CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

GUI-Frameworks und -Anwendungen II

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- JavaFX
  - Multimedia
  - Event-Handling
- Observer-Pattern
- Bindings
- MVC-Pattern

CARL VON OSSIETZKY universität OLDENBURG

Multimedia

Grafik

- 2 Modi: Retained Mode und Immediate Mode
- Retained Mode:
  - Grafiken werden als spezielle **Node**-Objekte erzeugt
  - Grafik-Objekte werden in den **SceneGraph** eingebaut und durch ihn gerendert
  - Grafik-Objekte und andere Nodes lassen sich kombinieren
  - Vorteil: einfache Änderungen durch Änderung entsprechender Eigenschaften
  - Nachteil: hoher Speicherbedarf durch Objekte
- Immediate Mode (→ Canvas-API):
  - Prozeduraler Programmierstil
  - Aufruf konkreter Zeichen-Methoden
  - Vorteil: weniger Speicherbedarf
  - Nachteil: Änderungen durch Neuzeichnen

CARL VON OSSIETZKY universität OLDENBURG

Shapes

- **Shape**: Basisklasse aller Grafik-Objekte
- Einfügen von Grafik-Objekten in den **`SceneGraph`**
- JavaFX rendert **`SceneGraph`** unter Berücksichtigung von
  - Optimierungen
  - Effekten
  - Transformationen
- Konkrete Klassen
  - **`Arc`**, **`Circle`**, **`CubicCurve`**, **`Ellipse`**, **`Line`**, **`Path`**, **`Polygon`**, **`Polyline`**, **`QuadCurve`**, **`Rectangle`**, **`SVGPath`**, **`Text`**

CARL VON OSSIETZKY universität OLDENBURG

Audio

- Klasse **`AudioClip`** für kurze Clips
- Klasse **`MediaPlayer`** für längere Clips
- Alle gängigen Formate

```
URI resource = Paths.get("clip.wav").toUri();
AudioClip clip = new AudioClip(resource.toString());
Button button = new Button("Start");
button.setOnAction((e) -> clip.play(1.0));

resource = Paths.get("music.mp3").toUri();
Media media = new Media(resource.toString());
MediaPlayer mediaPlayer = new MediaPlayer(media);
mediaPlayer.play();
```

CARL VON OSSIETZKY universität OLDENBURG

Video

- Klassen **Media**, **MediaPlayer** und **MediaView**
- Viele gängige Formate

```
// Create the media source
Media media = new Media(Paths.get("video.mp4").toUri().toString());

// Create the player and set to play automatically
MediaPlayer mediaPlayer = new MediaPlayer(media);
mediaPlayer.setAutoPlay(true);

// Create the view and add it to the Scene
MediaView mediaView = new MediaView(mediaPlayer);
root.getChildren().add(mediaView);
```

CARL VON OSSIETZKY universität OLDENBURG

Events

CARL VON OSSIETZKY universität OLDENBURG

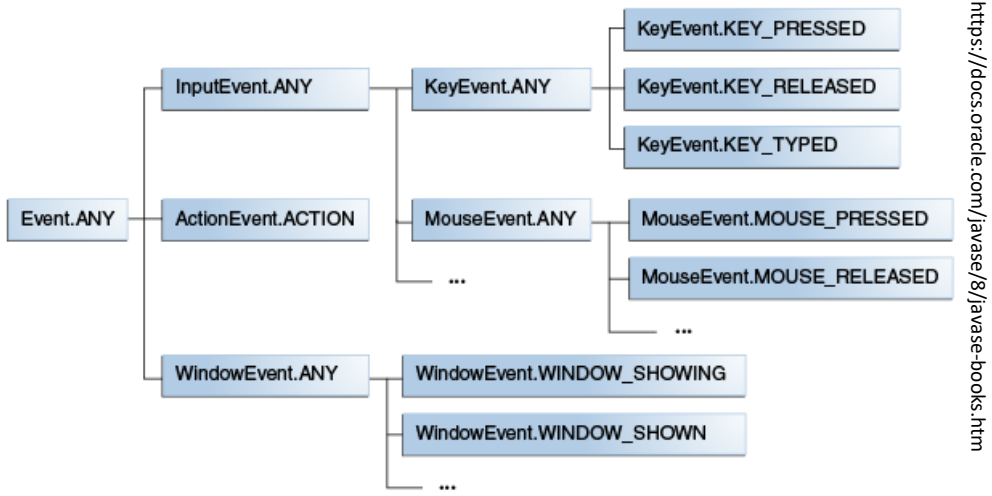
Events

- **`Event`**: Benachrichtigung, dass eine Benutzeraktion ausgelöst wurde
- Basis-Klasse **`javafx.event.Event`**
- Eigenschaften:
  - Event type: Typ des Events
  - Source: Auslöser des Events
  - Target: **Node**-Objekt, auf dem die Aktion ausgelöst wurde

EventTypes

- Instanz der Klasse **`EventType<T extends Event>`**

- **`ActionEvent`**
- **`InputEvent`**
  - **`ContextMenuEvent`**
  - **`DragEvent`**
  - **`KeyEvent`**
  - **`MouseEvent`**
    - **`MouseDragEvent`**
- **`WindowEvent`**



https://docs.oracle.com/javase/8/javase-books.htm

- Jeweils statische Attribute für spezifischere Events
  - `public static final EventType<MouseEvent> MOUSE_PRESSED`
  - `public static final EventType<MouseEvent> MOUSE_RELEASED`
  - `public static final EventType<MouseEvent> MOUSE_CLICKED`

CARL VON OSSIETZKY universität OLDENBURG

EventTargets

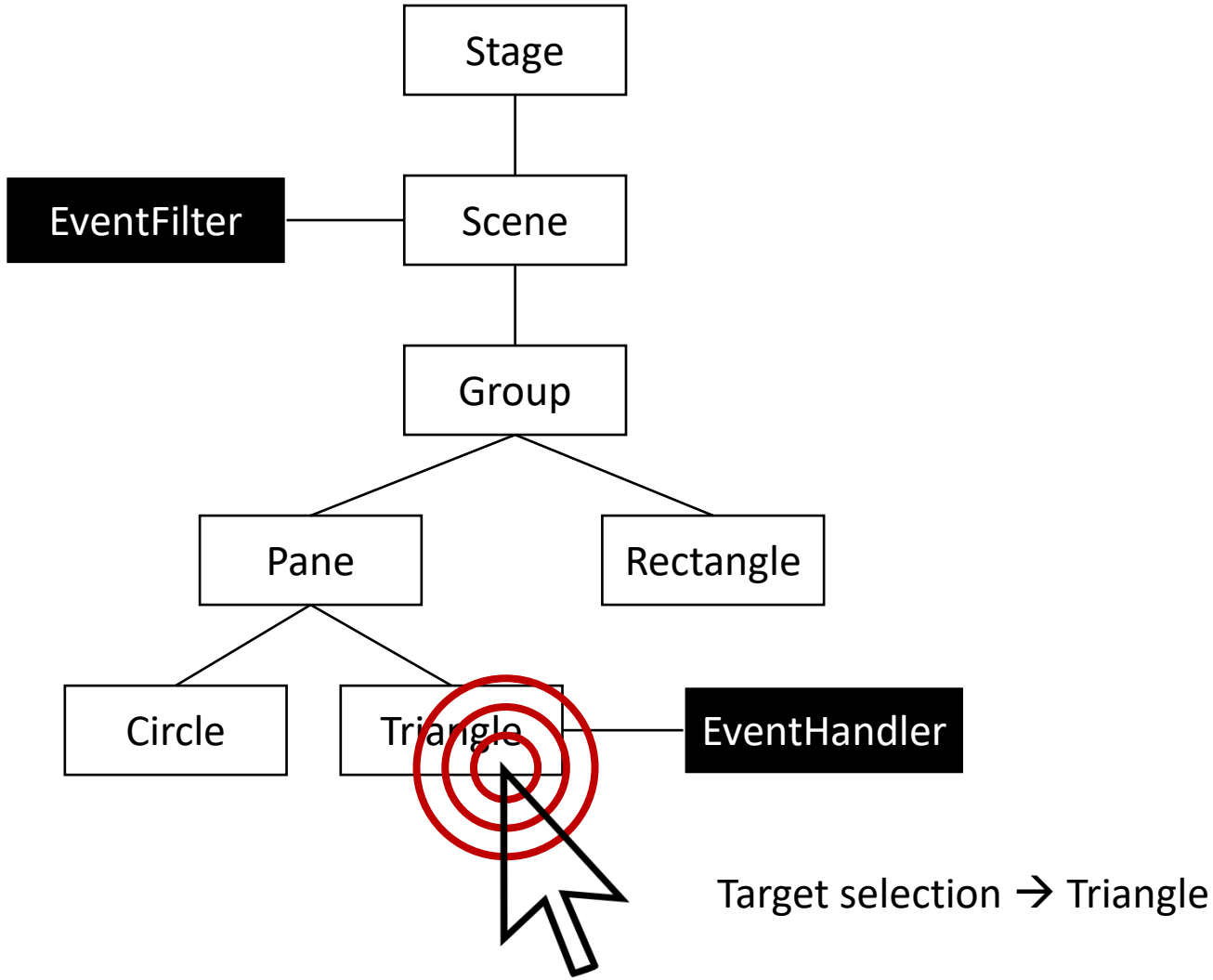
- Interface **`EventTarget`**
- Nahezu alle UI-Klassen implementieren das Interface

CARL VON OSSIETZKY universität OLDENBURG

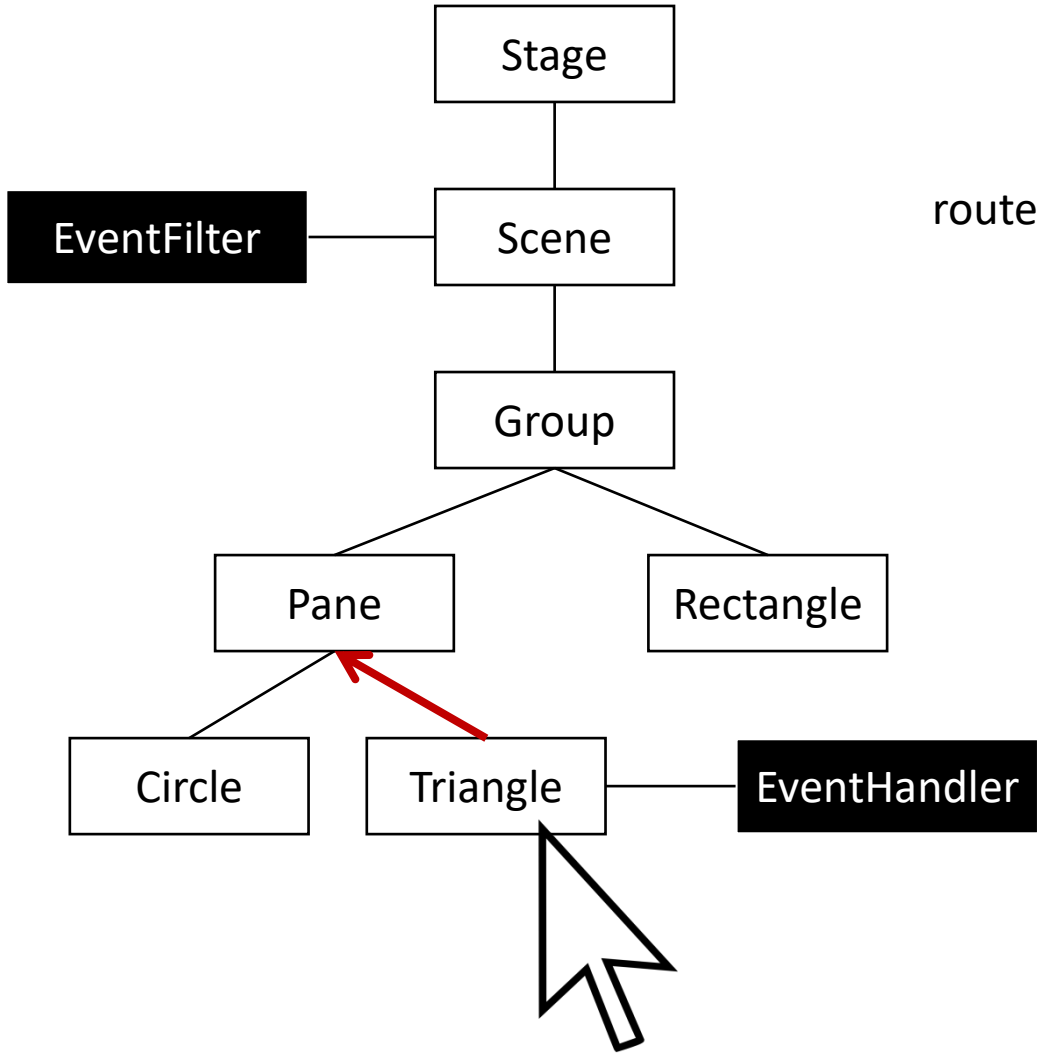
Event Delivery Process

1. Target selection
   - Key events: Target ist **Node**-Objekt mit Fokus
   - Mouse events: Target ist das oberste **Node**-Objekt, wo sich der Mauscursor befindet
2. Route construction
   - Event Dispatch Chain bezüglich der Hierarchie des **SceneGraph**
3. Event capturing
   - Event wird von der **SceneGraph**-Wurzel zum Target runtergereicht
   - dabei werden evtl. **EventFilter**-Operationen ausgeführt
   - wenn das Event nicht konsumiert wird, wird es bis zum Target durchgereicht
4. Event bubbling
   - Event wird vom Target zur **SceneGraph**-Wurzel wieder hochgereicht
   - dabei werden registrierte **EventHandler** ausgeführt, bis ein Handler ggf. das Event konsumiert

Event Delivery Process: Beispiel

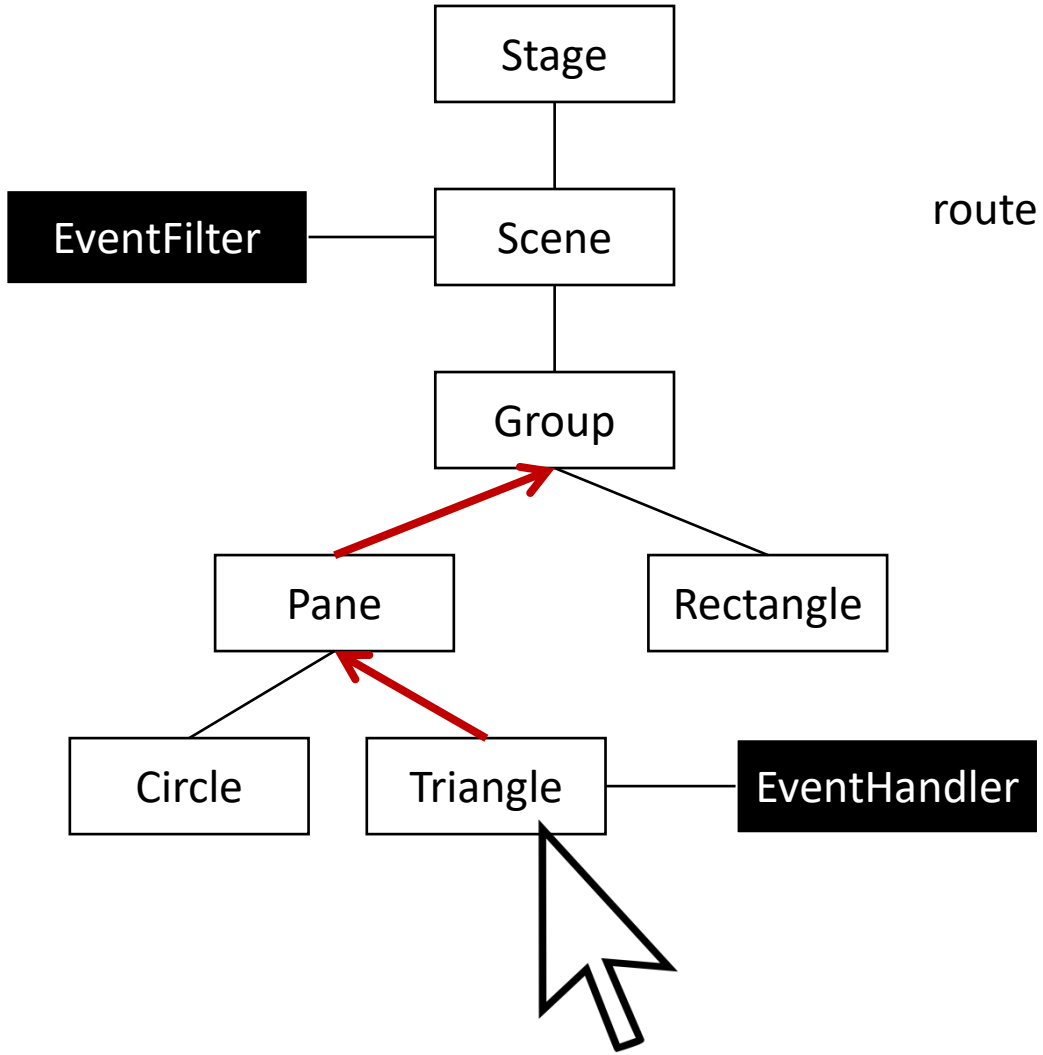


Event Delivery Process: Beispiel



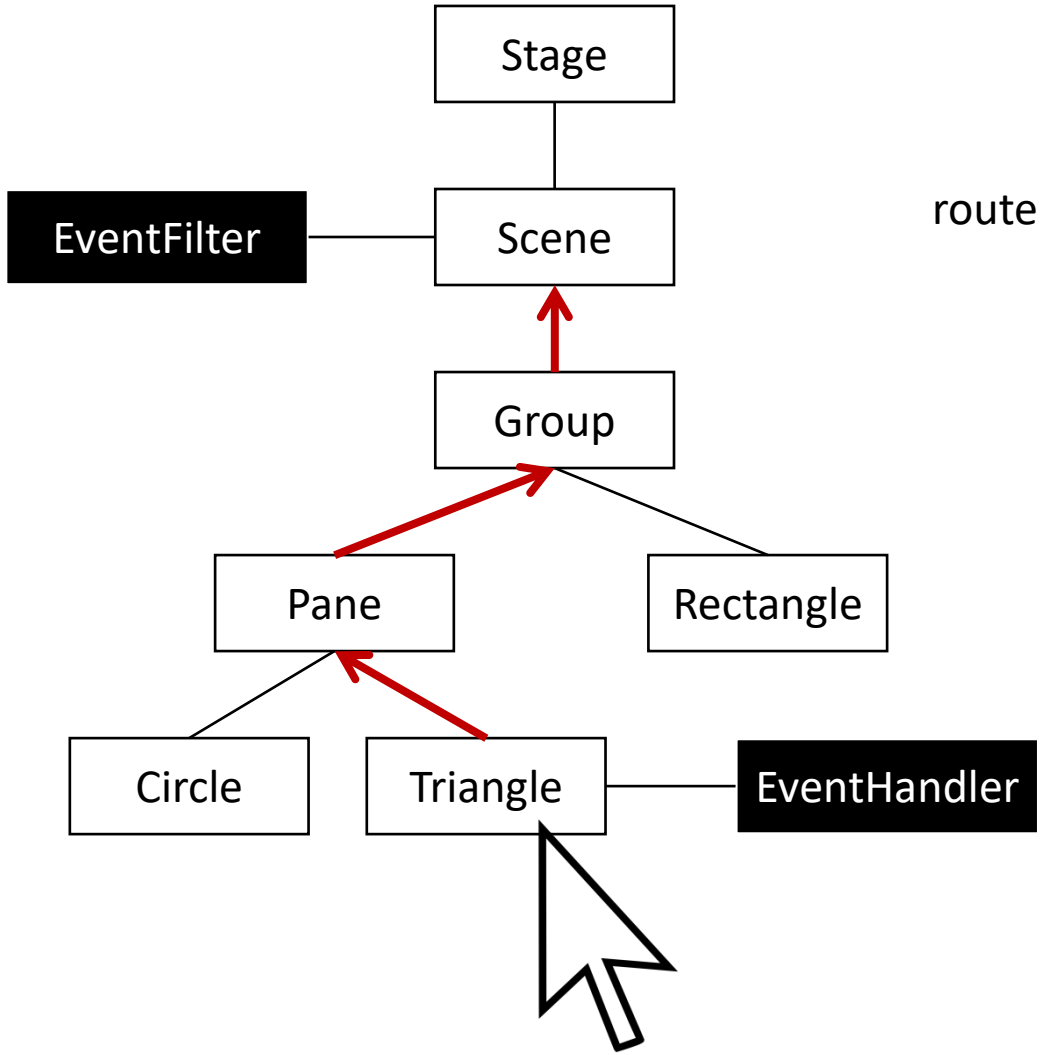
route construction

Event Delivery Process: Beispiel



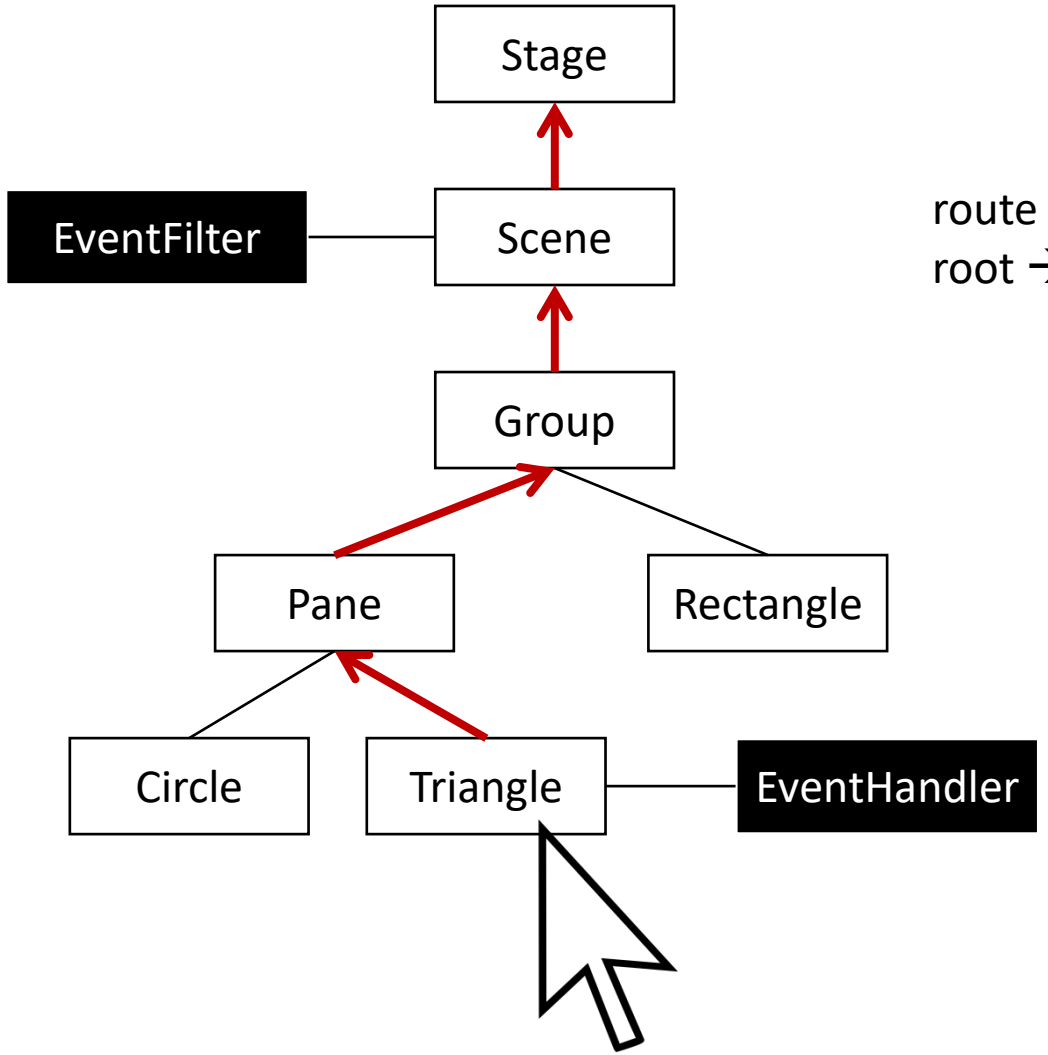
route construction

Event Delivery Process: Beispiel



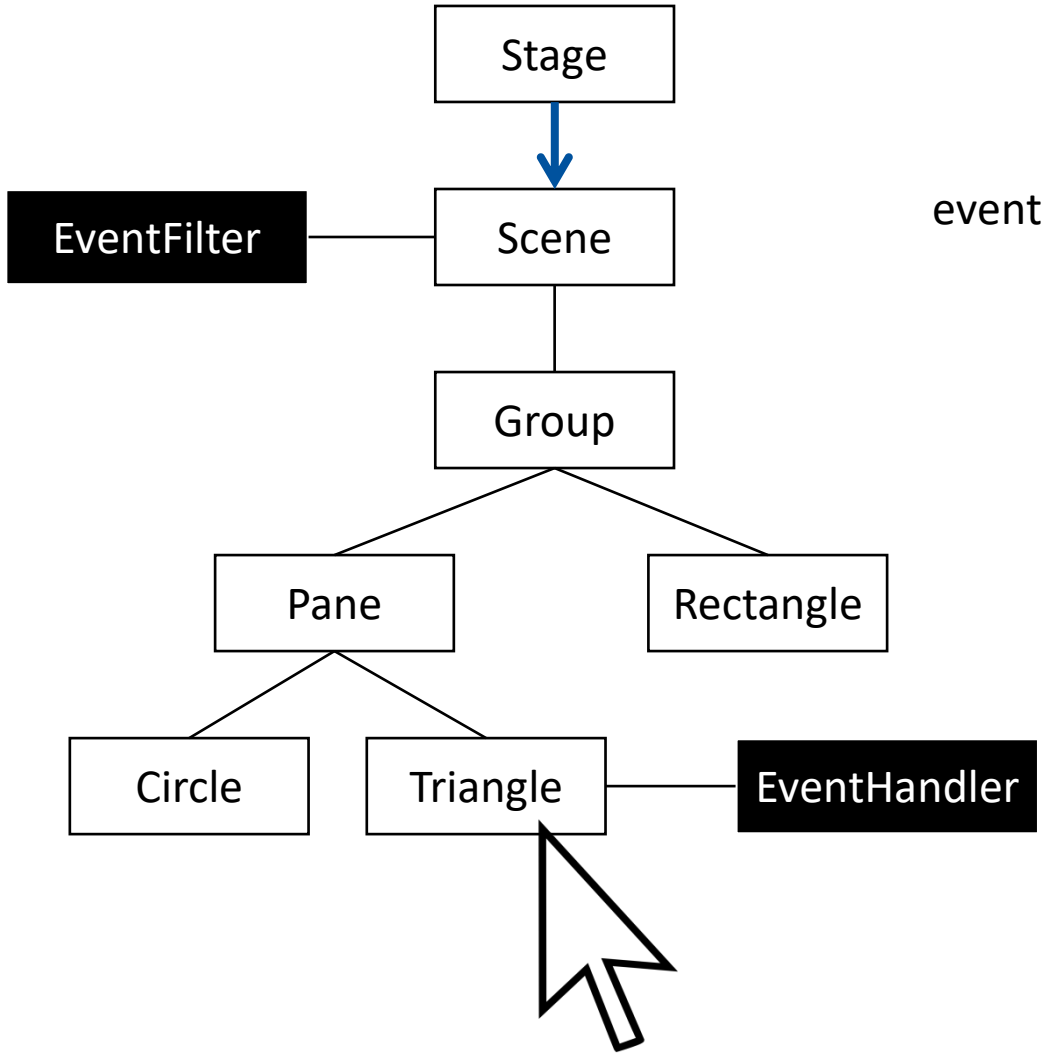
route construction

Event Delivery Process: Beispiel



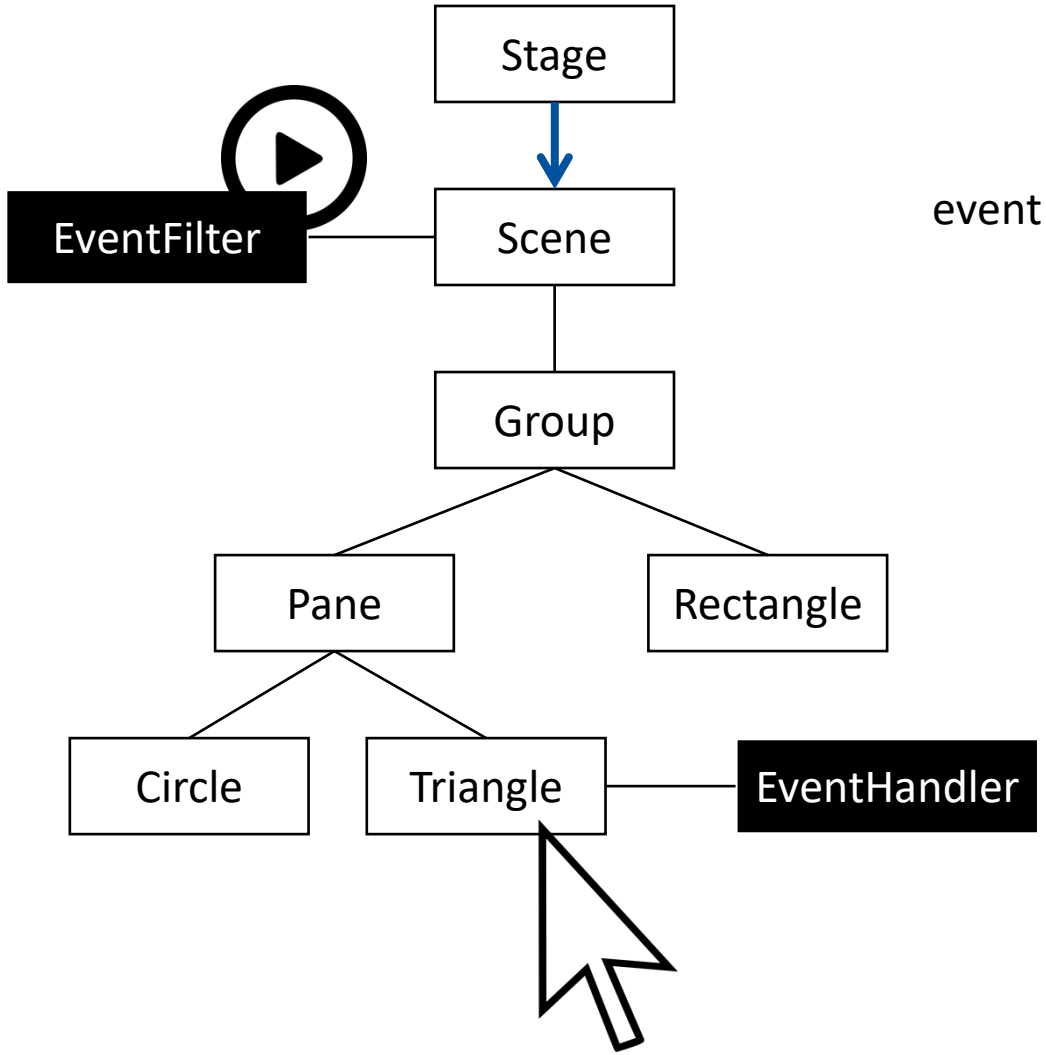
route construction
root → Stage

Event Delivery Process: Beispiel



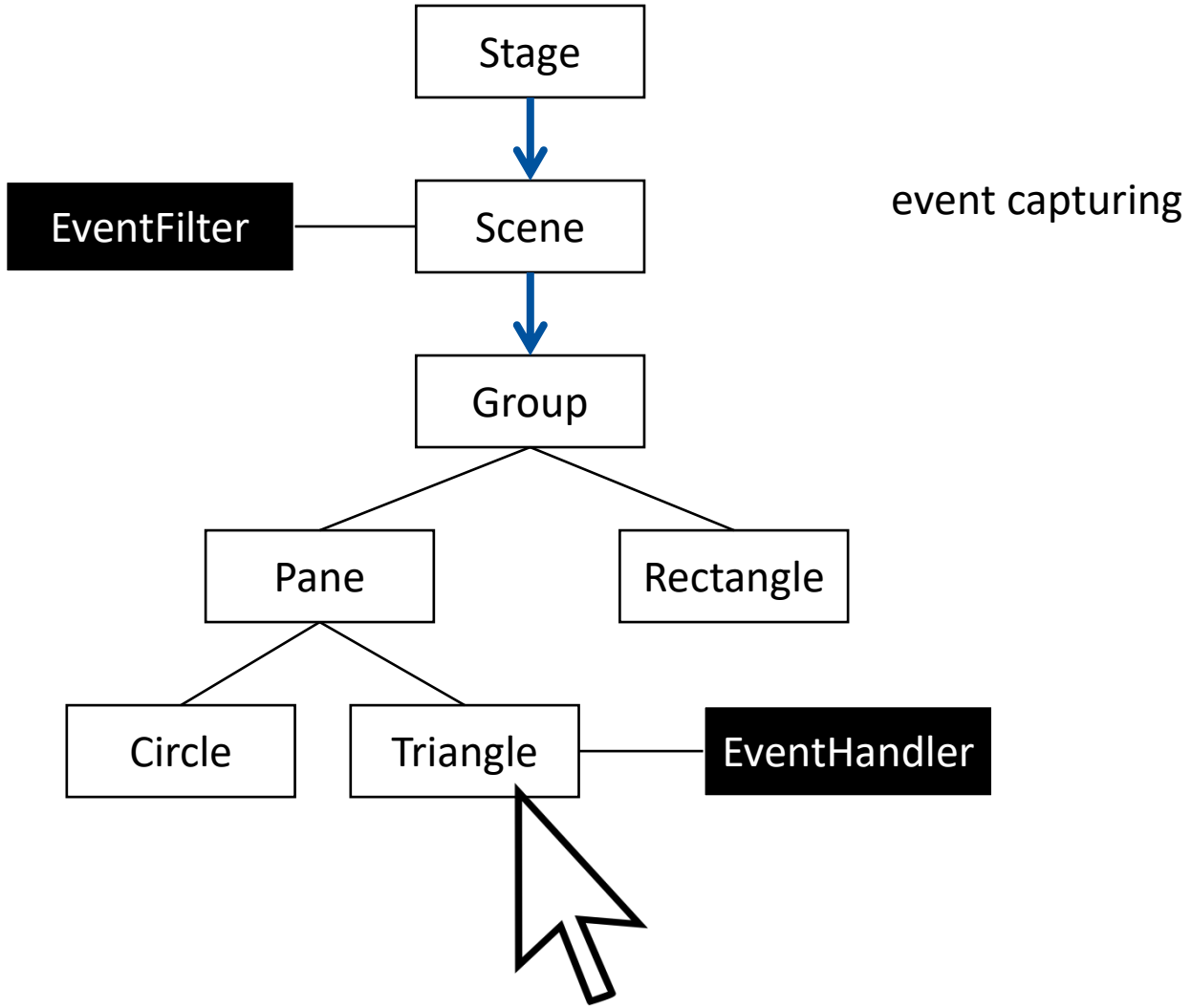
event capturing

Event Delivery Process: Beispiel



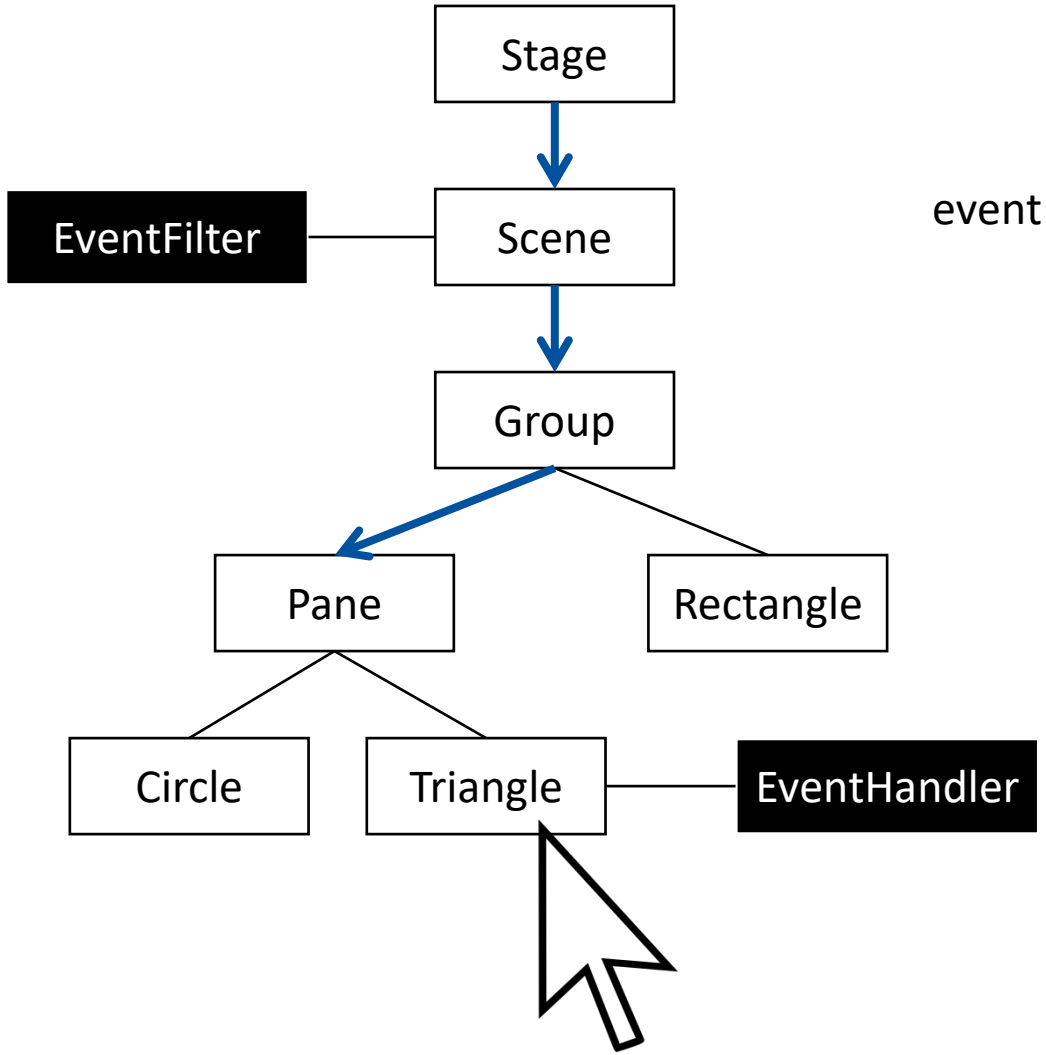
event capturing

Event Delivery Process: Beispiel



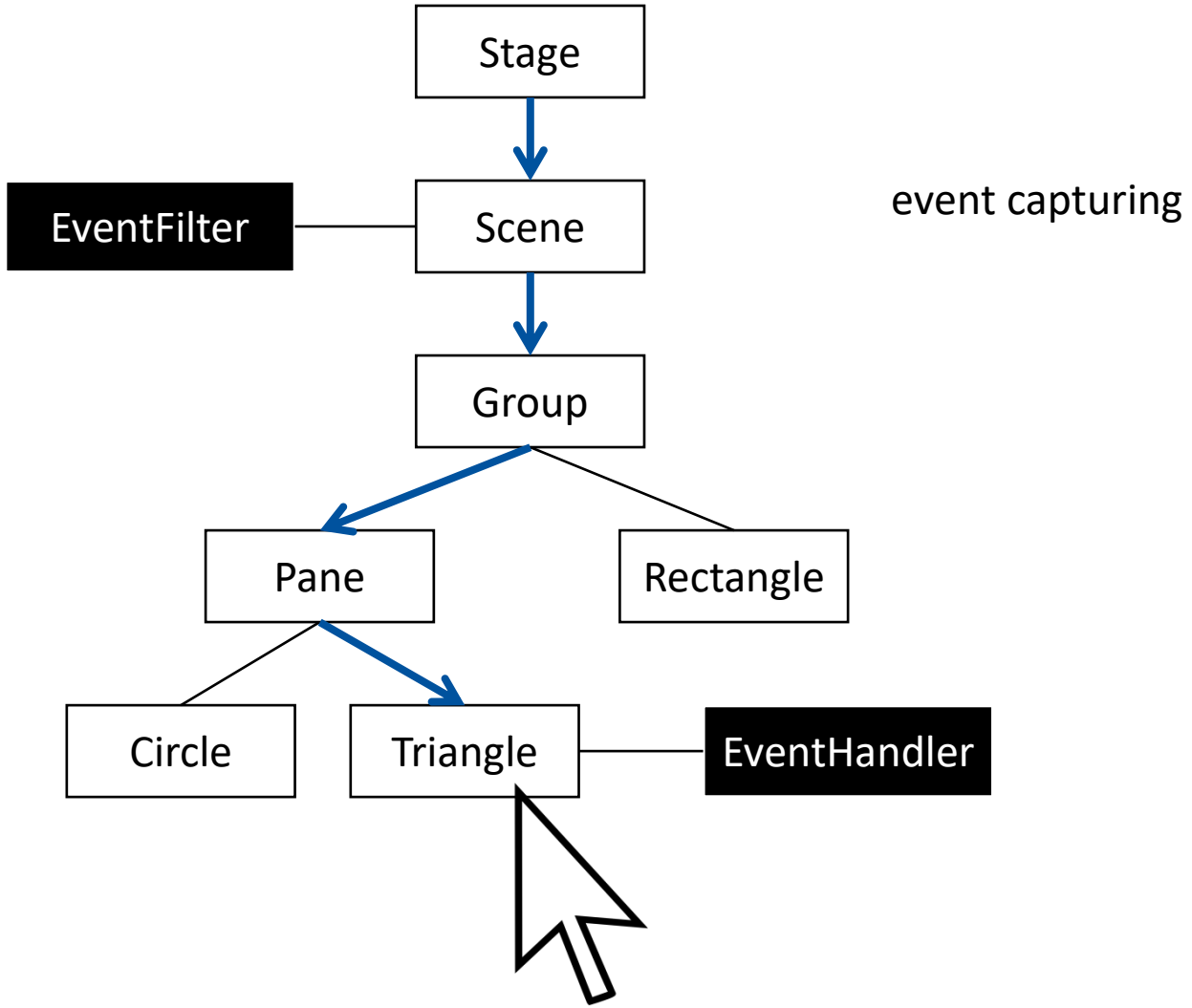
event capturing

Event Delivery Process: Beispiel

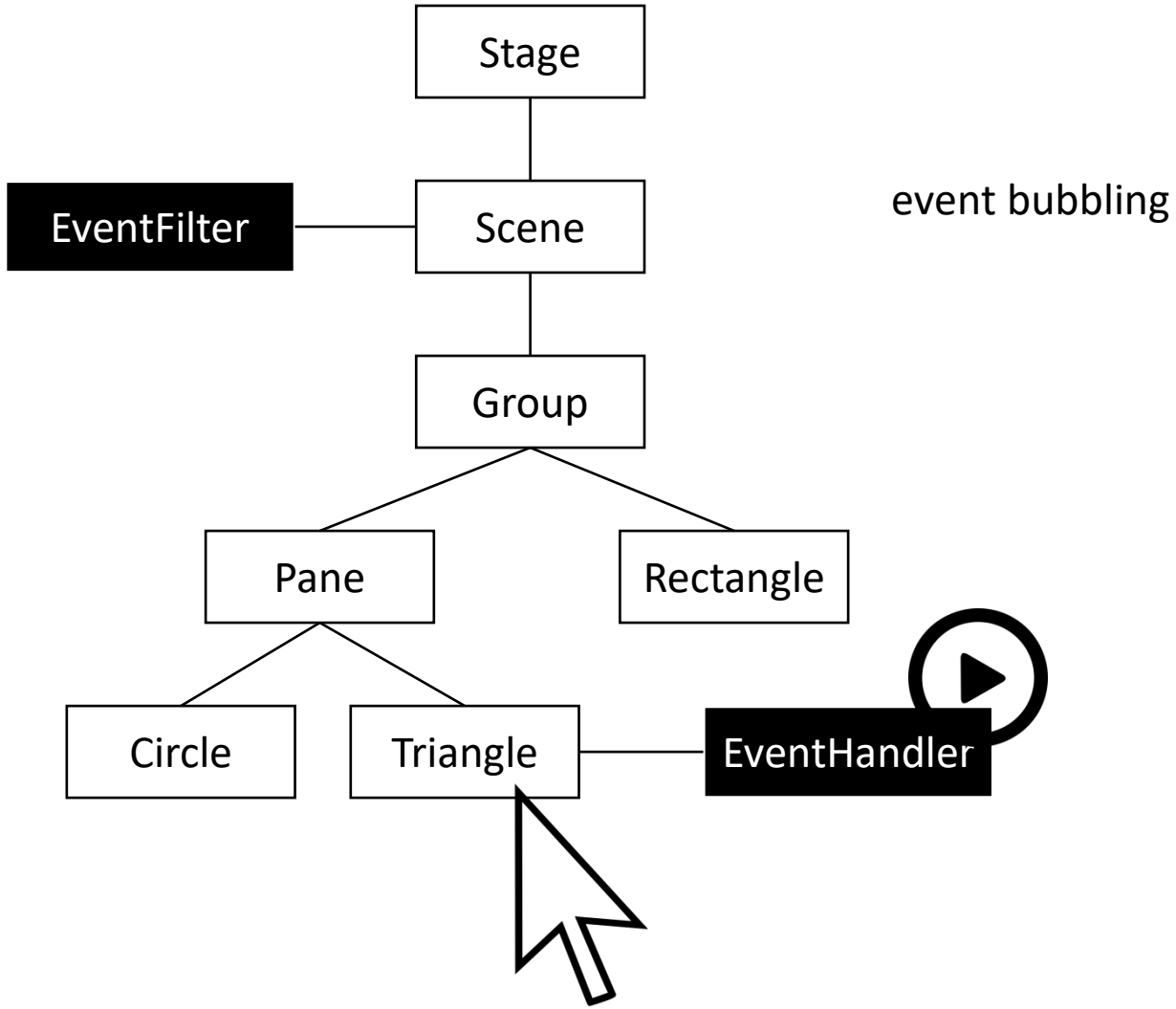


event capturing

Event Delivery Process: Beispiel

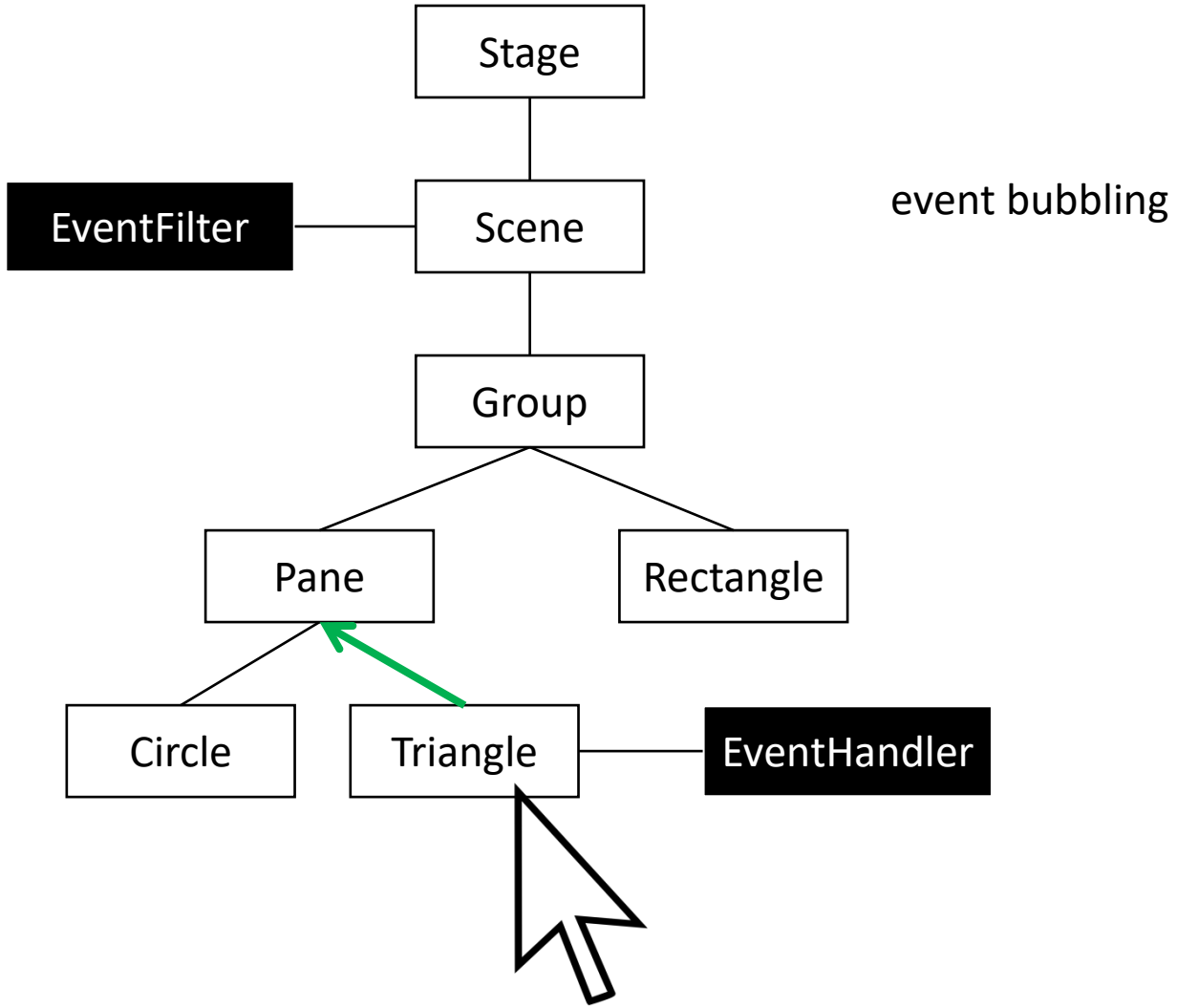


Event Delivery Process: Beispiel



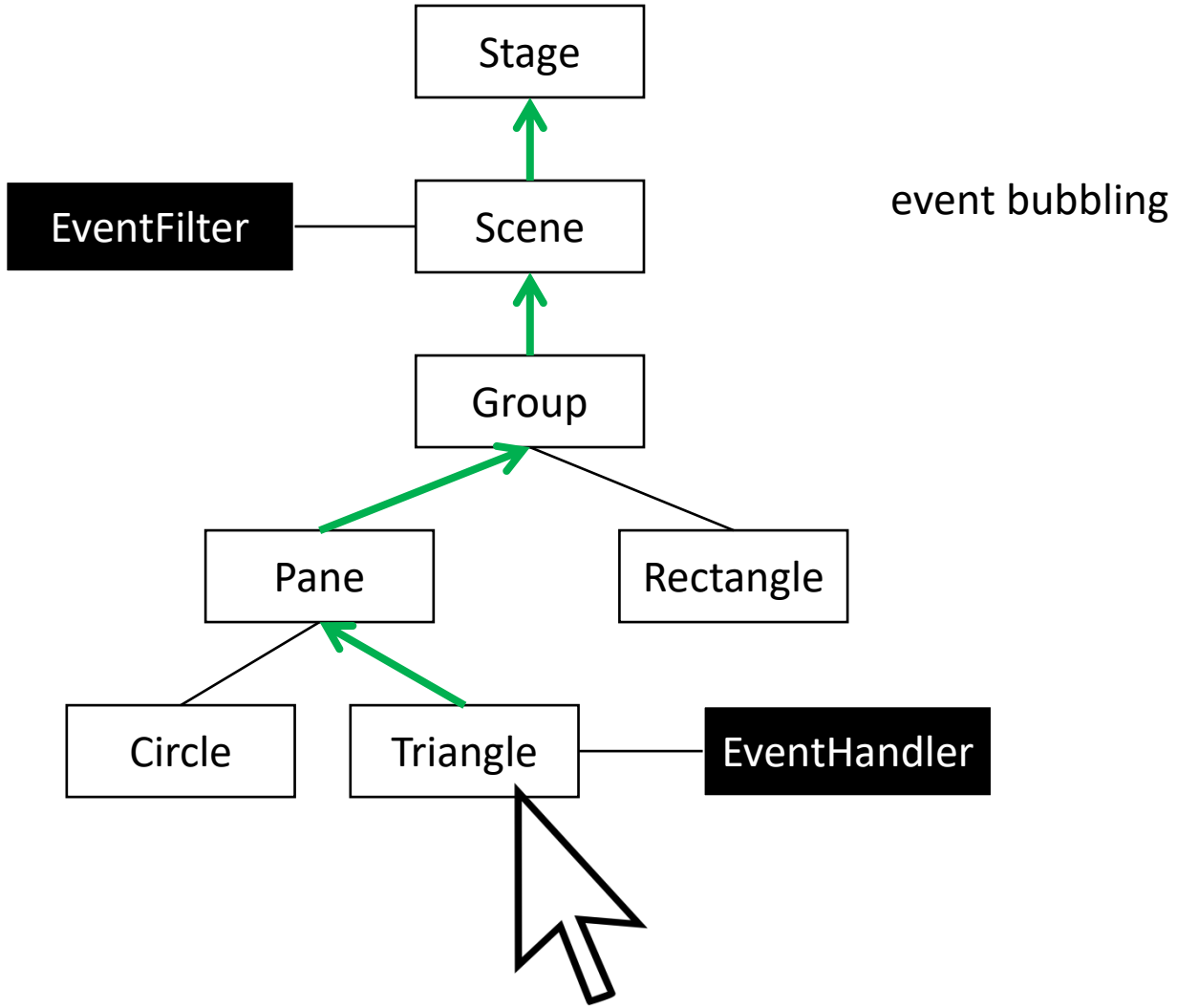
event bubbling

Event Delivery Process: Beispiel



event bubbling

Event Delivery Process: Beispiel



event bubbling

CARL VON OSSIETZKY universität OLDENBURG

Event Handling

- 2 Varianten:
  - **`EventFilter`**: Event Capturing Phase
  - **`EventHandler`**: Event Bubbling Phase

- Bei mehreren Filtern bzw. Handlers eines Knoten:
  - Spezifische EventTypen haben Priorität vor allgemeinen EventTypen (z.B. **`MouseEvent.PRESSED`** vor **`InputEvent.ANY`**)

- Konsumieren von Event:
  - Aufruf der Methode **`consume()`**
  - Event Dispatch Chain wird beendet
  - **`EventFilter`** bzw. -**`Handler`** des konsumierenden Knotens werden noch ausgeführt

CARL VON OSSIETZKY universität OLDENBURG

EventHandler

```
interface EventHandler<T extends Event> {
    void handle(T event);
}
```

- Herkömmliche Methode zum Setzen von EventHandlern:
  `setOn`*`EventKind`*`(EventHandler<EventType> value)`
  - ***EventKind***: Art des Events
  - **EventType**: Klasse/Typ des Events

```
Button btn = new Button();
btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        System.out.println("Hello World");
    }
});
```

EventHandler (2)

- Setzen von EventHandlern mit **`addEventHandler`**
  → hiermit sind mehrere Handler pro Node möglich

```
Button btn = new Button();
btn.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
        System.out.println("Hello World 1");
    };
});
```

EventFilter

- Setzen von EventFiltern mit **`addEventFilter`**
  → hiermit sind mehrere Filter pro Node möglich
- Nutzung der Klasse **`EventHandler`**

```
Label text1 = new Label("hallo");
text1.addEventFilter(MouseEvent.MOUSE_CLICKED, new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event) {
        System.out.println("1: Filtering " + event.getEventType());
    }
});
```

CARL VON OSSIETZKY universität OLDENBURG

MouseEvents

- Viele getter-Methoden zum Abfragen von Details

```
Circle circle = new Circle();
circle.setOnMouseEntered(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) {
        System.out.println("Mouse entered");
    }});
circle.setOnMousePressed(new EventHandler<MouseEvent>() {
    @Override
    public void handle(MouseEvent me) {
        System.out.println("Mouse pressed");
        if (me.isShiftDown()) {
            System.out.println("shift down");
        }
        int count = 0;
        if ((count = me.getClickCount()) > 1) {
            System.out.println("ClickCount: " + count);
        }
    }});
```

CARL VON OSSIETZKY universität OLDENBURG

KeyEvents

- Viele getter-Methoden zum Abfragen von Details

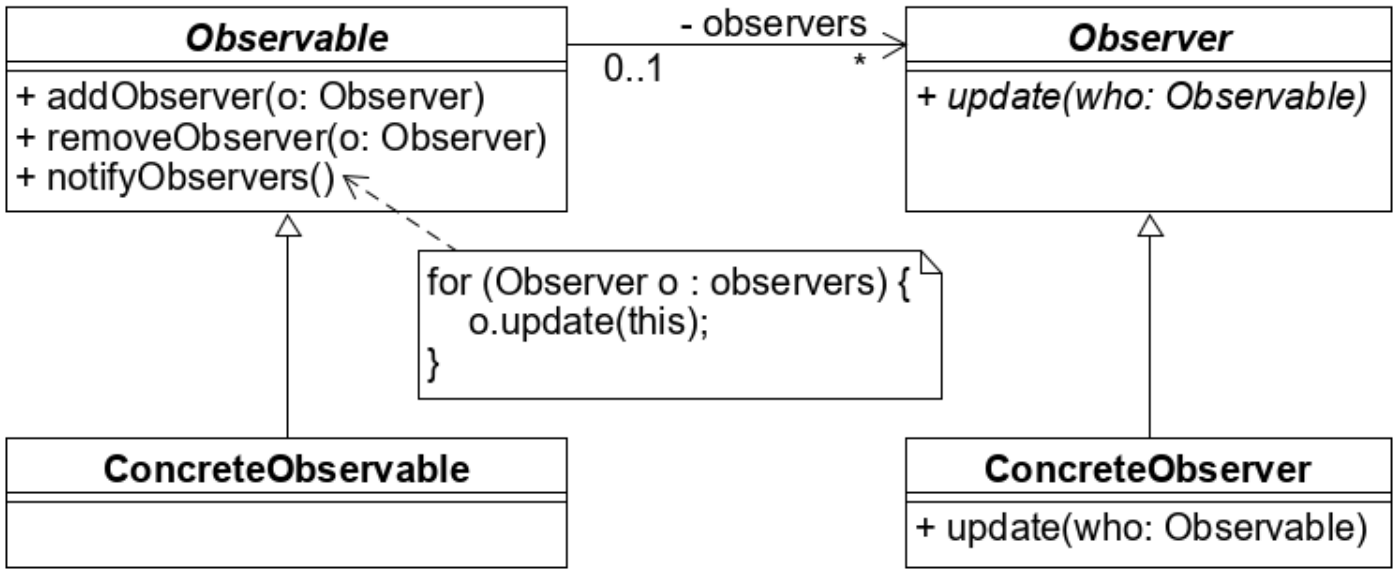
```
TextField textBox = new TextField();
textBox.setPromptText("Write here");

textBox.setOnKeyPressed(new EventHandler<KeyEvent>() {
    @Override
    public void handle(KeyEvent ke) {
        System.out.println("Key pressed: " + ke.getText());
    }
});

textBox.setOnKeyReleased(new EventHandler<KeyEvent>() {
    @Override
    public void handle(KeyEvent ke) {
        System.out.println("Key released: " + ke.getText());
    }
});
```

CARL VON OSSIETZKY universität OLDENBURG

Observer-Pattern

Observer-Pattern

Definiert eine eins-zu-vielen Abhängigkeit zwischen Objekten,
so dass wenn ein Objekt (**`Observable`**) den Zustand ändert,
alle abhängigen Objekte (**`Observer`**) benachrichtigt und aktualisiert werden.



CARL VON OSSIETZKY universität OLDENBURG

Bindings

CARL VON OSSIETZKY universität OLDENBURG

Begriffe

- Variable: Speicherung von Werten
- Property: Speicherung von Werten + automatische Benachrichtigung von ChangeListenern (Observern), wenn Wert sich ändert
- Bean: Objekte auf der Basis von Properties (→ Namenskonventionen)
- Binding: Abhängigkeiten zwischen Properties

CARL VON OSSIETZKY universität OLDENBURG

Bean: Beispiel

```
class MyValue {

    private final DoubleProperty value =
                new SimpleDoubleProperty(this, "value", 0);

    public Double getValue() {
        return value.get();
    }

    public void setValue(Double val) {
        value.set(val);
    }

    public DoubleProperty valueProperty() {
        return value;
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

ChangeListener: Beispiel

```
MyValue myBean = new MyValue();
VBox root = new VBox();
Button button = new Button("Zufallswert");
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        myBean.setValue(Math.random());
    }
});
Label label = new Label("");
root.getChildren().addAll(button, label);

myBean.valueProperty().addListener(new ChangeListener<Number>() {
    @Override
    public void changed(ObservableValue<? extends Number> obs,
                Number oldValue, Number newValue) {
        label.setText(newValue.toString());
    }
});
```

CARL VON OSSIETZKY universität OLDENBURG

Binding: Beispiel

```
VBox root = new VBox();
TextField field = new TextField("");
Label label = new Label("");
Slider slider = new Slider(10, 200, 100);
Rectangle rect = new Rectangle(100, 50);
root.getChildren().addAll(field, label, slider, rect);

label.textProperty().bind(field.textProperty());
rect.widthProperty().bind(slider.valueProperty());
```

Wichtige Klassen und Interfaces

- **`javafx.beans.Observable`**:
  - Kapselung von Daten
  - Registrierung von InvalidationListenern
- **`javafx.beans.value.ObservableValue`** (erweitert **`Observable`**)
  - getValue
  - Registrierung von ChangeListenern
- **`javafx.beans.property.ReadOnlyProperty`** (erweitert **`ObservableValue`**)
  - getBean
  - getName
- **`javafx.beans.property.Property`** (erweitert **`ReadOnlyProperty`**)
  - bind
  - bindBidirectional
- **`javafx.collections`**: observable Collection-Klassen

Property-Klassen

- Paket **`javafx.beans.property`**
- Abstrakte Klassen: ***Type*`Property`**
  (***`Type`*** = **`Float`**, **`String`**, **`Object`**, …)
- Konkrete Klassen: **`Simple`*`Type`*`Property`**
- Konstruktoren:
  - **`new SimpleIntegerProperty()`**
  - **`new SimpleIntegerProperty(wert)`**
  - **`new SimpleIntegerProperty(wert, name, bean)`**

- JavaFX:
  - Viele Eigenschaften von UI-Controls werden über Properties realisiert!

Binding mit ChangeListener

- Registrierung eines **`ChangeListeners`** bei einem **`ObservableValue`**
- Bei Änderung (nicht nur Setzen) des Wertes wird **`changed`** aufgerufen

```
myBean.valueProperty().addListener(new ChangeListener<Number>() {
    @Override
    public void changed(ObservableValue<? extends Number> obs,
                Number oldValue, Number newValue) {
        label.setText(newValue.toString());
    }
});
```

Binding mit InvalidationListener

- Registrierung eines **InvalidationListeners** bei einem **Observable**
- Nur bei erstmaliger Änderung einer Property wird **invalidated** aufgerufen
- Wird wieder scharf geschaltet, sobald **getValue** aufgerufen wurde

```
DoubleProperty number = new SimpleDoubleProperty(123);
number.addListener(new InvalidationListener() {
    @Override
    public void invalidated(Observable observable) {
        System.out.println("ungültig, neu: " + number.getValue());
    }
});
number.setValue(90);
// durch das "number.getValue()" im Listener wird das Binding
// wieder scharf geschaltet; ohne getValue würde ein Invalidation-
// Listener die nächste Änderung nicht mehr mitbekommen
number.setValue(10);
```

CARL VON OSSIETZKY universität OLDENBURG

JavaFX Collections

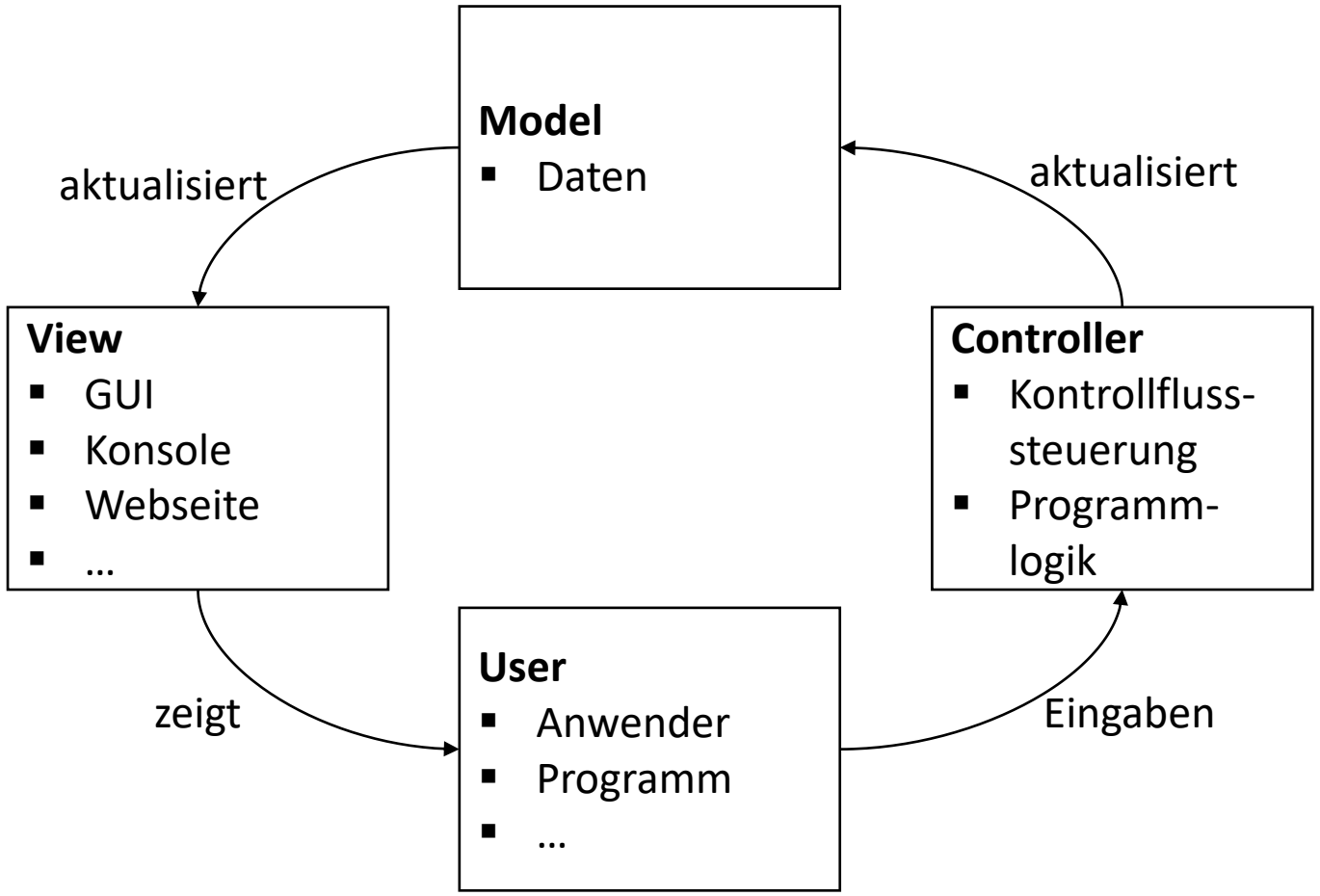
- Beobachtbare Collection-Klassen (List, Map, Set, …)

```
ObservableList<String> ol = FXCollections.observableArrayList();
ol.addListener(new ListChangeListener<String>() {
    @Override
    public void onChanged(Change<? extends String> c) {
        while (c.next()) {
            if (c.wasAdded()) {
                System.out.println(c.getAddedSubList());
            }
        }
    }
});
ol.add("a"); // [a]
ol.add("b"); // [b]

ArrayList<String> l = new ArrayList<String>();
Bindings.bindContent(l, ol);
// l enthält dieselben Elemente wie ol
```

CARL VON OSSIETZKY universität OLDENBURG

MVC-Pattern

MVC-Pattern


Model
Daten
aktualisiert
aktualisiert
View
GUI
Konsole
Webseite
…
Controller
Kontrollfluss-steuerung
Programm-logik
zeigt
Eingaben
User
Anwender
Programm
…

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- JavaFX
  - Multimedia
  - Event-Handling
- Observer-Pattern
- Bindings
- MVC-Pattern