Grafische Benutzeroberflächen mit JavaFX
Einführung

Grafische Benutzeroberflächen

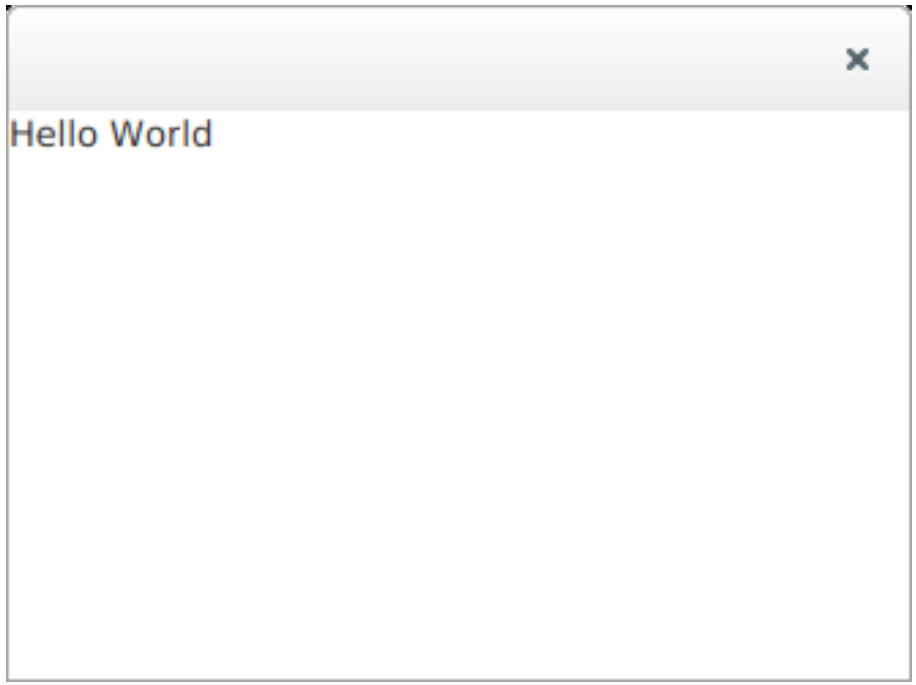
Objektorientierte Bibliotheken

- Windows Presentation Foundation
  (C#, andere .NET-Sprachen)
- Qt (C++)
- Cocoa (Objective C)
- GTK (Objektimplementierung in C)
- Swing, SWT, JavaFX (Java)
- ...

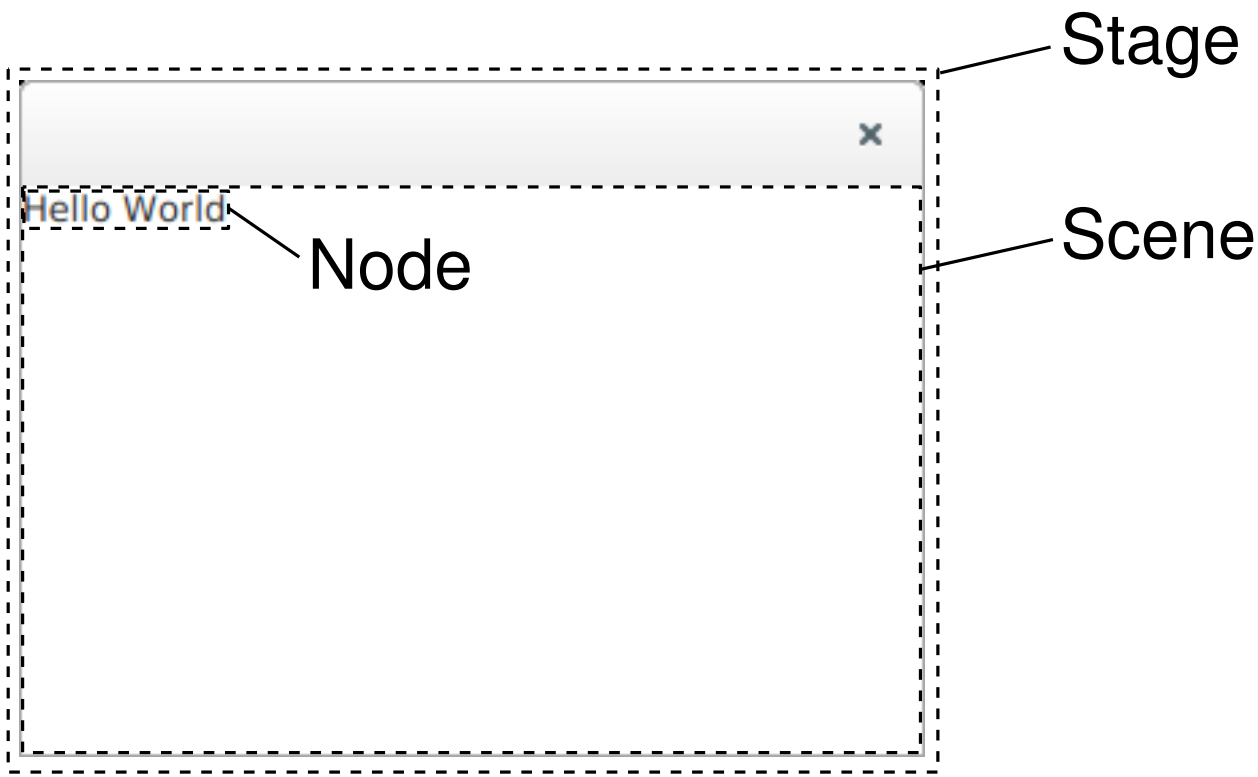
Im Praktikum behandelt: JavaFX

Hinweis: Beispiele beziehen sich auf JavaFX, enthalten ab Java SE Development Kit (JDK 8) und später.

Stage, Scene, Node



Stage, Scene, Node



Stage entspricht dem Fenster

Scene entspricht dem Fensterinhalt

JavaFX Grundgerüst

JavaFX-Programme erben von der abstrakten Klasse `Application`.

Die abstrakte Methode

```
public void start(Stage stage)
```

muss implementiert werden.

- Beim Start des Programms wird ein Fenster (`Stage`) erzeugt und der Methode `start` als Parameter übergeben.
- Dort kann eine Szene hinzugefügt werden und das Fenster dargestellt werden.
- Die Anwendung wird automatisch beendet, wenn das letzte Fenster geschlossen wird.

JavaFX Grundgerüst

```
import javafx.application.Application;
import javafx.scene.*;
import javafx.stage.Stage;
import javafx.scene.control.*;

public class Main extends Application {

  @Override
  public void start(Stage stage) throws Exception {

    Node node = new Label("Hello World");

    // Group erbt von Node und erlaubt die Gruppierung mehrerer Nodes.
    Group root = new Group();
    root.getChildren().add(node);

    Scene scene = new Scene(root, 300, 100);

    stage.setTitle("JavaFX Fenster");
    stage.setScene(scene);
    stage.show();
  }
}
```

Nodes

Die verschiedenen graphischen Elemente sind durch Unterklassen von `Node` repräsentiert.

Beispiele:

- Steuerelemente: `Label`, `TextField`, `Button`, `Slider`, `...`
- Geometrische Formen: `Line`, `Circle`, `Polygon`, `...`

Darstellung eines Polygons:

```
Polygon polygon = new Polygon();
polygon.getPoints().addAll(new Double[]{
    100.0, 100.0,
    320.0, 110.0,
    150.0, 130.0,
    170.0, 180.0,
    290.0, 200.0,
    110.0, 230.0 });
root.getChildren().add(polygon);
```

Nodes

Container-Nodes

Mehrere `Node`-Objekte können zu einem einzigen `Node`-Objekt zusammengefasst werden.

Beispiele: `Group`, `Region`, `Pane`, `BorderPane`, `HBox`, `VBox`, `...`

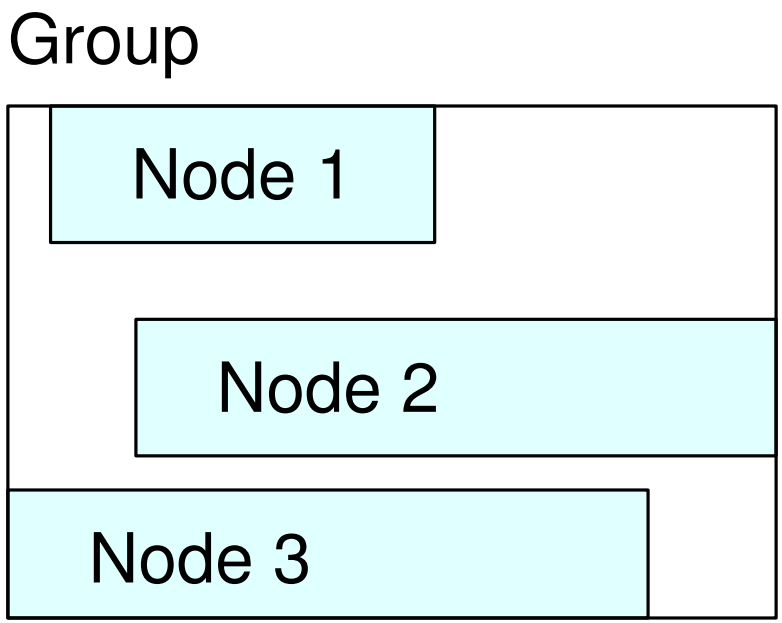
Instanzen dieser Unterklassen von `Node` speichern eine Liste von Kinder-Nodes.

```
Group root = new Group();
root.getChildren().add(polygon);
// root.getChildren() liefert Liste der Kinder-Nodes
```

Nodes

Container-Nodes

- `Group` (Unterklasse von `Node`):
  - Kind-Nodes erhalten ihre gewünschte Größe
  - Gruppe ist gerade groß genug, um alle enthaltenen Objekte zu umschließen



Nodes

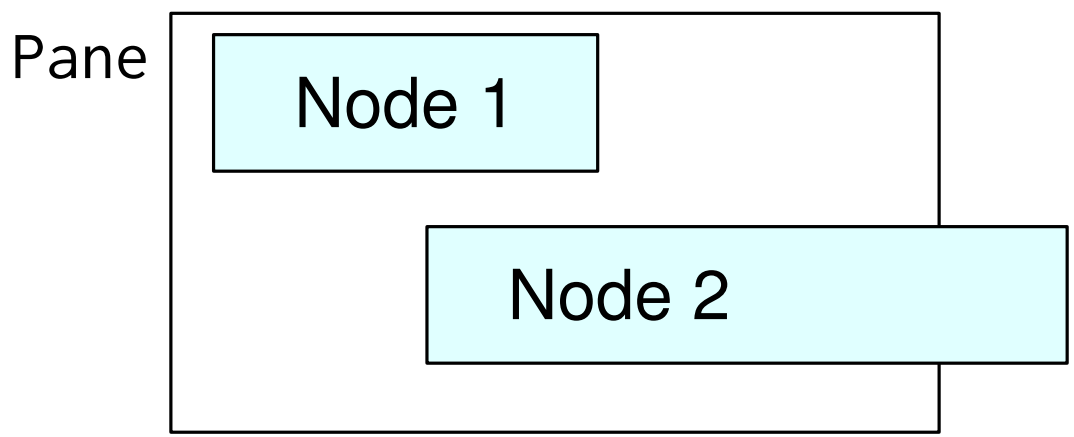
Container-Nodes

- `Region` (Unterklasse von `Node`):
  - Größe unabhängig von Größe der Kinder
  - jede Region definiert minimale, gewünschte und maximale Größe
  - bei Größenänderungen werden die Kinder neu angeordnet und evtl. ihre Größe angepasst
  - verschiedene Unterklassen implementieren verschiedene Layouts

Nodes

Unterklassen von `Region`

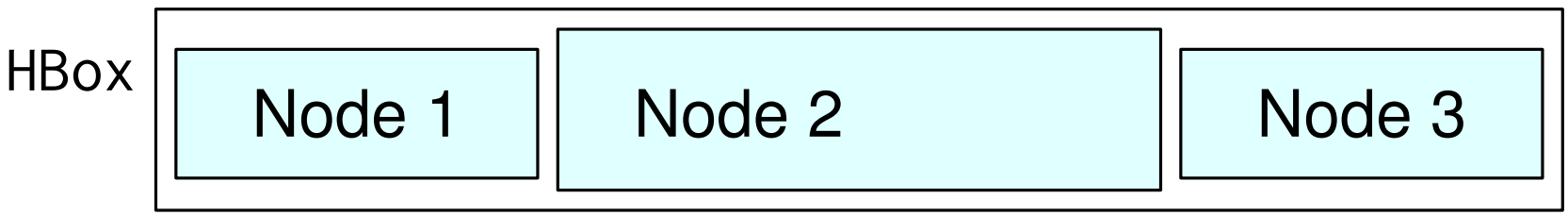
- Pane:
  - manuelle Größe und manuelles Layout der Kinder
  - Clipping möglich



Nodes

Unterklassen von `Region`

- `HBox`:
  - horizontale Anordnung der Kinder
  - Verhalten der Kinder bei Skalierung kontollierbar



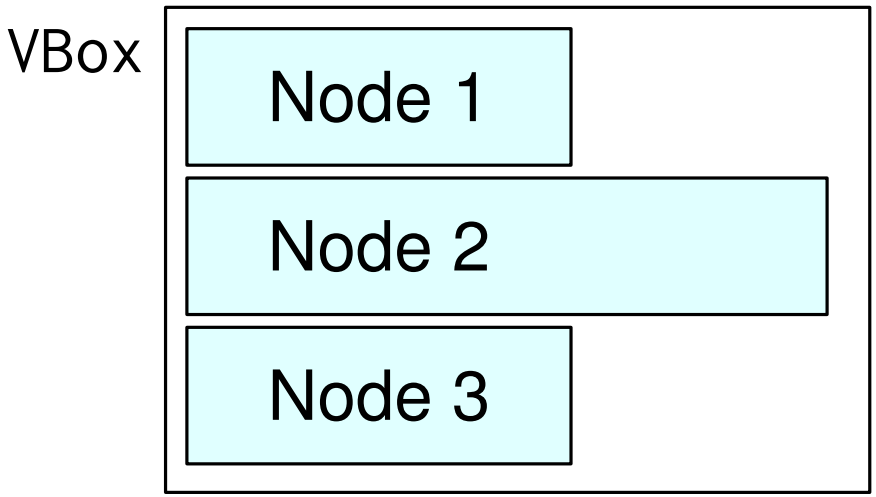
Beispiele für Verhalten bei Vergrößerung:

- die Kinder behalten ihre Größe und werden gleichmäßig angeordnet, z.B. zentriert
- die Kinder werden zu gleichen Teilen vergrößert, um den zusätzlichen Platz zu füllen
- nur Node 2 wird vergrößert, um den Platz zu füllen

Nodes

Unterklassen von `Region`

- `VBox`: analog zu `HBox`, nur vertikal

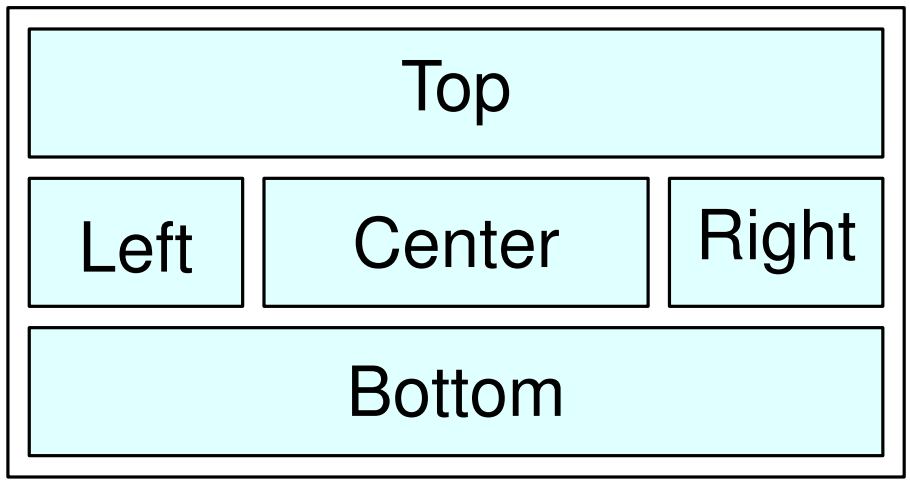


Nodes

Unterklassen von `Region`

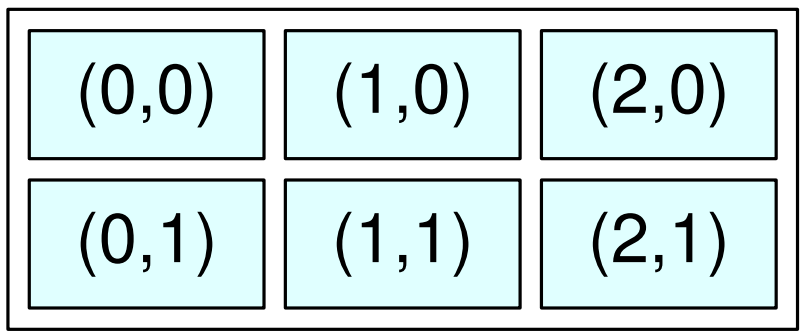
- `BorderPane`

`BorderPane`



- `GridPane`

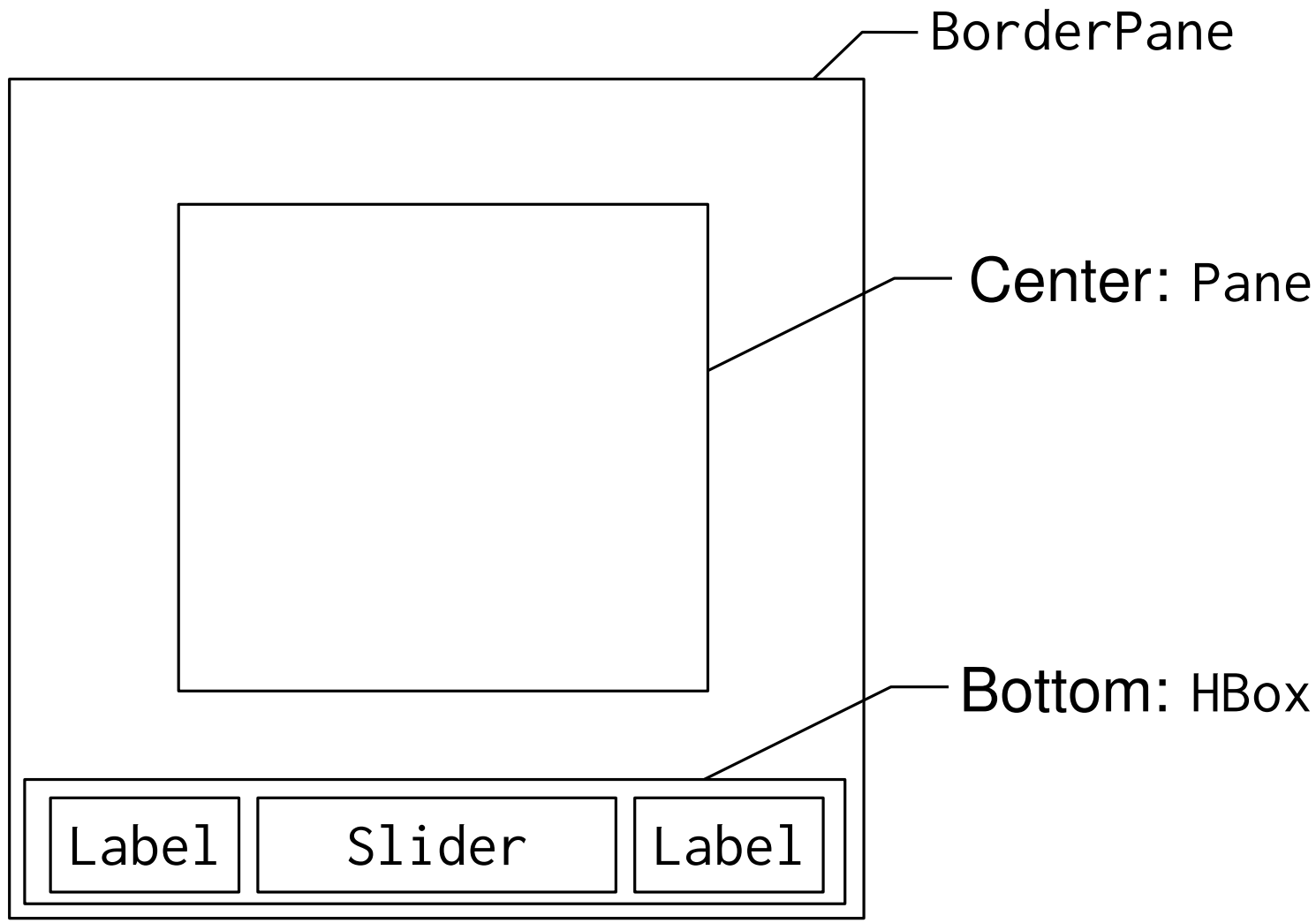
`GridPane`



- usw. (siehe Javadoc-Dokumentation)

Automatisches Layout

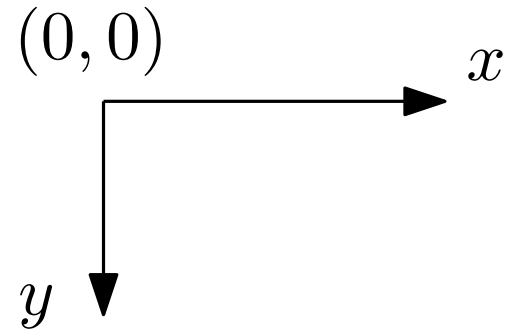
Das Layout der GUI-Elemente wird durch die geeignete Schachtelung von Container-Nodes bestimmt.



JavaFX Koordinaten

Jedes `Node`-Objekt hat sein eigenes Koordinatensystem.

In Container-Nodes beziehen sich Position und Größe der Kinder immer auf das Koordinatensystem der enthaltenden Node.



Koordinaten sind `double`-Werte. Sie können als Pixel verstanden werden.

Ausblick: Das Koordinatensystem kann geändert werden (Drehung, Skalierung, . . .)

- Dann sind die Koordinaten keine Pixel mehr.
- Spart oft Umrechnung von Daten.

Zeichenfläche: Canvas

`Canvas` ist eine Node-Klasse in JavaFX. Sie repräsentiert eine Fläche von Pixeln, auf die direkt gezeichnet werden kann.
Dargestellte Dinge sind keine eigenen Java-Objekte!

Vorteile:

- schnelles Zeichnen
- komplexe Grafiken

Nachteile:

- geometrische Objekte sind keine Java-Objekte, darum
  - keine Interaktion (z.B. Klick-Ereignisse für Linie)
  - keine Manipulation / Animation (z.B. Verschieben)
- Skalierung eines Canvas vergrößert Pixel

Canvas: Beispiel

```
// Canvas der Groesse 200x100 Pixel anlegen
Canvas canvas = new Canvas(200, 100);
root.getChildren().add(canvas);

// Grafik auf dem Canvas zeichnen:
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setFill(Color.BISQUE);
gc.fillOval(5, 5, 100, 90);
gc.strokeLine(50, 50, 70, 60);
gc.strokeText("Text", 10, 20);
```

Ereignissgesteuerte Programmierung

JavaFX basiert auf einem ereignisgesteuerten Ansatz (ebenso wie die meisten anderen graphischen Benutzeroberflächen)

- Alle Situationen, bei denen etwas passieren soll, werden als Event (Ereignis) bezeichnet
  - Physikalisches Ereignis: Maus bewegt, Taste k losgelassen, ...
  - Logisches Ereignis: Programmknopf gedrückt, ...
- Events kann Code zugeordnet werden: Wenn das Event auftritt, wird der Code ausgeführt

```
Pane p = new Pane();
p.addEventHandler(MouseEvent.MOUSE_CLICKED, handler);
```

- Genauer Ablauf der Ereignisbehandlung wird in einer späteren Vorlesung vertieft

Interaktion 1: Ereignisse — Maus, Tastatur, . . .

- Man benötigt ein Objekt einer Klasse, welches das Interface `EventHandler<MouseEvent>` oder `EventHandler<KeyEvent>` implementiert
- `addEventHandler(`zu behandelndes Ereignis, behandelndes Objekt`)` ordnet dieses Objekt diesem Ereignis zu
- Sowohl für die Maus, wie auch für die Tastatur stehen eine Vielzahl an Ereignissen zur Verfügung

  `MouseEvent.ANY` Maus hat etwas gemacht

  `MouseEvent.MOUSE_CLICKED` Ein Mausklick

  `MouseEvent.MOUSE_DRAGGED` Maus wurde bewegt, während eine Taste festgehalten wurde

  `KeyEvent.KEY_PRESSED` Eine Taste wurde gedrückt

  `KeyEvent.KEY_TYPED` Etwas wurde eingetippt (Taste gedrückt und wieder losgelassen)

Interaktion Beispiel: Maus

In CanvasMausHandler.java:

```
public class CanvasMausHandler implements EventHandler<MouseEvent> {
    Canvas canvas;
    public CanvasMausHandler(Canvas canvas) {
        this.canvas = canvas;
    }
    public void handle(MouseEvent ev){
        double x = ev.getX(); double y = ev.getY();
        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.setFill(Color.GREEN);
        gc.fillOval(x-20,y-20,40,40);
    }
}
```

In CanvasMaus.java:

```
Canvas canvas = new Canvas(800, 800);
root.getChildren().add(canvas);

EventHandler<MouseEvent> mouseevent = new CanvasMausHandler(canvas);
canvas.addEventHandler(MouseEvent.MOUSE_PRESSED, mouseevent);
```

Objekt mit anonymer Klasse erzeugen

Für jeden Eventhandler eine eigene Klasse in einer eigenen Datei zu schreiben ist oft umständlich und unübersichtlich

- $\Rightarrow$ Anonyme Klassen können hier helfen
- Anonyme Klasse ist ein abkürzende Schreibweise, der Compiler behandelt es wie eine normale Klasse mit einem nur ihm bekannten Namen
- Anonyme Klassen erweitern eine existente Klasse oder implementieren ein Interface (im Beispiel Foo)
- ```
  Foo name = new Foo(){
    ...klassenvariablen...
    ...methodendefinitionen...
  };
  ```
  ```

Interaktion Beispiel: Maus mit anonymer Klasse

```
Canvas canvas = new Canvas(800, 800);
root.getChildren().add(canvas);

EventHandler<MouseEvent> mouseevent =
    new EventHandler<MouseEvent>(){

  public void handle(MouseEvent ev){
    double x = ev.getX();
    double y = ev.getY();
    GraphicsContext gc = canvas.getGraphicsContext2D();
    gc.setFill(Color.GREEN);
    gc.fillOval(x-20,y-20,40,40);
  }
};

canvas.addEventHandler(MouseEvent.MOUSE_PRESSED,
    mouseevent);
```

Interaktion Beispiel: Maus mit Lambda

Lambda ist ein Konzept anonymer Methoden

- seit 1958 in vielen Programmiersprachen vorhanden
- Java hat seit Java 8 (März 2014) Lambdas
- ermöglicht es, Events noch kompakter zu schreiben

```
Canvas canvas = new Canvas(800, 800);
root.getChildren().add(canvas);

canvas.addEventHandler(MouseEvent.MOUSE_PRESSED,
  ev -> {
     double x = ev.getX();
     double y = ev.getY();
     GraphicsContext gc = canvas.getGraphicsContext2D();
     gc.setFill(Color.GREEN);
     gc.fillOval(x-20,y-20,40,40);
  });
```

Events: Weitere Abkürzungen

Es gibt noch weitere Abkürzungen

Beispiel:

```
node.setOnMouseClicked(handler);
```

hat den gleichen Effekt wie

```
node.addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
```

Weiter Abkürzungen finden sich in der Dokumentation
(setOnDragDropped, setOnKeyTyped, setOnScroll, ...)

Interaktion 2 Property — Slider, . . .

Veränderliche Werte werden in JavaFX durch **Property**-Objekte repräsentiert.

Beispiel: Klasse `Slider`

```
// Wert des Sliders
DoubleProperty valueProperty()

// Hoehe der Node
DoubleProperty heightProperty();

// Ist der Mauszeiger gerade ueber der Node?
ReadOnlyBooleanProperty hoverProperty();

usw.
```

Property-Objekte

Ziel: Reagiere auf jede Änderung eines Werts in der GUI, z.B. die Position des Schiebereglers.

Idee: Anstelle eines `double`-Werts wird die Position des Schiebereglers durch ein `DoubleProperty`-Objekt repräsentiert.

- `DoubleProperty` kapselt einen privaten `double`-Wert.
- Zugriff auf den gekapselten Wert mit Getter und Setter:

```
public void set(double v);
public double get();
```

- Bei jeder Änderung des Werts (nur möglich über `set`), werden alle Interessenten über die Änderung informiert.
- Interessenten können sich mit der Methode `addListener` als "Zuhörer" registrieren. Sie werden dann über alle Änderungen informiert.

Interaktion Beipiel: Slider

```
Canvas canvas = new Canvas(800, 800);

Slider slider = new Slider(4, 300, 40);

root.getChildren().add(canvas);
root.getChildren().add(slider);

canvas.addEventHandler(MouseEvent.MOUSE_PRESSED,
    ev -> {
      double size = slider.getValue();
      double x = ev.getX();
      double y = ev.getY();
      GraphicsContext gc = canvas.getGraphicsContext2D();
      gc.setFill(Color.hsb(size % 255, 1, 1, 0.5));
      gc.fillOval(x-size/2,y-size/2,size,size);
    });
```

Canvas: Aktualisierung

Im Gegensatz zu Container-Nodes speichert ein Canvas nicht die gezeichneten Objekte, sondern nur Pixel.

- Die Aktualisierung eines Canvas erfolgt üblicherweise durch komplettes Löschen und Neuzeichnen.
- Wegen Anti-Aliasing ist Löschen einzelner Objekte schwierig.

```
// Inhalt loeschen
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.clearRect(0, 0, canvas.getWidth(),
                   canvas.getHeight());

// alles nochmal zeichnen
...
```

- Was neu gezeichnet werden soll, muss gespeichert sein.

Wiederholte Ausführung: Timeline, KeyFrame

Eine Timeline speichert eine Reihe von Aktionen, in zeitlich definierter Abfolge

- Eine einzelne dieser Aktionen nennt man KeyFrame.
- Timeline setzt sich aus KeyFrames zu definierten Zeitpunkten zusammen.
- Ermöglicht flexible Spezifikation flüssiger Bewegung von Objekten (Details dazu später).
- Auch unbegrenzte Wiederholung von Code möglich.

Timeline mit Endloswiederholung starten

- Aufbau des Keyframes: Möglich über Klassen, anonyme Klassen oder durch Lambdas

```
EventHandler<ActionEvent> handler = event -> {
... Code ...
};
KeyFrame keyframe =
  new KeyFrame(Duration.seconds(0.01), handler);
```

- Timeline zusammenbauen

```
Timeline tl = new Timeline();
tl.getKeyFrames().addAll(keyframe);
tl.setCycleCount(Timeline.INDEFINITE);
```

- Timeline starten

```
tl.play();
```

Interaktion Beispiel: Timeline

Zum erneuten Zeichnen müssen die Daten gespeichert werden:
Definition einer Klasse zum Speichern von Kreisen

```
public class ColCircle {

    double x;
    double y;
    double size;

    public ColCircle(double x, double y, double size) {
        this.x = x;
        this.y = y;
        this.size = size;
    }
}
```

Interaktion Beispiel: Timeline

```
List<ColCircle> cclist = new ArrayList<ColCircle>();

Canvas canvas = new Canvas(800, 800);
root.getChildren().add(canvas);

canvas.addEventHandler(MouseEvent.MOUSE_PRESSED, ev -> {
      double x = ev.getX(); double y = ev.getY();
      cclist.add(new ColCircle(x,y,400));
    });
EventHandler<ActionEvent> draw = e -> {
  GraphicsContext gc = canvas.getGraphicsContext2D();
  gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
  gc.setFill(Color.GREEN);
  for(ColCircle cc : cclist){
    gc.fillOval(cc.x - cc.size/2,cc.y - cc.size/2,cc.size,cc.size);
    cc.size *= 0.997;
  }
};

KeyFrame drawframe = new KeyFrame(Duration.seconds(0.01), draw);
Timeline tl = new Timeline();
tl.getKeyFrames().addAll(drawframe);
tl.setCycleCount(Timeline.INDEFINITE); // endlos wiederholen
tl.play();
```

Model, View, Controller

Zielsetzung: Trennung der Einzelteile eines Programms für bessere Wartbarkeit und Austauschbarkeit von Einzelteilen. Von Anfang an sollten diese Funktionalitäten in eigene Klassen aufgeteilt werden.

Model:
Ansammlung des gesamten Datenstands und der Algorithmen; enthält keinen Code zum Anzeigen

View:
Zeigt dem Benutzer die Daten an; speichert keinen Zustand zu den Daten, nur zu der Anzeige

Controller:
Ansammlung aller Einflußmöglichkeiten auf den Datenstand