Poul Klausen

JAVA 14

Development of applications with JavaFX

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 14: DEVELOPMENT OF APPLICATIONS WITH JAVAFX

SOFTWARE DEVELOPMENT

Java 14: Development of applications with JavaFX: Software Development
1st edition

ISBN 978-87-403-2188-3
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the fourteenth in a series of books on software development and deals with JavaFX. The book introduces JavaFX as an alternative to Swing and Java2D, thus treating how to work with 2D graphics in JavaFX, as well as the main components and layout. The goal is that you will quickly be able to write programs with a graphical user interface using JavaFX. There are many topics, especially JavaFX properties, which the book does not include, but these topics are dealt with in the following book. JavaFX is intended as an alternative to Swing, and it is especially for better graphics and other media used in the user interface. With JavaFX, it is easier to develop applications with a modern user interface and with all the possibilities that users expect and get used to from web applications. The book does not require the reader to know Swing and Java2D in detail, but assumes that the reader has an introductory knowledge of developing programs with a graphical user interface.

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

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

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer

technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

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

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

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

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

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

1 INTRODUCTION

In the books Java 2, Java 9 and primarily Java 10, I have dealt with how to write a standalone Java application with a graphical user interface. All of it has been based on Swing, which is an API that makes the necessary available through a comprehensive class library. There is nothing wrong with Swing, and it will be there many years to come, among other things because there are a lot of programs based on Swing. However, there are alternatives and one of them is called *JavaFX*, as the new API for developing Java applications with a graphical user interface.

JavaFX is intended as an alternative to Swing, and it is especially for better graphics and other media used in the user interface. With JavaFX, it is easier to develop applications with a modern user interface and with all the possibilities that users expect and get used to from web applications. However, it is important to emphasize that you can still use Swing components, so that all known facilities from Swing are still available.

JavaFX is an alternative to Swing, but the idea actually originates from World Wide Web, where you can build the user interfaces using markup, and to a large extent you can design the applications user interface using styles. The goal of this book is to show how to use JavaFX, and I will alone look at the development of standalone applications, but the goal of JavaFX is also the development of the client side of web application, which means that a JavaFX application can be opened as

4. a standalone desktop application
5. with the *Java Web Start* tecnology
6. as a part of a web page

For the time being, I will only look at the first option.

1.1 HELLOFX

As a start, I will show a very simple application that opens a window with a few buttons and a little more, but a program that uses JavaFX instead of Swing. I start with a new project in Netbeans, but it must be a *JavaFX Application* project. I have called the project *HelloFX*, and after the project has been created, NetBeans has created a single class (where I have removed all comments):

```
package hellofx;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloFX extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  Button btn = new Button();
  btn.setText("Say 'Hello World'");
  btn.setOnAction(new EventHandler<ActionEvent>() {
   @Override
   public void handle(ActionEvent event)
   {
    System.out.println("Hello World!");
   }
  });
  StackPane root = new StackPane();
  root.getChildren().add(btn);
  Scene scene = new Scene(root, 300, 250);
  primaryStage.setTitle("Hello World!");
  primaryStage.setScene(scene);
  primaryStage.show();
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

In fact, it is a fully finished program, and starting the program opens the following window:

and clicking on the button it prints a message on the console. If you consider the code, first notice that the class inherits *Application*, which means that it is a *JavaFX* application. The class has a usual *main()* method as other Java applications, and it starts by calling a static method *launch()* (in the class *Application*). It basically performs the following:

1. instantiates an object of the class (here the class *HelloFX*)
2. performs a method *init()*, which in *Application* is an empty method
3. performs a method *start()*, which in *Application* is an empty method
4. waiting until the program terminates, where method *stop()*, as in *Application* is an empty method, is performed

If you want an application to open a window, it usually happens to override the method *start()* from the class *Application*, as is the case in the above example.

A JavaFX program basically consists of a *Stage* object that contains a *Scene* object, which again contains *Node* objects arranged in a tree. A *Stage* object represents a window and the primary *Stage* object is created by the runtime system and is sent as a parameter to the method *start()*, and in the above example it has the name *primaryStage*. A *Scene* object contains all of the window's objects organized into a tree called a *scene graph*. The tree's elements are *Node* objects, and a node can be an *branch node* (which has children) or a *leaf node* (which no children have). A node is a component in the window and may, for example, be a button or an entry field.

In the example above, the method *start()* begins by creating a button, which is a node. The object is called *btn*, and besides attaching a text to the button, an action is also associated, which is an object with a single method, which simply performs a *System.out.println()*. You should note that it looks like a usual event handling as it is known from Swing, but it is not the case, and other types are used, although they have the same names as some of the known types from Swing.

As a next step, a *root* object of the type *StackPane* is created, and it is the root of the scene graph and the button is inserted as a child to *root*. This program thus has a very simple scene graph consisting of a root and a single leaf node. The root object is then added to a *Sceen* object, which represents the window's content, while defining the window size. This *Scene* object must be attached to the *Stage* object, which is also assigned a title, and finally, the method *show()* is performed on the object *primaryStage*, which displays the window on the screen.

It is thus easy to follow what happens in the above program, but it is more difficult to spot the benefits. They come later, but I can immediately point out that one of the advantages is that the technology used to draw the window with all its content is extremely effective.

Before I leave this example, I will change the code a bit (for now, the code appears as NetBeans has created it). I have changed the code to the following:

```
package hellofx;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.application.Platform;

public class HelloFX extends Application
{
 @Override
 public void start(Stage stage)
 {
  System.out.println("Start " + Thread.currentThread().getId());
  TextField txtName = new TextField();
  Label lblText = new Label();
  lblText.setStyle("-fx-text-fill: blue;");
  Button cmdHello = new Button("Hello");
  cmdHello.setOnAction(new EventHandler<ActionEvent>()
  {
   @Override
   public void handle(ActionEvent e)
   {
    String name = txtName.getText();
    lblText.setText(name.trim().length() > 0 ? "Hello " + name :
     "Hello there");
```

```
    }
   });
   VBox root = new VBox();
   root.setSpacing(10);
   root.getChildren().addAll(new Label("Enter your name:"), txtName, lblText,
    cmdHello, createButton("Exit", e -> Platform.exit()));
   stage.setScene(new Scene(root, 300, 200));
   stage.setTitle("Hello FX");
   stage.show();
  }

  private Button createButton(String text, EventHandler<ActionEvent> handler)
  {
   Button cmd = new Button(text);
   cmd.setOnAction(handler);
   return cmd;
  }

  public static void main(String[] args)
  {
   launch(args);
  }

  public HelloFX()
  {
   System.out.println("Constructor " + Thread.currentThread().getId());
  }

  @Override
  public void init()
  {
   System.out.println("Init " + Thread.currentThread().getId());
  }

  @Override
  public void stop()
  {
   System.out.println("Stop " + Thread.currentThread().getId());
  }
}
```

If you run the program, it first writes the following lines on the console:

```
Constructor 12
Init 13
Start 12
```

after which you get the following window

If you enter the text *Frode Fredegod* and click *Hello*, the window will be updated to:

and clicking *Exit* terminates the program after writing the following text on the console:

```
Stop 12
```

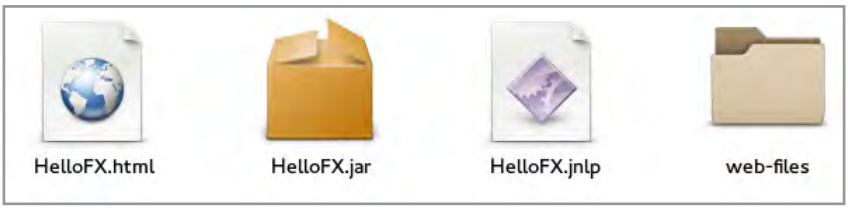
This time, the method *start()* creates a scene graph with a *root* that has 5 leaf nodes. Note that the type of *root* is *VBox*. There is no particular justification in addition to showing that there are more options, and the type is relevant to how the graph's nodes are laid out in the window. The 5 nodes have the types *Label*, *TextField*, *Label*, *Button* and *Button*, where the first and the last are created when they are added to *root* (the last created of the method *createButton()*), while the other three are created explicitly.

Note that the method *start()* starts by printing a text (on the console). I have also overwritten the two methods *init()* and *stop()* so they both print a text and finally I have added a default constructor. They show that the constructor is first executed, then *init()* and finally *start()*. Finally, they show that the method *stop()* is executed when the program terminates. Here you should note that the constructor, *start()* and *stop()* are executed in the same thread (called the *JavaFX Application Thread*), while *init()* is performed in its own thread (called the *JavaFX Launcher*).

If you look at the directory *dist*, you can see that a jar file named *HelloFX.jar* has been created in the usual way, and the program can therefore be executed from the command line as (if current directory is the folder *dist*):

```
java -jar HelloFX.jar
```

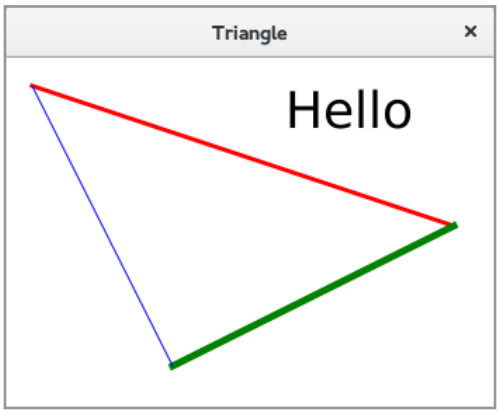
The directory *dist* also contains other files:



These files I do not want to mention in this place, but they all have to do with the execution of the program as a web application.

1.2 HELLOLINES

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



The project is called *HelloLines*, and the window contains a scene graph, consisting of a root with 4 child nodes: a *Text* object and three *Line* objects. The code is as follows:

```
package hellolines;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.shape.Line;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.scene.text.Font;

public class HelloLines extends Application
{
 private Line[] lines;

 public HelloLines()
 {
  lines = new Line[] { createLine(20, 20, 320, 120, 3, Color.RED),
                       createLine(320, 120, 120, 220, 5, Color.GREEN),
                       createLine(120, 220, 20, 20, 1, Color.BLUE) };
 }

 private Line createLine(double x1, double y1, double x2, double y2,
  double width, Color color)
 {
```

```
 Line line = new Line(x1, y1, x2, y2);
 line.setStroke(color);
 line.setStrokeWidth(width);
 return line;
}

private Text createText()
{
 Text text = new Text("Hello");
 text.setX(200);
 text.setY(50);
 text.setFont(new Font(36));
 return text;
}

@Override
public void start(Stage stage)
{
 Group root = new Group(lines);
 root.getChildren().add(createText());
 Scene scene = new Scene(root, 350, 250);
```

```
  stage.setTitle("Triangle");
  stage.setScene(scene);
  stage.show();
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

When you see the code, it is easy enough to understand and looks like the first example. The class defines an array of *Line* elements (a *Line* object represents a straight line), and the array is created in the constructor so that it contains three *Line* objects. A *Line* object is created using the method *createLine()*, in which you should especially note how an object is initialized and what types are used. There is also a method *createText()*, which creates and returns a *Text* object. If you consider the types used to create *Line* and *Text* objects, it is not the same types that you know from *Java2D*, but they have the same names and basically have the same characteristics. The scene graph is created in the same way as in the previous example in the method *start()*, but this time the type of the *root* element is *Group*, which is the basic type for a branch node. You should especially note how with *getChildren()* to get a list with all child nodes.

The example is simple, but you should note how geometric objects and text elements are included in the scene graph as nodes in the same way as other nodes such like *Button*, *Label*, and more.

2 ARCHITECTURE OF JAVAFX

JavaFX is a full API for creating GUI applications and consists of classes distributed on the following packages:

1. *javafx.animation*
2. *javafx.application*
3. *javafx.css*
4. *javafx.event*
5. *javafx.geometry*
6. *javafx.stage*
7. *javafx.scene*

where the latter has more sub packages. A JavaFX program consists as mentioned of a scene graph, which is a tree containing nodes that may be

- geometrical objects, that can be 2D and 3D objects, as *Circle*'s, *Sphere*'s and so on
- UI controls, such as *Label*, *Button* and so on
- Containers, as *StackPane*, *VBox*, *Group* and more
- Media elements, for images, audio and video

Each node in a scene graph has a unique parent, except the root, which has no parent, and all nodes that do not have child nodes are called as mentioned for leaf nodes, while all other nodes are called branch nodes. The same node can only occur in the tree once, and each node may have defined effects, opacity, transforms, and event handlers.

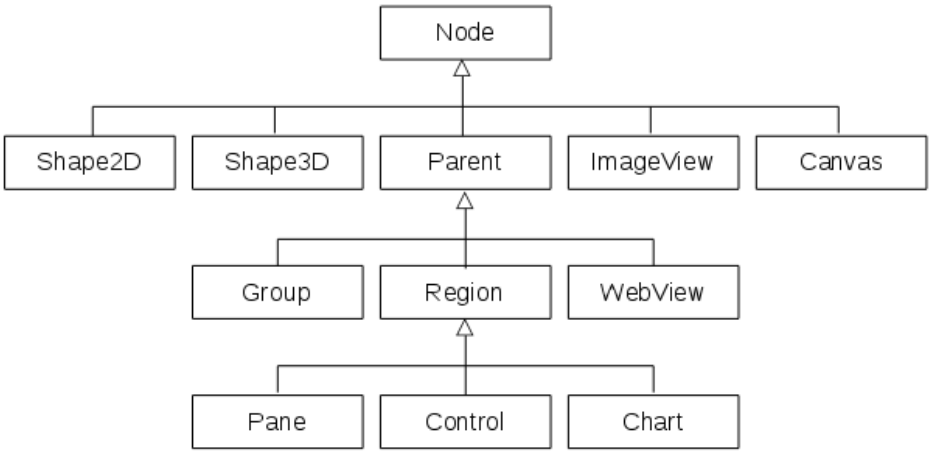
To render the graphics JavaFX uses *openGL* (on Linux systems) and, as far as possible, use the graphics card's hardware accelerator with the result that JavaFX is highly effective in rendering graphics.

As mentioned above, a JavaFX application is represented by a *Stage* object that contains all of the application's objects, and the *Stage* object is created by the actual platform. A *Stage* object has a width and a height in addition to the content (the scene graph), and there are decorations as a *title line* and a *frame*. There are the following kind of *Stage* objects:

- *Decorated*
- *Undecorated*
- *Transparent*
- *Unified*
- *Utility*

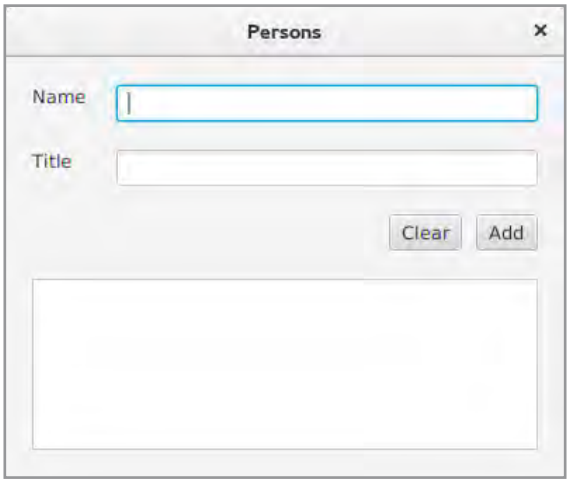
and the individual objects are discussed continuously in connection with the examples.

The *Stage* object's content is represented by a *Scene* object, which defines the width and height as well as the scene graph. The individual nodes are discussed in connection with the examples, and there are many with each there purpose. Among other things, there are different branch nodes, where the simplest is a *Group* node, which is primarily a collection of child nodes. Other examples are *Pane* nodes with several specific derivative classes, which play the same role as layout managers in Swing. All the specific nodes are part of a hierarchy of class's and are derived from one of the following classes:



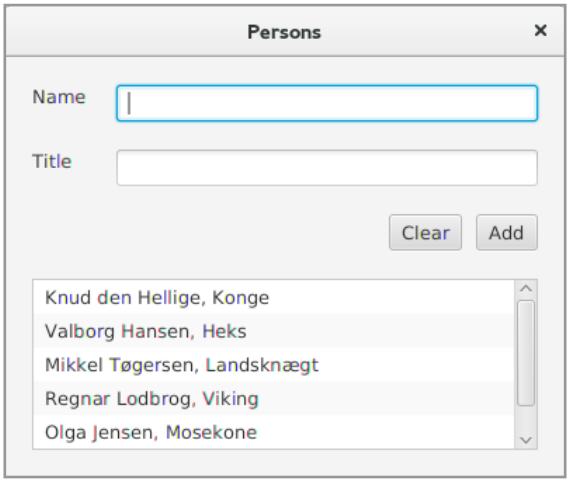
EXERSICE 1

You must write a JavaFX application that opens the following window, where there are two entry fields at the top and at the bottom of a list box:



If you click the *Add* button, (and something has been entered in both fields), a line must be inserted in the list box where the name and title are separated by comma. If you click on the other button, the content of the list box must be deleted. It is a requirement that the size of the two input fields as well as the size of the list box follows the size of the window and that the buttons follow the edge of the window.

Below is the window after 6 people have entered. Note that it is a part of the task to find out which controls are to be used and how to place them in the window:



3 2D SHAPES

In this chapter I will review how to draw geometric shapes in a window, and as may be seen in the last example of chapter 1, it is almost the same as in Java2D, only other classes are used. A geometric shape is an object of the type *Shape2D*, and thus an object that is a node and can be included in a scene graph. Generally, there are the following predefined shapes (nodes):

- *Line*
- *Rectangle*
- *Circle*
- *Ellipse*
- *Polygon*
- *Polyline*
- *CubicCurve*
- *QuadCurve*
- *Arc*

In addition, you can define general curves as *Path* objects consisting of segments of the types:

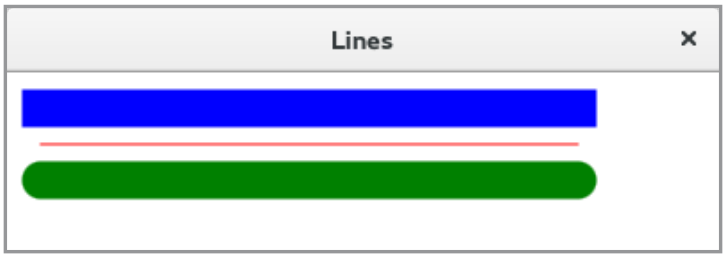
- *Line*
- *CubicCurve*
- *QuadCurve*
- *Arc*

and finally you can define a shape like a *SVG Path*.

The aim of this chapter is to show how to create and apply the above geometric shapes, but instead of reviewing the syntax (which is a lot about what you know from Java2D), I will instead formulate a series of exercises where the challenge is primarily to find out of what the types are called JavaFX.

EXERCISE 2

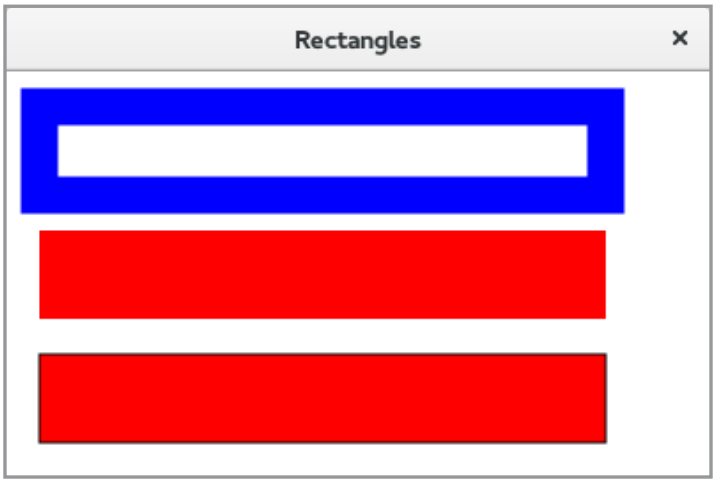
Write a JavaFX application that opens the following window:



The blue line has the endpoints (20, 20) and (320, 20) and width 21, while the red line has the endpoints (20, 40) and (320, 40) and width 1. Finally, the green line has the endpoints (20, 60) and (320, 60) and width 21.

EXERCISE 3

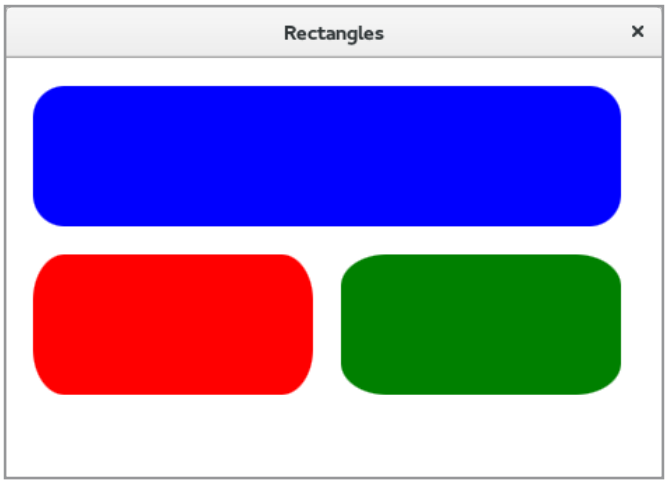
Write a JavaFX program that opens the following window:



which shows three rectangles, each of which has a size of 300 × 50. The left upper corner of the three rectangles are (20, 20), (20, 90) and (20, 160) respectively. The top has a blue frame with a width of 21, but an empty inner. The middle has no frame, while the latter has a black frame with the width 1.

EXERCISE 4

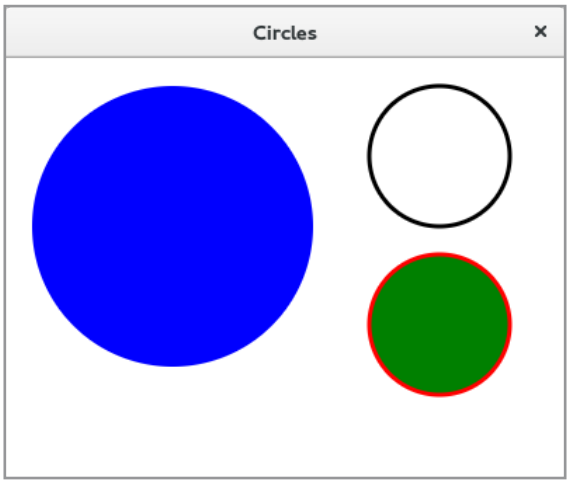
Write a program that shows a window with three rectangles with rounded corners:



Note that JavaFX does not have a type for rectangles with rounded corners, and the shapes are of the type *Rectangle*.

EXERCISE 5

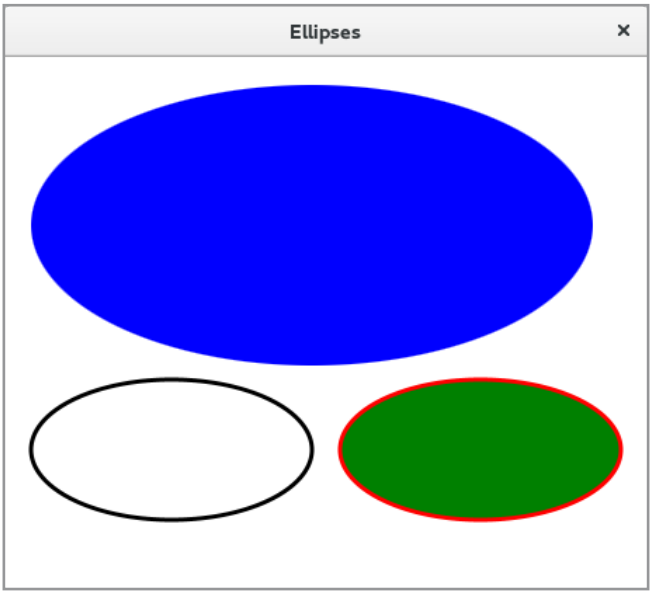
Write a JavaFX program that opens the following window:



when the type of the shapes must be *Circle*.

EXERCISE 6

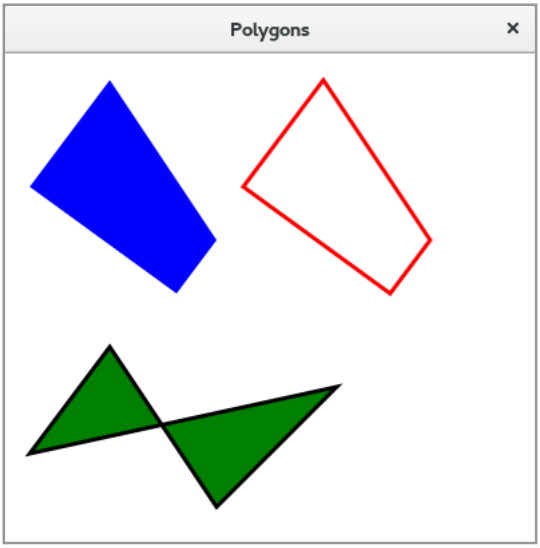
Write a program that opens the following window:



where the type of the three shapes are *Ellipse*.

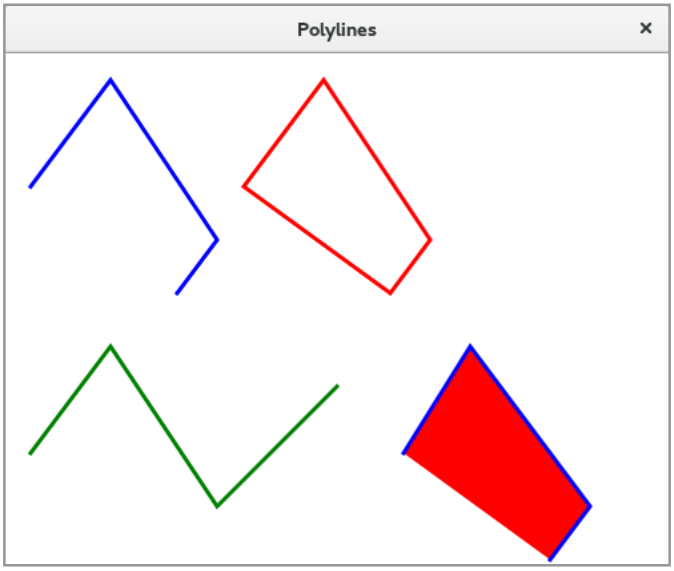
EXERCISE 7

A polygon is a closed shape bounded by straight lines, and you defines a polygon by specifying the coordinates of the corners. Write a JavaFX program that opens the following window, which shows three polygons:



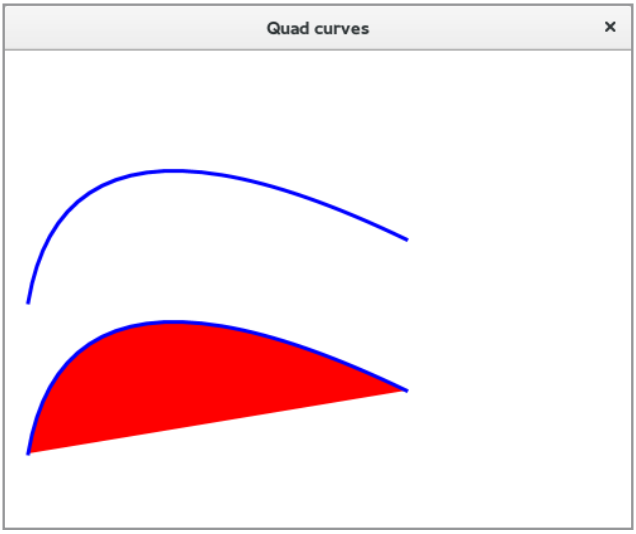
EXERCISE 8

A polyline is defined as a polygon and consists of straight lines. The shape is defined in the same way as a sequence of corners, but there is no need to be a coherent shape. Generally, a polyline is not a filled shape, but if you define a fill color, the shape is filled as a polygon. In this exercise, you must write a program that opens the following window with 4 polyline shapes:



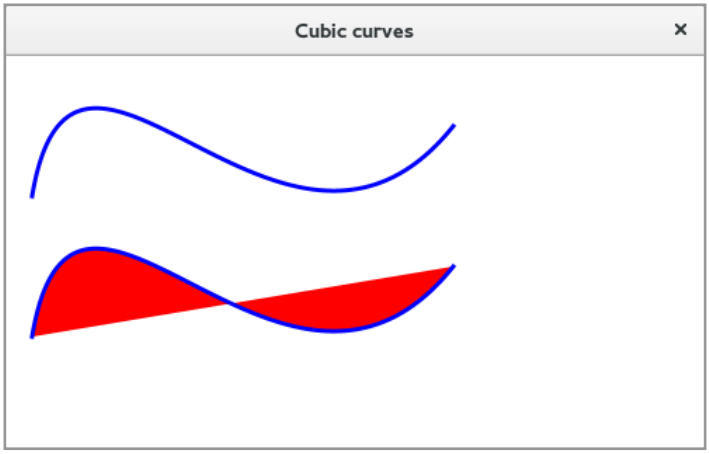
EXERCISE 9

You must write a program that shows two *quad curves*. If you do not remember the definition of a *quad curve*, you might want to read about the curve in the book Java 10. The program must open the following window:



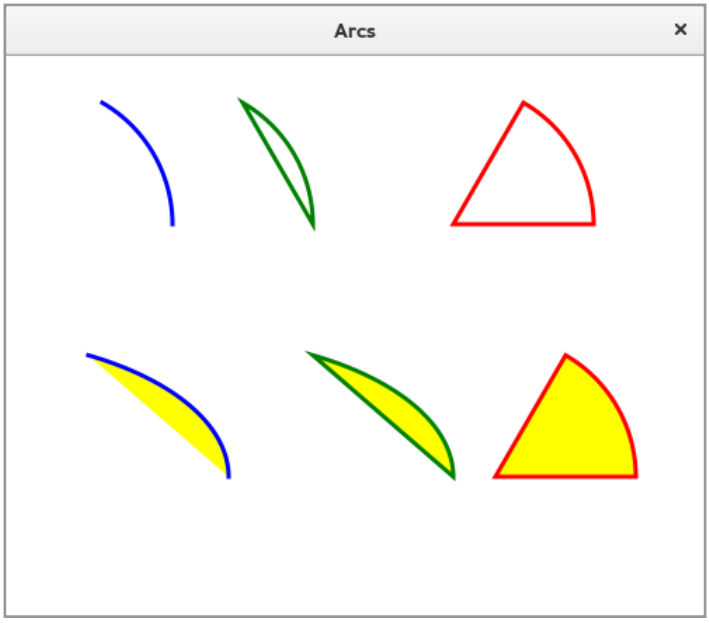
EXERCISE 10

You must write a program that shows two *cubic curves*. If you do not remember the definition of a *cubic curve*, you might want to read about the curve in the book Java 10. The program must open the following window:



EXERCISE 11

Write a program that opens the window below, showing 6 circle/ellipse slices. All slices start at 0 and span over an angle of 60 degrees. The top three are circle sections from a circle with radius 100, while the three bottom are slices of an ellipse with radiuses 200 and 100:



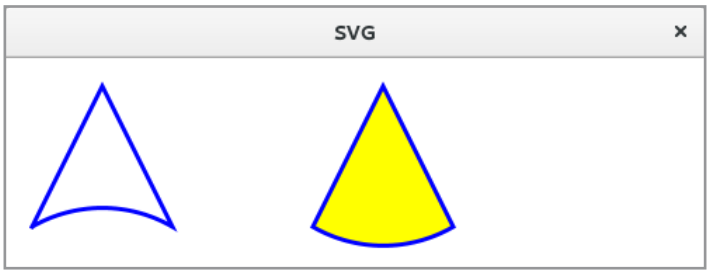
3.1 SVG

SVG stands for *Scalable Vector Graphics*, and in relation to JavaFX, you define a shape in the form of a path as a string containing commands. There are the following commands where the cursor denotes the drawing tool:

1. *M* or *m*, which means *move to*. As a result, the cursor is moved to the following coordinate. Is the command *M*, the coordinates designate an absolute position, and is the command *m*, the coordinates denote a relative position. If you indicate more coordinates, they are perceived as multiple subsequent move to operations.
2. *Z* or *z*, which means *close* and closes the path with a straight line from the current position of the cursor to the start of the curve.
3. *L* or *1*, which means *line to*. As a result, the cursor draws a straight line from the current point to the subsequent coordinate. If the command is *L*, the coordinates denote an absolute position and is the command *l*, the coordinates denote a relative position. If you indicate more coordinates, they are perceived as several subsequent line to operations.
4. *H* or *h*, which means *horizontal to*. As a result, the cursor draws a horizontal line from the current point to the following x-coordinate. Is the command *H*, the coordinates designate an absolute position and is the command *h*, the coordinates denote a relative position. If you specify multiple x coordinates, they are perceived as multiple subsequent horizontal to operations.
5. *V* or *v*, which means *vertical to*. As a result, the cursor draws a vertical line from the current point to the following y coordinate. If the command is *V*, the coordinates denote an absolute position and is the command *v*, the coordinate denotes a relative position. If you specify more y coordinates, they are perceived as multiple subsequent vertical to operations.
6. *C* or *c*, which means *cubic to*. As a result, the cursor draws a cubic curve from the current point to the last of the following three coordinate pairs. The first two are control points. Is the command *C*, the coordinates designate an absolute position, and is command *c*, the coordinates denote a relative position. If you specify more coordinate sets, they are perceived as several subsequent curve to operations.
7. *Q* or *q*, which means *quad to*. As a result, the cursor draws a quad curve from the current point to the last of the following two coordinate pairs. The first is the control point. Is the command *Q*, the coordinates designate an absolute position and is the command *q*, the coordinates denote a relative position. If you specify more coordinate sets, they are perceived as several subsequent quad to operations.
8. *A* or *a*, which means *arc to*. The result is that the cursor draws an arc starting in the current point to (x, y). The parameters are (*rx row rotation large sweep x y*), where *rx* and *row* are radii in an ellipse and rotation is the ellipse rotation about the x-axis. The two parameters *large* and *sweep* indicate which of the two arcs to be selected,

and the center of the ellipse is calculated automatically from the parameters. Is the command *A*, the coordinates designate an absolute position and is the command *a*, the coordinates denote a relative position. If you specify more parameter sets, they are perceived as several subsequent arc to operations.

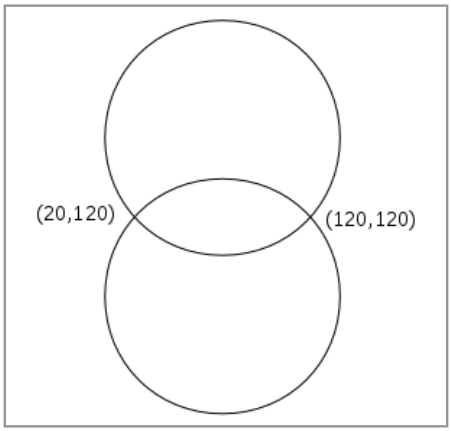
As an example, below is shown an application that opens the following window:



which shows two shapes, both of which have the type *SVGPath*. Both shapes are drawn with two *line to* commands and an *arc to* command. The code is as follows:

```
package svgprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.shape.*;
import javafx.scene.paint.*;

public class SVGProgram extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  Group root = new Group();
  root.getChildren().add(path1());
  root.getChildren().add(path2());
  Scene scene = new Scene(root, 500, 150);
  primaryStage.setTitle("SVG");
  primaryStage.setScene(scene);
  primaryStage.show();
 }

 private SVGPath path1()
 {
  SVGPath path = new SVGPath();
  path.setContent("M 20 120 L 70 20 120 120 A 100 100 0 0 0 20 120");
  path.setStroke(Color.BLUE);
  path.setStrokeWidth(3);
  path.setFill(null);
  return path;
 }

 private SVGPath path2()
 {
  SVGPath path = new SVGPath();
  path.setContent("M 220 120 L 270 20 320 120 A 100 100 0 0 1 220 120");
  path.setStroke(Color.BLUE);
  path.setStrokeWidth(3);
  path.setFill(Color.YELLOW);
  return path;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

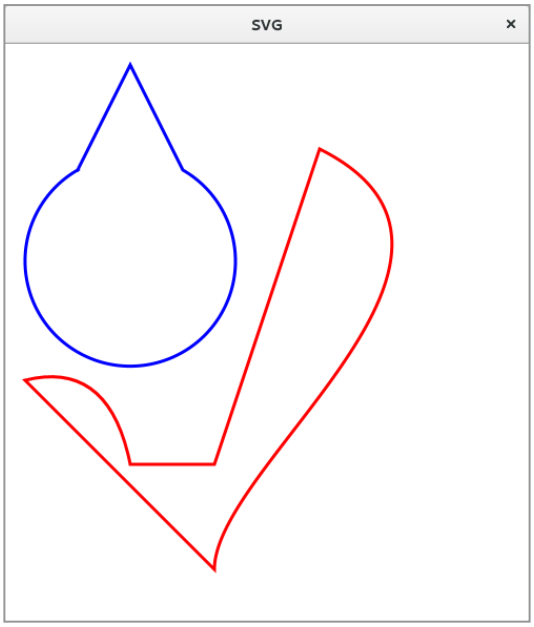
Here is the interesting of course the two SVG commands. You must note the syntax and how the commands are simply defined as a list separated by spaces. The only thing that is not obvious is command *A*, which defines an arc. In this case, it is (in the first case) a curve from (120, 120) to (20, 120) in a circle of radius 100 and it offers two options:



Which one chooses is determined by the two parameters *large* and *sweep* (whose values are 0 or 1).

EXERCISE 12

Write a program that opens the following window, which shows two *SVGPath* shapes where the blue curve consists of a *move to*, two *line to* and an *arc to* operation, while the red consists of one *quad to*, two *line to*, one *cubic to* and a *close* operation:



3.2 A PATH

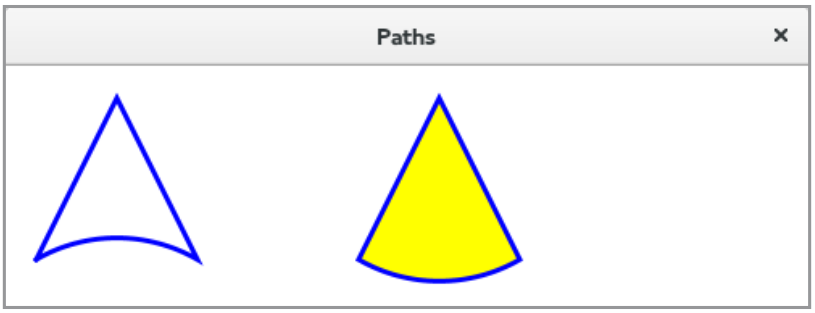
Instead of drawing a path as shown above, where you defines the drawing operations as a string, you can add drawing objects to a *Path* object that represents a collection of drawing objects. The advantage is primarily that the compiler can check the syntax and that any errors are thus detected before runtime. There are the following drawing objects:

- *MoveTo*
- *LineTo*
- *HLineTo*
- *VLineTo*
- *ArcTo*
- *QuadCurveTo*
- *CubicCurveTo*

For example, the following program draws the same window as shown in the previous example:

```
package pathprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.shape.*;
import javafx.scene.paint.*;

public class PathProgram extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  Group root = new Group();
  root.getChildren().add(path1());
  root.getChildren().add(path2());
  Scene scene = new Scene(root, 500, 150);
  primaryStage.setTitle("Paths");
  primaryStage.setScene(scene);
  primaryStage.show();
 }

 private Path path1()
 {
  Path path = new Path();
  path.getElements().add(new MoveTo(20, 120));
  path.getElements().addAll(new LineTo(70, 20), new LineTo(120, 120));
  path.getElements().add(new ArcTo(100, 100, 0, 20, 120, false, false));
  path.setStroke(Color.BLUE);
  path.setStrokeWidth(3);
  path.setFill(null);
  return path;
 }

 private Path path2()
 {
  Path path = new Path();
  path.getElements().addAll(new MoveTo(220, 120), new LineTo(270, 20),
   new LineTo(320, 120), new ArcTo(100, 100, 0, 220, 120, false, true));
  path.setStroke(Color.BLUE);
  path.setStrokeWidth(3);
  path.setFill(Color.YELLOW);
  return path;
 }
```

```
 public static void main(String[] args)
 {
  launch(args);
 }
}
```



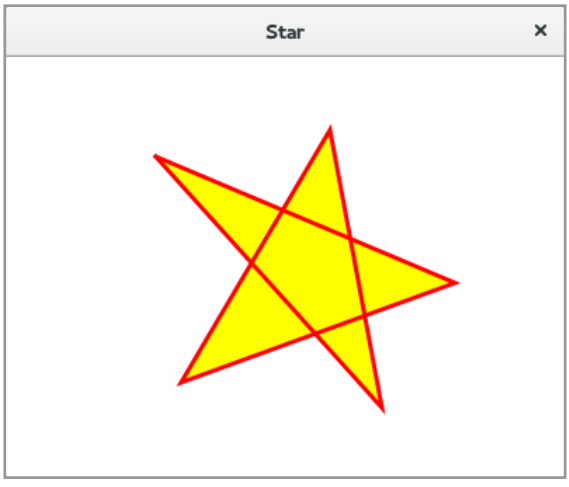
In fact, there is not much to explain, but you should notice how to create a *Path* object and how to add drawing objects. Also note how to create each drawing object, and that the objects are used in the order as they are added to the *Path* object.

EXERCISE 13

You must write a program that opens the same window as in exercise 12, but this time the type of the two shapes must be a *Path*.

EXERCISE 14

Write a program that opens the following window when the shape should be a *Path* object:



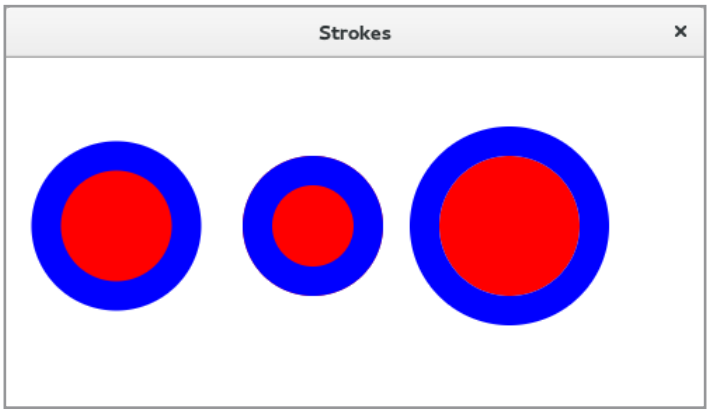
3.3 SHAPE PROPERTIES

The above has primarily concerned the shape of figures like straight lines, rectangles, circles, and more, but there are also some other properties attached to shapes, including how they are filled and how the perimeter is drawn. This section will discuss the most important of these features.

The perimeter – which is a line / curve – can be drawn in three ways:

1. *StrokeType.CENTERED*, which is default and means the perimeter is drawn so that the edge of the shape is exactly the center of the perimeter line.
2. *StrokeType.INSIDE*, where the perimeter is drawn as it falls within the shape.
3. *StrokeType.OUTSIDE*, where the perimeter is drawn outside the shape.

The program *StrokeProgram* opens the following window, which illustrates the three ways to draw the perimeter and where the first is the default:



The code is the following, where I have only shown the method *start()*:

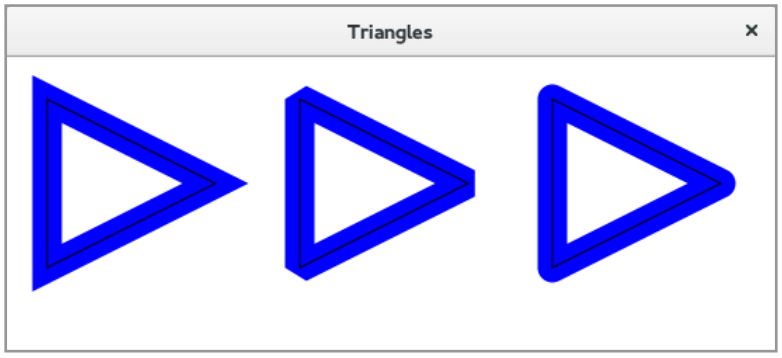
```
public void start(Stage primaryStage)
{
 Group root = new Group();
 root.getChildren().add(circ1());
 root.getChildren().add(circ2());
 root.getChildren().add(circ3());
 Scene scene = new Scene(root, 500, 250);
 primaryStage.setTitle("Strokes");
 primaryStage.setScene(scene);
 primaryStage.show();
}

private Circle circ1()
{
 Circle circ = new Circle(80, 120, 50, Color.RED);
 circ.setStroke(Color.BLUE);
 circ.setStrokeWidth(21);
 circ.setStrokeType(StrokeType.CENTERED);
 return circ;
}

private Circle circ2()
{
 Circle circ = new Circle(220, 120, 50, Color.RED);
 circ.setStroke(Color.BLUE);
 circ.setStrokeWidth(21);
 circ.setStrokeType(StrokeType.INSIDE);
 return circ;
}
```

```
private Circle circ3()
{
 Circle circ = new Circle(360, 120, 50, Color.RED);
 circ.setStroke(Color.BLUE);
 circ.setStrokeWidth(21);
 circ.setStrokeType(StrokeType.OUTSIDE);
 return circ;
}
```

Another thing is the endpoints of the line and here especially for shapes composed of straight lines, such as polygons. There are three options (as you also know from Java2D):

1. *StrokeLineJoin.MITTER* which is default
2. *StrokeLineJoin.BEVEL*
3. *StrokeLineJoin.ROUND*

You can find the exact explanation of how these endpoints are calculated in the book about Java2D. The application *JoinProgram* opens the following window, which shows the effect of each of the three options:




```
public void start(Stage primaryStage)
{
 Group root = new Group();
 root.getChildren().add(poly1());
 root.getChildren().add(poly2());
 root.getChildren().add(poly3());
 root.getChildren().add(poly4());
 root.getChildren().add(poly5());
 root.getChildren().add(poly6());
 Scene scene = new Scene(root, 550, 210);
 primaryStage.setTitle("Triangles");
 primaryStage.setScene(scene);
 primaryStage.show();
}
```

```
private Polygon poly1()
{
 Polygon poly = new Polygon(30, 30, 150, 90, 30, 150);
 poly.setStroke(Color.BLUE);
 poly.setStrokeWidth(21);
 poly.setFill(null);
 poly.setStrokeLineJoin(StrokeLineJoin.MITER);
 return poly;
}

private Polygon poly2()
{
 Polygon poly = new Polygon(210, 30, 330, 90, 210, 150);
 poly.setStroke(Color.BLUE);
 poly.setStrokeWidth(21);
 poly.setFill(null);
 poly.setStrokeLineJoin(StrokeLineJoin.BEVEL);
 return poly;
}
```

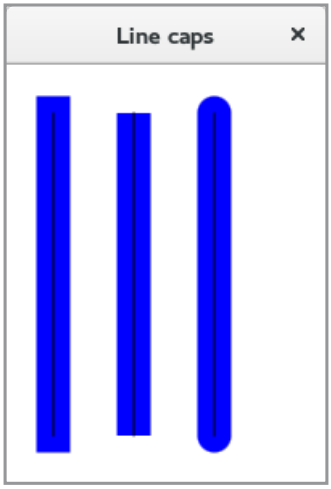
```
private Polygon poly3()
{
 Polygon poly = new Polygon(390, 30, 510, 90, 390, 150);
 poly.setStroke(Color.BLUE);
 poly.setStrokeWidth(21);
 poly.setFill(null);
 poly.setStrokeLineJoin(StrokeLineJoin.ROUND);
 return poly;
}

private Polygon poly4()
{
 Polygon poly = new Polygon(30, 30, 150, 90, 30, 150);
 poly.setStroke(Color.BLACK);
 poly.setStrokeWidth(1);
 poly.setFill(null);
 return poly;
}

private Polygon poly5()
{
 Polygon poly = new Polygon(210, 30, 330, 90, 210, 150);
 poly.setStroke(Color.BLACK);
 poly.setStrokeWidth(1);
 poly.setFill(null);
 return poly;
}

private Polygon poly6()
{
 Polygon poly = new Polygon(390, 30, 510, 90, 390, 150);
 poly.setStroke(Color.BLACK);
 poly.setStrokeWidth(1);
 poly.setFill(null);
 return poly;
}
```

There are similar options for specifying endpoints of lines (please check the book Java 10 for an explanation):

1. *StrokeLineCap.BUTT*
2. *StrokeLineCap.SQUARE*
3. *StrokeLineCap.ROUND*

EXERCISE 15

Write a program that you can call *LineCaps*. The program must open the following window, which shows 6 *Line* objects (each of which has a length of 200), and each endpoint (of the blue lines) is *BUTT*, *SQUARE* and *ROUND* respectively:

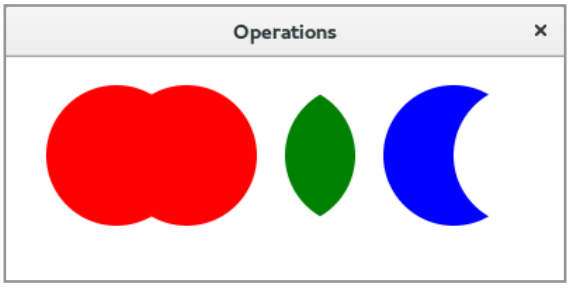


3.4 SHAPE OPERATIONS

In general, if you add two *Shape* objects to a *Group* and the two shapes overlap, then the shape added last will cover the part of the first shape that overlaps. The *Shape* class has three static methods

- *union()*
- *intersect()*
- *subtract()*

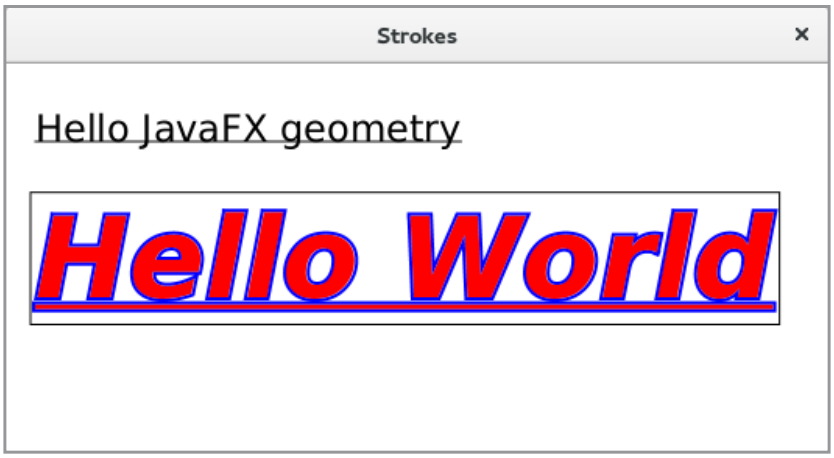
which all as parameters have two *Shape* objects and the methods combine these two objects into a single *Shape* object. The *OperationProgram* shows how to apply this operation and opens the following window:



```
public void start(Stage primaryStage)
{
 Group root = new Group();
 root.getChildren().add(shape1());
 root.getChildren().add(shape2());
 root.getChildren().add(shape3());
 Scene scene = new Scene(root, 400, 160);
 primaryStage.setTitle("Operations");
 primaryStage.setScene(scene);
 primaryStage.show();
}

private Shape shape1()
{
 Circle circ1 = new Circle(80, 70, 50);
 Circle circ2 = new Circle(130, 70, 50);
 Shape shape = Shape.union(circ1, circ2);
 shape.setFill(Color.RED);
 return shape;
}

private Shape shape2()
{
 Circle circ1 = new Circle(200, 70, 50);
 Circle circ2 = new Circle(250, 70, 50);
 Shape shape = Shape.intersect(circ1, circ2);
 shape.setFill(Color.GREEN);
 return shape;
}

private Shape shape3()
{
 Circle circ1 = new Circle(320, 70, 50);
 Circle circ2 = new Circle(370, 70, 50);
 Shape shape = Shape.subtract(circ1, circ2);
 shape.setFill(Color.BLUE);
 return shape;
}
```

4 TEXT

Text can be represented by a *Text* object, and the important thing is that *Text* inherits *Shape* and thus can be used as any other *Shape* object and enter the scene graph as a node. In reality, there is not much to explain, but the application *TextProgram* opens the following window:



It works almost the same way you know from Java2D, but other classes are used. The completed code is as follows:

It works almost the same way you know from Java2D, but other classes are used. The completed code is as follows:

```
package textprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;
import javafx.geometry.*;

public class TextProgram extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  Group root = new Group();
  Shape shape = text1();
  root.getChildren().add(shape);
  root.getChildren().add(line(shape));
  shape = text2();
  root.getChildren().add(shape);
  root.getChildren().add(rect(shape));
  Scene scene = new Scene(root, 530, 250);
  primaryStage.setTitle("Strokes");
  primaryStage.setScene(scene);
  primaryStage.show();
 }

 private Text text1()
 {
  Text text = new Text(20, 50, "Hello JavaFX geometry");
  text.setFont(Font.font("Liberation Sherif", FontWeight.NORMAL,
    FontPosture.REGULAR, 24));
  return text;
 }

 private Line line(Shape shape)
 {
  Bounds bounds = shape.getBoundsInLocal();
  return new Line(20, 50, 20 + bounds.getWidth(), 50);
 }
```

```
 private Text text2()
 {
  Text text = new Text("Hello World");
  text.setX(20);
  text.setY(150);
  text.setFont(Font.font("Liberation Sherif", FontWeight.BOLD,
   FontPosture.ITALIC, 72));
  text.setFill(Color.RED);
  text.setStroke(Color.BLUE);
  text.setStrokeWidth(2);
  text.setStrokeType(StrokeType.OUTSIDE);
  text.setUnderline(true);
  return text;
 }

 private Rectangle rect(Shape shape)
 {
  Bounds bounds = shape.getBoundsInLocal();
  Rectangle rect = new Rectangle(bounds.getMinX(), bounds.getMinY(),
   bounds.getWidth(), bounds.getHeight());
  rect.setFill(null);
  rect.setStroke(Color.BLACK);
  rect.setStrokeWidth(1);
  rect.setStrokeType(StrokeType.OUTSIDE);
  return rect;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The top text is created by the method *text1()* as a *Text* object. Here you should note that the constructor as parameters has the coordinates of the text and that the coordinates indicate the left end of the baseline (the line on which the text is "on"). Otherwise, you should especially note how to create a font. This happens with a static method *font()* in the class *Font* as well as two enumerations *FontWeight* and *FontPosture*.

The method *line()* has a *Shape* object as parameter, and the method creates a *Line* object that corresponds to the baseline of the text created in *text1()*. Here you should especially note the method *getBoundsInLocal()*, which returns a *Bounds* object (defined in the package *javafx.geometry*) that contains the *Shape* object width and height as well as its location, and thus the circumscribing rectangle. Regarding the latter, it is not a (x, y) value, but instead the smallest and largest x and y value that may occur, respectively.

The method *getBoundsInLocal()* returns the circumscribing rectangle before any transformation. There is also a method *getBoundsInParent()* that returns the circumscribing rectangle after a transformation. In this case, *Bounds* uses alone the object to determine the length of the line to be drawn.

The method *text2()* creates a *Text* object for the lower text. Since a *Text* object is also a *Shape* object, you can indicate both fill and stroke just like other shape objects. In this case, the location of the object is not defined by the constructor, but it is only to show that there are other options. In particular, note how to indicate that the text needs to be underlined.

Finally, there is method *rect()* that returns the circumscribing rectangle for the last text, and here you should primarily note how the rectangle is defined using a *Bound* object.

5 EFFECTS

A JavaFX program, as mentioned, represents the content of the user interface as nodes in a scene graph. It is possible to change how the individual nodes are displayed using so-called effects, where an effect is an object that is used to modify how a node is to be drawn. It is primarily interesting for nodes, which represent graphic objects like images and geometric figures. This chapter is a brief introduction to these effects. Basically, the following effects are represented as classes in the package *javafx.scene.effect*:

- *ColorAdjust*, which can be used to modify the color of an image where you can adjust the hue, saturation, brightness and contrast.
- *ColorInput*, which corresponds to drawing a rectangle, filled with a color.
- *ImageInput* used to fill a rectangular area with an image.
- *Blend*, which is used to combine pixels from two inputs.
- *Bloom*, which is used to provide a portion of a node more hue.
- *Glow*, which is used to increase the hue of a node.
- *BoxBlur*, used to perform a blur operation on a node with a box filter.
- *GausienBlur*, also used to perform a blur of a node, but instead using a kernel.
- *MotionBlur*, which is a gaussien blur, where there is also an angle.
- *Reflection* that performs a reflection of a node.
- *SepiaTone*, which tones a node with a brownish color.
- *Shadow*, resulting in a shadow effect by a blur of edges.
- *DropShadow*, where the shadow effect is created behind the node.
- *InnerShadow*, where the shadow effect is created within the edge of the node.
- *Lighting*, which is used to simulate a light effect, where you can indicate a light source such as point, distance or sport.
- *Light.Distant*, which is a variant of Lighting.
- *Light.Spot*, which is a variation of Lighting.
- *Point.Spot*, which is a variation of Lighting.

As an example I show a program that adjusts the colors of an image. The program is named *ColorAdjustProgram* and opens the following window:

where the left image is the original, while the right is the adjusted image. The program should show only the syntax for using a *ColorAdjust*, but in this case does not result in any improvement of the image. The program also shows how to load a picture from the local disk into a JavaFX program. The program code is as follows:

```
package coloradjustprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.effect.*;
import javafx.scene.image.*;

public class ColorAdjustProgram extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  String filename =
   "file:" + System.getProperty("user.home") + "/data/dollar.jpg";
  Image image = new Image(filename);
  Group root = new Group();
  root.getChildren().add(image1(image));
  root.getChildren().add(image2(image));
  Scene scene = new Scene(root, 460, 250);
  primaryStage.setTitle("ColorAdjust");
  primaryStage.setScene(scene);
  primaryStage.show();
 }
```

```
private ImageView image1(Image img)
{
 ImageView view = new ImageView(img);
 view.setX(20);
 view.setY(20);
 view.setFitWidth(200);
 view.setPreserveRatio(true);
 return view;
}

private ImageView image2(Image img)
{
 ImageView view = new ImageView(img);
 view.setX(240);
 view.setY(20);
 view.setFitWidth(200);
 view.setPreserveRatio(true);
 ColorAdjust colorAdjust = new ColorAdjust();
 colorAdjust.setContrast(0.5);
 colorAdjust.setHue(0.5);
 colorAdjust.setBrightness(-0.5);
```

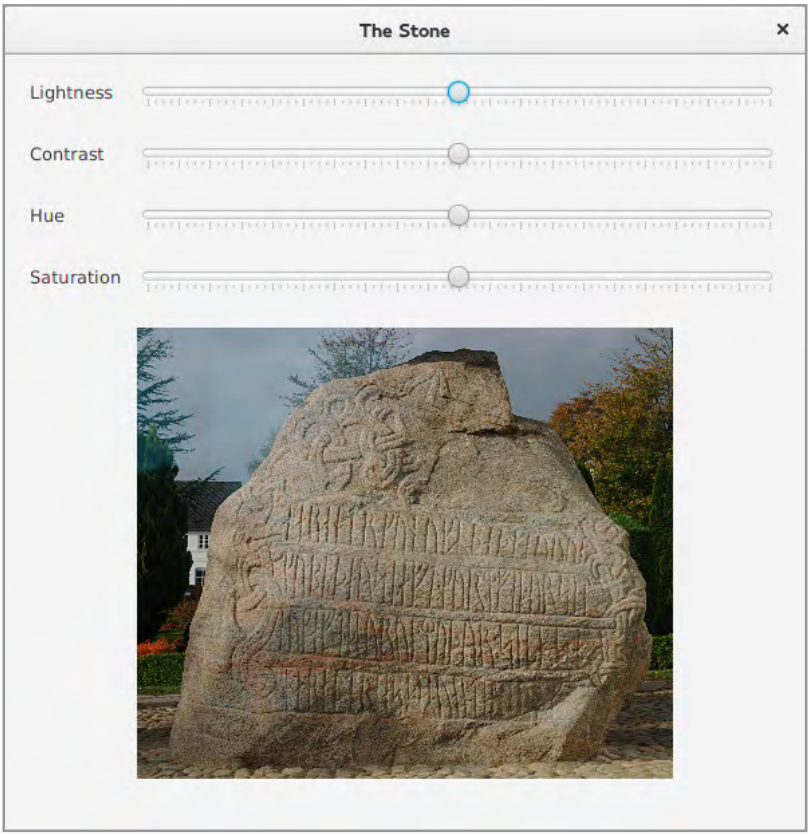
```
  colorAdjust.setSaturation(0.5);
  view.setEffect(colorAdjust);
  return view;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The image is loaded in the method *start()* and is usually done by defining a path to the image, which is a *jpg* image. In this case, the image is placed in a directory *data* under my home directory, but the image is also part of the book's zip file with examples. The only difference with regard to the file name is that it starts with the name of the protocol as the image is located in the local file system. In other cases, it would be the *http* protocol. The image is loaded by creating an *Image* object.

An image is inserted into the program's scene graph as an *ImageView* object, and the method *image1()* creates an *ImageView* object for the original image and specifies the coordinates of the image's upper left corner. In addition, you specify the width of the image, and that its propositions can not be changed.

The method *image2()* correspondingly creates another *ImageView* object for the same image, but this time with other coordinates. In addition, a *ColorAdjust* object is created that has methods for adjusting lightness, contrast, saturation and tint. The parameters have values between -1 and 1, where a negative value means reducing the property while a positive value means increasing the property.

PROBLEM 1



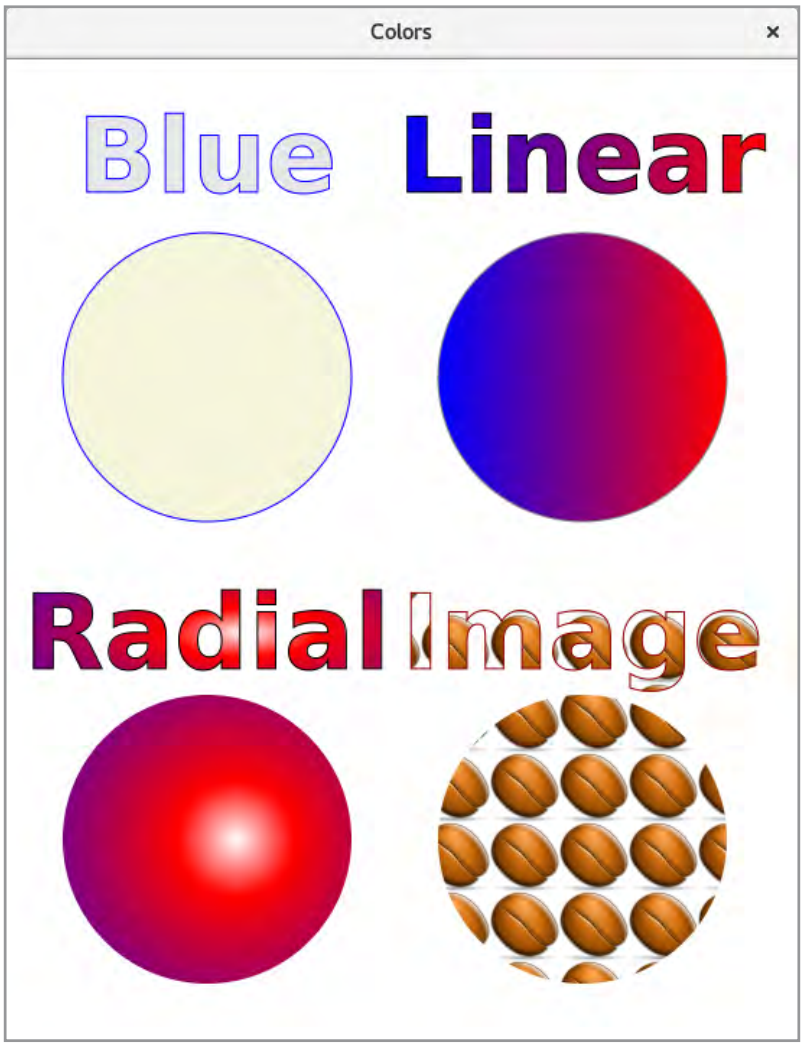
Write a program that opens the window above. The goal is that you can experiment with the four properties: Light, contrast, hue and saturation. The picture is called *stone.jpg* and is contained in the book's zip file. You must be able to adjust the four properties using the *Slider* controls at the top of the window. This means that you should handle the event that is firing when moving the slider. This happens with a *ChangedListener*, which is defined in the *javafx.beans.value* package.

5.1 COLORS

In the above examples (and also the exercises from the previous chapters) I have used colors and shown how to fill a shape with a color and draw the perimeter. Generally there are four options:

1. a uniform color and then objects of the class *Color*
2. a linear gradiant paint and then objects of the class *LinearGradian*
3. a radial gradiant paint and then objects of the class *RadialGradiant*
4. an image and then objects of the class *ImagePattern*

The program *ColorProgram* opens the following window:



The program's scene graph has 8 nodes, which are 4 *Text* nodes and 4 *Circle* nodes, respectively. A text node is created using the following method:

```
private Text text(String txt, double x, double y, Paint fill, Paint stroke)
{
 Text text = new Text(txt);
 text.setFont(
  Font.font("Liberation Sherif", FontWeight.BOLD, FontPosture.REGULAR, 72));
 text.setFill(fill);
 text.setStroke(stroke);
 text.setStrokeWidth(1);
 text.setX(x - text.getBoundsInParent().getWidth() / 2);
 text.setY(y + text.getBoundsInParent().getHeight() / 2);
 return text;
}
```

The parameters are the text, the coordinates of the shape's position and the colors to fill the figure and draw the perimeter respectively. You should note that a color may be one of the above four options, each of which has the type *Paint* as the base type.

A circle is created using the following method:

```
private Circle circle(double x, double y, Paint fill, Paint stroke)
{
 Circle circ = new Circle(x, y, 100, fill);
 if (stroke != null)
 {
  circ.setStroke(stroke);
  circ.setStrokeWidth(1);
 }
 return circ;
}
```

Here the parameters means the same as for a text node.

The window shows four figures, each character consisting of a *Text* node and a *Circle* node. The first is drawn using uniform colors, where there is nothing to say in relation to what

has already been said about colors. The next figure uses a linear gradient paint, which is defined as follows:

```
private Paint createLinear()
{
 Stop[] stops = new Stop[] { new Stop(0, Color.BLUE), new Stop(1, Color.RED)};
 return new LinearGradient(0, 0, 1, 0, true, CycleMethod.NO_CYCLE, stops);
}
```

A linear gradient paint is defined by two or more colors, which are defined as objects of the type *Stop*. Each stop indicates a color and a position that can either be relative or absolute. The first color stop is in this case blue and the position is 0, which means that the color is the start color. The other color is red and the position is 1, which means that the last color value must be red. All colors between these two extremes are blends of blue and red.

The *LinearGradient* parameters are the starting position (here (0.0)) and the final position (here (1.0)), and the next parameter indicates that the coordinates must be interpreted relative (*true*) or absolute. In this case, the coordinates indicate that they must be mixed along the x axis from the left edge of the shape to the right edge of the shape. The next last parameter indicates whether the pattern should be repeated if the coordinates do not define the entire shape, and finally the last parameter is the color stops.

A radial gradient paint is defined in almost the same way:

```
private Paint createRadial()
{
 Stop[] stops = new Stop[]
  { new Stop(0, Color.WHITE), new Stop(0.2, Color.
RED), new Stop(1, Color.BLUE) };
 return new RadialGradient(0, 0, 0.6, 0.5, 1, true, CycleMethod.NO_CYCLE, stops);
}
```

Again, two or more color stops must be defined, and in this case three are defined. They say that starting with a white color, and after 20% the color should be red for the last to be blue. Parameters for a *RadialGradient* begins by defining a starting point, which is the focus point and is the first stop. The first parameter is the angle of this focus point, while the next is the distance to the focus point. The next two parameters are the coordinates of the focus point, while the next is the radius of a circle that defines the area that the *Paint* object is about. The last three parameters means the same as for a *LinearGradient*.

Finally, there is an *ImagePattern*, which is a *Paint* object that draws an image. In the program's *start()* method an image (*Bean.png*) is loaded as an *Image* object. This image can then be used to create an *ImagePattern* object:

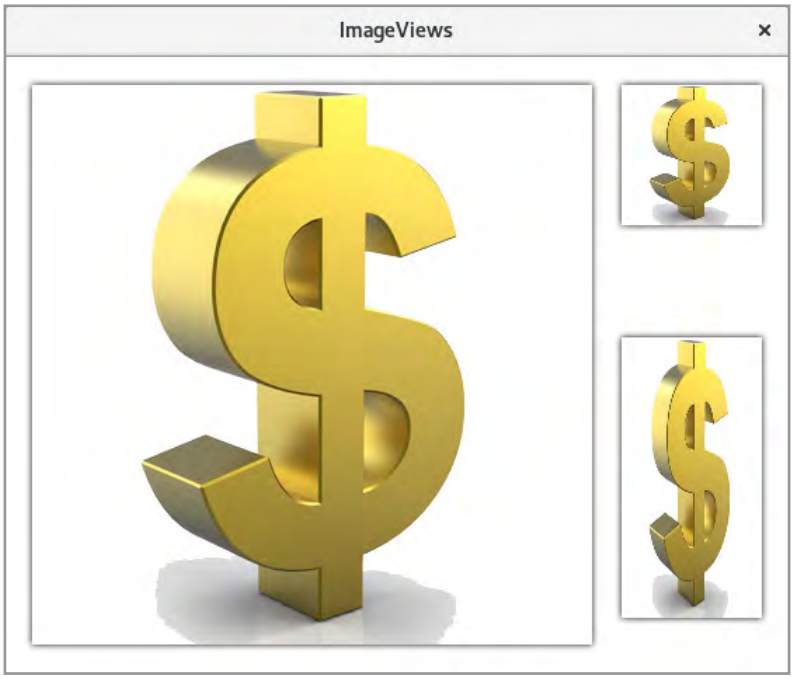
```
private Paint createImage()
{
 return new ImagePattern(image, 0, 0, 48, 48, false);
}
```

In addition to the image, the parameters are the starting coordinate of the figure (upper left corner), the width and height of the figure (the figure is scaled to this size), and whether the coordinates should be interpreted relative or absolute.

5.2 IMAGES

In the previous examples, I have already shown how to load an image and display it in a window. Actually, pictures do not have much to do with effects, but some are pictures used in the example above, and there are a few more remarks to add, so therefore a few additional examples.

An image is displayed in a scene graph as an *ImageView* node, and the image itself is represented in the program as an *Image* object and can be loaded from the machine's hard disk by specifying the path of the image as a parameter to the constructor. The following program is called *ImageViewProgram* and it opens the following window:



It is the same image shown three times, and the example should primarily show how to scale an image. The first image (the big one) is unscaled while the two others are scaled. The code is as follows:

```
package imageviewerprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.scene.effect.*;
import javafx.stage.Stage;
import javafx.scene.paint.*;
import javafx.scene.image.*;

public class ImageViewerProgram extends Application
{
 private Image image;

 public ImageViewerProgram()
 {
```

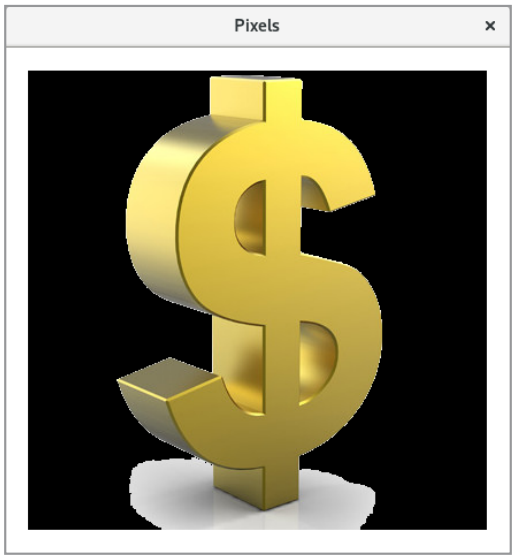
```
  String filename =
   "file:" + System.getProperty("user.home") + "/data/dollar.jpg";
  image = new Image(filename);
 }

 @Override
 public void start(Stage primaryStage)
 {
  Group root = new Group();
  root.getChildren().add(createView(20, 20, 0, 0, true));
  root.getChildren().add(createView(440, 20, 100, 200, true));
  root.getChildren().add(createView(440, 200, 100, 200, false));
  Scene scene = new Scene(root, 560, 440);
  primaryStage.setTitle("Colors");
  primaryStage.setScene(scene);
  primaryStage.show();
 }

 private ImageView createView(double x, double y, double width, double height,
  boolean preserve)
 {
  ImageView view = new ImageView(image);
  view.setX(x);
  view.setY(y);
  if (width > 0 && height > 0)
  {
   view.setFitHeight(height);
   view.setFitWidth(width);
  }
  view.setEffect(new DropShadow(5, Color.BLACK));
  view.setPreserveRatio(preserve);
  return view;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The image is loaded in the class' constructor and can then be referred to with the instance variable *image*. The method *createView()* creates an *ImageView* for the image, and it has five parameters, the first two being the coordinates of the *ImageView* object's upper left corner, while the two next indicate the width and height. The last indicates whether the image's propositions are to be retained. The statements of the method are easy to interpret, perhaps apart from the meaning of *setEffect()*, but the result is that a frame is drawn outside the

image. If no *width* or *height* is specified, these attributes are not set, and the result is that the physical dimensions of the image are applied.

Examining the method *start()* shows that there is no size for the first image. The image size therefore becomes the physical size, which is 400 × 400. For the next image, the size is set to 100 × 200, and it will be scaled within this frame, but as it is said that the propositions are to be saved, the result is an image that is scaled to ¼ in both directions and you gets a picture that is 100 × 100. For the last image, the same size is specified, but the propositions are not preserved, and the image is therefore scaled in both directions, so it will fill 100 × 200.

As the last example concerning images, I will show a program called *PixelProgram*. The program should show how to manipulate the individual pixels in an image. The starting point is the same picture as shown above. Here is a portion of the image's pixels white (the background), and the program should change the background to black. That is, all white pixels must be changed to black. In general, it is simple, but as it is a jpg image, pixels close to the figure will not be completely white, and it is therefore necessary to allow an uncertainty as to when a pixel is white. Performing the program gives you the result:




```
package pixelsprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.scene.effect.*;
import javafx.stage.Stage;
import javafx.scene.paint.*;
import javafx.scene.image.*;

public class PixelsProgram extends Application
{
 private Image image;

 public PixelsProgram()
 {
  String filename =
   "file:" + System.getProperty("user.home") + "/data/dollar.jpg";
  image = new Image(filename);
 }

 @Override
 public void start(Stage primaryStage)
 {
  Group root = new Group();
  root.getChildren().add(createImage());
  Scene scene = new Scene(root, 440, 440);
  primaryStage.setTitle("Pixels");
  primaryStage.setScene(scene);
  primaryStage.show();
 }
```

```
 private ImageView createImage()
 {
  int width = (int)image.getWidth();
  int height = (int)image.getHeight();
  WritableImage buffer = new WritableImage(width, height);
  PixelReader reader = image.getPixelReader();
  PixelWriter writer = buffer.getPixelWriter();
  for (int y = 0; y < height; ++y)
   for (int x = 0; x < width; ++x)
   {
    Color color = reader.getColor(x, y);
    if (isWhite(color)) color = Color.BLACK;
    writer.setColor(x, y, color);
   }
  ImageView view = new ImageView(buffer);
  view.setX(20);
  view.setY(20);
  return view;
 }

 private boolean isWhite(Color color)
 {
  double epsilon = 0.1;
  return 1 - color.getRed() < epsilon && 1 - color.getGreen() < epsilon &&
   1 - color.getBlue() < epsilon;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

Most importantly, of course, is the method *createImage()*, which returns an *ImageView* with the modified image. First, the width and height of the image are determined, and a memory representation of a picture of the same size is created. The type is *WritableImage* and I have called it *buffer*. Next, a *PixelReader* is defined for the original image, which means that you can refer to the individual pixels in the image. In addition, a *PixelWriter* is created for *buffer* that allows you to modify the buffer's pixels. Next, the program iterates using the reader over the pixels of the original image (line by line), and for each pixel, the corresponding pixel is updated in *buffer* – either modified or unchanged. Here you should note the method *isWhite()* which tests whether a pixel can be considered white.

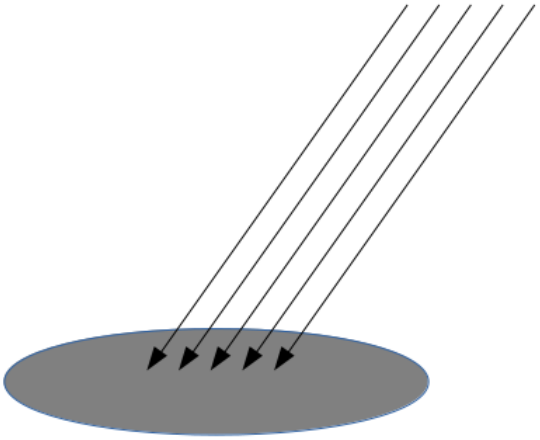
5.3 LIGHT

In the introduction to this chapter on effects, I have listed the different effects that you can apply, and I have far from illustrating them all (and will not), but I want to conclude this chapter with a little about the effect light. The principle is that a light source shines on a figure, and the color of the figure is determined by the nature of the light source and its position in relation to the figure.

Generally, you can associate a lighting effect to a figure with a *Lighting* object, and if you do not indicate a light source, a default light source is used, which corresponds to the light falling directly into the figure, resulting in a weak 3D effect towards the edge. You can also explicitly indicate a light source, where there are three options:

1. *Light.Distant*
2. *Light.Point*
3. *Ligth.Spot*

The first corresponds to the light source located at a distance, thus sending light that extends in one direction from a remote light source:

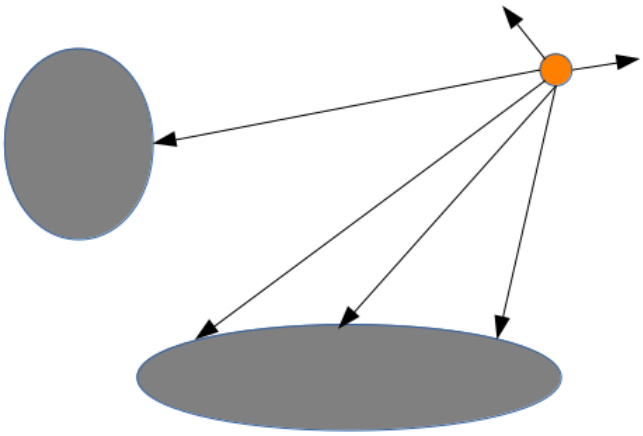


You can set as a parameter two values (both of the type *double*):

1. *azimuth*, which indicates the angle of the light source's rotation about the z axis
2. *elevation*, which indicates the angle of the light source's rotation about the x axis

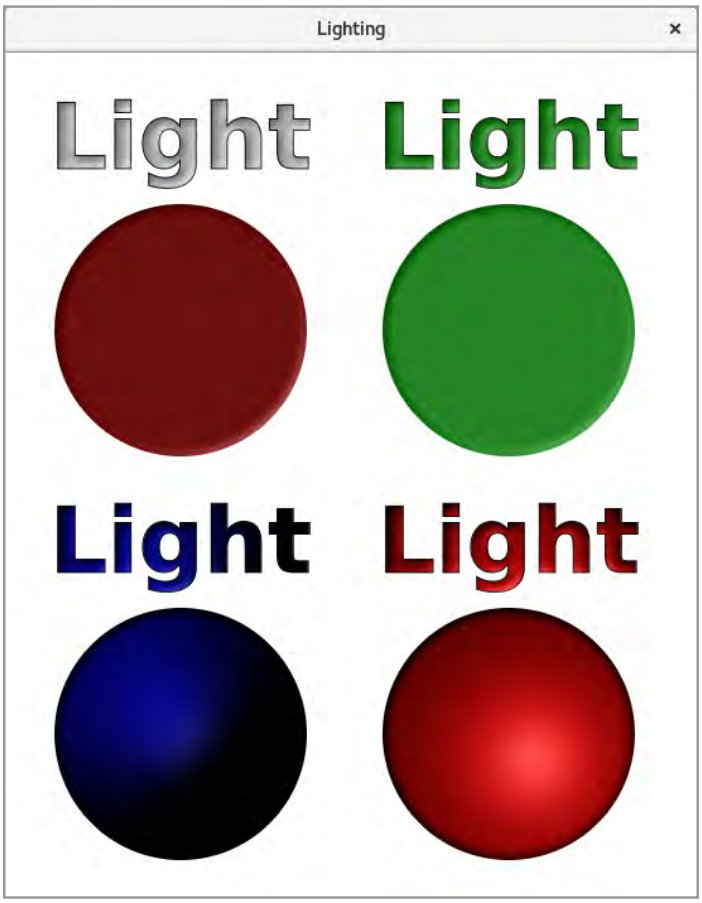
In addition, you can indicate the color of the light source.

The second option is a light source where the light is sent from a certain point, as indicated by 3 coordinates. The light spreads in all directions, and in the same way as a *Distant* light source, you can indicate the color of the light source.



The light intensity depends on the distance of the source to the figure. Finally, there is the third option that is similar to the above, but with the difference that you can specify a *specular* that controls the focus of the light source.

If you run the application *LightingProgram*, it opens the following window:



The example looks like *ColorProgram*, but instead of illustrating the effect of colors, the program shows the effect of light. The figures in the top row show the effect of a standard and a distant light source, respectively, while the two figures in the bottom row show the effect of a spot and a point light source, respectively. The code of the program is:

```
package lightingprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.effect.*;
import javafx.scene.shape.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class LightingProgram extends Application
{
 @Override
```

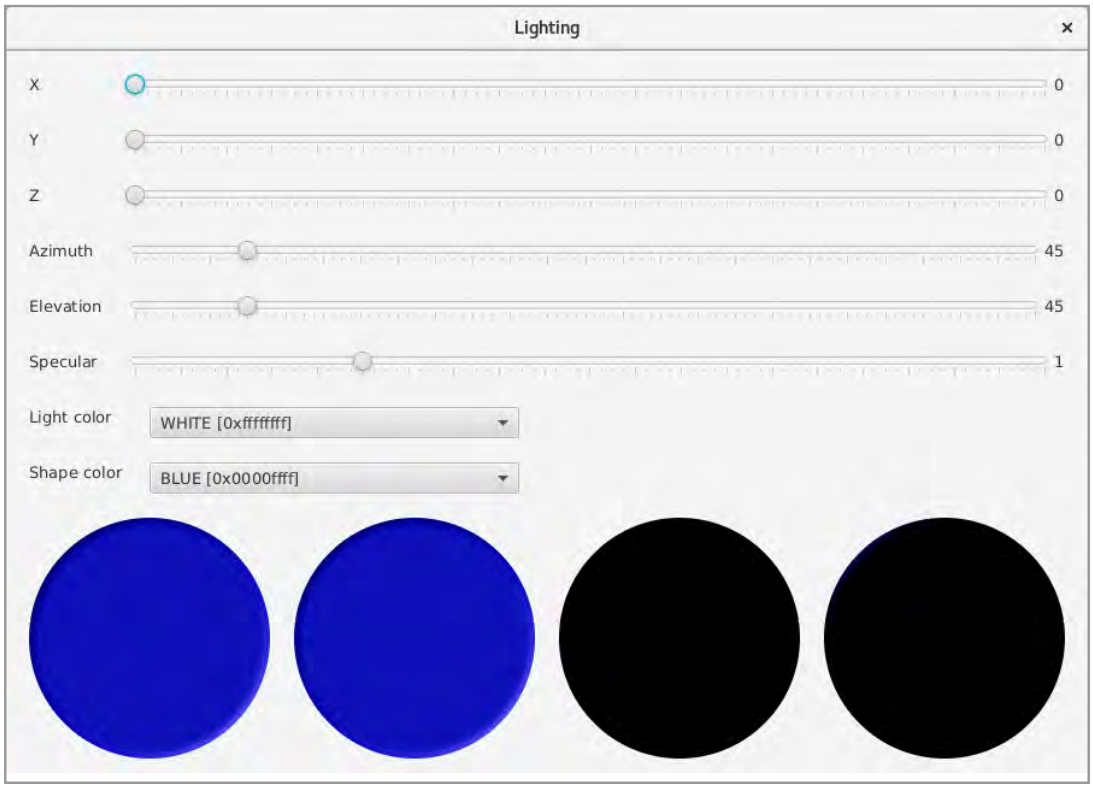
```
public void start(Stage primaryStage)
{
 Group root = new Group();
 root.getChildren().add(text(140, 50, Color.rgb(232, 232, 232),
  new Lighting()));
 root.getChildren().add(circle(140, 220, Color.DARKRED, new Lighting()));
 root.getChildren().add(text(400, 50, Color.GREEN,
  new Lighting(new Light.Distant(45, 60, Color.WHITE))));
 root.getChildren().add(circle(400, 220, Color.GREEN,
  new Lighting(new Light.Distant(45, 60, Color.WHITE))));
 root.getChildren().add(text(140, 370, Color.BLUE,
  new Lighting(new Light.Spot(120, 120, 50, 2, Color.WHITE))));
 root.getChildren().add(circle(140, 540, Color.BLUE,
  new Lighting(new Light.Spot(120, 120, 50, 2, Color.WHITE))));
 root.getChildren().add(text(400, 370, Color.RED,
  new Lighting(new Light.Point(120, 120, 50, Color.WHITE))));
 root.getChildren().add(circle(400, 540, Color.RED,
  new Lighting(new Light.Point(120, 120, 50, Color.WHITE))));
 Scene scene = new Scene(root, 550, 670);
 primaryStage.setTitle("Lighting");
 primaryStage.setScene(scene);
 primaryStage.show();
}
```

```
 private Text text(double x, double y, Paint fill, Lighting light)
 {
  Text text = new Text("Light");
  text.setFont(Font.font("Liberation Sherif", FontWeight.BOLD,
   FontPosture.REGULAR, 72));
  text.setFill(fill);
  text.setStroke(Color.BLACK);
  text.setStrokeWidth(1);
  text.setX(x - text.getBoundsInParent().getWidth() / 2);
  text.setY(y + text.getBoundsInParent().getHeight() / 2);
  text.setEffect(light);
  return text;
 }

 private Circle circle(double x, double y, Paint fill, Lighting light)
 {
  Circle circ = new Circle(x, y, 100, fill);
  circ.setEffect(light);
  return circ;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The two methods *text()* and *circle()* are similar to the corresponding methods from earlier. You should primarily consider the method *start()* and here how to create light sources as *Light* objects used as parameters for a *Lighting* object that is an effect. Note that the specific *Light* classes are defined as inner classes in the class *Light*.

PROBLEM 2

To experiment a little with light sources, write an application that opens the following window:



The four circles mentioned from the left are using the light sources:

1. default
2. distant
3. spot
4. point

The setting options are made with *Slider* controls and *ComboBox* controls, and the names should explain the meaning. A setting option does not necessarily have effect for all light sources, but changes a setting, the change must be reflected in the relevant shapes. As for *ComboBox* controls, they must be initialized with colors and I have enclosed a color in the following wrapper class:

```
class ColorWrapper
{
 private Color color;
 private String name;

 public ColorWrapper(Color color, String name)
 {
  this.color = color;
  this.name = name;
 }
```

```
 public Color getColor()
 {
  return color;
 }

 public String getName()
 {
  return name;
 }

 @Override
 public String toString()
 {
  return String.format("%s [%s]", name, color.toString());
 }
}
```

You can initialize the two *ComboBox* controls with an appropriate number of standard colors (static objects in the class *Color*), but you can actually decide all of them using the following method:

```
private java.util.List<ColorWrapper> getColors()
{
 java.util.List<ColorWrapper> list = new java.util.ArrayList();
 try
 {
  java.lang.reflect.Field[] fields = Color.class.getDeclaredFields();
  for (java.lang.reflect.Field field : fields)
  {
   if (java.lang.reflect.Modifier.isStatic(field.getModifiers()) &&
    field.getType().equals(Color.class))
   {
    Object obj = new Object();
    obj = field.get(obj);
    list.add(new ColorWrapper((Color)obj, field.getName()));
   }
  }
 }
 catch (Exception ex)
 {
  list.clear();
 }
 return list;
}
```

6 TRANSFORMATIONS

JavaFX also supports transformations of nodes in the scene graph, and it happens largely in the same way that you know it from Java2D. Generally, there are 4 transfomations

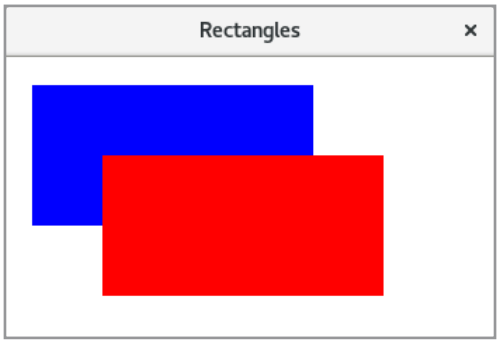
1. Translater
2. Scaling
3. Rotation
4. Shearing

and the meaning is the same as in Java2D. As a simple example, the following program performs a translation of a rectangle:

```
package translateprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.shape.*;
import javafx.scene.paint.Color;
import javafx.scene.transform.*;

public class TranslateProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  Group root = new Group();
  root.getChildren().add(createRect(Color.BLUE, null));
  root.getChildren().add(createRect(Color.RED, new Translate(50, 50)));
  Scene scene = new Scene(root, 350, 200);
  stage.setTitle("Rectangles");
  stage.setScene(scene);
  stage.show();
 }

 private Rectangle createRect(Color color, Translate trans)
 {
  Rectangle rect = new Rectangle(20, 20, 200, 100);
  rect.setFill(color);
  if (trans != null) rect.getTransforms().add(trans);
  return rect;
 }
```

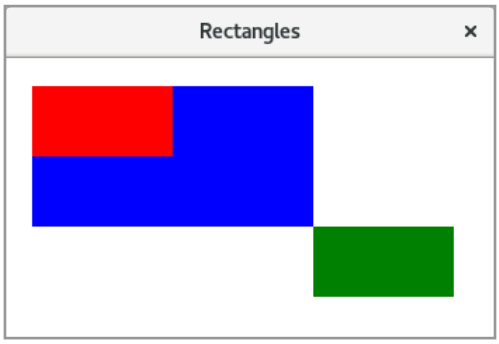
```
 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The method *createRect()* creates a *Rectangle* with a fixed position and size, as well as a parameter that is the color. Finally, there is a parameter that can indicate a transformation and thus a shift of the rectangle to another position. That is, the program creates two rectangles of the same size and position, but one (the red) translated 50 pxels in both directions:



EXERCISE 16

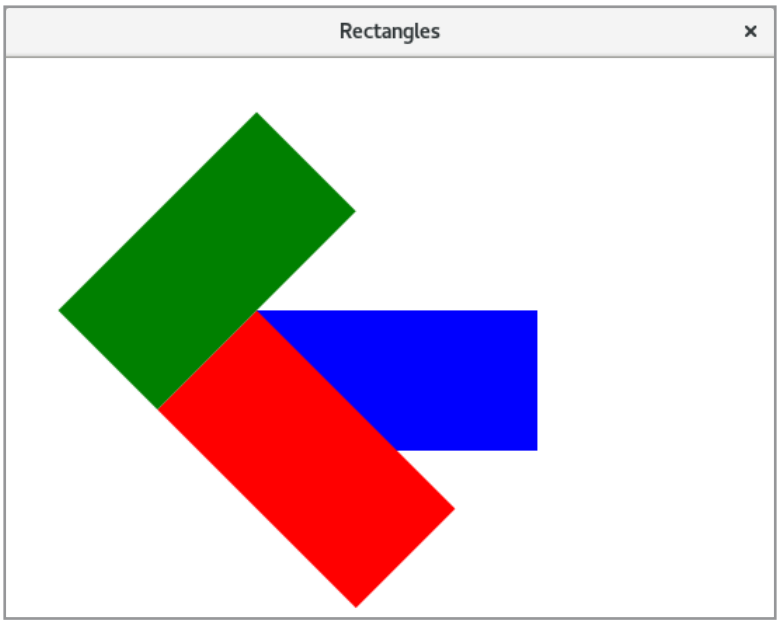
Write a program that opens the following window:



The blue rectangle has the top corner in (20, 20) while the size is 200 × 100. The red rectangle is a scaling of the blue where it is scaled to 50% in both directions and from the point (20, 20). The green rectangle is a corresponding scale, but followed by a translation. Note that in the case of compound transformations, the order means something.

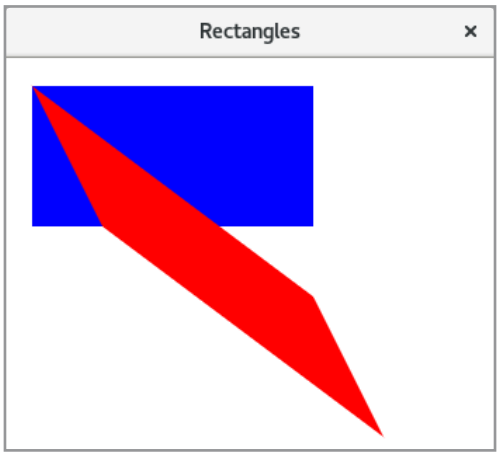
EXERCISE 17

Write an application that opens the window below. Here the blue rectangle has the upper left corner in (180, 180) while the size is 200 × 100. The red rectangle is a rotation of 45 degrees of the blue rectangle about the point (180, 180). The green rectangle is a rotation about the same point, but with the angle -45 degrees, and the rotation is followed by a translation.



EXERCISE 18

Write a program that opens a window similar to the following when the red rectangle is to be a shearing of the blue rectangle:

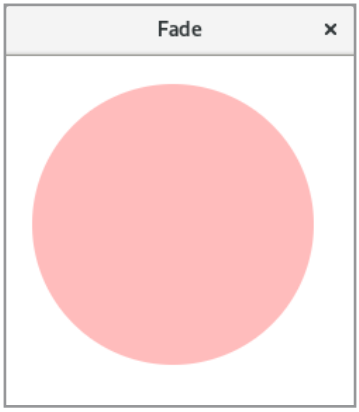


6.1 ANIMATIONS

In this section I introduce animations, which you also can perceive as a form of transformations of a node. An animation basically means changing one or the properties of a node, and if you do it many times and do it at short intervals, it will seem to the user as if something is "alive" with that node. As described in Java 10, a node needs to be redrawn at short intervals, and such that the value of an option has been changed for each redraw. In JavaFX, this logic is encapsulated in a number of classes, where each class defines an animation for a particular value or property. The base class is called a *Transition*, but there are the following specific transitions:

- *FadeTransition*
- *FillTransition*
- *RotateTransition*
- *ScaleTransition*
- *StrokeTransition*
- *TranslateTransition*
- *PathTransition*
- *SequentialTransition*
- *PauseTransition*
- *ParallelTransition*

and the names tells a little about what it is for properties that the class in question is an animation for. As an example, I will show you how to use a *FadeTransition*. A visual node has an *opacity* property that indicates the extent to which that node should be displayed transparent. If you perform an animation of this property, you can see that the figure appears fully drawn, after which it appears more transparent until it completely disappears. It is the task of a *FadeTransition*. If you run the program *FadeAnimation*, the following window shows a red circle:



The circle becomes weaker and weaker and finally disappears completely. Then it grows again until it is shown as it original was. This cycle is repeated 5 times. The code is as follows:

```
package fadeanimation;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.shape.*;
import javafx.scene.paint.Color;
import javafx.animation.*;
import javafx.util.*;

public class FadeAnimation extends Application
{
 @Override
 public void start(Stage stage)
 {
  Group root = new Group();
  Circle circ = createCircle();
  root.getChildren().add(circ);
  Transition trans = createTransition(circ);
  Scene scene = new Scene(root, 250, 250);
  stage.setTitle("Fade");
  stage.setScene(scene);
  stage.show();
  trans.play();
 }

 private Circle createCircle()
 {
  return new Circle(120, 120, 100, Color.RED);
 }

 private FadeTransition createTransition(Node node)
 {
  FadeTransition trans = new FadeTransition(Duration.millis(2000), node);
  trans.setFromValue(1);
  trans.setToValue(0);
  trans.setCycleCount(10);
  trans.setAutoReverse(true);
  return trans;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The method *createCircle()* is trivial, while it is the method *createTransition()* that creates the transition. In this case, it is a *FadeTransition*, and the constructor tells that the animation should take 2 seconds, and what it is for a node that the animation should work on. Otherwise, the animation must change the value from from 1 (not transparent) to 0 (total transparent). The next statement tells the animation to be repeated 10 times and finally, after the animation is complete, the next one must go backwards. In *start()*, a *Transition* object is created and the statement

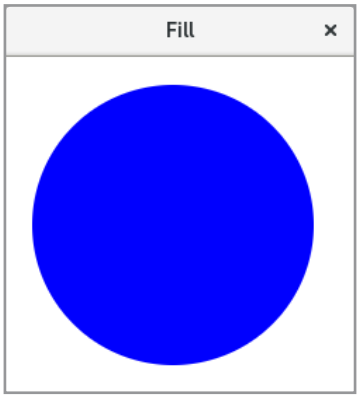
```
trans.play();
```

starts the animation.

The other animation classes are used in the same way, but instead of showing examples, I have formulated a series of exercises that will illustrate applications of the most important classes. You can immediately start from the above program, but for some of the exercises it is necessary to read the documentation to find out what to write – which values should be initialized.

EXERCISE 19

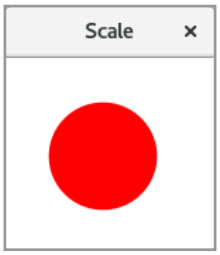
Write an application that opens a window with a blue circle:



Then the program should with a *FillTransition* change color to red. The animation must take 5 seconds and then start again with a blue circle. The animation must be repeated 50 times. Note that a *FillTransition* can not be applied to a general node, but it must be a *Shape*.

EXERCISE 20

Write a program that you can call *ScaleAnimation*, which opens a window with a red circle:

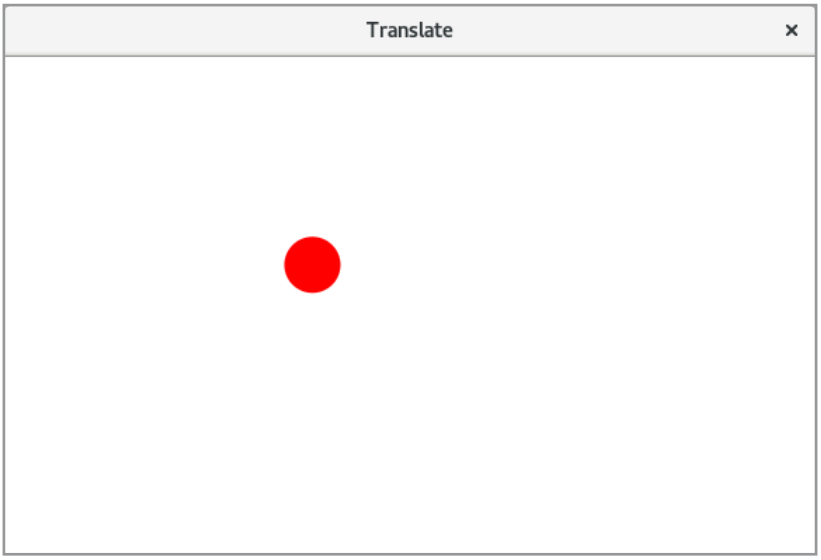


The program must, with an animation, change the size of the circle, so it will shrink and grow.

EXERCISE 21

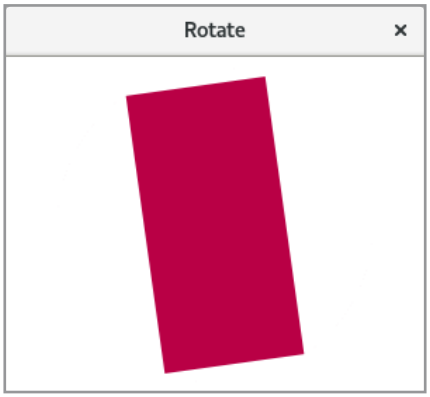
Write a program that you can call *TranslateAnimation*. The program must open a window, as shown below, where a red circle with an animation is moved from the upper left corner to the bottom right corner. You can use a *TranslateTransition*. The animation period must

be 2 seconds and the animation must be repeated 50 times. You must also set *AutoReverse* to true.



EXERCISE 22

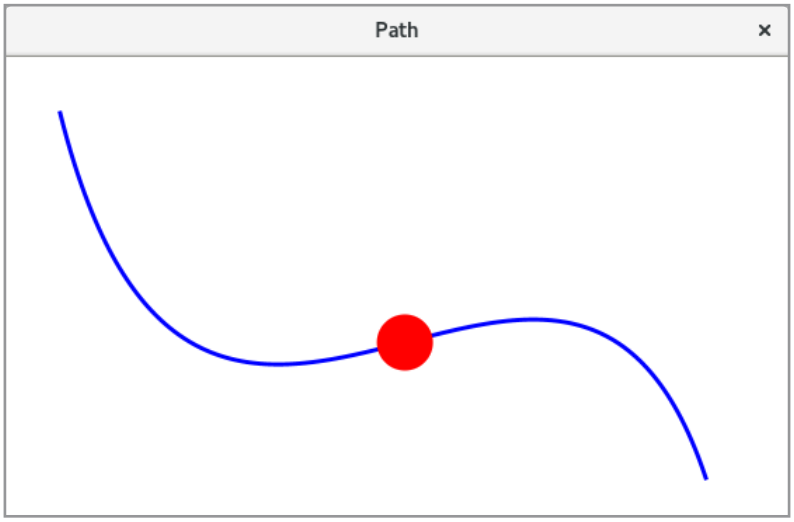
You must write a program called *RotateAnimation*, which shows a rectangle. The rectangle should rotate from 0 to 360 degrees, and simultaneously the color must change from red to blue – and back again in the next rotation:



Note that this time you will need to use two transitions.

EXERCISE 23

Write a program *PathAnimation*, where a red circle moves along a *CubicCurve* (see below). The curve must be defined as a *Path* object, and as a transition object you can use a *PathTransition*. An animation of the circle must run for two seconds, and you must set *AutoReverse* to *true*.



EXERCISE 24

As the last example of an animation, write a program *SequentialAnimation* similar to the above, where a red circle moves along a *CubicCurve*, but the animation is only repeated once. When the circle reaches the end of the curve (the animation stops), the circle should with an animation change color to blue and back to red again. The program must therefore use two animations that are performed sequentially.

7 COMPONENTS

Looking at the foregoing, I have primarily focused on how you in JavaFX works with geometric objects, but I have only overall mentioned components that the user can interact with. Viewed from practical programs it is of course the most interesting and is the subject of this and the following chapters. It is about which components are available and how they are laid out in a window. In addition, there is event management.

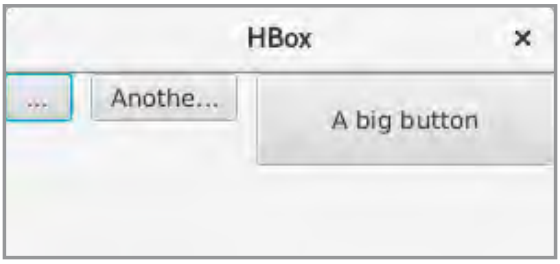
7.1 LAYOUT

The program's scene graph consists of components, and these components can be placed absolutely as I have used in conjunction with geometric objects, but typically, nodes are placed in the scene graph using layout panes where a layout pane is an branch node. Viewed from the programmer, a layout pane has the same characteristics (and purposes) as a layout manager in Swing, and the theory resembles what you know from Swing. All layout panes have a common basic class *Pane*, and there are the following specific panes:

- *HBox*
- *VBox*
- *BorderPane*
- *StackPane*
- *Flow Pane*
- *TextFlow*
- *AnchorPane*
- *TitlePane*
- *Grid Pane*

In this section I show how the layout panes are used to layout components, and of course they can be nested in the same way you know it from Swing. The following examples are similar to each other and briefly show the application of the different panes, but you should examine the online documentation and what properties are available.

I want to start with a *HBox* (the *HBoxProgram* application) which places components horizontally in a window, and here with three buttons:



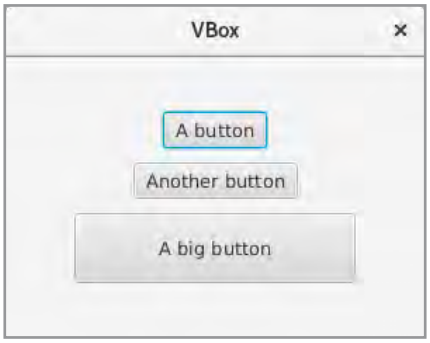
As you can see, the size of the buttons (width) changes according to the available space, and below I have shown the same window after the size has changed:

As you can see, the components are as default left-aligned, but it can be changed with a property. The code is as follows:

```
package hboxprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.stage.*;

public class HBoxProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  HBox root = new HBox();
  root.setSpacing(10);
  root.getChildren().add(createButton("A button"));
  root.getChildren().add(createButton("Another button"));
  root.getChildren().add(createButton("A big button", 200, 50));
  Scene scene = new Scene(root, 300, 100);
  stage.setTitle("HBox");
  stage.setScene(scene);
  stage.show();
 }

 private Button createButton(String text)
 {
  return new Button(text);
 }

 private Button createButton(String text, int width, int height)
 {
  Button cmd = new Button(text);
  cmd.setPrefSize(width, height);
  return cmd;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

You may notice that the last method *createButton()* assigns the button a preferred size, and in JavaFX, basically, the same applies to a component's size, as you know from Swing. As for the method *start()*, there is not much to explain, but you should note how to define a gap between the components. A *HBox* is a very simple layout pane and can only be used to arrange components on a line relative to their preferred size.

There is a layout pane, called *VBox*, which is parallel to *HBox*, the difference being that it places the components on a vertical line. The *VBoxProgram* program shows the application of this layout pane:




```
public void start(Stage stage)
{
 VBox root = new VBox();
 root.setSpacing(10);
 root.setAlignment(Pos.CENTER);
 root.getChildren().add(createButton("A button"));
 root.getChildren().add(createButton("Another button"));
 root.getChildren().add(createButton("A big button", 200, 50));
```

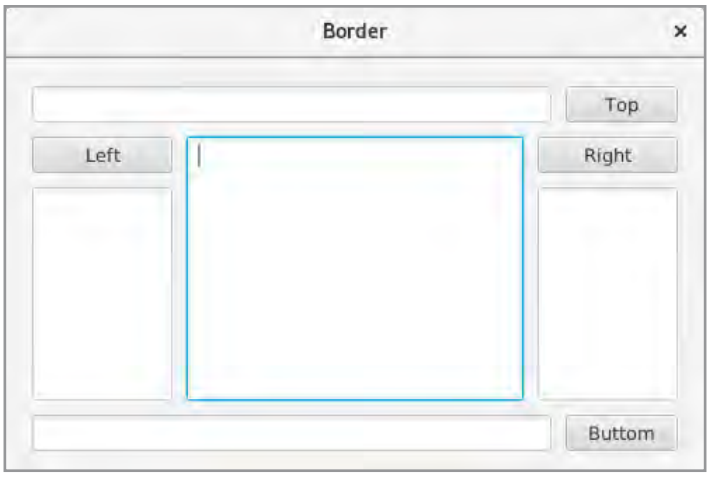
```
 Scene scene = new Scene(root, 300, 200);
 stage.setTitle("VBox");
 stage.setScene(scene);
 stage.show();
}
```

There is not much new to explain, but you should note how to indicate that the components should be displayed centered in the window. You are encouraged to investigate what options the type of *Pos* otherwise defines.

A *BorderPane* looks like a *BorderLayout* and divides the window into 5 areas, called *center*, *top*, *right*, *bottom* and *left* and usually referenced in this order. As with a *BorderLayout*, the center will fill what has not been used by the other areas. The program *BorderProgram* opens the window shown below. In addition to show a *BorderPane*, the program should also show the use of nested panes. The window has 9 components:

1. *center*: a *TextArea*
2. *top*: a *BorderPane* with a *TextField* and a *Button*
3. *right*: a *BorderPane* with a *Button* and a *TextArea*
4. *bottom*: a *BorderPane* with a *TextField* and a *Button*
5. *left*: a *BorderPane* with a *Button* and a *TextArea*

You should notice the spacing between the components and how they are defined in the code.




```
public class BorderProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  BorderPane root =
```

```
    new BorderPane(createField(), createField("Top"), createField("Right", 100),
     createField("Buttom"), createField("Left", 100));
  root.setPadding(new Insets(20, 20, 20, 20));
  Scene scene = new Scene(root, 500, 300);
  stage.setTitle("Border");
  stage.setScene(scene);
  stage.show();
 }

 private Pane createField()
 {
  BorderPane pane = new BorderPane(new TextArea());
  pane.setPadding(new Insets(10, 10, 10, 10));
  return pane;
 }

 private Pane createField(String text)
 {
  Button cmd = new Button(text);
  cmd.setPrefWidth(80);
  BorderPane pane = new BorderPane(new TextField(), null, cmd, null, null);
  BorderPane.setMargin(cmd, new Insets(0, 0, 0, 10));
  return pane;
 }

 private Pane createField(String text, int width)
 {
  Button cmd = new Button(text);
  cmd.setPrefWidth(width);
  TextArea field = new TextArea();
  field.setPrefWidth(width);
  BorderPane pane = new BorderPane(field, cmd, null, null, null);
  BorderPane.setMargin(cmd, new Insets(10, 0, 10, 0));
  BorderPane.setMargin(field, new Insets(0, 0, 10, 0));
  return pane;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The individual 5 areas are created with three helper methods named *createField()*. The first is used for the center and creates a *BorderPane* with a *TextArea* component. You should notice how to set a spacing of 10 to the remaining 4 areas with *setPadding()*. Also note that the *BorderPane* constructor specifies a single parameter, which is the component to be shown

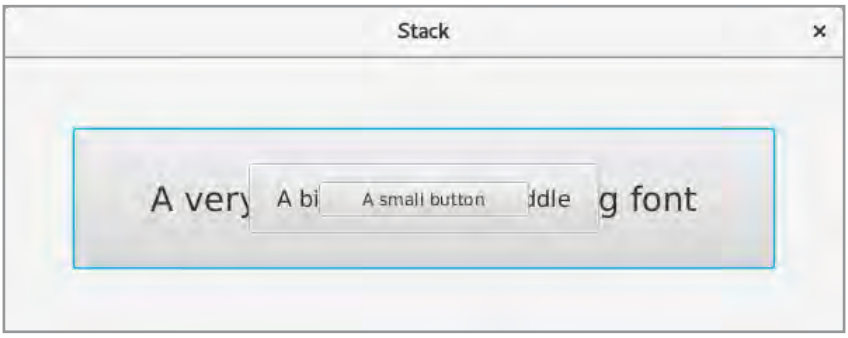
at the center. The second *createField()* is used to create a *BorderPane* to be placed at top and bottom respectively. Here you specify 5 parameters for the constructor in *BorderPane*, which is interpreted as *center*, *top*, *right*, *bottom* and *left*. In particular, note how to define that the button should have a left margin in the container:

```
BorderPane.setMargin(cmd, new Insets(0, 0, 0, 10));
```

Here, *setMargin()* is a static method in the *BorderPane* class, which defines the margin of the first parameter in the current layout pane. The last *createField()* method is used to create a *BorderPane* for the contents of the right and left area and, in principle, it works as the previous method. You should note that a bottom margin is defined for both components.

Finally, there is the *start()* method, where the *root* element this time is a *BorderPane* (which is a branch node). Here you should note how to create *root*, and to set a margin of 20.

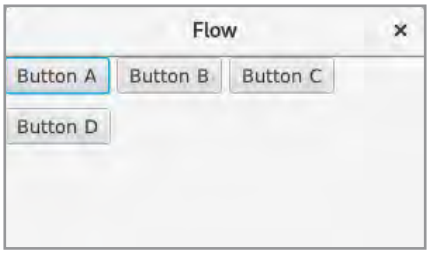
A *StackPane* is a layout pane that arranges the components on top of each other in the order they are added. The program *StackProgram* shows three buttons placed on top of each other:



The application is probably limited, but primarily for *Shape* objects, the possibility can be useful. It's quite simple to use a StackPane:

```
public void start(Stage stage)
{
 StackPane root = new StackPane();
 root.getChildren().add(createButton("A very big button with a big font",
  500, 100, 24));
 root.getChildren().add(createButton("A big button in the middle", 250, 50, 16));
 root.getChildren().add(createButton("A small button", 150, 25, 12));
 Scene scene = new Scene(root, 600, 200);
 stage.setTitle("Stack");
 stage.setScene(scene);
 stage.show();
}

private Button createButton(String text, int width, int height, int size)
{
 Button cmd = new Button(text);
 cmd.setFont(Font.font(size));
 cmd.setPrefSize(width, height);
 return cmd;
}
```

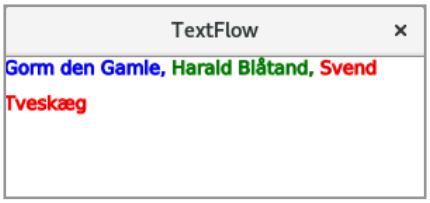
A *FlowPane* is a layout pane that essentially behaves like a *FlowLayout* in Swing:



If you increase the width of the window, all components will appear on the same line. Regarding the code, you should first notice how to define the gap between the components:

```
public void start(Stage stage)
{
 FlowPane root = new FlowPane();
 root.setAlignment(Pos.TOP_LEFT);
 root.setHgap(5);
 root.setVgap(10);
 root.getChildren().add(createButton("Button A"));
 root.getChildren().add(createButton("Button B"));
 root.getChildren().add(createButton("Button C"));
 root.getChildren().add(createButton("Button D"));
 Scene scene = new Scene(root, 300, 150);
 stage.setTitle("Flow");
 stage.setScene(scene);
 stage.show();
}
```

There is also a layout pane, which is called *TextFlow*, and behaving in the same way as a *FlowPane*, but only for *Text* nodes. The program *TextflowProgram* shows how to use this pane. It's probably not the most commonly used layout pane, but it may be useful if you need to write a custom component that supports text wrapping – something that is actually relatively complicated in Swing.




```
public void start(Stage stage)
{
 TextFlow root = new TextFlow();
 root.setLineSpacing(10);
 root.getChildren().add(createText("Gorm den Gamle, ", Color.BLUE));
 root.getChildren().add(createText("Harald Blåtand, ", Color.GREEN));
 root.getChildren().add(createText("Svend Tveskæg", Color.RED));
 Scene scene = new Scene(root, 300, 100);
 stage.setTitle("TextFlow");
 stage.setScene(scene);
 stage.show();
}
```

```
private Text createText(String text, Color color)
{
 Text node = new Text(text);
 node.setStroke(color);
 return node;
}
```

An AnchorPane places a component relative to the container's edge:

```
public void start(Stage stage)
{
 AnchorPane root = new AnchorPane();
 root.setPadding(new Insets(30, 30, 30, 30));
 Button cmd = createButton("A button");
 root.getChildren().add(cmd);
 AnchorPane.setRightAnchor(cmd, 30.0);
 AnchorPane.setBottomAnchor(cmd, 50.0);
 Scene scene = new Scene(root, 300, 200);
 stage.setTitle("Anchor");
 stage.setScene(scene);
 stage.show();
}
```

In this case, *root* has a margin of 30, and the component is positioned so that it sits 30 from the right edge (relative to the margin) and 50 above the bottom (relative to the margin). If the window's size changes, the component will follow the bottom and right edges.

Then there is a *TilePane*, which places components into cells of the same size. The size of the cells is determined by the largest preferred size (in this case the last button). Is there no room for all components on the line, the line wraps, so the last component appears on the next line.


```
public void start(Stage stage)
{
 TilePane root = new TilePane();
 root.getChildren().add(createButton("A button"));
 root.getChildren().add(createButton("Another button"));
 root.getChildren().add(createButton("A big button", 200, 50));
 Scene scene = new Scene(root, 650, 100);
 stage.setTitle("Tile");
 stage.setScene(scene);
 stage.show();
}
```

In this case, the components are set horizontally, which is default, but with a property, it is possible to specify that the components should be laid out vertically.

Finally, there is a *GridPane*, which is a very useful layout pane. Compared with Swing, you can achieve a bit of the same as with a *GridBagLayout*, but it is far easier to use. Basically,

it's a grid. but the cells do not necessarily have the same size. The program *GridProgram* opens the following window:

that has 5 components:

- 2 *Label* components
- 2 *TextField* components
- 1 *Button* component

If you change the window size, the components will stay at their positions and with the same size – at least as long as the window is large enough. It is therefore a typical dialog box. The code is quite simple:

```
package gridprogram;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.stage.*;
import javafx.geometry.*;

public class GridProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  GridPane root = new GridPane();
  root.setPadding(new Insets(20, 20, 20, 20));
  root.setVgap(20);
  root.add(createLabel("First name", 80), 0, 0);
  root.add(createField(300), 1, 0);
  root.add(createLabel("Last name", 80), 0, 1);
  root.add(createField(300), 1, 1);
```

```
 Button cmd = createButton("OK");
 root.add(cmd, 1, 2);
 GridPane.setHalignment(cmd, HPos.RIGHT);
 Scene scene = new Scene(root, 500, 200);
 stage.setTitle("Grid");
 stage.setScene(scene);
 stage.show();
}

private Label createLabel(String text, int width)
{
 Label label = new Label(text);
 label.setPrefWidth(width);
 return label;
}

private TextField createField(int width)
{
 TextField field = new TextField();
 field.setPrefWidth(width);
 return field;
}
```

```
 private Button createButton(String text)
 {
  return new Button(text);
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

In the method *start()*, a *GridPane* is created as root. Note that you do not say anything about the number of rows and columns. There is a margin of 20, and there must be a gap of 20 between the individual rows. Then I add a *Label* with the width 80, and I define that it should be in cell (0, 0), which is the cell in the upper left corner. The next component is a *TextField* with the width 300, which should be in cell (1, 0), which can be translated into column 1 and row 0. Similarly, the two next components are placed solely with the difference that they should be placed in row 1. Finally, place the button in cell (1, 2) and thus column 1 row 2. Note that nothing is placed in cell (0, 2). It is not necessary – a cell may be empty. In particular, you should note that a *GridPane* has a static method that you can use to specify aligment for a component within the component's cell:

```
GridPane.setHalignment(cmd, HPos.RIGHT);
```

7.2 EVENTS

JavaFX has its own event types, and the base type is called *Event*. Examples of specific event types include:

- *MouseEvent*
- *KeyEvent*
- *DragEvent*
- *WindowEvent*
- *ActionEvent*

An event is characterized by

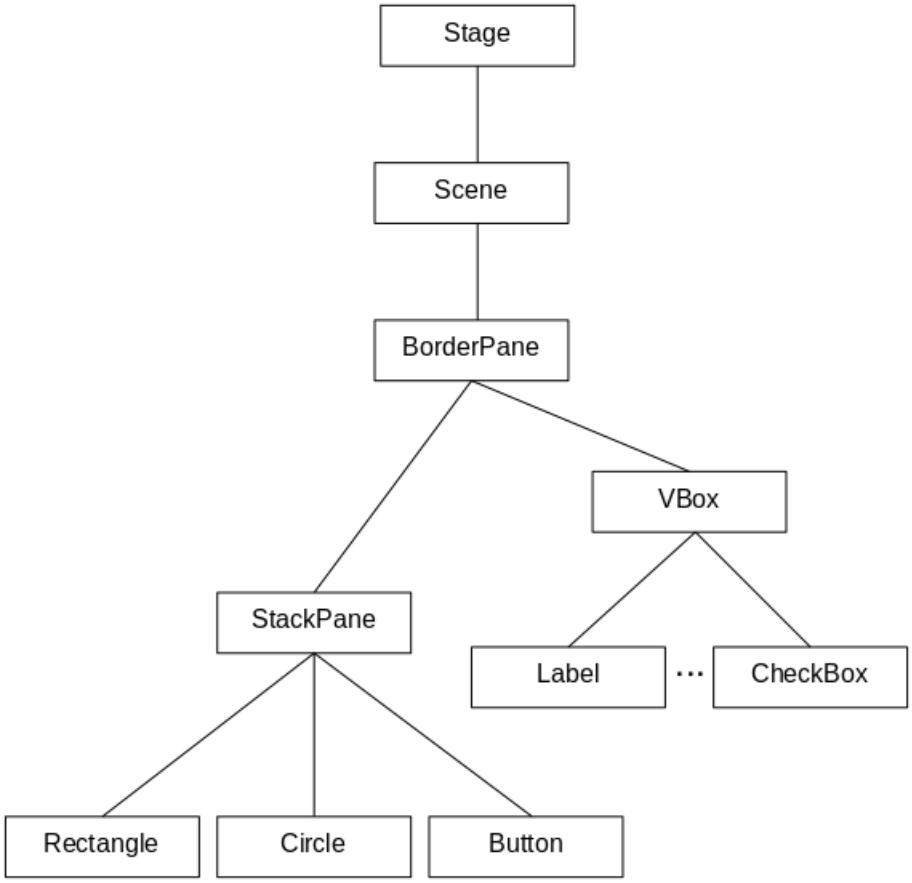
1. a target, which is the node where an event occurs and may for example be a window, a scene or a node
2. a source, which is the one that has generated an event and can, for example, be the mouse
3. an event type, such as mouse pressed, mouse released and so on

When an event occurs, one talks about an *event dispatch chain*, which is the route from the stage to that node. Is it a *Circle* node, for example, it could be

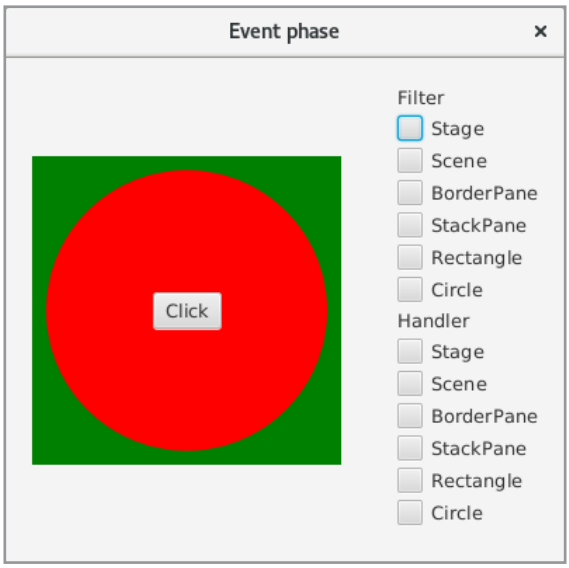
Stage – Scene – Group – Circle

When an event occurs, it is sent to all nodes in its *dispatch chain* from top to bottom. If there is a node that has registered a *filter* for that event, this will be done, after which the event will be sent to its target node where it will be processed if there is an event handler. This phase is called the *Event Capturing Phase*. After that, the event will follow the way back, and for each node it will be processed if there is a *handler* assigned. This phase is called the *Event Bubbling Phase*. An event can thus result in that multiple event handlers are performed and event handlers during the capturing phase are called *filters* while event handlers during the bubbling phase are called *handlers*.

When an event follows the dispatch chain, it can at any time be stopped by the method *consume()*, which means that the event will not be forwarded in the chain. As an example, I will show a program *EventPhaseProgram* that creates the following scene graph:



where under *VBox* there are 14 nodes. If you opens the program you get the window:



For example, if you click a location in the green area, the following is written on standard output:

```
Stage – capturing
Sceen – capturing
BorderPane – capture
StackPane – capture
Rectangle – capture
Rectangle – bubbling
StackPane – bubbling
BorderPane – bubbling
Sceen – bubbling
Stage – bubbling
```

Here you can see how the event of mouse clicks has been processed all the way through the dispatch chain and back again. The meaning with the checkboxes of the window is that you check where the processing of an event should be interrupted. For example, checking *Filter Scene* and clicking the green rectangle again gives you the result:

```
Stage – capturing
Sceen – capturing
```

You are encouraged to test the program and what happens when you click the mouse on the different nodes depending on which checkboxes are selected.

Then there is the code:

```
public class EventPhaseProgram extends Application
{
 private CheckBox filterStage = new CheckBox("Stage");
 private CheckBox filterScene = new CheckBox("Scene");
 private CheckBox filterBorder = new CheckBox("BorderPane");
 private CheckBox filterPane = new CheckBox("StackPane");
 private CheckBox filterRect = new CheckBox("Rectangle");
 private CheckBox filterCirc = new CheckBox("Circle");
 private CheckBox handleStage = new CheckBox("Stage");
 private CheckBox handleScene = new CheckBox("Scene");
 private CheckBox handleBorder = new CheckBox("BorderPane");
 private CheckBox handlePane = new CheckBox("StackPane");
 private CheckBox handleRect = new CheckBox("Rectangle");
 private CheckBox handleCirc = new CheckBox("Circle");

 @Override
 public void start(Stage stage)
 {
```

```
 stage.setTitle("Event phase");
 stage.addEventFilter(MouseEvent.MOUSE_CLICKED,
  new Handler("\nStage - capturing", filterStage));
 stage.addEventHandler(MouseEvent.MOUSE_CLICKED,
  new Handler("Stage - bubbling", handleStage));
 stage.setScene(createScene(createBorder(createPane(createRect(),
  createCircle(), createButton()), createPanel())));
 stage.show();
}

private Scene createScene(Pane root)
{
 Scene scene = new Scene(root, 400, 360);
 scene.addEventFilter(MouseEvent.MOUSE_CLICKED,
  new Handler("Sceen - capturing", filterScene));
 scene.addEventHandler(MouseEvent.MOUSE_CLICKED,
  new Handler("Sceen - bubbling", handleScene));
 return scene;
}

private Pane createBorder(Pane pane1, Pane pane2)
{
 BorderPane pane = new BorderPane(null, null, pane2, null, pane1);
 pane.setPadding(new Insets(20, 20, 20, 20));
 pane.addEventFilter(MouseEvent.MOUSE_CLICKED,
  new Handler("BorderPane - capture", filterBorder));
 pane.addEventHandler(MouseEvent.MOUSE_CLICKED,
  new Handler("BorderPane - bubbling", handleBorder));
 return pane;
}

private VBox createPanel()
{
 VBox pane = new VBox();
 pane.setSpacing(5);
 pane.getChildren().add(new Label("Filter"));
 pane.getChildren().addAll(filterStage, filterScene, filterBorder, filterPane,
  filterRect, filterCirc);
 pane.getChildren().add(new Label("Handler"));
 pane.getChildren().addAll(handleStage,handleScene,handleBorder,handlePane,
  handleRect, handleCirc);
 return pane;
}

private Pane createPane(Rectangle rect, Circle circle, Button cmd)
{
 StackPane pane = new StackPane(rect, circle, cmd);
 pane.addEventFilter(MouseEvent.MOUSE_CLICKED,
  new Handler("StackPane - capture", filterPane));
```

```
 pane.addEventHandler(MouseEvent.MOUSE_CLICKED,
  new Handler("StackPane - bubbling", handlePane));
 return pane;
}

private Button createButton()
{
 Button cmd = new Button("Click");
 cmd.setOnAction(new EventHandler<ActionEvent>()
 {
  @Override
  public void handle(ActionEvent e)
  {
   new Alert(Alert.AlertType.INFORMATION,
     "You have clicked the button!").showAndWait();
  }
 });
 return cmd;
}

private Circle createCircle()
{
 Circle circle = new Circle(20, 20, 100, Color.RED);
```

```
  circle.addEventFilter(MouseEvent.MOUSE_CLICKED,
   new Handler("Circle - capture", filterCirc));
  circle.addEventHandler(MouseEvent.MOUSE_CLICKED,
   new Handler("Circle - bubbling", handleCirc));
  return circle;
 }

 private Rectangle createRect()
 {
  Rectangle rect = new Rectangle(0, 0, 220, 220);
  rect.setFill(Color.GREEN);
  rect.addEventFilter(MouseEvent.MOUSE_CLICKED,
   new Handler("Rectangle - capture", filterRect));
  rect.addEventHandler(MouseEvent.MOUSE_CLICKED,
   new Handler("Rectangle - bubbling", handleRect));
  return rect;
 }

 public static void main(String[] args)
 {
  launch(args);
 }

 class Handler implements EventHandler<MouseEvent>
 {
  private String text;
  private CheckBox chkBox;

  public Handler(String text, CheckBox chkBox)
  {
   this.text = text;
   this.chkBox = chkBox;
  }

  @Override
  public void handle(MouseEvent e)
  {
   System.out.println(text);
   if (chkBox.isSelected()) e.consume();
  }
 }
}
```

First, 12 *CheckBox* components are created, where the 6 first are to be used for the filters (capturing phase), while the 6 next are to be used for the handlers (bubbling phase). If you consider the method *start(),* this time, both a filter and a handler are assigned to the *Stage* node – the node is the top of the scene graph. A filter associated with the method

addEventFilter(), while a handler is associated with the method *addEventHandler()*. In either case, an event handler is associated for mouse clicked, such as:

```
stage.addEventFilter(MouseEvent.MOUSE_CLICKED, new Handler("…", filterStage));
```

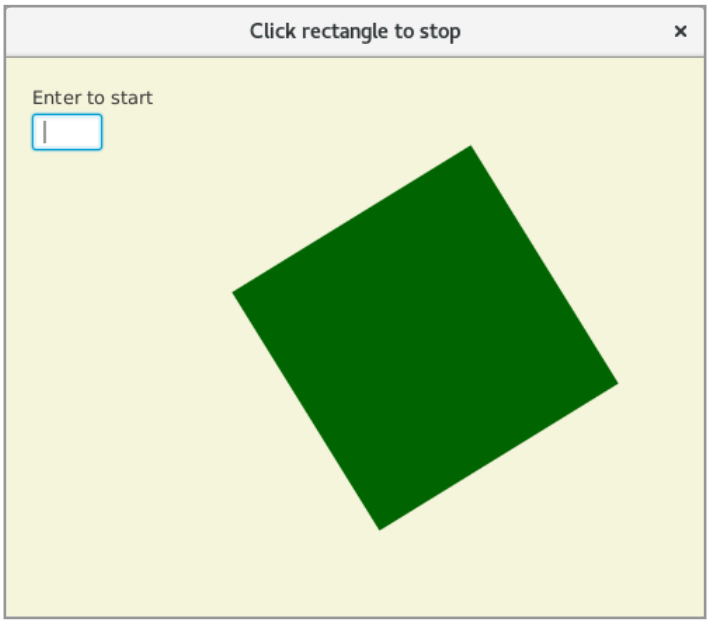
where the first parameter indicates which event to be listened to while the other is the event handler, there is an object that (in this case) implements the interface *EventHandler<MouseEvent>*. Such an object is defined by the class *Handler*, which is an inner class where the constructor has the text to be printed on the screen as a parameter, and a reference to the *CheckBox* to be tested. The class should implement the method:

```
public void handle(MouseEvent e)
{
}
```

In this case, the method performs a *System.out.println()*, and then tests whether the appropriate *CheckBox* is selected and, if necessary, sets the event as *consumed*, which means that it will not be forwarded in the dispatch chain. You should note how the scene graph is built by calling the methods that create the individual nodes.

These methods are all simple and all built in the same way, and you should primarily notice how to assign filters and handlers to the individual nodes in the same way as described above. You should note *createButton()*, where no filter or handler is attached. If you run the program and click on the button, you will find that no event handlers are performed in the dispatch chain, and the reason is that the button instead with *setOnAction()* is associated with an event handler of the type *EventHandler<ActionEvent>*. This means that with the mouse click, the button will take care of the event action and not fire a regular mouse event, but instead an *ActionEvent*. Also note that the event handler of the button's *ActionEvent* opens a simple message box, including the syntax for how to do it in JavaFX.

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



If you are in the entry field and enter the key *Enter*, the green square will rotate and clicking on the square it will stop the rotation. The code is as follows:

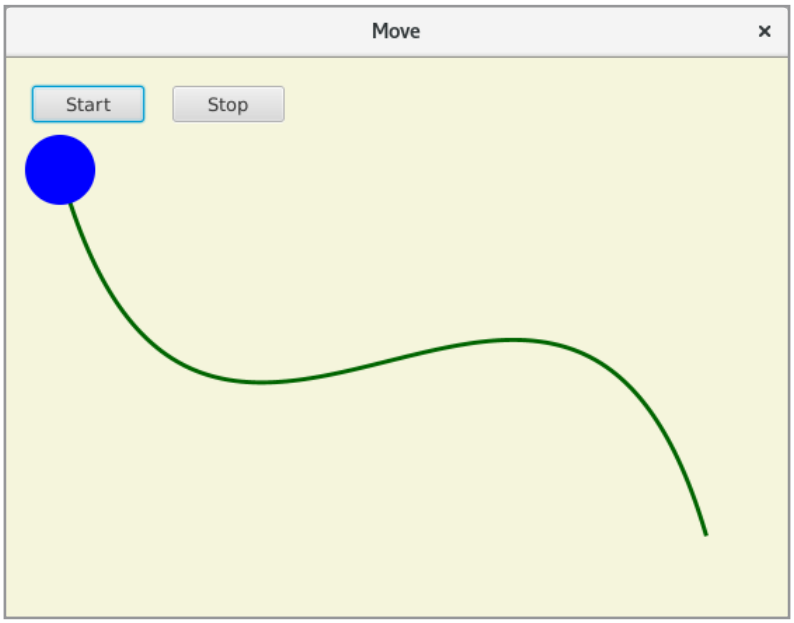
```
public class KeyEventProgram extends Application
{
 private Transition trans;

 @Override
 public void start(Stage stage)
 {
  Rectangle rect = createRect();
  trans = createRotate(rect);
  Group root = new Group(createLabel(), createField(), rect);
  Scene scene = new Scene(root, 500, 400);
  scene.setFill(Color.BEIGE);
  stage.setTitle("Click rectangle to stop");
  stage.setScene(scene);
  stage.show();
 }

 private Rectangle createRect()
 {
  Rectangle rect = new Rectangle(200, 100, 200, 200);
  rect.setFill(Color.DARKGREEN);
  rect.setOnMouseClicked(e -> trans.stop());
  return rect;
 }

 private Transition createRotate(Shape shape)
 {
  RotateTransition trans = new RotateTransition(Duration.millis(1000), shape);
  trans.setByAngle(360);
  trans.setCycleCount(50);
  trans.setAutoReverse(true);
  return trans;
 }

 private TextField createField()
 {
  TextField field = new TextField();
  field.setLayoutX(20);
  field.setLayoutY(40);
  field.setPrefWidth(50);
  field.addEventFilter(KeyEvent.KEY_TYPED, new EventHandler<KeyEvent>()
  {
   public void handle(KeyEvent e)
   {
    if (e.getCharacter().charAt(0) == '\r') trans.play();
    e.consume();
   }
  });
  return field;
 }
```

```
 private Label createLabel()
 {
  Label label = new Label("Enter to start");
  label.setLayoutX(20);
  label.setLayoutY(20);
  return label;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

The class defines a variable for a transition, which is the animation that rotates the square. It is created by the method *createRotate()*, which does not contain anything new in terms of animations. The actual *Transition* object is created in the method *start()*. Here you should also notice how to set the background color for the window. The square is created in the method *createRect()*. Here, the method *setOnMouseClicked()* is used to attach an event handler for mouse clicks to the square, and you should note how the event handler is defined using a lambda expression.

Then there is the method *createField()* that creates the entry field. It is a *TextField*, and the most important is the association of the event handler. It is a filter and the handler is associated for a *KeyEvent* of the type *KEY_TYPED*. The handler uses the method *getCharacter()* to test what has been entered and has the first character the code 13, it is the *Enter* key that is pressed, and if so, the animation will start. For any event, the event is marked as cosumed so that it is not passed on in the chain, and the effect is that the character for the key pressed is never displayed in the input field.

The example thus shows how to catch events for mouse and keyboard. In the example, it happens in two different ways, where in *createField()* it takes place at low level with *addEventFilter()* while in *createRect()* it occurs using a *convenience* method. Many of the JavaFX classes defines event handlers as properties, and you can then register an event handler using a set method for that *convenience* method. The goal is to make it easier to associate event handlers what the following program *HandlerProgram* should illustrate:



Clicking the Start button starts an animation where the circle moves along the curve and the animation runs until you click the stop button. If you click on the circle, it changes color.

```
public class HandlerProgram extends Application
{
 private boolean blue = true;
 private Transition trans;

 @Override
 public void start(Stage stage)
 {
  Circle circle = createCircle();
  Path path = createPath();
  trans = createTrans(circle, path);
  Group root = new Group(createButton("Start", 20, e -> trans.play()),
   createButton("Stop", 120, e -> trans.stop()), path, circle);
  Scene scene = new Scene(root, 560, 400);
  scene.setFill(Color.BEIGE);
  stage.setTitle("Move");
  stage.setScene(scene);
  stage.show();
 }

 private Button createButton(String text, int pos,
  EventHandler<ActionEvent> handler)
 {
  Button cmd = new Button(text);
  cmd.setLayoutX(pos);
  cmd.setLayoutY(20);
  cmd.setPrefSize(80, 25);
  cmd.setOnAction(handler);
  return cmd;
 }

 private Transition createTrans(Shape shape, Path path)
 {
  PathTransition trans = new PathTransition();
  trans.setDuration(Duration.millis(3000));
  trans.setNode(shape);
  trans.setPath(path);
  trans.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
  trans.setCycleCount(50);
  trans.setAutoReverse(true);
  return trans;
 }

 private Circle createCircle()
 {
  Circle circle = new Circle(40, 80, 25, Color.BLUE);
  circle.setStrokeWidth(20);
  circle.setOnMouseClicked(
```

```
    e -> { blue = !blue; circle.setFill(blue ? Color.BLUE : Color.RED); });
   return circle;
  }

  private Path createPath()
  {
   Path path = new Path(new MoveTo(40, 80),
    new CubicCurveTo(140, 440, 400, 0, 500, 340));
   path.setStroke(Color.DARKGREEN);
   path.setStrokeWidth(3);
   return path;
  }

  public static void main(String[] args)
  {
   launch(args);
  }
}
```

As how to create the animation and the curve there is nothing new and happens in the two methods *createTrans()* and *createRect()*. If you consider the method *createCircle()* that creates the circle, it associates an event handler for mouse clicks using a convenience method called *setOnMouseClicked()*. You should note that the event handler is the parameter and is a lambda expression.

The method *createButton()* is used to create a button. It has three parameters, which is the text of the button, the x-coordinate of the location in the window and the button's event handler. The latter has the type *EventHandler<ActionEvent>* and is associated with the method *setOnActionEvent()*.

7.3 COMPONENTS

Then there are the components (or controls) where JavaFX has many, but the most important are the following:

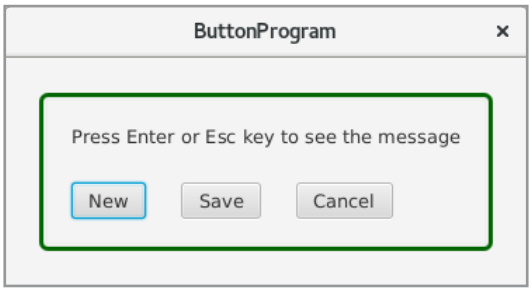
Label	MenuButton	Slider	ColorPicker
Button	MenuItem	ScrollPane	DatePicker
TextField	Menu	ScrollBar	ProgressIndicator
PassWordField	ContextMenu	TabPane	ProgressBar
TextArea	Separator	SplitPane	FileChooser

RadioButton	ChoieBox	TitlePane	HTMLEditor
Hyperlink	ComboBox	Accordion	TableView
CheckBox	ListView	Pagination	TreeView
ToggleButton	WebView	Tooltip	TreeTableView

Some of these components have already been used in the previous examples and the rest of this section consists of exercises and problems where you should try using the above components (but not the last three in the last column postponed to the next book). Most of the components are easy to use and take a good part of the way, as you know it from Swing. Others are more complex, and on the whole, the following exercises require that you use the documentation and, if necessary, also find examples and help on the Internet.

EXERCISE 25

You must write a program that opens the following window:



where there are 4 controls: 1 *Label* and 3 *Button* controls. If you click *New*, the text must be changed to

Creatring a new document…

and clicking *Save* should change the text to

Saving…

If you click the *Cancel* button, the text must be changed to the original text. It is required that the window must not be resizable, and that there must be shortcuts for all three buttons.

With regard to the green frame, it can be defined with a style. This is the subject of the next chapter, but you can try and see how far you can come.

EXERCISE 26

You must write an application that opens the following window:

In the top line there are three *Hyperlink* controls on the left. A *Hyperlink* is the same as a *Button*, but it is drawn in another way. In this case, the three links must refer to a web page (you decide which ones). On the right side there is a *MenuButton*, which is a button with a dropdown, where you can click on a *MenuItem* control. In this case, there must be three items, each of which refers to a web page (which you decide).

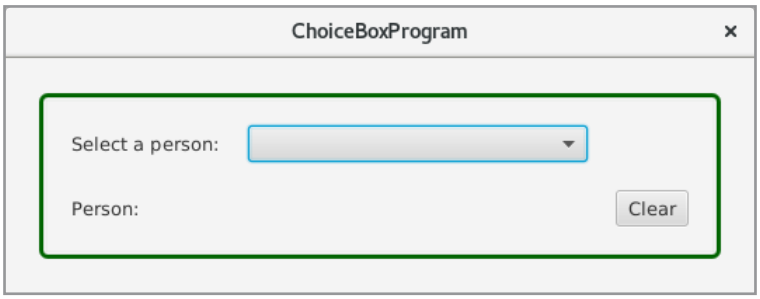
Center, there must be a *WebView* control, which is a control that can open and display a web page.

At the bottom there are three *ToggleButton* controls in a group and a *Label*. When you click a button, the *Label* control must be updated with the button's text, if it is clicked and otherwise the text *None*.

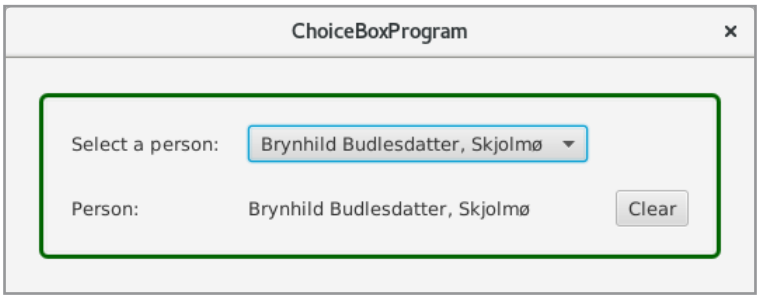
Note that the commands at the top and the commands at the bottom have nothing to do with each other, and the goal is only to illustrate typical controls.

EXERCISE 27

Write a program that uses a *ChoiceBox* control:



You must be able to select a person from the dropdown list when a person is represented by an object of the type *Person*. It must be a simple model class that represents a person by a name and a job title. It is a requirement that the *Person* class must not have any *toString()* method, and instead, a converter must be associated with the *ChoiceBox* control. After selecting a person, the name and the job title must be displayed with a *Label*:



When you click the *Clear* button, this label must be blanked and no person in the *ChoiceBox* control must be selected.

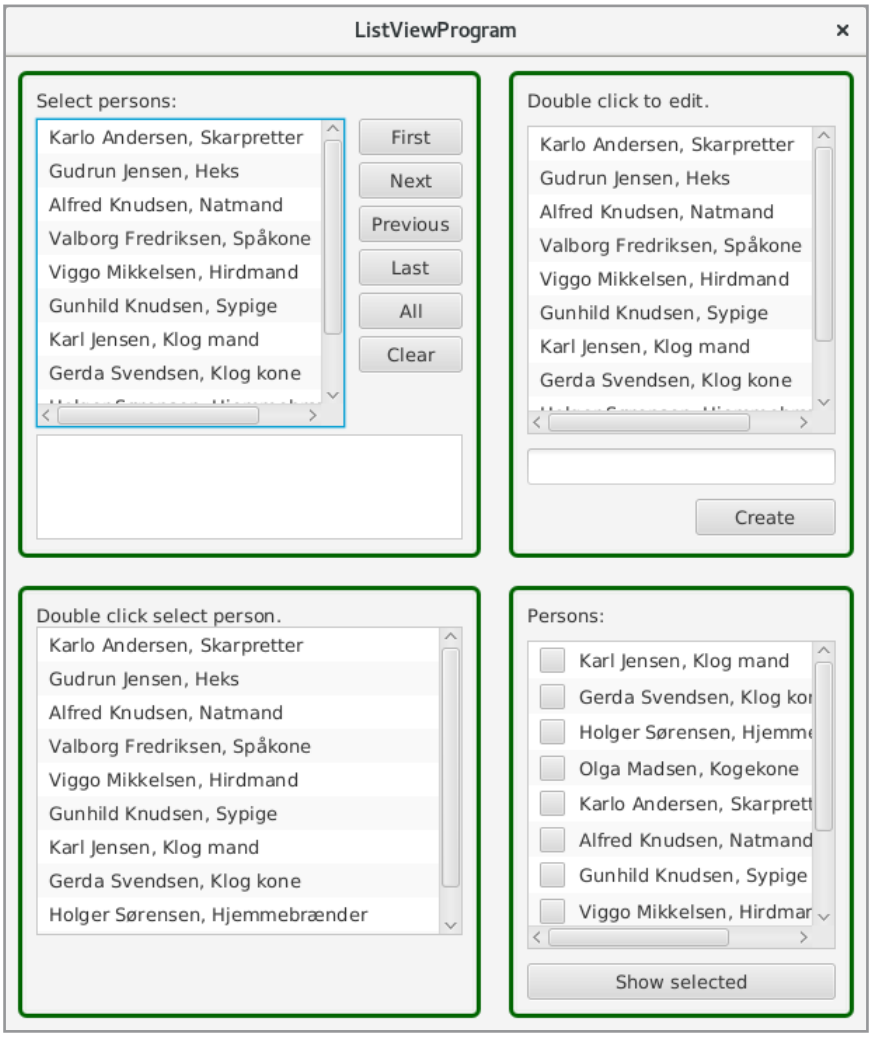
EXERCISE 28

You must write the same program as in the previous exercise, but with one difference: Instead of using a *ChoiceBox*, you use a *ComboBox*.

Once you have written the program, try to investigate what the difference is between a *ChoiceBox* and a *ComboBox*.

PROBLEM 3

From Swing you know the component *JList*, which is a list box and is one of the widely used components. In JavaFX, it is replaced by a *ListView*, which is a much more advanced component with many new options, which uses multiple help classes. You must write an application that opens the following window, where the window shows four *ListView* controls (actually 5), each *ListView* showing 10 *Person* objects where *Person* is the class from exercise 27.



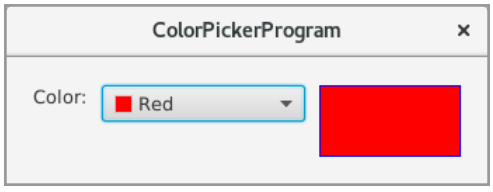
1. The *ListView* control in the upper left corner should show how to select items. You can select an item with the mouse and select an item using the top 5 buttons. Each time you select an item, the blank *ListView* must show which items are selected.
2. The *ListView* control in the upper right corner should show how to edit a *Person* object. If you double-click on a person, the list box must open a field so you can edit the content (it is built into the component). Under the control there is a *TextField* that will be used to add a new *Person* object to the list.

3. The *ListView* control at the bottom left should show items where each element is a *ChoiceBox*, from which you can update the item by selecting another *Person* object (it is also built into the component).
4. Finally, the last *ListView* shows the items as *CheckBox* controls (it is also incorporated into the *ListView* component). If you click on the button, the program must open a message box that displays the objects that are selected (by the checkboxes).

The task requires you to examine the documentation to find out what to write, and maybe you can also can find help online.

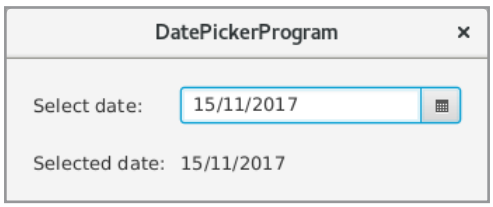
EXERCISE 29

JavaFX has a component called *ColorPicker*, which is a relatively complicated component for selecting a color. On the other hand, the component is both flexible and easy to use. You must write a program that opens a window where you can choose a color used to color a rectangle:



EXERCISE 30

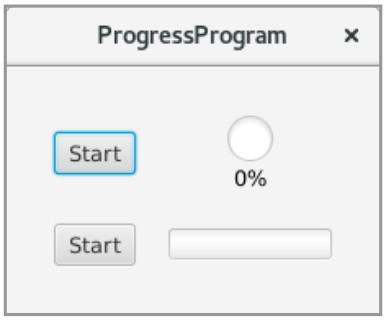
JavaFX has a good *DatePicker* control that can be used to select a date. You must write a program that opens the following window, where November 15, 2017 was selected:



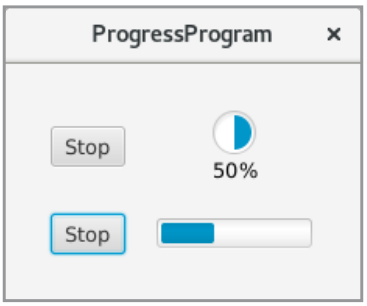
After the date is selected, the bottom *Label* must be uploaded.

EXERCISE 31

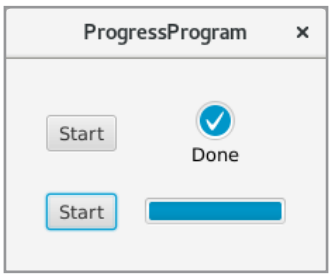
JavaFX has a progress bar like Swing, but there is also a *ProgressIndicator*, and the difference is how the two components are drawn. The *ProgressIndicator* control is bass class for *ProgressBar*. A progress component is updated in the same way as in Swing using a thread represented by a *Task* object contained in the *javafx.concurrent* package. You must write a program that shows a *ProgressIndicator* and a *ProgressBar*, both of which can be started (and stopped) by clicking a button:



and below is the window after clicking on both buttons:

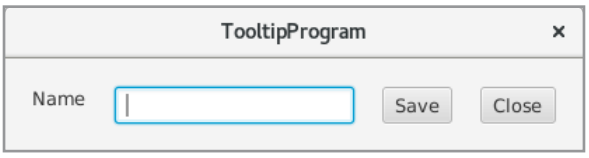


and below the window after the two progress components are completed:



EXERCISE 33

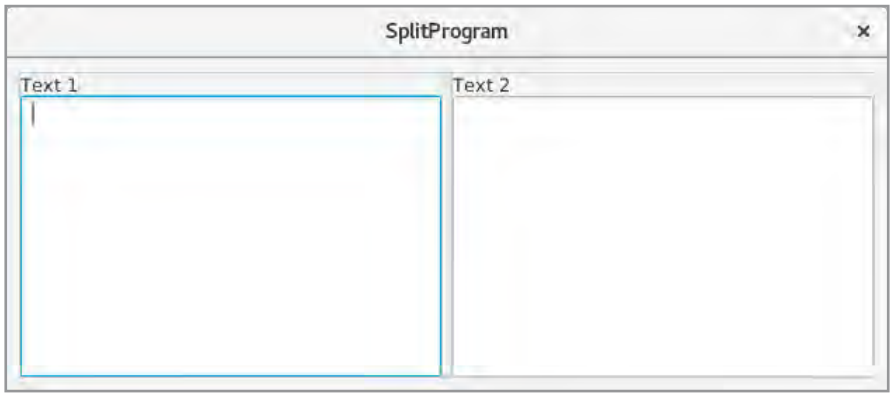
You must write a program that opens the following window, that has a *Label*, a *TextField* and two *Button* controls:



The buttons do not need an action, but the input field and the buttons must have a Tooltip. Here it is interesting that you can attach a style to a tooltip, where styles is the subject of the next chapter.

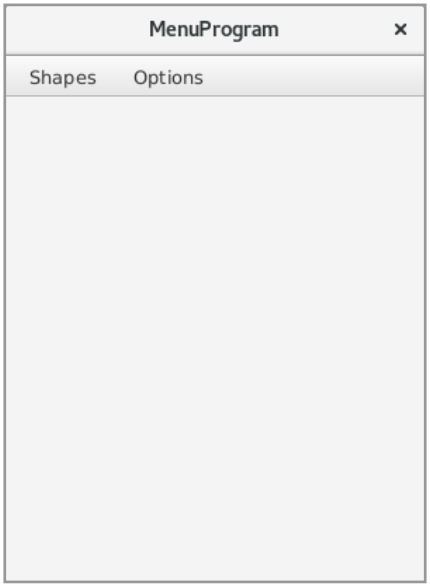
EXERCISE 34

Write a program that has a window with two *TextArea* controls located in a *SplitPane*:

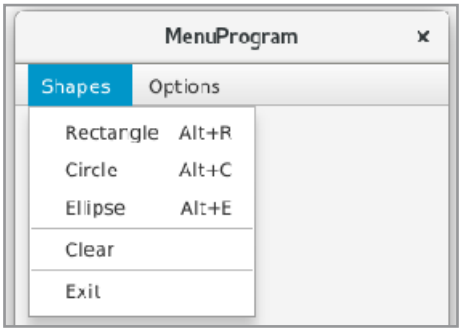


PROBLEM 4

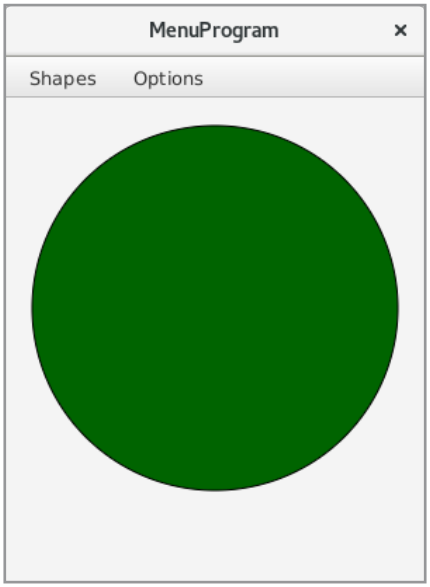
In this task you should work with menus in JavaFX. You must write an application that opens the following window:



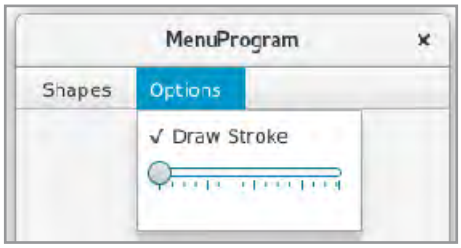
The window has a menubar with two menus, and the window itself (center) has a *Canvas* object. The first menu must have the following functions:



If you click on one of the three top menu items, a dark green shape with a black border in the window should appear, for example:



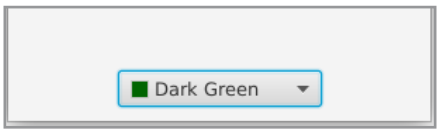
The fourth menu item must clear the window, while the last one should close the program. The *Options* menu has a menu item as well as a *Slider*:



The first menu item must indicate if the circumference of the shape is to be drawn while the slider indicates the size of the perimeter. If you move the slider, the perimeter of any shape should be updated.

If you right click on the canvas object you should get a context menu:

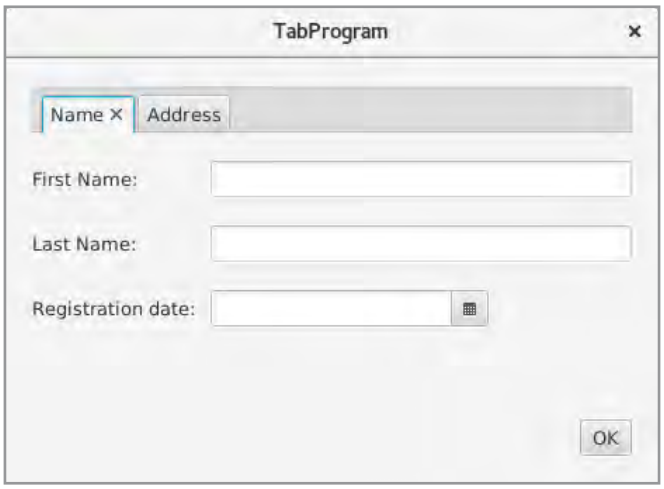
and clicking on one of the two menu items, a *ColorPicker* object should appear at the bottom of the window, so you can choose one of the two colors:



As a last requirement, shortcut keys must be available for all menu items, and accelerator keys must be defined for the three shapes.

EXERCISE 35

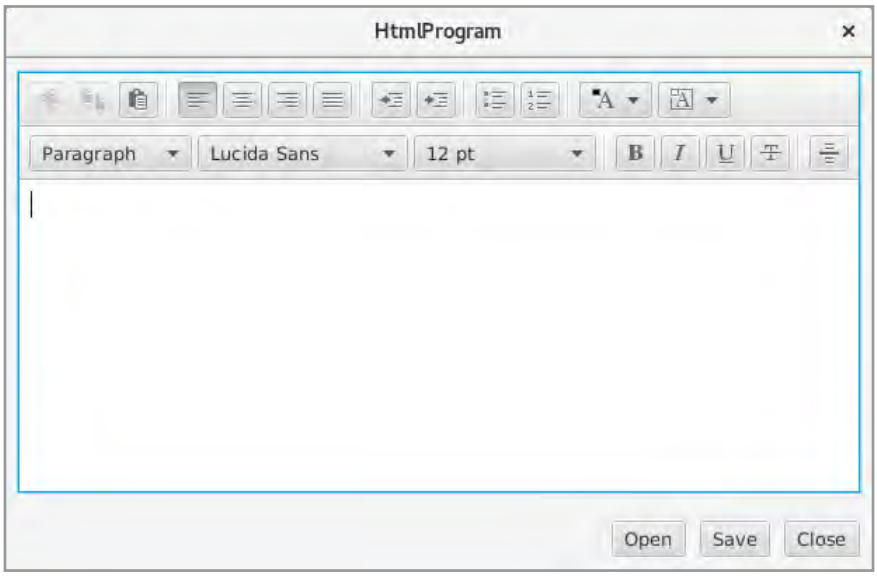
You must write an application that opens the following window:



and thus a program that uses tabs (two tabs). On the first one, you must enter a name and a registration date, while on the other you can enter the address and an email address. If you click on the button, you must get a message box that shows the data entered:

EXERCISE 36

JavaFX has a control called *HTMLEditor*, which is a complete HTML editor. You must write an application that opens the following window where you can edit a HTML document:

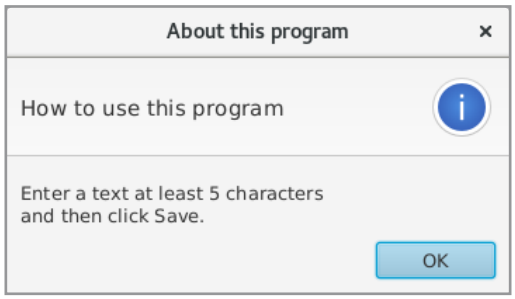


The program must have buttons so you can load and save an HTML document in a local file.

7.4 DIALOGS

You can also use dialog boxes, and there are dialogs as in Swing there are ready to use classes that can be used immediately and, and on the other hand, there are dialog boxes that are custom. The simplest is an *Alert* that corresponds to a *MessageDialog* in Swing. The *AlertProgram* application opens the following window:

If you click on the *Info* button, you get the message box below that shows a simple message and clicking on the *Save* button, you may receive a corresponding message box (but with an error message) and finally the last button will result in a message box with a warning, where you will be asked if you want to delete the entry field:




```
package alertprogram;

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;
import javafx.scene.text.*;
import javafx.scene.text.Text;
import javafx.scene.paint.Color;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.event.*;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import java.util.Optional;
```

```
public class AlertProgram extends Application
{
 private TextField txtField = new TextField();
 private Button cmdInfo;
 private Text status = new Text();

 @Override
 public void start(Stage stage)
 {
  txtField.setPromptText("Enter some text and save.");
  status.setFont(Font.font("Arial", FontWeight.NORMAL, 20));
  VBox vbox = new VBox(20, txtField, createCommands(), status);
  vbox.setPadding(new Insets(20, 20, 20, 20));;
  Scene scene = new Scene(vbox, 400, 180);
  stage.setTitle("AlertProgram");
  stage.setScene(scene);
  stage.show();
  cmdInfo.requestFocus();
 }

 private Pane createCommands()
 {
  HBox pane = new HBox(10, cmdInfo = createButton("Info", new InfoListener()),
   createButton("Save", new SaveListener()), createButton("Clear",
   new ClearListener()));
  pane.setAlignment(Pos.CENTER_RIGHT);
  return pane;
 }

 private Button createButton(String text, EventHandler<ActionEvent> handler)
 {
  Button cmd = new Button(text);
  cmd.setOnAction(handler);
  return cmd;
 }

 private class InfoListener implements EventHandler<ActionEvent>
 {
  @Override
  public void handle(ActionEvent e)
  {
   Alert alert = new Alert(AlertType.INFORMATION);
   alert.setTitle("About this program");
   alert.setHeaderText("How to use this program");
   alert.setContentText(
    "Enter a text at least 5 characters\nand then click Save.");
   alert.show();
  }
 }
```

```
 private class SaveListener implements EventHandler<ActionEvent>
 {
  @Override
  public void handle(ActionEvent e)
  {
   String txt = txtField.getText().trim();
   String msg = "Text is saved";
   boolean valid = true;
   if ((txt.isEmpty()) || (txt.length() < 5))
   {
    valid = false;
    Alert alert = new Alert(AlertType.ERROR);
    alert.setTitle("Error message");
    alert.setContentText(
"Text should be at least 5 characters long.\nEnter a valid text and save. ");
    alert.showAndWait();
    msg = "Invalid text entered";
   }
   status.setText(msg);
   status.setFill(valid ? Color.DARKGREEN : Color.DARKRED);
   if (!valid) txtField.requestFocus();
  }
 }
```

```
 private class ClearListener implements EventHandler<ActionEvent>
 {
  @Override
  public void handle(ActionEvent e)
  {
   Alert alert = new Alert(AlertType.CONFIRMATION);
   alert.setTitle("Warning message");
   alert.setContentText("Are you sure you want to delete the text field");
   Optional<ButtonType> result = alert.showAndWait();
   if ((result.isPresent()) && (result.get() == ButtonType.OK))
   {
    txtField.clear();
    status.setText("");
    cmdInfo.requestFocus();
   }
  }
 }

 public static void main(String [] args)
 {
  launch(args);
 }
}
```

The buttons are created in the method *createCommands()*, and here are the interesting the *ActionEvent* listeners, which are defined as classes at the end of the program. The *InfoListener* class creates an event object for the *Info* button, and in the handler is created an *Alert* object of the type *AlertType.INFORMATION* and the result is the dialog box shown above. You must note how the dialog box is otherwise initialized and how the individual messages appear in the message box. The dialog opens with the method *show()*. The method is not blocking and the program is not waiting for a return value.

The *SaveListener* class defines a listener to the *Save* button and simulates that the entered text is saved. It tests the text length and is the length less than 5, an alert similar to the above opens. The difference is the type as here is *AlertType.ERROR* corresponding to an error message, and then the dialog opens but instead with the method *showAndWait()*. This means that the message box is blocking and the following statement

```
msg = "Invalid text entered";
```

will only be executed after the dialog box is closed.

The last listener class is similar to the others, but this time is the type *AlertType.CONFIRMATION* that opens a dialog box with two buttons, where the user must select *YES* or *NO*. It also

opens with *showAndWait()* and you should note that this time there is a return value and how this value is tested.

There also is a *ChoiceDialog*, where the user can choose a value between several options. The program *ChoiceProgram* opens a window with a button and clicking on the button gives you the following dialog box:

where in a dropdown you can choose a text (in this case the name of a Danish king). If you click OK, the selected name will be returned to the application where it appears in the window. The code is as follows:

```
public class ChoiceProgram extends Application
{
 private List<String> dialogData = Arrays.asList("Gorm den Gamle",
  "Harald Blåtand", "Svend Tveskæg", "Harald d. 2.", "Knud den Store");
 private Text status = new Text();

 @Override
 public void start(Stage stage)
 {
  Button cmd = new Button("Get a King");
  cmd.setOnAction(e -> showDialog());
  HBox command = new HBox(cmd);
  command.setAlignment(Pos.CENTER);
  status.setFont(Font.font("Arial", FontWeight.NORMAL, 20));
  status.setFill(Color.DARKGREEN);
  HBox message = new HBox(status);
  message.setAlignment(Pos.CENTER);
  VBox root = new VBox(30, command, message);
  root.setPadding(new Insets(20, 20, 20, 20));
  Scene scene = new Scene(root, 400, 150);
  stage.setTitle("ChoiceProgram");
  stage.setScene(scene);
  stage.show();
 }
```

```
 private void showDialog()
 {
  ChoiceDialog<String> dialog =
   new ChoiceDialog<String>(dialogData.get(0), dialogData);
  dialog.setTitle("Danish Kings");
  dialog.setHeaderText("Choice a king");
  Optional<String> result = dialog.showAndWait();
  String selected = "cancelled…";
  if (result.isPresent()) selected = result.get();
  status.setText(selected);
 }

 public static void main(String [] args)
 {
  launch(args);
 }
}
```

Initially, a list named *dialogData*, which contains the names of 5 Danish kings, is defined. The method *start()* is simply and build a scene graph containing a button and a *Text* control. The button's event handler calls the method *showDialog()*, which creates a dialog of the type *ChoiceDialog*. The parameters of the constructor are the data to be displayed in the

dropdown field and which one should be selected by default. Otherwise, you should notice how to get the return value and test it.

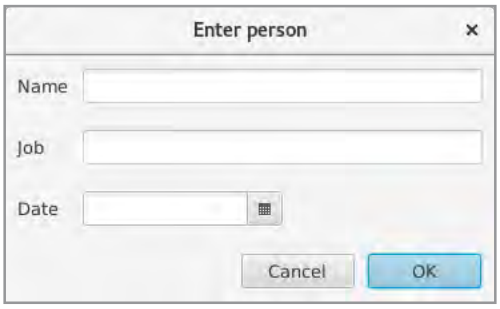
As the last of the standard dialog boxes, you have a *TextInputDialog* that is used to enter a text. The program *InputProgram* opens just like the previous program a window with a button and clicking on the button gives you a dialog box where you can enter a text:

The program is basically identical to the previous one, so I would like only to show the method called from the event handler for the button, and compared to the previous example, there is not much to explain, the fact that the type of the dialog box is now *TextInputDialog*:

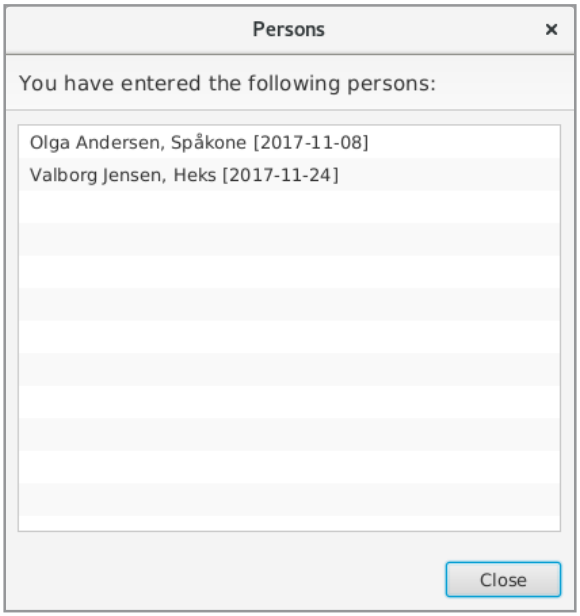
```
private void showDialog()
{
 TextInputDialog dialog = new TextInputDialog("A name?");
 dialog.setTitle("Enter text");
 dialog.setHeaderText("Enter some text.");
 Optional<String> result = dialog.showAndWait();
 status.setText(result.isPresent() ? result.get() : "");
}
```

As the last example in this section, the following application opens two custom dialogs. The program is called *DialogProgram*, and if you run the program, you get the following window:

If you click on the button at the top, you will get a dialog box where you should enter the name and job title of a person and choose a date (which can be interpreted as birthday, appointment date or something else):



If you click the bottom button, the program opens another dialog box that shows the persons entered in the first dialog box (here it is after two persons are entered):



The program looks like a program that I've seen before in connection with Swing (the book Java 2). A person is defined by a simple model class:

```
package dialogprogram;

import java.time.*;

public class Person
{
 private String name;
 private String job;
 private LocalDate date;

 …
```

```
 @Override
 public String toString()
 {
  return String.format("%s, %s [%s]", name, job, date.toString());
 }
}
```

Then there is the code of the program:

```
public class DialogProgram extends Application
{
 private List<Person> persons = new ArrayList();

 @Override
 public void start(Stage stage)
 {
  VBox root = new VBox(20, createButton("Enter person", 150, e -> enterDialog()),
   createButton("Show persons", 150, e -> showDialog()));
  root.setAlignment(Pos.TOP_CENTER);
  root.setPadding(new Insets(20, 20, 20, 20));;
  Scene scene = new Scene(root, 300, 150);
  stage.setTitle("DialogProgram");
```

```
 stage.setScene(scene);
 stage.show();
}

private Button createButton(String text, double width,
 EventHandler<ActionEvent> handler)
{
 Button cmd = new Button(text);
 cmd.setPrefSize(width, 25);
 cmd.setOnAction(handler);
 return cmd;
}

private void enterDialog()
{
 Dialog<Person> dialog = new Dialog();
 dialog.setTitle("Enter person");
 dialog.setResizable(false);
 TextField name = new TextField();
 TextField job = new TextField();
 DatePicker date = new DatePicker();
 name.setPrefWidth(300);
 job.setPrefWidth(300);
 date.setPrefWidth(150);
 GridPane grid = new GridPane();
 grid.setHgap(10);
 grid.setVgap(20);
 grid.setPadding(new Insets(10, 10, 10, 10));
 grid.add(new Label("Name"), 0, 0);
 grid.add(name, 1, 0);
 grid.add(new Label("Job"), 0, 1);
 grid.add(job, 1, 1);
 grid.add(new Label("Date"), 0, 2);
 grid.add(date, 1, 2);
 dialog.getDialogPane().setContent(grid);
 dialog.getDialogPane().getButtonTypes().add(
  new ButtonType("OK", ButtonData.OK_DONE));
 dialog.getDialogPane().getButtonTypes().add(
  new ButtonType("Cancel", ButtonData.CANCEL_CLOSE));
 dialog.setResultConverter(new Callback<ButtonType, Person>()
 {
  @Override
  public Person call(ButtonType b)
  {
   return b.getButtonData() == ButtonData.OK_DONE ?
     new Person(name.getText(), job.getText(), date.getValue()) : null;
  }
 });
```

```
  Optional<Person> result = dialog.showAndWait();
  if (result.isPresent()) persons.add(result.get());
 }

 private void showDialog()
 {
  ListView<Person> view = new ListView();
  view.setPrefSize(400, 300);
  view.getItems().addAll(persons);
  BorderPane pane = new BorderPane(view);
  Dialog<List<Person>> dialog = new Dialog();
  dialog.setTitle("Persons");
  dialog.setHeaderText("You have entered the following persons:");
  dialog.setResizable(true);
  dialog.getDialogPane().setContent(pane);
  dialog.getDialogPane().getButtonTypes().add(
   new ButtonType("Close", ButtonData.CANCEL_CLOSE));
  dialog.show();
 }

 public static void main(String [] args)
 {
  launch(args);
 }
}
```

At the start of the program, a list is defined for *Person* objects, which are the persons created in the input dialog. The method *start()* is trivial and the only thing to note is which event handlers are associated with the two buttons. The first calls the method *enterDialog()*, which creates an object of the type *Dialog* to *Person* objects. The parameter type *Person* specifies which object is returned if the dialog box ends by clicking *OK*. Next, the contents of the dialog box must be defined, and that happens by a *GridPane*, which laid out the dialog box components, and this *Pane* is assigned the dialog box as follows:

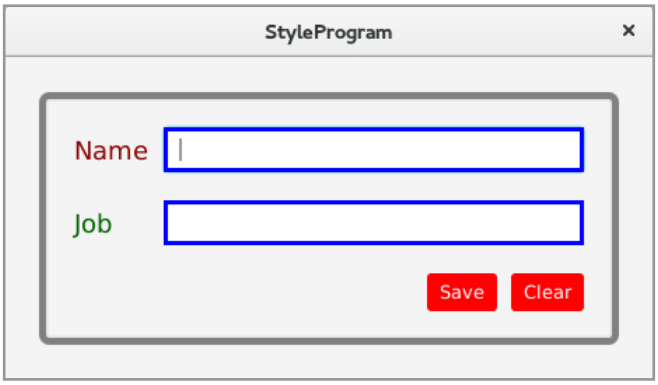
```
dialog.getDialogPane().setContent(grid);
```

The next step defines which buttons the dialog should have, and then is defined how to click on *OK* to create a *Person* object, which must be the return value of the dialog box.

The other dialog box is basically built in exactly the same way, but this time with a *BorderPane*. The main difference is that there is no return value this time. You should note that this dialog box has a header text, while the first one did not, and what a header text means for the result.

8 STYLING

From web applications you know cascading style sheets, and you can also use it in JavaFX. The idea is the same as for web application, namely to separate the definition of windows and their content from the presentation so that you can change the look and feel of a program without changing the code. On the other hand, the syntax is a little different, and as for web applications, the biggest challenge is to learn how styles can be written and what they are called. The subject appears best with an example, and the program *StyleProgram* opens the following window:



where the window contains two *Label* controls, two *TextField* controls and two *Button* controls. For all 6 components there is a style attached. For the *TextField* and *Button* controls, it happens in a style sheet, which is a file named *styles.css*:

```
.button {
 -fx-background-color: red;
 -fx-text-fill: white;
}
.text-field {
 -fx-border-width: 3;
 -fx-border-color: blue;
}
.border {
 -fx-padding: 20;
 -fx-border-style: solid inside;
 -fx-border-width: 5;
 -fx-border-insets: 25;
 -fx-border-radius: 5;
 -fx-border-color: gray;
}
```

Here are three styles defined. In the same way as for web pages, a style is defined with a selector that starts with a point followed by the type on the node in the scene graph, which should have that style – but with the class name written in lowercase. The first, therefore, defines a style for *Button* nodes. Here, two style attributes are defined, and the difficulty is of course to know which attributes can be specified, but in general, you can associate a style value with any of the class's properties. Here is the syntax that if there is a property called *backgroundColor*, the style attribute is named *-fx-background-color*. That is, if the name of an attribute consists of two words, the last one starting with a capital letter, it is all written in lowercase letters and with a hyphen is inserted between the words. The same syntax is used for a selector, and a selector for the class *TextField* is written as:

```
.text-field
```

You can also have a custom selector, what the last style is an example of. It defines a style for the gray frame in the window, and when you see the individual attributes, it is easy enough to understand the meaning. In practice, of course, it is more difficult to find out what to write, and it is important to note that NetBeans knows JavaFX style sheets and which attributes are available.

With respect to the syntax, the following applies to values of the individual attributes:

inherit, where the value is inherited from the parent's node, for example

```
-fx-border-color: inherit;
```

boolean, where the value can be *true* or *false*, for example

```
-fx-display-caret: true;
```

string, where you enter the value in quotation marks, for example

```
-fx-font: normal bold 24px 'areal';
```

number, where you enter the value as a decimal number, for example

```
-fx-border-width: 3.5;
```

You can also specify a unit such as *px* (pixels), *mm* (millimeters), *cm* (centimeters), *in* (inches), *pt* (points), *pc* (picas) or *em*. You can also specify a percent value. Examples could be:

```
-fx-font-size: 18px;
-fx-font-size: 24pt;
-fx-border-width: 10%;
```

angle, which is specified by a number and a unit that must be *deg* (degrees) or *rad* (radians) for example

```
-fx-rotate: 45deg;
```

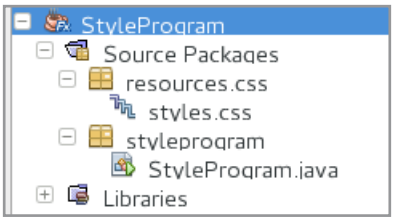
point, which is specified using x and y coordinates as two numbers separated by whitespaces, for example

```
-fx-background-color: linear-gradient(from 0 0 to 100 0, repeat, red, blue);
```

There are several other values that you can specify for properties, and you should examine the documentation for a complete overview, for example

```
https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html
```

Above is shown a style sheet with three styles and it must be somewhere. The project *StyleProgram* consists of the following files:



and the style sheet in question is thus located in the folder *css* under *resources* (that again is located under the *src* directory). It is not a requirement, but can be recommended as the file becomes part of the project's jar file. Then there is the code for the above program:

```
package styleprogram;

import javafx.application.Application;
import javafx.event.*;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;
import javafx.geometry.*;
```

```
public class StyleProgram extends Application
{
 private TextField txtName;
 private TextField txtJob;

 @Override
 public void start(Stage stage)
 {
  HBox commands = new HBox(10,
   createButton("Save", e -> { (new Alert(Alert.AlertType.INFORMATION,
    "Object saved…", ButtonType.CLOSE)).show(); }),
   createButton("Clear", e -> { txtName.clear(); txtJob.clear(); }));
  commands.setAlignment(Pos.CENTER_RIGHT);
  GridPane root = new GridPane();
  root.addRow(0, createLabel(
   "Name", "-fx-font-size: 18; -fx-text-fill: darkred"),
   txtName = createField(300));
  root.addRow(1, createLabel(
   "Job", "-fx-font-size: 18; -fx-text-fill: darkgreen"),
   txtJob = createField(300));
  root.add(commands, 1, 2);
  root.setHgap(10);
  root.setVgap(20);
  root.setPadding(new Insets(20, 20, 20, 20));
  root.getStyleClass().add("border");
  Scene scene = new Scene(root);
  scene.getStylesheets().add("resources/css/styles.css");
  stage.setTitle("StyleProgram");
  stage.setScene(scene);
  stage.show();
 }

 private Label createLabel(String text, String style)
 {
  Label label = new Label(text);
  label.setStyle(style);
  return label;
 }

 private TextField createField(double width)
 {
  TextField field = new TextField();
  field.setPrefWidth(width);
  return field;
 }
```

```
 private Button createButton(String text, EventHandler<ActionEvent> handler)
 {
  Button cmd = new Button(text);
  cmd.setOnAction(handler);
  return cmd;
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

It's a simple program where the components are laid out using a *GridPane*. A style sheet can be assigned to a scene graph or a branch node. In this case, it is assigned to the scene graph:

```
scene.getStylesheets().add("resources/css/styles.css");
```

Here is used a relative path to the style sheet, but you can of course also specify an absolute path. As a result, all of the program's *TextField* and *Button* nodes (unless stated otherwise) will apply the styles defined in the style sheet. The statement

```
root.getStyleClass().add("border");
```

says that the *GridPane* node should apply a style with the selector *border*, and the result is that the gray frame appears.

Styles can also be defined inline, as is the case with the two *Label* controls. It certainly has its uses, but it is generally advisable to define styles in a style sheet, as inline styles can only be changed in the code.

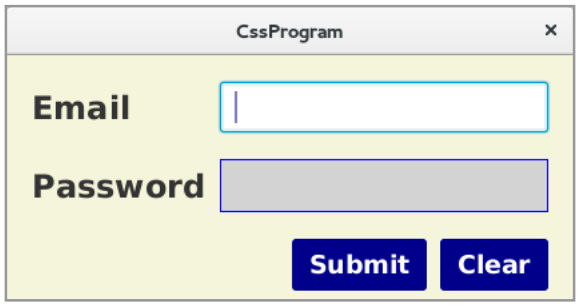
If you in an application considers an attribute, its value can be determined in several places, and the following list indicates which value has the highest priority:

1. inline (the highest priority)
2. parent node
3. scene graph
4. values assigned in the code (Java properties)
5. default style sheet

Here you should especially note that styles have higher priority than values assigned in the usual way in the Java code. Also note the last option with the lowest priority. If you do not specify something, the value of a property is determined by a default style sheet.

EXERCISE 37

You must write an application that opens the following login window:



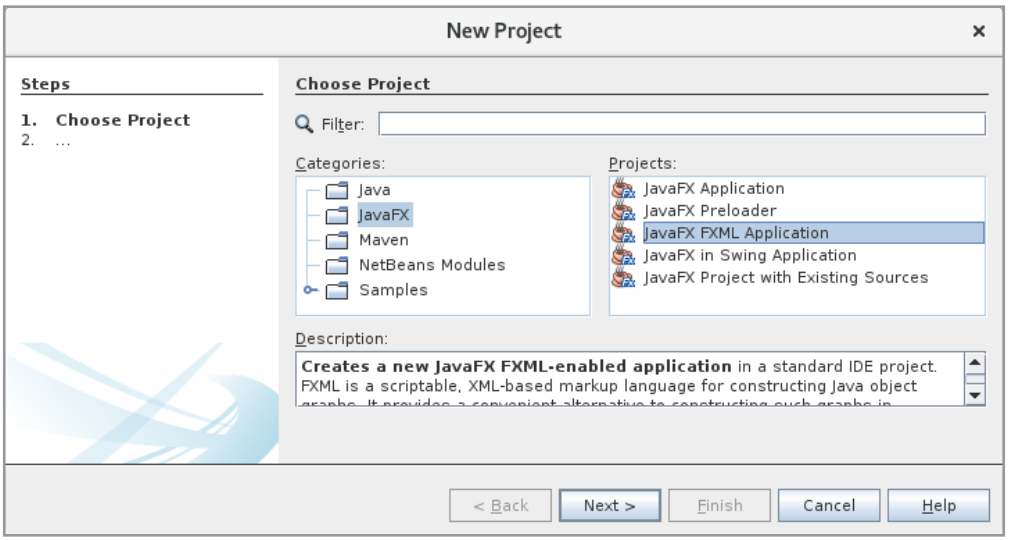
where there are 2 *Label* controls, 1 *TextField* control, 1 *PasswordField* control and 2 *Button* controls. For these controls, the following properties must be set:

1. *Label*: Must have a bold *Arial* font on 18 points
2. *TextField*: Must have a bold *Arial* font on 14 points and the text should be dark blue
3. *PasswordField*: Must have a gray background and a blue frame
4. *Button*: Must have a blue background and a white text with a bold *Arial* font on 14 points

These values must all be assigned using styles defined in a style sheet.

9 FXML

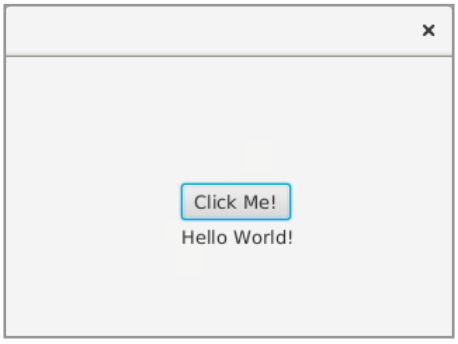
FXML is an XML language that can be used to define the user interface in a JavaFX program and thus it is an alternative to writing the code in Java. With FXML you can define the entire scene graph or just a part of it. If in NetBeans you select *JavaFX* and *JavaFX FXML Application* for *New Project*



and when you click *Next*, you must enter a project name where I have typed *HelloFXML*, and when you click *Finish*, NetBeans creates a project with three files:

and again it is a fully finished program that can be translated and executed:

If you click on the button, you get a window where a label is updated:



Then there is the code and I want to start with HelloFXML.java:

```
package hellofxml;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;
```

```
public class HelloFXML extends Application
{
 @Override
 public void start(Stage stage) throws Exception
 {
  Parent root = FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
  Scene scene = new Scene(root);
  stage.setScene(scene);
  stage.show();
 }

 public static void main(String[] args)
 {
  launch(args);
 }
}
```

Immediately, it is a usual JavaFX application with a method *start()*, and there is only one difference, which is the statement

```
Parent root = FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
```

which loads the file *FXMLDocument.fxml*. It is the XML document that defines the user interface and hence the program's sceen graph. When you see the content, it's almost a common XML document:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
 xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="hellofxml.FXMLDocumentController">
 <children>
  <Button layoutX="126" layoutY="90" text="Click Me!"
   onAction="#handleButtonAction" fx:id="button" />
  <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
   fx:id="label" />
 </children>
</AnchorPane>
```

The document starts with some import directives, which you can easily interpret. Otherwise, the document consists of elements that corresponds to classes for nodes in the scene graph, and even the same names apply. For the individual elements, you can define attributes that match the properties of the corresponding node classes, and here the same names apply. The only exception is the attribute *fx:id*, which refers to a variable defined in the controller class. In addition, note the value of the attribute *onAction* for the *Button* item, which refers to a method (an event handler) in the controller class. The class is as follows:

```
package hellofxml;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController implements Initializable
{
 @FXML
 private Label label;
```

```
 @FXML
 private void handleButtonAction(ActionEvent event)
 {
  System.out.println("You clicked me!");
  label.setText("Hello World!");
 }

 @Override
 public void initialize(URL url, ResourceBundle rb)
 {
 }
}
```

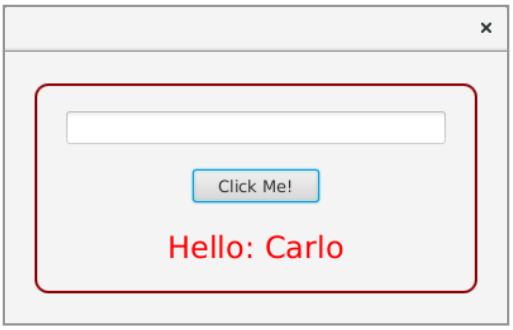
The class defines a variable and two methods. The first is a reference to the control defined in the XML section, and here you should note the annotation *@FXML*. It means that an *FXMLLoader* can create an object defined in FXML by injection. The same notation is used for the event handler for the button, which means it can be referenced from the FXML section. You should note that the class implements the interface *Initializable*, which defines a single method that can be used to initialize the controller.

The above is what NetBeans has automatically created, and I will now have to make changes to *FXMLDocument.fxml* and *FXMLDocumentController.java*. The class *HelloFXML.java* will be unchanged. *FXMLDocument.fxml* has been changed to:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.text.*?>

<VBox alignment="CENTER" spacing="20" prefWidth="400"
 xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="hellofxml.FXMLDocumentController">
 <style>
 -fx-padding: 20;
 -fx-border-style: solid;
 -fx-border-width: 2;
 -fx-border-insets: 25;
 -fx-border-radius: 10;
 -fx-border-color: darkred;
 </style>
 <children>
  <TextField fx:id="name" prefWidth="300" maxWidth="300" />
```

```
  <Button text="Click Me!" prefWidth="100" onAction="#handleButtonAction"
   fx:id="button" />
  <Label minHeight="20" fx:id="label" textFill="red" >
   <font>
    <Font name="Aria" size="24.0" />
   </font>
  </Label>
 </children>
</VBox>
```

Two new import directives have been added, and the *AnchorPane* element has been replaced by a *VBox* element. Then a style element and a *TextField* element are added, and many attributes are also changed. When you see the code, it is easy enough to understand – at least if you know JavaFX. As a result, the window changes to the following:



where the text *Carlo* is entered in the entry field and then clicked on the button. In the controller, nothing else has been added than an additional variable has been added and the event handler has been changed:

```
package hellofxml;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;

public class FXMLDocumentController implements Initializable
{
 @FXML
 private Label label;
 @FXML
 private TextField name;

 @FXML
 private void handleButtonAction(ActionEvent event)
```

```
 {
  label.setText("Hello: " + name.getText());
  name.setText("");
 }

 @Override
 public void initialize(URL url, ResourceBundle rb)
 {
 }
}
```

9.1 CREATE OBJECTS

As illustrated in the previous example, FXML is used to define a program's scene graph or part of the graph, but FXML may more generally be used to define an object graph. In this section, I will show a program called *ObjectsFXML*, which will show you how to create objects using FXML. The *ObjectsFXML.java* program class does not contain anything new and will not be shown here. The file *FXMLDocument.fxml* has the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>
<?import javafx.geometry.*?>
<?import java.lang.*?>
<?import javafx.collections.*?>
<?import objectsfxml.*?>

<VBox prefHeight="200" prefWidth="320" spacing="20" alignment="CENTER"
 xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="objectsfxml.FXMLDocumentController">
 <padding><Insets top="20" right="20" bottom="20" left="20"/></padding>
 <children>
  <Text fx:id="header" />
  <ComboBox fx:id="lstKings" >
   <items>
    <FXCollections fx:factory="observableArrayList">
     <String fx:value="Gorm den Gamle"/>
     <String fx:value="Harald Blåtand"/>
     <String fx:value="Svend Tveskæg"/>
     <String fx:value="Harald den 2."/>
     <String fx:value="Knud den Store"/>
    </FXCollections>
   </items>
  </ComboBox>
  <Button text="Select a king" onAction="#handleKings" fx:id="button" />
  <ComboBox fx:id="lstPersons" />
  <Button text="Select a king" onAction="#handlePersons" />
  <Label fx:id="label" >
   <String fx:value="A unknown Person" />
  </Label>
 </children>
</VBox>
```

For example, if you look at the item

```
<VBox prefHeight="200" …
```

this means that an object of the type *VBox* is being instantiated. First of all, it requires an import directive for the package containing the class *VBox*, and that the class has a default constructor, as the syntax does not contain any immediate way to transfer values to the constructor. Similarly, the element

```
<Text fx:id="header" />
```

means that a *Text* object is created and that it is referred to in the controller of the variable *header*. Here, the object is also assigned a value that I return to below. The next thing that happens is to create a *ComboBox* called *lstKings* (the name that the corresponding object is called in the controller). In particular, note how the *ComboBox* is initialized. This happens with an *ObservableList*, and here the syntax means that the list is created using a factory method, explained below. The individual items in the list are of the type *String* and the item

```
<String fx:value="Gorm den Gamle"/>
```

creates a *String* object by performing a static method *valueOf()* initialized with the current value. It requires that the class of which to instantiate an object has such a method and it has the class *String*. The result of all is that a *ComboBox* is created with 5 *String* objects.

As a next step, a button is created where there is nothing new to explain, and then an empty *ComboBox* called *lstPersons* and finally another button. Finally, a *Label* is created, and here you will notice how this *Label* is initialized with a *String* object. If you run the program, you get the following window:

where a name is selected in the top combobox and then clicked on the top button. The other *ComboBox* is also initialized, and it is done using a factory, as shown below. The *ComboBox* is initialized with Objects of the type:

```
package objectsfxml;

public class Person
{
 private String name;
 private String job;
```

```
public Person(String name, String job)
{
 this.name = name;
 this.job = job;
}

public String getName()
{
 return name;
}

public void setName(String name)
{
 this.name = name;
}

public String getJob()
{
 return job;
}
```

```
 public void setJob(String job)
 {
  this.job = job;
 }

 @Override
 public String toString()
 {
  return String.format("%s, %s", name, job);
 }
}
```

which is a simple model class that I have previously shown. Note that the class has no default constructor, and if you want to instantiate an object, it is necessary to transfer parameters to the constructor. Because you not can do that in FXML, a builder class is used, that is a class that implements an interface:

```
public interface Builder<T>
{
 public T build();
}
```

and a class that implements this interface how to instantiate objects of the type *T*. The class is used by a factory class that implements another interface:

```
public interface BuilderFactory
{
 public Builder<?> getBuilder(Class<?> type);
}
```

The *FXMLLoader* class uses a *BuilderFactory* if it can not create an object otherwise, and it happens bt calling the method *getBuilder()* with the type as argument. If this method returns a *Builder* object, a *FXMLLoader* uses this object to initialize the required properties of the object to be instantiated and eventually called the method *build()* that returns the object. The builder class must have get and set methods for all the constructor's parameters. In this case, you must define a *Builder* for the class *Person* as follows:

```
package objectsfxml;

import javafx.util.*;

public class PersonBuilder implements Builder<Person>
{
 private String name;
 private String job;
```

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

 public void setName(String name) {
  this.name = name;
 }

 public String getJob() {
  return job;
 }

 public void setJob(String job) {
  this.job = job;
 }

 @Override
 public Person build()
 {
  return new Person(name, job);
 }
}
```

A *Builder* is thus a class that solve the problem that you can not construct an object of a class (here *Person*) without transferring parameters to the constructor.

In addition to a *Builder*, you must have a *BuilderFactory*, and in this case it can be written as follows:

```
package objectsfxml;

import javafx.util.*;
import javafx.fxml.*;

public class PersonFactory implements BuilderFactory
{
 private final JavaFXBuilderFactory fxFactory = new JavaFXBuilderFactory();

 @Override
 public Builder<?> getBuilder(Class<?> type)
 {
  if (type == Person.class) return new PersonBuilder();
  return fxFactory.getBuilder(type);
 }
}
```

The method *getBuilder()* has a type as parameter, and if this type is *Person*, a *PersonBuilder* is returned. Otherwise, a default builder will be returned for that type.

For the project *ObjectsFXML*, I have created a directory *resources / fxml*, and I have added a file named *Persons.fxml* whose content are:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import objectsfxml.*?>
<?import java.util.ArrayList?>

<ArrayList>
 <Person name="Hardeknud" job="Konge"/>
 <Person name="Magnus den Gode" job="Konge"/>
 <Person name="Svend Estridsen" job="Konge"/>
 <Person name="Harald Hen" job="Konge"/>
 <Person name="Knud den Hellige" job="Konge"/>
</ArrayList>
```

That is, a XML document that defines 5 *Person* objects, but in order to instantiate these objects, a *Builder* is required. It happens in the controller:

```
public class FXMLDocumentController implements Initializable
{
 @FXML
 private Text header;
 @FXML
 private Label label;
 @FXML
 private ComboBox lstKings;
 @FXML
 private ComboBox lstPersons;

 @FXML
 private void handleKings(ActionEvent event)
 {
  label.setText("" + lstKings.getSelectionModel().getSelectedItem());
  lstKings.getSelectionModel().clearSelection();
 }

 @FXML
 private void handlePersons(ActionEvent event)
 {
  label.setText("" + lstPersons.getSelectionModel().getSelectedItem());
  lstPersons.getSelectionModel().clearSelection();
 }

 @Override
 public void initialize(URL url, ResourceBundle rb)
 {
  header.setText("Objects");
  header.setFont(Font.font("Arial", FontWeight.BOLD, FontPosture.REGULAR, 24));
  try
  {
   lstPersons.getPersons().addAll(
    FXCollections.observableArrayList(loadItems(new PersonFactory())));
  }
  catch (Exception ex)
  {
   System.out.println(ex);
  }
 }

 public ArrayList loadPersons(BuilderFactory builderFactory) throws Exception
 {
  URL url = FXMLDocumentController.class.getClassLoader().getResource(
   "resources/fxml/Persons.fxml");
  FXMLLoader loader = new FXMLLoader();
  loader.setLocation(url);
```

```
  loader.setBuilderFactory(builderFactory);
  return loader.<ArrayList>load();
 }
}
```

Note that at the start of the controller, variables are defined corresponding to elements defined in the FXML document. Next, there are the events of the two buttons that do not require any particular explanation. In the method *initialize()*, the object *header* is initialized by the loader, and the same applies to *lstPersons*, which is the bottom *ComboBox*. This is done by calling the method *loadPersons()*, as with a *FXMLLoader* loads the XML document with the definitions of *Person* objects, using a *BuilderFactory*, that here is a *PersonFactory*.

9.2 DIALOGFXML

I will write a program similar to the program *DialogProgram* from chapter 7, when the difference is that the user interface is written in FXML. I start with a *JavaFX FXML Application* project, which I have called *DialogFXML*. The main program is unchanged, but *FXMLDocument* is changed to the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox prefHeight="200" prefWidth="320" xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="dialogfxml.FXMLDocumentController"
 spacing="20" alignment="CENTER" >
 <children>
  <Button prefWidth="150" text="Enter person" onAction="#handleEnterAction" />
  <Button prefWidth="150" text="Show persons" onAction="#handleShowAction" />
 </children>
</VBox>
```

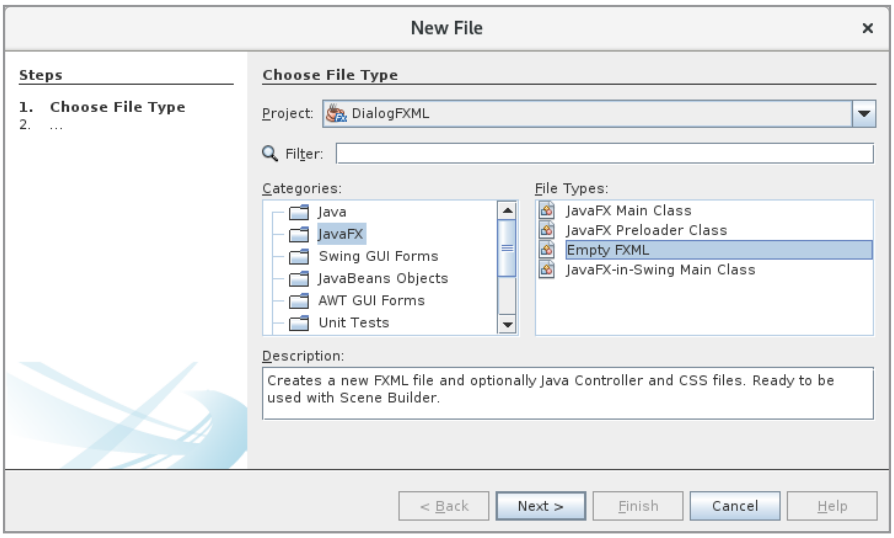
which defines a window with two buttons:

I have then added the class *Person* from the project *DialogProgram*. The window's controller can then be written as follows:

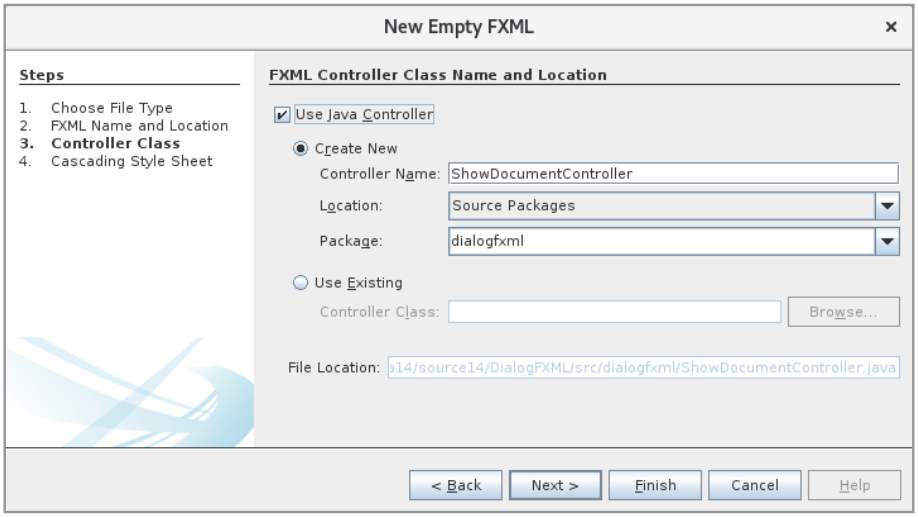
```
public class FXMLDocumentController implements Initializable
{
 private List<Person> list = new ArrayList();

 @FXML
 private void handleEnterAction(ActionEvent event)
 {
```

```
 try
 {
  FXMLLoader loader =
   new FXMLLoader(getClass().getResource("EnterDocument.fxml"));
  Parent root = loader.load();
  ((EnterDocumentController)loader.getController()).setList(list);
  Dialog<Person> dlg = new Dialog();
  dlg.getDialogPane().setContent(root);
  dlg.setTitle("Enter person");
  dlg.showAndWait();
 }
 catch (Exception ex)
 {
 }
}

@FXML
private void handleShowAction(ActionEvent e)
{
 try
 {
  FXMLLoader loader =
   new FXMLLoader(getClass().getResource("ShowDocument.fxml"));
  Parent root = loader.load();
  ((ShowDocumentController)loader.getController()).setList(list);
  Dialog<Person> dlg = new Dialog();
  dlg.setTitle("Persons");
  dlg.setHeaderText("You have entered the following persons:");
  dlg.getDialogPane().setContent(root);
  dlg.showAndWait();
 }
 catch (Exception ex)
 {
 }
}

@Override
public void initialize(URL url, ResourceBundle rb)
{
}
}
```

First, note that a list has been defined for *Person* objects, but otherwise the class consists primarily of the event handlers for the two buttons. In principle, the two methods work in the same way as they starts to load the XML that defines the user interface, and then the list of *Person* objects must be transferred to the controller for the dialog box. Here you should especially note how to refer to the controller. To create a dialog box, I add an Empty FMXL:



After clicking *Next*, I have to enter the name (for example, *ShowDocument*), and in the next window you can choose whether to create a controller:



Finally, there is a window where you can specify whether there should be a stylesheet, and then NetBeans has added two new files. If it is the dialog box for entering a person, they are called *EnterDocument.fxml* and *EnterDocumentController.java*. Below is the FXML code that defines a *GridPane*:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.*?>
```

```
<GridPane id="GridPane" xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="dialogfxml.EnterDocumentController" hgap="10" vgap="20" >
 <padding><Insets top="10" right="10" bottom="10" left="10"/></padding>
 <Label GridPane.rowIndex="0" GridPane.columnIndex="0" >Name</Label>
 <TextField fx:id="txtName" GridPane.rowIndex="0" GridPane.columnIndex="1"
  prefWidth="300" ></TextField>
 <Label GridPane.rowIndex="1" GridPane.columnIndex="0" >Job</Label>
 <TextField fx:id="txtJob" GridPane.rowIndex="1" GridPane.columnIndex="1"
  prefWidth="300" ></TextField>
 <Label GridPane.rowIndex="2" GridPane.columnIndex="0" >Date</Label>
 <DatePicker fx:id="ctlDate" GridPane.rowIndex="2" GridPane.columnIndex="1"
  prefWidth="150" ></DatePicker>
 <HBox GridPane.rowIndex="3" GridPane.columnIndex="1" alignment="CENTER_RIGHT"
  spacing="10" >
  <Button text="OK" onAction="#handleOkAction" />
  <Button text="Cancel" onAction="#handleCancelAction" />
 </HBox>
</GridPane>
```

Here is not much new compared to what has previously been shown, but you should note that a *DataPicker* has been added, which is a node in the same way as other nodes. The controller is as follows:

```
public class EnterDocumentController implements Initializable
{
 private List<Person> list;

 @FXML
 private TextField txtName;
 @FXML
 private TextField txtJob;
 @FXML
 private DatePicker ctlDate;

 public void setList(List<Person> list)
 {
  this.list = list;
 }

 @FXML
 private void handleOkAction(ActionEvent e)
 {
  list.add(new Person(txtName.getText(), txtJob.
  getText(), ctlDate.getValue()));
  ((Stage)((Node) e.getSource()).getScene().getWindow()).close();
 }

 @FXML
 private void handleCancelAction(ActionEvent e)
 {
  ((Stage)((Node) e.getSource()).getScene().getWindow()).close();
 }

 @Override
 public void initialize(URL url, ResourceBundle rb)
 {
 }
}
```

Also, here is not much new, but you should note that a method has been added as *setList()*, so there is access to the list of *Person* objects in the event handler for the *OK* button. You should also note how to close the window by obtaining a reference to the *Stage* object that represents the window. If you run the program and click on the top button, you get the following window:

The XML code of the other dialog box is:

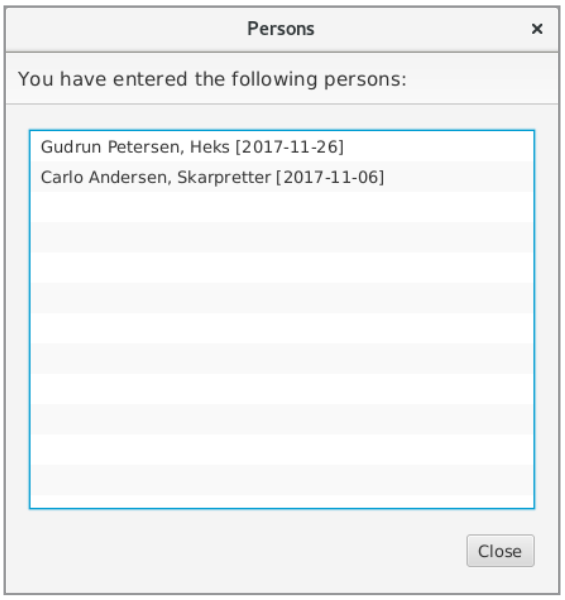
```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.geometry.*?>

<BorderPane xmlns:fx="http://javafx.com/fxml/1"
 fx:controller="dialogfxml.ShowDocumentController">
 <padding><Insets top="20" right="20" bottom="0" left="20"/></padding>
 <center>
  <ListView prefWidth="400" prefHeight="300" fx:id="lstPersons" />
 </center>
 <bottom>
  <HBox alignment="CENTER_RIGHT" >
   <padding><Insets top="20" right="0" bottom="0" left="0"/></padding>
   <Button text="Close" onAction="#handleCloseAction" />
  </HBox>
 </bottom>
</BorderPane>
```

and is basically a *BorderPane*. Note that the center is a *ListView*, which is not initialized here, but is instead done it in the controller, which is trivial:

```
public class ShowDocumentController implements Initializable
{
 @FXML
 private ListView lstPersons;
```

```
 public void setList(List<Person> list)
 {
  lstPersons.getItems().addAll(list);
 }

 @FXML
 private void handleCloseAction(ActionEvent e)
 {
  ((Stage)((Node) e.getSource()).getScene().getWindow()).close();
 }

 @Override
 public void initialize(URL url, ResourceBundle rb)
 {
 }
}
```

Opening the dialog the result is (where two persons are entered):



9.3 ABOUT FXML

This chapter has provided a brief introduction to FXML, and although there is more to explain, the above should be sufficient to use FXML in practice. The question is then what is gained from using FXML and there are two purposes:

1. to separate the design/definition of the user interface from the Java code
2. to make it easier to develop complex user interfaces

The first follows immediately from the separation of the code for a window in an XML part and a controller with the Java code. In principle, it is good, but it requires learning FXML, and although NetBeans provides good support, there is still a lot to learn, and in my opinion, it's at least as easy to write the user interface in Java as in FXML. The separation of the code can easily be achieved by moving the code to event handlers in their own classes in a controller layer. The conclusion is that I do not find the big gains in using FXML, and in fact, I feel that the forces are better used by learning JavaFX in detail than using the power to get good at FXML. It's my opinion and it's not necessarily true and I know that others look different. Therefore, there is only to say, to try and gain your own experiences.

According to what I've mentioned above, FXML does not make it easier to write user interfaces, but it is when using a tool. There is a program called *Scene Builder* that you can use to design the user interface. Instead of writing the code itself, you pull components

from a toolbox and place them in the window where you think they should be. The tool will then form the FXML code for the user interface, and it is in this context that FXML comes into its own. The product *Scene Builder* can be downloaded from

http://gluonhq.com/products/scene-builder/

and it is quite simple to install. Of course, you should learn to use the product, but also it is relatively easy, and if you often need to write complex user interfaces, it is worth knowing the product as it can generate significant progress in development projects.

10 A FINAL EXAMPLE

As the final example, I will use the same program as in the previous book, but with the difference that the program this time should be a desktop application. This means that I can reuse the class library *FunctionsLib*, which implements the mathematical functions, as well as the classes for parsing and evaluating an expression can also be reused. Therefore, I only have to implement the user interface. The aim of the program is, of course, to show an example of a JavaFX program, which is slightly larger than the examples that are otherwise shown in this book.

The program must have the same features as in the previous book and should be basically used in the same way where the user enters a mathematical expression that the program can then evaluates. Unlike the previous program, the program should have buttons for the mathematical functions where a button should do nothing but insert the function name in the input field of the expression. In the previous program you can save a value in a variable and the value can be inserted later in an expression. In the new version of the program you must be able to save an arbitrary expression so that you can insert an expression into an expression, and it should also be possible to save the program's variables in a file.

10.1 DEVELOPMENT

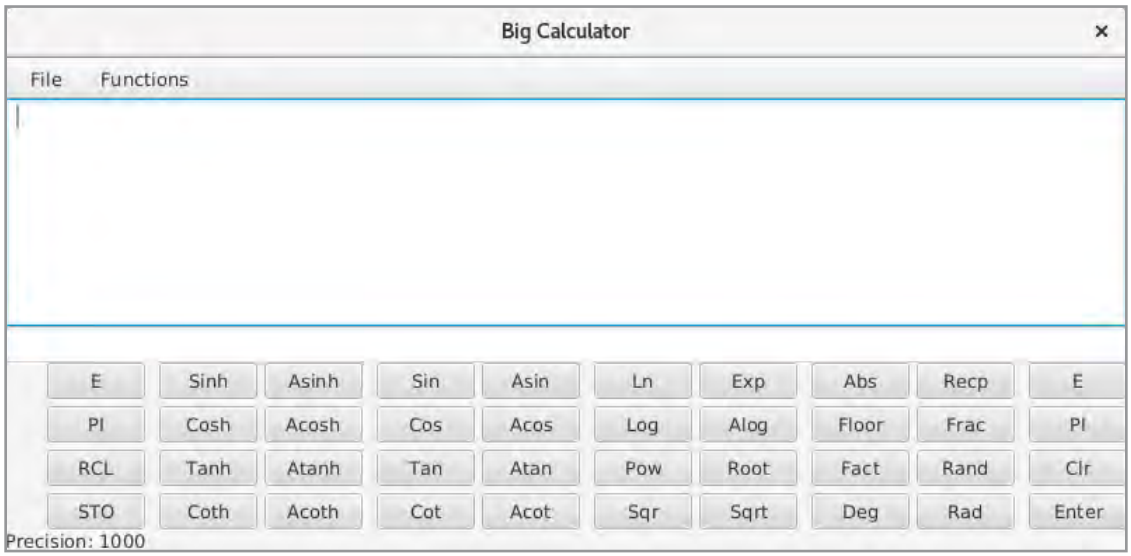
The development of the program has been implemented as a form of prototyping, and in the following I will briefly describe how the development has been completed. From the start, I have planned the following iterations:

1. First prototype
2. Enhanced prototype
3. Implements the square root and square function
4. Implements Store and Recall
5. Implements the other functions
6. Implements Set Precision
7. Implements the menu item Expressions
8. Implements the File menu
9. Implements missing features
10. Styling the program
11. Refactoring

and the goal is that after completing the last iteration, you will have a finished program. The iterations are small, but after each iteration you have a version of the program that can be tested and used with the functions that have been implemented.

First prototype

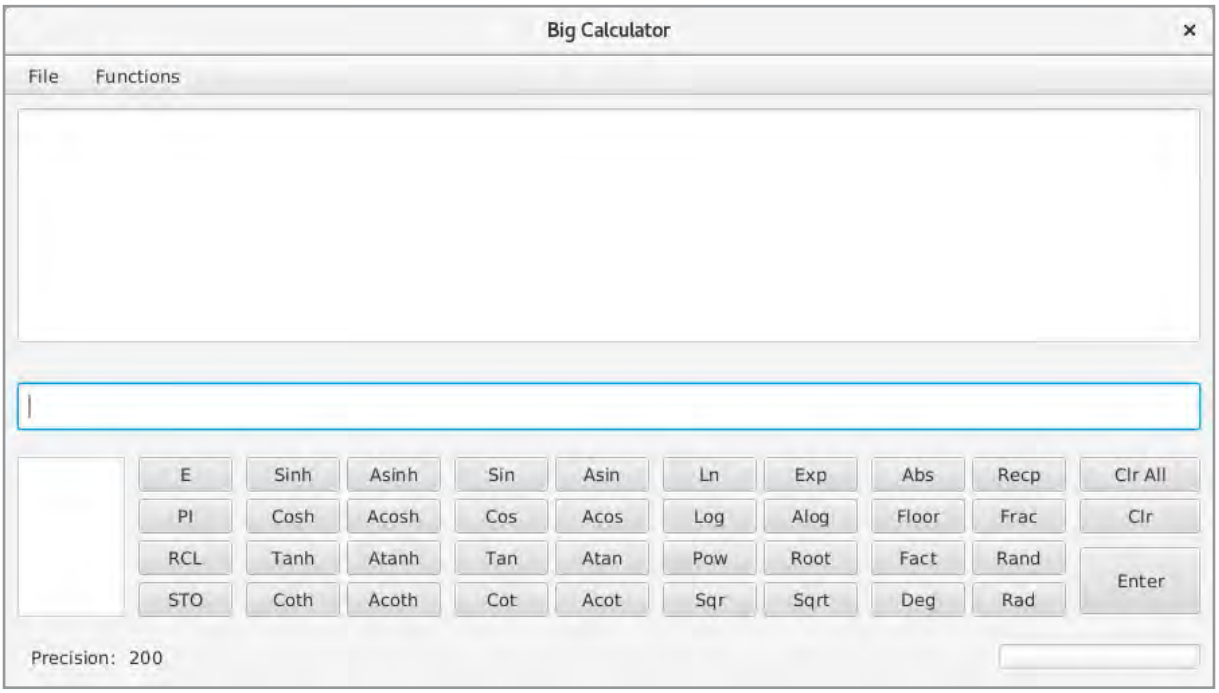
The goal is to write an early version of the program, which should alone be a sketch of the user interface. The program must therefore not have calculating functions.



Apart from the buttons, the program looks like the application from the previous book. The window must be resizable, and the two text fields must follow the size of the window, but the bottom must have a fixed height. The buttons should follow the right edge and bottom of the window. The *File* menu must have functions for saving and loading the variables, while the *Functions* menu must have a function to change the precision as well as a function that shows an overview and the variables that have been created.

Enhanced prototype

The result of the next iteration is still a prototype for the user interface, but with some changes with regard to buttons and with appropriate spacing and padding of the components.


Big Calculator
File
Functions
E
Sinh
Asinh
Sin
Asin
Ln
Exp
Abs
Recp
Clr All
PI
Cosh
Acosh
Cos
Acos
Log
Alog
Floor
Frac
Clr
RCL
Tanh
Atanh
Tan
Atan
Pow
Root
Fact
Rand
Enter
STO
Coth
Acoth
Cot
Acot
Sqr
Sqrt
Deg
Rad
Precision: 200

To the right of the buttons is a list box that displays the program's variables and the size must be that part of the window that is not used for buttons. Additionally, a progress bar has been added because some the calculations may take a long time.

Implements the square root and square function

I have added a reference to the jar file *FunctionsLib.jar* from the final project in the previous book. Then I have created a package *bigcalc.models* and copied the files *Tokens.java* and *Expression.java* from the project *FunctionsEJB* to this package. After that, the entire math should be in place and the required calculation functions should be available.

I have then created a package *bigcalc.ctrls* and here a class *MainController*:

```
package bigcalc.ctrls;

import bigcalc.models.*;

public class MainController
{
 private int precision = 200;

 public String calculate(String text)
 {
  try
  {
   return (new Expression(text, precision)).getValue().toString();
  }
  catch (Exception ex)
  {
   return "Error: " + ex.getMessage();
  }
 }

 public int getPrecision()
 {
  return precision;
 }

 public void setPrecision(int precision)
 {
  this.precision = precision;
 }
}
```

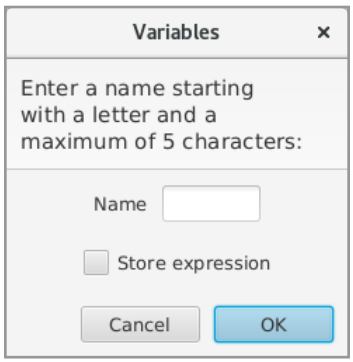
In the class *BigCalc*, event handlers are assigned to the following buttons:

- Sqr
- Sqrt
- Clear Result, which deletes the content of the upper field for the result
- Clear Editor, which deletes the content og the enter field
- Enter

It is all simple event handlers. Then you can perform arithmetic and evaluate expressions, and the calculator is in principle complete. In particular, you can paste text into the editor by clicking the buttons *Sqr* and *Sqrt*.

Implements Store and Recall

If you click the *STO* button, you should get the following dialog box:



where to enter the name of the variable. If you click *OK*, the last result that is calculated is saved. If, on the other hand, you have checked the *Store Expression* check box, it is the expression that is entered in the editor field that is saved. After clicking *OK* and the desired value is assigned to the variable, the list box is updated (the list box to the left of the buttons).

In order to keep track of the variables, the following model class has been added:

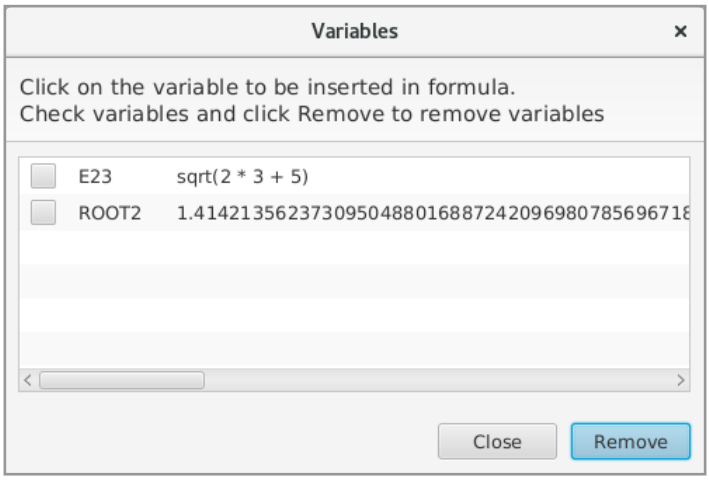
```
package bigcalc.models;

import java.util.*;

public class Memory implements Iterable<String>
{
 private Map<String, String> variables = new TreeMap();
```

```
 public void store(String key, String value)
 {
  variables.put(key, value);
 }

 public String getValue(String key)
 {
  return variables.get(key);
 }

 public boolean remove(String key)
 {
  return variables.remove(key) != null;
 }

 public Iterator<String> iterator()
 {
  return variables.keySet().iterator();
 }
}
```

If you click on a variable in the list box, the corresponding value is inserted in the editor field (at the cursor's position). The same syntax is used as in the previous program, that

is *@A1*, and the difference is that the value can be a value or an expression. This means that the controller must be updated so that values for variables are substituted before the expression is sent to the parser.

If you click the *RCL* button, you get the following window where two variables were created:



If you click on the name of a variable, the window closes and the value of the variable is inserted in the editor field at the cursor's position. If you check one or more checkboxes and click the button *Remove*, these variables will be deleted.

Center in the window is a *ListView* that is created as follows:

```
ListView view = new ListView();
for (String key : memory)
{
 CheckBox check = new CheckBox("");
 check.setPrefWidth(20);
 HBox pane = new HBox(10, check);
 pane.setAlignment(Pos.CENTER_LEFT);
 Label name = new Label(key);
 name.setOnMouseClicked(ev -> selectVariable(key));
 name.setPrefWidth(60);
 Label value = new Label(memory.getValue(key));
 pane.getChildren().add(name);
 pane.getChildren().add(value);
 view.getItems().add(pane);
}
```

Here you should note that a *ListView* can contain anything, and especially a layout pane with components. It is thus an extremely flexible component, and the list box on the main window is initially initialized in the same way.

In principle, this dialog box contains the same information as the list box to the left of the buttons, but the dialog is included partly to delete variables and partly to give a better overview.

Implements the other functions

This iteration is simple as the remaining buttons must do the same as for *Sqr* and *Sqrt*, where only a text must be inserted into the editor field at the cursor's position. The only challenge is that it is the right text.

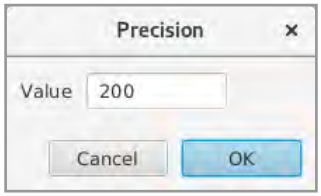
However, the impact of the progress bar must be implemented. I have changed the component to a *ProgressIndicator*, which is constantly running, but from the start is invisible. When a calculation is started (clicked *Enter*), the component is made visible and it becomes invisible again when the calculation is completed and the user interface is updated. The only challenge is get the the user interface (the *ProgressIndicator*) updated, and the calculation must be performed in its own thread:

```
private void calculate(ActionEvent e)
{
 String expression = txtEditor.getText().trim();
 calculator = new Thread()
 {
  public void run()
  {
   lastValue = ctrl.calculate(expression, memory);
   updateResult();
  }
 };
 progress.setVisible(true);
 calculator.start();
}

private void updateResult()
{
 String text = txtDisplay.getText();
 if (text.length() > 0) text += "\n";
 txtDisplay.setText(text + lastValue);
 calculator = null;
 progress.setVisible(false);
}
```

After this iteration, the machine is in principle completed, but with a fixed precision of 200.

Implements Set Precision

This function opens a simple dialog box where you can enter the desired precision:



If you click *OK*, both the controller and the status bar in the main window must be updated. The function has been moved from the menu to a button above the *Enter* key. The menu has been changed so that under *Functions* there is a menu item for each of the program's buttons. The reason is partly to have a better description of the individual functions and to define key combinations for all functions and commands.

Implements the menu item Expressions

This iteration is empty, as the function corresponds to a click on *RCL*, and the function is already implemented. Since *RCL* also has a menu item, the function already exists in the menu.

Implements the File menu

Five functions must be implemented, all located under the File menu:

1. *Save*, which saves all variables in an existing file
2. *Save as*, which saves all variables in a new file
3. *Open*, which opens a file with variables, that ealier are saved
4. *New*, which deletes all existing variables
5. *Exit*, which terminates the program

Variables must be saved with simple object serialization. Therefore, the class *Memory* is partly defined *Serializable*, and in addition, there are added two static methods:

```
public static boolean store(Memory memory, String filename)
{
 try (ObjectOutputStream stream =
  new ObjectOutputStream(new FileOutputStream(filename)))
 {
  stream.writeObject(memory);
  return true;
 }
 catch (Exception ex)
 {
  return false;
 }
}

public static Memory load(String filename)
{
 try (ObjectInputStream stream =
  new ObjectInputStream(new FileInputStream(filename)))
 {
  return (Memory)stream.readObject();
 }
 catch (Exception ex)
 {
  return null;
 }
}
```

Then it's simple to implement the 5 event handlers using a *FileChooser* component.

Implements missing features

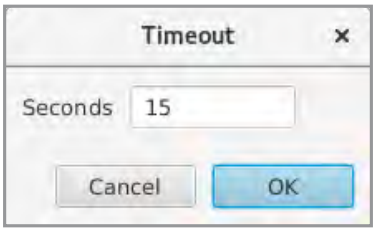
There is not much that are missing, but the following should be added:

1. There should be warnings if you try to overwrite existing variables by loading variables from a file or closing the program and the variables are not been saved.
2. Similarly, there should be a warning if you delete the content of the result field, and accordingly if you delete variables in the dialog that shows the list of variables (the command RCL). The same applies to the dialog boxes for entering the name of a variable and the dialog for change the precision if you enter an illegal value.
3. If you use many significant digits (heigh precision), a calculation can take a long time and therefore a timeout should be defined so that the program giving up after the timeout.

The first point is simple to solve and is primarily about include a *boolean* variable that keeps track of changes in the program's variables. The problem is thus limited to ensuring that this variable is set at the correct places and then test the variable in the event handlers for the *File* menu. Additionally, you can override the *stop()* method and test the variable there so that you do not close the program without a warning if there are non-saved variables.

The second point is similarly simple and refers to displaying a confirmation box in the event handler that deletes the content of the result field, as well as a test in the three dialog boxes.

The last problem, on the other hand, is not quite simple to solve. First I have in the controller class (*MainController*) included a property *timeout*, which indicates the number of seconds for the timeout value. It is as default 15. Next, I have expanded the program with the following dialog box:



where you can enter the timeout value. There is no button for the dialog, but it is opened from the menu or with a shortcut key.

When you click the *Enter* button in the main window, the following events are performed:

```
private void calculate(ActionEvent e)
{
 String expression = txtEditor.getText().trim();
 Thread calculator = new Thread()
 {
  public void run()
  {
   cmdCalc.setDisable(true);
   ExecutorService executor = Executors.newFixedThreadPool(1);
   Future<Void> task = executor.submit(new Calculator());
   try
   {
    task.get(ctrl.getTimeout(), TimeUnit.SECONDS);
   }
   catch (Exception ex)
   {
    lastValue = "Timeout";
   }
   executor.shutdownNow();
   Platform.runLater(new Runnable()
   {
```

```
    @Override
    public void run()
    {
     updateResult();
     cmdCalc.setDisable(false);
    }
   });
  }
 };
 progress.setVisible(true);
 calculator.start();
}
```

It looks a bit complex. The calculation is performed by a method in an *Expression* object in the controller and is the method that takes time, and if executed directly, it will block the main window, which will mean that the progress indicator does not run. The calculation must therefore be performed in its own thread, which is the thread *calculator*. In Java, you can not directly interrupt a running thread, which must happen in the case of a timeout. The problem can be solved with an *Executor* and a *Future* object (possibly see Java 8). The thread *calculator* therefore starts with disabling the button *Enter*, and then an *Executor* object is created that is used to start a task of the type *Calculator*:

```
class Calculator implements Callable<Void>
{
 public Void call()
 {
  try
  {
   lastValue = ctrl.calculate(txtEditor.getText().trim(), memory);
  }
  catch (Exception ex)
  {
   lastValue = "Error";
  }
  return null;
 }
}
```

It is then its *call()* method that performs the calculation. After this task has been created (in the thread *calculator*) the thread is set to wait for the task to be performed or a timeout occurs. In the case of the last the result is defined as the text *Timeout*, after which is executed an

```
executor.shutdownNow();
```

This causes that threats performed by the task to terminate, and even if it is still running due to a timeout. Hereafter the following method is called which updates the result:

```
private void updateResult()
{
 String text = txtDisplay.getText();
 if (text.length() > 0) text += "\n";
 txtDisplay.setText(text + lastValue);
 progress.setVisible(false);
 txtEditor.requestFocus();
 position = txtEditor.getText().trim().length();
 txtEditor.selectRange(position, position);
}
```

and the button will be enabled agian. After the thread *calculator* is defined, it is started and the result is that the calculation is performed in a thread (of a task object) that is started by the thread *calculator* that starts when clicking the *Enter* button.

Styling the program

After the above iterations, all windows and nodes are shown on basis of the default styles in JavaFX, and often, by a program of the current kind, it will be quite good, but if you want to put your own touch on how the components appear or for some reason want to change the specific components look and feel, you can add a style sheet. It is preferable not to directly assign values to properties in the code, as a style sheet makes it much easier to maintain the code.

There are several strategies, but you can proceed as follows. The program (the main window) has the following groups of components:

- 1 *TextArea* for the display (the result)
- 1 *TextField* to the editor field
- 1 *ListView* to variables
- 1 *Button* to the *Enter* button
- 5 *Button* controls to commands (*Clr Result*, *Clr Editor*, *Precision*, *RCL*, *STO*)
- 2 *Button* controls to consts (*Pi*, *e*)
- 36 *Button* controls to functions
- 7 *Menu* controls
- 41 *MenuItem* controls
- 1 *ProgressIndicator*

- 1 *Label* to the text in the status line
- 1 *Label* to precision
- 1 *Label* to the filename
- 1 *Label* to the variable name in the list box
- 1 *Label* to the variable valie in the list box

You can then in a style sheet define a class (an empty class) for each of these groups (15 classes in total) and then associate these classes with the individual components. You can then write the styles you want to use and the advantage is that you can change the presentation of controls without changing the Java code, and as the most important thing, you can delegate the work to developers with special skills in styling.

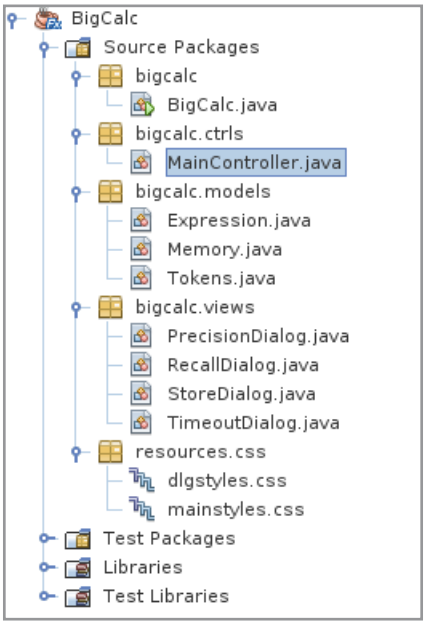
In this case, I have added a style sheet named *mainstyles.css* and written a few styles – and maybe even most for the example's sake.

You can then repeat it for all the application's windows and dialogs, and maybe use the same style sheet for all or most dialogs. That way, you only have to write styles somewhere, and all of the application's dialog boxes have the same look and feel. In this case, I have added a style sheet named *dlgstyles.css*, which is used by all dialogs. In many examples, you

may want to use a single style sheet, which is used by all windows, and of course (at least for small programs) it may be appropriate.

Refactoring

Now the program is in principle complete and consists of the following files:



I have previously argued that in this place you should conduct a code review, inspecting the code, adding comments, removing disabilities, and more. This code review often has character of a refactoring, which may result in code being moved to new classes or reorganizing the code in another way. Especially if the program is developed through many small iterations with the character of prototyping as in this case, there is a great risk that it may go beyond the architecture, and in particular there is a risk that the class of the main window swells up and becomes inappropriately design. For the sake of future maintenance, therefore, refactoring is important, and you should focus on the following:

- If there are *import* statements that are no longer necessary (NetBeans shows which ones), they should be removed.
- If there are variables that are no longer used, they should also be deleted.
- If there is code (such as methods) that are not used, it should be removed.
- Has instance variables and methods appropriate names. If not, consider changing the names so that they better reflect the use.
- Consider visibility, so only what it's going to be *public* is *public* while the rest is *private* and possibly *protected*.

- Are variables and properties located in the right place (in the right classes) and here you must have the MVC pattern in mind. Often the user interface will contain variables that should be moved to the controller or model.
- Pay special attention to code repetition where the same code is written in several places. Such a code should typically be moved to its own method, and the code repetitions can then be replaced by the call of this method.
- Pay attention to the size of classes. If classes are very large, such as many hundred of lines or more, you might want to consider splitting up the class into several classes. Not that you must necessarily, but a class on a thousand lines or more is hard to overlook.
- Be aware of event handlers, as they tend to contain code related to business logic, and if necessary, this code should be moved to controller classes (possibly to the model). In general, event handlers should be as simple as possible and contain only code that directly concerns the user interface. It's a tough balance, since delegation of the code to the controller and model layers can also lead to code that is difficult to read and that the user interface needs to be observer for the controller and model. However, it should be noted that the user interface should contain only what is necessary to update the components.

And so: Refactoring is an important activity for future maintenance, but make sure that the process does not introduce new errors in the program. The program must then be tested again after the activity is completed.

And then there are the comments, which are also part of the refactoring process or rather the maintenance of the comments. I have previously argued the importance of comments and especially the process of writing the comments. These views are still valid, but if you see my program examples (what the readers hopefully do), I have to admit that there is a long line between the comments. Should the truth come true, I must also admit that I am a bit ambivalent with regard to comments: I know it's right to comment on the code, but I do not do that often, so some remarks about this problem.

It's a big job to write comments – at least if you have to make it complete, and then the comments also has to tell you something. You must therefore necessarily refer to whether it is worth the work. Comments documents the code and it is certainly important, but I find that good program architecture and good names for variables and methods are far more important – it is also a form of comments. The goal of all is to make the code easier to read and understand, and here I find that the things I have mentioned above regarding refactoring, exceeds the value of comments. Comments have definitely their applications, but with good names and a good architecture, there is a tendency for comments to commenting

on self-esteem and comments make comments for their own sake, and if so, it's not worth the effort – then there is hardly anyone who will pay for the assignment.

Now it's probably not either, but to explain a little about my wings, I will mention that when I have to read and understand foreign code, one of the first things I do is to delete all comments. They simply seem disturbing to me – at least if it's a nice code. Is that the way it matters, of course, it does not make much sense of writing comments. Of course, it also has something to do with which code it is, where comments of the code in the user interface are rarely valuable, where comments of complex algorithms in control and model days may be required.

I know that not everyone will agree with these views, but at least they explain why my code often lacks comments, which also reflects that much of what I should write as comments is a part of the text in the books.