

GUI mit JavaFX I

Zentrale Konzepte:

- ⊙ MVC-Architektur
- ⊙ Aufbau einer JavaFX Anwendung
- ⊙ FXML und Scene Builder
- ⊙ Container und Komponenten
- ⊙ Ereignisbehandlung (Event-Handling) in Controller Klassen
- ⊙ *TextField* und *TextArea*

Model-View-Controller Architekturmuster (MVC)

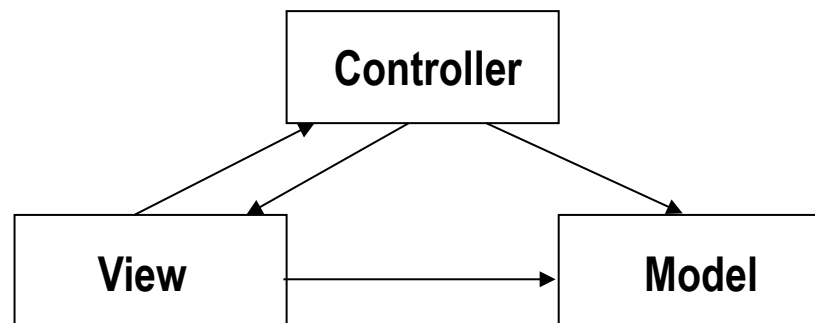
Einsatz: bei interaktiven Programmen mit flexibler Benutzerschnittstelle, d.h.

- ⊙ Das Programm stellt dem Nutzer bestimmte Funktionalitäten zur Verfügung, mit denen er die von dem Programm unterstützten Aufgaben umsetzen kann (z.B. Tabellenkalkulation). Die dahinter steckende Logik nennt man **Anwendungslogik**.
- ⊙ Die für die Ausführung der Aufgaben nötige Information (Datenbasis) wird in interaktiven Fenstern dargestellt. Diese Sichten auf die Daten nennt man auch **View**.
- ⊙ Bestimmte Interaktionen oder Änderungen der Daten können zu Änderungen im Verhalten und somit der Benutzeroberfläche des Systems führen. Die dahinter steckende Logik nennt man **Darstellungslogik**.
- ⊙ Änderungen/Erweiterungen an der Benutzeroberfläche des Systems treten häufig auf und sollten leicht integrierbar sein.

⇒ Benutzerschnittstelle (Darstellungslogik + View) sollte vom funktionalen Kern (Anwendungslogik und Daten) des Programms streng getrennt sein!

Model-View-Controller Muster (Model-View-Presenter)

- ⊙ **Model:** enthält die Anwendungslogik und die Daten. Das Model kennt weder die View noch den Controller.
- ⊙ **View:** Ist allein für die Darstellung der Daten des Models zuständig. Leitet die Nutzereingaben an den Controller weiter.
- ⊙ **Controller (Presenter):** enthält die Darstellungslogik der Anwendung. Nimmt die Eingaben der View entgegen, ruft Methoden auf dem Model auf und weist die View an, die resultierenden Änderungen der Model-Daten auf spezifische Art anzuzeigen.



Vorteile des MVC Architekturmusters:

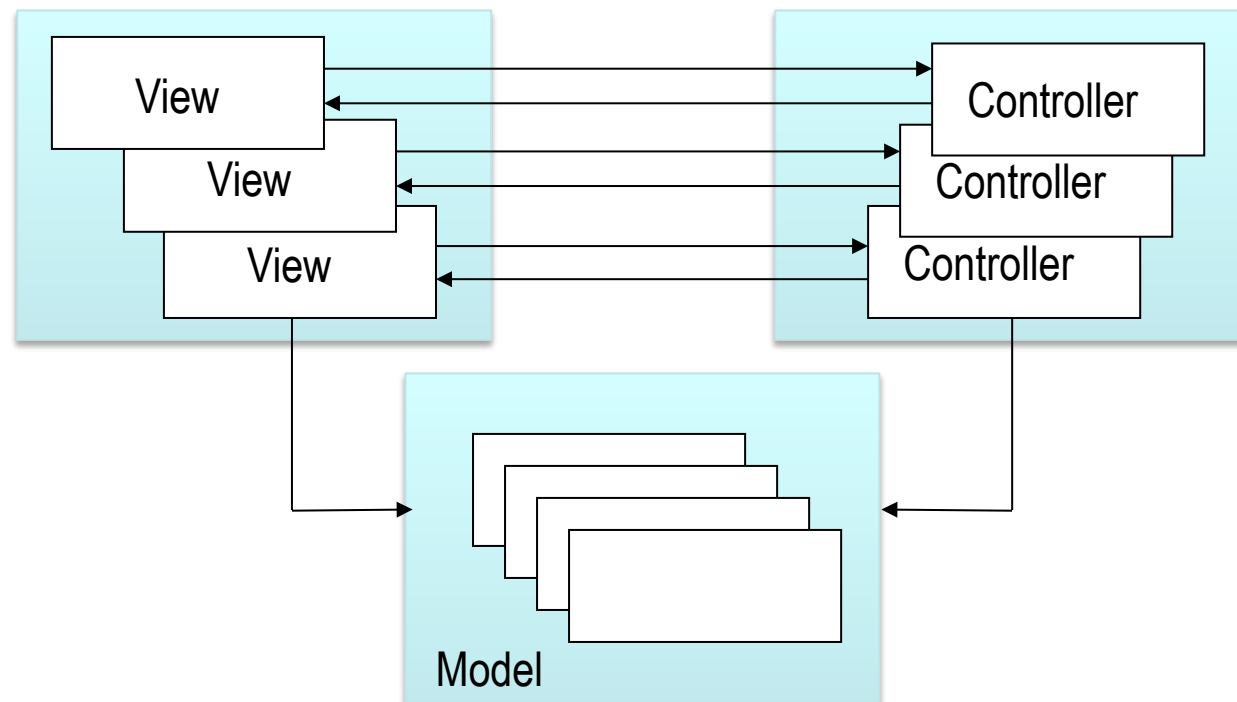
- ➦ Benutzerschnittstelle, die eher Änderungen unterliegt als das Model, lässt sich sehr leicht modifizieren oder austauschen, weil es keine Abhängigkeit vom Model zur View oder Controller gibt.
- ➦ Technologie der Benutzerschnittstelle lässt sich wechseln ohne dass der Rest des Programms von den Änderungen betroffen ist.
- ➦ Beliebige Sichten lassen sich hinzufügen, ohne dass der Rest des Programms angepasst werden muss.
- ➦ MVC Anwendungen sind leichter lesbar und leichter zu debuggen.

Bemerkungen:

- Die Begriffe Presenter und Controller werden oft synonym verwendet.

Wie lässt sich eine Java Anwendung nach MVC strukturieren?

- ⊙ Es gibt mehrere View Klassen.
- ⊙ Es gibt pro View eine Controller Klasse.
- ⊙ Das Model besteht aus mehreren Klassen: Klassen für die reine Datenhaltung und Klassen mit der Anwendungslogik.



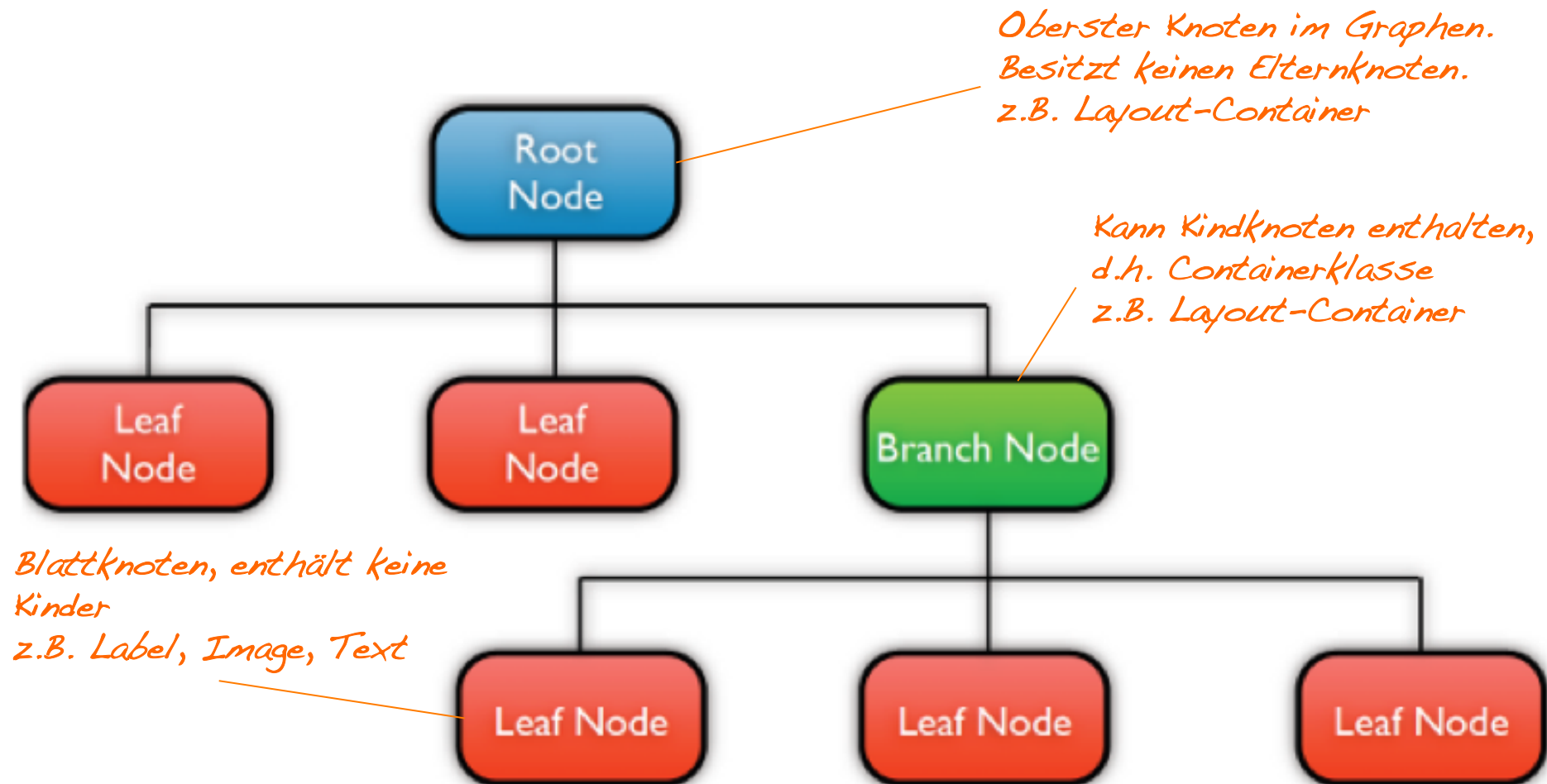
JavaFX

- ⊙ Plattform für die Entwicklung von Rich-Internet-Applications, die konsistent in unterschiedlichsten Umgebungen laufen.
- ⊙ API bietet Pakete für die vielseitige Gestaltung von multimedialen, grafiklastigen Benutzeroberflächen.
- ⊙ Unterstützt hardwarebeschleunigte Grafik
- ⊙ Enthält eine hochperformante Multimedia-Engine
- ⊙ Erlaubt die Einbettung von Web-Content in Anwendungen
- ⊙ FXML für die Spezifikation von GUIs (Trennung von View und Controller-Aspekten)
- ⊙ Scene Builder: Grafischer Editor für die Entwicklung von GUIs in FXML
- ⊙ API für die programmatische Umsetzung (oder Erweiterung) von GUIs
- ⊙ CSS3 für das Skinning und Layout von GUIs (Trennung von Struktur und Gestaltung von GUIs)
- ⊙ Enthalten in der aktuellen JDK 8

Übersicht



Eine GUI-Komponente wird als Szenegraph erstellt und verwaltet



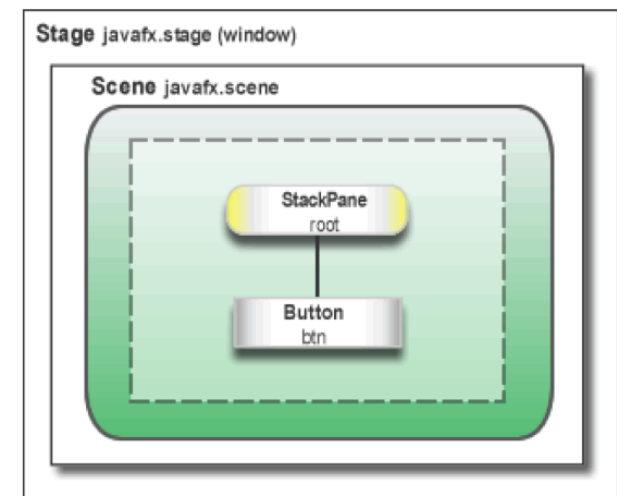
Startklasse einer JavaFX Anwendung

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    @Override
    public void start(Stage stage) {
        BorderPane root = new BorderPane();
        Button btn = new Button();
        btn.setText("Click Me");
        root.setCenter(btn);
        stage.setTitle("Hello World!");
        stage.setScene(new Scene(root, 300, 250));
        stage.show();
    }
}
```

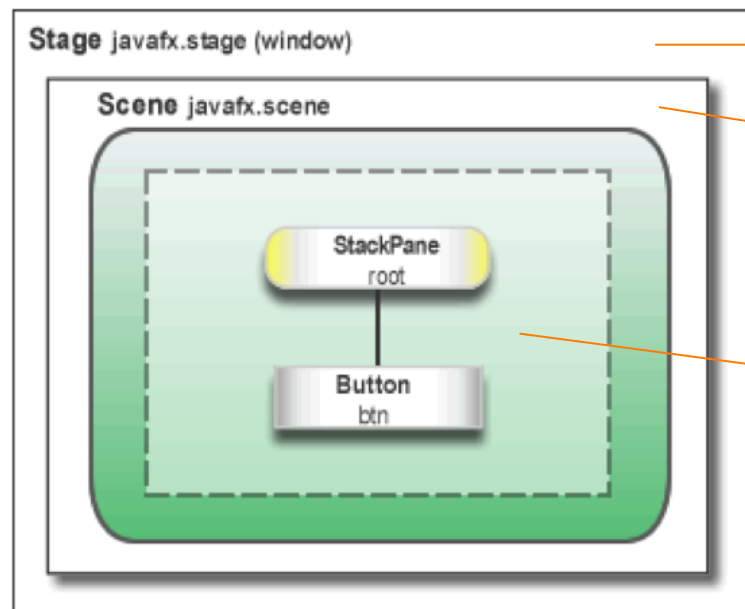
JavaFX-Anwendung ist von Application abgeleitet.

In main-Methode wird launch(..) aufgerufen.

In launch wird start(..) aufgerufen und die sog. Primary Stage übergeben.



Stage und Scene



Stage: Repräsentiert ein Fenster.

Scene: Im Fenster wird eine Scene angezeigt.

Scene Graph: Die Szene wird durch einen Scene Graph repräsentiert.

Stage

- ⊙ Eine Stage repräsentiert ein Fenster einer Anwendung
- ⊙ Die sog. Primary Stage wird durch die Virtual Machine erzeugt, sobald die Methode `launch(..)` aufgerufen wird.
- ⊙ Eine Stage zeigt ein Scene-Objekt an.
- ⊙ Der Fensterstatus (minimiert, Vollbild) und der Fenstertitel können über setter-Methoden des Stage-Objekts gesetzt werden

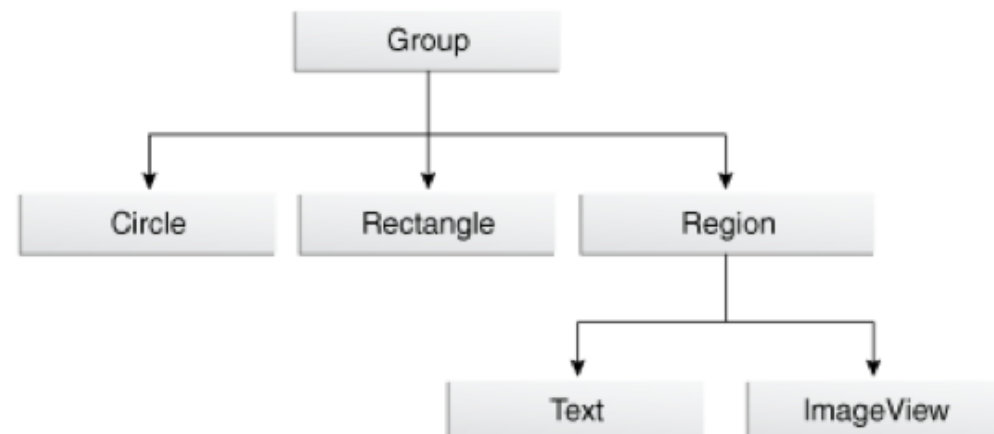


Scene

- ⊙ Eine Scene ist ein Container für die Elemente der GUI einer Anwendung (Szenegraph)
- ⊙ Sie legt fest, was wann wo angezeigt wird.

Szenegraph

- ⊙ Repräsentiert den Aufbau einer GUI in Form einer Baumstruktur
- ⊙ Die Knoten im Baum sind Instanzen von speziellen JavaFX Klassen.
- ⊙ Die Struktur definiert alle relevanten Details der GUI-Darstellung
 - ⊙ Welche Objekte angezeigt werden müssen
 - ⊙ Was wo platziert werden soll
 - ⊙ An welche Knoten Interaktionen des Nutzers weiter geleitet werden sollen
 - ⊙ Welche Bereiche neu gezeichnet werden müssen
 - ⊙ Wie Objekte am effizientesten gerendert werden



Erzeugen eines Szenegraphen: 2 Möglichkeiten

1. Programmieren: Aufbau von Szenegraphen mit Hilfe von Klassen aus der JavaFX-Klassenbibliothek.

```
@Override
```

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

```
    BorderPane root = new BorderPane();
```

```
    Button btn = new Button();
```

```
    btn.setText("Click Me");
```

```
    root.setCenter(btn);
```

```
    Scene scene = new Scene(root, 300, 250);
```

```
    stage.setTitle("Hello World!");
```

```
    stage.setScene(scene);
```

```
    stage.show();
```

```
}
```

Sog. Layout-Container, kann GUI-Komponenten aufnehmen

Hier wird ein Button-Objekt erzeugt und mit Aufschrift versehen

Dem Container-Objekt wird der Button hinzugefügt

Erzeugen eines Szenegraphen: 2 Möglichkeiten

2. Deklarieren: Szenegraphen durch FXML-Datei (View) spezifizieren ...

```
<?xml version="1.0" encoding="UTF-8"?>      ——— Datei Example.fxml  
  
<?import java.lang.*?>  
  
<?import java.util.*?>  
  
<?import javafx.scene.*?>  
  
<?import javafx.scene.control.*?>  
  
<?import javafx.scene.layout.*?>  
  
  
<BorderPane prefHeight="250.0" prefWidth="300.0" xmlns="http://javafx.com/  
    javafx/8" xmlns:fx="http://javafx.com/fxml/1">  
  
    <center>  
  
        <Button fx:id="button" text="Click Me!" BorderPane.alignment="CENTER" />  
  
    </center>  
  
</BorderPane>
```

Erzeugen eines Szenegraphen: 2 Möglichkeiten

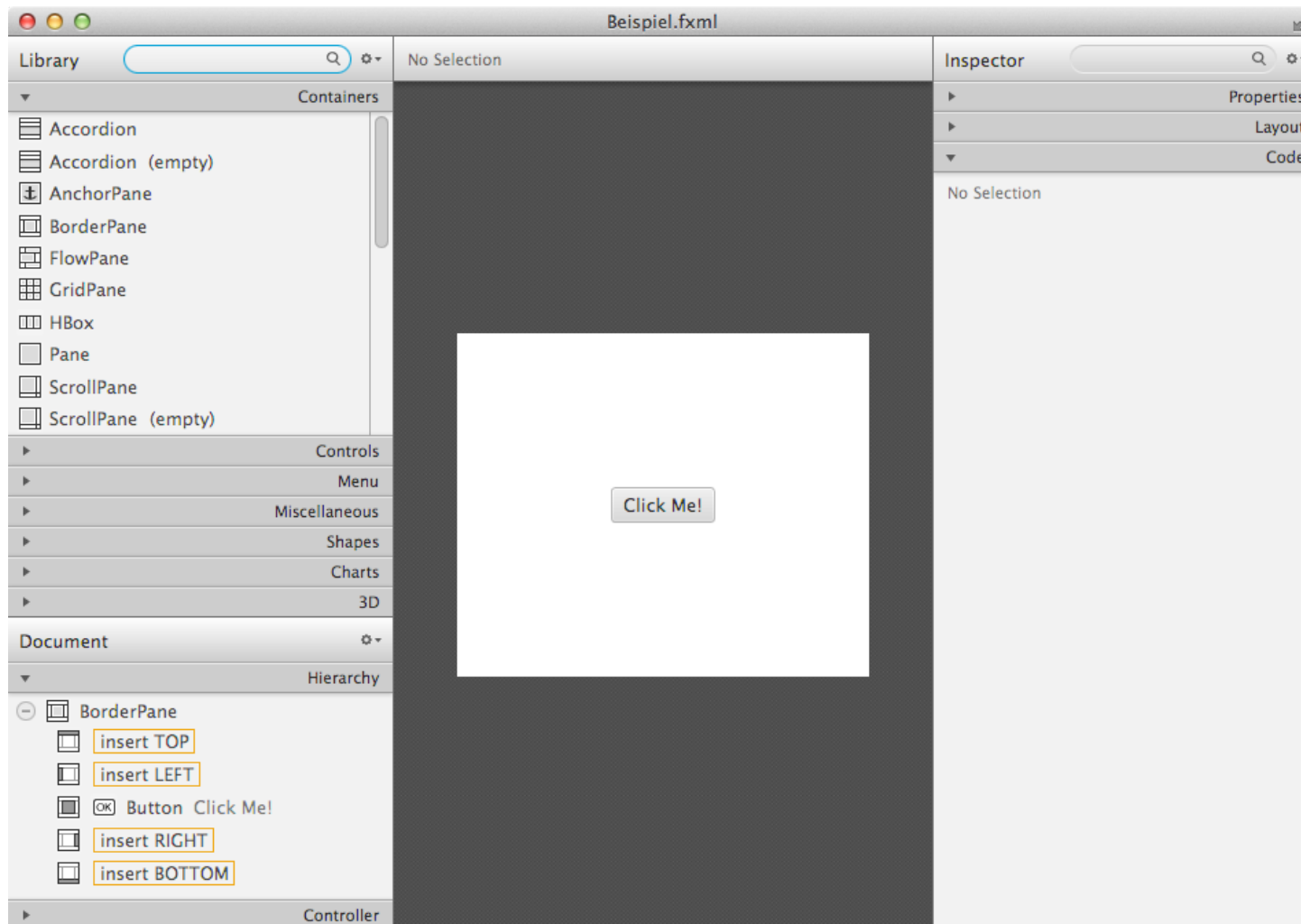
2. Deklarieren: ... und die Datei laden.

```
@Override
public void start(Stage stage) {
    Parent root = FXMLLoader.load(getClass().getResource("Example.fxml"));
    Scene scene = new Scene(root);
    stage.setTitle("Hello World!");
    stage.setScene(scene);
    stage.show();
}
```

*Methode load(..) erwartet
einen URI für die zu ladende
Datei.*

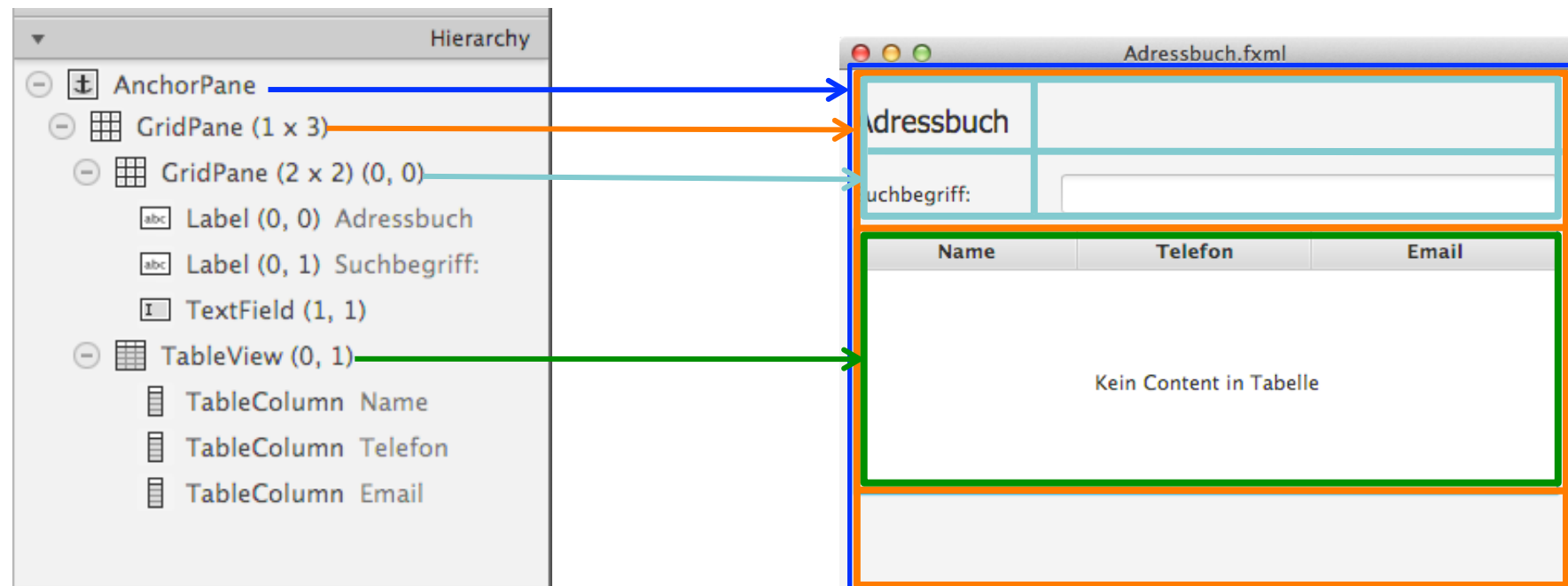
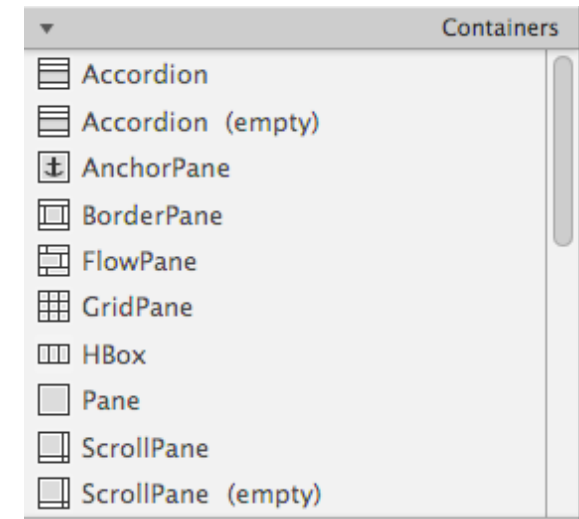
*Methode getResource(..)
liefert die URI für die
angegebene Datei.*

Scene Builder: GUI-basierte Erstellung von FXML-Views



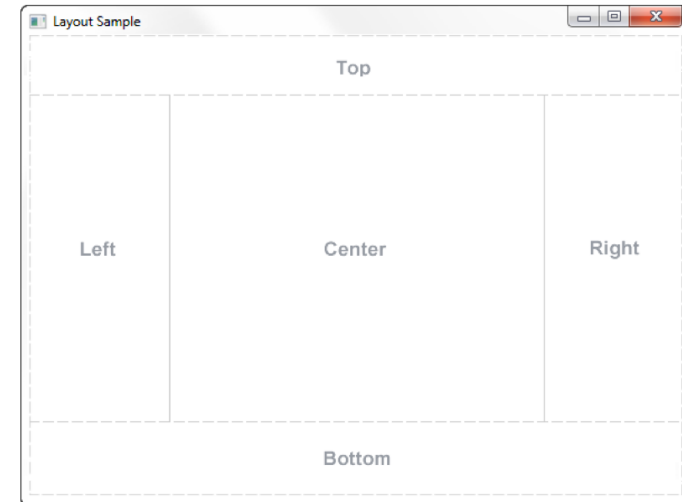
JavaFX Komponenten: Layout Container

- Ein Layout Container kann GUI-Elemente aufnehmen
- Jeder Layout Container ordnet die zugehörigen GUI-Elemente auf eine spezifische Weise an.
- Ein Layout-Container kann in einen anderen Layout Container eingefügt werden.

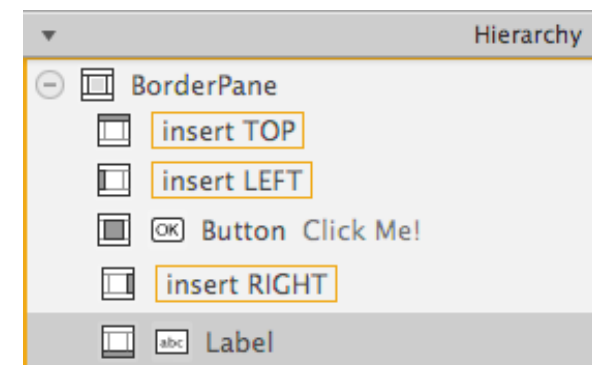


Layout Container: BorderPane

- ⦿ Layout-Bereich wird in fünf Bereiche aufgeteilt.
- ⦿ Jedem Bereich kann genau ein Element hinzugefügt werden (Container oder Komponente).
- ⦿ Ist das Fenster größer als die Elemente, die hinzugefügt wurden, so wird dem Bereich *Center* der verbleibende Platz zugeordnet.
- ⦿ Ist das Fenster kleiner als der benötigte Platz, so erfolgt eine Überlappung in Abhängigkeit der Reihenfolge, in der die Bereiche gefüllt wurden.

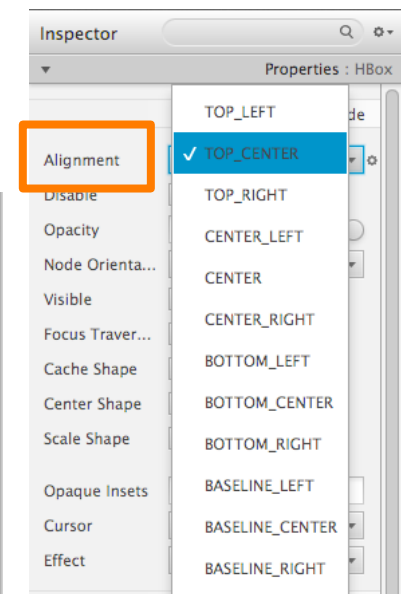
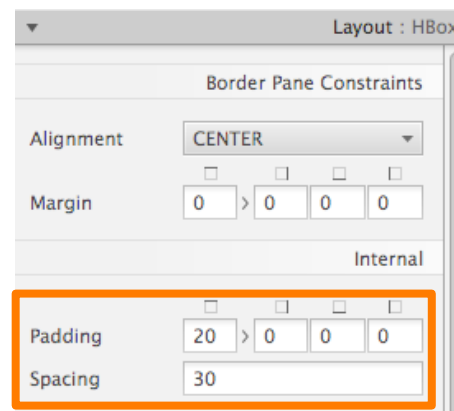
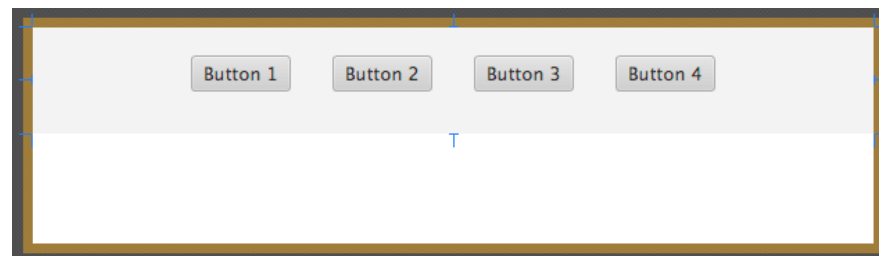
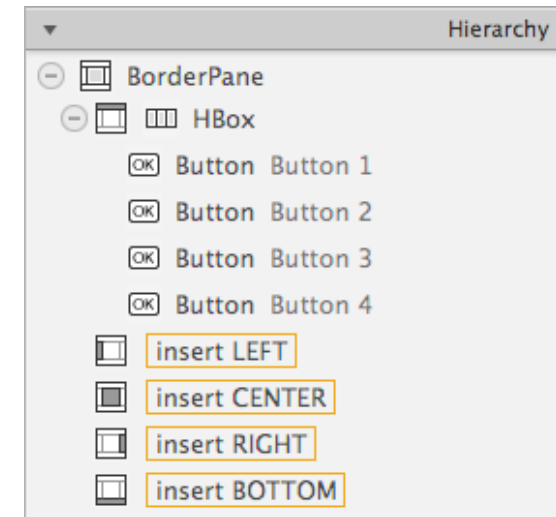


Bsp: Scene Builder



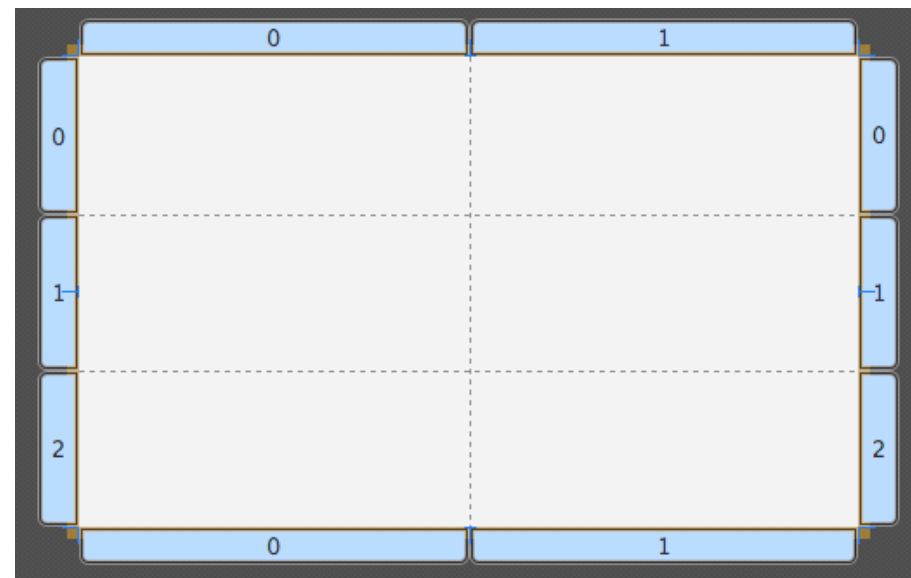
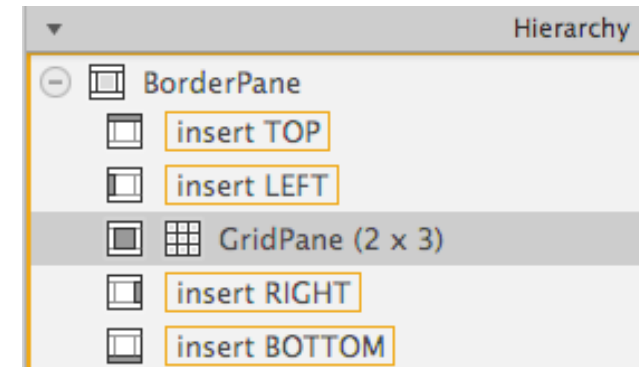
Layout Container: HBox und VBox

- ◉ Mehrere Elemente können hinzugefügt werden.
- ◉ Elemente werden horizontal (HBox) oder vertikal (VBox) in einer Linie angeordnet.
- ◉ Die Anordnung (**Alignment**) der Elemente in der Box kann eingestellt werden.
- ◉ Die Abstände zwischen den Elementen (**Spacing**) und die Abstände der Elemente zu den Seiten der Box (**Padding**) können eingestellt werden.



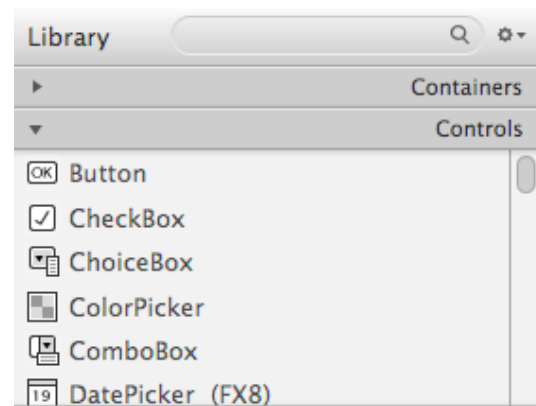
Layout Container: GridPane

- ⊙ Bereich wird in Spalten und Zeilen aufgeteilt (z.B. 2x3).
- ⊙ Spalten und Zeilen lassen sich in der Größe anpassen.
- ⊙ Jeder Zelle kann ein Element hinzugefügt werden (Container oder Komponente).
- ⊙ Spalten und Zeilen lassen sich im Kontextmenü einer selektierten Spalte oder Zeile hinzufügen



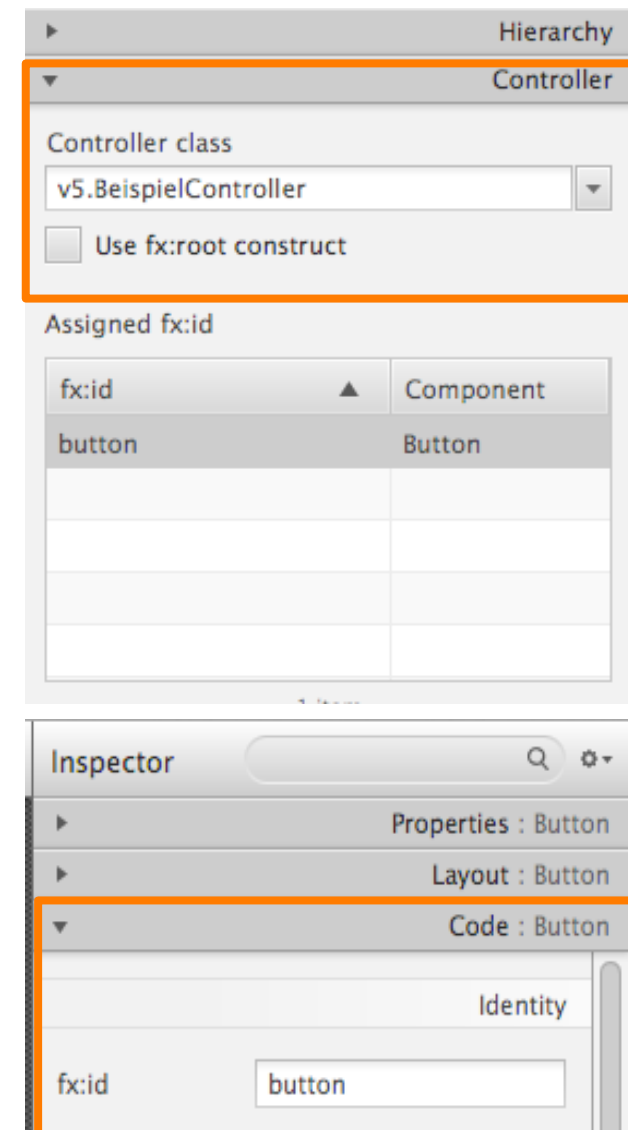
Scene Builder: Komponenten für die Nutzerinteraktion

- Die JavaFX Klassenbibliothek bietet eine Vielzahl an Steuerungselementen für Nutzerinteraktionen an. Diese stehen auch im Scene Builder zur Verfügung.



Scene Builder: Koppeln von Views mit Controller-Klassen

- Steuerungselemente einer View lassen sich für die Behandlung von Nutzerinteraktionen (Event Handling) oder der Erzeugung von dynamischen Inhalten (z.B. bei Tabellen) mit Attributen einer zugehörigen Controller-Klasse koppeln.
- Die View kann im Scene Builder mit einer Controller Klasse gekoppelt werden.
- Um in einer Controller-Klasse auf Knoten im Szenegraphen zugreifen zu können, müssen diese in der FXML-Datei mit einer *fx:id* versehen werden.
- Das lässt sich im Scene Builder im Inspector-Bereich unter *Code* eintragen.



Controller-Klassen

- ⊙ Eine Controller-Klasse muss das Interface *Initializable* implementieren.
- ⊙ Das Interface gibt eine Methode vor, die implementiert werden muss:

```
public void initialize(URL url, ResourceBundle rb)
```

- ⊙ Die Methode wird von der VM aufgerufen, wenn eine FXML-Datei geladen wird und die FXML-View mit der Controller-Klasse gekoppelt ist.
- ⊙ Alle mit der Annotation `@FXML` annotierten Attribute der Klasse bekommen die GUI-Objekte zugewiesen, die mit der zugehörigen *fx:id* versehen wurden und stehen damit in der Klasse zur Verfügung.

```
public class BeispielController implements Initializable {  
  
    @FXML  
    private Button button;  
  
    @Override  
    public void initialize(URL url, ResourceBundle rb) {  
        button.setText("Click Me!");  
    }  
}
```

Methode *initialize(..)*

- ⊙ In der Methode *initialize(..)* kann der Szenegraph programmatisch erweitert werden:
 - ⊙ Hinzufügen/Verändern von GUI-Elementen. So z.B. von dynamischen Inhalten (Tabellen- oder Menüinhalte), die erst berechnet oder geladen werden müssen.
 - ⊙ Koppeln von GUI-Komponenten mit Event Handlern

Bsp: Hinzufügen/Verändern von GUI-Elementen

```
public class BeispielController implements Initializable {  
  
    @FXML  
    private Button button;  
  
    @Override  
    public void initialize(URL url, ResourceBundle rb) {  
        button.setText("Click Me!");  
    }  
}
```


Koppeln von GUI-Elementen mit Event Handlern

- Bei GUI Elementen müssen sog. Event Handler angemeldet werden, die auf eine Interaktion reagieren.
- Für die erwarteten Event Handler sind in der Java API Interfaces vorgegeben, die eine Event Handler Klasse implementieren muss.
- Durch die Interfaces ist sichergestellt, dass speziellen Event Handler Klassen die Methoden bereit stellen, die von den GUI Elementen aufgerufen werden.

Bsp: Klasse *Button* besitzt eine Methode

```
public void setOnAction(EventHandler<ActionEvent> value)
```

Diese erwartet ein Objekt vom Typ *EventHandler<ActionEvent>*.

EventHandler<T extends Event> ist ein generisches Interface mit genau einer Methode:

```
void handle(T event)
```

Der Platzhalter *T* wird durch den konkreten Typ in spitzen Klammern ersetzt, hier *ActionEvent*

Event Handling in Controller-Klassen

Bsp: Klasse *ButtonHandler*

```
public class ButtonHandler implements EventHandler<ActionEvent>
{
    public void handle(ActionEvent event) {
        Button button = (Button) event.getSource();
        button.setText("Save");
    }
}
```

Eine Instanz dieser Klasse kann man beim Button als Event Handler setzen, weil die Klasse das erwartete Interface implementiert und daher die benötigte Methode besitzt:

```
button.setOnAction(new ButtonHandler());
```

Functional Interfaces

- In der Java-Bibliothek gibt es viele Interfaces, die lediglich eine einzige Methode vorgeben.
- Solche Interfaces bezeichnet man seit Java 8 als *Functional Interfaces*.
- D.h. um in einer Klasse *A* die eine Methodenimplementierung aufrufen zu können, die durch ein Functional Interface *I* vorgegeben ist, muss man eine Klasse *B* schreiben, die das Interface *I* implementiert und in der Klasse *A* eine Instanz der Klasse *B* für den Methodenaufruf nutzen.
- Diese unnötige Schreibaarbeit kann man sich bei *Functional Interfaces* sparen. Stattdessen kann eine Lambda Expression genutzt werden, um die aufzurufende Methode zu übergeben.

Event Handling in Controller-Klassen

Lambda Expressions statt Klassen

Bsp: Klasse *ButtonHandler*

```
button.setOnAction((ActionEvent e) -> handleButtonEvent(e));
```

Parameter die die Methode des erwarteten Functional Interface übergeben bekommt. Wenn es nur eine Version der Methode `setOnAction` gibt, können die Typangaben entfallen.

Methode, die aufgerufen werden soll. Sie kann die Parameter übernehmen, muss es aber nicht.

Typ muss mit dem übergebenen Wert übereinstimmen

```
public void handleButtonEvent(ActionEvent event) {  
    Button button = Button event.getSource();  
    button.setText("Save");  
}
```

Lambda Expression: Syntax

Parameterliste -> Ausdruck

Parameterliste:

- `x` -> Einzelner Parameter ohne Typ (Klammerung optional)
- `(x)` -> Einzelner Parameter ohne Typ (Klammerung optional)
- `(int x)` -> Parameter mit Typangabe (Klammerung obligatorisch)
- `(x, y)` -> Mehrere Parameter (Klammerung obligatorisch)
- `(int x, int y)` -> Mehrere Parameter (Klammerung obligatorisch)
- `()` -> Leere Parameterliste

Ausdruck:

- > `x + 1` Ausdruck, der ausgewertet werden soll
- > `{`
 - `anweisung1;` Sequenz von Anweisungen, die durchgeführt werden sollen
 - `anweisung2;` (immer in geschweiften Klammern)
 - `...`
 - `}`
- > `methode()` Angabe einer Methode, die ausgeführt werden soll (ohne Klammern)

Setzen von Event Handlern durch Lambda-Expressions

GUI-Komponenten können über sog. Lambda-Expressions mit Event Handlern gekoppelt werden (neu in Java 8)

```
public class ButtonViewController implements Initializable {  
  
    @FXML  
    Button button;  
  
    @Override  
    public void initialize(URL url, ResourceBundle rb) {  
        button.setText("Click Me!");  
        button.setOnAction((e) -> System.out.println("Hello World!"));  
    }  
}
```

*e: Event, das triggert, kann
mit oder ohne Typ angegeben
werden.*

*Was gemacht werden soll.
Bei mehrzeiligem Code muss
dieser in {}*

Setzen von Event Handler durch Lambda-Expressions

Alternativ kann eine Methode angegeben werden, die das Event konsumiert.

```
public class ButtonViewController implements Initializable {
```

```
    @FXML
```

```
        Button button;
```

```
    @Override
```

```
    public void initialize(URL url, ResourceBundle rb) {
```

```
        button.setOnAction((ActionEvent e) -> handleButton(e));
```

```
    }
```

```
    public void handleButton(ActionEvent event) {
```

```
        Button b = (Button) e.getSource();
```

```
        String txt = b.getText();
```

```
        System.out.println("Button gedrueckt: " + txt);
```

```
        System.out.println("Hello World!");
```

```
    }
```

```
}
```

*ActionEvent liefert Infos
über die Interaktion.*

Setzen von Event Handler durch Methodenreferenzen

Alternativ kann eine Methodenreferenz angegeben werden.

```
public class ButtonViewController implements Initializable {
```

```
    @FXML
```

```
    Button button;
```

```
    @Override
```

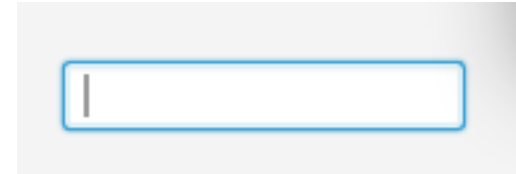
```
    public void initialize(URL url, ResourceBundle rb) {  
        button.setOnAction(this::handleButton);
```

*Hier MUSS der
Parameter hinein, den
der zugehörige Event
Handler erwartet.*

```
    public void handleButton(ActionEvent event) {  
        Button b = (Button) e.getSource();  
        String txt = b.getText();  
        System.out.println("Button gedrueckt: " + txt);  
        System.out.println("Hello World!");  
    }
```

```
}
```


TextField



- Ein *TextField* Objekt definiert ein einzeliges Texteingabefeld. Die zugehörige Klasse liefert Methoden für den lesenden und schreibenden Zugriff auf den Inhalt des Textfeldes:

```
public String getText()  
public void setText(String txt)
```

- Ein *TextField* Objekt kann mit einem Event Handler versehen werden, der ausgelöst wird, wenn der Nutzer seine Eingabe mit der Return-Taste beendet.

```
textField.setAction((ActionEvent e) -> handleTextEvent(e));
```

- Ein *TextField* Objekt kann mit einem Event Handler versehen werden, der ausgelöst wird, wenn der Text im Feld sich ändert (ohne Return).

```
textField.textProperty().addListener((e) -> handleEditAction());
```


- Der Inhalt eines TextField Objekt kann gelöscht werden.

```
public void clear()
```

TextField

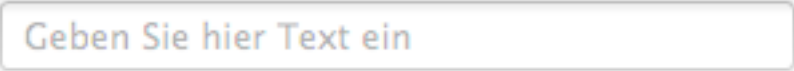
- Ein *TextField* Objekt kann editierbar oder nicht-editierbar gesetzt werden (Default: true).

```
public void setEditable(boolean e)
```



- Ein *TextField* Objekt kann mit einem Text unterlegt werden, der bei nicht selektiertem und leerem Textfeld angezeigt wird, um zu verdeutlichen, was hier eingetragen werden soll.

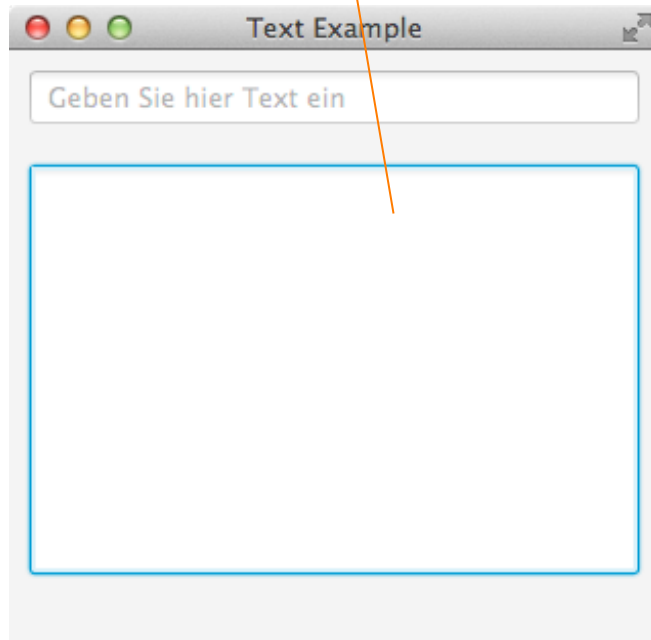
```
public void setPromptText(String text)
```



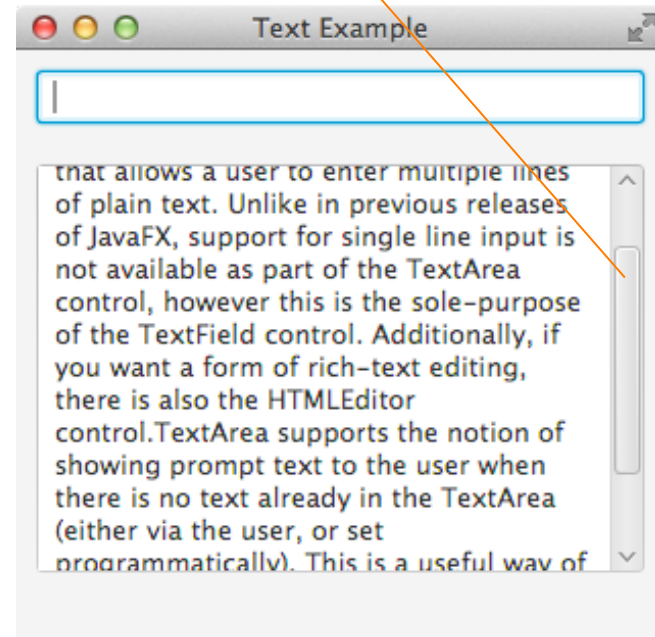
TextArea

- Ein *TextArea* Objekt definiert ein mehrzeiliges Textfeld. Sollte der Inhalt des Feldes dessen Größe überschreiten, so werden Scrollbars eingeblendet.

*TextArea
Scrollbars werden nur bei
Bedarf angezeigt.*



Anzeige vertikaler Scrollbar.



TextArea

- Die zugehörige Klasse liefert Methoden für den lesenden und schreibenden Zugriff auf den Inhalt der *TextArea*:

```
public String getText()  
public void setText(String txt)
```

- Bei einem *TextArea* Objekt kann Text ans Ende angehängt werden:

```
public void appendText(String txt)
```

- Bei einem *TextArea* Objekt kann der Inhalt umgebrochen werden (Default *false*).

```
public void setWrapText(boolean w)
```

- Ein *TextArea* Objekt kann nicht-editierbar bzw. editierbar gesetzt werden (Default *true*).

```
public void setEditable(boolean e);
```