Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024/2025

(Teil 14)

Visualisierung von Daten

- **Ziel**: Informationen in verständlicher, visueller Form darstellen
- **Vorteil**: Ein Bild sagt mehr als tausend Worte
- **Anwendungsfälle**:
  - Trends visualisieren
  - Datenpunkte vergleichen
  - Dynamische Veränderungen darstellen

Beispiel: Darstellung von Radfahrer-Statistiken
 Radfahrer-Statistiken

Lernziele

- Methoden zur Datenvisualisierung kennen
- **JavaFX-Schnittstellen** für Visualisierungen nutzen
- Dynamisch veränderliche Daten darstellen
- ...

Einführung: Desktop-Anwendungen und Notebook-Umgebungen

Zwei verschiedene Welten für Datenanalyse

Desktop-Anwendungen

- Traditionelle Programme mit eigenständiger Benutzeroberfläche.
- Persistente Speicherung, optimierte Performance.
- Beispiele: Excel, Tableau, MATLAB GUI.
- Vorteil: Oft effizienter für rechenintensive oder spezialisierte Anwendungen.

Notebook-Umgebungen

- Code und Visualisierung in interaktiven Dokumenten.
- Schrittweise Analyse von Daten, leicht anpassbar.
- Beispiele: Jupyter, Marimo.
- Vorteil: Ideal für explorative Datenanalyse, wissenschaftliches Arbeiten.

Visualisierung von Daten: Desktop vs. Notebook

Merkmal	Notebook-Umgebungen	Desktop-Anwendungen
Interaktion	Interaktive Zellen, Code änderbar	GUI-Steuerung, festgelegte Workflows
Flexibilität	Code kann jederzeit angepasst werden	Vorprogrammierte Funktionen
Performance	Speicherintensiv, aber flexibel	Optimiert für große Datenmengen
Anwendungsbereich	Data Science, Prototyping	Industrie, Unternehmenssoftware

Jupyter Notebook: Einführung und Datenvisualisierung

- Eine der beliebtesten Notebook-Umgebungen für Python.
- Ermöglicht das Schreiben von Code, Ausführen von Befehlen und Visualisierung in einer Umgebung.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.plot(x, y)
plt.title("Sinuskurve")
plt.show()
```

➡️ Erstellt eine einfache Sinuskurve direkt im Notebook.

siehe z.B. https://jupyter.org/

Jupyter Kernel: Was ist das?

- Ein (Jupyter) **Kernel** ist die Engine, die Code innerhalb eines Jupyter Notebooks ausführt.
- Jeder Kernel unterstützt eine bestimmte Programmiersprache.
- Standardmäßig verwendet Jupyter den **IPython-Kernel** für Python.
- Es gibt jedoch viele weitere Kernel für verschiedene Sprachen, z. B.:
  - **IJava** für Java
  - **IRKernel** für R
  - **IJavascript** für JavaScript
  - **Gophernotes** für Go
  - **Scala-Kernel** für Scala

➡️ Jupyter kann mehrere Kernel installieren und zwischen ihnen wechseln.

```
# Liste aller installierten Jupyter-Kernel anzeigen
jupyter kernelspec list
```

Java Notebooks: Einführung und Datenvisualisierung

- Jupyter kann auch (eher untypisch) mit Java verwendet werden, z. B. mit **IJava Kernel** (veraltet, aktiv weiterentwickelt: **JJava**).
- Ermöglicht interaktive Entwicklung mit Java.
- Kann für wissenschaftliche Berechnungen und Datenvisualisierung genutzt werden.

IJava

Live Demo : IJava

```
%maven org.knowm.xchart:xchart:3.5.2

import org.knowm.xchart.*;

double[] xData = new double[] { 0.0, 1.0, 2.0 };
double[] yData = new double[] { 2.0, 1.0, 0.0 };

XYChart chart = QuickChart.getChart("Sample Chart", "X", "Y", "y(x)", xData, yData);

BitmapEncoder.getBufferedImage(chart);
```

Besipiel Marimo: Einführung und Datenvisualisierung

- Erweiterung des Notebook-Konzepts mit reaktiver Programmierung (Python).
- Automatische Aktualisierung abhängiger Werte.
- Ideal für interaktive Dashboards.
- https://marimo.io/

```python
import marimo
import numpy as np
import matplotlib.pyplot as plt

x = mo.ui.slider(0, 10, value=5)
y = np.sin(np.linspace(0, x.value, 100))

fig, ax = plt.subplots()
ax.plot(np.linspace(0, x.value, 100), y)
mo.pyplot(fig)
```

➡️ Der Plot aktualisiert sich automatisch, wenn der Slider bewegt wird.

Reaktive Programmierung: Einführung

- Paradigma, das Abhängigkeiten automatisch aktualisiert.
- Statt manueller Neuberechnungen passt sich das System an Änderungen an.

Merkmale reaktiver Programmierung

✅ Automatische Datenaktualisierung.
✅ Geeignet für interaktive UIs und Dashboards.
✅ Flexibilität für explorative Datenanalysen.

✅ **Notebook-Umgebungen** bieten Flexibilität und explorative Analysen.
✅ **Jupyter** ermöglicht klassische interaktive Datenanalyse mit Python.
✅ **Java Notebooks** erlauben interaktive Programmierung mit Java.
✅ **Marimo** nutzt reaktive Programmierung für dynamische Updates.
✅ **Reaktive Programmierung** vereinfacht interaktive Visualisierungen.

Wichtige Programmierprinzipien:

- **Imperative Programmierung** (klassische Skripte).
- **Interaktive Entwicklung** (Jupyter Notebooks, Java Notebooks).
- **Reaktive Programmierung** (Marimo).

Rohdaten und ihre Verarbeitung

Beispiel für Rohdaten:

```
Päivämäärä;Huopalahti (asema);Kaisaniemi;...
ke 1 tammi 2014 00:00;;1;;;;;;2;...
```

Warum visualisieren?

- Rohdaten sind schwer lesbar
- Visualisierungen heben Muster und Trends hervor

Code zur Verarbeitung von CSV-Daten:

```java
String row = "Päivämäärä;Huopalahti (asema);...";
String[] pieces = row.split(";");
for (int i = 0; i < pieces.length; i++) {
    System.out.println(i + ": " + pieces[i]);
}
```

Diagramme in JavaFX

- **JavaFX** bietet vorgefertigte Klassen für:
  - Liniendiagramme
  - Balkendiagramme
  - Flächendiagramme
- **Dokumentation**:
  - JavaFX Charts API
  - Oracle Chart Guide (veraltet, aber hilfreich)

Liniendiagramm

Anwendungsfall:
Darstellung von Veränderungen über die Zeit.

Beispiel: Unterstützung für finnische Parteien (1968–2008).
Datenformat: Tab-getrennt

```
Party    1968    1972    ...
KOK      16.1    18.1    ...
```

Code-Schnipsel: Verarbeitung von Daten

```
String row = "Party    1968    1972    ...";
String[] pieces = row.split("\t");
for (String piece : pieces) {
    System.out.println(piece);
}
```

Liniendiagramm: Beispielcode

```
NumberAxis xAxis = new NumberAxis(1968, 2008, 4);
NumberAxis yAxis = new NumberAxis();
LineChart<Number, Number> chart = new LineChart<>(xAxis, yAxis);

XYChart.Series series = new XYChart.Series();
series.setName("RKP");
series.getData().add(new XYChart.Data(1968, 5.6));
series.getData().add(new XYChart.Data(1972, 5.2));

chart.getData().add(series);

Scene scene = new Scene(chart, 640, 480);
stage.setScene(scene);
stage.show();
```

Ergebnis:


RKP Unterstützung

Dynamische Datenvisualisierung von Zeitreihen

Anwendungsfälle:

- Aktienkurse , Wetterdaten, Server-Monitoring ...

Beispiel: Gesetz der großen Zahlen

```
new AnimationTimer() {
    @Override
    public void handle(long now) {
        int roll = random.nextInt(6) + 1;
        sum += roll;
        count++;
        series.getData().add(new XYChart.Data(count, sum / (double) count));
    }
}.start();
```

 Gesetz der großen Zahlen

Beachte: Ist das `ObservableInterface` implementiert? (hier: Nein)

Verwendung eines Observable-Modells für Animationen

Warum brauchen wir ein Observable Modell?

- In einer JavaFX-Anwendung sollen sich **Daten automatisch aktualisieren**.
- Wenn sich Werte ändern (z. B. Durchschnittswert eines Würfelwurfs), soll die UI **automatisch reagieren**, ohne expliziten Refresh.
- JavaFX bietet **Properties**, die Werte speichern und Änderungen melden.
- Dafür implementieren wir das **Observable-Interface**.

➡️ **Lösung**: Ein separates `DiceModel`, das `Observable` implementiert.

Das `DiceModel`: Trennung von Logik und UI

```
class DiceModel implements Observable {
    private final DoubleProperty averageRoll = new SimpleDoubleProperty();
    private final Random random = new Random();
    private long sum = 0;
    private long count = 0;
    private long previousTime = 0;

    private final AnimationTimer timer = new AnimationTimer() {
        @Override
        public void handle(long now) {
            if (now - previousTime < 100_000_000L) return;
            previousTime = now;
            int roll = random.nextInt(6) + 1;
            sum += roll;
            count++;
            averageRoll.set(sum / (double) count);
        }
    };

    public DiceModel() {
        timer.start();
    }

    public DoubleProperty averageRollProperty() {
        return averageRoll;
    }

    @Override
    public void addListener(InvalidationListener listener) {
        averageRoll.addListener(listener);
    }

    @Override
    public void removeListener(InvalidationListener listener) {
        averageRoll.removeListener(listener);
    }
}
```

➡️ `DiceModel` verwaltet die Berechnungslogik und läuft unabhängig von der UI.

Warum `Observable` implementieren?

- Damit die UI automatisch aktualisiert wird, muss sie über Änderungen am Model informiert werden.
- `DoubleProperty` speichert den Durchschnittswert und **meldet Änderungen**.
- Das UI-Element (`LineChart`) **reagiert auf Änderungen** und aktualisiert das Diagramm.

➡️ **Ohne `Observable` müsste die UI manuell aktualisiert werden!**

Die Verbindung von `DiceModel` mit der UI

```
// Erstellen des Observable-Modells
DiceModel diceModel = new DiceModel();

// UI an das Observable binden
XYChart.Series<Number, Number> series = new XYChart.Series<>();
lineChart.getData().add(series);

diceModel.averageRollProperty().addListener((obs, oldVal, newVal) -> {
    series.getData().add(new XYChart.Data<>(series.getData().size() + 1, newVal.doubleValue()));
});
```

- Jedes Mal, wenn sich der Wert von `averageRoll` ändert, wird ein neuer Punkt hinzugefügt.
- Live Demo

✅ **Trennung von Logik und UI**: `DiceModel` führt Berechnungen durch, `LineChart` zeigt die Daten.
✅ **Observable für automatische Updates**: Keine manuelle UI-Aktualisierung nötig.
✅ **AnimationTimer für Echtzeit-Updates**: Ständiges Aktualisieren neuer Werte.
✅ **Binding an `DoubleProperty`** : Änderungen werden direkt im Diagramm sichtbar.

➡️ **Ergebnis:** Eine (un)saubere Architektur für dynamische Visualisierungen!

Verwendung eines MVC-Modells für Animationen

Warum brauchen wir ein MVC-Modell?

- In einer JavaFX-Anwendung sollen sich **Daten automatisch aktualisieren**.
- Wenn sich Werte ändern (z. B. Durchschnittswert eines Würfelwurfs),
  soll die UI **automatisch reagieren**, ohne expliziten Refresh.
- JavaFX bietet **Bindings**, um Werte zu aktualisieren.
- Dafür trennen wir **Model, Controller und View**.

➡️ **Lösung**: Ein separates `DiceModel` und ein `DiceController`, der die Ablauflogik steuert.

Das `DiceModel`: Trennung von Logik und UI

```
class DiceModel {
    private final Random random = new Random();
    private long sum = 0;
    private long count = 0;
    private Consumer<Double> listener;

    public void setListener(Consumer<Double> listener) {
        this.listener = listener;
    }

    public void rollDice() {
        int roll = random.nextInt(6) + 1;
        sum += roll;
        count++;
        double average = sum / (double) count;
        if (listener != null) {
            listener.accept(average);
        }
    }
}
```

➡️ `DiceModel` verwaltet die Berechnungslogik und läuft unabhängig von der UI.

Warum MVC statt `Observable`?

- Bessere Testbarkeit: `DiceModel` kann ohne JavaFX getestet werden.
- `DiceController` steuert den Ablauf unabhängig von der UI.
- Die UI erhält nur Updates, ohne die Logik zu kennen.

➡️ **Ohne MVC müsste die UI direkt auf Model-Änderungen reagieren!**

Die Verbindung von `DiceModel` mit der UI

```
class DiceController {
    private final DiceModel model;
    private final AnimationTimer timer;

    public DiceController(DiceModel model, Consumer<Double> onUpdate) {
        this.model = model;
        this.model.setListener(onUpdate);

        this.timer = new AnimationTimer() {
            private long previousTime = 0;

            @Override
            public void handle(long now) {
                if (now - previousTime >= 100_000_000L) {
                    previousTime = now;
                    model.rollDice();
                }
            }
        };
    }

    public void start() {
        timer.start();
    }
}
```

➡️ `DiceController` steuert das `DiceModel` und meldet Änderungen an die UI.

✅ **Trennung von Logik und UI**: `DiceModel` speichert Daten, `DiceController` steuert Abläufe, UI zeigt an.
✅ **Event-Listener für automatische Updates**: Keine JavaFX-Abhängigkeiten im Modell.
✅ **AnimationTimer für Echtzeit-Updates**: Modell aktualisiert sich periodisch.
✅ **Flexible Architektur**: Modell kann in anderen Anwendungen (z. B. Web, CLI) genutzt werden.

➡️ **Ergebnis:** Eine saubere, testbare und flexible Architektur für dynamische Visualisierungen!

Die `DiceChartApp`: Erstellung des Controllers und Listener-Registrierung

```
public class DiceChartApp extends Application {
    @Override
    public void start(Stage stage) {
        LineChart<Number, Number> lineChart = new LineChart<>(new NumberAxis(), new NumberAxis(1, 6, 1));
        XYChart.Series<Number, Number> series = new XYChart.Series<>();
        lineChart.getData().add(series);

        DiceModel diceModel = new DiceModel();
        DiceController controller = new DiceController(diceModel, newVal -> {
            series.getData().add(new XYChart.Data<>(series.getData().size() + 1, newVal));
        });

        controller.start();

        stage.setScene(new Scene(lineChart, 400, 300));
        stage.setTitle("Durchschnittlicher Würfelwurf");
        stage.show();
    }
}
```

➡️ **Erstellt das Modell, registriert einen Listener und startet den Controller für Echtzeit-Updates.**

Erstellung des Controllers mit Listener-Registrierung

```
DiceController controller = new DiceController(diceModel, newVal -> {
    series.getData().add(new XYChart.Data<>(series.getData().size() + 1, newVal));
});
```

Was passiert hier?

- Der `DiceController` wird mit zwei Argumenten erstellt:
  i. Das `DiceModel`, das die Würfelstatistik verwaltet.
  ii. Eine **Lambda-Funktion** als Listener (`newVal -> {...}`), die automatisch auf Änderungen reagiert.
- Jedes Mal, wenn sich der Durchschnittswert (`newVal`) im `DiceModel` ändert, wird ein neuer Punkt zum `series`-Diagramm hinzugefügt.
- Dadurch aktualisiert sich die UI **automatisch** ohne expliziten Refresh.

➡️ **Vorteil:** Die UI bleibt synchron mit den Daten!

Datenfluss in `DiceController` und `DiceModel`

```
public DiceController(DiceModel model, Consumer<Double> onUpdate) {
    this.model = model;
    this.model.setListener(onUpdate);

    this.timer = new AnimationTimer() {
        private long previousTime = 0;

        @Override
        public void handle(long now) {
            if (now - previousTime >= 100_000_000L) {
                previousTime = now;
                model.rollDice();
            }
        }
    };
}
```

Wie funktioniert die Aktualisierung?

1. Der `AnimationTimer` ruft alle 100ms `model.rollDice()` auf.
2. Das `DiceModel` berechnet einen neuen Durchschnitt und ruft die Listener-Funktion (`onUpdate.accept(average)`) auf.
3. Die registrierte **Lambda-Funktion** fügt den neuen Wert ins Diagramm (`series`) ein.
4. **Das Diagramm aktualisiert sich automatisch!**

➡️ **Ergebnis:** Eine elegante, reaktive Architektur für dynamische Datenvisualisierung. 🚀

Schritt	Was passiert?
1	`AnimationTimer` ruft `model.rollDice()` auf.
2	`rollDice()` berechnet einen neuen Durchschnitt (`average`).
3	`listener.accept(average)` wird aufgerufen.
4	Der registrierte `Consumer<Double>` (`newVal -> {...}`) wird ausgeführt.
5	`series.getData().add(new XYChart.Data<>(series.getData().size() + 1, newVal))` fügt einen neuen Punkt ins Diagramm ein.
6	Das `LineChart` wird automatisch aktualisiert, weil `series.getData()` eine `ObservableList` ist.

Merkmal	Observable-Architektur	MVC
Ziel	Reaktive, Event-gesteuerte Aktualisierung	Klare Trennung von Logik & Darstellung
Datenfluss	Änderungen pushen Daten direkt in die UI	Controller entscheidet, wann Daten aktualisiert werden
Flexibilität	Sehr dynamisch, leicht anpassbar	Besser strukturiert, modularer
Testbarkeit	UI-abhängiger, schwierig zu isolieren	Model kann gut unabhängig getestet werden

Balkendiagramme

Anwendungsfall: Vergleich von Kategorien.

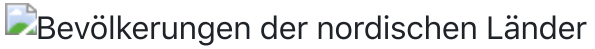
Beispiel: Bevölkerung der nordischen Länder.

```
Iceland, 343518
Norway, 5372191
...
```

Code-Schnipsel:

```
CategoryAxis xAxis = new CategoryAxis();
NumberAxis yAxis = new NumberAxis();
BarChart<String, Number> chart = new BarChart<>(xAxis, yAxis);

XYChart.Series series = new XYChart.Series();
series.getData().add(new XYChart.Data("Iceland", 343518));
chart.getData().add(series);
```

Ergebnis:


Bevölkerungen der nordischen Länder

Zusammenfassung

- **Liniendiagramme**: Trends und zeitliche Veränderungen
- **Balkendiagramme**: Vergleich von Kategorien
- **Dynamische Daten**: Echtzeit-Visualisierungen mit AnimationTimer

JavaFX bietet Klassen für Datenvisualisierungen, die mit wenig Code eingesetzt werden können.

Multimedia in Programmen

- **Multimedia-Inhalte in grafischen Oberflächen**:
  - Zeichnen (Canvas)
  - Bilder anzeigen
  - Sound abspielen
- **Anwendungsfälle**:
  - Interaktive Anwendungen (Zeichenprogramme)
  - Visualisierung komplexer Informationen
  - Benutzerfreundlichkeit durch Soundeffekte erhöhen

Lernziele

- Multimedia in grafischen Oberflächen (UI) nutzen
- **Zeichnen** auf einem Canvas-Objekt
- **Bilder** in einer grafischen Oberfläche anzeigen
- **Sound** in einer Anwendung abspielen

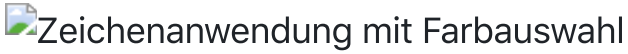
Zeichnen mit JavaFX

- **Canvas**: Zeichenfläche
- **GraphicsContext**: Stift zum Zeichnen
- **Ereignisse** (z. B. Mausbewegung) zum Zeichnen nutzen

Codebeispiel: Zeichenanwendung

```
Canvas paintingCanvas = new Canvas(640, 480);
GraphicsContext painter = paintingCanvas.getGraphicsContext2D();
ColorPicker colorPalette = new ColorPicker();

paintingCanvas.setOnMouseDragged((event) -> {
    painter.setFill(colorPalette.getValue());
    painter.fillOval(event.getX(), event.getY(), 4, 4);
});
```

Ergebnis:


Zeichenanwendung mit Farbauswahl

Bilder in JavaFX

- **Image**: Bild laden
- **ImageView**: Bild in der UI anzeigen

Codebeispiel:

```
Image imageFile = new Image("file:humming.jpg");
ImageView image = new ImageView(imageFile);

Pane frame = new Pane();
frame.getChildren().add(image);
```

Ergebnis:

 Beispiel eines Bildes in der UI

Bilder bearbeiten

Einzelne Pixel manipulieren:

- **PixelReader**: Pixelwerte auslesen
- **PixelWriter**: Pixelwerte setzen

Codebeispiel: rotieren, skalieren, verschieben

```
@Override
public void start(Stage stage) {
    Image imageFile = new Image("file:humming.jpg");
    ImageView image = new ImageView(imageFile);

    image.setRotate(180);
    image.setScaleX(0.5);
    image.setScaleY(0.5);

    image.setTranslateX(50);

    Pane frame = new Pane();
    frame.getChildren().add(image);

    stage.setScene(new Scene(frame));
    stage.show();
}
```

Ergebnis:


Negativ eines Bildes

Sound in JavaFX

- **AudioClip**: Soundeffekte einfach abspielen

Codebeispiel:

```
AudioClip sound = new AudioClip("file:bell.wav");
sound.play();
```

Ergebnis:

Der Soundeffekt wird abgespielt, wenn das Programm startet.

Zusammenfassung

- **Canvas**: Zeichnen auf einer UI
- **Image und ImageView**: Bilder anzeigen
- **AudioClip**: Soundeffekte abspielen
- **PixelReader/-Writer**: Bildmanipulation

Multimedia bereichert Anwendungen durch visuelle und akustische Inhalte.

Größere Anwendung: Asteroids

Lernziele

- Sie wissen, wie ein interaktives Spiel implementiert wird.
- Sie können sich vorstellen, wie eine größere Anwendung Schritt für Schritt aufgebaut wird.
- Sie üben, Schritt-für-Schritt-Anweisungen zu befolgen, um eine größere Anwendung zu erstellen.

Einführung: Asteroids

Asteroids wurde 1979 von Atari entwickelt und ist ein Videospiel-Klassiker.

- Steuerung eines Raumschiffs
- Ziel: Asteroiden abschießen

https://freeasteroids.org/

Erstellen des Spielfensters

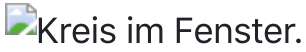
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class PaneExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        Pane pane = new Pane();
        pane.setPrefSize(300, 200);
        Scene scene = new Scene(pane);
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

- Fenstergröße: 300 x 200 Pixel
- Ursprung: oben links

Einfügen eines Kreises

```
pane.getChildren().add(new Circle(30, 50, 10));
```

- Position: (30, 50)
- Radius: 10 Pixel

 Kreis im Fenster.

Erstellen des Raumschiffs

```
@Override
public void start(Stage stage) throws Exception {
    Pane pane = new Pane();
    pane.setPrefSize(300, 200);

    Polygon parallelogram = new Polygon(0, 0, 100, 0, 100, 50, 0, 50);
    parallelogram.setTranslateX(100);
    parallelogram.setTranslateY(20);

    pane.getChildren().add(parallelogram);

    Scene scene = new Scene(pane);
    stage.setScene(scene);
    stage.show();
}
```

- Hier: Darstellung des "Raumschiffs" als Rechteck
- Zentriert auf das Spielfenster

 Raumschiff im Fenster.

Erstellen des Raumschiffs

```
Polygon ship = new Polygon(-5, -5, 10, 0, -5, 5);
ship.setTranslateX(300);
ship.setTranslateY(200);
pane.getChildren().add(ship);
```

- Hier: Darstellung des "Raumschiffs" als Dreieck

Drehen des Raumschiffs

```
scene.setOnKeyPressed(event -> {
    if (event.getCode() == KeyCode.LEFT) {
        ship.setRotate(ship.getRotate() - 5);
    }
    if (event.getCode() == KeyCode.RIGHT) {
        ship.setRotate(ship.getRotate() + 5);
    }
});
```

- Linke Pfeiltaste: -5° drehen
- Rechte Pfeiltaste: +5° drehen


Drehen des Raumschiffs.

Flüssiges Drehen

```
Map<KeyCode, Boolean> pressedKeys = new HashMap<>();

scene.setOnKeyPressed(event -> {
    pressedKeys.put(event.getCode(), Boolean.TRUE);
});

scene.setOnKeyReleased(event -> {
    pressedKeys.put(event.getCode(), Boolean.FALSE);
});

new AnimationTimer() {
    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
            ship.setRotate(ship.getRotate() - 5);
        }
        if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
            ship.setRotate(ship.getRotate() + 5);
        }
    }
}.start();
```

- `HashMap pressedKeys` mit `KeyCode`s und `Boolean`s
- AnimationTimer für kontinuierliche Updates (etwa 60 Updates in der Sekunde)
- Flüssiges Drehen des Raumschiffs

 Flüssiges Drehen des Raumschiffs.

Bewegen des Raumschiffs und Beschleunigung

Bewegen des Raumschiffs: Problem mit `Point2D`

```
Point2D movement = new Point2D(1, 0);

new AnimationTimer() {

    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
            ship.setRotate(ship.getRotate() - 5);
        }

        if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
            ship.setRotate(ship.getRotate() + 5);
        }

        ship.setTranslateX(ship.getTranslateX() + movement.getX());
    }
}.start();
```

`Point2D` um Bewegung darzustellen

Bewegen des Raumschiffs: Problem mit `Point2D`

- Die Klasse `Point2D` ist *unveränderlich* (immutable).
- Änderungen an der Bewegung (z. B. durch Beschleunigung) erfordern die Erstellung eines neuen Objekts.

Nicht mögliche Lösung:

```
movement = movement.add(new Point2D(1, 1));
```

Warum?

- Methodenaufrufe geben immer ein neues `Point2D`-Objekt zurück.
- Das direkte Ändern vorhandener Objekte ist nicht möglich.

Refactoring: Die `Ship`-Klasse

- Erstellung einer `Ship`-Klasse zur besseren Strukturierung.
- Enthält:
  - **`Polygon`**: Darstellung des Raumschiffs.
  - **`Point2D`**: Speicherung der Bewegung.

Konstruktor:

```
public Ship(int x, int y) {
    this.character = new Polygon(-5, -5, 10, 0, -5, 5);
    this.character.setTranslateX(x);
    this.character.setTranslateY(y);
    this.movement = new Point2D(0, 0);
}
```

```java
import javafx.geometry.Point2D;
import javafx.scene.shape.Polygon;

public class Ship {

    private Polygon character;
    private Point2D movement;

    public Ship(int x, int y) {
        this.character = new Polygon(-5, -5, 10, 0, -5, 5);
        this.character.setTranslateX(x);
        this.character.setTranslateY(y);

        this.movement = new Point2D(0, 0);
    }

    public Polygon getCharacter() {
        return character;
    }

    public void turnLeft() {
        this.character.setRotate(this.character.getRotate() - 5);
    }

    public void turnRight() {
        this.character.setRotate(this.character.getRotate() + 5);
    }

    public void move() {
        this.character.setTranslateX(this.character.getTranslateX() + this.movement.getX());
        this.character.setTranslateY(this.character.getTranslateY() + this.movement.getY());
    }
}
```

Methoden der `Ship`-Klasse

- **Drehen des Raumschiffs:**

```
public void turnLeft() {
    this.character.setRotate(this.character.getRotate() - 5);
}

public void turnRight() {
    this.character.setRotate(this.character.getRotate() + 5);
}
```

- **Bewegen des Raumschiffs:**

```
public void move() {
    this.character.setTranslateX(this.character.getTranslateX() + this.movement.getX());
    this.character.setTranslateY(this.character.getTranslateY() + this.movement.getY());
}
```

Verwendung der `Ship`-Klasse

- Anstelle eines `Polygon`-Objekts wird ein `Ship`-Objekt erstellt.
- Das `Pane`-Objekt erhält das Polygon des Raumschiffs:

```
Ship ship = new Ship(150, 100);
pane.getChildren().add(ship.getCharacter());
```

- `ship.getCharacter()` bleibt immer das gleiche Objekt, wird dadurch neu gezeichnet!
- `AnimationTimer` nutzt die Methoden der `Ship`-Klasse:

```
new AnimationTimer() {
    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
            ship.turnLeft();
        }
        if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
            ship.turnRight();
        }
        ship.move();
    }
}.start();
```

Beschleunigung des Raumschiffs

- Ziel: Das Raumschiff beschleunigt in die Richtung, in die es zeigt.
- Umsetzung durch Sinus- und Kosinus-Funktionen:

```
double changeX = Math.cos(Math.toRadians(this.character.getRotate()));
double changeY = Math.sin(Math.toRadians(this.character.getRotate()));
this.movement = this.movement.add(changeX, changeY);
```

Methoden:

- `Math.toRadians`: Umwandlung von Grad in Radiant.
- `Math.cos`, `Math.sin`: Berechnung der Richtung.

Beschleunigung in der `AnimationTimer`

- Pfeiltaste **oben** löst Beschleunigung aus.

Code:

```
new AnimationTimer() {
    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.UP, false)) {
            ship.accelerate();
        }
        ship.move();
    }
}.start();
```

Animation:


Das Raumschiff beschleunigt

Beschleunigung mit `Point2D` in JavaFX

Wie funktioniert die Methode `accelerate()`?

- Berechnet die Bewegungsrichtung basierend auf der Rotation (`getRotate()`).
- Nutzt `Math.cos()` und `Math.sin()`, um die X- und Y-Komponenten zu bestimmen.
- Fügt diese zur bestehenden Bewegung hinzu (`movement.add(changeX, changeY)`).

Methode:

```
public void accelerate() {
    double changeX = Math.cos(Math.toRadians(this.character.getRotate()));
    double changeY = Math.sin(Math.toRadians(this.character.getRotate()));
    this.movement = this.movement.add(changeX, changeY);
}
```

➡️ Das Objekt bewegt sich in die Richtung, in die es zeigt! 🚀

Beispiel: Rotation = 0° (Nach rechts)

Gegeben:

- `this.character.getRotate() = 0°`
- `this.movement = new Point2D(2, 3);`

Berechnung:

```
double angle = Math.toRadians(0);  // 0° → 0 Radian
double changeX = Math.cos(angle);  // 1.0
double changeY = Math.sin(angle);  // 0.0
this.movement = this.movement.add(changeX, changeY);
```

➡️ Neues **`Point2D`**: `(3.0, 3.0)`

➡️ Bewegung verstärkt sich nach rechts.

Beispiel: Rotation = 90° (Nach unten)

Gegeben:

- `this.character.getRotate() = 90°`
- `this.movement = new Point2D(2, 3);`

Berechnung:

```
double angle = Math.toRadians(90); // 90° → 1.5708 Radian
double changeX = Math.cos(angle);  // 0.0
double changeY = Math.sin(angle);  // 1.0
this.movement = this.movement.add(changeX, changeY);
```

➡️ Neues **`Point2D`**: `(2.0, 4.0)`

➡️ **Das Objekt bewegt sich nun nach unten.**

Beispiel: Rotation = 45° (Diagonal rechts unten)

Gegeben:

- `this.character.getRotate() = 45°`
- `this.movement = new Point2D(2, 3);`

Berechnung:

```
double angle = Math.toRadians(45); // 45° → 0.7854 Radian
double changeX = Math.cos(angle);  // ≈ 0.707
double changeY = Math.sin(angle);  // ≈ 0.707
this.movement = this.movement.add(changeX, changeY);
```

➡️ Neues **`Point2D`**: `(2.707, 3.707)`

➡️ Das Objekt bewegt sich nun diagonal nach rechts unten.

✅ `Math.cos()` und `Math.sin()` bestimmen die Bewegungsrichtung.
✅ Die Werte werden zu `this.movement` addiert, sodass das Objekt beschleunigt.
✅ Das Objekt bewegt sich in die Richtung, in die es zeigt.
✅ Ändert sich `getRotate()`, dann passt sich die Bewegung automatisch an.

Anpassung der Beschleunigung

- Problem: Beschleunigung des Raumschiffs ist zu stark.
- Lösung: Beschleunigung wird auf 5 % der bisherigen Werte reduziert.

Methode `accelerate`:

```
public void accelerate() {
    double changeX = Math.cos(Math.toRadians(this.character.getRotate()));
    double changeY = Math.sin(Math.toRadians(this.character.getRotate()));

    changeX *= 0.05;
    changeY *= 0.05;

    this.movement = this.movement.add(changeX, changeY);
}
```

Ergebnis:

Das Raumschiff lässt sich präziser steuern.

Animation:

 Das Raumschiff beschleunigt kontrollierter.

Verallgemeinerung und Erstellung von Asteroiden

Einführung: Erstellen eines Asteroiden

- Ein Asteroid hat ähnliche Eigenschaften wie ein Raumschiff:
  - **Form**
  - **Position**
  - **Bewegung**
- Ziel: Verallgemeinerung der gemeinsamen Eigenschaften in einer abstrakten Klasse `Character`.

Die abstrakte Klasse `Character`

- Speichert Eigenschaften und Methoden für beliebige Spielfiguren:
  - Form (Polygon)
  - Bewegung (Point2D)
  - Methoden für Drehen, Bewegung und Beschleunigung

Code:

```
import javafx.geometry.Point2D;
import javafx.scene.shape.Polygon;

public abstract class Character {

    private Polygon character;
    private Point2D movement;

    public Character(Polygon polygon, int x, int y) {
        this.character = polygon;
        this.character.setTranslateX(x);
        this.character.setTranslateY(y);
        this.movement = new Point2D(0, 0);
    }

    public Polygon getCharacter() {
        return character;
    }
    // Weitere Methoden: turnLeft, turnRight, move, accelerate
}
```

Änderung der Klasse `Ship`

- `Ship` erbt jetzt von `Character`:
  - Reduziert Redundanz.
  - Übernimmt Funktionalität der abstrakten Klasse.

Code:

```
import javafx.scene.shape.Polygon;

public class Ship extends Character {
    public Ship(int x, int y) {
        super(new Polygon(-5, -5, 10, 0, -5, 5), x, y);
    }
}
```

- Einfach und elegant.

Erstellung der Klasse `Asteroid`

- **Asteroid** ist ebenfalls ein `Character`.
- Erste Version mit Rechteckform.

Code:

```
import javafx.scene.shape.Polygon;

public class Asteroid extends Character {
    public Asteroid(int x, int y) {
        super(new Polygon(20, -20, 20, 20, -20, 20, -20, -20), x, y);
    }
}
```

Testen: Hinzufügen von Asteroiden

- Erstellung eines Asteroiden und eines Raumschiffs:

Code:

```
Ship ship = new Ship(150, 100);
Asteroid asteroid = new Asteroid(50, 50);

pane.getChildren().add(ship.getCharacter());
pane.getChildren().add(asteroid.getCharacter());
```

- **Bewegung des Asteroiden:**
  - Methode `turnRight` wird aufgerufen.
  - Mehrfacher Aufruf von `accelerate` für Geschwindigkeit.

Animation:

 Die Anwendung enthält ein Raumschiff und einen Asteroiden.

Integration: Bewegung von Asteroiden

- **AnimationTimer:**
  - Bewegt Raumschiff und Asteroiden.
  - Verwendet Methoden der `Character`-Klasse.

Code:

```
new AnimationTimer() {
    @Override
    public void handle(long now) {
        // [...] Rotation und Beschleunigung
        ship.move();
        asteroid.move();
    }
}.start();
```

Ergebnis:
Das Spielfenster enthält jetzt ein bewegliches Raumschiff und einen Asteroiden.

Animation:
 Die Anwendung enthält ein Raumschiff und einen Asteroiden.

Code: AnimationTimer

```
new AnimationTimer() {
    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
            ship.turnLeft();
        }

        if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
            ship.turnRight();
        }

        if (pressedKeys.getOrDefault(KeyCode.UP, false)) {
            ship.accelerate();
        }

        ship.move();
        asteroid.move();
    }
}.start();
```

Kollision zwischen Raumschiff und Asteroid

- **Ziel:** Erkennen, wenn das Raumschiff mit einem Asteroiden kollidiert.
- **Verhalten:** Stoppen der Animation bei einer Kollision.
- **Lösung:** Implementieren einer Methode in der `Character`-Klasse.
- **Initialisierung:** Derzeit kollidieren zwei Charaktere nie.

Code: (initial, nichts kollidiert)

```
public boolean collide(Character other) {
    return false;
}
```

Verbesserung der Kollisionsmethode

- **Nutzen der `Shape`-Klasse:** (`Polygon` erbt von `Shape` s)
  - Methode `intersect` bestimmt den Überschneidungsbereich von zwei Formen.
  - Gibt den Schnittbereich als `Shape` zurück.

Angepasste Methode:

```
public boolean collide(Character other) {
    Shape collisionArea = Shape.intersect(this.character, other.getCharacter());
    return collisionArea.getBoundsInLocal().getWidth() != -1;
}
```

Integration der Kollisionserkennung

- **AnimationTimer:** Stoppen der Animation bei einer Kollision.
- **Erweiterung des Timers:**
  - Bewegung des Raumschiffs und des Asteroiden.
  - Kollisionsprüfung.

Code:

```
new AnimationTimer() {

    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
            ship.turnLeft();
        }

        if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
            ship.turnRight();
        }

        if (pressedKeys.getOrDefault(KeyCode.UP, false)) {
            ship.accelerate();
        }

        ship.move();
        asteroid.move();

        if (ship.collide(asteroid)) {
            stop();
        }
    }
}.start();
```

Mehrere Asteroiden hinzufügen

- **Ziel:** Hinzufügen einer Liste von Asteroiden.
- **Implementierung:**
  - Erstellen von Asteroiden mit zufälligen Positionen.
  - Hinzufügen der Asteroiden zum `Pane`.

Code:

```
Ship ship = new Ship(150, 100);
List<Asteroid> asteroids = new ArrayList<>();
for (int i = 0; i < 5; i++) {
    Random rnd = new Random();
    Asteroid asteroid = new Asteroid(rnd.nextInt(100), rnd.nextInt(100));
    asteroids.add(asteroid);
}

pane.getChildren().add(ship.getCharacter());
asteroids.forEach(asteroid -> pane.getChildren().add(asteroid.getCharacter()));
```

Bewegung und Kollisionsprüfung für mehrere Asteroiden

- **Ziel:**
  - Bewegung aller Asteroiden.
  - Kollisionsprüfung zwischen Raumschiff und jedem Asteroiden.

Code:

```
new AnimationTimer() {

    @Override
    public void handle(long now) {
        if (pressedKeys.getOrDefault(KeyCode.LEFT, false)) {
            ship.turnLeft();
        }

        if (pressedKeys.getOrDefault(KeyCode.RIGHT, false)) {
            ship.turnRight();
        }

        if (pressedKeys.getOrDefault(KeyCode.UP, false)) {
            ship.accelerate();
        }

        ship.move();
        asteroids.forEach(asteroid -> asteroid.move());

        asteroids.forEach(asteroid -> {
            if (ship.collide(asteroid)) {
                stop();
            }
        });

    }
}.start();
```

Ergebnis:

- Mehrere Asteroiden bewegen sich im Fenster.
- Die Animation stoppt bei einer Kollision.

Bild:
 Mehrere Asteroiden.

Variation der Asteroiden

- **Ziel:** Unterschiedliche Formen und Bewegungen.
- **Methode:**
  - Zufällige Variationen in der Geometrie der Asteroiden.
  - Definieren eines Fünfecks als Grundform.
  - Anpassung der Eckenpositionen.

Code:

```
import java.util.Random;
import javafx.scene.shape.Polygon;

public class PolygonFactory {

    public Polygon createPolygon() {
        Random rnd = new Random();

        double size = 10 + rnd.nextInt(10);

        Polygon polygon = new Polygon();
        double c1 = Math.cos(Math.PI * 2 / 5);
        double c2 = Math.cos(Math.PI / 5);
        double s1 = Math.sin(Math.PI * 2 / 5);
        double s2 = Math.sin(Math.PI * 4 / 5);

        polygon.getPoints().addAll(
            size, 0.0,
            size * c1, -1 * size * s1,
            -1 * size * c2, -1 * size * s2,
            -1 * size * c2, size * s2,
            size * c1, size * s1);

        for (int i = 0; i < polygon.getPoints().size(); i++) {
            int change = rnd.nextInt(5) - 2;
            polygon.getPoints().set(i, polygon.getPoints().get(i) + change);
        }

        return polygon;
    }
}
```

Zufallsformen für Asteroiden

- **Ziel:** Erstellen von Asteroiden mit unterschiedlicher Geometrie.
- **Methode:** Verwenden einer `PolygonFactory`, um zufällige Fünfecke zu erzeugen.

Code:

```
public class Asteroid extends Character {

    public Asteroid(int x, int y) {
        super(new PolygonFactory().createPolygon(), x, y);
    }
}
```

Ergebnis:

- Jeder Asteroid hat eine leicht unterschiedliche Form.

Bild:
 Die Asteroiden sind variabler.

Bewegung und Richtung der Asteroiden

- **Ziel:** Hinzufügen von Zufälligkeit in Bewegung und Rotation.
- **Ansatz:**
  - Bewegung wird durch mehrere `accelerate`-Aufrufe erzeugt.
  - Rotation wird zufällig bestimmt.

Code:

```
import java.util.Random;

public class Asteroid extends Character {

    private double rotationalMovement;

    public Asteroid(int x, int y) {
        super(new PolygonFactory().createPolygon(), x, y);

        Random rnd = new Random();

        super.getCharacter().setRotate(rnd.nextInt(360));

        int accelerationAmount = 1 + rnd.nextInt(10);
        for (int i = 0; i < accelerationAmount; i++) {
            accelerate();
        }

        this.rotationalMovement = 0.5 - rnd.nextDouble();
    }

    @Override
    public void move() {
        super.move();
        super.getCharacter().setRotate(super.getCharacter().getRotate() + rotationalMovement);
    }
}
```

Begrenzung auf das Fenster

- **Problem:** Charaktere können das Spielfenster verlassen.
- **Lösung:** Definieren von Konstanten `WIDTH` und `HEIGHT` in der Hauptklasse.

Code:

```
public class AsteroidsApplication extends Application {

    public static int WIDTH = 300;
    public static int HEIGHT = 200;

    @Override
    public void start(Stage stage) throws Exception {
        Pane pane = new Pane();
        pane.setPrefSize(WIDTH, HEIGHT);

        Ship ship = new Ship(WIDTH / 2, HEIGHT / 2);
        List<Asteroid> asteroids = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            Random rnd = new Random();
            Asteroid asteroid = new Asteroid(rnd.nextInt(WIDTH / 3), rnd.nextInt(HEIGHT));
            asteroids.add(asteroid);
        }

        pane.getChildren().add(ship.getCharacter());
        asteroids.forEach(asteroid -> pane.getChildren().add(asteroid.getCharacter()));
    }
}
```

Erklärung:

- Die Konstanten `WIDTH` und `HEIGHT` sind `static`.
- Sie können von anderen Klassen verwendet werden.

Implementierung der Begrenzung

- **Ziel:** Charaktere bleiben im Fenster.
- **Ansatz:** Anpassung der Methode `move` in der Klasse `Character`.

Code:

```
public void move() {
    this.character.setTranslateX(this.character.getTranslateX() + this.movement.getX());
    this.character.setTranslateY(this.character.getTranslateY() + this.movement.getY());

    if (this.character.getTranslateX() < 0) {
        this.character.setTranslateX(this.character.getTranslateX() + AsteroidsApplication.WIDTH);
    }

    if (this.character.getTranslateX() > AsteroidsApplication.WIDTH) {
        this.character.setTranslateX(this.character.getTranslateX() % AsteroidsApplication.WIDTH);
    }

    if (this.character.getTranslateY() < 0) {
        this.character.setTranslateY(this.character.getTranslateY() + AsteroidsApplication.HEIGHT);
    }

    if (this.character.getTranslateY() > AsteroidsApplication.HEIGHT) {
        this.character.setTranslateY(this.character.getTranslateY() % AsteroidsApplication.HEIGHT);
    }
}
```

Ergebnis:

- Charaktere bleiben im Spielfenster.

Animation:
 Die Asteroiden bleiben im Fenster.

Projektile: Einführung

- Spiel ohne Projektile wäre nur ein Ausweichspiel.
- Projektile benötigen:
  - Form
  - Richtung
  - Bewegung

```
import javafx.scene.shape.Polygon;

public class Projectile extends Character {

    public Projectile(int x, int y) {
        super(new Polygon(2, -2, 2, 2, -2, 2, -2, -2), x, y);
    }
}
```

Hinzufügen von Projektilen

```
List<Projectile> projectiles = new ArrayList<>();
```

- Projektile werden bei Spielstart nicht angezeigt.
- Liste zur Verwaltung von Projektilen.

```
if (pressedKeys.getOrDefault(KeyCode.SPACE, false)) {
    Projectile projectile = new Projectile((int) ship.getCharacter().getTranslateX(), (int) ship.getCharacter().getTranslateY());
    projectile.getCharacter().setRotate(ship.getCharacter().getRotate());
    projectiles.add(projectile);

    pane.getChildren().add(projectile.getCharacter());
}
```

Animation:

Bewegung der Projektile

- `move`-Methode von `Character` zugänglich machen.
- Geschwindigkeit eines Projektils anpassen:

```
projectile.accelerate();
projectile.setMovement(projectile.getMovement().normalize().multiply(3));
```

- Bewegung in die allgemeine Bewegungsfunktion integrieren:

```
ship.move();
asteroids.forEach(asteroid -> asteroid.move());
projectiles.forEach(projectile -> projectile.move());
```

Begrenzung der Projektile

- Maximal drei Projektile:

```
if (pressedKeys.getOrDefault(KeyCode.SPACE, false) && projectiles.size() < 3) {
    Projectile projectile = new Projectile((int) ship.getCharacter().getTranslateX(), (int) ship.getCharacter().getTranslateY());
    projectile.getCharacter().setRotate(ship.getCharacter().getRotate());
    projectiles.add(projectile);

    projectile.accelerate();
    projectile.setMovement(projectile.getMovement().normalize().multiply(3));

    pane.getChildren().add(projectile.getCharacter());
}
```

Kollisionen mit Asteroiden

- Kollisionserkennung:

```
projectiles.forEach(projectile -> {
    List<Asteroid> collisions = asteroids.stream()
                                         .filter(asteroid -> asteroid.collide(projectile))
                                         .collect(Collectors.toList());

    collisions.stream().forEach(collided -> {
        asteroids.remove(collided);
        pane.getChildren().remove(collided.getCharacter());
    });
});
```

 Das Projektil zerstört den Asteroiden.

Projektile entfernen

- **Problem:** Projektile verschwinden nicht nach einer Kollision.
- **Ansatz:** Sammeln und entfernen getroffener Projektile.

Code:

```
List<Projectile> projectilesToRemove = projectiles.stream().filter(projectile -> {
    List<Asteroid> collisions = asteroids.stream()
                                         .filter(asteroid -> asteroid.collide(projectile))
                                         .collect(Collectors.toList());

    if(collisions.isEmpty()) {
        return false;
    }

    collisions.stream().forEach(collided -> {
        asteroids.remove(collided);
        pane.getChildren().remove(collided.getCharacter());
    });

    return true;
}).collect(Collectors.toList());

projectilesToRemove.forEach(projectile -> {
    pane.getChildren().remove(projectile.getCharacter());
    projectiles.remove(projectile);
});
```

Ergebnis:

- Projektile verschwinden nach einer Kollision.

Verbesserung: "alive"-Attribut

- **Ziel:** Übersichtlichere Verwaltung von Charakteren.
- **Ansatz:** Einführen eines „alive“-Attributes.

Code:

```
projectiles.forEach(projectile -> {
    asteroids.forEach(asteroid -> {
        if(projectile.collide(asteroid)) {
            projectile.setAlive(false);
            asteroid.setAlive(false);
        }
    });
});

projectiles.stream()
    .filter(projectile -> !projectile.isAlive())
    .forEach(projectile -> pane.getChildren().remove(projectile.getCharacter()));
projectiles.removeAll(projectiles.stream()
                        .filter(projectile -> !projectile.isAlive())
                        .collect(Collectors.toList()));

asteroids.stream()
        .filter(asteroid -> !asteroid.isAlive())
        .forEach(asteroid -> pane.getChildren().remove(asteroid.getCharacter()));
asteroids.removeAll(asteroids.stream()
                            .filter(asteroid -> !asteroid.isAlive())
                            .collect(Collectors.toList()));
```

Animation:

 Das Projektil verschwindet.

Punkte hinzufügen

- **Ziel:** Einführung eines Punktesystems.
- **Darstellung:** Verwendung eines Textobjekts.

Code:

```
@Override
public void start(Stage stage) throws Exception {
    Pane pane = new Pane();
    Text text = new Text(10, 20, "Points: 0");
    pane.getChildren().add(text);

    Scene scene = new Scene(pane);
    stage.setTitle("Asteroids!");
    stage.setScene(scene);
    stage.show();
}
```

Anzeige:

 Ein Fenster mit der Anzeige 'Points: 0'.

Punkte erhöhen

- **Problem:** Punkte müssen aktualisiert werden.
- **Lösung:** Verwendung von `AtomicInteger`.

Code:

```
@Override
public void start(Stage stage) throws Exception {
    Pane pane = new Pane();
    Text text = new Text(10, 20, "Points: 0");
    pane.getChildren().add(text);

    AtomicInteger points = new AtomicInteger();

    Scene scene = new Scene(pane);
    stage.setTitle("Asteroids!");
    stage.setScene(scene);
    stage.show();

    new AnimationTimer() {

        @Override
        public void handle(long now) {
            text.setText("Points: " + points.incrementAndGet());
        }
    }.start();
}
```

Animation:


Ein Fenster, in dem die Punkteanzeige ständig steigt.

Punkte beim Treffen eines Asteroiden

- **Ziel:** Punkte steigen, wenn ein Projektil einen Asteroiden trifft.
- **Ansatz:** Aktualisieren der Punkte innerhalb der Kollisionserkennung.

Code:

```
projectiles.forEach(projectile -> {
    asteroids.forEach(asteroid -> {
        if(projectile.collide(asteroid)) {
            projectile.setAlive(false);
            asteroid.setAlive(false);
        }
    });

    if(!projectile.isAlive()) {
        text.setText("Points: " + points.addAndGet(1000));
    }
});
```

Animation:

 Wie ein Profi.

Ständiges Hinzufügen von Asteroiden

- **Problem:** Asteroiden gehen aus.
- **Lösung:** Hinzufügen neuer Asteroiden während des Spiels.

Ansatz:

- Wahrscheinlichkeitsbasierter Asteroiden-Spawn.
- Vermeidung von Kollisionen beim Hinzufügen neuer Asteroiden.

Code:

```
if(Math.random() < 0.005) {
    Asteroid asteroid = new Asteroid(WIDTH, HEIGHT);
    if(!asteroid.collide(ship)) {
        asteroids.add(asteroid);
        pane.getChildren().add(asteroid.getCharacter());
    }
}
```

Animation:

 Wie ein Profi.

Zusammenfassung: Verbindung zu Programmierprinzipien

Modularisierung und Abstraktion

- Einführung von Klassen wie `Character`, `Ship`, und `Asteroid` zur klaren Trennung von Verantwortlichkeiten.
- Abstrakte Klassen fördern Wiederverwendbarkeit und Erweiterbarkeit.

Kapselung

- Interne Zustände wie `alive` werden durch Methoden geschützt.
- Kontrollierter Zugriff und Modifikation von Zuständen erhöhen die Robustheit.

Ereignisgesteuerte Programmierung

- Verwendung von Tastatur- und Timer-Ereignissen.
- Dynamische Reaktionen auf Benutzeraktionen illustrieren interaktive Programmkonzepte.

Zusammenfassung: Verbindung zu Programmierprinzipien (Fortsetzung)

Arbeiten mit Datenstrukturen

- Nutzung von Listen (`List`) zur Verwaltung mehrerer Objekte wie Asteroiden und Projektile.
- Streams und Filteroperationen optimieren Datenmanipulation.

Zustandsmanagement

- Objekte behalten eigene Zustände wie Position und Bewegung.
- Methoden zur Änderung dieser Zustände sorgen für klare Verantwortlichkeiten.

Performance und Effizienz

- Verwendung von `AnimationTimer` für zeitkritische Updates.
- Entfernen nicht mehr benötigter Objekte zur Ressourcenoptimierung.

Zusammenfassung: Verbindung zu Programmierprinzipien (Fortsetzung)

Refactoring

- Verbesserung der Kollisionsprüfung durch Einführung eines `alive`-Attributes.
- Refactoring erhöht Lesbarkeit, Wartbarkeit und Erweiterbarkeit des Codes.

Verantwortlichkeiten und Zuständigkeiten

- Jede Klasse übernimmt klar definierte Aufgaben.
- Beispiel: `Ship` verwaltet die Bewegung und Rotation des Raumschiffs.

Asteroids: Fokus auf MVC-Architektur

Übersicht

- Implementierung eines Asteroids-Klons mit **JavaFX**.
- Verwendung des **Model-View-Controller (MVC)-Prinzips**.
- Klare Trennung zwischen **Daten, UI und Logik**.

➡️ Das Ziel ist eine **strukturierte und erweiterbare Architektur** für das Spiel.

Grundstruktur der Anwendung

`AsteroidsApplication` (Einstiegspunkt)

```java
public class AsteroidsApplication extends Application {
    public static final int WIDTH = 800;
    public static final int HEIGHT = 600;
    private GameController controller;

    @Override
    public void start(Stage stage) {
        GameView view = new GameView();
        GameModel model = new GameModel();
        controller = new GameController(view, model);
        model.setController(controller); // ... nrrrrgs
        controller.startGameLoop();

        stage.setTitle("Asteroids!");
        stage.setScene(view.getScene());
        stage.show();
    }

    public static void main(String[] args) {
        launch(AsteroidsApplication.class);
    }
}
```

- Initialisiert **Model, View und Controller**.
- **MVC-Prinzip:** Das `GameModel` kennt den Controller, aber nicht die View.
- **Logik entkoppelt von der UI**, was spätere Änderungen erleichtert.

Das Model (`GameModel`): Spielzustand & Logik

```
public class GameModel {
    private final List<Asteroid> asteroids = new ArrayList<>();
    private final List<Projectile> projectiles = new ArrayList<>();
    private final Ship ship;
    private final AtomicInteger points = new AtomicInteger();
    private boolean isGameOver = false;
    private GameController controller;

    public GameModel() {
        ship = new Ship(AsteroidsApplication.WIDTH / 2, AsteroidsApplication.HEIGHT / 2);
        for (int i = 0; i < 15; i++) {
            asteroids.add(new Asteroid(new Random().nextInt(AsteroidsApplication.WIDTH / 3),
                                       new Random().nextInt(AsteroidsApplication.HEIGHT)));
        }
    }

    public void update() {
        if (isGameOver) return;
        ship.move();
        asteroids.forEach(Asteroid::move);
        projectiles.forEach(Projectile::move);
        checkCollisions();
    }
}
```

✅ **Trennung der Spiellogik von der UI.**

✅ Enthält **Spielzustand**: Schiff, Asteroiden, Projektile.

✅ **Steuert die Bewegung und Kollisionslogik.**

Die View (`GameView`): Darstellung des Spiels

```
public class GameView {
    private final Pane pane;
    private final Text scoreText;
    private final Scene scene;
    private final Text gameOverText;

    public GameView() {
        pane = new Pane();
        pane.setPrefSize(AsteroidsApplication.WIDTH, AsteroidsApplication.HEIGHT);
        scoreText = new Text(10, 20, "Points: 0");
        pane.getChildren().add(scoreText);
        gameOverText = new Text(AsteroidsApplication.WIDTH / 2 - 50, AsteroidsApplication.HEIGHT / 2, "Game Over!");
        gameOverText.setVisible(false);
        pane.getChildren().add(gameOverText);
        scene = new Scene(pane);
    }

    public void addGameObject(Character obj) {
        pane.getChildren().add(obj.getCharacter());
    }
}
```

✅ **Enthält nur UI-Elemente.**

✅ Aktualisiert Punktestand, zeigt `Game Over` an.

✅ Verwaltet die **graphischen Objekte** (Schiff, Asteroiden, Projektile).

Der Controller (`GameController`): Verbindet Model & View

```
public class GameController {
    private final GameView view;
    private final GameModel model;
    private final Map<KeyCode, Boolean> pressedKeys = new HashMap<>();
    private AnimationTimer gameLoop;

    public void startGameLoop() {
        gameLoop = new AnimationTimer() {
            @Override
            public void handle(long now) {
                if (model.isGameOver()) {
                    gameLoop.stop();
                    view.setGameOverVisible(true);
                    return;
                }
                handleInputs();
                model.update();
                updateView();
            }
        };
        gameLoop.start();
    }
}
```

✅ **Steuert das Spiel und verbindet Model & View.**
✅ **AnimationTimer verwaltet den Spielablauf.**
✅ Liest **Tastatureingaben** und aktualisiert das Model.

Kritische Analyse der Architektur

Stärken:

✓ Saubere **Trennung von Model, View und Controller (MVC)**.
✓ Leicht **erweiterbar**: Man kann neue Features ohne große Änderungen hinzufügen.
✓ **Einfache Wartung**: Jede Klasse hat eine **klare Verantwortung**.

Schwächen & Verbesserungsmöglichkeiten:

❌ **Das Model kennt den Controller:**

- **Besser:** Statt `model.setController(controller);` wäre eine **lose Kopplung** besser (Events, Observables).

❌ **Projektile werden direkt in `GameController` erstellt:**

- **Besser:** In `GameModel` verwalten, um klare Trennung beizubehalten.

❌ **Mehrere Methoden greifen direkt auf `pane.getChildren()` zu:**

- **Besser:** Eigene `GameObjectManager`-Klasse für effizientere Verwaltung der UI-Elemente.

Zusammenfassung & Fazit

✅ MVC-Prinzip verbessert die Struktur des Spiels.
✅ Spielobjekte (`Ship`, `Asteroids`, `Projectile`) sind gut getrennt.
✅ `AnimationTimer` sorgt für reibungslosen Ablauf.
❌ Verbesserungen in der Kopplung zwischen Model und Controller sind möglich.

➡️ Insgesamt eine solide Architektur mit Potenzial zur Optimierung!

Wichtigkeit von Drittanbieter-Bibliotheken

- Erweitern den Funktionsumfang einer Sprache, ohne alles selbst zu programmieren
- Vereinfachen komplexe Aufgaben (Parsing, Netzwerk, Datenbank, Kryptografie, etc.)
- Beispiele in vielen Sprachen:
  - **C++**: Boost (strenge Qualitätsanforderungen; oft Wegbereiter für C++-Standards)
  - **Java**: Apache Commons, Google Guava, Spring Framework, usw.
- Drittanbieter-Bibliotheken fördern Best Practices und Community-Wissen

C++ Boost und Standards

- **Boost**-Bibliotheken sind oft sehr leistungsfähig und qualitativ hochwertig
- Viele C++-Features (z.B. Smart Pointers) stammen ursprünglich aus Boost
- Wechselwirkung mit Hardware-Design:
  - Sprach-Features können Hardware-Optimierungen inspirieren
  - Hardware-Anforderungen beeinflussen Sprachweiterentwicklung
- Balance zwischen **Effizienz** (nahe an der Hardware) und **Flexibilität** (Abstraktion und Wiederverwendung)

Flexibilität vs. Effizienz

- **Drittanbieter-Bibliotheken**:
  - Fokus auf einfache Integration und Wiederverwendbarkeit
  - Entwickelt für unterschiedliche Anwendungsszenarien
- **Hardware-Design**:
  - Fokus auf Performance und Energieeffizienz
  - Abstraktionen werden geringer gehalten
- **Ziel**: Geeignete Bibliothek / Technologie finden, die Performance und Entwicklungsaufwand ausbalanciert

Typische Anwendungsfelder von Bibliotheken

- **Datenverarbeitung** (z.B. JSON/XML-Parsen, Serialisierung)
- **Web & Netzwerk** (HTTP-Clients, Server-Frameworks)
- **Datenbanken** (ORM, Migrationstools)
- **Testing** (Unit-Tests, Mocking, Integrationstests)
- **Security** (Kryptografie, Authentifizierung)
- **GUI und Spiele** (JavaFX, LibGDX, FXGL)
- **Nebenläufigkeit und Reaktivität** (RxJava, Project Reactor)

Exemplarischer Java-Code (JSON mit Jackson)

```
<!-- pom.xml -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.13.3</version>
</dependency>
```

```
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonExample {
    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        String json = "{\"name\":\"Alice\",\"age\":30}";

        Person person = mapper.readValue(json, Person.class);
        System.out.println("Name: " + person.getName());
    }
}
```

Use Case:

- Einfache Konvertierung zwischen Java-Objekten und JSON

Ausgewählte Bibliothek: Apache Commons

```xml
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.12.0</version>
</dependency>
```

```java
import org.apache.commons.lang3.StringUtils;

public class CommonsExample {
    public static void main(String[] args) {
        String input = " ";
        if (StringUtils.isBlank(input)) {
            System.out.println("Input is blank!");
        }
    }
}
```

Use Case:

- Utilities für Strings, Collections und mehr
- Vereinfachung wiederkehrender Standardaufgaben (z.B. String-Prüfungen)

Ausgewählte Bibliothek: Google Guava

```xml
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>31.1-jre</version>
</dependency>
```

```java
import com.google.common.collect.ImmutableList;

public class GuavaExample {
    public static void main(String[] args) {
        ImmutableList<String> names = ImmutableList.of("Alice", "Bob");
        names.forEach(System.out::println);
    }
}
```

Use Case:

- Erweiterte Collections, Caching, Funktionsprogrammierung
- Garantierte Immutability und hilfreiche Helper-Methoden

Ausgewählte Bibliothek: JUnit

```xml
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.8.2</version>
  <scope>test</scope>
</dependency>
```

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class MyTest {
    @Test
    void additionTest() {
        assertEquals(4, 2 + 2);
    }
}
```

Use Case:

- Unit- und Integrationstests für Java
- Defacto-Standard für Testautomatisierung

Ausgewählte Bibliothek: Hibernate

```xml
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.14.Final</version>
</dependency>
```

```java
@Entity
public class Person {
    @Id @GeneratedValue
    private Long id;
    private String name;
    // ...
}
```

Use Case:

- ORM (Object-Relational Mapping)
- Verknüpft Java-Objekte mit relationalen Datenbanken
- Vereinfacht CRUD-Operationen

Hibernate mit H2 (Erweitertes Beispiel)

- Nutzung einer **In-Memory-Datenbank** (H2) für Testzwecke
- **Vollständiges ORM-Beispiel** mit Hibernate:
  - Maven-Abhängigkeiten
  - Hibernate-Konfiguration (`hibernate.cfg.xml`)
  - Java-Entity
  - Hauptklasse mit CRUD-Operationen

Projektkonfiguration (pom.xml)

```xml
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.14.Final</version>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.1.214</version>
  </dependency>

  <dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
  </dependency>
</dependencies>
```

- **hibernate-core**: Hauptbibliothek
- **h2**: In-Memory-Datenbank
- **javax.persistence**: JPA-API, die Hibernate nutzt

Hibernate-Konfiguration (hibernate.cfg.xml)

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>

    <!-- H2-Inmemory-Datenbank -->
    <property name="hibernate.connection.driver_class">org.h2.Driver</property>
    <property name="hibernate.connection.url">jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>

    <!-- Dialect für H2 -->
    <property name="hibernate.dialect">org.hibernate.dialect.H2Dialect</property>

    <!-- Tabellen bei jedem Start neu erstellen -->
    <property name="hibernate.hbm2ddl.auto">create</property>
    <property name="show_sql">true</property>

    <!-- Zu mappende Klasse -->
    <mapping class="com.example.Person"/>
  </session-factory>
</hibernate-configuration>
```

Entität: Person.java

```java
package com.example;

import javax.persistence.*;

@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column
    private String name;

    public Person() {}

    public Person(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

- `@Entity`, `@Table`: Klassendefinition für DB-Tabelle
- `@Id`, `@GeneratedValue`: Primärschlüssel-Definition

Main.java

```java
package com.example;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration()
                .configure() // Lädt hibernate.cfg.xml
                .buildSessionFactory();

        try (Session session = sessionFactory.openSession()) {
            Transaction tx = session.beginTransaction();

            // Create
            Person alice = new Person("Alice");
            session.save(alice);

            Person bob = new Person("Bob");
            session.save(bob);

            // Read
            List<Person> personList =
                session.createQuery("from Person", Person.class).list();

            for (Person p : personList) {
                System.out.println("ID: " + p.getId()
                                   + ", Name: " + p.getName());
            }

            tx.commit();
        }

        sessionFactory.close();
    }
}
```

Erklärung

- **SessionFactory**: Baut Verbindung zur DB auf und verwaltet Sessions.
- **Session**: Repräsentiert eine Einheit des Datenbankzugriffs.
- **Transaction tx**: Bündelt SQL-Befehle (Insert, Update, Delete).
- **CRUD**:
  - *Create*: `session.save(...)`
  - *Read*: `session.createQuery(...)`
  - *Update*: z. B. `session.update(...)` (nicht im Beispiel)
  - *Delete*: z. B. `session.delete(...)` (nicht im Beispiel)

Ergebnis (Konsole)

- Generierte SQL-Statements:

```
Hibernate:
    create table person (
       id bigint generated by default as identity,
       name varchar(255),
       primary key (id)
    )
Hibernate:
    insert into person (name) values (?)
Hibernate:
    insert into person (name) values (?)
Hibernate:
    select person0_.id as id1_0_, person0_.name as name2_0_ ...
```

- Konsolenausgabe:

```
ID: 1, Name: Alice
ID: 2, Name: Bob
```

- **Hibernate** + **H2** = ideales Setup für einfache Test- oder Lernumgebungen
- Automatisches Mapping von Java-Objekten auf Datenbanktabellen
- Transaktionen gewährleisten konsistente und gebündelte DB-Zugriffe
- Die Kernideen gelten auch für Produktionsdatenbanken (z. B. MySQL, PostgreSQL) – nur URL & Dialect anpassen

Ausgewählte Bibliothek: Spring

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.7.5</version>
</dependency>
```

```java
@RestController
public class MyController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

Use Case:

- Entwicklung von Web- und Enterprise-Anwendungen
- Konfigurationsarm, bietet Inversion of Control & Dependency Injection

Ausgewählte Bibliothek: OkHttp (Client)

```xml
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
  <version>4.10.0</version>
</dependency>
```

```java
import okhttp3.*;

public class OkHttpExample {
    public static void main(String[] args) throws Exception {
        OkHttpClient client = new OkHttpClient();
        Request request = new Request.Builder()
            .url("https://example.com")
            .build();

        try (Response response = client.newCall(request).execute()) {
            System.out.println(response.body().string());
        }
    }
}
```

Use Case:

- Moderner und effizienter HTTP-Client
- Ideal für REST-Services und Web-APIs

Ausgewählte Bibliothek: LibGDX

```xml
<dependency>
  <groupId>com.badlogicgames.gdx</groupId>
  <artifactId>gdx</artifactId>
  <version>1.11.0</version>
</dependency>
```

```java
public class MyGame extends ApplicationAdapter {
   SpriteBatch batch;
   Texture img;

   @Override
   public void create() {
      batch = new SpriteBatch();
      img = new Texture("badlogic.jpg");
   }

   @Override
   public void render() {
      Gdx.gl.glClearColor(1, 0, 0, 1);
      Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
      batch.begin();
      batch.draw(img, 0, 0);
      batch.end();
   }
}
```

Use Case:

- 2D- und 3D-Spieleentwicklung
- Cross-Plattform (Desktop, Android, iOS, HTML5)

Ausgewählte Bibliothek: FXGL

```
<dependency>
  <groupId>com.github.almasb</groupId>
  <artifactId>fxgl</artifactId>
  <version>11.17</version>
</dependency>
```

```
public class MyGameApp extends GameApplication {
    @Override
    protected void initSettings(GameSettings settings) {
        settings.setWidth(800);
        settings.setHeight(600);
        settings.setTitle("My FXGL Game");
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Use Case:

- JavaFX-basierte 2D-Game Engine
- Einfache Erstellung von Spielen mit UI-Elementen und Effekten

Eigenschaft	libGDX	FXGL
Zielplattformen	**PC, Android, iOS, Web (HTML5), macOS, Linux**	**PC (Windows, macOS, Linux) – basiert auf JavaFX**
Rendering-Technologie	OpenGL (LWJGL), Vulkan (teilweise)	JavaFX Canvas & Scene Graph
2D / 3D Unterstützung	2D & 3D	Hauptsächlich 2D, begrenztes 3D
Geeignet für...	Professionelle, plattformübergreifende Spieleentwicklung	Schnelle Java-Spielentwicklung mit JavaFX
Programmierparadigma	Low-Level, hoher Grad an Kontrolle	High-Level, schnelle Prototypenerstellung
Eingebaute Physik-Engine	Box2D, Bullet (3D)	Ja, basiert auf JavaFX
Asset-Handling	Texturen, Sounds, Animationen, Partikeleffekte	Einfacher Asset-Manager für JavaFX
Benutzeroberflächen (UI)	Scene2D (einfache UI-Komponenten)	Vollständige JavaFX-Integration
Netzwerkunterstützung	Manuell implementierbar	Integrierte Multiplayer-Unterstützung
Community & Support	Sehr groß, professionell genutzt (Minecraft Bedrock basiert teilweise auf libGDX)	Kleiner, aber aktiv gepflegt
Lernkurve	Mittel bis hoch (eher für erfahrene Entwickler)	Einfach (gut für Anfänger)

Ausgewählte Bibliothek: Project Reactor

```xml
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
  <version>3.4.22</version>
</dependency>
```

```java
import reactor.core.publisher.Flux;

public class ReactorExample {
    public static void main(String[] args) {
        Flux.just("Alice", "Bob", "Charlie")
            .filter(name -> name.startsWith("A"))
            .subscribe(System.out::println);
    }
}
```

Use Case:

- Reaktive Programmierung
- Asynchrone und non-blocking Datenströme

Unterschied: Blockierend vs. Nicht-Blockierend

Java Stream vs. Reactor Flux

- Java `Stream` ist **synchron und blockierend**.
- **Reactor** `Flux` ist **asynchron und nicht-blockierend**.
- `Flux` wird häufig in **reaktiven Anwendungen** eingesetzt, z. B. für **WebSockets, Microservices, Daten-Streaming**.

Java Stream (Blockierend)

```
import java.util.stream.Stream;

public class StreamExample {
    public static void main(String[] args) {
        System.out.println("Start Stream Verarbeitung");

        Stream.of("Alice", "Bob", "Charlie")
            .peek(name -> sleep(1000))  // Verzögerung simulieren
            .forEach(name -> System.out.println("Stream gibt aus: " + name));

        System.out.println("Ende Stream Verarbeitung");
    }

    private static void sleep(int millis) {
        try { Thread.sleep(millis); } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

Erwartete Ausgabe (Stream)

```
Start Stream Verarbeitung
(Stream schläft 1 Sekunde)
Stream gibt aus: Alice
(Stream schläft 1 weitere Sekunde)
Stream gibt aus: Bob
(Stream schläft 1 weitere Sekunde)
Stream gibt aus: Charlie
Ende Stream Verarbeitung
```

✅ Alles läuft sequenziell im `main`-Thread.
✅ Nächste Zeile wird erst nach kompletter Verarbeitung ausgeführt.
❌ Blockiert den gesamten Ablauf.

Reactor Flux (Nicht-Blockierend)

```java
import reactor.core.publisher.Flux;
import java.time.Duration;

public class FluxExample {
    public static void main(String[] args) {
        System.out.println("Start Flux Verarbeitung");

        Flux.just("Alice", "Bob", "Charlie")
            .delayElements(Duration.ofSeconds(1)) // Asynchron verzögern
            .subscribe(name -> System.out.println("Flux gibt aus: " + name));

        System.out.println("Ende Flux Verarbeitung");

        try {
            Thread.sleep(4000);  // Warten, sonst würde das Programm vorher enden
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Erwartete Ausgabe (Flux)

```
Start Flux Verarbeitung
Ende Flux Verarbeitung  <-- ⚠️ Erscheint sofort!
(1 Sekunde später) Flux gibt aus: Alice
(2 Sekunden später) Flux gibt aus: Bob
(3 Sekunden später) Flux gibt aus: Charlie
```

✅ Die Hauptmethode läuft sofort weiter, während `Flux` Elemente verzögert ausgibt.

✅ Die Verarbeitung ist asynchron, `main` wird nicht blockiert.

Vergleich: Java Stream vs. Reactor Flux

Eigenschaft	Java Stream (`Stream<T>`)	Reactor Flux (`Flux<T>`)
Verarbeitung	Synchron, blockierend	Asynchron, nicht-blockierend
Wann erscheint `"Ende Verarbeitung"`?	Nach allen Elementen	Sofort, bevor die Daten erscheinen!
Thread-Nutzung	Einzelner Thread (`main`)	Kann andere Threads nutzen (reaktiv)
Latenzsteuerung	❌ Nein	✅ `delayElements(Duration.ofMillis(x))`

Anwendungsfälle für `Flux`

✅ Daten-Streaming (z. B. WebSockets, Kafka, RxJava)
✅ Reaktive Microservices (Spring WebFlux, vert.x)
✅ Event-getriebene Systeme
✅ Datenbank-Abfragen mit Rückgabe mehrerer Ergebnisse (R2DBC, MongoDB)

Wenn synchrone Verarbeitung nicht ausreicht, ist `Flux` die bessere Wahl!

Ausgewählte Bibliothek: RxJava

```xml
<dependency>
  <groupId>io.reactivex.rxjava3</groupId>
  <artifactId>rxjava</artifactId>
  <version>3.1.5</version>
</dependency>
```

```java
import io.reactivex.rxjava3.core.Observable;

public class RxJavaExample {
    public static void main(String[] args) {
        Observable.just("Alice", "Bob", "Charlie")
                  .filter(name -> name.startsWith("C"))
                  .subscribe(System.out::println);
    }
}
```

Use Case:

- Reaktive, eventgetriebene Architektur
- Deklaratives Datenflussmodell

Anwendungsfall	Reactor `Flux`	RxJava `Observable`
Spring WebFlux / Spring Boot	✅ Ja, perfekt integriert	❌ Nein, kein direkter Support
Reaktive Microservices	✅ Ja (Spring Cloud)	✅ Ja
Einfache reaktive Streams (Standalone-Java)	✅ Ja	✅ Ja
Datenbank-Reaktive Verarbeitung (R2DBC)	✅ Ja	❌ Nein
Android-Apps	❌ Nicht empfohlen	✅ Ja (RxJava ist Android-Standard)
Fehlertolerante Streams	✅ Gut integriert	✅ Gut integriert
Backpressure benötigt?	✅ Ja (Standard in `Flux`)	❌ Nur mit `Flowable`

Ausgewählte Bibliothek: Bouncy Castle

```xml
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcprov-jdk15on</artifactId>
  <version>1.70</version>
</dependency>
```

```java
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import javax.crypto.Cipher;
import java.security.Security;

public class BouncyCastleExample {
    public static void main(String[] args) throws Exception {
        Security.addProvider(new BouncyCastleProvider());

        Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "BC");
        // ... Konfiguration und Nutzung
    }
}
```

Use Case:

- Erweiterte Kryptografie-Funktionen
- Verschlüsselung, Signaturen, Zertifikate

.... und ... so ... weiter ...

Fazit

- Drittanbieter-Bibliotheken sind **essentiell**, um Entwicklungsaufwände zu reduzieren und Fokus auf Kernlogik zu legen
- **Boost** in C++ als Beispiel: Bibliotheken beeinflussen oft sogar Sprachstandards
- In **Java** gibt es eine riesige Auswahl, je nach Use Case (Web, Datenbank, Testing, Spiele, etc.)
- Drittanbieter-Bibliotheken sind eine der stärksten Ressourcen für schnelle, robuste und flexible Softwareentwicklung