Programmierkurs Anwendungsentwicklung

Einführung JavaFX

Wo stehen wir …

- Klassen und Objekte
- Vererbung und Schnittstellen
- Generische Klassen und Methoden
- Exceptions
- Parallele Ausführung von Prozessen / Threads
- Lambda-Ausdrücke
- Collections
- Streams

- Dateien und Verzeichnisse
- Serialisierung
- Parallelprogrammierung in Java: Das Concurrent-Paket
- JavaFX (FXML, Grahpics, Properties und Bindings)
- Fallstricke und Lösungen im Praxisalltag
  - Bed Smells, Design Pattern, …

HowTo

- Unter:
  https://gluonhq.com/products/javafx/
  die passende Version herunterladen.
- Anleitung für verschiedene IDEs:
  https://openjfx.io/openjfx-docs/#install-javafx

Es gibt natürlich noch viel mehr Anleitungen …

Einführung in JavaFX

- Motivation und Eigenschaften
- „Hello World“ in JavaFX
- Komponenten und Szenegraph
- Nutzeraktionen
- GUI gestalten mit CSS

➔Lernziele:

- Grundlagen in JavaFX kennenlernen
- GUIs erstellen und verwenden können
- Konzepte von GUI-Programmierung in Java verstehen

Motivation

GUI – **G**raphical **U**ser **I**nterface

- Anschauliche und leichtere Möglichkeit zur Dateneingabe und Kontrolle von Programmen

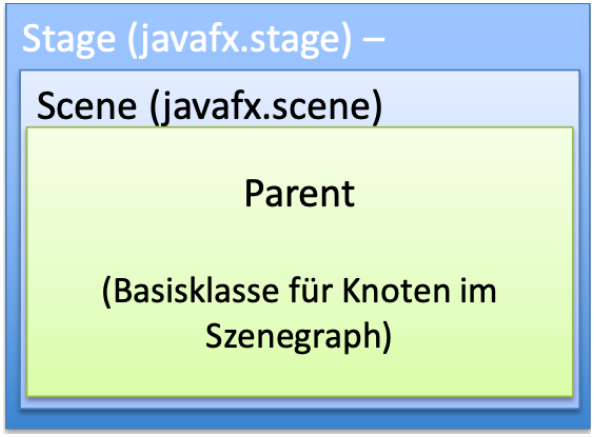
JavaFX:

- Große und vielseitige Bibliothek zur Gestaltung grafischer Oberflächen
- War seit JDK 7 in das JDK integriert
- Wird aber seit Java 11 unabhängig vom JDK entwickelt und veröffentlicht
- Zur Entwicklung von Desktop und RIAs (Rich Internet Applications)
- Schnell erstellbare neue UI-Komponenten (per CSS gestaltbar)
- JavaFX Anwendungen auf nahezu allen Geräten ausführbar (auch im Browser)
- Grafische WYSIWYG Tools: JavaFX Scene Builder

JavaFX-Anwendung

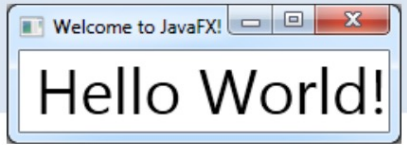
Grundlagen sind Szenegraphen im Hauptfenster. Die wichtigsten Komponenten:

- Hauptfenster ist unsere Bühne (**Stage**)
- Auf der Bühne gibt es Szenen (**Scene**)
- Die Szene besitzt die Elemente in einer baumartigen Struktur, dem Szenegraph (**Nodes**)



„Hello World“ in JavaFX

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class HelloWorld extends Application {

    @Override public void start(Stage stage) {
        Text text = new Text(10, 40, "Hello World!");
        text.setFont(new Font(40));
        Scene scene = new Scene(new Group(text));

        stage.setTitle("Welcome to JavaFX!");
        stage.setScene(scene);
        stage.sizeToScene();
        stage.show();
    }

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



Hauptklasse einer JavaFX Applikation erbt von javafx.application.Application

Einstiegspunkt für jede JavaFX Anwendung

Bekommt die Bühne

Die Szene mit dem Text errichten

Die Szene auf die Bühne holen

Die Bühne anzeigen

Klassische Main-Methode: Für JAR-Files ohne JavaFX Launcher (Optional)

Komponenten im Detail: Stage

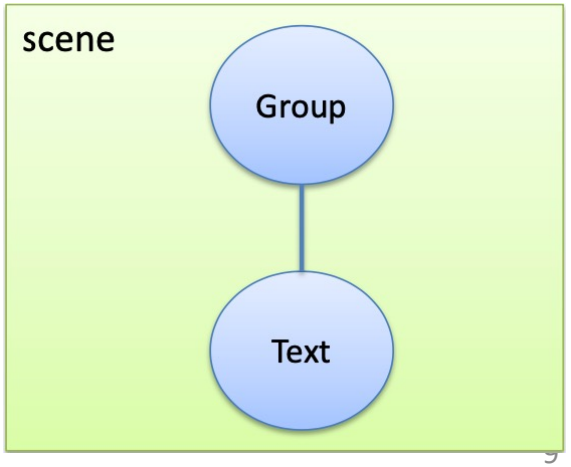
Hauptfenster (Stage) aus `javafx.stage`

- `public class Stage extends Application`
- Stellt den obersten JavaFX Container bereit
- Die Hauptbühne wird von der Plattform erzeugt und beim Start als Argument übergeben, häufig: `public void start(Stage primaryStage){...}`
- Zusätzliche Bühnen können von der Anwendung (Application Thread) erzeugt und verändert werden.
- Kann Eigenschaften des Hauptfensters anzeigen oder festlegen, wie bsp.:
  - `void setTitle(String title);` // Titel des Fensters festlegen
  - `void setMaxWidth(double value);` // Maximale Breite festlegen
  - `Boolean isMaximized();` // Ist das Fenster maximiert?
- Kann aber nicht direkt Elemente aufnehmen, sondern braucht eine Szene:
  - `primaryStage.setScene(meineSzene);`

Komponenten im Detail: Scene

Szene (Scene) aus `javafx.scene`

- `public class Scene`
- Stellt den Container für alle Inhalte des Szenegraph bereit
  - Dazu muss ein Wurzelknoten (Root-Node) vom Typ Parent angegeben werden:
    - `public Scene(Parent root);` // Ein Konstruktor von Scene
- Ausgehend von diesem Wurzelknoten können nun Elemente hinzugefügt werden.
  - Meist verwendet man als `Parent root` eine Glasscheibe (`Pane`) auch für das Layout (`javafx.scene.layout.Pane`), die wiederum mehrere Objekt enthalten können - auch weitere Panes
- So entsteht der Szenegraph als eine Art Baum, dessen Blätter graphische Elemente (Buttons, Textfelder, usw.) und die Verzweigungen weitere Panes sind.
- `Scene scene = new Scene(new Group(text));`



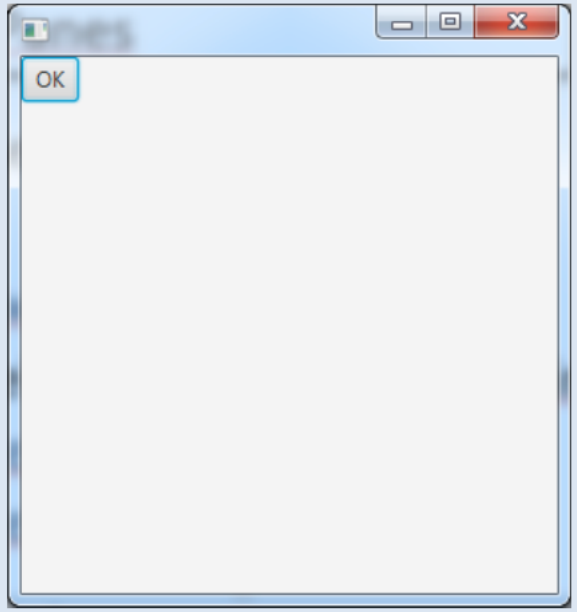
Arbeiten mit Panes

Wir können mit Glasscheiben sehr einfach arbeiten:

- Erstellen:
```
Pane glasscheibe = new Pane();
```
- Elemente Hinzufügen (Indirekt über die Liste der Kindknoten):
```
glasscheibe.getChildren().add(text);
glasscheibe.getChildren().add(button);
```

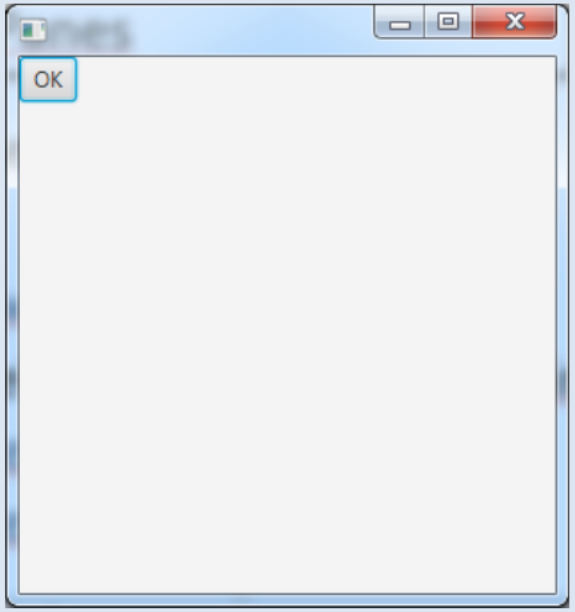
Arbeiten mit Panes

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

      // Elemente
      Text text = new Text("Hello World!");
      Button button = new Button("OK");

      // Glasscheibe
      Pane glasscheibe = new Pane();

      // Elemente auf die Glasscheibe stellen
      glasscheibe.getChildren().add(text);
      glasscheibe.getChildren().add(button);

      //Scene erstellen
      Scene szene = new Scene(glasscheibe,300,300);

       // Die Szene auf die Bühne holen
       stage.setScene(szene);
       stage.show();
}
```



Arbeiten mit Panes

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

      // Elemente
      Text text = new Text("Hello World!");
      Button button = new Button("OK");

      // Glasscheibe
      Pane glasscheibe = new Pane();

      // Elemente auf die Glasscheibe stellen
      glasscheibe.getChildren().add(text);
      glasscheibe.getChildren().add(button);

      //Scene erstellen
      Scene szene = new Scene(glasscheibe,300,300);

       // Die Szene auf die Bühne holen
       stage.setScene(szene);
       stage.show();
}
```



Es wurde kein Layout spezifiziert:
Daher werden alle Elemente übereinander gelegt.

Layouts

- Layouts können direkt mit einer Glasscheibe verknüpft sein: Built-in Layout Panes (javafx.scene.layout.Pane)
- Guter Überblick unter: https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

Layouts

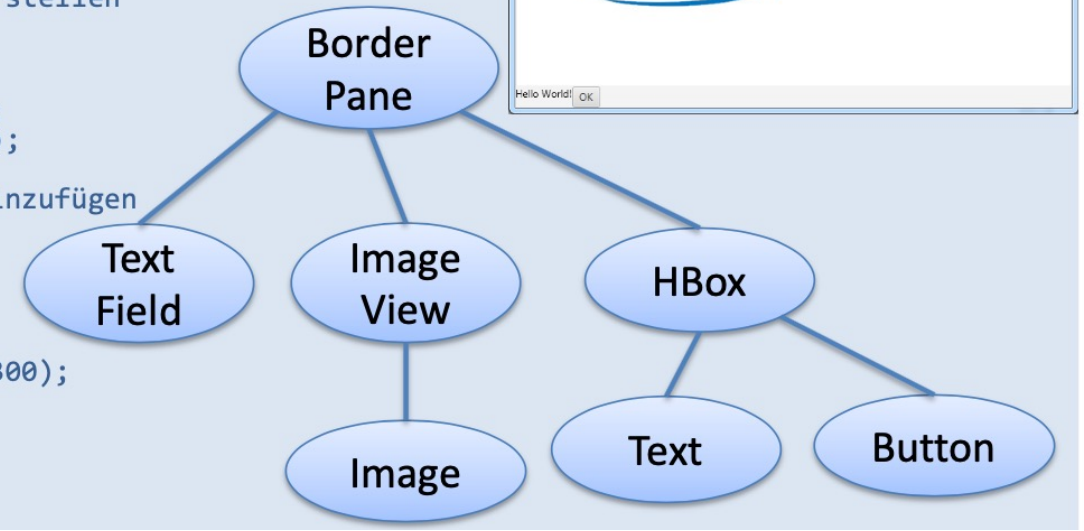
```
@Override public void start(Stage stage) {

    // Elemente
    Text text = new Text("Hello World!");
    Button button = new Button("OK");

    // Glasscheibe mit HBox Layout:
    HBox box = new HBox();

    // Elemente auf die Glasscheibe stellen
    box.getChildren().add(text);
    box.getChildren().add(button);

    //Scene erstellen
    Scene szene = new Scene(box,300,300);

       // Die Szene auf die Bühne holen
       stage.setScene(szene);
       stage.show();
  }
```

Erweiterter Szenengraph

Panes können Elemente und auch wieder Panes mit Elementen enthalten:

- Bsp.: eine BorderPane hat eine HBox Pane und andere Elemente:

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

      // Elemente
      Text text = new Text("Hello World!");
      Button button = new Button("OK");
      TextField txt_field = new TextField();
      txt_field.setMinWidth(20);
      ImageView iv = new ImageView(new Image("java.png"));

      // Glasscheibe mit HBox Layout:
      HBox box = new HBox();

      // Elemente auf die Glasscheibe stellen
      box.getChildren().add(text);
      box.getChildren().add(button);

      // Glasscheibe mit BorderLayout:
      BorderPane bp = new BorderPane();

      // Glasscheibe zur Borderpane hinzufügen
      bp.setTop(txt_field);
      bp.setCenter(iv);
      bp.setBottom(box);

      //Scene erstellen
      Scene szene = new Scene(bp,300,300);

       //…
   }
```

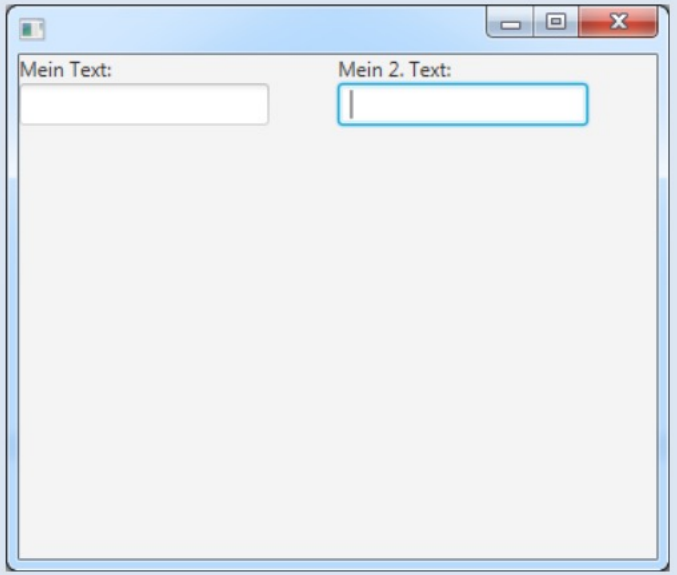


Umgang mit Komponenten

Ein paar Hinweise im Umgang mit Komponenten:

- Aktive Komponenten (bspw.: Textfelder, Buttons, etc.) sollten als Instanzvariablen deklariert werden, um sie später benutzen zu können. Bsp.:
  - private Button ok_btn
- Logisch zusammengehörige Komponenten sollten i.d.R. auch in einer Pane vereinigt werden. Bsp.:
  - Textfelder und deren Beschriftungen (Labels)
  - Buttons (Ok und Cancel)

```
// Beispiel für Labels und Textfelder in einer FlowPane:

      // Elemente
      Label txt_label = new Label("Mein Text: ");
      TextField txt_field = new TextField();
      Label txt_label2 = new Label("Mein 2. Text: ");
      TextField txt_field2 = new TextField();

      // Flow Layout:
      FlowPane fp1 = new FlowPane();
      FlowPane fp2 = new FlowPane();

      // Elemente hinzufügen
      fp1.getChildren().addAll(txt_label,txt_field);
      fp2.getChildren().addAll(txt_label2,txt_field2);

      GridPane gp = new GridPane();
      gp.add(fp1, 0, 0);
      gp.add(fp2,0,1);

      Scene szene = new Scene(gp,300,300); //…
```



Auf Nutzeraktionen reagieren

- TextFelder können per setText(String text) und getText() verändert bzw. ausgelesen werden.
- Um auf Button-Klicks reagieren zu können, müssen wir dem Button mittels setOnAction einen action handler (Behandler) anheften.
  - Reagiert immer dann, wenn der Button gedrückt wurde
  - Implementiert das funktionale Interface EventHandler<ActionEvent>, welches die Methode public void handle(ActionEvent e) bereitstellt
  - Sehr ähnlich zu Swing:
    - mit ActionListener und public void actionPerformed(ActionEvent e)
  - Das geht sehr einfach über eine anonyme innere Klasse oder über Lambda-Ausdrücke

Wdh: Anonyme Innere Klassen

Anonyme Klassen haben

- Keinen Namen
- Erzeugen bei der Klassendeklaration automatisch ein Objekt
- Klassendeklaration und Objekterzeugung sind so zu einem Sprachkonstrukt verbunden

```
// Allgemeine Syntax

new KlasseOderSchnittstelle() { /* Eigenschaften der inneren Klasse */ }
```

Zwei Möglichkeiten

- Steht hinter new ein Klassentyp, dann ist die anonyme Klasse eine Unterklasse dieser Klasse
- Steht hinter new der Name einer Schnittstelle, dann erbt die anonyme Klasse von Object und implementiert die Schnittstelle

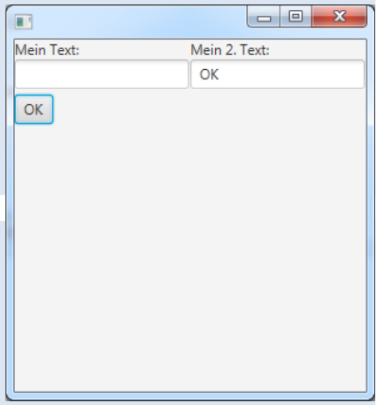
Wdh: Anonyme Innere Klassen

- Können keine weiteren Schnittstellen implementieren oder von weiteren Klassen erben
- Erlauben keine eigenen Konstruktoren
- Nur Objektmethoden und finale statische Variablen erlaubt

```
// Beispiel
public class HelloWorld {
    public void greet() {
        System.out.println("Hello world!");}
}
…
public static void main(String... args) {

        HelloWorld germanGreeting = new HelloWorld() {
            public void greet() {
                System.out.println("Hallo Welt!" ); }};
        germanGreeting.greet();
   }
```

setOnAction

```
//Einfaches Beispiel mit anonymer innerer Klasse:

//…
   TextField txt_field2 = new TextField();

   Button mein_button = new Button("OK");
   mein_button.setOnAction(new EventHandler<ActionEvent>() {

      @Override
      public void handle(ActionEvent event) {
         txt_field2.setText("OK");
      }
   });
```




```
//Einfaches Beispiel mit Lambda-Ausdrücken (Seit Java 8)

//…
   Label txt_label2 = new Label("Mein 2. Text: ");
   TextField txt_field2 = new TextField();

   Button mein_button = new Button("OK");
   mein_button.setOnAction(event -> txt_field2.setText("OK"));
```

Beispiel: Bank

```java
public class BankGUI extends Application {

    @Override
    public void start(Stage primaryStage) {

        final Text msg = new Text();

        primaryStage.setTitle("Willkommen");
        GridPane gp = new GridPane();
        gp.setHgap(10);
        gp.setVgap(10);
        gp.setAlignment(Pos.CENTER);
        gp.setPadding(new Insets(25));

        Scene scene = new Scene(gp);

        primaryStage.setScene(scene);

        Text t_welcome = new Text("Willkommen bei Ihrer Bank!");

        Label l_user = new Label("Nutzername: ");
        Label l_pswd = new Label("Passwort: ");

        TextField tf_user = new TextField();
        PasswordField tf_pswd = new PasswordField();

        gp.add(t_welcome, 0, 0, 2, 1);
        gp.add(l_user, 0, 1);
        gp.add(tf_user, 1, 1);
        gp.add(l_pswd, 0, 2);
        gp.add(tf_pswd, 1, 2);
        gp.add(msg, 1, 4);

        Button login_button = new Button("Einloggen");

        login_button.setOnAction(e -> {
            if (tf_user.getText().equals("")) {
                msg.setText("Es wurde kein Nutzername eingegeben!");
            } else if (tf_pswd.getText().equals("")) {
                msg.setText("Es wurde kein Passwort eingegeben!");
            } else {
                msg.setText("Nutzer eingeloggt!");
            }
        });

        gp.add(login_button, 1, 3);
        primaryStage.sizeToScene();

        primaryStage.show();

    }

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

GUI designen

In Swing mussten die Elemente i.d.R. einzeln designed werden:

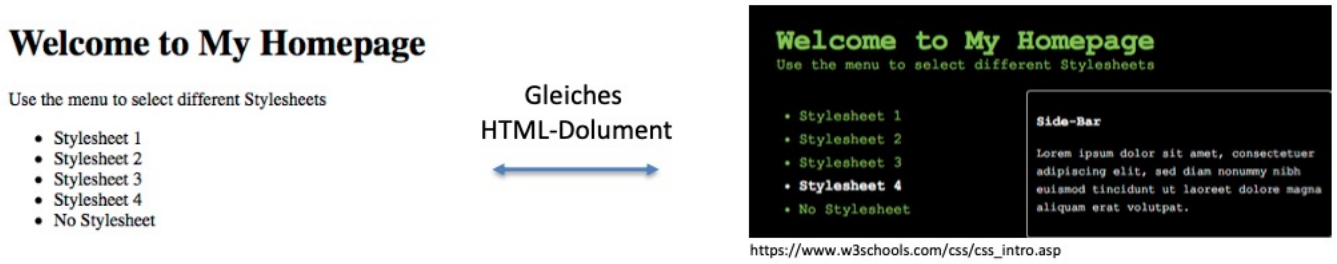
- myButton.setBackground(new Color...);
- myButton.setBorder(...);

In JavaFX ist die Trennung von Inhalt und Layout fest verankert und wird mit *JavaFX CSS* verwirklicht.

- Basiert auf W3C CSS 2.1 und folgt den bekannten Regeln für CSS
- Reference Guide unter: https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

Cascading Style Sheets (CSS)

- CSS beschreibt wie HTML-Elemente angezeigt werden sollen
- CCS wird verwendet um Design und Layout für eine Website zu definieren



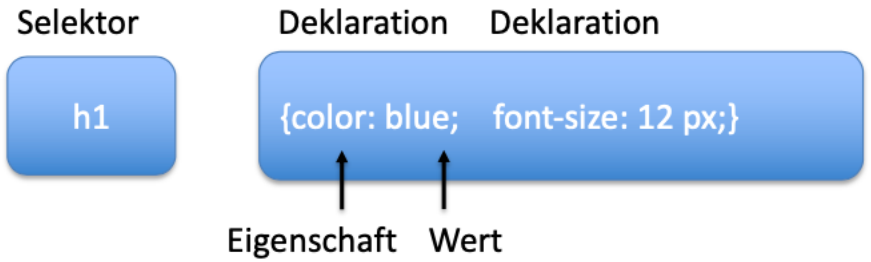
https://www.w3schools.com/css/css_intro.asp

Exkurs: Warum nicht einfach in das HTML integrieren?

- HTML war nie dazu gedacht, Tags zur Formatierung einer Website zu beinhalten
- In HTML 3.2 wurden Tags wie <font> und Farb-Attribute ergänzt
- Das führte allerdings zu sehr großen HTML-Dokumenten, da alle Informationen in jeder Seite enthalten sein mussten
- Deshalb entwickelte das World Wide Web Consortium (W3C) CSS
- Die Definition des Styles einer Website erfolgt i.d.R. über ein externes .css-File

CSS Syntax

- Der Selektor gibt das Element an, das gestylt werden soll
- Der Deklarationsblock enthält eine oder mehrere Deklarationen, die durch ein Semikolon getrennt sind
- Jede Deklaration enthält eine CSS-Eigenschaft (Property) und einen WertCSS-Selektoren werden verwendet, um Elemente zu finden basierend auf deren Namen, ID, Klasse, bestimmter Attribute etc.



CSS Selektoren

- CSS-Selektoren werden verwendet, um Elemente zu finden basierend auf deren Namen, ID, Klasse, bestimmter Attribute etc.

  - **Element-Selektor**: Selektion von Elementen anhand des Namens
    - Beispiel: Selektion aller <p> Elemente:

      ```
      p {text-align: center; color: red;}
      ```

    - Ergebnis: Alle Absätze (Paragraphs) werden zentriert und rot angezeigt

  - **ID-Selektor**: Selektion anhand der ID
    - Verwendung von Hashtag gefolgt von ID:

      ```
      #para1 {text-align: center; color: red;}
      ```

  - **Klassen-Selektor**: Selektion anhand eines bestimmten Klassenattributes
    - Verwendung eines Punktes gefolgt von Klassenname:

      ```
      .center {text-align: center;}
      ```

JavaFX CSS- Ein paar Unterschiede zu CSS:

- JavaFX Eigenschaften werden mit dem Präfix fx erweitert
- verbietet CSS Layout-Eigenschaften, wie float, position, usw.
- bietet einige Erweiterungen, bspw.: (Hintergrund-)Farben, Ränder, usw.

CSS Styles werden nun für Knoten im JavaFX Szenegraphen verwendet.

- Für das Mapping gelten die bekannten CSS Regeln für Selektoren:
  - Element-Selektoren:
    - Analog zu einem Element in HTML
    - I.d.R. einfach der Name der Klasse, bspw.: Button, Text, usw.
    - Kann abgefragt werden mittels `public String getTypeSelector()`

```
// Beispiel für Typ-Selektor Definition:

Button {
-fx-font-size: 25px;
-fx-font-weight: bold;
-fx-font-style: italic;
-fx-font-family: "Arial Blank"
}
```

JavaFX CSS

Klassen-Selektoren:

- Jeder Knoten im Szenegraph kann zu einer oder mehreren Klassen gehören (analog zum class-Attribut in HTML)
- Elemente können mittels getStyleClass().add(String class) zu einer Klasse hinzugefügt werden:
  - Bsp.: myButton.getStyleClass().add(„MeineKlasse“);
- Ansprechbar dann mit bekannter Punkt-Notation:

```
.MeineKlasse{
   -fx-font-weight: bold;
}
```

ID- Selektoren:

- Jeder Knoten besitzt ein ID-Attribut (analog zu id in HTML)
  - Kann mit setID(String id) gesetzt werden.
  - Bsp.: myButton.setId(“button1“);
- Ansprechbar dann mit bekannter #-Notation

```
#button1{
   -fx-font-weight: bold;
}
```

JavaFX CSS – Verwendung der sog. Pseudoklassen

Pseudoklasse	Auswirkung
Focused	Wenn das Element den Fokus erhält
Hover	Wenn der Mouse-Zeiger über dem Element steht
Pressed	Wenn das Element angeklickt wird

```
// Beispiel für Pseudoklassen:

Button:hover {

   -fx-border-width: 0.0 ;
   -fx-font-size: 10px;
   -fx-font-weight: bold;
   -fx-font-style: italic;
   -fx-font-family: "Arial Blank";

}
```

CSS in Code einbinden

2 Varianten:

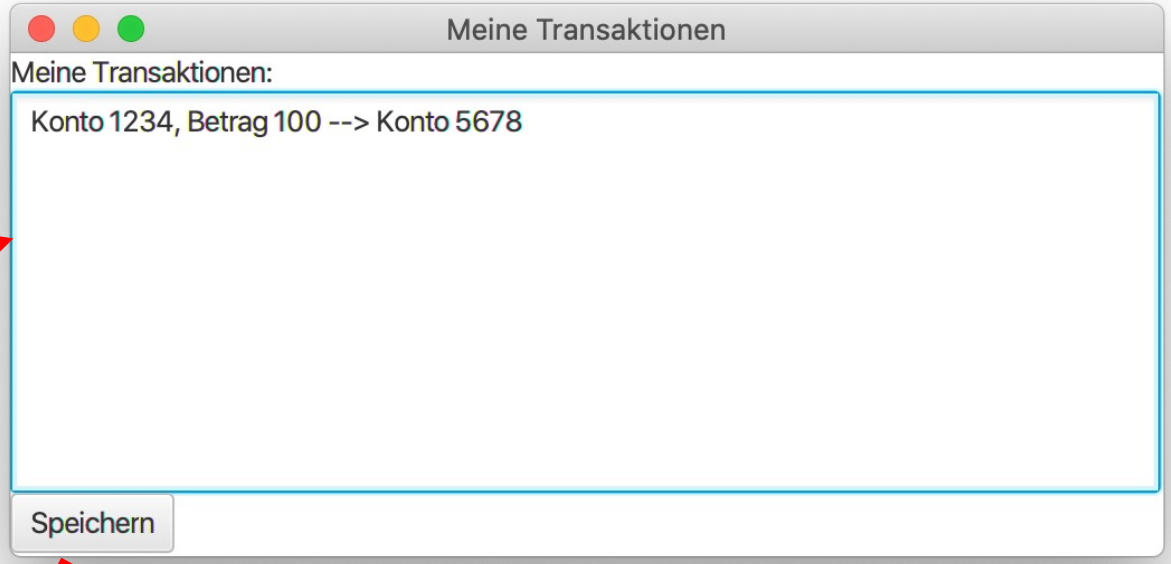
- Variante 1: Inline Styles
  - Direkt im Code (Class File)
  - Jeder Node verfügt über die Methode public final void setStyle(String value)
    - Bsp.: myButton.setStyle("-fx-font-size: 20px");
- Variante 2: Style sheets
  - Eigene separate CSS Datei.css
  - Wird i.d.R. der Szene mitgeteilt:
    - scene.getStylesheets().add(getClass().getResource("application.css").toString());
  - Damit können globale Design-Einstellungen für eine Szene definiert werden
    - Sehr komfortabel

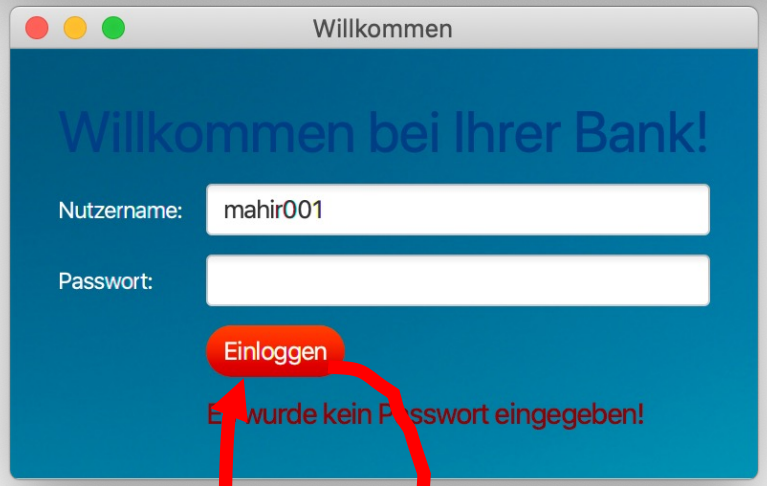
Neuer Inhalt im gleichen Fenster

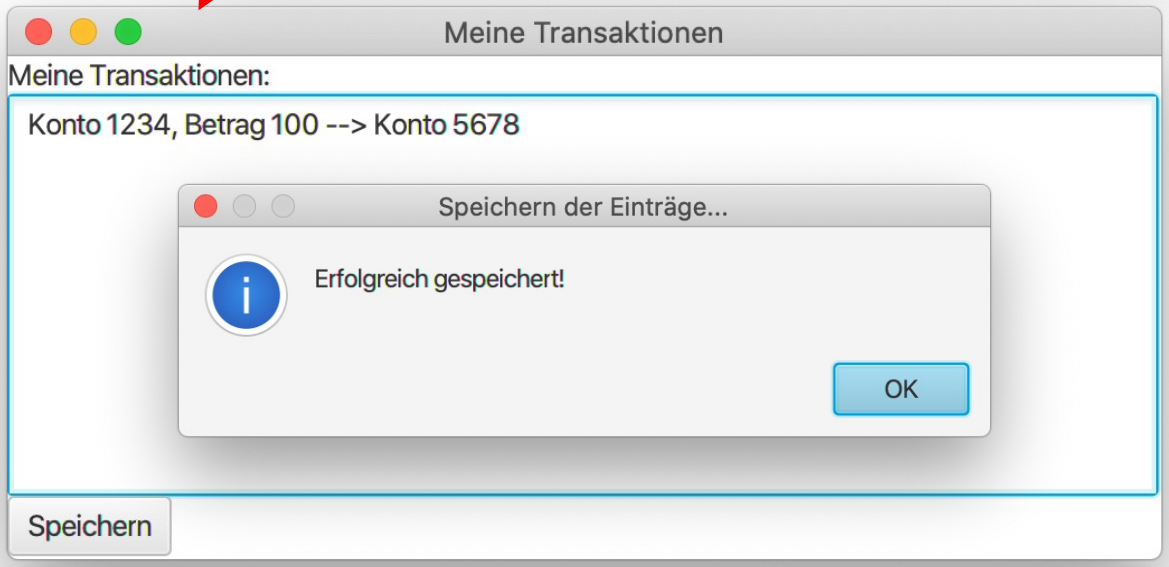
Will man nun den Inhalt in seinem Fenster neu gestalten, bspw. wenn der Nutzer eingeloggt ist, wird einfach eine neue Szene eingesetzt:

- primaryStage.setScene(new Scene(new NeueSzene()));
- Kann als separate Klasse Definiert werden

Beispiel: Bank







Grafikprogrammierung

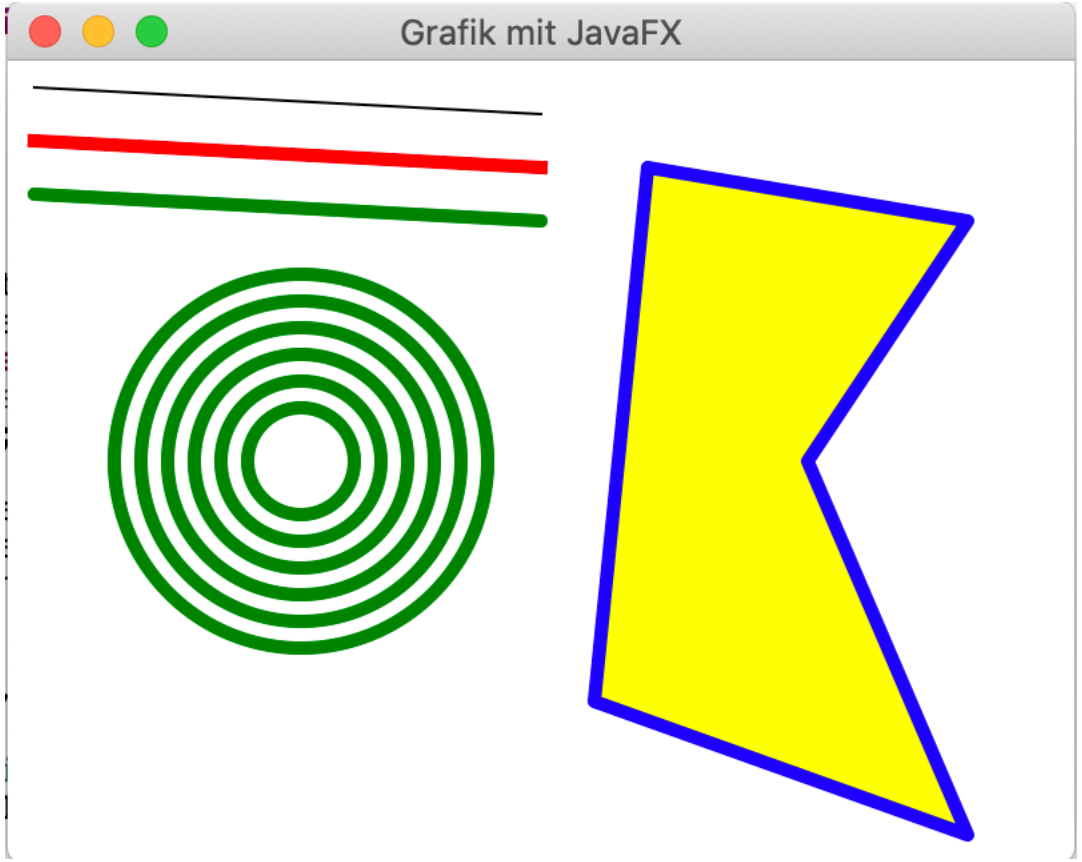
- Problem: JavaFX-Steuerelemente sehr vielseitig, aber man kann nicht ohne Weitere in ihnen zeichnen

➔Für diese Aufgabe gibt es die Canvas-Klasse

- Mit der Methode getGraphicsContext2D ermitteln Sie das GraphicContext-Objekt des Canvas

➔in erster Linie mit den strokeXxx- und fillXxx- Methoden – werden Zeichenoperationen durchgeführt

Beispiel



```
public class GrafikTest extends Application {
  final int W=400, H=300; // Canvas-Größe

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

  // Initialisierung
  @Override
  public void start(Stage primaryStage) {
    primaryStage.setTitle("Grafik mit JavaFX");
    var canvas = new Canvas(W, H);
    var gc = canvas.getGraphicsContext2D();
    drawShapes(gc);
    var root = new Group();
    root.getChildren().add(canvas);
    primaryStage.setScene(new Scene(root));
    primaryStage.show();
  }
```

```
// im Canvas zeichnen
private void drawShapes(GraphicsContext gc) {
  // Linie, standardmäßig schwarz, 1 Pixel breit
  gc.strokeLine(10, 10, 200, 20);

  // rot, 5 Pixel breit
  gc.setStroke(Color.RED);
  gc.setLineWidth(5);
  gc.strokeLine(10, 30, 200, 40);

  // grün, mit runden Endpunkten
  gc.setStroke(Color.GREEN);
  gc.setLineCap(StrokeLineCap.ROUND);
  gc.strokeLine(10, 50, 200, 60);

  // einige Kreise
  for(int r=20; r<80; r+=10)
    gc.strokeOval(110-r, 150-r, 2*r, 2*r);

  // gefülltes Polygon
  double[] x = new double[] {240, 360, 300, 360, 220};
  double[] y = new double[] {40, 60, 150, 290, 240};

  gc.setFill(Color.YELLOW);          // Innenfarbe
  gc.fillPolygon(x, y, x.length);    // Polygon füllen

  gc.setStroke(Color.BLUE);              // Linienfarbe
  gc.setLineJoin(StrokeLineJoin.ROUND);  // runde Ecken
  gc.strokePolygon(x, y, x.length);      // Rand
 }
}
```

Properties & Bindings

Motivation

Properties und Binding

- 2 wichtige Mechanismen
- Häufig in Kombination
- Verbindungen zwischen Variablen herstellen und gestalten
  - Meistens, um Werte zu aktualisieren und konsistent zu halten
  - Bsp.: GUI-Programmierung

Properties:

- Erweitern das bekannte JavaBean-Konzept
  - Zugriff auf Eigenschaften über Getter/Setter
- Ermöglichen die Kopplung von Eigenschaften einer Klasse mit anderen Objekten (bspw. Nodes)
- Wir können auf Datenänderungen (Events) reagieren

Binding:

- Setzen Properties in eine Beziehung:
  - Unidirektional (Einseitig)
  - Bidirektional (Beidseitig)
- Überwachen den Zustand
- Führen Änderungen automatisch aus

Properties

Das Packet `javafx.beans.property` stellt Property-Klassen für alle primitiven und auch komplexe Datentypen zur Verfügung.

- Beispiele:

Typ	Property Typ
`int`	`IntegerProperty`
`double`	`DoubleProperty`
`short`	`ShortProperty`
`long`	`LongProperty`
`String`	`StringProperty`
`Object`	`ObjectProperty<T>`

- Die primitiven Datentypen werden somit gekapselt zu einem komplexen Objekt, welches die Benachrichtigungsfunktion der Property implementiert
- Diese können nun wie folgt benutzt werden
  - Einige Regeln sind zu beachten

Properties: Beispiel Counter

```
package application;

import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

public class ModelwithProperties {

    private DoubleProperty counter;


    public final double getCounter() {
        if (counter == null)
            return 0;
        return this.counter.get();
    }

    public final void setCounter(double c){
        this.counter.set(c);
    }

    public DoubleProperty counterProperty(){
        if(counter==null)
            this.counter = new SimpleDoubleProperty(0);
        return this.counter;
    }

    public void increment(){
        this.counter.set(this.counter.get()+1);
    }

    public void decrement(){
        this.counter.set(this.counter.get()-1);
    }
}
```

Die Instanzvariable als `DoubleProperty`

Getter und Setter für jede Feldvariable als `public final` anlegen

Zu beachten: Auf Instanziierung prüfen!

Getter/Setter über `get()` und `set()` Funktion der Property realisieren

Einen `public` Getter für die Property anlegen! Konvention: `<feldName>Property()`

Falls noch nicht instanziiert: `Simple<Type>Property(value)`

Weitere Methoden

Properties: Eigenschaften

Die Implementierung der abstrakten Property-Klassen wird ebenfalls von `javafx.beans.property` bereitgestellt:

- **`Simple`**`<Type>Property`
  - Mit Konstruktorparameter (initialValue) oder ohne
    - Weitere Parameter: ObjectBean, Name /+ initValue
  - Erlaubt den Lese- **und** Schreibzugriff (Read/Write)
  - Bsp.: `private IntegerProperty ip = new SimpleIntegerProperty();`

- **`ReadOnly`**`<Type>Wrapper`
  - Mit Konstruktorparameter (initialValue) oder ohne
    - Weitere Parameter: ObjectBean, Name /+ initValue
  - Erlaubt **nur** Lesezugriff
  - Bsp.: `private IntegerProperty ip2 = new ReadOnlyIntegerWrapper(1);`

Da bei einem Aufruf nicht klar ist, ob die Property bereits instanziiert wurde, muss das häufig noch geprüft werden! Sonst evtl. `NullPointerException`

Auf Events (Änderungen) reagieren

Mittels Properties können wir nun ganz leicht auf Datenänderungen reagieren

- Anfügen eines `ChangeListener<T>`
  - Funktionales Interface mit der Methode:

```
public void changed(ObservableValue<?> obs, Object oldVal, Object newVal)
```

  - Reagiert auf Änderungen an der Property und führt den entspr. Code aus.
  - Implementierung über anonyme innere Klasse oder Lambdas:

```
counterProperty().addListener(new ChangeListener<Object>(){

   @Override
   public void changed(ObservableValue<? extends Object> observable, Object
      oldValue, Object newValue){
       System.out.println("Change value from "+oldV+" to "+newV+".");
      }
});

//Mit Lambda:
counterProperty().addListener((obs,oldV,newV)->{
        System.out.println("Change value from "+oldV+" to "+newV+".");
      });
```

Binding

Mittels Binding können wir nun Properties in eine Beziehung zueinander setzen:

- **Unidirectional** (Einseitige Bindung)
  - Bindet Property p1 an Property p2.
  - D.h.: Wird der Wert von p2 geändert, ändert sich automatisch der Wert von p1 entsprechend dem neuen Wert von p2
- **Bidirectional** (Beidseitige Bindung)
  - Bindet entsprechend beide Properties aneinander
  - D.h.: Wird der Wert von einer Property verändert, ändert sich automatisch der Wert der anderen Property entsprechend.

```
// Beispiel Unidirectional Binding:
this.widthProperty().bind(this.heightProperty());


// Beispiel Bidirectional Binding:
this.widthProperty().bindBidirectional(this.heightProperty());
```

Ändert sich die Höhe, dann ändert sich entspr. auch die Breite

Ändert sich die Höhe/Breite, dann ändert sich entspr. auch die Breite/Höhe

Binding

Das Packet `javafx.beans.binding` bietet noch weitere Möglichkeiten von Bindings an:

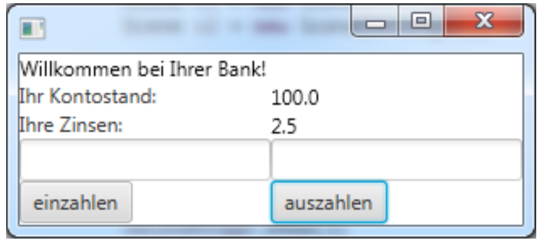
- Für spezifischere Bindings
  - Bsp.: `BooleanBinding`, `DoubleBinding`, `StringBinding`, usw.
- Für komplexere Bindings
  - Interface `Binding`, Interface `NumberBinding`
  - Bsp.: Wenn sich ein Ergebnis einer Berechnung ändert

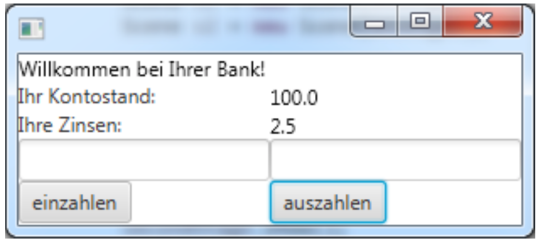
```
// Beispiel für Number Binding:

NumberBinding nb = Bindings.multiply(seiteAProperty(), seiteBProperty());
this.surfaceProperty().bind(nb);
```

Ändert sich das Produkt von Seitenlänge A und B, dann ändert sich auch die Property für die Fläche

Bank Beispiel

- Verwenden Sie nun den `ChangeListener` auf Ihre Properties um auf Ein- und Auszahlungen bei all Ihren GUIs zu reagieren!
- Binden Sie Ihre Zinsen-Property so an den Kontostand, dass eine Änderung des Kontostands auch gleichzeitig die aktuellen Zinsen berechnet und speichert
  - **Hinweis**: Verwenden Sie `Bindings.multiply`





Main

```
 7 public class Main extends Application {
 8     @Override
 9     public void start(Stage primaryStage) {
10         try {
11
12             Bank b = new Bank(100);
13
14             Anzeige anzeige1 = new Anzeige(b);
15             Anzeige anzeige2 = new Anzeige(b);
16
17             Scene s1 = new Scene(anzeige1);
18             Scene s2 = new Scene(anzeige2);
19
20             primaryStage.setScene(s1);
21             primaryStage.setX(10);
22             primaryStage.show();
23
24             Stage secondStage = new Stage();
25             secondStage.setScene(s2);
26             secondStage.setX(100);
27             secondStage.show();
28
29         } catch (Exception e) {
30             e.printStackTrace();
31         }
32     }
33
34     public static void main(String[] args) {
35         launch(args);
36     }
37 }
38
```

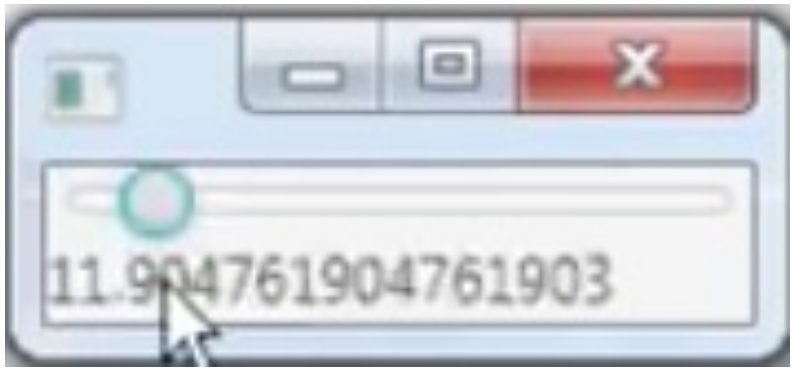
Bank

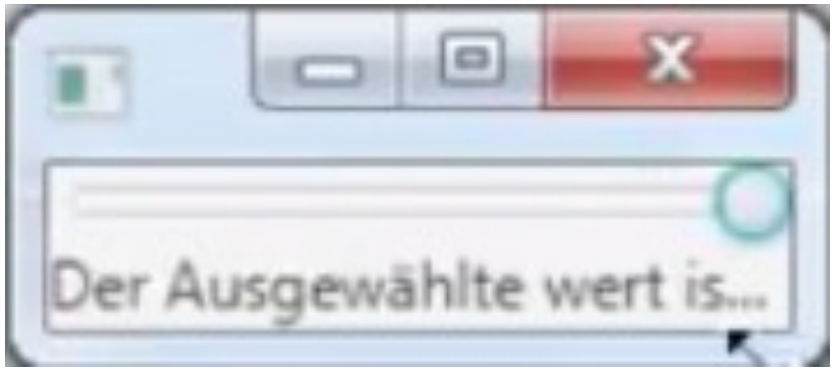
```
public class Bank {

    private DoubleProperty balance, interest;
    private double zinssatz = 0.025;

    public Bank(double value) {
        if (this.balance == null)
            this.balance = new SimpleDoubleProperty(value);
        else
            this.balance.set(value);
    }

    public DoubleProperty balanceProperty() {
        if (this.balance == null)
            this.balance = new SimpleDoubleProperty(0);
        return this.balance;
    }

    public DoubleProperty interestProperty() {

        if (this.interest == null) {
            this.interest = new SimpleDoubleProperty(0);
            this.interest.bind(Bindings.multiply(zinssatz, this.balanceProperty()));
        }
        return this.interest;
    }

    public final void setBalance(double value) {
        this.balance.set(value);
    }

    public final double getBalance() {
        if (this.balance == null)
            return 0.0;
        return this.balance.get();
    }

    public final double getInterest() {
        if (this.interest == null)
            return 0.0;
        return this.interest.get();
    }

    public void einzahlen(double betrag) {
        this.balance.set(this.getBalance() + betrag);
    }

    public void auszahlen(double betrag) {
        this.balance.set(this.getBalance() - betrag);
    }

}
```

Anzeige

```java
public class Anzeige extends Parent {

    private Text balance_info, interest_info;

    public Anzeige(Bank bank) {

        BorderPane root = new BorderPane();

        Text header = new Text("Willkommen bei Ihrer Bank!");

        root.setTop(header);

        header.setId("header");

        GridPane gp = new GridPane();
        root.setCenter(gp);

        Label c_label = new Label("Ihr Kontostand: ");
        balance_info = new Text("" + bank.getBalance());

        Label int_label = new Label("Ihre Zinsen: ");
        interest_info = new Text("" + bank.getInterest());

        TextField tf_einzahlen, tf_auszahlen;

        tf_einzahlen = new TextField();
        tf_auszahlen = new TextField();

        Button btn_einzahlen = new Button("einzahlen");
        Button btn_auszahlen = new Button("auszahlen");

        btn_einzahlen.setOnAction(e -> {

            bank.einzahlen(Double.parseDouble(tf_einzahlen.getText()));
            tf_einzahlen.setText("");
        });

        btn_auszahlen.setOnAction(e -> {

            bank.auszahlen(Double.parseDouble(tf_auszahlen.getText()));
            tf_auszahlen.setText("");
        });

        gp.add(c_label, 0, 0);
        gp.add(balance_info, 1, 0);

        gp.add(int_label, 0, 1);
        gp.add(interest_info, 1, 1);

        gp.add(tf_einzahlen, 0, 2);
        gp.add(tf_auszahlen, 1, 2);

        gp.add(btn_einzahlen, 0, 3);
        gp.add(btn_auszahlen, 1, 3);

        this.getChildren().add(root);

        bank.balanceProperty().addListener((obs, old, newV) -> {
            this.balance_info.setText("" + newV);
        });

        bank.interestProperty().addListener((obs, old, newV) -> {
            this.interest_info.setText("" + newV);
        });

    }

}
```

Beispiel: bind / unbind





Ist der Slider <=50, soll der aktuelle Wert ausgegeben werden.
Ist der Slider >50, soll der Text „Der ausgewählte Wert ist größer als die Hälfte“ ausgegeben werden.

Zusammenfassung

Wo stehen wir …

- Klassen und Objekte
- Vererbung und Schnittstellen
- Generische Klassen und Methoden
- Exceptions
- Parallele Ausführung von Prozessen / Threads
- Lambda-Ausdrücke
- Collections
- Streams

- Dateien und Verzeichnisse
- Testen mit Junit
- Parallelprogrammierung in Java: Das Concurrent-Paket
- JavaFX (FXML, Grahpics, Properties und Bindings)
- Fallstricke und Lösungen im Praxisalltag
  - Bed Smells, Design Pattern, …

Java Pitfalls

Carefully Overloading

```
public class Overloading {


   public static void main(String[] args) {
      myMethod(123);
   }


   public static void myMethod(Integer p) {
      System.out.println("Integer");
   }
   public static void myMethod(int p) {
      System.out.println("int");
   }
   public static void myMethod(long p) {
      System.out.println("long");
   }
   public static void myMethod(Integer... p) {
      System.out.println("Integer ...");
   }
   public static void myMethod(int... p) {
      System.out.println("int ...");
   }
   public static void myMethod(Object p) {
      System.out.println("Object");
   }
}
```