LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Javakurs für Fortgeschrittene

Einheit 06: FXML, Properties & Binding

Kyrill Schmid
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Teil 1: FXML und Scene Builder

- Model einbinden

Teil 2: Properties & Binding

- Properties
- Reagieren auf Änderungen (ChangeListener)
- Binding

Praxis:

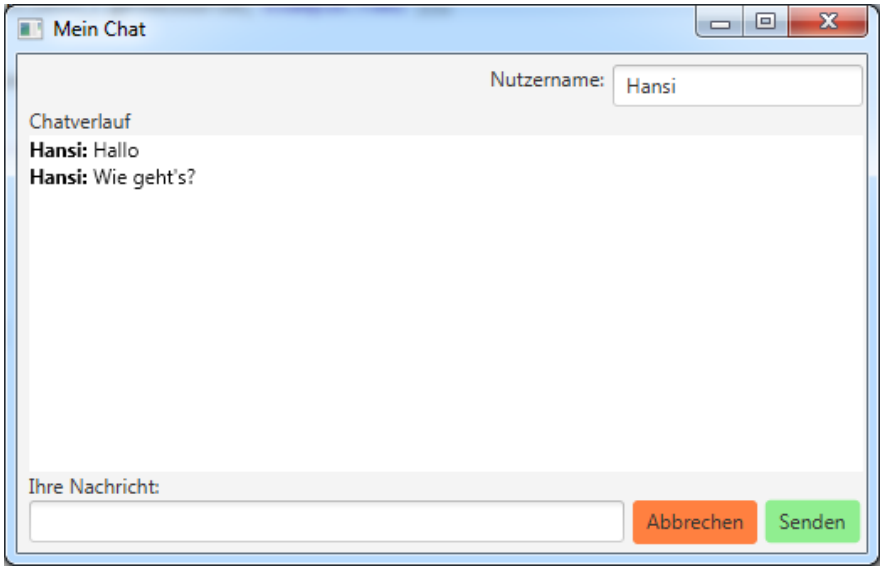
- Chat über Model
- Chat mit Properties
- Bank mit Zinsen

Lernziele

- Scene Builder und FXML in JavaFX nutzen können
- MVC Muster für JavaFX mit FXML anwenden
- Das Konzept von Properties & Bindig kennenlernen

Wiederholung: Aufgabe zu FXML mit Scene Builder

Nutzen Sie nun Ihr erlangtes Wissen über FXML und Scene Builder und gestalten Sie damit eine Chat-GUI, die ungefähr so aussehen könnte:



Schreiben Sie einen geeigneten Controller, der

- Nutzereingaben für Nachrichten und Nutzernamen entgegennimmt
- Beim Drücken des Senden-Buttons diese entsprechend der Grafik im Verlaufsfenster anzeigt.
- Beim Drücken des Abbrechen-Buttons den eingetragenen Text im Nachrichtenfeld wieder löscht

Achten Sie auf eine herkömmliche Usability, d.h.:

- Das Chat-Verlaufsfenster darf nicht direkt editiert werden können
- Beim Absenden einer Nachricht wird automatisch das Nachrichtenfeld geleert.

Ein Model für verschiedenen Controller

Will man sein Model für verschiedene Controller nutzen, muss man die Instanz dem Controller bekanntmachen (übergeben).

- Ein einfache Parameterübergabe mittels Konstruktor funktioniert hier nicht, da der Controller mittels FXML vom Framework erzeugt wird!
- Daher 2 Möglichkeiten:

1. Möglichkeit:
   - Die Controller-Instanz selbst erzeugen und nicht durch FXML
     - Ganz normal mittels `new Controller (param1, param2,…)`
   - und beim FXML-Loader bekannt machen:
     - `setController(myController);`

```
Model model = new Model();
FXMLLoader loader = new FXMLLoader(getClass().getResource("counter.fxml"));

Controller c = new Controller(model);
loader.setController(c);

Parent root = loader.load();
//…
```

Der Controller brauch natürlich einen entspr. Konstruktor!

Nicht vergessen: Im FXML File den Controller zu löschen!

Dann erst laden!

Ein Model für verschiedenen Controller

2. Möglichkeit:
   - Den Controller durch FXML erzeugen lassen
   - Nachdem der Controller geladen ist
     - die entsprechende Controller-Instanz durch getController() vom loader holen
     - Einen eigenen Setter zum Setzen des Models aufrufen

```
Model m = new Model();
FXMLLoader loader = new FXMLLoader(getClass().getResource("counter.fxml"));

// Zuerst laden:
Parent root = loader.load();

// Dann Controller-Instanz holen:
Controller c = loader.<Controller>getController();

// Dann eigenen Setter aufrufen:
c.setModel(m);
```

Chat-Fenster mit Logik erweitern

Schreiben Sie nun ein Model für Ihre zuvor erstellte Chat-Anwendung, welches die eingegebenen Nachrichten zusammen mit dem Nutzernamen speichert.

Verknüpfen Sie anschließend das Model mit 2 Chat-Anzeigen, so dass über dieses Model ein lokaler (einfacher) Chat realisiert werden kann.

Ihr Model soll ganz einfach gehalten sein:

- Es besitzt 2 ArrayLists für Nutzernamen und Nachrichten
- Es hat eine Methode, um neue Nachrichten + Nutzernamen zu speichern
  - Bsp.: `public void addMsg(String uname, String txt)`
- Es hat 2 Getter für die beiden ArrayLists

Wenn Sie nun eine Nachricht in das Chat-Fenster eingeben, soll diese Nachricht mit dem Nutzernamen im Model gespeichert werden.

- Für die Ausgabe müssen Sie nun auf den Inhalt der beiden ArrayLists im Model zugreifen und diese wie gewohnt anzeigen.

Problem

Wir haben wieder das Problem von inkonsistenten Anzeigen, obwohl das Model immer die richtigen Datenbestände hat.

- Wir brauchen wieder einen Mechanismus, wie das Observer-Pattern
- Dieses mal: **Properties & Binding**!

Teil 2: Properties & Binding

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+".");
      });
```

Beispielsweise um Textfelder zu updaten (Controller Beispiel)


```
public class Controller implements Initializable{

   private ModelwithProperties mwp;

   public Controller(){

      System.out.println("Klasse instanziiert");
   }

   @FXML private Text actionTarget;

   @FXML protected void handleIncreaseButton(){
      mwp.increment();
   }

   @FXML protected void handleDecreaseButton(){
      mwp.decrement();
   }

   @Override
   public void initialize(URL location, ResourceBundle resources) {

      this.mwp = new ModelwithProperties();

      mwp.counterProperty().addListener((obs,oldV,newV)->{
         this.actionTarget.setText(""+newV);
      });
   }
}
```

`ChangeListener` mit Property aus dem Model verbinden. Textfeld wird mit neuem Wert befüllt, sobald Wert geändert wurde.

Aufgabe zu Properties

Erweitern Sie nun Ihr Model aus der vorherigen Aufgabe um eine Property-Variabel, so dass die beiden Chat-GUIs darüber synchronisiert werden können.

- Greifen Sie in Ihrem Controller auf die neuen Property zu und reagieren Sie auf Veränderungen mittels einem geeigneten `ChangeListener`
  - **Hinweis**: Nutzen Sie einen einfachen Counter vom Typ `IntegerProperty`, der die Anzahl an Nachrichten in der `ArrayList` mitzählt.
    - Jedes Mal, wenn eine Nachricht in den Chat eingegeben wird, muss der Counter erhöht werden.
    - Daraufhin reagiert der `ChangeListener` und aktualisiert die GUIs um die entsprechende neue Nachricht+Nutzernamen