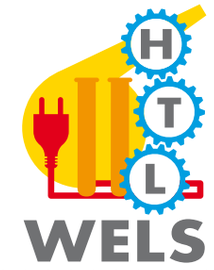
JAVAFX 8 – FORTGESCHRITTENE THEMEN

SEW

DI Thomas Helml

INHALT



- Properties & Bindings
- Concurrency in JavaFX

PROPERTIES & BINDINGS

PROPERTIES & BINDINGS

- Benutzeroberfläche
  - stellt Zustand von Datenobjekten dar
  - gibt Benutzer die Möglichkeit diesen Zustand zu ändern
- Bsp: Schieberegler für Breite eines Rechtecks
  - Wert auslesen
  - width des Models aktualisieren
  - Berechnung der Rechtecksfläche anstoßen
  - zeichnen

PROPERTIES

- Neu mit JavaFX: Properties

```
public class MyBean {

    private StringProperty sample = new SimpleStringProperty();

    public String getSample() {

        return sample.get();

    }

    public void setSample(String value) {

        sample.set(value);

    }

    public StringProperty sampleProperty() {

        return sample;

    }
}
```

PROPERTIES

- Einfache Properties (abstrakte Klassen):
  - BooleanProperty
  - DoubleProperty
  - FloatProperty
  - IntegerProperty
  - LongProperty
  - StringProperty

PROPERTIES

- Erzeugen von Properties über **konkrete** Klassen (Konstruktor mit maximalen Parametern):

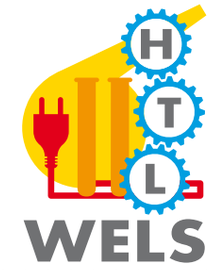
```
BooleanProperty booleanProperty = new
SimpleBooleanProperty(true, „b”, this);

DoubleProperty doubleProperty = new
SimpleDoubleProperty(1.5, „d”, this);

FloatProperty floatProperty = new
SimpleFloatProperty(1.5f, „f”, this);

IntegerProperty integerProperty = new
SimpleIntegerProperty(123, „i”, this);

LongProperty longProperty = new
SimpleLongProperty(12345678991, „l”, this);

StringProperty stringProperty = new
SimpleStringProperty("hallo", „s”, this);
```

PROPERTIES

- Object Properties
  - speichern beliebige Objekte

```
ObjectProperty<Image> objectProperty =
new SimpleObjectProperty<>();
```

BINDINGS



- Bindings „verknüpfen“ Properties mit Werten

```
label.textProperty().bind(myBean.sampleProperty());
```

- Binding lösen:

```
label.textProperty().unbind();
```

INITIALIZE

- Wo werden Bindings erstellt?
  - im Controller!
  - 2 Möglichkeiten:
    - `Interface Initializable`
    - `@FXML initialize`

INITIALIZE

- Variante 1:

```
public class Controller implements Initializable {

  @Override
  public void initialize(URL location, ResourceBundle resources) {
     // Bindings here
  }
  …
}
```

INITIALIZE

- Variante 2:

```
public class Controller {

  @FXML
  public void initialize() {
     // Bindings here
  }
  …
}
```

ÜBUNG PROPERTIES

- Übung: **204_SimpleBindings**

CALCULATED BINDINGS

- Bindings realisieren eine komplexe Beziehungen zwischen Werten (Properties):
  - Unterscheide: High-Level- und Low-Level-API
    - High-Level:
      - Bindings-Klasse
      - Fluent-API
  - Low-Level-API

BINDINGS

- Bindings-Klasse

```
DoubleProperty number1 = new SimpleDoubleProperty(1);

DoubleProperty number2 = new SimpleDoubleProperty(2);

DoubleProperty number3 = new SimpleDoubleProperty(3);
```

```
NumberBinding calculated = Bindings.add(

    number1, Bindings.multiply(number2,number3));
```

FLUENT API

- Fluent-API

```
DoubleProperty number1 = new SimpleDoubleProperty(1);

DoubleProperty number2 = new SimpleDoubleProperty(2);

DoubleProperty number3 = new SimpleDoubleProperty(3);
```

```
NumberBinding calculated =

        number1.add(number2.multiply(number3));
```

LOW LEVEL API

- Low-Level-API

```
DoubleProperty number1 = new SimpleDoubleProperty(1);
DoubleProperty number2 = new SimpleDoubleProperty(2);
DoubleProperty number3 = new SimpleDoubleProperty(3);


NumberBinding calculated = new DoubleBinding() {
    {
      super.bind(number1, number2, number3);
    }
    @Override
    protected double computeValue() {
      return number1.get() + (number2.get() * number3.get());
    }
};
```

NUMERISCHE BINDINGS

- Berechnungen mit numerischen Bindings:
  - `.add / .substract`
  - `.multiply / .divide`
  - `.negate`
  - `.min / .max`

BIDIREKTIONALE BINDINGS

- 2 Properties können gegenseitig aneinander gebunden werden:

```
DoubleProperty number1 = new SimpleDoubleProperty(1);
DoubleProperty number2 = new SimpleDoubleProperty(2);
number2.bindBidirectional(number1);
```

- Dann können Sie auch gebundene Properties gesetzt werden:

```
number2.setValue(3);
number1.setValue(4);
System.out.println(„number2 hat Wert: „ + number2.getValue());
```

ÜBUNG BINDINGS

- Übung: **205_CalculatedBindings**

OBJECT BINDINGS

- Object Bindings
  - mit ObjectBindings können beliebige Objekte an Properties gebunden werden
  - Vorgehensweise:
    - eigene Klasse ableiten von `ObjectBinding<T>`
    - Im `Konstruktor` entsprechendes Property annehmen
    - `T computeValue()` Methode implementieren, welches das entsprechende Objekt retourniert, das an Property gebindet wurde

OBJECT BINDINGS

```
public class ImageViewerBinding extends ObjectBinding<Image> {
    StringProperty p;


    public ImageViewerBinding(StringProperty property) {
        super.bind(property);
        p = property;
    }

    @Override
    protected Image computeValue() {
      try {
          Image image = new Image(p.get(), true);
          return image;
       } catch (Exception e) {
       }
       return null;
    }
}
```

➤

ÜBUNG BINDINGS

- Übung: **206_ImageViewer**

BOOLEAN BINDINGS

- Boolean Bindings: Logische Verknüpfung von BooleanProperty
  - .greaterThan / .greaterThanOrEqualTo
  - .isEqualTo / .isNotEqualTo
  - .lessThan / .lessThanOrEqualTo
  - .and / .or
  - isNotEmpty
  - …

BOOLEAN BINDINGS

- Bsp:
  - in einem Textfield überprüfen, ob Eingabe mit mindestens 3 Zeichen:

```
BooleanBinding textFieldEntered =
   textField.textProperty()
   .isNotEmpty()
   .and(textField.textProperty().length().greaterThan(3));
```

  - Button soll deaktiviert werden, wenn im Textfield nichts steht

```
button.disableProperty().bind(textFieldEntered.not());
```

ÜBUNG BINDINGS

- Übung: **207_BooleanBindings**

SERIALISIERUNG VON PROPERTIES

- Properties können NICHT serialisiert werden
  - daher müssen sie in dem Fall mit transient gekennzeichnet werden
  - sie werden dann aber auch NICHT gespeichert!

```
public class MyBean implements Serializable {

    private transient StringProperty sample = new SimpleStringProperty();

    public String getSample() {
        return sample.get();
    }

    public void setSample(String value) {
        sample.set(value);
    }

    public StringProperty sampleProperty() {
        return sample;
    }
}
```

SERIALISIERUNG VON PROPERTIES

- Möchte man Datenobjekte mit Properties serialisieren, dann muss man muss man die Serialisierung selbst in die Hand nehmen (z.B. StringProperty):

```
public class xyz implements Externalizable {

    private SimpleStringProperty x = new SimpleStringProperty("");

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        setX ((String) in.readObject());
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(getX());
    }
```

KLASSEN MIT PROPERTIES SERIALISIEREN

```java
public class Packet implements Externalizable {
    private static final long serialVersionUID = -8256294089416034037L;
    private SimpleStringProperty varName = new SimpleStringProperty("");
    private SimpleStringProperty varValue = new SimpleStringProperty("");

    public Packet() {
        this("", "");
    }
    public Packet(String varName, String varValue) {
        setVarName(varName);
        setVarValue(varValue);
    }
    public String getVarName() {
        return varName.get();
    }
    public void setVarName(String var) {
        varName.set(var);
    }

    public String getVarValue() {
        return varValue.get();
    }
    public void setVarValue(String value) {
        varValue.set(value);
    }
    public SimpleStringProperty getVarNameProperty() {
        return varName;
    }
    public SimpleStringProperty getVarValueProperty() {
        return varValue;
    }

    @Override
    public String toString() {
        return getVarName() + ": " + getVarValue();
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        setVarName((String) in.readObject());
        setVarValue((String) in.readObject());
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(getVarName());
        out.writeObject(getVarValue());
    }
}
```

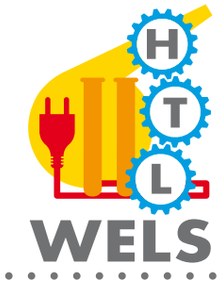
BINDINGS VON COLLECTIONS

- Beispiel 208_ListBindings
  - `ListView` verfügt über `itemsProperty` an welches ein `ListProperty` „gebindet“ werden kann
  - `ObserverableArrayList` ist eine spezielles `ListProperty`, welches „beobachtbar“ ist
  - `ObservableArrayList` kann über Hilfsklasse `FXCollections` aus einer Standard-Collection erzeugt werden

BINDINGS VON COLLECTIONS

- Beispiel 208_ListBindings

CONCURRENCY



CONCURRENCY

- JavaFX Application Thread darf unter keinen Umständen blockiert werden
  - „Eingefrorene“ Anwendung
  - Ereignisbehandlung reagiert nicht mehr
  - sämtliche Rechenintensiven Tätigkeit müssen in Threads ausgelagert werden

INTERFACE WORKER

CONCURRENCY – WORKER 1

- `Interface` ***`Worker`***
  - Interface (Basis für alle Concurrent Klassen)
  - `Worker` Objekte erledigen Arbeit in Hintergrund-Thread
  - Zustände können abgefragt werden:
    - `Worker.State.READY`
    - `Worker.State.SCHEDULED`
    - `Worker.State.RUNNING`
    - `Worker.State.CANCELLED`
    - `Worker.State.SUCCEEDED`
    - `Worker.State.FAILED`

CONCURRENCY – WORKER 2

- `Worker`
  - Fortschritt über Properties beobachtbar:
    - `totalWork`: DoubleProperty, Maximalwert für die Arbeit
    - `workDone`: DoubleProperty, Anteil an erledigter Arbeit
    - `progress`: Wert zw. 0 und 1 (prozentueller Anteil)

TASK

TASK

- Klasse ***Task***
  - Implementierung des `Worker` Interface
  - für **einmalige** Hintergrundberechnungen (abgeleitet von `FutureTask`)
- Task implementiert `Runnable`, somit auch Start über `Executor` möglich
- Ergebnis der Berechnung mit Methode `get()`, wenn Berechnung zu Ende ist
  - Berechnung noch nicht am Ende: `get()` blockiert
- in `call()` Methode wird Arbeit verrichtet und Properties aktualisiert

TASK

- Fortschritt aktualisieren:
  - in `call()` wird Methode `updateProgress` aufrufen
  - über `task.progressProperty()` kann ein Binding auf z.B. eine Progressbar realisiert werden

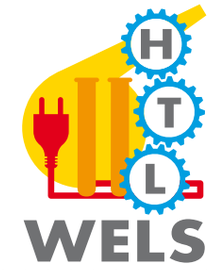
TASK

- Tasks unterbrechen
  - vgl. „interrupt“ in Threads
  - im Controller wird mit `myTask.cancel()` versucht den Task zu beenden
  - bei Tasks: in Methode „`call`“ prüfen auf **`isCancelled()`**

- Aktionen nach Beendigung des Tasks:

```
task.setOnSucceeded((WorkerStateEvent event) -> {
    Object value = task.getValue();
    // do anything with the result
    updateTheUI(value);
});
// setOnFailed
// setOnScheduled
// setOnCanceled
// setOnRunning
```

TASK



- TODO:
  - Lies dir die API Dokumentation zu Tasks
  - Finde heraus, wie man Tasks ohne/mit Parameter/mit Rückgabewert implementiert!

CONCURRENCY – TASK

- Bsp.: 209_SimpleTask

SERVICE

CONCURRENCY – SERVICE 1

- Klasse **`Service`**
  - verwaltet einen `Task`
  - Tasks können über Service mehrfach ausgeführt werden
  - Task ohne Service kann nur 1x ausgeführt werden!
- Klasse **`ScheduledService`**
  - führt Tasks in vorbestimmten Intervallen

CONCURRENCY – SERVICE 1

- Klasse **`Service`** - wichtige Methoden:
  - `start()` - startet den Service
  - `reset()` - resettet den Service, funktioniert aber nur, wenn Thread in finished Status ist (SUCCEEDED, FAILED, CANCELLED, READY)
  - `restart()` - laufender Thread wird gecancelt und dann neu gestartet
  - `cancel()` - canceled laufenden Thread

CONCURRENCY – SERVICE 2

➤ Bsp. Service: Task definieren

```
public CounterTask extends Task<Integer>{
    public CounterTask(int max) {
      this.max = max;
      updateMessage("Ready to count...");
    }
    @Override protected Integer call() throws Exception {
      updateMessage("Counting...");
      for (int i = 0; i < max; i++) {
        Thread.sleep(10);
        updateProgress(i, max);
      }
      updateMessage("READY");
      return max;
    }
}
```

CONCURRENCY – SERVICE 3

➤ Service definieren:

```
public class CounterService extends Service<Integer>{
    private final int max;
    public CounterService(int max) {
      this.max = max;
    }
    @Override
    protected Task<Integer> createTask() {
      return new CounterTask(max);
    }
}
```

CONCURRENCY – TASK 7

- Bsp.: 210_SimpleService

SCHEDULEDSERVICE

CONCURRENCY – SCHEDULEDSERVICE 1

- `ScheduledService`
  - führt Tasks in vorbestimmten Intervallen wieder aus
  - Ändere im vorigen Beispiel folgende Zeile und es wird der Task immer wieder ausgeführt:

```
public class CounterService extends
Scheduled
```

CONCURRENCY – SCHEDULEDSERVICE 2

- Verzögerung des Restarts um 2 Sekunden

```
public CounterService(int max) {
    super();
    setPeriod(Duration.seconds(2));
    this.max = max;
}
```

CONCURRENCY – SCHEDULEDSERVICE 3

- Bei `Task/Service` Wert (z.B. in GUI) an `valueProperty` binden - somit ist Wert immer aktuell

```
Label label = new Label();
label.textProperty().bind(Bindings.concat("Value: ",
        counterService.valueProperty()));
```

CONCURRENCY – SCHEDULEDSERVICE 4

- bei `ScheduledService` wird `valueProperty` regelmäßig `null` sein, da der `Service` immer wieder neu gestartet wird (und somit der Wert zurückgesetzt wird)
  - daher gibt es die Property `lastValue`

```
label.textProperty().bind(Bindings.concat("Value: ",
counterService.lastValueProperty()));
```

CONCURRENCY – SCHEDULEDSERVICE 5

- Was passiert im Fehlerfall? Server nicht erreichbar, …

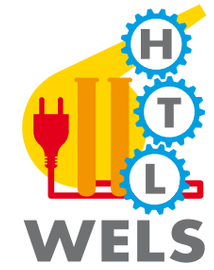
```
protected Integer call() throws Exception {
  updateMessage("Counting...");
  for (int i = 0; i < max; i++) {
    Thread.sleep(10);
    updateProgress(i, max);
  }
  if (max>=3) throw new Exception("Das ist zu
kompliziert!");
   updateMessage("READY");
   return max;
  }
```

CONCURRENCY – SCHEDULEDSERVICE

- Abbruch bei Misserfolg
  - Service muss manuell wieder gestartet werden!

```
counterService.setRestartOnFailure(false);
counterService.start();
```

CONCURRENCY – SCHEDULEDSERVICE



- Festlegen, wie oft es der Service im Fehlerfall versuchen soll:

```
counterService.setRestartOnFailure(true);
counterService.setMaximumFailureCount(3);
counterService.start();
```

CONCURRENCY – SCHEDULEDSERVICE

- Nach Fehler ist es meist nicht sinnvoll es sofort neu zu versuchen
- Daher unterschiedliche Strategien:
  - LOGARITHMIC_BACKOFF_STRATEGY
  - EXPONENTIAL_BACKOFF_STRATEGY
  - LINEAR_BACKOFF_STRATEGY

```
counterService.setRestartOnFailure(true);
counterService.setMaximumFailureCount(3);
counterService.setBackoffStrategy(
        ScheduledService.EXPONENTIAL_BACKOFF_STRATEGY );
counterService.start();
```

CONCURRENCY – PROGRESS ANZEIGEN

```
Task task = new Task<Void>() {
    @Override public Void call() {
        static final int max = 1000000;
        for (int i=1; i<=max; i++) {
            if (isCancelled()) {
                break;
            }
            updateProgress(i, max);
        }
        return null;
    }
};
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

PLATFORM.RUNLATER

PLATFORM.RUNLATER

- Soll eine GUI Komponente von einem Nicht-GUI-Thread heraus modifiziert werden, so kann `Platform.runLater` verwendet werden

```
public static void runLater(Runnable runnable)
```

  - die Aufgabe wird in den GUI Thread eingereiht und frühest möglich abgearbeitet
  - kleinere Aufgaben können ebenso mit `Platform.runlater()` realisiert werden
  - größere/rechenintensivere Aufgaben mittels Threads!

PLATFORM.RUNLATER

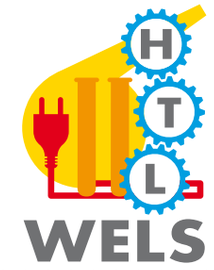
- Annahme: Eine ListView wird über ein Property an einen Service „gebindet“

```
...
listView.itemsProperty().bind(myListService.resultProperty());
...
```

  - in dem Fall muss eine Änderung der ListView über runlater realisiert werden:

```
...
Platform.runLater(() -> result.add("Element " + finalI));
...
```

JAVA DOCUMENTATION



- Java Dokumentation für Concurrency in JavaFX:
  - https://docs.oracle.com/javase/8/javafx/interoperability-tutorial/concurrency.htm#JFXIP546
-