JavaFX:
Ereignisse, Properties, Animationen, Tasks, CSS, FXML, Scene Builder

Ereignisgesteuerte Programmierung

Beim Start einer JavaFX-Anwendung wird die Methode `start` aufgerufen, um die GUI-Elemente zu erzeugen.

Danach wird eine **Event-Loop** ausgeführt.

Pseudocode:

```
while (nicht alle Fenster geschlossen) {

  Warte auf Ereignis (Maus bewegt, Taste gedrueckt,
                      Timer abgelaufen,...);

  Informiere Steuerelemente von eingetretenem Ereignis;

  Aktualisiere die Anzeige;

}
```

Wiederholung

Ereignisse werden durch Objekte der Klasse `Event` repräsentiert.

- physikalische Events, z.B. Maus bewegt
  - `InputEvent extends Event`
  - `MouseEvent extends InputEvent`
  - `ScrollEvent extends InputEvent`
  - `KeyEvent extends InputEvent`
  - `...`
- logische Events, z.B. Button betätigt
  - `ActionEvent extends Event`
  - `...`

Wiederholung

Für die Behandlung von Ereignissen verwendet man Objekte, die das Interface `EventHandler` implementieren.

```
class SayClickHandler implements EventHandler<MouseEvent>

  @Override
  public void handle(MouseEvent event) {
    System.out.println("Click!");
  }

}
```

Wiederholung

Die Java 8-Notation

```
myButton.setOnAction(event -> {
    System.out.println("Click!");
});
```

hat die gleiche Bedeutung wie

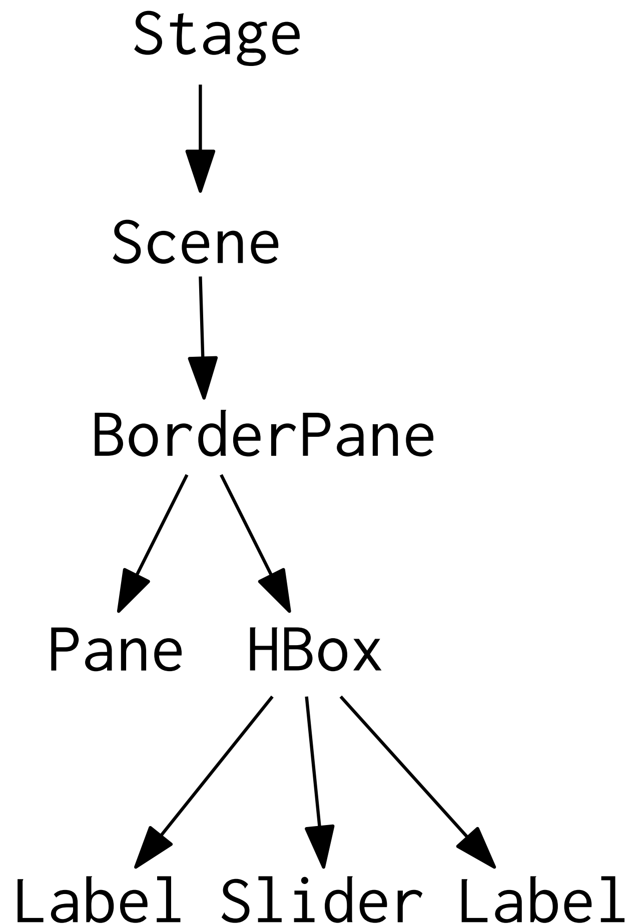
```
myButton.setOnAction(new SayClickHandler());
```

Man braucht also nicht extra eine Klasse zu definieren und kann den Code der einzigen Methode gleich an Ort und Stelle angeben.

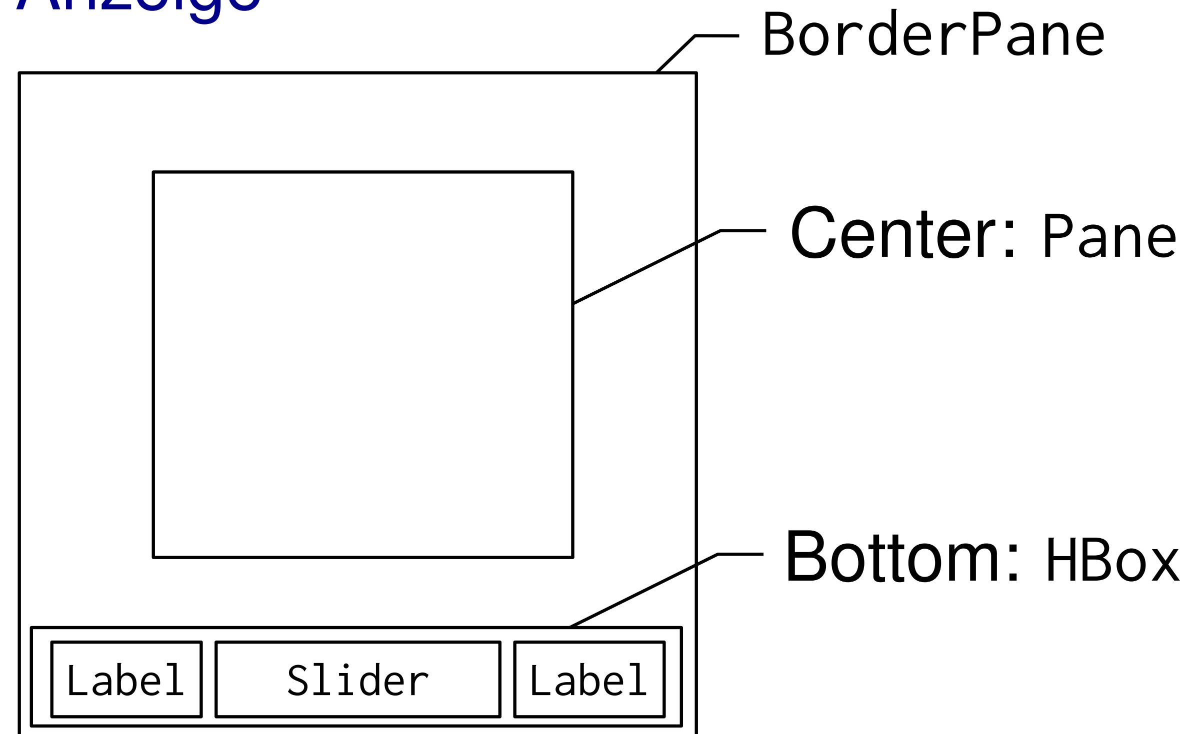
Ablauf der Ereignisbehandlung

Beispiel: Ablauf der Ereignisbehandlung bei Klick auf Slider.

Objektdiagramm

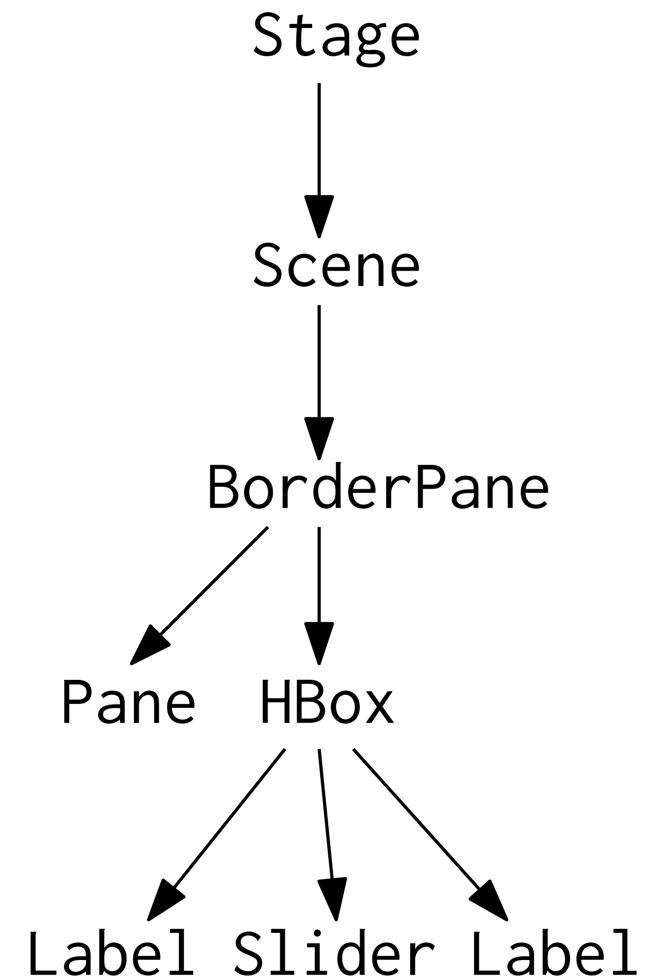


Anzeige



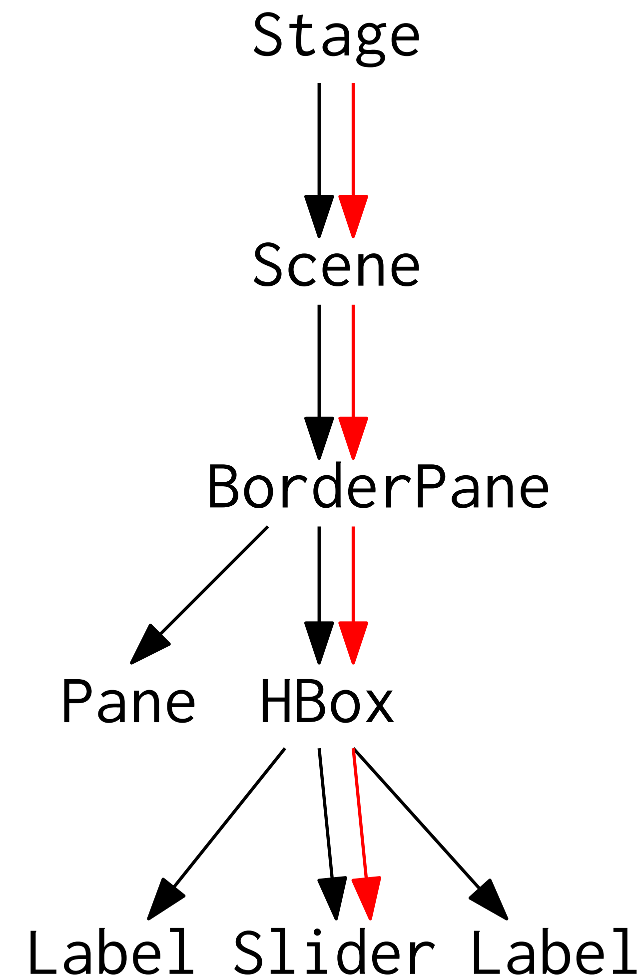
Ablauf der Ereignisbehandlung

- Informationen über den Mausklick werden in ein `MouseEvent`-Objekt kodiert



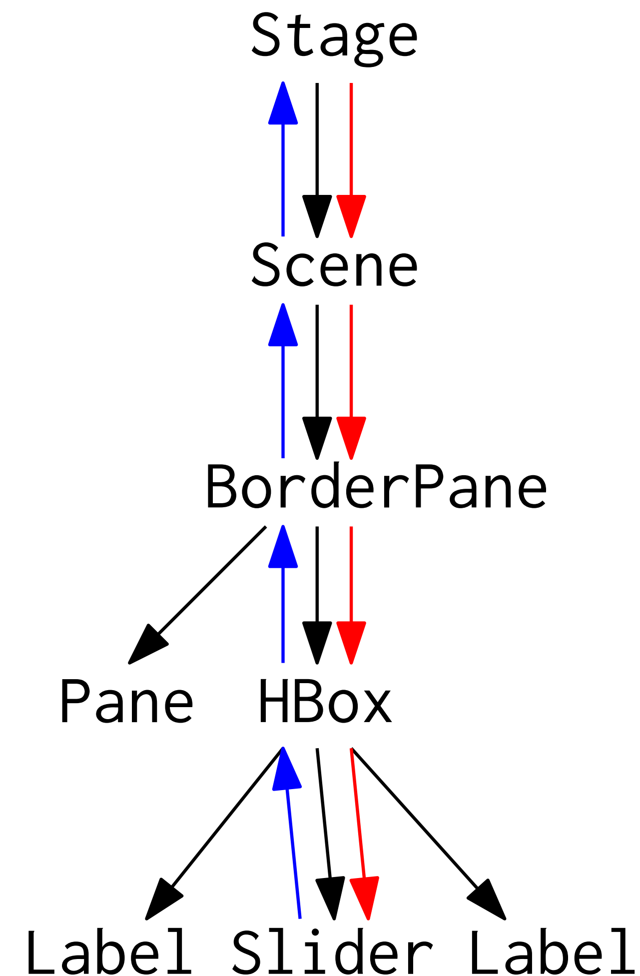
Ablauf der Ereignisbehandlung

- Informationen über den Mausklick werden in ein `MouseEvent`-Objekt kodiert
- Der `MouseEvent` wird zum Zielobjekt (der geklickte Slider) durchgereicht. In jedem Schritt werden **Event-Filter** angewendet. (Beispiel: Verschlucken des Events)



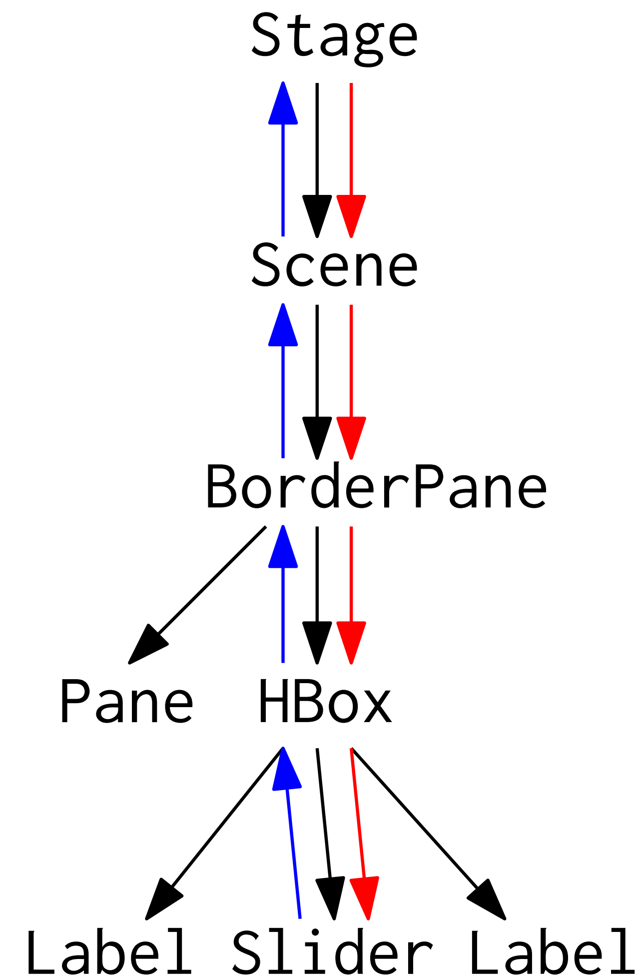
Ablauf der Ereignisbehandlung

- Informationen über den Mausklick werden in ein `MouseEvent`-Objekt kodiert
- Der `MouseEvent` wird zum Zielobjekt (der geklickte Slider) durchgereicht. In jedem Schritt werden **Event-Filter** angewendet. (Beispiel: Verschlucken des Events)
- Das Event-Objekt wird wieder nach oben durchgereicht. In jedem Schritt werden **Event-Handler** aufgerufen. (Beispiel: Ändern des Slider-Werts)



Ablauf der Ereignisbehandlung

- Informationen über den Mausklick werden in ein `MouseEvent`-Objekt kodiert
- Der `MouseEvent` wird zum Zielobjekt (der geklickte Slider) durchgereicht. In jedem Schritt werden **Event-Filter** angewendet. (Beispiel: Verschlucken des Events)
- Das Event-Objekt wird wieder nach oben durchgereicht. In jedem Schritt werden **Event-Handler** aufgerufen. (Beispiel: Ändern des Slider-Werts)
- Event-Filter und -handler können beliebig vom Benutzer definiert werden.



Event-Handler

Jedes `Node`-Objekt hat eine Liste von aktiven Event-Handlern.

- `addEventHandler` fügt einen Event-Handler zur Liste hinzu.

```
Pane pane = new Pane();
EventHandler<MouseEvent> h = new SayClickHandler();
pane.addEventHandler(MouseEvent.MOUSE_CLICKED, h);
```

- `removeEventHandler` entfernt aus der Liste.

Event-Filter

Jedes `Node`-Objekt hat eine Liste von aktiven Event-Filtern.

- `addEventFilter` fügt einen Event-Filter zur Liste hinzu.

  Beispiel: Ignorieren aller Tastaturereignisse

```
Pane pane = new Pane();
pane.addEventFilter(KeyEvent.ANY, new ForgetFilter());
```

```
class ForgetFilter implements EventHandler<KeyEvent> {

  @Override
  public void handle(KeyEvent event) {
    // Verschlucke den event. Er wird nicht an die Kinder weitergeleite
    event.consume();
  }
}
```

Event-Filter unterscheiden sich von Event-Handlern nur im Zeitpunkt der Ausführung.

Informationen in Events

Bei der Ausführung eines Event-Handlers oder Event-Filters erhält man Informationen über den Event.

Jedes Event-Objekt hat Source und Target:

- Source: Das Node-Objekt, auf dem der Event gerade behandelt wird.
- Target: Das Node-Objekt am Ende der Ereigniskette.

Koordinaten:

- `getX`, `getY`: Koordinaten in Bezug auf das Source-Objekt.
- `getSceneX`, `getSceneY`: Koordinaten in Bezug auf die gesamte Scene.

Hinweise

Alle Event-Handler und Event-Filter werden von der Event-Loop aufgerufen.

- Die Ereignisbehandlung erfolgt *nicht* nebenläufig.
  Grund: Synchronisierung aufwendig und fehleranfällig.
  (Praktisch alle GUI-Bibliotheken verwenden nur einen einzigen Ausführungsthread für die GUI.)
- $\Rightarrow$ Wenn ein Event-Handler lange für die Berechnung braucht, dann blockiert das Programm.

Ablauf der Ereignisbehandlung

- Ein aufgetretenes Ereignis wird als `Event`-Objekt kodiert.
- Das `Event`-Objekt muss zu seinem Zielobjekt (z.B. die Node, auf die geklickt wurde) geschickt werden.
- Dazu wird eine Dispatch-Kette zum Zielobjekt erstellt.
  Das ist normalerweise der Pfad von der Wurzel (Stage) zum Zielobjekt.
- Das Event-Objekt wird entlang der Dispatch-Kette zum Zielobjekt durchgereicht. In jedem Schritt werden die `handle`-Methoden der Event-Filter des erreichten Objekts aufgerufen.
- Das Event-Objekt wird wieder nach oben durchgereicht. In jedem Schritt werden die `handle`-Methoden der Event-Handler des erreichten Objekts angewendet.

JavaFX: Properties

Properties

JavaFX verwendet das Observer-Pattern für alle Werte, die sich ändern können.

Änderbare double-Werte werden zum Beispiel in `DoubleProperty`-Objekten gespeichert.

Die Klasse `DoubleProperty` kapselt einen einzigen `double`-Wert und implementiert das Observer-Pattern.

- `void set(double value);`
- `double get();`
- `void addListener(ChangeListener<? super Double> l);` wobei

```
public interface ChangeListener<T> {
  void changed(ObservableValue<? extends T> observable
               T oldValue, T newValue);
}
```

Properties

Beispiel: Automatisches Resizing eines Canvas, wenn sich die Größe einer Pane ändert.

Die Methoden `widthProperty()` und `heightProperty()` in `Node` liefern jeweils eine `DoubleProperty`.

```
pane.widthProperty().addListener(a -> {
  canvas.setWidth(pane.getWidth());
  draw(canvas); // eigene Methode zum Neuzeichnen
});

pane.heightProperty().addListener(a -> {
  canvas.setHeight(pane.getHeight());
  draw(canvas); // eigene Methode zum Neuzeichnen
});
```

Properties

Beispiel: Einfache Behandlung von Mausereignissen.

Die Methode `hoverProperty()` in `Node` liefert eine `ReadOnlyBooleanProperty` (ohne Setter).

Verwendung:

```
Circle circle = new Circle(20);

circle.hoverProperty().addListener(
  (obs, oldValue, newValue) -> {
    if (newValue) {
      circle.setFill(Color.RED);
    } else {
      circle.setFill(Color.BLUE);
    }
});
```

Properties

Direkte Verbindungen zwischen Werten können kompakt hingeschrieben werden.

Beispiel: Der Wert einer Property kann an den einer anderen gebunden werden.

```
canvas.widthProperty().bind(pane.widthProperty());
```

Dieses Binding hat den gleichen Effekt wie:

```
pane.widthProperty().addListener(event -> {
  canvas.widthProperty().set(pane.widthProperty().get()
});
```

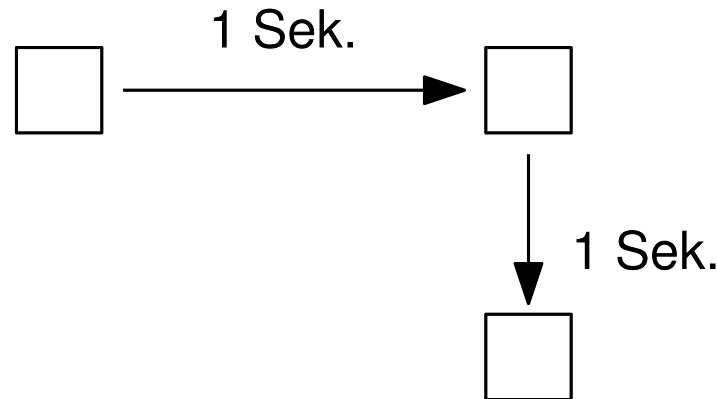
Bidirektionale Bindings:

```
Bindings.bindBidirectional(p1, p2);
```

JavaFX: Animationen

Timeline

Beispiel: Verschieben eines Rechtecks (Anfangsposition (0,0))



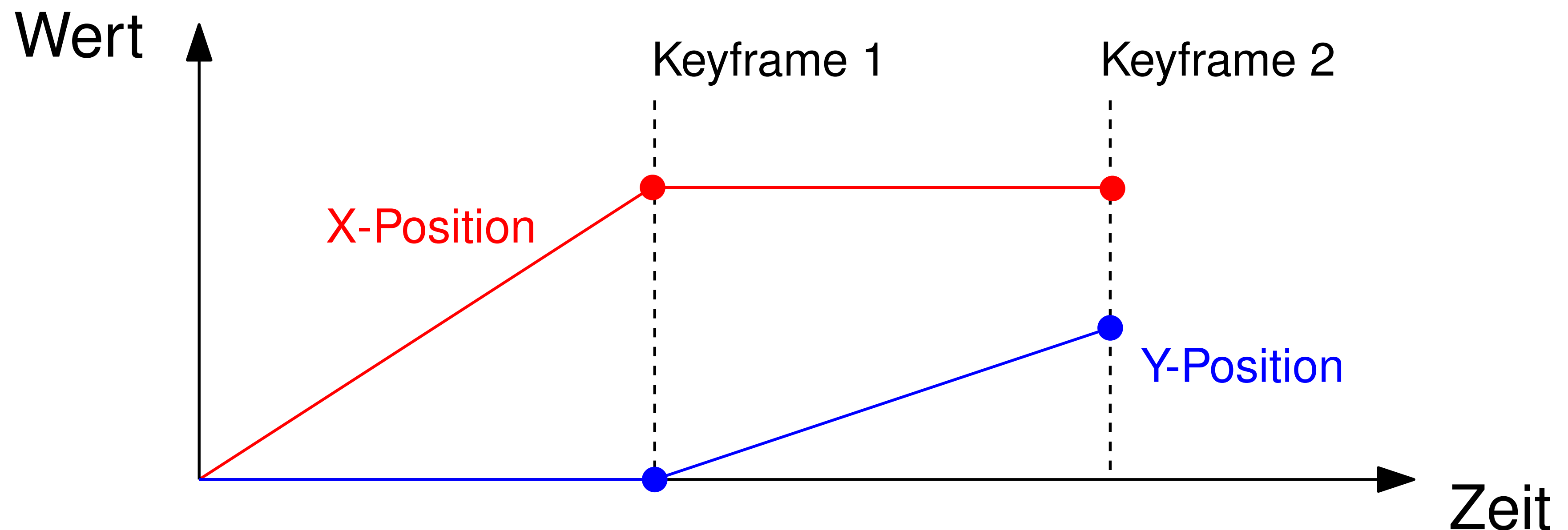
Idee: Spezifikation der Position zu bestimmten Zeitpunkten, automatische Ausführung

- Wert der X-Position nach 1 Sekunde: 100
- Wert der Y-Position nach 1 Sekunde: 0
- Wert der X-Position nach 2 Sekunden: 100
- Wert der Y-Position nach 2 Sekunde: 50

Timeline

Spezifikation mit einer Folge von Keyframes:

- ein Keyframe definiert die Zielwert verschiedener Properties zu einem bestimmeten Zeitpunkt
- mehrere Keyframes werden zu einer Timeline zusammengesetzt
- automatische Animation durch Interpolation, z.B. linear



Timeline

In JavaFX:

```
// nach einer Sekunde
KeyFrame f1 = new KeyFrame(Duration.seconds(1),
    new KeyValue(rectangle.translateXProperty(), 100),
    new KeyValue(rectangle.translateYProperty(), 0));

// nach zwei Sekunden
KeyFrame f2 = new KeyFrame(Duration.seconds(2),
    new KeyValue(rectangle.translateXProperty(), 100),
    new KeyValue(rectangle.translateYProperty(), 50));

// beide Keyframes in eine Timeline zusammensetzen
Timeline tl = new Timeline();
tl.getKeyFrames().addAll(f1, f2);

// Eigenschaften der Timeline setzen:
tl.setCycleCount(Timeline.INDEFINITE); // endlos wiederholen
tl.setAutoReverse(true); // Animation nach Ende umgedreht wiederholen

// Animation starten
tl.play();
```

Timeline

Keyframes werden durch Objekte der Klasse `KeyFrame` repräsentiert.

Jedes `KeyFrame`-Objekt hat eine Liste von Werten.

- Werte sind `KeyValue`-Objekte.
  Beispiel: Spezifikation des Zielwerts für eine Property

  ```
  new KeyValue(rectangle.translateXProperty(), 100)
  ```

  “Die Property der X-Position soll den Wert 100 haben.”
- Die Liste aller Werte eines `KeyFrame` erhält man mit `keyframe.getValues()` und kann dazu Werte hinzufügen.
- Werte können auch direkt im Konstruktor von `KeyFrame` übergeben werden.

Vorgefertigte Animationen

Die Klasse `Transition` und ihre Unterklassen stellen vorgefertigte Animationen zur Verfügung.

- `PathTransition`: Bewegung einer Node entlang eines Pfades
- `FadeTransition`: Ausblenden einer Node
- `StrokeTransition`: Veränderung der Linienfarbe
- …

Wiederholte Aktionen

Die Werte in einem `KeyFrame` können nicht nur Zielwerte für Properties sein.

Das Erreichen eines Keyframe ist ein logisches Ereignis, auf das man mit einem `EventHandler` reagieren kann.

Beispiel: Aufruf einer Methode alle 400ms.

```
EventHandler<ActionEvent> handler = new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent event) {
      // wird alle 400 Millisekunden ausgefuehrt
      ...
    }
  };
KeyFrame f = new KeyFrame(Duration.millis(400), handler);
Timeline timer = new Timeline(f);
timer.setCycleCount(Timeline.INDEFINITE);
timer.play();
```

Nebenläufigkeit

Nebenläufige Berechnungen

Erinnerung: Wenn ein Event-Handler lange für die Berechnung braucht, dann blockiert das Programm.

Aufwändige Berechnungen sollten nebenläufig zum GUI-Programm ausgeführt werden.

- Java erlaubt die nebenläufige Abarbeitung mehrerer Ausführungsstränge (Threads).
- Die Datenstrukturen in JavaFX sind aber nur auf Zugriffe aus dem eigenen JavaFX-Thread ausgelegt.
- Zugriffe auf JavaFX-Daten aus anderen Threads führen zu Fehlern.

JavaFX-Tasks sind eine vorgefertigte Möglichkeit zur kontrollieren nebenläufigen Programmabarbeitung.

Tasks

Die Klasse `Task<T>` kapselt Berechnungen, die nebenläufig ausgeführt werden sollen und die einen Wert vom Typ `T` als Ergebnis liefern.

Solche Tasks können parallel zur GUI berechnet werden.

Beispiel:

```java
public class Expensive extends Task<Integer> {
  public Integer call() {
    int i = 0;
    int sum = 0;
    while (!isCancelled() && i < 10000000) {
      sum += someSlowFunction(i);
      i++;
    }
    return sum;
  }
}
```

Tasks

Tasks werden so gestartet, dass die Methode `call` von einem nebenläufigen Thread ausgeführt wird.

Mit nebenläufigen Ausführungsthreads ist Vorsicht geboten.

- In der Methode `call` darf man nicht auf JavaFX-Steuerelemente oder Instanzvariablen anderer Klassen zugreifen.
- Umgekehrt darf man nicht ohne Synchronisierung auf Instanzvariablen eines Task-Objekts zugreifen.
  Der direkte Zugriff vom JavaFX-Thread auf eine Instanzvariable (z.B. über Getter-Methode) wäre nicht korrekt.
- Zugriff sollte *nur* über die bereits existierenden Methoden von `Task<T>` erfolgen.

Verwendung von Tasks

- Nebenläufigen Task im JavaFX-Programm starten:

```
Task<Integer> task = new Expensive();
new Thread(task).start();
```

- Abfragen des Werts (`null` wenn noch nicht fertig):

```
Integer i = task.getValue();
```

- Reagieren auf erfolgreiche Beendigung:

```
task.onSucceeded(event -> {
  Integer i = task.getValue();
  // Ergebnis der Berechnung ist i.
  // Hier koennen JavaFX-Methoden verwendet werden.
  label.setText("Ergebnis: " + i);
});
```

Verwendung von Tasks

- Task abbrechen:

  ```
  task.cancel();
  ```

  Diese Anweisung markiert den Task als abgebrochen. Die Methode `task.isCancelled()` gibt danach **`true`** zurück.

  Die eigentliche Berechnung wird aber nicht abgebrochen.

  Der nebenläufige Berechnungsthread wird erst beendet wenn die `call`-Methode einen Wert zurückgibt. Deshalb muss dort immer wieder `isCancelled()` abgefragt werden.

Tasks mit Anfangswerten

Anfangswerte können über einen Konstruktor übergeben werden.

```java
public class Expensive extends Task<Integer> {
  private int n; // keine Getter oder Setter!

  public Expensive(int n) {
    this.n = n;
  }
  public Integer call() {
    int i = 0;
    int sum = 0;
    while (!isCancelled() && i < n) {
      sum += someSlowFunction(i);
      i++;
    }
    return sum;
  }
}
```

Tasks mit Anfangswerten

- Nebenläufigen Task mit Anfangswerten starten:

```
int n = ...;
Task<Integer> task = new Sum(n);
new Thread(task).start();
```

- Abfragen des Ergebniswerts wie vorher.

Lange Berechnungen

Man kann schon während der Berechnung Zwischenergebnisse zurückgeben.

```
public class Expensive extends Task<Integer> {
  public Integer call() {
    int i = 0;
    int sum = 0;
    while (!isCancelled() && i < 10000000) {
      sum += someSlowFunction(i);
      updateValue(sum); // setze das Zwischenergebnis
      i++;
    }
    return sum;
  }
}
```

Dann gibt `task.getValue()` schon ein Zwischenergebnis zurück.

Lange Berechnungen

Für Fortschrittsbalken und Nachrichten vom Rechen-Task gibt es spezielle Methoden wie `updateProgress` und `updateMessage`.

Siehe Dokumentation.

Warnung

- Die JavaFX-Event-Loop läuft in einem einzigen Ausführungsthread.
- JavaFX-Objekte dürfen nicht aus einem anderen Ausführungsthread verändert werden.

Beispiel: Timer

Die Java-Klasse `Timer` verwendet einen nebenläufige Thread (im Gegensatz zu `Timeline` in JavaFX).

Das folgende Beispiel funktioniert *nicht*:

```
Button button = new Button("Ok");

// Nach einer Sekunde wird button.setText("ko") aufgerufen,
// aber aus einem falschen Ausfuehrungsthread
new Timer().schedule(new TimerTask() {

  @Override
  public void run() {
    button.setText("ko");

  }
}, 1000);
```

Ausnahme bei Ausführung:

```
Exception in thread "Timer-0" java.lang.IllegalStateException:
Not on FX application thread; currentThread = Timer-0
```

Warnung: Beispiel

Ein anderer Thread kann `Platform.runLater` aufrufen.

```
Button button = new Button("Ok");
new Timer().schedule(new TimerTask() {

  @Override
  public void run() {

    // Bitte den JavaFX-Thread, die folgende Methode "run" in
    // sobald wie moeglich in der Event-Loop auszufuehren.
    Platform.runLater(new Runnable() {

      @Override
      public void run() {
        button.setText("ko");

      }
    });

  }
}, 1000);
```

$\Rightarrow$ Nebenläufigkeit nur einsetzen, wenn unvermeidbar.

JavaFX:
CSS, FXML, Scene Builder

Styling mit CSS

Details des Aussehens können über Stylesheets gesteuert werden.

Datei: `style.css`

```
.hbox {
    -fx-background-color: lightblue;
    -fx-alignment: center;
    -fx-padding: 15;
    -fx-spacing: 5;
}

#rect {
    -fx-fill: linear-gradient(from 0% 0% to 100% 100%, red 0%, black 100%);
    -fx-stroke: darkblue;
    -fx-stroke-width: 5;
}

#line {
    -fx-stroke: darkblue;
    -fx-stroke-width: 5;
}
```

Styling mit CSS

- `.hbox` ist eine Stilklasse und betrifft möglicherweise mehrere Nodes.
- `#line` und `#rect` sind CSS-Ids und betreffen nur eine einzige Node.

Auswahl des Stylesheets im Java-Code:

```
// hbox soll Stil von CSS-Klasse .hbox lesen
hbox.getStyleClass().add("hbox");

// line soll Stil von CSS-Id #line lesen
line.setId("line");
// rec soll Stil von CSS-Id #rect lesen
rectangle.setId("rect");

// CSS-Datei kann ueber den Classpath (d.h. im Projekt) angegeben werde
borderPane.getStylesheets().add("beispiel/style.css");
```

Dokumentation des CSS-Formats: **JavaFX CSS Reference**

FXML und Scene Builder

In JavaFX wird das Layout der GUI durch die geeignete Schachtelung von Klassen wie `HBox`, `BorderPane`, etc. festgelgt.

Die Festlegung von Layoutdetails (Abstände, Anordnung, Farben, etc.) im Programmcode ist aufwendig und unflexibel.

Ansatz zur Vereinfachung:

- Formattierung durch CSS
- Dateiformat FXML zur Spezifikation der GUI
- Graphische Tools zur interaktiven Bearbeitung von FXML-Dateien

Man muss den Aufbau der GUI immer noch verstehen, kann aber die Details mit Tools festlegen.

FXML

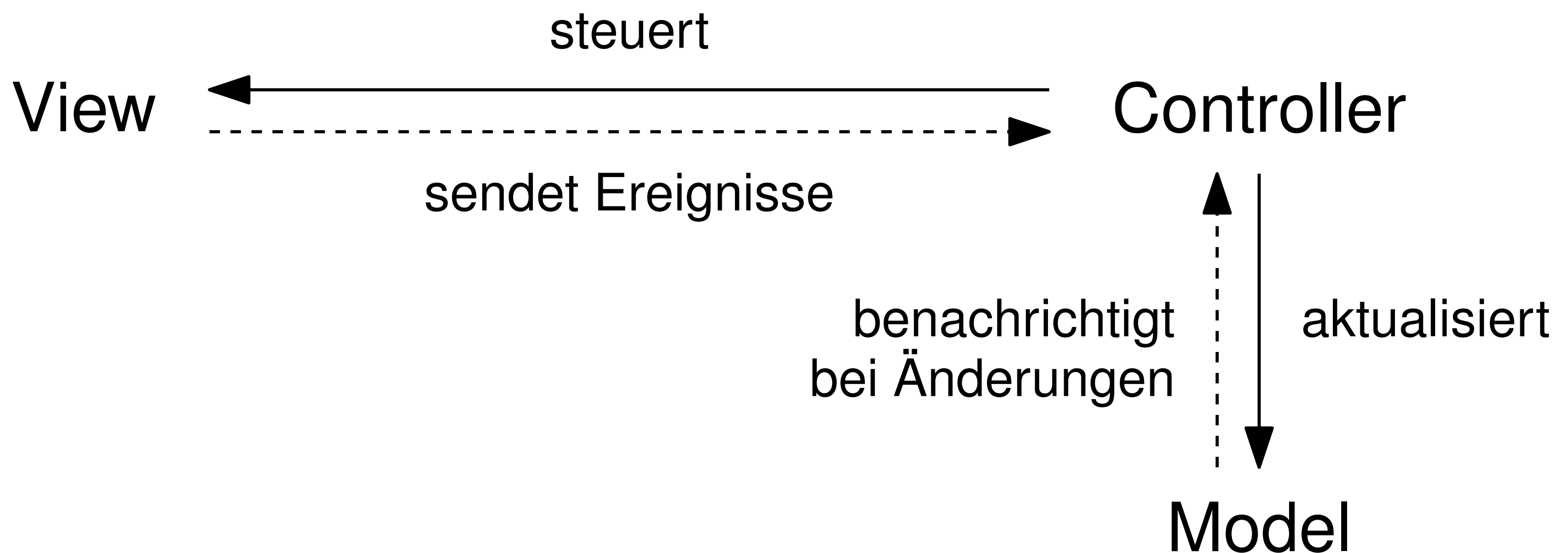
Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.shape.*?>
...
<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity'
            minWidth="-Infinity" prefHeight="347.0" prefWidth="382.0"
            xmlns="http://javafx.com/javafx/8.0.66"
            xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <TextField fx:id="inputField" layoutX="15.0" layoutY="307.0"
          onAction="#textEntered" prefHeight="26.0" prefWidth="353.0"
          AnchorPane.bottomAnchor="14.0" AnchorPane.leftAnchor="15.0"
          AnchorPane.rightAnchor="14.0" />
      <ListView fx:id="listView" layoutX="14.0" layoutY="14.0"
          prefHeight="283.0" prefWidth="353.0" AnchorPane.bottomAnchor="50.
          AnchorPane.leftAnchor="14.0" AnchorPane.rightAnchor="15.0"
          AnchorPane.topAnchor="14.0" />
   </children>
</AnchorPane>
```

FXML: Model-View-Presenter

Die View ist vollständig durch diese FXML-Datei beschrieben. Es gibt keine eigene Klasse.

- $\Rightarrow$ Man kann die View nicht als Observer implementieren.
- $\Rightarrow$ Der Controller übernimmt die Aktualisierung der View.
- Das entspricht dem Model-View-Presenter Muster.



FXML

Laden dieser FXML-Datei im Java-Programm:

```
@Override
public void start(Stage primaryStage) {
  try {
    BeispielController controller = new BeispielController();
    FXMLLoader loader = new FXMLLoader(getClass().getResource("view.fxml"))
    loader.setController(controller);
    Parent view = loader.load();

    Scene scene = new Scene(view);
    primaryStage.setScene(scene);
    primaryStage.show();
  } catch (IOException ex) {
    // Problem beim Laden von view.fxml
  }
}
```

Der FXMLLoader liest die XML-Datei, baut die View auf und initialisiert Variablen im Controller.

FXML: Controller

```java
public class BeispielController {

  @FXML
  private TextField inputField;

  @FXML
  private ListView<String> listView;

  @FXML
  void textEntered(ActionEvent event) {
    listView.getItems().add(inputField.getText());
    inputField.clear();
  }
}
```

In `loader.load()` werden die Steuerelemente erzeugt.

- Für jedes Steuerelement kann in der FXML-Datei ein Variablenname (Attribut `fx:id`) angegeben werden.
- Beim Laden wird das Steuerelement in die gleichnamige Variable im Controller geschrieben. Die Variable muss mit `@FXML` annotiert sein.

Scene Builder

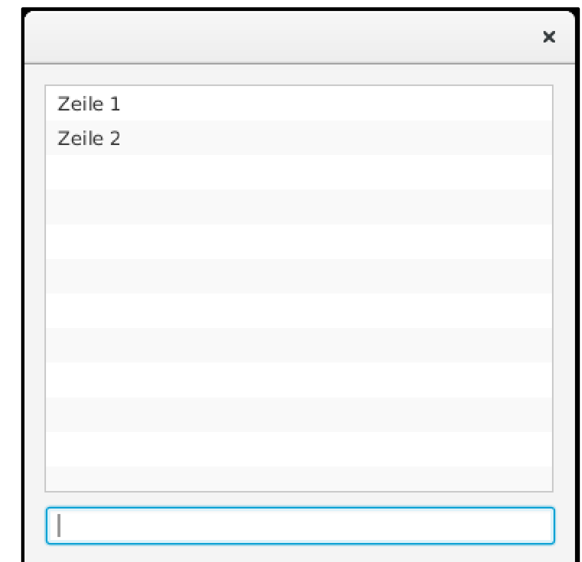
Tool zur Erstellung von FXML-Dateien

- entwickelt von Oracle
- veröffentlicht im Quelltext unter Open-Source Lizenz
- Jar-Dateien und Installer verfügbar von:
  `http://gluonhq.com/open-source/scene-builder/`

Scene Builder: Beispiel

1. Erstellung der GUI-Elemente

- In der Library (links) kann man verschiedene Node-Typen auswählen und zusammenstellen.
- In diesem Beispiel: `AnchorPane` mit zwei Kindern: `ListView` und `TextField`
- Vorschau mit Ctrl+P.
- Speichern der Datei in dem Verzeichnis, in dem man sonst die View-Klasse anlegen würde.



Scene Builder: Beispiel

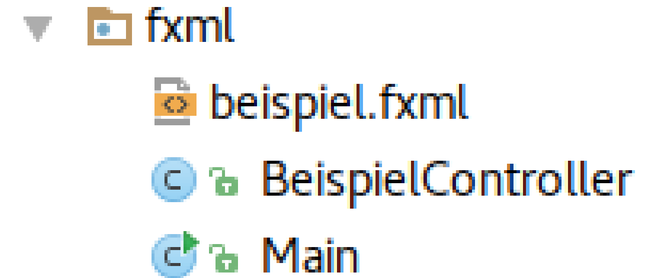
2. Variablen und Methoden für Controller bennenen

- Das `TextField` markieren und rechts bei "Code" (Ctrl-3) unter "fx:id" einen Variablennamen eintragen.
  Im Beispiel: `listView`.
- Die `ListView` markieren und rechts bei "Code" (Ctrl-3) unter "fx:id" einen Variablennamen eintragen.
  Im Beispiel: `inputField`.
- Wenn man im `TextField` auf Enter drückt, soll der eingegebene Text in die Liste eingetragen werden.
  Trage dazu bei "Code" unter "On Action" den Namen der aufzurufenden Methode des Controllers ein.
  Im Beispiel: `textEntered`

Scene Builder: Beispiel

Ein Skelett für die Controller-Klasse kann man sich nun mit "View > Show Sample Controller Skeleton" anzeigen lassen.

Lege diese Klasse im Projekt an.



Scene Builder: Beispiel

3. Controller fertig implementieren

```
package fxml;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;

public class BeispielController {

  @FXML
  private TextField inputField;

  @FXML
  private ListView<String> listView;

  @FXML
  void textEntered(ActionEvent event) {
    listView.getItems().add(inputField.getText());
    inputField.clear();
  }

}
```

Scene Builder: Beispiel

Die FXML-Datei kann jetzt benutzt werden:

```
package fxml;
...

public class Main extends Application throws Exception {

    @Override
    public void start(Stage primaryStage) {
      BeispielController controller = new BeispielController();
      FXMLLoader loader =
        new FXMLLoader(getClass().getResource("beispiel.fxml"));
      loader.setController(controller);
      Parent view = loader.load();

      Scene scene = new Scene(view);
      primaryStage.setScene(scene);
      primaryStage.show();
    }
}
```