LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Javakurs für Fortgeschrittene

Einheit 05: CSS und MVC in JavaFX

Lorenz Schauer
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Teil 1: GUIs designen mittels CSS

- Prinzip
- Selektoren
  - Typ, Klasse, ID
- Einbinden in den Quellcode

Teil 2: MVC

- Motivation und Grundprinzip
- Die 3 Teile des MVC

Praxis:

- CSS in JavaFX
- MVC realisieren

Lernziele

- Mit CSS in JavaFX arbeiten
- MVC als Entwurfsmuster für grafische Anwendungen kennenlernen

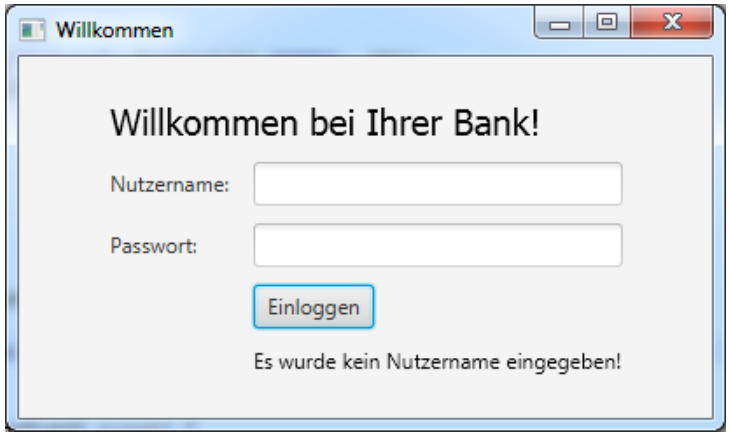
Wiederholung

Füllen Sie nun Ihre zuvor erstellte GUI mit Leben, in dem Sie auf den Einloggen-Button reagieren.

Erstellen Sie zunächst einen Platzhalter für einen Status-Text unterhalb des Einloggen-Buttons.

Reagieren Sie nun wie folgt, wenn der Einloggen-Button gedrückt wurde:

- Falls kein Nutzername eingegeben wurde, erscheint als Status-Text: „Es wurde kein Nutzername eingegeben!“
- Falls kein Passwort Eingegeben wurde, erscheint als Status-Text: „Es wurde kein Passwort eingegeben!“
- Falls Nutzername oder Passwort nicht mit einem von Ihnen akzeptierten Nutzername oder Passwort übereinstimmt, dann geben Sie aus: „Nutzername oder Passwort falsch!“
- Falls beides korrekt, dann geben Sie aus: „Nutzer wird eingeloggt.“



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

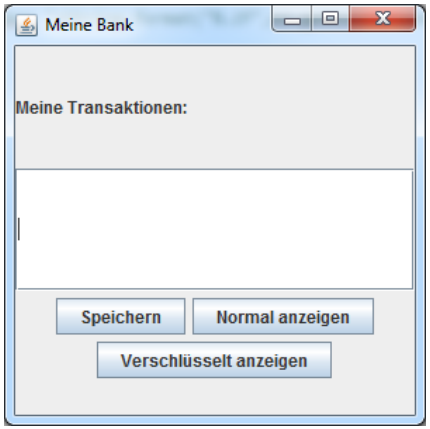
- `primaryStage.setScene(new Scene(new NeueSzene()));`
- Kann als separate Klasse Definiert werden

Hausaufgabe:

Nachdem Sie nun bereits ein funktionsfähiges Log-In Fenster gestaltet haben, sollten Sie nun eine Szene in JavaFX für die Bank-Anwendung von letzter Stunde schreiben und bei einem erfolgreichen Log-In diese aufrufen.

Die Funktionen der Buttons sollten dann genauso funktionieren, wie bei der letzten Hausaufgabe gefordert.

Zum Vergleich, dies war die GUI für die Bankanweisung in Swing =>



Bisher haben wir mit Standard Design-Elementen gearbeitet!

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

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

Oder man nutzte open-source Frameworks (Javaxx, Java CSS,…) um cascading style sheets (CSS) mit Swing Komponenten zu verwenden.

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

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:
  - **Typ-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"
}
```

- **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:
    - `.root` ist das Wurzel-Element

```
.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;
}
```

Daneben stehen uns auch ein Teil der sog. Pseudoklassen zur Verfügung:

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

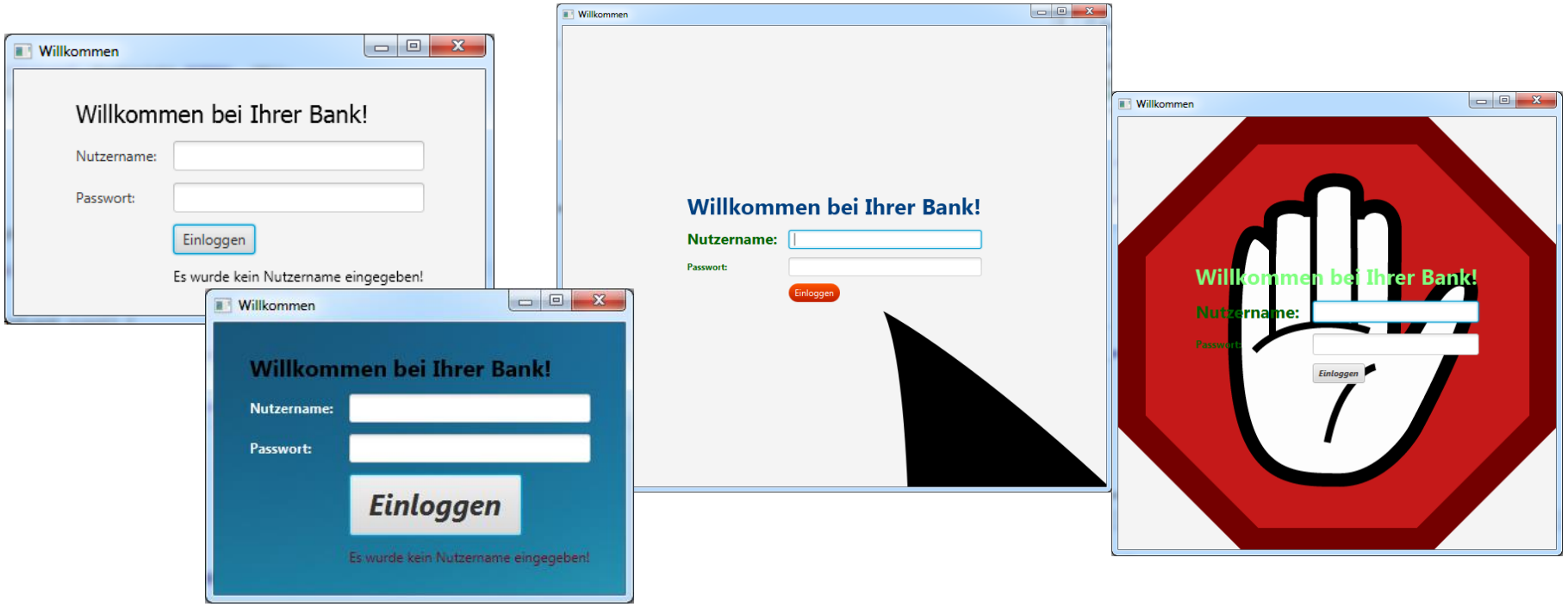
2 Arten, um CSS in unsere JavaFX-Applikation einzubinden (analog zu HTML):

- 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");`
  - Hat höchste Priorität, wäre aber sehr umständlich
- Style sheets
  - Eigene separate CSS Datei. Bspw.: *application.css*
  - Wird i.d.R. der Szene mitgeteilt. Bspw. über:
    - `scene.setUserAgentStylesheet("./ application/application.css");`
      - `//ersetzt auch die vorgegebenen Styles!`
    - `scene.getStylesheets().add("application/application.css");`
    - `scene.getStylesheets().add(getClass().getResource("application.css")`
      `.toExternalForm());`
  - Damit können wir globale Design-Einstellungen für eine Szene definieren.
    - Sehr komfortabel!

Aufgabe zu JavaFX: GUI individuell designen

Verwenden Sie nun JavaFX CSS Elemente, um Ihre zuvor erstellte GUI individuell nach Ihren Bedürfnissen zu gestalten.

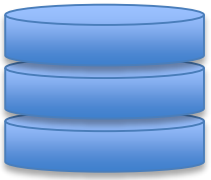
- Seien Sie dabei etwas kreativ und spielen Sie ein wenig mit den Möglichkeiten, die Ihnen JavaFX CSS bietet.
- Verwende Sie auch einmal inline Styles bzw. verschiedene Style Sheets.
- Ihre GUI könnte sich wie folgt verändern:



MVC: **M**odel **V**iew **C**ontroller

Bei der Programmierung einer grafischen Anwendung haben wir meistens 3 größere Bereiche zu implementieren:

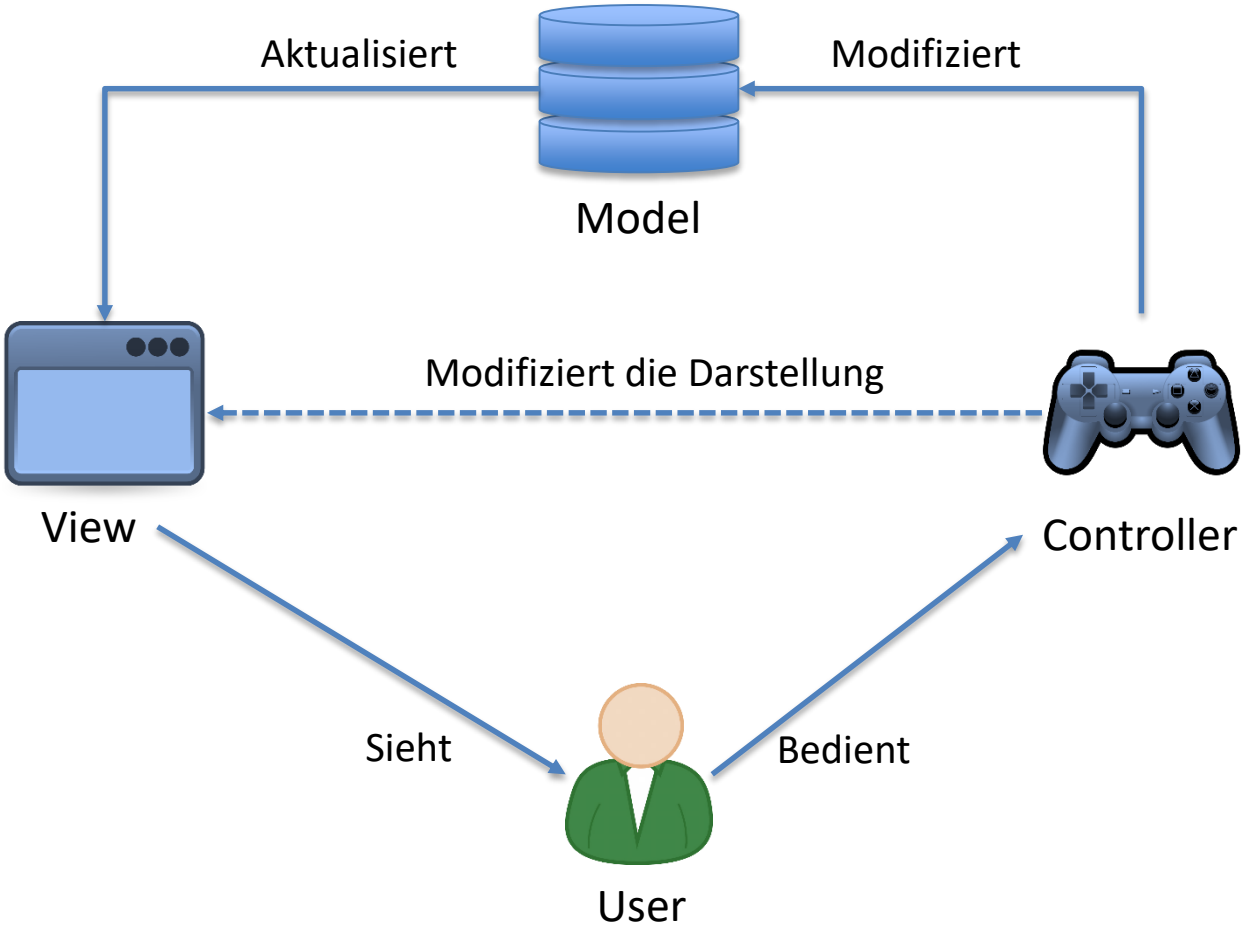
- Das Modell (**M**odel):
  - Ist zuständig für die gesamte Datenhaltung, also dem Zustand des Systems
    - Können über Nutzereingaben (Interaktionen) verändert werden
  - Auch sind hier häufig Regeln und Logik verankert (Backend)



- Die Ansicht (**V**iew):
  - Das ist das, was der Anwender letztendlich sieht (Frontend)
    - also Buttons, Textfelder, Fenster, usw.
  - Mehrere Anzeigen möglich:
    - Engl. vs. deutsche Ansicht, verschiedene Graphen für gleiche Daten, usw.

- Die Steuerung (**C**ontroller):
  - Über die Steuerung kann der Nutzer Einfluss auf die Daten im Model nehmen
    - Bspw.: Anzeigen oder manipulieren
  - Gängige Elemente der Steuerung sind `EventHandler` und `ActionListener`

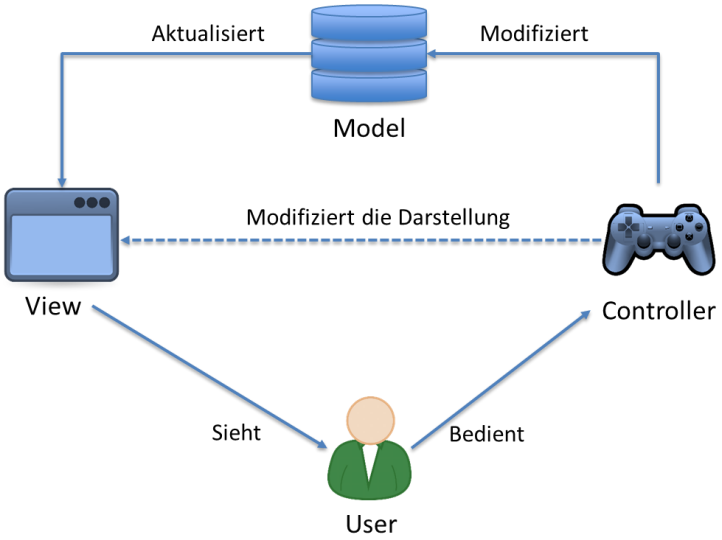
MVC: Logische Zusammenhänge



MVC: Objekt Zusammenhänge

Model:

- Unabhängig von View & Controller
- Aktualisierung der View über Beobachter

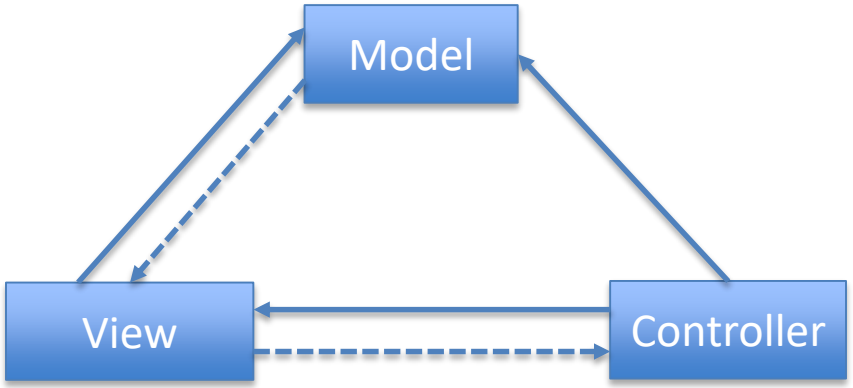


View:

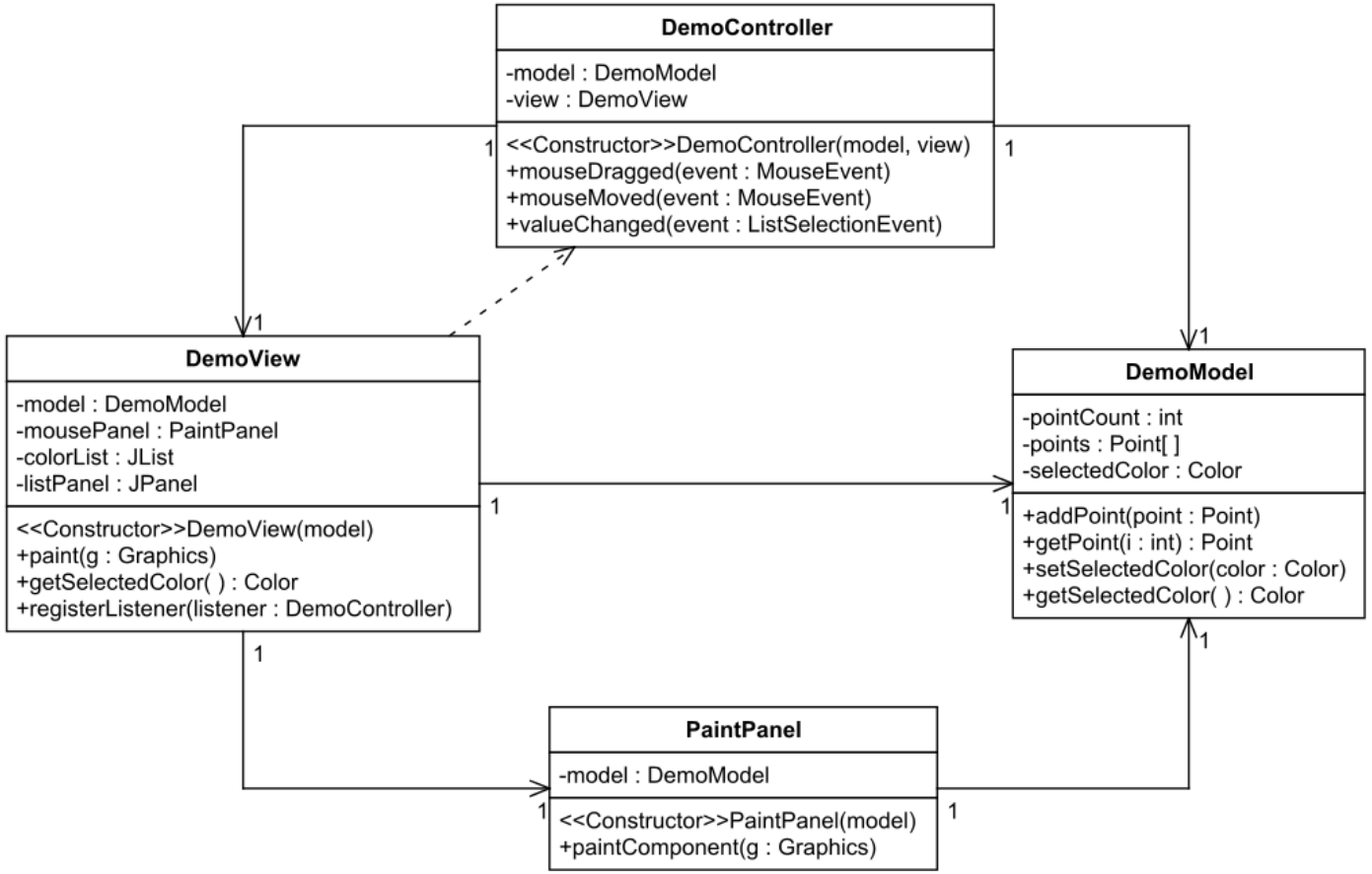
- Kennt ihre Steuerung und ihr Modell
- Wird bei Datenänderung benachrichtigt
- Braucht eine eigene Steuerung

Controller:

- Verwaltet mind. eine View
- Nimmt Interaktionen entgegen
- Bewirkt Änderungen im Model bzw. der View



MVC: Konkretes Beispiel als UML



Quelle: http://www.cs.utsa.edu/~cs3443/demomvc/demomvc-uml.png

MVC ist also ein sehr geläufiges Entwurfs- bzw. Architekturmuster zum Programmieren von grafischen Anwendungen.
Die Einhaltung der Prinzipien von MVC wird i.d.R. empfohlen!

MVC bei JavaFX

Bei JavaFX ist eine komplette Trennung zwischen View und Controller nicht immer leicht, da wir hier mit den Elementen des Frameworks arbeiten.

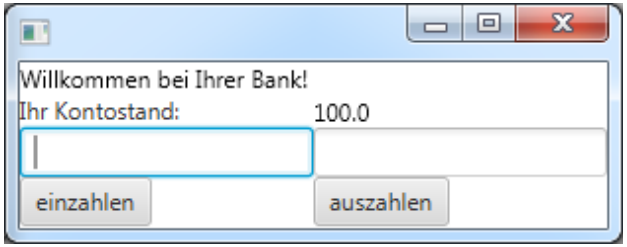
- Das Model sollte aber nach wie vor unabhängig und für die Datenhaltung zuständig sein
  - Bsp.: Bankkonto
    - Eigene Klasse
    - Datenhaltung
    - Keine Kenntnis über Model bzw. View (Keine Referenz)
- Die View (`Stage` und `Scene`) ist häufig mit dem Controller über anonymen innere Klassen (`EventHandler`) verbunden.
  - Beide kennen das Model
  - Zugriff über Controller

Aufgabe zu MVC

Schreiben Sie eine kleine Anwendung unter Berücksichtigung des MVC Musters:

Erstellen Sie ein neues JavaFX Projekt „MVC“ und generieren Sie 3 Klassen:

- **Anzeige**: Erbt von `Parent` und bildet die Anzeige (und den Controller) für eine einfache Bank-Anwendung, bei welcher der Nutzer Ein- und Auszahlungen mit Hilfe eines Textfeldes und den Buttons tätigen kann.
  - Der aktuelle Kontostand soll darüber angezeigt und bei jeder Ein- bzw. Auszahlung entspr. angepasst werden.



- **Bank**: Ist das eigtl. Model und besitzt eine Instanzvariable für den Kontostand (`double`)
  - Außerdem die Methoden `einzahlen(double Betrag), auszahlen(double Betrag)`
  - Zusätzlich einen Getter für die Instanzvariable des Kontostands
  - Im Konstruktor sollte ein Initialkontostand angegeben werden können.

- **Main**: Die Main-Klasse beinhaltet die `start(Stage primaryStage)` Methode. Sie erzeugt: eine Bankinstanz und die Anzeige, welche die GUI mit dem integrierten Controller enthält

Verknüpfen Sie nun die Teile, so dass Sie über Ihre GUI Ein- und Auszahlungen tätigen können, die sich im Model und in der Anzeige auswirken.

Erweiterung auf 2 oder mehrere Anzeigen ?!?

Was passiert nun, wenn Sie 2 Anzeigen in dem vorherigen Beispiel erzeugen und in der einen Anzeige den Kontostand erhöhen?

Problem: Die GUIs sind nicht konsistent in ihrer Anzeige!

- Müssen also benachrichtigt werden sobald eine Änderung in den Daten erfolgt!
- Abhilfe schafft hier:
  - das Observer Pattern (Kommt nächste Stunde)
  - Oder auch Properties (Kommt später)