TEKO
SCHWEIZERISCHE FACHSCHULE

Objektorientiertes Programmieren 2

Grafical User Interface 2

Stephan Kessler, BSc in Systems Engineering

Kontakt: stephan.kessler@edu.teko.ch

Vorbereitung

Lesen Sie folgende Abschnitte im Buch ***Sprechen Sie Java?*** als Vorbereitung für dieses Thema. Beachten Sie, dass bei Angaben von Unterkapiteln nur diese auch gelesen werden müssen:

- Pakete (S. 237-254)

Package

In Java kann es vorkommen, das Klassennamen mehrmals in unterschiedlichen Klassen verwendet werden. Z.B. existiert in der Java-Bibliothek die Klasse *Color* mehrfach (j*ava.awt.Color* und *javafx.scene.paint.Color*). Dank dem Package, welcher Sie schon beim *import*-Schlüsselwort verwendet haben, ist es möglich, genau anzugeben, welche Klasse gemeint ist. Durch das Package wird ein sogenannter *Namensraum* erstellt.

Package

```java
package bereich;

public class Test{
    public String text = "Hallo";

    public void printText(){
        System.out.println(text);
    }
}
```

Um ein Package für eine Klasse zu bestimmen, muss das Schlüsselwort ***package*** in der obersten Zeile von der .java-Datei verwendet werden. Anschliessend folgt der Name des Bereiches.

In diesem Beispiel liegt die Klasse *Test* im Package *bereich*. Wenn man diese Klasse «importieren» möchte, würde der Import-Befehl wie folgt aussehen: *import bereich.Test;*

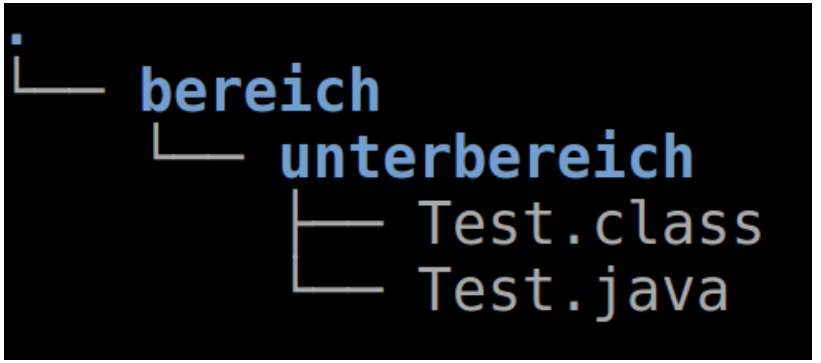
Package

```
package bereich.unterbereich;

public class Test{
    public String text = "Hallo";

    public void printText(){
        System.out.println(text);
    }
}
```

Dabei können auch Unterbereiche erstellt werden. Dazu werden die Bereichsnamen durch einen Punkt getrennt.

Package

```java
package bereich.unterbereich;

public class Test{
    public String text = "Hallo";

    public void printText(){
        System.out.println(text);
    }
}
```



Für das Ausführen des Programmes ist es von Vorteil, wenn die Klassen in der selben Ordnerstruktur bestehen, wie die Package-Struktur definiert ist, da standardmässig die Klassen nach dieser Struktur von der Java-Runtime-Environment gelesen werden.

Viele IDEs können diese Struktur automatisch aufbauen.

Package Konvention

ch.gruppenname.projektname

Um auch eine Überschneidung bei den Package-Namen zu vermeiden, wird eine Namens-Konvention verwendet. Dabei baut diese auf dem Prinzip der Domain-Namen auf. Wenn Sie ein Website für Ihr Projekt erstellen, würden Sie die Domain entsprechend mit *projektname.gruppenname.ch* nennen.

In Java wird diese Struktur in der umgekehrten Form als Package verwendet: *ch.gruppenname.projektname*

Beispiel: Die Game-Klasse aus der Programm-Bibliothek LibGDX (gdx) der Gruppe Badlogic besitzt den Package-Name: *com.badlogic.gdx*

Zugriffsmodifikatoren

Access Modifier	Zugang in der Klasse	Zugang in der package?	Zugang ausserhalb Package durch die Sub-Klasse?	Zugang ausserhalb der Klasse und nicht in der Sub-Klasse?
public	X	X	X	X
protected	X	X	X	
Default	X	X		
private	X			

In Java gibt es 4 Zugriffsmodifikationen, die beschreiben ob ein Zugriff in einer gewissen Situation möglich ist oder nicht. Dies erlaubt Programmsegmente vor fehlerhaften oder unnötigen Zugriffen zu schützen.

Diese Modifikationen lassen sich an Klassen, Schnittstellen, Konstruktoren, Methoden und Attribute verwenden.

Zugriffsmodifikatoren public

Access Modifier	Zugang in der Klasse	Zugang in der package?	Zugang ausserhalb Package durch die Sub-Klasse?	Zugang ausserhalb der Klasse und nicht in der Sub-Klasse?
public	X	X	X	X

```
package test;

public class Test{
    public String text = "Hallo";

    public void printText(){
        System.out.println(text);
    }
}
```

Programmsegmente mit ***public*** beschrieben, können von überall zugegriffen werden:

- Ein Objekt aus der Klasse *Test* lässt sich von überall instanziieren.
- Auf das Attribut *text* kann von überall zugegriffen werden.
- Die Methode *printText()* lässt sich von überall aufrufen.

Zugriffsmodifikatoren protected

Access Modifier	Zugang in der Klasse	Zugang in der package?	Zugang ausserhalb Package durch die Sub-Klasse?	Zugang ausserhalb der Klasse und nicht in der Sub-Klasse?
protected	X	X	X	

```
package test;

protected class Test{
    protected String text = "Hallo";

    protected void printText(){
        System.out.println(text);
    }
}
```

Programmsegmente mit ***protected*** beschrieben, können von überall zugegriffen werden, wenn diese geerbt werden oder die zugreifende Klasse im Package-Bereich liegt:

- Eine Klasse besitzt eine protected-Methode. Wird diese Klasse auf eine weitere Klasse vererbt, kann dort auch auf die protected-Methode zugegriffen werden. Zudem kann auf diese Methode zugegriffen werden, wenn die Klasse in dem selben Packaga-Bereich ist.
- In diesem Beispiel kann *Test* nur dann in einer Klasse instantiiert werden, wenn diese Klasse auch im Package *test* liegt. (kein Nested-Code).

Zugriffsmodifikatoren Default

Access Modifier	Zugang in der Klasse	Zugang in der package?	Zugang ausserhalb Package durch die Sub-Klasse?	Zugang ausserhalb der Klasse und nicht in der Sub-Klasse?
Default	X	X		

```
package test;

class Test{
    String text = "Hallo";

    void printText(){
        System.out.println(text);
    }
}
```

Programmsegmente ohne ein Schlüsselwort („Default“), können in der eigenen Klasse oder in erbenden Klassen verwendet werden:

- Eine Klasse die eine „Default“-Methode besitzt, kann diese Methode in einer weiteren eigenen Methode verwenden. Wird diese Klasse auf eine weitere Klasse vererbt, kann dort auch auf die „Default“-Methode zugegriffen werden.

Zugriffsmodifikatoren private

Access Modifier	Zugang in der Klasse	Zugang in der package?	Zugang ausserhalb Package durch die Sub-Klasse?	Zugang ausserhalb der Klasse und nicht in der Sub-Klasse?
private	X			

```
package test;

private class Test{
    private String text = "Hallo";

    private void printText(){
        System.out.println(text);
    }
}
```

Programmsegmente mit **private** beschrieben, können **nur** in der eigenen Klasse verwendet werden.

Aufgabe

```java
package test;

public class Test{
    public String text = "Hallo";
    public int value = 12;

    public void printText(){
        System.out.println(text);
    }

    public void add(int i){
        printText();
    }
}
```

- Erstellen Sie die oben dargestelle Klasse *Test*.
- Erstellen Sie eine weitere Klasse *Test2,* die von der Klasse *Test* erbt und im selben Package-Bereich ist. Überschreiben Sie die Methode *add* mit der Logik, dass der Parameter *i* auf den Wert *value* addiert wird (ohne *printText())*.
- Erstellen Sie eine weitere Klasse *Test3*, die in Package-Bereich *test3* liegt und auf das Attribut *text* durch ein instanziiertes Objekt von *Test* in einer eigenen Methode zugreift.
- Erstellen Sie eine weitere Klasse *Test4,* die von der Klasse *Test* erbt und im Package-Bereich *test4* ist. Überschreiben Sie die Methode *add* mit der Logik, dass der Parameter *i* an den String *text angehängt* wird (ohne *printText())*.

Aufgabe

```java
package test;

public class Test{
    public String text = "Hallo";
    public int value = 12;

    public void printText(){
        System.out.println(text);
    }

    public void add(int i){
        printText();
    }
}
```

- Überlegen Sie sich nun in Gedanken, welche Zugriffmodifikatoren in der Klasse *Test* mit einer “niedrigeren” (weniger Zugriffsmöglichkeiten) Modifikator ersetzt werden können, so, dass alle beschrieben Zugriffe noch möglich sind.
- Testen Sie ihre Annahme im Code. Erstellen Sie auch eine Main-Methode, die die Funktionalität der einzelnen Klassen testet. Überprüfen Sie das Ergebnis.

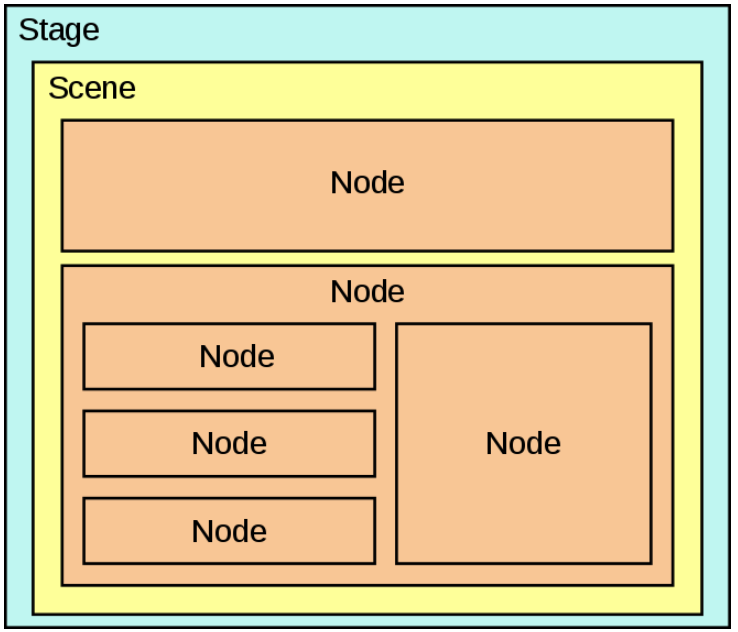
JavaFX

- AWT
- Swing (JFC)
- JavaFX

Wenn Sie in Java eine GUI-Applikation schreiben wollen, stehen Ihnen grundlegend 3 Programm-Bibliotheken zur Verfügung:

- **AWT** Die erste Bibliothek zur Erstellung von GUI-Applikationen. Sie ist relativ schlicht gehalten, wodurch komplexere Anwendungen nur schwer Umsetzbar sind.
- **Swing** Nachfolger von AWT welcher teils auf AWT-Elemente aufbaut. Die bisherigen GUI-Aufgaben wurden von Ihnen in der Swing-Umgebung erstellt. Der Funktionsumfang in Swing ist sehr gross, jedoch teils sehr komplex.
- **JavaFX** Nachfolger von Swing, welcher die Komplexität verringern soll und trotzdem für komplexeren Anwendungen zu gebrauchen ist. Erlaubt «neuartige» Elemente, wie Animationen, Videos usw.

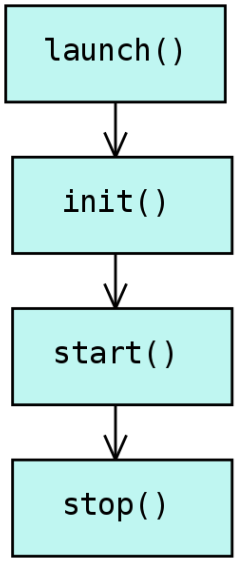
JavaFX Aufbau



In JavaFX ist der Aufbau eines Fensters an die Struktur eines Theaters angelehnt. Die Bühne (**Stage**) führt verschiedene Szenen (**Scene**) durch. In diesen Szenen tauchen Schauspieler (**Node**) auf, die vom Betrachter gesehen werden.

Eine JavaFX-Anwendung besitzt also eine **Stage** die wiederum eine **Scene** besitzt. Die Scene selber enthält **Nodes**, welche wiederum weitere **Nodes** enthalten können.

JavaFX

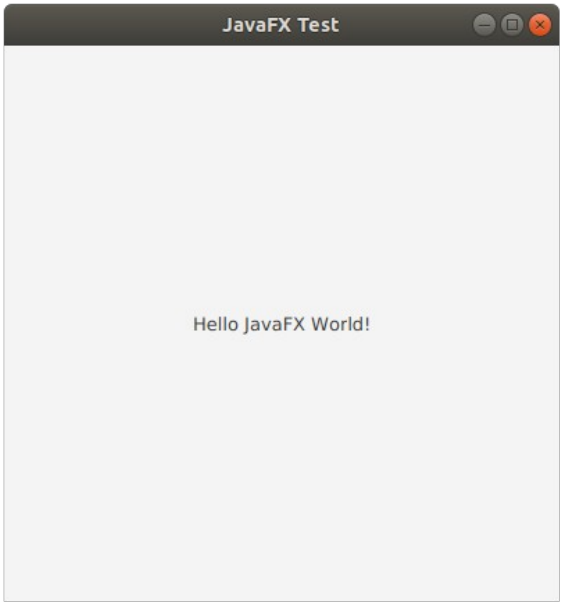


JavaFX-Anwendungen erweitern die Basisklasse *Application*, die Lebenszyklus-Methoden wie *init()*, *start()* oder *stop()* vererbt. Diese Methoden können in der JavaFX-Anwendung überschrieben werden. Der JavaFX-Launcher, welcher mit der Methode *launch()* ausgeführt wird, kümmert sich darum, dass diese entsprechend aufgerufen werden.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Im nebenstehenden Beispiel wird folgendes JavaFX-Fenster erstellt:



JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Die eigene statische *main(String[])*-Methode leitet an die statische *launch(String[])*-Methode der *Application*-Klasse weiter und übergibt alle Aufrufparameter.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Dadurch wird die Methode *start()* aufgerufen. JavaFX übergibt der Methode eine ***Stage***, was etwa der Aufgabe eines Haupt-Containers entspricht.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Über die ***Stage*** kann der Titel des Fensters gesetzt werden.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Nun wird die ***Scene*** vorbereitet. Dazu werden auch die entsprechenden «Schauspieler» bzw. ***Nodes*** instanziiert.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Label ist ein ***Node*** um einen Text auf dem Bildschirm darzustellen.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Wie Sie bereits kennen gelernt haben, können *Nodes* auch weitere *Nodes* beinhalten. So sind die Layout-Elemente auch *Nodes*. Im nebenstehenden Beispiel wird ein StackPane vom Typ ***Node*** erstellt. Dieser Zeigt seine *Nodes* «aufgestapelt» an.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

In diesem Schritt erhält der *StackPane* den *Node Label*.

JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Da alle «Schauspieler» vorbereitet sind, kann nun die Szene (*Scene*) erstellt und die Entsprechenden *Nodes* hinzugefügt werden. Die Szene gibt auch die grösse des Fensters an. In diesem Beispiel wird ein 400x400 Fenster erzeugt.

JavaFX Beispiel

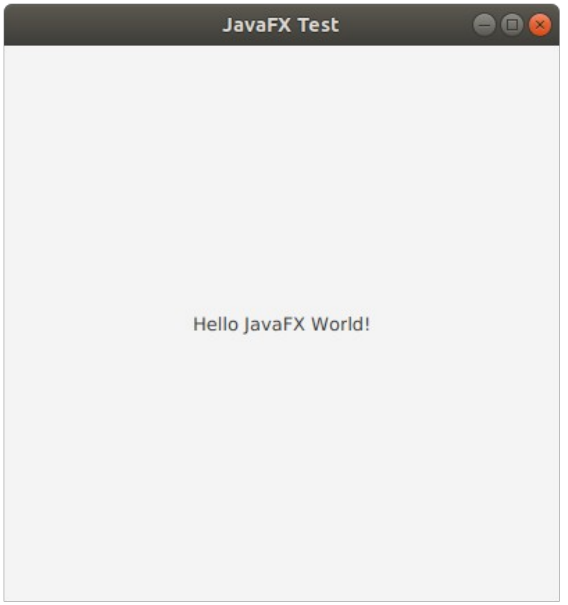
```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Mit dem Befehl primaryS*tage.setScene(scene)* kommt der Szenengraph über die Klasse *Scene* auf die Bühne. Zu jedem Zeitpunkt sind alle grafischen Objekte einer JavaFX-Anwendung präsent, d. h. sie existieren nicht nur zum Zeitpunkt des Zeichnens.

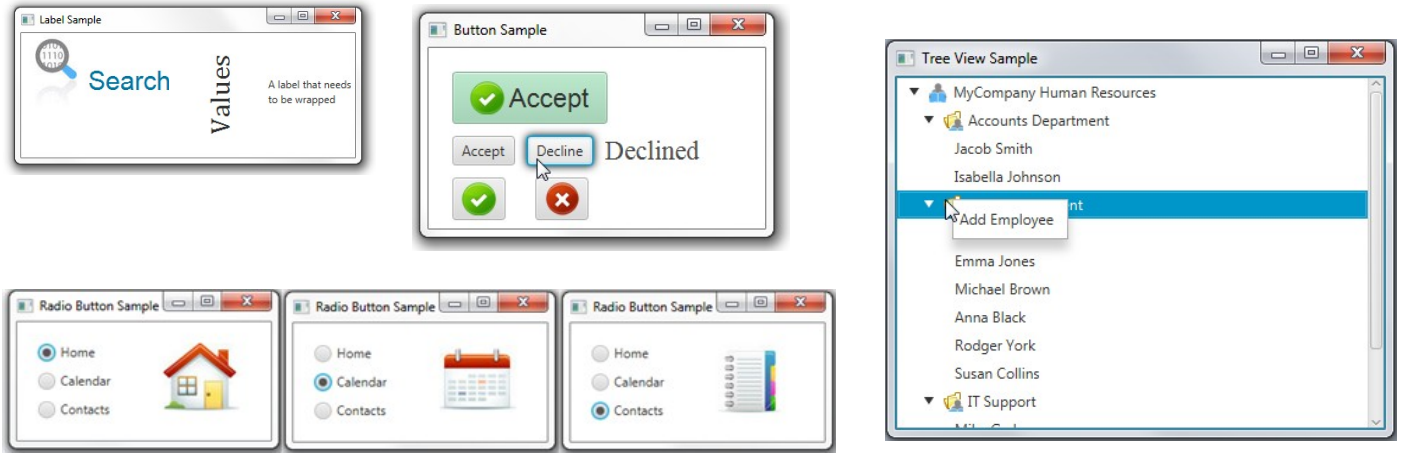
JavaFX Beispiel

```
public class Test extends Application{
    public static void main(String[] args) {
        launch(args);
    }

    public void start(Stage primaryStage){
        primaryStage.setTitle("JavaFX Test");
        Label label = new Label("Hello JavaFX World!");

        StackPane stack_pane = new StackPane();
        stack_pane.getChildren().add(label);
        Scene scene = new Scene(stack_pane, 400, 400);

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

Mit primaryS*tage.show()* wird zum Schluss das Fenster dargestellt:



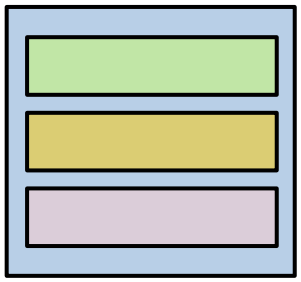
JavaFX Node Controls



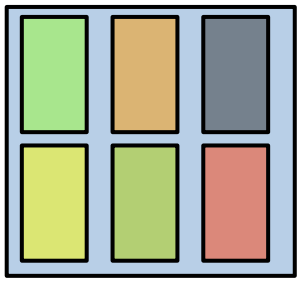
Die interaktiven Fensterelemente (z.B. Knopf) werden in JavaFX als **Controls** genannt. Folgende Beispiele stehen Ihnen in zur Verfügung: *Label*, *Hyperlink*, *Button*, *RadioButton*, *ToggleButton*, *CheckBox*, *ChoiceBox*, *TextField*, *TextArea*, *ListView*, *TableView*, *TitledPane*, *ScrollPane*

Eine Übersicht mit entsprechender Beschreibung der Elemente finden Sie hier:
https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm#JFXUI336

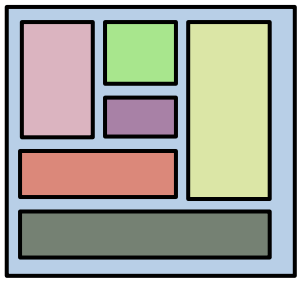
JavaFX Node Layouts



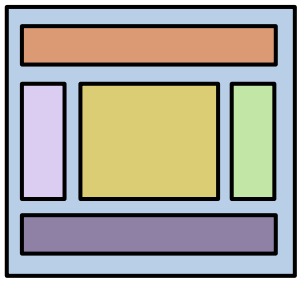
VBox



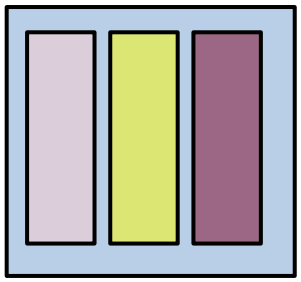
TilePane



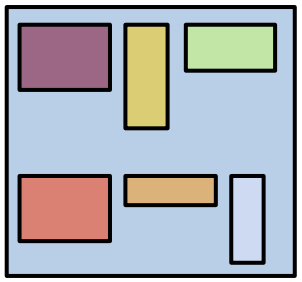
GridPane



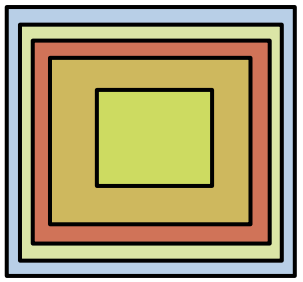
BorderPane



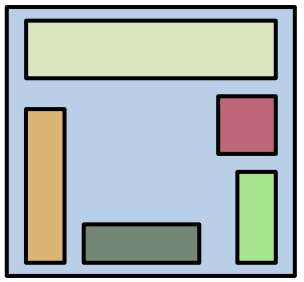
HBox



FlowPane



StackPane



AnchorPane

In der oberen Darstellung sind die verschiedenen Layout-Varianten und deren Eigenschaften aufgeführt.

Eine genauere Beschreibung der Layout-Klassen finden Sie unter folgendem Link:
https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

JavaFX Aktionen

```
Button button = new Button("Ok");

button.setOnAction(new EventHandler<ActionEvent>(){
    @Override
    public void handle(ActionEvent e) {
        System.out.println("Button wurde gedrückt");
    }
});
```

Um eine Aktion zu beschreiben (z.B. was passieren soll, wenn ein Knopf gedrückt wird), können Sie über die entsprechende Methode einen *EventHandler* übergeben.

Im oberen Beispiel ist dies anhand eines Knopfes dargestellt. Dabei wird zusätzlich als generischer Typ *ActionEvent* angegeben, welcher Ihnen in der *handle*-Mehtode zur Verfügung steht.

JavaFX Aktionen

```
Button button = new Button("Ok");

button.setOnAction((ActionEvent) -> {
    System.out.println("Button wurde gedrückt");
});
```

Beachten Sie dabei, dass Sie auch die oben dargestellte Schreibweise antreffen können.

Dabei handelt es sich um einen **Lambda-Ausdruck**. Beide Varianten sind programmiertechnisch gleich. Diese Variante ist jedoch kürzer und nur Möglich, da die Klasse *EventHandler* nur **eine** abstrakte Methode aufweist.

Dieser Kurs wird jedoch nicht die Lamba-Ausdrücke weiter behandeln bzw. prüfen. Diese Information dient daher nur zur Vorbeugung von Unklarheiten, falls Sie solche Programmzeilen antreffen sollten.

JavaFX Bindings

Bis anhin haben Sie in der Swing-Bibliothek bei Veränderungen von Zuständen alle grafischen Elemente mit einem *repaint()* neu zeichnen lassen. Dies ist in JavaFX nicht mehr nötig, da die JavaFX-Elemente automatisch bei Veränderungen sich grafisch anpassen.

In JavaFX können jedoch Eigenschaften (*Properties*) miteinander verknüpft werden. Diese Verknüpfung erlaubt es eine automatische Anpassung über mehrere Elemente.

JavaFX Bindings

Im oberen Beispiel wurde der Text aus dem Textfeld von «Hallo» zu «Hallo Welt» erweitert. Das nebenstehende Label passt sich dabei automatisch und während dem Tippen direkt an.

JavaFX Bindings

```
Label label = new Label();
TextField field = new TextField();

label.textProperty().bind(field.textProperty());
```

Dies wird mit folgendem Programmcode erreicht. Dabei wird die Text-Eigenschaft des Labels über dessen Methode *bind()* mit der Eigenschaften des Textfeldes verknüpft. Dies ist möglich da die JavaFX-Eigenschaften die *Observable*-Schnittstelle implementiert haben. Diese Struktur erlaubt die automatische Anpassung.

Da in diesem Beispiel eine Anpassung am Textfeld nur das Label anpasst, handelt es sich hier um ein **unidirektionale Bindung**.

JavaFX Bindings

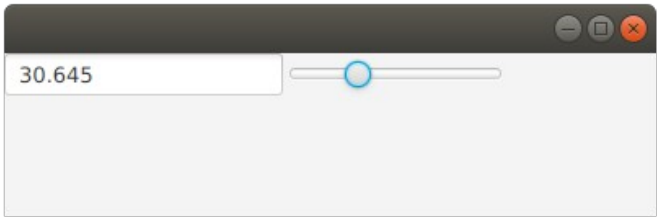
```
Label label = new Label();
TextField field = new TextField();

label.textProperty().bindBidirectional(field.textProperty());
label.setText("value");
```

Ist eine automatische Anpassung in beiden Richtungen erwünscht, kann dies mit der Methode *bindBidirectional()* erreicht werden.

Im oberen Beispiel wird nun durch die letzte Zeile auch das Textfeld auf «value» gesetzt, obwohl dies über das Label geschieht. Veränderungen im Textfeld haben immer noch den Effekt das Label entsprechend zu verändern.

Diese Variante nennt man **bidirektionale Bindung**.

JavaFX Bindings



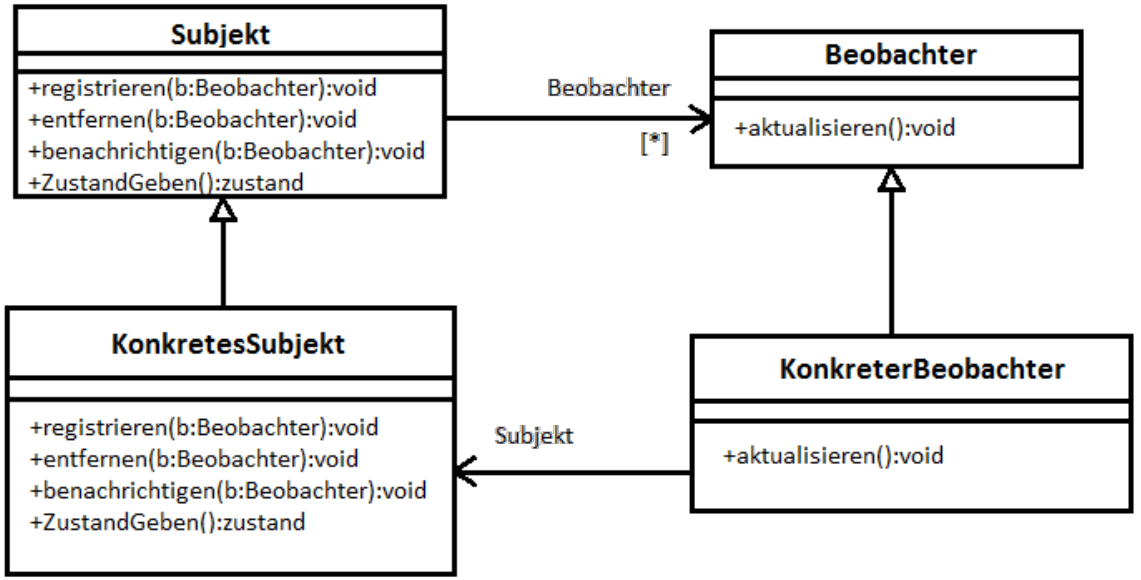

```
TextField field = new TextField();
Slider slider = new Slider();

NumberStringConverter converter = new NumberStringConverter();
field.textProperty().bindBidirectional(slider.valueProperty(), converter);
```

Da nicht jede Eigenschaft direkt übernommen werden kann, können auch Converter-Klassen verwendet werden.

Im oberen Beispiel wird ein Text-Feld mit einem Slider verbunden. Da das Textfeld einen String als Eigenschaft aufweist und der Slider die Position jedoch als Zahl interpretiert, ist hier eine Umwandlung von Text zu Zahl notwendig.

Beim Befehl *bindBidirectional()* kann das Objekt *NumberStringConverter* zusätzlich übergeben werden, wodurch eine Umwandlung bei einer Veränderung statt findet.

JavaFX Bindings



Wie bereits Erwähnt, sind diese Bindungen wegen den *Observable*-Schnittstellen möglich. Diese Strukturen enthalten eine Benachrichtigungsfunktionalität: Passiert etwas, werden die Beobachter benachrichtigt und deren Aktualisierungsmethode durchgeführt.

In JavaFX existieren weitere Beobachter-Klassen: *ObservableList*, *ObervableSet*, *ObservableMap* usw.