

Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024/2025

(Teil 13)

GUI-Programmierung mit Java: Einführung

- **Graphische Benutzeroberflächen (GUIs)** : die Schnittstelle zwischen Benutzern und Software.
- Java bietet seit Jahrzehnten Werkzeuge für die Erstellung von GUIs (z. B. Swing, JavaFX).
- **Warum lernen wir GUIs mit Java?**
 - GUIs sind ein praktischer Einstieg, um **programmatische Prinzipien** zu verstehen.
 - Sie ermöglichen das Lernen von Konzepten wie **Event-Handling, Modularisierung** und **MVC-Architekturen**.

Prinzipien statt Technologien

- **Technologien verändern sich**, aber grundlegende Prinzipien bleiben, z.B.:
 - Event-Handling zeigt, wie Software auf **Ereignisse** reagiert.
 - MVC (Model-View-Controller) trennt **Daten, Darstellung** und **Logik**.
 - Frameworks lehren uns, **Client-Server-Kommunikation** zu strukturieren.
- **Java GUIs als Beispiel:**
 - Sie sind eine Grundlage, um komplexere Softwarearchitekturen zu verstehen.
 - Konzepte, die hier gelernt werden, sind auf andere Plattformen übertragbar.

Herausforderung: Modernität

- **GUIs mit Java** gelten oft als "veraltet".
 - Swing: Seit den 1990er Jahren etabliert, aber nicht mehr "modern".
 - JavaFX: Moderner, aber von anderen Technologien wie Web-Frameworks überholt.
- **Warum trotzdem Java für GUIs lernen?**
 - Fokus auf **universelle Konzepte**, nicht auf die "neueste" Technologie.
 - Grundlegende Prinzipien sind mit modernen Frameworks wie React, Vue.js oder Flutter vergleichbar.

Prinzipien: Event-Handling

- **Was ist Event-Handling?**
 - Ein Mechanismus, der Ereignisse wie **Klicks** oder **Tastendrücke** erkennt und darauf reagiert.
- **Warum wichtig?**
 - Grundlage für jede interaktive Software.
- **Beispiele:**
 - **Java:** `addActionListener()` in Swing.
 - **Moderne Frameworks:** React mit `onClick` oder JavaScript `addEventListener()`.

Prinzipien: Client-Server-Modelle

- **Server-seitige GUIs:**
 - Rendern der GUI erfolgt auf dem Server (z. B. JSP, JSF).
- **Client-seitige Frameworks:**
 - Rendering und Logik laufen im Browser oder auf dem Client (z. B. React, Angular).
- **Java GUIs als Einstieg:**
 - **Standalone-Anwendungen** zeigen, wie Logik und Darstellung auf einem Gerät laufen.
 - Erweiterung: Interaktion mit einem Server (z. B. REST-APIs).

Positives Lernziel

- **Was Sie mitnehmen:**
 - Ein Verständnis für **zeitlose Prinzipien** in der Softwareentwicklung.
 - Übertragbare Fähigkeiten, die auch für moderne Technologien relevant sind.
- **Warum Java?**
 - Einfache Einstiegsmöglichkeiten.
 - Breite Unterstützung und viele vorhandene Beispiele.
- **Ein Blick in die Zukunft:**
 - Die gelernten Prinzipien können auf moderne Frameworks angewendet werden.
 - Java GUIs bieten einen sicheren, didaktischen Einstieg.

- **GUIs mit Java:**

- Sind ein Werkzeug, um **zeitlose Prinzipien** zu lernen.
- Helfen, die Grundlage für moderne Softwareentwicklung zu legen.

- **Prinzipien vor Technologien:**

- Technologien ändern sich, aber Konzepte wie Event-Handling, Modularisierung und Client-Server-Architekturen bleiben bestehen.

- **Vorteil:**

- Lernen Sie Java GUIs, um die Mechanismen moderner Anwendungen zu verstehen.

Honestly, diese Einführungsfolien habe ich geschrieben, um mich selbst zu motivieren, weil ich die Voreinstellung von JavaFX an der Grenze dessen (oder auf der falschen Seite hinter der Grenze) sehe, was an einer Universität gelehrt werden sollte...

Grafische Benutzungsoberflächen (GUIs)

- **Ziel:** Programme um **grafische Benutzungsoberflächen** erweitern
- **Vorteil:** intuitive Interaktion mit Buttons, Textfeldern etc.
- **Bibliotheken:** z. B. **JavaFX**, aber auch weitere Möglichkeiten:
 - **Swing** (älter, in Java integriert)
 - **AWT** (ganz alt, Teil des JDKs, nur rudimentäre Funktionen)
 - Andere plattformabhängige Bibliotheken oder **Frameworks** (SWT, Qt via JNI, etc.)

Einleitung

- **Ziel:** Verständnis für Desktop- und Web-Frameworks entwickeln.
- **Vergleich von Technologien:**
 - i. Desktop-orientierte Frameworks (AWT, Swing, SWT, JavaFX, Qt).
 - ii. Web-orientierte Frameworks (Vaadin, React, Jetpack Compose).
- **Schwerpunkt:**
 - Paradigmen: **Imperativ vs. Deklarativ.**
 - **Code-Beispiele**, um Unterschiede zu verdeutlichen.

Vergleich: Desktop- vs. Web-orientierte Frameworks

Desktop-orientierte Frameworks

- Beispiele : **Swing, SWT, JavaFX, Qt**
 - Ziel: Native Desktop-Anwendungen.
 - GUI und Logik eng gekoppelt.
 - Fokus auf **plattformübergreifende Desktop-Apps**.

Web- und deklarative Frameworks

- Beispiele : **Vaadin, React, Jetpack Compose**
 - Ziel: Moderne, oft cloudbasierte Anwendungen.
 - Trennung von Frontend (Benutzeroberfläche) und Backend (Datenverarbeitung).
 - Fokus auf **Webbrowser** oder **deklarative UI-Ansätze**.

Desktop-orientierte Frameworks: Unvollständiger Überblick

- Technologien:
 - i. **AWT**: Grundlegende Widgets, Teil des JDK.
 - ii. **Swing**: Erweiterung von AWT, plattformunabhängig.
 - iii. **SWT**: Native Widgets, Eclipse-Ökosystem.
 - iv. **JavaFX**: Modernes Toolkit mit CSS-Integration.
 - v. **Qt**: Plattformübergreifend, leistungsstark (C++ mit Java-Bindings).

Vergleich: Desktop Frameworks

Technologie	Vorteile	Nachteile
AWT	Native Plattform-Widgets	Veraltet, rudimentär
Swing	Plattformunabhängig, Teil von Java SE	Veraltete Optik
SWT	Native Performance	Eclipse-Ökosystem
JavaFX	CSS-Unterstützung, "modern"	Nicht mehr Teil des JDK
Qt	Plattformübergreifend	Lizenzkosten möglich

Maven-Abhängigkeiten für JavaFX (Teaser)

Schritt 1: Abhängigkeit in `pom.xml` hinzufügen

```
<dependencies>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>17</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>17</version>
  </dependency>
</dependencies>
```

Schritt 2: JavaFX-Laufzeitoptionen

- JVM-Argumente (z. B. in der IDE einstellen):

```
--module-path /pfad/zu/javafx --add-modules javafx.controls,javafx.fxml
```

Maven-Abhängigkeiten für SWT

Schritt 1: Abhängigkeit in `pom.xml` hinzufügen

```
<dependencies>
  <dependency>
    <groupId>org.eclipse.swt</groupId>
    <artifactId>org.eclipse.swt.win32.win32.x86_64</artifactId>
    <version>4.24</version>
  </dependency>
</dependencies>
```

Hinweis:

- Die spezifische SWT-Bibliothek hängt von der Zielplattform ab (z. B. `win32`, `gtk` für Linux).

Maven-Abhängigkeiten für Vaadin

Schritt 1: Abhängigkeit in `pom.xml` hinzufügen

```
<dependencies>
  <dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin</artifactId>
    <version>24.1.0</version>
  </dependency>
</dependencies>
```

Hinweis:

- Vaadin benötigt keine zusätzliche Konfiguration für den Modulpfad.

Vergleich: AWT, Swing und JavaFX

Merkmal	AWT	Swing	JavaFX
Alter	1995	1997	2008
Abhängigkeit	Native Plattform-Widgets	Plattformunabhängige Komponenten	Eigenes, modernes Toolkit
Rendering	Plattformabhängig	Softwarebasiert	Hardwarebeschleunigt
Optik	Veraltet	Veraltet	Modern, anpassbar

Web-orientierte Frameworks: Unvollständiger Überblick

- Technologien:
 - i. **Vaadin**: Java-basiertes Framework für Web-Apps.
 - ii. **React**: Komponentenbasierte UIs mit JavaScript.
 - iii. **Jetpack Compose**: Moderne, deklarative UI-Entwicklung (Kotlin).

Vergleich: Web Frameworks (völlig unvollständig)

Framework	Vorteile	Nachteile
Vaadin	Einfacher Einstieg	Begrenzte Kontrolle
Jetpack Compose	Modern, deklarativ	Kleinere Desktop-Community
React + Java	Flexibel, weit verbreitet	Frontend-/Backend-Kenntnisse

Einführung in React-Komponenten

- **Komponenten** sind die Bausteine von React-Anwendungen.
- Jede Komponente ist eine **unabhängige Einheit** mit eigener Logik und Darstellung.
- Sie ermöglichen **Wiederverwendbarkeit, Modularität** und **Kapselung**.

Funktionale Komponente: Beispiel

```
function Greeting(props) {  
  return <h1>Hello, {props.name}!</h1>;  
}  
  
// Verwendung  
<Greeting name="Alice" />
```

- **Greeting** ist eine funktionale Komponente.
- Sie nimmt **props** (Eigenschaften) als Eingabe.
- Gibt ein **JSX-Element** zurück.

Klassensbasierte Komponente: Beispiel

```
class Greeting extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}!</h1>;  
  }  
}  
  
// Verwendung  
<Greeting name="Bob" />
```

- **Greeting** ist eine Klassenkomponente.
- Verwendet die Methode **render()** zum Erstellen des UI.
- Zugriff auf **props** über **this.props**.

Komponenten-Hierarchie: Beispiel

```
function App() {
  return (
    <div>
      <Header />
      <MainContent />
      <Footer />
    </div>
  );
}

function Header() {
  return <h1>Welcome to my website</h1>;
}

function MainContent() {
  return <p>This is the main content of the page.</p>;
}

function Footer() {
  return <footer>© 2025 My Website</footer>;
}
```

Integration von React mit Java

Backend mit Spring Boot

Erstelle eine einfache API:

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @GetMapping
    public List<String> getUsers() {
        return List.of("Alice", "Bob", "Charlie");
    }
}
```

- **Spring Boot** stellt RESTful APIs bereit.
- Diese APIs liefern Daten an das React-Frontend.

Integration von React mit Java

Frontend mit React

Nutze die API im React-Frontend:

```
import React, { useEffect, useState } from "react";

function App() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("/api/users")
      .then((response) => response.json())
      .then((data) => setUsers(data));
  }, []);

  return (
    <div>
      <h1>User List</h1>
      <ul>
        {users.map((user) => (
          <li key={user}>{user}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

- **fetch** ruft die Daten vom Java-Backend ab.
- Die API-Antwort wird in der Benutzeroberfläche angezeigt.
- REST Beispiel: <https://jsonplaceholder.typicode.com/users/1>

Vorteile des Komponentenbasierten Ansatzes

- **Wiederverwendbarkeit:**
 - Komponenten können mehrfach genutzt werden.
- **Kapselung:**
 - Jede Komponente verwaltet ihre eigene Logik.
- **Modularität:**
 - Große Anwendungen werden in kleine, übersichtliche Teile zerlegt.
- **Verbesserte Lesbarkeit:**
 - Strukturiertes und wartbarer Code.

Fazit

- React-Komponenten machen UIs **strukturiierter** und **flexibler**.
- Der komponentenbasierte Ansatz erleichtert die **Entwicklung, Wartung** und **Erweiterung** von Anwendungen.
- Die Integration von React und Java ermöglicht die Erstellung moderner, skalierbarer Webanwendungen.

Mit kleinen, gut definierten Komponenten wird die Arbeit an komplexen Anwendungen erheblich vereinfacht.

Zusammenfassung

- **Maven** erleichtert die Integration moderner Frameworks wie JavaFX oder Vaadin.
- **AWT, Swing, JavaFX**: Für Desktop-Anwendungen geeignet, je nach Anforderungen.
- **Vaadin, React, Jetpack Compose**: Für moderne, deklarative Web-UIs.
- Die Wahl des Frameworks hängt von **Projektzielen** und **Teamkompetenzen** ab.

Desktop-orientierte Frameworks (völlig unvollständig)

Technologie	Vorteile	Nachteile
Swing	Plattformunabhängig, Teil von Java SE	Veraltete Optik
SWT	Native Performance	Abhängigkeit von Eclipse
JavaFX	CSS-Unterstützung, (modern)	Nicht mehr Teil des JDK
Qt	Plattformübergreifend, leistungsstark	evtl. Lizenzkosten

Imperative GUIs: Schritt-für-Schritt-Anweisungen

- **Imperativ** bedeutet hier, dass jede Aktion zur Erstellung der GUI explizit programmiert wird.
- **Entwickler*innen kontrollieren jeden Schritt**, wie Elemente hinzugefügt und manipuliert werden.

Beispiel (Swing):

```
JFrame frame = new JFrame("Imperative GUI");  
JButton button = new JButton("Klick mich!");  
frame.add(button);  
frame.setSize(400, 300);  
frame.setVisible(true);
```

- **Nachteil:** Viel Boilerplate-Code, schwerer wartbar.
- **Vorteil:** Volle Kontrolle über jeden Aspekt der GUI.

Deklarative GUIs: Was statt Wie

- **Deklarativ** bedeutet, dass die Struktur und der Zustand der GUI beschrieben werden.
- **Automatische Aktualisierungen:** Änderungen am Zustand passen die GUI an.

Beispiel (Jetpack Compose):

```
@Composable
fun MyApp() {
    var text by remember { mutableStateOf("Hallo!") }
    Column {
        Text(text)
        Button(onClick = { text = "Geklickt!" }) {
            Text("Klick mich!")
        }
    }
}
```

- **Vorteil:** Weniger Boilerplate, wartbarer Code.
- **Nachteil:** Weniger direkte Kontrolle über den Ablauf.

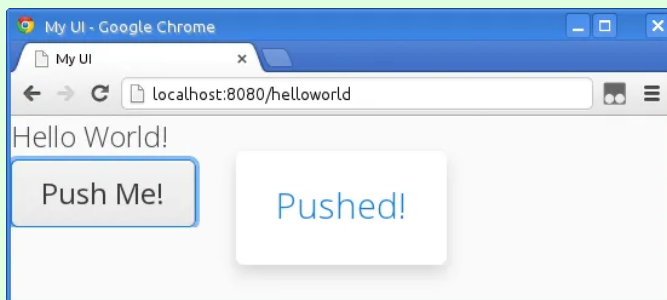
Vaadin Code-Beispiel (Läuft Server-seitig)

```
import com.vaadin.server.VaadinRequest;
import com.vaadin.ui.Label;
import com.vaadin.ui.UI;

@Title("My UI")
public class HelloWorld extends UI {
    @Override
    protected void init(VaadinRequest request) {
        // Create the content root layout for the UI
        VerticalLayout content = new VerticalLayout();
        setContent(content);

        // Display the greeting
        content.addComponent(new Label("Hello World!"));

        // Have a clickable button
        content.addComponent(new Button("Push Me!",
            click -> Notification.show("Pushed!")));
    }
}
```



Jetpack Compose Code-Beispiel

Hinweis:

- **Jetpack Compose ist ausschließlich in Kotlin verfügbar.**
- Es gibt zwei Versionen:
 - i. **Jetpack Compose (für Android):** Für mobile Apps.
 - ii. **Jetpack Compose for Desktop:** Für plattformübergreifende Desktop-Anwendungen.

```
import androidx.compose.desktop.Window
import androidx.compose.foundation.layout.*
import androidx.compose.material.*
import androidx.compose.runtime.*

fun main() = Window {
    var text by remember { mutableStateOf("Hallo, Jetpack Compose!") }

    Column(Modifier.fillMaxSize().padding(16.dp)) {
        Text(text)
        Spacer(Modifier.height(16.dp))
        Button(onClick = { text = "Button geklickt!" }) {
            Text("Klick mich!")
        }
    }
}
```

Unterschiede zwischen Android und Desktop:

- **Desktop:** Nutzt plattformübergreifende Technologien für Windows, macOS und Linux.
- **Android:** Spezifisch für mobile Geräte mit Android-APIs.

React + Java Code-Beispiel

Frontend (React):

```
function App() {
  const [message, setMessage] = React.useState("");

  React.useEffect(() => {
    fetch("/api/hello")
      .then(response => response.text())
      .then(setMessage);
  }, []);

  return <div>{message}</div>;
}

export default App;
```

Backend (Spring Boot):

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
  @GetMapping("/api/hello")
  public String sayHello() {
    return "Hallo, React mit Java-Backend!";
  }
}
```

Gatsby: React als Fundament

- **React** bildet die Basis von Gatsby:
 - Komponentenbasierter Aufbau.
 - JSX-Syntax für die Erstellung von UIs.
- Gatsby erweitert React durch zusätzliche Features:
 - **Static Site Generation (SSG)** für ultraschnelle Ladezeiten.
 - **SEO-Optimierung** durch vorgerendertes HTML.
 - **Plugins** für Datenquellen und erweiterte Funktionalitäten.

Gatsby kombiniert die Flexibilität von React mit Tools, die speziell für statische Seiten optimiert sind.

Gatsby: Statische Webseiten

React vs. Gatsby: Rendering

React	Gatsby
Läuft komplett im Browser.	Rendert Seiten bei der Build-Zeit.
Daten werden dynamisch zur Laufzeit geladen.	Daten werden vorab integriert und gerendert.
Eignet sich für dynamische UIs.	Perfekt für statische Inhalte und SEO.

Beispiel:

Die PdP-Webseiten

Vorteile von Gatsby

- **Performance:** Optimierte Bildverarbeitung, Code-Splitting, Prefetching.
- **Datenintegration:** Plugins für CMS (z. B. WordPress, Contentful) und APIs.
- **Flexibilität:** Volle React-Kompatibilität:
 - Du kannst React-Bibliotheken wie Material-UI oder TailwindCSS verwenden.
- **SEO und Sicherheit:** Kein Server erforderlich, statische Dateien reduzieren Angriffsfläche.

Fazit: Gatsby ist ideal für Blogs, Landing Pages und Dokumentationen, die auf Geschwindigkeit und SEO setzen.

Back to Square 1

Lernziele

- Aufbau und Bestandteile einer **Benutzungsoberfläche** verstehen
- Starten einer **grafischen** Anwendung in Java
- Komponenten (z. B. Buttons) verwenden und anordnen

Aufbau: GUI vs. textbasierte Oberfläche

- **Textbasiert:** Eingaben werden vom Terminal verarbeitet (Scanner, System.in).
- **Grafisch:**
 - Benutzer*innen interagieren über visuelle Komponenten (z. B. Button).
 - Ereignisse (Events) lösen Methoden aus.

Beispiel: Button-Klick ruft eine Methode auf, statt wie beim Textmodus eine Eingabezeile zu interpretieren.

JavaFX – Eine Möglichkeit

```
public class JavaFxApplication extends Application {  
    @Override  
    public void start(Stage window) {  
        window.setTitle("Hei Maailma!");  
        window.show();  
    }  
    public static void main(String[] args) {  
        launch(JavaFxApplication.class);  
    }  
}
```

Vergleich: AWT, Swing und JavaFX

Merkmal	AWT	Swing	JavaFX
Alter	1995	1997	"Modern" (2008)
Abhängigkeit	Native Plattform-Widgets	Plattformunabhängige Komponenten	Eigenes, modernes Toolkit
Funktionalität	Grundlegende GUI-Features	Erweiterte Komponenten	Moderne UI, CSS, FXML
Aussehen	Veraltet	Veraltet	Zeitgemäß, anpassbar
Rendering	Plattformabhängig	Softwarebasiert	Hardwarebeschleunigt
Weiterentwicklung	Abgeschlossen	Wartungsmodus	Aktiv (OpenJFX)

JavaFX ist von diesen die modernste und leistungsstärkste Option, während Swing und AWT für bestehende Projekte relevant bleiben.
(Alternativen: Compose for Desktop, JRebirth, Griffon, ...)

Swing Code-Beispiel

```
import javax.swing.*;

public class SwingExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Swing Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        JButton button = new JButton("Klick mich!");
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Hallo, Swing!"));

        frame.add(button);
        frame.setVisible(true);
    }
}
```

Warum wurde JavaFX aus dem JDK entfernt?

- **Entkopplung von Java SE:**
 - Ziel: Ein schlankeres, modulares JDK.
 - JavaFX war nicht für alle Java-Entwickler*innen relevant.
- **Eigenständige Weiterentwicklung:**
 - Als OpenJFX wird es unabhängig und schneller weiterentwickelt.
 - Beiträge aus der Community (z. B. Gluon).
- **Integration:**
 - Einfaches Einfügen über **Maven** oder **Gradle**.

Fazit: JavaFX ist zwar nicht mehr Teil des JDK, bleibt jedoch eine aktive, relativ moderne Lösung für GUIs.

JavaFX: Maven-Abhängigkeit hinzufügen

Schritt 1: Maven-Projekt erstellen

1. Erstelle ein neues Maven-Projekt in deiner IDE (z. B. IntelliJ IDEA, Eclipse).
2. Stelle sicher, dass Java 11+ verwendet wird.

Schritt 2: Abhängigkeit in `pom.xml` hinzufügen

```
<dependencies>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>17</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>17</version>
  </dependency>
</dependencies>
```

Schritt 3: JavaFX-Laufzeitoptionen

Füge folgende Argumente für die JVM hinzu (z. B. in deiner IDE):

```
--module-path /pfad/zu/javafx --add-modules javafx.controls,javafx.fxml
```

JavaFX: Beispielcode mit Maven

Einfacher Start mit einer `Application`-Klasse:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class JavaFxApp extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Klick mich!");
        button.setOnAction(e -> System.out.println("Hallo, JavaFX!"));

        StackPane root = new StackPane(button);
        Scene scene = new Scene(root, 400, 300);

        primaryStage.setTitle("JavaFX mit Maven");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Mit Maven ist JavaFX leicht in Projekte integrierbar und bleibt flexibel für Anwendungen.

Grundstruktur einer Benutzungsoberfläche (JavaFX)

```
Button button = new Button("Tämä on nappi");  
FlowPane group = new FlowPane();  
group.getChildren().add(button);  
  
Scene scene = new Scene(group);  
window.setScene(scene);  
window.show();
```

Zusammenfassung

- Grafische Benutzungsoberflächen ermöglichen **Interaktion** via Maus, Tastatur, etc.
- **JavaFX** ist nur eine von mehreren Optionen, heutzutage recht modern und flexibel.
- **Layout**-Konzepte: Container in Container, um flexible Anordnungen zu schaffen.
- Auch andere Toolkits sind gängige Alternativen.

Lernziele

- Aufbau und Bestandteile einer **Benutzungsoberfläche** verstehen
- Starten einer **grafischen** Anwendung in Java
- Komponenten (z. B. Buttons) verwenden und anordnen

Aufbau: GUI vs. textbasierte Oberfläche

- **Textbasiert:** Eingaben werden vom Terminal verarbeitet (Scanner, System.in).
- **Grafisch:**
 - Benutzer*innen interagieren über visuelle Komponenten (z. B. Button).
 - Ereignisse (Events) lösen Methoden aus.

Beispiel: Button-Klick ruft eine Methode auf, statt wie beim Textmodus eine Eingabezeile zu interpretieren.

JavaFX

```
public class JavaFxApplication extends Application {  
  
    @Override  
    public void start(Stage window) {  
        Button button = new Button("Knopf");  
  
        FlowPane componentGroup = new FlowPane();  
        componentGroup.getChildren().add(button);  
  
        Scene scene = new Scene(componentGroup);  
  
        window.setScene(scene);  
        window.show();  
    }  
  
    public static void main(String[] args) {  
        launch(JavaFxApplication.class);  
    }  
}
```

- **Stage** = Fenster
- **Scene** = „Szene“, enthält Layout
- **Layout-Container** (hier `FlowPane`)
- **Komponenten** (z. B. `Button`) als Kinder des Layouts, werden in Layout-Containern platziert

So funktioniert das Zusammenspiel

1. **Fenster** (`Stage`)
2. **Szene** (`Scene`) beschreibt Inhalte
3. **Layout-Objekt** (z. B. `FlowPane`) ordnet Komponenten an
4. **Benutzungsoberflächenkomponenten** (z. B. `Button`, `TextField`) werden ins Layout hinzugefügt

Viele Bibliotheken (z. B. `Swing`) folgen einem ähnlichen Prinzip, nur mit anderen Klassennamen.

UI-Komponenten und deren Layout

Lernziele

- Kennenlernen verschiedener UI-Komponenten:
 - **Label** für Text
 - **Button** für Schaltflächen
 - **TextField, TextArea** etc.
- Verstehen, wie man diese zu einer **Benutzungsoberfläche** hinzufügt
- Methoden zur Konfiguration von Komponenten (z. B. `setText`)
- **Layouts:** FlowPane, BorderPane, HBox, VBox, GridPane

Grundidee: Vorgefertigte Komponenten

- Für grafische Benutzungsoberflächen wollen wir nicht jede Schaltfläche „from scratch“ zeichnen
- Stattdessen nutzen wir **Bibliotheken** mit fertigen Komponenten:
 - Buttons, Textfelder, Labels, Listen, ...
- Jede Komponente ist ein eigenes Objekt, das wir unserer Oberfläche hinzufügen

Label: Text anzeigen


```
Label textComponent = new Label("Tekstielementti");  
FlowPane componentGroup = new FlowPane();  
componentGroup.getChildren().add(textComponent);  
  
Scene view = new Scene(componentGroup);  
window.setScene(view);  
window.show();
```

 Window with a textComponent. The window shows the text 'Text element'.

- `Label` zeigt statischen Text an
- Text wird im Konstruktor oder per `setText` festgelegt
- Kann man im Layout (z. B. `FlowPane`) platzieren


Button: Interaktive Schaltfläche

```
Button buttonComponent = new Button("Tämä on nappi");  
FlowPane componentGroup = new FlowPane();  
componentGroup.getChildren().add(buttonComponent);
```

-  Ikkuna, jossa on nappi. Napissa on teksti 'This is a button'.
- Ähnlich wie Label, nur klickbar
 - Üblich: Button löst Event aus (in späteren Teilen relevant)

Mehrere Komponenten


```
Button buttonComponent = new Button("Tämä on nappi");  
Label textComponent = new Label("Tekstielementti");  
  
FlowPane group = new FlowPane();  
group.getChildren().add(buttonComponent);  
group.getChildren().add(textComponent);
```

 Ikkuna, jossa on nappi sekä textComponent. Napissa on teksti 'This is a button' ja textComponent enthält den Text 'Text element'.

- Wir können beliebig viele Komponenten in einen Layout-Container hinzufügen
- Reihenfolge des Hinzufügens bestimmt die Reihenfolge der Anzeige (z. B. in FlowPane)


Layout: FlowPane

- **FlowPane** reiht Komponenten **nebeneinander**, ggf. auch in mehreren Zeilen
- Verkleinert man das Fenster, rutschen Komponenten automatisch in die nächste Zeile

 Window that has a button and a textComponent. The button has the text 'This is a button' and the textComponent contains the text 'Text element'. The window's width is so narrow that the components are placed on separate rows.

Layout: BorderLayout

```
BorderPane layout = new BorderLayout();  
layout.setTop(new Label("top"));  
layout.setRight(new Label("right"));  
layout.setBottom(new Label("bottom"));  
layout.setLeft(new Label("left"));  
layout.setCenter(new Label("center"));
```

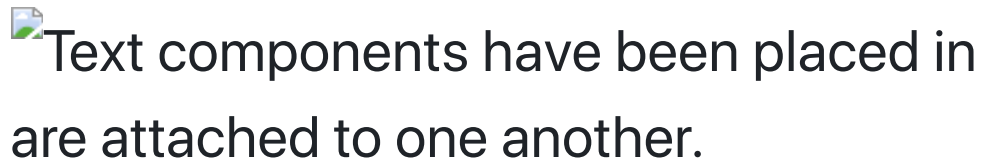
 A user interface using the BorderLayout layout, which contains a textComponent in each primary location

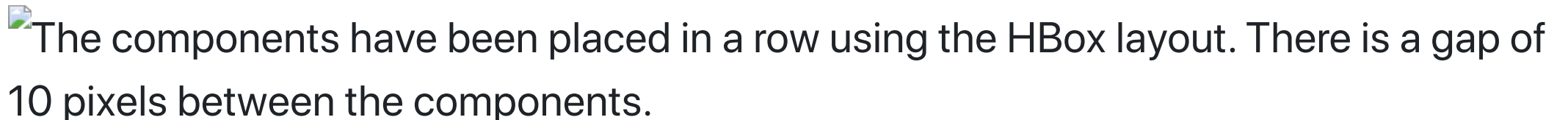
- Anordnung in **fünf Hauptbereiche**: oben, rechts, unten, links, Mitte
- Ideal für typische Programme mit Menüleiste oben, Hauptinhalt in der Mitte etc.

Layout: HBox und VBox

- **HBox:** Horizontale Anordnung

```
HBox layout = new HBox();  
layout.setSpacing(10);  
layout.getChildren().addAll(  
    new Label("Eins"), new Label("Zwei"), new Label("Drei")  
);
```

Text components have been placed in a row using the HBox layout. The components are attached to one another.

The components have been placed in a row using the HBox layout. There is a gap of 10 pixels between the components.

- **VBox:** Vertikale Anordnung

Text components have been placed in a column using the VBox layout.

Layout: GridPane

```
GridPane layout = new GridPane();
for (int x = 1; x <= 3; x++) {
    for (int y = 1; y <= 3; y++) {
        layout.add(new Button(x + "", "" + y), x, y);
    }
}
```

 3 times 3 grid containing 9 buttons.


- Layout als **Raster** (Tabellenstruktur)
- Angabe von Zeilen- und Spaltenindex beim Hinzufügen

Kombinieren von Layouts

```
public void start(Stage window) {
    HBox buttons = new HBox(10, new Button("Eka"), new Button("Toka"), new Button("Kolmas"));
    VBox texts = new VBox(10, new Label("Eka"), new Label("Toka"), new Label("Kolmas"));

    BorderPane layout = new BorderPane();
    layout.setTop(buttons);
    layout.setLeft(texts);
    layout.setCenter(new TextArea("Hei..."));

    window.setScene(new Scene(layout));
    window.show();
}
```

 Multiple layouts have been used in a single interface. A BorderPane creates the frame, a HBox is at the top and a VBox on the left side. The text area in the center has some text written in it.

- Layout-Objekte lassen sich **verschachteln**
- Häufig: Ein Container als Hauptlayout (z. B. BorderPane) und darin weitere Layouts

Zusammenfassung

- **UI-Komponenten:** z. B. Label, Button, TextField
- **Layout:** FlowPane, BorderPane, HBox/VBox, GridPane etc.
- **Vorgehensweise:**
 - i. Komponente erstellen
 - ii. Zu einem Layout-Container hinzufügen
 - iii. Layout in `Scene` , Scene in `Stage`
- **Kombination** verschiedener Layouts für komplexe Oberflächen

Event Handling in Benutzungsoberflächen

Einführung in GUI-Event-Handling mit JavaFX

Event-Handler in C

- Ein **Event-Handler** ist eine Funktion, die auf ein bestimmtes Ereignis oder Signal reagiert.
- In C können Event-Handler mithilfe der **POSIX-Signal-API** implementiert werden.
- **Beispiel:** Reaktion auf ein Signal wie `SIGUSR1` .

Beispiel: Signal-Handler

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Globale Variable zum Hochzählen
volatile sig_atomic_t counter = 0;

// Signal-Handler-Funktion
void handle_signal(int signal) {
    if (signal == SIGUSR1) {
        counter++;
        printf("\nSIGUSR1 empfangen. Zähler: %d\n", counter);
    }
}
```

Hauptprogramm: Registrierung und Ausführung

```
int main() {
    // Signal-Handler registrieren
    if (signal(SIGUSR1, handle_signal) == SIG_ERR) {
        perror("Fehler beim Registrieren des Signal-Handlers");
        return 1;
    }

    printf("Prozess-ID: %d\n", getpid());
    printf("Sende SIGUSR1 mit: kill -USR1 %d\n", getpid());

    // Endlos-Schleife, um das Programm am Laufen zu halten
    while (1) {
        pause(); // Warten, bis ein Signal empfangen wird
    }

    return 0;
}
```

Erklärung des Codes

1. Globale Variable `counter` :

- `volatile sig_atomic_t` : Sicher für die Nutzung im Signal-Handler.
- Erhöht den Zähler bei jedem empfangenen Signal `SIGUSR1` .

2. Signal-Handler `handle_signal` :

- Wird aufgerufen, wenn das Signal `SIGUSR1` empfangen wird.
- Gibt den aktuellen Zählerstand aus.

3. `signal()` -Funktion:

- Registriert die Funktion `handle_signal` als Handler für `SIGUSR1` .

4. `pause()` -Funktion:

- Hält das Programm an, bis ein Signal empfangen wird.

Ausführung des Programms

1. Programm kompilieren und starten:

```
gcc -o signal_example signal_handler.c  
./signal_example
```

2. Signal senden:

- Sende `SIGUSR1` an das Programm:

```
kill -USR1 <PID>
```

- Die Prozess-ID (`PID`) wird beim Start des Programms ausgegeben.

3. Ergebnis:

- Für jedes Signal wird der Zähler erhöht und ausgegeben:

```
SIGUSR1 empfangen. Zähler: 1  
SIGUSR1 empfangen. Zähler: 2  
SIGUSR1 empfangen. Zähler: 3
```

Verbindung zu Java GUIs: Event-Handling

- **Event-Handling** ist auch in Java ein zentraler Bestandteil für interaktive Anwendungen, z. B. GUIs mit **Swing** oder **JavaFX**.
- In Java GUIs:
 - Ein Event-Handler ist ein Objekt, das auf Ereignisse wie **Mausklicks** oder **Tastendrücke** reagiert.
 - Beispiele für Ereignisse: `ActionEvent`, `MouseEvent`.
- **Parallelen zu Signal-Handling in C:**
 - Beide reagieren auf spezifische Ereignisse (Signale in C, Benutzerinteraktionen in Java).
 - Registrierung eines Handlers, der beim Ereignis aufgerufen wird.
 - Im Kern verwenden beide Mechanismen betriebssystemnahe Funktionen zur Ereignisverarbeitung.

Beispiel: Event-Handler in Java (Swing)

```
import javax.swing.*;
import java.awt.event.*;

public class EventHandlerExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Event-Handler Beispiel");
        JButton button = new JButton("Klick mich");

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button wurde geklickt!");
            }
        });

        frame.add(button);
        frame.setSize(200, 100);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```


Vergleich: C vs. Java Event-Handling

Merkmal	Signal-Handling in C	Event-Handling in Java
Ereignisse	Systemsignale (z. B. <code>SIGINT</code> , <code>SIGUSR1</code>)	Benutzerinteraktionen (z. B. Button-Klicks)
Registrierung	<code>signal()</code>	Swing: <code>addActionListener()</code> , JavaFX: <code>EventHandler</code> , <code>ChangeListener</code>
Handler	Funktion (<code>void handler()</code>)	Methode eines Listener-Objekts
Flexibilität	systemnah	Objektorientiert, vielseitig
Mechanismus	Betriebssystem löst Signale aus	JVM verarbeitet Benutzereingaben (letztendlich aus Betriebssystem)

- **Signal-Handling in C und Event-Handling in Java basieren auf ähnlichen Konzepten:**

- Registrierung eines Handlers.
- Reaktion auf spezifische Ereignisse.

- **Gemeinsamkeiten im Kern:**

- Beide Mechanismen beruhen auf betriebssystemnahen Funktionen.
- Java abstrahiert diese Mechanismen für Benutzerinteraktionen.

- **Anwendungsbereich:**

- C eignet sich für systemnahe Ereignisse wie Signale.
- Java ist ideal für Benutzerschnittstellen und komplexe Ereignisketten.

Lernziele

- **Verständnis:** Was ist ein Event-Handler?
- **Fähigkeit:** Wie können Benutzungsoberflächen-Ereignisse (z. B. Button-Klicks) verarbeitet werden?

Event Handling: Grundlagen

- Ereignisse wie **Button-Klicks** erfordern einen **Event-Handler**.
- JavaFX bietet das Interface `EventHandler`.
- Beispiel für einen Button-Klick:

```
Button button = new Button("This is a button");  
button.setOnAction((event) -> System.out.println("Pressed!"));
```

Ergebnis:

Beim Klick auf den Button wird `Pressed!` in der Konsole ausgegeben.

Komponenten & Ereignisse verknüpfen

- Jeder **Event-Handler** ist mit einer **Komponente** verbunden.
- Beispiel: Der Klick auf einen Button führt zu einer Aktion.
- **Komponenten können mehrere Ereignisse verwalten.**

Beispiel: Text kopieren

Ziel: Der Inhalt eines Textfelds wird in ein anderes kopiert.

```
@Override
public void start(Stage window) {
    TextField leftText = new TextField();
    TextField rightText = new TextField();
    Button button = new Button("Kopioi");

    button.setOnAction((event) -> rightText.setText(leftText.getText()));

    HBox group = new HBox(20, leftText, button, rightText);
    window.setScene(new Scene(group));
    window.show();
}
```

Ergebnis:

Der Button "Kopioi" kopiert den Text von links nach rechts.

 Two text fields and a button with the text 'Copy'.

Ergebnis: Text erfolgreich kopiert

Nach dem Klick auf "Kopioi":

 Two text fields and a button with the text 'Copy'.

Lambda-Ausdrücke im Event Handling

- Lambda-Ausdrücke sind kompakter als klassische Implementierungen:

```
button.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent event) {  
        System.out.println("Clicked!");  
    }  
});
```

- Mit Lambda-Ausdruck:

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

- Ergebnis ist identisch.

Fehler bei Referenzänderungen

- **Achtung:** Event-Handler können nur mit bestehenden Referenzen arbeiten.
- **Dieser Code führt zu einem Fehler:**

```
button.setOnAction((event) -> {  
    rightText = new TextField(); // Fehler!  
});
```

Erweiterte Events: **ChangeListener**

- Zeichenweise Änderungen verfolgen:

```
leftText.textProperty().addListener((change, oldValue, newValue) -> {  
    rightText.setText(newValue);  
});
```

Vorteil:

Echtzeit-Reaktion auf Benutzereingaben.

Live-Statistiken aus Textfeldern

- Text analysieren:
 - Zeichenanzahl
 - Wörter zählen
 - Längstes Wort

```
leftText.textProperty().addListener((change, oldValue, newValue) -> {  
    int characters = newValue.length();  
    int words = newValue.split(" ").length;  
    String longest = Arrays.stream(newValue.split(" "))  
        .max(Comparator.comparingInt(String::length))  
        .orElse("");  
});
```

Erweiterung: Dynamische Statistiken

Funktionalität:

Die Anwendung zeigt automatisch aktualisierte Statistiken:

- Zeichenanzahl
- Wörteranzahl
- Längstes Wort

Beispiel:

 A example of the functionality of an application made for text statistics.

Zusammenfassung

- **Event-Handler:**
 - Verknüpfen Aktionen mit GUI-Ereignissen.
 - Lambda-Ausdrücke vereinfachen die Implementierung.
- **ChangeListener:** Echtzeit-Reaktion auf Eingaben.
- **Beispiele:**
 - Textfeld-Kopierer
 - Echtzeit-Statistiken

Codebeispiel:

```
button.setOnAction((event) -> {  
    // Aktionen beim Button-Klick  
});
```

Parameter grafischer Benutzungsoberflächen

Übergabe von Parametern an JavaFX-Anwendungen

Lernziele

- Verstehen, wie **Parameter** an eine JavaFX-Benutzungs Oberfläche übergeben werden.
- Anwendungen dynamisch starten und konfigurieren.

Einführung: Start einer JavaFX-Anwendung

Eine einfache JavaFX-Anwendung:

```
import javafx.application.Application;
import javafx.stage.Stage;

public class JavaFxApplication extends Application {
    @Override
    public void start(Stage window) {
        window.setTitle("Hello World!");
        window.show();
    }
}
```

Die Anwendung wird typischerweise über die `launch` -Methode gestartet.

Starten außerhalb der **Application**-Klasse

- JavaFX-Anwendungen können auch **außerhalb** der Klasse gestartet werden.

Beispiel:

```
import javafx.application.Application;

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

Übergabe von Parametern zur Laufzeit

- **Parameterübergabe** an die Anwendung durch `launch` -Methode.
- Aufrufbeispiel:

```
Application.launch(JavaFxApplication.class,  
    "--organization=Once upon a time",  
    "--course=Title");
```

- Zugriff auf die Parameter in der `start` -Methode:

```
Parameters params = getParameters();  
String organization = params.getNamed().get("organization");  
String course = params.getNamed().get("course");
```

Anwendung: Parameter nutzen

```
import javafx.application.Application;
import javafx.stage.Stage;

public class JavaFxApplication extends Application {
    @Override
    public void start(Stage window) {
        Parameters params = getParameters();
        String organization = params.getNamed().get("organization");
        String course = params.getNamed().get("course");

        window.setTitle(organization + ": " + course);
        window.show();
    }
}
```

Ergebnis:

Die Benutzungsoberfläche zeigt den dynamischen Titel basierend auf den übergebenen Parametern.

Start mit Parametern

```
public class Main {  
    public static void main(String[] args) {  
        Application.launch(JavaFxApplication.class,  
            "--organization=Once upon a time",  
            "--course=Title");  
    }  
}
```

Ergebnis:

Die Benutzungsoberfläche wird mit dem Titel **"Once upon a time: Title"** gestartet.

Verwendungsmöglichkeiten

- **Dateipfade:** Übergabe einer Datei, die geladen oder gespeichert werden soll.
- **Konfigurationen:** Weitergabe von Einstellungen oder Themen.
- **Webadressen:** Übergabe einer URL für Daten oder APIs.

Multiple Views

**Hinzufügen mehrerer Ansichten und Wechsel zwischen Szenen in
JavaFX**

Lernziele

- Hinzufügen mehrerer Ansichten zu einer **grafischen Benutzungsoberfläche**.
- Methoden zum Wechseln zwischen Ansichten kennenlernen.
- Trennung von **Anwendungslogik** und **Benutzungsoberfläche** üben.

Einführung: Mehrere Ansichten

- Ansichten werden als **Scene-Objekte** erstellt.
- Wechsel zwischen Szenen durch **Ereignisse** (z. B. Button-Klick).

Beispiel: Zwei Szenen mit einem Button, um zwischen den Ansichten zu wechseln.

```
Button back = new Button("Back ..");
Button forth = new Button(".. forth.");
Scene first = new Scene(back);
Scene second = new Scene(forth);

back.setOnAction(e -> window.setScene(second));
forth.setOnAction(e -> window.setScene(first));


window.setScene(first);
window.show();
```


Beispielanwendung: Szenenwechsel

```
public class BackAndForthApplication extends Application {  
    @Override  
    public void start(Stage window) {  
        Button back = new Button("Back ..");  
        Button forth = new Button(".. forth.");  
  
        Scene first = new Scene(back);  
        Scene second = new Scene(forth);  
  
        back.setOnAction(e -> window.setScene(second));  
        forth.setOnAction(e -> window.setScene(first));  
  
        window.setScene(first);  
        window.show();  
    }  
}
```

Eigenes Layout für jede Ansicht

Beispiel: Passwortabfrage mit Szenenwechsel nach erfolgreicher Eingabe.

 Passwortabfrage mit Szenenwechsel.

- **Erste Ansicht:** Passwortabfrage.
- **Zweite Ansicht:** Begrüßungsnachricht bei korrektem Passwort.

Beispiel: Passwortgeschützte Anwendung

```
Label instruction = new Label("Write the password and press Log in");
PasswordField passwordField = new PasswordField();
Button login = new Button("Log in");
Label error = new Label("");

GridPane layout = new GridPane();
layout.add(instruction, 0, 0);
layout.add(passwordField, 0, 1);
layout.add(login, 0, 2);
layout.add(error, 0, 3);

Scene passwordView = new Scene(layout);

login.setOnAction(e -> {
    if (!passwordField.getText().equals("password")) {
        error.setText("Unknown password!");
    } else {
        window.setScene(welcomeView);
    }
});
```

Beispiel: Ansichten mit Menü

- **Permanentes Menü** für Navigation.
- Inhalt wechselt bei Klick auf Menütasten.

 Wechsel zwischen Ansichten über ein Menü.

Menü mit wechselnden Ansichten

```
BorderPane layout = new BorderPane();

HBox menu = new HBox();
menu.setSpacing(10);
Button first = new Button("First");
Button second = new Button("Second");
menu.getChildren().addAll(first, second);

layout.setTop(menu);

StackPane firstView = createView("First view");
StackPane secondView = createView("Second view");

first.setOnAction(e -> layout.setCenter(firstView));
second.setOnAction(e -> layout.setCenter(secondView));

layout.setCenter(firstView);
```

Trennung von Logik und UI

- **Vorteil:** Verbesserte Lesbarkeit und Testbarkeit.
- **Ansatz:**
 - Logik in separaten Klassen.
 - UI verwendet diese Klassen über Interfaces.

Beispiel: Personenspeicher

```
interface PersonWarehouse {  
    void save(Person person);  
    Collection<Person> getAll();  
}
```

UI für Eingabe von Personen:

```
Label name = new Label("Name:");  
TextField nameField = new TextField();  
Button add = new Button("Add person!");  
  
add.setOnAction(e -> warehouse.save(new Person(nameField.getText())));
```

 UI für Personeneingabe.

Vokabeltrainer: Anwendung

 Vokabeltrainer mit mehreren Ansichten.

1. Wörterbuch pflegen (Wort und Übersetzung eingeben).
2. Übersetzungen üben (Wörter zufällig auswählen).
3. Trennung der Logik und UI in Klassen.

Beispiel: Wörterbuchklasse

```
public class Dictionary {
    private Map<String, String> translations = new HashMap<>();

    public void add(String word, String translation) {
        translations.put(word, translation);
    }

    public String get(String word) {
        return translations.get(word);
    }

    public String getRandomWord() {
        List<String> words = new ArrayList<>(translations.keySet());
        return words.get(new Random().nextInt(words.size()));
    }
}
```

Beispiel: Eingabeansicht

```
public class InputView {
    private Dictionary dictionary;

    public InputView(Dictionary dictionary) {
        this.dictionary = dictionary;
    }

    public Parent getView() {
        GridPane layout = new GridPane();
        TextField wordField = new TextField();
        TextField translationField = new TextField();
        Button addButton = new Button("Add");

        addButton.setOnAction(e -> {
            dictionary.add(wordField.getText(), translationField.getText());
            wordField.clear();
            translationField.clear();
        });

        layout.add(new Label("Word:"), 0, 0);
        layout.add(wordField, 1, 0);
        layout.add(new Label("Translation:"), 0, 1);
        layout.add(translationField, 1, 1);
        layout.add(addButton, 1, 2);

        return layout;
    }
}
```

Zusammenfassung

- Mehrere **Ansichten** in JavaFX möglich:
 - Szenenwechsel per Ereignisse (z. B. Buttons).
 - Permanente Elemente (z. B. Menü) in Layouts integrieren.
- **Best Practice:** Trennung von Logik und UI.
- Anwendungen wie Vokabeltrainer und Menüsysteme basieren auf diesen Konzepten.

MVC am Wörterbuchbeispiel

Ziel: Verdeutlichen, wie Model-View-Controller in einer JavaFX-Anwendung mit einem **Dictionary** (Wörterbuch) umgesetzt werden kann.

Architekturüberblick

Model (Dictionary)

- Verwaltet Daten und Logik (z. B. Wörter und Übersetzungen).

View (JavaFX GUI)

- Präsentiert Daten dem Benutzer (z. B. Textfelder, Labels).
- Bietet Interaktionsmöglichkeiten (Buttons, Menüs).

Controller

- Vermittelt zwischen Model und View.
- Reagiert auf Benutzeraktionen in der View.
- Aktualisiert Model und View entsprechend.

Model: Dictionary

```
import java.util.*;

public class Dictionary {
    private Map<String, String> translations = new HashMap<>();

    public void add(String word, String translation) {
        translations.put(word, translation);
    }

    public String get(String word) {
        return translations.get(word);
    }

    public String getRandomWord() {
        List<String> words = new ArrayList<>(translations.keySet());
        if (words.isEmpty()) {
            return null; // oder Exception werfen
        }
        Random rand = new Random();
        return words.get(rand.nextInt(words.size()));
    }
}
```

Aufgaben:

- Speichern und Abfragen von Übersetzungen.
- Enthält keinerlei Code für Darstellung oder Benutzerinteraktion.

View: Eingabeansicht (Beispiel)

```
import javafx.scene.Parent;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;

public class DictionaryView {

    private TextField wordField;
    private TextField translationField;
    private Button addButton;

    public DictionaryView() {
        this.wordField = new TextField();
        this.translationField = new TextField();
        this.addButton = new Button("Add Translation");
    }

    public Parent getView() {
        GridPane layout = new GridPane();
        layout.add(new Label("Word:"), 0, 0);
        layout.add(wordField, 1, 0);
        layout.add(new Label("Translation:"), 0, 1);
        layout.add(translationField, 1, 1);
        layout.add(addButton, 1, 2);

        return layout;
    }

    public String getWordInput() {
        return wordField.getText();
    }

    public String getTranslationInput() {
        return translationField.getText();
    }

    public Button getAddButton() {
        return addButton;
    }

    public void clearInputs() {
        wordField.clear();
        translationField.clear();
    }
}
```

Merkmale:

- Stellt die Oberflächenelemente bereit.
- Keine Logik zum Speichern, Lesen oder Verarbeiten von Daten.

Controller: Anbindung Model & View

```
public class DictionaryController {  
  
    private Dictionary model;    // Das Datenmodell  
    private DictionaryView view; // Die View-Komponenten  
  
    public DictionaryController(Dictionary model, DictionaryView view) {  
        this.model = model;  
        this.view = view;  
    }  
  
    public void initController() {  
        // Button-Click-Event abfangen und verarbeiten  
        view.getAddButton().setOnAction(event -> {  
            String word = view.getWordInput();  
            String translation = view.getTranslationInput();  
  
            // Logik des Models aufrufen  
            model.add(word, translation);  
  
            // View zurücksetzen, damit neue Einträge eingegeben werden können  
            view.clearInputs();  
        });  
    }  
}
```

Aufgaben:

- Reagiert auf Aktionen in der View (z. B. Klick auf "Add Translation").
- Ruft Model-Methoden auf, um Einträge zu speichern.

Hauptanwendung (Main / Application)

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

public class DictionaryApp extends Application {

    @Override
    public void start(Stage window) {
        // Model erstellen
        Dictionary model = new Dictionary();

        // View erstellen
        DictionaryView view = new DictionaryView();

        // Controller erstellen und verbinden
        DictionaryController controller = new DictionaryController(model, view);
        controller.initController();

        // JavaFX-Szene erstellen
        Scene scene = new Scene(view.getView(), 400, 200);

        window.setTitle("Dictionary (MVC)");
        window.setScene(scene);
        window.show();
    }

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

Vorteile des MVC-Ansatzes

1. Trennung der Verantwortlichkeiten

- Model: Daten und Logik
- View: Darstellung und Benutzerinteraktion
- Controller: Steuerung und Vermittlung

2. Wartbarkeit

- Änderungen im GUI betreffen nur die View, ohne das Model zu beeinflussen.
- Logik kann unabhängig getestet und weiterentwickelt werden.

3. Erweiterbarkeit

- Neue Views oder zusätzliche Controller können hinzugefügt werden, ohne das Model zu ändern.

4. Testbarkeit

- Logik (Model) und Controller-Funktionen lassen sich leichter in Unit-Tests prüfen.
- GUI-Komponenten können teils separat getestet oder mit Mock-Objekten ersetzt werden.

- **MVC** fördert eine **klare Struktur** in JavaFX-Anwendungen.
- Das **Wörterbuchbeispiel** zeigt, wie Datenmodell, GUI und Controller entkoppelt zusammenarbeiten.
- Durch konsequente Anwendung von **MVC** entsteht wartbare und erweiterbare Software.

Empfehlung:

- Auch bei kleineren Projekten lohnt sich eine (leichte) Trennung von Model, View und Controller, um spätere Änderungen zu erleichtern.

Tic Tac Toe Min(i)Max

