Java User Group

Zühlke Java FX Hands On Workshop

michael.inden@zuehlke.com

Hinweise

Auf dem USB-Stick gibt es ein Eclipse-Projekt mit Aufgaben. Nach dessen Import finden sich die nachfolgend aufgeführten Übungsaufgaben als eigene kleine Programme, die an den jeweiligen Stellen ergänzt werden müssen:

Beispiel

```
final CheckBox applyStyle = new CheckBox("Apply style");
 applyStyle.setOnAction(event -> {
     // TODO
 });
```

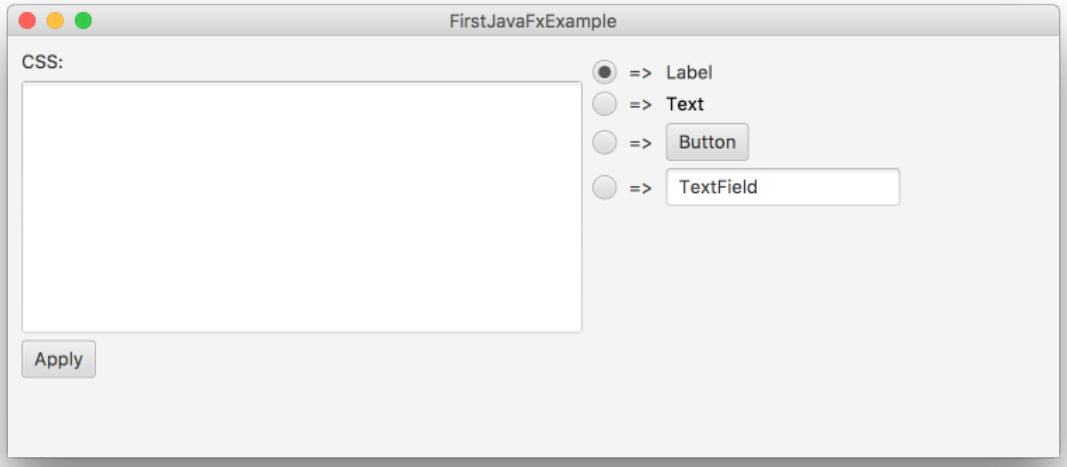
Gliederung der Aufgaben

Die Übungen untergliedern sich wie der Vortrag in 5 Teile, die sich jeweils in eigenen Packages wiederfinden:

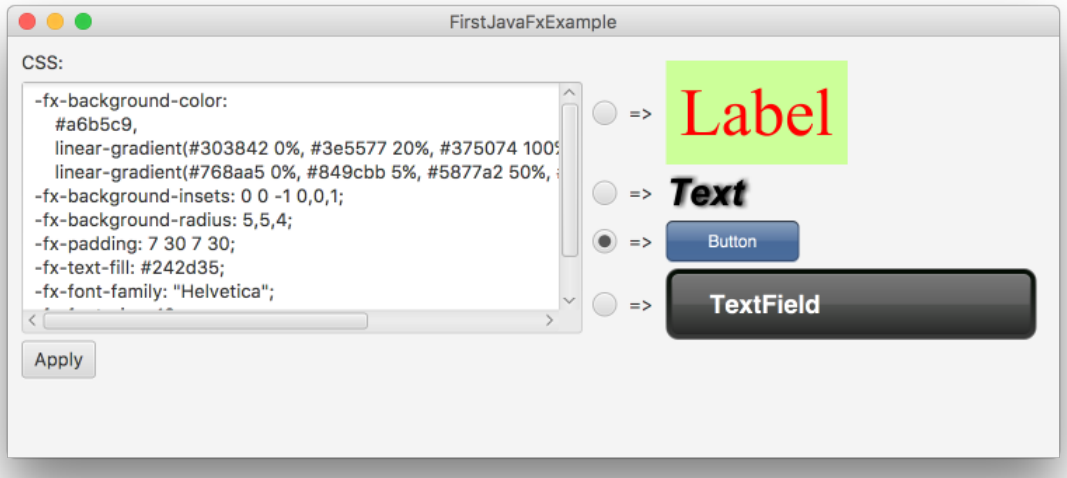
Teil	Package	Thema
1	`javafx.a_basics`	Grundlagen
2	`javafx_b_properties`	Properties, Binding, Obsverable Collections
3	`javafx.c_table`	Trees, Tables und TreeTables
4	`javafx.d_charts`	Diagramme mit Charts
5	`javafx.e_update40`	Die Neuerungen aus JavaFX 8 Update 40

Teil 1: Bedienelemente & Styling

Aufgabe 1a:
Erstelle eine einfache Applikation, die es erlaubt, verschiedene Bedienelemente mithilfe von CSS zu stylen. Beginne mit dem Grundgerüst einer Applikation in Form der Klasse `Application` und unterschiedlichen Bedienelementen, wie z. B. `Label`, `Text`, `Button`, `Textfield`, `TextArea`, `RadioButton` sowie Layomutcontainern wie `HBox`, `VBox` und `GridPane`. Das Ganze sollte in etwa wie folgt aussehen:



Aufgabe 1b:
Ergänze die Applikation, um ein Event Handling, sodass die rechten Bedienelemente mithilfe von CSS gestylt werden können und eine Steuerung über ein `Radiobutton` erfolgt.



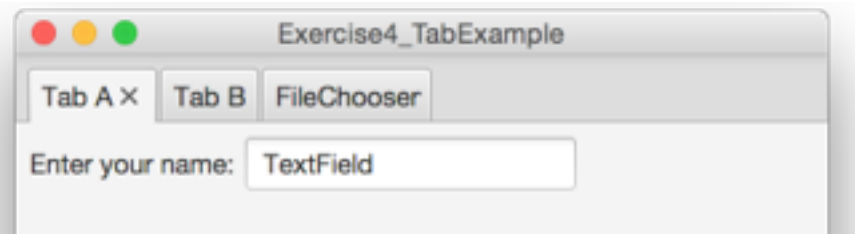
Tipp 1: Verwende explizites Styling per `setStyle()` und orientiere dich an den im Programmkommentar vorgegebenen CSS.

Tipp 2: Vermutlich hast du eine Auswertung der RadioButtons in etwa wie folgt gelöst:

```
if (selectedToggle == rb1)
{
      label.setStyle(cssStyle);
}
else if (selectedToggle == rb2)
{
      text.setStyle(cssStyle);
}
else if (selectedToggle == rb3)
{
      button.setStyle(cssStyle);
}
else if (selectedToggle == rb4)
{
      textfield.setStyle(cssStyle);
}
```

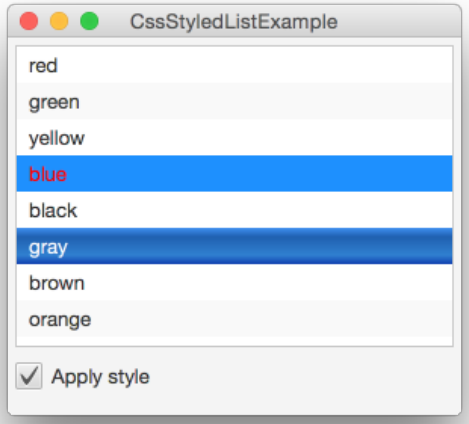
Das wirkt wenig elegant. Wie kann man es besser machen? Jedes Bedienelement besitzt in JavaFX ein spezielles UserData-Objekt. Zugriff darauf bieten die Methoden `setUserData(Object)` und `Object getUserData()`. Vereinfache damit die obige Lösung.

Aufgabe 2:
Erstelle eine Applikation, die verschiedene Bedienelemente in Tabs präsentiert. Unter anderem einen Button zum Öffnen eines Dateiauswahl-Dialogs sowie in einem Tab auch ein `Label` und ein `TextField`, dass die selbe Baseline besitzen.



Tipp: Zum Alignment besitzt die `Box` eine Methode `setAlignment()`.

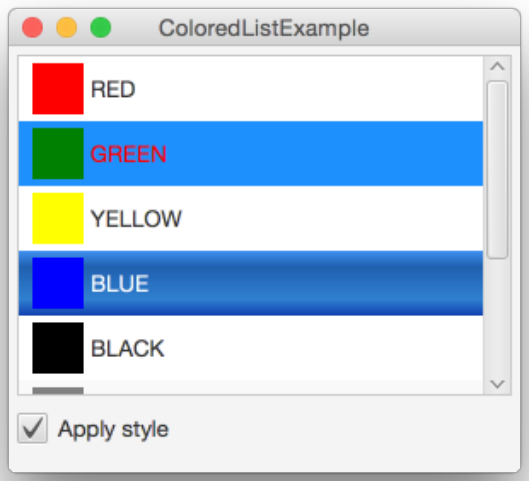
Aufgabe 3a:
Erstelle eine Applikation, die eine Auswahl aus einer Menge von Farben erlaubt. Für eine ansprechende optische Präsentation soll folgendes CSS aus der Datei `list.css` genutzt werden:



Tipp: Lade die Datei mit `getResource()`als `URL`, die mit `toExternalForm()` in einen String konvertiert wird. Diese Verweise können für die Scene mit `getStylesheets()` verwaltet werden.

Aufgabe 3b:
Werte die Applikation noch weiter optisch auf und ergänze dazu einen Cell-Renderer. Dessen Aufgabe ist es, in jeder Zeile zusätzlich zu dem Text ein farbiges Rechteck zu zeichnen. Nutze folgenden Sourcecode als Ausgangsbasis:

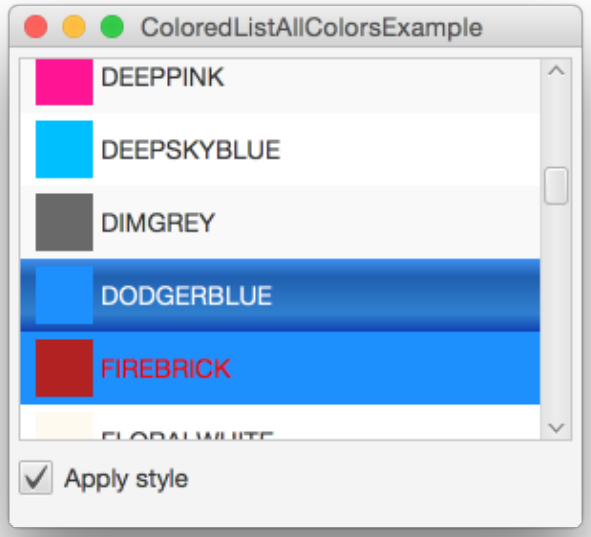
```
// "Renderer"
static class ColorRectCell extends ListCell<String>
{
    @Override
    public void updateItem(String item, boolean empty)
    {
        super.updateItem(item, empty);
        setText(item);
    }
}
```



Tipp: Nutze die Klasse `Rectangle` und die Methoden `setGraphic(Node)`. Farben verarbeitet man mit der Klasse `Color`. Diese bietet eine Methode `web()`.

Aufgabe 3c: KÜR & Programmier-Tricks

Ermittle alle von JavaFX durch die Klasse `Color` bereitgestellten Farben per Reflection und stelle diese in einer Liste sortiert nach Name der Farbe dar. Das Resultat sollte in etwa wie folgt aussehen:



Als Hilfestellung dient folgender Sourcecode zum Ermitteln der Farbinformationen:

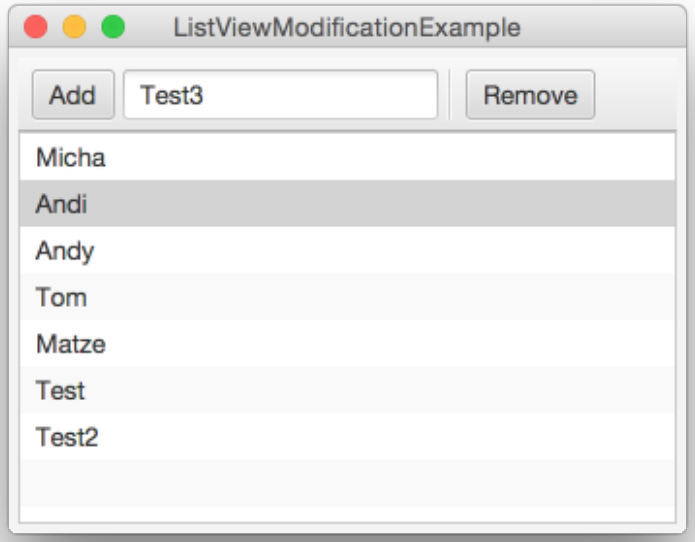
```
static Map<Color, String> fillColorLookUpMap()
{
    final Map<Color, String> colorLookUpMap = new HashMap<>();

    final Field[] fields = Color.class.getFields(); // only public
    for (final Field field : fields)
    {
        if (field.getType() == Color.class)
        {
            try
            {
                final Color color = (Color) field.get(null);
                final String colorName = field.getName();

                colorLookUpMap.put(color,  colorName);
            }
            catch (IllegalAccessException illegalAccessEx)
            {
                System.out.println("not allowed to access field '" +
                                   field.getName() + "'.");
            }
        }
    }

    return colorLookUpMap;
}
```

Auch benötigt man das Sortieren einer Map nach Wert. Das Ganze ist etwas trickreich und der Sourcecode dazu sieht wie folgt aus:

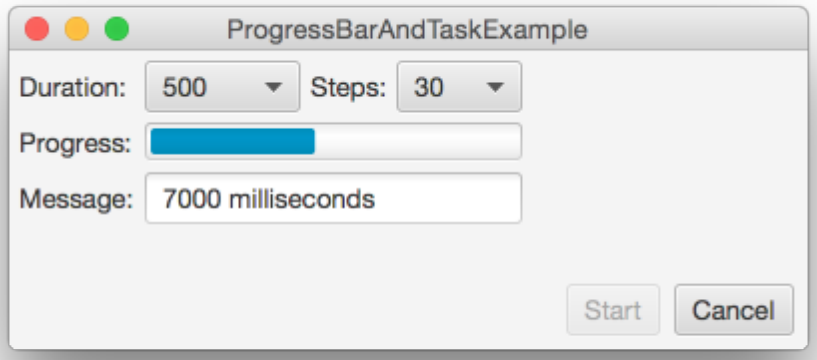
```
public static Map<Color, String> sortMapByValue(final Map<Color, String> map)
{
    class ValueComparator implements Comparator<Color>
    {
        public int compare(final Color first, final Color second)
        {
            final int result = map.get(first).compareTo(map.get(second));
            if (result == 0) // Der Wert 0 würde ungewünschte Ergebnisse liefern
            {
                return -1;
            }
            return result;
        }
    }

    // Sortierung
    final ValueComparator comparator = new ValueComparator();
    final TreeMap<Color, String> valueSortedMap = new TreeMap<>(comparator);
    valueSortedMap.putAll(map);

    // Bereitstellung als LinkedHashMap, die die Ordnung des Einfügens aufrecht erhält
    final LinkedHashMap<Color, String> valueSortedMap2 = new LinkedHashMap<>();
    valueSortedMap2.putAll(valueSortedMap);
    return valueSortedMap2;
}
```

Teil 2: Properties, Binding & Observable-List und Concurrency

Aufgabe 1:
Erstelle eine Listendarstellung von Namen, die das Hinzufügen und Löschen von Einträgen erlaubt und die den Zustand des Add- bzw. Remove-Buttons anpasst, sodass nur gültige und sinnvolle Operationen angeboten werden. Nutze dazu Data-Binding.



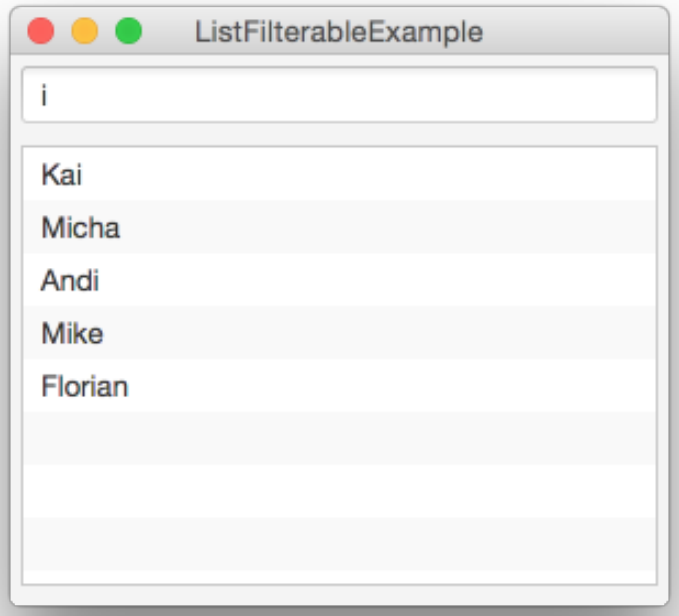
Aufgabe 2:
Erstelle eine Applikation, die Nebenläufigkeit in Form vom `Tasks` nutzt. Simuliere eine Aufgabe auf einer einstellbaren Dauer und Schrittanzahl und visualisiere den Fortschritt mithilfe eines `ProgressBars`. Verwende Property-Binding für den Datenabgleich.



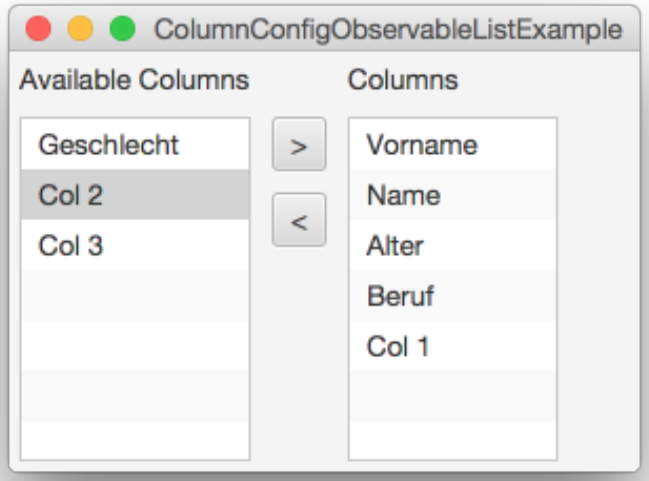
Tipp: Implementiere die Methode `Integer call() throws Exception` dd und rufe dort unter anderem `updateMessage()` auf.

Aufgabe 3:
Lerne die Möglichkeiten aus JDK 8 zum Filtern von Listeninhalten kennen. Gestalte das vorgegebene Programm `Exercise3_FilterListExample_JDK7` mithilfe von JDK 8 um:

a) In Schritt 1 soll zunächst lediglich eine Konvertierung des Algorithmus auf JDK 8 vorgenommen werden.
b) In Schritt 2 soll die Klasse `FilteredList` aus dem JDK genutzt werden.



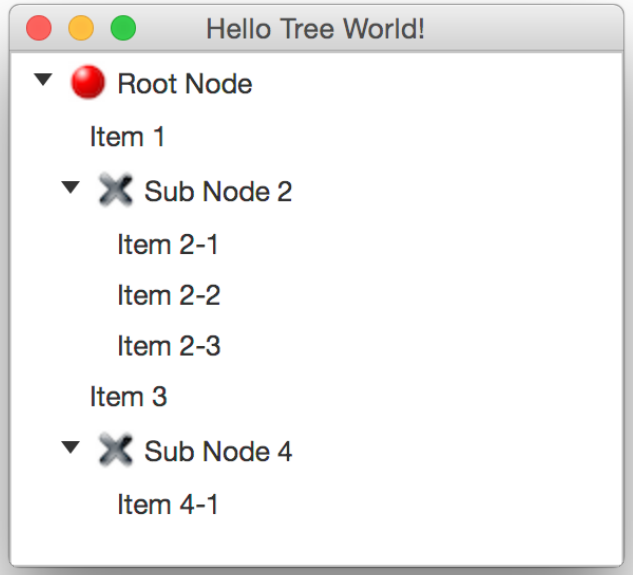
Aufgabe 4 KÜR:
Baue eine Mini-Spalten-Konfigurationsapplikation. Profitiere von `ObservableList`.



Teil 3: Tree, Table & TreeTable

Zur Visualisierung von komplexeren Daten eignen sich anstelle von Listen eher Bäume oder Tabellen oder die `TreeTableView` als Kombination daraus.

Aufgabe 1:
Erstelle ein Programm, das folgenden Tree aufbaut:



Tipp: Nutze die Klassen `TreeItem<T>` und `TreeView` sowie `ImageView` für die Icons.

```
TreeItem<String> root = new TreeItem<String>("Root Node“, redBallImage);
root.getChildren().addAll(
     new TreeItem<String>("Item 1"),
     new TreeItem<String>("Item 3"),
);
TreeView<String> treeView = new TreeView<String>(root);
```

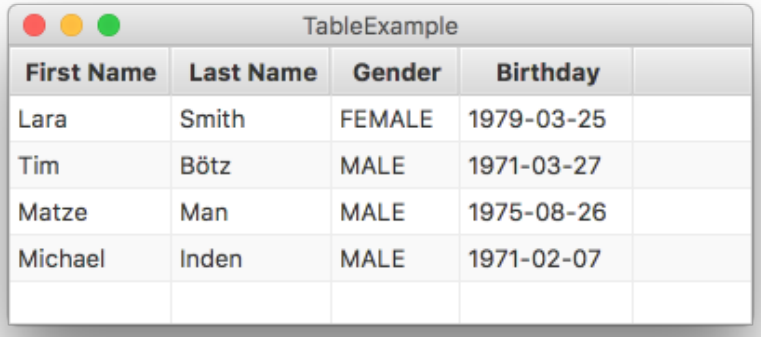
Aufgabe 2:

Wir wollen neben Standardtypen auch spezielle Typen nutzen, etwa eine Aufzählung `Gender` für das Geschlecht:

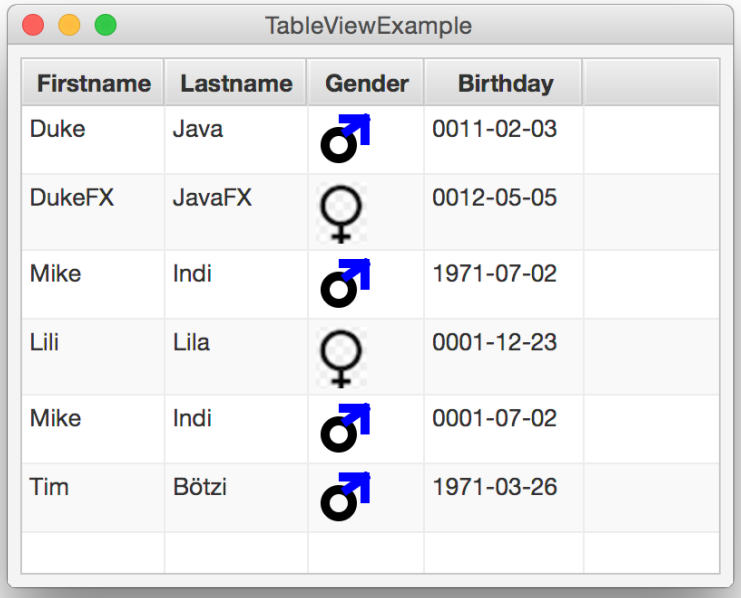
```
enum Gender { MALE, FEMALE }
```

Auch das Geburtsdatum soll durch die Klasse `LocalDate` aus dem mit Java 8 neu eigeführten Daten And Time API repräsentiert werden:

```
public class PersonEx
{
      final String firstName;
      final String lastName;
      final Gender gender;
      final LocalDate birthday;

      …
}
```



Zur gelungenen Darstellung soll das Geschlecht als kleine Grafik repräsentiert werden. Dazu soll ein Bild eingelesen werden und die männliche Darstellung durch das Zeichnen in Komponenten mit dem `Canvas` realisiert werden:



Die Darstellung übernimmt folgende Klasse `GenderOwnRenderer`:

```
class GenderOwnRenderer implements Callback<TableColumn<PersonEx, Gender>,
                                            TableCell<PersonEx, Gender>>
{
    @Override
    public TableCell<PersonEx, Gender> call(TableColumn<PersonEx, Gender> param)
    {
        return new GenderCell();
    }
}
```

Insbesondere die dort genutzte Klasse `GenderCell` muss noch passend ergänzt werden:

```
    @Override
    public void updateItem(Gender item, boolean empty)
    {
            if (item == Gender.MALE)
            {
                setTextFill(Color.BLUE);

                final Canvas canvas = new Canvas(30, 30);
                final GraphicsContext gc = canvas.getGraphicsContext2D();

                 …
            }
        }
    }
```

Die Klasse `Canvas` bzw. der `GraphicsContext` ist ähnlich zu der Klasse `Graphics2D` aus Java2D. Auf einem `Canvas` können verschiedene Zeichenoperationen ausgeführt werden, unter anderem folgende:

- Malfarbe: gc.setStroke(Color.FIREBRICK);
- Strichstärke: gc.setLineWidth(10);
- Linien: gc.strokeLine(40, 10, 10, 40);
- Kreise / Ovale: gc.strokeOval(120, 40, 50, 50);
- Füllfarbe: gc.setFill(Color.BLUE);
- Gefülltes Rechteck: gc.fillRect(10,10,100,100);
- Pfade:

```
gc.beginPath();
gc.moveTo(50, 50);
gc.lineTo(100, 100);
gc.bezierCurveTo(150, 20, 150, 150, 75, 150);
gc.closePath();
gc.stroke(); / gc.fill();
```

Weitere Infos
http://www.informatik-aktuell.de/entwicklung/programmiersprachen/zeichnen-in-javafx-komponenten.html

Aufgabe 3:

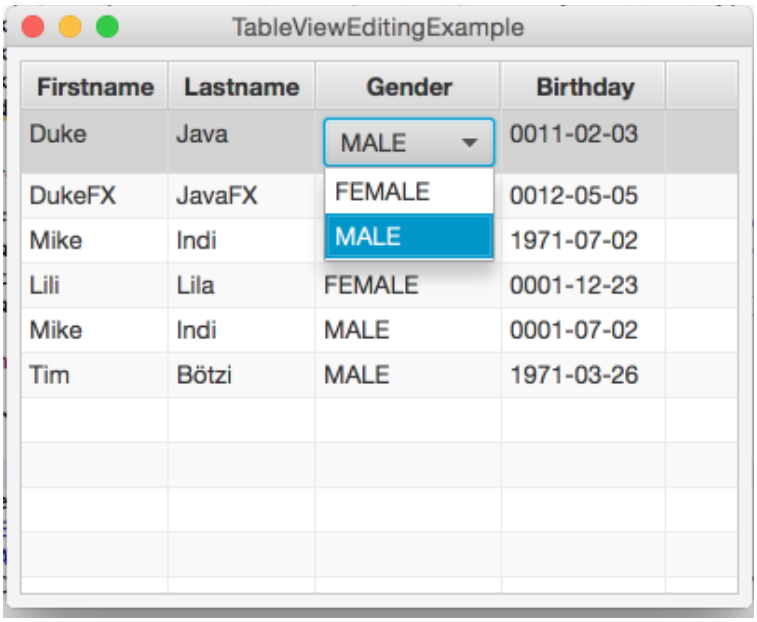
Ziel ist es, Editoren in der `TableView` kennenzulernen. Befülle dazu ein `TableView` mit `PersonEx`-Objekten, bei der einige Attribute nicht `final` sind, um diese editieren zu können.

```
public class PersonEx {

      final String firstName;
      final String lastName;
      final Gender gender;
      final LocalDate birthday;

   …
}
```

Ergänze die Editierbarkeit für den Vornamen. Nutze dazu die vordefinierte Klasse `TextFieldTableCell`. Ergänze die folgenden Zeilen nach der Definition der Spalten:

```
// Zugriff auf Default
firstNameCol.setCellFactory(TextFieldTableCell.<PersonEx>forTableColumn());

// Editing
tableView.setEditable(true);
```

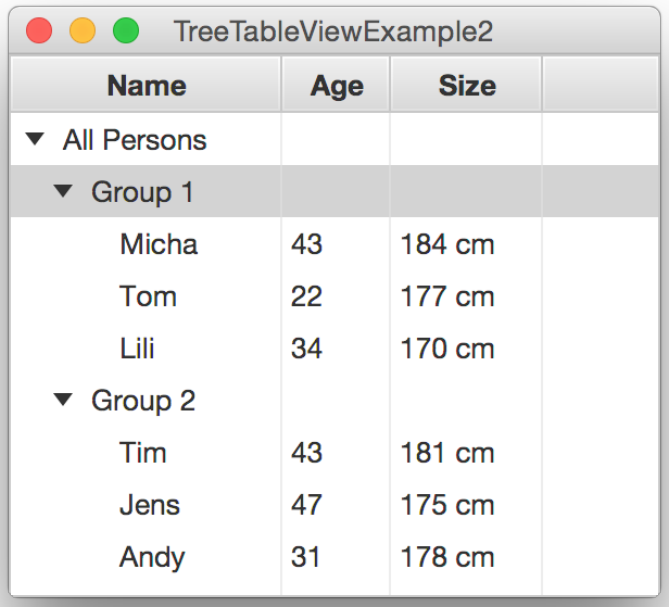
Ergänze die Editierbarkeit für das Geschlecht mithilfe einer `Combobox`:



Tipp: Zur Implementierung von Editoren existieren diverse Editor-Typen im JDK: `CheckBoxTableCell`, `ChoiceBoxTableCell`, `ComboBoxTableCell`, `ProgressBarTableCell` und `TextFieldTableCell`.

Aufgabe 4 KüR:

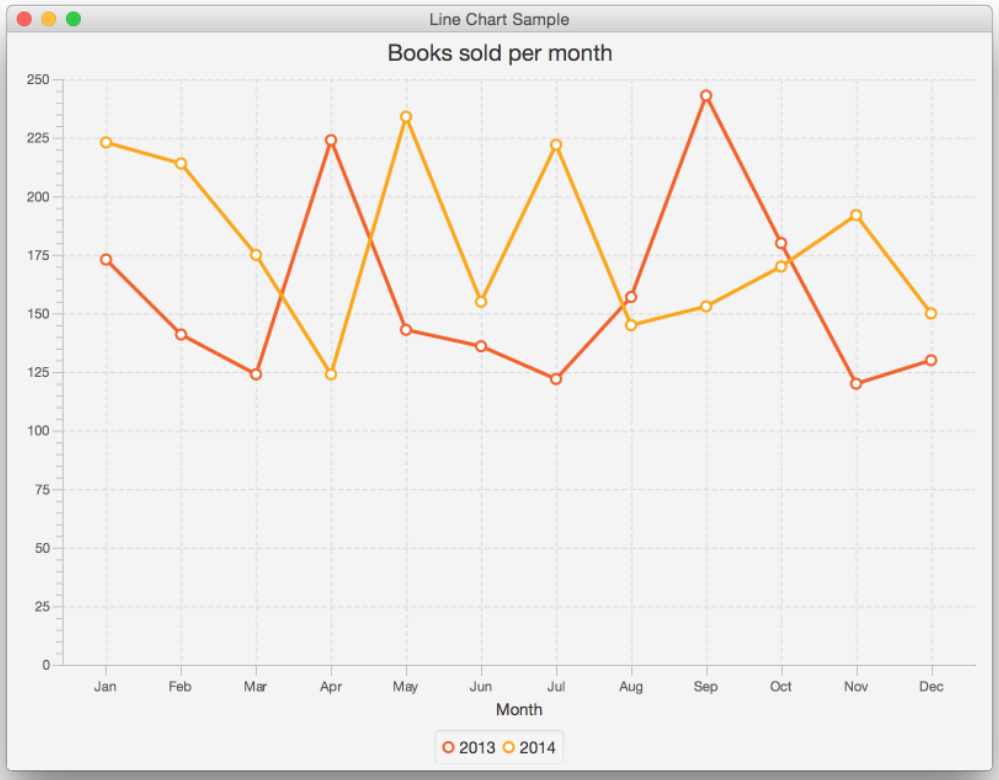
Erstelle eine `TreeTableView` für Personen. Überlege, welche Varianten der Modellierung für Gruppeneinträge existieren. Erstelle einen eigenen Renderer, um gruppenunspezifische Angaben wie Alter oder Größe nicht anzuzeigen.



Teil 4: Charts

Nicht immer ist eine tabellarische oder hierarchische Sicht als Table, Tree bzw. TreeTable angemessen. Teilweise wünscht man sich grafische Darstellungen in Form verschiedener Charts. Das wird durch JavaFX bereits ohne weitere Bibliotheken hervorragend unterstützt, weil es diverse vordefinierte Chart-Typen gibt, womit sich u. ä. Linien-, Balken-, Torten- und Verteilungsdiagramme erstellen lassen.

Aufgabe 1:
Lerne die Möglichkeiten von Charts kennen. Erstelle ein Liniendiagramm, das zwei Linienverläufe zeigt:



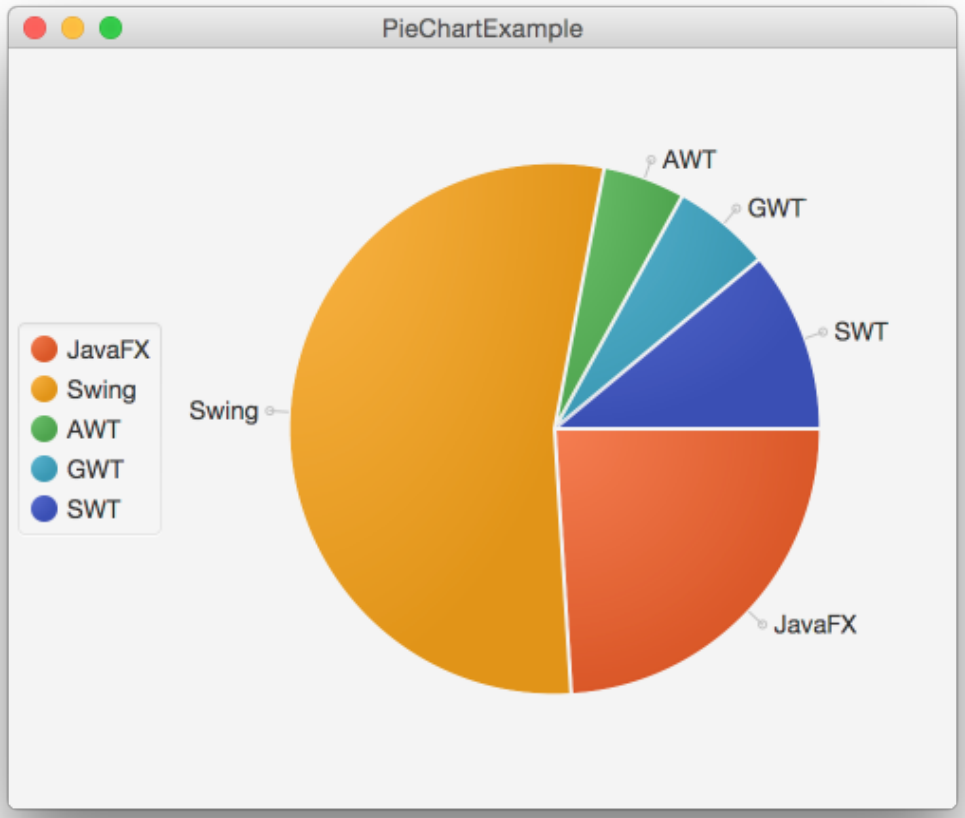
Starte mit folgenden Programmzeilen

```
final LineChart<String, Number> lineChart = new LineChart<>(xAxis, yAxis);

lineChart.setTitle("Books sold per month");

final XYChart.Series<String, Number> series2013 = createSeriesFor2013();
final XYChart.Series<String, Number> series2014 = createSeriesFor2014();
```

Einzelne Datensätze lassen sich wie folgt erzeugen:

```
series.getData().add(new XYChart.Data<String, Number>("Jan", 173));
```

Aufgabe 2:
Ein weiteres praktisches Diagramm ist ein Tortendiagramm. Erstelle ein Ranking der beliebtesten Java-Grafik-Frameworks in etwa wie folgt:



Beim `PieChart` werden die Daten in einem anderen Format als beim Liniendiagramm benötigt:

```
final ObservableList<PieChart.Data> pieChartData = FXCollections.observableArrayList();

pieChartData.add(new PieChart.Data("JavaFX", 24.0));
```

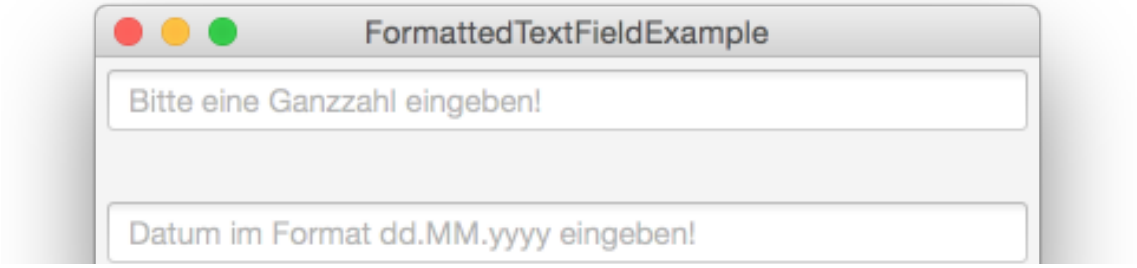
Teil 5: JavaFX 8 — Update 40

In Update 40 von JavaFX 8 findet sich endlich eine Unterstützung für Dialoge sowie für formatierte Textfelder.

Aufgabe 1:
Lerne die Möglichkeiten der neuen Dialog-Elemente kennen. Nachfolgend soll eine kleine Sicherheitsabfrage vor dem Programm-Schliessen eingebaut werden:

Tipp: Nutze einen Lambda und die Methode `setOnCloseRequest()` der `Stage`.

Aufgabe 2:
Erstelle eine kleine Applikation, die zwei Textfelder zur Eingabe anbietet. Im ersten sollen lediglich Ganzzahlen erlaubt sein. Im zweiten Textfeld sollen Datumswerte im Format `dd.MM.yyyy` eingeben sein. Das Ganze sieht dann etwa wie folgt aus:



Nachfolgende Utility-Klasse kann man nutzen, um Fehler geeignet aufzubereiten:

```
public class DialogUtils
{
    public static void showExceptionDialog(final String hint,
                                           final Exception ex)
    {
        final Alert alert = new Alert(AlertType.ERROR);
        alert.setTitle("Internal Software Error");
        alert.setHeaderText(hint);
        alert.setContentText(ex.toString());

        final Pane detailsPane = createStacktracePane(ex);
        alert.getDialogPane().setExpandableContent(detailsPane);

        alert.showAndWait();
    }

    private static Pane createStacktracePane(final Exception ex)
    {
        final StringWriter sw = new StringWriter();
        final PrintWriter pw = new PrintWriter(sw);
        ex.printStackTrace(pw);

        final Label details = new Label("Stacktrace:");
        final TextArea textArea = new TextArea(sw.toString());
        textArea.setEditable(false);
        textArea.setWrapText(true);

        final FlowPane contentPane = new FlowPane();
        contentPane.getChildren().addAll(details, textArea);
        return contentPane;
    }
}
```

Bonus: Grafische Elemente

In JavaFX werden grafische Figuren als Subklassen vom Typ `Node` bereitgestellt, etwa durch die Klassen `Arc`, `Circle`, `Line` und `Rectangle` – alle mit dem Basistyp `Shape` und aus dem Package `javafx.scene.shape`. Diese lassen sich in Containerkomponenten, wie z. B. einer `FlowPane`, anordnen – wie alle anderen `Nodes` auch. Wir schreiben Folgendes:

```
@Override
public void start(final Stage primaryStage) throws Exception
{
  // Kreisbogen mit Beleuchtung
  final Arc arc = new Arc(10, 10, 50, 50, 45, 270);
  arc.setType(ArcType.ROUND);
  arc.setFill(Color.GREENYELLOW);
  arc.setEffect(new Lighting());

  // Kreis mit Reflexion
  final Circle circle = new Circle(10, 30, 30, Color.FIREBRICK);
  circle.setEffect(new Reflection());

  // Linie mit Schatten
  final Line line = new Line(10, 10, 40, 10);
  line.setEffect(new DropShadow());

  // Rechteck mit Beleuchtung
  Rectangle rectangle = new Rectangle(10, 10, 120, 120);
  rectangle.setArcWidth(20);
  rectangle.setArcHeight(20);
  rectangle.setFill(Color.DODGERBLUE);
  rectangle.setEffect(new Lighting());

  final FlowPane flowPane = new FlowPane();
  flowPane.getChildren().addAll(arc, circle, line, rectangle);
  primaryStage.setScene(new Scene(flowPane, 300, 130));
  primaryStage.setTitle(this.getClass().getSimpleName());
  primaryStage.show();
}
```

Aufgabe:
Experimentiere ein wenig mit den Figuren.