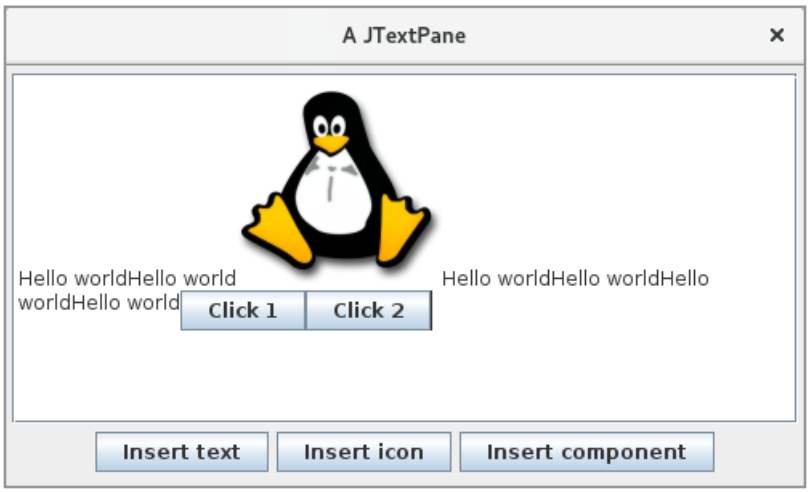
8.4 A JTEXTPANE

The subject of this section is the component *JTextPane*, which is an extremely advanced component with many options for settings and many related types in the form of interfaces and classes, and the following will also only be an introduction but sufficient to get an impression of what a *JTextPane* can do and how you can use the component in practice.

Basically, you can think of a *JTextPane* as a *JTextArea* and thus a component where you can enter and edit text lines, but with the big difference that the component supports styling of the text. This means that individual text elements can have assigned style objects in the form of, for example, font and color. The component can actually more than it, as it can also display images and even other Swing components. Like the other Swing components, a *JTextPane* is designed by MVC, and the model is defined (similar to the other text components) of the *Document* interface, and a *JTextPane* uses the concrete class *StyledDocument* as model. It is thus the model that keeps track of the individual style elements. The model is dealt with first in the next section, and this section will primarily show how to use a *JTextPane* in a program.

I want to start with an application that opens a window with three buttons. Over the three buttons is a *JTextPane* in a *JScrollPane*. If you click on the *Insert Test* button, the text *Hello World* is inserted at the cursor's position. Clicking the button *Insert icon* instead adds an icon at the cursor's position, and clicking the *Insert component* button adds a button. Below is the window after I have

1. clicked the button *Insert text* twice
2. clicked the button *Insert icon*
3. clicked the button *Insert text* four times
4. clicked the button *Insert component* twice



If you instead select some of the content and click on one of the buttons, then the selected will be replaced by what are inserted. You should note that the buttons are real buttons and clicking on one of them will give you a message box. In fact, you can insert any Swing component and hence a *JPanel*, and then you can insert a form. The example here has nothing to do with styling, but it still shows what data the component is capable of rendering.

The code is shown below and does not fill as much as you might expect:

```
package atextpane;

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

public class MainView extends JFrame
{
 private static ImageIcon icon = createImage();
 private int counter = 0;
 private JTextPane txtPane = new JTextPane();

 public MainView()
 {
  super("A JTextPane");
  setSize(500, 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));
  panel.add(new JScrollPane(txtPane));
  panel.add(createBottom(), BorderLayout.SOUTH);
  add(panel);
 }

 private JPanel createBottom()
 {
  JPanel panel = new JPanel(new FlowLayout());
  panel.add(createButton("Insert text", e ->
   { txtPane.replaceSelection("Hello world"); txtPane.requestFocus(); }));
  panel.add(createButton("Insert icon", e ->
   { txtPane.insertIcon(icon); txtPane.requestFocus(); }));
  panel.add(createButton("Insert component", e ->
   { txtPane.insertComponent(createButton("Click " + (++counter),
     a -> JOptionPane.showMessageDialog(this,
     "Button " + counter))); txtPane.requestFocus(); }));
  return panel;
 }

 private JButton createButton(String text, ActionListener listener)
 {
  JButton cmd = new JButton(text);
  cmd.addActionListener(listener);
  return cmd;
 }

 private static ImageIcon createImage()
 {
  java.net.URL imgURL =
   MainView.class.getResource("/atextpane/images/linux.png");
  return new ImageIcon(new ImageIcon(imgURL, "").
   getImage().getScaledInstance(128, 128, Image.SCALE_SMOOTH), "");
 }
}
```

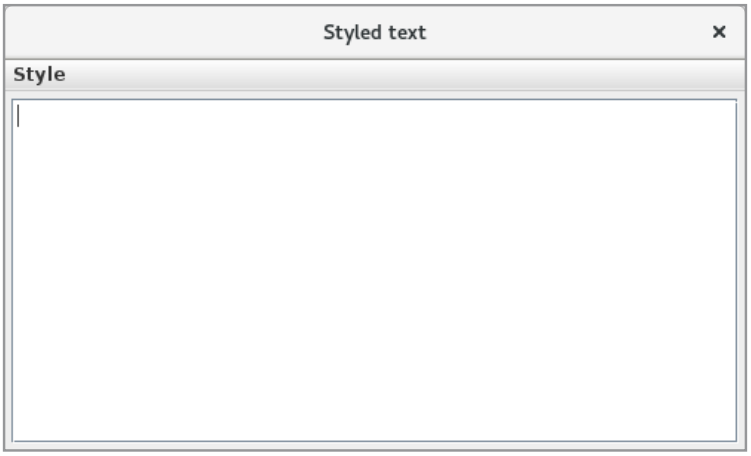
There are defined three instance variables, where the first is the icon to be inserted, and the last method is the same method that I have used many times that loads the image from the program's jar file and scales it. The variable *counter* is used for the text of the buttons that may be inserted so that they can be known from each other. Finally, there is a reference to the *JTextPane* component so it can be referenced from the three button's event handlers. The component is inserted into the window as other components, and there is something to note in the three event handlers. There you should notice the following methods on a *JTextPane*:

- *replaceSelection()*
- *insertIcon()*
- *insertComponent()*

which is used to insert a text, insert an icon, and insert a component, respectively. Regardless of the names, they all work in the same way and replace the content marked with what to insert – if something is selected and otherwise inserted at the caret's position.

Then there is styling in the form of attributes and styles. You can attach attributes to the individual characters, to paragraphs, and to the entire document. Character attributes relate to font and text color. Attributes regarding paragraphs may also concern indentation and line spacing. A style is a group of attributes that can be used in multiple places in the document, and may be a named style known by a name. If you change an attribute by a style (attached to the document), the attribute of all text in the document that uses that style changes. If a style is assigned to a paragraph, its attributes can be overridden by attaching attributes to a portion of the paragraph's text. An attribute is a key/value pair, and families of attributes are defined by three interfaces: *AttributeSet*, *MutableAttributeSet* and *Style*.

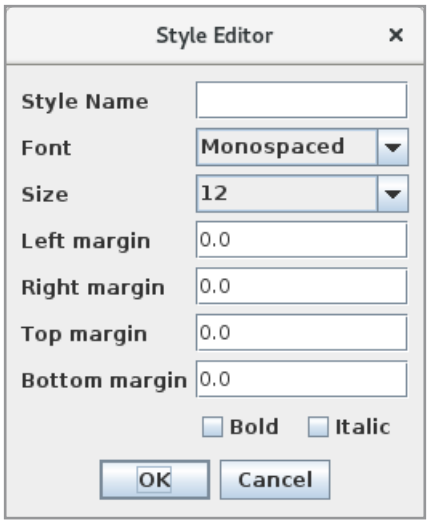
I will now show a program with a *JTextPane*, which uses styles. The program opens a window with a *JTextPane* and a menu:



The menu has three menu items

- Set style
- Modify style
- Create style

where the first two are two empty submenus, while the bottom one is a function used to create a style for a paragraph. If you choose this menu item you will get a window as shown below, where you can create a named style that defines a font. Here you should especially note that you can specify how much air there should be outside the section.



Once you have created a style, a menu item is added to each of the two submenus, and then you can apply that style to the paragraph that the cursor is in and you can edit that style that opens the same dialog as shown above. Below is a run of the program where 7 lines are entered:

Note that each line is a paragraph. There are then created 4 styles which are subsequently applied to the 7 lines. The result is as follows:

Then there is the program code that fills some and I want to start with the dialog to maintenance styles:

```
class StylePanel extends JPanel
{
 private static final String[] fonts =
  { "Monospaced", "Serif", "Sans", "SansSerif" };
 private static final String[] sizes =
  { "8", "10", "12", "14", "18", "24", "36", "48", "72" };
 private JTextField txtName = new JTextField();
 private JComboBox lstFont = new JComboBox(fonts);
 private JComboBox lstSize = new JComboBox(sizes);
 private JTextField txtLeft = new JTextField();
 private JTextField txtRight = new JTextField();
 private JTextField txtTop = new JTextField();
 private JTextField txtBottom = new JTextField();
 private JCheckBox cmdBold = new JCheckBox("Bold");
 private JCheckBox cmdItalic = new JCheckBox("Italic");

 public StylePanel()
 {
  super(new BorderLayout(5, 5));
  JPanel left = new JPanel(new GridLayout(8, 1, 0, 5));
  JPanel right = new JPanel(new GridLayout(8, 1, 0, 5));
  add(left, BorderLayout.WEST);
  add(right);
  left.add(new JLabel("Style Name", JLabel.LEFT));
  right.add(txtName);
  left.add(new JLabel("Font", JLabel.LEFT));
  right.add(lstFont);
  left.add(new JLabel("Size", JLabel.LEFT));
  right.add(lstSize);
```

```
 left.add(new JLabel("Left margin", JLabel.LEFT));
 right.add(txtLeft);
 left.add(new JLabel("Right margin", JLabel.LEFT));
 right.add(txtRight);
 left.add(new JLabel("Top margin", JLabel.LEFT));
 right.add(txtTop);
 left.add(new JLabel("Bottom margin", JLabel.LEFT));
 right.add(txtBottom);
 left.add(new JLabel());
 JPanel panel = new JPanel(new GridLayout(1, 2));
 panel.add(cmdBold);
 panel.add(cmdItalic);
 right.add(panel);
 clear();
}

public void clear()
{
 txtName.setText("");
 txtName.setEditable(true);
 lstFont.setSelectedIndex(0);
 lstSize.setSelectedIndex(2);
 txtLeft.setText("0.0");
```

```
  txtRight.setText("0.0");
  txtTop.setText("0.0");
  txtBottom.setText("0.0");
  cmdBold.setSelected(false);
  cmdItalic.setSelected(false);
 }

 public String getStyleName()
 {
  String name = txtName.getText().trim();
  if (name.length( ) > 0) return name;
  return null;
 }

 public void initStyle(Style style)
 {
  StyleConstants.setFontFamily(style, (String)lstFont.getSelectedItem());
  StyleConstants.setFontSize(style,
   Integer.parseInt((String)lstSize.getSelectedItem()));
  StyleConstants.setLeftIndent(style,
   Float.valueOf(txtLeft.getText()).floatValue());
  StyleConstants.setRightIndent(style,
   Float.valueOf(txtRight.getText()).floatValue());
  StyleConstants.setSpaceAbove(style,
   Float.valueOf(txtTop.getText()).floatValue( ));
  StyleConstants.setSpaceBelow(style,
   Float.valueOf(txtBottom.getText()).floatValue());
  StyleConstants.setBold(style, cmdBold.isSelected());
  StyleConstants.setItalic(style, cmdItalic.isSelected());
 }

 public void initFields(Style style)
 {
  txtName.setText(style.getName());
  txtName.setEditable(false);
  lstFont.setSelectedItem(StyleConstants.getFontFamily(style));
  lstSize.setSelectedItem(Integer.toString(StyleConstants.getFontSize(style)));
  txtLeft.setText(Float.toString(StyleConstants.getLeftIndent(style)));
  txtRight.setText(Float.toString(StyleConstants.getRightIndent(style)));
  txtTop.setText(Float.toString(StyleConstants.getSpaceAbove(style)));
  txtBottom.setText(Float.toString(StyleConstants.getSpaceBelow(style)));
  cmdBold.setSelected(StyleConstants.isBold(style));
  cmdItalic.setSelected(StyleConstants.isItalic(style));
 }
}
```

There is not much to explain about the design, but you should note that the class inherits *JPanel* and not *JDialog*. The explanation follows below. Otherwise, the two last methods are the most interesting. The first initializes a *Style* object using the *StyleConstants* class. The class has a number of constants for typical values for attributes and a number of static methods that are used to initialize attributes in a style object. Similarly, the last method is used to initialize the dialog box's fields with values from a *Style* object, and note again the use of the class *StyleConstants* that has static methods that returns the values.

Then there is the code for the window, where I have only shown that part of the code that relates to the *JTextPane* component and styles:

```
public class MainView extends JFrame
{
 private JTextPane txtPane = new JTextPane();
 private StylePanel stylePanel = new StylePanel();
 private JMenu setMenu;
 private JMenu modMenu;

 public MainView()
 {
  …
 }

 private void createView()
 {
  createMenuBar();
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(5, 5, 5, 5));
  panel.add(new JScrollPane(txtPane));
  add(panel);
 }

 private void createMenuBar()
 {
  …
 }

 private void createMenuItem(JMenu menu, String text, ActionListener listener)
 {
  JMenuItem item = new JMenuItem(text);
  item.addActionListener(listener);
  menu.add(item);
 }
```

```
public void create(ActionEvent e)
{
 stylePanel.clear();
 if (JOptionPane.showConfirmDialog(this, stylePanel, "Style Editor",
  JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE) ==
  JOptionPane.OK_OPTION && stylePanel.getStyleName().length() > 0)
 {
  Style style = txtPane.addStyle(stylePanel.getStyleName(), null);
  stylePanel.initStyle(style);
  createMenuItem(setMenu, stylePanel.getStyleName(), this::setStyle);
  createMenuItem(modMenu, stylePanel.getStyleName(), this::modStyle);
 }
}

public void setStyle(ActionEvent e)
{
 String styleName = ((JMenuItem)e.getSource()).getActionCommand();
 txtPane.setLogicalStyle(txtPane.getStyle(styleName));
}
```

```
 public void modStyle(ActionEvent e)
 {
  String styleName = ((JMenuItem)e.getSource()).getActionCommand();
  Style style = txtPane.getStyle(styleName);
  stylePanel.initFields(style);
  if (JOptionPane.showConfirmDialog(this, stylePanel, "Style Editor",
   JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE) ==
   JOptionPane.OK_OPTION) stylePanel.initStyle(style);
 }
}
```

Note that there is an instance variable whose type is the above dialog box (a *JPanel*). If you look at the event handler regarding creating a style, it starts by performing the dialog's *clear()* method, thus deleting all fields. The Swing class *JOptionPane* has several methods that open a message box, and I have earlier used *showConfirmDialog()* that has two buttons as an *OK* button and a *Cancel* button. The method has at least two parameters, where the second has the type *Object* and general it is the text that appears in the message box. A function that I have not mentioned previously is that if this parameter may be a Swing component, and in this case it is a *JPanel*, and the result is therefore a dialog box with an *OK* button and an *Cancel* button. This approach is actually an easy option if there is a need to open a simple dialog box. You should note how the event handlers adds two menu items if *OK* is clicked.

The event handler to modify a style works in principle the same way (opens the same dialog box) but does not add menu items to the menu. You should also note the event handler to apply a style for a paragraph and here how to assign a style to a paragraph.

EXERCISE 6

You must create a copy of the above project (*StyledText*). You must add a menu named *Select color*, which must have a menu item for the colors:

- red, green, blue, yellow, orange, gray lightGray, magenta, pink and black

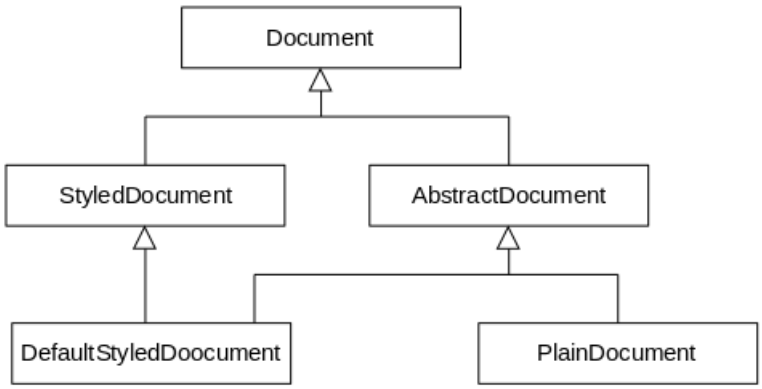
If you select one of these menu items, the text in the paragraph where the caret is displayed must be shown with the selected color.

When it works, change the program so that you can mark a text, and if you then in the menu selects a font or color then the highlighted text must be displayed with the chosen style. An example could be as shown below, where there are 7 paragraphs:

You need to examine the help for *JTextPane* to find out how to determine the area selected and in the first program in this chapter, you can see how to assign attributes to individual characters instead of a paragraph.

8.4 DOCUMENT

As mentioned, all text components use a model, as defined above by the *Document* Interface. The model represents the text that the user edits, and if it is a styled document, it is also the model that represents the attributes. The *Document* interface does not define anything about styles, but instead, it takes place in a sub-interface *StyledDocument*. The concrete models can thus be illustrated by the following figure, where *AbstractDocument* as evidenced by the name is an abstract class, while *PlainDocument* and *DefaultStyledDocument* are concrete classes. *PlainDocument* is the model for a *JTextField* and a *JTextArea* component, while *DefaultStyledDocument* is the model of a *JTextPane* and a *JEditorPane*.



A *Document* object divides the document into a hierarchy of objects where each node has the type *Element* and consists of an offset and an end position in the document. In addition, an *Element* may have associated an *AttributeSet* and thus a style is not used by, for example, by a *JTextField* and a *JTextArea*. An *Element* object thus describes a part of the document and seen from a *JTextPane* it is interesting that an *Element* may have its own style.

An important task for *AbstractDocument* is to implement a locking mechanism that ensures that at one point only a single writer can use the document, but no or more can read the document.

As a simple example of how to attach your own document to a text component, the program *EnterProgram* opens the following window:

where the user can enter a text, but only 20 characters can be entered. The input field is a *JTextField* component and the challenge is solved by associating the component with another model. The model is defined as a derived class of the class *PlainDocument* that only overrides the method *insertString()*:

```
class MaxDocument extends PlainDocument
{
 private int max;

 public MaxDocument(int max)
 {
  this.max = max;
 }

 public void insertString(int offset, String str, AttributeSet attr)
  throws BadLocationException
 {
  if (getLength() + str.length( ) > max) Toolkit.getDefaultToolkit().beep();
  else super.insertString(offset, str, attr);
 }
}
```

With this class available, the program can be written as follows:

```
public class MainView extends JFrame
{
 private JTextField txtField = new JTextField();

 public MainView()
 {
  super("Enter text");
  setSize(600, 150);
  createView();
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setVisible(true);
 }

 private void createView()
 {
  txtField.setDocument(new MaxDocument(20));
  txtField.setPreferredSize(new Dimension(200, 20));
  JButton cmd = new JButton("OK");
  cmd.addActionListener(
   e -> JOptionPane.showMessageDialog(this, txtField.getText()));
```

```
    JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    panel.setBorder(new EmptyBorder(25, 25, 25, 25));
    panel.add(new JLabel("Enter text (max 20 characters)"));
    panel.add(txtField);
    panel.add(cmd);
    add(panel);
  }
}
```

It is a simple program, but the options for blocking the number of characters that can be entered are interestingly, as in practice it is a frequently encountered issue. However, keep in mind that the same problem could be solved simpler with a *JFormattedTextField* and appropriate format such as:

```
JFormattedTextField(new MaskFormatter("********************"))
```

EXERCISE 7

Write a program that you can call *ElementTree* when the program opens the following window:

where the upper component is a *JTextPane*, while at the bottom is a *JTextArea*. The upper component is initialized with the following string:

```
"Gorm den Gamle\nHarald Blåtand\nSvend Tveskæg\nHarald d. 2.\nKnud den Store"
```

Before the string is displayed, each of the five names must be assigned attributes that indicate the color and size of the font respectively.

If you click on the button, you must get a list in the lower component where each line represents an *Element* and for each are displayed offset for both the beginning and end of the characters spanning the element and any attributes. Note that as the elements constitute a hierarchy, the method that fills text in the lower component must be written recursively.

Try to interpret the results in the bottom *JTextArea*.

You can try entering more text in the upper *JTextPane* and then clicking *OK* to see if you get the expected result.

8.5 A JEDITORPANE

As the last of Swing's text components, a *JEditorPane* should be mentioned. It's a component that rarely has any interest in itself, but it can be perceived as the base class of styled text components, and a *JTextPane* is an example of a text component derived from *JEditorPane*. As can be seen from the above, a *JTextPane* is a component that can be used to edit a document to which style information is assigned. The actual text and style information is represented by a *StyledDocument*, but some logic is required to control how text and styles can be edited (what actions can be performed) and it is all encapsulated in an *EditorKit* object. The simple text components such as *JTextField* and *JTextArea* uses a *DefaultEditorKit* while a *JTextPane* uses is derived class called *StyledEditorKit*. The goal of a *JEditorPane* is that you can define its own text component derived from *JEditorPane* by defining its own *EditorKit* and its *Document* type, thus defining text components that exactly support the tasks you need. In fact, Swing comes with two examples that can be used to edit *html* and *rtf* documents.

In this section I will very overall illustrate how you can write your own html editor. The goal is not to reach a finished html editor (which is comprehensive), but to show a little more about text components and a little about what an *EditorKit* is. I want to start with an example called *AEditorPane*. If you run the program, you first get the following message box:

and when you click *OK*, you get the following window:

The window shows the following html document that are saved with the project:

```
<html>
 <head>
  <meta charset="UTF-8">
  <title>Simple html page</title>
 </head>
 <body>
  <h1>Danish Kings</h1>
  <h2>The wee know about</h2>
  <p>Gorm den Gamle</p>
  <p>Harald Blåtand</p>
  <p>Svend Tveskæg</p>
  <h3>But we have heard of others before them</h3>
  <p><a href="http://danmarkshistorien.dk/forside/">The danish history</a></p>
 </body>
</html>
```

That is, the document is interpreted as the component knows the individual html elements and thus knows how to interpret them. In particular, note the last line, which is a link to the page

```
http://danmarkshistorien.dk/forside
```

and clicking this link will actually show that page in the window. The result is not too good, but it is because the page contains many things (modern websites are very complex in terms of content) that the component can not interpret, but the example illustrates that you can load any webpage and with more or less good results display the page's content in a text component. The example is thus a step towards a simple browser. The program's code is as follows:

```
package aeditorpane;

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

public class MainView extends JFrame
{
 private JEditorPane txtPane = null;
```

```
public MainView()
{
 super("A JEditorPane");
 setSize(500, 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));
 try
 {
  txtPane = new JEditorPane(loadPage());
  txtPane.addHyperlinkListener(new LinkHandler());
  JOptionPane.showMessageDialog(this,
    txtPane.getEditorKit().getClass().getName());
 }
 catch (Exception ex)
 {
```

```
    JOptionPane.showMessageDialog(this, ex.toString());
    System.exit(1);
   }
   txtPane.setEditable(false);
   panel.add(new JScrollPane(txtPane));
   add(panel);
  }

  private static URL loadPage()
  {
   return MainView.class.getResource("/aeditorpane/kings.html");
  }

  class LinkHandler implements HyperlinkListener
  {
   public void hyperlinkUpdate(HyperlinkEvent e)
   {
    try
    {
     if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      txtPane.setPage(e.getURL());
    }
    catch (Exception ex)
    {
     JOptionPane.showMessageDialog(MainView.this, ex.toString());
    }
   }
  }
 }
```

The program defines a single instance variable, whose type is *JEditorPane*. It is created in the method *createView()*, where the above html document is transferred to the class' constructor as an URL. The html document is packaged with the program in the jar file, and it is loaded from the jar file in the method *loadPage()*. After the *JEditorPane* component is created there is associated an event handler for click on a link. It is a class that implements the interface *HyperlinkListener*, which defines a single method called *hyperlinkUpdate()*, and the method is executed when a link is clicked in the window. The result is that the content of the component is overwritten with the content of the website that is linked to.

After this event handler is associated with the component, the first message box opens. Viewed from the program, it makes no sense, but it shows which *EditorKit* is being used. In this case, it is an *HTMLEditorKit*. It was created by the constructor in *JEditorPane*, since the argument is an HTML document. *HTMLEditorKit* is a finished *EditorKit* that is part of Swing, but it meets far from all that one needs to demand from a modern browser and when the result of clicking the link in the window not is as desired, it is because the *HTMLEditorKit* class is not sufficiently comprehensive, but you can write your own – if you have the need.

An *EditorKit* is an abstract class, but *DefaultEditorKit* is a concrete class, as used by the simple text components, and by a derived class *StyledEditorKit*, used by a *JTextPane*, and the *HTMLEditorKit* class is again derived from *StyledEditorKit*. An *EditorKit* defines how to create the document, and also defines methods that can be read the document from a stream and write it to a stream.

ACTIONS

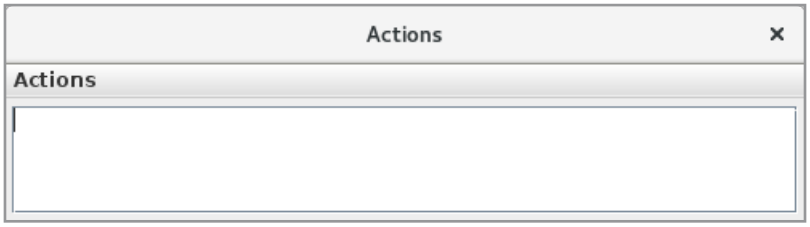
There are many events associated with text components and especially keyboard and mouse events, and you can also make many settings regarding fonts and colors, and more. All these options are represented as actions. An *Action* is an interface that inherits two other interfaces: *ActionListener* and *EventListener*, and one should think of an *Action* as an object representing an *ActionEvent*. The goal is that the same functionality should be made available to several components. An *Action* may have associated several key / value pairs that you have access to with the methods:

Object getValue(String key)
void putValue(String key, Object value)

Particular attention should be paid to the last method as it changes the value if the key already exists.

The interface is implemented by the class *AbstractAction*, but the class obviously does not implement *actionPerformed()*, and if you write your own *Action* types, it is equivalent to writing a class that inherits *AbstractAction* and then implementing *actionPerformed()*. The advantage of actions is that classes like *JButton*, *JMenuItem*, etc. have constructors that have an *Action* as parameter, and it makes it easy to assign functionality to those kinds of objects, and finally is the reason that *Actions* are mentioned here, while the text components uses a lot of *Actions*.

If you run the program *ActionProgram*, it opens a window with a *JTextArea* component as well as a menu:



If you open the menu, there are two submenus and each of these submenus shows an overview of the actions that a *JTextArea* knows. When you see the names, it's easy enough to figure out what the individual actions means. The program is simple, and only the method that creates the menu is shown below:

```
private void createMenu()
{
 Action[] actions = txtArea.getActions();
 int n = actions.length / 2;
 JMenu menu1 = new JMenu("Some actions");
 for (int i = 0; i < n; ++i) menu1.add(actions[i]);
```

```
  JMenu menu2 = new JMenu("More actions");
  for (int i = n; i < actions.length; ++i) menu2.add(actions[i]);
  JMenu menu = new JMenu("Actions");
  menu.add(menu1);
  menu.add(menu2);
  JMenuBar bar = new JMenuBar();
  bar.add(menu);
  setJMenuBar(bar);
 }
```

EXERCISE 8

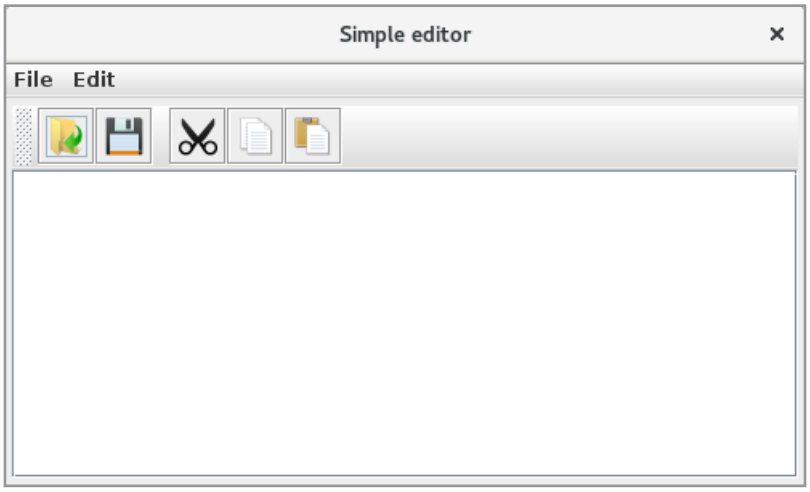
Create a copy of the above program. You need to change the program so that the component instead of a *JTextArea* is a *JTextPane*, and so the menu shows all actions distributed on three submenus.

Try to investigate the differences between the actions that this program shows and the above program.

A SIMPLEEDITOR

I want to show a program that is a simple text editor and the program looks like the corresponding program that I have shown in the book Java 2, but the program is simpler and partly because there are some features that are not implemented in this solution and partly because I here, to a greater extent, utilizes that many of the necessary features are built into the class *JTextArea*, and then I do not have to implement them.

The program opens a window with a *JTextArea*:



The window also has a toolbar with features to open and save a document, and there are icons for cut, copy and paste. The same functions are found as menu items. The completed code is as follows:

```
public class MainView extends JFrame
{
 private JTextComponent txtComp = new JTextArea();
 private Action openAction = new OpenAction();
 private Action saveAction = new SaveAction();

 public MainView()
 {
  …
 }

 private void createView()
 {
  ((JTextArea)txtComp).setLineWrap(true);
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(5, 5, 5, 5));
  panel.add(new JScrollPane(txtComp));
  defineActions();
  panel.add(createToolbar(), BorderLayout.NORTH);
  createMenu();
  add(panel);
 }

 protected void defineActions()
 {
  defineAction(txtComp.getActionMap().get(DefaultEditorKit.cutAction), "Cut",
   "/simpleeditor/images/cut.png");
  defineAction(txtComp.getActionMap().get(DefaultEditorKit.copyAction), "Copy",
   "/simpleeditor/images/copy.png");
  defineAction(txtComp.getActionMap().get(DefaultEditorKit.pasteAction), "Paste",
   "/simpleeditor/images/paste.png");
 }

 private void defineAction(Action action, String name, String icon)
 {
  action.putValue(Action.NAME, name);
  if (icon != null) action.putValue(Action.SMALL_ICON, createImage(icon));
 }
```

```
private JToolBar createToolbar()
{
 JToolBar toolBar = new JToolBar();
 toolBar.add(openAction).setText("");
 toolBar.add(saveAction).setText("");
 toolBar.addSeparator();
 toolBar.add(
  txtComp.getActionMap().get(DefaultEditorKit.cutAction)).setText("");
 toolBar.add(
  txtComp.getActionMap().get(DefaultEditorKit.copyAction)).setText("");
 toolBar.add(
  txtComp.getActionMap().get(DefaultEditorKit.pasteAction)).setText("");
 return toolBar;
}

private void createMenu()
{
 JMenu file = new JMenu("File");
 file.add(openAction);
 file.add(saveAction);
 file.add(new ExitAction());
 JMenu edit = new JMenu("Edit");
 edit.add(txtComp.getActionMap().get(DefaultEditorKit.cutAction));
```

```
 edit.add(txtComp.getActionMap().get(DefaultEditorKit.copyAction));
 edit.add(txtComp.getActionMap().get(DefaultEditorKit.pasteAction));
 JMenuBar bar = new JMenuBar();
 bar.add(file);
 bar.add(edit);
 setJMenuBar(bar);
}

class ExitAction extends AbstractAction
{
 public ExitAction()
 {
  super("Exit");
 }

 public void actionPerformed(ActionEvent e)
 {
  System.exit(0);
 }
}

class OpenAction extends AbstractAction
{
 public OpenAction()
 {
  super("Open", createImage("/simpleeditor/images/open.png"));
 }

 public void actionPerformed(ActionEvent e)
 {
  JFileChooser chooser = new JFileChooser();
  if (chooser.showOpenDialog(MainView.this) != JFileChooser.APPROVE_OPTION)
   return;
  File file = chooser.getSelectedFile();
  if (file == null) return;
  try (BufferedReader reader = new BufferedReader(new FileReader(file)))
  {
   txtComp.read(reader, null);
  }
  catch (IOException ex)
  {
   JOptionPane.showMessageDialog(MainView.this, "File Not found: " +
    ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
  }
 }
}
```

```
 class SaveAction extends AbstractAction
 {
  public SaveAction()
  {
   super("Save", createImage("/simpleeditor/images/save.png"));
  }

  public void actionPerformed(ActionEvent e)
  {
   JFileChooser chooser = new JFileChooser( );
   if (chooser.showSaveDialog(MainView.this) != JFileChooser.APPROVE_OPTION)
    return;
   File file = chooser.getSelectedFile();
   if (file == null) return;
   try (BufferedWriter writer = new BufferedWriter(new FileWriter(file)))
   {
    txtComp.write(writer);
   }
   catch (IOException ex)
   {
    JOptionPane.showMessageDialog(MainView.this, "File not Saved: " +
     ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
   }
  }
 }

 private static ImageIcon createImage(String path)
 {
  java.net.URL imgURL = MainView.class.getResource(path);
  return new ImageIcon(new ImageIcon(imgURL, "").getImage().
   getScaledInstance(24, 24, Image.SCALE_SMOOTH), "");
 }
}
```

The program defines three instance variables, the first being the *JTextArea* component, while the other are *Action* objects that are used to open and save a file, respectively. When the two objects are created as instance variables, it is because it is the same *Action* objects to be used both from the menu and from the toolbar. The objects type is defined by an inner class in the program.

The method *defineActions()* is used to customize three actions. A *JTextArea* has actions for copy, paste or cut, but typically you want to give them another name and possibly attach an icon to them. That's what happens in the method *defineActions()*. This happens by setting the key / value pair, which is performed by the method *defineAction()*. For example, you reference to the action for cut as:

```
txtComp.getActionMap().get(DefaultEditorKit.cutAction)
```

Here, *DefaultEditorKit.cutAction* is a constant that identifies this action, and the *Action* object is then sent to the method *defineAction()* along with the text to be displayed in the menu as well as the name of the icon to be used. The method *defineActions()* is called from *createView()* so the three actions are initialized as desired.

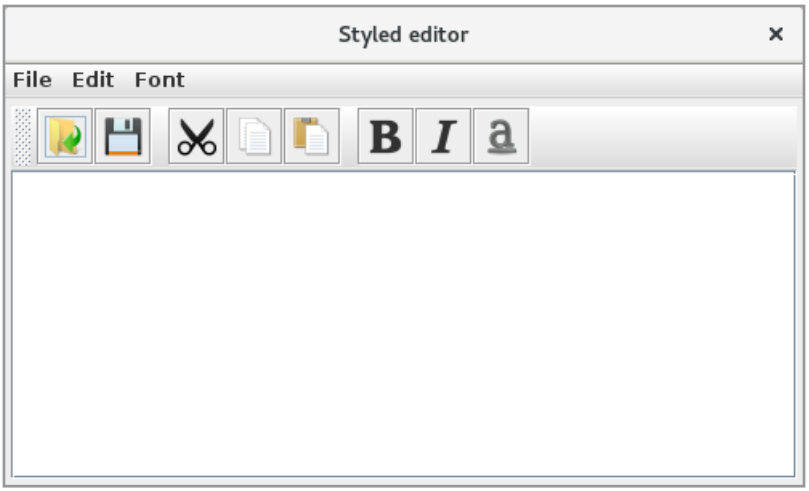
Then there is the method *createToolbar()* that creates a toolbar with 5 icons or actions. You should note that the first two are the two actions defined as instance variables at the start of the program, while the other three are referenced in the same way as shown above. Also note that for all 5 *Action* objects, the *setText()* method is performed, which puts the text to blank. The reason is that the toolbar should not display the object's text property.

The method *createMenu()* is in principle identical. Instead, it is *JMenuItem* objects that are associated with an *Action*. Also note that the *File* menu has an *ExitAction* action, which is also an inner class that represents an action.

Back there are the three inner classes that defines the *Action* objects. I will not go through the three classes in detail, but you will notice how they all inherit *AbstractAction*. Also note how the two last associate an icon and then note that a *JTextComponent* has methods that can directly initialize the component by reading from a stream and write the content of the component to a stream.

PROBLEM 5

You must write a program that you can call *StyledEditor*, which will open the following window:



The program should in principle work as the previous program, where the first two menus are the same (and correspondingly the first 5 icons in the toolbar), but instead of using a *JTextArea*, the program must use a *JTextPane*. The *Font* menu should be as follows:

```
Font
 Style
  Bold
  Underline
  Italic
 Family
  SansSerif
  Serif
  Monospaced
```

```
Size
 8
 10
 12
 14
 18
 24
 36
 48
 144
```

In principle, the program can be written as in the previous example, but it is more extensive to create the toolbar and the menu. However, there is a small problem as a *JTextPane* uses the *read()* and *write()* methods, which is defined in *JTextComponent* and these methods can only read and write common plan text. Although it is not a suitable solution, you can solve the problem by serialization.

9 INTERNATIONALIZING

Sometimes programs must be used in several countries and thus in multiple languages, and it demands the application's user interface and how data are displayed. One option would be to write several versions of the same program (an English version, a Danish version and so on), but for maintenance reasons it is not an optimal solution. However, Java supports internationalization, so the program itself determines how data should be displayed depending on where it is used. It is the subject of this chapter, and it is by no means simple, and the following is just an introduction to what it's all about.

Basically, it relates to the language used by the program to display text, as well as the formats for dates, times and numbers. In reality, there are two things, and internationalization means that the program supports multiple languages and formats, while by localization is understand as the process that the program determines which language and formats to use. It is all complicated by the fact that more countries use the same language, and some countries even use more languages, and in the same way there is no clear distinction between language and formatting rules. Seen from Java, the starting point is the class *Locale*, which basically consists of three properties:

1. a language code (ISO-639)
2. a country code (ISO-3166)
3. a variant code that is not required

The language code is two or three characters and as examples can be mentioned

English	en	eng
French	fr	fra
Germande	deu	
Danish	da	dan

Similarly, the country codes are two or three characters, and as examples can be mentioned

English	gb	gbr
French	fr	fra
German	de	deu
Denmark	dk	dnk

The class *Locale* uses 2-character codes (however, 3-character language codes are allowed if there is no 2-character code). The class generally does not offer much, but many methods have a *Locale* parameter, which is the primary purpose of the class. The following program shows examples of using the *Locale* class:

```
package localeprogram;

import java.util.*;

public class LocaleProgram
{
 public static void main(String[] args)
 {
  test1();
  test2();
  test3();
 }

 private static void test1()
 {
  Locale loc = Locale.getDefault();
  print(loc);
 }

 private static void test2()
 {
  Locale.setDefault(Locale.US);
  test1();
 }

 private static void test3()
 {
  Locale[] locales = Locale.getAvailableLocales();
  System.out.println(locales.length);
  for (Locale loc : locales) System.out.println(String.format("%s: %s, %s: %s",
   loc.getDisplayCountry(), loc.getCountry(),
   loc.getDisplayLanguage(), loc.getLanguage()));
 }

 private static void print(Locale loc)
 {
  System.out.println(String.format("%s, %s",
   loc.getDisplayCountry(), loc.getCountry()));
  System.out.println(String.format("%s, %s",
   loc.getDisplayLanguage(), loc.getLanguage()));
```

```
    System.out.println(String.format(loc, "%1.2f", 1234.56));
    System.out.println(String.format(loc, "%tD", Calendar.getInstance()));
  }
}
```

The method *test1()* determines the program's default *Locale* as determined by the machine's language settings. Then the program prints information about the language and the country. In addition, a number is formatted with *String.format()* and here you should note that the first parameter is a local object. This ensures that the result is displayed with the correct decimal point. The last statement in the method *print()* works the same way, but this time for a *Calendar* object. You should note that the String class's *format()* method has incredible options for formating the result – especially regarding dates and time – and you are encouraged to investigate the help for the options available.

The method *test2()* shows that you have the option to define which default locale to use. You can create a *Local* object yourself by specifying the language and country, but it is rarely necessary since the *Locale* class has defined objects for most languages. The method *test3()* shows an overview of which *Locale* objects are defined.

9.1 RESOURCE BUNDLES

If a program is to be internationalized, it means that the text should be available in several languages, and although Java supports internationalization, it is of course the task of the development department to translate the text. The translated text must be saved somewhere, and it happens in resource bundles, which allows to save text, images and other resources outside of the program and not as part of the code. Resource bundles are represented by the class *ResourceBundle*, which encapsulates key/value pairs, where the key is a string, while value can be any arbitrary object.

ResourceBundle is an abstract class, and although you can define your own specific class, you usually use

- *PropertyResourceBundle* to text
- *ListResourceBundle* to images and other resources

I will as an example show how to use a resource bundle of the first kind, as it is the most commonly used in internationalization, primarily to achieve a program that is independent of the language.

The starting point is a *poperties* file, which is simply a text file where the filename has the extension *.properties*. An example might be the following:

```
di=Ruder
he=Hjerte
sp=Spar
cl=Kl\u00f8r
```

which contains four texts with keys. The content can be interpreted as the colors for playing cards. Note that the 'ø' in the last value is encoded with the code for the unicode of the character. The file name is

```
Cards.properties
```

that is important. There is also a different version of the file with English names written as:

```
di=Diamond
he=Heart
sp=Spade
cl=Club
```

and here the filename is

```
Cards_en_US.properties
```

and again the name is absolutely crucial. Then consider the following program:

```
package resourceprogram;

import java.util.*;

public class ResourceProgram
{
 public static void main(String[] args)
 {
  ResourceBundle bundle1 = ResourceBundle.getBundle("Cards");
  System.out.println(bundle1.getString("di"));
  System.out.println(bundle1.getString("he"));
  System.out.println(bundle1.getString("sp"));
  System.out.println(bundle1.getString("cl"));
  ResourceBundle bundle2 = ResourceBundle.getBundle("Cards", Locale.US);
  for (Enumeration<String> keys = bundle2.getKeys(); keys.hasMoreElements(); )
  System.out.println(bundle2.getString(keys.nextElement()));
 ResourceBundle bundle3 = ResourceBundle.getBundle("Cards", Locale.GERMAN);
 System.out.println(bundle1.getString("di"));
 }
}
```

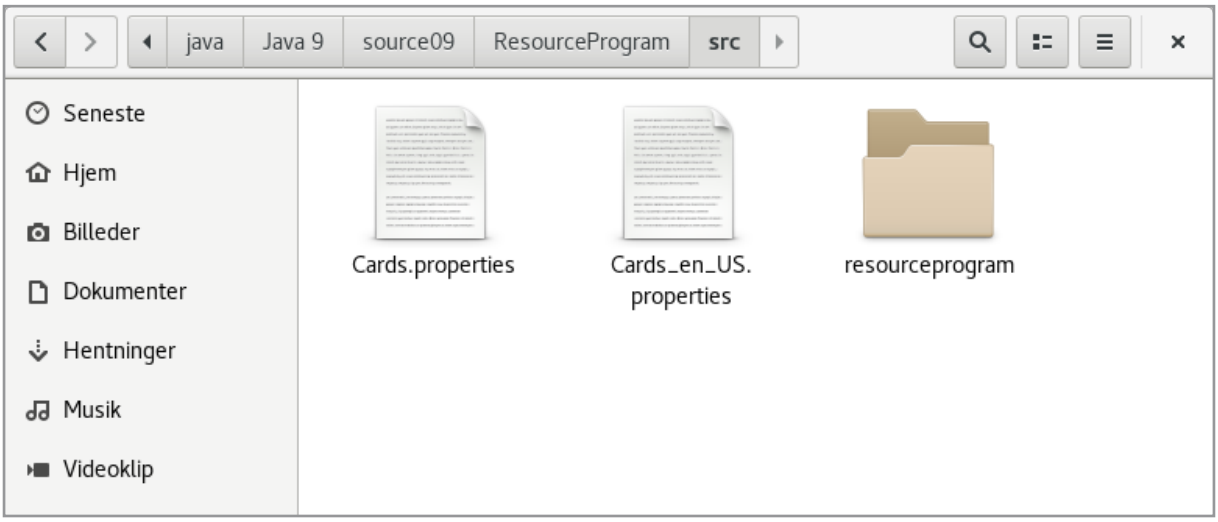
Running the program is the result:

```
Ruder
Hjerte
Spar
Klør
Club
Diamond
Spade
Heart
Ruder
```

The program starts by creating a *ResourceBundle* named *bundle1*, and it will when not specify any *Locale* read the file *Cards.properties*, which is called the *base properties file*. Next, the values for the four keys are printed, and the result is the first four lines. Next, another *ResourceBundle* is created with the name *bundle2*, but this time a *Locale* is added. The result is that the file for this *Locale* is read and that is *Cards_en_US.properties*. The next statement is iterating over all the keys and printing the English names. Finally, a third *ResourceBundle* is created and called *bundle3*, but with a local for German. It will then look for the file *Cards_de.properties*, but such a file does not exist and it will then read the file *Cards.properties* which is the base properties file. Therefore, the last statement writes the text

```
Ruder
```

*ResourceBundle*s based on properties provide the great advantage that they can be maintained outside of the program and thus make it flexible to language versions of programs. However, it should be noted that the keys can not be changed as they are referenced from the program. Another question is where the properties files should be and they should simply be placed in the project directory in the *src* directory (se below). Should the program be installed elsewhere (as is the case in practice), the properties must be copied together with the jar file and placed in the same location.



9.2 FORMATTING VALUES

In the context of internationalization of programs, formatting of numbers plays an important role, and even more important is the formatting of date and time. It is partly about how these values are displayed, but also how to parse a string in connection with data entry.

Date and time have long been represented by the *Calendar* interface, implemented by the class *GregorianCalendar*, but now is added a package *java.time* containing new types for dates and time. I do not want to treat these types here but just point out their existence and there are actually some types and you are encouraged to examine the API. To indicate what it is, the *DateTimeProgram* application is a simple console application that shows how to format and enter dates and times. As an example, the following method shows how to create an *Instant* object, which is the new type of time:

```
private static void test1()
{
  Instant dt = Instant.now();
  System.out.println(dt);
  System.out.println(String.format("%d %d", dt.getEpochSecond(), dt.getNano()));
}
```

Here you should note the method *getEpochSecond()* that returns the number of seconds after 1970-01-01T00:00:00, while the method *getNano()* returns the number of nano seconds within the current second.

The method *test2()* shows how to convert between the new representation of times and *Calender* objects:

```
private static void test2()
{
 Calendar dt1 = Calendar.getInstance();
 Instant dt2 = Instant.ofEpochMilli(dt1.getTimeInMillis());
 Calendar dt3 =
  GregorianCalendar.from(ZonedDateTime.ofInstant(dt2, ZoneId.systemDefault()));
 System.out.println(dt1);
 System.out.println(dt2);
 System.out.println(dt3);
}
```

The next method shows how to convert dates and times corresponding to a given *Locale*:

```
private static void test3()
{
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.SHORT),
  DateFormat.getTimeInstance(DateFormat.SHORT));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.SHORT,
  Locale.US), DateFormat.getTimeInstance(DateFormat.SHORT, Locale.US));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.MEDIUM),
  DateFormat.getTimeInstance(DateFormat.MEDIUM));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.MEDIUM,
  Locale.US), DateFormat.getTimeInstance(DateFormat.MEDIUM, Locale.US));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.LONG),
  DateFormat.getTimeInstance(DateFormat.LONG));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.LONG,
  Locale.US), DateFormat.getTimeInstance(DateFormat.LONG, Locale.US));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.FULL),
 DateFormat.getTimeInstance(DateFormat.FULL));
 print(Calendar.getInstance(), DateFormat.getDateInstance(DateFormat.FULL,
 Locale.US), DateFormat.getTimeInstance(DateFormat.FULL, Locale.US));
}

private static void print(Calendar dt,
 DateFormat dFormatter, DateFormat tFormatter)
{
 System.out.println(dFormatter.format(dt.getTime()));
 System.out.println(tFormatter.format(dt.getTime()));
}
```

Performing the method gives you the result:

```
27-04-17
11:49
4/27/17
11:49 AM
27-04-2017
```

```
11:49:27
Apr 27, 2017
11:49:27 AM
27. april 2017
11:49:27 CEST
April 27, 2017
11:49:27 AM CEST
27. april 2017
11:49:27 CEST
Thursday, April 27, 2017
11:49:27 AM CEST
```

Finally, the last test method shows how to use a formatter to parse a string representing a date and a time, but where I only have shown one of the two input methods:

```
private static void test4()
{
 System.out.println(enterDate(DateFormat.getDateInstance(DateFormat.MEDIUM)));
 System.out.println(enterDate(DateFormat.getDateInstance(DateFormat.MEDIUM,
  Locale.US)));
 System.out.println(enterTime(DateFormat.getTimeInstance(DateFormat.SHORT)));
 System.out.println(enterTime(DateFormat.getTimeInstance(DateFormat.SHORT,
  Locale.US)));
}
```

```
private static Date enterDate(DateFormat formatter)
{
 Scanner kb = new Scanner(System.in);
 while (true)
 {
  System.out.print("Enter date: ");
  try
  {
   return formatter.parse(kb.nextLine());
  }
  catch (Exception ex)
  {
   System.out.println("Illegal date…");
  }
 }
}
```

To format numbers, you have the *NumberFormat* class and in this context it is important that the class supports the use of a *Locale* object. The class is illustrated with the following program:

```
package numberprogram;

import java.text.*;
import java.util.*;

public class NumberProgram
{
 public static void main(String[] args)
 {
  test1();
  test2();
 }

 private static void test1()
 {
  System.out.println(NumberFormat.getNumberInstance().format(123456.789));
  System.out.println(NumberFormat.getNumberInstance(Locale.US).
   format(123456.789));
  System.out.println(NumberFormat.getCurrencyInstance().format(123456.789));
  System.out.println(NumberFormat.getCurrencyInstance(Locale.US).
   format(123456.789));
  System.out.println(NumberFormat.getPercentInstance().format(123456.789));
  System.out.println(NumberFormat.getPercentInstance(Locale.US).
   format(123456.789));
 }
```

```
 private static void test2()
 {
  System.out.println(enterNumber(NumberFormat.getNumberInstance()));
  System.out.println(enterNumber(NumberFormat.getNumberInstance(Locale.US)));
 }

 private static Number enterNumber(NumberFormat formatter)
 {
  Scanner kb = new Scanner(System.in);
  while (true)
  {
   System.out.print("Enter number: ");
   try
   {
    return formatter.parse(kb.nextLine());
   }
   catch (Exception ex)
   {
    System.out.println("Illegal number…");
   }
  }
 }
}
```

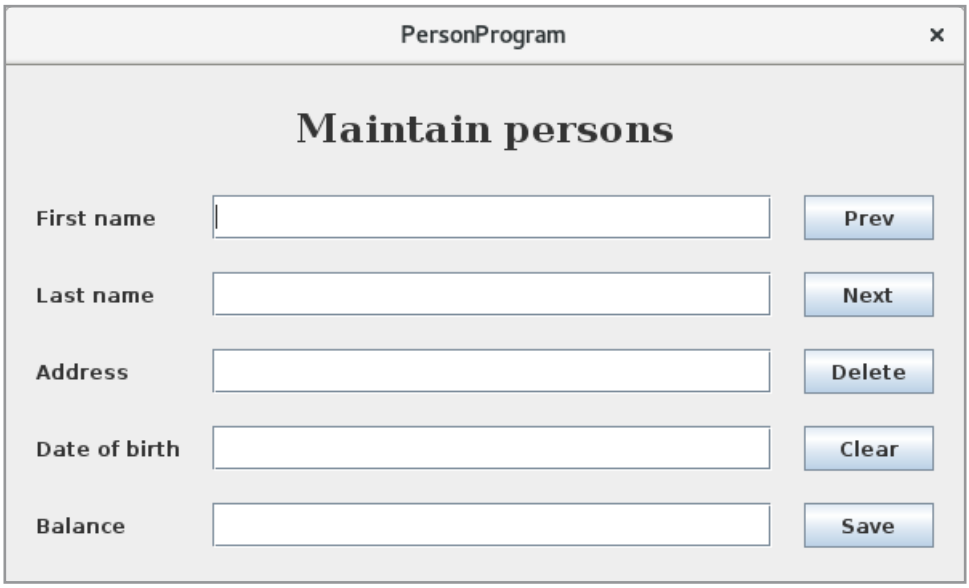
If the program is executed, the result could be:

```
123.456,789
123,456.789
kr 123.456,79
$123,456.79
12.345.679%
12,345,679%
Enter number: 123456,789
123456.789
Enter number: 123456.789
123456.789
```

EXERCISE 9

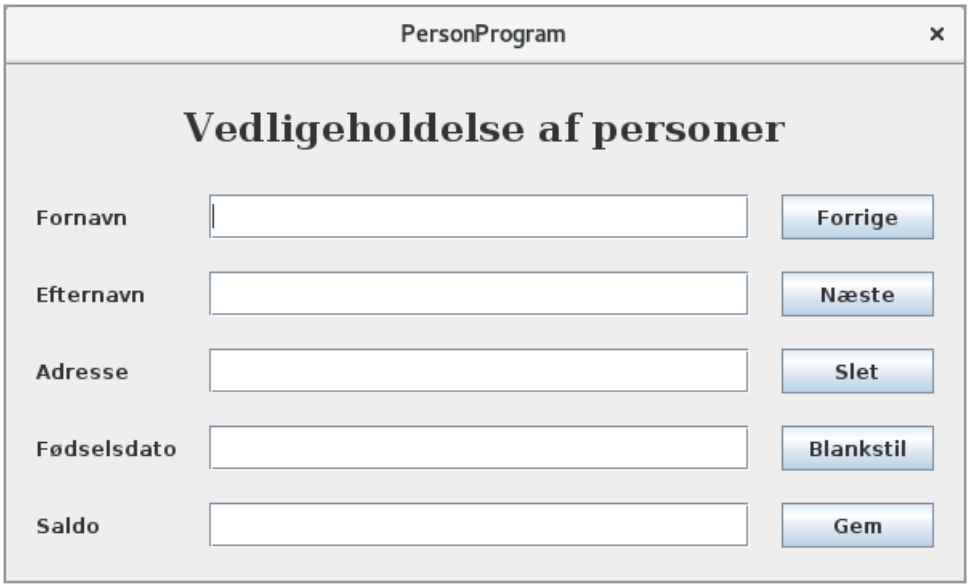
Create a copy of the program *PersonProgram*. If you run the program, the window below opens.

It is a simple program where you can create a person with a date of birth and a balance. The persons are saved in an *ArrayList* that is serialized whenever a person is created, a person is changed or a person is deleted. You can browse the list using the two top buttons. Test the program and study the code so you are sure you know how the code works.



The program is not internationalized and it is your task to language versioning the program. Start by creating a *base properties file* with English texts there must be a key / value pair for all texts that the program uses. Replace all texts in the program with texts from a *ResourceBundle* for your properties file.

Then write another properties file but with texts in another language (according to your choice). In the class's constructor, you can set default *Locale* to this language, and check if the program then uses your chosen language. Below is a version of the program where the language is Danish:



And what about the formatting of the date of birth and the balance – does it work as expected?

10 A SLOT MACHINE

As the final example of this book, I will show the development of a program that will simulate a slot machine. The aim is, of course, to show the use of some of the concepts introduced in this book, but where the focus is also on the process. A slot machine consists of (typical) three wheels on which some figures are placed. When you play the machine, the wheels rotate and the figure combination displayed when the wheels stops, determines whether there is a win and, if so, how much the winnings are. To write a program that can simulate such a machine and thus with wheels that rotates, I need concepts that are dealt with first in the next book, and the code can therefore first be written after you have read the next book (or equivalent about Java). The following version of the program will therefore not simulate rotating wheels, but just show the figures.

As mentioned, I want to focus on the process. At least on analysis and design, but the program is relatively large and during the programming I will primarily describe what is made and what you should especially notice when you read the code, but not much about how it was made since the number of pages else will be very extensive. For the same reasons, I will not describe how to write an installation script, but you can do it in the same way as shown for other programs.

10.1 TASK FORMULATION

A program must be written to simulate a slot machine on a regular PC. There must be some form of user administration, and in order to play the machine, you must be created as a user with an account and have deposited money on the account. It must also be possible to receive an amount from the account if there is money on it which in practice will mean you have won.

You should not be able to play with “real” money, but the program must be able to simulate that the player puts in money and that the player gets paid money.

The machine must be configurable and you should be able to choose whether to play with 3 or 4 wheels. In terms of configuring the machine, one must be able to choose which and how many figures there should be on each wheel, which combinations should give a win and what the winnings should be, and finally it should also be possible to choose which images to use. It must be possible for the player to choose from several configurations. When setting up a configuration, ensure that in the long-term configuration provides profits to the owner of the machine (the house), but at the same time, the machine must often make a gain to be sure that the machine gives so much back that there is someone who want to play on the machine.

There should also be an opportunity for some kind of statistics, where you can see when the machine gives a win and how big the winnings are.

10.2 ANALYSIS

The following requirement specification has been prepared on the basis of an analysis of the task formulation as well as contact with the customer for clarification of wishes and requirements. In addition, a prototype has been developed for the application's user interface.

REQUIREMENTS SPECIFICATION

The machine can be in one of two modes

1. User mode
2. Admin mode

where in the last mode besides playing, you can configure the machine and administrator users. The requirement specification is divided according to these two use patterns.

User mode

In user mode, this is a simple program with few features. When a user (a player) meets the program, the player must do one of the following:

1. Log in with username (email address) and password
2. Register yourself as a user (if the player is not already a user)

If the player has forgotten his password, the player must contact the administrator, who may change the password. If the player is logged in, the player can change the password at any time.

In the latter case (where a new player has to be created) the user must inform

- email address
- password
- name
- phone number
- debit card (number, expiry date, control number – should simulate a real card)

After that, the user can play the machine if there is money on the account. In addition to play, the user can perform the following functions:

1. Log out
2. Deposit money on the account
3. Display account information and change these (but not the email address), and at the same time be able to get paid money if there is money on the account
4. View a summary of winning combinations for the current configuration and what the winnings are
5. View an account summary that shows transactions and winnings
6. Select another configuration

In terms of payment, the program must simulates that money is raised on the player's account, and if you put money on the account, the program must simulate the deposit of the player's account. All transactions must be recorded so that a player can always see how much has been inserted and raised.

Of course, the most important feature is to play, which is performed by clicking on a button or equivalent. It must be possible to hold the individual wheels similar to the following:

1. You may not hold all wheels
2. You should not be able to hold a wheel if it was held in the previous game
3. You can not hold a wheel if there was a win in the previous game

When playing the machine, the result of a game must be registered, where the following information should be registered:

1. the user who has played
2. the time for the play
3. the configuration for play
4. the result as the wheels combination
5. the win (0, if there is no gain)

Admin mode

If you log in as administrator, besides playing the machine, you can configure it and manage user accounts.

Regarding the configuration of the machine, you must be able to maintain specific configurations where a configuration indicates:

1. A name so the configuration can be chosen by the players
2. Number of wheels (3 or 4)
3. Which figures are to be used on the wheels (typically in the neighborhood of 10)
4. The position of the figures (order) on the wheels – the same figure may appear several times
5. Which combinations should give a win and the amounts of the winnings
6. What the cost is to play on the machine
7. If configuration is active and can be selected by players

Regarding the administration of player accounts, you should be able to:

1. Change player's password (for example, if forgotten)
2. Delete a player where the player's balance is transferred to the player's account

Finally, the administrator should be able to print a statistic that can show how each configuration behaves. Here you can see, among other things

1. Whether a configuration gives positive returns or not
2. The distribution of the single winnings over time

THE PROPTOTYPE

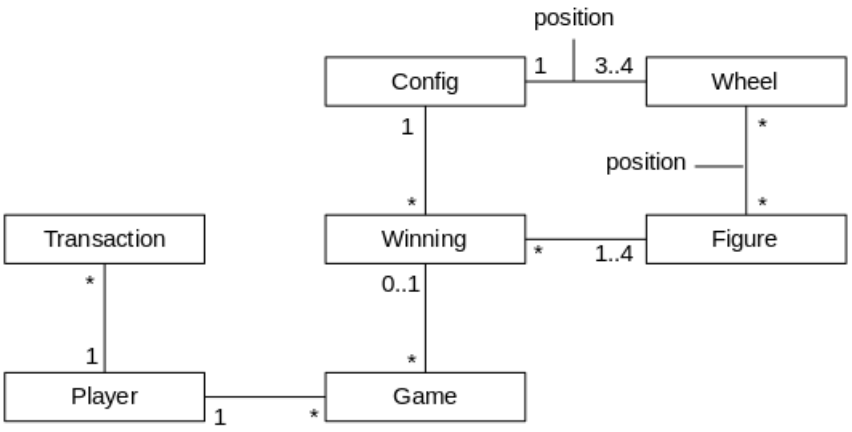
A prototype has been developed to illustrate the user interface and how it will be playing on the machine. If you click on the game button, the prototype selects random figures (there are 15 figures available), but the figures are changed for a period of time to simulate that the wheels rotates. The toolbar at the top of the window is the features available to the player, where the icon on the right is intended as login to the administrator. The prototype is called *SlotMashine0*.

10.3 DESIGN

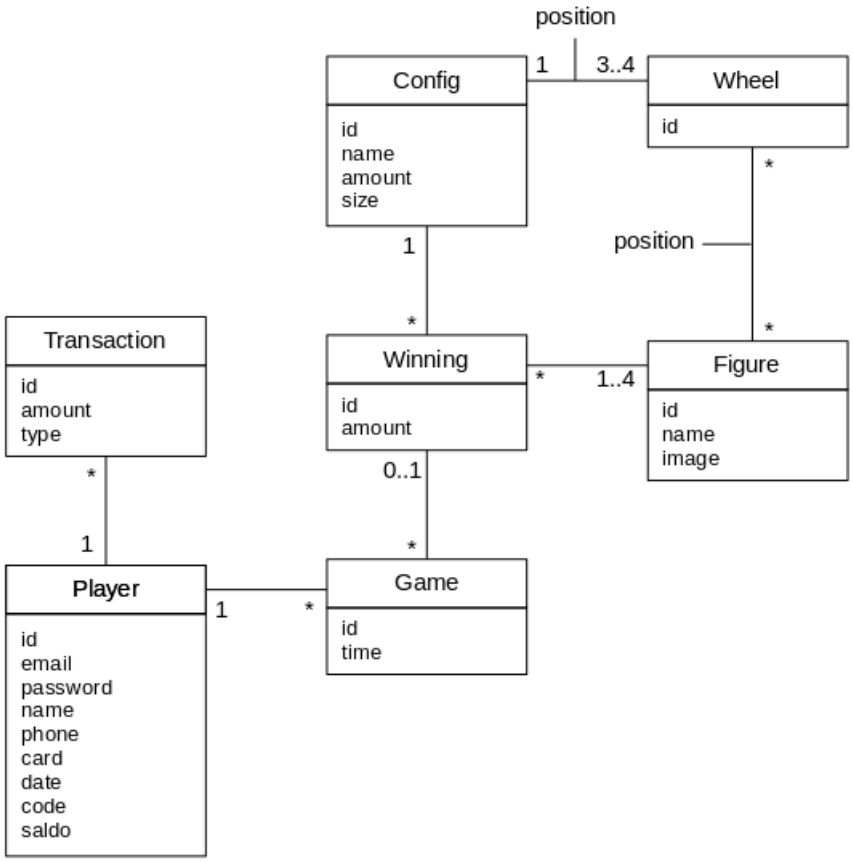
Information about players and configurations must be saved somewhere, and I will use a database. I will start with the design of this database, and the design is outlined in the following ER diagram, where there are the following entities:

1. *Config* that represents a concrete configuration of the machine
2. *Wheel* which represents a wheel for a configuration
3. *Figure*, that is a figure on a wheel
4. *Winning* har represents a gain with an amount for a given combination of figures
5. *Player* which represents a player
6. *Game* that rrepresents a game on the machine for a player
7. *Transaction* that is an amount that the player has inserted or raised on the account

The attribute between *Config* and *Wheel* indicates the position of the wheel, and in the same way, the attribute between *Wheel* and *Figure* indicate the position of a figure on a wheel.



If you add attributes to the diagram, the conceptual database design can be illustrated as follows:



Datadictionary:

Player		
id	INT autonumber	primary key
emal	VARCHAR(100)	must be unique
password	VARCHAR(200)	encrypted password
name	VARCHAR(50)	players name, NOT NULL
phone	VARCHAR(20)	players phone number
card	VARCHAR(20)	players cardnumber, NOT NULL
date	DATE	expiration date for card, NOT NULL
code	INT	card control code, NOT NULL
balance	DECIMAL	
Transaction		
id	INT autonumer	primary key
amount	INT	the amount of the transaction, NOT NULL
type	BOOLEAN	true = deposited, false = paid amount from
Config		
id	INT autonumner	primary key
name	VARCHAR(50)	the configurations name, NOT NULL
amount	INT	price to play with this configuration, NOT NULL
size	INT	number of wheels, must be 3 or 4
Figure		
id	INT autonumber	primary key
name	VARCHAR(50)	the name of the figure, NOT NULL
image	BINARY	the figure, NOT NULL
Wheel		
id	INT autonumber	primary key

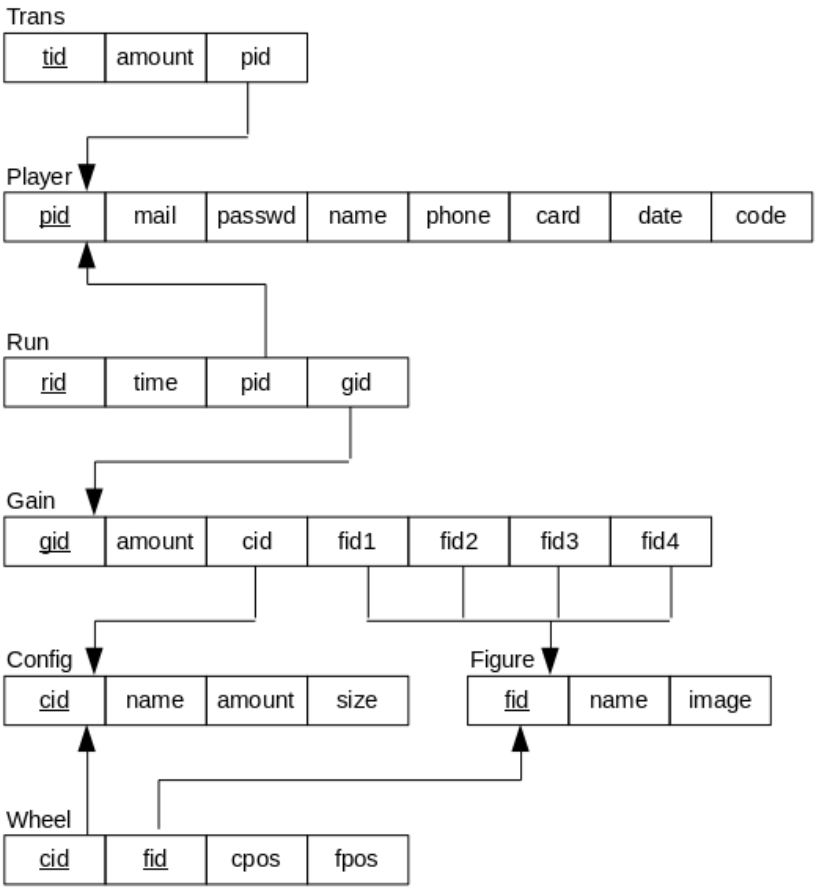
Winning		
id	INT autonumber	primary key
amount	INT	the winning, may be 0
Game		
id	INT autonumber	primary key
time	DATETIME	date and time for the winning

The above ER model can be mapped to a relational model as shown below. In addition to names, there are only a few changes.

Regarding transactions, the type has been removed and it has been decided that a positive transaction means that it has been inserted into the account while a negative transaction means that the account has been raised.

The *Wheel* entity has been changed so that it is a relation between *Config* and *Figure*, and the two positions are attributes in this relation, indicating the wheel number and the position of a figure, respectively.

All relationships are simple and are in third normal form, and with reference to the above data dictionary, I can immediately write a script and create the database. The script is not shown here.



As part of the design, a NetBean project has been created as a copy of the project from the analysis that defines a design of the model layer.

There is also a program (the project *CreateDefault*) that creates a default configuration. The goal of this program is to have a configuration in place before the programming starts.

With regard to figures (images) for the wheels it has been decided that the machine must use square images in a small resolution so they does not fill too much.

Following these steps, two additional changes have been added to the database:

1. The table *figure* is expanded with a column *series*. The column contains a short text, and is used to categorizing the figures. The above program has uploaded 16 figures to the database, which is located in the category *Default*.
2. The database is expanded with a simple table *admin* with two columns. The table must contain the password and email address of the administrator.

10.4 PROGRAMMING

If you have done a good and careful analysis and possibly combined with one or more prototypes, you should have all the requirements in place at the start of the programming phase. It is at least the theory and, with practice, it will also work for smaller projects. However, one must also acknowledge that no matter how careful you has been, then there during the programming can happen changes to the specification, including wishes for something that should work in a different way or maybe even new wishes. In fact, it is not strange, and during the development of a program, the understanding of the application's use and also the possibility of recognizing new features that the program should also have or something that is agreed in the specification that can be advantageously modified all may cause changes. Obviously, the larger a program, the greater the chances are that during the programming there will be a desire to change the requirement specification.

Should it happen – or when it happens – you must contact the customer and present the changes, including what the changes will mean in terms of resource usage and the time horizon for when the program can be completed. Then there must be an agreement as to whether the new wishes are included in the project or not, or they may have to be postponed to later. In practice, it is important that such wishes come true, but it is also important that, as a developer, you not just implements things that are not part of the requirement specification, since ultimately it is the customer who decides (and pays for) what should be part of the finished program.

The conclusion is that even after a thorough in-depth analysis, there is an opportunity for during the programming there will be wishes to change the requirement specification, wishes that may be clarified with the customer.

The same applies to the design, and here is more the rule than the exception. This is due to several factors, among other things, that the boundary between design and programming is not sharp. Thus, there will almost always be changes in the design, and usually changes that must be made by the developer without the customer's approval.

A typical design activity is database design, and changes will almost always occur during the programming phase. This does not mean that the database design is not important during the design phase, as programming typically just involves minor adjustments, but they will, almost certainly, also be there. This is also the case with the current program, where some columns have been added to some of the tables, as well as changes to the foreign keys. It is worth paying attention to the design of the database during the design phase to avoid this kind of changes during programming, but vice versa, with just a small size database, it is difficult to avoid any adjustments during programming. When such changes are problematic, it is because they sometimes means that the database data content needs to be recreated, which may be time consuming. In this case, some changes have meant that it has also been necessary to change the initialization program *CreateDefault* and it has been necessary to run the program again.

Another design activity is the design of the model layer and including the most important model classes. It is rarely associated with the major challenges, but you must be prepared to make changes to these model classes and to create new ones. It is not a problem and just states that during the design you are not at the same level of details as during programming, and that is not the idea, as the design should have focus on the overall concepts alone.

As mentioned, the boundary between design and programming is not sharp, which can result in significant decisions are postponed to the programming phase. In general, you should be careful and there is a tendency or the risk under the design not to find challenges and problems that should be solved. This indicates that during the design, sufficient time is not used, thus neglecting or underestimating problems that may occur during the programming. It's hard to put the right distinction between design and programming, but conversely, just under the design work with the big lines and not writing the program, but the result is, that if significant decisions are postponed for programming, there is a great risk that parts of the code that is written must be changed or, at worst, rewritten. Therefore, it is important that all major challenges are dealt with before you write the program.

In connection with the development of this program, there are actually examples of tasks that should have been treated better during the design and, for example, I would like to mention two.

It is part of the requirements that the administrator must be able to test the individual configurations as to how they earns and how much the house earns on the configuration. How it will happen is not treated in either the analysis or the design. That it has not been processed during the analysis is an indication that it allows developers to come up with a solution, which is not unusual, and therefore there should be a sketch of a solution under the design. For example, it should have been considered whether the database has the data needed to make a reasonable analysis of a given configuration.

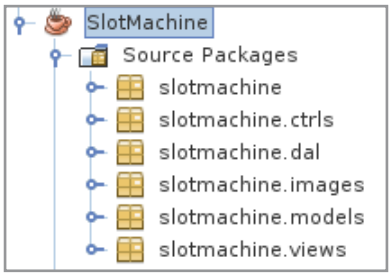
Another issue is what should happen if you change or delete a configuration. Doing so, the data recorded for games with the configuration is not necessarily longer valid, meaning that information about the individual player's use of the machine is no longer valid. The data in question can not only be deleted, as it also deletes how much a player has won or lost. It is also an example of a problem that should have been solved during the design.

THE CURRENT PROGRAM

With a slightly larger program, the programming phase should be divided into multiple iterations, where an iteration is a defined as a part of the program that can be developed and tested independently, something that I have previously illustrated. This also applies to the current program, but I do not want to review the individual iterations this time, but only focus on the final result, pointing out what has been made and where the biggest challenges have been.

I want to use the library *PaLib*, and to the extent that it does not contains the required library methods it should be updated. The result of the task is thus both the current program and an updated library.

The program has a classic MVC architecture:

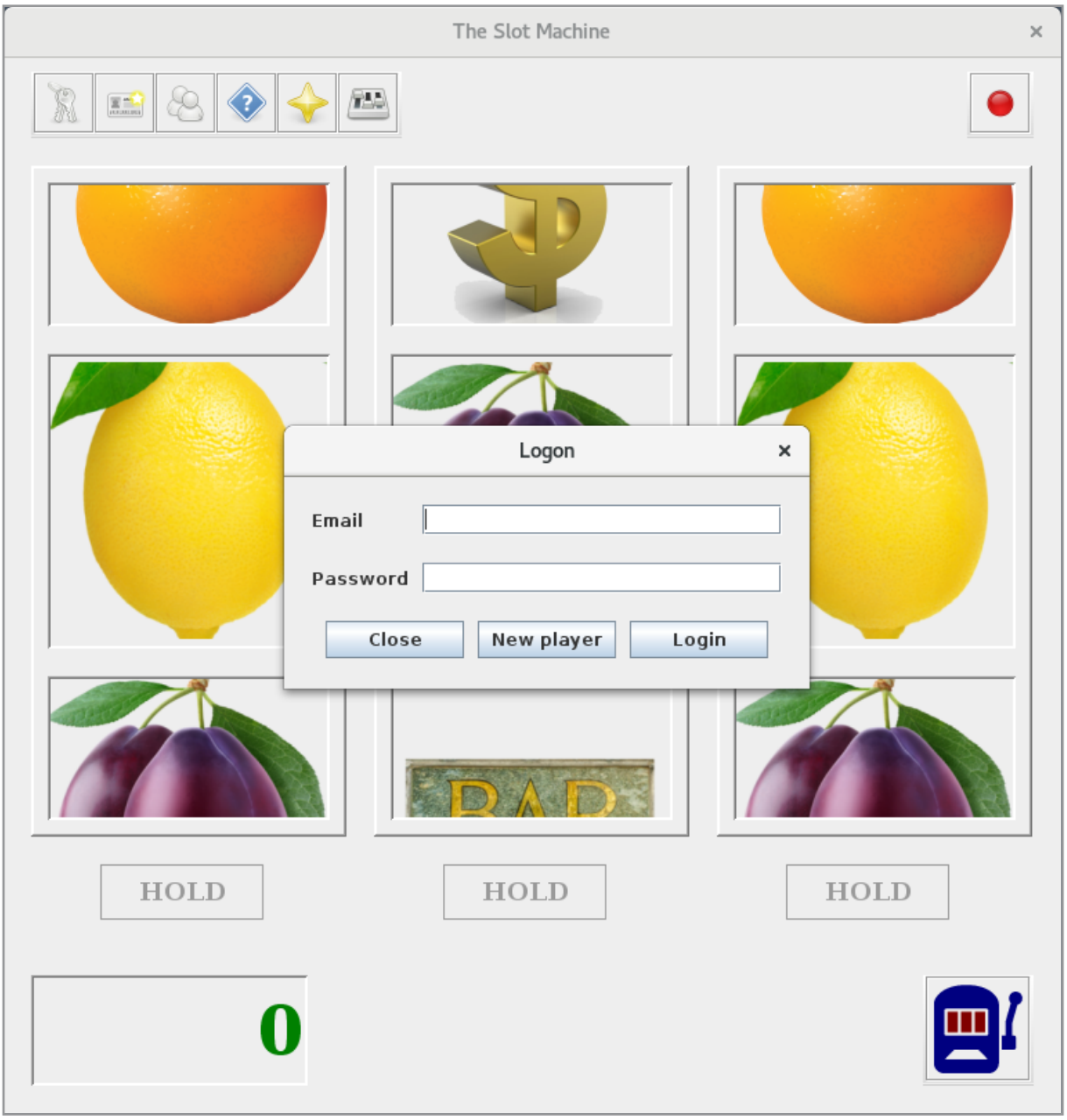


The *dal* layer has only one class called *DB*. The class is written as a singleton and has all methods that maintains the database. The class is extensive, but does not contain anything new.

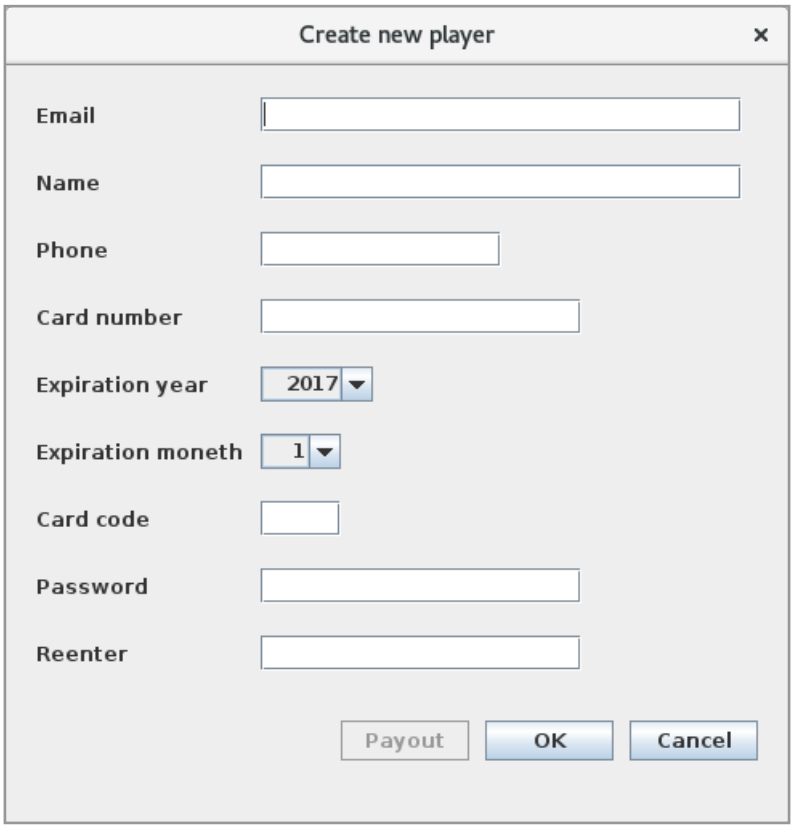
The model layer has the classes from the design. They are all modified where the methods are implemented, and some new classes have also been added, but in general there are few and simple extensions. Specifically, there is the class *Repository*, which is a fairly thin adapter class to the dal layer. In principle, the class is meaningless as it does nothing but delegate call of methods to the class *DB*, and its sole purpose is that the dal layer should not be known in the view and controller layers.

A very large part of the code is in the view layer, which defines no less than 33 types (classes and interfaces). The number exaggerates as several classes are model classes for *JTable* components as well as *CellRenderer* classes, and many of the other classes relate to simple dialogs. This corresponds to that there are almost two windows, that are not simple dialog boxes, and the control layer has only 7 classes.

When you opens the program, you will meet the following window:



If you click on *Close* in the *Logon* dialog box or click on the cross in the upper right corner, the program ends so you can not play before there is a player. If you are not a player, you can click on *New player*, and you will get a dialog box where you can be created as a player:

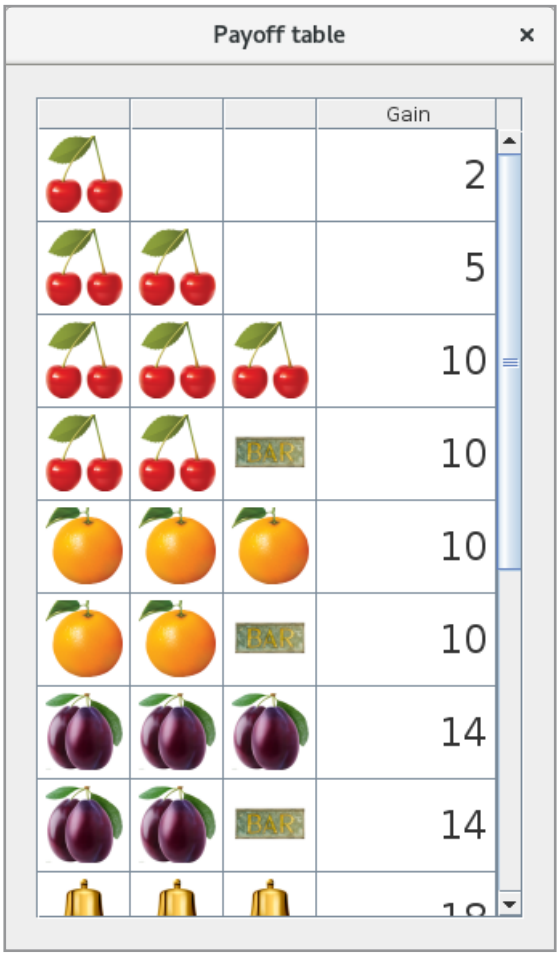

Create new player
Email
Name
Phone
Card number
Expiration year
2017
Expiration moneth
1
Card code
Password
Reenter
Payout
OK
Cancel

With regard to payment cards, it is pseudo, but the card number must be 16 digits, where a space may have been inserted between 4 digits groups. The check code must be 3 characters, but in the same way as the card number, nothing is used. Finally, there is the expiration date that the program checks against the current date when playing on the machine. The email address must have a legitimate format for an email address and must be unique as it is used as user identification. The password must be at least 10 characters.

If you create a user with legitimate data, the dialog closes and the dialog for the login window also closes and you are ready to play. The two dialogs for logon and creation of a user, both have a controller attached, but are otherwise simple. As for the dialog to create a player, it is the same dialog box that opens from the main toolbar if the player wishes to maintain the account.

To implement the game itself (and thus the *MainView* class, which is one of the two complex windows in the program), a class *WheelComponent* has been written, which is a view class representing a wheel, and *MainView* uses three instances or four of this component. It is this component that starts a thread that will simulate the wheels rotating. The game function resembles the prototype, and only minor changes have been made that primarily concerns rotation of the wheels. The game feature must also ensure that the logic of the hold buttons are properly implemented, to test whether there is a win and that the player's balance is updated. In addition, the feature must ensure that you can not play if the player's account is empty. These features are implemented in the class *MainModel*, and the database is updated for each game as it is a requirement that all games are written to a log (the table *run*). Compared to the database design, this table is expanded with two columns: The *amount* of the result of the game, and the *id* of the configuration played with. This is to simplify some of the other features.

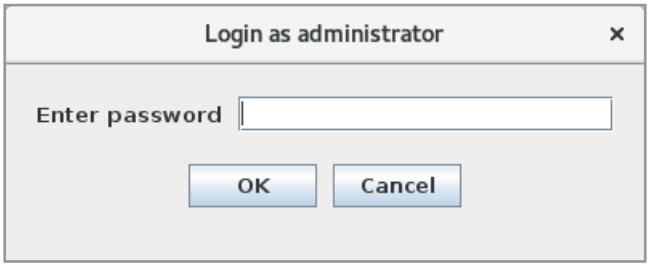
Then there are the features in the toolbar. The first feature is logout, and if you click on it, you will get the login window again. The next function is used to deposit amounts into the account and clicking on the button will give you a simple dialog box where you enter the amount and password. The last is to ensure that others do not suddenly deposit money into the account and thus raise on the registered payment card. The third feature of the toolbar is used by the player to maintain user data and, as mentioned, opens a dialog box similar to the above to create a user. A player can not change his email address as it is perceived as a key. The fields for password are empty, and you should not enter the old password. These fields must only be filled in if you want to change the password. The fourth features open a simple dialog box that shows an overview of which combinations give a win:



and the content is nothing but a *JTable* that shows the winning opportunities. The next last tool in the toolbar is used to display an overview of the current player's results. It is a dialog box with a *JTable*, where you can see the player's transactions in the account and which games have been. In principle, it is also a simple dialog box, but complicates a bit because of of filters for which transactions to display. Finally, there are the last tool in the toolbar, where you can choose a different configuration. Programmatically, it's only a *JTable* with an overview of active configurations. You select a configuration by double click the name of the configuration.

The above describes how the machine works for a player and, purely programmatically, the work is in the class *DB* to implement the database functions, the class *MainModel*, which will keep track of all the logic about playing the machine and finally the class *MainView* that contains all the visual. This class is complicated by the choice of 3 and 4 wheels, which means that the components in and layout of the window depends on this choice, and the same applies to the above-mentioned dialog with the win combinations.

Then there is the administration part. If you click the icon to the right of the toolbar, you get the following dialog box:

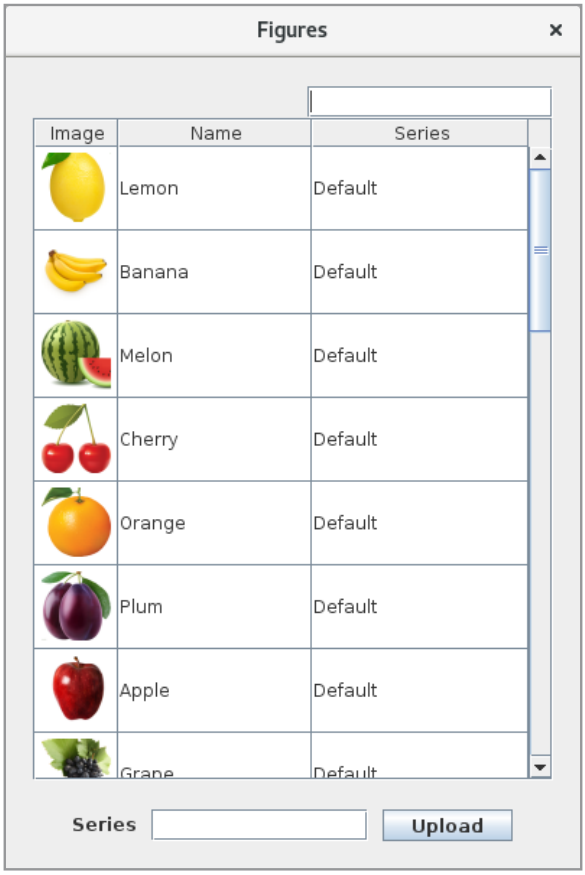


There can only be one administrator. You must enter the administrator password (which must also be at least 10 characters), and if the administrator is not created, the administrator will be created, and then you can log in as administrator with the appropriate password (as the administrator may change). You should note that this maintenance of administrator information for practical use is too simple and the solution might be that a particular user could be registered with administrator privileges – a problem that I will return to in the next book. After logging in as an administrator, the toolbar is expanded with 4 new icons:

which is used to upload images to the database, to maintain users, to maintain configurations, and to change the administrator's password. The last feature is simple and should not be mentioned further here. The user maintenance feature opens a dialog box with a *JTabel* that shows all users. If you double-click a user in the table, you get the following dialog box:

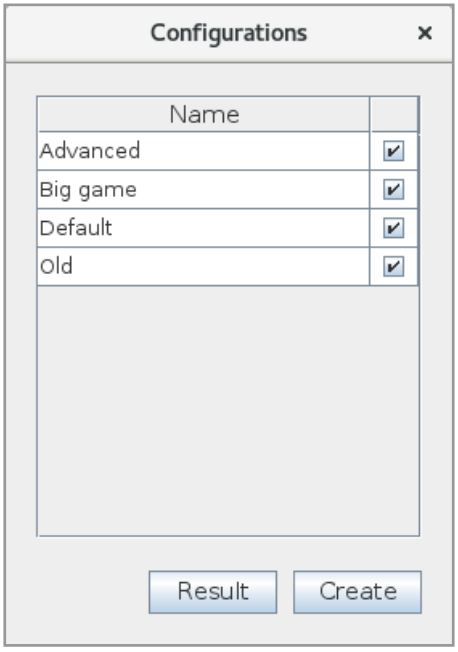
where the administrator can change the user's password (and thus assign the user a new password) and possibly delete the user.

Then there is the function to upload images to the database. If you select the function you get the following dialog box:

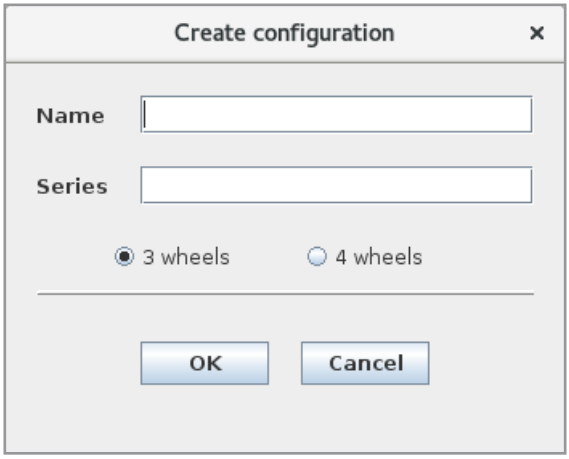


The dialog box shows all figures in the database, but you can set a filter on the series. When you upload a figure you most first enter then series name, and then you browse the file system for the image. The program use the filename as name for the figure, but you can edit the table and change both the name and the series. If you want to remove a figure, you can double-click on the image.

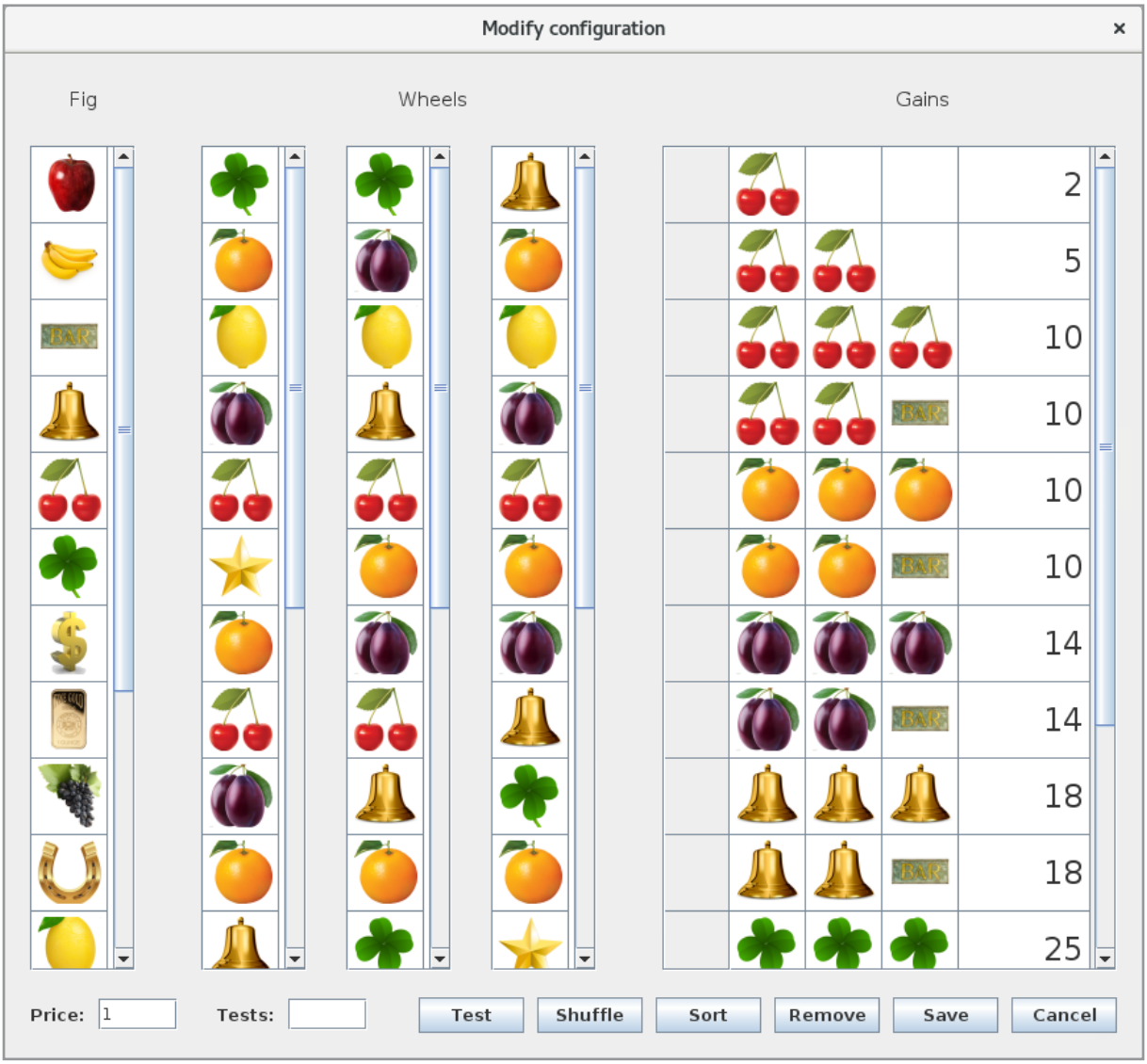
If you in the toolbar click the icon to maintain configurations you get the following dialog box:



that shows a table with all configurations. Here you can activate og deactivate configurations. If you click on *Result* you opens a window that shows the result of using this configuration and how much the configurations has returned. I will not show the window here. If you click *Create*, you can create a new configuration:



where you must enter the name for the configuration, the series og images to use, and the number og wheels. These data can not later be change. When you click *OK* you get the following window, that is the most complex dialog in the program (you get the same window when you double-click on a configuration in the dialog box *Configurations*):



To the left is a *JTable* with all shapes in the current series. In the center there is a *JTable* for each wheel, and to the right there is a *JTable* with all winning combinations. You add figures to the wheels and the winning table by dragging them with the mouse from the left table. If you want to remove a figure, just double-click it. To add a new row in the winning table you just drop a figure in the table, and to add a new figure to a row you drop the figure at the row header. In the winning table, the column with the value is editable so you can enter the amount. At the bottom there is an entry field for what it costs to play on the machine, and there are also the following buttons:

1. *Cancel* undo the changes.
2. *Save* that save the changes. Doing so will delete all historical data for this configuration, that is all rows in the table *run* regarding this configuration. The amounts are counted for each player and added as a transaction to the player's account.
3. *Remove* deletes the configuration, and here the same goes as above where the current rows in the table *run* are deleted and counted together and inserted as transactions on players accounts.
4. *Sort* sorts the gain table.
5. *Shuffle* blends the figures on the reels such they get a random location.
6. *Test* simulates a number of games with the configuration. The number of games is entered in the *Tests* field, and the result is a window with the test results. The goal of the function is to validate the frequencies for profit, and how much the machine returns before saving the configuration.

There are many details related to that feature, including implementing drag and drop, and among other things, the database functions are in the class *DB*.

10.5 TEST

After the program is finished, it must be tested and it is not easy to test an application like this, with the primary emphasis on the user interface. When you (as part of the programming) have found that the program seems to work as expected, there is not much more to do than simply to play on the machine and in the following I will outline what I have done.

Before the test I have cleaned the database (executed the script *slot.sql* and executed the program *CreateDefault*). The result is a database without players and with a single default configuration.

Then I have performed a test for how to play at the machine:

1. Create a player and inserts 100 on the players account
2. Play at least 100 games with the player
3. Log out and create a second player and inserts 1000 on the players account
4. Play at least 50 games with that player
5. Close the program
6. Start the program again and create a third player
7. Log out and then log in as the first player and plays at least 50 games
8. Check the players gains by clicking the button in the toolbar
9. Check the overview of winnings by clicking the button in the toolbar
10. Close the program

To test the aministrator functions I have change the last statement in the constructor for *MainView* as:

```
// new LoginView(this, model);
try
{
  model.setPlayer(Repository.getPlayer("poul.klausen@mail.dk",
  palib.util.Tools.encrypt("1234567890".toCharArray())));
  model.setAdminMode(true);
}
catch (Exception ex)
{
}
```

where the player is hardcoded (as the first of the above three players) and the machine is in admin mode (I do not the have to login all the time). Then I

1. Opens the program
2. Log out as administrator (click the right button in the toolbar)
3. Log in as administrator by entering a password (if it is the first time it means that you creates an adminstrator password)
4. Change the administrator's password
5. Log out as administrator
6. Log in as administrator again (with the new password)

Next I have created a configurationen called *Big game*, that uses de same figures as the default configuration. The new configuration has fewer but bigger winnings, and the price to play is 2. After sawing the configuration, I opens it again and try to test it with 1000 and 10000 runs.

Then I in the same way creates another configuration called *Advanced*, but a configuration with 4 wheels.

Now I have three configurations, but only one is active and can be used by the players.

The folder for this project has a subfolder called *Figures2*, that have some gif images for a slot machine. Next I have uploadet this figures to the database as a series with the name *Old figures*.

As the next step I have created a configuration with the series *Old figures*. The configuration has 3 wheels and is called *Old*. The configuration looks like the default configuration.

Then I have done the following:

1. From the administrator icon for users I opens the table with the users (there are three)
2. I changed the password for the user that above was created last
3. Click the icon for select a configuration as a player – only the default configuration is active
4. In the window for configurations the three new configurations are defined as active
5. Click the icon to select configuration again now there should be four configurations
6. Close the program

Next I have changed the *MainView* again, such I have to login. Then I opens the program and performs the following:

1. Log in as the first of the above three users
2. Play 20 games with the default configurationen
3. Select the configuration *Big game*
4. Play at least 50 games with that configuration
5. Select the default configuration again
6. Play 20 games with the default configuration
7. Log out
8. Log in as the second player
9. Select the configuration *Advanced*
10. Play 100 games with that configurationen
11. Log in as administrator
12. Open the configuration *Advanced*
13. Shuffle the wheels and save the configuration
14. Log out as administrator
15. Check the transactions for the current user.
16. Log out
17. Log in as the last user
18. Play 100 games with the *Old* configuration
19. Log out
20. Log in as the first user
21. Log in as administrator
22. Delete the *Old* configuration
23. Delete the two last users

If all goes as expected, I will consider the program to be completed – at least to be able to be used by house-selected test users.