

Benutzerschnittstellen mit JavaFX

Inhaltsverzeichnis

- 1. Einführung in JavaFX
 - 1.1. Was ist JavaFX?
 - 1.2. Warum ist JavaFX wichtig?
 - 1.3. JavaFX-Anwendungsarchitektur
 - 1.4. Struktur einer JavaFX-Anwendung
 - 1.4.1. Struktur eines JavaFX - Projekts
 - 1.5. Erstellen einer JavaFX - Anwendung mit Maven **Archetype**
 - 1.5.1. Voraussetzungen an das zu verwendende SDK
 - 1.5.2. Erstellen des Maven Projekts in Eclipse
 - 1.5.3. Einrichten des JavaFX Projekts mittels IntelliJ
 - 1.6. Erstellen einer JavaFX - Anwendung mit Maven OHNE Archetype
 - 1.6.1. Voraussetzungen
 - 1.6.2. Erstellen des leeren Maven Projekts in IntelliJ
 - 1.6.3. Anpassen der **pom.xml**
 - 1.6.4. Erstellen der Java-Hauptklasse
 - 1.6.5. Ausführen und Debuggen in IntelliJ (ohne Maven Goal)
- 2. JavaFX-Klassen und Komponenten
 - 2.1. Die beiden Haupt Klassen **Stage** und **Scene**
 - 2.1.1. Die **State** Klasse
 - 2.1.2. Die **Scene** Klasse
 - 2.1.3. Das Scene - Graph Modell
 - 2.2. Control-Klassen (z.B. Button, Label, Textfield)
 - 2.3. Layout-Klassen (z.B. BorderPane, VBox, HBox)
 - 2.3.1. Tipps zum Desig von Grafischen Oberflächen
- 3. Event-Handling in JavaFX
 - 3.1. Verwendung einer Event-Handler Klasse
 - 3.2. Identifizierung von Ereignisquellen und Ereignisarten
 - 3.2.1. Identifizierung von Ereignisquellen
 - 3.2.2. Ereignisart bzw. Typ identifizieren
- 4. Verwendung von FXML
 - 4.1. Was ist FXML
 - 4.2. Grundlegende Struktur von FXML
 - 4.2.1. XML - Grundlagen
 - 4.2.2. FXML - Dokument Struktur
 - 4.3. Das Zusammenspiel von Controller Klasse und FXML - Datei
 - 4.3.1. Definition von Event - Methoden
 - 4.3.2. Beispiel von FXML - Datei und zugehörigem Controller
 - 4.3.3. Verknüpfen der Controller Klasse mit der FXML - Datei
 - 4.3.4. Laden der FXML - Datei und Instanzierung der Controllerklasse.
- 5. Das MVC-Patterns (Model-View-Controller) in JavaFX
 - 5.1. Was ist das MVC - Pattern

- 5.2. Umsetzung des MVC-Patterns in JavaFX
 - 5.2.1. Das **Model**
 - 5.2.2. Die **View**
 - 5.2.3. View OHNE Angabe der konkreten Implementierung des Controller - Interfaces
 - 5.2.4. Der **Controller**
- 6. Eingabeelemente in JavaFX
 - 6.1. Eingabeelemente mit Auswahlmöglichkeit
 - 6.1.1. Dropdown (ComboBox)
 - 6.1.2. Radio-Buttons
 - 6.1.3. Checkbox
 - 6.2. Weitere Eingabeelemente
 - 6.2.1. Textarea
- 7. Layouting, Dialoge und Datenpräsentation in JavaFX
 - 7.1. Layout - Klassen in JavaFX
 - 7.1.1. Überblick über die gängigsten Layout - Klassen
 - 7.2. Dialoge in JavaFX
 - 7.2.1. JavaFX Standard Dialoge
 - 7.2.2. Benutzerdefinierte Dialoge in JavaFX
 - 7.2.3. Verwenden eines **FileChooser** Dialogs
 - 7.3. Verwendung von TableView zur Verwaltung von Tabellendaten
 - 7.3.1. Was ist eine Table View
 - 7.3.2. Defintion einer TableView in FXML
 - 7.3.3. Initialisieren und verwalten einer TableView im Controller
 - 7.4. Realsierung von mehrsprachigen Benutzeroberflächen

Version History

Version	Änderungen	Autor
2023-06-19	offizielle Erstversion für SJ 23/24	KUW
2024-03-14	- 3.1 Verwendung einer EventHandler Klasse - 5.6 Überblick über gängige Layout Klassen	KUW
2024-03-30	Überarbeitung des Abschnitts 6 - Benutzerinteraktionen und Datenpräsentation in JavaFX - neu: JavaFX - Standard Dialoge	KUW
2025-04-09	Neuer Abschnitt: 1.6. Erstellen einer JavaFX - Anwendung mit Maven OHNE Archetype	KUW
2025-05-14	Neuer Abschnitt: 6.1. Eingabeelemente mit Auswahlmöglichkeit	KUW

1. Einführung in JavaFX

1.1. Was ist JavaFX?

JavaFX ist eine GUI-Toolkit-Plattform und eine umfassendes Software Development Kit (= SDK) für die Entwicklung von plattformübergreifenden Desktop-, Mobil- und Webanwendungen in Java. Es ist ein Teil des Java Development Kit (JDK) und wurde erstmals im Jahr 2008 veröffentlicht.

JavaFX bietet eine Fülle von UI-Elementen, Layout-Optionen und Anwendungsframeworks, die Entwicklern dabei helfen, ansprechende und interaktive Anwendungen zu erstellen. Mit JavaFX können Entwickler visuell ansprechende Anwendungen erstellen, die über moderne UI-Designs verfügen und auf verschiedenen Plattformen ausgeführt werden können.

JavaFX ist besonders gut für die Erstellung von reaktionsfähigen und interaktiven Anwendungen geeignet, da es eine hohe Leistung und schnelle Rendering-Fähigkeiten bietet. Es wird oft als Alternative zu Swing, der älteren GUI-Toolkit-Plattform von Java, betrachtet und hat sich als die bevorzugte Wahl für die Entwicklung moderner Anwendungen etabliert.

1.2. Warum ist JavaFX wichtig?

1. **Plattformübergreifende Entwicklung:** JavaFX ermöglicht die Entwicklung von plattformübergreifenden Anwendungen, die auf verschiedenen Betriebssystemen wie Windows, macOS, Linux und mobilen Plattformen wie Android und iOS ausgeführt werden können.
2. **Moderne UI-Designs:** Mit JavaFX können Entwickler ansprechende Benutzeroberflächen und moderne UI-Designs erstellen, die für eine bessere Benutzererfahrung sorgen.
3. **Leistung:** JavaFX bietet eine hohe Leistung und schnelle Rendering-Fähigkeiten, die für die Erstellung von interaktiven und reaktionsfähigen Anwendungen erforderlich sind.
4. **Integration mit Java-Ökosystem:** JavaFX ist eng in das Java-Ökosystem integriert, was bedeutet, dass es leicht mit anderen Java-Technologien wie Spring und Hibernate integriert werden kann.
5. **Unterstützung durch Oracle und einer breiten Community:** Oracle ist der primäre Entwickler und Sponsor von JavaFX und bietet eine umfassende Dokumentation, Support und Schulungen für die Entwicklergemeinschaft.

Insgesamt bietet JavaFX eine leistungsstarke Plattform für die Entwicklung moderner Anwendungen und ermöglicht es Entwicklern, schnell ansprechende UI-Designs zu erstellen, die auf einer Vielzahl von Plattformen ausgeführt werden können.

1.3. JavaFX-Anwendungsarchitektur

Die JavaFX-Anwendungsarchitektur umfasst die folgenden Komponenten:

1. **Application Klasse:** Dies ist die **Hauptklasse** der JavaFX-Anwendung. Diese Klasse erweitert die abstrakte Klasse "`javafx.application.Application`" und enthält die `main`-Methode. Es ist verantwortlich für das Starten der Anwendung und das Erstellen der primären Stage.
2. **Stage Klasse:** Die Stage-Klasse repräsentiert ein Fenster in der Anwendung. Es ist das oberste Level des Scene Graphs und enthält die Szene, die auf dem Fenster angezeigt wird. Eine Anwendung kann mehrere Stages haben.

3. **Scene Klasse:** Die Scene-Klasse repräsentiert den Inhalt, der auf der Stage angezeigt wird. Es ist der Container für alle UI-Elemente der Anwendung und kann eine Hierarchie von Nodes (Knoten) enthalten.
4. **Node Klasse:** Die Node-Klasse ist die Basisklasse für alle UI-Elemente in JavaFX. Jedes Element der Benutzeroberfläche, einschließlich Layouts, Steuerelemente, Formen und Grafiken, ist eine Node. Nodes können in einer Baumstruktur (Scene-Graph) organisiert werden, um eine hierarchische Struktur der Anwendungsoberfläche zu erstellen.
5. **Layout Manager Klassen:** JavaFX bietet eine Vielzahl von Layout-Managern wie **BorderPane**, **FlowPane**, **HBox**, **VBox**, **StackPane**, **AnchorPane** und **GridPane**, um die Positionierung und Größenverwaltung von Nodes zu erleichtern.
6. **Event - Handling:** JavaFX bietet eine umfangreiche Unterstützung für die Ereignisbehandlung, um auf **Benutzeraktionen** wie **Mausklicks**, **Tastatureingaben** und Änderungen von Daten zu reagieren. Die Ereignisbehandlung kann mithilfe von Event-Handling-Methoden oder einem **Observer Pattern** implementiert werden.

Zusammengefasst bietet die JavaFX-Anwendungsarchitektur eine robuste und flexible Struktur für die Entwicklung von modernen Anwendungen. Es bietet eine einfache Möglichkeit, UI-Elemente zu erstellen, zu organisieren und zu verwalten, während es gleichzeitig eine umfangreiche Unterstützung für Ereignisbehandlung und Layout-Management bietet.

1.4. Struktur einer JavaFX-Anwendung

Die Struktur einer JavaFX-Anwendung kann je nach Entwicklerpräferenz und Projektanforderungen variieren. Eine **typische Struktur** einer JavaFX-Anwendung kann jedoch folgende Komponenten umfassen:

1. **Application Klasse:** Dies ist die Hauptklasse der JavaFX-Anwendung und erweitert die abstrakte Klasse "javafx.application.Application". Es enthält die main-Methode und ist verantwortlich für das Starten der Anwendung und das Erstellen der primären Stage.
2. **View-Komponenten:** Die View-Komponenten umfassen **alle UI-Elemente**, die in der Anwendung angezeigt werden, wie Labels, Buttons, Textfelder, Listenansichten und Tabellen. Diese Komponenten werden **in der Regel als FXML-Dateien** definiert, die mit dem **Scene Builder** erstellt werden können.
3. **Controller-Komponenten:** Die Controller-Komponenten sind für Event-Handling und die Verwaltung der Interaktionen zwischen den View-Komponenten verantwortlich. Es kann eine oder mehrere Controller-Klassen geben, die spezifische Views verwalten und Aktionen auf Benutzerinteraktionen ausführen.
4. **Model-Komponenten:** Das Model ist für die Verwaltung der Geschäftslogik und der Daten der Anwendung verantwortlich. Es enthält Klassen, die Datenobjekte und Methoden definieren, die auf die Daten zugreifen und sie aktualisieren können.
5. **Ressourcen-Ordner:** Der Ressourcen-Ordner enthält alle Ressourcen, die von der Anwendung verwendet werden, wie Bilder, CSS-Dateien und Übersetzungsdateien.
6. **Libraries:** Libraries sind externe Bibliotheken, die von der Anwendung verwendet werden können, um zusätzliche Funktionalität bereitzustellen, wie z.B. **Datenbankzugriff oder HTTP-Anforderungen**.

Insgesamt bietet die Struktur einer JavaFX-Anwendung eine klare Trennung der Verantwortlichkeiten zwischen UI-Komponenten, Datenverwaltung und Ereignisbehandlung. Diese Trennung erleichtert die Entwicklung und Wartung der Anwendung und ermöglicht es Entwicklern, effizienter zu arbeiten.

1.4.1. Struktur eines JavaFX - Projekts

```

MyJavaFXApp/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   ├── com/
│   │   │   │   ├── example/
│   │   │   │   │   ├── MyJavaFXApp.java      // Die Hauptklasse der Anwendung
│   │   │   │   │   ├── controller/
│   │   │   │   │   │   ├── MainController.java // Controller-Klasse für die
│   │   │   │   │   │   │   ├── model/
│   │   │   │   │   │   │   │   ├── User.java      // Datenmodell-Klasse
│   │   │   │   │   │   │   │   ├── resources/
│   │   │   │   │   │   │   │   │   ├── view/
│   │   │   │   │   │   │   │   │   │   ├── MainView.fxml // Die Hauptansicht in FXML-Format
│   │   │   │   │   │   │   │   │   │   ├── images/
│   │   │   │   │   │   │   │   │   │   ├── css/
│   │   │   │   │   │   │   │   │   │   └── lang/
│   │   │   │   │   └── test/
│   │   │   │   │       ├── java/
│   │   └── build.gradle // Build-Skript
│   └── README.md        // Projektbeschreibung

```

1.5. Erstellen einer JavaFX - Anwendung mit Maven **Archetype**

Um ein einfaches JavaFX Projekt mittels Maven zu erstellen werden folgende **archetype** mit folgender Definition bereitgestellt:

1. archetypeGroupId: **org.openjfx**
2. archetypeArtifactId:
 1. **org.openjfx:javafx-archetype-simple**: Für JavaFX Anwendungen **OHNE FXML**
 2. **org.openjfx:javafx-archetype-fxml**: Für JavaFX Anwendungen **MIT FXML**
3. Version: **0.0.6**

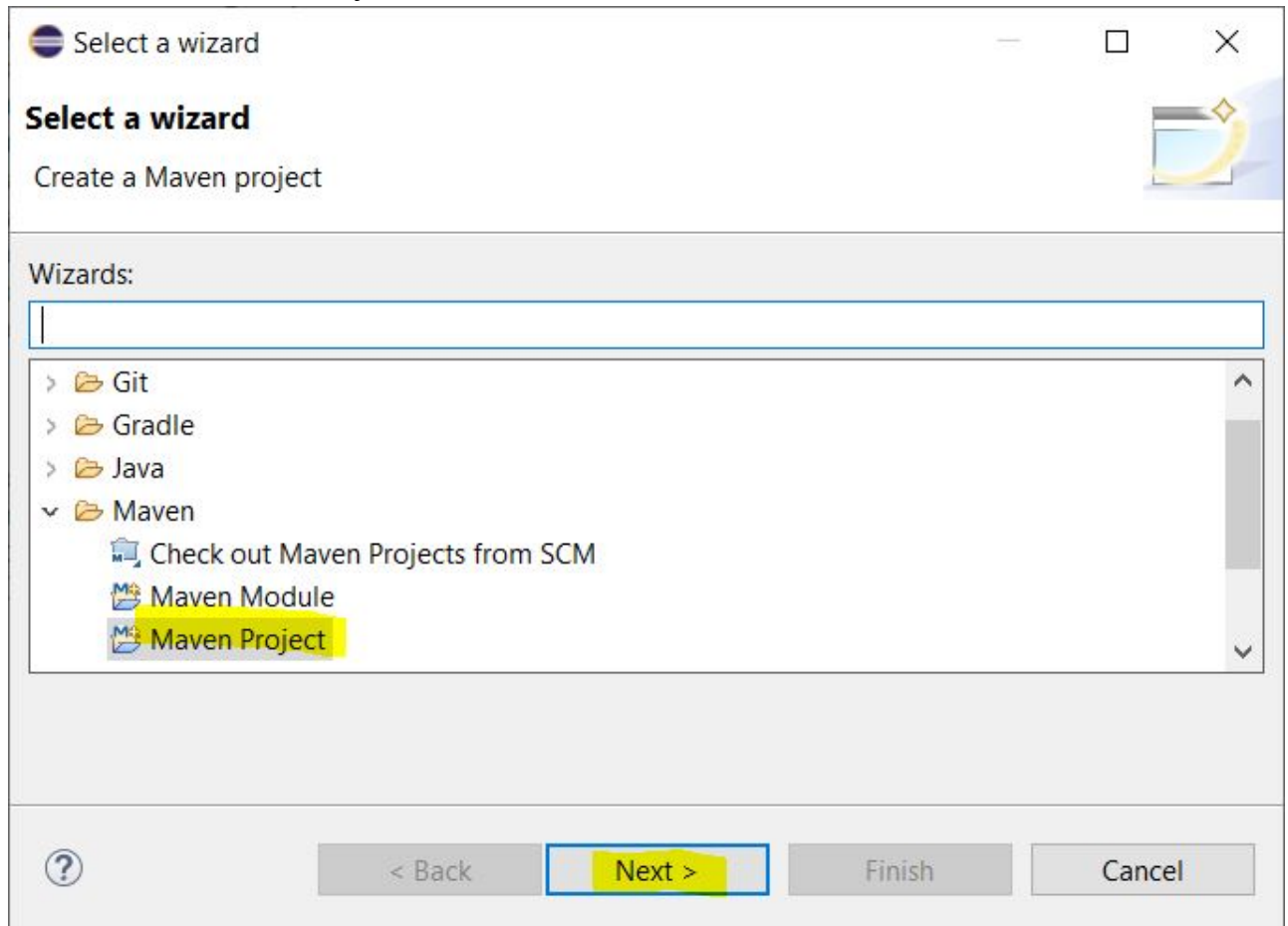
1.5.1. Voraussetzungen an das zu verwendende SDK

Erfolgreich getestet wurden beide zuvor erwähnten **archetypes** mit der JDK - Version **11**

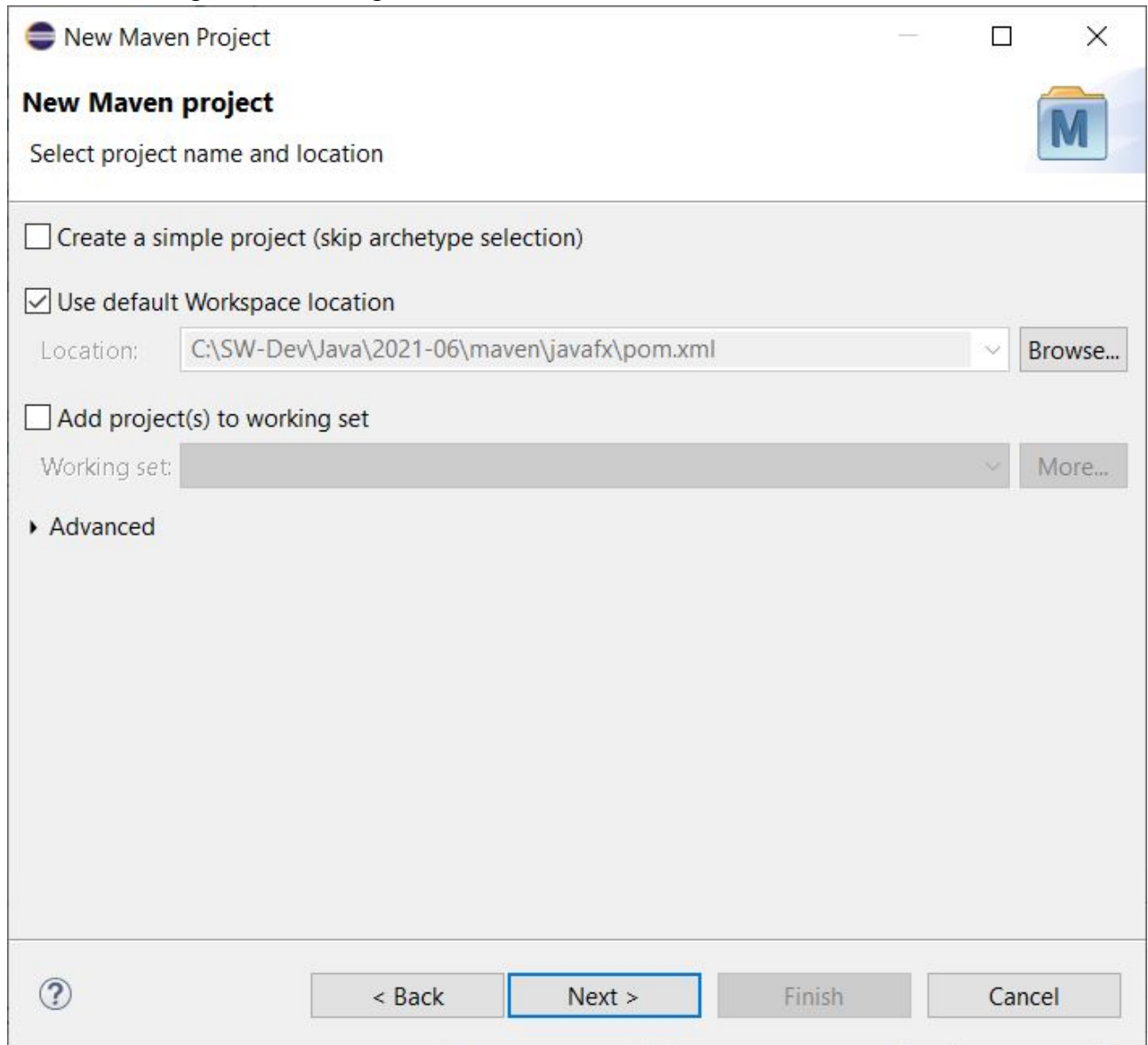
Bitte beachten Sie auch, dass bei der verwendeten IDE das **Standard SDK** die **Version 11** ist.

1.5.2. Erstellen des Maven Projekts in Eclipse

1. New -> Other -> Maven Project



2. Im zweiten Dialog keine Änderungen vor nehmen und einfach **Next** klicken



3. Im dritten Dialog muss der **archetype** gewählt werden, hier reicht es in dem Suchfeld: **openjfx** einzugeben. Danach sollten die zwei archetypen mit der Artifactid:

- javafx-archetype-fxml
- javafx-archetype-simple

zur Verfügung stehen (siehe Screenshot)

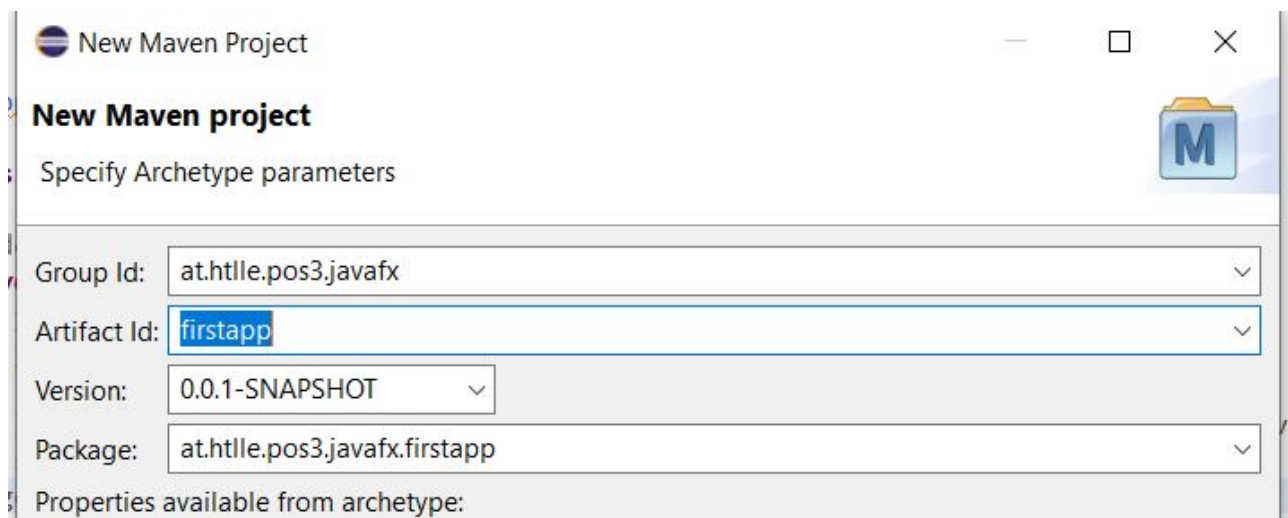
New Maven project

Select an Archetype

Catalog:	All Catalogs		
Filter:	openjfx		
Group Id	Artifact Id	Version	
org.openjfx	javafx-archetype-fxml	0.0.6	
org.openjfx	javafx-archetype-simple	0.0.6	

4. Im 4. Dialog geben wählen sie eine passende groupId, sowie artifactId für ihr Maven Projekt (vgl. Screenshot):

- groupId: `at.htlle.pos3.javafx`
- artifactId: `firstapp`



New Maven Project

New Maven project

Specify Archetype parameters

Group Id: `at.htlle.pos3.javafx`

Artifact Id: `firstapp`

Version: `0.0.1-SNAPSHOT`

Package: `at.htlle.pos3.javafx.firstapp`

Properties available from archetype:

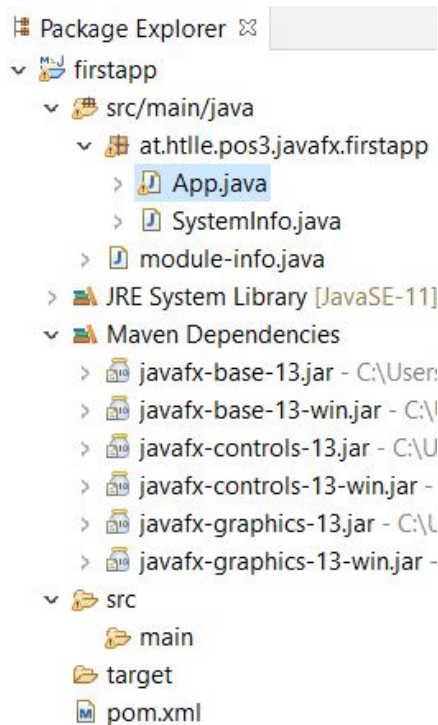
Nach diesen Schritten sollte nun das entsprechende Maven Projekt fertig erstellt sein:

Ansicht Package Explorer

JavaFX - App.java

Ansicht Package Explorer

JavaFX - App.java



```

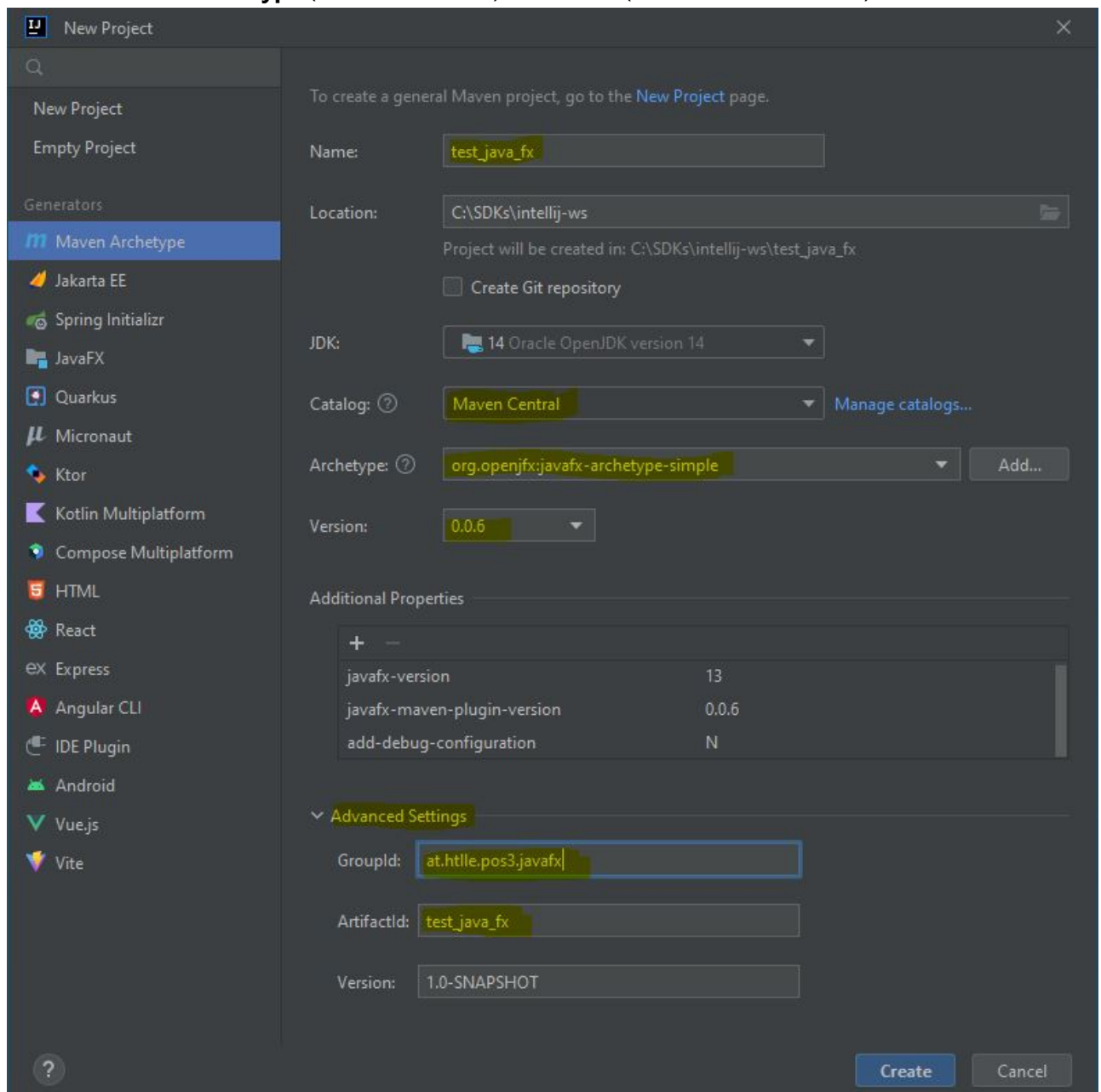
1 package at.htlle.javaafx;
2
3 import javafx.application.Application;
4
5 /**
6  * JavaFX App
7  */
8 public class App extends Application {
9
10     @Override
11     public void start(Stage stage) {
12         var javaVersion = SystemInfo.javaVersion();
13         var javafxVersion = SystemInfo.javafxVersion();
14
15         var label = new Label("Hello, JavaFX " + javafxVersion);
16         var scene = new Scene(new StackPane(label), 640, 480);
17         stage.setScene(scene);
18         stage.show();
19     }
20
21     public static void main(String[] args) {
22         Launch();
23     }
24 }

```

1.5.3. Einrichten des JavaFX Projekts mittels IntelliJ

Unter File -> **New Project**:

1. Auswahl **Maven ArcheType** (unter Generators) auswählen (siehe auch Screenshot)



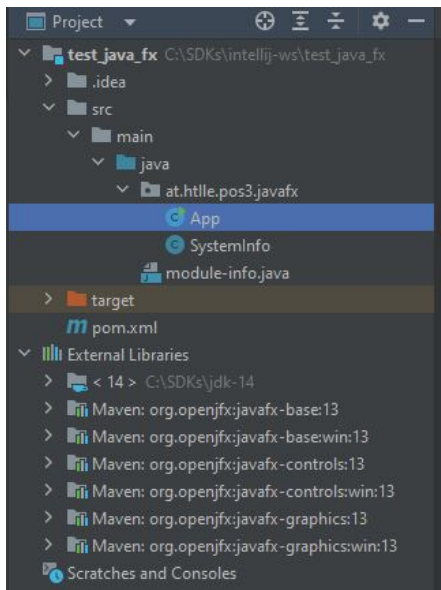
2. Unter **Name**: Name des Projekts (z.B. test_java_fx)
3. Unter **Location**: Ihr Verzeichnis unter dem sie das Projekt speichern möchten
4. Unter **JDK**: sollte mindesten JDK 11 sein
5. Unter **Catalog**: Hier **Maven Central** auswählen!
6. Unter **Archetype**: Nach Eingabe von `openfxml` sollten folgende beide Archetypen angezeigt werden:
 1. `org.openfx:javaafx-archetype-simple`: Für JavaFX Anwendungen **OHNE FXML**
 2. `org.openjfx:javaafx-archetype-fxml`: Für JavaFX Anwendungen **MIT FXML**
7. Unter **Advanced Settings**:
 1. Unter GroupId: `at.htlle.pos3.javaafx`
 2. Unter ArtifactId: Paketname der Anwendung (z.B. `test_java_fx`) - Achtung Keine Leerzeichen, Bindstriche udgl. verwenden - muss ein gültiger Java - Paketname sein)

Nach diesen Schritten sollte nun das entsprechende Maven Projekt fertig erstellt sein.

1.5.3.1. Projekt **OHNE FXML**

Ansicht Projekt Explorer

JavaFX - App.java



```
public class App extends Application {

    @Override
    public void start(Stage stage) {
        var javaVersion = SystemInfo.javaVersion();
        var javafxVersion = SystemInfo.javafxVersion();

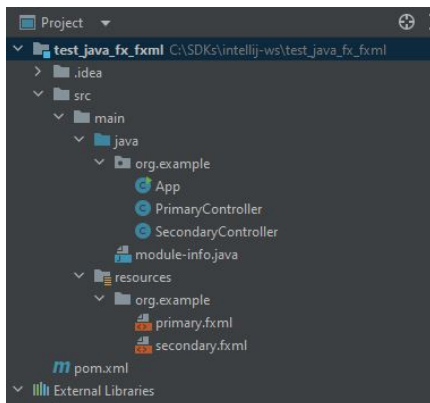
        var label = new Label( text: "Hello, JavaFX " + javafxVersion + ", run");
        var scene = new Scene(new StackPane(label), width: 640, height: 480);
        stage.setScene(scene);
        stage.show();
    }

    no usages
    public static void main(String[] args) { launch(); }
}
```

1.5.3.2. Projekt MIT FXML

Ansicht Projekt Explorer

JavaFX - App.java



```
public class App extends Application {

    3 usages
    private static Scene scene;

    @Override
    public void start(Stage stage) throws IOException {
        scene = new Scene(LoadFXML("primary"), width: 640, height: 480);
        stage.setScene(scene);
        stage.show();
    }

    2 usages
    static void setRoot(String fxml) throws IOException {
        scene.setRoot(LoadFXML(fxml));
    }

    2 usages
    private static Parent loadFXML(String fxml) throws IOException {
        FXMLLoader fxmlLoader = new FXMLLoader(App.class.getResource( name: fxml + ".fxml"));
        return fxmlLoader.load();
    }

    no usages
    public static void main(String[] args) { launch(); }
}
```

Die **primary.xml** Datei:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Button?>
<?import javafx.geometry.Insets?>

<VBox alignment="CENTER" spacing="20.0" xmlns="http://javafx.com/javafx/8.0.171"
```

```
xmlns:fx="http://javafx.com/fxml/1" fx:controller="org.example.PrimaryController">
  <children>
    <Label text="Primary View" />
    <Button fx:id="primaryButton" text="Switch to Secondary View"
onAction="#switchToSecondary"/>
  </children>
  <padding>
    <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
  </padding>
</VBox>
```

Okay, hier ist der ergänzte Abschnitt, der beschreibt, wie man ein JavaFX-Projekt mit Maven *ohne* Archetype erstellt und wie man IntelliJ für das direkte Ausführen und Debuggen mittels VM-Parametern konfiguriert.

vorheriger Text ... Die `primary.xml` Datei:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Button?>
<?import javafx.geometry.Insets?>

<VBox alignment="CENTER" spacing="20.0" xmlns="http://javafx.com/javafx/8.0.171"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="org.example.PrimaryController">
  <children>
    <Label text="Primary View" />
    <Button fx:id="primaryButton" text="Switch to Secondary View"
onAction="#switchToSecondary"/>
  </children>
  <padding>
    <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
  </padding>
</VBox>
```

1.6. Erstellen einer JavaFX - Anwendung mit Maven OHNE Archetype

Während Archetypen den Einstieg erleichtern, möchten erfahrene Entwickler oft die volle Kontrolle über die Projektstruktur und Abhängigkeiten haben oder einfach verstehen, wie ein JavaFX-Maven-Projekt von Grund auf aufgebaut ist. Dieses Vorgehen erfordert die manuelle Konfiguration der `pom.xml`.

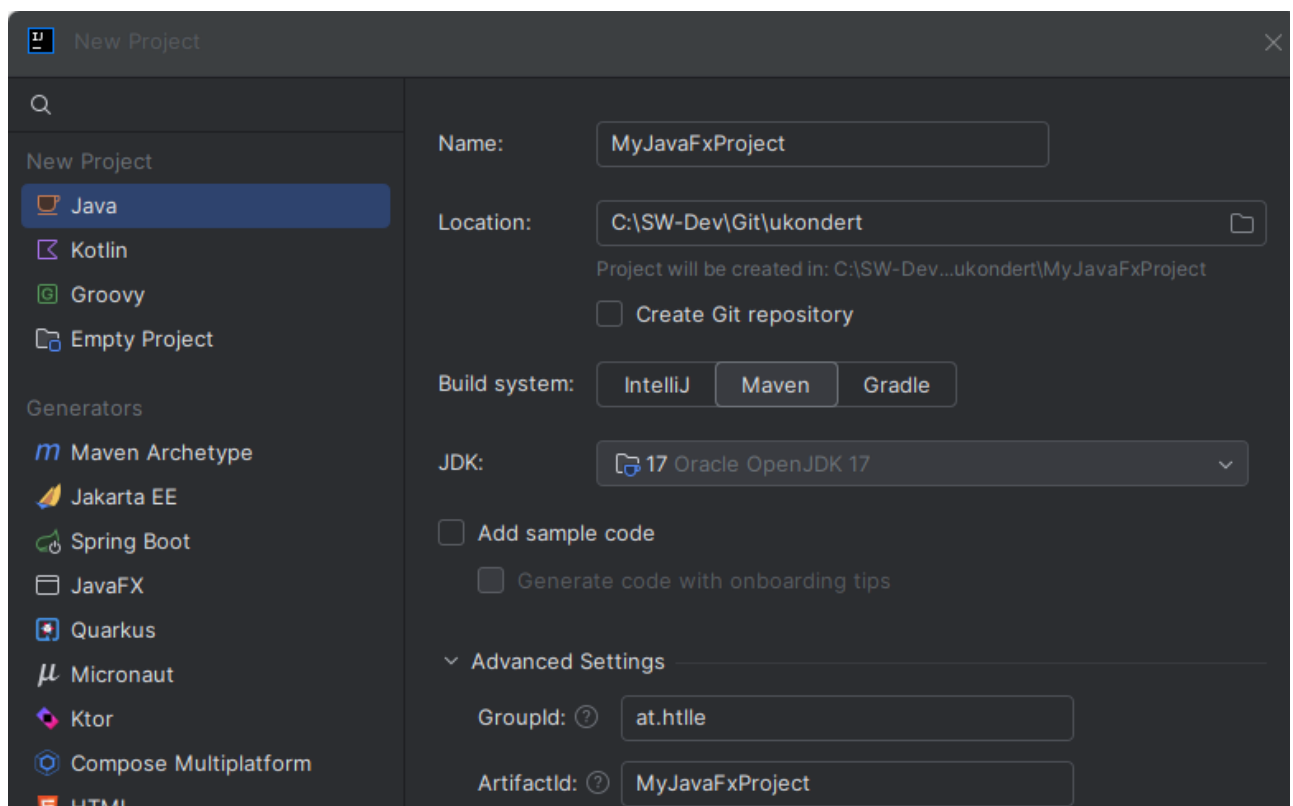
1.6.1. Voraussetzungen

- **JDK:** Version 11 oder höher. Stellen Sie sicher, dass Ihr `JAVA_HOME` korrekt gesetzt ist und Ihre IDE dieses JDK verwendet.
- **Maven:** Eine aktuelle Maven-Installation.

- **JavaFX SDK:** Laden Sie das JavaFX SDK (nicht das JMOD-Archiv) für Ihr Betriebssystem und Ihre JDK-Version von der [Gluon Website](#) herunter. Entpacken Sie dieses an einen bekannten Ort (z.B. `C:\javafx-sdk-17` oder `/opt/javafx-sdk-17`). Sie benötigen den Pfad zum `lib`-Verzeichnis dieses SDKs später für die VM-Optionen in IntelliJ. **Wichtig:** Die Version des heruntergeladenen SDK sollte mit der Version übereinstimmen, die Sie in Ihrer `pom.xml` deklarieren.

1.6.2. Erstellen des leeren Maven Projekts in IntelliJ

1. Öffnen Sie IntelliJ IDEA.
2. Wählen Sie File -> **New Project**.
3. Wählen Sie im linken Bereich **Maven**.
4. Stellen Sie sicher, dass die Option **Create from archetype NICHT** ausgewählt ist.
5. Wählen Sie das korrekte **JDK** (Version 11 oder höher).
6. Klicken Sie auf **Next**



7. Geben Sie einen **Namen** für Ihr Projekt ein (z.B. `javafx-manual-setup`).
8. Passen Sie ggf. die **Location** an.
9. Erweitern Sie **Artifact Coordinates** und geben Sie eine **GroupId** (z.B. `at.htlle.pos3.javafx`) und eine **ArtifactId** (z.B. `manualapp`) ein. Die ArtifactId wird oft als Modulname verwendet.
10. Klicken Sie auf **Create**.

IntelliJ erstellt nun ein einfaches Maven-Projekt mit einer Standard `pom.xml` und einer einfachen Verzeichnisstruktur (`src/main/java`, `src/test/java`).

1.6.3. Anpassen der `pom.xml`

Öffnen Sie die generierte `pom.xml`-Datei und passen Sie sie an, um die JavaFX-Abhängigkeiten und das notwendige Maven-Plugin hinzuzufügen:


```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>at.htlle.pos3.javafx</groupId>
  <artifactId>manualapp</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source> <!-- Oder höher -->
    <maven.compiler.target>11</maven.compiler.target> <!-- Oder höher -->
    <!-- Wählen Sie hier die gewünschte JavaFX Version -->
    <javafx.version>17.0.2</javafx.version>
    <!-- Definieren Sie Ihre Hauptklasse -->
    <javafx.main.class>at.htlle.pos3.javafx.manualapp.App</javafx.main.class>
  </properties>

  <dependencies>
    <!-- JavaFX Controls Modul -->
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-controls</artifactId>
      <version>${javafx.version}</version>
    </dependency>

    <!-- JavaFX FXML Modul (optional, aber oft benötigt) -->
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-fxml</artifactId>
      <version>${javafx.version}</version>
    </dependency>

    <!-- Fügen Sie hier bei Bedarf weitere JavaFX Module hinzu (z.B. javafx-
web, javafx-media) -->

  </dependencies>

  <build>
    <plugins>
      <!-- Maven Compiler Plugin: Stellt sicher, dass die richtige Java-
Version verwendet wird -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version> <!-- Oder eine neuere Version -->
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

```

        </plugin>

        <!-- JavaFX Maven Plugin: Für das Kompilieren und Ausführen von JavaFX
Anwendungen -->
        <plugin>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-maven-plugin</artifactId>
            <version>0.0.8</version> <!-- Oder eine neuere Version -->
            <configuration>
                <!-- Hier wird die Hauptklasse für das Plugin definiert -->
                <mainClass>${javafx.main.class}</mainClass>
            </configuration>
            <executions>
                <!-- Optional: Bindet das Plugin an Lifecycle-Phasen -->
                <execution>
                    <!-- Default configuration for running with: mvn clean
javafx:run -->
                    <id>default-cli</id>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>

```

Wichtige Punkte in der `pom.xml`:

1. Properties:

- `maven.compiler.source/target`: Legt die Java-Version für die Kompilierung fest (hier 11).
- `javafx.version`: Definiert zentral die zu verwendende JavaFX-Version. **Stellen Sie sicher, dass diese Version mit dem heruntergeladenen JavaFX SDK übereinstimmt!**
- `javafx.main.class`: Der vollqualifizierte Name Ihrer Hauptklasse (die von `javafx.application.Application` erbt). Passen Sie dies an Ihren Paketnamen und Klassennamen an.

2. Dependencies:

- Mindestens `javafx-controls` wird benötigt.
- Fügen Sie `javafx-fxml` hinzu, wenn Sie FXML verwenden möchten.
- Fügen Sie weitere JavaFX-Module (`javafx-graphics`, `javafx-base` sind transitiv oft schon dabei; `javafx-media`, `javafx-web` etc.) nach Bedarf hinzu.

3. Build Plugins:

- `maven-compiler-plugin`: Stellt sicher, dass der Code mit der korrekten Java-Version kompiliert wird.
- `javafx-maven-plugin`: Ermöglicht das Ausführen der Anwendung über Maven (`mvn javafx:run`) und hilft beim Erstellen von Paketen. Die `mainClass` wird hier konfiguriert.

Nachdem Sie die `pom.xml` gespeichert haben, wird IntelliJ Sie wahrscheinlich auffordern, die Maven-Änderungen zu laden (ein kleines Popup unten rechts oder ein Maven-Symbol oben rechts im Editor). Bestätigen Sie dies.

1.6.4. Erstellen der Java-Hauptklasse

Erstellen Sie nun die Java-Klasse, die Sie in der `pom.xml` als `javafx.main.class` definiert haben.

1. Navigieren Sie im Project Explorer zu `src/main/java`.
2. Erstellen Sie die notwendige Paketstruktur (z.B. `at.htlle.pos3.javafx.manualapp`).
3. Erstellen Sie in diesem Paket eine neue Java-Klasse (z.B. `App.java`).

Fügen Sie den folgenden Beispielcode ein:

```
package at.htlle.pos3.javafx.manualapp; // Passen Sie dies an Ihren Paketnamen an!

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

/**
 * Eine einfache JavaFX Anwendung (manuell konfiguriert).
 */
public class App extends Application {

    @Override
    public void start(Stage primaryStage) {
        // Ein einfaches Label erstellen
        Label label = new Label("Hallo, JavaFX (manuell konfiguriert)!");

        // Ein Layout-Container (StackPane legt Elemente übereinander)
        StackPane root = new StackPane();
        root.getChildren().add(label);

        // Eine Szene erstellen und das Layout hinzufügen
        Scene scene = new Scene(root, 400, 300); // Breite: 400px, Höhe: 300px

        // Den Titel des Fensters setzen
        primaryStage.setTitle("Manuelle JavaFX App");
        // Die Szene dem Fenster (Stage) zuweisen
        primaryStage.setScene(scene);
        // Das Fenster anzeigen
        primaryStage.show();
    }

    public static void main(String[] args) {
        // Startet die JavaFX Anwendung
        launch(args);
    }
}
```

1.6.5. Ausführen und Debuggen in IntelliJ (ohne Maven Goal)

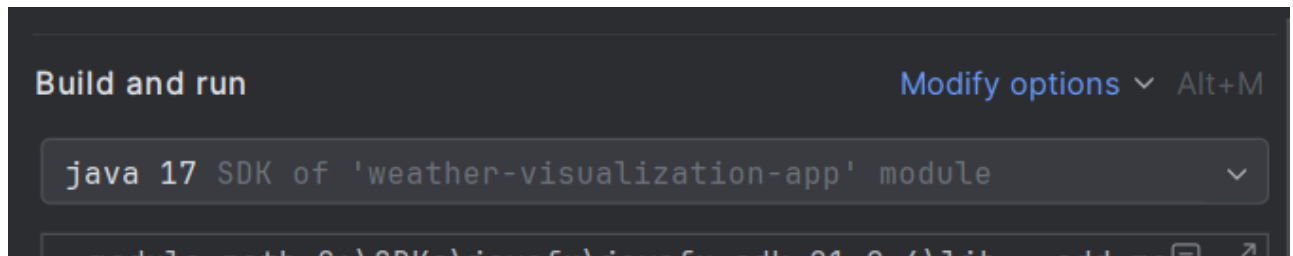
Standardmäßig kann IntelliJ die `main`-Methode einer JavaFX-Anwendung seit Java 11 nicht mehr direkt ausführen, da die JavaFX-Module nicht automatisch im Modulpfad enthalten sind. Das `javafx-maven-`

`plugin` löst dies, wenn Sie `mvn javafx:run` verwenden, aber für das direkte Starten/Debuggen aus der IDE sind zusätzliche Schritte nötig.

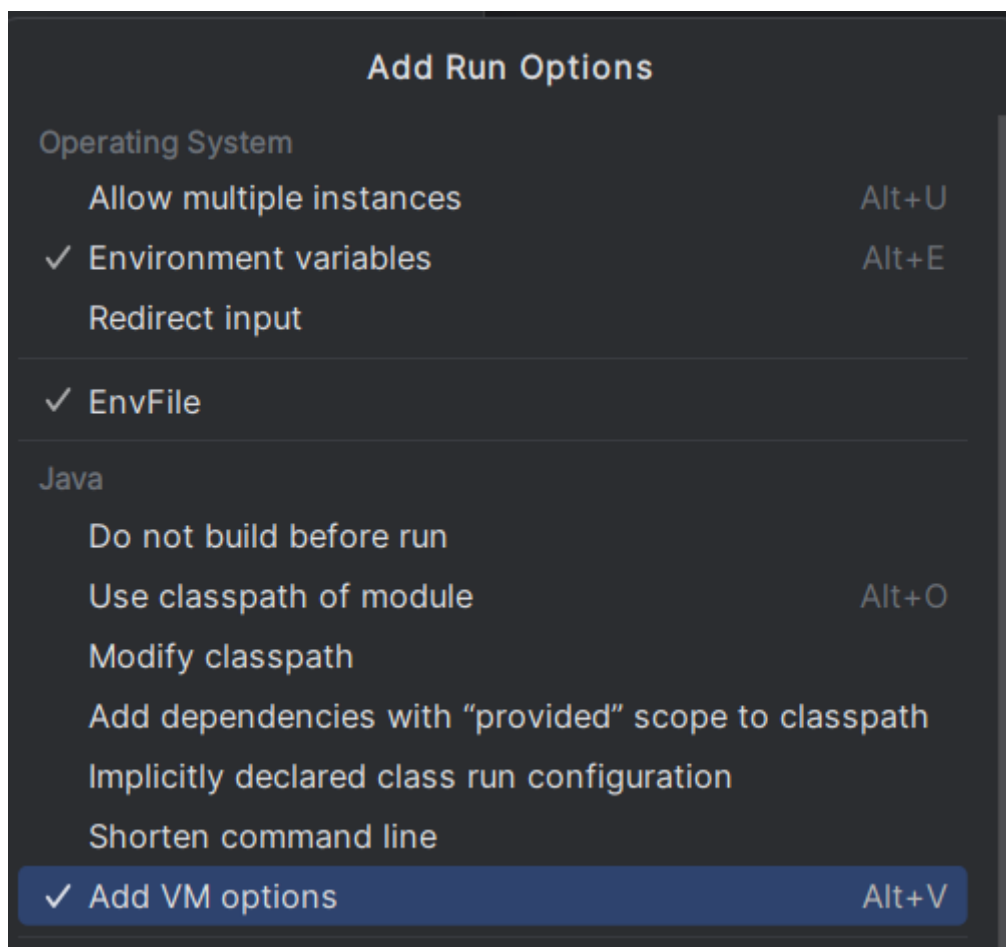
1. Gehen Sie zu **Run** -> **Edit Configurations...**
2. Klicken Sie auf das **+** Symbol oben links und wählen Sie **Application**.
3. Geben Sie der Konfiguration einen Namen, z.B. `Run JavaFX App`.
4. Wählen Sie bei **Main class**: Ihre Hauptklasse aus (z.B. `at.htlle.pos3.javafx.manualapp.App`).

Konfigurieren der Run Configuration mit VM-Optionen

1. Klicken Sie unter **Build and run** - `Modify options`



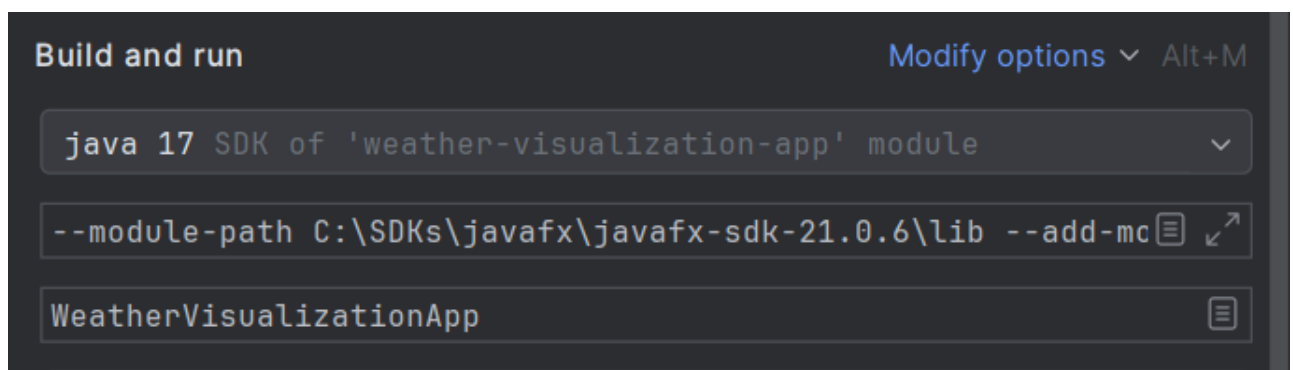
2. wählen sie aus der Drop-Down Box `Add VM-Opstion` aus



3. in dem neu hinzugefügten Feld fügen sie folgendes hinzu:

```
--module-path /pfad/zum/javafx-sdk-XX.Y.Z/lib --add-modules  
javafx.base,javafx.controls,javafx.fxml
```

1. Ersetzen Sie `/pfad/zum/javafx-sdk-XX.Y.Z/lib` durch den **tatsächlichen, vollständigen Pfad** zum `lib`-Verzeichnis des JavaFX SDK, das Sie zuvor heruntergeladen haben (z.B. `C:/Program Files/Java/javafx-sdk-17.0.2/lib` oder `/Users/ihrname/javafx-sdk-17.0.2/lib`). **Achten Sie darauf, dass die Version XX.Y.Z mit der in Ihrer `pom.xml` übereinstimmt!**
2. `--add-modules`: Listet die benötigten JavaFX-Module auf. Mindestens `javafx.base`, `javafx.controls` ist fast immer erforderlich. Fügen Sie `javafx.fxml` hinzu, wenn Ihre Anwendung FXML verwendet. Trennen Sie mehrere Module durch Kommas (ohne Leerzeichen).
4. Stellen Sie sicher, dass das korrekte **JRE** ausgewählt ist (das gleiche JDK 11+, das Sie für das Projekt verwenden).
5. Klicken Sie auf **Apply** und dann auf **OK**.



(*WeatherVisualizateionApp* ist hier die Main Start-App)

Nun können Sie diese Konfiguration (**Run App (Direct)**) auswählen und Ihre Anwendung direkt in IntelliJ starten (grüner Pfeil) oder debuggen (Käfer-Symbol). IntelliJ wird nun die JVM mit den angegebenen Optionen starten, wodurch die JavaFX-Module gefunden und geladen werden können.

Mit dieser manuellen Konfiguration haben Sie ein voll funktionsfähiges JavaFX-Maven-Projekt erstellt und können es bequem direkt in IntelliJ entwickeln, ausführen und debuggen, ohne auf Maven-Goals für jeden Start angewiesen zu sein.

2. JavaFX-Klassen und Komponenten

2.1. Die beiden Haupt Klassen `Stage` und `Scene`

2.1.1. Die `State` Klasse

1. Die `Stage`-Klasse repräsentiert ein Fenster in einer JavaFX-Anwendung.
2. Eine JavaFX-Anwendung muss mindestens eine Stage haben, die vom Application-Thread erstellt werden muss.
3. Eine Stage enthält eine oder mehrere **Szenen** (Scenes), die als Container für alle **Nodes** (Knoten) fungieren, die in der Anwendung angezeigt werden.
4. Die **Größe** und **Position** einer Stage können festgelegt werden, ebenso wie das **Verhalten** beim **Schließen** und **Minimieren**.

5. Die Stage-Klasse hat eine Methode `show()`, die das Fenster auf dem Bildschirm anzeigt, und eine Methode `hide()`, die das Fenster ausblendet.
6. Eine Stage kann auch mit **verschiedenen Styles** dekoriert werden, wie z.B. mit einem **Standard-Window-Style**, einem **transparenten Style** oder einem benutzerdefinierten Style.
7. Die Stage-Klasse bietet auch die Möglichkeit, das Fenster in den **Vollbildmodus** oder den **modalen Dialogmodus** zu versetzen.
8. Eine Stage kann auch über verschiedene Methoden wie `setTitle()`, `setIconified()`, `setResizable()` und `setFullScreen()` konfiguriert werden.

Die `Stage`-Klasse ist somit ein zentrales Element in einer JavaFX-Anwendung und bietet eine Vielzahl von Konfigurationsmöglichkeiten, um das Aussehen und Verhalten des Fensters anzupassen.

2.1.2. Die `Scene` Klasse

1. Die Scene-Klasse ist ein Container für alle visuellen Elemente (Nodes) in einer JavaFX-Anwendung.
2. Eine `Scene` wird **innerhalb einer Stage** angezeigt.
3. Die **Größe** und **Position** einer Scene können festgelegt werden, ebenso wie das Verhalten bei Größenänderungen und das **CSS-Styling**.
4. Eine `Scene` besteht aus einem **Root-Node**, der alle anderen Nodes enthält. Der Root-Node kann ein beliebiger JavaFX-Node sein, wie z.B. ein `Pane`, `StackPane`, `BorderPane`, ein `Group` udgl.
5. Die Nodes innerhalb einer Scene können mit **CSS** gestaltet werden.
6. Eine Scene kann auch mit **verschiedenen Effekten** wie Schatten, Beleuchtung oder Blendeffekten versehen werden.
7. Die Scene-Klasse bietet auch die Möglichkeit, **Maus- und Tastatureingaben** sowie **Drag & Drop**-Operationen zu verarbeiten.
8. Eine Scene kann auch mit verschiedenen **Fokus- und Cursor-Einstellungen** konfiguriert werden.
9. Die Scene-Klasse hat eine Methode `getRoot()`, die den Root-Node der Scene zurückgibt, und eine Methode `setRoot()`, um den Root-Node zu ändern.

Die Scene-Klasse ist somit ein wichtiger Bestandteil einer JavaFX-Anwendung, da sie alle visuellen Elemente enthält und die Möglichkeit bietet, diese mit CSS zu gestalten und Ereignisse zu verarbeiten.

2.1.3. Das Scene - Graph Modell

Das **Scene-Graph-Modell** ist ein wichtiger Bestandteil der JavaFX-Architektur und beschreibt die **hierarchische Struktur aller visuellen Elemente (Nodes)** einer Anwendung. Es handelt sich dabei um eine **Baumstruktur**, in der jeder Knoten (Node) als Kindknoten einen oder mehrere weitere Knoten (Nodes) besitzen kann.

Das Modell beginnt immer mit einem Root-Node, der alle anderen Knoten enthält und der Scene-Klasse zugewiesen wird. Jeder **Knoten** kann **verschiedene Eigenschaften** besitzen, wie z.B. Größe, Position, Transformationen, Effekte und Stile.

Die `Node`-Klasse ist die **Basisklasse für alle visuellen Elemente** in JavaFX und definiert die grundlegenden Methoden und Eigenschaften für alle Knoten. Die wichtigsten Eigenschaften/Methoden sind:

1. `id`: Eine eindeutige ID, die verwendet werden kann, um den Knoten zu identifizieren.
2. `styleClass`: Eine Liste von CSS-Klassen, die auf den Knoten angewendet werden können.
3. `style`: Ein CSS-Style-String, der auf den Knoten angewendet wird.

4. **visible**: Ein Boolean, der angibt, ob der Knoten sichtbar ist oder nicht.
5. **opacity**: Die Transparenz des Knotens.
6. **layoutX** und **layoutY**: Die Position des Knotens innerhalb seines übergeordneten Knotens.

Die **Node**-Klasse hat auch verschiedene Methoden, die verwendet werden können, um die Knoten im Baum zu durchlaufen und zu manipulieren, z.B. `getParent()`, `getChildren()`, `setTranslateX()`, `setTranslateY()` und viele weitere.

Insgesamt bietet das Scene-Graph-Modell eine leistungsstarke und flexible Möglichkeit, um visuelle Elemente zu erstellen und zu verwalten, und ist ein wichtiger Bestandteil der JavaFX-Plattform.

2.2. Control-Klassen (z.B. Button, Label, Textfield)

In JavaFX gibt es viele verschiedene Control-Klassen, die für die Erstellung von Benutzeroberflächen verwendet werden können. Hier sind einige der häufigsten Control-Klassen und ihre Verwendung:

1. **Label**: Wird verwendet, um Text anzuzeigen, der nicht bearbeitet werden kann.
2. **Button**: Wird verwendet, um Aktionen auszulösen, wenn der Benutzer darauf klickt.
3. **TextField**: Wird verwendet, um Benutzereingaben in Form von Text zu akzeptieren.
4. **TextArea**: Wird verwendet, um größere Mengen von Text als Benutzereingabe zu akzeptieren.
5. **ComboBox**: Wird verwendet, um eine Liste von Optionen anzuzeigen, aus der der Benutzer eine auswählen kann
6. **CheckBox**: Wird verwendet, um eine Ja/Nein-Entscheidung des Benutzers anzuzeigen und auszuwählen.
7. **RadioButton**: Wird verwendet, um eine Auswahl aus mehreren Optionen zu ermöglichen.
8. **ToggleButton**: Wird verwendet, um eine Option zu aktivieren/deaktivieren oder zwischen zwei Optionen hin- und herzuschalten.
9. **ListView**: Wird verwendet, um eine Liste von Elementen anzuzeigen, aus der der Benutzer eine oder mehrere auswählen kann.
10. **TableView**: Wird verwendet, um Daten in einer tabellarischen Darstellung anzuzeigen.
11. **WebView**: Wird verwendet, um Webinhalte wie HTML-Seiten anzuzeigen.

Jede Control-Klasse verfügt über verschiedene Eigenschaften und Methoden, mit denen Sie das Aussehen und Verhalten der Komponente konfigurieren können. Hier ist ein Beispiel für die Erstellung eines Labels und eines Buttons:

```
...
Label label = new Label("Dies ist ein Label"); // Erstellen eines Labels
Button btn = new Button(); // Erstellen eines Buttons
btn.setText("Drücke mich"); // Text des Buttons setzen

// Aktion festlegen, die bei Klick ausgeführt wird
btn.setOnAction(event -> {
    label.setText("Button wurde gedrückt!"); // Text des Labels ändern
});
...
```

2.3. Layout-Klassen (z.B. BorderPane, VBox, HBox)

In JavaFX gibt es verschiedene Layoutklassen, die verwendet werden können, um die Anordnung der Steuerelemente in einer GUI zu definieren. Hier sind einige der häufigsten Layoutklassen und ihre Verwendung:

1. **BorderPane**: Ordnet Steuerelemente an den Rändern des Fensters an (oben, unten, links, rechts) und ein zentrales Steuerelement.
2. **FlowPane**: Ordnet Steuerelemente horizontal oder vertikal an, wobei die Steuerelemente in einer Zeile umgebrochen werden, wenn der verfügbare Platz ausgeht.
3. **GridPane**: Ordnet Steuerelemente in einer tabellarischen Anordnung an.
4. **HBox**: Ordnet Steuerelemente horizontal an.
5. **VBox**: Ordnet Steuerelemente vertikal an.
6. **StackPane**: Stapelt Steuerelemente aufeinander und zeigt immer nur das oberste Steuerelement an.
7. **AnchorPane**: Ordnet Steuerelemente relativ zu einem Ankerpunkt an.
8. **ScrollPane**: Stellt einen Bereich zur Verfügung, in dem der Benutzer durch das Scrollen durch einen größeren Bereich navigieren kann.

Jede Layoutklasse verfügt über verschiedene Eigenschaften und Methoden, mit denen Sie das Aussehen und Verhalten der Anordnung konfigurieren können.

2.3.1. Tips zum Desig von Grafischen Oberflächen

1. Beginnen Sie mit einem groben Layout: Überlegen Sie sich, wie die Hauptbereiche der GUI angeordnet sein sollen. Wählen Sie eine Layoutklasse aus, die am besten zu Ihrer Anforderung passt.
2. Verwenden Sie Container-Layouts: Verwenden Sie Layoutklassen wie **BorderPane**, **HBox**, **VBox** oder **GridPane**, um mehrere Steuerelemente in einem Container zusammenzufassen und ihre Gruppierung und Positionierung zu vereinfachen.
3. Verwenden Sie Constraints und Bindings: Verwenden Sie Layout-Constraints, um die Positionierung von Steuerelementen innerhalb eines Containers genau zu definieren. Bindings können verwendet werden, um die Größe und Position von Steuerelementen automatisch an die Größe des Containers anzupassen.
4. Vermeiden Sie überladene Layouts: Versuchen Sie, eine einfache und übersichtliche Layout-Hierarchie zu erstellen. Vermeiden Sie überladene Layouts, die schwierig zu warten und zu aktualisieren sind.
5. Verwenden Sie visuelle Hilfsmittel: Verwenden Sie grafische Design-Tools (z.B. **SceneBuilder**, **Draw.io**) oder Skizzen, um ein visuelles Layout-Modell der GUI zu erstellen. Dies kann bei der Planung und Implementierung der GUI sehr hilfreich sein.

Beispiel zur Verwendung einer **HBox** in Kombination mit einer **VBox** und Layout Constraints (Padding, Alignment):

```
...
public void start(Stage primaryStage) throws Exception {
    // Erstellung der Labels und TextFields
    Label nameLabel = new Label("Name:");
    TextField nameTextField = new TextField();
    Label ageLabel = new Label("Age:");
    TextField ageTextField = new TextField();

    // Erstellung der Buttons
    Button submitButton = new Button("Submit");
    Button clearButton = new Button("Clear");
}
```

```
// Erstellung der HBox und Einfügen der Labels und TextFields
HBox hbox = new HBox();
hbox.setSpacing(10);
hbox.setPadding(new Insets(10, 10, 10, 10));
hbox.getChildren().addAll(nameLabel, nameTextField, ageLabel,
ageTextField);

// Erstellung der VBox und Einfügen der HBox und der Buttons
VBox vbox = new VBox();

vbox.setSpacing(10);
vbox.setPadding(new Insets(10, 10, 10, 10));
HBox hboxButton = new HBox();
// Zentrieren der HBox
hboxButton.setAlignment(Pos.CENTER);
hboxButton.setSpacing(10.0);
hboxButton.getChildren().addAll(submitButton, clearButton);
vbox.getChildren().addAll(hbox, hboxButton);

// Erstellung der Scene und Anzeige des Hauptfensters
// Reduzierung der Größe der Scene auf die Minimale Größe der VBox (= Root
Node)
Scene scene = new Scene(vbox, vbox.getMinWidth(), vbox.getMinHeight());
primaryStage.setScene(scene);
primaryStage.show();
}
...
```

Hier das erstellte Layout:



3. Event-Handling in JavaFX

Das Event Handling in JavaFX ermöglicht es, auf Benutzerinteraktionen wie Mausklicks oder Tastatureingaben zu reagieren und entsprechende Aktionen auszuführen. Dabei gibt es verschiedene Arten von Events, die von JavaFX bereitgestellt werden, wie z.B. `MouseEvent`, `KeyEvent` oder `ActionEvent`.

Die grundlegende Idee des Event Handlings in JavaFX besteht darin, dass ein Event von einem `Node` (z.B. einem `Button`) erzeugt wird und an einen `EventHandler` weitergeleitet wird. Der `EventHandler` ist eine Schnittstelle, die eine `"handle()"`-Methode definiert, die das Event entgegennimmt und entsprechende Aktionen ausführt.

Es gibt verschiedene Möglichkeiten, wie man ein Event behandeln kann. Die beiden häufigsten Arten einen EventHandler zu definieren sind:

1. direkt als **anonyme innere Klasse**
2. als **Lambda-Ausdruck**

zu implementieren. Eine andere Möglichkeit ist, einen EventHandler als **separate Klasse** zu definieren und diese als **Listener** (= Observer) für bestimmte Nodes zu **registrieren**.

Um einen EventHandler zu registrieren, kann man die **addEventHandler()**-Methode des Nodes verwenden. Alternativ kann man auch die **setOn()**-Methoden des Nodes verwenden, um direkt eine Methode aufzurufen, wenn ein bestimmtes Event auftritt.

Hier ist ein Beispiel für die Verwendung von **Event Handling** in JavaFX:

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

In diesem Beispiel wird ein Button erstellt und ein EventHandler registriert, der aufgerufen wird, wenn der Button geklickt wird. Der EventHandler gibt einfach eine Nachricht aus, dass der Button geklickt wurde.

Bei Verwendung der **Lambda - Notation** sieht dies wie folgt aus:

```
Button button = new Button("Click me!");
button.setOnAction(event -> {
    System.out.println("Button clicked!");
});
```

3.1. Verwendung einer Event-Handler Klasse

Um analog zu dem gegebenen Lambda-Ausdruck oder anonyme Klasse, eine eigene EventHandler-Klasse in JavaFX zu definieren, musst du man Klasse erstellen, die das Interface **EventHandler<ActionEvent>** implementiert. Hier ist ein einfaches Beispiel, wie du eine solche Klasse erstellen und verwenden kannst:

1. **Definiere die EventHandler-Klasse:** Zuerst definierst du eine neue Klasse, die **EventHandler<ActionEvent>** implementiert. In der Implementierung der **handle**-Methode fügst du den Code ein, der ausgeführt werden soll, wenn der Button geklickt wird.

```
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
```



```
public class MyButtonEventHandler implements EventHandler<ActionEvent> {  
    @Override  
    public void handle(ActionEvent event) {  
        System.out.println("Button clicked!");  
    }  
}
```

2. **Verwende die EventHandler-Klasse:** Nun kann man eine Instanz der eigenen EventHandler-Klasse erstellen und sie dem Button zuweisen.

```
import javafx.application.Application;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.layout.StackPane;  
import javafx.stage.Stage;  
  
public class MyApplication extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        Button button = new Button("Click me!");  
        MyButtonEventHandler buttonHandler = new MyButtonEventHandler();  
        button.setOnAction(buttonHandler);  
  
        StackPane root = new StackPane();  
        root.getChildren().add(button);  
        Scene scene = new Scene(root, 300, 250);  
        primaryStage.setTitle("Hello World!");  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

In diesem Beispiel erstellt die `MyButtonEventHandler`-Klasse eine spezifische Implementierung von `EventHandler`, die bei jedem Klick auf den Button eine Nachricht in der Konsole ausgibt. Dieser Ansatz ist besonders nützlich, wenn die Logik für die Behandlung des Events komplex ist oder wenn du den gleichen `EventHandler` an verschiedenen Stellen in deiner Anwendung wiederverwenden möchtest.

3.2. Identifizierung von Ereignisquellen und Ereignisarten

Die Identifizierung von Ereignisquellen und Ereignisarten in JavaFX erfolgt in der Regel in der Event-Handler Methode. Diese Methode wird aufgerufen, wenn das entsprechende Ereignis ausgelöst wird, z.B. wenn ein Button geklickt wird. Dabei gibt es zwei Schritte, die bei der Identifizierung der Ereignisquelle und Ereignisart helfen können:

1. **Ereignisquelle identifizieren:** In der `Event`-Handler-Methode kann man auf das `Event`-Objekt zugreifen, welches Informationen über die Quelle des Ereignisses enthält. Dies kann zum Beispiel durch die Methode `getSource()` erfolgen.
2. **Ereignisart identifizieren:** Das `Event`-Objekt enthält auch Informationen über die Art des Ereignisses. Die Ereignisart kann durch den Namen der `Event`-Klasse abgeleitet werden, z.B. `MouseEvent` für Mausereignisse oder `ActionEvent` für Aktionen wie Klicken auf einen Button.

3.2.1. Identifizierung von Ereignisquellen

Es gibt viele Situationen, in denen es notwendig ist, Ereignisquellen in der Event-Methode zu identifizieren. Hier sind einige Beispiele:

1. Wenn mehrere Elemente auf der Benutzeroberfläche denselben Typ von Ereignissen auslösen (z.B. Klick auf einen Button), aber unterschiedliche Aktionen ausgelöst werden sollen, je nachdem, welches Element ausgelöst hat.
2. Wenn eine Benutzeroberfläche dynamisch generiert wird und Elemente hinzugefügt oder entfernt werden können. In diesem Fall müssen die Ereignisquellen identifiziert werden, um die richtigen Aktionen auszuführen.
3. Wenn eine Gruppe von Elementen als Einheit behandelt wird, aber dennoch jedes Element individuell identifiziert werden muss, um unterschiedliche Aktionen auszulösen.
4. Wenn es mehrere Instanzen desselben Elements auf der Benutzeroberfläche gibt und jede Instanz individuell identifiziert werden muss, um unterschiedliche Aktionen auszulösen.

Insgesamt ist die Identifizierung von Ereignisquellen in der Event-Methode eine wichtige Technik, um eine robuste und flexible Benutzeroberfläche zu erstellen, die auf Benutzeraktionen reagieren kann.

Hier ein anschauliches Beispiel für die Verwendung der `getSource()` - Methode um die gedrückten Button zu identifizieren :

```
public class ButtonExample extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        VBox vbox = new VBox();
        vbox.setAlignment(Pos.CENTER);
        vbox.setSpacing(10);
        vbox.setPadding(new Insets(10, 10, 10, 10));

        Button button1 = new Button("Button 1");
        Button button2 = new Button("Button 2");
        Button button3 = new Button("Button 3");

        button1.setOnAction(event -> handleButtonAction(event));
        button2.setOnAction(event -> handleButtonAction(event));
        // auch diese Notation ist möglich
        button3.setOnAction(this::handleButtonAction);

        vbox.getChildren().addAll(button1, button2, button3);

        Scene scene = new Scene(vbox, 300, 200);
```

```
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private void handleButtonAction(ActionEvent event) {
        Button button = (Button) event.getSource();
        String buttonName = button.getText();
        System.out.println("Button " + buttonName + " wurde gedrückt.");
    }

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

In diesem Beispiel werden drei Buttons erstellt und jeder Button wird mit der gleichen Event-Methode `handleButtonAction()` verknüpft. Wenn ein Button gedrückt wird, wird die Methode aufgerufen und die Ereignisquelle (der Button) wird identifiziert, indem das `getSource()`-Methode aufgerufen wird. Der Text des Buttons wird dann extrahiert und ausgegeben.

3.2.2. Ereignisart bzw. Typ identifizieren

Hier sind einige Situationen oder Beispiele, in denen es notwendig ist, den Ereignistyp in der Event-Methode zu identifizieren:

1. Unterschiedliche Aktionen bei verschiedenen Ereignissen: Wenn Sie verschiedene Aktionen ausführen möchten, basierend auf dem Ereignistyp, ist es wichtig, den Ereignistyp zu identifizieren. Zum Beispiel möchten Sie bei einem Klick auf einen Button ein Popup-Fenster anzeigen und bei einem Mouse-Over-Ereignis die Hintergrundfarbe ändern.
2. Fehlerbehandlung: Bei der Behandlung von Fehlern oder Ausnahmen kann es hilfreich sein, den Ereignistyp zu identifizieren, um den Fehler genauer zu diagnostizieren und zu beheben.
3. Dynamisches Styling: Sie können das Styling von Elementen dynamisch ändern, basierend auf dem Ereignistyp. Beispielsweise können Sie die Farbe eines Buttons ändern, wenn der Mauszeiger darüber bewegt wird, oder das Styling eines Textfelds ändern, wenn der Fokus darauf gesetzt wird.
4. Filtern von Ereignissen: Manchmal möchten Sie bestimmte Ereignisse ignorieren oder filtern, um unerwünschte Aktionen zu vermeiden. Zum Beispiel können Sie alle Mausklicks auf ein bestimmtes Element ignorieren, oder nur reagieren, wenn bestimmte Tasten auf der Tastatur gedrückt werden.

In der Regel ist es eine gute Praxis, Ereignisse so genau wie möglich zu identifizieren, um sicherzustellen, dass Ihre Anwendung genau das tut, was von ihr erwartet wird, und um unerwünschte oder fehlerhafte Aktionen zu vermeiden.

Hier ist ein Beispiel, das die Hintergrundfarbe eines Textfelds ändert, wenn es den Fokus erhält, und die Farbe zurücksetzt, wenn es den Fokus verliert:

```
TextField textField = new TextField();

// Event-Handler-Methode für den Fokus-Wechsel
```

```
public void handleFocusChange(FocusEvent event) {
    if (event.getEventType() == FocusEvent.FOCUS_GAINED) {
        // Hintergrundfarbe ändern, wenn Textfeld Fokus erhält
        textField.setStyle("-fx-background-color: yellow;");
    } else if (event.getEventType() == FocusEvent.FOCUS_LOST) {
        // Hintergrundfarbe zurücksetzen, wenn Textfeld Fokus verliert
        textField.setStyle("-fx-background-color: white;");
    }
}

// Zuweisung des Event-Handlers zum Textfeld
textField.setOnFocusChanged(event -> handleFocusChange(event));
```

In diesem Beispiel wird die Methode `handleFocusChange` als Event-Handler verwendet und erhält das `FocusEvent` als Parameter. Wenn das Textfeld den Fokus erhält (`FOCUS_GAINED`), wird die Hintergrundfarbe auf gelb gesetzt, und wenn es den Fokus verliert (`FOCUS_LOST`), wird die Hintergrundfarbe auf weiß zurückgesetzt. Der Event-Handler wird dem Textfeld mit `setOnFocusChanged` zugewiesen.

4. Verwendung von FXML

4.1. Was ist FXML

XML ist eine **XML**-basierte Sprache, die es ermöglicht, die Benutzeroberfläche von JavaFX-Anwendungen zu definieren. FXML-Dateien können verwendet werden, um die Struktur und das Layout einer Benutzeroberfläche zu definieren, sowie um die Beziehungen zwischen den einzelnen Komponenten zu beschreiben.

Die Verwendung von FXML hat mehrere Vorteile. Zum einen erleichtert es das Erstellen und Bearbeiten von Benutzeroberflächen, da es eine klare Trennung zwischen der Darstellung und der Logik der Anwendung ermöglicht. Zum anderen ist es einfacher, die Benutzeroberfläche durch Änderungen an der FXML-Datei anzupassen, anstatt den Code der Anwendung zu ändern.

FXML-Dateien werden normalerweise in einem Texteditor erstellt und können dann mit dem Scene Builder-Tool von Oracle oder anderen FXML-Editoren bearbeitet werden. In der Regel wird die FXML-Datei von einem Controller gesteuert, der die Interaktion mit der Benutzeroberfläche übernimmt.

FXML-Dateien können auch verwendet werden, um CSS-Stylesheets und Ressourcen wie Bilder oder Audio-Dateien zu importieren, um das Erscheinungsbild der Benutzeroberfläche anzupassen.

4.2. Grundlegende Struktur von FXML

4.2.1. XML - Grundlagen

XML (Extensible Markup Language) ist eine Auszeichnungssprache zur Strukturierung und Beschreibung von Daten. XML-Dokumente bestehen aus Tags, Attributen und Daten. Tags sind Elemente, die einen bestimmten Bereich markieren. Attribute sind Eigenschaften eines Elements und Daten sind der Inhalt des Elements.

Ein Beispiel für ein einfaches XML-Dokument:

```
<book>
  <title>Harry Potter and the Philosopher's Stone</title>
  <author>J.K. Rowling</author>
  <publisher>Bloomsbury</publisher>
  <year>1997</year>
</book>
```

In diesem Beispiel ist "book" das Hauptelement, das die anderen Elemente "title", "author", "publisher" und "year" enthält.

XML-Dokumente müssen gut geformt sein, d.h. sie müssen korrekt verschachtelt und abgeschlossen sein. Es ist auch möglich, XML-Schemas zu verwenden, um die Struktur und den Inhalt von XML-Dokumenten zu definieren und zu validieren.

FXML ist eine **Erweiterung** von **XML**, die speziell für JavaFX entwickelt wurde. Es ermöglicht die **Definition von Benutzeroberflächen** und deren **Verhalten** in einer XML-Datei, die dann von einer JavaFX-Anwendung geladen und ausgeführt wird.

4.2.2. FXML - Dokument Struktur

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox alignment="CENTER" spacing="20.0" style="-fx-background-color: green;"
xmlns="http://javafx.com/javafx/17.0.2-ea" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="at.htlle.pos3.javafx.PrimaryController">
  <children>
    <Label text="Primary View" />
    <Button fx:id="primaryButton" onAction="#switchToSecondary" text="Switch to
Secondary View" />
  </children>
  <padding>
    <Insets bottom="20.0" left="20.0" right="20.0" top="20.0" />
  </padding>
</VBox>
```

Die ersten sechs Zeilen sind die sogenannte XML-Deklaration, die am Anfang eines XML-Dokuments stehen muss. Sie besteht aus dem XML-Header, der die Version der XML-Syntax angibt, und der Zeichenkodierung (hier UTF-8), in der das Dokument geschrieben wurde:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Die Version "1.0" ist die am häufigsten verwendete XML-Version. Die Kodierung "UTF-8" gibt an, dass das Dokument in der UTF-8-Zeichenkodierung geschrieben wurde. UTF-8 ist eine weit verbreitete Kodierung, die alle Unicode-Zeichen unterstützt.

Nach der XML-Deklaration können weitere spezielle Anweisungen folgen, die mit dem Präfix `<?` beginnen. Diese Anweisungen werden als Verarbeitungsanweisungen bezeichnet. Im Beispiel werden drei Verarbeitungsanweisungen verwendet, die angeben, welche JavaFX-Klassen und -Layouts im Dokument verwendet werden:

```
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
```

Das FXML-Dokument verwendet für das **VBox**-Element und alle Elemente innerhalb einen **XML-Namespace** (`xmlns:fx`), um Elemente und Attribute zu definieren. Der Namespace "`http://javafx.com/javafx/17.0.2-ea`" definiert die JavaFX-Elemente und -Attribute, während der Namespace "`http://javafx.com/fxml/1`" das **fx**-Präfix definiert, das zum Verweisen auf Elemente und Attribute im FXML-Dokument verwendet wird. Der **fx:controller**-Attribut gibt den **vollqualifizierten Namen des Controllers** an, der das FXML-Dokument steuert.

Die **VBox**-Containerkomponente hat drei Attribute:

1. Das **"alignment"**-Attribut legt fest, wie die Kinder in der **vertikalen** Ausrichtung angeordnet werden sollen. Hier ist es auf "CENTER" gesetzt, was bedeutet, dass die Kinder zentriert werden.
2. Das **"spacing"**-Attribut legt den Abstand zwischen den Kindern fest. Hier ist es auf **20.0** gesetzt.
3. Das **"style"**-Attribut legt das Styling der VBox-Komponente fest. Hier ist es auf eine grüne Hintergrundfarbe gesetzt.

Innerhalb der VBox-Komponente gibt es zwei Kinder:

1. Ein **Label** mit dem Text "Primary View".
2. Ein **Button** mit dem Text "Switch to Secondary View" und der ID **"primaryButton"**. Der Button hat auch das Attribut **onAction**, welches das Event beschreibt und dessen Wert festlegt, welche Methode aufgerufen werden soll, wenn der Button gedrückt wird. In diesem Fall wird die Event-Methode **switchToSecondary** aufgerufen, die im zugehörigen Controller definiert ist.

Schließlich hat die VBox-Komponente auch ein **"padding"**-Attribut, das den Abstand zwischen der VBox und ihrem umgebenden Container festlegt.

4.3. Das Zusammenspiel von Controller Klasse und FXML - Datei

FXML-Dateien und Controller in JavaFX interagieren über das **fx:controller**-Attribut in der **Root-Tag**-Definition der FXML-Datei. Das **fx:controller**-Attribut wird verwendet, um dem FXML-Loader zu sagen, welcher Controller für diese Ansicht verwendet werden soll.

Wenn eine FXML-Datei geladen wird, erzeugt der **FXML-Loader** eine Instanz des Controllers und verknüpft diese Instanz mit der FXML-Datei. Der Controller kann dann auf die in der FXML-Datei definierten

Steuerelemente (Nodes) zugreifen. Dabei müssen im Controller alle Node-Elemente, die in der FXML-Datei mit dem Attribut `fx:id` gekennzeichnet, sind deklariert werden.

Damit die definierten Steuerelemente (Nodes), welche mit dem Attribut `fx:id` in der FXML-Datei gekennzeichnet sind im Controller instanziiert werden, müssen die Deklarationen der Node Attribute im Controller mit dem `@FXML`-Annotation-Tag versehen werden.

4.3.1. Definition von Event - Methoden

In der FXML-Datei können Event-Methoden durch das Attribut `on<Action>` definiert werden, wobei `<Action>` für den Ereignistyp steht. Zum Beispiel wird für einen Klick auf einen Button das Attribut `onAction` verwendet.

Hier ist ein Beispiel für eine Button-Element-Definition in der FXML-Datei, die auf die `handleButtonClick`-Methode im Controller verweist:

```
<Button text="Click me" onAction="#handleButtonClick"/>
```

Die `handleButtonClick`-Methode muss im Controller definiert werden und folgendes Format haben:

```
@FXML
public void handleButtonClick(ActionEvent event) {
    // Implementierung der Event-Behandlung
}
```

Das `@FXML`-Annotation kennzeichnet die Methode als eine Methode, die von der FXML-Datei aufgerufen wird. Der Name der Methode muss dem Wert von `on<Action>` in der FXML-Datei entsprechen. Die Signatur der Methode muss den richtigen Ereignis-Typ als Parameter haben.

4.3.2. Beispiel von FXML - Datei und zugehörigem Controller

FXML - Datei

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane fx:id="anchorPane" xmlns="http://javafx.com/javafx/17.0.2-ea"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="com.example.MyController">
    <children>
        <Label text="Hello, World!" />
        <Button text="Click me" onAction="#handleButtonClick" />
    </children>
</AnchorPane>
```

Controller

```
package com.example;

import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;

public class MyController {

    @FXML
    private AnchorPane anchorPane;

    @FXML
    private Label label;

    @FXML
    private Button button;

    @FXML
    private void handleButtonClick() {
        label.setText("Button clicked!");
    }

}
```

In diesem Beispiel wird die `view.fxml`-Datei verwendet, um das Layout der Benutzeroberfläche zu definieren. Der `fx:controller`-Attribut in der `AnchorPane`-Zeile zeigt an, dass diese FXML-Datei mit dem `MyController`-Controller verknüpft ist.

In der Controller-Klasse werden die mit `@FXML` annotierten Felder verwendet, um auf die in der FXML-Datei definierten UI-Elemente zuzugreifen (in diesem Fall `AnchorPane`, `Label` und `Button`). Die `handleButtonClick`-Methode wird als Event-Methode für den Button definiert und ändert den Text des Labels, wenn der Button geklickt wird.

Die FXML-Datei und der Controller werden normalerweise im Hauptprogramm zusammengeführt, indem die FXML-Datei geladen und der Controller initialisiert wird. Hier ist ein Beispiel für das Laden der FXML-Datei und die Initialisierung des Controllers in der `start`-Methode der Hauptklasse:

```
package com.example;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MainApp extends Application {
```



```

@Override
public void start(Stage stage) throws Exception {
    FXMLLoader loader = new FXMLLoader(getClass().getResource("view.fxml"));
    Parent root = loader.load();
    MyController controller = loader.getController();
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.show();
}

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

```

Hier wird der **FXMLLoader** verwendet, um die **view.fxml**-Datei zu laden und den zugehörigen Controller zu initialisieren. Der **FXMLLoader** analysiert die **FXML**-Datei und erzeugt das Layout der Benutzeroberfläche sowie die UI-Elemente, die in der Controller-Klasse verwendet werden können. Der Controller wird dann aus dem **FXMLLoader**-Objekt abgerufen und die **Scene** wird mit dem geladenen Layout erstellt. Schließlich wird das Fenster angezeigt.

In JavaFX wird häufig die **MVC-Architektur** (= **Model View Controller**) verwendet, wobei:

- die **FXML**-Datei: das **View-Element** (= GUI bzw. Benutzeroberfläche) darstellt
- die **Controller**-Klasse: für die **Logik** und **Interaktionen** zuständig ist.

4.3.3. Verknüpfen der Controller Klasse mit der FXML - Datei

Um eine **Controller**-Klasse mit einer **FXML**-Datei zu verbinden, gibt es folgende Schritte:

1. Erstellen Sie eine **Controller-Klasse**, welche die **Initializable**-Schnittstelle implementiert. Diese Schnittstelle enthält die **initialize()**-Methode, die aufgerufen wird, nachdem die **FXML**-Datei geladen wurde. Beispiel:

```

public class MyController implements Initializable {

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        // Initialisierungscode hier
        // dieser wird aufgerufen, nachdem alle mit @FXML annotierten
        // Variablen initialisiert wurden
    }

}

```

2. In der **FXML**-Datei müssen Sie das Attribut **fx:controller** im Root-Element der Datei auf die vollständig qualifizierte Klassenbezeichnung der Controller-Klasse setzen. Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.StackPane?>
<StackPane xmlns="http://javafx.com/javafx/17.0.2-ea"
xmlns:fx="http://javafx.com/fxml/1" fx:controller="mypackage.MyController">
    <Button fx:id="myButton" text="Click me!" onAction="#handleButtonAction" />
</StackPane>
```

3. Verknüpfen Sie FXML - Elemente mit Attribute und Methoden in der Controller - Klasse mit Hilfe von @FXML - Annotationen:

1. Verwenden Sie das Attribut `fx:id` in den FXML-Elementen, die Sie mit der Controller-Klasse verbinden möchten. Diese Elemente werden dann als Instanzvariablen in der Controller-Klasse erstellt, und Sie können auf sie zugreifen (hier `myButton`).
2. Mit Hilfe von `on[EventArt]` (z.B. `onAction`) Attributen können Methoden als **Event-Methoden** (hier `#handleButtonAction`) definiert werden, die in der Controller Klasse mit Hilfe der `@FXML` - Annotation versehen und implementiert werden. Beispiel:

```
public class MyController implements Initializable {

    @FXML
    private Button myButton;

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        // Initialisierungscode hier
    }

    @FXML
    private void handleButtonAction(ActionEvent event) {
        // Event-Handling-Code hier
        myButton.setText("Button clicked!");
    }
}
```

In diesem Beispiel wird das Button-Element in der FXML-Datei mit der Instanzvariable `myButton` in der Controller-Klasse verknüpft. In der `handleButtonAction()`-Methode wird der Text des Buttons geändert, wenn er geklickt wird.

1. Laden Sie die FXML-Datei in Ihrer JavaFX-Anwendung und verknüpfen Sie sie mit der Controller-Klasse. Beispiel:

```
public class MyApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        // Laden der FXML-Datei
    }
}
```

```

        FXMLLoader loader = new
FXMLLoader(getClass().getResource("MyScene.fxml"));
        Parent root = loader.load();

        // Verknüpfung mit der Controller-Klasse
        MyController controller = loader.getController();

        // Anzeigen der Szene
        Scene scene = new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

```

4.3.4. Laden der FXML - Datei und Instanziierung der Controllerklasse.

1. **Speicherort der FXML - Datei:** Diese Datei (z.B. `MyScene.fxml`) sollte sich im gleichen Verzeichnis wie die Controller-Klasse (z.B. `MyController.java`) befinden, damit sie leicht gefunden werden kann. In **Maven** - Projekten wird diese unter dem Resource Verzeichnis `main/resources/` angelegt. (siehe unten)

```

my-javafx-project
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   ├── com
    │   │   │   └── example
    │   │       ├── Main.java
    │   │       └── MyController.java
    │   └── resources
    │       ├── com
    │       │   └── example
    │       │       └── MyScene.fxml
    │       └── application.css

```

2. **Controller-Klasse angeben:** In der FXML-Datei (z.B. `MyScene.fxml`) wird die Controller-Klasse mit samt dem Package-Pfad (z.B. `com.example.MyController`) mithilfe des `fx:controller` Attributs angegeben, das im Wurzelement der FXML-Datei gesetzt wird:

```

<VBox fx:id="root" xmlns:fx="http://javafx.com/fxml"
fx:controller="com.example.MyController">
    <!-- Benutzeroberflächenelemente hier -->
</VBox>

```

3. **FXML-Datei laden:** Um die FXML-Datei in Ihrer Anwendung zu laden, verwenden Sie die **FXMLLoader**-Klasse, wie im folgenden Beispiel (Mit `getClass().getResource()` wir auf das Verzeichnis der Klasse verwiesen, in der dieser Aufruf erfolgt - hier *Main*):

```
FXMLLoader loader = new FXMLLoader(getClass().getResource("MyScene.fxml"));
```

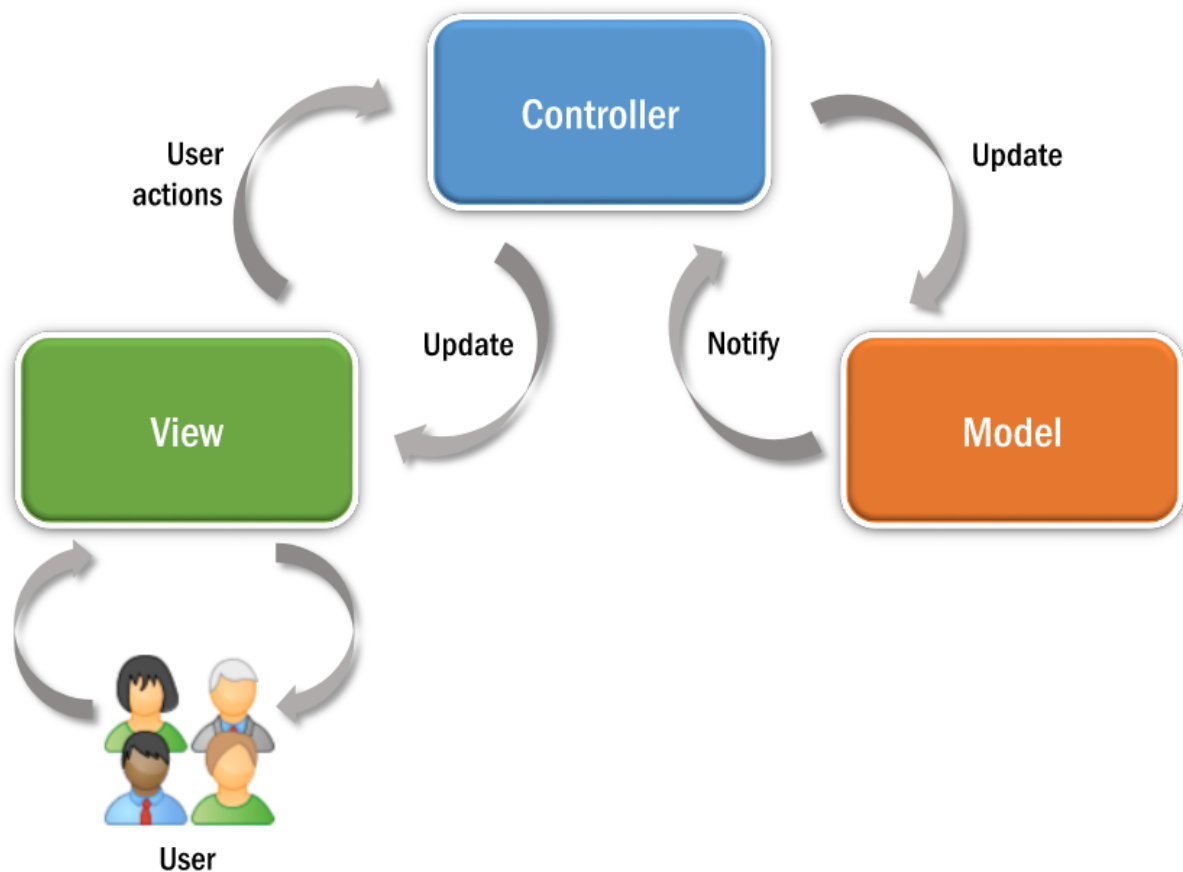
4. **Erstellen des Wurzel - Elements (= root):** Die `load()`-Methode erstellt das Wurzelement der Benutzeroberfläche und instanziert die Controller-Klasse, die der FXML-Datei zugeordnet ist.

```
Parent root = loader.load();
```

5. **Aufrufen der Controller - Instanz** der FXML - Datei: Nachdem die FXML-Datei geladen wurde und das Wurzelement sowie die Controller-Klasse erstellt wurden, können Sie die Instanz der Controller-Klasse abrufen, indem Sie die `getController()`-Methode des FXMLLoader aufrufen:

```
MyController controller = loader.getController();
```

5. Das MVC-Patterns (Model-View-Controller) in JavaFX



5.1. Was ist das MVC - Pattern

Das MVC-Pattern (Model-View-Controller) ist ein Design-Pattern, das häufig in der Softwareentwicklung angewendet wird, um eine klare Trennung von Aufgaben und Verantwortlichkeiten innerhalb einer Anwendung zu ermöglichen. Es wird oft verwendet, um die Benutzeroberfläche (View) von der Geschäftslogik (Model) zu trennen, um die Wartbarkeit, Erweiterbarkeit und Testbarkeit der Anwendung zu verbessern.

Das Pattern besteht aus drei Kernkomponenten:

1. **Model:** Das Model ist die Komponente, die die Geschäftslogik und die Daten enthält. Es ist verantwortlich für das Verwalten und Aktualisieren der Daten und bietet Methoden zum Lesen, Schreiben und Validieren von Daten an.
2. **View:** Die View ist die Benutzeroberfläche der Anwendung. Sie zeigt die Daten an, die vom Model bereitgestellt werden, und bietet eine Schnittstelle zur Interaktion mit dem Benutzer.
3. **Controller:** Der Controller ist die Komponente, die zwischen dem Model und der View vermittelt. Er empfängt Benutzeraktionen von der View, aktualisiert das Model entsprechend und aktualisiert die View, um die Änderungen anzuzeigen.

Durch die Trennung von Model, View und Controller wird die Anwendung modular und leichter zu warten und zu erweitern. Änderungen an einem Teil des Patterns haben keinen Einfluss auf die anderen Teile, solange die Schnittstellen zwischen ihnen unverändert bleiben.

5.2. Umsetzung des MVC-Patterns in JavaFX

Das Zusammenspiel zwischen Model und View wird in JavaFX mithilfe des Controllers umgesetzt. Der Controller ist für die Verbindung zwischen Model und View verantwortlich und enthält die Geschäftslogik der Anwendung.

5.2.1. Das **Model**

Die Aufgabe des Models besteht darin, Daten zu verwalten, zu validieren und zu transformieren. Es stellt eine **Schnittstelle** für den Zugriff auf Daten bereit und führt Aktionen auf diesen Daten aus.

Das Model ist unabhängig von der View und dem Controller und enthält keine Kenntnis über die Implementierung der anderen Komponenten des Patterns. Es wird lediglich über den Controller aufgerufen und gibt Daten an diesen zurück.

Folgendes Beispiel zeigt das Interface für ein einziges Objekt, die `IllegalArgumentException` soll bei einer fehlgeschlagenen Validierung der Daten geworfen werden.

```
public interface ProductModel {
    String getName();
    void setName(String name) throws IllegalArgumentException;
    int getId();
    void setId(int id) throws IllegalArgumentException;
    double getPrice();
    void setPrice(double price) throws IllegalArgumentException;
    String getDescription();
    void setDescription(String description) throws IllegalArgumentException;

    void save();
    void delete();
}
```

Werden Views verwendet, die **Listen von Objekten** verwenden, so ist es auch die Aufgabe des Models, diese Listen von Objekten zu verwalten. Es ist jedoch ratsam, dass diese Objekte das Model - Interface (hier `ProductModel`) für das einzelne Objekt implementieren.

```
public interface ProductListModel {
    List<ProductModel> getAllProducts();
    ProductModel getProductById(int id);
    void addProduct(ProductModel product);
    void updateProduct(ProductModel product);
    void deleteProduct(ProductModel product);
}
```

Durch die Verwendung einer Schnittstelle für das Model, kann insbesondere bei komplexeren Model - Implementierungen zunächst eine **Dummy Implementierung** des Model - Interface verwendet werden, um bestimmte Funktionalitäten auf der GUI zu testen.

5.2.2. Die **View**

Die View wird mit Hilfe von FXML erstellt, wobei hier für das grafische Layouting Werkzeuge wie z.B. der **SceneBuilder** zur Anwendung kommen.

5.2.2.1. View mit Angabe der konkreten Implementierung des Controller - Interfaces

Wird im FXML-Code der View angegeben, so ist hier die konkrete Implementierung des Controller Interfaces anzugeben. Dies erfolgt im Root - Element der View, durch das Attribut **fx:controller** (hier z.B. der **VBox**).

```
<!-- Imports -->
<VBox xmlns="http://javafx.com/javafx/13" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="at.htlle.pos3.javafx.ProductControllerImpl">
    <Label text="Product" />
    <TextField fx:id="nameField" promptText="Name" />
    <TextField fx:id="idField" promptText="ID" />
    <!-- and so on -->
</VBox>
```

Die Erstellung der View erfolgt dann wie bereits erwähnt mit Hilfe der **FXMLLoader** Klasse, indem die FXML - Datei geladen und die darin definierten GUI - Elemente instanziiert werden.

```
FXMLLoader loader = new FXMLLoader(getClass().getResource("MyView.fxml"));
Parent root = loader.load();
```

5.2.3. View OHNE Angabe der konkreten Implementierung des Controller - Interfaces

Wird im FXML - Code der View die Controller - Klasse, bzw. die Implementierung des Controller Interfaces **nicht** angegeben, so muss diese gesetzt werden, **nachdem** eine Instanz der **FXMLLoader** Klasse durch Angabe der FXML-Datei erstellt worden ist.

```
FXMLLoader loader = new FXMLLoader(getClass().getResource("MyView.fxml"));

MyController myController = new MyControllerImpl();

loader.setController(myController);
// Instanzieren der View NACH dem Setzen der Controller Klasse
Parent root = loader.load();
```



ACHTUNG: Das Erstellen der View (durch den Aufruf **loader.load()**) muss unbedingt erst NACH dem Setzen des Controllers (**loader.setController()**) erfolgen!

5.2.4. Der Controller

Auch hier ist es möglich eine **Schnittstelle als Controller** zu definieren. Hierbei kann man sich im Vorfeld darüber einigen, welche Methoden dieser Controller nach aussen hin bereit stellen soll.

```
public interface ProductController {
    public void updateProduct() throws NoValidInputException;
    public void updateView();
    public void initialize(ProductModel product);
    public void initialize(Class<? extends ProductModel> clazz) throws
    ClassNotFoundException, InstantiationException, IllegalAccessException;
}
```

Die **konkrete Implementierung** der Controller Klasse soll eine `initialize` - Methode, die mit `@FXML` annotiert ist, bereitstellen, die als Parameter die Model - Schnittstelle besitzt.

```
public class ProductControllerImpl implements ProductController{

    // mit @FXML annotierte GUI Elemente und Event-Methoden

    @FXML
    public void initialize(ProductModel product) {
        // Initialisierung des Controllers
        this.product = product;
        updateView();
    }
}
```

Somit kann beim Aufruf der Controller Klasse eine Instanz der gewünschten Implementierung der Model - Schnittstelle übergeben werden.

```
public class ProductApplication extends Application {

    @Override
    public void start(Stage stage) throws Exception {
        // some other stuff
        FXMLLoader loader = new
        FXMLLoader(getClass().getResource("product.fxml"));
        VBox root = loader.load();
        ProductController controller = loader.getController();
        controller.initialize(new ProductModelImpl(1, "TestProduct", 20.0, "Das
ist ein Produkt"));
        // Erstellen des Szenengraphen
        Scene scene = new Scene(root);
        // Anzeigen der Szene
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```


Wie man an dem zuvor angeführten Beispiel sieht, wird in der Applikation selbst das **Controller - Interface** (**ProductController**) verwendet.

5.2.4.1. (V) Verwendung von Reflection

Durch die Verwendung von **Reflection** kann zur Laufzeit die Klasse durch Angabe dessen Namens instanziiert werden. Somit wäre es möglich über eine **Konfigurationsdatei**, den Namen der Klasse zu definieren, die bei der Ausführung des Programms verwendet werden soll.

```
@FXML
public void initialize(Class<? extends ProductModel> clazz) throws
ClassNotFoundException, InstantiationException, IllegalAccessException {
    Object instance = clazz.newInstance();
    if (instance instanceof ProductModel) {
        this.product = (ProductModel) instance;
    } else {
        // Fehlerbehandlung: Die Klasse implementiert das Interface nicht
    }
}
```

Die Definition in der Konfigurationsdatei (z.B. **properties** Datei), sowie der Aufruf der Controller Klasse würde dann wie folgt aussehen:

```
model.impl.product=at.htlle.pos3.javafx.model.Product
```

```
controller.initialize((Class<? extends ProductModel>)
Class.forName(configProperties.getProperty("model.impl.product")));
```

Hier ist der angepasste Abschnitt, der auch erklärt, wie die vom Benutzer selektierten Werte abgefragt werden können und wie diese Elemente in FXML definiert werden:

6. Eingabeelemente in JavaFX

JavaFX bietet eine Vielzahl von Steuerelementen, die Benutzereingaben ermöglichen. Im folgenden wird beschrieben, wie diese Element mit und ohne FXML definiert, vordefinierte Werte gesetzt werden und die Auswahl des Benutzers in der Controller Klasse abgefragt werden kann.



Wie die Beispiele zeigen, sollen vordefinierte Werte (z.B. im Dropdown Feld), oder eine Standardauswahl (z.B. bei Radiobuttons und Checkboxes), in der **initialize()** Methode des Controllers gesetzt werden.

Das Auslesen der ausgewählten Werte des Benutzers muss innerhalb einer **Event**-Methode, die üblicherweise beim Betätigen eine Buttons erfolgt (z.B. in FXML über die Actionattribute definiert:

```
onAction="#submit", onClick="#submit")
```

6.1. Eingabeelemente mit Auswahlmöglichkeit

Um die vom Benutzer eingegebenen oder ausgewählten Werte zu verarbeiten, werden typischerweise Event-Handler-Methoden verwendet. Diese Methoden werden aufgerufen, wenn ein bestimmtes Ereignis eintritt, z.B. ein Klick auf einen Button. Innerhalb dieser Methoden können dann die aktuellen Werte der Eingabeelemente abgefragt werden.

6.1.1. Dropdown (ComboBox)

Eine **ComboBox** wird verwendet, um dem Benutzer eine Auswahl aus einer Liste von Optionen anzubieten. Sie ermöglicht es, eine Option auszuwählen oder einen eigenen Text einzugeben.

FXML-Deklaration: Hier deklarieren wir die ComboBox und einen Button, der das Auslesen des Wertes auslöst.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.ComboBox?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>

<VBox spacing="10" alignment="CENTER" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.example.MyController">
    <children>
        <ComboBox fx:id="comboBox" promptText="Bitte auswählen"/>
        <Button text="Auswahl bestätigen" onAction="#handleComboBoxAction"/>
    </children>
</VBox>
```

Java Controller Code: Im Controller greifen wir auf die ComboBox zu, initialisieren sie und definieren eine Event-Handler-Methode, die beim Klick auf den Button aufgerufen wird.

```
package com.example; // Ersetze dies mit deinem Paketnamen

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ComboBox;

public class MyController {

    @FXML
    private ComboBox<String> comboBox;

    @FXML
    public void initialize() {
        // ComboBox mit Optionen füllen
        comboBox.getItems().addAll("Option 1", "Option 2", "Option 3", "Eigene
Eingabe");
    }
}
```

```

        // Optional einen Standardwert setzen
        comboBox.setValue("Option 1");
        // Erlaubt dem Benutzer, eigenen Text einzugeben, der nicht in der Liste
        ist
        comboBox.setEditable(true);
    }

    /**
     * Diese Methode wird aufgerufen, wenn der Button "Auswahl bestätigen"
     geklickt wird.
     * Sie liest den aktuell ausgewählten oder eingegebenen Wert aus der ComboBox.
     */
    @FXML
    private void handleComboBoxAction(ActionEvent event) {
        String selectedValue = comboBox.getValue();
        if (selectedValue != null && !selectedValue.isEmpty()) {
            System.out.println("Ausgewählte/Eingegebene Option in ComboBox: " +
selectedValue);
            // Hier könntest du den Wert weiterverarbeiten
        } else {
            System.out.println("Keine Option in ComboBox ausgewählt oder
eingegeben.");
        }
    }
}

```

In der `handleComboBoxAction` Methode wird `comboBox.getValue()` aufgerufen, um den aktuell in der ComboBox ausgewählten oder eingegebenen Wert zu erhalten.

6.1.2. Radio-Buttons

RadioButtons werden verwendet, um eine Auswahl aus einer Gruppe von Optionen zu ermöglichen, wobei immer nur eine Option gleichzeitig gewählt werden kann. Sie werden üblicherweise in einer `ToggleGroup` zusammengefasst.

FXML-Deklaration: Wir definieren eine `ToggleGroup` und mehrere `RadioButtons`, die zu dieser Gruppe gehören, sowie einen Button zur Bestätigung.

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.RadioButton?>
<?import javafx.scene.control.ToggleGroup?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.layout.HBox?>

<VBox spacing="10" alignment="CENTER_LEFT" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.example.MyController">
    <fx:define>
        <ToggleGroup fx:id="genderGroup" />
    </fx:define>
    <children>

```

```

        <HBox spacing="10">
            <children>
                <RadioButton fx:id="radioMale" text="Männlich"
toggleGroup="$genderGroup" userData="Männlich"/>
                <RadioButton fx:id="radioFemale" text="Weiblich"
toggleGroup="$genderGroup" userData="Weiblich"/>
                <RadioButton fx:id="radioDiverse" text="Divers"
toggleGroup="$genderGroup" userData="Divers"/>
            </children>
        </HBox>
        <Button text="Auswahl abrufen" onAction="#handleRadioAction"/>
    </children>
</VBox>

```

Java Controller Code: Im Controller wird die `ToggleGroup` injiziert. Die Event-Handler-Methode fragt den ausgewählten `RadioButton` aus der Gruppe ab.

```

package com.example; // Ersetze dies mit deinem Paketnamen

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;

public class MyController {

    @FXML
    private ToggleGroup genderGroup;

    // Einzelne RadioButtons sind nicht zwingend notwendig als @FXML-Felder,
    // wenn die Auswahl über die ToggleGroup erfolgt.
    // @FXML private RadioButton radioMale;
    // @FXML private RadioButton radioFemale;
    // @FXML private RadioButton radioDiverse;

    @FXML
    public void initialize() {
        // Optional einen RadioButton vorauswählen
        // (Hier den ersten RadioButton der Gruppe, falls vorhanden)
        if (genderGroup.getToggles() != null &&
!genderGroup.getToggles().isEmpty()) {
            // Zum Beispiel den ersten Button vorauswählen:
            // ((RadioButton) genderGroup.getToggles().get(0)).setSelected(true);
        }
    }

    /**
     * Diese Methode wird aufgerufen, wenn der Button "Auswahl abrufen" geklickt
     wird.
     * Sie liest den ausgewählten RadioButton aus der ToggleGroup.
     */
}

```

```

@FXML
private void handleRadioAction(ActionEvent event) {
    RadioButton selectedRadioButton = (RadioButton)
genderGroup.getSelectedToggle();

    if (selectedRadioButton != null) {
        String selectedValue = selectedRadioButton.getText(); // Oder
        userData, falls gesetzt
        // String userDataValue = (String) selectedRadioButton.getUserData();
        System.out.println("Ausgewählter RadioButton: " + selectedValue);
        // System.out.println("UserData des RadioButtons: " + userDataValue);
    } else {
        System.out.println("Kein RadioButton ausgewählt.");
    }
}
}

```

In `handleRadioAction` wird `genderGroup.getSelectedToggle()` verwendet, um den aktuell ausgewählten `RadioButton` zu erhalten. Davon kann dann der Text oder ein anderer Wert (z.B. `userData`) abgefragt werden.

6.1.3. Checkbox

Eine **Checkbox** ermöglicht es dem Benutzer, eine Ja/Nein-Entscheidung zu treffen oder mehrere Optionen unabhängig voneinander auszuwählen.

FXML-Deklaration: Hier eine `CheckBox` und ein `Button` zur Verarbeitung.

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.CheckBox?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>

<VBox spacing="10" alignment="CENTER_LEFT" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.example.MyController">
    <children>
        <CheckBox fx:id="agbCheckBox" text="AGB akzeptieren"/>
        <Button text="Status prüfen" onAction="#handleCheckBoxAction"/>
    </children>
</VBox>

```

Java Controller Code: Der Controller greift auf die `CheckBox` zu und prüft ihren Status in der Event-Handler-Methode.

```

package com.example; // Ersetze dies mit deinem Paketnamen

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.CheckBox;

```

```

public class MyController {

    @FXML
    private CheckBox agbCheckBox;

    @FXML
    public void initialize() {
        // Optional den Standardstatus der CheckBox setzen
        agbCheckBox.setSelected(false);
    }

    /**
     * Diese Methode wird aufgerufen, wenn der Button "Status prüfen" geklickt
     wird.
     * Sie liest den aktuellen Status (ausgewählt/nicht ausgewählt) der CheckBox.
     */
    @FXML
    private void handleCheckBoxAction(ActionEvent event) {
        boolean isSelected = agbCheckBox.isSelected();
        System.out.println("CheckBox 'AGB akzeptieren' ist ausgewählt: " +
            isSelected);

        if (isSelected) {
            System.out.println("Benutzer hat die AGB akzeptiert.");
        } else {
            System.out.println("Benutzer hat die AGB nicht akzeptiert.");
        }
    }
}

```

In `handleCheckBoxAction` wird `agbCheckBox.isSelected()` genutzt, um zu prüfen, ob die CheckBox angehakt ist.

6.2. Weitere Eingabeelemente

6.2.1. Textarea

Ein **Textarea** (TextArea) ermöglicht die Eingabe von mehrzeiligem Text.

FXML-Deklaration: Eine TextArea und ein Button zum Abrufen des Inhalts.

```

<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.TextArea?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>

<VBox spacing="10" alignment="CENTER" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.example.MyController">
    <children>
        <TextArea fx:id="commentTextArea" promptText="Geben Sie Ihren Kommentar
hier ein..." />
    </children>
</VBox>

```

```
<Button text="Kommentar absenden" onAction="#handleTextAreaAction"/>
</children>
</VBox>
```

Java Controller Code: Der Controller liest den Text aus der TextArea in der Event-Handler-Methode.

```
package com.example; // Ersetze dies mit deinem Paketnamen

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.TextArea;

public class MyController {

    @FXML
    private TextArea commentTextArea;

    @FXML
    public void initialize() {
        // Optional einen anfänglichen Text oder weitere Einstellungen vornehmen
        // commentTextArea.setText("Vorbelegter Text...");
    }

    /**
     * Diese Methode wird aufgerufen, wenn der Button "Kommentar absenden"
     geklickt wird.
     * Sie liest den aktuellen Text aus der TextArea.
     */
    @FXML
    private void handleTextAreaAction(ActionEvent event) {
        String enteredText = commentTextArea.getText();
        System.out.println("Eingegebener Text in TextArea:");
        System.out.println(enteredText);

        if (enteredText.isEmpty()) {
            System.out.println("Die TextArea ist leer.");
        } else {
            // Hier könnte der Text weiterverarbeitet werden (z.B. Speichern)
        }
    }
}
```

In `handleTextAreaAction` wird `commentTextArea.getText()` verwendet, um den gesamten vom Benutzer eingegebenen Text abzurufen.



Wichtige Hinweise:

- Ersetze `com.example.MyController` in den FXML-Dateien und `package com.example;` in den Java-Dateien durch deinen tatsächlichen Paket- und Controllernamen.

- Die Event-Handler-Methoden (z.B. `handleComboBoxAction`) müssen mit `@FXML` annotiert sein und entweder `public` oder `protected` sein, oder wenn sie `private` sind, muss sichergestellt sein, dass das Modul-System (falls verwendet) den Zugriff erlaubt. Die Methode kann optional einen `ActionEvent` als Parameter haben.
- Die `initialize()`-Methode wird automatisch von JavaFX aufgerufen, nachdem alle `@FXML`-annotierten Felder injiziert wurden. Sie eignet sich gut für die einmalige Einrichtung der UI-Komponenten, oder Komponenten, welche die Daten weiter verarbeiten (z.B. *speichern der Daten in einer DB oder einer Datei*)

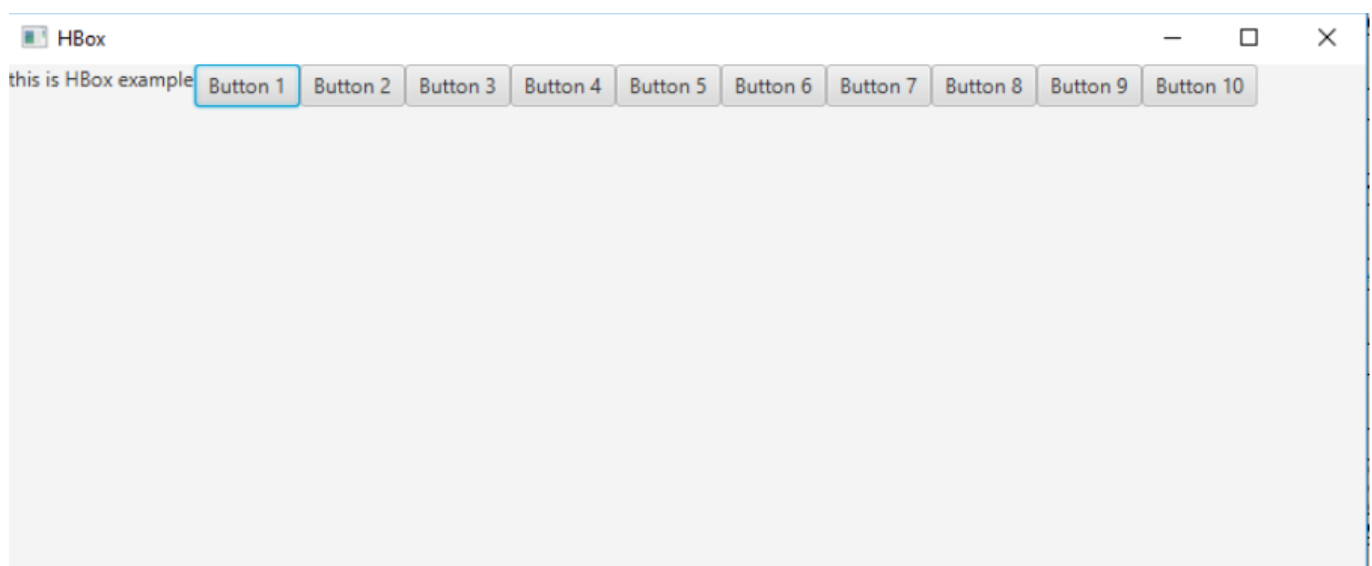
7. Layouting, Dialoge und Datenpräsentation in JavaFX

7.1. Layout - Klassen in JavaFX

In der Welt der JavaFX-Entwicklung spielen Layoutklassen eine entscheidende Rolle bei der Gestaltung und Organisation der Benutzeroberfläche einer Anwendung. Diese Klassen bieten eine Vielzahl von Möglichkeiten, um Komponenten innerhalb eines Fensters zu positionieren und zu verwalten, wodurch eine ansprechende und funktionale Benutzeroberfläche entsteht. Von grundlegenden Anordnungen, die Elemente in einer Linie oder einem Raster ausrichten, bis hin zu fortgeschrittenen Layouts, die flexible Anpassungen an Größe und Position der Komponenten erlauben, decken die Layoutklassen in JavaFX ein breites Spektrum an Designanforderungen ab. In diesem Abschnitt befassen wir uns mit den Schlüsseleigenschaften und der Anwendung der verschiedenen Layoutklassen, darunter `HBox`, `VBox`, `GridPane`, `FlowPane`, und `BorderPane`, um nur einige zu nennen. Durch das Verständnis dieser Layoutmechanismen können Entwickler intuitive und responsive Benutzeroberflächen kreieren, die die Interaktion mit der Anwendung zu einem angenehmen Erlebnis machen.

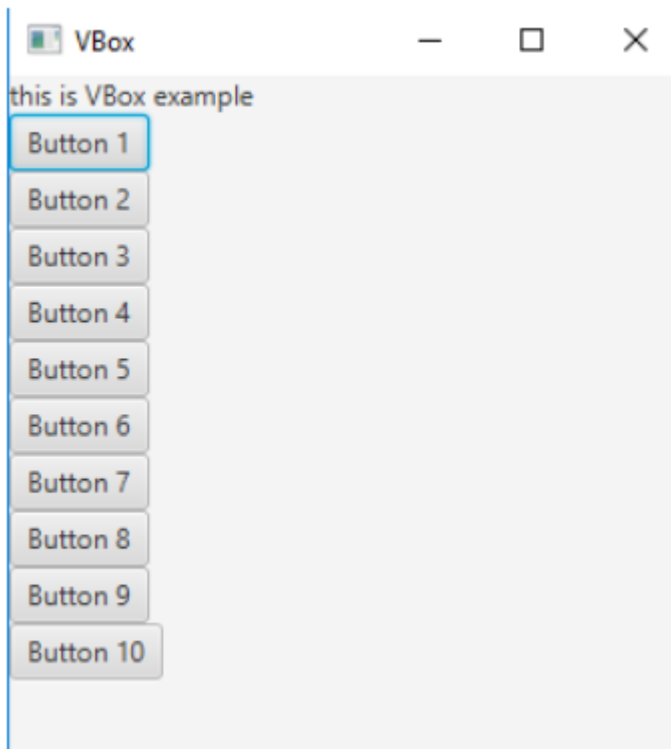
7.1.1. Überblick über die gängigsten Layout - Klassen

HBox:



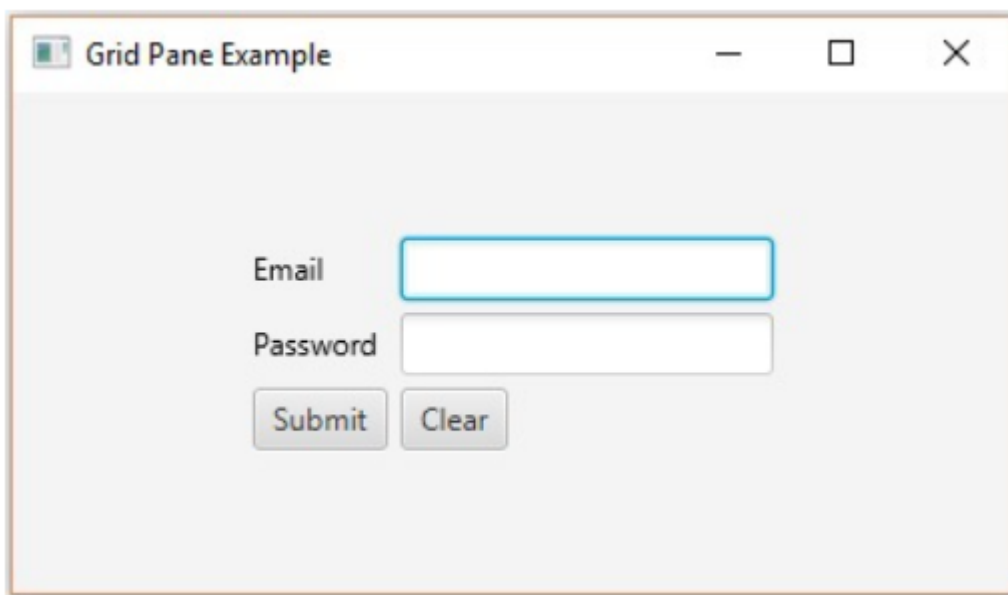
Das `HBox`-Layout ordnet seine untergeordneten Elemente horizontal an. Es ist nützlich zum Erstellen von Symbolleisten, Menüs und anderen horizontalen Layouts.

VBox:



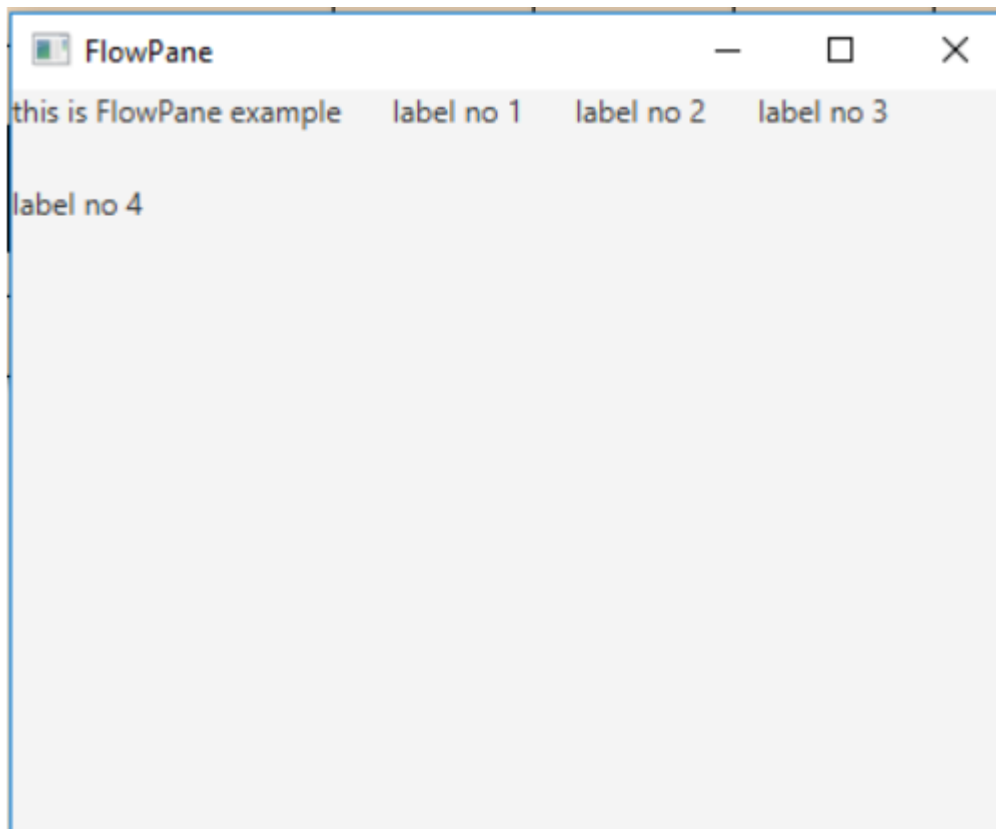
Das VBox-Layout ordnet seine untergeordneten Elemente vertikal an. Es ist nützlich zum Erstellen von Formularen, Dialogen und anderen vertikalen Layouts.

GridPane:



Das GridPane-Layout ordnet seine untergeordneten Elemente in einem Raster an. Es ist nützlich zum Erstellen von komplexeren Layouts, z. B. Tabellen und Formularen mit mehreren Spalten.

FlowPane:

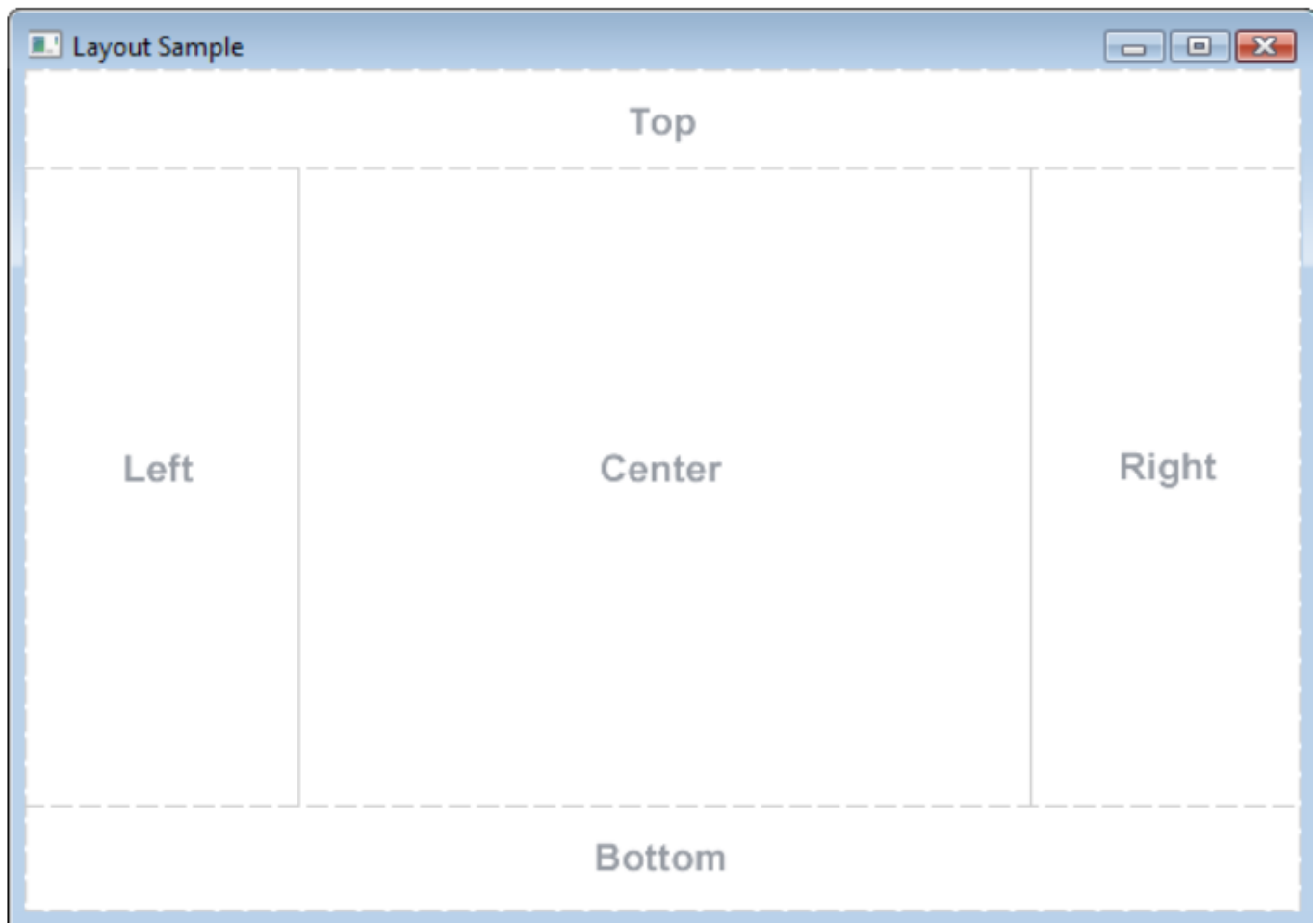


Das FlowPane-Layout ordnet seine untergeordneten Elemente in einem fließenden Layout an. Es ist nützlich zum Erstellen von Layouts, bei denen die Größe der untergeordneten Elemente nicht bekannt ist, z. B. einer Liste von Tags.

StackPane:



Das StackPane-Layout ordnet seine untergeordneten Elemente übereinander an. Es ist nützlich zum Erstellen von Layouts, bei denen ein untergeordnetes Element über den anderen angezeigt wird, z. B. einer Popup-Nachricht.

BorderPane:

Das BorderPane-Layout ordnet seine untergeordneten Elemente in fünf Bereichen an: oben, unten, links, rechts und Mitte. Es ist nützlich zum Erstellen von Layouts mit einem Hauptbereich und mehreren Randbereichen, z. B. einem Fenster mit einem Menü, einer Symbolleiste und einem Statusbereich.

Scene Builder:

Scene Builder ist ein grafisches Tool zum Erstellen von JavaFX-Layouts. Es ermöglicht Ihnen, Ihre Layouts per Drag-and-Drop zu erstellen und die Eigenschaften Ihrer untergeordneten Elemente visuell zu bearbeiten.

Weitere Informationen:

- JavaFX-Layouts: <https://docs.oracle.com/javafx/2/layout/index.html>
- Scene Builder: <https://gluonhq.com/products/scene-builder/>

7.2. Dialoge in JavaFX

In der Entwicklung von Desktop-Anwendungen mit JavaFX spielen modale Dialoge eine wichtige Rolle. Sie sind spezielle Fenster, die die Interaktion mit anderen Teilen der Anwendung verhindern, solange sie geöffnet sind. Dies zwingt den Benutzer, mit dem Dialog zu interagieren, bevor er zur Hauptanwendung zurückkehrt. JavaFX unterstützt generell zwei Haupttypen von Modalität:

- **Applikationsmodalität:** es werden alle Fenster der gesamten Anwendung blockiert solange der Dialog geöffnet ist.
- **Fenstermodalität:** es wird nur das Fenster, von dem der Dialog aufgerufen wurde, blockiert

Des Weiteren werden seit Java 8 Standard Dialoge bereitgestellt, auf die im folgenden Abschnitt näher eingegangen wird.

7.2.1. JavaFX Standard Dialoge

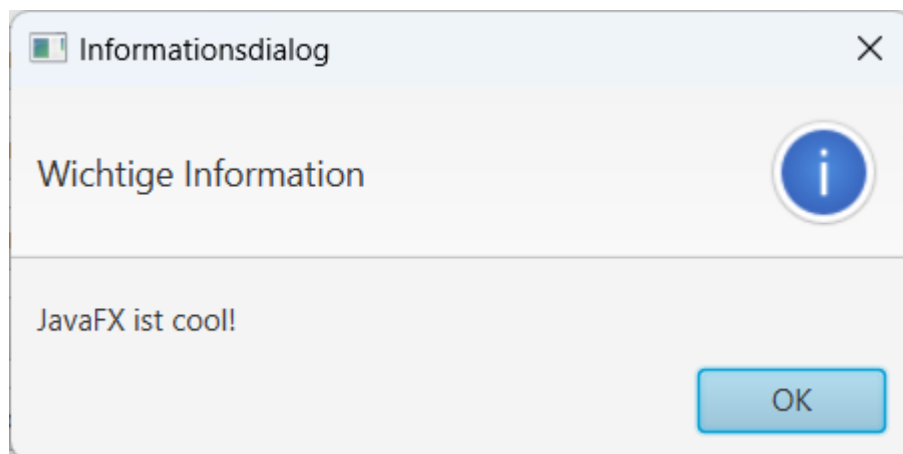
JavaFX hat mit der Version **8u40** die Unterstützung für Standarddialoge eingeführt, um Entwicklern die Arbeit zu erleichtern. Diese Dialoge sind:

- **Alert:** Ein einfacher Dialog zum Anzeigen von Nachrichten, Warnungen oder Fehlern.
- **Confirmation:** Ein Dialog, der eine Bestätigung vom Benutzer verlangt.
- **TextInputDialog:** Ein Dialog, der vom Benutzer Texteingaben anfordert.
- **ChoiceDialog:** Ein Dialog, der es dem Benutzer ermöglicht, eine Auswahl aus einer Liste von Optionen zu treffen.

Information Dialog

Der **Alert**-Dialog ist einer der häufigsten Dialoge. Er wird verwendet, um dem Benutzer Informationen, Warnungen oder Fehlermeldungen anzuzeigen. Hier ist ein einfaches Beispiel für die Erstellung und Anzeige eines Alert-Dialogs:

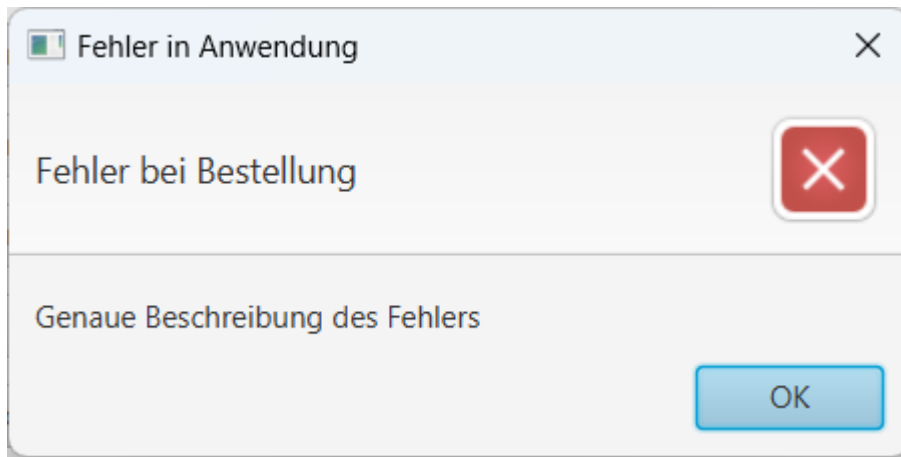
```
Alert alert = new Alert(Alert.AlertType);
alert.setTitle("Informationsdialog");
alert.setHeaderText("Wichtige Information");
alert.setContentText("JavaFX ist cool!");
alert.showAndWait();
```



Error Dialog

```
Alert alert = new Alert(Alert.AlertType.ERROR);
alert.setTitle("Fehler in Anwendung");
alert.setHeaderText("Fehler bei Bestellung");
alert.setContentText("Genaue Beschreibung des Fehlers");

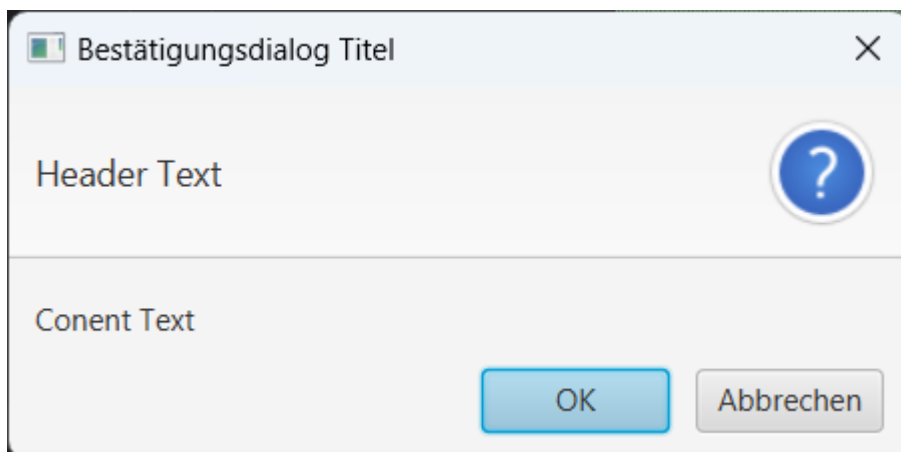
alert.showAndWait();
```



Confirmation Dialog

Der **Confirmation**-Dialog fragt den Benutzer, ob er eine bestimmte Aktion fortsetzen möchte. Ein typisches Beispiel wäre die Bestätigung des Benutzers, bevor Daten gelöscht werden:

```
Alert confirmation = new Alert(Alert.AlertType.CONFIRMATION);
confirmation.setTitle("Bestätigungsdialog Titel");
confirmation.setHeaderText("Header Text");
confirmation.setContentText("Content Text");
Optional<ButtonType> result = confirmation.showAndWait();
if (result.get() == ButtonType.OK){
    // Benutzer wählt OK; fortfahren mit Aktion
} else {
    // Benutzer wählt Abbrechen; Aktion abbrechen
}
```

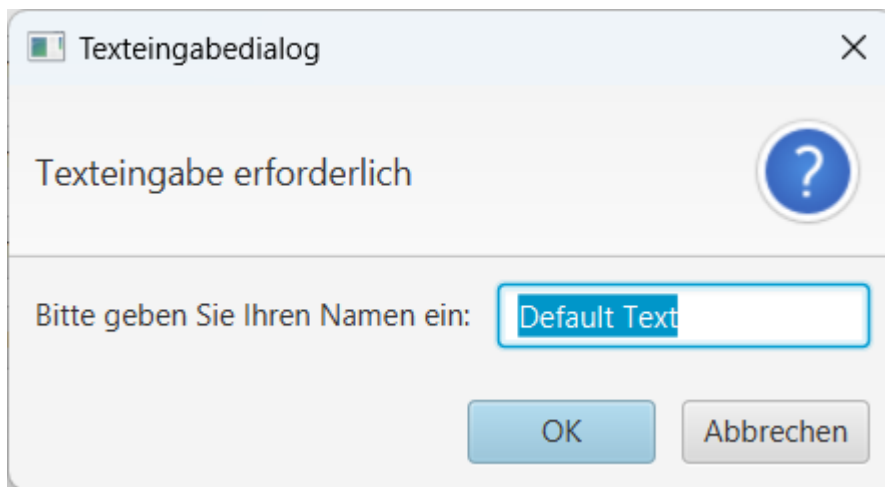


TextInputDialog

Mit dem **TextInputDialog** kann der Benutzer eine Zeichenfolge eingeben. Dies ist nützlich für die Eingabe von Namen, IDs oder anderen kurzen Texten:

```
TextInputDialog textInputDialog = new TextInputDialog("Default Text");
textInputDialog.setTitle("Texteingabedialog");
```

```
textInputDialog.setHeaderText("Texteingabe erforderlich");
textInputDialog.setContentText("Bitte geben Sie Ihren Namen ein:");
Optional<String> result = textInputDialog.showAndWait();
result.ifPresent(name -> System.out.println("Benutzername: " + name));
```



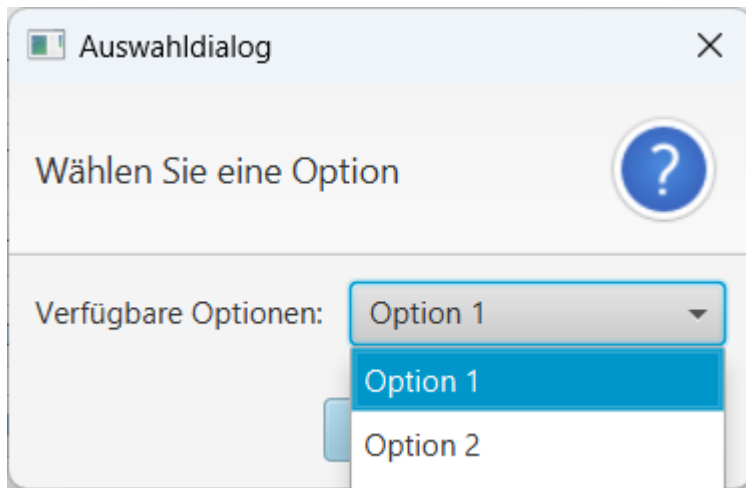
ChoiceDialog

Der `ChoiceDialog` ermöglicht es dem Benutzer, eine Auswahl aus einer Liste von Optionen zu treffen. Dies ist hilfreich für Einstellungen, Auswahlverfahren oder jede Situation, in der der Benutzer aus mehreren Optionen wählen muss:

```
List<String> choices = new ArrayList<>();
choices.add("Option 1");
choices.add("Option 2");
choices.add("Option 3");

ChoiceDialog<String> dialog = new ChoiceDialog<>("Option 1", choices);
dialog.setTitle("Auswahldialog");
dialog.setHeaderText("Wählen Sie eine Option");
dialog.setContentText("Verfügbare Optionen:");

Optional<String> result = dialog.showAndWait();
result.ifPresent(choice -> System.out.println("Auswahl: " + choice));
```



Fazit

JavaFX Standard Dialoge vereinfachen die Entwicklung interaktiver Anwendungen, indem sie eine einfache und effiziente Möglichkeit bieten, mit dem Benutzer zu kommunizieren. Durch die Verwendung dieser Dialoge können Entwickler die Benutzererfahrung ihrer Anwendungen verbessern, ohne sich um die Grundlagen der Dialogerstellung kümmern zu müssen.

Weiterführende Links:

- <https://o7planning.org/11529/javafx-alert-dialog>

7.2.2. Benutzerdefinierte Dialoge in JavaFX

Applikationsmodale Dialoge

Applikationsmodale Dialoge blockieren alle Fenster der gesamten Anwendung. Das bedeutet, dass der Benutzer nicht mit anderen Teilen der Anwendung interagieren kann, bis der modale Dialog geschlossen wird. Applikationsmodale Dialoge werden häufig für kritische Aktionen verwendet, bei denen eine Entscheidung des Benutzers erforderlich ist, bevor die Anwendung fortgesetzt werden kann.

Beispiel

```
Dialog<String> dialog = new Dialog<>();
dialog.setTitle("Applikationsmodaler Dialog");
dialog.setContentText("Dieser Dialog ist applikationsmodal.");
// Setzen der Modalität
dialog.initModality(Modality.APPLICATION_MODAL);

dialog.showAndWait();
```

Fenstermodale Dialoge

Fenstermodale Dialoge blockieren nur das Fenster, von dem sie aufgerufen wurden, und lassen die Interaktion mit anderen Fenstern der Anwendung zu. Diese Art von Modalität eignet sich gut für Aufgaben, die sich nur auf einen Teil der Anwendung beziehen.

Beispiel

```
// Annahme, dass primaryStage das Hauptfenster der Anwendung ist
Dialog<String> dialog = new Dialog<>();
dialog.setTitle("Fenstermodaler Dialog");
dialog.setContentText("Dieser Dialog ist fenstermodal.");
// Setzen der Modalität und Angabe des Elternfensters
dialog.initModality(Modality.WINDOW_MODAL);
dialog.initOwner(primaryStage);

dialog.showAndWait();
```

Gestaltung Modaler Dialoge

Die Gestaltung modaler Dialoge in JavaFX ist flexibel. Entwickler können die Dialoge mit verschiedenen Steuerelementen und Layouts anpassen, um eine reiche Benutzererfahrung zu bieten. Zum Beispiel können Sie Buttons, Textfelder, Dropdown-Menüs und andere JavaFX-Komponenten in Ihren Dialogen verwenden.

Best Practices

- **Klare Kommunikation:** Verwenden Sie klare und präzise Botschaften in Ihren Dialogen. Benutzer sollten verstehen, warum der Dialog angezeigt wird und was von ihnen erwartet wird.
- **Angemessene Modalität wählen:** Entscheiden Sie weise zwischen Applikationsmodalität und Fenstermodalität, basierend auf dem Kontext, in dem der Dialog verwendet wird.
- **Feedback:** Geben Sie dem Benutzer eindeutiges Feedback nach einer Aktion im Dialog, insbesondere bei kritischen oder irreversiblen Aktionen.

Erstellen einfacher Dialoge ohne FXML

Im Folgenden Beispiel wird gezeigt, wie ein einfacher Dialog erstellt wird, in dem man die Daten eines Produkts ändern kann.

Edit Product [X]

Edit the data for the selected product:

Name:
Amazon Echo Dot

ID:
5

Price:
59.99

Description:
A smart speaker from Amazon with Alexa voice assistant and great sound for its size.

[Save] [Abbrechen]

```
// Wenn ein Produkt ausgewählt ist, öffne einen Dialog, um die Daten zu
bearbeiten
if (selectedProduct != null) {
    // Erstelle das Dialogfenster
    Dialog<Product> dialog = new Dialog<>();
    dialog.setTitle("Edit Product");
    dialog.setHeaderText("Edit the data for the selected product:");
    dialog.initModality(Modality.APPLICATION_MODAL);

    // Erstelle die UI-Komponenten im Dialogfenster
    Label nameLabel = new Label("Name:");
    TextField nameTextField = new TextField(selectedProduct.getName());
    Label idLabel = new Label("ID:");
    TextField idTextField = new
TextField(Integer.toString(selectedProduct.getId()));
    // weitere Labels und Textfelder
    VBox dialogContent = new VBox(nameLabel, nameTextField, idLabel,
idTextField, priceLabel, priceTextField, descriptionLabel, descriptionTextArea);

    // Füge die UI-Komponenten zum Dialogfenster hinzu
    dialog.getDialogPane().setContent(dialogContent);

    // Erstelle die Buttons im Dialogfenster
    ButtonType saveButtonType = new ButtonType("Save",
ButtonBar.ButtonData.OK_DONE);
    dialog.getDialogPane().getButtonTypes().addAll(saveButtonType,
ButtonType.CANCEL);
```

```

        // Wenn der "Save"-Button geklickt wird, übernehme die neuen Daten und
        // schließe das Dialogfenster
        dialog.setResultConverter(buttonType -> {
            if (buttonType == saveButtonType) {
                String name = nameTextField.getText();
                int id = Integer.parseInt(idTextField.getText());
                double price = Double.parseDouble(priceTextField.getText());
                String description = descriptionTextArea.getText();
                return new Product(name, id, price, description);
            }
            return null;
        });

        // Öffne das Dialogfenster und warte auf die Eingabe des Benutzers
        Optional<Product> result = dialog.showAndWait();

        // Wenn der Benutzer die Eingabe bestätigt hat, aktualisiere die Daten
        // des Produkts in der Tabelle
        result.ifPresent(newProduct -> {
            int selectedIndex =
productTable.getSelectionModel().getSelectedIndex();
            productList.set(selectedIndex, newProduct);
        });

```

Schritt-für-Schritt-Erklärung

1. `Dialog<Product> dialog = new Dialog<>();` erstellt ein neues Dialog-Objekt, das eine Instanz der Klasse `Product` zurückgibt.
2. `dialog.setTitle("Edit Product");` setzt den Titel des Dialogfensters auf "Edit Product".
3. `dialog.setHeaderText("Edit the data for the selected product:");` setzt den Text im Header des Dialogfensters auf "Edit the data for the selected product:".
4. Es werden UI-Komponenten erstellt, die dem Dialogfenster hinzugefügt werden. Dazu gehören Labels und Textfelder für den Namen, die ID, den Preis und die Beschreibung des Produkts. Der Wert der Textfelder wird auf die aktuellen Werte des ausgewählten Produkts gesetzt.
5. `VBox dialogContent = new VBox(nameLabel, nameTextField, idLabel, idTextField, priceLabel, priceTextField, descriptionLabel, descriptionTextArea);` erstellt ein `VBox`-Objekt, das die UI-Komponenten enthält, die dem Dialogfenster hinzugefügt werden sollen.
6. `dialog.getDialogPane().setContent(dialogContent);` fügt die UI-Komponenten dem Dialogfenster hinzu.
7. Es werden Buttons für das Dialogfenster erstellt, darunter ein "Save"-Button und ein "Cancel"-Button.
8. `dialog.getDialogPane().getButtonTypes().addAll(saveButtonType, ButtonType.CANCEL);` fügt die Buttons dem Dialogfenster hinzu.
9. Ein `resultConverter` wird erstellt, der die Eingabe des Benutzers in eine Instanz der Klasse `Product` umwandelt. Wenn der "Save"-Button geklickt wird, werden die neuen Werte aus den Textfeldern ausgelesen und als neues `Product`-Objekt zurückgegeben.
10. `Optional<Product> result = dialog.showAndWait();` zeigt das Dialogfenster an und wartet auf die Eingabe des Benutzers. Wenn der Benutzer auf "Save" klickt, enthält `result` das aktualisierte `Product`-Objekt.

Zusammengefasst ermöglicht dieser Code die Bearbeitung von Produktdaten durch den Benutzer über ein Dialogfenster und gibt die aktualisierten Daten als **Product**-Objekt zurück.

Aufruf einer FXML - View als Dialog

Um eine JavaFX FXML View als Dialog aufzurufen, können Sie das Dialogfenster in einem separaten Controller definieren und dann die entsprechende FXML-Datei laden.

Hier ist ein Beispiel, wie Sie das tun können:

1. Erstellen Sie einen separaten Controller für das Dialogfenster:

```
public class DialogController {
    // Ihre Controller-Logik hier

    private Product product;

    @FXML
    private Button okButton;

    @FXML
    private TextField productNameTxt;
    ...

    @FXML
    public void handleOkButton(ActionEvent e) {
        this.product = new Product();
        product.setName(productNameTxt.getText());
        ...

        // Schließen Sie das Dialogfenster
        Stage stage = (Stage) okButton.getScene().getWindow();
        stage.close();
    }

    // Initialisierung des Dialogs falls notwendig
    @FXML
    public void initialize() {

        // Initialisierung des Dialogs
    }

    public Product getProduct() {
        return product;
    }
}
```

2. Erstellen Sie eine FXML-Datei für das Dialogfenster (z.B. **my-dialog.fxml**) und definieren Sie das Layout und die Steuerelemente des Dialogfensters in dieser Datei.
3. Laden Sie die FXML-Datei in Ihrem Hauptcontroller und rufen Sie das Dialogfenster auf:

```

public class ParentController {
    @FXML
    private Button dialogButton;

    @FXML
    public void openDialog(ActionEvent event) throws IOException {
        FXMLLoader loader = new FXMLLoader(getClass().getResource("my-
dialog.fxml"));
        Parent dialogRoot = loader.load();

        MeinDialogController dialogController = loader.getController();
        // Initialisieren Sie den Dialogcontroller hier, wenn nötig
        dialogController.initialize();

        Scene dialogScene = new Scene(dialogRoot);
        Stage dialogStage = new Stage();
        dialogStage.setScene(dialogScene);
        dialogStage.initModality(Modality.APPLICATION_MODAL);
        dialogStage.showAndWait();

        // Hier können Sie auf Daten des Dialogs über
        // dessen Controller zugreifen
        Product newProduct = dialogController.getProduct();
    }
}

```

In diesem Beispiel wird die `FXMLLoader`-Klasse verwendet, um die FXML-Datei zu laden und den Dialogcontroller zu erhalten. Der Dialogcontroller kann initialisiert werden, bevor der Dialog angezeigt wird.

Dann wird ein neues Fenster mit der `Stage`-Klasse erstellt und die **modale Eigenschaft** auf `APPLICATION_MODAL` gesetzt, um zu verhindern, dass der Benutzer auf das Hauptfenster zugreift, bis der Dialog geschlossen wird. Schließlich wird der Dialog angezeigt und mit `showAndWait()` blockiert, bis der Benutzer den Dialog schließt.

Wenn der Dialog einen Rückgabewert hat, können Sie auf den Rückgabewert zugreifen, nachdem der Dialog geschlossen wurde.

Will man die `Stage` - Instanz, von welcher der Dialog aufgerufen wird (= "Parent - Stage") **explizit setzen**, so kann dies wie folgt geschehen: `dialogStage.initOwner(parentStage)`. Dies ist notwendig, wenn der Dialog über den DialogController direkt erstellt wird.

7.2.3. Verwenden eines `FileChooser` Dialogs

Die `FileChooser` Klasse ermöglicht es auf eine einfache Weise einen File - Dialog aufzurufen.

Der folgende Code öffnet einen File - Dialog und ermöglicht es dem Benutzer, eine Datei auszuwählen. Der ausgewählte Dateiname wird in der Variable `selectedFile` gespeichert. Der Code prüft auch, ob eine Datei ausgewählt wurde, bevor mit der Datei gearbeitet wird.

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Open CSV file");
fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("CSV
files", "*.csv"));
File selectedFile =
fileChooser.showOpenDialog(root.getScene().getWindow());

if (selectedFile != null) {
    // do anything with the selected file
}
```

Im Detail sieht der Code folgendermaßen aus:

1. `FileChooser fileChooser = new FileChooser();` erstellt eine neue Instanz des `FileChooser`-Objekts.
2. `fileChooser.setTitle("Open CSV file");` setzt den Titel des Dialogs auf "Open CSV file".
3. `fileChooser.getExtensionFilters().add(new FileChooser.ExtensionFilter("CSV files", "*.csv"));` definiert die zulässigen Dateierweiterungen, die der Benutzer auswählen kann. In diesem Fall ist es ".csv" und der Filtername ist "CSV files".
4. `File selectedFile = fileChooser.showOpenDialog(root.getScene().getWindow());` öffnet den Dateiauswahldialog und speichert den ausgewählten Dateinamen in der Variable "selectedFile".
5. `if (selectedFile != null) { // do anything with the opened file }` überprüft, ob eine Datei ausgewählt wurde. Wenn ja, können beliebige Operationen auf der geöffneten Datei ausgeführt werden.

7.3. Verwendung von TableView zur Verwaltung von Tabellendaten

7.3.1. Was ist eine Table View

Eine `TableView` ist ein grafisches Steuerelement in JavaFX, das zur Anzeige von tabellarischen Daten verwendet wird. Mit der `TableView` können Daten in Form einer Tabelle dargestellt werden, ähnlich wie in einer Excel-Tabelle oder einer Datenbank-Tabelle.

Die `TableView` besteht aus Zeilen und Spalten, die in der Regel durch eine `ObservableList` von Objekten dargestellt werden. Jedes Objekt in der Liste repräsentiert eine Zeile in der `TableView`, und die Attribute des Objekts werden in den Spalten der `TableView` angezeigt.

Die `TableView` bietet eine Vielzahl von Funktionen, um Daten anzuzeigen und zu bearbeiten, darunter **Sortierung, Filterung, Gruppierung, Bearbeitung von Zellen und Zeilen, Anpassung der Spaltenbreite** und viele mehr. Die `TableView` ist ein wichtiges Steuerelement in JavaFX und wird häufig in Anwendungen verwendet, die Daten in tabellarischer Form anzeigen und bearbeiten müssen.

7.3.2. Definition einer TableView in FXML

Eine `TableView` kann in JavaFX mithilfe von FXML sehr einfach definiert werden. Hier ist ein Beispiel für die Definition einer `TableView` in FXML:

```
<TableView fx:id="productTable">
  <columns>
    <TableColumn text="Name" fx:id="nameColumn" />
    <TableColumn text="ID" fx:id="idColumn" />
    <TableColumn text="Preis" fx:id="priceColumn" />
    <TableColumn text="Beschreibung" fx:id="descriptionColumn" />
  </columns>
</TableView>
```

In diesem Beispiel wird eine `TableView` definiert, die vier Spalten für `Name`, `ID`, `Preis` und `Beschreibung` enthält. Jede Spalte wird mit einem `fx:id` versehen, um sie im Controller ansprechen zu können.

7.3.3. Initialisieren und verwalten einer TableView im Controller

Um die `TableView` mit Daten im Controller zu initialisieren, können Sie die Spalten-Objekte mit den entsprechenden Attributen der `Product`-Klasse verknüpfen und dann eine `ObservableList` von `Product`-Objekten erstellen und der `TableView` hinzufügen. Hier ist ein Beispiel:

```
@FXML
private TableView<Product> productTable;

@FXML
private TableColumn<Product, Integer> idColumn;

@FXML
private TableColumn<Product, String> nameColumn;

@FXML
private TableColumn<Product, Double> priceColumn;

@FXML
private TableColumn<Product, String> descriptionColumn;

private ObservableList<Product> productList = FXCollections.observableArrayList();

public void initialize() {
    // Verknüpfen Sie jede Spalte mit dem entsprechenden Attribut der Product-
    // Klasse
    idColumn.setCellValueFactory(new PropertyValueFactory<>("id"));
    nameColumn.setCellValueFactory(new PropertyValueFactory<>("name"));
    priceColumn.setCellValueFactory(new PropertyValueFactory<>("price"));
    descriptionColumn.setCellValueFactory(new PropertyValueFactory<>
        ("description"));

    // Erstellen Sie eine ObservableList von Product-Objekten und fügen Sie sie
    // der TableView hinzu
    productList.addAll(new CSVProductReader().readProductsFromCSV("products.csv",
        ";"));
    productTable.setItems(productList);
}
```

In diesem Beispiel werden die Spalten-Objekte mit den entsprechenden Attributen der **Product**-Klasse verknüpft. Dann wird eine **ObservableList** von **Product**-Objekten instanziiert (= **productList**). Die **initialize()**-Methode wird automatisch aufgerufen, wenn das FXML-Layout geladen wird, so dass die **TableView** beim Start der Applikation mit Daten, welche in diesem Beispiel aus einer CSV - Datei (mit Hilfe der **CSVProductReader** Klasse) gelesen werden, initialisiert wird (= **productTable.setItems(productList)**)

Beachten Sie, dass die **PropertyValueFactory**-Klasse verwendet wird, um die Spalten-Objekte mit den **Attributen/Eigenschaften** der **Product**-Klasse zu verknüpfen. Dies ist ein wichtiger Bestandteil der Verwendung einer **TableView** in JavaFX. Bei der **Product** - Klasse handelt es sich um eine **POJO** - Klasse:

```
public class Product {
    private int id;
    private String name;
    private String description;
    private double price;

    // getter- und setter- für alle Properties
}
```



Achten Sie darauf, dass die POJO Klasse über Getter- und Setter Methoden für all jene Attribute besitzt, welche in der **TableView** angezeigt werden sollen. Der definierte Name, der im Konstruktor der Klasse **PropertyValueFactory** gesetzt wird (z.B. **new PropertyValueFactory<>("id")**), muss genau dem Namen der Getter-/Setter Methoden (**ohne get bzw. set**) der POJO Klasse (hier **id** mit den Methoden **getId()/setId(int id)**) entsprechen.

7.4. Realisierung von mehrsprachigen Benutzeroberflächen

Um mehrsprachige Oberflächen mit FXML und JavaFX zu erstellen, können Sie das **ResourceBundle**-Konzept verwenden. Hier sind die Schritte, um dies zu erreichen:

1. **Ressourcenbündel erstellen:** Erstellen Sie für jede unterstützte Sprache eine Eigenschaftsdatei (.properties) mit den Übersetzungen für die Benutzeroberflächen-Texte. Benennen Sie die Dateien nach dem folgenden Muster: **messages_.properties**. Zum Beispiel:

- **messages_en.properties** für Englisch
- **messages_de.properties** für Deutsch
- **messages_fr.properties** für Französisch

Platzieren Sie die Eigenschaftsdateien im **src/main/resources** Verzeichnis oder in einem passenden Paketverzeichnis.

2. **Übersetzungen in Eigenschaftsdateien hinzufügen:** Fügen Sie für jedes Benutzeroberflächen-Element, das übersetzt werden soll, einen Eintrag in den Eigenschaftsdateien hinzu. Zum Beispiel: **messages_en.properties**:

```
label.greeting=Hello  
button.submit=Submit
```

3. **Ressourcenbündel in FXML laden**: Laden Sie das Ressourcenbündel in Ihrer FXML-Datei und verwenden Sie die `fx:id`-Attribute für die Elemente, die übersetzt werden sollen. Zum Beispiel:

```
<Label fx:id="greetingLabel" text="%label.greeting" />  
<Button fx:id="submitButton" text="%button.submit" />
```

Die `%`-Syntax zeigt an, dass der Text aus dem Ressourcenbündel geladen werden soll.