Poul Klausen

JAVA 2

Programs with a graphical user interface

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 2: PROGRAMS WITH A GRAPHICAL USER INTERFACE

SOFTWARE DEVELOPMENT

Java 2: Programs with a graphical user interface: Software Development
1st edition

ISBN 978-87-403-1643-8
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the second 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. The subject of the current book is an introduction to development of programs with a graphical user interface and thus also an introduction to *Swing*. Programs with a graphical user interface is also called GUI programs and *Swing* is the Java's API for the development of GUI programs. *Swing* is extensive and is first treated in detail in the book Java 9, and the goal of the current book is to present as much of *Swing*, that the reader will be able to write small applications that have practical interest. The book assumes a basic knowledge of Java corresponding to the book Java 1 of this series, but since the objective is that the reader soon should be introduced to the development of programs with a graphical user interface the book bypasses many object-oriented concepts, which naturally are part of a book about GUI programming. These concepts are discussed in detail in the next book in the series.

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

This book is an introduction to writing GUI applications in Java using the *Swing* API. Far from everything is treated, but conversely sufficient that you after reading the book can write small applications with a graphical user interface.

A program with a graphical user interface is a program that runs in a window. Since it is largely based on the principles of object-oriented programming, the topic should be examined only after the next book in the series of this books, and the following will also use a lot about classes and interfaces that are first mentioned there. When I still have chosen to address the subject now, it is because I want to be able to write applications that are more interesting and better match the programs that you meet in everyday life, than what is possible purely as commands or console applications.

To write a program with a graphical user interface you must work on basis of a wide range of finished classes, for example classes that creates and opens a window, classes representing a button, classes for an input field, etc. These are very many classes, and they are assembled in an API called *Swing*. In fact, Swing classes are based on an older API called *AWT*, so there are two APIs that you have to learn about, and each of these has several packages of classes. In the following I will call programs with a graphical user interface for *GUI* programs where GUI stands for *Graphical User Interface*, and I will start to make it clear that you can not learn to write GUI applications by learning all the many classes and their methods. Instead, you must learn some basic principles for the development of these kinds of programs, and once you have learned it, it's all not so difficult. You quickly get an idea of what it takes, and the question is what it's all are called and what you actually have to write. Here, however, there is only one way and that is to turn up the help, which fortunately is online available and there is also a variety of other sources on the Internet that provides advice on how to solve specific problems.

It is very different to writing GUI applications than console applications, and immediately it seems as if you have to write a lot and you also has to (at least as I will introduce *Swing*), but you will also quickly find that there is a lot of repetition and it is the same you have to write every time. Therefore, to reducing the task after you've written the first programs, and to reduce the work, you can create your own class library with frequently used methods. You will find an example of that in the book's appendix.

A GUI application will typically consist of several or many classes, but starting you can think of each window as a class. In addition, the data that a program must treat, are usually also be defined using classes, and possibly there will also be classes that implement the logic that must process the application's data. It is primarily in this context, that I am missing something of the theory of classes and interfaces, as first is explained in the third book on object-oriented programming. The following will therefore to some extent deal with object-oriented concepts, but so that I postpone all details for later.

The focus is on how to write a GUI application using Swing, and I will mainly use the most basic components, while more complex components is delayed. The goal is that after reading this book and in detail studied the related examples and solved the related exercises and problems you should be able to write less GUI programs for practical use.

2 HELLO SWING

I'll start a little, like I did in the first book to write a simple program, but this time a program which opens a window where you can enter a name. When you then click the *Add* button, the entered name is added to a list box, and you can enter a new name. At the bottom of the window, there is also a button, allowing you to delete the contents of the list box.

In addition you can, what you else can with a window. You can move the window, and you can change the window size. Right-clicking on the title bar, you get the usual menu and double-click at the title bar, the window maximizes. If for a moment you think about it, it's actually a lot that the program can do.

For writing the program I have in NetBeans created a common project in the same way as in the first book, and I have called the project *HelloSwing*. Next, I added a class to the project, called *MainWindow*, and it is the file for this class, which contains all the code:

```
package helloswing;

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

public class MainWindow extends JFrame
{
 private JTextField txtName = new JTextField();
 private JButton cmdAdd = new JButton("Add");
 private JButton cmdClr = new JButton("Clear");
 private JList lstNames;
 private DefaultListModel model = new DefaultListModel();
```

```
public MainWindow()
{
 setTitle("Hello Swing");
 setSize(500, 300);
 setDefaultCloseOperation(EXIT_ON_CLOSE);
 addListeners();
 createWindow();
 setVisible(true);
}

private void createWindow()
{
 add(createLabel(), BorderLayout.NORTH);
 add(createLabel(), BorderLayout.WEST);
 add(createLabel(), BorderLayout.EAST);
 add(createLabel(), BorderLayout.SOUTH);
 JPanel panel = new JPanel(new BorderLayout());
 panel.add(createTop(), BorderLayout.NORTH);
 panel.add(createBottom(), BorderLayout.SOUTH);
 panel.add(createCenter());
 add(panel);
}

private JLabel createLabel()
{
 JLabel label = new JLabel("");
 label.setPreferredSize(new Dimension(10, 10));
 return label;
}

private JPanel createTop()
{
 JPanel panel = new JPanel(new BorderLayout(10, 10));
 JLabel label = new JLabel("Enter a name");
 panel.add(label, BorderLayout.WEST);
 panel.add(cmdAdd, BorderLayout.EAST);
 panel.add(txtName);
 return panel;
}

private JPanel createCenter()
{
 lstNames = new JList(model);
 JPanel panel = new JPanel(new BorderLayout());
 panel.add(createLabel(), BorderLayout.NORTH);
 panel.add(createLabel(), BorderLayout.SOUTH);
 panel.add(new JScrollPane(lstNames));
 return panel;
}
```

```
 private JPanel createBottom()
{
 JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
 panel.add(cmdClr);
 return panel;
}

private void addListeners()
{
 cmdAdd.addActionListener(new AddAction());
 cmdClr.addActionListener(new ActionListener()
 {
  public void actionPerformed(ActionEvent e)
  {
   model.clear();
  }
 });
}

class AddAction implements ActionListener
{
 public void actionPerformed(ActionEvent e)
 {
```

```
    String name = txtName.getText().trim();
    if (name.length() > 0)
    {
     model.addElement(name);
     txtName.setText("");
     txtName.requestFocus();
    }
   }
  }
 }
```

It looks undeniably like much (there are 97 lines), and I will explain below what it all means. Of course one can not know that the code should be written as shown above, but if you look through the code, you can actually easily understand the meaning of most.

There are three *import* statements, which imports three packages with classes, two packages from the *AWT API* and one package from the *Swing API*. These *import* statements will be part of any GUI program and in all the classes that define windows.

The class is named *MainWindow*, and it inherits the class *JFrame*. It is a *Swing* class, and this is the class that defines all that is necessary to create a window and offers all the services that are needed to define how the window should behave.

A window is in principle just a simply rectangular area without any content, but it may contain components. Each component is defined by a class, and in this case the window must contain an *input field*, two *buttons* and a *list box*. They are defined as instance variables at the start of the class. An input field has the type *JTextField*, a button the type *JButton*, and a list box the type *JList*. They are all *Swing* classes. You should note that the list box is not created, when the variables are defined, but the other three components are. Finally, a last instance variable of the type *DefaultListModel* is defined and is explained below.

The class has a constructor that does the following:

- defines the text in the title bar
- defines the window size, which is the size of the window has when it opens
- defines that the program should close when the user clicks the cross in the title bar
- calls a method *addListeners()* that assigns functionality to the window's buttons
- calls a method *createWindow()* placing the components in the window
- define that the window must be displayed on the screen

The class has a method called *createLabel()*, which creates an empty label. A label is a component, and has the type *JLabel*, and it is a component that shows a text. In this case it is a blank text, but the important thing is that it has a size which is defined with *setPreferredSize()*.

Components are placed in a window using a layout manager, which is an object that determines the component's size and position. A window (that is a *JFrame*) has by default a *BorderLayout*. It is a layout manager that divides the window into 5 areas, and each area may contain a component. The five areas are

1. *NORTH*, and it has always the same width as the window while the height is determined by the height of the component
2. *SOUTH*, and it has always the same width as the window while the height is determined by the height of the component
3. *WEST*, the height is the height of the window, except for what is used for NORTH and SOUTH, while the width is determined by the width of the component
4. *EAST*, the height is the height of the window, except for what is used for NORTH and SOUTH, while the width is determined by the width of the component
5. *CENTER*, that is the rest of the window

If an area is empty – there is not a component in the area – it fills nothing and is collapsed. It is important to note that a *BorderLayout* can contain only 5 components.

If you now considers the method *createWindow()*, it starts using the method *createLabel()* to place a component in each of the four areas NORTH, SOUTH, WEST and EAST. The goal is to define a margin of 10 corresponding to the size of the components that are created with the method *createLabel()*. Next, the method creates a new component of the type *JPanel*. It is a component which in itself is not visible, but it has a layout manager (in this case a *BorderLayout*) and can contain other components. The component is called *panel*, and the last statement in the method *createWindow()* places the component CENTER in the window. Before it is filled with content using three methods, called respectively *createTop()*, *createCenter()* and *createBottom()*.

If I start with the last one, that creates a *JPanel* with a *FlowLayout* manager. It is a layout manager, where components are flowing horizontally. In this case, I have indicated that the components must be aligned to the right edge. After the panel is created, I adds one of the two buttons (the one to delete the list box). The result is that you gets a button located in the lower right corner of the window, but because of the two layout managers properties, the component will follow the window's bottom and right edge, when the window size is changed.

The method *createTop()* creates a component to the top, a *JPanel* with a label on the left, a button on the right and an input box in the center. The three components are placed by a *BorderLayout*, where the input field *txtName* is placed CENTER. As a CENTER area always fills it all, is the result of the layout, that the input field will always fill the part of the panel, which is not used by the other two components and thereby adjust to the window width. You should note that the panel is created as follows:

```
JPanel panel = new JPanel(new BorderLayout(10, 10));
```

Here you specify by the parameters for the *BorderLayout* class's constructor, how much free space should be between the components.

The method *createCenter()* creates the list box. When you do that you assign the list box to a data model – an object of the type *DefaultListModel* – which must contain the data that the list box should display. The method creates a *JPanel* with a *BorderLayout*, and assigns an empty label NORTH and SOUTH to get some space, respectively at top and bottom. When the list box is placed in the panel, it is encapsulated in a *JScrollPane*, which allows that you can scroll the content.

All the foregoing concerning only how to create and design the window. You should note that it is a stable design in the sense that the components follows the window size. What remains is to attach functionality to the two buttons. When the user clicks a button, it raises an event. Specifically, this means that other objects can register as listeners for this event by specifying a method with a specific signature. This method is called an *event handler*. When the button is clicked, it will check whether there are listeners, and if so, will it call these listeners. This sends messages to the listener objects that tells that the button is clicked, and the listeners can do something.

Above, there is an inner defined class, which is simply a class within another class. It's called *AddAction* and implements an interface called *ActionListener*. This interface defines only a single method called *actionPerformed()* and is an example of an event handler. It should be linked to the *Add* button, as happens width the method *addActionListener()*. The event handler retrieves the text from the input field, and if the text is not empty, the handler updates the model of the list box with the text, and the result is that the entered name appears in the list box. Next, the content of the input field is deleted, and finally the field is given focus, and the input field is ready to enter the next name. That the class *AddAction* implements the interface *ActionListener* means that an object of type *AddAction* can be used as a listener object of a button. It happens in the method *addListeners()* with the following statement:

```
cmdAdd.addActionListener(new AddAction());
```

cmdAdd is the name of the button, and the class *JButton* (which are of the button's type) have a method *addActionListener()*, which can register a listener object. The result is that when you click the *Add* button, then listener object's *actionPerformed()* method executes. You should note that this method can refer to the instance variables in the class *MainWindow*. This is possible because *AddAction* is an inner class.

There must now be done the same for the second button, but this time the event handler is very simple, since it alone must delete the contents of the model for the list box. I have therefore instead of writing a new listener class added a listener object created on the basis of an anonymous class:

```
cmdClr.addActionListener(new ActionListener()
{
 public void actionPerformed(ActionEvent e)
 {
  model.clear();
 }
});
```

At this point you just need to accept the syntax, but *cmdClear* is the name of a button, and I use its *addActionListener()* to associate a listener object. That is, as explained above, an object that implements the interface *ActionListener*, and such an object can be instantiated on the basis of an anonymous class with a syntax, as shown above. One can discuss the pros and cons with respect to anonymous classes, since they can be difficult to understand and read, but are they very simple classes – as in this case – it can be quite excellent.

Now the window is finished, but this is not in itself a program. There must be instantiated an object whose type is *MainWindow*, and it should be in the main program:

```
package helloswing;

public class HelloSwing
{
 public static void main(String[] args)
 {
  new MainWindow();
 }
}
```

Then the program is finished and can be run. As already mentioned, much is being written, although it is a very simple program, but the example exaggerating, partly because it is the same to happen every time, and secondly, parts of the code can be written simpler.

EXERCISE 1

In this exercise you have to make some changes and improvements to the program *HelloSwing*. Start by creating a copy and open it in NetBeans. In *MainWindow* is a method called *createLabel()*, which is used to add a margin outside the window's content and to create space between the components. This can be done in other way. Remove this method. You get 6 statements (which refers to this method), where NetBeans reports an error. Delete these 6 statements. If you then run the program, the result is the following:

To define margins you need to add an import statement:

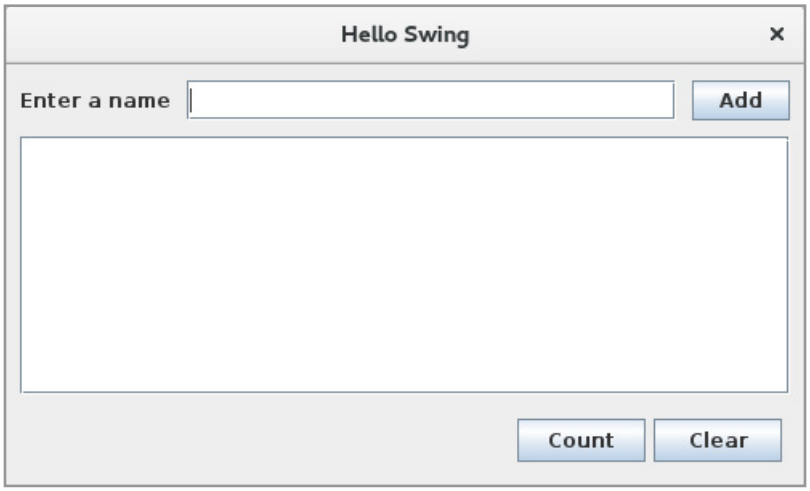
```
import javax.swing.border.*;
```

The method *createWindow()* must then be changed to the following:

```
private void createWindow()
{
 JPanel panel = new JPanel(new BorderLayout(0, 10));
 panel.setBorder(new EmptyBorder(10, 10, 10, 10));
 panel.add(createTop(), BorderLayout.NORTH);
 panel.add(createBottom(), BorderLayout.SOUTH);
 panel.add(createCenter());
 add(panel);
}
```

So would it all look right again.

You must then add a new button to the bottom of the window:



You do this as follows:

1. create a new instance variable to a button that you can call *cmdCount*
2. add the button (a component) to the window in the method *createBottom()*

Now the window should have another button. The two buttons at the bottom is probably not equal in size, but because they are laid out with a *FlowLayout*, you can define their size (in *createBottom()*) as follows:

```
cmdAnt.setPreferredSize(new Dimension(80, 27));
```

Assign the *Clear* button the same size.

What remains is to attach an action for the new button. When the button is clicked, the program must open the following message box that shows how many names are entered:

Start by adding another inner class that defines a listener object:

```
class CountAction implements ActionListener
{
 public void actionPerformed(ActionEvent e)
 {
  JOptionPane.showMessageDialog(MainWindow.this, "You have entered " +
    model.getSize() + " names", "Message", JOptionPane.INFORMATION_MESSAGE);
 }
}
```

Here is *showMessageDialog()* a static method in the class *JOptionPane*, which opens a message box. It has four parameters, wherein only the first two are required. The first tells who owns the message box, and the next is the text to be displayed. The third is the text in the title bar, while the remaining is telling the icon the message box will display.

After you have defined this class, you only need in the method *addListeners()* to make your application to listener for events from the button.

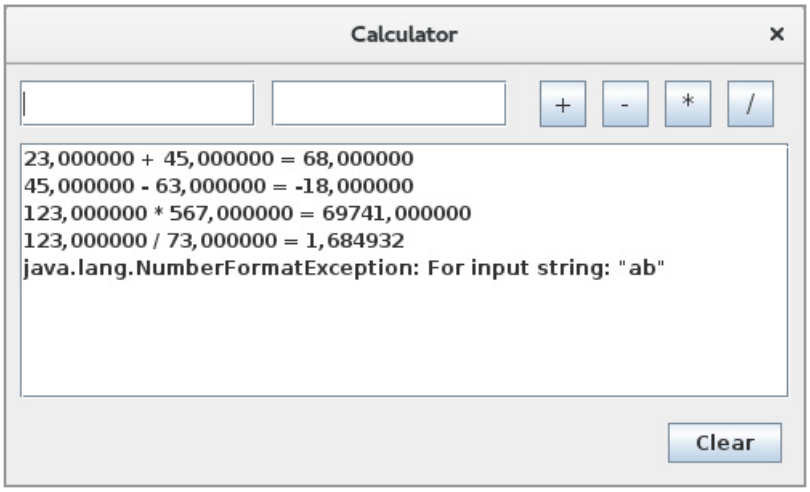
You must add one last change to the program. The *Clear* button deletes the content of the list box, but without a warning, which in practice can be unlucky. It would be better with a warning:

To add that, you must change the event handler for the *Clear* button so that its code is as follows:

```
if (JOptionPane.showConfirmDialog(MainWindow.this,
  "Are you sure you want to delete the list?", "Warning",
  JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE)
   == JOptionPane.YES_OPTION) model.clear();
```

EXERCISE 2

Create a new NetBeans project, that you can call *Calculator*. You should write a program that opens a window, as shown below:



The window has two input fields where you must enter numbers. Furthermore, there are four buttons – one for each of the four arithmetical operations. When you click a button, the program should insert a line in the list box, which shows the results of that calculation. If there is an error, the program simply inserts an error message. The *Clear* button should work in the same way as in the first program.

When the window size is changed, all buttons must follow the window's right edge, while the two input boxes will use the remaining space equally.

It is clear that the program is similar to the *HelloSwing*, but I would suggest that you start from scratch and follow the guidelines below. You can of course use *HelloSwing* to see how the individual statements should be written.

1) Add a new class named *MainWindow* to the project. Start by typing the following code:

```
package calculator;

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

public class MainWindow extends JFrame
{
 public MainWindow()
 {
```

```
   setTitle("Calculator");
   setSize(500, 300);
   setDefaultCloseOperation(EXIT_ON_CLOSE);
   setVisible(true);
  }
}
```

You must then edit the *main* class to open the window:

```
package calculator;

public class RegneMaskine
{
 public static void main(String[] args)
 {
  new MainWindow();
 }
}
```

Test the program. This should open a window, and it is in a way a minimal GUI program, but nevertheless a program with a window with a title bar that can be moved around on the screen and resized.

2) Add instance variables to the program. There is a need for 9 variables:

- two input fields that you should call *txtNum1* and *txtNum2*
- five buttons with the names *cmdAdd*, *cmdSub*, *cmdMul*, *cmdDiv* and *cmdClr*
- a list box with the name *lstRes* (you must not create an object, but just define a variable)
- a model for the list box, and the name must be *model*

Compile the program and run it. There are no visible differences, and this is just to ensure that you do not have any syntax errors.

3) You must next add a method named *createCenter()*. It's basically the same method as in exercise 1, and the task is to create the list box and place it in a *JScollPane*. The only difference is that the list box is now called something else. Next, add the method c*reateWindow()*, which creates the window's components:

```
private void createWindow()
{
 JPanel panel = new JPanel(new BorderLayout(0, 10));
 panel.setBorder(new EmptyBorder(10, 10, 10, 10));
 panel.add(createCenter());
 add(panel);
}
```

Keep in mind that it is necessary to add an *import* statement

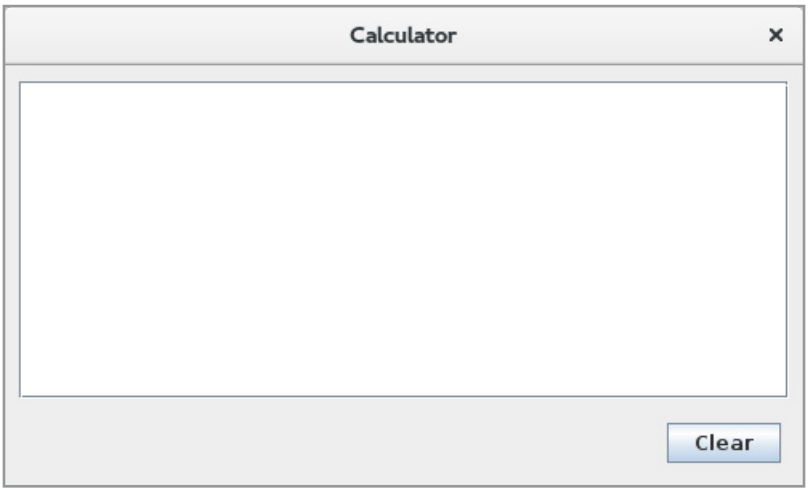
```
import javax.swing.border.*;
```

You should also call the method *createWindow()* from the constructor. Run the program, the result should now be that you get a window with a list box.

4) Add a method *createBottom()*. It must be the same method as in the program *HelloSwing* which adds the *Clear* button to the window. You must then add a statement to *createWindow()*, so the panel that *createBottom()* creates are added to the bottom of the window:

```
private void createWindow()
{
 JPanel panel = new JPanel(new BorderLayout(0, 10));
 panel.setBorder(new EmptyBorder(10, 10, 10, 10));
 panel.add(createBottom(), BorderLayout.SOUTH);
 panel.add(createCenter());
 add(panel);
}
```

If you then run the program, the result should be the following:



5) You now need to write the code to the top panel, which is a little more difficult. Start by adding the following method:

```
private void initButton(JButton cmd)
{
 cmd.setPreferredSize(new Dimension(29, 29));
 cmd.setFont(new Font("Liberation Sherif", Font.PLAIN, 16));
 cmd.setMargin(new Insets(0, 0, 0, 0));
}
```

It has a button as a parameter, and assign the button a size of 29 × 29, defines the font, the button should apply and remove an internal margin, as a button has by default.

You can then add the following method:

```
private JPanel createRight()
{
 JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 10, 0));
 initButton(cmdAdd);
 initButton(cmdSub);
 initButton(cmdMul);
 initButton(cmdDiv);
 panel.add(cmdAdd);
 panel.add(cmdSub);
 panel.add(cmdMul);
 panel.add(cmdDiv);
 return panel;
}
```

The method calls *initButton()* for each of the four calculation buttons, and then places the four buttons in a *JPanel* with *FlowLayout*.

As the next step you need to create a panel to the two input fields:

```
private JPanel createLeft()
{
 JPanel panel = new JPanel(new GridLayout(1, 2, 10, 0));
 panel.add(txtNum1);
 panel.add(txtNum2);
 return panel;
}
```

Here I use a *GridLayout*. It is a layout manager that divides a panel in a number of rows and columns. This divides a panel in a number of cells that all have the same size. In this case, there is one row of two columns, and the result is that the panel consists of two cells, which will always be of the same size. The two input fields are added to the panel, and each field will automatically fill the cell that it is located in. The constructor for *GridLayout* object has two additional parameters that indicates how much space there should be between the cells horizontally and vertically.

Finally, add the following method:

```
private JPanel createTop()
{
 JPanel panel = new JPanel(new BorderLayout(10, 10));
 panel.add(createRight(), BorderLayout.EAST);
 panel.add(createLeft());
 return panel;
}
```

It returns a panel with a *BorderLayout*. For this panel is added the panel with the 4 calculation buttons so it sits to the right, and the panel with the two input fields so that it uses the remaining space. If you then apply the method *createTop()* in *createWindow()*, you can add the top panel and the design of the window is now finished.

6) What remains is to attach event handlers to the 5 buttons. Start by adding the following method:

```
private void calculate(char ch)
{
 try
 {
  double tal1 = Double.parseDouble(txtTal1.getText());
  double tal2 = Double.parseDouble(txtTal2.getText());
  double res = 0;
  switch (ch)
  {
   case '+': res = tal1 + tal2; break;
   case '-': res = tal1 - tal2; break;
   case '*': res = tal1 * tal2; break;
   case '/': res = tal1 / tal2; break;
  }
  model.addElement(String.format("%f %s %f = %f", tal1, "" + ch, tal2, res));
 }
 catch (Exception ex)
 {
  model.addElement(ex.toString());
 }
```

```
  txtTal1.setText("");
  txtTal2.setText("");
  txtTal1.requestFocus();
 }
```

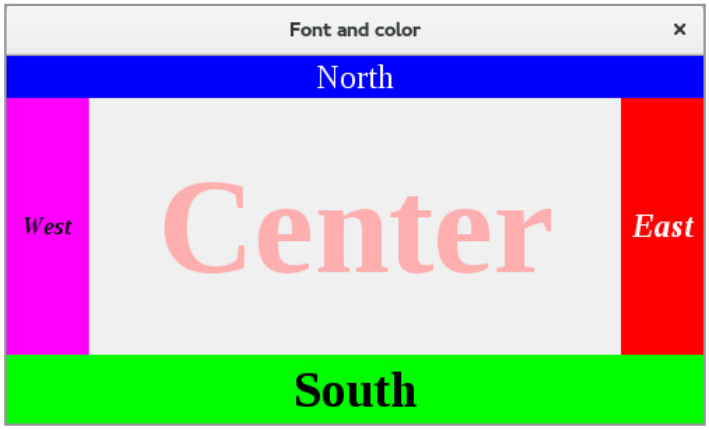
It has a parameter that tells which calculation to be carried out. The method performs the calculation and inserts a line in the model for the list box.

This method must be called from the buttons event handlers. This should be done by the same way as in the program *HelloSwing* to add a method *addListeners()*, which adds event handlers for the buttons, but this time for all 5 buttons, with the help of anonymous classes. Remember to call the methods *addListeners()* from the constructor in *MainWndow*.

Then the program should be finished and could be tested.

3 FONTS AND COLORS

Most of the components that can be inserted into a window/panel displays text and you can define the font, which they must use. Similarly, you can define the color of the text uses, and finally you can define the background color. The program *TextColor* opens the following window:



It is a very simple program that in fact do nothing. It has the same architecture as the previous programs, in which the main class opens a window, which in this case is defined as follows:

```
package textcolor;

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

public class MainWindow extends JFrame
{
 public MainWindow()
 {
  setTitle("Font og farver");
  setSize(500, 300);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  setLocationRelativeTo(null);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  add(createLabel("North", Color.blue, Color.white,
   new Font("Liberation Serif", Font.PLAIN, 24), 0, 30),
   BorderLayout.NORTH);
```

```
 add(createLabel("South", Color.green, Color.black,
  new Font("Liberation Serif", Font.BOLD, 36), 0, 50),
  BorderLayout.SOUTH);
 add(createLabel("West", Color.magenta, Color.black,
  new Font("Liberation Serif", Font.ITALIC, 18), 60, 0),
  BorderLayout.WEST);
 add(createLabel("East", Color.red, Color.white,
  new Font("Liberation Serif", Font.BOLD | Font.ITALIC, 24), 60, 0),
  BorderLayout.EAST);
 add(createLabel("Center", new Color(240, 240, 240), Color.pink,
  new Font("Liberation Serif", Font.BOLD, 96), 0, 0));
}

private JLabel createLabel(String text, Color bg, Color fg, Font font,
 int width, int height)
{
 JLabel label = new JLabel(text);
 label.setOpaque(true);
 label.setBackground(bg);
 label.setForeground(fg);
 label.setFont(font);
 label.setHorizontalAlignment(JLabel.CENTER);
 label.setPreferredSize(new Dimension(width, height));
```

```
    return label;
  }
}
```

I'll start with the method *createLabel()*, which creates and returns a component of the type *JLabel*. It is a component that shows a text. The method has 6 parameters:

- The first is the text.
- The next is the background color, which is an object of the type *Color*.
- The third is the foreground color and thus the text color. It is likewise an object of type *Color*.
- The fourth parameter is the font, which is an object of type *Font*.
- The last two parameters are respectively the width and height og the component.

The method's statements are easy enough to figure out, but you must again note how to define the size of a component with *setPreferredSize()*. A *JLabel* is shown with transparent background by default, and therefore can not have a background color unless you say that the background should not be transparent. This is done with *setOpaQue()*.

A frame window is by default born with a *BorderLayout*, and you can immediately add 5 components, as happens in the method c*reateWindow()*.

A *BorderLayout* divides as mentioned a window/ panel in 5 areas, called respectively *NORTH*, *EAST*, *SOUTH*, *WEST* and *CENTER*, with the latter as default. Each region may contain a single component, but may also be empty. If so, the area fills nothing on the screen. If an area contains a component, the following applies concerning the size:

- *NORTH*: The width is the width of the panel, and the height is determined by the component's height defined by its preferred size. The width of the component in terms of its preferred size is ignored.
- *SOUTH*: The width is the width of the panel, and the height is determined by the component's height defined by its preferred size. The width of the component in terms of its preferred size is ignored.
- *WEST*: The height is the height of the panel minus the height that is used to NORTH and SOUTH, and the width is determined by the component's width defined by its preferred size. The component's height in terms of its preferred size is ignored.
- *EAST*: The height is the height of the panel minus the height that is used to NORTH and SOUTH, and the width is determined by the component's width defined by its preferred size. The component's height in terms of its preferred size is ignored.
- *CENTER*: The width is the width of the panel minus the width used by the WEST and EAST, and the height is the height of the panel minus the height that is used to NORTH and SOUTH. The component's preferred size is ignored.

In this case, there is placed 5 *JLabel* components in the panel, where a component is created by the method *createLabel()*. Here you should note, how the size is defined, and that the values for width and height where they are ignored are set to 0. It is not necessary and is only done to clarify that the values is not used.

Otherwise note how you define colors and fonts. The *Color* class has a number of constants for frequently used colors, and most of the colors in this example are indicated by means of these colors. Colors are defined using three intensities of red, green and blue, and wee talk about this color coding as *RGB colors*. Each intensity has a value between 0 and 255, and the number of possible colors is

$$256^3 = 16777216$$

The following table shows some examples of color encodings

R	G	B	Farve
0	0	0	Black
255	255	255	White
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	0	Yellow
128	128	128	Gray

Here you need to specifically note the last encoding, where the same value for all intensities provides a grayscale. In Java you can define a color as an object to the type *Color* with color intensities to the constructor. It is used in the label that sits center in the window where there is defined the following color:

```
new Color(240, 240, 240)
```

This means that the component will have a faint gray background.

You define a font from the font name, whether it should be *normal*, *italic*, *bold* or possibly a combination of the last two. Finally, you specify the font size.

You should note that this time the program do not define any event handling. It is of course because it is a simple demo program, and in practice GUI programs always have an event handling.

Finally, in the constructor you should note the statement:

```
setLocationRelativeTo(null);
```

It is a statement that ensures that the window opens in the middle of the screen.

EXERCISE 3

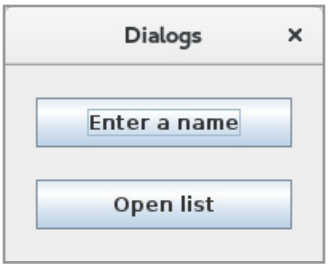
Make a copy of the program *TextColor*. You should modify the 5 *JLabel* components so

1. North must use a *Liberation Sans* font that is bold and 16 point.
2. South must have a dark green text color.
3. West must have a width on 100 and a font on 48 point.
4. East must have a standard yellow text color.
5. Center must have at light gray background and a gray text color.

4 DIALOG BOXES

In the above examples are each time been a single window, but in practice, a GUI program have several or many windows, and in this chapter I will show how you from a window can open another window. The main window is called a *frame window*, and is derived from the class *JFrame*. Another window, is usually a *dialog box*, which is a class derived from *JDialog*. I will show a program that has three windows:

Here is *MainView* a usual frame window, which in this case must have two buttons, which are used to open each of the other two windows:

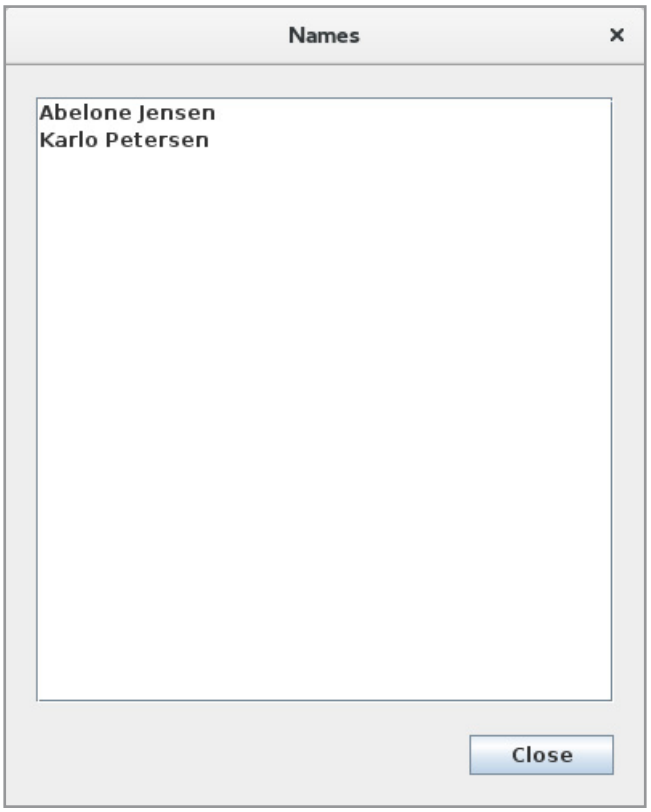


The window to the right is an example of a dialog box where you should enter a name (first name and last name):

It must be a *modal dialog*, and it means that when the dialog box is open, the other windows of the application does not have focus, and you can not interact with the other windows before this dialog box is closed.

The third window must also be a dialog and has to show a list box with the names that have been entered (see below), but it must be a *modless dialog* box, which means that other windows can get focus while the dialog box is open.

Dialog boxes are defined (almost) the same way as frame windows, and as such there is nothing new in the program, but the three windows has to communicate, so that if you enter a name in the dialog box above, then dialog box with the list box must be updated with the entered name.



The program should represent a name as an object, and for that I have used the class *Name* from the program *Comparison* in the book Java 1. The class is simple and completely unchanged and is therefore not discussed further here.

I'll start with the dialog box for entering a name:

```
package dialogs;

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

```
public class EnterView extends JDialog
{
 private DefaultListModel model;
 private JTextField txtFirstname = new JTextField();
 private JTextField txtLastname = new JTextField();

 public EnterView(DefaultListModel model)
 {
  super(null, "Enter a name", Dialog.ModalityType.APPLICATION_MODAL);
  this.model = model;
  setSize(400, 200);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createView();
  setVisible(true);
 }

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

```
 panel.add(createBottom());
 add(panel);
}

private JPanel createTop()
{
 JPanel panel = new JPanel(new GridLayout(2, 1, 0 ,10));
 panel.add(createLine("First name", txtFirstname));
 panel.add(createLine("Last name", txtLastname));
 return panel;
}

private JPanel createBottom()
{
 JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT, 0, 0));
 panel.add(createButton("OK", 90, 25, this::ok));
 panel.add(createSpace());
 panel.add(createButton("Close", 90, 25, this::close));
 return panel;
}

private JPanel createLine(String text, JTextField field)
{
 JPanel panel = new JPanel(new BorderLayout());
 JLabel label = new JLabel(text);
 label.setPreferredSize(new Dimension(90, 22));
 panel.add(label, BorderLayout.WEST);
 panel.add(field);
 return panel;
}

private JButton createButton(String text, int width, int height,
  ActionListener listener)
{
 JButton cmd = new JButton(text);
 cmd.addActionListener(listener);
 cmd.setPreferredSize(new Dimension(width, height));
 return cmd;
}

private JLabel createSpace()
{
 JLabel label = new JLabel();
 label.setPreferredSize(new Dimension(10, 20));
 return label;
}
```

```
 private void clear()
 {
  txtFirstname.setText("");
  txtLastname.setText("");
  txtFirstname.requestFocus();
 }
 private void ok(ActionEvent e)
 {
  String firstname = txtFirstname.getText().trim();
  String lastname = txtLastname.getText().trim();
  if (firstname.length() > 0 && lastname.length() > 0)
  {
   model.addElement(new Name(firstname, lastname));
   clear();
  }
  else JOptionPane.showMessageDialog(this,
    "You must enter both first name and last name",
    "Error", JOptionPane.WARNING_MESSAGE);
 }

 private void close(ActionEvent e)
 {
  dispose();
 }
}
```

The class is called *EnterView*, and the first thing to note is that the class inherits *JDialog* instead of *JFrame*. It is telling that it is a dialog box. There are three instance variables, to the two input fields and a model for a list box. It may seems strange, since the dialog box does not contain a list box, but it must be used to keep the *Name* objects to be displayed in the second dialog. The model is sent as a parameter in the constructor. Aside from that, there is not much new in the constructor, but you must notice the first line, which is the place where one says that it is a modal dialog box by sending a parameter to the constructor of *JDialog*. Finally, note the statement *setDefaultCloseOperation()*, which uses a different parameter, indicating that if you click the cross in the title bar, then the dialog box must be closed, and the program should not terminate.

Then there is the method *createView()*, which initializes the window components. As in the other examples, it is spread out in several methods and is certainly complex enough, but conversely, there is not much new compared to what was previously mentioned. However, you should datailed study how the user interface is defined and, in particular, observe what happens to the window when you run the program and changes the window size.

However, there is one place where there are added something new. The dialog box has two buttons, and they must have attached event handlers. In the preceding examples, I have attached event handlers for buttons at either defining inner classes or by defining anonymous classes. Inner classes requires to be written much code and anonymous classes results in code that can be hard to read. However, there are alternatives and in this example, I have used a simple and readable notation. If, as an example, the event handler for the *Close* button, there shall be no more statements that a statement that close the dialog box. You do this with the statement *dispose()*, and I added the following method:

```
private void close(ActionEvent e)
{
 dispose();
}
```

Here you must notice the parameter, which is the one that says that this method can be used as an event handler. It must be attached to the button, which is done in the following way in the method *createBottom()*:

```
this::close
```

Simpler it can hardly be. What exactly happens, I will not explain here, but in principle what happens is that the compiler automatically generates the code that I usually write. A more detailed explanation follows in the book Java 4, but the notation enhances readability so much that I would recommend that you start taking it as is and use it already.

There is a similar event handler for the second button. It is obviously more extensive, but shortly the following happens. The entered values are copied to variables from the two input fields. If there is typed something for both first and last name a *Name* object is created, and the model is updated. If not a message box displays an error message.

Then there is the other dialog box, showing the entered names:

```
package dialogs;

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

public class ShowView extends JDialog
{
 private DefaultListModel model;
 private CloseListener listener;

 public ShowView(DefaultListModel model, CloseListener listener)
 {
  super(null, "Names", Dialog.ModalityType.MODELESS);
  this.model = model;
  this.listener = listener;
  setSize(400, 500);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  addWindowListener(new ClosingHandler());
  createView();
  setVisible(true);
 }

 private void createView()
 {
  setLayout(new BorderLayout());
  JPanel panel = new JPanel(new BorderLayout(0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(new JScrollPane(new JList(model)));
  panel.add(createBottom(), BorderLayout.SOUTH);
  add(panel);
 }
```

```
  private JPanel createBottom()
  {
   JPanel panel = new JPanel(new FlowLayout(FlowLayout.RIGHT, 0, 0));
   panel.add(createButton("Close", 90, 25, this::close));
   return panel;
  }

  private JButton createButton(String text, int width, int height,
   ActionListener listener)
  {
   JButton cmd = new JButton(text);
   cmd.addActionListener(listener);
   cmd.setPreferredSize(new Dimension(width, height));
   return cmd;
  }
  private void close(ActionEvent e)
  {
   listener.dialogClosed();
   dispose();
  }

  class ClosingHandler extends WindowAdapter
  {
   public void windowClosing(WindowEvent e)
   {
    listener.dialogClosed();
   }
  }
}
```

It is similar a class that inherits *JDialog*. There are two instance variables, the first being the model and thus the objects that must be displayed in the list box, while the other has the type *CloseListener*. I explains the type in a moment, but values for both variables are sent to the constructor. Otherwise, there is only one thing to note, that the first line defines that it is a *modeless dialog box*.

The design of the window is this time simple since it is merely to place a list box and a button. The button is assigned an event handler with the same syntax as mentioned above, but there is one complication back. The dialog is opened by clicking a button in the main window, and because it is a modeless dialog box, you can click the button again and open the dialog several times. I are not interested in that, but conversely, it must be possible to opened the dialog again if it is closed. It is therefore necessary to send a message back to the main window when the dialog closes. As mentioned, the constructor has a parameter named *listener* that has the type *CloseListener*. It's a simple interface defined in the same file as the class *MainView*:

```
interface CloseListener
{
 public void dialogClosed();
}
```

The interface does nothing more than define a single method, but when the dialog box through the constructor get an object of this type the dialog box know that object has such a method, and can thus sent a notfikation when the dialog box is closed by clicking the Close button:

```
private void close(ActionEvent e)
{
 listener.dialogClosed();
 dispose();
}
```

Now the dialog box, in principle, also could be closed by clicking on the cross in the titleline, and in this case, sending a similar notification. It is therefore necessary to capture the event that occurs when clicking on the cross. It is a *WindowEvent*, and it occurs in several contexts, but above I have shown how to catch it with the class *CloseHandler*. Note that the handler must be associated with the dialog box, which happens in the constructor:

```
 addWindowListener(new ClosingHandler());
```

Back is the main window where there is not much to explain as it is just an ordinary window with two buttons:

```
package dialogs;

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

public class MainView extends JFrame implements CloseListener
{
 private DefaultListModel model = new DefaultListModel();
 private JButton cmdShow;

 public MainView()
 {
  setTitle("Dialogs");
  setSize(200, 160);
  setResizable(false);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  JPanel panel = new JPanel(new GridLayout(2, 1, 0, 20));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(createButton("Enter a name", this::openEnter));
  panel.add(cmdShow = createButton("Open list", this::openShow));
  add(panel);
 }

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

 private void openEnter(ActionEvent e)
 {
```

```
   new EnterView(model);
  }

  private void openShow(ActionEvent e)
  {
   cmdShow.setEnabled(false);
   new ShowView(model, this);
  }

  public void dialogClosed()
  {
   cmdShow.setEnabled(true);
  }
}
```

There are two instance variables, respectively the model for the list box and a button. The model is sent to both dialog boxes and the button is a reference to the button that opens the dialog box with the list. You should note that the class implements the interface *CloseListener* and must therefore define the method *dialogClosed()*. When the dialog box for the list box is opened, two things happen. First disables the button so the dialog box can not be opened again, and then the dialog box opens by sending parameters as the model for the list box and a reference to the window itself, but the window is special a *CloseListener*, and therefore can be used as a current parameter. The result is that when the dialog box closes, it calls the method *dialogClosed()*, which enables the button, and the dialog box can be opened again. When you test the program, especially noting that if you create a name then the list box is updated automatically without you must do anything. The technique for that is hided in the class *DefaultListModel*.

EXERCISE 4

In this exercise you have to make some improvements to the program above. Start by creating a copy and open the copy in NetBeans.

You should start to add another button to the dialog *ShowView* where the button should delete the contents of the list box. This change should not lead to major challenges.

Next, it must be such that if you double-click on a name in the list box, then the dialog box *EnterView* should opens initialized with the name that is clicked on, and you should then be able to edit the name. It is immediately a little more complicated, but you can go as follows.

Modify the class *Name*, so it has *set* methods for both variables.

Add a parameter *index* to the constructor in the class *EnterView*:

```
public EnterView(DefaultListModel model, int index)
```

and store the value in an instance variable. If the value is negative, it shal indicate that the dialog box is used to create a *Name*, and otherwise the *index* is interpreted as the index of the object in the model to be edited. Note that the change means that it is necessary to change the class *MainView* where you must add a parameter (value -1) when the dialog box *EnterView* opens.

In the class *ShowView* add the following inner class:

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

```
    JOptionPane.showMessageDialog(null, model.getElementAt(n));
   }
  }
 }
```

It defines an event handler concerning mouse clicks, and must be linked to the list box, which you can do by changing the method *createView()*:

```
JList list = new JList(model);
list.addMouseListener(new MouseHandler());
panel.add(new JScrollPane(list));
```

Before continuing, you should test the program. Create a few names and open the dialog with the list box. Try double-clicking on a name and see if you get a message box that displays the name. Doing so is everything regarding double-click into place.

The above event handler for the mouse should now be changed, so it does not open a message box, but instead opens *EnterView*, where the last parameter now is the index of the name that is double clicked. *EnterView* should show the name, and to do this, change in the method *createTop()* so that it initializes the input fields with the name that is double-clicked. Remember that the variable *index* is the index of the *Name* object relative to the model. When you then click *OK*, do not create a new *Name* object, but the object that is clicked must instead be updated. It is therefore necessary to modify the event handler *ok()* so that it (using the variable *index*) distinguishes between the two cases, where to create a new object, and where an existing object is edited.

Once you've made these changes, you will find that the list box does not seem to be updated, but close the dialog box *ShowView* and open it again, and you will now see that the changes are visible, so the model has therefore been updated. To solve this problem, you must add the following class to file with the main window:

```
class EnhancedListModel extends DefaultListModel
{
 public void update(int index)
 {
  fireContentsChanged(this, index, index);
 }
}
```

It is a class that extends *DefaultListModel* with a new method. You must also change the definition of the model in the *MainView*:

```
private DefaultListModel model = new EnhancedListModel();
```

The new method on the model must be called in the event handler *ok()* in the class *EnterView* after the object has been updated:

```
((EnhancedListModel)model).update(index);
```

Note the typecast. After that the list box will be updated correctly.

You must add one last improvement. The dialog box *EnterView* must have an additional button to be used to delete the object being edited, but the button must only be enabled if the dialog box is used to edit a *Name*.

5 MORE COMPONENTS

In this section I will show an example that, in principle is similar to the first examples, where the program simply consists of a single window, but it is a somewhat more complex example:

1. there are several components both in terms of the number of componentes and the type of components
2. it is an example of a complex layout
3. there is a comprehensive event management

If you run the program it opens the following window:

The window should illustrate a sign with a text. You can then using the window's other components to adjust how the sign should be displayed:

- the font used to draw the text
- how the text is aligned (left, right or center)
- the color used to draw the text
- the background color to the sign

Compared to the first examples, this example use more kinds of components:

- *JLabel*, which is a component showing a text and was used in the previous examples
- *JTextField*, which is an input field, and is also used in the previous examples
- *JComboBox*, which is component with a list of objects, and you can then select one of these objects
- *JCheckBox*, which is a simple component, where you can select a property
- *JRadioButton*, where several components are organized in a group, so you can choose one of them
- *JSlider*, which is a component, where you by means of the "shoots" can select a value within a range

If on top of that you adds the components *JButton* and *JList* that is used in the previous examples, but are not used in this example, you have actually met most of the basic *Swing* components, and in practice you will get far with these components.

Then there is the program code, which is extensive with nearly 350 lines. Rather than show alle the code together, I will show parts in connection with the description of the individual concepts. It is recommended that you open the full code while you read the following.

I'll start with the layout, which this time is quite complex. You should start to run the application and notice how some of the components change their sizes as the window size changes. The components size and location are determined by the layout manager being used, and it is the subject of the next chapter, but for now I have mentioned three layout managers

- *BorderLayout*
- *FlowLayout*
- *GridLayout*

and they are all used in this example. By using nested layout managers, that is having panels inside each other with their own layout managers, you can actually by using the above three layout managers design a rather complicated layout.

The starting point is similar to the previous examples and starts with the following method:

```
private void createWindow()
{
 JPanel panel = new JPanel(new BorderLayout());
 panel.setBorder(new EmptyBorder(10, 10, 10, 10));
 panel.add(createTop(), BorderLayout.NORTH);
 panel.add(createCenter());
 add(panel);
}
```

that by means of a *BorderLayout* with a margin around dividing the window into two parts, where the top has a panel with all of the adjustment components, while at the bottom has the sign, and it will use the part of the window, which is not used for the top panel.

I'll start with the bottom panel, which is the simplest. The sign is represented by a *JLabel* defined as an instance variable:

```
private JLabel lblText = new JLabel("Det er teksten");
```

The bottom panel is created by the following method:

```
private JPanel createCenter()
{
 lblText.setOpaque(true);
 lblText.setBackground(Color.white);
 JPanel panel = new JPanel(new BorderLayout());
 panel.setBorder(new EmptyBorder(20, 0, 20, 0));
 JPanel sign = new JPanel(new BorderLayout());
 sign.setBorder(new LineBorder(Color.black));
 JPanel inner = new JPanel(new BorderLayout());
 inner.setBorder(new LineBorder(Color.white, 5));
 inner.add(createDots(), BorderLayout.NORTH);
 inner.add(createDots(), BorderLayout.SOUTH);
 inner.add(createMargin(), BorderLayout.WEST);
 inner.add(createMargin(), BorderLayout.EAST);
 inner.add(lblText);
 sign.add(inner);
 panel.add(sign);
 return panel;
}
```

The first thing that happens is that the panel's background is set to white. The panel has a *BorderLayout* with a nested *BorderLayout* with another nested *BorderLayout*. The goal of all this is to define some margins. The outer panel has a margin of 20 on the top and bottom. The middle panel (called *sign*) is intended to define a thin black frame. This is done by assigning a *LineBorder*, which by default is a line of 1 pixel. Then there is the inner panel called *inner*. It has a white margin of 5 pixels (to create distance to the black frame. The panel places in the corners some small squares (to symbolize screw holes) and finally the label component is placed center. To build it all the following methods are used:

```
private JLabel createMargin()
{
 JLabel label = new JLabel();
 label.setPreferredSize(new Dimension(5, 5));
 label.setOpaque(true);
 label.setBackground(Color.white);
 return label;
}
```

```
private JPanel createDots()
{
 JPanel panel = new JPanel(new BorderLayout());
 panel.add(createDot(), BorderLayout.WEST);
 panel.add(createDot(), BorderLayout.EAST);
 panel.add(createMargin());
 return panel;
}

private JLabel createDot()
{
 JLabel label = new JLabel();
 label.setPreferredSize(new Dimension(5, 5));
 label.setOpaque(true);
 label.setBackground(Color.black);
 return label;
}
```

The first is used for a *BorderLayout* to define a white background. The bottom creating the black square (screw hole), while the middle creates a panel with two screw holes and filled with a white background between the holes.

Then there is the top panel, which is somewhat more complex. It actually consists of four lines with tools that are laid out with a *GridLayout*:

```
private JPanel createTop()
{
 JPanel panel = new JPanel(new GridLayout(4, 1));
 panel.add(createFont());
 panel.add(createFg());
 panel.add(createBg());
 panel.add(createText());
 return panel;
}
```

I'll start with the bottom line, which defines the input field to the text of the sign. The field is defined as an instance variable:

```
private JTextField txtText = new JTextField();
```

and the panel with the input field is created as follows:

```
private JPanel createText()
{
 txtText.setText(lblText.getText());
 JPanel panel = new JPanel(new BorderLayout(10, 0));
 panel.setBorder(new EmptyBorder(5, 0, 5, 0));
 JLabel label = new JLabel("Tekst");
 panel.add(new JLabel("Tekst"), BorderLayout.WEST);
 panel.add(txtText);
 return panel;
}
```

The input field is initialized with the content of the label component to the sign, but otherwise this is primarily a *BorderLayout* with a label and a input field, added center. In this way one obtains that the field follows the window size.

Then there is the line to the background color, which consists of a *JLabel*, and three pairs consisting of a *JLabel*, and a *JSlider*. The following method creates such a couple of components:

```
private JPanel createColor(String text, int width, JSlider slider,
   int min, int max, int value)
{
 JPanel panel = new JPanel(new BorderLayout());
 JLabel label = new JLabel(" " + text);
 label.setPreferredSize(new Dimension(width, 30));
 panel.add(label, BorderLayout.WEST);
 slider.setMinimum(min);
 slider.setMaximum(max);
 slider.setValue(value);
 slider.setMajorTickSpacing(50);
 slider.setMinorTickSpacing(10);
 slider.setPaintTicks(true);
 panel.add(slider);
 return panel;
}
```

The first parameter specifies the text of the label component and the next the width of the component. The other four parameters are for the slider component and the values it must be initialized with. A *JSlider* is a component that represents an interval of integers defined with *setMinimum()* and *setMaximum()*. The component has at any time a value within this range, and it can be defined with *setValue()*. The two methods *setMajorTickSpacing()* and *setMinorTickSpacing()* are used to define a visual partition of the component. The method *createColor()* returns a *JPanel* with a *BorderLayout* containing the label component and the slider. The result is that the sliders size will follow the width of the panel.

This method is used by the following method to define three pairs of label and slider:

```
private JPanel createColors(JSlider sldRed,
JSlider sldGreen, JSlider sldBlue,
  int red, int green, int blue)
{
 JPanel panel = new JPanel(new GridLayout(1, 3));
 panel.add(createColor("Rød", 40, sldRed, 0, 255, red));
 panel.add(createColor("Grøn", 50, sldGreen, 0, 255, green));
 panel.add(createColor("Blå", 40, sldBlue, 0, 255, blue));
 return panel;
}
```

They are laid out with a *GridLayout* with 1 row and 3 columns, and each pair will always fill the same. The *JSlider* components are defined as instance variables:

```
private JSlider sldRbg = new JSlider(JSlider.HORIZONTAL);
private JSlider sldGbg = new JSlider(JSlider.HORIZONTAL);
private JSlider sldBbg = new JSlider(JSlider.HORIZONTAL);
```

After that is the line for adjusting the background color defined as follows:

```
private JPanel createBg()
{
 JPanel panel = new JPanel(new BorderLayout());
 panel.setBorder(new EmptyBorder(5, 0, 5, 0));
 JLabel label = new JLabel("Baggrungsfarve");
 label.setPreferredSize(new Dimension(120, 30));
 panel.add(label, BorderLayout.WEST);
 panel.add(createColors(sldRbg, sldGbg, sldBbg, 255, 255, 255));
 return panel;
}
```

The goal of all this is to ensure that the slider components are resized equally in the line when the window size is changed.

The line to define the text color is designed in exactly the same way, and I will not show it here, but the three slider components are defined as instance variables:

```
private JSlider sldRfg = new JSlider(JSlider.HORIZONTAL);
private JSlider sldGfg = new JSlider(JSlider.HORIZONTAL);
private JSlider sldBfg = new JSlider(JSlider.HORIZONTAL);
```

Then there is the top toolbar, which is the most complex. It uses the following components, all of which are defined as instance variables:

```
private JComboBox lstFont;
private JTextField txtSize = new JTextField();
private JCheckBox chkBold = new JCheckBox("Bold");
private JCheckBox chkItalic = new JCheckBox("Italic");
private JRadioButton cmdLeft = new JRadioButton("Left", true);
private JRadioButton cmdCenter = new JRadioButton("Center");
private JRadioButton cmdRight = new JRadioButton("Right");
```

The following method creates a *JPanel* with a *BorderLayout* containing a *JLabel* and a *JComboBox*, and the goal is that the combo box must follow the width of the window:

```
private JPanel createFonts()
{
 JPanel panel = new JPanel(new BorderLayout(10, 0));
 JLabel label = new JLabel("Fonte");
 panel.add(new JLabel("Fonte"), BorderLayout.WEST);
 DefaultComboBoxModel model = new DefaultComboBoxModel();
 String fonts[] =
  GraphicsEnvironment.getLocalGraphicsEnvironment().
getAvailableFontFamilyNames();
 for (int i = 0; i < fonts.length; ++i) model.addElement(fonts[i]);
 lstFont = new JComboBox(model);
 Font df = label.getFont();
 for (int n = 0; n < fonts.length; ++n)
  if (df.getFamily().equals(fonts[n]))
  {
   lstFont.setSelectedIndex(n);
   break;
  }
 panel.add(lstFont);
 chkBold.setSelected(df.isBold());
 chkItalic.setSelected(df.isItalic());
 txtSize.setText("" + (size = df.getSize()));
 return panel;
}
```

After the label component is added, the method creates a data model for the combo box. Next, an array of the names of all fonts available on the current machine is created, and this array is used to initialize the model. After the combo box is created the label component is used to determine the current default font for a *JLabel*, and it is used to determine which element in the combo box, that should be selected. Finally, the default font is used to initialize the two check boxes and the input field to the font size.

With this method in place, the toolbar to fonts are defined as follows:

```
private JPanel createFont()
{
 JPanel panel = new JPanel(new BorderLayout(10, 0));
 panel.setBorder(new EmptyBorder(5, 0, 5, 0));
 panel.add(createFonts());
 JPanel east = new JPanel(new FlowLayout(FlowLayout.LEFT, 10, 0));
 east.add(new JLabel("Størrelse"));
```

```
 txtSize.setPreferredSize(new Dimension(40, 25));
 east.add(txtSize);
 east.add(chkBold);
 east.add(chkItalic);
 east.add(new JLabel(" | Justering"));
 ButtonGroup group = new ButtonGroup();
 group.add(cmdLeft);
 group.add(cmdCenter);
 group.add(cmdRight);
 east.add(cmdLeft);
 east.add(cmdCenter);
 east.add(cmdRight);
 panel.add(east, BorderLayout.EAST);
 return panel;
}
```

Here is not much new to explain, and the method must mainly insert the panel from *createFonts()* and the remaining components in a *BorderLayout*.

After the design is completed – after a long road, but wee also talk about a very complex design, and by testing you should observe that it is a stable design where the components adapts to the window size.

Back is event handling, and to help with that, the program provides the following methods:

```
private Font createNewFont()
{
 boolean bold = chkBold.isSelected();
 boolean italic = chkItalic.isSelected();
 String name = (String)lstFont.getSelectedItem();
 int size = Integer.parseInt(txtSize.getText());
 if (bold && italic) return new Font(name, Font.BOLD | Font.ITALIC, size);
 if (bold) return new Font(name, Font.BOLD, size);
 if (italic) return new Font(name, Font.ITALIC, size);
 return new Font(name, Font.PLAIN, size);
}

private void alignText()
{
 if (cmdLeft.isSelected()) lblText.setHorizontalAlignment(JLabel.LEFT);
 else if (cmdCenter.isSelected()) lblText.
setHorizontalAlignment(JLabel.CENTER);
 else lblText.setHorizontalAlignment(JLabel.RIGHT);
}

private void foreground()
{
 lblText.setForeground(
  new Color(sldRfg.getValue(), sldGfg.getValue(), sldBfg.getValue()));
}

private void background()
{
 lblText.setBackground(
  new Color(sldRbg.getValue(), sldGbg.getValue(), sldBbg.getValue()));
}
```

The first creates and returns a font based on the settings selected in the top toolbar. The next sets the horizontal alignment of the component *lblText* corresponding to the radio button that is pressed. Finally, the last two methods defines the component's (the sign's) text color and background color from the settings for the *JSlider* components.

The various components can fire an event when an action occurs, and in the previous examples I have shown and used, how a button can fire an *ActionEvent* when clicked, and how this event can be caught and handled by an *ActionListener*. In this example I will use that

- A *JTextField* can fire a *FocusEvent*, when it get the focus (when the user select the field by the tab key or the mouse), and when it lost focus (when the user leave the field). These events can be caught with a *FocusListener*.
- A *JComboBox* firing an *ActionEvent*, when the selection is changed (when the user select another element), that can be caught with an *ActionListener*.
- A *JCheckBox* and a *JRadioButton* firing a *ChangeEvent*, when the state is changed, and it can be caught with a *ChangeListener*.
- A *JSlider* fires a *ChangeEvent*, when the value is changed, and it can be caught with a *ChangeListener*.

With this knowledge the event handlers can be written. They are except for a single simple, and they are written therefore as anonymous classes in method *addListeners()*:

```
private void addListeners()
{
 txtText.addFocusListener(new FocusListener() {
  public void focusLost(FocusEvent e)
  {
   lblText.setText(txtText.getText());
  }
  public void focusGained(FocusEvent e)
  {
  }
 });
 txtSize.addFocusListener(new FocusListener() {
  public void focusLost(FocusEvent e)
  {
   try
   {
    int t = Integer.parseInt(txtSize.getText());
    if (t > 5)
    {
     lblText.setFont(createNewFont());
     return;
    }
   }
   catch (Exception ex)
   {
   }
   txtSize.setText("" + size);
  }
```

```
 public void focusGained(FocusEvent e)
 {
  size = Integer.parseInt(txtSize.getText());
 }
});
lstFont.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e)
 {
  lblText.setFont(createNewFont());
 }
});
chkBold.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  lblText.setFont(createNewFont());
 }
});
chkItalic.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  lblText.setFont(createNewFont());
 }
});
```

```
cmdLeft.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  alignText();
 }
});
cmdCenter.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  alignText();
 }
});
cmdRight.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  alignText();
 }
});
sldRfg.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  foreground();
 }
});
sldGfg.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  foreground();
 }
});
sldBfg.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  foreground();
 }
});
sldRbg.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  background();
 }
});
sldGbg.addChangeListener(new ChangeListener() {
 public void stateChanged(ChangeEvent e)
 {
  background();
 }
});
```

```
  sldBbg.addChangeListener(new ChangeListener() {
   public void stateChanged(ChangeEvent e)
   {
    background();
   }
  });
 }
```

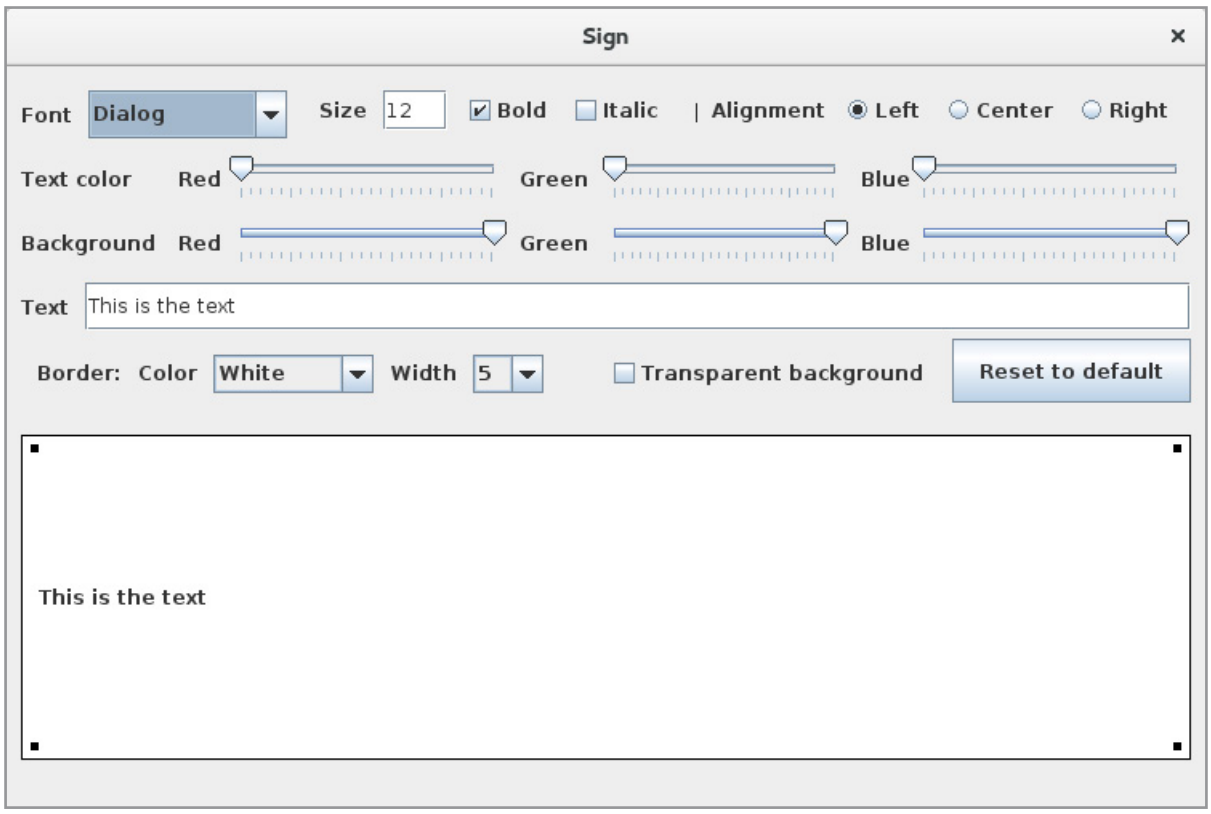
There is not much to explain, but you must note that a *FocusListener* defines two methods, both to be implemented. You should also note that the event handler to the input field for the font size is relatively complex since it must take into account that the user may enter anything illegal. You are invited to test the program and study the code thoroughly, as the example contains much of what is necessary in practice to write a GUI program.

EXERCISE 5

Make a copy of the project *CreateSign*. You have to expand the layout with a new toolbar (see below). The two combo boxes and associated labels and check box must all have a fixed size and be placed to the left as shown below. The button must follow the right edge. The meaning of the new components are as follows. The first combo box should contain the colors that are defined as constants in the class *Color* – except for the colors black and dark gray. The second combo box should contains the numbers from 3 to 20 inclusive. The 4 black square (screw holes) form the corners of a border around the sign. The two combo boxes indicates respectively the color and width of this edge when the width must also apply the size of the 4 squares (they must always be black). The check box must be used to specify whether the sign should have a transparent background – only the interior of the sign and not the edge. Finally, the button is used to restore all settings to default – that is, as they were when the program starts.

Your first task is to add the new toolbar to the program and thus expand the program with the necessary design. You can start by creating the components and then place them in a panel with a *BorderLayout*, where the button is placed EAST while the rest of the components are in a *FlowLayout*, which then is inserted WEST in the first panel.

Next, you define event handlers in *addListeners()* – four in all. The handler for the *JCheckBox* component is simple, but there is little problem to getting the component to update itself when switching from non-transparent to transparent background:



```
public void stateChanged(ChangeEvent e)
{
 lblText.setOpaque(!chkTrans.isSelected());
 lblText.repaint();
}
```

Here is the last statement is required for the changes to take effect. To change the color and width of the frame, it is necessary to have references to the components that make up the frame: The 4 black squares and the 4 edges. It is eight in total, and it is all *JLabel* components. Start therefore by defining 8 instance variables for these components and initialize them to the respective components when they are created. Then it's simple to write a handler for the first combo box to change the color – here, only 4 of the 8 components must change color. As regards to the final combo box that is used to change the components preferred size is a little more difficult. The following method can be used to change the size of a component (the times that a *JLabel*):

```
private void resize(JLabel label, int width)
{
 label.setPreferredSize(new Dimension(width, width));
 label.doLayout();
 label.revalidate();
}
```

Finally, there is the button to the right. When you click on it, all settings should return as at program startup and the sign should be displayed as when the program starts.

6 LAYOUT AND THE COMPONENT'S SIZE

As shown in the above examples, a component's size and location is determined of the so-called layout managers, and the result is not always quite so easy to figure out. In this section I will explain in part how a component's size is calculated, and also where it is placed in the window. When there is a lot to tell it is due to a desire that the window and its components should behave sensibly if the window size changes. If, for example you consider the above program several of the components change sizes when the window is resized, while others components follows the window's right edge. It is controlled by layout managers, which determines how the components of a window or panel are placed.

6.1 THE COMPONENT'S SIZE

A component has a size which is determined by a width and a height, and to define and modify these values a component has four methods

- *setSize()*
- *setPreferredSize()*
- *setMinimumSize()*
- *setMaximumSize()*

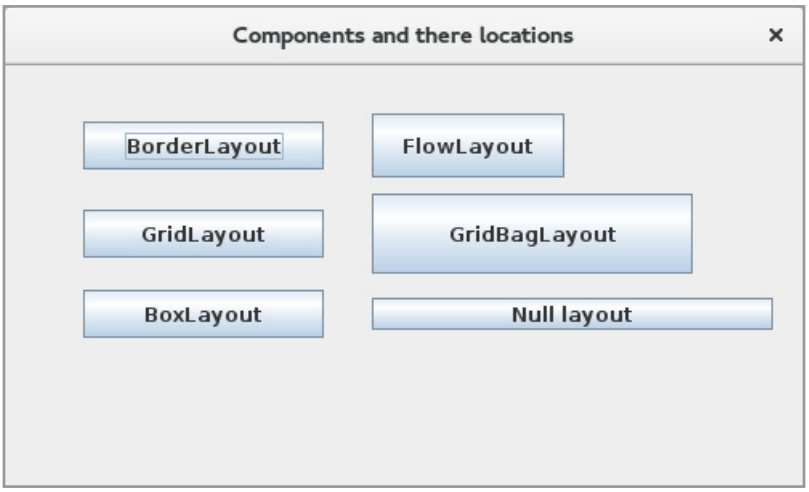
These methods all have a parameter of the type *Dimension*, however, the first has an overloading, where the parameters are two *int* values. The first has only effect if the panel is not using a layout manager. In all other cases it is ignored. It can therefore be used to define the size of the window, because a frame Window does not have a layout manager. In practice are components almost always placed in a panel using one or more layout managers, and here it is the last three methods, which are the interesting. The main rule is that a layout manager will try to customize a component's size to its preferred size, but the manager will not reduce component size to below its minimum size and not increase the size to more than its maximum size. So there are three sizes attached to a component, but the whole thing is complicated because how these quantities are used are determined by the specific layout manager. As a start is shown below the code for a window with 6 buttons:

```
package layoutpanels;

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

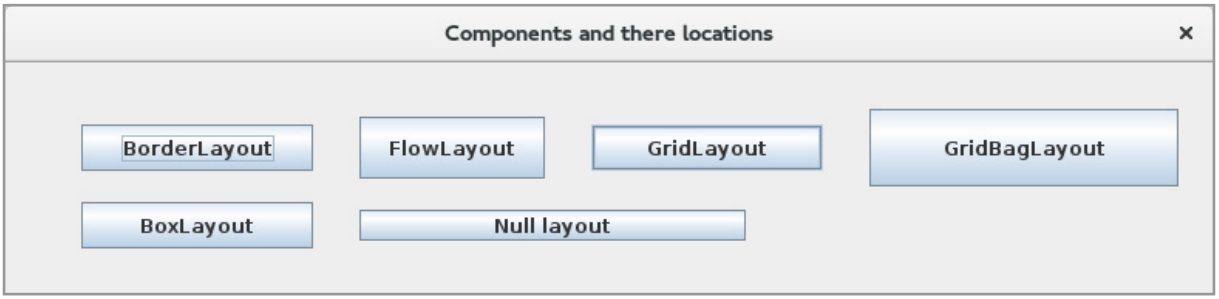
```
public class MainView extends JFrame
{
 public MainView()
 {
  setTitle("Components and there locations");
  setSize(500, 300);
  setLocationRelativeTo(null);
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 30, 10));
  panel.setBorder(new EmptyBorder(20, 20, 20, 20));
  panel.add(createButton("BorderLayout", 150, 30, null));
  panel.add(createButton("FlowLayout", 120, 40, null));
  panel.add(createButton("GridLayout", 150, 30, null));
  panel.add(createButton("GridBagLayout", 200, 50, null));
  panel.add(createButton("BoxLayout", 150, 30, null));
  panel.add(createButton("Null layout", 250, 20, null));
  add(panel);
 }
```

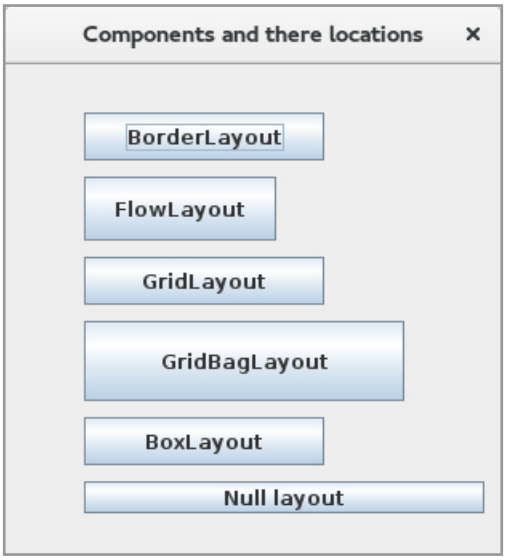
```
  private JButton createButton(String text, int width, int height,
   ActionListener listener)
  {
   JButton cmd = new JButton(text);
   cmd.addActionListener(listener);
   cmd.setPreferredSize(new Dimension(width, height));
   return cmd;
  }
}
```

You should note the method *createButton()*, which creates a button with a preferred size. In addition, the button's event handler is a parameter. The method is used in *createView()* to create the buttons, but so far the last parameter is always *null*, which simply means that the button not yet have an event handler, but the meaning is that when you click on a button, it opens another window which illustrates the effect of a layout manager. If you run the program, you get the following window:



The method *createWindow()* creates a panel with a *FlowLayout*, which is already used several times as a layout manager that places components in a row from left to right. Is there not enough space, the components continue on the next line, and the component's size are determined by their preferred size.





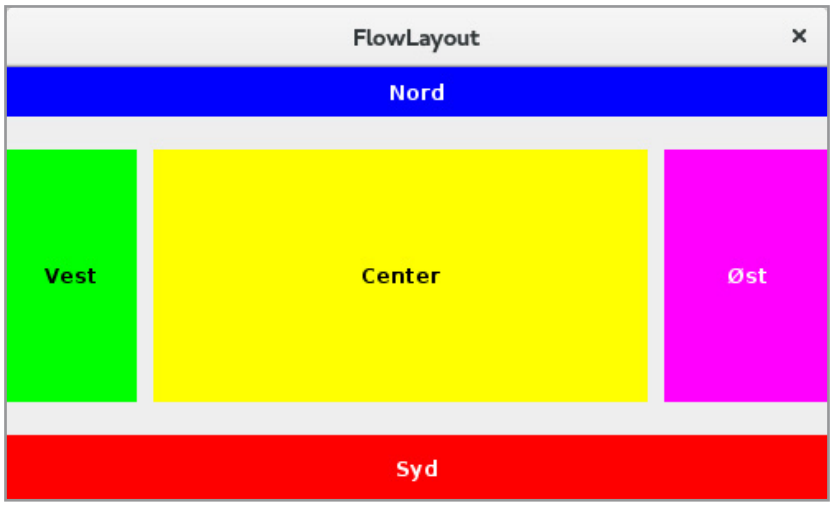
6.2 BORDERLAYOUT

If you click on the top button, a window opens, whose code is as follows:

```
package layoutpanels;

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

public class BorderlayoutView extends JDialog
{
 public BorderlayoutView()
 {
  super(null, "BorderLayout", Dialog.ModalityType.APPLICATION_MODAL);
  setSize(500, 300);
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  setLayout(new BorderLayout(10, 20));
  setBackground(Color.lightGray);
  add(createLabel("North", 0, 30, Color.white, Color.blue), BorderLayout.NORTH);
  add(createLabel("South", 0, 40, Color.white, Color.red), BorderLayout.SOUTH);
  add(createLabel("West", 80, 0, Color.black, Color.green), BorderLayout.WEST);
```

```
  add(createLabel("East", 100, 0, Color.white, Color.magenta),
    BorderLayout.EAST);
  add(createLabel("Center", 0, 0, Color.black, Color.yellow));
 }

 private JLabel createLabel(String text, int width, int height, Color color1,
  Color color2)
 {
  JLabel label = new JLabel(text);
  label.setHorizontalAlignment(JLabel.CENTER);
  label.setOpaque(true);
  label.setBackground(color2);
  label.setForeground(color1);
  label.setPreferredSize(new Dimension(width, height));
  return label;
 }
}
```

This time it is not a *JFrame* window, but a dialog – the class inherits *JDialog*. The dialog box has a *BorderLayout* with five *JLabel* components



Above I have explained how a *BorderLayout* works, and I will not further comments on that layout manager. To open the dialog, the *MainView* must have an event handler for the first button and there is nothing new compared to what has previously been said:

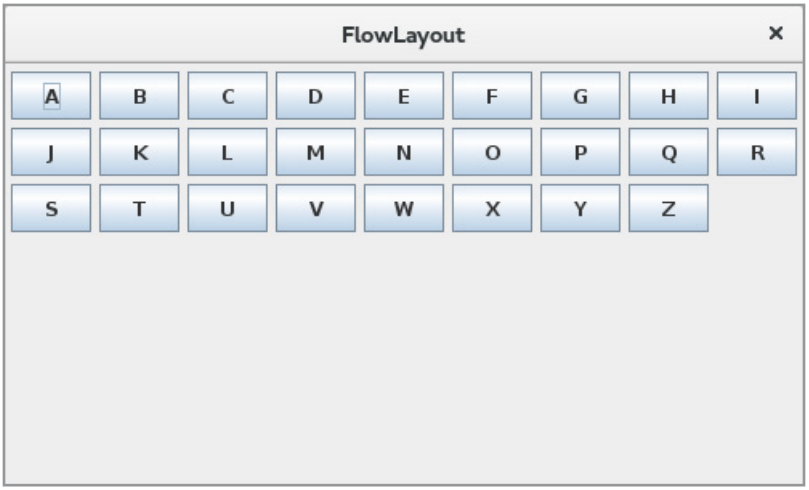
```
private void border(ActionEvent e)
{
 new BorderlayoutView();
}

panel.add(createButton("BorderLayout", 150, 30, this::border));
```

6.3 FLOWLAYOUT

Clicking the second top button (in the *MainView*), you get a window as shown below, to illustrate the use of a *FlowLayout* manager:

```
package layoutpaneler;

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

public class FlowlayoutView extends JDialog
{
 public FlowlayoutView()
 {
  super(null, "FlowLayout", Dialog.ModalityType.APPLICATION_MODAL);
  setSize(500, 300);
```

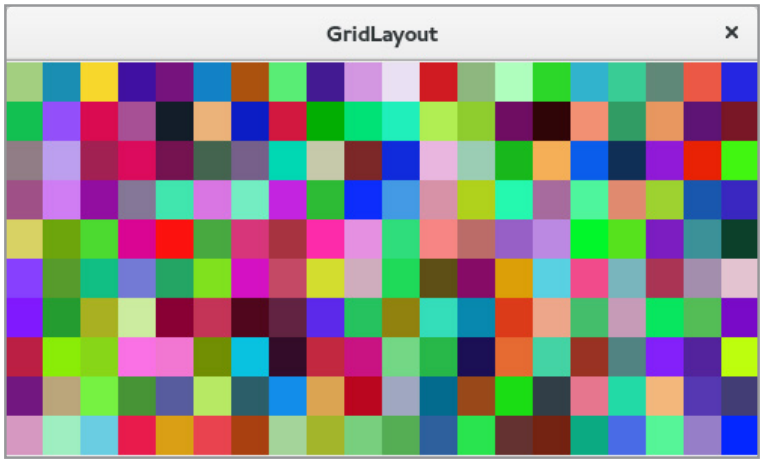
```
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  setLayout(new FlowLayout(FlowLayout.LEFT, 5, 5));
  for (char c = 'A'; c <= 'Z'; ++c) add(createButton("" + c, 50, 30));
 }

 private JButton createButton(String text, int width, int height)
 {
  JButton cmd = new JButton(text);
  cmd.setPreferredSize(new Dimension(width, height));
  cmd.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    JOptionPane.showMessageDialog(FlowlayoutView.this,
     "You have clicked " + text); } });
  return cmd;
 }
}
```



The window is again a modal dialog box, but this time with a *FlowLayout*. There are 27 buttons, and the only thing to watch is what happens when you run the program and change the window size. You can also note how to attach an anonymous event handler for the buttons. It is not particularly readable, but is a short way of writing.

6.4 GRIDLAYOUT

A *GridLayout* is a layout manager that subdivides a panel into a number of rows and a number of columns. If the panel for example has 10 rows and 20 columns, it contains 200 cells, that all are of the same size. Each cell can contain a component, and the component will always fill the entire cell, and a *GridLayout* ignores everything regarding the components preferred size. Below is a window with 200 components located in the panel using a *GridLayout*. The components are all *JLabel* components, which all is displayed with a random background color:



The code is the following:

```
package layoutpaneler;

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

public class GridlayoutView extends JDialog
{
 private static Random rand = new Random();

 public GridlayoutView()
 {
  super(null, "GridLayout", Dialog.ModalityType.MODELESS);
  setSize(500, 300);
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  setLayout(new GridLayout(10, 20));
  for (int r = 0; r < 10; ++r) for (int c = 0; c < 20; ++c) add(createLabel());
 }

 private JLabel createLabel()
 {
  JLabel label = new JLabel();
  label.setOpaque(true);
  label.setBackground(new Color(rand.nextInt(256), rand.nextInt(256),
   rand.nextInt(256)));
  return label;
 }
}
```

You should note how, in the *createWindow()* a *GridLayout* is created. The parameters are, respectively the number of rows and number of columns. You can also specify how much gap there should be between the individual cells – both horizontally and vertically.

EXERCISE 6

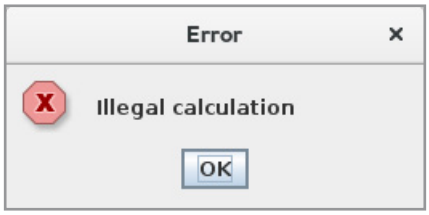
Write a program that you can call *MiniCalc*. The program must open a window as shown below. The window has an input field and 20 buttons. The program should simulate a simple calculator and should only be operated using the mouse. The input field is read only (use method *setEditable()*), and the text is right-justified. The 10 top buttons should be self-explanatory. The buttons in the third row are from left:

1. remove the last character in the display
2. exponentation
3. division
4. subtraction
5. decimal point

The buttons in the bottom row are from left:

1. clear the display
2. shift sign on the content in the display
3. multiplication
4. addition
5. enter, that calculates the value of the expression in the display and update the display with the value

The program must validate that there only are added legal characters in the display (character which leads to a legitimate expression). If you try to perform an illegal calculation by clicking OK, you should get an error message as shown below:



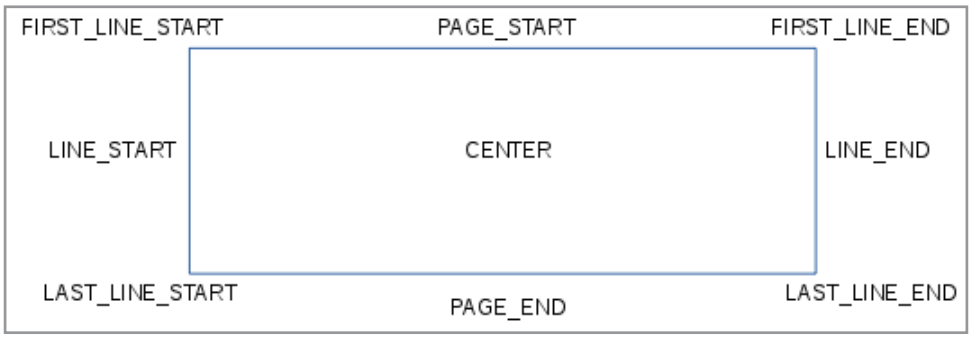
It is obviously a very simplified calculator, and you can only work with very simple expressions, especially because you can not enter parentheses.

6.5 GRIDBAGLAYOUT

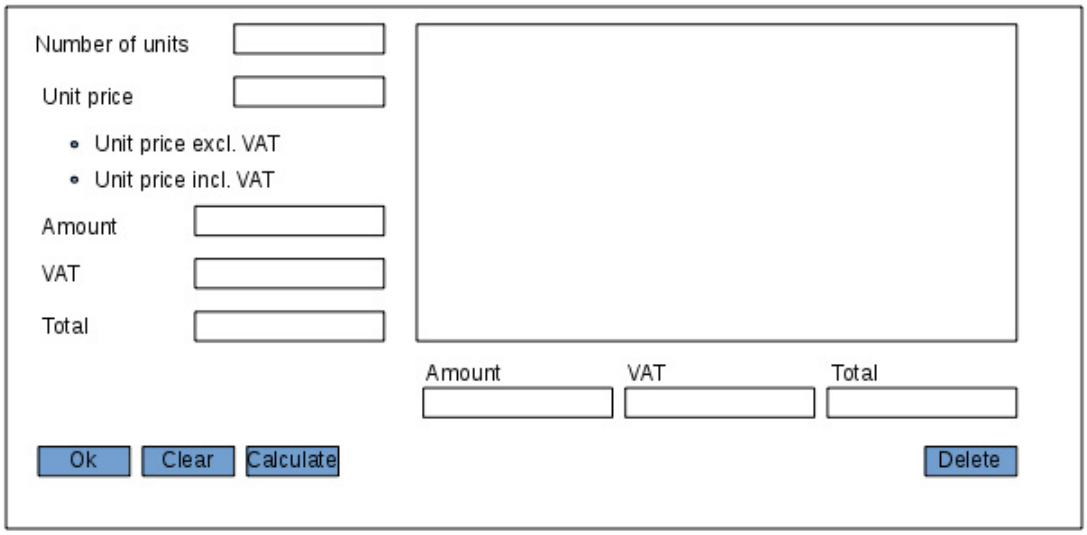
The next layout manager is a very a flexible manager, but is also very complex to work with. As the name says, it is a grid that divides a panel in row and columns, but such that all rows do not have to have the same height, and all the columns do not have to have the same width. The result is the same as in a *GridLayout*, that the panel is divided into a number of cells, but a component may this time spans multiple cells. In addition to what is the exact size of the individual cells is determined by the components's preferred size. How much the components fills, and how they behave when the window size is changed, is determined by a data structure called a *GridBagConstraints* and an object of this kind is associated with the individual components. This data structure has the following fields:

1. *gridx* and *gridy* indicating the component's column and row index in the panel as the upper left corner is (0, 0). You can also specify the value as *GridBagConstraints. RELATIVE* for both column and row, which means relative to the last component, placed in the panel. Default value for both columns and rows are *GridBagConstraints. RELATIVE*.
2. *gridwidth* and *gridheight* indicating respectively how many columns and how many rows this component should span. The default value of both values is 1. For both values it is possible to specify *GridBagConstraints.REMAINDER,* which means that the component will span over the remainder of the row or the remainder of the column.
3. *fill* that indicates how the component should fill the cell if its preferred size is less than the size of the cell. You can specify *GridBagConstraints.NONE*, (which means that the preferred size is used) *GridBagConstraints.HORIZONTAL*, *GridBagConstraints. VERTICAL* and *GridBagConstraints.BOTH* where the first is default.
4. *ipadx* and *ipady* that indicates how much gap there must be outside of the component. Default value is 0. You can think of this value as an edge on the component, but an internal edge which is part of the area that is used for the component.
5. *insets* which indicates an external margin, and therefore how much space, there must be around of the component. The value is a *Insets* object, and the default is no margin.
6. *weightx* and *weighty* indicating how the space will be distributed on respectively columns and rows. The impact of these values is a little difficult to interpret, but they have a value between 0 and 1, where the default is 0. This means that the width of columns and the height of the rows is the maximum preferred width and the maximum preferred height of the components in the column or row. If, you for a component indicates a weight, this means that any free space in the panel should be distributed corresponding to these weights. The weights are therefore central to how the components will behave when the window is resized.

7. *anchor* as in the case where the component's preferred size is less than the cell indicates where the component is to be placed in the cell. The options are shown in the figure below, where the names all refer to the constants in *GridBagConstraints*:



The above sounds complicated, and it also is, and it takes some experimentation to get a *GridBagLayout* to behave as desired. The starting point is to start with a sketch, which can illustrate the window that you want to design. In this case I would design a window where you can enter the number of units and unit price of an item. One must also be able to check whether the unit price is entered with or without VAT. When you click on a button, the program must calculate the total and VAT, and if you click on another button, a line item is inserted in a list box. The design should be something like what is shown as below:



The three fields under the list box must contain totals for the product lines. The list box and the three fields to the totals should follow the window size and the buttons should be positioned relative to the bottom.

In fact, it is a quite complex design and it is an example of a design that can be solved with a *GridBagLayout*. The code is shown below, and it fills much:

```
package layoutpanels;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;

public class GridbaglayoutView extends JDialog
{
 private DefaultListModel model = new DefaultListModel();
 private ButtonGroup group = new ButtonGroup();
 private JTextField txtUnits;
 private JTextField txtPrice;
 private JTextField txtExcl;
 private JTextField txtVAT;
 private JTextField txtIncl;
 private JRadioButton cmdExcl;
 private JRadioButton cmdIncl;
 private JTextField txtExclSum;
 private JTextField txtVATSum;
 private JTextField txtInclSum;
```

```
private double units;
private double price;
private double excl;
private double vat;
private double incl;
private double exclSum;
private double vatSum;
private double inclSum;

public GridbaglayoutView()
{
 super(null, "GridBagLayout", Dialog.ModalityType.APPLICATION_MODAL);
 setSize(800, 500);
 this.setMinimumSize(new Dimension(800, 450));
 this.setLocationRelativeTo(null);
 setDefaultCloseOperation(DISPOSE_ON_CLOSE);
 createWindow();
 setVisible(true);
}

private void createWindow()
{
 setLayout(new BorderLayout());
 JPanel panel = new JPanel(new GridBagLayout());
 panel.setBorder(new EmptyBorder(20, 20, 30, 20));
 addComponent(panel, createLabel("Number of units", 120, 20), 0, 0, 2, 1, 0, 0,
  GridBagConstraints.NONE, GridBagConstraints.LINE_START,
  new Insets(0, 0, 0, 0));
 addComponent(panel, txtUnits = createField(120, 20, true), 2, 0, 1, 1, 0, 0,
  GridBagConstraints.NONE, GridBagConstraints.LINE_START,
  new Insets(0, 0, 20, 20));
 addComponent(panel, createLabel("Unit price", 120, 20), 0, 1, 2, 1, 0, 0,
  GridBagConstraints.NONE, GridBagConstraints.LINE_START,
  new Insets(0, 0, 0, 0));
 addComponent(panel, txtPrice = createField(120, 20, true), 2, 1, 1, 1, 0, 0,
  GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
  new Insets(0, 0, 20, 20));
 addComponent(panel, cmdExcl = createRadio("Unit price excl. VAT", 200, 20,
  true, group), 0, 2, 3, 1, 0, 0,
  GridBagConstraints.NONE, GridBagConstraints.LINE_START,
  new Insets(0, 0, 10, 0));
 addComponent(panel, cmdIncl = createRadio("Unit price incl. VAT", 200, 20,
  false, group), 0, 3, 3, 1, 0, 0,
  GridBagConstraints.NONE, GridBagConstraints.LINE_START,
  new Insets(0, 0, 20, 0));
 addComponent(panel, createLabel("Amount", 100, 20), 0, 4, 1, 1, 0, 0,
  GridBagConstraints.NONE, GridBagConstraints.FIRST_LINE_START,
  new Insets(0, 0, 0, 0));
```

```
addComponent(panel, txtExcl = createField(150, 20, false), 1, 4, 2, 1, 0, 0,
 GridBagConstraints.NONE, GridBagConstraints.LINE_END,
 new Insets(0, 0, 20, 20));
addComponent(panel, createLabel("VAT", 100, 20), 0, 5, 1, 1, 0, 0,
 GridBagConstraints.NONE, GridBagConstraints.FIRST_LINE_START,
 new Insets(0, 0, 0, 0));
addComponent(panel, txtVAT = createField(150, 20, false), 1, 5, 2, 1, 0, 0,
 GridBagConstraints.NONE, GridBagConstraints.FIRST_LINE_END,
 new Insets(0, 0, 20, 20));
addComponent(panel, createList(), 3, 0, 3, 7, 1, 1, GridBagConstraints.BOTH,
 GridBagConstraints.LINE_START, new Insets(0, 0, 10, 0));
addComponent(panel, createLabel("Total", 100, 20), 0, 6, 1, 1, 0, 0,
 GridBagConstraints.NONE, GridBagConstraints.FIRST_LINE_START,
 new Insets(0, 0, 0, 0));
addComponent(panel, txtIncl = createField(150, 20, false), 1, 6, 2, 1, 0, 0,
 GridBagConstraints.NONE, GridBagConstraints.FIRST_LINE_END,
 new Insets(0, 0, 20, 20));
addComponent(panel, createLabel("Amount", 80, 20), 3, 7, 1, 1, 1, 0,
 GridBagConstraints.NONE, GridBagConstraints.LINE_START,
 new Insets(0, 0, 0, 0));
addComponent(panel, createLabel("VAT", 80, 20), 4, 7, 1, 1, 1, 0,
 GridBagConstraints.NONE, GridBagConstraints.LINE_START,
 new Insets(0, 0, 0, 0));
addComponent(panel, createLabel("Total", 80, 20), 5, 7, 1, 1, 1, 0,
 GridBagConstraints.NONE, GridBagConstraints.LINE_START,
 new Insets(0, 0, 0, 0));
addComponent(panel, createButton("OK", 90, 23, this::ok), 0, 9, 1, 1, 0, 0,
 GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
 new Insets(20, 0, 0, 10));
addComponent(panel, createButton("Clear", 90, 23, this::clear), 1, 9, 1, 1, 0,
 0, GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
 new Insets(20, 0, 0, 10));
addComponent(panel, createButton("Calculate", 90, 23, this::calc), 2, 9, 1, 1,
 0, 0, GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
 new Insets(20, 0, 0, 40));
addComponent(panel, txtExclSum = createField(100, 20, false), 3, 8, 1, 1, 1, 0,
 GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
 new Insets(0, 0, 0, 10));
addComponent(panel, txtVATSum = createField(100, 20, false), 4, 8, 1, 1, 1, 0,
 GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
 new Insets(0, 0, 0, 10));
addComponent(panel, txtInclSum = createField(100, 20, false), 5, 8, 1, 1, 1, 0,
 GridBagConstraints.HORIZONTAL, GridBagConstraints.LINE_START,
 new Insets(0, 0, 0, 0));
addComponent(panel, createButton("Delete", 90, 23, this::delete), 5, 9, 1, 1,
 0, 0, GridBagConstraints.NONE, GridBagConstraints.LINE_END,
 new Insets(20, 0, 0, 0));
```

```
 add(panel);
}
private JScrollPane createList()
{
 JList list = new JList(model);
 list.setEnabled(false);
 JScrollPane scroll = new JScrollPane(list);
 scroll.setPreferredSize(new Dimension(400, 200));
 return scroll;
}

private JRadioButton createRadio(String text, int width, int height,
 boolean checked, ButtonGroup group)
{
 JRadioButton cmd = new JRadioButton(text);
 cmd.setPreferredSize(new Dimension(width, height));
 cmd.setSelected(checked);
 group.add(cmd);
 return cmd;
}
```

```
private JLabel createLabel(String text, int width, int height)
{
 JLabel label = new JLabel(text);
 label.setPreferredSize(new Dimension(width, height));
 return label;
}

private JButton createButton(String text, int width, int height,
 ActionListener listener)
{
 JButton cmd = new JButton(text);
 cmd.setPreferredSize(new Dimension(width, height));
 cmd.addActionListener(listener);
 return cmd;
}

private JTextField createField(int width, int height, boolean editable)
{
 JTextField field = new JTextField();
 field.setPreferredSize(new Dimension(width, height));
 field.setEditable(editable);
 field.setHorizontalAlignment(JTextField.RIGHT);
 return field;
}

public static void addComponent(Container container, Component component,
 int gridx, int gridy, int gridwidth, int gridheight, double weightx,
 double weighty, int fill, int anchor, Insets insets)
{
 GridBagConstraints constraints = new GridBagConstraints();
 constraints.gridx = gridx;
 constraints.gridy = gridy;
 constraints.gridwidth = gridwidth;
 constraints.gridheight = gridheight;
 constraints.weightx = weightx;
 constraints.weighty = weighty;
 constraints.fill = fill;
 constraints.anchor = anchor;
 constraints.insets = insets;
 container.add(component, constraints);
}

private void calc(ActionEvent e)
{
 try
 {
  units = Double.parseDouble(txtUnits.getText().trim());
```

```
   price = Double.parseDouble(txtPrice.getText().trim());
   if (units > 0 && price > 0)
   {
    if (cmdIncl.isSelected()) price *= 0.8;
    excl = price * units;
    vat = excl * 0.25;
    incl = excl + vat;
    txtExcl.setText(String.format("%1.2f", excl));
    txtVAT.setText(String.format("%1.2f", vat));
    txtIncl.setText(String.format("%1.2f", incl));
    return;
   }
  }
  catch (Exception ex)
  {
  }
  JOptionPane.showMessageDialog(this,
   "Illegal value for the number of units or unit price",
   "Error message", JOptionPane.ERROR_MESSAGE);
}

private void ok(ActionEvent e)
{
 if (incl > 0)
 {
  model.addElement(String.format(
   "%1.1f units á kr. %1.2f, amount = %1.2f, VAT = %1.2f, total = %1.2f",
   units, price, excl, vat, incl));
  exclSum += excl;
  vatSum += vat;
  inclSum += incl;
  txtExclSum.setText(String.format("%1.2f", exclSum));
  txtVATSum.setText(String.format("%1.2f", vatSum));
  txtInclSum.setText(String.format("%1.2f", inclSum));
  clear(e);
 }
}

private void clear(ActionEvent e)
{
 units = price = excl = incl = vat = 0;
 txtUnits.setText("");
 txtPrice.setText("");
 txtExcl.setText("");
 txtVAT.setText("");
 txtIncl.setText("");
 txtUnits.requestFocus();
}
```

```
  private void delete(ActionEvent e)
 {
  model.clear();
  clear(e);
 }
}
```

The window contains this time many components:

- 8 label components that are objects of the type *JLabel*
- 8 input fields that are objects of the type *JTextField*
- 2 radio buttons that are objects of the type *JRadioButton*
- 1 list box that is an object of the type *JList*
- 4 buttons that are objects of the type *JButton*

If you start at the top, there is a model for the list box and a *ButtonGroup*. The last object should be used for the radio buttons that are attached to a *ButtonGroup*. This ensures that only one radio button can be pressed. Next, are defined the components that you could refer to in the event handlers. It is the input fields and the two radio buttons (actually only one of them). Final is defined variables to the values of the input fields.

The class has several auxiliary methods:

- *createLabel()* is a simple method that creates a label with a preferred size.
- *createField()* creates in the same way an input field. It has a parameter that specify where the content can be edited. That is where the field should be used for input or it only should be used to show a value. You should note that the content is right aligned.
- *createRadio()* is a method that creates a radio button. It has a parameter that tells if the button should be checked, and also the group is a parameter. The group is required, because a window may well have more groups of radio buttons.
- *createButton()* is a method that creates a button and there are parameters for the text, the size and the event handler.
- *createList()* creates the list box and encapsulates the list box in a *JScrollPane*, so you will be able to scroll the content.

Furthermore, there is defined four event handlers for the four buttons:

1. *calc()* is event handler to the button *Calculate*. When pressing it retrieves the contents of the input fields respectively the number of units and the unit price and converts the values and stored them in the respective variables. If entered legal values, the VAT if necessary pulled out of the unit price, and otherwise the product amounts and VAT are calculated and the corresponding fields are updated. Are the values of the one reason or another illegal, you gets an error message.
2. *clear()* is event handler to the *Clear* button, and it clears the input fields and fields to the result and set the variables used for the calculation to zero.
3. *ok()* is event handler for the *OK* button. If there is a calculation, the method add a line to the list box for the current item and updates the fields for the totals. The method also call the handler *clear()* and clears the calculation fields.
4. *delete()* is event handler to the *Delete* button and clears the list box.

Now there's the method *createWindow()*, and that is where it all happens. To add a component the method uses a method called *addComponent()*. This method creates a *GridBagConstraints* object and initialize it using parameters. This object is attached to the component, which is then added to the panel. *createWindow()* creates a *JPanel* with a *GridBagLayout*. In order to have a margin the panel is assigned a border, but otherwise the work consists of placing the 23 components in the panel. I will not go through all 23 components, but I'll look at two as the principle is much the same for all components.

I'll start with the second component, which is the input field to the number of units:

```
addComponent(panel, txtUnits = createField(120, 20, true), 2, 0, 1, 1, 0, 0,
 GridBagConstraints.NONE, GridBagConstraints.LINE_START,
 new Insets(0, 0, 20, 20));
```

The input field is created with a suitable preferred size. This has implications for the size of the cell that contains the component. Then is defined that the cell should be column 2, row 0. When it is placed in column 2, it is because the label component in front og it is spanning two columns. The next two parameters indicates that this component does not need to span the cells other than the cell in which it is placed. The next two parameters again, are the weights, and they indicate with the next fill parameter, the size of this component must always be its preferred size. The second last parameter tells that the component must be adjusted to the left side of the cell, and finally the last parameter indicates that there has to be a margin of 20 to the right of and below the component.

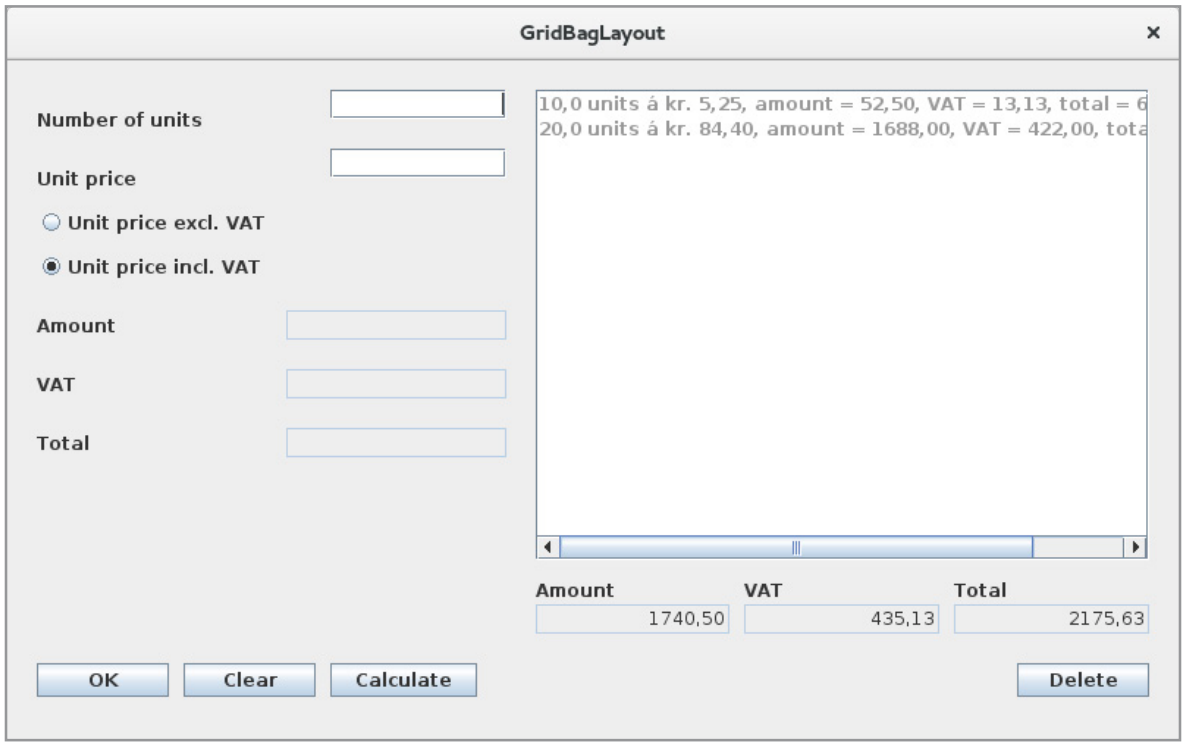
Below is the code that adds the list box:

```
addComponent(panel, createList(), 3, 0, 3, 7, 1, 1, GridBagConstraints.BOTH,
 GridBagConstraints.LINE_START, new Insets(0, 0, 10, 0));
```

It is placed in column 3 row 0, but it should span 3 columns and 7 rows. Next, the weights are set to 1, and it says that the component must use all the available space both horizontally and vertically. At the same time telling the next parameter, the component should fill the entire cell out both horizontally and vertically, and the result is that the component's size follows the window size. Its preferred size is ignored. The two last parameters tells the component to be adjusted to the cell's left edge (ignored in this case, but the method *addComponent()* requires a value), and there must be a margin of 10 below the component.

As a final note to the dialog box, notice that the window in the constructor is assigned a minimum size. The reason is not to made the window smaller, then the layout manager can not display the components.

If you opens the dialog box and enter values for a few items, the result could be as shown below:



EXERCISE 7

Write a program that you can call *Positions* that opens a window as shown below:

The user must enter the name and position of a person, and when you click on OK, the program must inserts a line in the list box. The goal of the exercise is that the window should be designed using a *GridBagLayout*. The top entry fields must have a fixed size, but the list box and the bottom button should follow the window size.

You should first create a simple class that represents a *person*:

```
package positions;

public class Person
{
 private String firstname;
 private String lastname;
 private String position;
```

```
  public Person(String firstname, String lastname, String position)
  {
   this.firstname = firstname;
   this.lastname = lastname;
   this.position = position;
  }

  public String toString()
  {
   return firstname + " " + lastname + ", " + position;
  }
 }
```

To create the window *ManView* you can copy the method *addComponent()* from the above example, and maybe it can also be a good idea to copy the methods *createLabel(), createField()* and *createButton()*. Subsequently, the window may be designed in the same manner as in the above example. I may mention that my *GridBagLayout* has five rows and four columns.

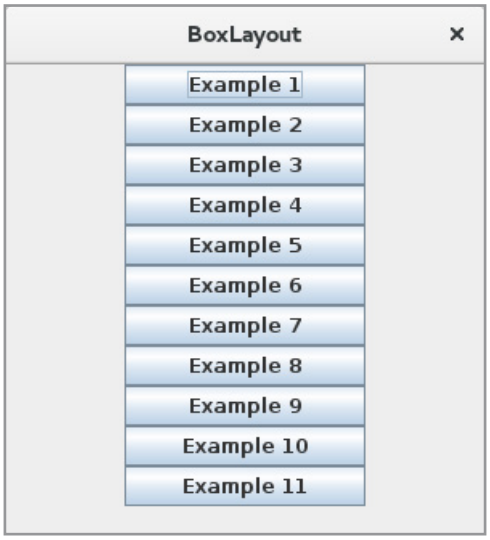
6.6 BOXLAYOUT

The next layout manager is in principle a very simple layout manager, but is also complex, as there are many options, and it can be difficult to figure out how the different settings affects the layout. Basically, it is a layout manager that organizes its components in a row either horizontally or vertically. Immediately it looks like a *FlowLayout*, but there are some more things to be aware of.

When a *BoxLayout* organize components, their location and size will be determined by

- the component's preferred size
- the component's minimum size
- the component's maximum size
- the component's alignment

It is best illustrated through examples, and if you click the button *BoxLayout* in the demo program, you get a window where you can open 11 examples (se below). Here are the first 10 examples virtualy the same, while the latter is a little different. The window's buttons is also laid out using a *BoxLayout*. You are encouraged to run the program and see what happens with the buttons when the window is resized. Here you particularly should notice three things:


BoxLayout
Example 1
Example 2
Example 3
Example 4
Example 5
Example 6
Example 7
Example 8
Example 9
Example 10
Example 11

1. The buttons are located centered in the window. When using a *BoxLayout*, all the container's components usually have the same aligment as different alignment often leads to unexpected results.
2. The component's sizes are not changed, but is defined of their preferred size. This is because they have the same minimum, maximum and preferred size, which then determines the component's size. To ensure that a *BoxLayout* gives the expected result, you should always specify all three sizes of a component.
3. The part of the window, which is not occupied by components is blank, and the blank space (the components are laid out in a column) is always below the components.

The window code is the following, where I have not shown the event handlers:

```
package layoutpanels;

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

public class BoxlayoutView extends JDialog
{
 public BoxlayoutView()
 {
  super(null, "BoxLayout", Dialog.ModalityType.APPLICATION_MODAL);
  setSize(300, 330);
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  setLayout(new BorderLayout());
  JPanel panel = new JPanel();
  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
  panel.add(createButton("Example 1", 150, 25, this::box01));
  panel.add(createButton("Example 2", 150, 25, this::box02));
  panel.add(createButton("Example 3", 150, 25, this::box03));
  panel.add(createButton("Example 4", 150, 25, this::box04));
  panel.add(createButton("Example 5", 150, 25, this::box05));
  panel.add(createButton("Example 6", 150, 25, this::box06));
  panel.add(createButton("Example 7", 150, 25, this::box07));
  panel.add(createButton("Example 8", 150, 25, this::box08));
  panel.add(createButton("Example 9", 150, 25, this::box09));
```

```
    panel.add(createButton("Example 10", 150, 25, this::box10));
    panel.add(createButton("Example 11", 150, 25, this::box11));
    add(panel);
  }

  private JButton createButton(String text, int width, int height,
    ActionListener listener)
  {
    JButton cmd = new JButton(text);
    cmd.setPreferredSize(new Dimension(width, height));
    cmd.setMinimumSize(new Dimension(width, height));
    cmd.setMaximumSize(new Dimension(width, height));
    cmd.setAlignmentX(Component.CENTER_ALIGNMENT);
    cmd.addActionListener(listener);
    return cmd;
  }
}
```

You must primarily note how, the method *createWindow()* defines a *BoxLayout*, and how that

```
BoxLayout.Y_AXIS
```

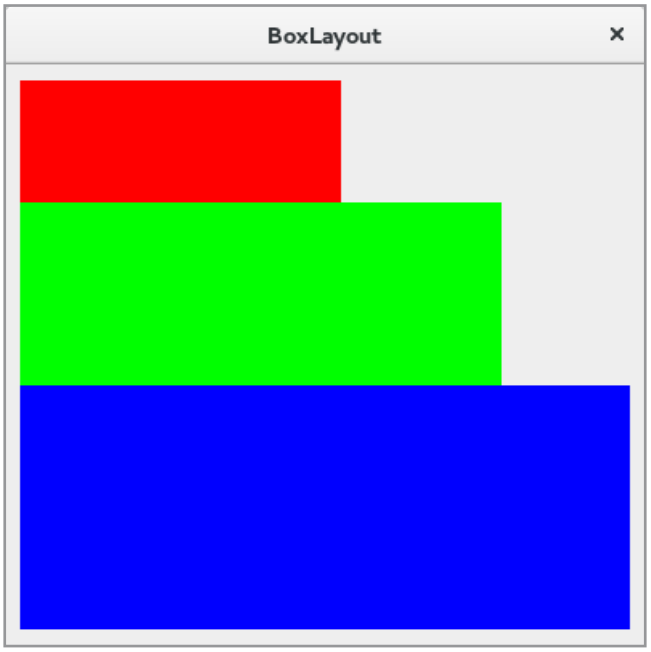
indicates that the components must be laid out in a column. Note also that a *BoxLayout* is defined slightly differently than the other layout managers. Furthermore, note how the method *createButton()* defines the alignment of the individual components:

```
cmd.setAlignmentX(Component.CENTER_ALIGNMENT);
```

that means a horizontal alignment. The parameter is a constant of the type *float* and has a value between 0 and 1. The value indicated the degree to which the component must be aligned from left to right, and there is defined the following constants:

0 = *Component.LEFT_ALIGNMENT*
0.5 = *Component.CENTER_ALIGNMENT*
1 = *Component.RIGHT_ALIGNMENT*

If you open *Example 1* you get the window shown below. The window shows three *JLabel* components that are laid out in a column by a *BoxLayout*. When you test this example, you need to observe what happens to the components when the window is resized. The components changes to their maximum size and are compressed to their minimum size, and finally you should note that they are adjusted with the left edge.

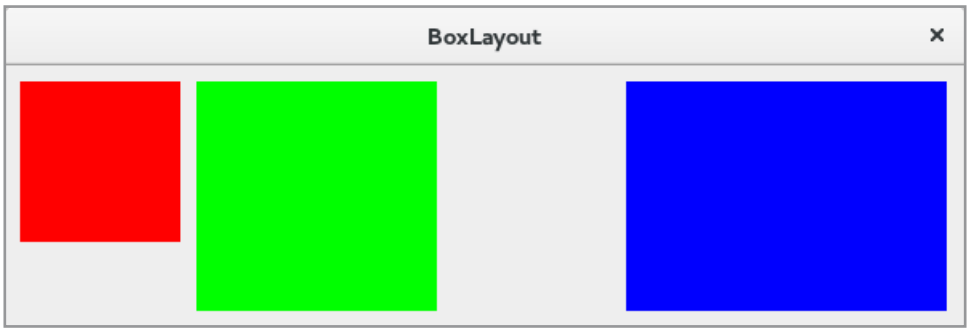

BoxLayout

The code is as follows, which requires no further explanation:

```
package layoutpanels;

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

public class Box01View extends JDialog
{
 public Box01View()
 {
  super(null, "BoxLayout", Dialog.ModalityType.MODELESS);
  setSize(400, 400);
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  setLayout(new BorderLayout());
  JPanel panel = new JPanel();
  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
  panel.setBorder(new EmptyBorder(10, 10, 10, 10));
  panel.add(createLabel(100, 50, Color.red));
  panel.add(createLabel(150, 75, Color.green));
  panel.add(createLabel(200, 100, Color.blue));
  add(panel);
 }

 private JLabel createLabel(int width, int height, Color color)
 {
  JLabel label = new JLabel();
  label.setAlignmentX(Component.LEFT_ALIGNMENT);
  label.setOpaque(true);
  label.setBackground(color);
  label.setPreferredSize(new Dimension(width, height));
  label.setMinimumSize(new Dimension(width / 2, height / 2));
  label.setMaximumSize(new Dimension(width * 2, height * 2));
  return label;
 }
}
```

The following 9 examples are substantially identical to the above, and shows the same three components. The difference is how the components are adjusted, and whether they are laid out vertically or horizontally. I will not show these examples here, but you should open the dialog boxes to see what is happening.

Generally a BoxLayout does not inserts gaps between the components, but it is possible to add no visual components, and I will as an example to explain *Example 10*. If you opens the dialog box, you get the following window:



where the components this time is laid out horizontally. Between the first two components, has been added an invisible component of width 10, while there between the two last components is inserted a gap that fills the part of the panel that are not used. The components height is as previously limited by their maximum height, and the width is their preferred width. The code is as follows:

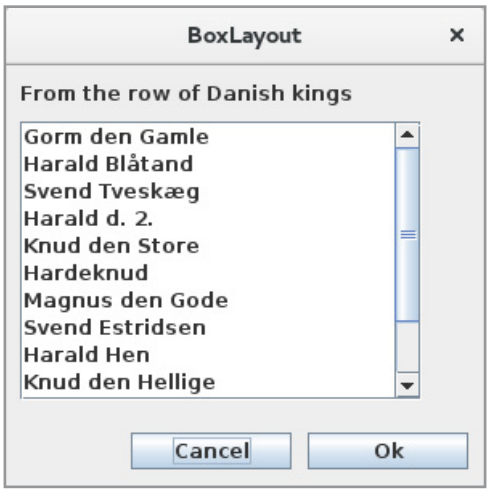
```
package layoutpaneler;

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

public class Box10View extends JDialog
{
 public Box10View()
 {
  super(null, "BoxLayout", Dialog.ModalityType.MODELESS);
  setSize(600, 200);
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }
```

```
private void createWindow()
{
 setLayout(new BorderLayout());
 JPanel panel = new JPanel();
 panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
 panel.setBorder(new EmptyBorder(10, 10, 10, 10));
 panel.add(createLabel(100, 50, Color.red));
 panel.add(Box.createRigidArea(new Dimension(10, 0)));
 panel.add(createLabel(150, 75, Color.green));
 panel.add(Box.createHorizontalGlue());
 panel.add(createLabel(200, 100, Color.blue));
 add(panel);
}

private JLabel createLabel(int width, int height, Color color)
{
 JLabel label = new JLabel();
 label.setAlignmentY(Component.TOP_ALIGNMENT);
 label.setOpaque(true);
 label.setBackground(color);
 label.setPreferredSize(new Dimension(width, height));
 label.setMinimumSize(new Dimension(width / 2, height / 2));
```

```
    label.setMaximumSize(new Dimension(width * 2, height * 2));
    return label;
  }
}
```

which is almost identical to the previous example, but you should note how the *createWindow()* inserts spaces between the components. The important thing about this example is that the blue label follows the window's right edge.

Finally, the last example (Example 11), which opens the following dialog box:



The window places three components with a *BoxLayout*:

1. a *JLabel*
2. a *JScrollPane* with a list box
3. a *JPanel* with two buttons

You should study the code and see what happens when the window is resized.

EXERCISE 8

You must write a program that you can call *Boxes*. The program should open a window as shown below, that shows 12 *JLabel* components in different colors. The program do not perform anything and there should be no event handling.

The design is a panel with a *BoxLayout* that containts two other panels with a *BoxLayout* (a left and a right). In addition, apply the following

- The components must have a fixed size.
- Vertical between the components there must be a gap of 5.
- When the window is resized, the 6 components to the right must follow the window's right edge, while the 6 lower components must follow the window's bottom edge.

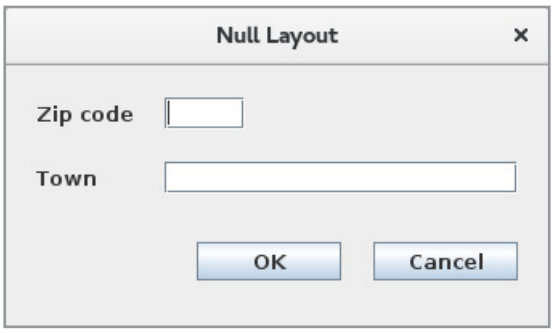
6.7 NULL LAYOUT

Above I have mentioned the most important of Java's layout managers, but it is actually possible to place components in a window without using a layout manager. If, in the demo program you clicks the button *Null layout*, you get the following dialog, that has six components (the buttons have no function)

```
package layoutpanels;

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

```
public class NulllayoutView extends JDialog
{
 public NulllayoutView()
 {
  super(null, "Null Layout", Dialog.ModalityType.MODELESS);
  setSize(340, 200);
  setResizable(false);
  this.setLocationRelativeTo(null);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  createWindow();
  setVisible(true);
 }

 private void createWindow()
 {
  setLayout(null);
  addComponent(this, new JLabel("Zip code"), 20, 20, 100, 20);
  addComponent(this, new JLabel("Town"), 20, 60, 100, 20);
  addComponent(this, new JTextField(), 100, 20, 50, 20);
  addComponent(this, new JTextField(), 100, 60, 220, 20);
  addComponent(this, new JButton("Cancel"), 230, 110, 90, 24);
  addComponent(this, new JButton("OK"), 120, 110, 90, 24);
 }
```

```
  private void addComponent(Container container, Component component,
   int left, int top, int width, int height)
  {
   component.setBounds(left, top, width, height);
   container.add(component);
  }
}
```



The first statement in *createWindow()* sets the layout manager to *null*, which means the window has no layout manager. Then the method adds components using the method *addComponent()*. It defines the components size and location with the method *setBounds()* where the two first parameters are the upper left corner of the component's location in the panel, while the two last parameters are the width and height. That is, the component is assigned an absolute position and size. These values can also be assigned with *setLocation()* and *setSize()*.

Immediately above works simple, but in general it is advisable to use a layout manager, as the component's sizes can involve for example the current font. If you use a particular font, and the window size does not change (note that this is not possible in the above example), the use of components at fixed positions, however is a possibility.

PROBLEM 1

You must write a program that is a loan calculation program and thus a program where the user can enter the amount of a loan, the interest rate and number of peridos. The program should then calculate the payment when the loan is an annuity. It should be mentioned that there are many such programs on the Internet that you can compare the result with. An annuity is a loan that is amortized with a fixed payment each period. A payment consist of interest and repayment and in the beginning is a big part of the payment interest and a smaller part is repayment. This situation is changing continuously, so that towards the end of the loan period, the largest part of the payment is repayment. If

- G = the loan
- y = the payment
- n = number of periods
- r = interest rate

the relationship between the loan and the payment is given by the following formula:

$$G = y\frac{1-(1+r)^{-n}}{r} \Leftrightarrow y = \frac{rG}{1-(1+r)^{-n}}$$

This formula assumes that the interest rate is constant throughout the loan period and the first payment must take place 1 period after you got the loan, and you should assume that these assumptions apply. The following formula determines the outstanding debt immediately after the kth payment is paid:

$$Rest = G(1+r)^{k}\text{-}y\frac{(1+r)^{k}\text{-}1}{r}$$

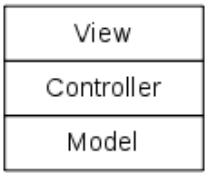
The program must open a window where the user must enter:

- cost of loan formation
- the size of the loan
- the interest rate in percent pro ano
- the repayment period in years
- number of periods a year

Using this information, the program must calculate the payment. In addition, it should be possible to open a window that shows an amortization, and thus an overview of the loan that for each period shows the payment, the interest, the repayment and the debt outstanding after this date.

6.8 MVC

When you study my solution of the above problem, primarily focusing on the program's architecture and the classes that are used. The program can be written differently – and simpler – but the chosen architecture is a step toward a design pattern for a GUI program that is called for MVC for *Model View Controller*. In the book Java 7 I will return to this pattern, and although the pattern is first treated in this book, I will already start to use it. The pattern is very simple and means to develop a GUI program with a three-layer architecture



where the model consists of the classes that defines the program's data and state, while controller layer has classes who mainly perform control of entries etc. and optionally also has essential calculation functions (business logic). Finally, the view layer has all the classes for the user interface and thus for windows and dialogs.

The goal of the pattern is to separate the code so that the code regarding the program's data are placed in classes in the model and code used for data control and logical operations are placed in the control layer, whereas the view layer alone must contain the code which has to do with the visual representation and user interaction. Conforms to the pattern you get a code which might be a bit bigger, but in return is far easier to read and understand.

I will in the following books when there is slightly larger programs begin using the pattern, and so far it is only a question of the division of the program's classes in logical layers, and although it does not sound like much, the pattern has proved very appropriate as architecture for a GUI program. Therefore, I would in a small way begin using the pattern only as a way to a reasonable division of the code. There is much more to say about MVC, and including how each layer should communicate with the others, and there may also be several layers, but the details I'll defer to the book Java 7.

7 PAEDIT

In this chapter I will show a program that is a simple text editor and thus a program where the user can enter text and save the text in a file. The program is relatively simple and has only a single window. The layout is solved with a single *BorderLayout*, and concerning GUI programs the program mainly shows the following:

- how to create and use a menu
- how to create and use a toolbar
- how to use a *JTextArea* component
- how to use finished dialog boxes from the Swing API
- how to use the clipboard

Regarding point 4 I have previously shown the use of *JOptionPane.showMessageDialog()*, but the class *JOptionPane* has other dialog boxes that I will use. Moreover, I shows the use of the class *JFileChooser* that implements a dialog box for browsing the file system.

The program requires that you can read and write a text file, and in the book Java 1 I explained how to do that. In fact, the program does not very much concerning algorithms and thus problem solving, so most of the program deals with how to write text to a file and read text from a file, and it's something you just have to take note of, but behind it all, there are many details that I first are able to explain at a later time.

Similar to what is said above, the program has a very simple architecture, consisting of a view and a model, and by far most of the program's code is in the view. The program has a 2-tier architecture.

7.1 THE MODEL

The model consists only of a single class, called *Document*, which encapsulates a text file, and thus represents the document that the program must be able to edit. The class is written as follows:

```
package paedit;

import java.io.*;

public class Document
{
 private String text;   // the documents text
 private File file;     // object that represents the document's file
```

```
public Document()
{
 text = "";
 file = null;
}

public Document(File file) throws Exception
{
 BufferedReader reader = null;
 try
 {
  StringBuilder builder = new StringBuilder();
  reader = new BufferedReader(new FileReader(file));
  for (String line = reader.readLine(); line != null; line = reader.readLine())
  {
   builder.append(line);
   builder.append("\n");
  }
  text = builder.toString();
  this.file = file;
 }
 catch (Exception ex)
 {
  text = "";
  this.file = null;
  throw new Exception("The content of the file could not be read");
 }
 finally
 {
  if (reader != null) reader.close();
 }
}

public String getText()
{
 return text;
}

public void setText(String text)
{
 this.text = text;
}
public boolean save()
{
 if (file == null) return false; else return save(file);
}
```

```
 public boolean save(File file)
{
 BufferedWriter writer = null;
 try
 {
  writer = new BufferedWriter(new FileWriter(file));
  writer.write(text);
  this.file = file;
  return true;
 }
 catch (Exception ex)
 {
  return false;
 }
 finally
 {
  if (writer != null)
   try
   {
    writer.close();
   }
```

```
      catch (Exception ex)
      {
      }
    }
  }
}
```

There are two variables, the first being for the text, while the other represents a file, and is a *File* object that represents a file path.

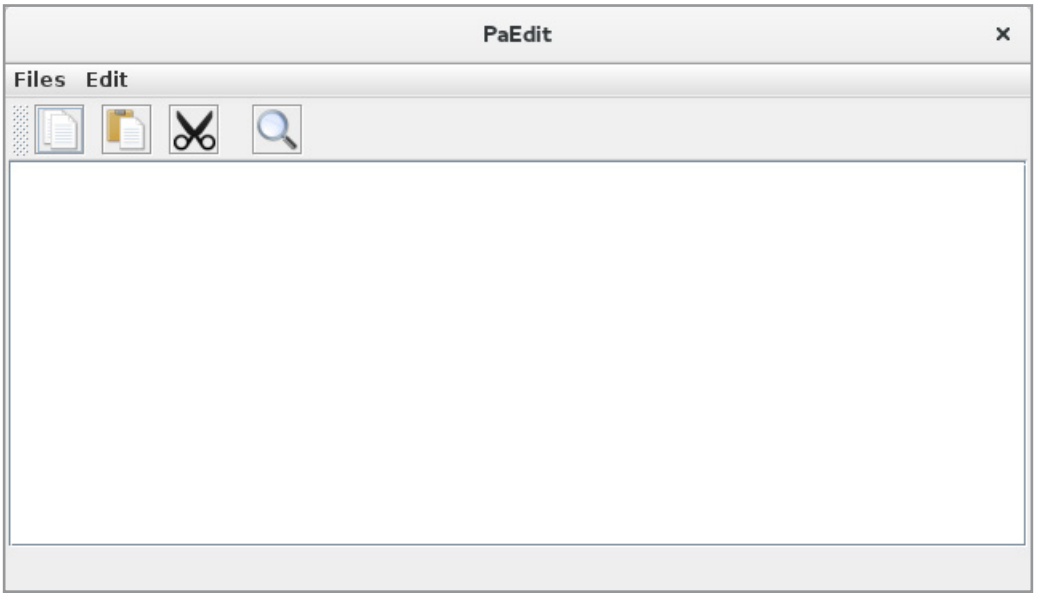
The default constructor creates a blank document, not yet saved and associated with a file. The other constructor has a parameter that is a *File* object, and the constructor trys to read the contents of this file as a text. Most methods concerning files can raise exceptions, as there may be many reasons why a particular file operation can not be performed properly. As an example it may be that the file does not exist, or that you may not have the right to open it. Therefore, statements regarding files almost always are placed in try/catch blocks. In this case raises the constructor an exception if the content of the file for one reason or another can not be read. A text file can be read with a *BufferedReader*, an object that reads the file line by line. As the lines are reading the are added to a *StringBuilder*, which is used to build up the document. If an error occurs, the constructor go to the catch block, and the result is again a blank document.

The method *save()* trys to save the content (the document) in a file, but this is only possible if the variable *file* refers to a file. If it does not, the method returns false. Otherwise the method calls another *save()* method but with a *File* as a parameter. It looks like the constructor, and you write the text to a file using a *BufferedWriter*. The text is saved with the method *write()*, that saves all the text as a whole and also all the line breaks.

Note that the class also has *get* and *set* methods for variable *text*, such the program's view can read the text and update it again.

7.2 THE VIEW

The program's view layer has two classes that is the class *MainView* and a class *Tools*. The last is a simple class that contains a few tools that can also be interested in other programs. If you executes the program, it opens a window as shown below, where there at the top is a menu and a toolbar and in the bottom a status line (is empty when the program starts). Center is a *JTextArea* component, that is an input field like a *JTextField*, but a field where you can enter more lines.



THE CLASS *TOOLS*

The program has a class *Tools* that has only static members, which means that the members can be refered without an object of the type *Tools*:

```
package paedit;

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

public class Tools
{
 public static Font defFont = new Font("Liberation Sans", Font.PLAIN, 14);
 public static Font txtFont = new Font("FreeMono", Font.PLAIN, 16);

 public static Color statusLine = new Color(240, 240, 240);

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

```
 public static JButton createCommand(Icon icon, String toolTip,
  ActionListener listener)
 {
  JButton cmd = new JButton();
  cmd.setFont(defFont);
  cmd.setIcon(icon);
  cmd.setMargin(new Insets(0, 0, 0, 0));
  cmd.setToolTipText(toolTip);
  cmd.addActionListener(listener);
  return cmd;
 }
}
```

Initially are defined three constants where the first two defines the font to respectively default and buttons. The third defines the color of the status bar. In many contexts you must desire to control the fonts and colors, a window must apply, and it is recommended to define that kind of values as constants, as shown above, since in this way you can easily change the values, and as such you can also use them in multiple windows.

The class also has two methods, which are difficult to understand (again because you still lack many Java details), so you should largely accept them as they are. The first is used to load an icon from the application's jar file. The two parameters are the image name and the icon size. The first statement defines a reference to the image in the jar file, while the next statement creates an icon and scale it to the desired size. The last method creates a button with an icon and associate an event handler for the button. In addition a tooltip is added to the button.

THE MENU

It is easy to add a menu to a window. A menu is basically just a collection of buttons, just shown in a different way as a menu item, but the effect is the same, that you can click on a menu item and the item sends an *ActionEvent*. The menu must be defined, and it can fill a lot, but below shows how the menu is defined i *MainView* in this case:

```
private void createMenu()
{
 JMenuBar menuBar = new JMenuBar();
 menuBar.add(createFileMenu());
 menuBar.add(createEditMenu());
 setJMenuBar(menuBar);
}

private JMenu createFileMenu()
{
 JMenu menu = new JMenu("Files");
 menu.add(createMenuItem("New document", this::blank));
 menu.add(createMenuItem("Open document", this::open));
 menu.add(createMenuItem("Savve document", this::save));
 menu.add(createMenuItem("Save document as", this::saveas));
 menu.addSeparator();
 menu.add(createMenuItem("Exit", this::close));
 return menu;
}

private JMenu createEditMenu()
{
 JMenu menu = new JMenu("Edit");
 menu.add(createMenuItem("Copy", this::copy));
 menu.add(createMenuItem("Paste", this::paste));
 menu.add(createMenuItem("Cut", this::cut));
 menu.addSeparator();
 menu.add(createMenuItem("Search", this::search));
 return menu;
}
```

```
private JMenuItem createMenuItem(String text, ActionListener listener)
{
 JMenuItem item = new JMenuItem(text);
 item.addActionListener(listener);
 return item;
}
```

In principle, the code is quite simple and easy of understand. There are three components. The menu is a *JMenuBar* which is a component, that contains the menu and can be added to the window and automatically is placed at the top of the window. The *JMenuBar* is added to the window with the method *setJMenuBar()*. The second component is *JMenu* and represents a menu, while the last is *JMenuItem* and represents a menu item. You should note that you assign a listener in quite the same way you assign a listener to a button. The individual event handlers are written at the end of the class.

THE TOOLBAR

The program's window has a toolbar that is merely a container that can contains components. In this case, there are four buttons, but the buttons are this time not represented by a text, but an image.

Each of these controls include an image (an icon), which must be available for the program. This can be done in several ways, but if, as here it are small icons, you can use the following procedure:

1. add package to the NetBeans project – in this case it is a sub package to *paedit* called *images*, and after you creates the package the name is *paedit.images*
2. copy the images to the corresponding folder (the folder *images* has in this case is 4 png files)

The advantage of this method is that the icons are packed together with the class files in the project's jar file.

Then there is the toolbar, which is defined as follows:

```
private JToolBar createToolbar()
{
 JToolBar toolBar = new JToolBar();
 toolBar.setBackground(Tools.statusLine);
 toolBar.add(Tools.createCommand(
  Tools.createImageIcon("/paedit/images/copy.png", 26),
  "Copy text to the clip board", this::copy ));
 toolBar.addSeparator(new Dimension(10, 10));
```

```
 toolBar.add(Tools.createCommand(
  Tools.createImageIcon("/paedit/images/paste.png", 26),
  "Insert text from the clip board", this::paste ));
 toolBar.addSeparator(new Dimension(10, 10));
 toolBar.add(Tools.createCommand(
  Tools.createImageIcon("/paedit/images/cut.png", 26),
  "Delete text and copy the text to the clip board", this::cut ));
 toolBar.addSeparator(new Dimension(20, 20));
 toolBar.add(Tools.createCommand(
  Tools.createImageIcon("/paedit/images/search.png", 26),
  "Search the document", this::search ));
 toolBar.setPreferredSize(new Dimension(0, 36));
 return toolBar;
}
```

Here I am using the methods from the *Tools* class and you should primarily notice how one refers to the individual images. An image is a resource in the application's jar file, and you must specify the path leading to the current image, the path is relative to the project. Note also the use of the same event handlers as used in the menu. The class *JTextArea* supports also copy/paste, so there is actually not really need for the buttons, but they are included because the goal is to show how to create a toolbar. In this case includes the toolbar buttons, but it may contain any other components.

THE STATUS LINE

It is actually not a real status bar, but just a JLabel which can display a left-aligned text:

```
private JLabel createStatus()
{
 JLabel label = new JLabel();
 label.setFont(Tools.defFont);
 label.setOpaque(false);
 label.setBackground(Tools.statusLine);
 label.setHorizontalAlignment(JLabel.LEFT);
 label.setPreferredSize(new Dimension(0, 25));
 return label;
}
```

Assign with the label is a method that is used to update the text:

```
private void setStatus()
{
 status.setText(String.format("%d linje, %d tegn",
  txtDoc.getLineCount(), txtDoc.getText().length()+ 1));
}
```

It refers to the *JTextArea* component called *txtDoc*, and the status line shows the number of lines and the number of characters entered in the field.

THE LAYOUT

MainView defines four instance variables:

```
private JTextArea txtDoc = new JTextArea();
private JLabel status;
private Document doc = new Document();
private boolean changed = false;
```

The first is the *JTextArea* component which is a component, where one can enter and edit any number of lines of text, and thus an arbitrary document. The next is a *JLabel* to the status line, and the third is the *Document* and then the model. The last keeps track of whether the document is changed.

The method *createWindow()* is similar to the previous examples, and defines the window's design:

```
private void createWindow()
{
 createMenu();
 setLayout(new BorderLayout());
```

```
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(3, 3, 3, 3));
  panel.add(createToolbar(), BorderLayout.NORTH);
  panel.add(status = createStatus(), BorderLayout.SOUTH);
  panel.add(createField());
  add(panel);
  txtDoc.requestFocus();
 }
```

The first statement creates and adds a menu to the window. Otherwise consists the window only of a *JPanel* with a *BorderLayout* which *NORTH* has a toolbar, *SOUTH* has a status bar (the above label) while the *JTextArea* component encapsulated in a *JScrollPane* fills the rest of the window.

The following method initializes the component *txtDoc* which is the editor:

```
 private JScrollPane createField()
 {
  txtDoc.setFont(Tools.txtFont);
  txtDoc.setWrapStyleWord(true);
  txtDoc.setLineWrap(true);
  txtDoc.addKeyListener(new TextChanged());
  return new JScrollPane(txtDoc);
 }
```

The method defines the text to wrap to the next line when the line does not have room for more, and that the division must be done on word boundaries. Finally, there is attached an *KeyListener*, an event handler that is performed each time a key is pressed. It sets the variable *changed* to *true*, which says that the document has changed, and then called the method *setStatus()*, which updates the status line with the current number of lines and the current number of characters:

```
 class TextChanged extends KeyAdapter
 {
  public void keyTyped(KeyEvent e)
  {
   changed = true;
   setStatus();
  }
 }
```

The event handler is defined as a method in a class that inherits *KeyAdater*. When a key is pressed, a *JTextArea* raises more events, and the class *KeyAdapter* is a class that implements the interface *KeyListener* which defines three event handlers regarding the keyboard. *KeyAdapter* implements these handlers as empty methods, and you can then write a class that overrides these handlers that you want to use. Note that it is exactly the same principle that I used in the previous program related events regarding change of the window size, and you also used the same principle in exercise 4 to catch the event when double-click on in item in a list box.

EVENT HANDLERS

Finally, there are 8 event handlers (there are 8 menu items). Below is the event handler that opens a document:

```
private void open(ActionEvent e)
{
 if (changed && JOptionPane.showConfirmDialog(this,
  "The document is changed. Should document be saved?", "Warning",
  JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE) ==
  JOptionPane.YES_OPTION) save();
 JFileChooser fileChooser = new JFileChooser();
```

```
  fileChooser.setCurrentDirectory(new File(System.getProperty("user.home")));
  fileChooser.setDialogTitle("Open text document");
  if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
  {
   File file = fileChooser.getSelectedFile();
   try
   {
    doc = new Document(file);
    txtDoc.setText(doc.getText());
    changed = false;
   }
   catch (Exception ex)
   {
    JOptionPane.showMessageDialog(this, "The document could not be opened",
     "Error message", JOptionPane.OK_OPTION);
    txtDoc.setText("");
    changed = false;
   }
   setStatus();
  }
 }
```

The first statement tests whether the current document is modified. In this case it opens a popup, but this time it's a confirm dialog, a dialog box with a *Yes* and a *No* button, and the user will be asked if the document should be saved. In this case, the method *save()* is called, which saves the document. Next I defined a *JFileChooser* which opens a dialog box where you can browse the file system for the file that you want to open. Accepting a file, you get a *File* object representing the file's path, and the object is used in the constructor in the class *Document* to open the file. After that the input field is initialized with the file's content. You should note that the creation of the document is placed in a try/catch, as the constructor of the class *Document* may raise an exception.

The event handler will possibly call the method *save()*

```
private void save()
{
 doc.setText(txtDoc.getText());
 if (doc.save()) changed = false; else saveas();
}
```

This method copies the contents of the input field to the model and ask the model to write the text back to the file using the model's *save()* method. Is it not possible (because it is a new document that has not been saved) the method calls *saveas()*, where the user again using a *JFileChooser* object will be able to browse the file system and enter a file name.

The event handlers to *Save*, *Saveas* and *New Document* works in principle the same way, and I will not show the code here.

Then there are the three event handlers regarding the clipboard. Below is the handler that copies text to the clipboard:

```
private void copy(ActionEvent e)
{
 try
 {
  String text = txtDoc.getSelectedText();
  Clipboard cb = Toolkit.getDefaultToolkit().getSystemClipboard();
  cb.setContents(new StringSelection(text), null);
 }
 catch (Exception ex)
 {
  JOptionPane.showMessageDialog(this,
   "Text could not be copied to the clipboard",
   "Error message", JOptionPane.OK_OPTION);
 }
}
```

First the selected text is stored in a variable. Next, define a reference to the clipboard, and the text is stored in an object of the type *StringSelection*, and this object is saved om the clipboard. The next handler inserts text stored on the clipboard in the document:

```
private void paste(ActionEvent e)
{
 try
 {
  Clipboard cb = Toolkit.getDefaultToolkit().getSystemClipboard();
  Transferable data = cb.getContents(this);
  if (data == null) return;
  String str = (String)data.getTransferData(DataFlavor.stringFlavor);
  String text = txtDoc.getText();
  int p = txtDoc.getSelectionStart();
  int q = txtDoc.getSelectionEnd();
  if (q > p) txtDoc.setText(q < text.length() ?
   text.substring(0, p) + str + text.substring(q + 1) :
   text.substring(0, p) + str);
  else
  {
   p = txtDoc.getCaretPosition();
   txtDoc.setText(text.substring(0, p) + str + text.substring(p));
  }
```

```
  changed = true;
  setStatus();
 }
 catch (Exception ex)
 {
  JOptionPane.showMessageDialog(this,
   "The text could not be pasted from the clipboard",
   "Error message", JOptionPane.OK_OPTION);
 }
}
```

The handler first defines a reference to the clipboard, and the value is taken as an object of the type *Transferable*. The object is converted into a *String* with a method *getTransfereData()*. This string must be inserted into the input field, either to replace a selected text or inserted at the cursor position. The event handler *cut()* works in principle in the same way.

Back is the event handler for search:

```
private void search(ActionEvent e)
{
 String str = JOptionPane.showInputDialog(this, "Enter a search text", "Search",
  JOptionPane.INFORMATION_MESSAGE);
```

```
  if (str != null && str.length() > 0)
  {
   String text = txtDoc.getText();
   int p = text.indexOf(str);
   if (p >= 0)
   {
    txtDoc.select(p, p + str.length());
    txtDoc.requestFocus();
   }
   else JOptionPane.showMessageDialog(this, "The text does not exist",
    "Information", JOptionPane.OK_OPTION);
  }
 }
```

It does not work quite as it should, since it only finds the first occurrence of the search string. It is a limitation of a *JTextArea*. The code is easy enough to understand, but you must the enter search text. For this purpose is used an input dialog, which is a simple popup where you can enter a string, and the popup is opend with the method *showInputDialog()* that is a method in the class *JOptionPane*.

When the window is closed (and the program stops), I want to test whether the document is changed and possibly should be saved. For this purpose, I has written an inner class that implements an event handler for the event, that the window closes:

```
class WindowCloser extends WindowAdapter
{
 public void windowClosing(WindowEvent e)
 {
  if (changed && JOptionPane.showConfirmDialog(MainView.this,
   "The document is changed. Do you want to save it?",
   "Warning", JOptionPane.YES_NO_OPTION) == JOptionPane.YES_OPTION) save();
 }
}
```

There is no mystery in the code, but the window must be defined as a listener for the event, that happens in the constructor:

```
addWindowListener(new WindowCloser());
```

TEST

When the program is finished, it must be tested. In this case there is not much else to do than try to use the program. However, it can be hard to ensure that you get tested all the situations that may occur. Here it may help to make a test plan where you write down what it is you want to test. It naturally gives no guarantee, but partly just that to write the plan means that you think through what it is you must test, and secondly it helps you remember to get all the points with when you performs the test. A test plan could for instance be something like the following:

1. the program's visual look and feel
2. enter a document
3. save the document
4. enter more text in the same document
5. save it again
6. close the program
7. open the document again
8. edit the content
9. save the document
10. copy text to the clipboard
11. paste the text in the document without replace
12. paste the text in the document and replace selected text
13. cut a selected text
14. paste the text in the document again
15. copy text from another program to this program
16. test the search functionen
17. save the document
18. create a new document
19. open the test document again
20. modify the document
21. close the program without saving the document
22. open the program again
23. open test document
24. open another text document, as an example a java program
25. open a document, that is not text

Later I will deal with testing programs, but a simple test plan as above is better than nothing. One should however be aware that if you find errors during the test, the errors must of course be corrected, and then repeat the test – all the points in the test plan again – and repeating the process until you have completed a test without errors.

8 FINAL EXAMPLE

The task is to write a program that simulates a solitaire. It is a very simple solitaire game where the rules are as follows:

On the table is 16 cards. One must then remove the cards on two criteria:

1. cards of the same color with value less than 10, and where the sum of the card's values are 15 (thus as an example es, five and nine in the same color, or seven and eight in the same color)
2. 10, Jack, Queen and King of the same color

When you have removed cards, the unused places must be filled with cards from the deck – if there are more cards. The solitaire is solved when all cards are removed, but when you get into a situation where you can not remove cards, it means that the solitaire can not be solved.

8.1 THE PROGRAM'S CLASSES

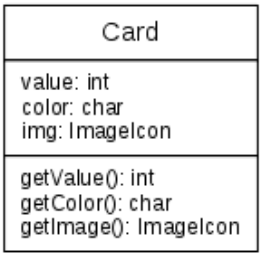
In principle, it is a very simple program in which the user interface has to display images of playing cards, which the user then should be able to click on with the mouse. The program has in principle no other functions, but there must be a possibility to be able to select new game.

There must be kept track of which cards are on the table, which are left in the deck and which have been taken, and for this purpose the program should define a model that always represents the program's current state. The program must therefore in principle work in that way, that when the user clicks on a card, a message is sent to the model on which card is clicked, and the model should then from the cards that are marked determine whether the cards can is removed from the table according to the solitaires rules. If this is the case the model must remove the cards from the table, and selecting new cards from the deck.

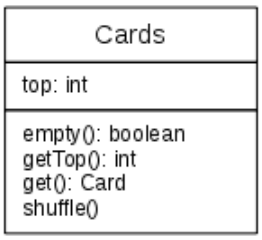
The basic concept in the program is a playing card, which is a very simple concept. A playing card must have three characteristics:

1. a value that must be a number from 1 to 13: 1 = es, 11 = Jack, 12 = Queen and 13 = king
2. a color which should be a character: D (diamonds), H (Hearts), S (spade) C (clubs)
3. an image of the card

Otherwise, a card is passive and can be defined as follows:



Besides a playing card it is also natural to think of a deck of 52 cards. It is not much more than an array of 52 *Card* objects and a variable that can keep track of how many cards are left in the deck:

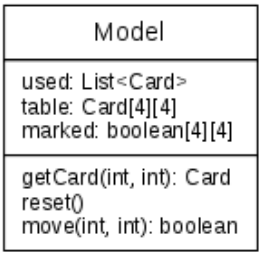


The first method returns while the deck is empty, and the next how many cards left in the deck. The third method returns and remove the card that is on top of the deck and as so simulates that a dealer share a card. The last method shuffles the cards and should be used when selecting a new game.

The model must have an object of type *Cards*, and thus a deck of cards. In order to control the game's state is needed three data structures

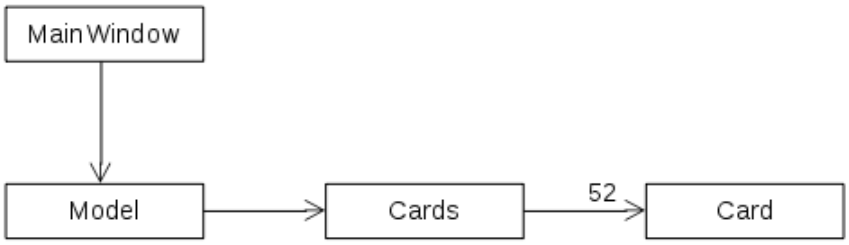
- a list of the cards that has been taken
- an array that keeps track of which cards are on the table
- an array which keeps track of which of the cards on the table is marked

and thus you can outline the model class as follows:



The class has only two essential methods, where *move()* is the most important and is used each time the user clicks on a card. The method *reset()* is for selecting new game.

The program's design will consist of the following classes:



8.2 PROGRAMMING

The starting point is a NetBeans project called *Solitaire16*. To write the program you must have pictures of playing cards. The folder for this book has a package with the name *cards.tar.gz* containing images of the cards. In addition to the 52 cards, there are also two back side cards and an icon to be used in a simple toolbar. To get the pictures packed in the application's jar file, I've created a package named *solitaire16.images*, unpackaged the tar file and copied all the images to that folder.

THE MODEL LAYER

The model layer is composed of the three classes referred to above, and the class *Card* is trivial and are not discussed further. The same applies in principle to the class *Cards*, but it is this class that has to load the images to the cards, and for this I added a helper class with a method that can load an image from the jar file:

```
package solitaire16;

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

```
public class Tools
{
 public static Color selected = Color.darkGray;

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

The class really does not belong in the model layer, since its purpose is to create *ImageIcon* objects that are objects that belongs to the user interface. On the whole, the boundary between model and view layers in the current solution is not quite sharp because the classes in the model layer are using objects that belongs to the view layer, and the program then does not meets the principle behind MVC as the model layer in a manner knows the view layer. One can made a different design of the program, where the classes in the model layer does not know the images, but you can also take the current solution of the program as an example of how patterns and principles are good, but conversely you should not complicate the code just to comply a pattern.

With the above class available, it is simple to write the code for the class *Cards*. It must load the 52 playing cards into an array, and in addition load the images to two back cards. The class also has a method to shuffle the cards, which is done with the following algorithm:

```
public void shuffle()
{
 for (int n = 0; n < 1000; ++n)
 {
  int i = rand.nextInt(array.length);
  int j = rand.nextInt(array.length);
  Card temp = array[i];
  array[i] = array[j];
  array[j] = temp;
 }
 top = array.length;
}
```

The class *Model* defines the following instance variables:

```
public class Model
{
 private Cards deck = new Cards();                  // the deck of cards to be used
 private ArrayList<Card> used = new ArrayList(); // the cards are taken
 private Card[][] table = new Card[4][4];        // the cards on the table
 private boolean[][] marked = new boolean[4][4]; // cards that is marked
```

In addition to what is shown in the design is the class extended by a number of other methods, which has the sole purpose that the user interface can read the model's state. As mentioned, the method *move()* is the most complex, as it is the method to test whether the cards may be removed from the table. This is done with the following algorithm:

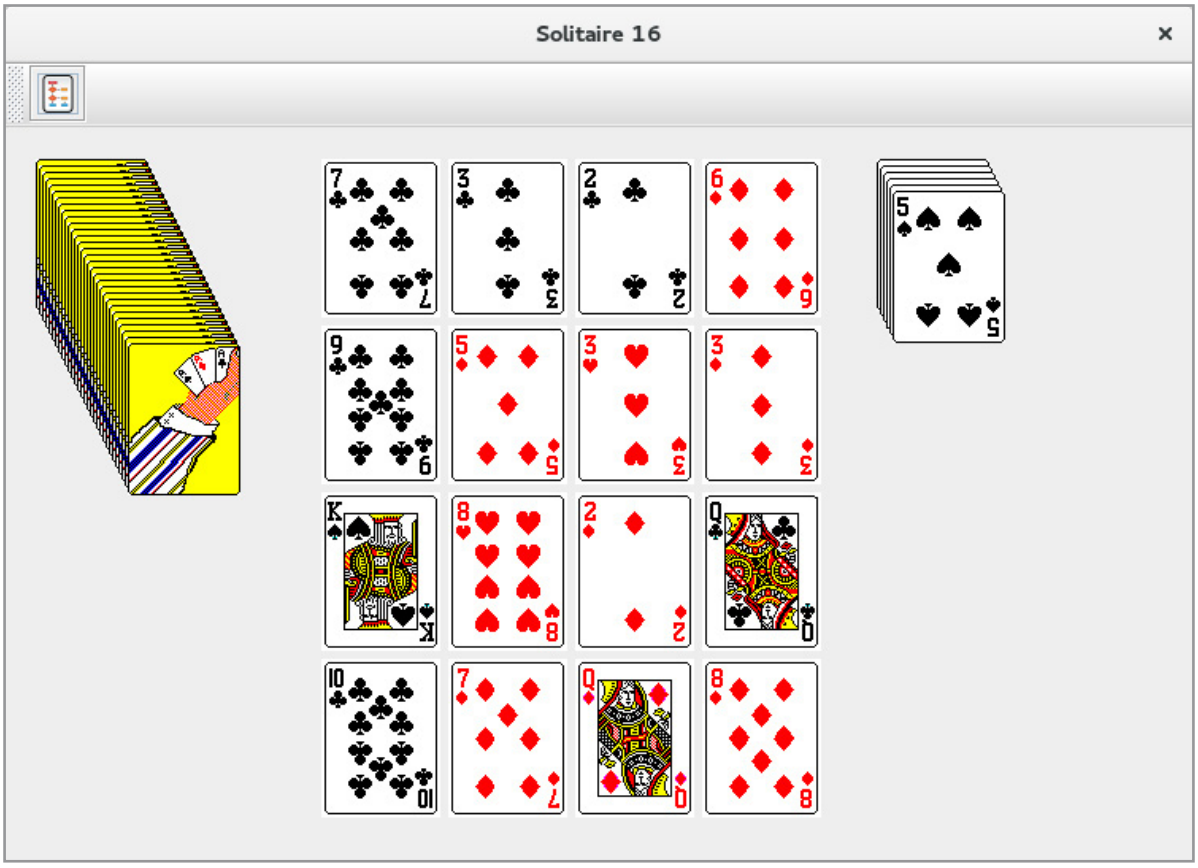
```
if all cards has the same color then
 calculate the sum of the cards values
 if there is a face card then
  if the sum = 46 then move cards
 else
  if the sum = 15 then move cards
```

THE PROGRAM'S VIEW

The user interface contains only a single class, which is *MainWindow*. The class does not take up so much and defines the following instance variables:

```
public class MainWindow extends JFrame
{
 private Model model = new Model();
 private JLabel[][] center = new JLabel[4][4];
 private JLabel[] left = new JLabel[36];
 private JLabel[] right = new JLabel[52];
```

Here is the *center* for the cards on the table, *left* is the deck and must show which cards are left in the deck, and finally *right* for the cards that are taken. *center* shows cards spread out over the table with the front up, while *right* shows the cards that is taken as a deck. *left* shows just a backing card:


Solitaire 16

The window layout is a *BorderLayout*, where *NORTH* has a toolbar with a single button, *WEST* has the deck with the cards not yet dealt, *EAST* has a deck with the cards taken, and finally the 16 cards on the table are *CENTER*.

The images for the cards are displayed with a *JLabel* that specifically can display an image. To show that a card is marked when it is clicked, the background color for the *JLabel* component that contains the image is changed, and it appears as a frame around the picture. The color is defined in the class *Tools*.

It is a very simple solitaire, but it is relatively difficult to solve the solitaire, and you get very quick in a situation where you have to give up, so there is no question of that it is not the most entertaining solitaire.

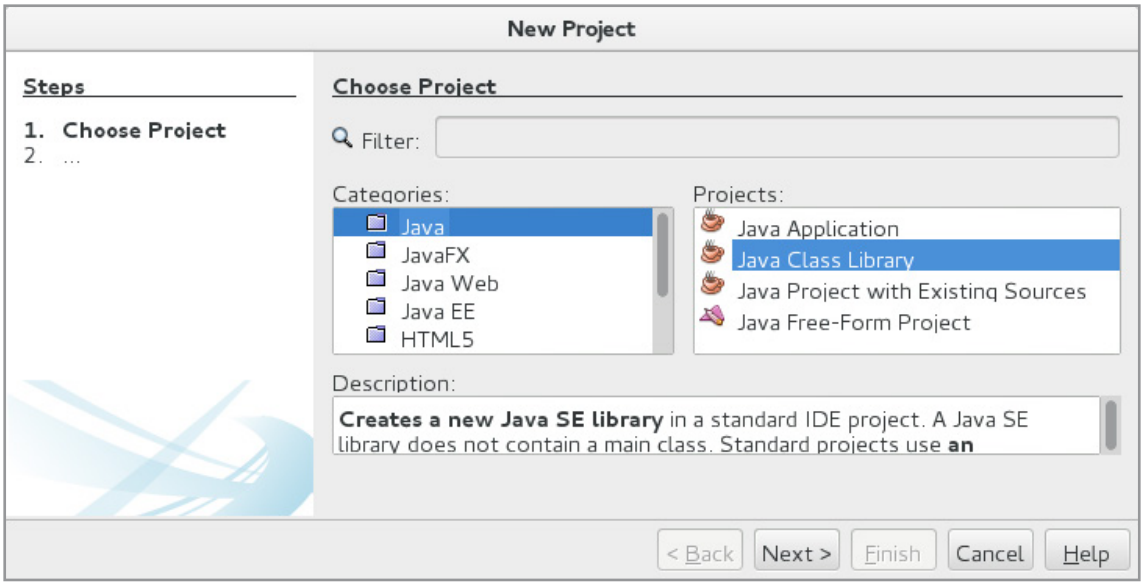
9 A LAST EXAMPLE

As shown in this book require the programming of a graphical user interface many lines of code, and there is even talk of code that is difficult to remember how to write. Conversely, the book's examples also shows that it is the same thing you have to write every time, or at least that the development of the various user interfaces have a number of common features. One can facilitate the work by creating a *class library* that contains classes with methods of the tasks typically encountered, and this library can then be made available to all of the GUI programs that you develop.

I will in this last chapter show how to write such a class library, which I will call *PaLib,* and I also shows a program that use the library. The class library is used in all subsequent books and not only that, the library will continuously be expanded and more of the upcoming exercises and problems actually has to do with how to expand the library. You are therefore encouraged to carefully study how the class library is made and how it is used in the program *History*, a test program that I shows last in this chapter. Although the program relative to GUI programs do not contain anything new, it is nevertheless an appropriate conclusion to this introduction to GUI applications and *Swing*.

9.1 CREATING THE LIBRARY

In principle, it is quite simple to create a class library, and in NetBeans you creates a project in the usual way, just it have to be a *Java Class Library* project:



The result is a project that is empty, and you can then start to add packages and classes. My library is as mentioned called *PaLib*, and so far it consists only of a single package named *palib.gui* and a single class called *Tools* (se below). The class consists exclusively of static methods that are methods that I find useful in the design of user interfaces. I will not show the code here, as it is extensive (it takes up 800 lines), but many of the lines are comments, and the methods are almost all used in the previous examples. As an example I will show a single method (in several overloadings), which shows a little about what the class contains.

It is a method that creates a *JTextField*:

```
/**
 * Creates and returns a JTextField with the following properties:
 * @param width The width of the field
 * @param height The height of the field
```

```
 * @return A JTextField component
 */
public static JTextField createField(int width, int height)
{
 return createField(null, JTextField.LEFT, true, width, height, null, null, null);
}

/**
 * Creates and returns a JTextField with the following properties:
 * @param text The text that must initialize the field
 * @param width The width of the field
 * @param height The height of the field
 * @return A JTextField component
 */
public static JTextField createField(String text, int width, int height)
{
 return createField(text, JTextField.LEFT, true, width, height, null, null, null);
}

/**
 * Creates and returns a JTextField with the following properties:
 * @param text The text that must initialize the field
 * @param width The width of the field
 * @param height The height of the field
 * @param font The font to be used
 * @return A JTextField component
 */
public static JTextField createField(String text, int width, int height, Font font)
{
 return createField(text, JTextField.LEFT, true, width, height, font, null, null);
}

/**
 * Creates and returns a JTextField with the following properties:
 * @param text The text that must initialize the field
 * @param alignment The text alignment, can be JTextField.LEFT or JTextField.RIGHT
 * @param editable Where the contents of the field may be edited or not
 * @param width The width of the field
 * @param height The height of the field
 * @param font The font to be used
 * @return A JTextField component
 */
public static JTextField createField(String text,
int alignment, boolean editable,
 int width, int height, Font font)
{
 return createField(text, alignment, editable, width, height, font, null, null);
}
```

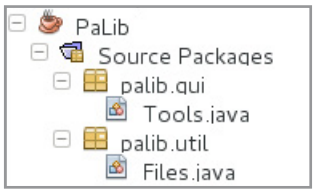
```
/**
 * Creates and returns a JTextField with the following properties:
 * @param text The text that must initialize the field
 * @param alignment The text alignment, can be JTextField.LEFT or JTextField.RIGHT
 * @param editable Where the contents of the field may be edited or not
 * @param width The width of the field
 * @param height The height of the field
 * @param font The font to be used
 * @param fg The field's text color
 * @param bg The field's background color
 * @return A JTextField component
 */
public static JTextField createField(String text, int alignment, boolean editable,
 int width, int height, Font font, Color fg, Color bg)
{
 JTextField field = new JTextField();
 if (text != null) field.setText(text);
 field.setHorizontalAlignment(alignment);
 field.setEditable(editable);
 if (font != null) field.setFont(font);
 if (fg != null) field.setForeground(fg);
 if (bg != null) field.setBackground(bg);
 field.setPreferredSize(new Dimension(width, height));
 return field;
}
```

As regards the methods themselves, and how they work, there is not much to explain, but I will tell you about the considurations concerning the method *createField()*. In practice, I often need to create a *JTextField* and I often has to initialize several properties. It may therefore be useful with a method which parameters has values for the properties to be set, and a method that creates and returns a *JTextField* with these properties initialized. Now you can define many properties of a *JTextField*, and it would not be appropriate by methods of all possible combination of properties, but the last version of the method *createField()* has parameters for the most common properties – at least in relation to the programs that I write. Since there is far from always need of all these parameters, I have written several overloading methods that has fewer parameters.

When you have to write that kind of library classes the biggest considerations are what parameters the methods should have. On the one hand, methods should have so many parameters that they are flexible enough to be used as tools in many applications. On the other hand there should not be so many parameters that it is confusing and the methods equally becomes difficult to use. The choice should reflect typical applications, and in this case the needs I typically have to create a *JTextField*.

I shall not show the rest of the class, but there are similar methods to create the most basic *Swing* components. Furthermore, there are methods to place components in the containers. This is a class with tools, and it is natural that the class is continuously expanded with new methods as you need them. Similarly, the class library may be extended with new classes, and as an example I will extend the library with a class that has methods to write an object to and read an object from a file.

First I has extended the library with a new package called *palib.util* and here a class called *Files*:



In the book Java 1 I shows how to store an abritary object in a file (object serialization) and how to read the object again (object deserialization). It can be difficult to remember how write the statements, and when it is the same you has to write every time it pays to write two library methods for these purposes:

```
package palib.util;

import java.io.*;

public class Files
{
 public static boolean serialize(Object obj, String filename)
 {
  try
  {
   ObjectOutputStream stream =
    new ObjectOutputStream(new FileOutputStream(filename));
   stream.writeObject(obj);
   stream.close();
   return true;
  }
  catch (Exception ex)
  {
   return false;
  }
 }

 public static Object deserialize(String filename)
 {
  try
  {
   ObjectInputStream stream =
    new ObjectInputStream(new FileInputStream(filename));
   Object obj = stream.readObject();
   stream.close();
   return obj;
  }
  catch (Exception ex)
  {
   return null;
  }
 }
}
```

9.2 THE TEST PROGRAM

I will finally show how my class library can be used in a program. I have written a program called *History*, where you can maintain a list of historical persons. It requires that the program can save the list somewhere. When the program starts the first time, it initialize the list of the Danish kings, and each time the list is changed, it is stored in a file on your hard disk. When the program opens the next time, it starts to read the updated list from the file – if possible. Otherwise the list is initialized of the Danish kings. The file is stored in the same location as the program and you do not need to study how the file is stored and read, as the library has the necessary methods.

A person is defined as follows (where I removed the comments and all the *get* and *set* methods):

```
package history;

import java.io.*;

public class Person implements Comparable<Person>, Serializable
{
 private String name;     // the person's name
 private String job;      // the person's position
 private String text;     // a description
 private int from;        // birth, start of reign or otherwise
 private int to;          // year of when the person is dead

 public Person(String name, String job, String text, int from, int to)
 {
  this.name = name;
  this.job = job;
  this.text = text;
  this.from = from;
  this.to = to;
 }

 ….

}
```

When you read the code, you should specifically noting how the comparison of objects are defined by the method *compareTo()* where the objects compared by the year numbers (the class implements the interface *Comparable<Person>*), and note that the class implements the interface *Serializable*. You should also note that the class overrides *equals()*, so people are compared solely on their names.

Then there is the class *Persons* representing the list of *Person* objects.

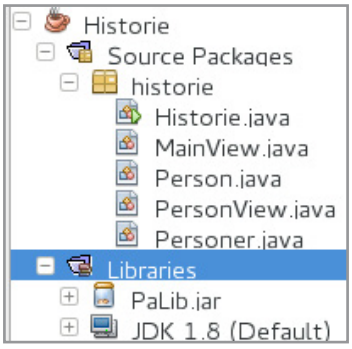
```
package history;

import java.util.*;
import palib.util.*;

public class Persons implements Iterable<Person>
{
 private static String filename = "persons.dat";
 private ArrayList<Person> list;

 public Persons()
 {
  list = (ArrayList<Person>)Files.deserialize(filename);
  if (list == null) initialize();
 }
 public Iterator<Person> iterator()
 {
  return list.iterator();
 }
```

```
 public void add(Person pers)
 {
  list.add(pers);
  Collections.sort(list);
  Files.serialize(list, filename);
 }

 public void remove(Person pers)
 {
  list.remove(pers);
  Files.serialize(list, filename);
 }

 public void update(Person pers)
 {
  for (Person p : list)
  {
   if (p.equals(pers))
   {
    p.setJob(pers.getJob());
    p.setText(pers.getText());
    p.setFrom(pers.getFrom());
    p.setTo(pers.getTo());
    break;
   }
  }
  Files.serialize(list, filename);
 }

 private void initialize()
 {
  list = new ArrayList();
  for (int i = 0;i < navne.length; ++i)
  {
   String job = navne[i][0].equals("Margrete d. 1.") ||
    navne[i][0].equals("Margrethe d. 2.") ? "Queen" : "King";
   int fra = navne[i][1].length() > 0 ? Integer.parseInt(navne[i][1]) : -9999;
   int til = navne[i][2].length() > 0 ? Integer.parseInt(navne[i][2]) : 9999;
   list.add(new Person(navne[i][0], job, "", fra, til));
  }
  Collections.sort(list);
 }

 private static String[][] navne = {
  { "Gorm den Gamle", "", "958" },
  { "Harald Blåtand", "958", "986" },
   …
 };
}
```

First you note the import statement

```
import palib.util.*;
```

referring to the package containing the class *Files*. In order to be possible, the jar file for the class library must be available for the program. It is obtained for the project (the program *History*), by right-click on *Libraries* and here choose *Add JAR / Folder…* You can then browse to class library's jar file and add it to the project:



Now the application can use the jar file and its classes. If we now builds the project and copy the file *History.jar* to another folder and try to run the program from a Terminal window, you get an error message because the program can not find the class library. The class library is not part of the program's jar-file. If you opens the project's *dist* folder, you will see that there is a subdirectory named *lib*. It contains the jar file to class library, and for the program to run, this subfolder must also be copied.

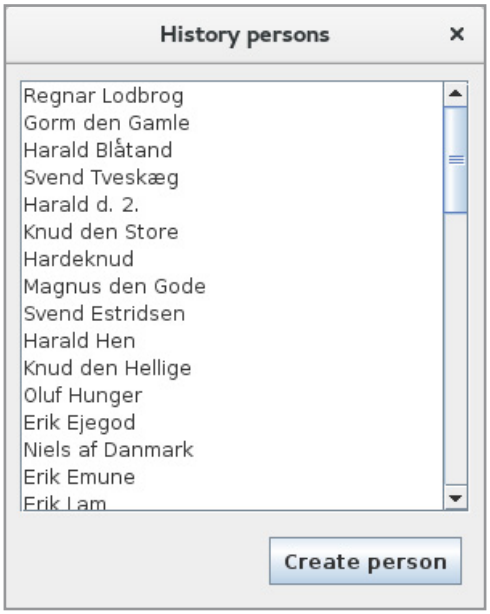
The class *Persons* is in principle simple and you should notice how *Person* objects are stored in an *ArrayList*, and the class has methods to add an item, update an item, and delete an item. Note specifically that each of these methods serialize the list using the method *serialize()* in the class *Files*, which are a class in the class library.

The most complex in the class is the constructor. It tries to deserialize the list from a file. If this fails, for example because the file does not exist (and indeed it does not the first time the program runs), the list is initialized using data that is laid out in the program (Danish kings and I have only shown the first two). The list is created in the method *initialize()*, which runs through the static array with data.

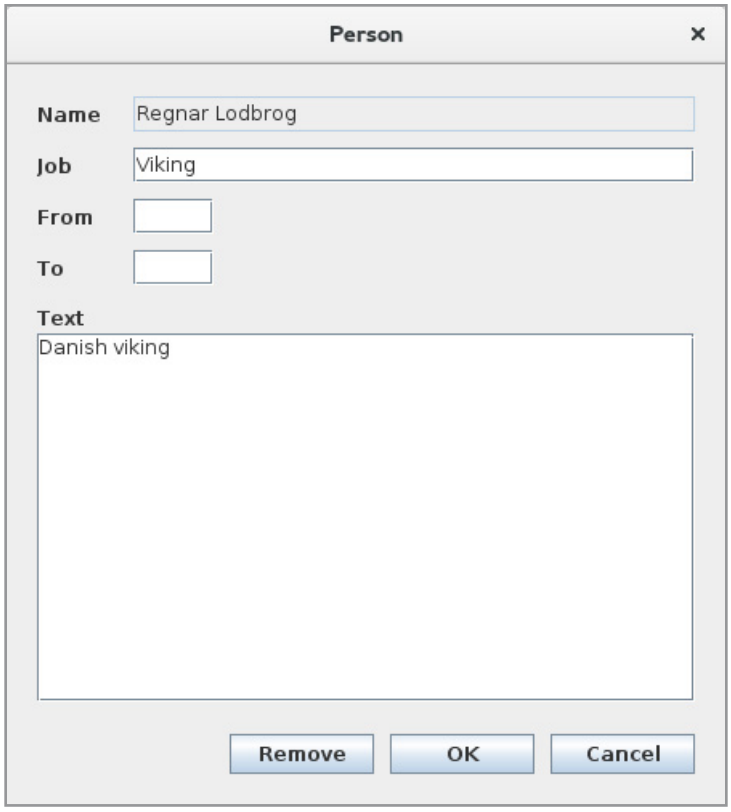
You should also note the method *iterator()*. It will first be explained in the next book, but it means that it is possible to iterate over a *Persons* object's items with a *for* statement.

If you run the program, you get a window as shown below. The window is the program's main window, and I will not show the code here, as there not is something new, but you should notice how the class use the class library for constructing the user interface.

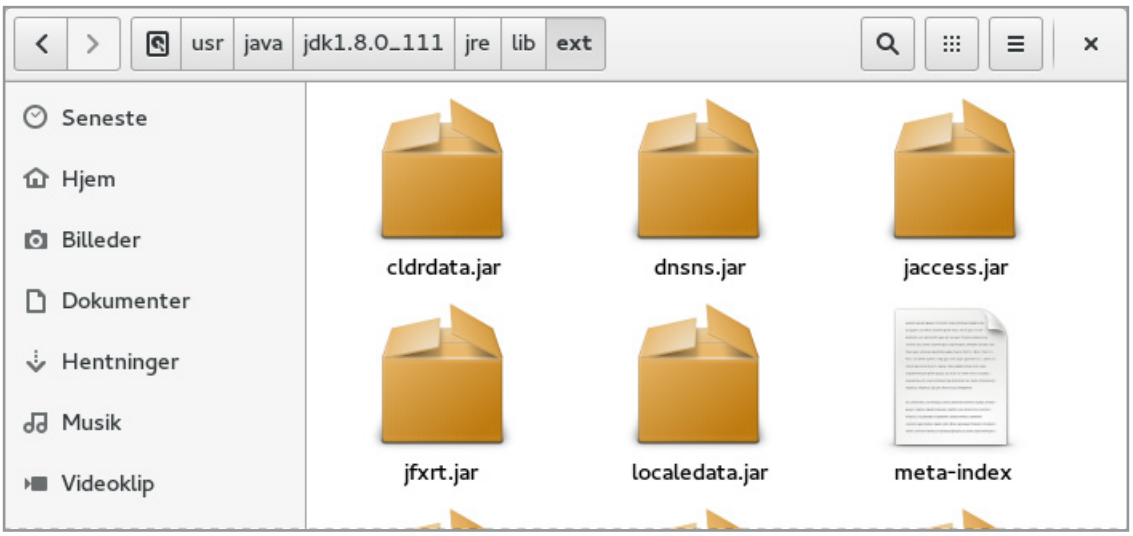
If you double click on a line (a name) in the list, you get a window where you can see all the information, edit them and possibly delete the person (see below). The class is named *PersonView*, where I will not show the code, but the window layout is defined by a *GridBagLayout* and the class makes widely use of the class library. If you in the main window, click the button *Create person* opens the same window, but with all fields blank, and you can then enter the name of a person and thus add a new *Person* to the list.



Below is an example of the window *PersonView*, when double clicked on a line in the list box:



If you now have many programs that use the class library, each program must have a copy of the class library's jar file. Of course it is not particularly useful, and it should be such that there is only one copy, that all programs are using. It can be solved in several ways, but a very simple way (although far from the best) is to place the jar file as shown below:



Here are *jdk1.8.0_111* the folder containing my Java and can of course be replaced by another. To make a jar file available in that way by placing it in the folder as shown above is not necessarily the best way, but it's a simple way, which is well worth knowing.