Poul Klausen

# JAVA 15

## More about JavaFX

Software Development

bookboon
The eBook company

POUL KLAUSEN

# JAVA 15: MORE ABOUT JAVAFX

## SOFTWARE DEVELOPMENT

Java 15: More about JavaFX: Software Development
1st edition
© 2018 Poul Klausen & bookboon.com
ISBN 978-87-403-2200-2
Peer review by Ove Thomsen, EA Dania

# CONTENTS

**Foreword** **7**

**1 Introduction** **9**

**2 JavaFX properties** **10**
2.1 Binding properties 20
Exercise 1 26
2.2 Observable collections 28
2.3 Binding observable collections 35
Exercise 2 41
2.4 Binding persons 43
2.5 The screen 47
2.6 Decorations 50
2.7 Modality 53
Problem 1 56

Free eBook on
Learning & Development

By the Chief Learning Officer of McKinsey

Download Now

bookboon
Prof. Dr. Nick H.M. van Dam
21st Century Corporate Learning & Development
Trends and Best Practices

**3 Advanced controls 59**
3.1 TableView 59
Exercise 3 71
3.2 Edit cells in a TableView 71
Problem 2 83
3.3 Filters 85
Exercise 4 87
3.4 A TreeView control 88
Exercise 5 96
3.5 A TreeView with Country objects 98
3.6 A TreeTableView 103
3.7 A TreeTableView, an extended example 110

**4 Drag and drop 114**
4.1 Simple press-drag-release gesture 115
4.2 Full press-drag-release gesture 118
4.3 Drag-and-drop gesture 120

**5 MVC 134**

**6 User defined controls 144**
6.1 A LabelField 146
6.2 A Canvas 148
6.3 A Spinner 150

**7 JavaFX and concurrency 153**
7.1 A Task 156
7.2 A Service 162

**8 3D Shapes 163**
8.1 Box, Sphere and Cylinder 166
8.2 Material 169
8.3 Draw mode 171
8.4 Cull face 172
8.5 Camera and Light 175
Exercise 6 181
Exercise 7 182
8.6 A last remark 182

**9 Charts 183**

**10 Final Example 191**
10.1 Development 192
10.2 A simple prototype 193
10.3 Drawing the axes 194
10.4 Settings for the coordinate system 196
10.5 Drawing a function from a formal 198
10.6 The program architecture 200
10.7 Drawing a plot 202
10.8 Refactoring the expression dialog 205
10.9 Implementing the Functions menu 207
10.10 Implementing the Zoom menu 212
10.11 Implementing the Edit menu 213
10.12 Implementing the Calculations menu 218
10.13 Implementing the File menu 218
10.14 A final iteration 224
10.15 A last remark 226

# FOREWORD

This book is the fifteenth in a series of books on software development and the book is a natural continuation of the previous book on programming of GUI applications with JavaFX. The book focuses primarily on JavaFX properties and data bindings, but also treats the basic architecture of a JavaFX application as Model-View-Presenter. Other important topics are advanced controls like TableView and TreeView and also charts and 3D graphics are mentioned. The book requires knowledge of JavaFX corresponding to what is dealt with in the book Java 14. After reading this book and solving the corresponding exercises and tasks, you should be able to write completed GUI applications using JavaFX and use JavaFX as a alternative to Swing.

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

This book is an immediate continuation of the previous book on JavaFX, which was not included due to the number of pages, or topics that are more technical:

- Properties and databinding
- Advanced controls
- Drag and drop
- Architecture and MVC
- User defined controls
- JavaFX and concurrency
- 3D Shapes
- Charts
- Final example

You can also say that the book completes what is needed to write Java desktop applications with a graphical user interface.

As shown in the above list, this book includes many topics, but apart from the first about properties and binding, the individual topics do not matter to each other and you can read the topics in line with current needs. The book should therefore show how to solve concrete problems as illustrated by examples, and compared to the previous book there will be only a few exercises.

# 2 JAVAFX PROPERTIES

Properties and beans have been mentioned several times in the previous books, and all classes in JavaFX defines properties that you can read or modify as a programmer. In a typical bean, a property is nothing but *get* methods and possibly *set* methodd that represents a property of the current class, and typically corresponds to a variable with associated *get* and *set* methods. In addition, a Java Bean uses a particular name convention. In JavaFX, however, a property is a bit more, as it is sometimes necessary to know. All properties in JavaFX are observable, where an observer can receive notifications regarding invalidations and changes. There is a strict distinction between read/write properties and read-only properties. The value of a property can be either a single value or a collection.

Another important difference is that a property is always an object whose class is part of a particular class hierarchy for properties. As examples can be mentioned

- *IntegerProperty*
- *DoubleProperty*

(and there are corresponding classes for the other simple data types). These classes are actually abstract, and for each class there are two specific classes that for an *IntegerProperty* are

- *SimpleIntegerProperty*
- *ReadOnlyIntegerWrapper*

where the first represents a read/write property, while the other represents a read-only property. A property class defines as usual *get* and *set* methods, but the class also defines two methods called *getValue()* and *setValue()*. For primitive types, the *get* and *set* methods works on primitive types (*int*, *double* and so on), while the other two works on objects (such as *Integer*, *Double*, etc.). For reference types like

- *StringProperty*
- *ObjectProperty<T>*

all four methods works on objects, and because of autoboxing, there is no difference in practice, for example on *get* and *getValue()*. As an example, below is shown a method that creates a read/write *IntegerProperty*

```
private static void test01()
{
 IntegerProperty p = new SimpleIntegerProperty(100);
 System.out.println(p.get());
 p.set(200);
 System.out.println(p);
}
```

and the method (which is a metodin the project *FXProperties*) is performed you get the result:

```
100
IntegerProperty [value: 200]
```

Here you should note the last line that is the result of *toString()* in the class *SimpleIntegerProperty*, while the first prints the value of an *int*.

Read-only properties are actually a bit more complex as they are a wrapper about two properties that the system ensures is synchronized. Here, the one is a read-only property, while the other is a read/write. The idea is that from the outside it is a read-only property, but it can be changed internally from the class in which it is defined, and the application is typical as a private instance variable.

As an example, below is shown a class with two JavaFX properties:

```
package fxproperties;

import javafx.beans.property.*;

public class Counter
{
 private IntegerProperty step = new SimpleIntegerProperty(1);
 private ReadOnlyIntegerWrapper value = new ReadOnlyIntegerWrapper(0);

 public final IntegerProperty stepProperty()
 {
  return step;
 }

 public final int getStep()
 {
  return step.get();
 }
```

```
public final void setStep(int step)
{
 this.step.set(step);
}

public final ReadOnlyIntegerProperty valueProperty()
{
 return value.getReadOnlyProperty();
}

public final int getValue()
{
 return value.get();
}

public void increase()
{
 value.set(value.get() + step.get());
}
```

www.sylvania.com

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA

```
 public void decrease()
 {
  value.set(value.get() - step.get());
 }

 @Override
 public String toString()
 {
  return "" + value.get();
 }
}
```

This is an example of a JavaFX bean. The two properties are called *step* and *value* and are a read/write property and a read-only property. For both properties is defined a method that returns the value, for example:

```
public final IntegerProperty stepProperty()
{
 return step;
}
```

Here you should note the name conventions, which is the name followed by the word property, and furthermore that it is defined final. Finally, there are defined common *get* and *set* methods (which are also defined final). In principle, they are not necessary, but it is also part of the convention, to be in accordance with the usual Java beans convention. You should note that the methods *increase()* and *decrease()* perform a set method on the property *value* – even if it is readonly. That's okay, because it takes place internally in the class *Counter*. Below is a method that uses the class:

```
private static void test02()
{
 Counter counter = new Counter();
 System.out.printf("%d %s\n", counter.getValue(), counter);
 for (int i = 0; i < 10; ++i) counter.increase();
 System.out.printf("%d %s\n", counter.getValue(), counter);
 counter.setStep(10);
 for (int i = 0; i < 10; ++i) counter.increase();
 System.out.printf("%d %s\n", counter.getValue(), counter);
// counter.valueProperty().setValue(10);
}
```

There is not much to note, but you must note the last statement, which is a comment, and the statement can not actually be translated, since *valueProperty()* returns a read-only property and therefore does not have a *set()* method.

A property class encapsulates three values

1. a reference to the bean, that the property belongs to
2. a name, that is only a String
3. a value

and the class has constructors, so you can initialize these values as desired. For example, for the class *SimpleIntegerProperty*:

- *SimpleIntegerProperty()*
- *SimpleIntegerProperty(int value)*
- *SimpleIntegerProperty(Object bean, String name)*
- *SimpleIntegerProperty(Object bean, String name, int value)*

and a property has a *getBean()* and a *getName()* method. To show another example of the pattern that JavaFX uses regarding properties, below is shown a class, which is a normal bean:

```
package fxproperties;

import javafx.beans.property.*;

public class King
{
 private ReadOnlyStringWrapper name = new ReadOnlyStringWrapper(this, "name");
 private IntegerProperty from = new SimpleIntegerProperty(this, "from");
 private IntegerProperty to = new SimpleIntegerProperty(this, "to", 9999);

 public King()
 {
 }

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

 public final String getName()
 {
  return name.get();
 }
```

```
public final ReadOnlyStringProperty nameProperty()
{
 return name.getReadOnlyProperty();
}

public final int getFrom()
{
 return from.get();
}

public final void setFrom(int from)
{
 this.from.set(from);
}

public final IntegerProperty fromProperty()
{
 return from;
}

public final int getTo()
{
 return to.get();
}
```

360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

```
 public final void setTo(int to)
 {
  this.to.set(to);
 }

 public final IntegerProperty toProperty()
 {
  return to;
 }
}
```

The class has three properties, the first being a read-only property, while the others are read/write properties. You should note that in JavaFX, properties are not created as primitive types, but as *Property* objects. You define *get* and *set* methods in the usual way and with the same names convention for what is standard for Java beans. You should note how these methods refer to the individual properties, and that they are defined final. It is part of the pattern that JavaFX uses. For each property, a method is also defined that returns the actual property object, such as *fromProperty()*. It is also a part of the pattern. Finally, you should note that *name* is a read-only property and therefore has no set method. Also note that the method *nameProperty()* returns a *ReadOnlyStringWrapper* and not a *StringProperty*.

That properties as illustrated above are class types means that many objects must be instantiated. The reason that a property is defined as an object type is a number of advanced features that are associated with a property (as fire events and binding), but often they are not used, and therefore JavaFX uses largely *lazy updating*, where an object is first instantiated when needed. The principle can be illustrated with the class *King* as follows:

```
public class King
{
 private static final String DK = "DK";
 private StringProperty country;
 …

 public String getCountry()
 {
  return country == null ? DK : country.get();
 }

 public void setCountry(String country)
 {
  if (country != null || !DK.equals(country)) countryProperty().set(country);
 }
 public StringProperty countryProperty()
 {
  if (country == null) country = new SimpleStringProperty(this, "country", DK);
  return country;
 }
}
```

where the class now has an additional property, representing the name of the country from where the king is from. If the vast majority of kings come from Denmark, and when you only need to read this value, it is not appropriate (for the sake of performance) to instantiate a *Property* object for each *King* object. Therefore, the variable *country* from the start is *null* and an object is first instantiated if you try to change the value to something else than *DK* or if you refer to the property with *countryProperty()*.

The above may seem overwhelming, but the reason is that properties in JavaFX are observable and may be associated with an *InvalidationListener* and a *ChangeListener <? super T>*. The first interface defines a method *invalidated()*, while the other defines a method *changed()*. When the value of a property changes from valid to invalid, an invalidation event is generated. When it does not necessarily happens every time the value of a property changes (or otherwise becomes invalid), it is to avoid generating a line of invalidation events. *Change* events are, however, generated each time the value of a property is changed. The following method will illustrate when these events occurs:

```
private static void test03()
{
 IntegerProperty p = new SimpleIntegerProperty(100);
 p.addListener(FXProperties::invalidated);
//    p.addListener(FXProperties::changed);
 System.out.println("Set to 101");
 p.set(101);
 System.out.println("Changed to 101");
 System.out.println("Set to 102");
 p.set(102);
 System.out.println("Changed to 102");
 System.out.println("Set to 102");
 p.set(102);
 System.out.println("Changed to 103");
 System.out.println("Set to " + p.get());
 p.set(103);
 System.out.println("Changed to 103");
}

public static void invalidated(Observable p)
{
 System.out.println("Property is invalid");
}
```

```
public static void changed(ObservableValue<? extends Number> p,
 Number oldValue, Number newValue)
{
 System.out.println("Property is changed");
}
```

At the bottom, the event handlers are defined for an *invalidation* event and a *change* event, respectively. You must note the parameters where the first has a reference to the property that has fired the event in question. The other has a similar reference to the property, and also the old value and the value after the change. Note that the type of the value is *Number*, which is the base class for an *Integer*. If you performs the function (without removing the comment) the result is:

```
Set to 101
Property is invalid
Changed to 101
Set to 102
Changed to 102
Set to 102
Changed to 102
Set to 103
Property is invalid
Changed to 103
```

SIMPLY CLEVER

ŠKODA

We will turn your CV into
an opportunity of a lifetime

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

and that is, the event handler *invalidated()* is performed twice. The function creates a read/write *IntegerProperty* with the value 100, and is then valid. Then an event handler for invalidate events is added. When the value of *p* changes to 101, the property becomes invalid and an invalidate event is triggered. When the value changes to 102, an event is not fired because it is already invalid. Then the method reads the property in question (in *System.out.println()*), which, among other things, means that it becomes valid. When it is subsequently set to 102, no invalidation event is fired, as the value is not changed. This happens when the value changes to 103.

If you then remove the comment and set a comment in front of the first *addListener()*, instead, a *ChangeListener* is added, and if you then performs the method you get the result:

```
Set to 101
Property is changed
Changed to 101
Set to 102
Property is changed
Changed to 102
Set to 102
Changed to 102
Set to 103
Property is changed
Changed to 103
```

Note that a change event is performed each time the value changes, but not when the value is set to 102 the second time as the value is not changed.

Another difference between invalidation events and change events is that JavaFX for invalidation events uses *lazy evaluation*, while change events use *eager evaluation* as a value must be transferred to the event handler.

To some extent, one can achieve the same with an invalidation event and a change event, but an invalidation event is slightly more effective as it is not necessarily fired whenever the value changes and as it does not necessarily update the value. If you want to choose which event you should listen to, the main rule is that if you do not read the value in the event handler, you should listen to invalidation events, but if you read the value, you should instead listen to a change event as it automatically updates the value (and thus validates the property). You can see the effect if you in the above program removes both comments:

```
Set to 101
Property is invalid
Property is changed
Changed to 101
```

```
Set to 102
Property is invalid
Property is changed
Changed to 102
Set to 102
Changed to 102
Set to 103
Property is invalid
Property is changed
Changed to 103
```

The result is that an invalidation event is firing every time the value changes.

## 2.1 BINDING PROPERTIES

JavaFX supports binding of properties where a property of a component or node can bind to a property on another object. In general, it is relatively simple to use, but in fact, many details are assigned to the concept, however, the classes in JavaFX are hidden the most. In the following, I will illustrate with some simple examples what binding is about. Consider the following method:

```
private static void test04()
{
 IntegerProperty x = new SimpleIntegerProperty(3);
 IntegerProperty y = new SimpleIntegerProperty(5);
 IntegerProperty z = new SimpleIntegerProperty(7);
 z.bind(x.add(y));
 System.out.println("Bound = " + z.isBound() + ", z = " + z.get());
 x.set(11);
 y.set(13);
 System.out.println("Bound = " + z.isBound() + ", z = " + z.get());
 z.unbind();
 x.set(17);
 y.set(19);
 System.out.println("Bound = " + z.isBound() + ", z = " + z.get());
}
```

The method defines three properties of the type *IntegerProperty* initialized with values 3, 5 and 7. An *IntegerProperty* has an *add()* method that performs an addition of values of two properties and returns a binding of the sum of the two properties which is an expression of the type *NumberBinding*, representing the sum of the values of the two properties. This binding is then linked to the property *z*. The next statement will print

```
Bound = true, z = 8
```

which states that the variable *z* is bound, and that its value is 8 and hence the sum of *x* and *y*. That is, the value of *z* has been changed. Next, the two properties *x* and *y* are changed, and the next statement prints

```
Bound = true, z = 24
```

That is, changes of the two properties *x* and *y* automatically update *z*, which exactly is what binding is about. The next statement again removes the binding of *z*, after which *x* and *y* are changed again. The last statement will then print

```
Bound = false, z = 24
```

which partly shows that *z* is no longer bound and its value has not changed since it is no longer linked to the two properties *x* and *y*.

A binding is thus an expression that is evaluated to a value. The expression consists of one or more observable values, known as dependencies. The binding observes changes of its dependencies and its value is automatically updated when its dependencies change values. All property classes in JavaFX have built-in support for binding. There are two forms of binding: *Unidirectional binding* and *bidirectional binding*. The above is an example of

e-learning for kids
• The number 1 MOOC for Primary Education
• Free Digital Learning for Children 5-12
• 15 Million Children Reached
**About e-Learning for Kids** Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

unidirectional binding, which is defined by the method *bind()*. There are several limitations to unidirectional bindings, and you can not, for example, directly change the value of the property that is bound. You can not directly change the value of *z* above. In addition, a property can only be bound to one expression with unidirectional bindings, and trying to bind it again, overrides the first binding.

A binding can also be bidirectional, which means that you can also change the value of the property that is bound. Obviously, a binding like the above can not be bidirectional, for changing *z*, there is no way to automatically update the values of *x* and *y*. For a binding to be bidirectional, it must be between properties of the same type, but in return, a property may be bound to several other properties. The following method shows the principles of bidirectional binding:

```
private static void test05()
{
 IntegerProperty x = new SimpleIntegerProperty(3);
 IntegerProperty y = new SimpleIntegerProperty(5);
 IntegerProperty z = new SimpleIntegerProperty(7);
 System.out.println("x = " + x.get() + ", y = " + y.get() + ", z = " + z.get());
 x.bindBidirectional(y);
 System.out.println("x = " + x.get() + ", y = " + y.get() + ", z = " + z.get());
 x.bindBidirectional(z);
 System.out.println("x = " + x.get() + ", y = " + y.get() + ", z = " + z.get());
 z.set(11);
 System.out.println("x = " + x.get() + ", y = " + y.get() + ", z = " + z.get());
 x.unbindBidirectional(y);
 x.unbindBidirectional(z);
 System.out.println("x = " + x.get() + ", y = " + y.get() + ", z = " + z.get());
 x.set(13);
 y.set(17);
 z.set(19);
 System.out.println("x = " + x.get() + ", y = " + y.get() + ", z = " + z.get());
}
```

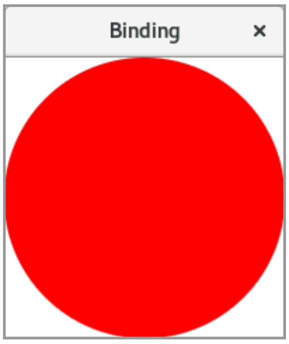
Three properties are created again, and the first *println()* displays the values of these properties before they are bound. Next, *x* is bound to *y* and the next *println()* statement shows that *x* now has the same value as *y*. Then *x* is bound to *z*, which means that *x* has the same value as *z*, and since *x* is bound to *y* and since the binding is bidirectional, the value of *y* also becomes the value of *x*. The next *println()* statement shows that the three properties now all have the same value. Now, the value of *z* changes to 11, and when the three properties are printed again, they all have the value 11. Then the bindings are removed and the three properties are printed again, and you can see that they still have the value 11. Finally, the values of the three properties are changed, after which they are printed and the new values are shown:

```
x = 3, y = 5, z = 7
x = 5, y = 5, z = 7
x = 7, y = 7, z = 7
x = 11, y = 11, z = 11
x = 11, y = 11, z = 11
x = 13, y = 17, z = 19
```

In JavaFX, all read/write properties support bidirectional binding, and as an example of how it can be used, consider the following class:

```
package fxproperties;

import javafx.application.Application;
import javafx.scene.*;
import javafx.scene.paint.*;
import javafx.stage.Stage;
import javafx.scene.shape.*;
import javafx.beans.binding.*;

public class CircleView extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  Circle c = new Circle();
  c.setFill(Color.RED);
  Group root = new Group(c);
  Scene scene = new Scene(root, 200, 200);
  c.centerXProperty().bind(scene.widthProperty().divide(2));
  c.centerYProperty().bind(scene.heightProperty().divide(2));
  c.radiusProperty().bind(Bindings.min(scene.widthProperty(),
   scene.heightProperty()).divide(2));
  primaryStage.setTitle("Binding");
  primaryStage.setScene(scene);
  primaryStage.show();
 }
}
```

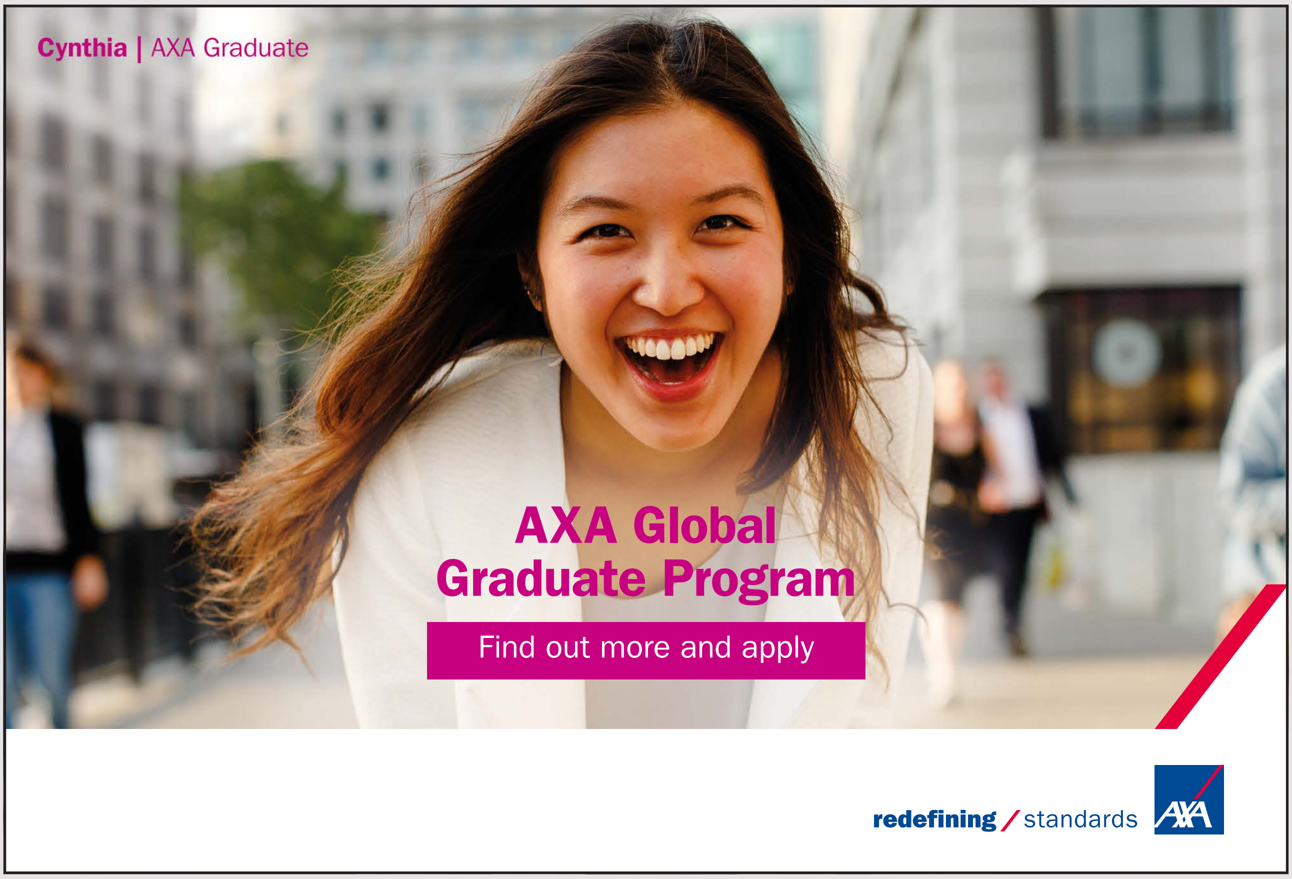
Note that it is a JavaFX window (inherits the class *Application*) and there is only a *start()* method. Here a red circle is inserted as a node in a scene graph. Next, three properties are bound, that are the circle's center and radius, to the width and height of the scene object so that the circle's center becomes the center of the window while the radius becomes the largest possible so that the entire circle is displayed. The window will thus show a circle:



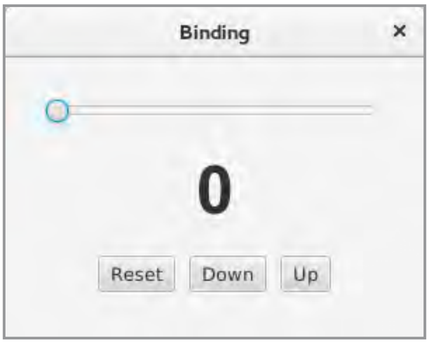
and change the size of the window, you will see that the circle follows the window as it always sits in the middle of the window, and the radius changes depending on the window size.

You should also note how to open the window from the main program:

```
private static void test06()
{
 Application.launch(CircleView.class);
}
```


Cynthia | AXA Graduate
AXA Global
Graduate Program
Find out more and apply
redefining / standards
AXA

As another example, the method *test07()* opens the following window:



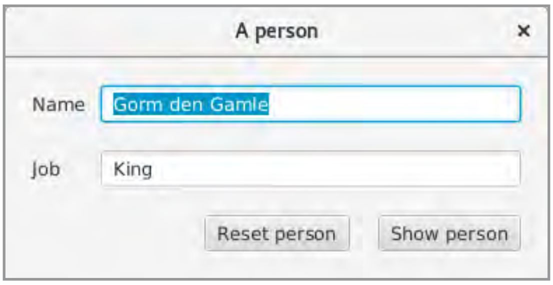
which includes a *Slider*, a *Label* and three *Button* controls. Here, the *Slider* component is bound to the property *step* in the class *Counter* with a bidirectional binding while the *Label* component is bound to the property *value* in the class *Counter* with a unidirectional binding:

```
public class CounterView extends Application
{
 private Counter counter = new Counter();

 @Override
 public void start(Stage primaryStage)
 {
  Slider slider = new Slider(1, 10, 1);
  Label label = new Label("");
  label.setFont(Font.font("Arial", FontWeight.BOLD, FontPosture.REGULAR, 48));
  Button cmd1 = new Button("Reset");
  cmd1.setOnAction(e -> counter.setStep(1));
  Button cmd2 = new Button("Down");
  cmd2.setOnAction(e -> counter.decrease());
  Button cmd3 = new Button("Up");
  cmd3.setOnAction(e -> counter.increase());
  HBox commands = new HBox(10, cmd1, cmd2, cmd3);
  commands.setAlignment(Pos.CENTER);
  VBox root = new VBox(20, slider, label, commands);
  root.setAlignment(Pos.TOP_CENTER);
  root.setPadding(new Insets(30, 30, 30, 30));
  Scene scene = new Scene(root, 300, 200);
  label.textProperty().bind(counter.valueProperty().asString());
  slider.valueProperty().bindBidirectional(counter.stepProperty());
  primaryStage.setTitle("Binding");
  primaryStage.setScene(scene);
  primaryStage.show();
 }
}
```

A *Label* control has a *StringProperty* property *textProperty*, and a *Counter* has a property *valueProperty* of the type *ReadOnlyIntegerWrapper*. Since the latter is read-only, you can not of course create a bidirectional binding, but a *ReadOnlyIntegerWrapper* has an *asString()* method, which returns a *StringBinding* object, and allows it to create a unidirectional binding. The result is that if the property *value* in the *Counter* object changes (clicking the *Down* and *Up* buttons), then the *Label* object will automatically be updated. A *Slider* control has a *valueProperty* of the type *IntegerProperty*, and the same applies to the class *Counter*, which has a *stepProperty* of the same type. You can therefore create a bidirectional binding between these two properties. The result is that if you change the slider, the *Counter* object's step property is updated. If you click the *Reset* button, it sets the value of *step* to 1, and since the slider binding is bidirectional, the slider component is automatically updated.

## EXERCISE 1

You must write an application that opens the following window:



Name: Gorm den Gamle
Job: King

where you can maintain information about a person. The displayed name and job title are default values. If you click on the last button, you must get an alert that shows the person's name and job title, and clicking on the first button, the values must be changed to default.

The program must use the following model class:

```
package editperson;

import javafx.beans.property.*;

public class Person
{
 private StringProperty name = new SimpleStringProperty("Gorm den Gamle");
 private StringProperty job = new SimpleStringProperty("King");
```

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

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

public StringProperty nameProperty()
{
 return name;
}

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

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

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

 public StringProperty jobProperty()
 {
  return job;
 }
}
```

and the two input fields must be bound to the model with bidirectional bindings. When you click the *Default* button, the default values must be set by directly updating the model, thus performing the *setName()* and *setJob()* methods.

## 2.2 OBSERVABLE COLLECTIONS

JavaFX defines multiple collections, which are extensions of the classic collection classes. There are three interfaces defined

- *ObervableList<T>*
- *ObervableSet<T>*
- *ObervableMap<K,V>*

which inherits *List<T>*, *Set<T>* and *Map<K, V>* respectively, but also inherits the interface *Observable*. JavaFX does not immediately have classes that implements those interfaces, but instead, there is a class of *FXCollections* that has static methods that return objects that implement these interfaces. Viewed from the programmer, an observable colletion is a list, a set or a map that can be observed for invalidation and changes of the content.

Consider the following example:

```
private static void test08()
{
 ObservableList<String> list =
  FXCollections.observableArrayList("Gorm den Gamle", "Harald Blåtand");
 list.addListener(FXProperties::onChanged);
 list.addAll("Svend Tveskæg", "Harald d. 2.");
 list.add("Knud d. Store");
 list.remove(3);
 show(list);
 java.util.Collections.sort(list);
 show(list);
}
```

```
private static void show(java.util.List<String> list)
{
 System.out.println();
 for (String name : list) System.out.println(name);
}

private static void onChanged(ListChangeListener.
 Change<? extends String> change)
{
 System.out.println("List has changed");
}
```

If the method is performed the result is:

```
List has changed
List has changed
List has changed

Gorm den Gamle
Harald Blåtand
Svend Tveskæg
Knud d. Store
List has changed

Gorm den Gamle
Harald Blåtand
Knud d. Store
Svend Tveskæg
```

The list is created with two elements, and then a *ChangeListener* is associated that do nothing but prints a message on the console. The first event occurs after additional two items have been added and the next after another element is added. Finally, the third occurs after an item has been deleted. The method *show()* shows the content of the list on the console and then reads the list. Note that the parameter is a *List<String>* which is ok, since an *ObservableList<String>* is specially a *List<String>*. After the list is printed, it is sorted, and as it means that the order of the list's elements changes, another *ChangeEvent* occurs.

The class *ListChangeListener.Change* has a variety of different methods that tells you about the reason for the event and you are encouraged to investigate which ones. The following example should show a little bit about how these methods can be used. First, I've added a simple model class with two properties that adhere to the JavaFX beans pattern:

```
package fxproperties;
```

```
import javafx.beans.property.*;

public class Person implements Comparable<Person>
{
 private StringProperty name = new SimpleStringProperty();
 private StringProperty job = new SimpleStringProperty();

 public Person(String name, String job)
 {
  setName(name);
  setJob(job);
 }

 public final String getName()
 {
  return name.get();
 }

 public final void setName(String name)
 {
  this.name.set(name);
 }
```

ie business school

93%
OF MIM STUDENTS ARE
WORKING IN THEIR SECTOR 3 MONTHS
FOLLOWING GRADUATION

MASTER IN MANAGEMENT

- STUDY IN THE CENTER OF MADRID AND TAKE ADVANTAGE OF THE UNIQUE OPPORTUNITIES THAT THE CAPITAL OF SPAIN OFFERS
- PROPEL YOUR EDUCATION BY EARNING A DOUBLE DEGREE THAT BEST SUITS YOUR PROFESSIONAL GOALS
- STUDY A SEMESTER ABROAD AND BECOME A GLOBAL CITIZEN WITH THE BEYOND BORDERS EXPERIENCE

Length: **10 MONTHS**
Av. Experience: **1 YEAR**
Language: **ENGLISH / SPANISH**
Format: **FULL-TIME**
Intakes: **SEPT / FEB**

5 SPECIALIZATIONS
PERSONALIZE YOUR PROGRAM

#10 WORLDWIDE
MASTER IN MANAGEMENT
FINANCIAL TIMES

55 NATIONALITIES
IN CLASS

www.ie.edu/master-management | mim.admissions@ie.edu | Follow us on IE MIM Experience

```
 public StringProperty nameProperty()
 {
  return name;
 }

 public final String getJob()
 {
  return job.get();
 }

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

 public StringProperty jobProperty()
 {
  return job;
 }

 @Override
 public int compareTo(Person p)
 {
  int val = getName().compareTo(p.getName());
  if (val == 0) val = this.getJob().compareTo(p.getJob());
  return val;
 }

 @Override
 public String toString()
 {
  return getName() + ": " + getJob();
 }
}
```

and in relation to what has been said before, there is nothing new. Note, however, that the class's objects are comparable and can thus be sorted. I have then defined a listener class for change events for an *ObservableList* collection with *Person* objects:

```
package fxproperties;

import javafx.collections.ListChangeListener;

public class PersonChangeListener implements ListChangeListener<Person>
{
 @Override
 public void onChanged(ListChangeListener.Change<? extends Person> change)
 {
  while (change.next())
  {
   if (change.wasPermutated()) permutated(change);
   else if (change.wasUpdated()) updated(change);
   else if (change.wasReplaced()) replaced(change);
   else if (change.wasRemoved()) removed(change);
   else if (change.wasAdded()) added(change);
  }
 }

 private void permutated(ListChangeListener.Change<? extends Person> change)
 {
  System.out.println("Sort: " + change.getFrom() + " - "+ change.getTo());
 }

 private void updated(ListChangeListener.Change<? extends Person> change)
 {
  System.out.println("Updated: " +
   change.getList().subList(change.getFrom(), change.getTo()));
 }

 private void replaced(ListChangeListener.Change<? extends Person> change)
 {
  removed(change);
  added(change);
 }

 private void removed(ListChangeListener.Change<? extends Person> change)
 {
  System.out.println("Removed " + change.getRemovedSize() + " person(s): "
+
   change.getRemoved());
 }

 private void added(ListChangeListener.Change<? extends Person> change)
 {
  System.out.println("Added " + change.getAddedSize() + " person(s): " +
   change.getAddedSubList());
 }
}
```

A single event handler must be implemented with the name *onChanged()*, whose parameter is a

```
ListChangeListener.Change<? extends Person>
```

object named *change*. When an event occurs and the method is called, the event may be triggered by one or more changes to the list, and therefore, the method starts with a loop that iterates over these changes. As shown by the method, there may be 5 different types of changes, and each iteration of the loop calls a method corresponding to the change in question. In this case, the methods are all trivial and do nothing but print a text on the screen, and the purpose is to show only when the event occurs.

With the above two classes available, you can perform the following method:

```
private static void test09()
{
 Person p1 = new Person("Gudrun Jensen", "Heks");
 Person p2 = new Person("Carlo Andersen", "Skarpretter");
 Person p3 = new Person("Valborg Kristensen", "Spåkone");
 Person p4 = new Person("Egon Knudsen", "Kriger");
 Person p5 = new Person("Abelone Jensen", "Sin mands kone");
 Person p6 = new Person("Viktor Hansen", "Høvding");
 Callback<Person, Observable[]> cb =
  (Person p) -> new Observable[] { p.nameProperty(), p.jobProperty() };
 ObservableList<Person> list = FXCollections.observableArrayList(cb);
 list.addListener(new PersonChangeListener());
 System.out.println(list);
```

“I studied English for 16 years but... ...I finally learned to speak it in just six lessons”
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

```
 list.add(p1);
 System.out.println(list);
 list.addAll(p2, p3);
 System.out.println(list);
 FXCollections.sort(list);
 System.out.println(list);
 p1.setName("Gunhild Mikkelsen");
 p2.setJob("Natmand");
 list.set(0, new Person("Olga Hansen", "Sypige"));
 System.out.println(list);
 list.setAll(p4, p5, p6);
 System.out.println(list);
 list.removeAll(p4, p6);
 System.out.println(list);
}
```

Performing the method gives you the result:

```
[]
Added 1 person(s): [Gudrun Jensen: Heks]
[Gudrun Jensen: Heks]
Added 2 person(s): [Carlo Andersen: Skarpretter, Valborg Kristensen: Spåkone]
[Gudrun Jensen: Heks, Carlo Andersen: Skarpretter, Valborg Kristensen: Spåkone]
Sort: 0 – 3
[Carlo Andersen: Skarpretter, Gudrun Jensen: Heks, Valborg Kristensen: Spåkone]
Updated: [Gunhild Mikkelsen: Heks]
Updated: [Carlo Andersen: Natmand]
Removed 1 person(s): [Carlo Andersen: Natmand]
Added 1 person(s): [Olga Hansen: Sypige]
[Olga Hansen: Sypige, Gunhild Mikkelsen: Heks, Valborg Kristensen: Spåkone]
Removed 3 person(s): [Olga Hansen: Sypige, Gunhild Mikkelsen: Heks,
  Valborg Kristensen: Spåkone]
Added 3 person(s): [Egon Knudsen: Kriger, Abelone Jensen: Sin mands kone,
  Viktor Hansen: Høvding]
[Egon Knudsen: Kriger, Abelone Jensen: Sin
 mands kone, Viktor Hansen: Høvding]
Removed 1 person(s): [Egon Knudsen: Kriger]
Removed 1 person(s): [Viktor Hansen: Høvding]
[Abelone Jensen: Sin mands kone]
```

The test method starts by creating 6 *Person* objects. Next, a *Callback* object is defined which indicates which properties for a *Person* object to be observed regarding changes. As the next step, the class *FXCollections* is used to create an *ObservableList<Person>* with the above *Callback* object as parameter. Next, the list is printed on the screen, which is the first line in the result, and it is just an empty list. The next statement adds *p1* to the list, after which the list is printed again. The result shows that the event handler has performed the method *added()* and added an object list, and the list then contains one object. As the next operation, *p2* and *p3* are added to the list and it is printed again. The result shows that the event handler has performed the method *added()* again and added two objects, and the list now contains 3 objects. Next, the list is sorted and it is printed again. As shown by the result, the event handler has performed the method *permuted()* corresponding to two or more of the list's objects have been replaced. Next, the value of a property for *p1* and *p2*, respectively, is changed and here you should note that the event handler performs the method *updated()* and note that it is not the list's content that has been changed but the objects that the list contains. The next statement again changes the actual contents of the list:

```
list.set(0, new Person("Olga Hansen", "Sypige"));
```

and you can see from the result that the event handler has performed the method *replaced()*. After the list is printed again, the method will performe the statement

```
list.setAll(p4, p5, p6);
```

which means that the list content are replaced with three new objects. From the result, you can see that both the method *removed()* and *added()* are performed. Finally, two objects are deleted, and then one is left.

Looking at the above example, which should illustrate how an *ObservableList* works, you could write similar programs that can illustrate how an *ObservableSet* and an *ObservableMap* works. I do not want to show examples here as there are no big differences compared to the above example.

## 2.3 BINDING OBSERVABLE COLLECTIONS

There is a class *ListProperty* that represents a property for an *ObservableList*, and you can think of the class as a wrapper for an *ObervableList*. It is a property type similar to other properties presented in this chapter. You can associate three kinds of listeners with a *ListProperty*:

- *InvallidationListener*
- *ChangeListener*
- *ListChangeListener*

and these listeners will receive notifications when the *OberservableList* is changed as the *ListProperty* object in question wrapper. Consider the following method:

```
private static void test10()
{
 ListProperty<String> property =
  new SimpleListProperty(FXCollections.observableArrayList());
 IntegerProperty count = new SimpleIntegerProperty();
 count.bind(property.sizeProperty());
 System.out.println(count.get());
 property.addListener(FXProperties::propertyInvalidated);
 property.addListener(FXProperties::propertyChanged);
 property.addListener(FXProperties::propertyListChanged);
 property.add("Gorm");
 property.add("Harald");
 System.out.println(count.get());
 property.set(FXCollections.observableArrayList("Svend", "Knud", "Valdemar"));
 System.out.println(count.get());
 property.remove("Knud");
 System.out.println(count.get());
 System.out.println(property.get());
}
```

The method creates a *ListProperty* named *property* as a *SimpleListProperty*, which wrapper an *ObservableList*. In addition, an *integerProperty* is defined with the name count, and it is bounded to the list's size. Here, you should note that a *ListProperty* implements the

Excellent Economics and Business programmes at:
university of groningen
AACSB ACCREDITED
"The perfect start of a successful, international career."
CLICK HERE
to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

*ObservableList* interface and therefore has the same properties as an *ObservableList*. *property* therefore has a *sizeProperty* which is an *IntegerProperty* and can therefore be bounded to *count*. The next print statement will therefore print 0 on the screen as the list is currently empty. As the next step, the listener methods are defined for the three events. The first one is trivial and does nothing but prints a text:

```
private static void propertyInvalidated(Observable list)
{
 System.out.println("Property invalid…");
}
```

You must note the parameter and that it is an *Observable*. The next listener method is also trivial and it prints the contents of the list before and after it has been changed:

```
private static void propertyChanged(
 ObservableValue<? extends ObservableList<String>> observable,
 ObservableList<String> oldList, ObservableList<String> newList)
{
 System.out.println("Old: " + oldList);
 System.out.println("New: " + newList);
}
```

You should primarily note the parameters that are a reference to the list (the object that caused the event in question) and the list's value before the change and its value after the change. The last event handler does nothing but prints a text (possibly more), but distinguishes the reason for the event:

```
private static void propertyListChanged(
 ListChangeListener.Change<? extends String> change)
{
 while (change.next())
 {
  if (change.wasPermutated()) System.out.println("Permutated");
  else if (change.wasUpdated()) System.out.println("Updated");
  else if (change.wasReplaced()) System.out.println("Replaced");
  else if (change.wasRemoved()) System.out.println("Removed");
  else if (change.wasAdded()) System.out.println("Added");
 }
}
```

Below is shown the result of performing the test method:

```
0
Property invalid…
Old: [Gorm]
```

```
Added
Property invalid…
Old: [Gorm, Harald]
New: [Gorm, Harald]
Added
2
Property invalid…
Old: [Gorm, Harald]
New: [Svend, Knud, Valdemar]
Replaced
3
Property invalid…
Old: [Svend, Valdemar]
New: [Svend, Valdemar]
Removed
2
[Svend, Valdemar]
```

It is easy to follow the program code and compare with the results, and the important thing is of course to observe when each event handler is performing.

You can also bind two *ListProperty* objects to each other, which means that the lists that they wrapper are bound. Consider the following method:

```
private static void test11()
{
 ListProperty<String> property1 =
  new SimpleListProperty<>(FXCollections.observableArrayList());
 ListProperty<String> property2 =
  new SimpleListProperty<>(FXCollections.observableArrayList());
 property1.add("Kristian");
 property2.add("Frederik");
 System.out.println(property1.get());
 System.out.println(property2.get());
 property1.bind(property2);
 property1.addAll("Svend", "Knud", "Valdemar");
 System.out.println(property1.get());
 System.out.println(property2.get());
 property2.set(FXCollections.observableArrayList("Gorm", "Harald"));
 System.out.println(property1.get());
 System.out.println(property2.get());
//    property1.set(FXCollections.observableArrayList("Harld", "Oluf"));
 property1.unbind();
 property1.bindBidirectional(property2);
 property1.set(FXCollections.observableArrayList("Erik", "Kristoffer"));
 property2.set(FXCollections.observableArrayList("Niels", "Abel"));
 property1.add("Oluf");
 property2.add("Knud");
 System.out.println(property1.get());
 System.out.println(property2.get());
}
```

Above this is defined two *ListProperty* objects that wrapper each their *ObservableList*, and the first two print statements will therefore print:

```
[Kristian]
[Frederik]
```

what there is no strange in. Because of the JavaFX properties, they can of course be bound:

```
property1.bind(property2);
```

which binds *property1* to *property2* with a unidirectional binding. Quite exactly, it means that the list for which *property1* is wrapper for is the same list that *property2* is wrapper for. The next statement adds three names to *property1*, and the next statement again a name to *property2*. The next two print statments results in

```
[Frederik, Svend, Knud, Valdemar, Hans]
[Frederik, Svend, Knud, Valdemar, Hans]
```

American online
LIGS University
is currently enrolling in the
Interactive Online BBA, MBA, MSc,
DBA and PhD programs:

- enroll **by September 30th, 2014** and
- **save up to 16%** on the tuition!
- pay in 10 installments / 2 years
- Interactive **Online education**
- visit www.ligsuniversity.com to find out more!

**Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education. More info here.**

and perhaps it was not what one would expect as it is a unidirectional binding, but since the two property objects wrapper the same list (and the one that *property2* was originally wrapper for), it is in both cases that list that is being updated. If you then perform the statement

```
property2.set(FXCollections.observableArrayList("Gorm", "Harald"));
```

*property2* is now wrapper for a list with two names, and because of the binding, both properties will refer to this list. If you try instead to execute the statement, which is comment out, you get an exception due to the fact that *propert1* is bound to *property2* with a unidirectional binding. Therefore, do not change the object as *property1* is wrapper for. Next, the binding is removed and a bidirectional binding is instead created. Now you can change the list for both properties, since the binding now is bidirectional, and it is the last value that applies. The last print statements therefore results in:

```
[Niels, Abel, Oluf, Knud]
[Niels, Abel, Oluf, Knud]
```

The above bindings bind the lists that the properties are wrappers for, but you can instead bind their contents as you do with the methods *bindContent()* and *bindContentBidirectional()*:

```
private static void test12()
{
 ListProperty<String> property1 =
  new SimpleListProperty<>(FXCollections.observableArrayList());
 ListProperty<String> property2 =
  new SimpleListProperty<>(FXCollections.observableArrayList());
 property1.bindContent(property2);
 property1.addAll("Svend", "Knud", "Valdemar");
 System.out.println(property1.get());
 System.out.println(property2.get());
 property2.set(FXCollections.observableArrayList("Gorm", "Harald"));
 System.out.println(property1.get());
 System.out.println(property2.get());
 property1.unbindContent(property2);
 property1.bindContentBidirectional(property2);
 property1.add("Oluf");
 property2.add("Knud");
 System.out.println(property1.get());
 System.out.println(property2.get());
}
```

Above this defines a binding of the two lists' content, but this means that the two lists' content by the unidirectional binding is not synchronized:

```
[Svend, Knud, Valdemar]
[]
[Gorm, Harald, Svend, Knud, Valdemar]
[Gorm, Harald]
[Gorm, Harald, Oluf, Knud]
[Gorm, Harald, Oluf, Knud]
```

As the last example, I will show how to bind to a single element in an *ObservableList*:

```
private static void test13()
{
 ListProperty<Person> property =
  new SimpleListProperty<>(FXCollections.observableArrayList());
 ObjectBinding<Person> last =
  property.valueAt(property.sizeProperty().subtract(1));
 property.add(new Person("Gudrun Jensen", "Heks"));
 property.add(new Person("Carlo Andersen", "Skarpretter"));
 property.add(new Person("Valborg Kristensen", "Spåkone"));
 System.out.println(property.get());
 System.out.println(last.get());
}
```

*property* is a *ListProperty*, which wrapper an *ObservableList* with *Person* objects. *last* is an *ObjectBinding* object for a *Person* object that is bounded to the last item in the list that *property* is wrapper for. Note the syntax. The method *valueAt()* has as parameter an index and returns an *ObjectBinding* to the object in the list to which the index refers. After adding three objects to the list, the print statements shows:

```
[Gudrun Jensen: Heks, Carlo Andersen: Skarpretter, Valborg Kristensen: Spåkone]
Valborg Kristensen: Spåkone
```

In the same way as shown in this section, there are also wrapper properties called *SetProperty* and *MapProperty* for an *ObeservableSet* and an *ObservableMap* respectively.

## EXERCISE 2

Create a simple console application, which you can call *BindingElements*. The project *FXProperties* has a class *Person*. Copy this class to the new project and add the following simple method to the class *BindingElements*:

```
private static void print(List<Person> list)
{
 for (Person p : list) System.out.println(p);
 System.out.println();
}
```

Then write the following *main()* method:

```
public static void main(String[] args)
{
 // Create an ObservableList<Person> to Person objects
 // Add 5 Person objects to list, what names and job titles do not matter
 // Create a ListProperty for the above list
 // Define a binding for the element with index 1
 // Define a binding for the element with index 3
 // Print the list on the screen
 // Insert a new Person in list at position 1
 // Insert a new Person in list at position 3
 // Modify the name of the Person that is bound with p1
 // Modify the job title of the Person that is bound with p3
 // Print the list on the screen
}
```

Test the program. Do you get the expected result? Are there the expected objects that have been changed?

DON'T EAT YELLOW SNOW
What will your advice be?
GrantThornton
An instinct for growth™

**Some advice just states the obvious.** But to give the kind of advice that's going to make a real difference to your clients you've got to listen critically, dig beneath the surface, challenge assumptions and be credible and confident enough to make suggestions right from day one. At Grant Thornton you've got to be ready to kick start a career right at the heart of business.

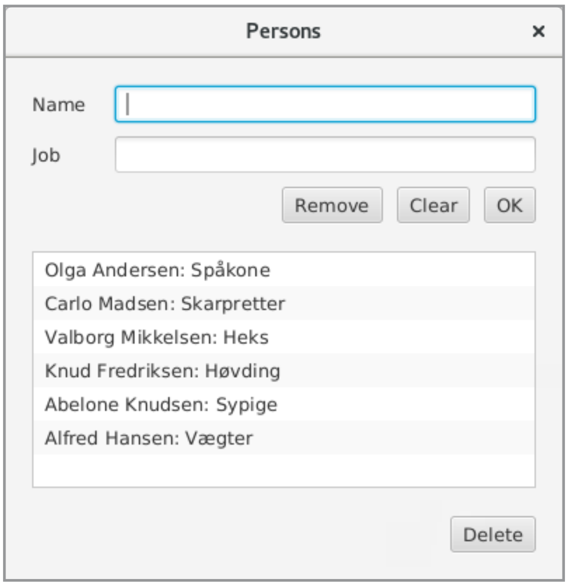
Sound like you? Here's our advice: visit
**GrantThornton.ca/careers/students**

Scan here to learn more about a career with Grant Thornton.

© Grant Thornton LLP. A Canadian Member of Grant Thornton International Ltd

## 2.4 BINDING PERSONS

As a final example of properties, the application *PersonProgram* opens the following window:



Olga Andersen: Spåkone
Carlo Madsen: Skarpretter
Valborg Mikkelsen: Heks
Knud Fredriksen: Høvding
Abelone Knudsen: Sypige
Alfred Hansen: Vægter

where you in the two top entry fields, can enter the name and job title of a person. If you click *OK*, a person will be added to a *ListView*. In the window above 6 persons have been created / entered. If you double-click on a person in the *ListView* control, the person's data is inserted into the input fields, and they can then be edited. If you double click on a person, this can be deleted by clicking the *Remove* button. Finally, the *Clear* button is used to delete the entry fields and remove a selction in the *ListView* control. The bottom button is used to delete the content of the list.

The program uses the class *Person* from the project *FXProperties* – expanded by two trivial methods. The code for the *PersonProgram's* window is as follows:

```
package personprogram;

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.*;
import javafx.scene.input.*;
import javafx.collections.*;
import javafx.beans.property.*;
```

```
public class PersonProgram extends Application
{
 private TextField txtName = new TextField();
 private TextField txtJob = new TextField();
 private ListView lstView = null;
 private Person person = new Person("", "");
 private ObservableList<Person> persons =
  FXCollections.observableArrayList();
 private IntegerProperty selected = new SimpleIntegerProperty(-1);

 @Override
 public void start(Stage primaryStage)
 {
  lstView = new ListView(persons);
  lstView.setPrefHeight(300);
  lstView.setOnMouseClicked(this::clickHandler);
  selected.bind(lstView.getSelectionModel().selectedIndexProperty());
  BorderPane root =
   new BorderPane(lstView, createTop(), null, createBottom(), null);
  root.setPadding(new Insets(20, 20, 20, 20));
  BorderPane.setMargin(lstView, new Insets(20, 0, 20, 0));
  Scene scene = new Scene(root);
  primaryStage.setTitle("Persons");
  primaryStage.setScene(scene);
  primaryStage.show();
 }

 private Pane createBottom()
 {
  HBox pane = new HBox(createButton("Delete", this::deleteHandler));
  pane.setAlignment(Pos.CENTER_RIGHT);
  return new BorderPane(null, null, pane, null, null);
 }

 private Pane createTop()
 {
  txtName.textProperty().bindBidirectional(person.nameProperty());
  txtJob.textProperty().bindBidirectional(person.jobProperty());
  txtName.setPrefWidth(300);
  HBox commands = new HBox(10, createButton("Remove", this::removeHandler),
   createButton("Clear", this::clearHandler),
   createButton("OK", this::okHandler));
  commands.setAlignment(Pos.CENTER_RIGHT);
  GridPane pane = new GridPane();
  pane.setVgap(10);
  pane.setHgap(20);
  pane.addRow(0, new Label("Name"), txtName);
  pane.addRow(1, new Label("Job"), txtJob);
 pane.add(commands, 1, 2);
 return pane;
}

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

private void okHandler(ActionEvent e)
{
 if (person.getName().trim().length() > 0 &&
  person.getJob().trim().length() > 0)
 {
  if (selected.get() < 0)
   persons.add(new Person(person.getName(), person.getJob()));
  else persons.set(selected.get(),
   new Person(person.getName(), person.getJob()));
  clearHandler(e);
 }
}
```

Alcatel·Lucent
www.alcatel-lucent.com/careers
What if you could build your future and create the future?
One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

```
 private void clearHandler(ActionEvent e)
 {
  person.clear();
  lstView.getSelectionModel().clearSelection();
  txtName.requestFocus();
 }

 private void removeHandler(ActionEvent e)
 {
  if (selected.get() >= 0) persons.remove(selected.get());
  clearHandler(e);
 }

 private void deleteHandler(ActionEvent e)
 {
  persons.clear();
  clearHandler(e);
 }

 private void clickHandler(MouseEvent e)
 {
  if (e.getButton() == MouseButton.PRIMARY && e.getClickCount() == 2)
  {
   if (selected.get() >= 0) person.update(persons.get(selected.get()));
  }
  else lstView.getSelectionModel().clearSelection();
 }

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

The class has 6 instance variables, where the first three are for controls. The next has the type *Person* and represents the object on which you work, and thus the object whose properties appear in the two entry fields. The next again is an *ObservableList* and represents the content of the *ListView* control. Finally, the last one is a property that must represent the index of the *Person* object in the list that is selected.

Examining the method *start()* is the first thing that happens that the *lstView* object is created with *persons* as a parameter for the constructor. This means that a bidirectional binding is created between the *ListView* component and the list *persons*. An event handler is also associated with the *ListView* component for mouse events. Finally, the property *selected* is bound to the selected index in the *ListView* component. The rest of the method *start()*

does not contain anything new and is only intended to create the window's scene graph. However, you must note the method *createTop()* that creates a *GridPane* for entering a person. Here you should especially note how the two *TextField* controls bind to properties of the *Person* object.

Then there are the event handlers where there are 5, but they are generally simple. The first concerns the *OK* button and starts by testing whether something has been entered for both name and job title. If this is the case, the *property* selected is used to determine whether to add a new object or whether an existing object is to be modified. You should note that in either case, a new *Person* object is instantiated. It is necessary because the variable *person* can not be set to refer to another object as it is bound to the input fields. Whether you add an object or modify an object, the event handler for the *Clear* button is called that clears the fields in the *person* object and removes any selection in the *ListView* component. The event handlers for the *Remove* and *Delete* buttons are both trivial and the event handler for clicks with the mouse in the *ListView* component is also simple, and you should especially note how to test for double-click.

When you test the program, note how the user interface is automatically updated for reasons of the bindings.

## 2.5 THE SCREEN

In these and the following two sections I briefly will mention how to refer to the screen and a little more remarks about the program's *Stage* object – although it does not have much to do with properties, but the examples are part of the project *FXProperties*. The following class opens a window that prints informations on the console about the screen and components size:

```
public class ScreenView extends Application
{
 @Override
 public void start(Stage stage)
 {
  BorderPane root = new BorderPane(new Label("Tekst"));
  root.setPadding(new Insets(30, 30, 30, 30));
  Scene scene = new Scene(root, 300, 200);
  stage.setScene(scene);
  stage.setTitle("Screen");
  stage.setWidth(500);
  stage.setHeight(400);
  stage.show();
 showSize();
 showSize(stage);
 showSize(root);
}

private void showSize()
{
 Screen screen = Screen.getPrimary();
 Rectangle2D r1 = screen.getBounds();
 Rectangle2D r2 = screen.getVisualBounds();
 System.out.printf("(%1.0f, %1.0f) %1.0f x %1.0f\n", r1.getMinX(),
  r1.getMinY(), r1.getWidth(), r1.getHeight());
 System.out.printf("(%1.0f, %1.0f) %1.0f x %1.0f\n\n", r2.getMinX(),
  r2.getMinY(), r2.getWidth(), r2.getHeight());
}

private void showSize(Stage s)
{
 System.out.printf("(%1.0f, %1.0f) %1.0f x %1.0f\n\n", s.getX(), s.getY(),
  s.getWidth(), s.getHeight());
}
```

Maastricht University *Leading in Learning!*

Join the best at the Maastricht University School of Business and Economics!

Top master's programmes
- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

*Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012*

Maastricht University is the best specialist university in the Netherlands (Elsevier)

Visit us and find out why we are the best!
Master's Open Day: 22 February 2014

www.mastersopenday.nl

```
 private void showSize(Pane p)
 {
  Bounds b1 = p.getBoundsInLocal();
  Bounds b2 = p.getBoundsInParent();
  Bounds b3 = p.getLayoutBounds();
  System.out.printf("(%1.0f, %1.0f) %1.0f x %1.0f\n", b1.getMinX(),
   b1.getMinY(), b1.getWidth(), b1.getHeight());
  System.out.printf("(%1.0f, %1.0f) %1.0f x %1.0f\n", b2.getMinX(),
   b2.getMinY(), b2.getWidth(), b2.getHeight());
  System.out.printf("(%1.0f, %1.0f) %1.0f x %1.0f\n\n", b3.getMinX(),
   b3.getMinY(), b3.getWidth(), b3.getHeight());
 }
}
```

The program is simple and does nothing but create a window with a button. You should note that when the root object is added to the scene object, the window size is defined, and thus becoming the *Stage* object's size, but also the size of the root object, that is a *BorderPane*. This size is changed later in the method *start()*, partly to show that it is possible, and partly to show that it does not change the size of the root object. After the window is displayed on screen, three methods are called, which prints text on the console. You should note that these methods must be called after the *Stage* object is displayed and the scene graph's components are rendered.

The first method uses a class *Screen* that has static methods that return a *Screen* object that represents the screen and which is used to determine properties of the screen. In this case, two methods are used which return a *Rectangle2D* object with information about the screen size (measured in pixels and thus the resolution of the screen). The first returns the physical size of the screen while the other returns the size available for the program. The difference depends on the current platform, but the result could be as follows:

```
(0, 0) 1920 x 1080
(0, 27) 1920 x 1020
```

The difference is because my screen at the top has the *Activity* line (*Fedora*) and below a taskbar.

The next method prints the *Stage* object's size and position on the screen of its upper left corner. Here you should note which properties the class *Stage* has regarding position and size, and that the size is determined by the values I have assigned in the *start()* method:

```
(710, 234) 500 x 400
```

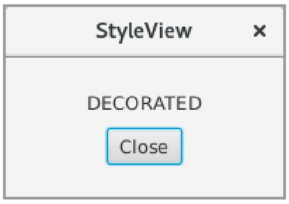
The last method prints information about the size of a node, as here is *root* and thus a *BorderPane*. The size is determined using three methods, all of which return a *Bound* object:

1. *getBoundsInLocal()*, as the position and size before any possible transformation
2. *getBoundsInParent()*, which is the position and size after a possible transformation
3. *getLayoutBounds()*, which is the position and size as a layout pane uses for calculations and may be different from the other sizes and is used if you write custom controls

In this case they are all alike.

## 2.6 DECORATIONS

A typical window looks something like the following:

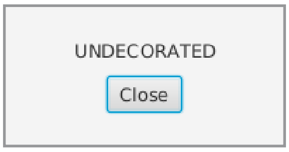


where there is a title bar and a border so you can change the window size. In addition, there is the window content, like here a scene graph with a label and a button. How the title bar and border are displayed depends on the current platform, and for the title bar, the platform determines which buttons are available. The window's title bar is also used to move the window on the screen by dragging it with the mouse and for example maximizing it by double-clicking the title bar.

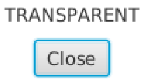
Title bar and border are referred to as the window's decorations or styles, and here are actually more options, which are defined as properties of the *Stage* object. I want to mention three:

1. *StageStyle.DECORATED*, which is the default where the window has a title bar and a border
2. *StageStyle.UNDECORATED*, where the window is not decorated and therefore does not have a title bar or border
3. *StageStyle.TRANSPARENT*, where the window is not decorated and also has a transparent background

For example, if you defines the above window as *UNDECORATED*, the result is:


UNDECORATED
Close

and if you defines the window as TRANSPARENT is the result:


TRANSPARENT
Close

The above can be illustrated by the following class:

```
public class StyleView extends Application
{
 private Stage stage;
 private double xpos;
 private double ypos;
```

Business
Strategic Marketing Management
Financial Economics
International Business
Leadership & Organisational Psychology
Shipping Management
BI
BI NORWEGIAN BUSINESS SCHOOL
EFMD EQUIS ACCREDITED

Empowering People. Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

```
 public void start(Stage stage)
 {
  this.stage = stage;
  Label lbl = new Label();
  Button cmd = new Button("Close");
  cmd.setOnAction(e -> stage.close());
  VBox root = new VBox(10, lbl, cmd);
  root.setAlignment(Pos.CENTER);
  Scene scene = new Scene(root, 200, 100);
  stage.setScene(scene);
  stage.setTitle("StyleView");
  this.show(scene, lbl, StageStyle.DECORATED);
//    this.show(scene, lbl, StageStyle.UNDECORATED);
//    this.show(scene, lbl, StageStyle.TRANSPARENT);
 }

 private void show(Scene scene, Label lbl, StageStyle style)
 {
  lbl.setText(style.toString());
  stage.initStyle(style);
  if (style == StageStyle.UNDECORATED || style == StageStyle.TRANSPARENT)
  {
   scene.setOnMousePressed(e -> handleMousePressed(e));
   scene.setOnMouseDragged(e -> handleMouseDragged(e));
  }
  if (style == StageStyle.TRANSPARENT)
  {
   stage.getScene().setFill(null);
   stage.getScene().getRoot().setStyle("-fx-background-color: transparent");
  }
  stage.show();
 }

 private void handleMousePressed(MouseEvent e)
 {
  xpos = e.getScreenX() - stage.getX();
  ypos = e.getScreenY() - stage.getY();
 }

 private void handleMouseDragged(MouseEvent e)
 {
  stage.setX(e.getScreenX() - xpos);
  stage.setY(e.getScreenY() - ypos);
 }
}
```

where the comments in the method *start()* indicate how the window should be decorated. There is not much going on in the *start()* method in addition to creating the scene graph and initializing the *Stage* object. Note that the class this time has an instance variable that refers to the *Stage* object so that it can be referenced from other class methods. Also note the event handler to the *Close* button, which closes the window, thus terminating the program. It works in this case, but not necessarily in other programs. If you perform a *close()* on a *Stage* object, it will hide the window (perform the *hide()* method in the base class to *Stage* that is named *Window*) and are there no other windows, the program will terminate. If you want to terminate the program as soon as a button is clicked, you should instead execute *Platform.exit()*.

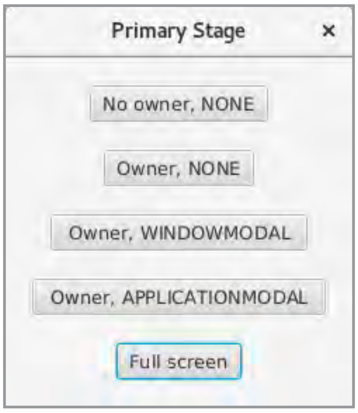
The window opens in the method *show()*, which has three parameters, which are the scene graph, its *Label* control and the style of the window and hence how it should be decorated. The method starts by setting the text for the window's *Label* control and then how the *Stage* object has to be decorated. Here you must note the syntax and that it must be done before the *Stage* object appears on the screen. If the style is *TRANSPARENT*, it indicates that there is no background. If the window is *UNDECORATED*, there is a problem, as you can not move it with the mouse. If you wish, you can, as shown above, associate an event handler for the mouse to the *Scene* object and then control that the window can be moved.

## 2.7 MODALITY

As you know it from Swing, a dialog box can be modeless or modal, and of course it also applies in JavaFX, where it is a property of the *Stage* object. There are three options:

1. *Modality.NONE*, where the result is a modeless window, which is the default for a *Stage*
2. *Modality.WINDOW_MODAL*, which is modal and blocks all windows that directly or indirectly own this window
3. *Modality.APPLICATION_MODEL*, which is modal and blocks all of the application's other windows

The following example shows the syntax, but also shows that an application may well have multiple *Stage* objects and thus windows (or dialogs) that explicitly create a *Stage* object. The *StageView* class opens the following window:


Primary Stage
×
No owner, NONE
Owner, NONE
Owner, WINDOWMODAL
Owner, APPLICATIONMODAL
Full screen

where each of the four top buttons opens a dialog box by instantiating a new *Stage* object, and the difference is partly modality and partly if the dialog has an owner. For example, the first is modeless and without an owner. This means clicking on the top button, the program will open a dialog box and then click at the cross in the main window's title bar (and you can because the dialog is modeless) closes the main window but not the dialog and the program does not terminate, before closing the other window.

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now

Go to www.helpmyassignment.co.uk for more info

Helpmyassignment

The last button is included to show that a *Stage* object also has a method, so you can say that the window should be displayed full screen. Note that displaying a full screen window you returns to normal viewing by pressing ESC – or clicking the *Full screen* button again.

The code is as follows:

```
public class StagesView extends Application
{
 @Override
 public void start(Stage stage)
 {
  VBox root = new VBox(20,
   createButton("No owner, NONE", e -> showDialog(null, null)),
   createButton("Owner, NONE", e -> showDialog(stage, Modality.NONE)),
    createButton("Owner, WINDOW_MODAL",
     e -> showDialog(stage, Modality.WINDOW_MODAL)),
    createButton("Owner, APPLICATION_MODAL",
     e -> showDialog(stage, Modality.APPLICATION_MODAL)),
    createButton("Full screen",
     e -> stage.setFullScreen(!stage.isFullScreen())));
  root.setAlignment(Pos.CENTER);
  root.setPadding(new Insets(20, 20, 20, 20));
  Scene scene = new Scene(root);
  stage.setScene(scene);
  stage.setTitle("Primary Stage");
  stage.show();
 }

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

 private void showDialog(Window owner, Modality modality)
 {
  Stage stage = new Stage();
  stage.initOwner(owner);
  if (modality != null) stage.initModality(modality);
  VBox root = new VBox(20, new Label(owner == null ? "Default" :
   "Parent Window"), new Label(modality == null ? "Default" :
   modality.toString()), createButton("Close", e -> stage.close()));
  root.setAlignment(Pos.CENTER);
  root.setPadding(new Insets(20, 20, 20, 20));
  Scene scene = new Scene(root);
  stage.setScene(scene);
  stage.setTitle("Stage");
  stage.show();
 }
}
```

There is not much to explain, but you should note the method *showDialog()*, which creates a dialog box by instantiating a new *Stage* object.

## PROBLEM 1

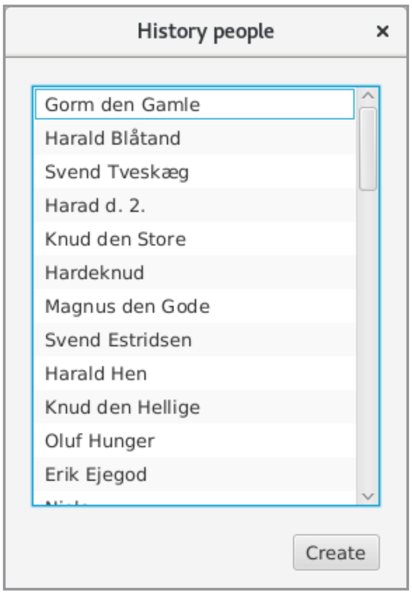
In this task you must write a program that works in the same way as the program from section 2.4. The database *padata* has a table with the name *history* which contains information about historical persons (see, if applicable, the book Java 6). The table is created with the following script:

```
use padata;
drop table if exists history;
create table history
(
 id int not null auto_increment primary key, # autogenerated surrogat key
 name varchar(50) not null,   # the person's name
 title varchar(30),           # the person's job title
 birth int,                   # birth, start of reign, or equivalent
 death int,                   # the year of death, end of reign, or equivalent
 country char(2),             # the country the person comes from
 description text             # a description
);
```

You must write a program that can maintain this database table. For example, you can call the project for *History*.

The task can be solved in several ways, but the idea is to use data binding, partly when data is displayed, and partly when editing information about a single person.

When the program starts, it must display a list of all persons in the database, and it could be a window as shown below, with a *ListView* showing the names of all persons:



Gorm den Gamle
Harald Blåtand
Svend Tveskæg
Harad d. 2.
Knud den Store
Hardeknud
Magnus den Gode
Svend Estridsen
Harald Hen
Knud den Hellige
Oluf Hunger
Erik Ejegod

Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

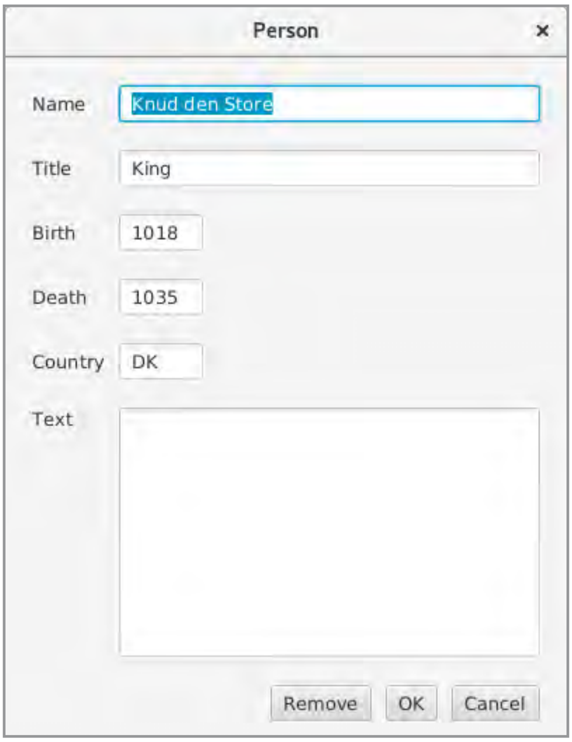
By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.

Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

The button should be used to create a new person and if you double-click a name in the list, the program must show all information about a person and it must be possible to edit the information. In both cases, you should use the same dialog box that could be:



Name Knud den Store
Title King
Birth 1018
Death 1035
Country DK
Text

# 3 ADVANCED CONTROLS

In this chapter I will illustrate the use of three controls that did not fit in the previous book:

1. *TableView*
2. *TreeView*
3. *TreeTableView*

where the first corresponds to the component *JTable*, the second to the component *JTree*, while the latter is best characterized as a combination. When the three components were not included in the previous book, it is partly because they are complex with many possibilities, and partly that their way of working can best be described after the mentions in the previous chapter of properties and binding.

## 3.1 TABLEVIEW

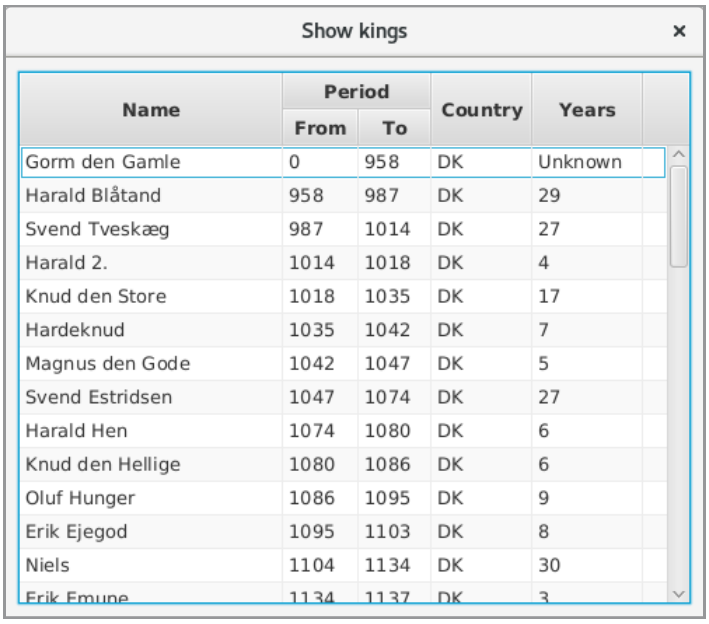
I want to start with the *TableView* component, that like *JTable*, is an extremely complex control that arranges data in rows and columns, and it is also the most useful of the three controls. The component's class is called *TableView*, but together with the component are several helper classes:

- *TableColumn*
- *TableRow*
- *TableCell*
- *TablePosition*
- *TableView.TableViewFocusModel*
- *TableView.TableViewSelectionModel*

and the names should tell a little about the purpose of the individual classes. The class *TableView* is used a bit like a *JTable*, where you must define a data model, which is the data that the component should display, and which columns there should be. I will start with an example, which I have called *ShowKingsProgram*, which shows an overview of Danish kings represented as objects of the type *King* (the class from the previous chapter):

```
public class King
{
 private static final String DK = "DK";
 private ReadOnlyStringWrapper name = new ReadOnlyStringWrapper(this, "name");
 private IntegerProperty from = new SimpleIntegerProperty(this, "from", 0);
private IntegerProperty to = new SimpleIntegerProperty(this, "to", 9999);
private StringProperty country;
```

If you run the program, it opens the window below which shows a table with 5 columns and thus one column more than the type of *King* has properties for:



| Name | Period | | Country | Years |
|---|---|---|---|---|
| | From | To | | |
| Gorm den Gamle | 0 | 958 | DK | Unknown |
| Harald Blåtand | 958 | 987 | DK | 29 |
| Svend Tveskæg | 987 | 1014 | DK | 27 |
| Harald 2. | 1014 | 1018 | DK | 4 |
| Knud den Store | 1018 | 1035 | DK | 17 |
| Hardeknud | 1035 | 1042 | DK | 7 |
| Magnus den Gode | 1042 | 1047 | DK | 5 |
| Svend Estridsen | 1047 | 1074 | DK | 27 |
| Harald Hen | 1074 | 1080 | DK | 6 |
| Knud den Hellige | 1080 | 1086 | DK | 6 |
| Oluf Hunger | 1086 | 1095 | DK | 9 |
| Erik Ejegod | 1095 | 1103 | DK | 8 |
| Niels | 1104 | 1134 | DK | 30 |
| Erik Emune | 1134 | 1137 | DK | 3 |

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

If you study the program code (the main program) it is:

```
public class ShowKingsProgram extends Application
{
 private KingTableModel model = new KingTableModel();

 @Override
 public void start(Stage stage)
 {
  TableView<King> table = new TableView(model.getKings());
  table.getColumns().addAll(model.getNameCol(), model.getPeriodCol(),
   model.getCountryCol(), model.getYearsCol());
  BorderPane root = new BorderPane(table);
  root.setPadding(new Insets(10, 10, 10, 10));
  Scene scene = new Scene(root, 500, 400);
  stage.setScene(scene);
  stage.setTitle("Show kings");
  stage.show();
 }

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

which is very simple. Initially, a model is defined (explained below). Otherwise, nothing happens except the method *start()*, where a *TableView* is created for *King* objects and initialized with the data model, and finally, four columns are added using methods in the data model. The respective *TableView* is inserted into the program's scene graph encapsulated in a *BorderPane*. It is the model class *KingTableModel* that contains the most:

```
public class KingTableModel
{
 private final ObservableList<King> kings = FXCollections.observableArrayList();

 public KingTableModel()
 {
  for (String[] arr : data) kings.add(new King(arr[0], Integer.parseInt(arr[1]),
   Integer.parseInt(arr[2])));
 }

 public ObservableList<King> getKings()
 {
  return kings;
 }
```

```
public TableColumn<King, String> getNameCol()
{
 TableColumn<King, String> col = new TableColumn("Name");
 col.setCellValueFactory(new PropertyValueFactory("name"));
 return col;
}

public TableColumn<King, Integer> getFromCol()
{
 TableColumn<King, Integer> col = new TableColumn("From");
 col.setCellValueFactory(new PropertyValueFactory("from"));
 return col;
}

public TableColumn<King, Integer> getToCol()
{
 TableColumn<King, Integer> col = new TableColumn("To");
 col.setCellValueFactory(new PropertyValueFactory("to"));
 return col;
}

public TableColumn<King, String> getCountryCol()
{
 TableColumn<King, String> col = new TableColumn("Country");
 col.setCellValueFactory(new PropertyValueFactory("country"));
 return col;
}

public TableColumn<King, String> getPeriodCol()
{
 TableColumn<King, String> col = new TableColumn<>("Period");
 col.getColumns().addAll(getFromCol(), getToCol());
 return col;
}

public TableColumn<King, String> getYearsCol()
{
 TableColumn<King, String> col = new TableColumn<>("Years");
 col.setCellValueFactory(c -> {
  King king = c.getValue();
  int a = king.getFrom();
  int b = king.getTo();
  if (a == 0 || b == 9999) return new ReadOnlyStringWrapper("Unknown");
  return new ReadOnlyStringWrapper("" + (b – a));
 });
 return col;
}
```

```
 private static final String[][] data = {
  { "Gorm den Gamle", "0", "958" },
  { "Harald Blåtand", "958", "987" },
  …
 };
}
```

The class starts by creating an *ObservableList* for *King* objects, and the list is initialized in the constructor using data defined in an array at the end of the class. Note that the list as an alternative could be initialized by reading a database table. The class has a method that can return the list, which was used in the method *start()* in the constructor for the *TableView* component. The rest of the class consists of methods that creates the individual columns. The first is for the *name* property, and the type of a column is *TableColumn*, with parameters indicating which the objects are (here *King*) and the type of the property in question (here *String*). The parameter of the constructor in *TableColumn* is the text shown in the header of the columns. Next, you must specify which values each cell should contain, and it happens with a *PropertyValueFactory*, where the parameter of the constructor is the name of the property in the class *King* that the cell must contain, and here it is *name*.

What do you want to do?

No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!

VOLVO
AB Volvo (publ)
www.volvogroup.com

Volvo Trucks I Renault Trucks I Mack Trucks I Volvo Buses I Volvo Construction Equipment I Volvo Penta I Volvo Aero I Volvo IT
Volvo Financial Services I Volvo 3P I Volvo Powertrain I Volvo Parts I Volvo Technology I Volvo Logistics I Business Area Asia

The two next columns are basically defined in the same way, just the type is this time *Integer*. However, these columns are not inserted directly in the table, but via another column created by the method *getPeriodCol()*. Columns can be nested and the column created by *getPeriodCol()* is a column consisting of two other columns. At the user interface, it corresponds to that the column with the header *Period* having two subcolumns, respectively *from year* and *to year*. In this case, there is no particular reason for it in addition to showing that it is possible and what the syntax is.

The class also creates a column for the property *country*, and here is nothing new, and actually the method *getCountryCol()* is not used. The goal is to show that you do not have to display all columns in the user interface.

Finally, there is the method *getYearCol()*, which shows how many years that king has ruled, or the text *Unknown* if you do not know the government period. This column is different, as there is no corresponding property in the *King* class, and the values to be displayed in the cells must therefore be calculated. This happens again using the method *setCellValueFactory()*. It has the following prototype:

```
setCellValueFactory(Callback<TableColumn.CellDataFeatures<S,T>,
 ObservableValue<T> value)
```

*Callback<P, R>* is an interface parameterized with two types, and the interface defines a single method

```
R call(P param)
```

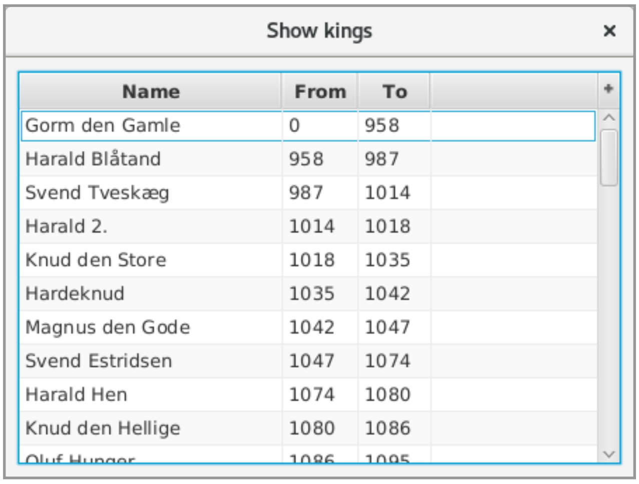
In this case for the column *Year*, it means that the cells of the column are initialized by a method of the form

```
ObservableValue<String> call(TableColumn.CellDataFeatures<King, String> value)
```

which is used to calculate the years in which the king has ruled. You can thus add your own custom-defined columns. You should note that the value is returned as a read-only property, as you can not edit such a value in the user interface.

If you run the program, note that by default you can change the columns width using the mouse, change the order of columns, and sort them by clicking the header.

The next example is called *MapKingProgram* and is essentially the same program and opens the following window:



| Name | From | To |
|---|---|---|
| Gorm den Gamle | 0 | 958 |
| Harald Blåtand | 958 | 987 |
| Svend Tveskæg | 987 | 1014 |
| Harald 2. | 1014 | 1018 |
| Knud den Store | 1018 | 1035 |
| Hardeknud | 1035 | 1042 |
| Magnus den Gode | 1042 | 1047 |
| Svend Estridsen | 1047 | 1074 |
| Harald Hen | 1074 | 1080 |
| Knud den Hellige | 1080 | 1086 |
| Oluf Hunger | 1086 | 1095 |

and that is, the program shows a *TableView* with 3 columns corresponding to three properties in the class *King*. However, the difference is that the objects (rows) that the component shows are not *King* objects, but instead, *Map<String, Object>* objects. There is no particular reason for that in this example besides showing the syntax, but in situations where the rows in the table do not match a domain object, the option can be used.

The class *King* is the same as in the previous example, but the class *KingTableModel* has been changed:

```
public class KingTableModel
{
 private final ObservableList<Map<String, Object>> kings =
  FXCollections.observableArrayList();

 public KingTableModel()
 {
  int id = 0;
  for (String[] arr : data)
  {
   King king =
    new King(arr[0], Integer.parseInt(arr[1]), Integer.parseInt(arr[2]));
   Map map = new HashMap<String, Object>();
   map.put("id", String.format("%s%02d", king.getCountry(), ++id));
   map.put("name", king.getName());
   map.put("from", king.getFrom());
   map.put("to", king.getTo());
   kings.add(map);
  }
 }
```

```
public ObservableList<Map<String, Object>> getKings()
{
 return kings;
}

public TableColumn<Map, String> getIdCol()
{
 TableColumn<Map, String> col = new TableColumn("Id");
 col.setCellValueFactory(new MapValueFactory("id"));
 return col;
}

public TableColumn<Map, String> getNameCol()
{
 TableColumn<Map, String> col = new TableColumn("Name");
 col.setCellValueFactory(new MapValueFactory("name"));
 return col;
}

public TableColumn<Map, Integer> getFromCol()
{
 TableColumn<Map, Integer> col = new TableColumn("From");
```

gaiteye®
Challenge the way we run
EXPERIENCE THE POWER OF FULL ENGAGEMENT...
RUN FASTER.
RUN LONGER..
RUN EASIER...
READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM

```
  col.setCellValueFactory(new MapValueFactory("from"));
  return col;
 }

 public TableColumn<Map, Integer> getToCol()
 {
  TableColumn<Map, Integer> col = new TableColumn("To");
  col.setCellValueFactory(new MapValueFactory("to"));
  return col;
 }

 private static final String[][] data = {
  { "Gorm den Gamle", "0", "958" },
  { "Harald Blåtand", "958", "987" },
  …
 };
}
```

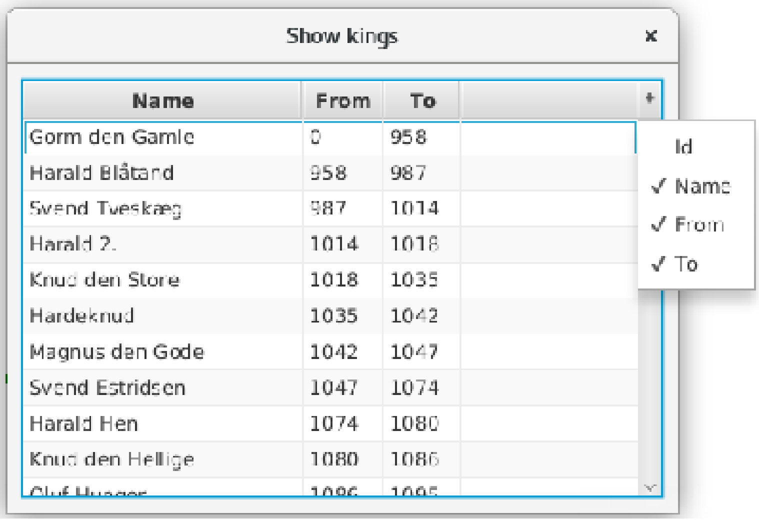
First, the collection *persons* is this time an *ObservableList* with objects, which are *Map<String, Object>*. The list is created in the constructor, and the difference is that this time, the Map objects must be instantiated where the key is a String while the value is an object (a *String* or an *Integer*). Note the keys that are created with a continuous number, but preceded the country code.

Then there are the methods that creates the columns. Column objects are created as in the first example, but the parameter to *setCellValueFactory()* is a *MapValueFactory* object, where the parameter of the constructor is the key. The result is that the columns are initialized with the values in the Map object that the key refers to.

In the method *start()* (the class *MapKingsProgram*), there are no major changes, and I do not want to show the code here, but after the table has been created and initialized with columns, two statements have been added:

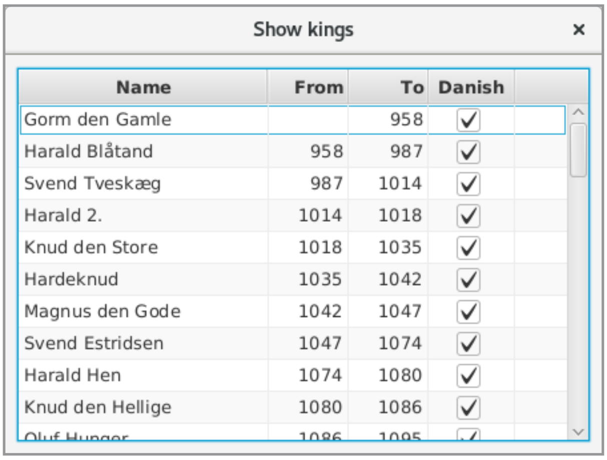
```
table.setTableMenuButtonVisible(true);
idCol.setVisible(false);
```

where *idCol* is the name of the first column with the rows *id* (and thus the key). The first statement indicates that it should be possible to hide columns, while the second statement indicates that *idCol* should be hidden from startup. If you run the program, you can notice that in the header line above the scroll bar there is a small plus. If you click on it, you'll get a little popup where you can click from or to if the individual columns should be hidden.



| Name | From | To |
|---|---|---|
| Gorm den Gamle | 0 | 958 |
| Harald Blåtand | 958 | 987 |
| Svend Tveskæg | 987 | 1014 |
| Harald 2. | 1014 | 1018 |
| Knud den Store | 1018 | 1035 |
| Hardeknud | 1035 | 1042 |
| Magnus den Gode | 1042 | 1047 |
| Svend Estridsen | 1047 | 1074 |
| Harald Hen | 1074 | 1080 |
| Knud den Hellige | 1080 | 1086 |

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



| Name | From | To | Danish |
|---|---|---|---|
| Gorm den Gamle | | 958 | ☑ |
| Harald Blåtand | 958 | 987 | ☑ |
| Svend Tveskæg | 987 | 1014 | ☑ |
| Harald 2. | 1014 | 1018 | ☑ |
| Knud den Store | 1018 | 1035 | ☑ |
| Hardeknud | 1035 | 1042 | ☑ |
| Magnus den Gode | 1042 | 1047 | ☑ |
| Svend Estridsen | 1047 | 1074 | ☑ |
| Harald Hen | 1074 | 1080 | ☑ |
| Knud den Hellige | 1080 | 1086 | ☑ |

and will show you how to define how the content of the individual cells should be displayed. In this case, the content of the *From* column appear as blank if the value is 0 corresponding to the meaning that you do not know the start of the king's government period. The same applies to the *To* column if the value is 9999. In addition, the two columns are right-aligned. Finally, the last column shows the value of the property *country* as a checkbox that is checked if the value is DK.

The objects that are displayed again have the type *King*, and the most important changes are again in the class *KingTableModel*:

```
public class KingTableModel
{
 private final ObservableList<King> kings = FXCollections.observableArrayList();


 …
```

```
public TableColumn<King, Integer> getFromCol()
{
 TableColumn<King, Integer> col = new TableColumn("From");
 col.setCellValueFactory(new PropertyValueFactory("from"));
 col.setCellFactory(c -> {
  TableCell<King, Integer> cell = new TableCell<King, Integer>()
  {
   @Override
   public void updateItem(Integer item, boolean empty)
   {
    super.updateItem(item, empty);
    this.setText(null);
    this.setGraphic(null);
    if (!empty && item != 0) this.setText("" + item);
   }
  };
  return cell;
 });
 col.setPrefWidth(60);
 col.setStyle("-fx-alignment: CENTER_RIGHT;");
 col.getStyleClass().add("salary-header");
 return col;
}
```

This e-book
*is made with*
**SetaPDF**

SETASIGN

**PDF** components for **PHP** developers

**www.setasign.com**

```
 public TableColumn<King, Boolean> getCountryCol()
 {
  TableColumn<King, Boolean> col = new TableColumn<>("Danish");
  col.setCellValueFactory(cell -> {
   King king = cell.getValue();
   return new ReadOnlyBooleanWrapper(king.getCountry().equals("DK"));
  });
  col.setCellFactory(CheckBoxTableCell.<King>forTableColumn(col));
  return col;
 }

 private static final String[][] data = {
  { "Gorm den Gamle", "0", "958" },
  …
 };
}
```

The start of the class is essentially the same as in the first example. That is, the construter, the method *getKing()* and the method *getNamCol()* are unchanged and are therefore not shown above. However, the method *getFromCol()* is different. It still has to return a *TableColumn* for the property *from*, and this column object is created as before, but then the method *setCellFactory()* is used to tell how the value should be displayed. This happens with a *TableCell* object, which is an interface that defines a method *update()*. The method has as parameter, the object to be rendered, and a *boolean* that tells if the cell is empty. If there is a non-zero content, it is used to update the cell. After being associated with a *TableCell*, a column is assigned a preferred width and a style is attached to the column and to its header, where the last is defined in a style sheet. The method *getToCol()* is written in the same way and does not appear here.

Then there is method *getCountryCol()*, which creates a *TableColumn* for *Boolean* objects. A *CellValueFactory* is assigned to specify which value a *King* object should result in, depending on the value of the *country* property. Is it *DK*, the value must be *true* and otherwise *false*. You should note that the value wrappers in a read-only property, and the reason is that the *TableView* component should be able to bind to the value. Finally, a *CellFactory* is associated with the column telling how the value should be displayed and that it should be like a *CheckBoxTableCell*, which is a wrapper for a checkbox. Note that you also indicate that the checkbox must be initialized with the value in the current column.

Then there is the main program with the method *start()*:

```
public void start(Stage stage)
{
 TableView<King> table = new TableView(model.getKings());
 table.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
 table.getSelectionModel().setCellSelectionEnabled(false);
 table.getSelectionModel().getSelectedIndices().addListener(
  (ListChangeListener.Change<? extends Integer> change) -> {
   String text = "";
   List<Integer> list = table.getSelectionModel().getSelectedIndices();
   for (Integer n : list) text += n + " ";
   System.out.println(text);
  });
 table.getColumns().addAll(model.getNameCol(), model.getFromCol(),
  model.getToCol(), model.getCountryCol());
 BorderPane root = new BorderPane(table);
 root.setPadding(new Insets(10, 10, 10, 10));
 Scene scene = new Scene(root, 450, 300);
 scene.getStylesheets().add("resources/css/styles.css");
 stage.setScene(scene);
 stage.setTitle("Show kings");
 stage.show();
}
```

It looks like the above examples, but should show a little about how to select rows. First, you should be able to select *MULTIPLE*, and you should not be able to select single cells. Next, an event handler that fires every time you change selection. The handler is trivial and does nothing but write a text on the console, a text that shows the indexes of the rows that are selected.

## EXERCISE 3

The database *padata* contains a table *zipcode* with Danish zip codes. You must write a program that you can call *PostProgram*. The program should only show the content of the database table in a *TableView*, and you should not do anything specifically about the formatting of the individual columns (there are only two).

## 3.2 EDIT CELLS IN A TABLEVIEW

I will now show an example that, in principle, looks like the above examples, but where you can edit the content of the individual cells. Although it may not be surprisingly new, it is still relatively simple. In general, it works that way that you double-click on the cell you want to edit, after which the cell opens with the option of changing the value. When the cell is opened, it happens by the cell shows another control that basically can be

1. *CheckBoxTableCell* (see the previous example)
2. *ChoiceBoxTableCell*
3. *ComboBoxTableCell*
4. *TextFieldTableCell*

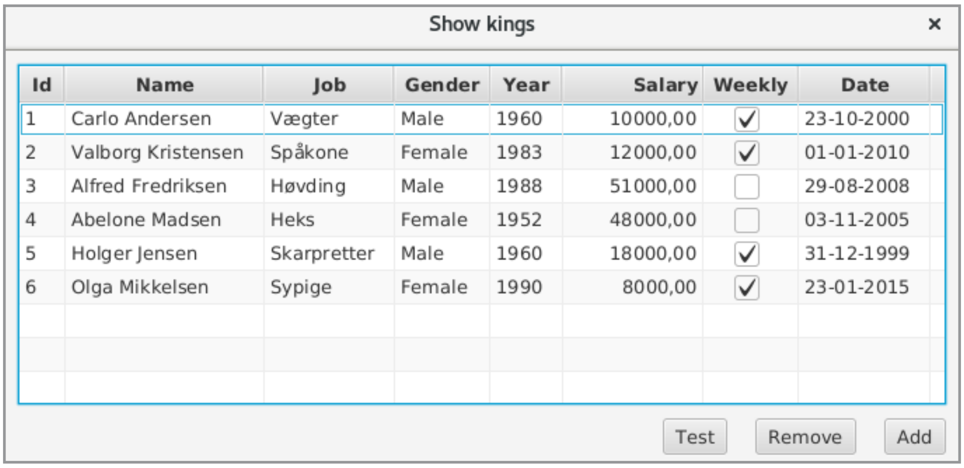
and you can also define your own controls. To show how it works, I will use another object type this time, which is a class that represents a Person:

```
public class Person implements Comparable<Person>
{
 private static int ID = 0;
 private final ReadOnlyIntegerWrapper id = new ReadOnlyIntegerWrapper();
 private final StringProperty name = new SimpleStringProperty();
 private final StringProperty job = new SimpleStringProperty();
 private final StringProperty gender = new SimpleStringProperty();
 private final IntegerProperty year = new SimpleIntegerProperty();
 private final DoubleProperty salary = new SimpleDoubleProperty();
 private final BooleanProperty weekly = new SimpleBooleanProperty();
 private final ObjectProperty<LocalDate> date = new SimpleObjectProperty();
```

Free eBook on
Learning & Development
By the Chief Learning Officer of McKinsey
Download Now
bookboon
Prof. Dr. Nick H.M. van Dam
21st Century Corporate Learning & Development
Trends and Best Practices

```
public Person(String name, String job, String gender, Integer year,
 Double salary, Boolean weekly, LocalDate date)
{
 id.set(++ID);
 setName(name);
 …
}
```

It’s a standard JavaFX model class, so I do not show the details here. The class is an extension of the class *Person* that I have used in the previous chapter, but this time, objects are assigned a current ID represented as a readonly property. In fact, the interpretation of the last four properties, it is not important, but it could be birth year (for *year*), salary (for *salary*) and whether the salary is weekly or monthly (for *weekly*). Finally, the last property could be interpreted as the date of employment. The important is not the interpretation, but that the parameters have different types. If you run the program, the result could be as shown below, where all values except the *Id* column can be edited. The *Add* button is used to add a new row to the table, thus creating a *Person*, while the *Remove* button is used to delete the row that is selected.



| Id | Name | Job | Gender | Year | Salary | Weekly | Date |
|---|---|---|---|---|---|---|---|
| 1 | Carlo Andersen | Vægter | Male | 1960 | 10000,00 | ☑ | 23-10-2000 |
| 2 | Valborg Kristensen | Spåkone | Female | 1983 | 12000,00 | ☑ | 01-01-2010 |
| 3 | Alfred Fredriksen | Høvding | Male | 1988 | 51000,00 | ☐ | 29-08-2008 |
| 4 | Abelone Madsen | Heks | Female | 1952 | 48000,00 | ☐ | 03-11-2005 |
| 5 | Holger Jensen | Skarpretter | Male | 1960 | 18000,00 | ☑ | 31-12-1999 |
| 6 | Olga Mikkelsen | Sypige | Female | 1990 | 8000,00 | ☑ | 23-01-2015 |

In addition to the class *Person*, the project has a class called *PersonModel*:

```
package editpersonsprogram;
import java.time.LocalDate;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;

public class PersonsModel
{
 private final ObservableList<Person> persons =
  FXCollections.observableArrayList();
 public PersonsModel()
 {
  initialize();
 }

 public ObservableList<Person> getPersons()
 {
  return persons;
 }

 public void add()
 {
  persons.add(new Person("", "", "", null, null, true, null));
 }

 public void remove(int n)
 {
  persons.remove(n);
 }

 private void initialize()
 {
  …
 }
}
```

which represents the persons on whom the program is working. The method *initialize()* creates 6 persons, so the table is not empty when the program starts. Note that the objects are stored in an *ObservableList* to *Person* objects and note how the methods *add()* and *remove()* maintains this list.

Otherwise I want to start with the class *EditPersonsProgram*:

```
public class EditPersonsProgram extends Application
{
 private PersonsModel model = new PersonsModel();
 private TableView<Person> table = null;

 @Override
 public void start(Stage stage)
 {
  table = new TableView(model.getPersons());
  PersonTableModel cols = new PersonTableModel();
  table.getColumns().addAll(cols.getIdCol(), cols.getNameCol(),
   cols.getJobCol(), cols.getGenderCol(), cols.getYearCol(),
   cols.getSalaryCol(), cols.getWeeklyCol(), cols.getDateCol());
  table.setEditable(true);
 BorderPane root = new BorderPane(table, null, null, createBottom(), null);
 root.setPadding(new Insets(10, 10, 10, 10));
 Scene scene = new Scene(root);
 scene.getStylesheets().add("resources/css/styles.css");
 stage.setScene(scene);
 stage.setTitle("Show kings");
 stage.show();
}

private Pane createBottom()
{
 HBox pane = new HBox(20, createButton("Test", this::test),
  createButton("Remove", this::remove), createButton("Add", e -> model.add()));
 pane.setAlignment(Pos.CENTER_RIGHT);
 pane.setPadding(new Insets(10, 0, 0, 0));
 return pane;
}

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

www.sylvania.com

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA

```
 private void remove(ActionEvent e)
 {
  int row = table.getSelectionModel().getSelectedIndex();
  if (row >= 0)
  {
   model.remove(row);
   table.getSelectionModel().clearSelection();
  }
 }

 private void test(ActionEvent e)
 {
  for (Person p : model.getPersons()) System.out.println(p);
  System.out.println();
 }

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

The class starts by defining a model object – not for the *TableView* component, but for the data that the program needs to maintain. In addition, a *TableView* for *Person* objects are defined. In the method *start()* where the table is created, there is not much new to explain, but you should note that the table columns are created by methods in a class *PersonTableModel* this time. Finally, note that the table is defined as *editable*:

```
table.setEditable(true);
```

and that's all you need to edit the cells if a column relates to read/write property and if the column has a *CellFactory* that is a *TableCell* control. You should also note the event handlers where the *Add* button handler is trivial, while the handler for the *Remove* button requires you to determine the index for the row that is selected. Finally, there is the handler for the *Test* button, which on the console prints the objects in the list. The goal of this handler is to test whether there is consistency with what the *TableView* control shows and what the model contains. It is to show that changes in the *TableView* control automatically updates the model.

Then there is the class *PersonTableModel* that has methods that create the table columns. On the other hand, it is also the most complex of the program's classes.

```
public class PersonTableModel
{
 public TableColumn<Person, Integer> getIdCol()
 {
  TableColumn<Person, Integer> idCol = new TableColumn("Id");
  idCol.setCellValueFactory(new PropertyValueFactory("id"));
  return idCol;
 }

 public TableColumn<Person, String> getNameCol()
 {
  TableColumn<Person, String> col = new TableColumn("Name");
  col.setCellValueFactory(new PropertyValueFactory("name"));
  col.setCellFactory(TextFieldTableCell.<Person>forTableColumn());
  return col;
 }

 public TableColumn<Person, String> getJobCol()
 {
  …
 }

 public TableColumn<Person, String> getGenderCol()
 {
  TableColumn<Person, String> col = new TableColumn("Gender");
  col.setCellValueFactory(new PropertyValueFactory("gender"));
  col.setCellFactory(ChoiceBoxTableCell.<Person, String>forTableColumn(
   "Male", "Female"));
  return col;
 }

 public TableColumn<Person, Integer> getYearCol()
 {
  TableColumn<Person, Integer> col = new TableColumn("Year");
  col.setCellValueFactory(new PropertyValueFactory("year"));
  col.setCellFactory(ComboBoxTableCell.<Person,
   Integer>forTableColumn(new YearConverter(), getYears()));
  return col;
 }

 public TableColumn<Person, Double> getSalaryCol()
 {
  TableColumn<Person, Double> col = new TableColumn("Salary");
  col.setCellValueFactory(new PropertyValueFactory("salary"));
  col.setCellFactory(TextFieldTableCell.<Person,
   Double>forTableColumn(new SalaryConverter()));
  col.setOnEditCommit((TableColumn.CellEditEvent<Person, Double> e) -> {
   int row = e.getTablePosition().getRow();
  Person person = e.getTableView().getItems().get(row);
  if (Math.abs(e.getNewValue()) < 0.1)
  {
   e.getTableView().getItems().set(row, person);
  }
  else person.setSalary(e.getNewValue());
 });
 col.setPrefWidth(100);
 col.setStyle("-fx-alignment: CENTER_RIGHT;");
 col.getStyleClass().add("salary-header");
 return col;
}

public TableColumn<Person, Boolean> getWeeklyCol()
{
 TableColumn<Person, Boolean> col = new TableColumn("Weekly");
 col.setCellValueFactory(new PropertyValueFactory("weekly"));
 col.setCellFactory(CheckBoxTableCell.<Person>forTableColumn(col));
 return col;
}
```

360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

```
 public TableColumn<Person, LocalDate> getDateCol()
 {
  TableColumn<Person, LocalDate> col = new TableColumn("Date");
  col.setCellValueFactory(new PropertyValueFactory("date"));
  col.setCellFactory(DatePickerTableCell.<Person>forTableColumn());
  return col;
 }

 private ObservableList<Integer> getYears()
 {
  int year = LocalDate.now().getYear() - 10;
  ObservableList<Integer> list = FXCollections.observableArrayList();
  for (int y = year - 90; y < year; ++y) list.add(y);
  return list;
 }


lass YearConverter extends StringConverter<Integer>


 @Override
 public Integer fromString(String string)
 {
  try
  {
   return Integer.parseInt(string);
  }
  catch (Exception ex)
  {
   return 0;
  }
 }

 @Override
 public String toString(Integer value)
 {
  return value == 0 ? "" : "" + value;
 }


lass SalaryConverter extends StringConverter<Double>


 @Override
 public Double fromString(String string)
 {
  try
  {
   return Double.parseDouble(string);
  }
```

```
   catch (Exception ex)
   {
    return 0.0;
   }
  }

  @Override
  public String toString(Double value)
  {
   return Math.abs(value) < 0.1 ? "" :
    String.format("%1.2f", value.doubleValue());
  }
 }
```

The first method returns a *TableColumn* for the column *id* and is the simplest, and all you need to note is that it does not have a *CellFactory* as the column is not editable.

The two next methods are in principle identical, as in both cases it is a column whose values are text. Therefore, they have a *CellFactory* which is a *TextFieldTableCell*, and the result is if you double-click in a cell in these columns, a *TextField* control opens where the content can be edited. You should note that it is the method *forTableColumn()* that opens the entry field.

Then there is the method *getGenderCol()* that returns a *TableColumn* for the *Gender* column. It has a *CellFactory* of the type *ChoiceBoxTableCell*, and the result is, if you double-click in a cell, a *ChoiseBox* with two values: *Male* and *Female*, where the user can select a value. Here you must note the method *forTableColumn()*, which as parameters has the values for which the ChoiceBox control should be initialized.

The next method is for the year column, and it uses a *ComboBoxTableCell* with the result that the cell opens a *ComboBox*. The class has a private method that creates an *ObservableList* with the years to be selected. It is used as a parameter for *forTableColumn()*, but there is also a *YearConverter* parameter (as defined at the end of the file). There is generally an override of *forTableColumn()*, where you can specify a converter as parameter, and in this case it is only to ensure that a missing year is not displayed as 0.

Then there is the method *getSalaryCol()*, which returns a *TableColumn* for a *Double*, and where it should be possible to edit a *Double*. Here are several things to notice. First, as *CellFactory*, a *TextFieldTableCell* is used to enter a random decimal number. In order for the result to look nice, a converter of the type *SalaryConveter*, which shows a Double with two decimal, is attached – but only if the number is not 0. If that happens, the result will appear as blank. When the user enters a number, they can of course enter something illegal, and if that is the case (the entered can not be converted to a number), an exception appears with the result that the value is set to 0. However, the problem is that the model is then updated with the value 0, which is not really the thought, but the model should retains the old value. The problem is solved by adding an event handler that is executed when the entry is completed (and the *TextField* component closes). If the value is 0, the old value is set, and otherwise the new value. Reintroducing the old value looks a bit weird and in fact the model still has the old value, but the user interface is not updated. The problem is solved by assigning the *Person* object to itself, which means that the model fires an event that updates the user interface. As the last, the column content is right aligned, but it is the same as in the previous example.

Then there is the method *getWeeklyCol()* that uses a *CheckBoxTableCell*. Here is not much new, but note the parameter for *forTableColumn()*, which is the column to be edited.

Finally, there is *getDateCol()*, and the goal of this method is to show a column that uses a custom *CellFactory*. It is a *DatePickerTableCell* since it is desired that the user should get a *DatePicker* by double-clicking the cell. A custom *CellFactory* is a class that implements the interface *TableCell* and otherwise has a control of the desired kind (in this case a *DatePicker*). The class must be able to override the following methods

SIMPLY CLEVER
ŠKODA

We will turn your CV into
an opportunity of a lifetime

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

- *startEdit()*
- *commitEdit()*
- *cancelEdit()*
- *updateItem()*

but otherwise the class consists of constructors and static *forTableColumn()* methods. Below, only the overriding methods of the class are shown:

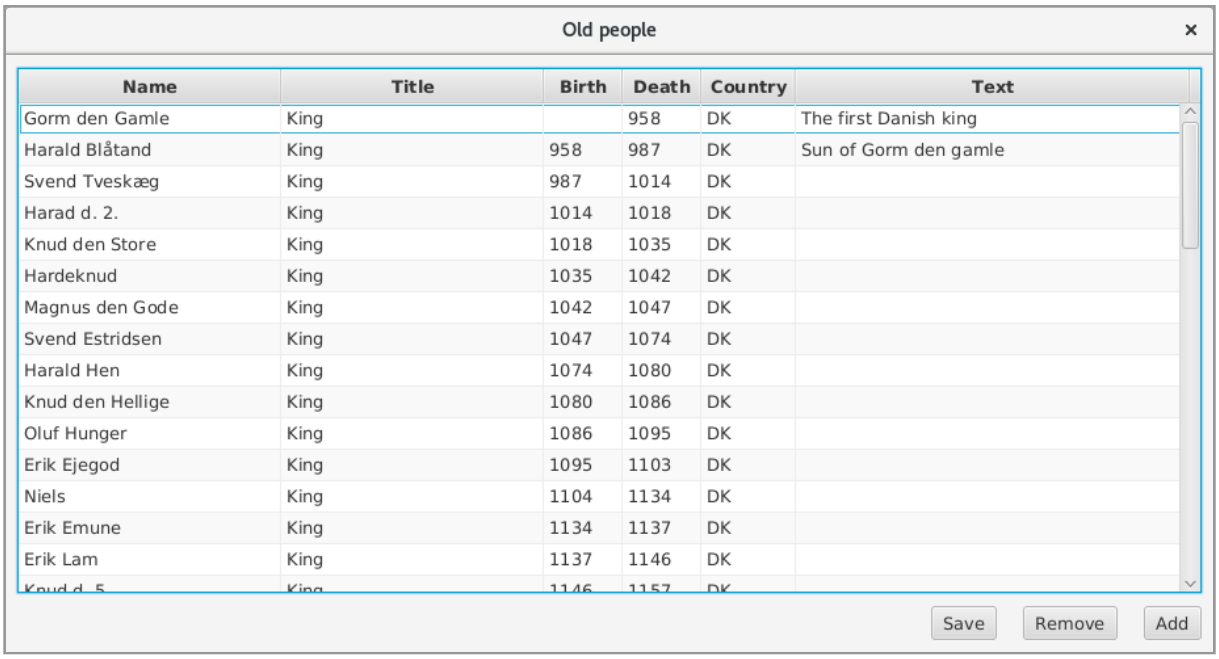
```
public class DatePickerTableCell<S, T> extends TableCell<S, LocalDate>
{
 private DatePicker datePicker;
 private StringConverter converter = null;
 private boolean editable = true;


 …


 @Override
 public void startEdit()
 {
  if (!isEditable() || !getTableView().isEditable() ||
   !getTableColumn().isEditable()) return;
  super.startEdit();
  if (datePicker == null) this.createDatePicker();
  setGraphic(datePicker);
 }

 @Override
 public void cancelEdit()
 {
  super.cancelEdit();
  setText(converter.toString(getItem()));
  setGraphic(null);
 }

 @Override
 public void updateItem(LocalDate item, boolean empty)
 {
  super.updateItem(item, empty);
  if (empty)
  {
   setText(null);
   setGraphic(null);
  }
  else
  {
    if (this.isEditing())
    {
     if (datePicker != null) datePicker.setValue((LocalDate)item);
     setText(null);
     setGraphic(datePicker);
    }
    else
    {
     setText(converter.toString(item));
     setGraphic(null);
    }
   }
  }

  …

}
```

When you run the program, including editing persons as well as creating new ones and deleting existing ones, please note that the model is automatically updated and the window is automatically updated (*Add* and *Remove* buttons). All this happens because of bidirectional bindings. To test that the window has the *Test* button.

## PROBLEM 2

You must solve the same task as in problem 1, but the user interface should be as shown in the window below, where the individual persons appear as a row in a *TableView*. Clicking the *Add* button will add a new person (where all fields are blank) to the list and if you click the *Remove* button, the person selected must be deleted. All cells must be editable and for the *Name*, *Title*, *Birth*, and *Death* columns, a *TextFieldTableCell* should be used, while for the *Country* column a *ComboBoxTableCell* should be used. The database has a table country, and the combobox must be initialized with all the country codes from this table. Back there is column *Text*, where the cells must be edited with a *TextArea* control and thus a custom *TableCell*. Note that it may mean that you need to add one way or another to finish entering text, for example, by typing F12.

A particular problem is when physically writing to the database. You can of course write each time you create a person and delete a person, but writing each time you edit a cell may not be appropriate. You can therefore choose a different strategy, where you write back all changes when you click on a button. That's the purpose of the *Save* button. In my solution, I simply update all rows (with a batch update) when clicking on the *Save* button. That's probably the easy solution, as it means I'll update all rows, regardless of whether they are changed. Perhaps you can find a better solution?



| Name | Title | Birth | Death | Country | Text |
|---|---|---|---|---|---|
| Gorm den Gamle | King | | 958 | DK | The first Danish king |
| Harald Blåtand | King | 958 | 987 | DK | Sun of Gorm den gamle |
| Svend Tveskæg | King | 987 | 1014 | DK | |
| Harad d. 2. | King | 1014 | 1018 | DK | |
| Knud den Store | King | 1018 | 1035 | DK | |
| Hardeknud | King | 1035 | 1042 | DK | |
| Magnus den Gode | King | 1042 | 1047 | DK | |
| Svend Estridsen | King | 1047 | 1074 | DK | |
| Harald Hen | King | 1074 | 1080 | DK | |
| Knud den Hellige | King | 1080 | 1086 | DK | |
| Oluf Hunger | King | 1086 | 1095 | DK | |
| Erik Ejegod | King | 1095 | 1103 | DK | |
| Niels | King | 1104 | 1134 | DK | |
| Erik Emune | King | 1134 | 1137 | DK | |
| Erik Lam | King | 1137 | 1146 | DK | |
| Knud d. 5 | King | 1146 | 1157 | DK | |

e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

**About e-Learning for Kids** Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

## 3.3 FILTERS

One of the important features of a *JTable* is the use of filters, and with a *TableView*, it's all a bit easier. A *TableView* shows the content of an *ObersvableList* and you can set a filter with a filter wrapper to the list and encapsulated in a sort wrapper. To conclude this review of the *TableView* component, I will show an example that sets a filter.

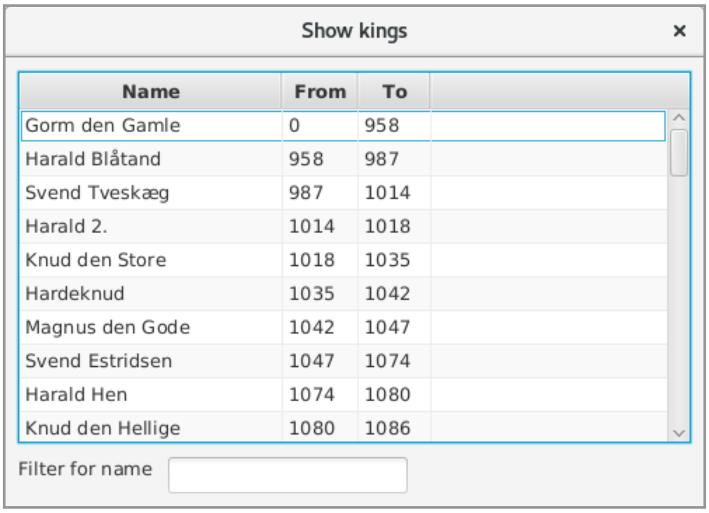
The example is called *FilterKingsProgram* and I want to reuse objects of the type *King*, and the data model for the *TableView* component is essentially unchanged from previously and will not be shown here.

The window's code is:

```
public class FilterKingsProgram extends Application
{
 private TextField txtName = new TextField();

 @Override
 public void start(Stage stage)
 {
  TableView<King> table = new TableView();
  table.getColumns().addAll(KingTableModel.getNameCol(),
   KingTableModel.getFromCol(), KingTableModel.getToCol());
  initialize(table);
  BorderPane root = new BorderPane(table, null, null, createBottom(), null);
  root.setPadding(new Insets(10, 10, 10, 10));
  Scene scene = new Scene(root, 500, 400);
  stage.setScene(scene);
  stage.setTitle("Show kings");
  stage.show();
 }

 private void initialize(TableView table)
 {
  FilteredList<King> filter =
   new FilteredList<>(KingTableModel.getKings(), king -> true);
  txtName.textProperty().addListener((observable, oldValue, newValue) ->
  {
   filter.setPredicate(king ->
   {
    if (newValue == null || newValue.isEmpty()) return true;
    String lowerCaseFilter = newValue.toLowerCase();
    if (king.getName().toLowerCase().contains(lowerCaseFilter)) return true;
    return false;
   });
  });
  SortedList<King> sorter = new SortedList<>(filter);
  sorter.comparatorProperty().bind(table.comparatorProperty());
  table.setItems(sorter);
 }

 private Pane createBottom()
 {
  HBox pane = new HBox(10, new Label("Filter for name"), txtName);
  pane.setPadding(new Insets(10, 0, 0, 0));
  return pane;
 }

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

There is nothing new in the *start()* method, and the new is done in the method *initialize()*, which initializes the table control. Note first that a filter is created for the data to be shown in the table, and the type for a filter is *FilteredList*. Next, a listener must be associated with the input field, which is defined as a predicat for the filter, and as in this case returns true if a *King* object's name property contains the value of the input field – without distinguishing between upper and lowercase letters. Finally the filter is inserted in a wrapper as a *SortedList*, which has a comparator that binds to the table's comparator. Finally, a *sorter* object is used as the data model for the table.

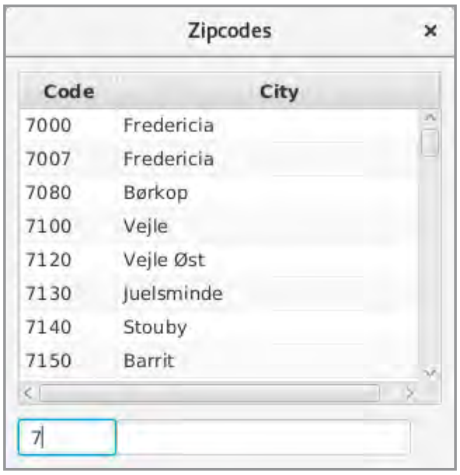
If you run the program, you get the following window, where at the bottom there is a entry field used to enter a filter for the *Name* column:



| Name | From | To |
|---|---|---|
| Gorm den Gamle | 0 | 958 |
| Harald Blåtand | 958 | 987 |
| Svend Tveskæg | 987 | 1014 |
| Harald 2. | 1014 | 1018 |
| Knud den Store | 1018 | 1035 |
| Hardeknud | 1035 | 1042 |
| Magnus den Gode | 1042 | 1047 |
| Svend Estridsen | 1047 | 1074 |
| Harald Hen | 1074 | 1080 |
| Knud den Hellige | 1080 | 1086 |

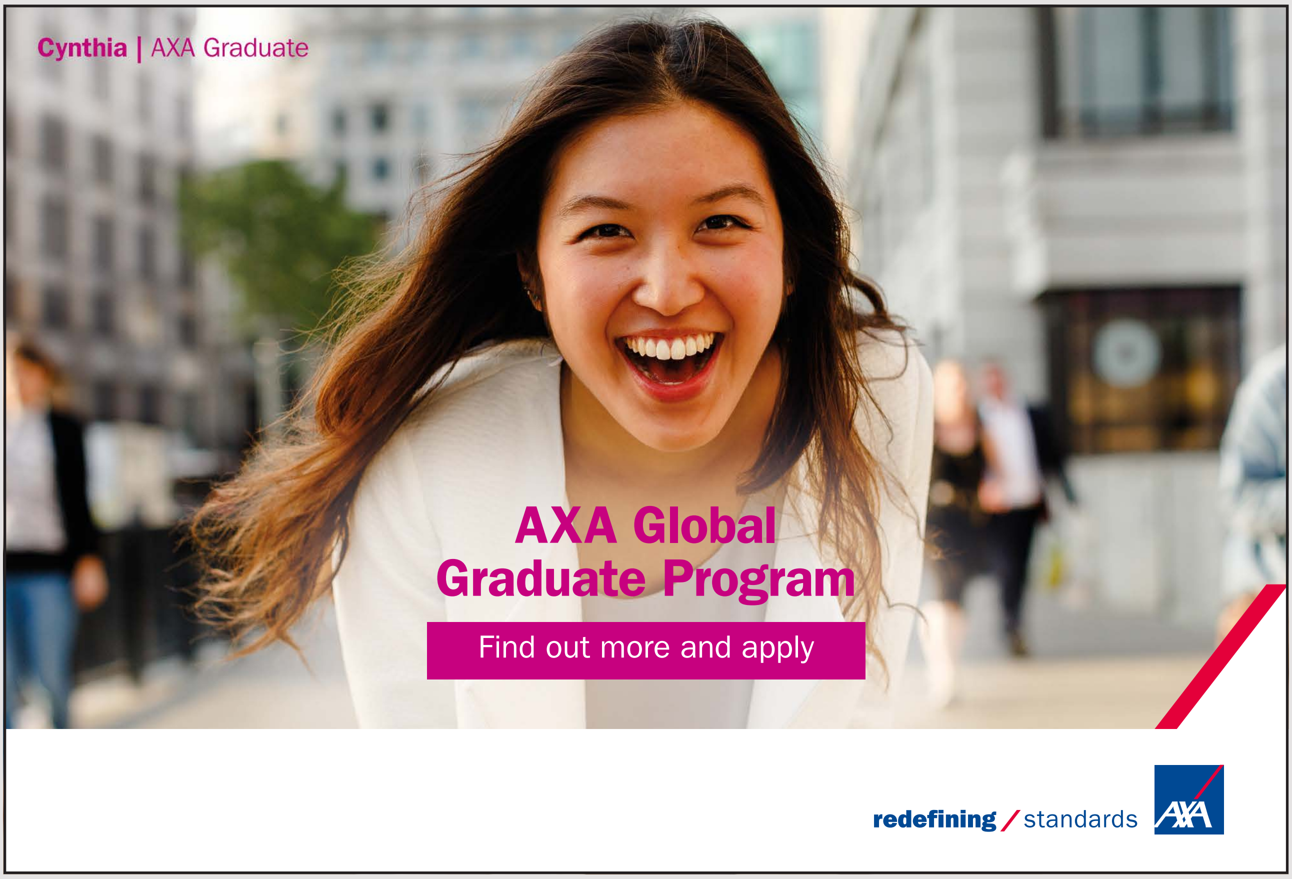
## EXERCISE 4

Create a copy of the project *PostProgram* from exercise 3. You need to expand the program so it has a filter for both zip code and city name:



| Code | City |
|---|---|
| 7000 | Fredericia |
| 7007 | Fredericia |
| 7080 | Børkop |
| 7100 | Vejle |
| 7120 | Vejle Øst |
| 7130 | Juelsminde |
| 7140 | Stouby |
| 7150 | Barrit |

7



**Cynthia** | AXA Graduate

AXA Global Graduate Program

Find out more and apply

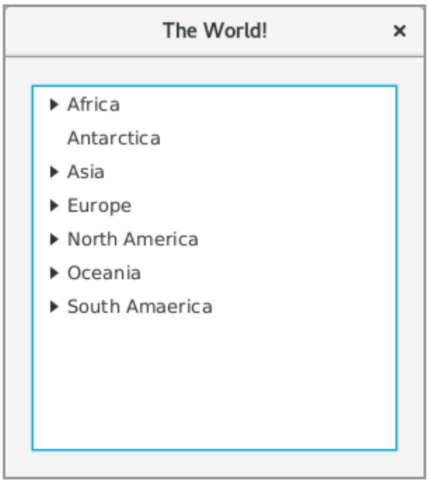
**redefining** / standards

AXA

## 3.4 A TREEVIEW CONTROL

JavaFX also has an *TreeView* control, which corresponds to a *JTree* from Swing and is a control that visualizes hierarchical data. Basically, there are two classes *TreeItem* and *TreeView*, the first representing an element in the tree, while the latter is the actual component. A *TreeItem* is either a composite or a leaf and it is determined by whether it has child elements.

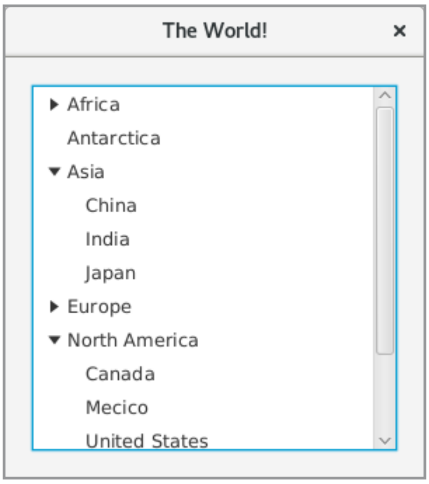
The program *ShowWorldProgram* shows some of the world's countries organized in a hierarchy:



The World!

▸ Africa
Antarctica
▸ Asia
▸ Europe
▸ North America
▸ Oceania
▸ South Amaerica

Above, all continents in these world, which all of the Antarctica are composite nodes, as are shown with the arrow, showing that it is a node with child nodes and that it can be expanded by clicking the arrow with the mouse. Below is the same window where Asia and North America are expanded (you should note that the component automatically displays a scrollbar when necessary):



The World!

▸ Africa
Antarctica
▾ Asia
    China
    India
    Japan
▸ Europe
▾ North America
    Canada
    Mecico
    United States

A *TreeView* shows data defined in a model, which is a hierarchy of *TreeItem* elements, and in this case, the model is defined in the method *build()*:

```
package showworldprogram;

import javafx.scene.control.*;

public class TreeWorldModel
{
 private TreeItem<String> root = new TreeItem("This World");

 public TreeWorldModel()
 {
  build();
 }

 public TreeItem<String> getData()
 {
  return root;
 }

 private void build()
 {
  TreeItem<String> af = new TreeItem("Africa");
  af.getChildren().addAll(new TreeItem("South Africa"), new TreeItem("Namibia"),
   new TreeItem("Botswana"), new TreeItem("Zimbabwe"));
  TreeItem<String> an = new TreeItem("Antarctica");
  TreeItem<String> as = new TreeItem("Asia");
  as.getChildren().addAll(new TreeItem("China"), new TreeItem("India"),
   new TreeItem("Japan"));
  TreeItem<String> eu = new TreeItem("Europe");
  eu.getChildren().addAll(new TreeItem("Denmark"), new TreeItem("Norway"),
   new TreeItem("Sweden"));
  TreeItem<String> na = new TreeItem("North America");
  na.getChildren().addAll(new TreeItem("Canada"), new TreeItem("Mecico"),
   new TreeItem("United States"));
  TreeItem<String> oc = new TreeItem("Oceania");
  oc.getChildren().addAll(new TreeItem("Australia"),
   new TreeItem("New Zealand"));
  TreeItem<String> sa = new TreeItem("South Amaerica");
  sa.getChildren().addAll(new TreeItem("Argentina"), new TreeItem("Brazil"),
    new TreeItem("Chile"), new TreeItem("Bolivia"), new TreeItem("Peru"));
  root.getChildren().addAll(af, an, as, eu, na, oc, sa);
 }
}
```

The code for the program is quite simple:

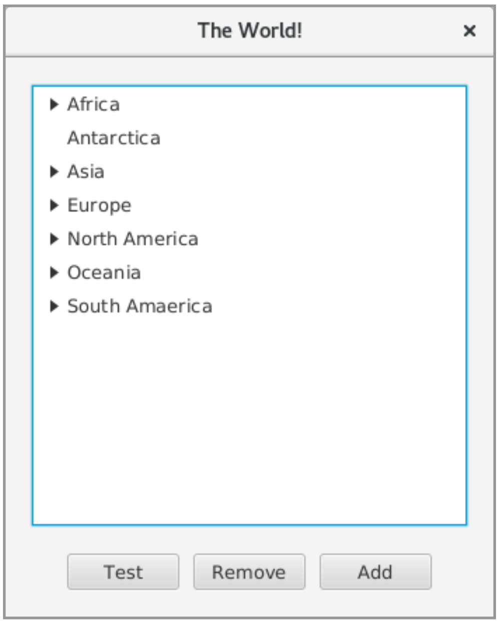
```
package showworldprogram;

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

public class ShowWorldProgram extends Application
{
 @Override
 public void start(Stage primaryStage)
 {
  TreeView view = new TreeView((new TreeWorldModel()).getData());
  view.setShowRoot(false);
  BorderPane root = new BorderPane(view);
  root.setPadding(new Insets(20, 20, 20, 20));
  Scene scene = new Scene(root, 300, 300);
  primaryStage.setTitle("The World!");
  primaryStage.setScene(scene);
```

I joined MITAS because
I wanted real responsibility
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

```
  primaryStage.show();
 }

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

and there is nothing to explain. However, you must note that with *setShowRoot()* I defined that the root of the tree should not be displayed. You can try to set a comment in front of the line and see what happens.

The next example is called *MaintainWorldProgram* and is a variation of the above program and opens the following window:


The World!
Africa
Antarctica
Asia
Europe
North America
Oceania
South Amaerica
Test
Remove
Add

As shown by the buttons, one of the differences is that you should also be able to edit the content of the tree:

1. Clicking the *Add* button adds a node to the item that has been selected – if it is not a leaf node
2. Clicking on the *Delete* button deletes the node that is selected, but only if it is a leaf node
3. Double-clicking a leaf node will allow you to change the name

Finally, there is a button *Test* that only makes it possible to print the number of leaf nodes in the tree, and you should note that it is number of leaf nodes in the *TreeView* component's model, and the button is used to test that the above operations not only update the *TreeView* controller but also the model.

The program should also show something about what events occurs when you use a *TreeView*.

The model for the *TreeView* component is the same as in the previous example and is not shown here. The class *MaintainWorldProgram*, on the other hand, fills a part, so I will just show the code for the most important methods.

The method *start()* is essentially unchanged, but there are now instance variables for both the *TreeView* control and the model:

```
public class MaintainWorldProgram extends Application
{
 private TreeWorldModel model = new TreeWorldModel();
 private TreeView<String> view;
```

Otherwise, most important in the method *start()* are that it calls a method *addHandlers()*, which adds event handlers to the *TreeView* component:

```
private void addHandlers()
{
 model.getData().addEventHandler(TreeItem.<Str
 ing>branchExpandedEvent(), e -> );
 model.getData().addEventHandler(TreeItem.<Stri
 ng>branchCollapsedEvent(), e -> );
 model.getData().addEventHandler(TreeItem.<Str
 ing>childrenModificationEvent(), );
 model.getData().addEventHandler(TreeItem.<St
 ring>valueChangedEvent(), e -> );
 view.setOnMouseClicked(new EventHandler<MouseEvent>()
 {
  @Override
  public void handle(MouseEvent mouseEvent)
  {
   if(mouseEvent.getClickCount() == 2)
   {
    TreeItem<String> item = view.getSelectionModel().getSelectedItem();
    if (item != null) modify(item);
   }
  }
 });
}
```

There are a total of 5 event handlers. The names tell you when the event handlers are performed. The first three (where only the code for first is shown) does nothing but print a text on the console. The fourth uses a method *printModel()* that prints the contents of a subtree on the console and actually prints the entire tree. Its purpose is to show that the model is updated and you are encouraged to examine the code. The last event handler concerns the mouse and tests for double click with the mouse. If so, is executed the method *modify()*:

```
private void modify(TreeItem<String> item)
{
 if (item.isLeaf())
 {
  TextInputDialog dialog = new TextInputDialog(item.getValue());
  dialog.setTitle("Modify country");
  dialog.setHeaderText("Change the country's name");
  Optional<String> result = dialog.showAndWait();
  if (result.isPresent())
  {
   String name = result.get().trim();
   if (name.length() > 0) item.setValue(name);
  }
  view.getSelectionModel().clearSelection();
 }
}
```

ie business school

93%
OF MIM STUDENTS ARE
WORKING IN THEIR SECTOR 3 MONTHS
FOLLOWING GRADUATION

MASTER IN MANAGEMENT

- STUDY IN THE CENTER OF MADRID AND TAKE ADVANTAGE OF THE UNIQUE OPPORTUNITIES THAT THE CAPITAL OF SPAIN OFFERS
- PROPEL YOUR EDUCATION BY EARNING A DOUBLE DEGREE THAT BEST SUITS YOUR PROFESSIONAL GOALS
- STUDY A SEMESTER ABROAD AND BECOME A GLOBAL CITIZEN WITH THE BEYOND BORDERS EXPERIENCE

Length: 10 MONTHS
Av. Experience: 1 YEAR
Language: ENGLISH / SPANISH
Format: FULL-TIME
Intakes: SEPT / FEB

5 SPECIALIZATIONS
PERSONALIZE YOUR PROGRAM

#10 WORLDWIDE
MASTER IN MANAGEMENT
FINANCIAL TIMES

55 NATIONALITIES
IN CLASS

www.ie.edu/master-management | mim.admissions@ie.edu | Follow us on IE MIM Experience

In the case of a leaf node, a simple *TextInputDialog* opens where you can change the value. Doing so, updates the current *TreeItem*, and you should note that it also updates the user interface.

The event handler for the *Add* button is in principle identical, while the event handler for the *Remove* button is:

```
private void remove(ActionEvent e)
{
 TreeItem<String> item = view.getSelectionModel().getSelectedItem();
 if (item != null && item.isLeaf() && item.getParent() != null)
 {
  TreeItem<String> parent = item.getParent();
  parent.getChildren().remove(item);
  view.getSelectionModel().clearSelection();
 }
}
```

Here you should note how to refer to the item that is selected and again that the user interface is automatically updated. The last event handler is about the Test button counting the number of leaf nodes and then opening an *Alert* with the result.

In the above example, the content of a *TreeItem* are edited by opening a simple *TextInputDialog*, but you can also edit the content inline like a *TableView*. It is used in the example *EditWorldProgram*. The user interface is the same as above, but there is a minor change to the model:

```
public class TreeWorldModel
{
 private TreeItem<String> root = new TreeItem("This World");

 public TreeWorldModel()
 {
  build();
 }

 public TreeItem<String> getData()
 {
  return root;
 }

 public void add(TreeItem<String> parent)
 {
  parent.getChildren().add(new TreeItem(""));
 }
 public void remove(TreeItem item)
 {
  item.getParent().getChildren().remove(item);
 }
 …
}
```

The method *start()* is:

```
public void start(Stage primaryStage)
{
 view = new TreeView(model.getData());
 view.setShowRoot(false);
 view.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
 view.setEditable(true);
 view.setCellFactory(TextFieldTreeCell.forTreeView());
 view.setOnEditStart(this::start);
 view.setOnEditCommit(this::commit);
 view.setOnEditCancel(this::cancel);
 BorderPane root = new BorderPane(view, null, null, createCommands(), null);
 root.setPadding(new Insets(20, 20, 20, 20));
 Scene scene = new Scene(root, 350, 400);
 primaryStage.setTitle("This World");
 primaryStage.setScene(scene);
 primaryStage.show();
}
```

Here you should note that the *TreeView* component must be defined as editable, and in addition, a *CellFactory* must be attached, as here is a *TextFieldTreeCell*. This means that if you double-click on a *TreeItem*, a *TextField* opens, allowing you to edit the content. Just as you've seen it for a *TableView*, there are also other *CellFactory* types that can be used for other data types. There are also associated three event handlers that do nothing but print a text on the console and in the example they are used to show that the first is executed when the entry field opens while the other two are executed depending on whether you quit with Enter or ESC.

The method *createCommands()* creates a *Pane* with the three buttons, and the associated event handlers are all simple and use the two new methods in the model.

# EXERCISE 5

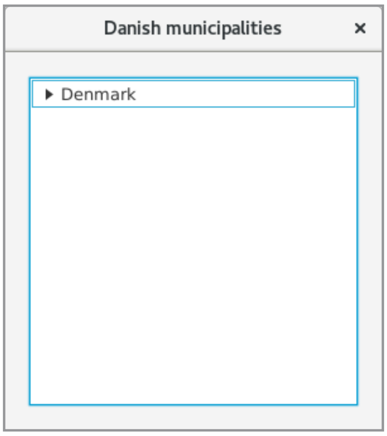
The database *padata* has four tables that contain information about Danish regions, Danish municipalities and Danish zip codes, while the fourth table represents a many-many relationship between *municipality* and *zipcode*. The four tables were created with the following script:

```
create table region
(
 regnr int not null primary key,
 name varchar(30) not null
);

create table municipality
(
 munnr int not null primary key,
 name varchar(30) not null,
 regnr int not null,
 area decimal(10, 2),
 number int,
 year int,
 foreign key (regnr) references region(regnr)
);
```

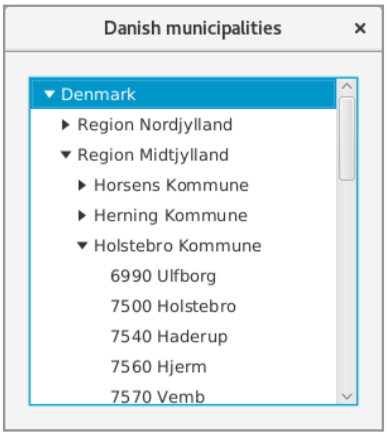
"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

```
create table zipcode
(
 code char(4) not null primary key,
 city varchar(30) not null
);

create table post
(
 code char(4) not null,
 munnr int not null,
 primary key (code, munnr),
 foreign key (code) references zipcode(code),
 foreign key (munnr) references municipality(munnr)
);
```

You must write a program that you can call *Denmark*, which displays this data in an *TreeView*. Note that the program only should display names, but you should not be able to edit the content. When the program starts, you should see the following window:

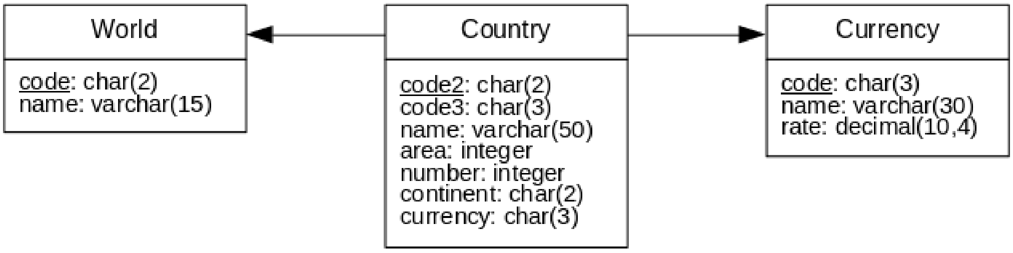

Danish municipalities
×
Denmark

and below the same window after the root, *Region Midtjylland* and the municipality *Holstebro* are expanded:

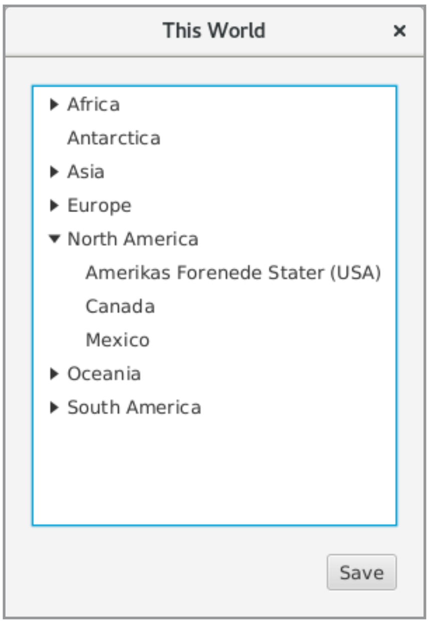

Danish municipalities
×
Denmark
Region Nordjylland
Region Midtjylland
Horsens Kommune
Herning Kommune
Holstebro Kommune
6990 Ulfborg
7500 Holstebro
7540 Haderup
7560 Hjerm
7570 Vemb

## 3.5 A TREEVIEW WITH COUNTRY OBJECTS

As a final example of using a *TreeView*, I will show a program called *UpdateWorldProgram*. The program is similar to the previous one, but partly the individual objects are of a custom type and not just strings, and partly there are more data. The database *padata* (see, if applicable, the book Java 6) contains three tables that contain data about currencies, continents and countries:


World
code: char(2)
name: varchar(15)
Country
code2: char(2)
code3: char(3)
name: varchar(50)
area: integer
number: integer
continent: char(2)
currency: char(3)
Currency
code: char(3)
name: varchar(30)
rate: decimal(10,4)

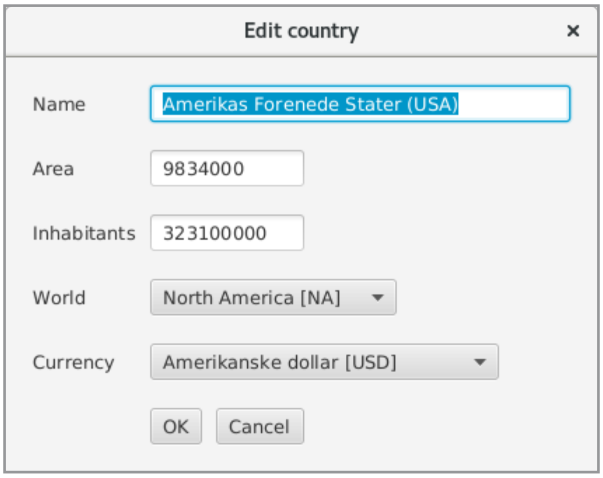
where the last two columns in the country table are foreign keys. The program will show an overview of all countries, but organized in an *TreeView*, and where it should be possible to edit the information about a country. If you open the program, you get the following window where North America is expanded:


This World
×

- Africa
- Antarctica
- Asia
- Europe
- North America
  - Amerikas Forenede Stater (USA)
  - Canada
  - Mexico
- Oceania
- South America

Save

For example, if you double-click on the United States, you will see the following window where you can edit data about the United States:


Edit country

Name: Amerikas Forenede Stater (USA)

Area: 9834000

Inhabitants: 323100000

World: North America [NA]

Currency: Amerikanske dollar [USD]

The two comboboxes contains all continents and all currencies, respectively, and are used to change a country's currency or continent. Note that the above window with the *TreeView* component does not show all the world's countries as only the countries in the database where a foreign key to the table *world* is included. You should also note that the program does not provide opportunities to create new countries or delete existing countries, but of course it would be easy to expand the program with these features.

Excellent Economics and Business programmes at:
university of groningen
AACSB ACCREDITED
"The perfect start of a successful, international career."
CLICK HERE to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

As a first step, model classes must be written to the three database tables. For *currency*, it is the following class:

```
public class Currency
{
 private ReadOnlyStringWrapper code = new ReadOnlyStringWrapper(this, "code");
 private ReadOnlyStringWrapper name = new ReadOnlyStringWrapper(this, "name");
```

where I have not included the column for exchange rates. Since the information can not be changed, both properties are defined readonly. The table *world* uses the following model class:

```
public class World
{
 private ReadOnlyStringWrapper code = new ReadOnlyStringWrapper(this, "code");
 private StringProperty name = new SimpleStringProperty();
```

In the program, it should not be possible to change the name of a continent, but when the property *name* is defined read/write, it is because the class *Country* is defined as a derivative class:

```
public class Country extends World
{
 private IntegerProperty area = new SimpleIntegerProperty();
 private IntegerProperty inhabitants = new SimpleIntegerProperty();
 private StringProperty world = new SimpleStringProperty();
 private StringProperty currency = new SimpleStringProperty();
```

The reason is primarily that a *TreeView* basically contains elements of the same type, as in this case is *World*, and the tree can therefore also immediately contain *Country* objects.

With these model classes in place, I have the following data model:

```
public class Model
{
 private ObservableList<World> worlds =
 FXCollections.observableArrayList();
 private ObservableList<Currency> currencies =
 FXCollections.observableArrayList();
 private ObservableList<Country> countries =
 FXCollections.observableArrayList();

 public Model()
 {
  load();
 }
 public ObservableList<World> getWorlds()
 {
  return worlds;
 }

 public ObservableList<Currency> getCurrencies()
 {
  return currencies;
 }

 public ObservableList<Country> getCountries()
 {
  return countries;
 }

 public World getWorld(String code)
 {
  for (World world : worlds) if (world.getCode().equals(code)) return world;
  return null;
 }

 public Currency getCurrency(String code)
 {
  for (Currency currency : currencies) if (currency.getCode().equals(code))
   return currency;
  return null;
 }

 private void load()
 {
  …
 }

 public void save()
 {
  …
}
```

The constructor calls the method *load()*, which initializes the three lists by reading the content of the database. I have not displayed the code for *load()*, but it only contains simple database operations. It should also be possible to write the changes that the program returns to the database, and therefore the method *save()*. When examining the code, note that it writes all countries back with a batch update. It might be a little exaggerated, and it could easily be solved by expanding the class *Country* with an additional *Boolean* property that could be set to true when a value was changed. For the class *Model*, you should also note the methods *getWorld()* and *getCurrency()* that from the continent's code and the currency code, respectively, returns the corresponding object. It is used when editing data for a country.

The class *TreeModel* is data model for the *TreeView* component and is not much more than a thin layer between the *Model* class and the user interface, but its task is to arrange the data from the model layer in a hierarchy of *TreeItem* objects:

```
public class TreeModel
{
 private Model model = new Model();
 private TreeItem<World> root = new TreeItem(new World("", "This world"));

 public TreeModel()
 {
  build();
 }

 public Model getModel()
 {
  return model;
 }
```

American online
LIGS University
is currently enrolling in the
Interactive Online BBA, MBA, MSc,
DBA and PhD programs:

- enroll by September 30th, 2014 and
- save up to 16% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

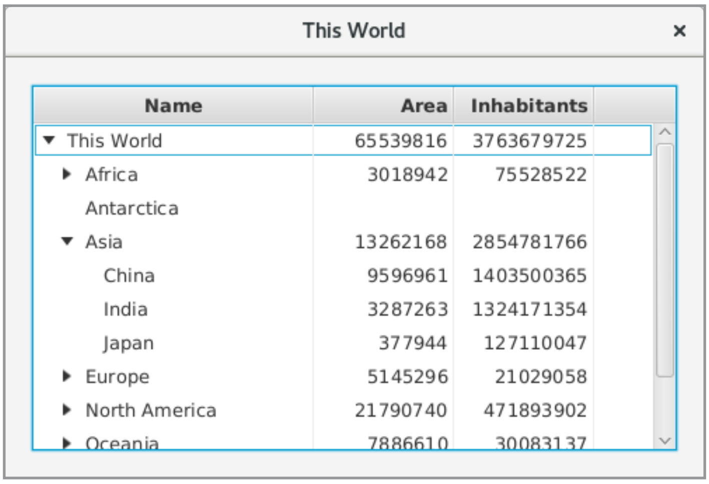
Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

```
 public TreeItem getData()
 {
  return root;
 }

 public void save()
 {
  model.save();
 }

 private void build()
 {
  List<TreeItem<World>> worlds = new ArrayList();
  for (World world : model.getWorlds()) worlds.add(new TreeItem(world));
  for (Country country : model.getCountries())
  {
   for (TreeItem<World> item : worlds)
    if (item.getValue().getCode().equals(country.getWorld()))
    {
     item.getChildren().add(new TreeItem(country));
     break;
    }
  }
  root.getChildren().addAll(worlds);
 }
}
```

With these 5 classes in place, the program's classes can be written for the main window and the dialog box. Since none of these classes in principle contains something new, I do not want to show the code here, but you are encouraged to study the code, and the dialog is not quite simple and, among other, it complicates the task changing the continent for a country, as it means that the country has to be moved in the tree. To resolve this, delete the old node and create a new one somewhere else, and note that the visual representation of the tree is automatically updated.

## 3.6 A TREETABLEVIEW

JavaFX also has a *TreeTableView* control, which can be characterized as a combination of a *TableView* and *TreeView*. This corresponds to that the component as a *TreeView* shows a hierarchy of objects, but where each object appears as a row of multiple values and thus more columns. It sounds complicated and is it, too, but the component works in principle like the previous ones, though there are more sophistry for the programming. I will start with a simple example that will show the basic syntax and I will again use data about countries organized in a hierarchy where a country's parent is the continent to which the country belongs (and it is assumed – not entirely correct – that countries belong to one particular continent). If you run the program, you will see the window below, where *This World* and *Asia* are expanded.



| Name | Area | Inhabitants |
|---|---|---|
| ▼ This World | 65539816 | 3763679725 |
| ▶ Africa | 3018942 | 75528522 |
| Antarctica | | |
| ▼ Asia | 13262168 | 2854781766 |
| China | 9596961 | 1403500365 |
| India | 3287263 | 1324171354 |
| Japan | 377944 | 127110047 |
| ▶ Europe | 5145296 | 21029058 |
| ▶ North America | 21790740 | 471893902 |
| ▶ Oceania | 7886610 | 30083137 |

That is, for each country, name, area and number of inhabitants are displayed. Here you should note that a number for area and inhabitants is shown, if the row is a continent, which is the sum of the corresponding numbers for child nodes.

The program is called *ShowCountriesProgram* and uses two model classes called *World* and *Country*. These classes are identical to the corresponding classes in the program *UpdateWorldProgram* and should not be mentioned further. The class *CountriesModel* defines a model for the *TreeTableView* component:

```
public class CountriesModel
{
 private TreeItem<Country> root =
  new TreeItem(new Country("", "This World", null, null, null, null));

 public CountriesModel()
 {
  build();
 }

 public TreeItem<Country> getData()
 {
  return root;
 }

 public TreeTableColumn<Country, String> getNameCol()
 {
```

```
 TreeTableColumn<Country, String> col = new TreeTableColumn<>("Name");
 col.setCellValueFactory(new TreeItemPropertyValueFactory("name"));
 col.setPrefWidth(200);
 return col;
}

public TreeTableColumn<Country, Long> getAreaCol()
{
 TreeTableColumn<Country, Long> col = new TreeTableColumn<>("Area");
 col.setCellValueFactory(new SumValueFactory("area"));
 col.setCellFactory(new LongFactory());
 col.setStyle("-fx-alignment: CENTER_RIGHT;");
 col.getStyleClass().add("salary-header");
 col.setPrefWidth(100);
 return col;
}

public TreeTableColumn<Country, Long> getInhabitantsCol()
{
 TreeTableColumn<Country, Long> col = new TreeTableColumn<>("Inhabitants");
 col.setCellValueFactory(new SumValueFactory("inhabitants"));
 col.setCellFactory(new LongFactory());
 col.setStyle("-fx-alignment: CENTER_RIGHT;");
```

DON'T EAT YELLOW SNOW
What will your advice be?
GrantThornton
An instinct for growth™

**Some advice just states the obvious.** But to give the kind of advice that's going to make a real difference to your clients you've got to listen critically, dig beneath the surface, challenge assumptions and be credible and confident enough to make suggestions right from day one. At Grant Thornton you've got to be ready to kick start a career right at the heart of business.

Sound like you? Here's our advice: visit
**GrantThornton.ca/careers/students**

Scan here to learn more about a career with Grant Thornton.

© Grant Thornton LLP. A Canadian Member of Grant Thornton International Ltd

```
 col.getStyleClass().add("salary-header");
 col.setPrefWidth(100);
 return col;
}

private void build()
{
 TreeItem<Country> af =
  new TreeItem(new Country("AF", "Africa", null, null, null, null));
 af.getChildren().add(new TreeItem(new
 Country("ZAF", "South Africa", 1221037L,
   54956900L, "AF", "ZAR")));
 …
 root.getChildren().addAll(af, an, as, eu, na, oc, sa);
}

class LongFactory implements
 Callback<TreeTableColumn<Country, Long>, TreeTableCell<Country, Long>>
{
 @Override
 public TreeTableCell<Country, Long> call(TreeTableColumn<Country, Long> col)
 {
  return new TreeTableCell<Country, Long>()
  {
   @Override
   public void updateItem(Long item, boolean empty)
   {
    super.updateItem(item, empty);
    this.setText(null);
    this.setGraphic(null);
    if (!empty && item != 0) this.setText("" + item);
   }
  };
 }
}

class SumValueFactory implements
 Callback<TreeTableColumn.CellDataFeatures<Country,
 Long>, ObservableValue<Long>>
{
 private String field;
 private long sum = 0;

 public SumValueFactory(String field)
 {
  this.field = field;
 }
```

```
  @Override
  public ObservableValue<Long>
   call(TreeTableColumn.CellDataFeatures<Country, Long> cellData)
  {
   TreeItem<Country> item = cellData.getValue();
   if (item.getValue().getWorld() == null ||
    item.getValue().getWorld().length() == 0)
   {
    sum = 0;
    calculate(item);
    return new SimpleLongProperty(sum).asObject();
   }
   return new TreeItemPropertyValueFactory<Cou
   ntry, Long>(field).call(cellData);
  }

  private void calculate(TreeItem<Country> item)
  {
   sum += getValue(item.getValue());
   for (TreeItem<Country> child : item.getChildren())
    if (child.getValue().getCode().length() == 3)
     sum += getValue(child.getValue());
    else calculate(child);
  }

  private long getValue(Country country)
  {
   if (field.equals("area"))
    return country.getArea() == null ? 0 : country.getArea();
   return country.getInhabitants() == null ? 0 : country.getInhabitants();
  }
 }
}
```

The class immediately looks like model classes for the other components (*TableView* and *TreeView*) and consists primarily of methods that return column objects. The constructor calls a method *load()*, where I have only shown few statements, but the method organizes data for 20 countries in a hierarchy under continents. Aside from the fact that the code fills a part, there is nothing mysterious in the code. You should note that a country consists of more data than the program shows, and the class *CountryModel* actually has a column method for each data column. Above, I have only included the methods for the three columns used by the program. There is no particular reason not to show all columns besides pointing out that of course it is not necessary.

Similar to the columns for the two previous controls, a *CellValueFactory* object is associated with which a parameter specifies where to retrieve the individual values. The name of the property is specified by the specific objects to be displayed, and here it are *Country* objects. A *CellValueFactory* thus indicates which values should be displayed in the columns cells. For the *name* column, a *TreeItemPropertyValueFactory* uses the content of the individual cells as a *Label* containing the value. The type *TreeItemPropertyValueFactory* implements a *Callback* interface, and you can therefore type your own *CellValueFactory* as a class that implements this interface if you want to determine the value of the columns cells. I have done that for the next two columns in the form of the class *SumValueFactory*. The class has a parameter *field*, which is the name of the property in the class *Country* to which the column relates. The goal is that the same class should apply for both columns *Area* and *Inhabitants*. Otherwise, the class consists primarily of overriding the method *call()*, which is defined by the interrface *Callback*. The method has parameters that represent the current cell in the model, and tests whether the cell contains a *Country* that does not have a reference to a continent. If that is the case, the object itself represents a continent and the method calls the method *calculate()* which determines the sum of all child object values for the current property. You should note that the method *calculate()* is recursive, and that is why the root of the tree shows the sum of all the world's countries. If the current cell is not for a continent, the value is determined by a default *TreeItemPropertyValueFactory*.

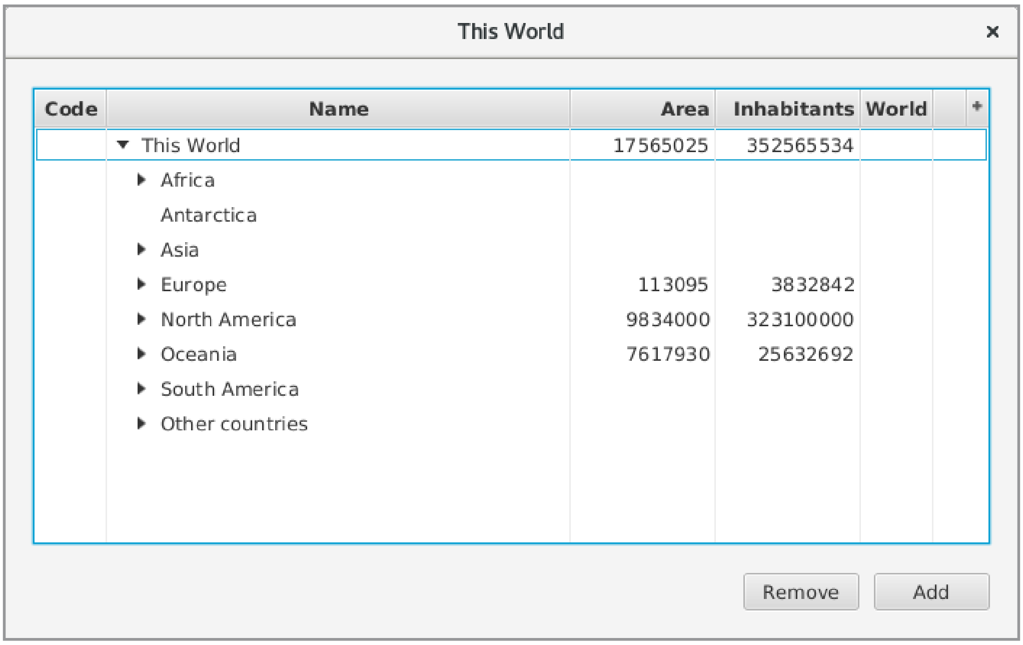
Alcatel·Lucent
www.alcatel-lucent.com/careers
What if you could build your future and create the future?
One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

A column may also be associated with a *CellFactory*, and while a *CellValueFactory* determines the value, a *CellFactory* determines which object will display the value. The meaning of a *CellFactory* is to associate an object that is used to edit the content of the cell (something to be treated in the next example), but in this case, a *CellFactory* is used simply to display 0 values as blank (what could also be achieved with a converter). The content of a cell is a *TreeTableCell* object, and the class *LongFactory* is a *CellFactory* (implementing a *Callback* interface) that implements the method *call()*, so it returns a *TreeTableCell* where the method *updateItem()* is overridden.

The above code can be difficult to understand, but if you think on, that a *TreeTableView*, like a *TableView* and an *TreeView*, is based on a data model and where the model (for a *TreeTableView*) should organize data in a hierarchy and define the individual columns then it's not that bad again and happily happening the same way every time. Have you first written the model, then the rest goes by itself, and for this example is the code of the method *start()*:

```
public void start(Stage primaryStage)
{
 TreeTableView<Country> view = new TreeTableView<>(model.getData());
 view.getColumns().addAll(model.getNameCol(), model.getAreaCol(),
  model.getInhabitantsCol());
 BorderPane root = new BorderPane(view);
 root.setPadding(new Insets(20, 20, 20, 20));
 Scene scene = new Scene(root, 500, 300);
 scene.getStylesheets().add("resources/css/styles.css");
 primaryStage.setTitle("This World");
 primaryStage.setScene(scene);
 primaryStage.show();
}
```

## 3.7 A TREETABLEVIEW, AN EXTENDED EXAMPLE



| Code | Name | Area | Inhabitants | World |
|---|---|---|---|---|
| | ▼ This World | 17565025 | 352565534 | |
| | ▶ Africa | | | |
| | Antarctica | | | |
| | ▶ Asia | | | |
| | ▶ Europe | 113095 | 3832842 | |
| | ▶ North America | 9834000 | 323100000 | |
| | ▶ Oceania | 7617930 | 25632692 | |
| | ▶ South America | | | |
| | ▶ Other countries | | | |

To conclude this chapter, I will show an example similar to the above, but showing countries created in the database *padata*, where you can also edit the countries, create new and delete a country, but such that a country is edited inline, as shown for a *TableView* in the example *EditPersonProgram*. The current example is called *UpdateCountriesProgram* and opens the above window, where the row with the name *Other countries* contains the countries from the database that are not located under a continent and where the column *Code* displays the country code with three characters. In fact, the model actually has two more columns, showing the country code as two characters, and partly showing the currency code. In order to simplify the program a bit, I have not loaded the table of currencies from the database and the program therefore does not validate the codes that are entered. If you enter an none existing currency code, that country can not be updated due to the foreign key in the database. Therefore, the column is hidden by default.

In fact, it's not easy to edit the content of the individual cells in a *TreeTableView*, and the result easily leads to many lines of code, as is the case in the current example – and it does not even work anywhere. I do not want to show the code here, but just highlight the key things that you should be aware of when studying the code.

The class *World* is unchanged compared to the previous examples. The class *Country*, on the other hand, has changed a bit, consisting of adding a property *code3* to the country code of three characters, and then the type for the two properties *area* and *inhabitants* are changed to *Long*. The reason is that the world's population is so large that it can not be represented by an *int*.

The class *Model* is similar to the corresponding class in the program *UpdateWorldProgram*, but with the difference that, as mentioned, that this time it did not include the database table *currency*. The class has as previously a method *load()*, which loads the two tables *world* and *country* and creates two *ObservableList* objects for these data. This time, a *ListChangeListener* has been added to the list of countries that observe events regarding changes to list contents:

```
public class Model
{
 private Callback<Country, Observable[]> cb = (Country c) ->
  new Observable[] { c.nameProperty(),
  c.areaProperty(), c.inhabitantsProperty(),c.
  worldProperty(), c.currencyProperty() };
 private ObservableList<World> worlds = FXCollections.observableArrayList();
 private ObservableList<Country> countries =
  FXCollections.observableArrayList(cb);

 public Model()
 {
  load();
  countries.addListener(new CountryChangeListener());
 }
```

Maastricht University
Leading in Learning!

Join the best at the Maastricht University School of Business and Economics!

Top master's programmes
- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

*Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012*

Maastricht University is the best specialist university in the Netherlands (Elsevier)

Visit us and find out why we are the best!
Master's Open Day: 22 February 2014

www.mastersopenday.nl

This event handler is used to update the database, and thus, as in the program *UpdateWorldProgram*, a batch update is not used, which uncritically writes all countries back to the database. In the current application, physical data is written to the database each time a new country is added, each time a country is deleted and each time a cell is changed. If there are many changes, it is not necessarily a good solution as it can lead to many write-ups to the database.

Then there is the class *CountriesModel*, which is the most complex and then also the largest of the program's classes. Basically, the class looks like the corresponding class in *ShowCountriesProgram* and creates the data model for the *TreeTableView* component and hence methods that creates the individual columns. The first two do not add anything new, but the method *getNameCol()* has this time a *CellFactory*, as you can edit the contents of a cell. It is basically a *TextFieldTreeTableCell*, and the effect is to open a *TextField* when you double-click the cell. In this case, however, there is an additional challenge as not all names are editable and therefore I have defined my own *CellFactory*:

```
class StringFactory implements Callback<TreeTableColumn<Country, String>,
 TreeTableCell<Country, String>>
{
 private Callback<TreeTableColumn<Country, String>,
  TreeTableCell<Country, String>> cellFactory =
   TextFieldTreeTableCell.<Country>forTreeTableColumn();

 public TreeTableCell<Country, String>
 call(TreeTableColumn<Country, String> col)
 {
  TreeTableCell<Country, String> cell = cellFactory.call(col);
  cell.itemProperty().addListener((obs, oldValue, newValue) ->
  {
   TreeTableRow<Country> row = cell.getTreeTableRow();
   if (row == null) cell.setEditable(false);
   else
   {
    Country item = cell.getTreeTableRow().getItem();
    if (item != null)
    {
     if (item.getCode3() == null || item.getCode3().length() < 3)
      cell.setEditable(false);
     else cell.setEditable(true);
    }
   }
  });
  return cell ;
 }
}
```

The syntax does not look nice, but here you have to remember that a *CellFactory* is just a class that implements a *Callback* interface and hence the method *call()*. The class creates a *TextFieldTreeTableCell* used to edit the content. The method *call()* is performed every time a cell becomes visual and it has a handler that primarily checks whether the cell represents an object for a country and not a continent and sets the cell's editable property accordingly.

The next method *getAreaCol()* works almost the same as *getNameCol()*, but another *TextFieldTreeTableCell* is used, as this time you have to edit a *Long*. The type is *LongFactory* and is in principle identical to *StringFactory*. Finally, the method also uses a *CellValueFactory* (because of the sum of all child nodes areas), but it is the same class as in the previous program. The *getInhabitantsCol()* method is in principle identical.

The method *getCurrencyCol()* works in the same way as *getNameCol()*, but the method *getWorldCol()* is a bit different. When you click on a cell, you must have a combobox so you can choose a continent. It has the type *ComboBoxTreeTableCell* and must as parameter have the items to display. The object *CellFactory* now has the type *WorldFactory*, and except that a *ComboBoxTreeTableCell* is used instead of a *TextFieldTreeTableCell*, the class is basically identical to the other two *CellFactory* classes. However, the *getWorldCol()* method requires extra action, as a change in a country's continent means that the corresponding node must be moved in the tree (see, if applicable, the program *UpdateWorldProgram*). This is solved by associating an event handler for *editSubmit*, which takes care of what is needed.

Back there are two classes, though both are relatively simple and at least do not add anything new. The class *CountryDialog* is a dialog box and is used when creating a new country. The reason is that a country with a name and legal country codes must be created. Back is the main program, which does not contain anything new, but you should note that in order to create a country (and thus click on the *Add* button), the continent for which the country is to be created must be selected. Similarly, to delete a country you must select the country you want to delete. Note that you do not get any warning about what should be used in practice.

# 4 DRAG AND DROP

JavaFX supports direct drag and drop, such the functionality is built into both a *Scene* and a *Node* object. You are talking about the operation as a *press-drag-release gesture*, which means that the user holds one of the mouse's buttons down, drags the mouse and releases the mouse again to complete the operation. A gesture can be started of the *Scene* object or on any *Node* object, called the source object, and a gesture can include several objects (*Nodes*). During the operation, several events are firing and the use of these events depends on the purpose of the current gesture, which may be:

1. That you want to change a node's shape by dragging its perimeter or by dragging it to another location. In this gesture, only the node that initiates the operation is involved.
2. That you want to draw a source and drop it to another node (target) and in one way or another combine the two nodes. When the source node is dropped on the target node, some action is performed.
3. To drag a node and drop it over another node to transfer data from the source node to the target node, and the actual data transfer occurs when the source node is dropped.

Business
Strategic Marketing
Management
BI
International
Business
Financial
Economics
Leadership &
Organisational
Psychology
Shipping
Management
BI NORWEGIAN BUSINESS SCHOOL
EFMD
EQUIS
ACCREDITED

Empowering People.
Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

To describe these gestures, the documentation divides them into three types:

1. simple press-drag-release gesture
2. full press-drag-release gesture
3. drag-and-drop gesture

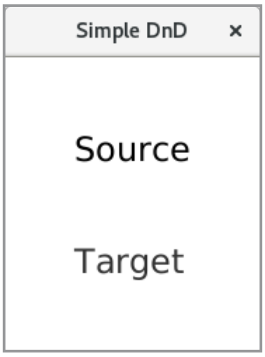
and here is the last one of the most interesting (which is most to be aware of), and I will divide the chapter into sections corresponding to this division.

## 4.1 SIMPLE PRESS-DRAG-RELEASE GESTURE

As the name says, it is the simplest form for drag-and-drop and is used where the operation relates only to a single node and which is the node that starts the gesture. During the operation, all *MouseDragEvent* types are fired:

- MouseDragEntered
- MouseDragOver
- MouseDragExited
- MouseDragReleased

but they are only sent to the source node. As an example, below is shown an application that opens a window with two *Label* controls. By pointing at the top and holding down the mouse, the background of the window changes to light green, and moves the mouse, the text changes the color to yellow and the background turns light blue. If you pull the mouse around in the window, nothing happens – even if you drag the mouse over the bottom *Label*. However, if you release the mouse, the background color and the text color of the top Label are returned to white and black respectively. The example should show what happens if you start a drag gesture for a *Label* (and thus any other node) and drag the mouse around in the window.



```
package simplednd;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;
import javafx.geometry.*;
import javafx.scene.text.Font;
import javafx.scene.paint.*;
import javafx.beans.property.*;

public class SimpleDnD extends Application
{
 private Label lbl1 = new Label("Source");
 private Label lbl2 = new Label("Target");
 private final ObjectProperty<Color> fg =
  new SimpleObjectProperty<Color>(Color.BLACK);
 private final ObjectProperty<Background> bg =
  new SimpleObjectProperty<Background>(background(Color.WHITE));

 @Override
 public void start(Stage stage)
 {
  Scene scene = new Scene(getRoot());
  lbl1.setOnMousePressed(e -> bg.set(background(Color.LIGHTGREEN)));
  lbl1.setOnMouseDragged(e -> bg.set(background(Color.LIGHTBLUE)));
  lbl1.setOnDragDetected(e -> fg.set(Color.YELLOW));
  lbl1.setOnMouseReleased(
   e -> { bg.set(background(Color.WHITE)); fg.set(Color.BLACK); });
  lbl2.setOnMouseDragEntered(e -> bg.set(background(Color.DARKGREEN)));
  lbl2.setOnMouseDragOver(e -> bg.set(background(Color.DARKBLUE)));
  lbl2.setOnMouseDragReleased(e -> fg.set(Color.RED));
  lbl2.setOnMouseDragExited(e -> bg.set(background(Color.MAGENTA)));
  stage.setScene(scene);
  stage.setTitle("Simple DnD");
  stage.show();
 }

 private Pane getRoot()
 {
  lbl1.setFont(Font.font(24));
  lbl2.setFont(Font.font(24));
  lbl1.textFillProperty().bind(fg);
  VBox pane = new VBox(50, lbl1, lbl2);
  pane.setPadding(new Insets(50, 50, 50, 50));
  pane.backgroundProperty().bind(bg);
  return pane;
 }
 private static Background background(Color color)
 {
  return new
   Background(new BackgroundFill(color, CornerRadii.EMPTY, Insets.EMPTY));
 }

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

The program has two *Label* controls defined as instance variables, as well as two properties for a *Color* and a *Background* respectively. Here you should especially note that a *Background* object is created for a specific color of the static method *background()*. The method *getRoot()* intializes the components and creates the program's view. Here you should note that root is a *VBox*, and that the method binds its background to the property *bg*, and similarly, the text color of the top *Label* binds to the property *fg*. Finally, the method *start()*, and here is the most important thing that 4 event handlers are associated with each of the *Label* controls. To the top, two drag event handlers are assigned, while all 4 drag event handlers

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now

Go to www.helpmyassignment.co.uk for more info

Helpmyassignment

are assigned to the bottom. The meaning is that you must observe that these event handlers are never performed. A simple press-drag-release gesture relates only to the node – and sends only notifications regarding drag to that Node – that initiates the operation, which is the Node the mouse points when the button is pressed. The operation ends when the mouse is released and wherever where the mouse points.

## 4.2 FULL PRESS-DRAG-RELEASE GESTURE

When the source node receives a notification for a *MouseDragDetected* event, it can launch a full press-drag-release gesture by calling the method *startFullDrag()*. In addition, the property *mouseTransparent* must be set to *false* for the source node so this node does not receive all drag notifications. The example *FullpressDnD* is almost identical to the above program and opens the same window. Clicking on the top *Label* starts a drag operation, which immediately results in the same as above, but when the mouse is draged over the bottom *Label*, the background of the window changes to dark blue while the text in the bottom label becomes white. If the mouse again is draged out from the bottom label, the text color in the top label changes to violet. When the mouse is released, it returns to default. The code is this time the following:

```
public class FullpressDnD extends Application
{
 private Label lbl1 = new Label("Source");
 private Label lbl2 = new Label("Target");
 private final ObjectProperty<Color> fg1 =
  new SimpleObjectProperty<Color>(Color.BLACK);
 private final ObjectProperty<Color> fg2 =
  new SimpleObjectProperty<Color>(Color.BLACK);
 private final ObjectProperty<Background> bg =
  new SimpleObjectProperty<Background>(background(Color.WHITE));

 @Override
 public void start(Stage stage)
 {
  Scene scene = new Scene(getRoot());
  lbl1.setOnMousePressed(e -> startDrag());
  lbl1.setOnMouseDragged(e -> bg.set(background(Color.LIGHTBLUE)));
  lbl1.setOnDragDetected(e -> { lbl1.startFullDrag();
  fg1.set(Color.YELLOW); });
  lbl1.setOnMouseReleased(e -> endDrag());
  lbl2.setOnMouseDragEntered(e -> fg2.set(Color.WHITE));
  lbl2.setOnMouseDragOver(e -> bg.set(background(Color.DARKBLUE)));
  lbl2.setOnMouseDragReleased(e -> lbl1.setText("End"));
  lbl2.setOnMouseDragExited(
   e -> { if (lbl1.isMouseTransparent()) fg1.set(Color.MAGENTA); });
  stage.setScene(scene);
  stage.setTitle("Full Press DnD");
  stage.show();
 }

 private void startDrag()
 {
  lbl1.setText("Source");
  lbl1.setMouseTransparent(true);
  bg.set(background(Color.LIGHTGREEN));
 }

 private void endDrag()
 {
  lbl1.setMouseTransparent(false);
  bg.set(background(Color.WHITE));
  fg1.set(Color.BLACK);
  fg2.set(Color.BLACK);
 }

 private Pane getRoot()
 {
  lbl1.setFont(Font.font(24));
  lbl2.setFont(Font.font(24));
  lbl1.textFillProperty().bind(fg1);
  lbl2.textFillProperty().bind(fg2);
  VBox pane = new VBox(50, lbl1, lbl2);
  pane.setPadding(new Insets(50, 50, 50, 50));
  pane.backgroundProperty().bind(bg);
  return pane;
 }

 private static Background background(Color color)
 {
  return new
   Background(new BackgroundFill(color, CornerRadii.EMPTY, Insets.EMPTY));
 }

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

This time, two properties for *Color* objects are defined so that the text color for both *Label* controls is bound to a property (the method *getRoot()*). Otherwise, you should first notice which events handlers are associated with the two labels. When the mouse points to the top label and the mouse is pressed, *mouseTransparent* is set to *true* (the method *startDrag()*) and it is put back to *false* when the mouse is released (the method *endDrag()*). Finally, note that *lbl1* at *dragDetected* executes *startFullDrag()*, and the result is that the component *lbl2* this time receives drag notifications. You are encouraged to experiment with the program and observe when each event occurs. Here you should especially note *dragReleased* for the lower label, which changes the text in the top label. It is executed if you release the mouse while it points to the bottom label.

## 4.3 DRAG-AND-DROP GESTURE

The last form of drag and drop gesture is mostly the general and probably also the most used and used to extract data from the source node and drop them on a target node. The gesture in question may concern nodes within the same application, but it may also concern nodes in two different Java applications. In fact, both applications does not necessarily have to be a Java application.

Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.

Therefore we need the best employees who can meet this challenge!

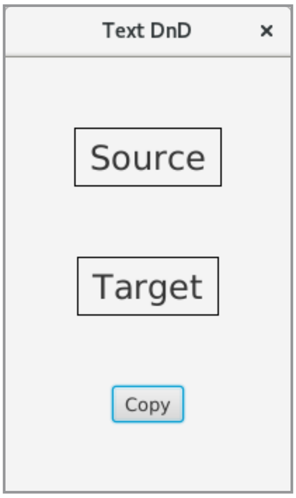
The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

In general, a drag-and-drop operation includes the following actions:

- Point to a node and hold down one of the mouse's buttons
- Drag the mouse while the button is held down and the node gets a *DragDetected* event and must perform a *startDragAndDrop()*, after which the node is a gesture source node and the current data is placed on the clipboard
- After starting a drag-and-drop gesture, the system no longer sends *MouseEvents*, but instead *DragEvents*
- If a gesture source is dragged over a gesture target, it will check if it will accept data on the clipboard and do it, it will with a *DragEvent* indicate that data is accepted.
- If the user drops the mouse while pointing to a gesture target, it will apply the current data and send a *DragDropped* event
- When the source gesture node receives a *DragDone* event, it tells that the operation has been completed

In this section I will illustrate drag-and-drop with three examples, and the first one is called *TextDnd* and opens the following window:



It is basically the same window as in the two previous examples, except that there is a frame outside of the two Label controls, and a button has been added. In this example, drag and drop gestures drag the text from the top label and drop it on the bottom label. A drag-and-drop gesture can be performed in two ways (actually three and you can read about drag-and-drop in the book Java 10), where one drops a copy of the data element (here the text in the top label) while the another one moves it. Here the text *Copy* in the button means that it is a copy gesture that is being executed. Clicking the button changes the text and it is a move gesture, while resetting the window. The code is as follows:

```
public class TextDnD extends Application
{
 private Button cmd = new Button("Copy");
 private Label lbl1 = new Label("Source");
 private Label lbl2 = new Label("Target");
 private TransferMode mode = TransferMode.COPY;

 @Override
 public void start(Stage stage)
 {
  Scene scene = new Scene(getRoot());
  lbl1.setOnDragDetected(this::dragDetected);
  lbl2.setOnDragOver(this::dragOver);
  lbl2.setOnDragDropped(this::dragDropped);
  lbl1.setOnDragDone(this::dragDone);
  stage.setScene(scene);
  stage.setTitle("Text DnD");
  stage.show();
 }

 private Pane getRoot()
 {
  lbl1.setFont(Font.font(24));
  lbl2.setFont(Font.font(24));
  lbl1.setPadding(new Insets(5, 10, 5, 10));
  lbl2.setPadding(new Insets(5, 10, 5, 10));
  lbl1.setStyle("-fx-border-color: black;");
  lbl2.setStyle("-fx-border-color: black;");
  cmd.setOnAction(e -> reset());
  VBox pane = new VBox(50, lbl1, lbl2, cmd);
  pane.setPadding(new Insets(50, 50, 50, 50));
  pane.setAlignment(Pos.CENTER);
  return pane;
 }

 private void dragDetected(MouseEvent e)
 {
  String text = lbl1.getText();
  if (text == null)
  {
   e.consume();
   return;
  }
  Dragboard dragboard = lbl1.startDragAndDrop(mode);
  ClipboardContent content = new ClipboardContent();
  content.putString(text);
  dragboard.setContent(content);
  e.consume();
 }
```

```
private void dragOver(DragEvent e)
{
 Dragboard dragboard = e.getDragboard();
 if (dragboard.hasString()) e.acceptTransferModes(mode);
 e.consume();
}

private void dragDropped(DragEvent e)
{
 Dragboard dragboard = e.getDragboard();
 if (dragboard.hasString())
 {
  String text = dragboard.getString();
  lbl2.setText(text);
  e.setDropCompleted(true);
 }
 else e.setDropCompleted(false);
 e.consume();
}

private void dragDone(DragEvent e)
{
 TransferMode mode = e.getTransferMode();
```

TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

```
  if (mode == TransferMode.MOVE) lbl1.setText("");
  e.consume();
 }

 private void reset()
 {
  lbl1.setText("Source");
  lbl2.setText("Target");
  if (mode == TransferMode.COPY)
  {
   cmd.setText("Move");
   mode = TransferMode.MOVE;
  }
  else
  {
   cmd.setText("Copy");
   mode = TransferMode.COPY;
  }
 }

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

With regard to instance variables, they are self explanatory, but note the last one used to indicate whether a gesture should be *Copy* or *Move*. The type is an enumeration. The method *getRoot()* that creates the program's view does not contain anything new, but in the method *start()* you should note which events handlers are being associated. Generally, a source node must have associated two event handlers: *DragDetected* and *DragDone*. A target node must correspondingly have associated two event handlers: *DragOver* and *DragDropped*. Corresponding to this is the most important in the example these 4 event traders.

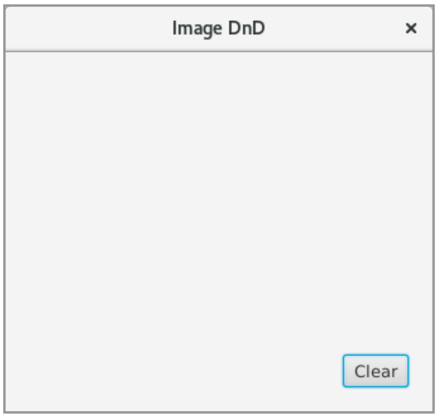
The method *dragDetected()* starts by testing whether there is actually a data element and here if the top label shows a text. If so, *startDragAndDrop()* is performed on the source node, and the parameter is which operation to be performed. The method returns a *DragBoard* object, and is an object that represents the clipboard. In fact, *DragBoard* is a subclass of the class *Clipboard*. The method *dragDetected()* also has a reference to the clipboard and it saves the text on the clipboard and finally updates the *DragBoard* object with the current content on the clipboard.

The method *dragDone()* is not always necessary, but in this case it tests whether it is a *Move* operation, and if so, the method is responsible for deleting the text in the *Label* component –

that is, in source node. The method *dragOver()* is performed when the user passes the mouse over a target node. The method tests whether something is on the clipboard, and if that is the case, *acceptTransferModes()* is performed, which means that the cursor changes, so you can visually see that you are over a node where you can drop.

Finally, there is the method *dragDropped()*. In this case, it starts testing whether there is a string on the clipboard, and if so, the text is retrieved and the label component's text is updated. Then the method *dropCompleted()* is performed, which sends a notification to the source node that the operation has been completed. Before leaving the program, you should also note the method *reset()* used as event handler for the button and among other switches mode, so the next drag-and-drop operation may be a *Move* gesture.

As the next example, I'll show you how to drag an image into an application where it can be either an image or a file that contains an image. That is, you can drag a data element that represents an image from another program (such as the *Files* program) to the current application. The example is called *ImageDnD* and opens the following window, which contains a button and an *ImageView* control:



```
public class ImageDnD extends Application
{
 private ImageView view = new ImageView();

 @Override
 public void start(Stage stage)
 {
  Scene scene = new Scene(createRoot());
  scene.setOnDragOver(this::dragOver);
  scene.setOnDragDropped(this::dragDropped);
  stage.setScene(scene);
  stage.setTitle("Image DnD");
  stage.show();
 }
```

```
private Pane createRoot()
{
 view.setFitWidth(300);
 view.setFitHeight(200);
 view.setSmooth(true);
 view.setPreserveRatio(true);
 HBox pane = new HBox(10, createButton("Clear", e -> view.setImage(null)));
 pane.setAlignment(Pos.CENTER_RIGHT);
 VBox root = new VBox(20, view, pane);
 root.setPadding(new Insets(20, 20, 20, 20));
 return root;
}

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

private void dragOver(DragEvent e)
{
 Dragboard dragboard = e.getDragboard();
```

What do you want to do?
No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!

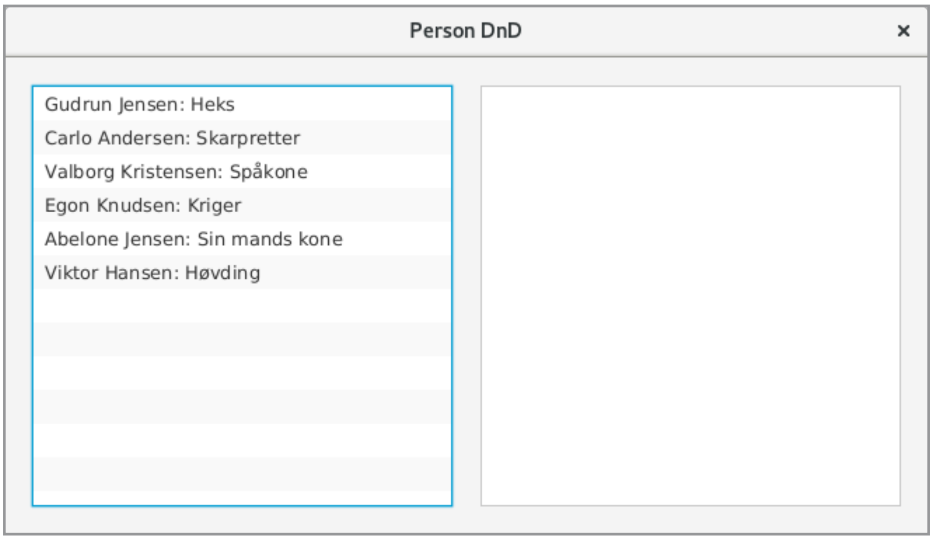
VOLVO
AB Volvo (publ)
www.volvogroup.com

Volvo Trucks I Renault Trucks I Mack Trucks I Volvo Buses I Volvo Construction Equipment I Volvo Penta I Volvo Aero I Volvo IT
Volvo Financial Services I Volvo 3P I Volvo Powertrain I Volvo Parts I Volvo Technology I Volvo Logistics I Business Area Asia

```
  if (dragboard.hasImage() || dragboard.hasFiles())
   e.acceptTransferModes(TransferMode.ANY);
  e.consume();
 }

 private void dragDropped(DragEvent e)
 {
  boolean completed = false;
  Dragboard dragboard = e.getDragboard();
  if (dragboard.hasImage()) completed = transferImage(dragboard.getImage());
  else if (dragboard.hasFiles()) completed =
   transferImageFile(dragboard.getFiles());
  else System.out.println("Error - Illegal format: Image, File, URL");
  e.setDropCompleted(completed);
  e.consume();
 }

 private boolean transferImage(Image image)
 {
  view.setImage(image);
  return true;
 }

 private boolean transferImageFile(List<File> files)
 {
  for(File file : files)
  {
   try
   {
    String mimeType = Files.probeContentType(file.toPath());
    if (mimeType != null && mimeType.startsWith("image/"))
    {
     view.setImage(new Image(file.toURI().toURL().toExternalForm()));
     return true;
    }
   }
   catch (IOException ex)
   {
    System.out.println(ex.getMessage());
   }
  }
  return false;
 }

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

The method *createRoot()* that creates the program's view does not contain anything new, and the task is to initialize the *ImageView* controller. Note, however, that the button is assigned an event handler which deletes the content of the *ImageView* component. The method *start()* assigns two event handlers to the *Scene* object, which means that you can drop an object in the application's window. There are only two handlers since the program this time does not have a source gesture node – a drag-and-drop operation is initiated in another application. *dragOver()* has the same function as in the previous example, and you should note how to test if the clipboard has a data element that the program can accept. It is likewise tested in the *dragDropped()* method and depending on what data item is (an image or a file), a method is called that performs the data transfer. Here you should especially note the method *transferImageFile()* and the syntax used. This means that the file name must be used to load that image from a file.

As the last example of drag-and-drop, I will show a program where you can drag an object of a custom type. The example is called *PersonDnD* and opens the following window:



Gudrun Jensen: Heks
Carlo Andersen: Skarpretter
Valborg Kristensen: Spåkone
Egon Knudsen: Kriger
Abelone Jensen: Sin mands kone
Viktor Hansen: Høvding

The window contains two *ListView* controls. Each line in the left *ListView* represents a *Person* object, and these objects can be draged to the right by a drag-and-drop gesture. Similarly, you can drag objects from right to left. A *Person* object is a very common model class that represents a person with two characteristics:

```
public class Person implements Serializable
{
 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;
}
```

gaiteye®
Challenge the way we run
EXPERIENCE THE POWER OF FULL ENGAGEMENT…
RUN FASTER.
RUN LONGER..
RUN EASIER…
READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM

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

 @Override
 public String toString()
 {
  return getName() + ": " + getJob();
 }
}
```

There is only one thing to observe, namely that the class is serializable. It is a prerequisite for objects to be used in a drag-and-drop gesture. Also note that this means that the types of the class's instance variables must be serializable. The class *PersonDnD* is:

```
public class PersonDnD extends Application
{
 private static final DataFormat PERSON_LIST =
  new DataFormat("persons/personlist");
 ListView<Person> view1 = new ListView();
 ListView<Person> view2 = new ListView();

 @Override
 public void start(Stage stage)
 {
  Scene scene = new Scene(getRoot());
  stage.setScene(scene);
  stage.setTitle("Person DnD");
  stage.show();
 }

 private Pane getRoot()
 {
  view1.setPrefSize(300, 300);
  view2.setPrefSize(300, 300);
  view1.getItems().addAll(getPersons());
  view1.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
  view2.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
  view1.setOnDragDetected(e -> dragDetected(e, view1));
  view2.setOnDragDetected(e -> dragDetected(e, view2));
  view1.setOnDragOver(e -> dragOver(e, view1));
  view2.setOnDragOver(e -> dragOver(e, view2));
  view1.setOnDragDropped(e -> dragDropped(e, view1));
  view2.setOnDragDropped(e -> dragDropped(e, view2));
  view1.setOnDragDone(e -> dragDone(e, view1));
  view2.setOnDragDone(e -> dragDone(e, view2));
 GridPane pane = new GridPane();
 pane.setPadding(new Insets(20, 20, 20, 20));
 pane.setHgap(20);
 pane.addRow(0, view1, view2);
 return pane;
}

private ObservableList<Person> getPersons() { … }

private void dragDetected(MouseEvent e, ListView<Person> view)
{
 int selectedCount = view.getSelectionModel().getSelectedIndices().size();
 if (selectedCount == 0)
 {
  e.consume();
  return;
 }
 Dragboard dragboard = view.startDragAndDrop(TransferMode.COPY_OR_MOVE);
 List<Person> items = getSelectedItems(view);
 ClipboardContent content = new ClipboardContent();
 content.put(PERSON_LIST, items);
 dragboard.setContent(content);
 e.consume();
}

private void dragOver(DragEvent e, ListView<Person> view)
{
 Dragboard dragboard = e.getDragboard();
 if (e.getGestureSource() != view && dragboard.hasContent(PERSON_LIST))
  e.acceptTransferModes(TransferMode.COPY_OR_MOVE);
 e.consume();
}

private void dragDropped(DragEvent e, ListView<Person> view)
{
 boolean completed = false;
 Dragboard dragboard = e.getDragboard();
 if(dragboard.hasContent(PERSON_LIST))
 {
  view.getItems().addAll((ArrayList<Person>)
  dragboard.getContent(PERSON_LIST));
  completed = true;
 }
 e.setDropCompleted(completed);
 e.consume();
}
```

```
private void dragDone(DragEvent e, ListView<Person> view)
{
 TransferMode mode = e.getTransferMode();
 if (mode == TransferMode.MOVE) removeSelectedItems(view);
 e.consume();
}

private List<Person> getSelectedItems(ListView<Person> listView)
{
 return new ArrayList(listView.getSelectionModel().getSelectedItems());
}

private void removeSelectedItems(ListView<Person> view)
{
 List<Person> list = new ArrayList();
 for(Person pers : view.getSelectionModel().getSelectedItems()) list.add(pers);
 view.getSelectionModel().clearSelection();
 view.getItems().removeAll(list);
}
```

This e-book
is made with
SetaPDF

SETASIGN

PDF components for PHP developers

www.setasign.com

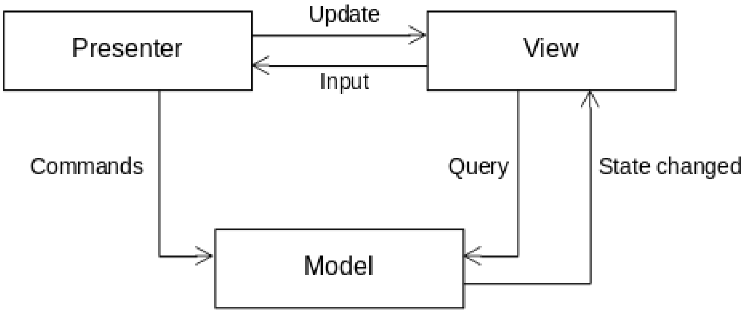
```
 public static void main(String[] args)
 {
  launch(args);
 }
}
```

Custom types can not be immediately transferred via the clipboard as they do not have a known mime type. This means that a target node can not test what is on the clipboard. Therefore, the program starts by creating a static object of the type *DataFormat* named *PERSON_LIST*. The value is not important. It just has to be a name that is unique (for the program). The method *getRoot()* fills a part, and it must, among other things, initialize the left *ListView* with 6 Person objects. These objects are created in the method *getPersons()*, but I have not shown the code. Note that for both *ListView* controls, are select *MULTIPLE*, so the user can select more *Person* objects. However, the most important thing is that 4 event handlers are assigned to each control, since both controls can be both source and target gesture nodes.

The methods (there are 4) that the event handlers call has a parameter that tells what it is for a *ListView* that the event concerns. *dragDetected()* initiates a drag operation, and you must note that its mode is *COPY_OR_MOVE* so that it can be used in either case. You start a *COPY* operation by pressing the mouse button, if you also holding down the *SHIFT* key, it is a MOVE operation. *dragDetected()* uses the method *getSelectedItems()* to return the *Person* objects that are selected, and it is this list of objects that is placed on the clipboard. *dragOver()* and *dragDropped()* appear as shown in the two previous examples but note how *dragDropped()* using the *PERSON_LIST* object tests whether it is data of the correct kind that ara located on the clipboard before the list is copied to the target *ListView* component. Finally, there is the method *dragDone()* that tests whether it is a *MOVE* operation, and if necessary, the objects that are moved will be deleted from the source node.

# 5 MVC

I have treated the *Module-View-Controller* pattern earlier, which is the design pattern of a GUI program, and in this context, I also mentioned that there are several versions of the pattern, partly because of the development tools used, but also on due to the API used. Should you use JavaFX fully, you should use the *Module-View-Presenter* pattern, which you sometimes see shortened as *MVP*. The pattern can be sketched as follows:



The principle is that the program's view takes care of everything regarding the visual, but can turn to the model to get the data to be displayed. The user interaction takes place in the view layer, but when there is a user interaction, it is sent to the presenter module, which takes care of it. That is, a window's event handlers are placed a presenter class that as a result can update the view component and execute commands on the model. When the model is updated as a result of commands from the present, it can send notifications, which the view component can listen to and possibly read the model to ensure the synchronization between model and view.

It's the principle, and in fact, is not very different than I've previously mentioned about MVC, just the pattern is drawn in a way that directly supports JavaFX. Another thing, however, is what it means in the code, and especially who is responsible for creating the individual objects, and it is best illustrated with an example.

Looking at the previous examples in this book, they basically had a 2 layer architecture consisting of a view layer and a model layer. The most important thing in the above pattern is actually a more accurate breakdown of the view layer in a view component and a presenter component. The goal of all is to make the code as simple and manageable as possible, and even as the most important thing to ensure that GUI programs are written in a standardized manner, and one of the patterns that JavaFX recommends is a systematic use of data binding. In previous books, I have used a database *addresses* that only have a single table:

```
use mysql;

create database addresses;

use addresses;

create table address (
 id int not null auto_increment,
 firstname varchar(50),
 lastname varchar(30),
 address varchar(50),
 code varchar(4),
 city varchar(30),
 mail varchar(50),
 date varchar(10),
 title varchar(50),
 primary key (id)
);
```

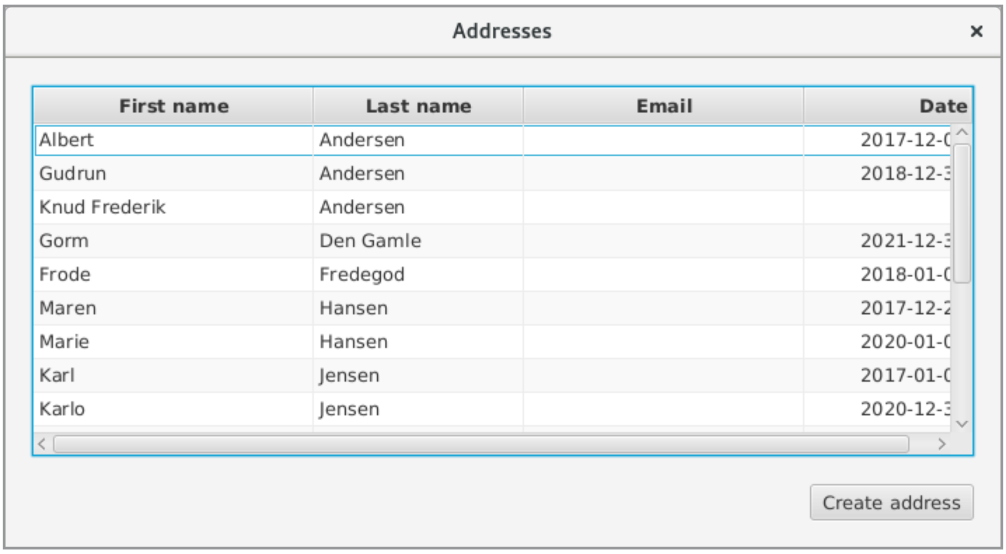
Free eBook on
Learning & Development

By the Chief Learning Officer of McKinsey

Download Now

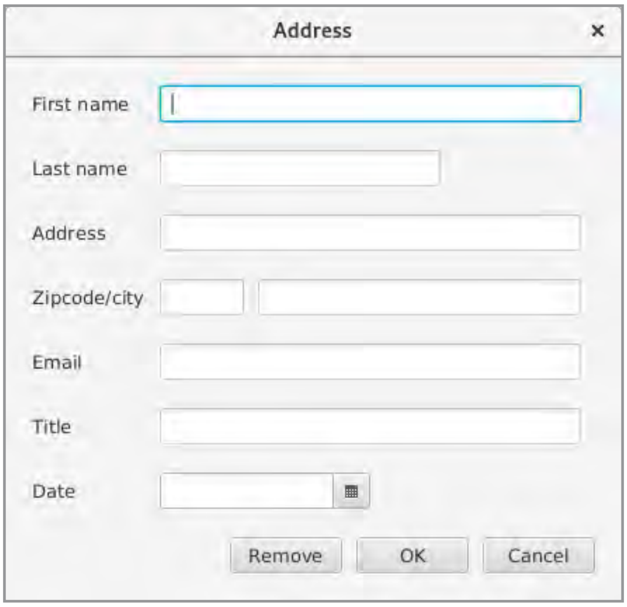
bookboon
Prof. Dr. Nick H.M. van Dam
21st Century Corporate Learning & Development
Trends and Best Practices

As example, I will look at a program that can maintain this database, and when the program starts, it must open the following window:

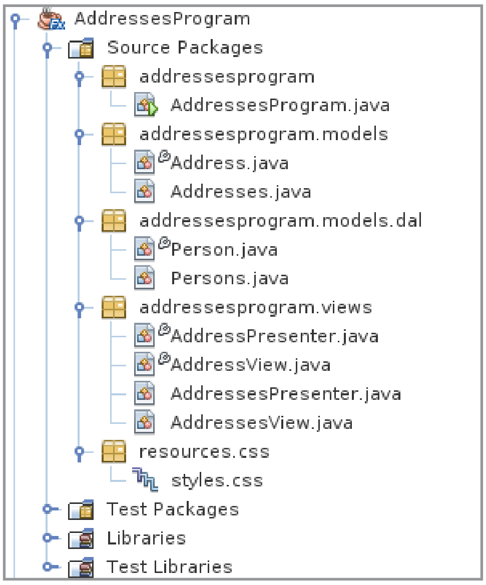

Addresses
×
Create address

| First name | Last name | Email | Date |
|---|---|---|---|
| Albert | Andersen | | 2017-12- |
| Gudrun | Andersen | | 2018-12- |
| Knud Frederik | Andersen | | |
| Gorm | Den Gamle | | 2021-12- |
| Frode | Fredegod | | 2018-01- |
| Maren | Hansen | | 2017-12- |
| Marie | Hansen | | 2020-01- |
| Karl | Jensen | | 2017-01- |
| Karlo | Jensen | | 2020-12- |

as in a *TableView* shows an overview of the database's addresses. If you click on the button, you will see the window below where you can create a new address and you get the same window (but initialized with data) if you double-click on an address in the table:


Address
×
First name
Last name
Address
Zipcode/city
Email
Title
Date
Remove
OK
Cancel

The program consists of 9 classes:


AddressesProgram
Source Packages
addressesprogram
AddressesProgram.java
addressesprogram.models
Address.java
Addresses.java
addressesprogram.models.dal
Person.java
Persons.java
addressesprogram.views
AddressPresenter.java
AddressView.java
AddressesPresenter.java
AddressesView.java
resources.css
styles.css
Test Packages
Libraries
Test Libraries

and in the following I will mention the most important of these classes and the choices made. If you start with the model layer, it has two classes, and one of the purposes is that the model via data binding should send notifications to the user interface, but also the view component classes should automatically update the model. For example, the class *Address* must have JavaFX properties corresponding to all columns in the database table, and since the classs *Addresses* should represent the table in the above window, it must have an *ObservableList<Address>* and have methods that can return the columns to the *TableView* component. The model classes *Address* and *Addresses* must thus be written according to a precise pattern that supports JavaFX. In one way or another, it does not fit with that you want to separate the model and its classes from the rest of the code so that they can be written independently of how the model has to be used. Therefore, a DAL layer is defined under the model layer, which consists of two classes *Person* and *Persons*, the first being a usual model class, representing an entity in the database, and containing no other than instance variables using conventional *get* and *set* methods. The class *Persons* consists primarily of 4 methods, one of which returns a *List<Person>* with the content of the database, where the other three are for SQL INSERT, UPDATE and DELETE operations. With this DAL layer available, the class Address is no more than a thin wrapper of the class Person, where each property is encapsulated in a JavaFX property. However, the class has a method *invalidate()*, as in relation to the current task, can validate if an *Address* object is legal.

The class Addresses has an instance of the class *Persons*, starting with reading data in the database, to create an *ObservableList<Address>*. In the same way as shown in the examples, the class defines methods for individual columns so that they can bind to the *TableView*. In addition to this, the class has methods *save()* and *remove()* used to save and delete data. In principle, these methods are trivial and do not so much other than calling the corresponding methods in the class *Persons*, but at the same time they must ensure the class's *ObservableList* is synchronized with the database.

Back there are the presentations and view components. The class *AddressesView* must define the main window:

```
import addressesprogram.models.*;

public class AddressesView extends BorderPane
{
 Button cmdAdd = new Button("Create address");
 TableView<Address> tableView;

 public AddressesView(Addresses model)
 {
  tableView = new TableView(model.getAddresses());
  createForm(model);
 }
```

www.sylvania.com

We do not reinvent the wheel we reinvent light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA

and in addition to the above, there is only *createForm()*, which initializes a *BorderPane* with a *TableView* and a *Button*. There is therefore no logic including event handlers in this class. You should note that the class inherits *BorderPane*. Also note the parameter of the constructor, which is the model and thus an object of the type *Addresses*. Finally, note the two instance variables, which are references to the window's controls. These variables have no visibility, and thus they have the package visibility so that they can be used from the presenter class, which has the task of creating the event handlers. It is therefore necessary that the view and presenter are in the same package. The presenter class is as follows:

```
public class AddressesPresenter
{
 private final Addresses model;
 private final AddressesView view;
 private final Stage owner;

 public AddressesPresenter(Stage owner, Addresses model)
 {
  this.owner = owner;
  this.model = model;
  view = new AddressesView(model);
  addHandlers();
 }

 public Pane getView()
 {
  return view;
 }

 private void addHandlers()
 {
  view.cmdAdd.setOnAction(e -> add(new Address()));
  view.tableView.setOnMousePressed(new EventHandler<MouseEvent>() {
   @Override
   public void handle(MouseEvent event)
   {
    if (event.isPrimaryButtonDown() && event.getClickCount() == 2)
      add(view.tableView.getSelectionModel().getSelectedItem());
   }
  });
 }

 private void add(Address address)
 {
  AddressPresenter presenter = new AddressPresenter(model, address);
  Scene scene = new Scene(presenter.getView());
  Stage stage = new Stage();
  stage.initOwner(owner);
  stage.initModality(Modality.APPLICATION_MODAL);
  stage.setResizable(false);
  stage.setTitle("Address");
  stage.setScene(scene);
  stage.show();
 }
}
```

It is thus the presenter class that creates the view class. It is not actually completely consistent with the pattern as most people choose to create view objects outside the class and then transfer it as a parameter to the constructor. Since I have not done it, it is necessary to add a method *getView()* such that you can refer to the view object in the main class. The class's constructor has as parameters the model and a reference to the primary *Stage* object. It is used as the owner of the dialog window to edit an address. You should note the method *addHandlers()* as the method that associates event handlers to nodes defined in the view component. In this case, an event handler must be attached to double-click a row in the *TableView* component as well as an event handler for the button. In either case, the method *add()*, which opens the dialog to either create a new address (where the current parameter is a new *Address* object) or to edit an existing address (where the address is the object of the line that is double-clicked).

The dialog box is defined as the following class, which is a GridPane:

```
public class AddressView extends GridPane
{
 private final Address model;
 TextField txtFirstname = new TextField();
 TextField txtLastname = new TextField();
 TextField txtAddress = new TextField();
 TextField txtCode = new TextField();
 TextField txtCity = new TextField();
 TextField txtMail = new TextField();
 TextField txtTitle = new TextField();
 DatePicker datePicker = new DatePicker();
 Button cmdDel = new Button("Remove");
 Button cmdOk = new Button("OK");
 Button cmdCancel = new Button("Cancel");

 public AddressView(Address model)
 {
  this.model = model;
  createForm();
  bindFields();
 }
 public Address getModel()
 {
  return model;
 }

 private void createForm()
 {
  …
 }

 public void bindFields()
 {
  txtFirstname.textProperty().bindBidirectional(model.firstnameProperty());
  txtLastname.textProperty().bindBidirectional(model.lastnameProperty());
  txtAddress.textProperty().bindBidirectional(model.addressProperty());
  txtCode.textProperty().bindBidirectional(model.codeProperty());
  txtCity.textProperty().bindBidirectional(model.cityProperty());
  txtMail.textProperty().bindBidirectional(model.mailProperty());
  txtTitle.textProperty().bindBidirectional(model.titleProperty());
  datePicker.valueProperty().bindBidirectional(model.dateProperty());
 }
}
```

360°
thinking.

Discover the truth at www.deloitte.ca/careers

Deloitte.

© Deloitte & Touche LLP and affiliated entities.

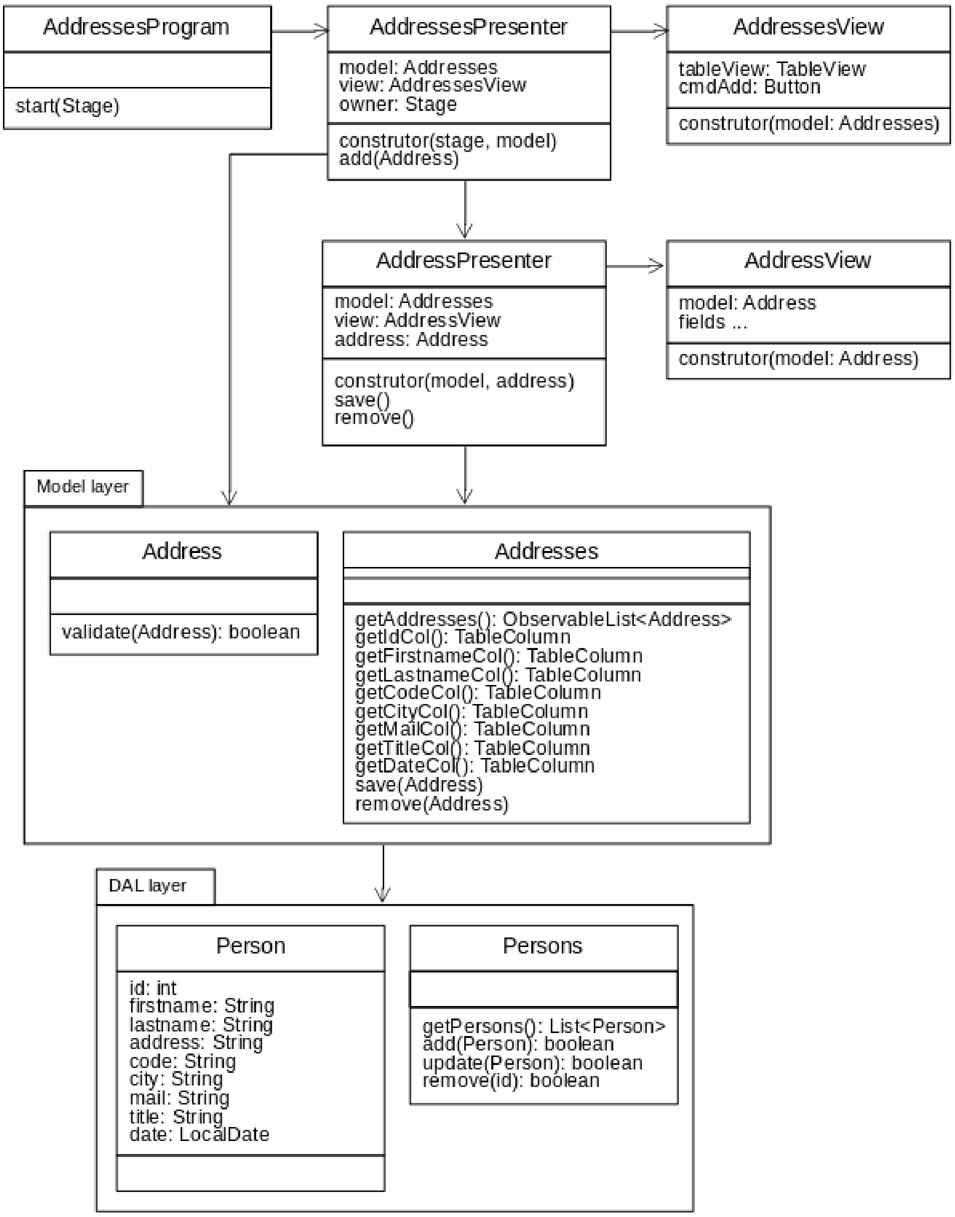
Here you should note that all nodes that can be referred by the event handlers are defined with package visibility so that they can be referenced in the presenter class. In some way, it violates object-oriented principles where variables must be private, but if one wishes to comply with these principles, it is necessary to write get methods for all instance variables. Therefore, you typically implement the MVP pattern as above and require that the view and presenter are in the same package. You should note the method *bindFields()* that creates bidirectional bindings for all fields, and it is here that it is necessary that the model class *Address* has JavaFX properties that you can bind to. This means that the values entered automatically update the model class, and vice versa that modifications of the model automatically update the user interface.

The corresponding presenter class is basically simple and resembles the previous presenter class, and among other things it is this class that creates the view component. However, there is a problem to be solved. If you edit an address, the model is automatically updated because of the bindings, which is incorrect if you close the dialog with cancel. In this case, the entries that have been made must be canceled. Therefore, the dialog must work on a copy, so only in case of clicking *OK*, updates the original *Address* object.

Then there is finally the main program, which starts it all:

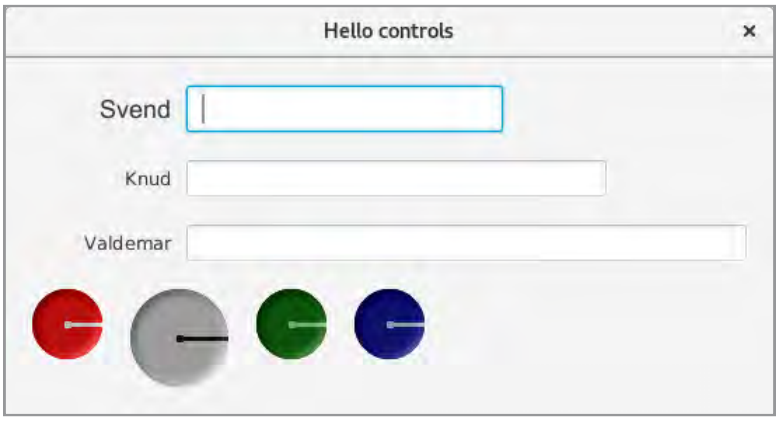
```
public class AddressesProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  AddressesPresenter presenter = new AddressesPresenter(stage, createModel());
  Scene scene = new Scene(presenter.getView());
  scene.getStylesheets().add("resources/css/styles.css");
  stage.setTitle("Addresses");
  stage.setScene(scene);
  stage.show();
 }

 private Addresses createModel()
 {
  try
  {
   return new Addresses();
  }
  catch (Exception ex)
  {
   System.out.println(ex);
   Platform.exit();
   return null;
  }
 }
 public static void main(String[] args)
 {
  launch(args);
 }
}
```

There is not much to explain, but you should note how the method *start()* creates a presenter and a model that is sent as a parameter for the presenter class's constructor. The architecture of the completed program can be illustrated as follows:


AddressesProgram
start(Stage)
AddressesPresenter
model: Addresses
view: AddressesView
owner: Stage
construtor(stage, model)
add(Address)
AddressesView
tableView: TableView
cmdAdd: Button
construtor(model: Addresses)
AddressPresenter
model: Addresses
view: AddressView
address: Address
construtor(model, address)
save()
remove()
AddressView
model: Address
fields ...
construtor(model: Address)
Model layer
Address
validate(Address): boolean
Addresses
getAddresses(): ObservableList<Address>
getIdCol(): TableColumn
getFirstnameCol(): TableColumn
getLastnameCol(): TableColumn
getCodeCol(): TableColumn
getCityCol(): TableColumn
getMailCol(): TableColumn
getTitleCol(): TableColumn
getDateCol(): TableColumn
save(Address)
remove(Address)
DAL layer
Person
id: int
firstname: String
lastname: String
address: String
code: String
city: String
mail: String
title: String
date: LocalDate
Persons
getPersons(): List<Person>
add(Person): boolean
update(Person): boolean
remove(id): boolean

# 6 USER DEFINED CONTROLS

It is rarely necessary to write your own controls, but the possibility exists, and it could even be the task to developing your own user defined controls. A control must be derived directly or indirectly from the class *Node* so that it can be part of a scene graph in the same way as all other controls, but in most cases, one will write a user defined control as a class inheriting an existing control, as it is made for the most important functionality. If you open the program *UserControlProgram*, you get the following window:



SIMPLY CLEVER
ŠKODA

We will turn your CV into
an opportunity of a lifetime

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

which contains 7 custom controls. The top three have the type *LabelField*, consisting of a *Label* and a *TextField*, while the 4 lower ones have the type *Spinner*. It's a kind of button and clicking on the circle it will rotate the line (default 2 seconds), after which the component will trigger an *ActionEvent*. The program's *start()* method is as follows:

```
public void start(Stage primaryStage)
{
 Spinner spin1 = new Spinner(50, Color.RED, Color.WHITE);
 spin1.setOnAction(e -> System.out.println("ok 1"));
 Spinner spin2 = new Spinner(70);
 spin2.setOnAction(e -> System.out.println("ok 2"));
 Spinner spin3 = new Spinner(50, Color.RED, Color.WHITE, 1000);
 spin3.setOnAction(e -> System.out.println("ok 3"));
 spin3.setBackground(Color.DARKGREEN);
 spin3.setForeground(Color.LIGHTGREEN);
 Spinner spin4 = new Spinner(50, Color.DARKBLUE, Color.LIGHTBLUE);
 spin4.setOnAction(e -> System.out.println("ok 4"));
 spin4.setTime(5000);
 HBox pane = new HBox(20, spin1, spin2, spin3, spin4);
 LabelField field = null;
 VBox root = new VBox(20, field = new LabelField("Svend", "", 100),
  new LabelField("Knud", "", 100, 300), new
  LabelField("Valdemar", "", 100, 400), pane);
 field.textProperty().addListener((ob, ov, nv) -> System.out.println(nv));
 field.setFont(Font.font("Arial", 18));
 root.setPadding(new Insets(20, 20, 20, 20));
 Scene scene = new Scene(root);
 primaryStage.setTitle("Hello controls");
 primaryStage.setScene(scene);
 primaryStage.show();
}
```

Initially, 4 *Spinner* controls are created, and the goal is to show how to create a *Spinner* (which parameters can be specified by the constructor) and which properties can be subsequently set. Here you should especially note how to associate an event handler. They are all trivial and the goal is to see (with a text on the console) when the events in question occurs. The 4 Spinner controls are placed in a *HBox* that can be added to the root of the window. When *root* is created (as a *VBox*), first 3 *LabelField* controls are added, where you will primarily observe the parameters of the constructor. For the first control is associated with an event handler – which is trivial and where the goal is to show that you can associate an event handlers to the entry field.

Both user controls are simple and hardly have not the great pratical interest, but they show something about how to create a user control. In the rest of this chapter I will describe how the two controls are written.

## 6.1 A LABELFIELD

When writing a custom control, the two main decisions are what properties should be and what events the control is going to fire. The actual control is, in fact, not much more than a *HBox* with a *Label* and a *TextField*, and in addition to the properties of a *HBox*, you should be able to set the width of the *Label* and the *TextField*, respectively, and set the text for both components. Finally, there must be a property for the font that should apply to both the *Label* and the *TextField*, so that both components always uses the same font. When you enter text or otherwise change the text for the *TextField* component, it should send *ChangeEvents*, and in order for these events to be captured from an application, the component returns the *TextField* component's *textProperty*. The component can then be written where I have only included one of 5 constructors and two properties:

```
public class LabelField extends HBox
{
 private Label label;
 private TextField field = new TextField();

 public LabelField(String caption, String text, double captionWidth,
  double fieldWidth)
 {
  setAlignment(Pos.BASELINE_LEFT);
  label = caption == null ? new Label() : new Label(caption);
  if (text != null) field.setText(text);
  if (captionWidth > 0) label.setPrefWidth(captionWidth);
  if (fieldWidth > 0) field.setPrefWidth(fieldWidth);
  label.setAlignment(Pos.CENTER_RIGHT);
  this.setSpacing(10);
  getChildren().addAll(label, field);
 }

 public double getCaptionWidth()
 {
  return captionWidthProperty().get();
 }

 public void setCaptionWidth(double captionWidth)
 {
  captionWidthProperty().set(captionWidth);
 }

 public DoubleProperty captionWidthProperty()
 {
  return label.prefWidthProperty();
 }
 public String getText()
 {
  return textProperty().get();
 }

 public void setText(String text)
 {
  textProperty().set(text);
 }

 public StringProperty textProperty()
 {
  return field.textProperty();
 }
}
```

In fact, there is not much to notice, and the most important code is found in the constructor, where the two components should be placed "nicely" in the container. Note that the class inherits *HBox* and therefore is a component that can be inserted into a scene graph like all other controls.

e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

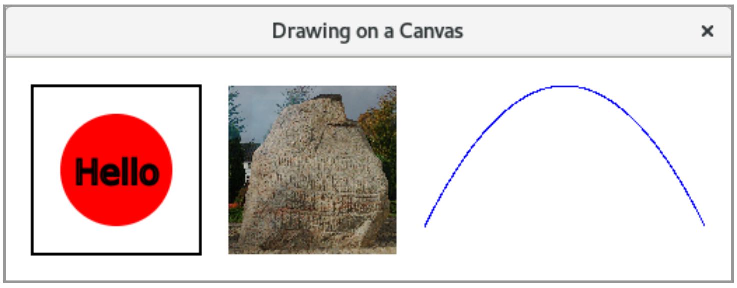
**About e-Learning for Kids** Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

## 6.2 A CANVAS

Before I look at the last control, I will mention a *Canvas*, which is a control I have not previously mentioned. A *Canvas* is a control consisting of a drawing surface where you can draw geometric figures and text and insert images using drawing features. You can also manipulate the individual pixels using a *PixelWriter*.

A *Canvas* has attached a *GraphicsContext* class that represents the graphic content and provides drawing functions available. You are encouraged to investigate which methods *GraphicsContext* makes available.

For example, the program *CanvasProgram* opens the following window:



where a square, a circle, a text and a curve (a parable) are drawn and a picture is added. The new is that it has been done using a *Canvas*:

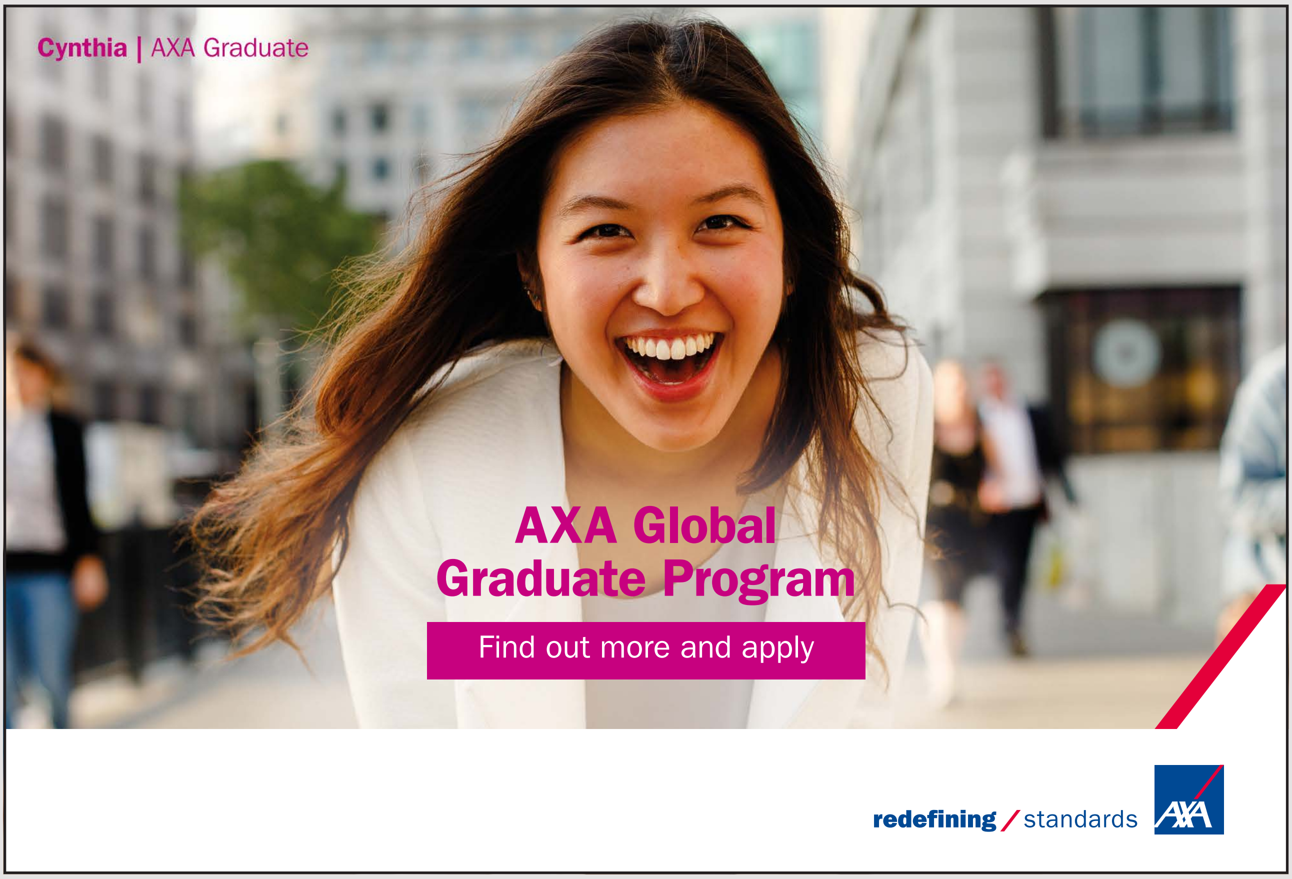
```
package canvasprogram;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.*;
import javafx.scene.image.*;
import javafx.scene.layout.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;
import javafx.stage.Stage;

public class CanvasProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  Canvas canvas = new Canvas(520, 160);
  GraphicsContext gc = canvas.getGraphicsContext2D();
  gc.setLineWidth(2.0);
  gc.strokeRect(20, 20, 120, 120);
  gc.setFill(Color.RED);
  gc.fillOval(40, 40, 80, 80);
  gc.setFont(Font.font(24));
  gc.strokeText("Hello", 50, 90);
  Image image = new Image("resources/images/stone.jpg");
  gc.drawImage(image, 160, 20, 120, 120);
  writeGraph(gc);
  Pane root = new Pane();
  root.getChildren().add(canvas);
  Scene scene = new Scene(root);
  stage.setScene(scene);
  stage.setTitle("Drawing on a Canvas");
  stage.show();
 }

 private void writeGraph(GraphicsContext gc)
 {
  PixelWriter writer = gc.getPixelWriter();
  for (double x = 300; x < 500; x += 0.25)
   writer.setColor((int)x, (int)y(x), Color.BLUE);
 }

 private double y(double x)
 {
  return x * x / 100 - 8 * x + 1620;
 }

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

The code is easy enough to figure out. In the method *start()*, a *Canvas* object is created of a certain size, and a reference is created to the *Canvas* object's *GraphicsContext*. For this object, the width of the pen to be drawn with is defined, then a drawing tool is used to draw a rectangle. Since nothing is said about the color, it is drawn with a black pen. Next, a red fill color is added and a filled circle is drawn. Note specially what the drawing function is called. As a next step, a font is defined and a text is drawn. Finally, an image is inserted and finally the method *writeGraph()* is called, which draws the curve. The curve is not too "nice" – is some pixelated – and there are other and better ways to draw such a graph, but the example should show how to manipulate the individual pixels in a *Canvas*. Finally, note that a *Canvas* is a node, and therefore can be added to a scene graph.

## 6.3 A SPINNER

The custom control can be written as follows, where I have not shown the code for the three properties for setting the rotation speed and color and only one constructor:

```
public class Spinner extends Canvas
{
 private ObjectProperty<EventHandler<ActionEvent>> onActionProperty =
  new SimpleObjectProperty<EventHandler<ActionEvent>>();
 private IntegerProperty time = new SimpleIntegerProperty();
 private ObjectProperty<Color> background = new SimpleObjectProperty();
 private ObjectProperty<Color> foreground = new SimpleObjectProperty();
 private double size;
 private Circle circle;
 private Transition transition;

 …

 public Spinner(double size, Color background, Color foreground, int time)
 {
  super(size, size);
  this.size = size;
  setBackground(background);
```


Cynthia | AXA Graduate
AXA Global
Graduate Program
Find out more and apply
redefining / standards
AXA

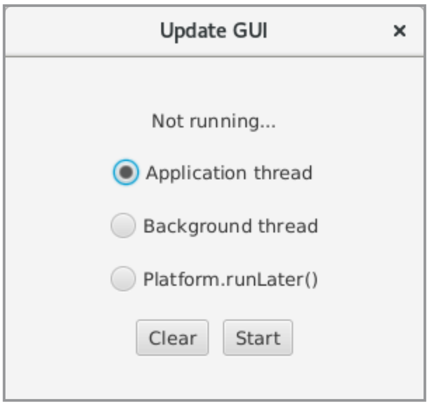
```
 setForeground(foreground);
 setTime(time);
 circle = new Circle(size / 2, size / 2, size / 2);
 transition = createTransition();
 transition.setOnFinished(this::clicked);
 draw();
 this.addEventHandler(MouseEvent.MOUSE_CLICKED, new ClickHandler());
 this.setEffect(new Lighting());
}

private void draw()
{
 GraphicsContext gc = getGraphicsContext2D();
 gc.setFill(getBackground());
 gc.fillOval(0, 0, size, size);
 gc.setFill(getForeground());
 double s = size / 2;
 gc.fillRect(s, s – 1, s, 3);
 gc.fillRect(s – 2, s – 2, 5, 5);
}

…

public EventHandler<ActionEvent> getOnAction()
{
 return onActionProperty.get();
}

public void setOnAction(EventHandler<ActionEvent> handler)
{
 onActionProperty.set(handler);
}

public ObjectProperty<EventHandler<ActionEvent>> onActionProperty()
{
 return onActionProperty;
}

private void clicked(ActionEvent e)
{
 if (getOnAction() != null) getOnAction().handle(new ActionEvent());
}

private RotateTransition createTransition()
{
 RotateTransition trans =
  new RotateTransition(Duration.millis(getTime()), this);
  trans.setByAngle(360);
  trans.setCycleCount(1);
  return trans;
 }

 class ClickHandler implements EventHandler<MouseEvent>
 {
  public void handle(MouseEvent e)
  {
   if (circle.contains(e.getX(), e.getY())) transition.play();
  }
 }
}
```

Mostly happens in the constructor that initially initiates properties and creates a *Transition* for a rotation. You should note that an event handler is associated with an event that occurs when the rotation ends and that it fires an *ActionEvent* if there is a listener. The constructor also calls the method *draw()*, which draws the component on a *Canvas*. Note that there is no particular reason for using a *Canvas* (in addition to showing how), as you could achieve the same with existing *Node* classes. The class also has a *Circle* object that is used to make it easy to test in the event handler for click with the mouse and if the circle is clicked.

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

# 7 JAVAFX AND CONCURRENCY

Java GUI applications are generally multi-threaded, and similar to Swing JavaFX uses a special thread to update the user interface. This thread is called *JavaFX Application Thread*. Since all nodes in a the program's scene graph are not thread-safe (for the reasons of performence), there are the same challenges that you know from Swing that you can not directly update them from another thread, but it should be done by calling a method performed in the *JavaFX Application Thread*. In this chapter I want to show what JavaFX makes available to make it simple. The program *UpdateGUI* opens the following window:



If you click the *Start* button, the program starts a method that takes a long time and after the method is completed, the top label is updated. Clicking the *Clear* button deletes the content of the top label. If you try the program while the top radio button is pressed, you will find that nothing happens when you click the *Clear* button – at least not before the method started of the *Start* button is completed. The user experiences that the program "hangs". Solutions are, of course, to execute the method that takes time, in its own thread. Pressing the middle radio button and clicking *Start* again executes the method in a background thread, but now you get an exception. The reason is that the background thread tries to update the JavaFX Application Thread, which raises an exception. On the other hand, press the bottom radio button and click *Start* again, so it runs as it should, where the top label is updated on a regular basis, and you can delete that label at any time by clicking the *Clear* button.

```
public class UpdateGUI extends Application
{
 private Label label = new Label("Not running…");
 private RadioButton cmd1 = new RadioButton("Application thread");
 private RadioButton cmd2 = new RadioButton("Background thread");
 private RadioButton cmd3 = new RadioButton("Platform.runLater()");
 private Button cmdOk;
```

```
@Override
public void start(Stage stage)
{
 ToggleGroup group = new ToggleGroup();
 cmd1.setToggleGroup(group);
 cmd2.setToggleGroup(group);
 cmd3.setToggleGroup(group);
 cmd1.setSelected(true);
 HBox commands = new HBox(10, createButton("Clear", e -> label.setText("")),
  cmdOk = createButton("Start", this::work));
 commands.setAlignment(Pos.CENTER);
 VBox root = new VBox(20, label, cmd1, cmd2, cmd3, commands);
 root.setPadding(new Insets(20, 20, 20, 20));
 root.setAlignment(Pos.CENTER);
 Scene scene = new Scene(root, 300, 250);
 stage.setScene(scene);
 stage.setTitle("Update GUI");
 stage.show();
}

private void work(ActionEvent e)
{
 cmdOk.setDisable(true);
 if (cmd1.isSelected()) work1();
 else
 {
  Thread th = cmd2.isSelected() ? new Thread(() -> work1()) :
   new Thread(() -> work2());
  th.setDaemon(true);
  th.start();
 }
}

private void work1()
{
 for(int i = 1; i <= 10; i++)
 {
  label.setText("Start calulation " + i);
  calculate();
 }
 label.setText("All calulations terminated");
 cmdOk.setDisable(false);
}

private void work2()
{
 for(int i = 1; i <= 10; i++)
 {
  String text = "Start calulation " + i;
  Platform.runLater(() -> label.setText(text));
  calculate();
 }
 String text = "All calulations terminated";
 Platform.runLater(() -> label.setText(text));
 Platform.runLater(() -> cmdOk.setDisable(false));
}

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

private void calculate()
{
 double y = 0;
 for (int i = 0; i < 10; ++i) for (int j = 0; j < 10000000; ++j)
  y = Math.sin(Math.sqrt(2));
}
```

ie business school

93%
OF MIM STUDENTS ARE
WORKING IN THEIR SECTOR 3 MONTHS
FOLLOWING GRADUATION

MASTER IN MANAGEMENT

- STUDY IN THE CENTER OF MADRID AND TAKE ADVANTAGE OF THE UNIQUE OPPORTUNITIES THAT THE CAPITAL OF SPAIN OFFERS
- PROPEL YOUR EDUCATION BY EARNING A DOUBLE DEGREE THAT BEST SUITS YOUR PROFESSIONAL GOALS
- STUDY A SEMESTER ABROAD AND BECOME A GLOBAL CITIZEN WITH THE BEYOND BORDERS EXPERIENCE

Length: 10 MONTHS
Av. Experience: 1 YEAR
Language: ENGLISH / SPANISH
Format: FULL-TIME
Intakes: SEPT / FEB

5 SPECIALIZATIONS
PERSONALIZE YOUR PROGRAM

#10 WORLDWIDE
MASTER IN MANAGEMENT
FINANCIAL TIMES

55 NATIONALITIES
IN CLASS

www.ie.edu/master-management | mim.admissions@ie.edu | Follow us on IE MIM Experience

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

The method to be performed is performed in *work1()* and *work2()*, which both performs the method *calculate()* 10 times, respectively. Before each execution of the method, the program's label is updated. *calculate()* performs nothing interesting, but it does many calculations that take a long time and the goal is that it is a method that constantly uses the machine's processor.

With regard to the user interface, there is not much to explain, and you should primarily note the method *work()*, which is the event handler for the *Start* button. The method tests which radio button is clicked in and if it is the top nothing else happens than the method *work1()* is called. That is, it is executed in the JavaFX Application Thread with the result that the program "hangs" until *work1()* is completed. For example, you will not see that the *Start* button is being disabled. Is it the middle radio button that is pushed in, the method *work1()* is performed again, but this time in its own thread. The result is an exception when the method tries to update the *Label* component. On the other hand, if the bottom radio button is pressed, the method *work2()* is performed, which is in principle is identical to *work1()*, but when the user interface is to be updated, it happens with the statement:

```
Platform.runLater(() -> label.setText(text));
```

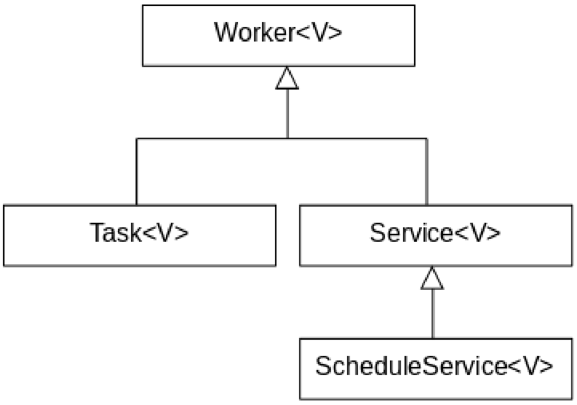
which simply means that the component *label* is updated in the JavaFX Application Thread. The class *Platform* has two static methods that relates to the JavaFX Application Thread:

- *public static boolean isFxApplicationThread()*
- *public static void runLater(Runnable runnable)*

where the first returns *true* if the calling thread are the JavaFX Application Thread, while the other creates a *Runnable* object to be executed by the JavaFX Application Thread at some point when the thread is running.

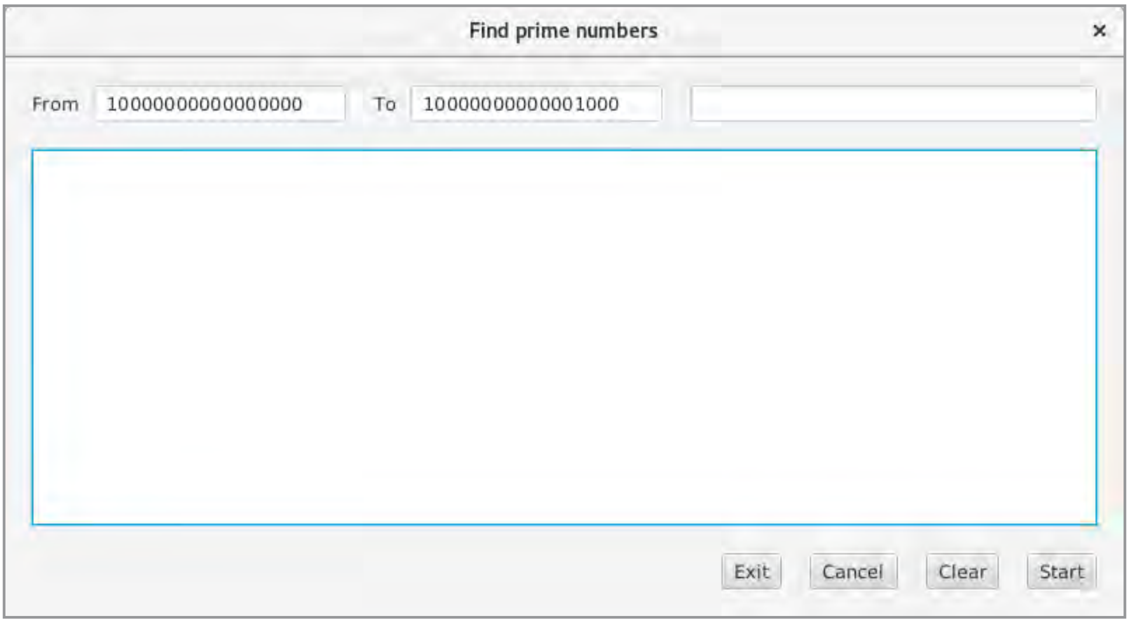
## 7.1 A TASK

To support threads in GUI programming, JavaFX offers a very simple concurrency framework based on the existing Java framework for concurrency. The framework consists of a single enumeration, called *State*, that defines the states that a thread may be in, as well as a single event type called *WorkerStateEvent*, and a thread fires such an event when it switches state. Else there are only four types of threads that constitute a simple hierarchy, the top one being an interface, while the other three are specific types:



Precisely, an instance of the *Worker* interface is a task to be performed by one or more background threads. The task's state is observable from the JavaFX Application Thread. The *Task*, *Service*, and *ScheduledService* classes are abstract classes that implement the *Worker* interface and represent different kinds of tasks. The *Task* class represents a task that can be performed once and a *Task* object can not be reused. The *Service* class represents a *Task* that can be repeated, and finally, the *ScheduledService* class represents a *Service* that can be performed at certain times. In the following I will show an example of a *Task* and a *Service*.

The task must be to determine prime numbers – a task you have seen many times in previous books. The goal is to have a task that for big numbers take a long time. Running the program *PrimesProgram1* opens the following window:



From 10000000000000000 To 10000000000001000

If you click on the *Start* button, the program determines all the primes within the entered interval and inserts them continuously in the list box. This happens, of course, in a secondary thread, and while the thread is running, the *Clear* button can delete the content of the list box, and with the *Cancel* button, you can stop the thread, and finally with the *Exit* button you can terminate the program.

To performe the task I will define a *Task* object. You should examine the documentation for both *Worker<V>* and *Task<V>*, but in the present case I have defined the following *Task* where I have not shown the code regarding the class's properties:

```
public class PrimesTask extends Task<ObservableList<Long>>
{
 private final ObservableList<Long> list =
  FXCollections.<Long>observableArrayList();
 private final LongProperty from = new SimpleLongProperty();
 private final LongProperty to = new SimpleLongProperty();

 …

 public void clear()
 {
```

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

```
 list.clear();
}

@Override
protected ObservableList<Long> call()
{
 list.clear();
 long count = getTo() - getFrom() + 1;
 long counter = 1;
 for (long i = getFrom(); i <= getTo(); ++i, ++counter)
 {
  if (this.isCancelled()) break;
  updateMessage("Check " + i + " as a prime number");
  if (isPrime(i))
  {
   long n = i;
   Platform.runLater(() -> list.add(n));
   updateValue(FXCollections.<Long>unmodifiableObservableList(list));
  }
  updateProgress(counter, count);
 }
 return list;
}

@Override
protected void cancelled()
{
 super.cancelled();
 updateMessage("The task was cancelled");
}

@Override
protected void failed()
{
 super.failed();
 updateMessage("The task failed");
}

@Override
public void succeeded()
{
 super.succeeded();
 updateMessage("The task finished successfully");
}

private boolean isPrime(long number)
{
 if (number == 2 || number == 3 || number == 5 || number == 7) return true;
```

```
  if (number < 11 || number % 2 == 0) return false;
  for (int t = 3, m = (int)Math.sqrt(number) + 1; t <= m; t += 2)
   if (number % t == 0) return false;
  return true;
 }
}
```

Note first that the class inherits *Task<ObservableList<Long >>*. Here the parameter type specifies the type that the abstract method *call()* returns. The two properties *from* and *to* defines the range within which the prime numbers are to be determined. They are defined as JavaFX properties as they must be able to be bound to the user interface. Then there is the method *call()*, which is the method that is performed in its own thread. It starts with a deletion of the list's content (what is not necessary in this example) and then a loop is executed that is iterating over the current range. For each iteration, the methods *updateMessage()*, *updateValue()* (only if a prime is found) and *updateProgress()* are called. These are the methods defined in the class *Task*, representing observable properties that the user interface can bind to.

Lastly, three methods are overriden, which are executed according to the state change, and the only thing that happens are that a property is updated on the *Task* object.

Then there is the class *PrimesProgram1*:

```
public class PrimesProgram1 extends Application
{
 private PrimesTask task = new PrimesTask();
 private LabelField txtFrom = new LabelField("From");
 private LabelField txtTo = new LabelField("To");
 private ListView<Long> lstPrimes = new ListView();
 private TextField txtMessages = new TextField();
 private ProgressBar progressBar = new ProgressBar(0);
```

The class has an instance of *PrimesTask* that represents the task to be performed. Note that the class uses the custom control from the previous chapter. I do not want to display the full code here, but the following method is called from *start()*:

```
public void bind(Worker<ObservableList<Long>> worker)
{
 txtFrom.textProperty().bindBidirectional(task.fromProperty(),
  new NumberStringConverter());
 txtTo.textProperty().bindBidirectional(task.toProperty(),
  new NumberStringConverter());
 progressBar.progressProperty().bind(worker.progressProperty());
 progressBar.visibleProperty().bind(worker.progressProperty().isNotEqualTo(
  new SimpleDoubleProperty(ProgressBar.INDETERMINATE_PROGRESS)));
 lstPrimes.itemsProperty().bind(worker.valueProperty());
 txtMessages.textProperty().bind(worker.messageProperty());
}
```

which binds the components in the user interface to properties in the *Task* object. Finally, there is the event handler for the *Start* button, which creates a background thread to execute the *Task* object:

```
private void start(ActionEvent e)
{
 Thread th = new Thread(task);
 th.setDaemon(true);
 th.start();
}
```

When the thread performs the method *start()*, it is the method *call()* in the class *PrimesTask* that is performed.

When you try out the program, it should all work as described, but click on the *Start* button again, you will find that nothing happens. As mentioned above, a *Task* object can only be executed once and this is where a *Service* object is an option.

Excellent Economics and Business programmes at:
university of groningen
AACSB ACCREDITED
“The perfect start of a successful, international career.”
CLICK HERE
to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

## 7.2 A SERVICE

The program *PrimesProgram2* opens the same window as shown in the previous example, and the class *PrimesTask* is unchanged. On the other hand, the following class has been added which represents a *Service*, and again I have not shown the code for the two properties:

```
public class PrimesService extends Service<ObservableList<Long>>
{
 private PrimesTask task;
 private final LongProperty from = new SimpleLongProperty();
 private final LongProperty to = new SimpleLongProperty();

 …

 public void clear()
 {
  if (task != null ) task.clear();
 }

 @Override
 protected Task<ObservableList<Long>> createTask()
 {
  task = new PrimesTask();
  task.setFrom(from.get());
  task.setTo(to.get());
  return task;
 }
}
```

A *Service* is not much more than a wrapper for a *Task*, and then you must override the method *createTask()* that is performed every time the service is performed. It creates the task to be executed and in this case initialized with the current values.

The class *PrimesProgram2* is also almost identical to *PrimesProgram1*, and the main difference is that the *Task* object has been replaced by a *Service* object and that the event handler for the *Start* button has been changed.

# 8 3D SHAPES

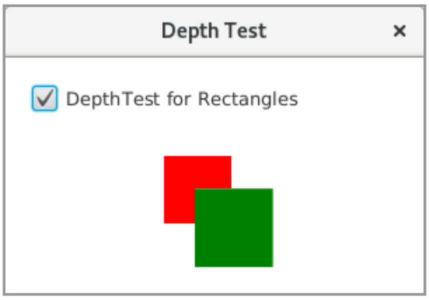
JavaFX also supports 3D graphics in some degree, although the possibilities are a bit limited, but future versions of the language will undoubtedly offer more. The following is a brief introduction to what it's all about, more than examples of practical applications.

There are only a few classes, all derived from *Shape3D*, and in fact, there are only four examples of specific shapes: *Box*, *Sphere*, *Cylinder* and *MeshView*, where the latter represents a customized shape. You can achieve 3D visualization using light and cameras that are also nodes and can be included in the scene graph and how a shape is displayed is determined by the position of the light sources and cameras in the scene graph.

However, Java3D is not necessarily supported, but it will be on most modern machines. You can test where it is the case with the following program (or equivalent statements):

```
package check3d;

import javafx.application.*;

public class Check3D
{
 public static void main(String[] args)
 {
  if (Platform.isSupported(ConditionalFeature.SCENE3D))
   System.out.println("3D is supported");
  else System.out.println("No 3D support");
 }
}
```

The starting point for 3D graphics is a 3D coordinate system with the origin in the upper left corner of the screen and a z axis that points into the screen while the x-axis and the y-axis are oriented in the same way as in 2D graphics. If in 2D you add two shapes to the scene graph that overlap, it is the figure added last that overlaps the first, but this is not necessarily in 3D, as the depth also plays a role. It can be illustrated with the program *Rectangles3D* that opens the following window:



where you should note that the green rectangle is drawn in front of the red. The program's code is:

```
public class Rectangles3D extends Application
{
 @Override
 public void start(Stage stage)
 {
  Rectangle red = new Rectangle(100, 100);
  red.setFill(Color.RED);
  red.setTranslateX(100);
  red.setTranslateY(100);
  red.setTranslateZ(400);
  Rectangle green = new Rectangle(100, 100);
  green.setFill(Color.GREEN);
  green.setTranslateX(150);
  green.setTranslateY(150);
  green.setTranslateZ(300);
  Group center = new Group(green, red);
  CheckBox check = new CheckBox("DepthTest for Rectangles");
  check.setSelected(true);
  check.selectedProperty().addListener((prop, oldValue, newValue) -> {
```

American online
LIGS University
is currently enrolling in the
Interactive Online BBA, MBA, MSc,
DBA and PhD programs:

- enroll by September 30th, 2014 and
- save up to 16% on the tuition!
- pay in 10 installments / 2 years
- Interactive Online education
- visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education.
More info here.

```
      if (newValue)
      {
        red.setDepthTest(DepthTest.ENABLE);
        green.setDepthTest(DepthTest.ENABLE);
      }
      else
      {
        red.setDepthTest(DepthTest.DISABLE);
        green.setDepthTest(DepthTest.DISABLE);
      }
    });
    BorderPane root = new BorderPane(center, check, null, null, null);
    root.setStyle("-fx-background-color: transparent;");
    root.setPadding(new Insets(20, 20, 20, 20));
    Scene scene = new Scene(root, 300, 200, true);
    scene.setCamera(new PerspectiveCamera());
    stage.setScene(scene);
    stage.setTitle("Depth Test");
    stage.show();
  }

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

In the method *start()*, two rectangles, a red and a green are defined, and the new is that this time, there is also defined a value for the z axis, that respectively is 400 and 300. That is, the red rectangle lies farther away than the green one. The two rectangles are added to a *Group* node, but so that the red is added last, and it should therefore overlap the green, but that is the opposite that happens as the red is farther away. To make it happen, two more things have to be done. When you create the *Scene* object, you must specify another parameter that tells to use 3D graphics, and you should also attach a camera to the scene graph.

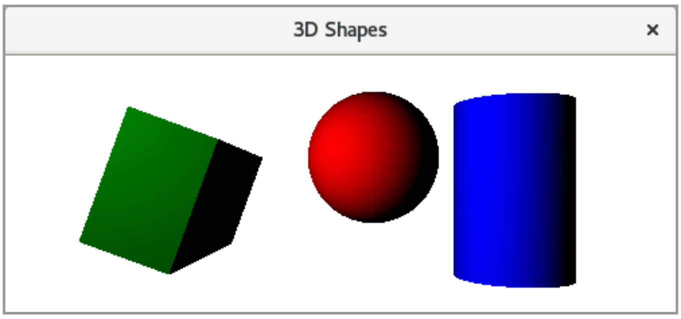
The window also has a checkbox and the meaning is that you can turn the 3D effect on or off and unchecking the box displays the graphics as the usual 2D graphics, where the red rectangle will overlap the green.

## 8.1 BOX, SPHERE AND CYLINDER

As mentioned, JavaFX is born with three shape classes, representing respectively a box, a sphere and a cylinder. All classes inherit from *Shape3D* and inherit here three crucial properties:

1. material
2. drawing mode
3. cull face

as all are explained later, but the first is used in the following example, and as the name says, it tells how the surface should be drawn. When you create a *Shape3D*, it has its center in the starting point of the coordinate system, but as it is a node, it can be moved using a transformation. Finally, its position and size are determined by the camera used to display the figure, and where the camera is located in the coordinate system. The current location of the camera can make it difficult to predict the actual location of a figure. The application *ShapesProgram* opens the following window and shows examples of the three shapes:



The box object has been moved with a translation but is also rotated. The others are moved with a translation, but when the cylinder is slightly deformed, it is because the camera is located at the left. You should also note that the color is determined by the location of the light. The code is as follows:

```
package shapesprogram;

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

public class ShapesProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
```

```
 Scene scene = new Scene(new Group(createBox(), createSphere(),
  createCylinder(), createLight()), 500, 200, true);
 scene.setCamera(createCamera());
 stage.setScene(scene);
 stage.setTitle("3D Shapes");
 stage.show();
}

private Box createBox()
{
 Box b = new Box(100, 150, 300);
 b.setTranslateX(150);
 b.setTranslateY(200);
 b.setTranslateZ(200);
 b.setRotate(20);
 b.setMaterial(new PhongMaterial(Color.GREEN));
 return b;
}

private Sphere createSphere()
{
 Sphere s = new Sphere(100);
 s.setTranslateX(400);
 s.setTranslateY(150);
```

DON'T EAT YELLOW SNOW
What will your advice be?

**Some advice just states the obvious.** But to give the kind of advice that's going to make a real difference to your clients you've got to listen critically, dig beneath the surface, challenge assumptions and be credible and confident enough to make suggestions right from day one. At Grant Thornton you've got to be ready to kick start a career right at the heart of business.

Sound like you? Here's our advice: visit
**GrantThornton.ca/careers/students**

Scan here to learn more about a career with Grant Thornton.

Grant Thornton
An instinct for growth™

© Grant Thornton LLP. A Canadian Member of Grant Thornton International Ltd

```
   s.setTranslateZ(300);
   s.setMaterial(new PhongMaterial(Color.RED));
   return s;
  }

  private Cylinder createCylinder()
  {
   Cylinder c = new Cylinder(100, 300);
   c.setTranslateX(650);
   c.setTranslateY(200);
   c.setTranslateZ(400);
   c.setMaterial(new PhongMaterial(Color.BLUE));
   return c;
  }

  private PointLight createLight()
  {
   PointLight p = new PointLight();
   p.setTranslateX(100);
   p.setTranslateY(100);
   p.setTranslateZ(-100);
   return p;
  }

  private PerspectiveCamera createCamera()
  {
   PerspectiveCamera c = new PerspectiveCamera(false);
   c.setTranslateX(100);
   c.setTranslateY(100);
   c.setTranslateZ(-100);
   return c;
  }

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

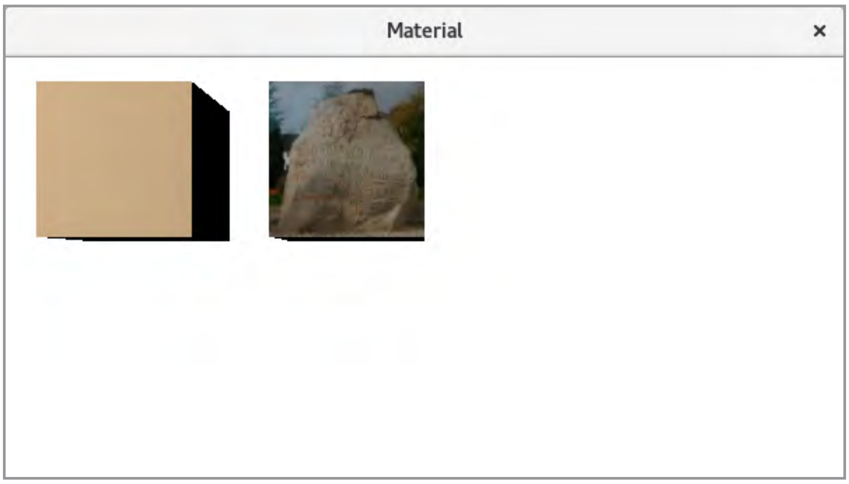
The code is actually quite simple and easy enough to figure out, but on the other hand, the effect is more difficult to figure out. All of this happens in the last 5 methods that create a *Box* object, a *Sphere* object, a *Cylinder* object, a *PointLight* object and a *PerspectiveCamera* object, respectively. With respect to the *Box* object, it is a box of 100 × 150 × 300 which is displaced at the point (150, 200, 200) and thus at a point that is 200 behind the screen. Afterwards the box is rotated 20 degrees. In particular, you should note how to assign a color to a surface with a *Material* object. A *Sphere* and a *Cylinder* are, in principle, created in the same way, such that for a *Sphere* object, a radius must be defined while for a cylinder it is necessary to indicate both the radius and the height. The camera is located at the point (100, 100, -100) and thus on the left and slightly in front of the screen. Note that the light source is located in the same place, which obviously does not have to be the case.

In the method *start()*, the light source is attached to the scene graph in exactly the same way as any other node while the camera is directly attached to the *Stage* object.

You are encouraged to experiment with the program and note what happens if you change the values in the 5 methods. In particular, try changing the size of the window. Then the shapes are changed as well.

## 8.2 MATERIAL

A Shape3D object has a property of the type *Material* that determines how the surface appears, and it can be a color, but can also be a picture. The program *BoxProgram* opens a window with two *Box* objects:



where one has a color (*Color.BEIGE*), while the surface of the other is a picture.

```
public class BoxProgram extends Application
{
 @Override
 public void start(Stage stage)
 {
  PhongMaterial m1 = new PhongMaterial();
  m1.setDiffuseColor(Color.TAN);
  PhongMaterial m2 = new PhongMaterial();
 m2.setDiffuseMap(new Image("resources/images/stone.jpg"));
 Scene scene = new Scene(new Group(createBox(500, 310, 500, m1), createBox(800,
  310, 500, m2), createLight()), 600, 300, true);
 scene.setCamera(createCamera());
 stage.setScene(scene);
 stage.setTitle("Material");
 stage.show();
}

private Box createBox(double x, double y, double z, Material m)
{
 Box box = new Box(200, 200, 200);
 box.setMaterial(m);
 box.setTranslateX(x);
 box.setTranslateY(y);
 box.setTranslateZ(z);
 return box;
}

private PointLight createLight()
{
 PointLight p = new PointLight();
 p.setTranslateX(200);
```

Alcatel·Lucent
www.alcatel-lucent.com/careers

What if you could build your future and create the future?

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

```
  p.setTranslateY(200);
  p.setTranslateZ(-600);
  return p;
 }

 private PerspectiveCamera createCamera()
 {
  PerspectiveCamera c = new PerspectiveCamera(false);
  c.setTranslateX(600);
  c.setTranslateY(300);
  c.setTranslateZ(-50);
  return c;
 }

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

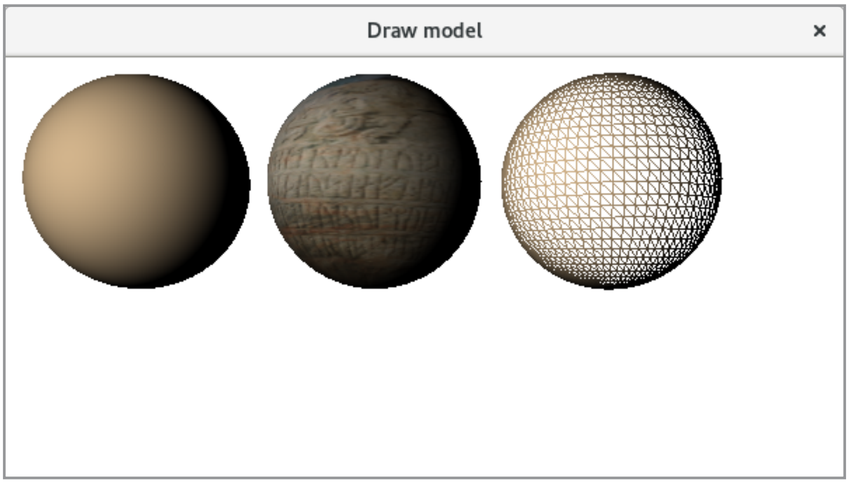
The two *Box* objects are created by the method *createBox()*, where the parameters are the coordinates of a translation as well as a *Material*. It is an abstract class, and there is only one specific class called *PhongMaterial*. It has a number of methods (such as *setDiffuseColor()* and *setDiffuseMap()*) used to specify how the surface should be and you are encouraged to examine the documentation. In this case, two *PhongMaterial* objects are created in the *start()* method, where the last refers to an image.

## 8.3 DRAW MODE

The surface of *Shape3D* objects is actually drawn as a number of triangles, and you can, with a parameter, indicate to the constructor how many there are to be. By default, these triangles are drawn as filled, but you can also indicate that you just have to draw the perimeter. There are thus two draw modes:

1. *DrawMode.FILL (default)*
2. *DrawMode.LINE*

The program *SphereProgram* opens the following window, which shows three *Sphere* objects, and the last one is drawn with *DrawMode.LINE*:



The code is basically the same as in the previous program, and there is only one significant change in which the method *createBox()* is replaced by the following method;

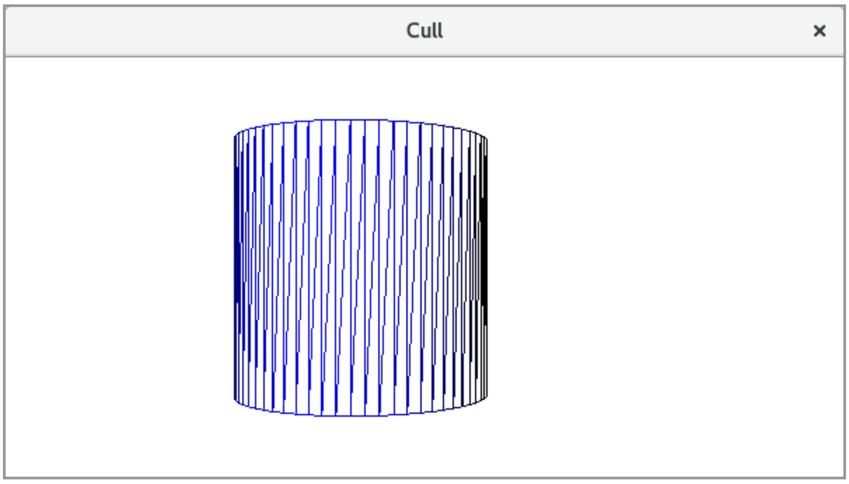
```
private Sphere createSphere(double x, double y, double z, Material m,
 DrawMode mode)
{
 Sphere sphere = new Sphere(150);
 sphere.setMaterial(m);
 sphere.setDrawMode(mode);
 sphere.setTranslateX(x);
 sphere.setTranslateY(y);
 sphere.setTranslateZ(z);
 return sphere;
}
```

## 8.4 CULL FACE

When a 3D figure appears on the screen, for natural reasons, you can not see the whole figure – you can not see what lies behind it. What you can see depends on where the camera is located. As mentioned in the previous section, a 3D shape is drawn as a number of triangles. A triangle has two sides in the form of the outside and the inside, and as a default one can see the outside. In general are dawn only the treangles, which are visible while the others are sorted out. There is a property called *CullFace*, which can assume three values:

1. CullFace.BACK
2. CullFace.FRONT
3. CullFace.NONE

where the first is default and means that you only see the triangles where you can see the outside while the other means that you can only see the inside of the triangles which are usually not visible. The last option means that you can see both kinds of triangles and thus draw all the triangles. The program *CylinderProgram* opens the following window:



Maastricht University *Leading in Learning!*

**Join the best at the Maastricht University School of Business and Economics!**

**Top master's programmes**

- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

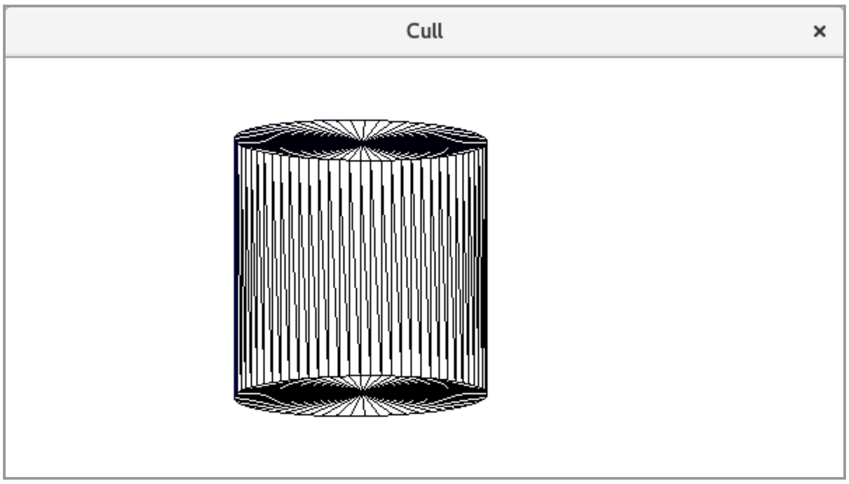
*Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012*

**Maastricht University is the best specialist university in the Netherlands** (Elsevier)

**Visit us and find out why we are the best!**
**Master's Open Day: 22 February 2014**

**www.mastersopenday.nl**

which shows a cylinder drawn with *DrawMode.LINE* and *CullFace.BACK*. If you click the cylinder with the mouse, you switch to *CullFace.FRONT*:

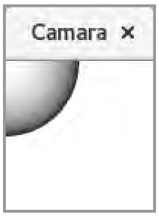


and click once to switch to *CullFace.NONE*, and then it all repeats itself. The code is simple and similar to the above programs, just an event is added for the mouse:

```
public class CylinderProgram extends Application
{
 private CullFace[] culls = { CullFace.BACK, CullFace.FRONT, CullFace.NONE
};
 private int pos = 0;
 private Cylinder cylinder;

 @Override
 public void start(Stage stage)
 {
  Group root = new Group(cylinder =
   createCylinder(500, 450, 600), createLight());
  root.addEventHandler(MouseEvent.MOUSE_CLICKED,
   new EventHandler<MouseEvent>() {
   public void handle(MouseEvent e) { pos = (pos + 1) % culls.length;
   cylinder.setCullFace(culls[pos]); };
  });
  Scene scene = new Scene(root, 600, 300, true);
  scene.setCamera(createCamera());
  stage.setScene(scene);
  stage.setTitle("Cull");
  stage.show();
 }

 private Cylinder createCylinder(double x, double y, double z)
 {
  Cylinder c = new Cylinder(200, 400);
  c.setMaterial(new PhongMaterial(Color.BLUE));
  c.setDrawMode(DrawMode.LINE);
  c.setTranslateX(x);
  c.setTranslateY(y);
  c.setTranslateZ(z);
  return c;
 }
```

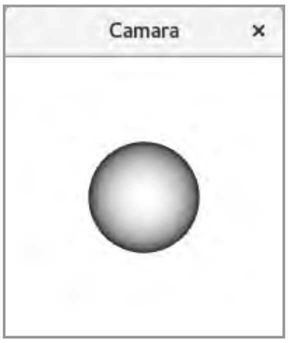
## 8.5 CAMERA AND LIGHT

The camera is the most crucial factor in 3D graphics, but it can be difficult to predict the effect. For example, if you create a sphere with radius 50 and without any kind of transition and create a camera in the same way, you get the following window:

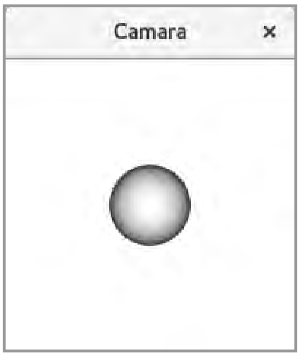


This corresponds to that the starting point of the coordinate system (0, 0, 0), which is the center of the sphere. The camera is located in the same position, and if you changed the window size, nothing happens with the shape. Since the sphere has no *Material*, a default value is given, which is light gray color, and since there is no mention of light, a light source with the same position as the camera is used.

If you move the shape 100 in each direction, the center of the sphere is shifted to (100, 100, 100) and the result is as shown below, where you should notice that the figure has become smaller as it is shown 100 into the screen and thus away from the viewer:



Note that the camera and the light are still in (0, 0, 0). If you then change the camera -200 in the z axis direction, you will see that the figure becomes even smaller as the camera has now moved further towards the viewer, and thus there is longer between the camera and the figure:



The camera used above has the type *PerspectiveCamera*, but there is also a camera *ParallelCamera*, and the difference is what kind of projection is used. A *PerspectiveCamera* is the most used since, as the name says, it sees the object in perspective. A clip area is attached to a camera so that shapes that are close to the camera do not appear and the same for shapes that are far away.

Business
Strategic Marketing Management
BI
Financial Economics
International Business
Leadership & Organisational Psychology
Shipping Management
BI NORWEGIAN BUSINESS SCHOOL
EFMD EQUIS ACCREDITED

Empowering People. Improving Business.

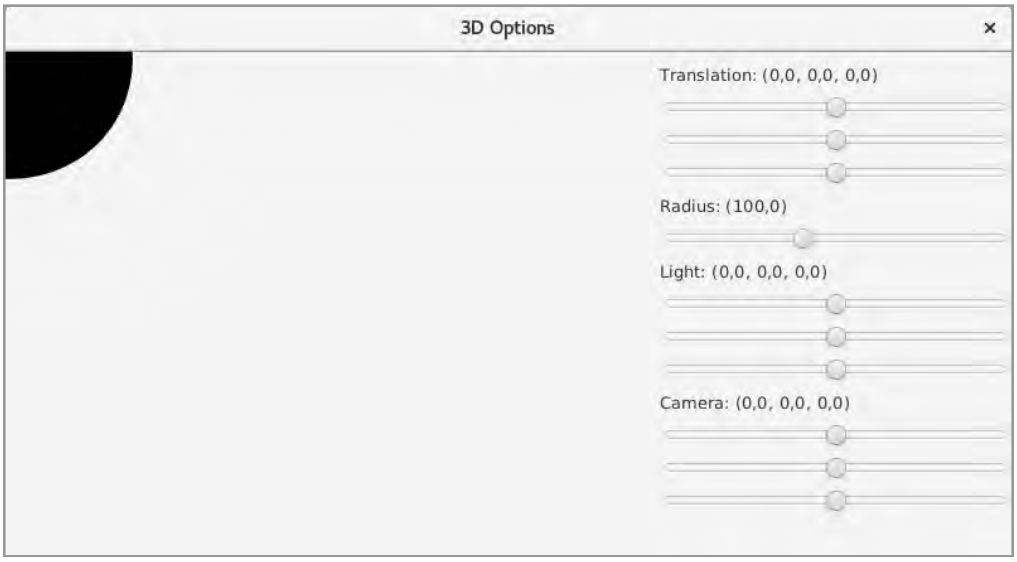
BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

To show a little about the effect of some of the many options and for experimentation, I will look at a program that I have called *Options3D*:



Translation: (0,0, 0,0, 0,0)

Radius: (100,0)

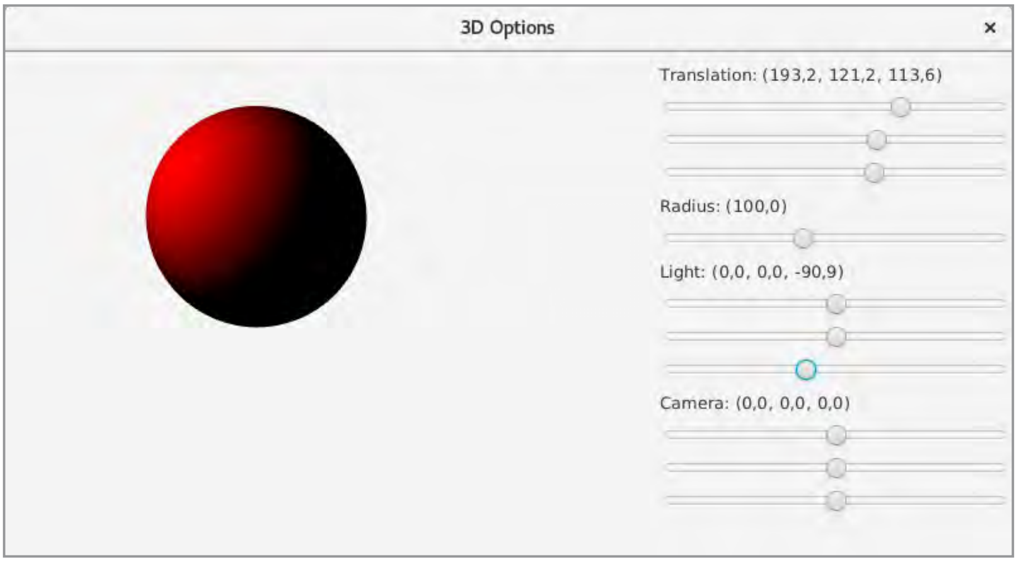
Light: (0,0, 0,0, 0,0)

Camera: (0,0, 0,0, 0,0)

The window has a *Sphere* object that sits in the upper left corner, and although it is not visible, the surface has a red *Material*. To the right there are 10 *Slider* controls that can be used to change the following settings:

1. position of the figure – (x, y, x) coordinate
2. radius
3. location of light source – (x, y, x) coordinate
4. camera location – (x, y, x) coordinate

Below I have shown the result after moving the figure and the light source:



Translation: (193,2, 121,2, 113,6)

Radius: (100,0)

Light: (0,0, 0,0, -90,9)

Camera: (0,0, 0,0, 0,0)

The program's code is as follows:

```
public class Options3D extends Application
{
 private Shape3D shape;
 private PointLight light = new PointLight();
 private PerspectiveCamera camera = new PerspectiveCamera(false);
 private Label tx = new Label();
 private Label ty = new Label();
 private Label tz = new Label();
 private Label lx = new Label();
 private Label ly = new Label();
 private Label lz = new Label();
 private Label cx = new Label();
 private Label cy = new Label();
 private Label cz = new Label();
 private Label ra = new Label();

 @Override
 public void start(Stage stage)
 {
  HBox root = new HBox(20, create3D(), create2D());
  Scene scene = new Scene(root, 800, 400, true);
  stage.setScene(scene);
  stage.setTitle("3D Options");
  stage.show();
 }

 private SubScene create3D()
 {
  Group root = new Group(shape = new Sphere(100), light);
  shape.setMaterial(new PhongMaterial(Color.RED));
  SubScene subScene = new SubScene(root, 500, 400, true,
   SceneAntialiasing.BALANCED);
  subScene.setCamera(camera);
  return subScene;
 }

 private SubScene create2D()
 {
  VBox commands = new VBox(10,
   createLabel("Translation", tx, ty, tz, shape.translateXProperty(),
    shape.translateYProperty(), shape.translateZProperty()),
   createSlider(-500, 500, 0, shape.translateXProperty()),
   createSlider(-500, 500, 0, shape.translateYProperty()),
   createSlider(-500, 500, 0, shape.translateZProperty()),
   createLabel("Radius", ra, ((Sphere)shape).radiusProperty()),
   createSlider(0, 250, 100, ((Sphere)shape).radiusProperty()),
  createLabel("Light", lx, ly, lz, light.translateXProperty(),
   light.translateYProperty(), light.translateZProperty()),
  createSlider(-500, 500, 0, light.translateXProperty()),
  createSlider(-500, 500, 0, light.translateYProperty()),
  createSlider(-500, 500, 0, light.translateZProperty()),
  createLabel("Camera", cx, cy, cz, camera.translateXProperty(),
   camera.translateYProperty(), camera.translateZProperty()),
  createSlider(-500, 500, 0, camera.translateXProperty()),
  createSlider(-500, 500, 0, camera.translateYProperty()),
  createSlider(-500, 500, 0, camera.translateZProperty()));
 commands.setPrefWidth(280);
 HBox root = new HBox(10, commands);
 root.setPadding(new Insets(10, 10, 10, 0));
 return new SubScene(root, 300, 380);
}

private Pane createLabel(String text, Label r, DoubleProperty p)
{
 r.textProperty().bind(p.asString("%3.1f"));
 HBox box = new HBox(new Label(text + ": ("), r, new Label(")"));
 return box;
}
```

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now

Go to www.helpmyassignment.co.uk for more info

Helpmyassignment

EXCELLENT
GOOD
AVERAGE
POOR
COMMENTS

```
 private Pane createLabel(String text, Label x, Label y, Label z,
  DoubleProperty px, DoubleProperty py, DoubleProperty pz)
 {
  x.textProperty().bind(px.asString("%3.1f"));
  y.textProperty().bind(py.asString("%3.1f"));
  z.textProperty().bind(pz.asString("%3.1f"));
  HBox box = new HBox(new Label(text + ": ("), x, new Label(", "), y,
   new Label(", "), z, new Label(")"));
  return box;
 }

 private Slider createSlider(double min, double max, double val,
  DoubleProperty property)
 {
  Slider slider = new Slider(min, max, val);
  slider.valueProperty().bindBidirectional(property);
  return slider;
 }

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

There is a lot to note in the code. There are no less than 13 instance variables, the first being a reference to the *Sphere* object. The two next are the light source and the camera respectively. Finally, the 10 *Label* controls that are used to display the values of the 10 *Slider* controls.

As for the method *start()*, there is not much to say besides placing two components in a *HBox*. However, there is a challenge. A *Scene* object can generally not display both 3D and 2D objects – at least not correct. To solve this problem, I have introduced the concept of a *SubScene* that allows you to have more *Scene* objects in a window. It can be used for multiple purposes (primarily in 3D graphics), and it can be used to split the window into a part for 3D objects and a part for usual 2D objects. That is exactly what is the case in this example.

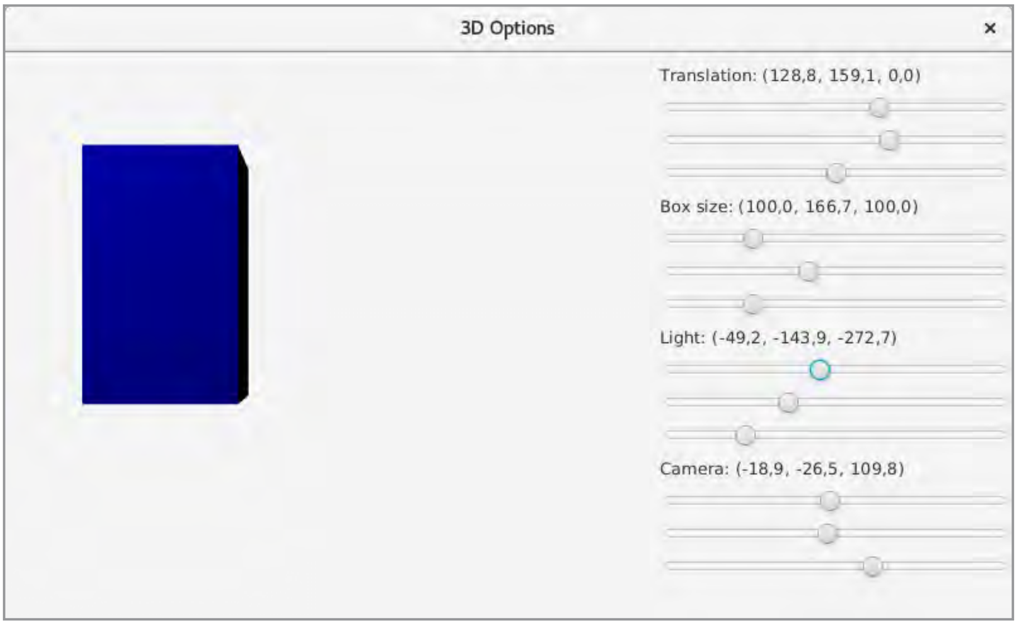
The method *create3D()* creates a 3D *Scene*, where the root is a *Group* with a *Sphere* that has a radius 100. Note that a *Material* is also defined. For this root, a *SubScene* is created. In particular, note how to specify that anti-aliasing is used and that the camera is attached to the *SubScene* object. This is why the image from the start is displayed black, because the camera's location in (0,0,0) corresponds to it is inside the ball.

The method *create2D()* also creates a *SubScene* object, but this time it's a usual 2D *Scene*. The code is a part, but it is because many controls must be created, and the code does not contain anything new in principle. When the individual controls (*Slider* controls) are created, note how the value property binds to a property by either the *Shape* object, the *Light* object or the *Camera* object. Similarly, the 10 *Label* controls are bound to the same properties, but here with a unidirectional binding, as the values are to be converted to strings.

You are encouraged to experiment with the program.

## EXERCISE 6

You must write a program similar to the program *Options3D*, but instead of a red *Sphere*, the program must show blue *Box*, which is initially (100, 100, 100). The window below shows the figure after several *Slider* controls have been moved. Note that two new *Slider* controls (and associated *Label* controls) have been added so you can set both width, height and depth of the *Box*. The easiest thing is to start with a copy of the *Options3D* program.



Translation: (128,8, 159,1, 0,0)

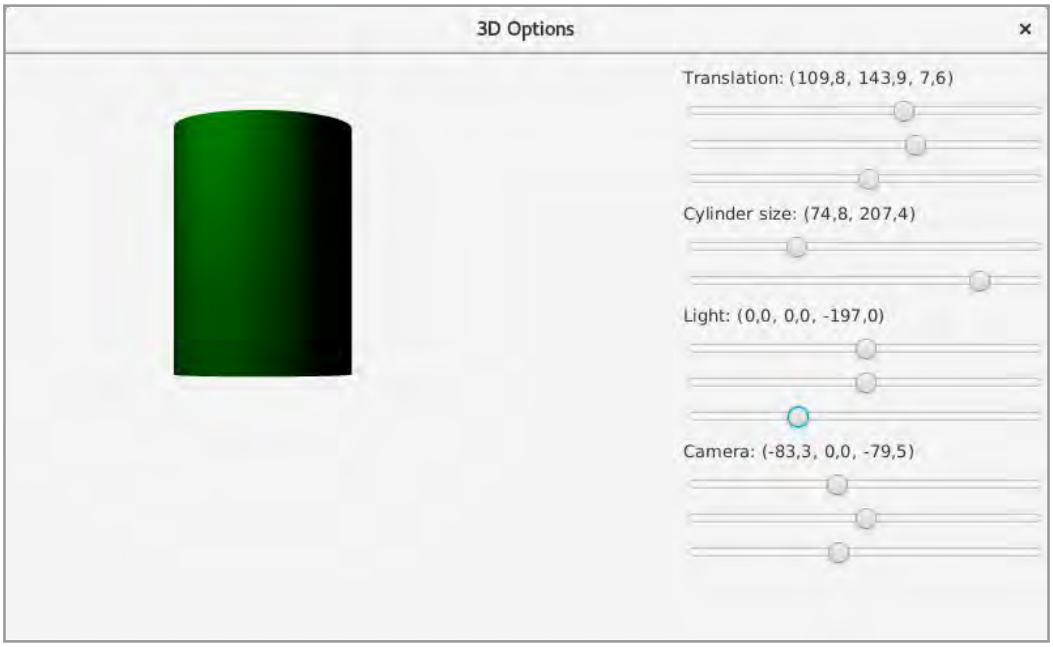
Box size: (100,0, 166,7, 100,0)

Light: (-49,2, -143,9, -272,7)

Camera: (-18,9, -26,5, 109,8)

## EXERCISE 7

Write another version of the above program, but where the *Shape3D* object this time is a green *Cylinder*, the size from start is (50, 100). You must be able to set both the radius and the height, and the result could be as shown below:



Translation: (109,8, 143,9, 7,6)

Cylinder size: (74,8, 207,4)

Light: (0,0, 0,0, -197,0)

Camera: (-83,3, 0,0, -79,5)

## 8.6 A LAST REMARK

As mentioned, a *Shape3D* shape is constructed using triangles, and it is possible – but far from easy – to define custom shapes by defining the individual triangles. I do not want to get closer to this in this book, and it takes a lot of practice, but the construction of custom shapes is done using the class *MeshView*.

# 9 CHARTS

In the final example of the book Java 10, I looked at a library that has classes for representation of graphs. As mentioned, there are many such libraries, and there are actually many examples of tasks where there is a desire for a graphical presentation of data, but in JavaFX it is easy as there is an API directly for that purpose, which has the most common graphs. The following is an introduction to this API, which basically include the following classes:

Brain power
SKF

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

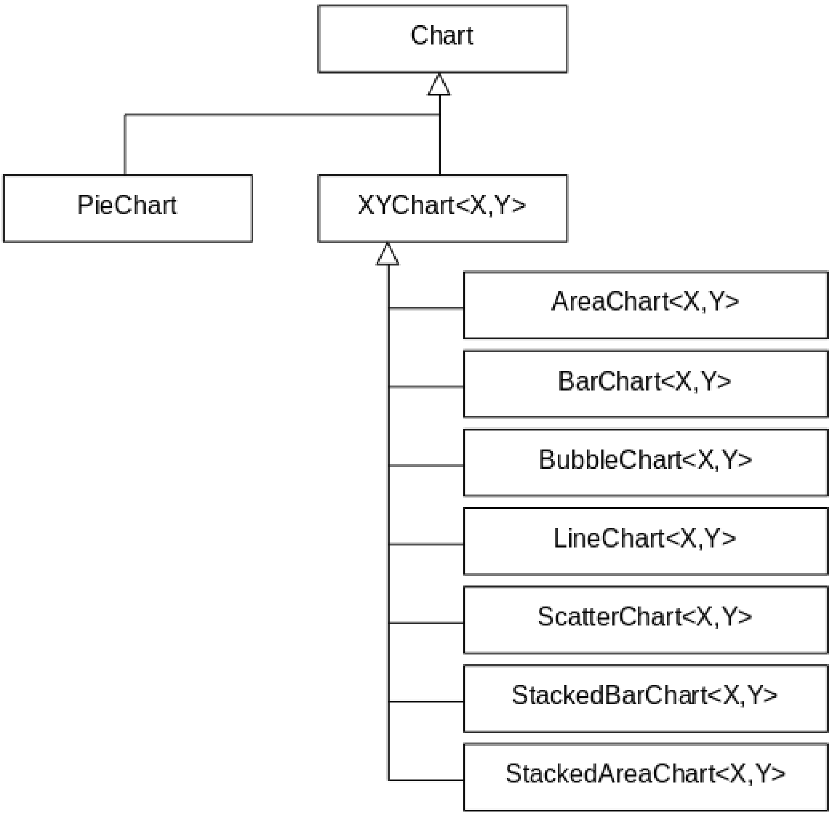
Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.

Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

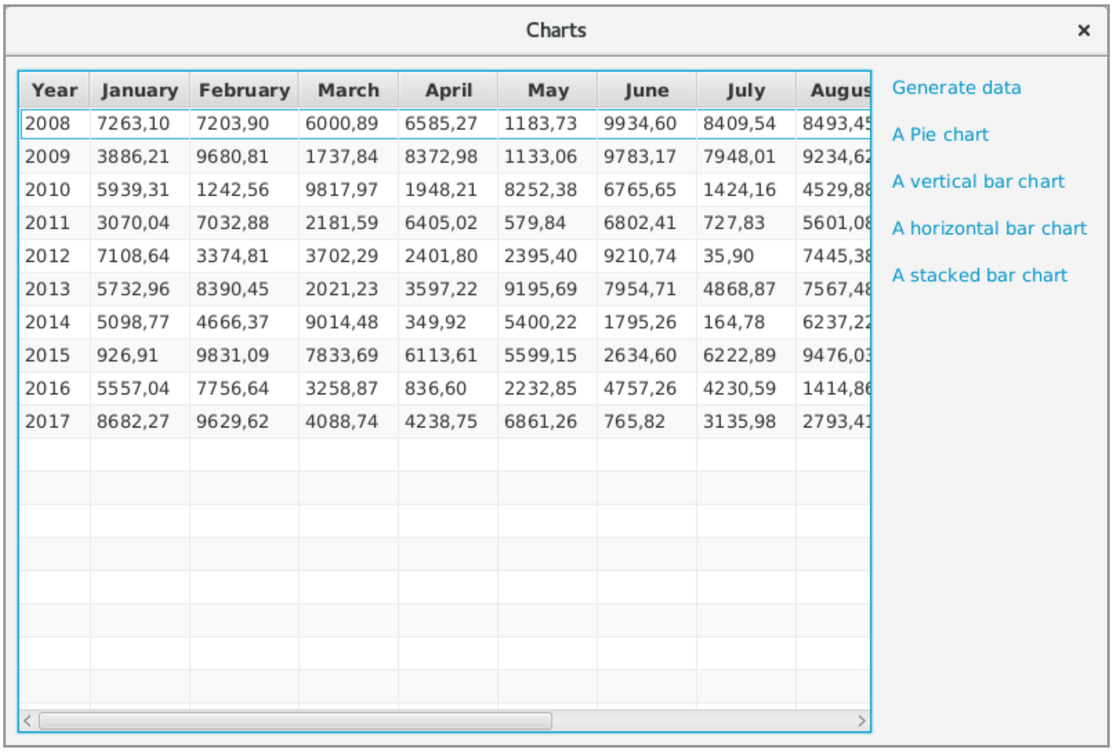


In addition, there are other auxiliary classes, including classes that represents the data that the individual graphs should illustrate, and there are classes to the axes of the coordinate system. The API can thus show 8 different graphs divided into two categories consisting of a *PieChart* and then the others. A graph is a graphical illustration of a number sequence, and a *PieChart* can show a single number sequence, while the others can display more called series in a xy-coordinate system. In the following example I would like to illustrate the following graphs:

- PieChart
- BarChart (two versions)
- StackedBarChart

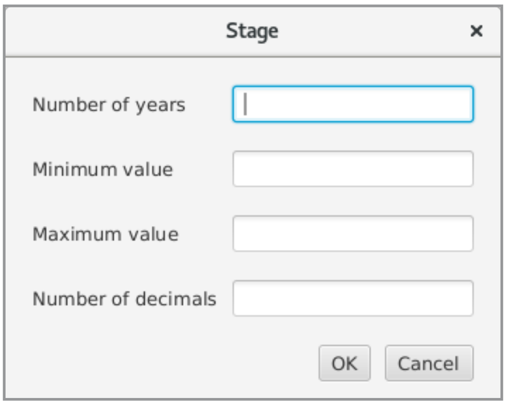
For all graphs, there are of course a number of settings, and the example does not show all settings, and the goal is to search the documentation yourself and experiment with the different settings, including expanding the program with new graphs using other types than the three mentioned above. It should be mentioned immediately that it is quite easy to work with the API and that it is similarly independent of the graph type.

The program is called *ChartProgram* and it opens the following window:


Charts
×
Generate data
A Pie chart
A vertical bar chart
A horizontal bar chart
A stacked bar chart

| Year | January | February | March | April | May | June | July | Augus |
|---|---|---|---|---|---|---|---|---|
| 2008 | 7263,10 | 7203,90 | 6000,89 | 6585,27 | 1183,73 | 9934,60 | 8409,54 | 8493,4 |
| 2009 | 3886,21 | 9680,81 | 1737,84 | 8372,98 | 1133,06 | 9783,17 | 7948,01 | 9234,6 |
| 2010 | 5939,31 | 1242,56 | 9817,97 | 1948,21 | 8252,38 | 6765,65 | 1424,16 | 4529,8 |
| 2011 | 3070,04 | 7032,88 | 2181,59 | 6405,02 | 579,84 | 6802,41 | 727,83 | 5601,0 |
| 2012 | 7108,64 | 3374,81 | 3702,29 | 2401,80 | 2395,40 | 9210,74 | 35,90 | 7445,3 |
| 2013 | 5732,96 | 8390,45 | 2021,23 | 3597,22 | 9195,69 | 7954,71 | 4868,87 | 7567,4 |
| 2014 | 5098,77 | 4666,37 | 9014,48 | 349,92 | 5400,22 | 1795,26 | 164,78 | 6237,2 |
| 2015 | 926,91 | 9831,09 | 7833,69 | 6113,61 | 5599,15 | 2634,60 | 6222,89 | 9476,0 |
| 2016 | 5557,04 | 7756,64 | 3258,87 | 836,60 | 2232,85 | 4757,26 | 4230,59 | 1414,8 |
| 2017 | 8682,27 | 9629,62 | 4088,74 | 4238,75 | 6861,26 | 765,82 | 3135,98 | 2793,4 |

When displaying a graph (a *Chart*), you must have some data in the form of a number material. What the numbers means is not so important for the current example, but the numbers could be interpreted as revenue over the 12 months of the year and the numbers above shows the revenue for the years 2008–2017. In order to experiment with graphs, the numbers are generated random and you can at any time create new numbers by clicking the *Generate Data* link:


Stage
×
Number of years
Minimum value
Maximum value
Number of decimals
OK
Cancel

You can also change the data by editing the table directly. The above dialog is called *ModelView*, but I do not want to show the code here as it contains nothing new. The table itself shows objects of the type *Year* and the datamodel of the table is called *DataModel*, and these classes are not shown for the same reasons either.

If you click on the *A Pie chart* link, the following events are performed:

```
private void pie(ActionEvent e)
{
 int n = table.getSelectionModel().getSelectedIndex();
 if (n >= 0)
 {
  Year year = model.getYears().get(n);
  ObservableList<PieChart.Data> data = FXCollections. observableArrayList();
  for (int i = 0; i < DataModel.names.length; ++i)
   data.add(new PieChart.Data(DataModel.names[i], year.getValue(i)));
  new PieView(parent, year.getYear(), data);
 }
}
```

The method determines the index for the first row that is selected, and for this row, a list of the type *ObservableList<PieChart.Data>* is created. You should note how this list is initialized with *PieChart.Data* objects, consisting of pairs where the first value is the month name (above the array *names*), while the other value is the numeric values from the row that is selected. Next, a window will appear showing a *PieChart*:

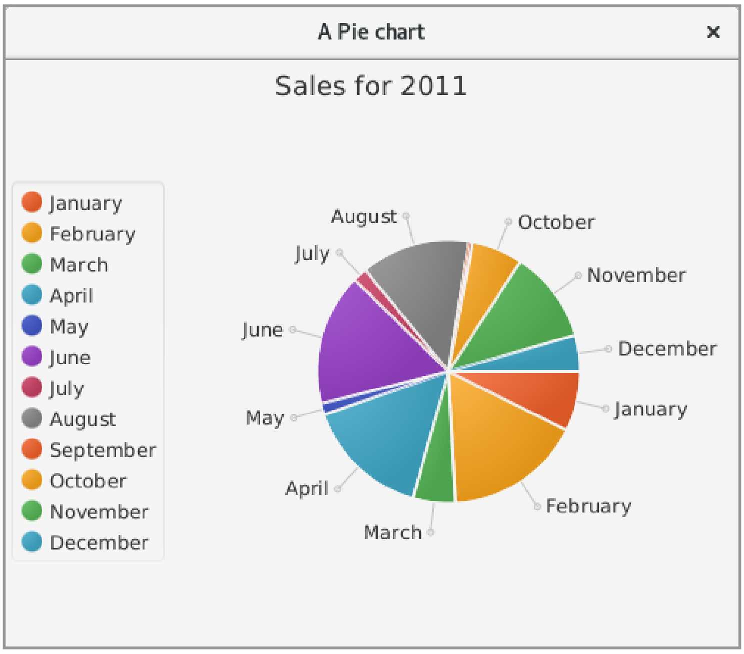
TURN TO THE EXPERTS FOR
SUBSCRIPTION CONSULTANCY

Subscrybe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develope acquisition and retention strategies.

Learn more at linkedin.com/company/subscrybe or contact
Managing Director Morten Suhr Hansen at mha@subscrybe.dk

SUBSCRYBE - to the future

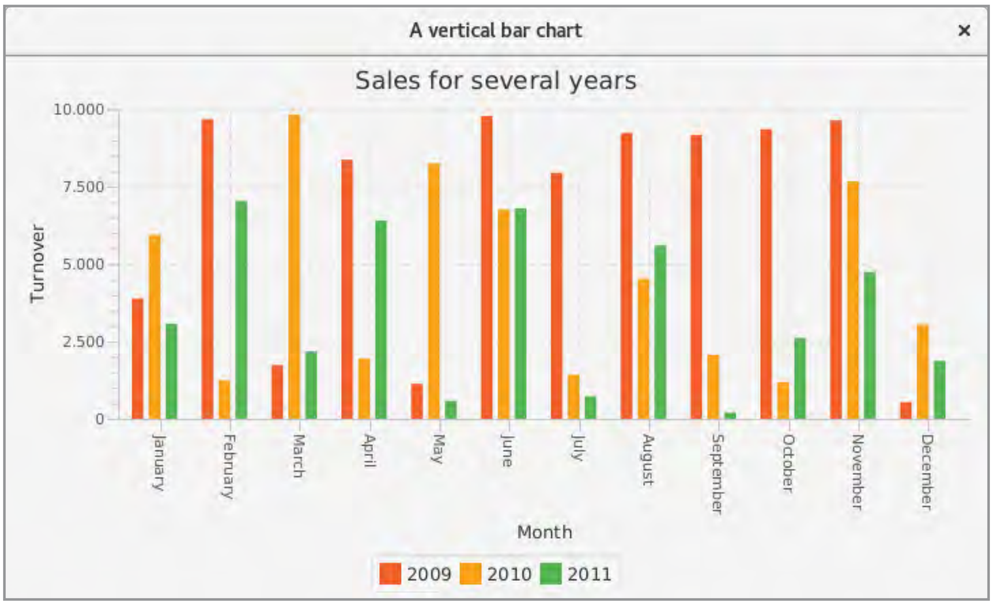

A Pie chart
×
Sales for 2011
January
February
March
April
May
June
July
August
September
October
November
December
August
October
July
November
June
December
January
May
April
February
March

The graph uses as default up to 8 colors, after which they are repeated, but you can specify more colors if you wish. The window's code is as follows:

```
public class PieView
{
 public PieView(Window owner, int year, ObservableList<PieChart.Data> data)
 {
  Stage stage = new Stage();
  stage.initOwner(owner);
  stage.initModality(Modality.APPLICATION_MODAL);
  PieChart chart = new PieChart();
  chart.setTitle("Sales for " + year);
  chart.setLegendSide(Side.LEFT);
  chart.setData(data);
  addTooltips(chart);
  StackPane root = new StackPane(chart);
  Scene scene = new Scene(root);
  stage.setScene(scene);
  stage.setTitle("A Pie chart");
  stage.show();
 }

 private void addTooltips(PieChart chart)
 {
  double sum = 0;
  for (PieChart.Data data : chart.getData()) sum += data.getPieValue();
  for (PieChart.Data data : chart.getData())
  {
   double value = data.getPieValue();
   Tooltip tip = new Tooltip(data.getName() + "= " +
    String.format("%.2f", value) + " (" +
    String.format("%.2f", value / sum * 100) + "%)");
   tip.setStyle("-fx-background-color: yellow; -fx-text-fill: black;");
   Tooltip.install(data.getNode(), tip);
  }
 }
}
```

and there is not much to explain. The hardest thing is actually the last method that links tooltips to the *PieChart* component. You should note how to create a *PieChart* component and specifically what properties are defined.

If in the main window you marks the numbers for three years and click on the link *A vertical bar chart*, you get the following window:


A vertical bar chart
×
Sales for several years
Turnover
10.000
7.500
5.000
2.500
0
January
February
March
April
May
June
July
August
September
October
November
December
Month
2009
2010
2011

which is an example of an XYChart. The event handler is as follows:

```
private void bar1(ActionEvent e)
{
 ObservableList<Integer> indices = table.
 getSelectionModel().getSelectedIndices();
 if (indices.size() > 0)
 {
  ObservableList<XYChart.Series<String, Number>> data = getXYData(indices);
  new VBarView(parent, data);
 }
}
```

First, the indexes are determined on the rows that are selected, and on that basis, the graph data is created as an object of the type *ObservableList<XHChart.Series<String, Number>>* to be used to display the graph. The object is determined using the following method:

```
private ObservableList<XYChart.Series<String, Number>>
 getXYData(ObservableList<Integer> indices)
{
 ObservableList<XYChart.Series<String, Number>> data =
  FXCollections.<XYChart.Series<String, Number>>observableArrayList();
 for (int i = 0; i < indices.size(); ++i)
 {
  Year year = model.getYears().get(indices.get(i));
  XYChart.Series<String, Number> series = new XYChart.Series<>();
  series.setName("" + year.getYear());
  for (int j = 0; j < DataModel.names.length; ++j)
   series.getData().add(new XYChart.Data(DataModel.
   names[j], year.getValue(j)));
  data.add(series);
 }
 return data;
}
```

What do you want to do?

No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!

VOLVO
AB Volvo (publ)
www.volvogroup.com

Volvo Trucks | Renault Trucks | Mack Trucks | Volvo Buses | Volvo Construction Equipment | Volvo Penta | Volvo Aero | Volvo IT
Volvo Financial Services | Volvo 3P | Volvo Powertrain | Volvo Parts | Volvo Technology | Volvo Logistics | Business Area Asia

The method is, in principle, simple, but you should note how the graph's data model is built up as objects of the type *XYChart.Series<String, Number>*, each object being a pair consisting of a name and a value.

The code for the graph window is as follows:

```
public class VBarView
{
 public VBarView(Window owner, ObservableList<XYChart.
 Series<String,Number>> data)
 {
  Stage stage = new Stage();
  stage.initOwner(owner);
  stage.initModality(Modality.APPLICATION_MODAL);
  CategoryAxis xAxis = new CategoryAxis();
  xAxis.setLabel("Month");
  NumberAxis yAxis = new NumberAxis();
  yAxis.setLabel("Turnover");
  BarChart<String, Number> chart = new BarChart<>(xAxis, yAxis);
  chart.setTitle("Sales for several years");
  chart.setData(data);
  StackPane root = new StackPane(chart);
  Scene scene = new Scene(root);
  stage.setScene(scene);
  stage.setTitle("A vertical bar chart");
  stage.show();
 }
}
```

Here you should note how to create the axes and how to create the *Chart* object itself as a node. In fact, it is simple to display a *Chart* as a *Node* in a scene graph once you have created the data model. As mentioned in the introduction to this chapter, you should investigate what other options exist for a *BarChart*.

The program has two other links, the first showing a horizontal bar chart, while the latter shows a stacked bar chart. As for the latter, it is done in the same way as above, and the data model is created in the same way, just the type is *StackedBarChart*. As for the first one, it is a *BarChart* and the only difference is that the two axes need to be replaced, which in turn means that the model data must also swapped. I do not want to show the code for the last two graphs here, as they are, in principle, identical to the code for the first *BarChart*.

# 10 FINAL EXAMPLE

The final example of this book is a program where the user can enter an expression for a mathematical function in one real variable, for example

$$y = 3x^2 - 2x + 1$$

and the program must then be able to draw the function's graph. The requirements are formulated continuously as the program is to be developed through a form of protyping, but the main purpose of the program is that when you have drawn the graph for one or more functions in a coordinate system, you must be able to copy the graph to the clipboard and paste it into a word processor, for example. You can thus think of the program as a tool that can be used by a writer of mathematical notes or books, but perhaps even by students in daily mathematics education.

Regarding the other features of the program, there are primarily requirements for flexibility and ease of use, as well as the types of graphs that the program can draw. Some of the program's features will relate to settings, including, for example, color and line thickness settings, as well as settings for the coordinate system will be important.

More challenges are expected regarding the development, and the biggest challenge is assumed to relate to performance, as it may take time to draw a complete graph. In particular, expectations can be expected to change the size of the window, as it means both adjustments of (redrawing) the graph itself, but also the coordinate system.

Another challenge is how to save the graph to a file where the graphs are not saved by simple object serialization as it is sensitive to new versions of the program.

Regarding what a function must be (legal expressions), it is a problem that I have previously solved, so for parsing and evaluating an expression, more or less direct code can be used, which has previously been developed and tested.

The program is considered as a first version, and you must expect frequent changes/extensions of the program for a period of time.

## 10.1 DEVELOPMENT

As mentioned above, the program is to be developed through a form of proptotype where the following iterations are planned:

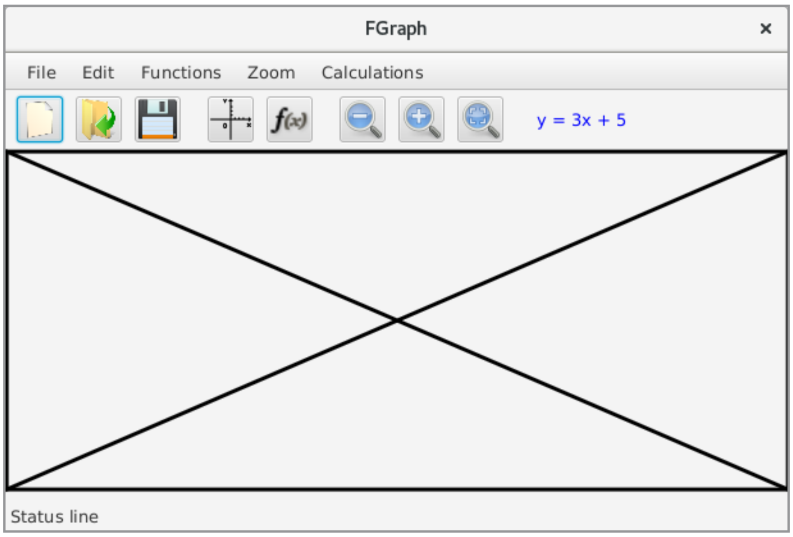
1. A simple prototype
2. Drawing the x-axis
3. Drawing the y-axis
4. Drawing the coordinat system
5. Drawing a function from a formal
6. Drawing a plot
7. Implementing other features
8. Implementing options
9. Styling the program
10. Refactoring

but there may occur several and other iterations. The result of every iteration is a complete program that can be used with the features that are implemenrated and each iteration is an extension of the previous iteration.

gaiteye®
Challenge the way we run
EXPERIENCE THE POWER OF
FULL ENGAGEMENT...
RUN FASTER.
RUN LONGER..
RUN EASIER...
READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM

## 10.2 A SIMPLE PROTOTYPE

The program's first prototype opens the following window:



The program's functions are defined as menu items, but for the time being, are only shown headings which all are menus that near the first are empty. The first menu defines 5 functions (as an example) but has no action. 8 functions have shortcuts in a toolbar and are from the left:

1. New drawing
2. Open existing drawing
3. Save drawing
4. Settings for coordinate system
5. Insert new function in the drawing
6. Zoom out
7. Zoom in
8. Same division om both axes

The toolbar also contains an expression, and the idea is that the drawen functions should be displayed as a link in the toolbar, so you can change the function's data by clicking the link.

The cross with the frame around should illustrate a drawing drawn by a method *showImage()*, and the figure consists of three *Shape* objects attached to a *Group* object. It is a preliminary solution that may be changed later. The figure is preliminary hard-coded, but follows the window size when it changes. There is a small problem with displaying the window in full screen if you double-click the titlebar. Here's a problem getting the window drawn correctly. It is apparently a JavaFX issue (under Linux) and is fixed with a timer that updates the window after ½ selund.

## 10.3 DRAWING THE AXES

This iteration involves drawing both axes of the coordinate system and thus the two next iterations are combined into a single iteration. Generally, there are many challenges in drawing the axes of the coordinate system, among other things because they have to behave nicely when the size of the window changes, but JavaFX has in the *Chart API* classes representing axes to a coordinate system, and especially the class *NumberAxis*. Since this class is actually general (has enough settings), it has been decided to use it instead of to write a custom component. Therefore, the two iterations to the coordinate system's axes are combined into a single iteration.

A single model class has been added that represents an axis:

```
public class AxisModel
{
 private DoubleProperty min = new SimpleDoubleProperty(-10);
 private DoubleProperty max = new SimpleDoubleProperty(10);
 private IntegerProperty dec = new SimpleIntegerProperty(0);
 private IntegerProperty ticks = new SimpleIntegerProperty(1);
 private StringProperty label = new SimpleStringProperty("");
 private DoubleProperty cross = new SimpleDoubleProperty(0);
 private BooleanProperty autoTicks = new SimpleBooleanProperty(false);
 private BooleanProperty showTicks = new SimpleBooleanProperty(true);
 private BooleanProperty showGitter = new SimpleBooleanProperty(false);
 private BooleanProperty showNumbers = new SimpleBooleanProperty(true);
```

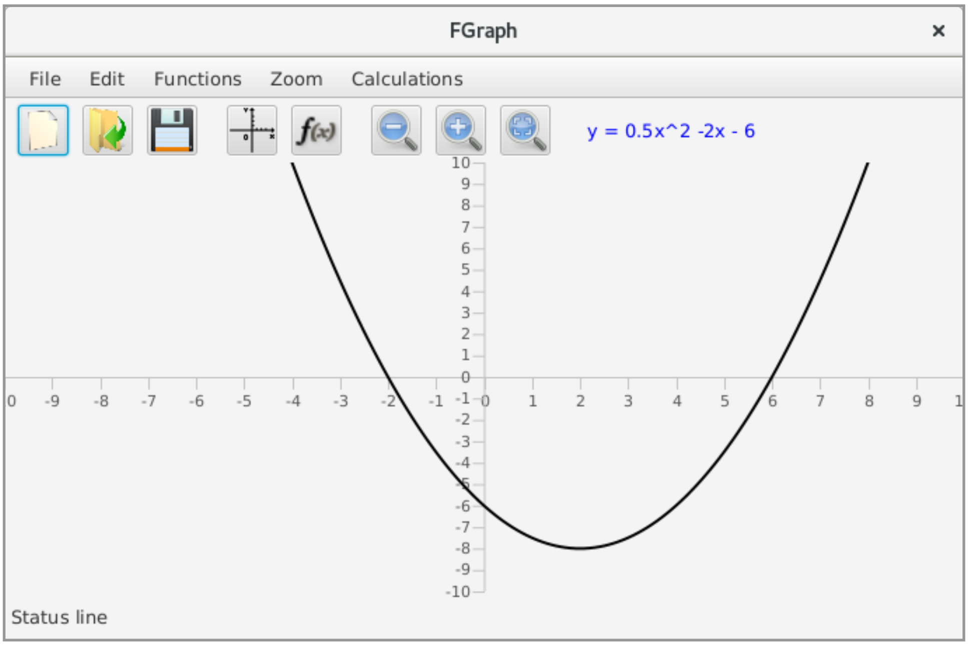
where the meaning is as follows:

- the lowest value on the axis
- the greatest value on the axis
- number of decimals to be used to display the axis numerical values
- the interval length between the points of the axis
- label (name) to the axis
- where the axis should cross the second axis
- if the axis automatically determines the division from the smallest and greatest value
- if the axis divisions are to be displayed
- if grid lines should appears
- if the numbers are to be displayed

The class *FDrawer* has two objects for the axes:

```
private AxisModel xModel = new AxisModel();
private AxisModel yModel = new AxisModel();
```

In addition, the class is expanded with two inner classes, one of which creates and drawes the axes, while the other draws the graph as a *Path* object. These two classes will be key and will be expanded several times. If you run the program, you get the following window where the function is still hard-coded:


FGraph
×
File
Edit
Functions
Zoom
Calculations
f(x)
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
Status line

y = 0.5x^2 -2x - 6

This e-book
*is made with*
**SetaPDF**

SETASIGN

**PDF** components for **PHP** developers

www.setasign.com

The important thing is that the coordinate system and the graph follows the window size. There is still a lack of a part about the coordinate system, including the use of above settings, which is only partly used until now. It is the subject of the next iteration.

## 10.4 SETTINGS FOR THE COORDINATE SYSTEM

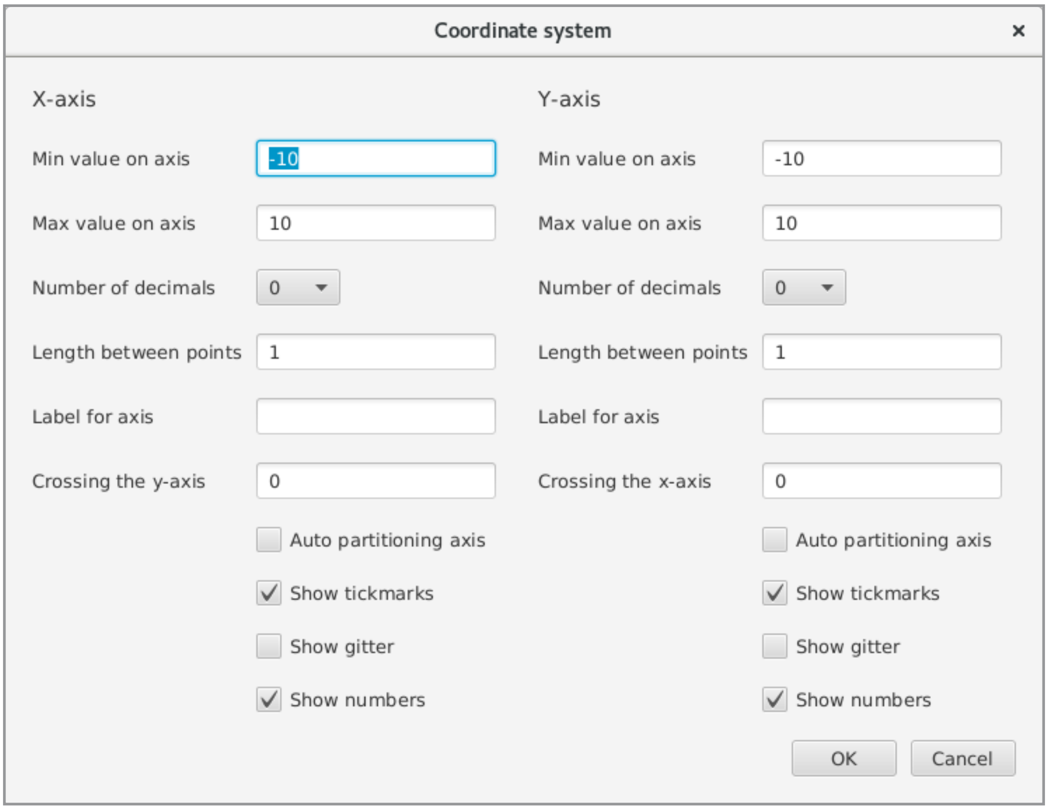
In this iteration I started with a little architecture. In the previous iteration, I have added a package *fdrawer.models*, which contains the class *AxisModel*. I have expanded the model with the following class:

```
package fdrawer.models;

public class AxesModel
{
 private AxisModel xaxis = new AxisModel();
 private AxisModel yaxis = new AxisModel();

 public AxesModel()
 {
 }

 public AxesModel(AxesModel axes)
 {
  xaxis = new AxisModel(axes.getXaxis());
  yaxis = new AxisModel(axes.getYaxis());
 }

 public AxisModel getXaxis()
 {
  return xaxis;
 }

 public AxisModel getYaxis()
 {
  return yaxis;
 }
}
```

which is nothing but an enclosure of two axes. In addition, I have added the following class

```
package fdrawer.models;

public class Model
{
 private AxesModel axes = new AxesModel();

 public AxesModel getAxes()
 {
  return axes;
 }
}
```

that for the time being, it is trivial, but in the future it should be the main model of the program. The main thing in the iteration is the following dialog box that is used to set the options for the axis of the coordinate system:



Coordinate system

X-axis

Min value on axis -10
Max value on axis 10
Number of decimals 0
Length between points 1
Label for axis
Crossing the y-axis 0
☐ Auto partitioning axis
☑ Show tickmarks
☐ Show gitter
☑ Show numbers

Y-axis

Min value on axis -10
Max value on axis 10
Number of decimals 0
Length between points 1
Label for axis
Crossing the x-axis 0
☐ Auto partitioning axis
☑ Show tickmarks
☐ Show gitter
☑ Show numbers

It is a relatively complex dialog box that basically consists of a *HBox* with two *GridPane* nodes. The code is placed in a package *fdrawer.views* and consists of two classes *AxesPresenter* and *AxesView*.

In addition to the dialog, the main window must be updated so that the current settings are also used, which is a relatively large work and relates to the two inner classes Axes and Plot, but after that the coordinate system is also essentially complete. However, there are two options that are not implemented:

1. *Number of decimals*, as JavaFX itself decides the number of decimals, and since it seems sensible, the function may later be removed.
2. *Auto division axis*. This feature is implemented by the *NumberAxis* class, but as this does not always seem appropriate, implementation of this feature is postponed to later, and perhaps the feature should be removed.

To open the dialog, an event handler has been added to the button in the toolbar, but the same event handler will later be added to the menu.

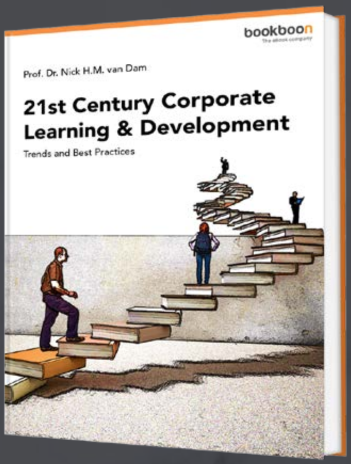
## 10.5 DRAWING A FUNCTION FROM A FORMAL

This iteration can be regarded as the most important iteration, and after the iteration is completed, you have in principle a ready-made drawer, where you can draw a typical

Free eBook on
Learning & Development

By the Chief Learning Officer of McKinsey

Download Now



bookboon
Prof. Dr. Nick H.M. van Dam
21st Century Corporate Learning & Development
Trends and Best Practices

mathematical function. The starting point is to expand the model with two classes *Expression* and *Tokens*, which represents a mathematical expression. The classes are implemented in the final example of the book Java 3 and support a mathematical expression where you can use the four arithmetical operations and the following functions:

| | | | | | |
|---|---|---|---|---|---|
| Sin | Cos | Tan | Cot | Ln | Log |
| Asin | Acos | Atan | Acot | Exp | Alog |
| Sqr | Sqrt | Pow | Root | Abs | Floor |

$$0.25x^3 + 0.75x^2 - 1.5x - 2$$

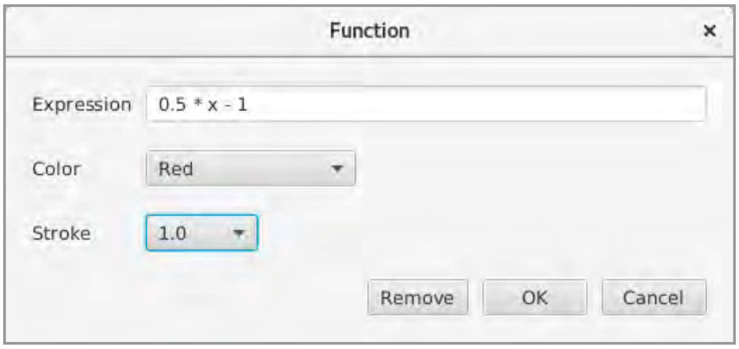
A function depends on one variable, and for example, the function

can be entered as

```
0.25 * pow(x, 3) + 0.75 * sqr(x) – 1.5 * x– 2
```

The classes *Expression* and *Tokens* from Java 3 can almost be used unchanged, but they support the use of multiple variables, and since it should not be an option in this program, the classes are modified so that you can only use a single variable called *x*.

In order to create a function, the program is expanded with the following dialog box (the classes *ExpressionView* og *ExpressionPresenter*):



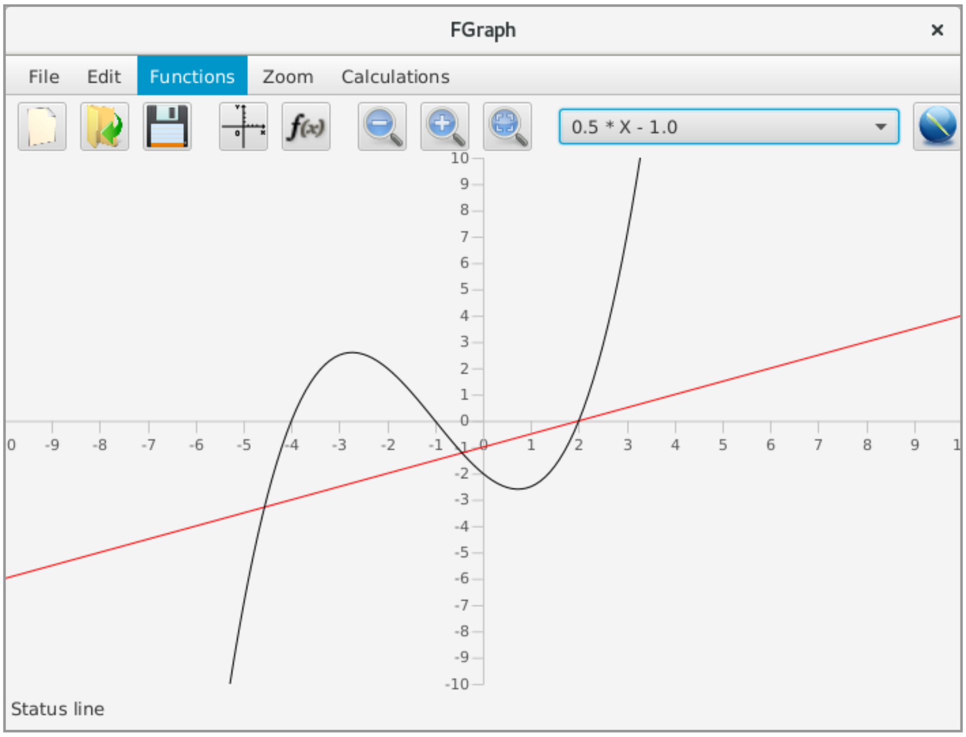
Function

Expression: 0.5 * x - 1

Color: Red

Stroke: 1.0

where you can enter a function (above a simple linear function). You can also specify the color based on a few fixed values, and in the same way you can specify line thickness (also based on a few fixed values). Below is the program window, where two functions are drawn:



Since you need to be able to draw more functions in the same coordinate system (in practice a few), the toolbar is changed where a combobox is added that shows the graph's functions. Additionally, a button has been added that opens the above dialog but for the function that is selected in the combobox.

Then the program is able to display the graphs for one or more functions in a coordinate system.

## 10.6 THE PROGRAM ARCHITECTURE

The result of the foregoing is that a very large part of the program code is located in the class *FDrawer*, and the subsequent development will, with unchanged architecture, mean that the class becomes even bigger and probably even much larger. This means that the class becomes irreversible, and partly because the program does not follow the MVP pattern, and both can make it more difficult to maintain the program in the future. That's why I've put in an extra iteration with the purpose of breaking down the class *FDrawer* into smaller classes and thus ensuring a better program architecture. It is therefore an iteration that does not directly generate momentum in the project.

It would of course be better to have made these considerations earlier in the project and from the outset have planned a better program architecture. However, it is not always that easy, and especially if you work on a program where the goal is not completely clear and where the requirements are determined ongoing. This may mean that you initially focus on getting something on the screen, so partial results can be presented to the future users to clarify the final requirements. The result will often be, as in the present case, a program with inappropriate architecture, and you should then stop and perform a first refactoring.

In this case, the following classes have been added to *fdrawer.views*:

- *Axes*, like the corresponding inner class in *FDrawer*, has been moved to its own view class
- *Plot*, like the corresponding inner class in *FDrawer*, has been moved to its own view class
- *MainMenu*, which is the code of the menu that has been moved to its own class primarily to increase the clarity
- *MainView*, which is the main window itself
- *MainPresenter*, that is presenter for the main window

The class *FDrawer* has been changed accordingly and is then a class that does nothing but instantiate a *Model* and a *MainPresenter*.

Finally, to *fdrawer.views* is added a class *ViewTools* that contains only static methods.

www.sylvania.com

We do not reinvent the wheel we reinvent light.

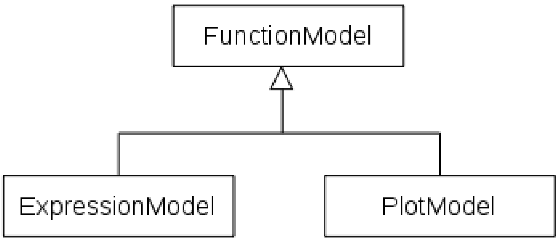
Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM SYLVANIA

## 10.7 DRAWING A PLOT

By a plot I want to understand a function defined by a number of points. The program must then display the function graph as a corresponding number of points in the coordinate system and possibly associated with lines. The coordinate system must thus be able to display two types of graphs, which are either a common function in one real variable as already implemented and a plot graph implemented in this iteration. So far, a function has been represented by the model class *ExpressionModel*, but in order to handle both types of functions in the same way, the model is expanded as follows:



The class *FunctionModel* must assign an ID to the individual functions and will only keep track of the color and the line thickness:

```
public abstract class FunctionModel
{
 private static int lastId = 0;
 private int id;
 private ObjectProperty<ColorWrapper> color =
  new SimpleObjectProperty(Colors.list.get(0));
 private ObjectProperty<Double> stroke = new SimpleObjectProperty(1.0);

 public FunctionModel()
 {
  id = ++lastId;
 }
```

Note that the class is defined abstract so that it can not be instantiated. The class *ExpressionModel* is then trivial

```
public class ExpressionModel extends FunctionModel
{
 private Expression expression;
```

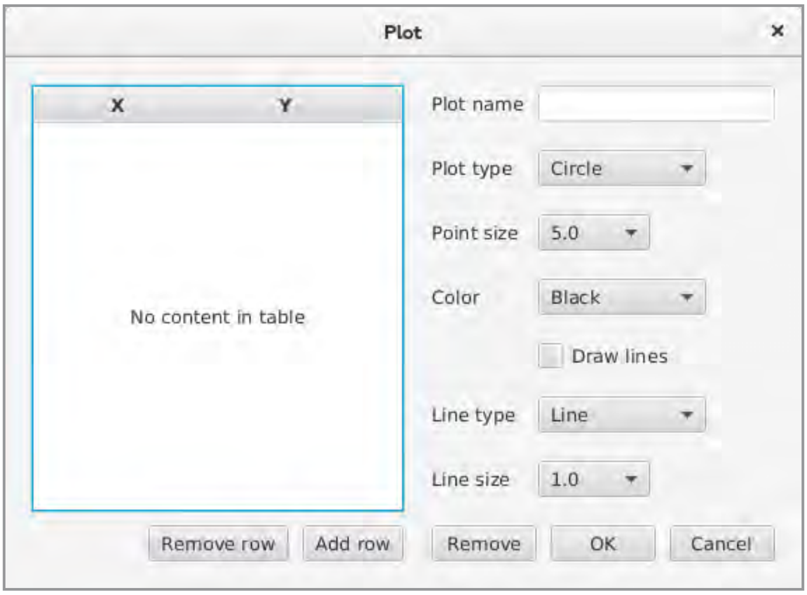
but should implement *equals()* and *toString()*. The class *PlotModel* requires a little more as there are more settings attached to a plot chart:

```
public class PlotModel extends FunctionModel
{
 private ObservableList<PlotPoint> points =
 FXCollections.observableArrayList();
 private StringProperty name = new SimpleStringProperty("");
 private StringProperty plotType =
  new SimpleStringProperty(PlotType.types.get(0));
 private ObjectProperty<Double> plotsize = new SimpleObjectProperty(5.0);
 private BooleanProperty drawLine = new SimpleBooleanProperty(false);
 private StringProperty lineType =
 new SimpleStringProperty(LineType.types.get(0));
```

This extension means some minor changes elsewhere in the code and, among other things, the model has been changed to the following:

```
package fdrawer.models;

import javafx.collections.*;

public class Model
{
 private AxesModel axes = new AxesModel();
 private ObservableList<FunctionModel> functions =
  FXCollections.observableArrayList();

 public AxesModel getAxes()
 {
  return axes;
 }

 public ObservableList<FunctionModel> getFunctions()
 {
  return functions;
 }
}
```

The iteration adds the following dialog that is used to define a plot graph:


Plot
×
X
Y
No content in table
Plot name
Plot type
Circle
Point size
5.0
Color
Black
☐ Draw lines
Line type
Line
Line size
1.0
Remove row
Add row
Remove
OK
Cancel

In the table on the left, you can enter points and the two buttons below the table are used to create a point and delete a point where the button *Remove row* deletes the row that is selected. The following type has been added to the model:

360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.

```
public class PlotPoint
{
 private DoubleProperty x = new SimpleDoubleProperty(0);
 private DoubleProperty y = new SimpleDoubleProperty(0);
```

which represents a point to a plot graph. To the right you can define options for the plot graph. A graph must have a name that can be used to identify the graph at the user interface. The type *PlotType* is used to specify the geometry of a point:

```
public class PlotType
{
 public static ObservableList<String> types =
 FXCollections.observableArrayList();

 static
 {
  types.addAll("Circle", "Square", "Rhombus", "Cross");
 }
}
```

while the next two comboboxes defines the size and color of the point. The following checkbox indicates whether the points should be joined by lines and, if applicable, there are two options:

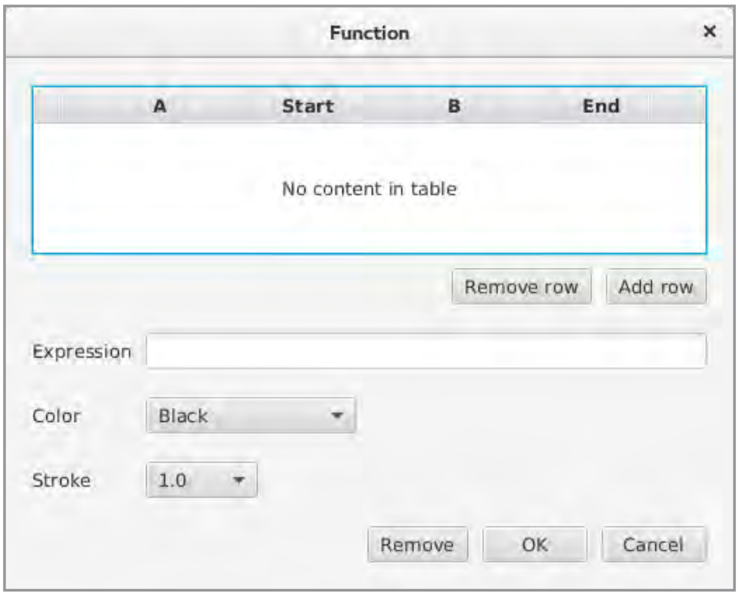
```
public class LineType
{
 public static ObservableList<String> types =
 FXCollections.observableArrayList();
 static
 {
  types.addAll("Line", "Curve");
 }
}
```

To implement the dialog, the view layer is expanded with two classes: *PlotView* and *PlotPresenter*. There was added a new button to the toolbar, but otherwise the biggest change is the extension of the class *Plot*, so it can now also draw plot graphs.

## 10.8 REFACTORING THE EXPRESSION DIALOG

The program has a few missings in drawing graphs, as the class *Plot* does not take into account the domain for a function. This may cause incorrect graphs or graphs that appear incorrect. I have therefore introduced an iteration solely for the purpose of solving this problem.

Basically, it is important to specify a function's domain, and the dialog *ExpressionView* is therefore expanded to:



It is now possible to add a number of intervals where the union of these intervals is the domain. If you do not enter any intervals, the *domain* is perceived as the real axis. For each interval you can enter 4 values:

1. *A* is the left end point. If the value is blank, the left end point is minus infinity.
2. *Start* indicating a mark for the starting point. If blank, no mark is displayed, but otherwise it may be *Close*, which means a filled circle or *Open*, which means an open circle.
3. *B* is the right end point. If the value is blank, the right end point is infinite.
4. *End* that indicates a mark for the end point. If blank, no mark is displayed, but otherwise it may be *Close*, which means a filled circle or *Open*, which means an open circle.

The extension means some changes to the model where there is a model class for an interval:

```
public class Interval
{
 private DoubleProperty a = new
  SimpleDoubleProperty(Double.NEGATIVE_INFINITY);
 private StringProperty lower = new SimpleStringProperty("");
 private DoubleProperty b = new SimpleDoubleProperty(Double.
 POSITIVE_INFINITY);
 private StringProperty upper = new SimpleStringProperty("");
```

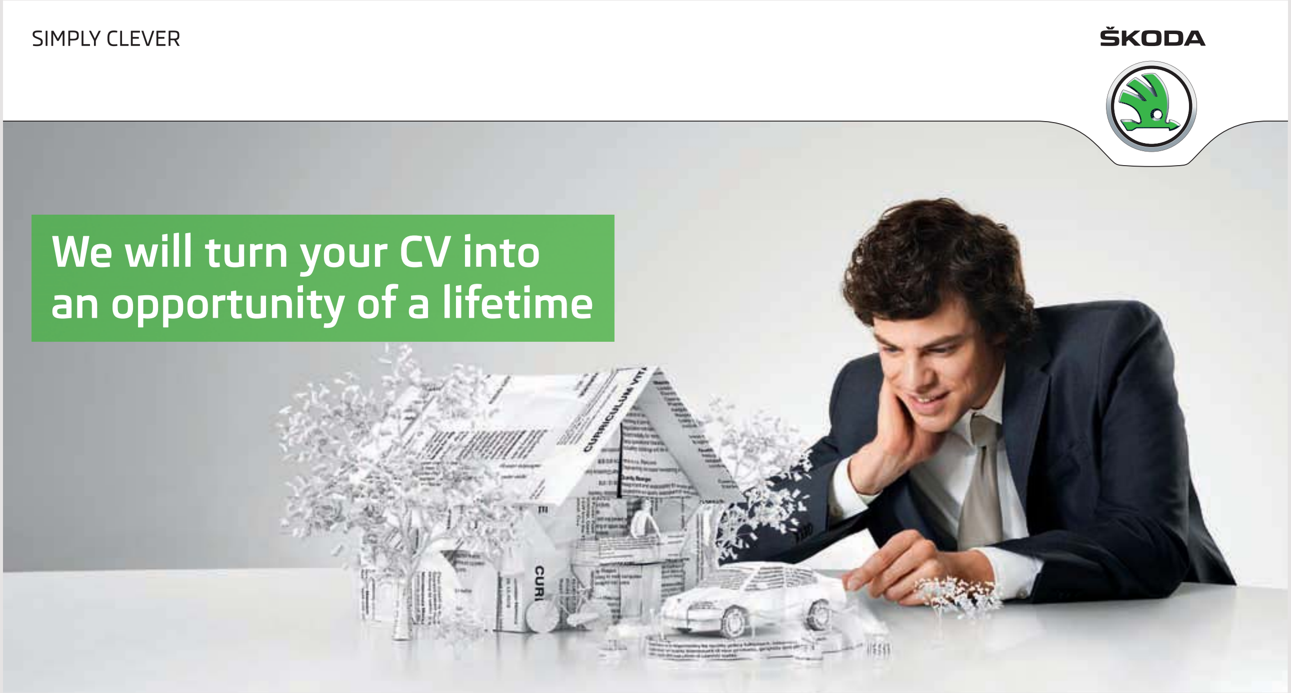
and the class *ExpressionModel* is also changed

```
public class ExpressionModel extends FunctionModel
{
 private Expression expression;
 private ObservableList<Interval> intervals =
 FXCollections.observableArrayList();
```

The most comprehensive changes in this iteration, however, are the class *Plot*, since it must take into account that a function can now have a domain, and eventually endpoints may be drawn. Finally, the class should behave nicely if you draw a graph that is not defined on the entire real axis, but fails to specify a domain. As a result, the class has gradually become relatively complex, and the class may eventually be included in a refactoring.

## 10.9 IMPLEMENTING THE FUNCTIONS MENU

The next iteration includes, according to the plan, to implement all functions in the menu. There are some, and some of them are even quite complex, so here I shared the iteration



SIMPLY CLEVER
ŠKODA
We will turn your CV into an opportunity of a lifetime

Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

into multiple iterations with one iteration for each menu. In this iteration I will look at the *Functions* menu that should include the following menu items:

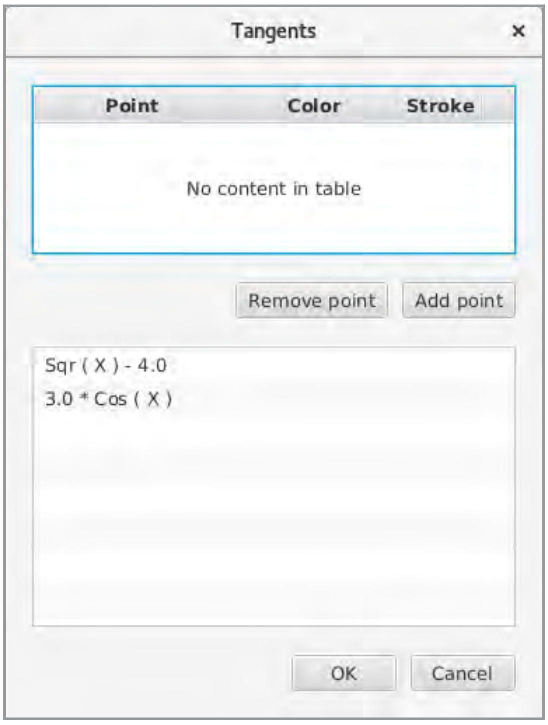
- *Create new function*
- *Create/modify tanget*
- *Create/modify vertical lines*
- *Insert/modify hatching*
- *Create point series*
- *Modify functions*
- *Modify point series*

The class *MainMenu* is updated accordingly. The class has an instance variable

```
Map<String, MenuItem> items = new HashMap();
```

with package visibility, where each menu item is identified by a key, and the *map* is used to allow in *MainPresenter* to assign event handlers to the individual menu items. The first and third last menu items corresponds to functions (create new function and create plot) that are already implemented as buttons in the toolbar, so nothing else than the class *MainPresenter* has to associate the right event handler.

The menu item *Create/modify tangent* is a new function, where it should be possible to draw a tangent to a function at a given point. If you select the function, you will receive the following window, which at the bottom shows a summary of the functions added to the drawing, and if you select one of these functions, you can add points for tangents in the upper table, as well as parameters for the color and thickness of the tangent:



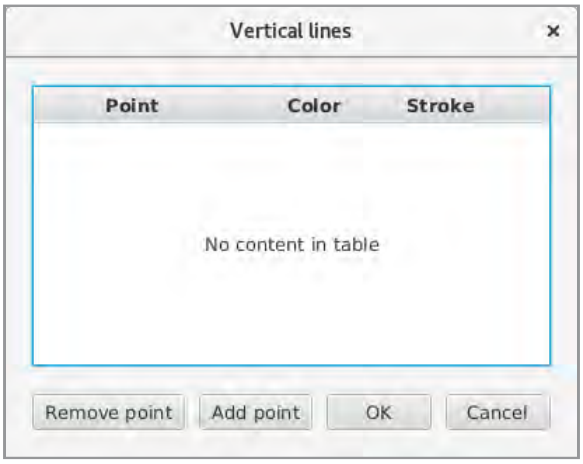
Sqr ( X ) - 4.0
3.0 * Cos ( X )

When you click OK, the appropriate tangents are drawn. The function requires an expansion of the class *ExpressionModel*, as a function can now have associated tangents:

```
public class ExpressionModel extends FunctionModel
{
 private Expression expression;
 private ObservableList<Interval> intervals =
 FXCollections.observableArrayList();
 private ObservableList<LineModel> tangents =
 FXCollections.observableArrayList();
```

Here, *LineModel* is a simple model class that extends the class *FunctionModel* with a single property for the tangent's foot point. In order to draw a tangent, the *Expression* class is expanded with a method that, as a limit value, can calculate the differential quotient in a point.

The menu item *Create / Modify vertical lines* should be used to inserts vertical lines into a drawing, for example, vertical asymptotes. The function opens the following window:



No content in table

where you can add points (x-values) where the vertical line has to cross the x-axis. The function is relatively simple, but requires an extension of the model:

```
public class Model
{
 private AxesModel axes = new AxesModel();
 private ObservableList<FunctionModel> functions =
  FXCollections.observableArrayList();
 private ObservableList<LineModel> lines = FXCollections.observableArrayList();
```

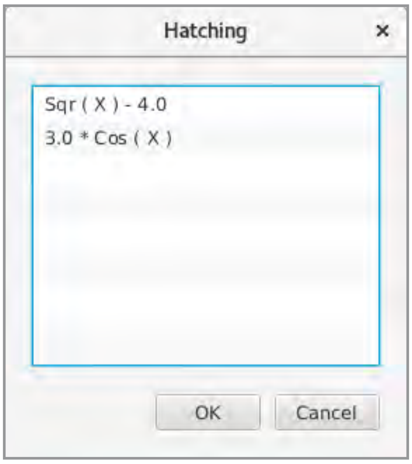
e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

**About e-Learning for Kids** Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

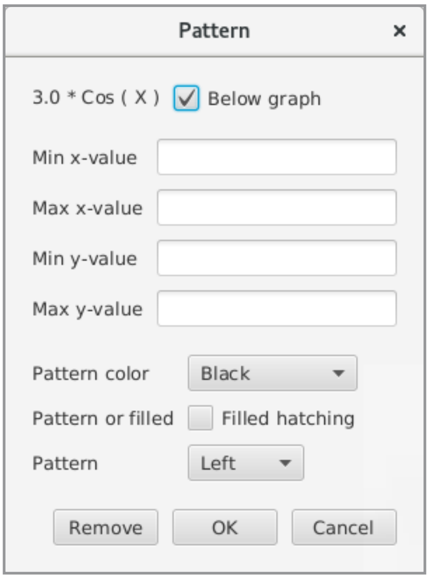
and similarly, the class *Plot* must be updated so that it draws the vertical lines.

The next feature *Insert/Modify hatching* allows you to insert hatching below or above a function and if you selects the menu item, you get the following window:



Sqr ( X ) - 4.0
3.0 * Cos ( X )

where to choose the function. If you click OK then you get the window:



3.0 * Cos ( X ) ☑ Below graph
Min x-value
Max x-value
Min y-value
Max y-value
Pattern color Black
Pattern or filled ☐ Filled hatching
Pattern Left

Here you can insert vertical and horizontal constraints on the area to be hatched, as well as selecting the color for the hatching, as well as whether it should be a standard fill or a pattern. The function is not quite simple to implement, and the class *Plot* should be significantly expanded. In addition, the model class *ExpressionModel* must be expanded with a new variable:

```
public class ExpressionModel extends FunctionModel
{
 private Expression expression;
 private HatchModel hatching;
 private ObservableList<Interval> intervals =
 FXCollections.observableArrayList();
 private ObservableList<LineModel> tangents =
 FXCollections.observableArrayList();
```

The function may possibly hatch an unintended area if restrictions are set on x and y and if the function's graph cross through the selected area. The reason is that the hatching area is defined as a *Path*, and it should be changed in a future version of the program.

The last two functions of the *Functions* menu are in principle implemented, as they correspond to edit a function or edit a plot already created. The only thing missing is a way to choose the function or plot to be edited, and it is done in the same way as above with hatching.

In connection with this iteration, the function for drawing a plot is expanded, so the line type to connect the points now can also can be a regression line.

## 10.10 IMPLEMENTING THE ZOOM MENU

The menu must have the following features:

- *Zoom out*
- *Zoom in*
- *Divide after x-axis*
- *Divide after y-axis*
- *Default*

Here, the third and fourth function means that the two axes determined by the division of the x-axis and the y-axis respectively, must use the same distance between the points on the axes. The last menu item must reset the axes to default. The toolbar initially had three buttons for zoom, but the latter is replaced by two other buttons, so there are now 4 buttons corresponding to the top 4 menu items.

In principle, it is quite simple to implement these functions and requires only simple changes in the two classes *AxisModel* and *AxesModel* and, of course, changes in the class *MainPresenter*, so that event handlers are associated.

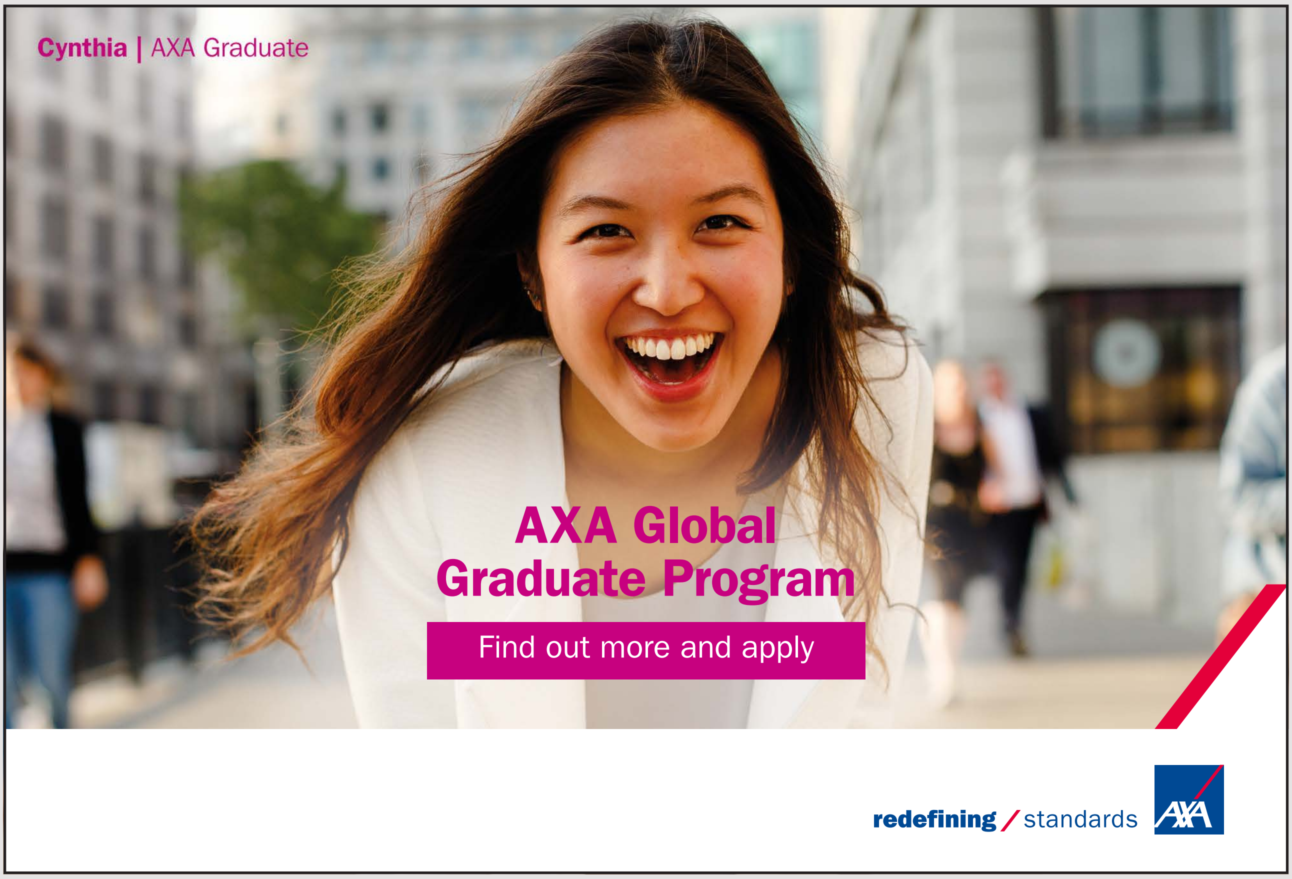
However, there is a single challenge to be solved. Until this place it is assumed that the two axes intersect at the center of the window. It does not work if it's possible to zoom, and more generally if you change the smallest and greatest values of the axes. It is necessary to change both in the *Axes* class and the class *Plot*, so they take into account that the coordinate system's origin is not necessarily centered in the window.

## 10.11 IMPLEMENTING THE EDIT MENU

This menu must have 4 menu items:

- *Undo*
- *Redo*
- *Screen clip*
- *Axes*

Here is the last the function for maintaining the coordinate system's axes and thus the same function implemented on one of the toolbar buttons. The *Screen clip* feature is one of the


Cynthia | AXA Graduate
AXA Global Graduate Program
Find out more and apply
redefining / standards
AXA

program's most important functions and is one of the goals of the entire application. The function, on the other hand, is simple to implement, as JavaFX offers it all:

```
private void clipGraph(ActionEvent e)
{
 WritableImage image = view.imagePane.
  snapshot(new SnapshotParameters(), null);
 ClipboardContent cc = new ClipboardContent();
 cc.putImage(image);
 Clipboard.getSystemClipboard().setContent(cc);
}
```

A Node – and here it is a *StackPane* – has a method *snapshot()* which simply takes a picture of the component. In fact, it is a PNG image and the image can be placed immediately on the clipboard.

Then there are the other two features, and on the other hand, it is not simple. For some reasons, JavaFX does not support Undo/Redo, but it must be assumed that it will be included in later versions of the API. One can find more open source APIs for Undo/Redo online, and it may be a good task to replace the following with one of these APIs. On the other hand, it is not entirely easy, partly because the APIs in question may not be comprehensive enough, and partly because they can be so general that it is in itself a piece of work to learn how they are used. In this case, I have solved the problem by programming the necessary from scratch. In fact, it is not particularly difficult, but the problem is of course that it is not modifiable and can not immediately be used in other programs.

In principle, Undo/Redo is about when you perform an operation in the program, it is simultaneously stored on a stack, and if you executes an Undo (keys Ctrl + Z), the operation must be rolled back and then placed on the redo stack . Redo (when keying Ctrl + Y) basically happens in the same way, just do not undo the operation but instead restore it and move it over to the undo stack. For so long it's all simple, but there are two outstanding problems:

1. What is an operation – what is it you should be able to undo and redo?
2. What should be pushed on the stack?

In this case, it has been decided that the operations to be rolled back are:

- Change the coordinate system's settings – an *AxesModel* object
- Add a new function – an *ExpressionModel* object
- Change a function – an *ExpressionModel* object
- Delete a function – an *ExpressionModel* object
- Add a new plot to the graph – an object of the type *PlotModel*

- Changing a plot – an object of the type *PlotModel*
- Delete a plot – an object of the type *PlotModel*
- Change a function's tangent – an *OberservableList object<LineModel>* object
- Changing the vertical lines of the graph – an *OberservableList object<LineModel>* object
- Add a hatching to a function – an object of the type *HatchModel*
- Change a function's hatching – an object of the type *HatchModel*
- Delete a function's hatching – an object of the type *HatchModel*

These possible operations (the operations to be saved for undo) are defined using the following enumeration:

```
public enum GraphOperations { AXES, ADDFUNC, MODFUNC, DELFUNC, ADDPLOT,
MODPLOT, DELPLOT, MODTANG, MODLINE, ADDHATCH, MODHATCH, DELHATCH }
```

An Undo / Redo operation is then defined as follows:

```
public class GraphOperation
{
 private GraphOperations operation;
 private Object value;

 public GraphOperation(GraphOperations operation, Object value)
 {
  this.operation = operation;
  this.value = value;
 }

 public GraphOperations getOperation()
 {
  return operation;
 }

 public Object getValue()
 {
  return value;
 }

 public void setValue(Object value)
 {
  this.value = value;
 }
}
```

and you can thus save any object, but with an indication of which operation the object is about. Back is the *UndoManager* class as the class that saves the operations and performs an *undo()* or a *redo()*:

```
public class UndoManager
{
 private static UndoManager instance;
 private Stack<GraphOperation> undoOperations = new Stack();
 private Stack<GraphOperation> redoOperations = new Stack();

 private UndoManager() {}

 public static UndoManager getInstance() { … }

 public void add(GraphOperation opr)
 {
  undoOperations.push(opr);
 }

 public void undo(Model model)
 {
  doOperation(model, undoOperations, redoOperations, true);
 }
```

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
**www.discovermitas.com**

**Month 16**
I was a construction supervisor in the North Sea advising and helping foremen solve problems

**Real work**
**International opportunities**
**Three work placements**

**MAERSK**

```
 public void redo(Model model)
 {
  doOperation(model, redoOperations, undoOperations, false);
 }

 private void doOperation(Model model, Stack<GraphOperation> stack1,
  Stack<GraphOperation> stack2, boolean undo)
 {
  if (!stack1.empty())
  {
   GraphOperation opr = stack1.pop();
   try
   {
    switch (opr.getOperation())
    {
     case AXES:
      AxesModel axes = model.getAxes();
      model.setAxes((AxesModel)opr.getValue());
      opr.setValue(axes);
      break;
      …
    }
    stack2.push(opr);
   }
   catch (Exception ex)
   {
   }
  }
 }
}
```

The class is written as a singleton, and the entire work is in the private method *doOperation()*, where there is a *case* entry for every possible operation.

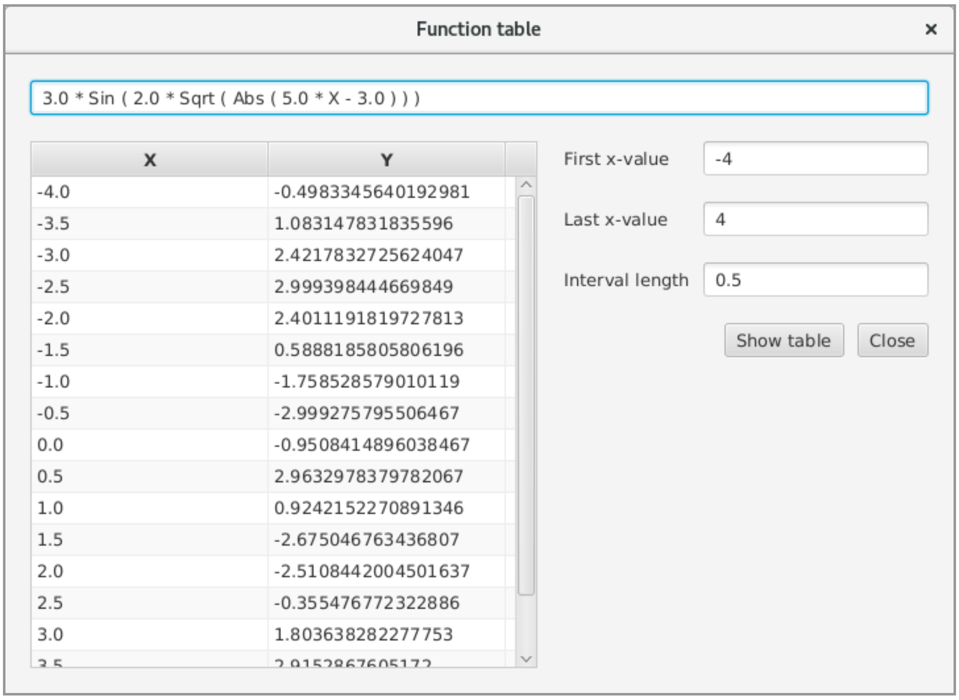
Each time the program performs an operation, it must be saved and thus the method *add()* above must be performed. It happens in the presenter classes of the individual dialog boxes, where copies of the individual operations are wrapped in *GraphOperation* objects that are then pushed on the undo stack. When you want to perform an undo or redo either by clicking the menu or by pressing Ctrl+Z or Ctrl+Y, the corresponding method is executed in the class *UndoManager*.

## 10.12 IMPLEMENTING THE CALCULATIONS MENU

It’s a very simple iteration as this menu only should have a single function:

- *Function table*

The idea is that the menu will later be expanded with other calculation functions. The current function opens a window for a particular function, where you can create a table of function values within an interval:



3.0 * Sin ( 2.0 * Sqrt ( Abs ( 5.0 * X - 3.0 ) ) )

| X | Y |
|---|---|
| -4.0 | -0.4983345640192981 |
| -3.5 | 1.083147831835596 |
| -3.0 | 2.4217832725624047 |
| -2.5 | 2.999398444669849 |
| -2.0 | 2.4011191819727813 |
| -1.5 | 0.5888185805806196 |
| -1.0 | -1.758528579010119 |
| -0.5 | -2.999275795506467 |
| 0.0 | -0.9508414896038467 |
| 0.5 | 2.9632978379782067 |
| 1.0 | 0.9242152270891346 |
| 1.5 | -2.675046763436807 |
| 2.0 | -2.5108442004501637 |
| 2.5 | -0.355476772322886 |
| 3.0 | 1.803638282277753 |
| 3.5 | 2.9152867605172 |

First x-value: -4
Last x-value: 4
Interval length: 0.5

Implementation of the function simply means writing the code for the above dialog box: *FuncTableView* and *FuncTablePresenter*, as well as for other functions in the menu, a simple dialog for selecting the function.

## 10.13 IMPLEMENTING THE FILE MENU

This iteration includes the last menu that must have 5 functions:

- *New*
- *Open*
- *Save*
- *Save as*
- *Exit*

Here is the last trivial, while the top three are also assigned to buttons in the toolbar. The goal is that it should be possible to save a drawing and later re-load the drawing. Since the program is a tool that should be easy to install, drawings must not be saved in a database, but in ordinary files. Here are four options:

1. to save drawings as JSON documents
2. to save drawings as XML documents
3. to save the drawings by conventional object serialization
4. to save drawings in a custom format

Here is the first the most obvious, as JSON does not fill up much, is a standard and is text and can therefore be read by everyone and thus other programs. It requires that a *Model* object can be converted to JSON, and that the JSON text loading again can create a Java object. There are open source APIs that can do that and examples are *GSON* and *Jackson*, which are both excellent products. Unfortunately, they do not work in the current case, and the reason is that it is a deep and complex object structure, but primarily that JavaFX is causing problems. Both products outline solutions that more or less directly solve the problems. It is apparently not quite simple, but one expects that future versions can be used directly. Because of that, I have abandoned using JSON.

ie business school

93%
OF MIM STUDENTS ARE
WORKING IN THEIR SECTOR 3 MONTHS
FOLLOWING GRADUATION

MASTER IN MANAGEMENT

- STUDY IN THE CENTER OF MADRID AND TAKE ADVANTAGE OF THE UNIQUE OPPORTUNITIES THAT THE CAPITAL OF SPAIN OFFERS
- PROPEL YOUR EDUCATION BY EARNING A DOUBLE DEGREE THAT BEST SUITS YOUR PROFESSIONAL GOALS
- STUDY A SEMESTER ABROAD AND BECOME A GLOBAL CITIZEN WITH THE BEYOND BORDERS EXPERIENCE

Length: **10 MONTHS**
Av. Experience: **1 YEAR**
Language: **ENGLISH / SPANISH**
Format: **FULL-TIME**
Intakes: **SEPT / FEB**

5 SPECIALIZATIONS
PERSONALIZE YOUR PROGRAM

#10 WORLDWIDE
MASTER IN MANAGEMENT
FINANCIAL TIMES

55 NATIONALITIES
IN CLASS

www.ie.edu/master-management | mim.admissions@ie.edu | Follow us on IE MIM Experience

As for XML, it's a bit the same, and the benefits are the same except that XML fills a lot. Also here are finished products that can be used, and *JAXB* is even an integral part of Java. If you often need to work with XML documents, it may be worth learning JAXB, and it would be a good job to replace the following with XML.

Storing drawings using object serialization is not appropriate in this case. Among other things, at the start of the project, I mentioned that it was not desirable, but in fact it also gives a problem to JavaFX, as properties and other JavaFX types are not serializable. At least, serialization of objects would require some extra work.

I will therefore choose the last of the above options and save drawings as text files, but in a custom format. In practice, it may sometimes be of interest, and it is also not very difficult, but in turn, there are a few significant disadvantages. Since it is a custom and non-transparent format, it is difficult to apply the files in other contexts than in the current application. The contents of the files do not follow any standard. That in itself will often mean that a method like the following is rejected. Secondly, a custom format like the following is not easy to maintain, and if the model is changed, the code to save drawings and read them again must be changed accordingly, which means it is very expensive to change the model. The question is then whether there are any benefits and that's hardly, but should I imply two, it should be that the format is very compact and saved drawings do not fill up much, which may be important if drawings should send over a network, and second as an example of what it takes to build an object hierarchy from a text (it is by default what parsing either JSON or XML does).

So to the specific solution. There are generally two things to do:

1. a *Model* object must be converted into a text which can be saved in a file
2. the content of a file (a saved drawing) must be split into values to be used to construct a *Model* object identical to the object that has been saved.

To solve these problems, it is necessary to use some separators that can be used to separate values, and the only requirement is that it should be characters that can not occur in the individual data elements. In this case, I need three, and I want to apply line breaks, carrige return and tabs, and thus \n, \r and \t. As the next step, each model class, whose objects are to be saved as part of the drawing, are expanded with a method *asText()*, which returns the object's data elements separated with the separators. As an example, the class *AxisModel* is expanded with the following method:

```
public String asText()
{
 StringBuilder builder = new StringBuilder();
 builder.append(getMin());
 builder.append('\t');
 builder.append(getMax());
 builder.append('\t');
 builder.append(getDec());
 builder.append('\t');
 builder.append(getTicks());
 builder.append('\t');
 builder.append(getLabel());
 builder.append('\t');
 builder.append(getCross());
 builder.append('\t');
 builder.append(isAutoTicks());
 builder.append('\t');
 builder.append(isShowTicks());
 builder.append('\t');
 builder.append(isShowGitter());
 builder.append('\t');
 builder.append(isShowNumbers());
 return builder.toString();
}
```

That is, the method returns all settings for a number axis as a string where the individual values are separated by the tab character. Similarly, the class *AxesModel* is expanded using the following method:

```
public String asText()
{
 return xaxis.asText() + '\r' + yaxis.asText();
}
```

which returns data for the x-axis and y-axis, respectively, separated by a carrige return. The same applies to the other model classes:

- *ColorWrapper*
- *PlotPoint*
- *PlotModel*
- *LineModel*
- *Interval*
- *HatchModel*
- *ExpressionModel*
- *Model*

where some of course are more complex than the above. As an example, below is shown *asText()* from the class *Model*, which returns the text to be saved:

```
public String asText()
{
 StringBuilder builder = new StringBuilder();
 builder.append(axes.asText());
 for (FunctionModel fm : functions)
 {
  builder.append('\n');
  builder.append(fm.asText());
 }
 if (lines.size() > 0)
 {
  for (LineModel lm : lines)
  {
   builder.append('\n');
   builder.append(lm.asText());
  }
 }
 return builder.toString();
}
```

“I studied English for 16 years but... ...I finally learned to speak it in just six lessons”
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

where the result technically is a text divided into lines. Then an event handler in *MainPresenter* can write the text of a file:

```
try (FileWriter writer = new FileWriter(filename))
{
 writer.write(model.asText());
}
catch (IOException ex)
{
 …
}
```

It is then easy to read the content of the file again:

```
try (FileReader reader = new FileReader(filename))
{
 char[] buffer = new char[2048];
 StringBuilder builder = new StringBuilder();
 for (int count = reader.read(buffer, 0, buffer.length); count != -1;
  count = reader.read(buffer, 0, buffer.length))
  builder.append(buffer, 0, count);
 if (!model.parse(builder.toString())) throw new Exception();
}
catch (Exception ex)
{
 …
}
```

where a string with the file's content is built, which is parsed by a method *parse()* in the class *Model*. It is this method that is responsible for splitting the content of the file into values and using these to initialize the *Model* object. The code of the method *parse()* is as follows:

```
public boolean parse(String text)
{
 try
 {
  String[] elems = text.split("\n");
  if (elems.length == 0) throw new Exception();
  AxesModel am = parseAxes(elems[0]);
  ObservableList<FunctionModel> functions =
  FXCollections.observableArrayList();
  ObservableList<LineModel> lines = FXCollections.observableArrayList();
  for (int i = 1; i < elems.length; ++i)
  {
   if (elems[i].charAt(0) == 'E') functions.add(parseExpression(elems[i]));
   else if (elems[i].charAt(0) == 'P') functions.add(parsePlot(elems[i]));
   else if (elems[i].charAt(0) == 'L') lines.add(parseLine(elems[i]));
  }
```

```
   this.axes = am;
   this.functions = functions;
   this.lines = lines;
   return true;
  }
  catch (Exception ex)
  {
  }
  return false;
}
```

but it uses several auxiliary methods. However, the above can give an impression of how the method works:

1. the string that is read in the file is split into lines (the separator character \n)
2. the first line is parsed to an *AxesModel* object
3. then iterates over the remaining lines (in principle, there only have to be the one for axes) and each element is parsed as either an *ExpressionModel*, a *PlotModel* or a *LineModel* object

If an exception occurs during the process, the parsing is interrupted and the method returns false, but if all goes well, the current *Model* object is initialized with the values parsed and the method returns true.

The Iteration includes a little more than what is shown above, but it is nothing but what is shown in other programs. You have to keep track of when a drawing has been changed, and before it should be deleted you should get a warning. It is a task left for the class *UndoHandler*, although the task because of cohesion probably not should be located there.

## 10.14 A FINAL ITERATION

According to the plan, there should now be two iterations left, but it has been merged into one. The program is in principle complete and can be used for what was the purpose. In the following, I will review the program's classes to remove inconvenience and adjust the user interface in some places. Below I have briefly mentioned the most important adjustments.

- A style sheet has been added to the application that primarily contains empty style classes that are used in the application's dialogs for later styling of the program.
- *MainView* has changed slightly where some icons have been replaced and the background color has been changed using a style. An icon has been added to the toolbar to copy a drawing to the clipboard, and tooltips have been added to all toolbar icons.

- *MainPresenter* has been updated in several areas. There are removed two event handlers (*axes()* and *func()*), which have been replaced by a call to the method *openDialog()*. This method was previously found as a static method in the class *ViewTools* and as a private method in *MainPresenter*. Here is the last removed, so the class *MainPresenter* now also uses the static method in *ViewTools*. Finally, all calls to *Alert* are replaced by static methods in the class *MessageDialogs*. The result is that unnecessary code has been removed.
- *AxesView* has two fields (*number of decimals* and *auto tickmarks*) on both axes that are not used. These four fields have been removed. Fields for entering numbers use a converter called *ValueConverter*. It's an inner class, but it is moved to a class with the package visibility in a file *Converters.java*. More dialogs uses converters, and the goal is to gather these converters somewhere.
- Three of the functions in the *Functions* menu and the function in *Calculations* menu opens a simple dialog box for selecting a function. The respective views are *FuncSelectView*, *FuncTableSelectView*, *HatchSelectView* and *TangentSelectView*. These four classes are actually the same and are replaced by a single view called *SelectView*. The corresponding presenter classes are also almost the same, but not entirely, and instead of replacing them with a new one, I have kept the old ones as they are, since they are small and quite simple.

Excellent Economics and Business programmes at:
university of groningen
AACSB ACCREDITED
"The perfect start of a successful, international career."
CLICK HERE
to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

- In general, I have been through all dialogs and assigned styles to all controls.
- The model classes are generally unchanged, but two of the classes use converters, and they are as in the same way as in the view layer moved into their own file called *Converters.java*. However, the classes *Model* and *UndoManager* are modified as follows.
- The class *UndoManager* actually encapsulates two concepts, namely undo/redo, and to keep track of whether the model has been changed. In principle, it is unfortunate, and the last functionality regarding the state of the model has therefore been moved to its own class called *ChangedManager*. The class is written as a singleton and is very simple.
- The class *Model* was expanded in the previous iteration with the method *parse()* used to parse the text read in a file. The disadvantage is that the class *Model* now fills a lot, and so I have moved the *parse()* method into its own class called *Parser*. The class has no other methods, and the only advantage is that the model class becomes simpler and more manageable, but it may also make it easier to replace how drawings are saved with something else.

After that, the program is complete and ready for use and as the last, I have written an installation script so that the program can be easily installed on the machine.

## 10.15 A LAST REMARK

As mentioned in the introduction to this chapter, the *FDrawer* program is developed by a form of prototyping. The term prototyping is sometimes used for a development as above, although the word is not correctly used. By the term prototyping one understands the development of a program, which is intended to illustrate the ideas of an application, and after the prototype has been developed and presented, the actual system development begins from the beginning, but not as a further development of the prototype. The prototype is itself a model that is not used as part of the final system development. It is therefore more correct to say that the program *FDrawer* is developed by a form of iterative system development through very small iterations, sometimes referred to as *sprints*.

Today, all just a bit bigger programs are developed through a number of iterations, and the difference is primarily how large these iterations are. In the present case they have been very small and it has its pros and cons. The advantage is that you often observe progress in the work and that the future users through presentations seeing frequently experience and something is happening and thus maintain the interest for the program. It also supports the fact that the requirements are determined continuously throughout the development period, and thus the situation that, from the start, it is not entirely clear where it all will end. It is also an advantage that you can quickly detect malicious decisions and unfortunate solutions and thus quickly correct before you reach too far in the development of a particular feature. A system development with many and very small iterations opens up a very close course with the upcoming users and helps to end with a solution that meets the users' requirements for the completed program. However, there are also disadvantages, and in particular there is a risk that you will always have to change the code that has been made. When you build on iteration on iteration, there is a great chance that you have to go back and change something that has been programmed in previous iterations, among other things because there are always new demands and wishes. That is precisely why traditional system development speaks of analysis and design, which defines the requirements for the finished program through the analysis, and under the design, it will be decided how to write the program before doing the programming itself.

Now, it's not a either or and you can perfectly combine iterative system development with more traditional system development and will even always do. Today, it has been acknowledged that by simply a slightly larger program it is impossible to carry out an analysis where all requirements are identified and determined. All experience shows that if you do it, it still does not meets the requirements. It's simply impossible to think about it all and there will certainly be new demands of requirements over the development period, and if there is no user interaction at the development, one can almost be sure that the end of a program has so many shortcomings and inconveniences that the user does not bother using it. Therefore, in iterative system development, an iteration is a system development in its own, but not necessarily as small iterations as the current example suggests. An iteration will generally have a limited amount in time, and must lead the program from one stable state to another. The result of an iteration is a milestone in system development, where the state of the program can be presented and assessed by the upcoming users or the customer and possibly lead to something being corrected and new requirements to be set. Typically, an iteration will be a system development process in itself, including analysis, design, programming and testing, but so that the weight changes over time, with the emphasis on analysis and design in the first iterations, but in later iterations the analysis and design fill less as the requirements are now established and the most important decisions regarding the design of the program are also in place.

There are (many) guidelines for how many iterations there should be, but the truth is that there are no golden rules. In general, experiments in system development, where the goals are not clear, and where you experience is gained along the way, speak for very short sprints, and perhaps something towards that illustrated in the current example, while tasks where the goals are clearer speaks for a little longer iterations involving traditional system development to ensure stable and secure progress in the development.