

Informatik 2

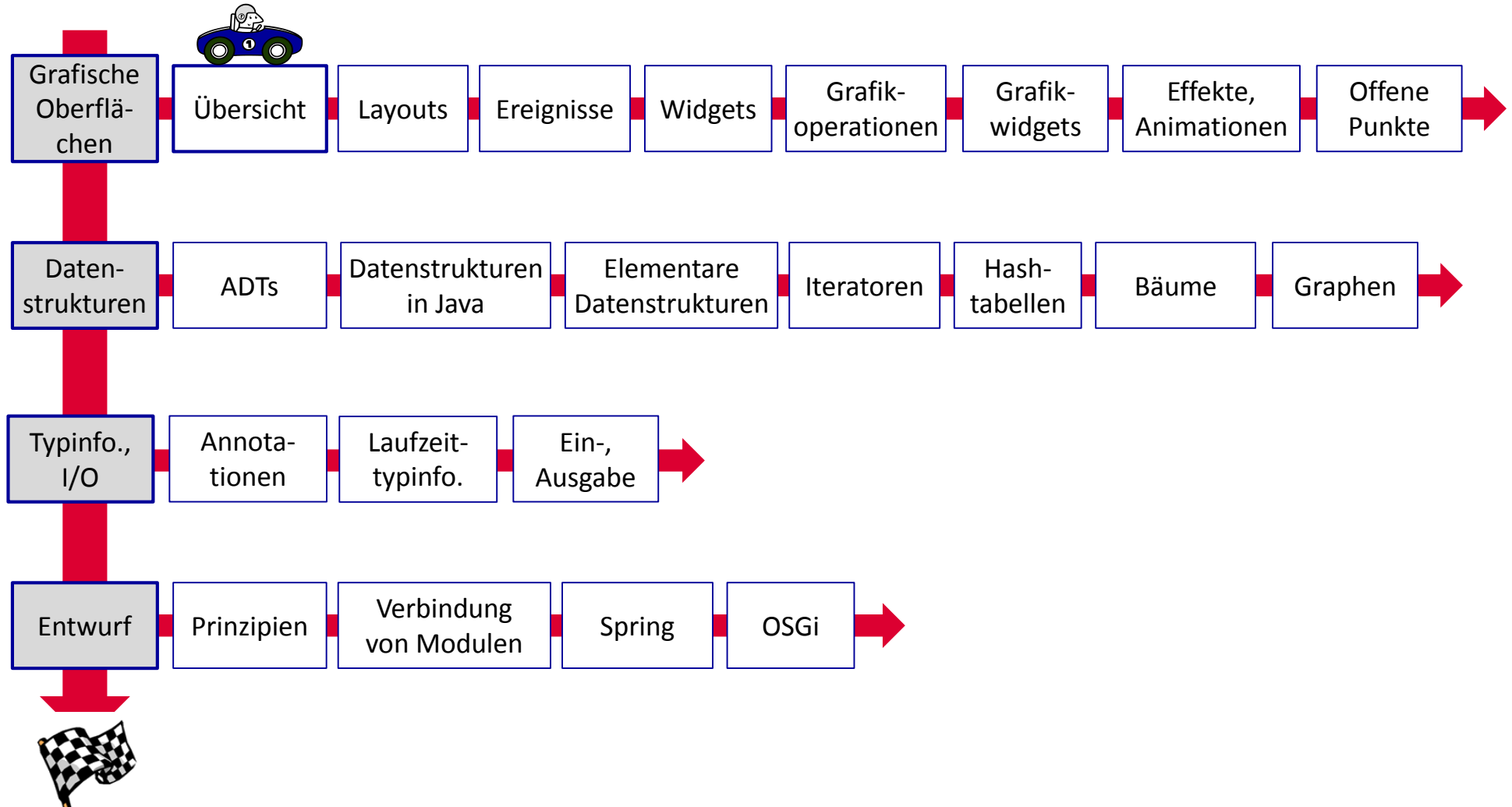
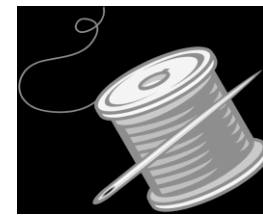
Grafische Oberflächen und Animationen mit JavaFX

Prof. Dr.-Ing. Holger Vogelsang
holger.vogelsang@hs-karlsruhe.de



- [Übersicht \(3\)](#)
- [Layouts \(12\)](#)
- [Ereignisbehandlung \(34\)](#)
- [Widgets \(38\)](#)
- [Grafikoperationen \(49\)](#)
- [Grafikwidgets \(63\)](#)
- [Effekte und Animationen \(82\)](#)
- [Offene Punkte \(94\)](#)

Übersicht





- *Wie lassen sich sehr einfache Oberflächen erstellen?*
- *Wie kann ein Programm auf Oberflächenereignisse reagieren?*
- *Welche Arten von Widgets gibt es?*
- *Wie kann in einem Fenster selbst gezeichnet werden?*
- *Wie lassen sich Effekte und Animationen einsetzen?*



Historie

1. **AWT:** Seit dem JDK 1.x gibt es das AWT (Abstract Window Toolkit) zur Erzeugung graphischer Benutzungsoberflächen. Nachteile des AWT:
 - ◆ Alle Fenster- und Dialogelemente werden von dem darunterliegenden Betriebssystem zur Verfügung gestellt → schwierig, plattformübergreifend ein einheitliches Look-and-Feel zu realisieren.
 - ◆ Die Eigenarten jedes einzelnen Fenstersystems waren für den Anwender unmittelbar zu spüren.
 - ◆ Im AWT gibt es nur eine Grundmenge an Dialogelementen, mit denen sich aufwändige grafische Benutzeroberflächen nicht oder nur mit sehr viel Zusatzaufwand realisieren ließen.
2. **Swing:** Einführung der Swing-Klassen als Bestandteil der JFC (Java Foundation Classes) mit Java 2.
 - ◆ Swing-Komponenten benutzen nur Top-Level-Fenster sowie grafische Grafikoperationen des Betriebssystems.
 - ◆ Alle anderen GUI-Elemente werden von Swing selbst gezeichnet.



- ◆ Vorteile:

- Plattformspezifische Besonderheiten fallen weg → einfachere Implementierung der Dialogelemente.
- Einheitliche Bedienung auf unterschiedlichen Betriebssystemen
- Nicht nur Schnittmenge der Komponenten aller Plattformen verfügbar
- Pluggable Look-and-Feel: Umschaltung des Aussehens einer Anwendung zur Laufzeit (Windows, Motif, Metal, ...).

- ◆ Nachteile:

- Swing-Anwendungen sind ressourcenhungrig. Das Zeichnen der Komponenten erfordert viel CPU-Leistung und Hauptspeicher (Unterstützung durch DirectDraw, OpenGL).

3. **JavaFX** wurde ursprünglich eingeführt, um mittels der Skriptsprache JavaFX-Script einfache und schnell Animation zu erstellen.

- ◆ JavaFX wird mittelfristig Swing ersetzen.
- ◆ Die aktuelle Version 2.2.7 bietet nur einer Untermenge der Swing-Funktionen.
- ◆ Grafikoperationen usw. sind denen aus Java 2D sehr ähnlich.

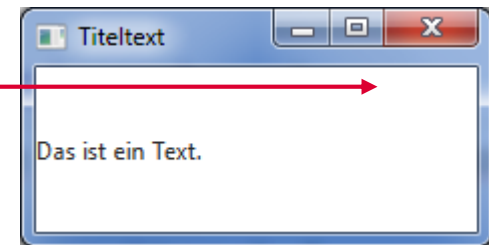
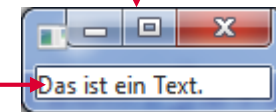


- Die folgende Einführung nimmt teilweise starke Vereinfachungen vor, weil fachliche Grundlagen wie „Multithreading“ und Internationalisierung noch fehlen.
- „Richtige“ GUI-Programmierung gibt es im Wahlfach „Benutzungsoberflächen“.
- Gute Quelle: <http://docs.oracle.com/javafx/index.html>



- Source-Code (**FirstApplication.java**):

```
public class FirstApplication extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        primaryStage.setTitle("Titeltext");  
  
        // Inhalt des Fensters  
        Label label = new Label("Das ist ein Text.");  
        Scene scene = new Scene(label);  
  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```



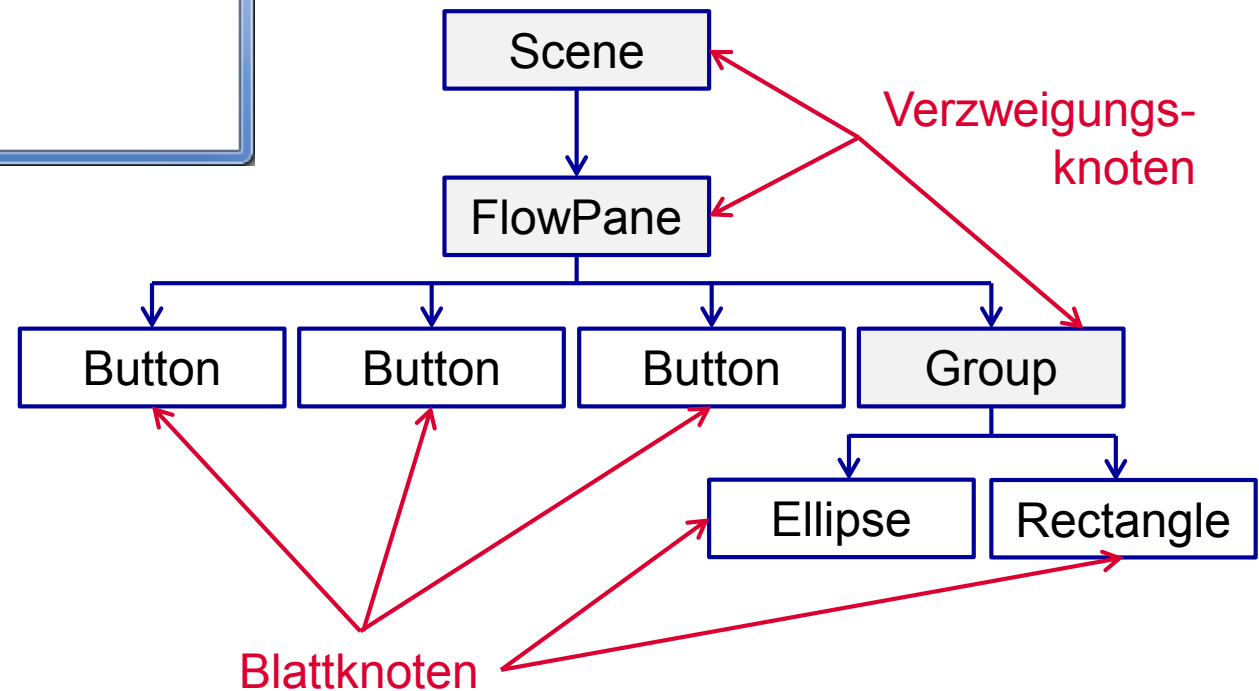
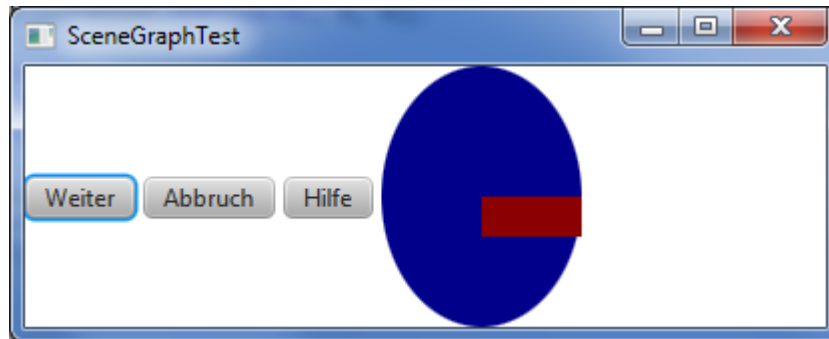
Originalausgabe und
manuelle Vergrößerung



- Erste Erklärungen:
 - ◆ **Application**: Anwendungsklasse; Sie stellt ein Fenster mit Rahmen, Systemmenü und Standardschaltflächen zur Verfügung.
 - ◆ Überschriebene Methode **start**: wird beim Erzeugen der Anwendung aufgerufen und legt den Inhalt des Fensters fest
 - ◆ **Stage**: oberster JavaFX-Container, in der Regel ein Fenster
 - ◆ **Label(String text)**: textuelle Bezeichnung
 - ◆ **Scene**: beschreibt den Inhalt des Fensters
 - ◆ **show**: zeigt das Fenster auf dem Bildschirm an
 - ◆ **launch()**: startet die Anwendung



- Die Widgets im Fenster werden in Form eines Baums angeordnet (JavaFX Scene Graph), Beispiel:

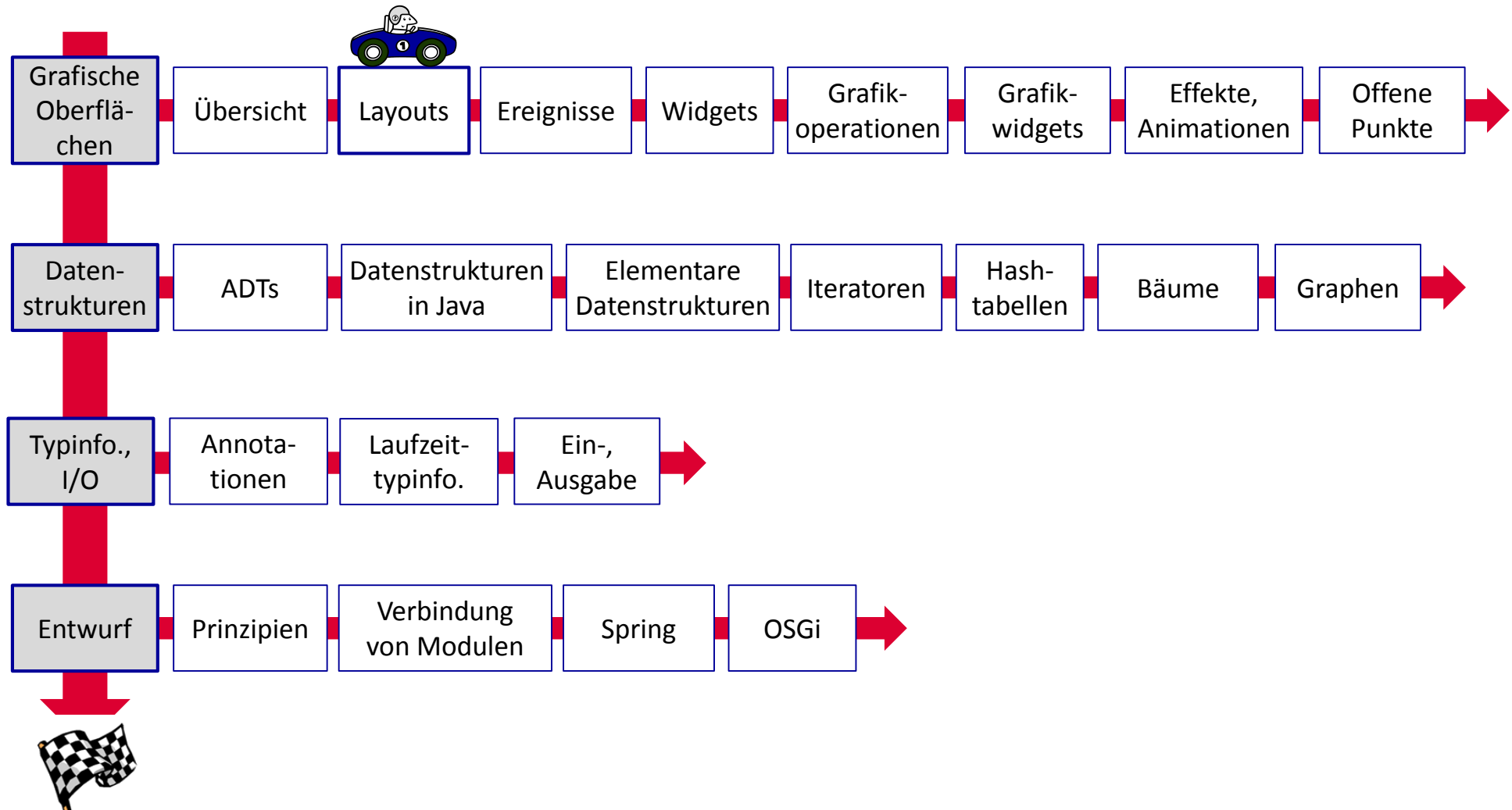
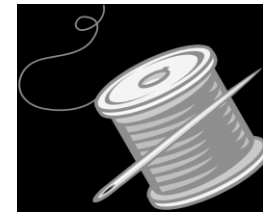




- Der Graph wird automatisch neu gezeichnet.
- Hardware-Unterstützung:
 - ◆ DirectX 9 unter Windows XP und Windows Vista
 - ◆ DirectX 11 unter Windows 7
 - ◆ OpenGL unter MacOS und Linux
 - ◆ Java2D, wenn es keine Hardware-Unterstützung gibt
- Unterstützung von
 - ◆ grafischen Effekten
 - ◆ Animationen
 - ◆ Videos
 - ◆ eingebettetem Browser
 - ◆ ...

Layouts

Übersicht

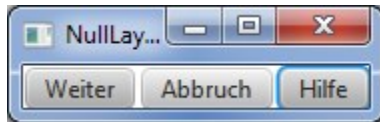




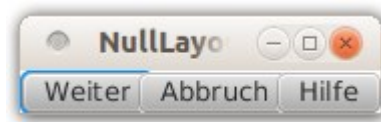
Warum sollen Widgets nicht einfach absolut positioniert werden?

Problem 1: Unterschiedliche Plattformen

Windows 7 mit Java 7

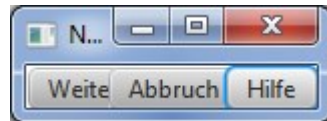


Ubuntu 12.04 mit Java 7



Problem 2: Internationalisierung

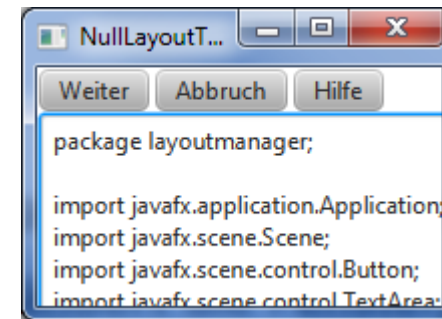
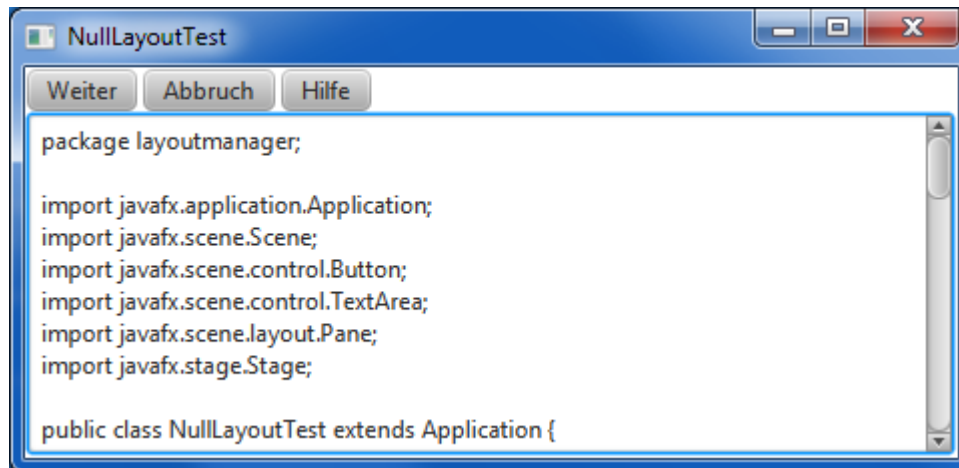
- Nachträgliche Übersetzung (Internationalisierung) der Texte in andere Sprachen
- Ausgaben (Sprachvarianten):





Problem 3: Interaktive Größenänderungen der Dialoge

- Der Anwender ändert die Fenstergröße interaktiv → der Inhalt passt sich nicht automatisch an (Platzverschwendung oder „Verschwinden“ von Inhalten):



Problem 4: Textorientierung

- Container unterstützen eine Ausrichtung der Komponenten anhand der Textausrichtung der eingestellten Sprache und des Landes → von vielen Layoutmanagern verwendet.



- Konsequenzen:
 - ◆ Plattformunabhängige Programmierung ist mit absoluten Layouts nicht sinnvoll möglich (unterschiedliche Zeichensätze oder Zeichengrößen).
 - ◆ Portierung einer Anwendung in mehrere Sprachen erfordert bei absoluten Layouts manuelle Nacharbeiten → nicht praktikabel.
 - ◆ Niemals ohne Layoutmanager mit JavaFX Benutzeroberflächen erstellen!
 - ◆ Auch andere Toolkits arbeiten mit Layoutmanagern: Swing, AWT, QT, GTK, SWT, WPF, ...



- Frage: Woher kennt ein Layoutmanager die Größen der einzelnen Widgets?
- Lösung: Jedes Widget kennt seine eigene Größe. Der Layoutmanager fragt die Widgets ab.
- Viele Layoutmanager beachten die folgenden Größen eines Widgets (Basisklasse **Control**):
 - ◆ **double minWidth, minHeight**: minimale Größe
 - ◆ **double maxWidth, maxHeight**: maximale Größe
 - ◆ **double prefWidth, prefHeight**: bevorzugte Größe, Beispiel: **prefWidth** eines Buttons ist Summe aus:
 - Breite des Textes oder des Bildes
 - Breite des Platzes für die Ränder
 - ◆ **double width, height**: aktuelle Größe, wird vom Layoutmanager gesetzt.



FlowPane: Platzierung der Widgets nacheinander in ihrer bevorzugten Größe

- Die Abstände zwischen den Widgets sowie deren Anordnung können im Konstruktor des Layoutmanagers angegeben werden.
- Anordnung:

Konstante	Anordnung der Widgets
Orientation.VERTICAL	vertikal (Widgets „übereinander“)
Orientation.HORIZONTAL	horizontal (Widgets „nebeneinander“)

- Ausrichtung der Widgets im Layout (wenn das Layout größer als der benötigte Platz ist), Auswahl:

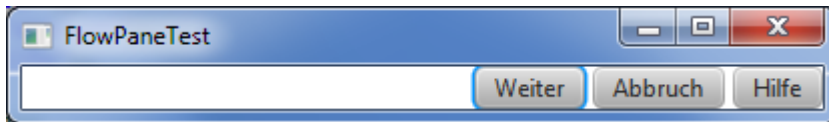
Konstante	Ausrichtung der Widgets
Pos.CENTER_LEFT	vertikal zentriert, horizontal linksbündig
Pos.TOP_RIGHT	vertikal oben, horizontal rechtsbündig
Pos.BOTTOM_CENTER	vertikal unten, horizontal zentriert



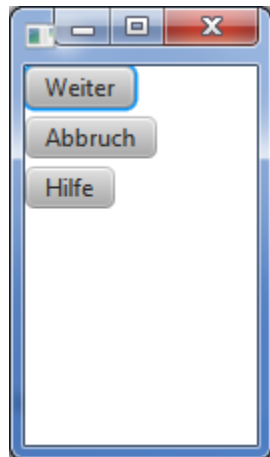
- Beispiele:



horizontal, linksbündig



horizontal, rechtsbündig



vertikal, linksbündig



- Programmcode (**FlowPaneTest.java**):

```
...
primaryStage.setTitle("FlowPaneTest");

// Inhalt des Fensters horizontal anordnen
FlowPane flowPane = new FlowPane(Orientation.HORIZONTAL, 4, 4);

Button okButton = new Button("Weiter");
Button cancelButton = new Button("Abbruch");
Button helpButton = new Button("Hilfe");

// Tasten zum FlowLayout hinzufügen
flowPane.getChildren().addAll(okButton, cancelButton, helpButton);
flowPane.setAlignment(Pos.TOP_RIGHT);

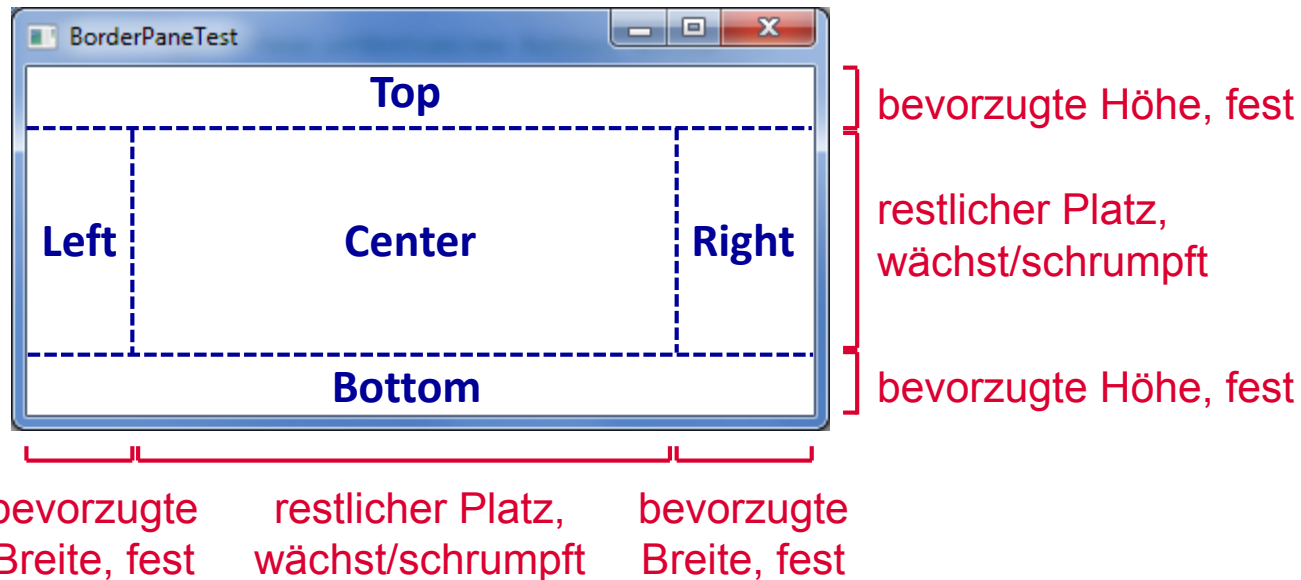
Scene scene = new Scene(flowPane);

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



BorderPane: Basis für Standarddialoge

- **BorderPane** besitzt 5 Bereiche, in die jeweils ein Widget eingetragen werden kann:





- Einfügen von Widgets in eine **BorderPane**:
 - ◆ **setTop(Control contr)**:
 - **contr**: einzufügendes Widget (z.B. **Label**, **Button**, ...)
 - ◆ Statt **setTop** gibt es auch die Varianten für die anderen Positionen (wie z.B. **setBottom**).
- Ist der Bereich größer als der Platz, den das Widget benötigt, dann kann die Position des Widgets festgelegt werden:
 - ◆ Taste soll zentriert werden: **BorderPane.setAlignment(button, Pos.CENTER);**
 - ◆ Die weiteren Positionsangaben sind ebenfalls statische Attribute der Klasse **Pos**.
- Es kann auch ein leerer Rand um ein Widget erzeugt werden, Beispiel:
BorderPane.setMargin(button, new Insets(4.0, 4.0, 4.0, 4.0));
- **Insets** nimmt die vier Abstände im Uhrzeigersinn auf (oben, rechts, unten, links).



- Beispiel (Datei **BorderPaneTest.java**):

```
...
// Inhalt des Fensters anordnen
BorderPane borderPane = new BorderPane();

// Widgets zur BorderPane hinzufügen
Label topLabel = new Label("Titeltext");
BorderPane.setAlignment(topLabel, Pos.CENTER_LEFT);
BorderPane.setMargin(topLabel, new Insets(4.0, 4.0, 4.0, 4.0));
borderPane.setTop(topLabel);

borderPane.setLeft(new Button("Left"));
borderPane.setRight(new Button("Right"));

TextArea centerText = new TextArea("Center");
BorderPane.setAlignment(centerText, Pos.CENTER);
BorderPane.setMargin(centerText, new Insets(4.0, 4.0, 4.0, 4.0));
borderPane.setCenter(centerText);

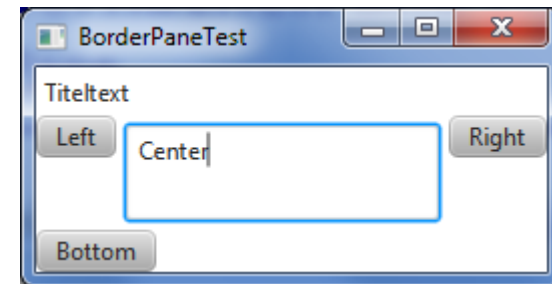
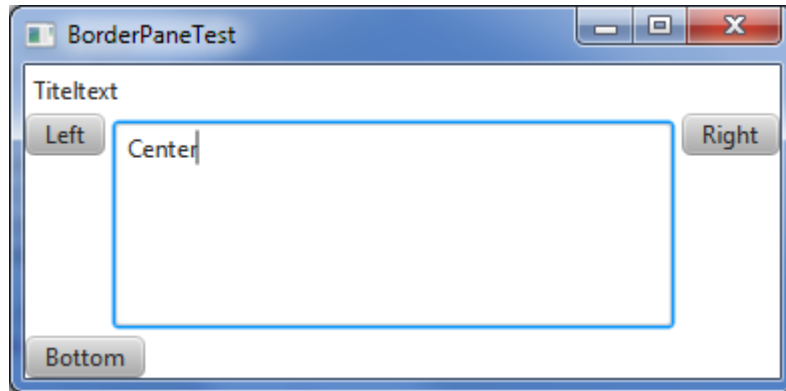
borderPane.setBottom(new Button("Bottom"));
...
```

Layouts

BorderPane



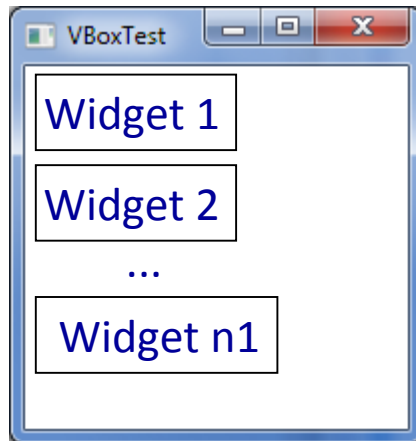
- Ausgabe (normal und verkleinert):





Platzierung der Widgets nacheinander in einheitlicher Größe

- Die Abstände zwischen den Widgets, die äußeren Ränder sowie die Ausrichtung können angegeben werden.
- Alle Widgets erhalten ihre bevorzugte Größe.
- Die **VBox** wird so groß wie die Summe der bevorzugten Größen der Widgets.



- Die Reihenfolge der Komponenten wird durch die Reihenfolge des Einfügens festgelegt.



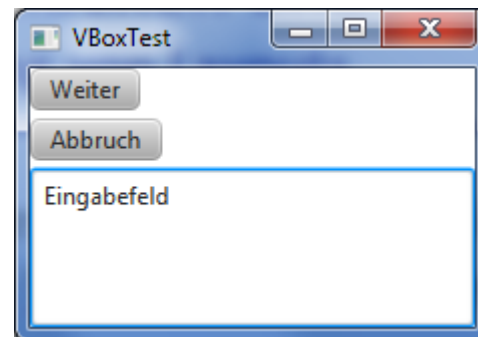
- Verhalten:
 - ◆ Wird der Dialog vergrößert, dann wird normalerweise der zusätzliche vertikale Platz nicht genutzt.
 - ◆ Soll ein Widget den zusätzlichen vertikalen Platz nutzen, dann kann das eingestellt werden: **`VBox.setVgrow(button, Priority.ALWAYS);`**
 - ◆ **Priority** bestimmt bei mehreren Widgets, welches Widget die Priorität beim Wachsen erhalten soll.



- Beispiel (**VBoxTest.java**):

```
...  
// Inhalt des Fensters vertikal anordnen  
VBox vbox = new VBox(4.0);  
  
Button okButton = new Button("Weiter");  
Button cancelButton = new Button("Abbruch");  
TextArea text = new TextArea("Eingabefeld");  
VBox.setVgrow(text, Priority.ALWAYS);  
  
// Tasten zum FlowLayout hinzufuegen  
vbox.getChildren().addAll(okButton, cancelButton, text);  
...
```

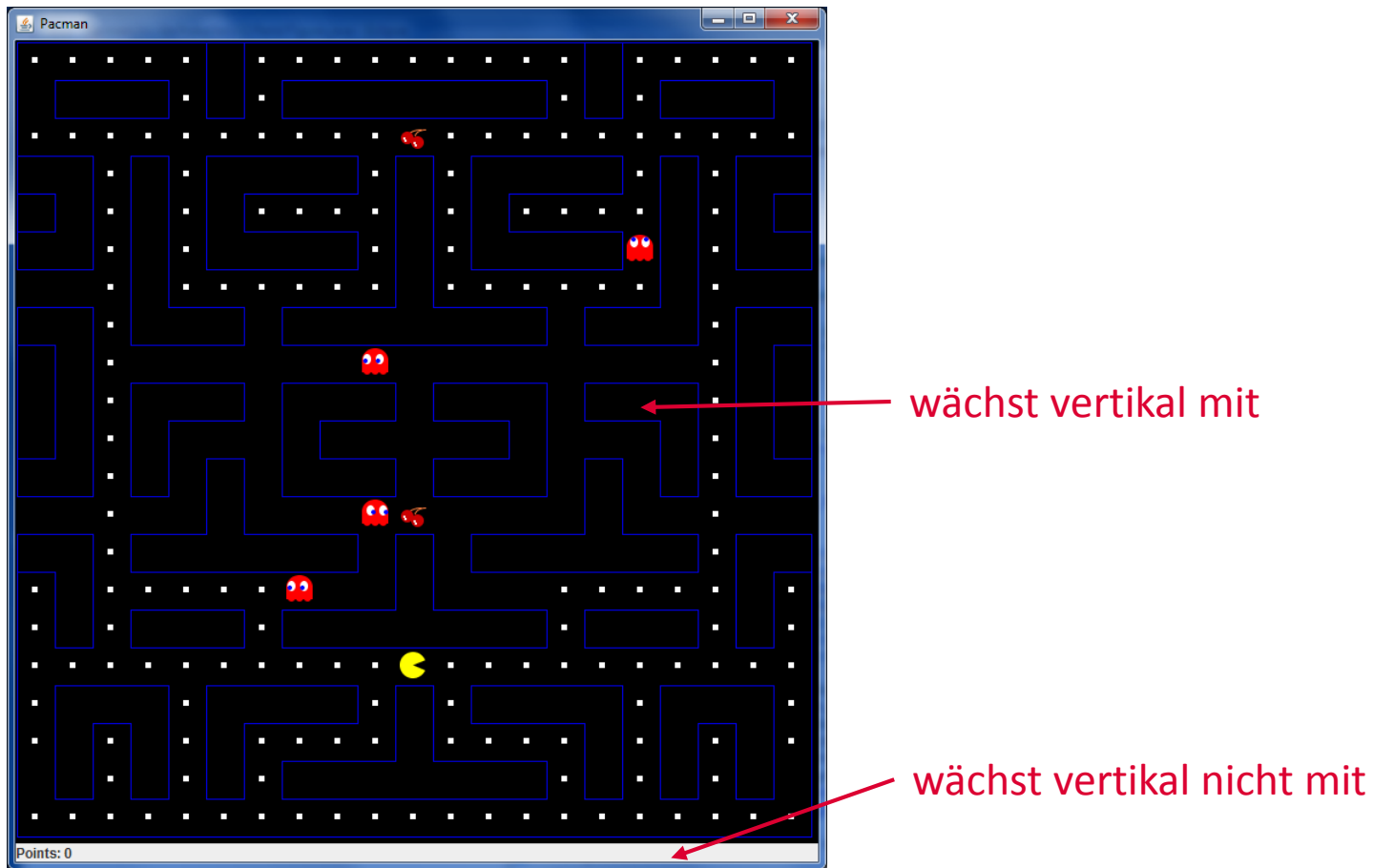
- Bildschirmausgabe:



wächst vertikal mit



- Pacman wurde mit der **VBox** erstellt:

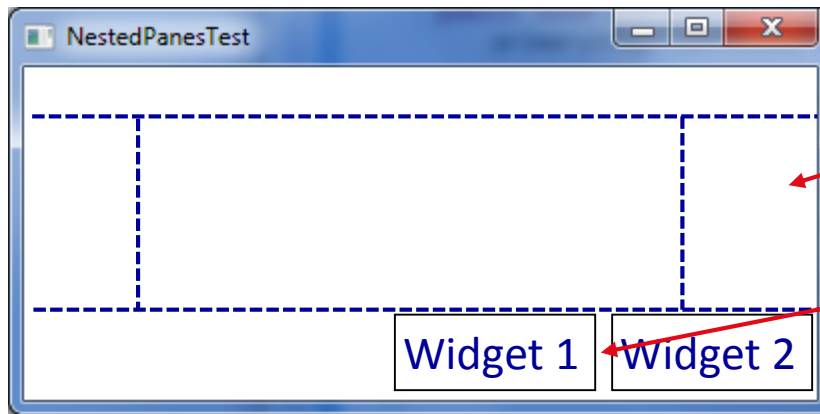


Layouts

Schachtelung von Layouts



- Bisher wurde innerhalb des Fensters immer ein Layoutmanager verwendet.
- Dieser Ansatz ist für komplizierte Layouts nicht flexibel genug.
- Ergänzung: Layouts lassen sich hierarchisch schachteln.

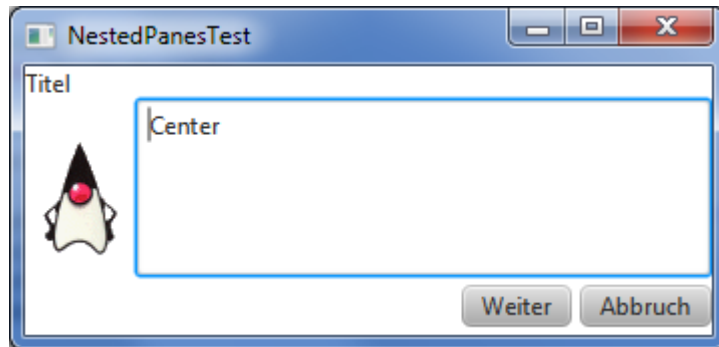


BorderPane

HBox (wie **VBox**, aber
horizontale Anordnung)



- Beispiel: Ein Dialog mit zwei nebeneinander angeordneten Tasten.



- Hinweis: Ein **Label** kann auch ein Bild enthalten (Quelltextbeispiel).



- Datei: **NestedPanesTest.java**

```
...
BorderPane borderPane = new BorderPane();

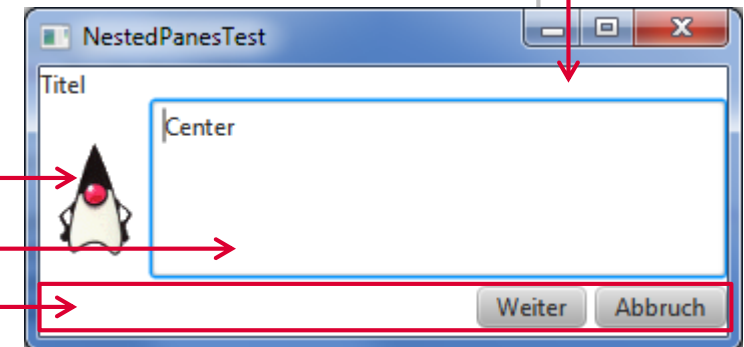
borderPane.setTop(new Label("Titel"));

Image image = new Image(getClass().getResourceAsStream("/resource/duke2.gif"));
Label leftLabel = new Label("", new ImageView(image));
BorderPane.setAlignment(leftLabel, Pos.CENTER);
borderPane.setLeft(leftLabel);

TextArea centerText = new TextArea("Center");
borderPane.setCenter(centerText);

HBox buttonBox = new HBox(4.0);
buttonBox.setAlignment(Pos.BOTTOM_RIGHT);
buttonBox.getChildren().add(new Button("Weiter"));
buttonBox.getChildren().add(new Button("Abbruch"));
BorderPane.setMargin(buttonBox, new Insets(4.0, 0.0, 4.0, 0.0));

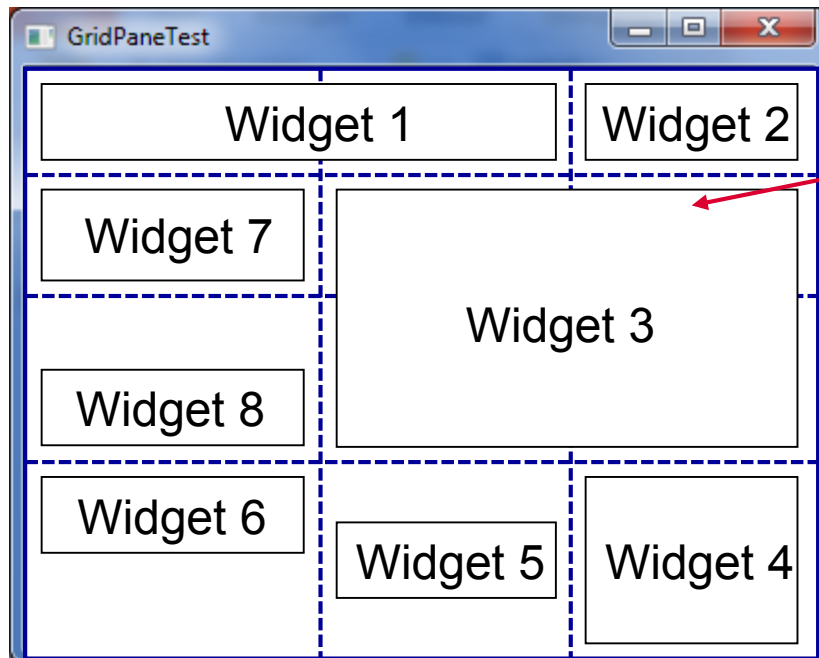
borderPane.setBottom(buttonBox);
...
```





Flexible Gestaltung aufwändiger Dialoge

- **GridPane** ist der flexibelste und komplexeste Standardlayoutmanager in JavaFX.
- **GridPane** platziert die Komponenten in einem Raster aus Zeilen und Spalten.



Widgets dürfen sich über mehrere Zeilen und/oder Spalten erstrecken.

Zeilen dürfen unterschiedliche Höhen, Spalte unterschiedliche Breiten besitzen.

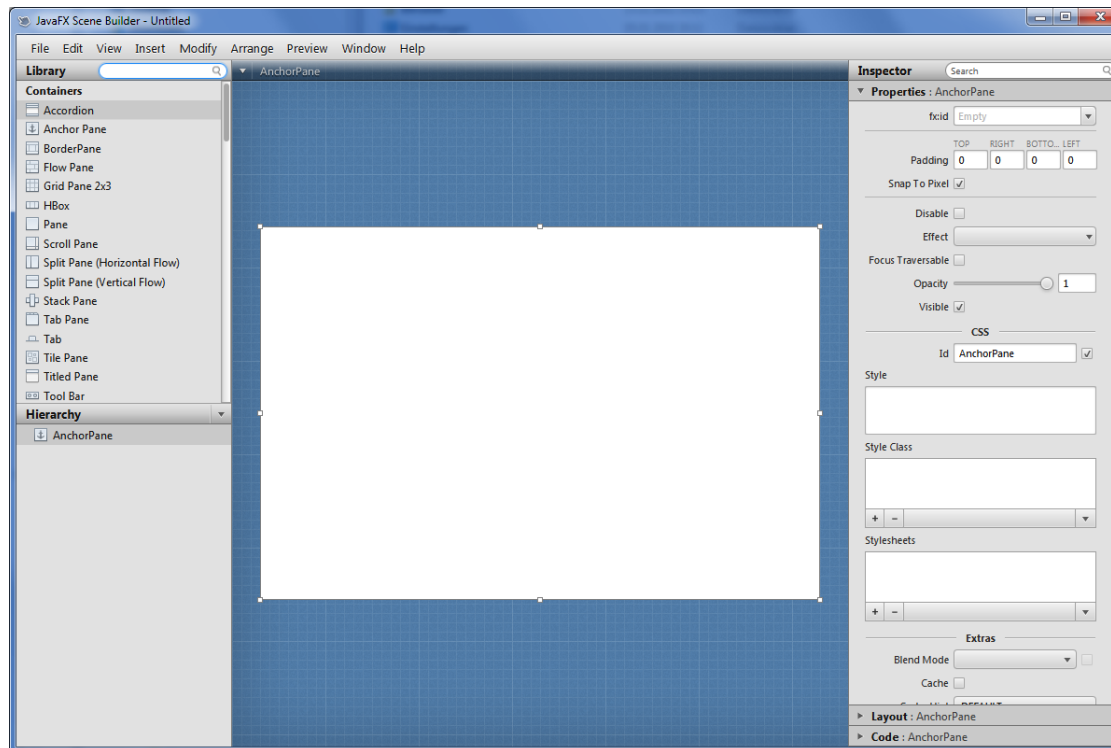
Die Größen der Zellen werden aus den bevorzugten Größen der Widgets berechnet.

Layouts

Scene Builder



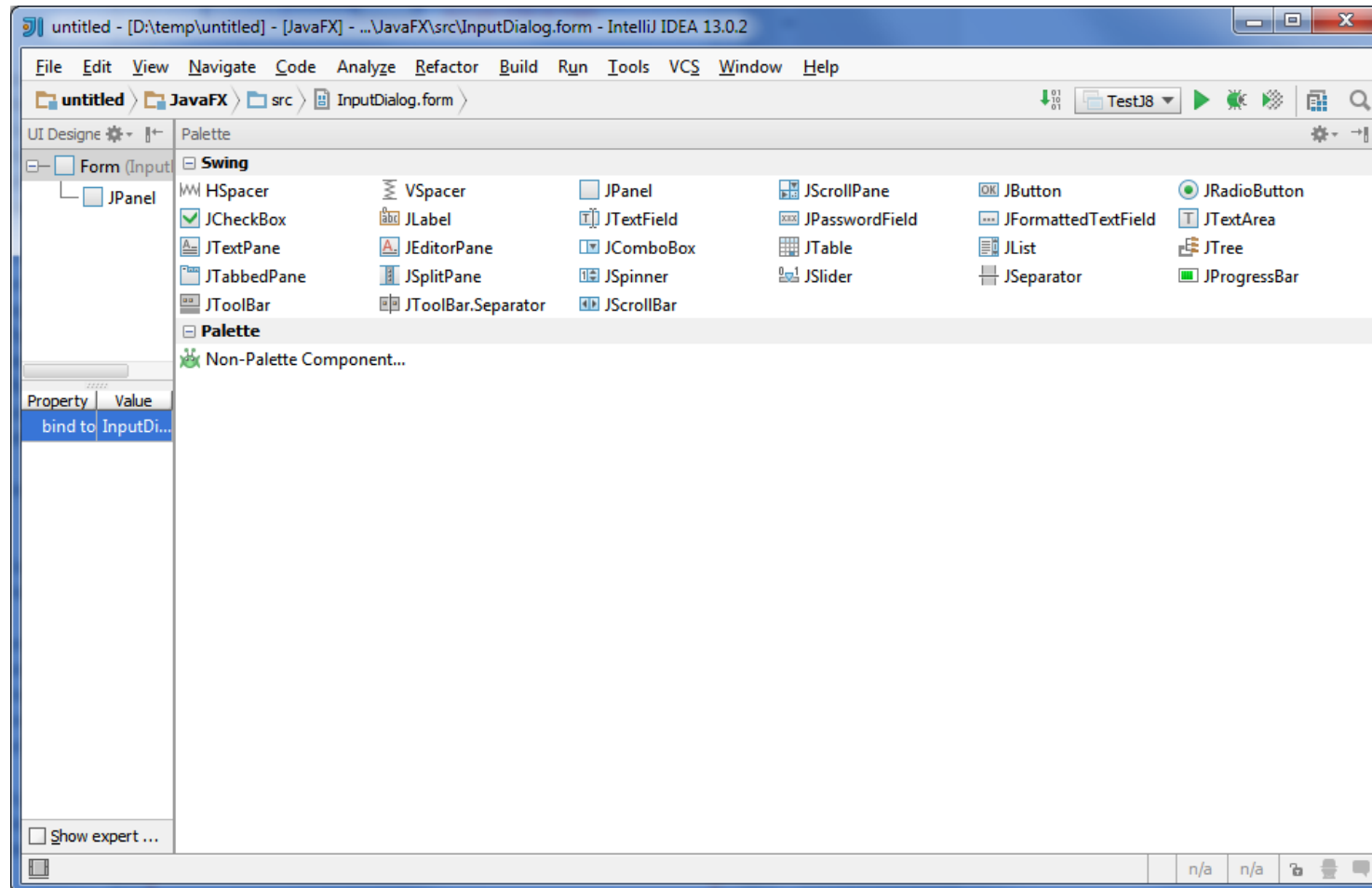
- Wer das manuelle Programmieren der Oberflächen nicht mag, nimmt einen interaktiven GUI-Builder wie den Scene Builder:



- Einbindung in Eclipse: http://docs.oracle.com/javafx/scenebuilder/1/use_java_ides/sb-with-eclipse.htm

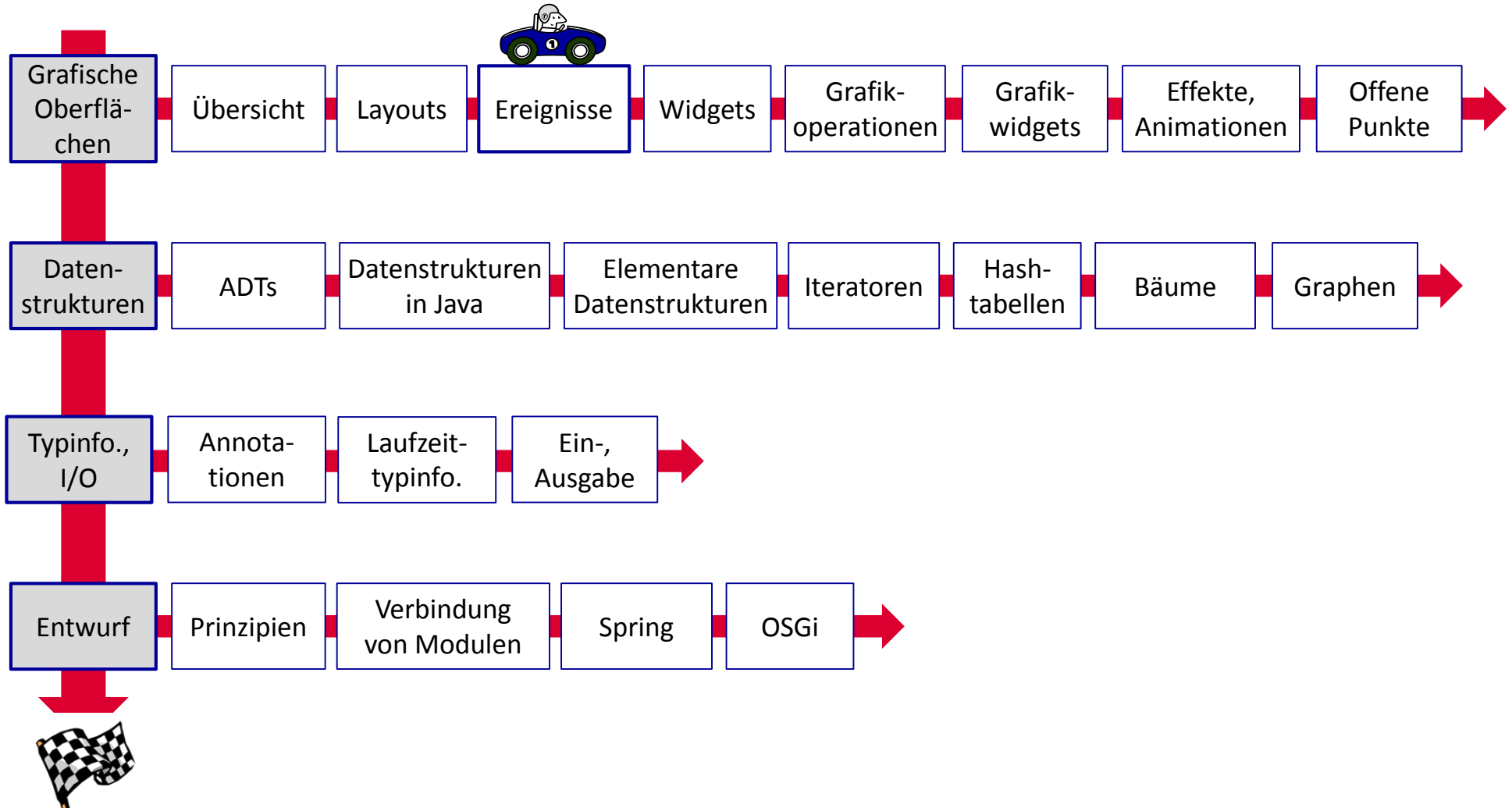
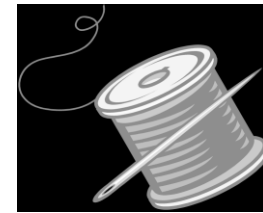


- In IntelliJ IDEA und Netbeans ist der interaktive GUI-Editor bereits integriert:



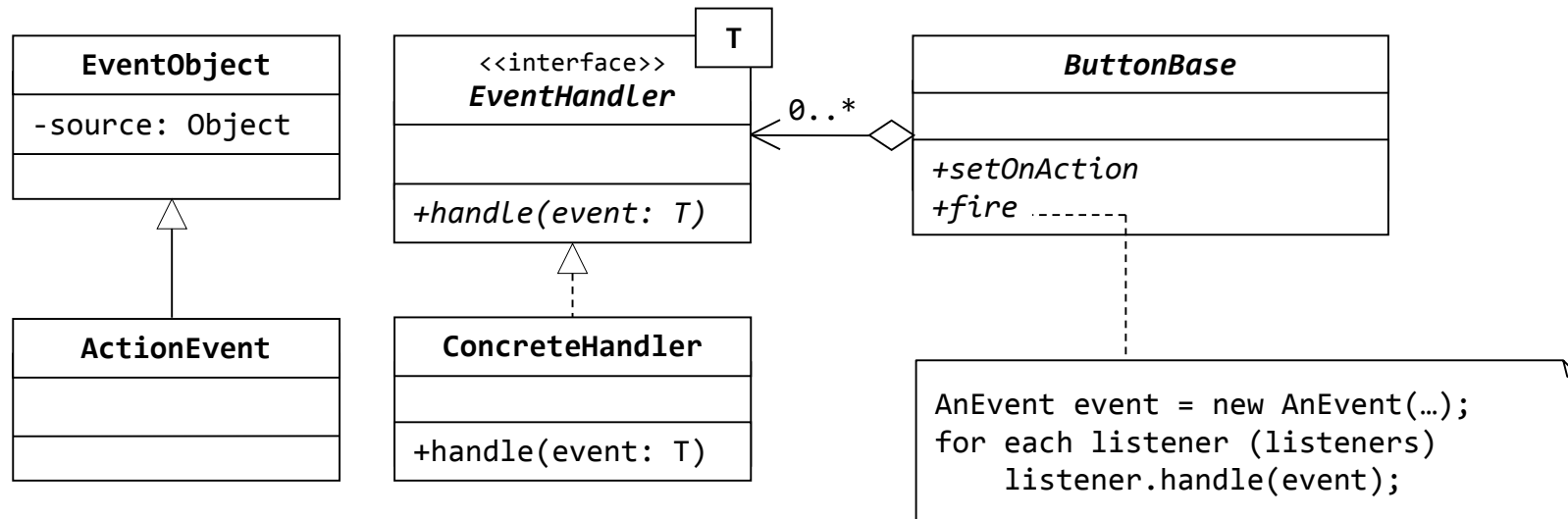
Ereignisbehandlung

Übersicht





- In JavaFX werden sehr viele unterschiedliche Arten von Ereignissen unterstützt.
- Als wichtiges Ereignis soll hier das **ActionEvent** näher betrachtet werden.
- Buttons lösen ein Action-Event aus, nachdem der Button gedrückt und wieder losgelassen wurde.
- Die Ereignisbehandlung basiert auf einem modifizierten Beobachter-Entwurfsmuster (Listener = Beobachter), vereinfacht:



- Eine genauere Behandlung folgt später.

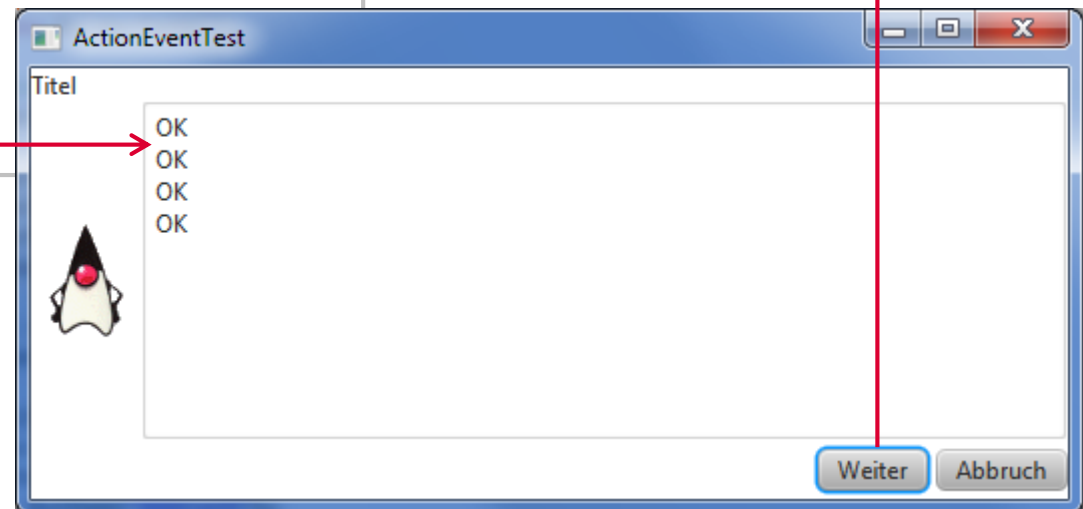


- Beispiel (Datei **ActionEventTest.java**):

```
...  
final TextArea centerText = new TextArea("");  
borderPane.setCenter(centerText);  
  
...  
Button okButton = new Button("Weiter");  
okButton.setOnAction(new EventHandler<ActionEvent>() {  
    @Override  
    public void handle(ActionEvent e) {  
        centerText.appendText("OK\n");  
    }  
});  
...
```

ruft auf

gibt aus





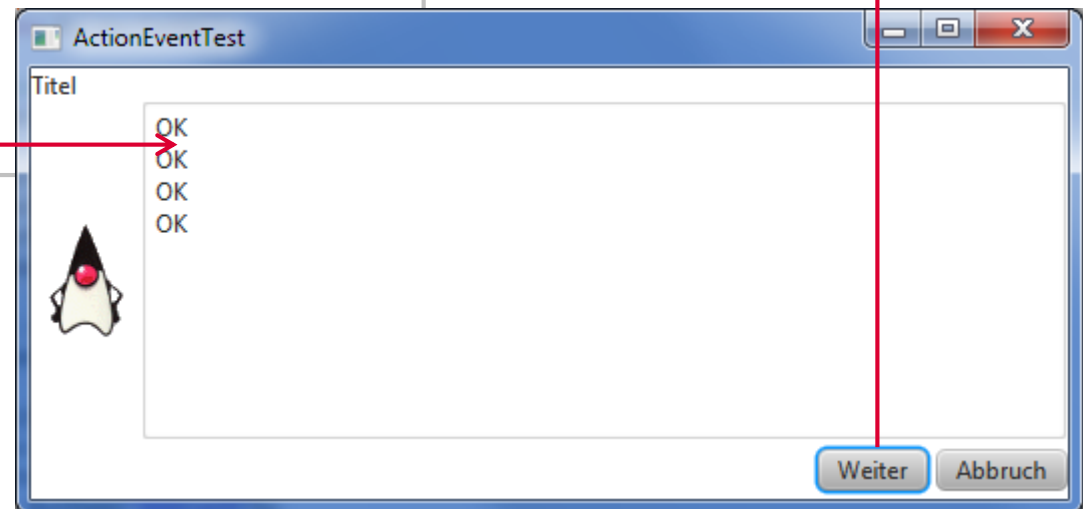
- Es gibt doch Lambda-Ausdrücke (Datei **ActionEventTest.java**):

```
...  
final TextArea centerText = new TextArea("");  
borderPane.setCenter(centerText);
```

```
...  
Button okButton = new Button("Weiter");  
okButton.setOnAction(e -> centerText.appendText("OK\n"));  
...
```

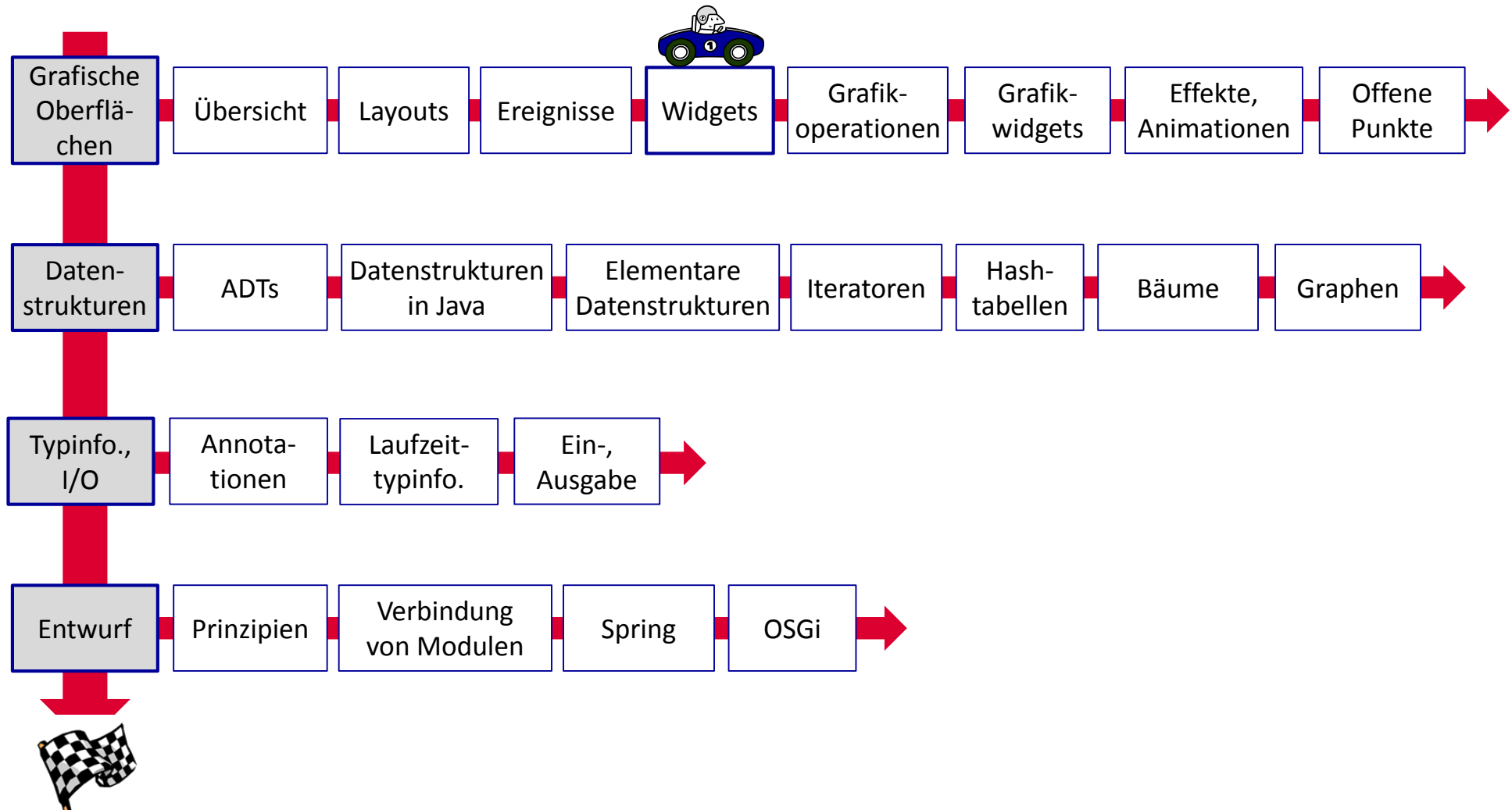
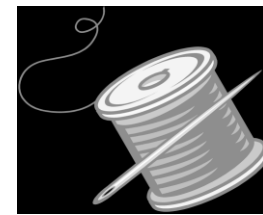
ruft auf

gibt aus

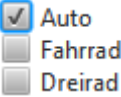



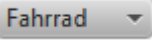


Widgets

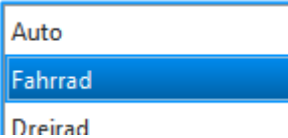
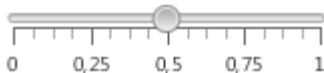
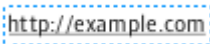
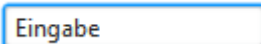

Übersicht



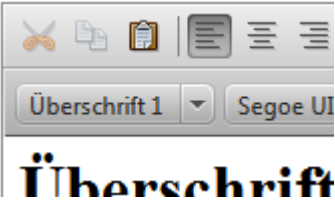

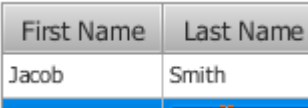



Name	Verhalten
CheckBox 	Kann selektiert und deselektiert werden. Der Selektionszustand bleibt bis zur Änderung erhalten.
RadioButton 	Kann selektiert werden. Der Selektionszustand bleibt erhalten, bis ein anderer Button derselben Gruppe gewählt wird.
Button 	Kann gedrückt werden. Der Selektionszustand bleibt nicht erhalten.
ToggleButton 	Kann selektiert werden. Verhält sich wie eine CheckBox oder ein RadioButton .
JComboBox 	Zeigt einen ausgewählten Eintrag sowie eine Liste weiterer möglicher Einträge an.



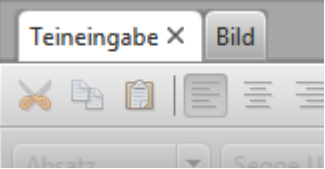
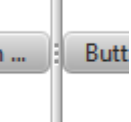


Name	Verhalten
ListView 	Zeigt einen ausgewählten Eintrag sowie eine Liste weiterer möglicher Einträge an.
Slider 	Kann horizontal oder vertikal verschoben werden und an bestimmten Markierungen einrasten.
Hyperlink 	Auswahl einer URL
Label Vorname:	kein Verhalten, Bezeichnung und/oder Bild (z.B. für eine anderes Widget)
TextField 	einzeiliges Freitexteingabefeld mit Cursorsteuerung und Selektion, keine variablen Textattribute
TextArea 	mehrzeiliges Freitexteingabefeld mit Cursorsteuerung und Selektion, keine variablen Textattribute



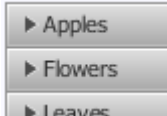
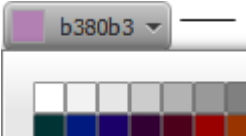


Name	Verhalten
HTMLEditor 	mehrzeiliges Freitexteingabefeld mit Cursor-Steuerung und Selektion, variable Textattribute, das Datenmodell im Hintergrund basiert auf HTML
PasswordField 	Eingegebene Zeichen werden z.B. durch * dargestellt.
TableView 	Darstellung bzw. Eingabe tabellarischer Daten
TreeView 	Baum mit auf- und zuklappbaren Knoten (wie z.B. beim Explorer)




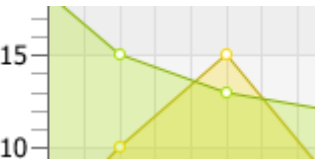


Name	Verhalten
Menu 	Menu mit Einträgen und Untermenüs zum Starten von Aktionen
ToolBar 	Buttonleiste für Aktionsauswahl
TabPane 	Umschaltung zwischen mehreren Teildialogen
SplitPane 	Variable Darstellung mehrerer Komponenten



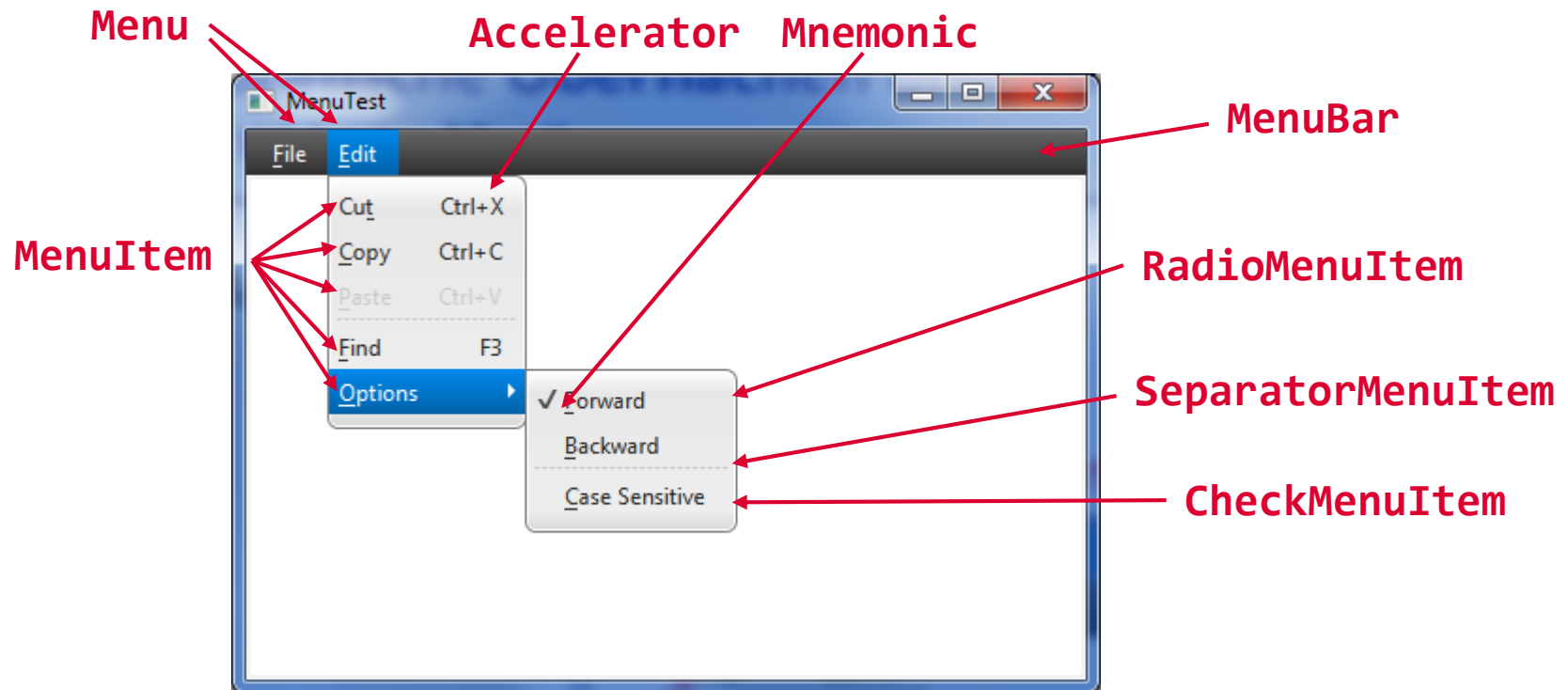
Name	Verhalten
ScrollPane 	Verschieben und Darstellung eines Ausschnittes
TitledPane 	auf- und zuklappbarer Teildialog
Accordion 	auf- und zuklappbarer Teildialog, bestehend aus TitlePanes
ColorPicker 	Farbauswahldialog



Name	Verhalten
FileChooser 	Dateiauswahldialog
DirectoryChooser 	Verzeichnisauswahldialog
LineChart 	Liniendiagramm
AreaChart 	Diagramm



Aufbau einer Menüstruktur





- Erzeugung einer **MenuBar** (aus Datei **MenuTest.java**):

```
MenuBar menuBar = new MenuBar();

// einzelnes Menü hinzufügen
Menu fileMenu = new Menu("_File");
menuBar.getItems().add(fileMenu);

// Menubar einem Layout hinzufügen
VBox vbox = new VBox();
vbox.getChildren().add(menuBar);
```

- Tastaturbedienung:

- ◆ Mnemonic: Navigation innerhalb des Menüs. Einzelnes Zeichen, dargestellt durch einen unterstrichenen Buchstaben. Die Navigation erfolgt plattformabhängig (z.B. ALT-C unter Windows).



- ◆ Accelerator: Aufruf eines Menüpunktes, ohne dass das Menü überhaupt sichtbar ist (z.B. F2, Ctrl-C). Der Accelerator wird rechts innerhalb des Menüeintrags dargestellt.





Accelerator

Ermittlung einer Tastenkombination durch Verwendung statischer Methoden der Klasse **KeyCombination** (eine der Möglichkeiten):

keyCombination(String s): **s** ist eine „lesbare“ Beschreibung, z.B.:

- ◆ "Ctrl-X"
- ◆ "F2": Die Taste **F2**
- Accelerator am **MenuItem** registrieren:
menuItem.setAccelerator(keyCombination);
- Beispiel (aus Datei **MenuTest.java**):

```
// Edit->Cut, T - Mnemonic, CTRL-X - Accelerator
MenuItem cutMenuItem = new MenuItem("_Cut");
cutMenuItem.setAccelerator(KeyCombination.keyCombination("Ctrl+X"));
editMenu.getItems().add(cutMenuItem);
```





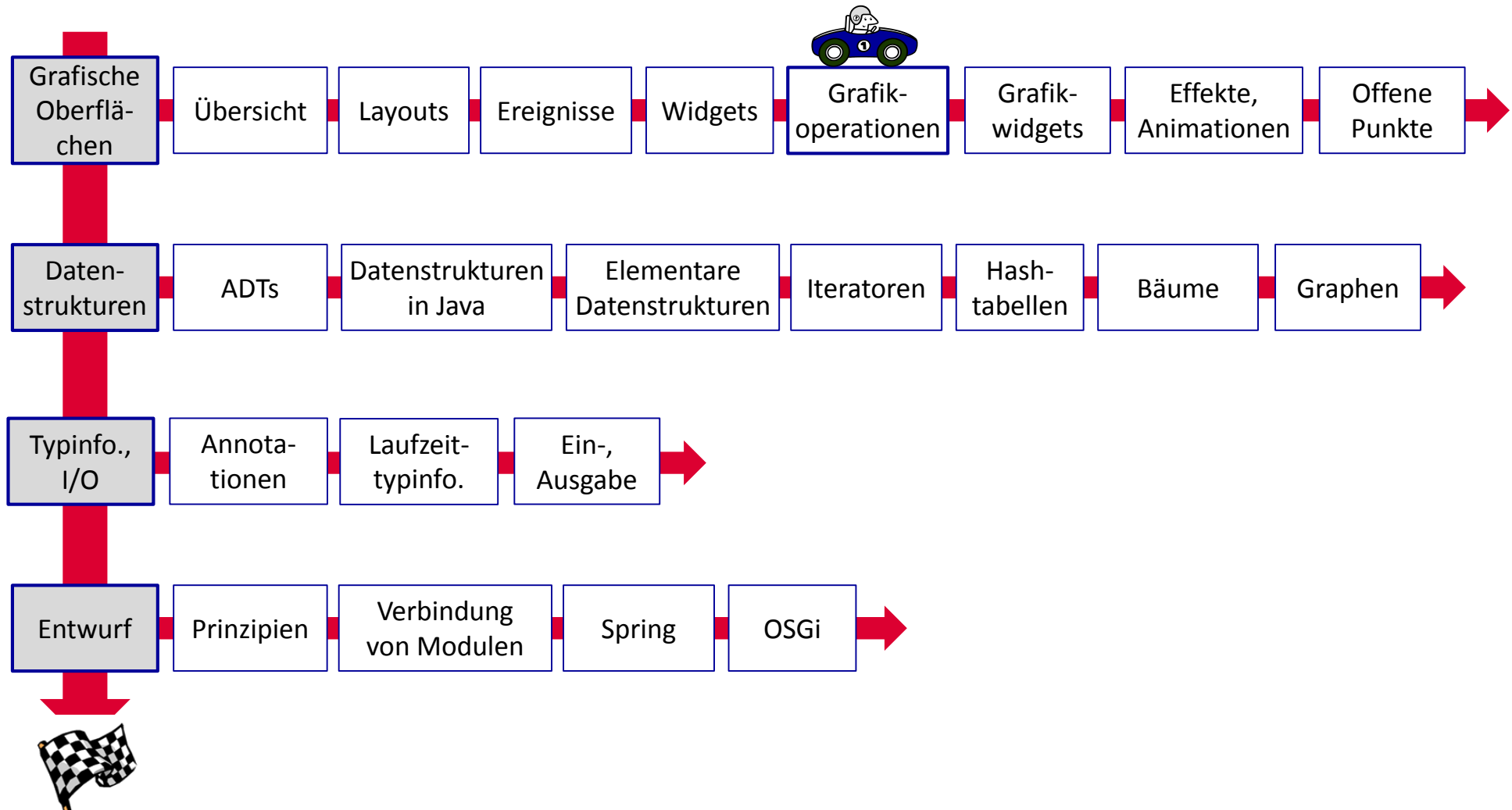
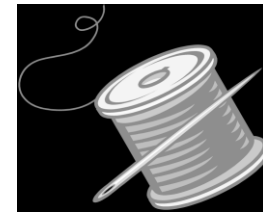
- Ein **MenuItem** kann ein **ActionEvent** auslösen, wenn der Menüeintrag ausgewählt wurde.
- Beispiel mit Ereignisbehandlung durch einen Lambda-Ausdruck (Datei **MenuTest.java**):

```
...
MenuBar menuBar = new MenuBar();
// File Menu, F - Mnemonic
Menu fileMenu = new Menu("_File");
menuBar.getItems().add(fileMenu);

// File->New, N - Mnemonic
MenuItem newItem = new MenuItem("_New");
newItem.setOnAction(e -> dumpArea.appendText(e.toString() + "\n"));
fileMenu.getItems().add(newItem);
...
```


Grafikoperationen

Übersicht





- Grafikoperationen:
 - ◆ 2-dimensionales Zeichnen von Grafik, Text und Bildern
 - ◆ einheitliches Zeichenmodell für Bildschirme und Drucker
 - ◆ große Anzahl primitiver geometrischer Figuren unterstützt: Kurven, Rechtecke, Ellipsen und nahezu beliebige Freiformen
 - ◆ Erkennung von Mausklicks auf Grafikformen, Text und Bildern
- Vorgehensweise:
 - ◆ Das Zeichnen kann direkt in einem **Canvas**-Objekt erfolgen.
 - ◆ Die Klasse **Canvas** erbt von **Node** und kann daher direkt im Szenen-Graph (also in anderen Layouts) eingesetzt werden.
 - ◆ Der Grafik-Kontext **GraphicsContext** besitzt alle Informationen zu Zeichen- und Füllfarbe usw.



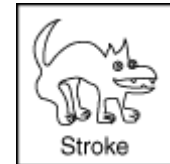
- Beispiel:

```
public class MyCanvas extends Canvas {  
    // ...  
    public void paint() {  
        GraphicsContext gc = getGraphicsContext2D();  
        // ...  
    }  
    // ...  
}
```

- Alle Grafikoperationen verwenden **double**-Werte als Koordinaten und Größen: höhere Genauigkeit beim Drehen, Skalieren, ...
- Der Ursprung liegt in der linken oberen Ecke (Default).

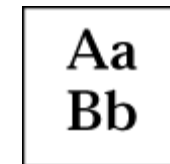


- Der Grafik-Kontext verwendet Zustands-Attribute, den sogenannten Zeichen-Kontext.
- Der Zeichen-Kontext wird für alle Grafikoperationen verwendet.
- Eine Einstellung bleibt bis zu ihrer Änderung erhalten.
- Zeichen-Kontexte beim Zeichnen einer geometrischen Form:
 - ◆ **Stroke** (Stift-Stil, Klasse **Paint**): Dieser Stil wird verwendet, um die Außenlinien einer Form zu zeichnen:
 - Farbe der Linie
 - ◆ **Fill** (Füll-Stil, Klasse **Paint**):
 - Farbe
 - Verlauf
 - Muster, Textur
 - ◆ **BlendMode** (Kombination bei Form-Überlappung):
 - Farbersetzungen, Aufhellungen, ...





- ◆ **Transform** (Transformationsmatrix, Klasse **Affine**):
 - Rotation
 - Verschiebung
 - Skalierung
 - Scherung
- ◆ **Effect** (grafischer Effekt):
 - unscharf maskieren, ...
- ◆ **Font** (Zeichensatz für Textausgaben):
 - Name
 - Größe
 - Stil
- ◆ Weitere wie Linienenden, Linienarten, Liniendicke, ...
- Im Folgenden sollen nur **Stroke** und **Paint** betrachtet werden.





- Mit **`graphicsContext.setStroke(Paint paint)`** kann die Linienfarbe eingestellt werden.
- Achtung: Der Grafik-Kontext legt keine Kopie des **Paint**-Objektes an.
- **Paint** ist eine Schnittstelle, die u.A. von **Color** implementiert wird.
- Beispiel:

```
public class MyCanvas extends Canvas {  
    // ...  
    public void paint() {  
        GraphicsContext gc = getGraphicsContext2D();  
        gc.setStroke(Color.BLUE); // Linienfarbe  
        gc.setLineWidth(1.0);    // Liniendicke in Punkt  
        // ...  
    }  
    // ...  
}
```

- **LinearGradientPaint** und **RadialGradientPaint** erlauben Farbverläufe.
- **ImagePattern** erlaubt es, eine Figur mit dem Inhalt eines Bildes zu füllen.







- Mit **`gc.setFill(Paint paint)`** können Füllart und Zeichenfarbe eingestellt werden.
- Achtung: Der Grafik-Kontext legt auch hier keine Kopie des **Paint**-Objektes an. Beispiel:

```
public class MyCanvas extends Canvas {  
    // ...  
    public void paint() {  
        GraphicsContext gc = getGraphicsContext2D();  
        gc.setFill(Color.WHITE);  
        // ...  
    }  
    // ...  
}
```

- JavaFX unterstützt eine Anzahl vordefinierter Zeichenobjekte (Formen).
- Diese Formen können über den Grafik-Kontext gezeichnet werden.
- Die Formen werden mit **`gc.strokeABC(Parameter)`** ohne Füllung sowie mit **`gc.fillABC(Parameter)`** gefüllt gezeichnet. **ABC** ist der Name der Figur.
- Die folgende Übersicht enthält eine Auswahl.






Form (Methode)	Beschreibung	Darstellung
strokeLine	Linie, die mittels zweier Koordinaten beschrieben wird.	
strokeRect, fillRect	Rechteck, beschrieben durch eine Koordinate (linke obere Ecke) sowie Breite und Höhe.	
strokeRoundRect, fillRoundRect	Rechteck mit runden Ecken, beschrieben durch eine Koordinate (linke obere Ecke) sowie Breite und Höhe. Weiterhin können die Breite und Höhe der Rundung angegeben werden.	
strokeArc, fillArc	Kreisbogen, beschrieben durch eine Koordinate (linke obere Ecke) sowie Breite und Höhe. Anfangs- und Endwinkel bezogen auf den Mittelpunkt sowie das Verbinden der Enden werden ebenso spezifiziert.	

Grafikoperationen

Geometrische Formen



Form (Methode)	Beschreibung	Darstellung
strokeOval, fillOval	Ellipse, beschrieben durch eine Koordinate (linke obere Ecke) sowie Breite und Höhe.	
strokePolygon, fillPolygon	Linienzug, bestehend aus beliebig vielen Linien	
stroke, fill	zeichnet einen vorab definierten geometrischen Pfad, bestehend aus geraden Linien, quadratischen und kubischen Kurven	
strokeText, fillText	Textausgabe	Hallo!
drawImage	vorab geladenes oder erzeugtes Bild	

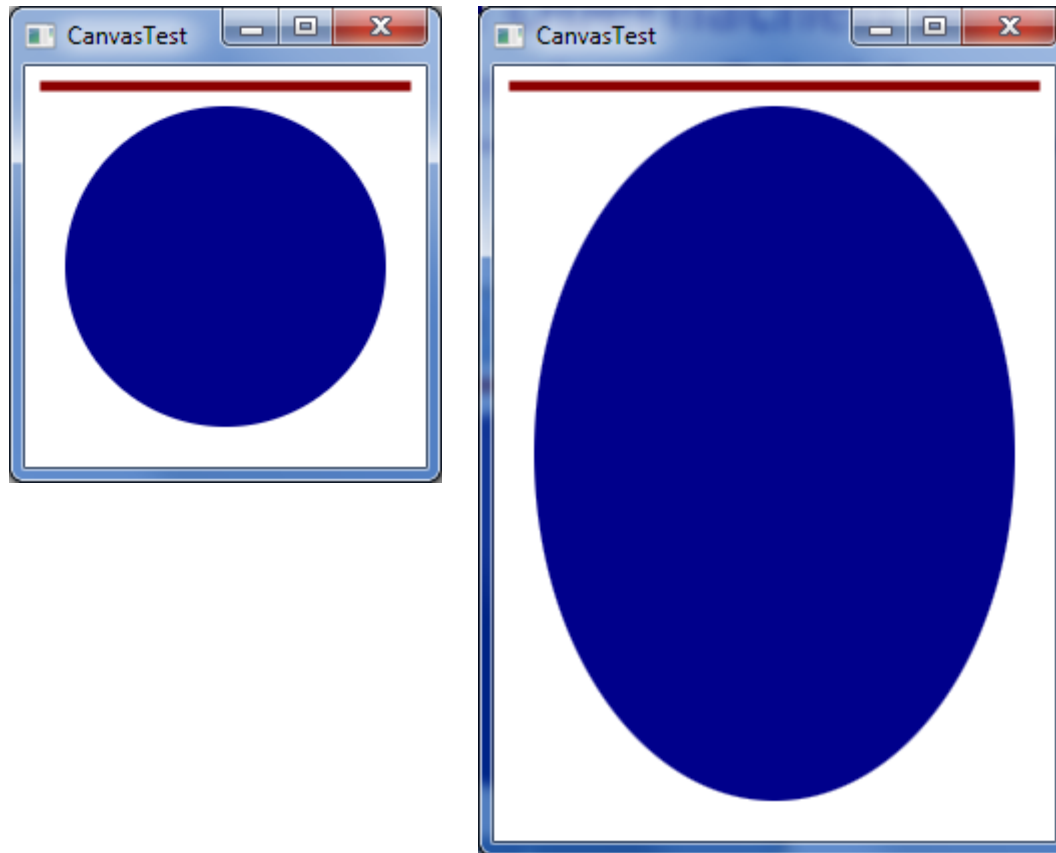
- Zur Erzeugung eines Pfades: siehe Methoden **beginPath**, **endPath**, ...

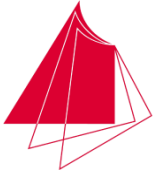
Grafikoperationen

Beispiel



- Ausgabe des folgenden Beispiels:





- Sourcecode (**PaintCanvas**):

```
public class PaintCanvas extends Canvas {  
    public PaintCanvas() {  
        setWidth(200.0);  
        setHeight(200.0);  
        // Hintergrundfarbe einstellen  
        setStyle("-fx-background-color: white;");  
        paint();  
    }  
  
    public void paint() {  
        GraphicsContext gc = getGraphicsContext2D();  
        double width = getWidth();  
        double height = getHeight();  
  
        // Hintergrund weiß ausfüllen → nur erforderlich,  
        // wenn der Canvas seine Größe ändert.  
        gc.setFill(Color.WHITE);  
        gc.fillRect(0.0, 0.0, getWidth(), getHeight());  
    }  
}
```



```
// Gefüllten Kreis zeichnen, dabei rund um den Kreis 20 Pixel frei lassen.  
// Liniendicke: 1 Punkt  
// Zeichenfarbe: dunkel-blau  
gc.setFill(Color.DARKBLUE);  
gc.setLineWidth(1.0);  
gc.fillOval(20, 20, width - 40, height - 40);  
  
// Linie oben in das Fenster zeichnen  
// Liniendicke: 5 Punkt  
// Zeichenfarbe: dunkel-rot  
gc.setStroke(Color.DARKRED);  
gc.setLineWidth(5.0);  
gc.strokeLine(10, 10, width - 10, 10);  
}  
}
```

- Eine **Canvas** hat normalerweise eine feste Größe, weil sie wie ein Bild behandelt wird.
- Die Größenänderung beim Verändern des Fensters kann erzwungen werden → siehe Beispielprojekt.



- Das Hauptprogramm ist dann sehr einfach (**CanvasTest**):

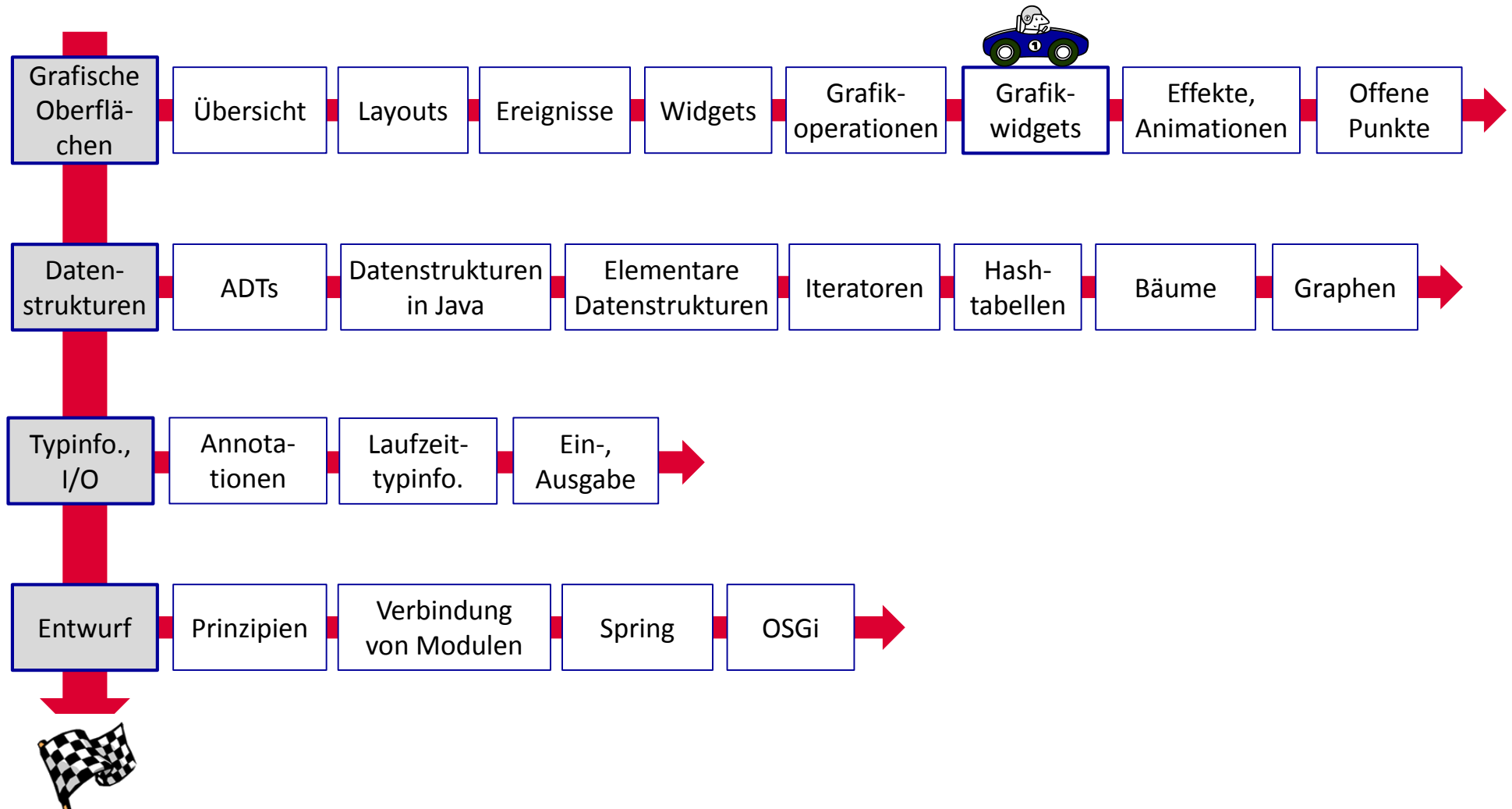
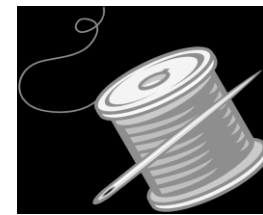
```
public class CanvasTest extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        primaryStage.setTitle("CanvasTest");  
  
        PaintCanvas canvas = new PaintCanvas();  
  
        VBox pane = new VBox();  
        pane.fillWidthProperty().set(true);  
        VBox.setVgrow(canvas, Priority.ALWAYS);  
        pane.getChildren().add(canvas);  
  
        Scene scene = new Scene(pane);  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```



- Die einfachste Art der Interaktion des Benutzers mit selbst gezeichneten Komponenten ist ein Beobachter vom Typ **EventHandler<MouseEvent>**:
 - ◆ **setOnMouseClicked**: Eine Maustaste wurde innerhalb der Komponente gedrückt und wieder losgelassen.
 - ◆ **setOnMouseDragged**: Der Mauszeiger wird mit gedrückter Maustaste bewegt.
 - ◆ **setOnMousePressed**: Die Maustaste wurde gedrückt und noch nicht losgelassen.
 - ◆ **setOnMouseReleased**: Die Maustaste wurde wieder losgelassen.
 - ◆ und weitere
 - ◆ → kommt gleich genauer

Grafikwidgets

Übersicht





- Die Grafikoperationen sind nicht immer optimal:
 - ◆ Es wird stets die komplette Fläche neu gezeichnet → kostet viel Zeit.
 - ◆ Die Grafikoperationen lassen sich schlecht in einem Layout mit anderen Widgets zusammen verwenden.
- Alle Figuren, die mit den Grafikoperationen gemalt werden können, gibt es auch als **Node**-Objekte für den Szenen-Graph:
 - ◆ Das Zeichnen übernimmt JavaFX → es werden nur die veränderten Objekte neu gezeichnet.
 - ◆ Die Figuren können mit anderen Widgets gemischt im Layout eingesetzt werden.
 - ◆ Die Figuren lassen sich sehr einfach animieren (kommt noch).
 - ◆ Die Figuren befinden sich im Paket **javafx.scene.shape**.

javafx.scene.shape

Class Line

```
java.lang.Object
  javafx.scene.Node
    javafx.scene.shape.Shape
      javafx.scene.shape.Line
```

All Implemented Interfaces:

EventTarget



- Beispiel mit absoluter Platzierung der Figuren in einem **Pane**-Objekt (also ohne Layout-Manager):

```
vBox = new VBox(); // Pane in ein Layout einsetzen
vBox.fillWidthProperty().set(true);

Pane field = new Pane();
field.setOpacity(1.0);
vBox.getChildren().add(field);
VBox.setVgrow(vBox, Priority.ALWAYS);

Ellipse ball = new Ellipse(600, 20, BALL_RADIUS, BALL_RADIUS);
ball.setFill(Color.DARKBLUE);
ball.setEffect(new GaussianBlur());
field.getChildren().add(ball);
```



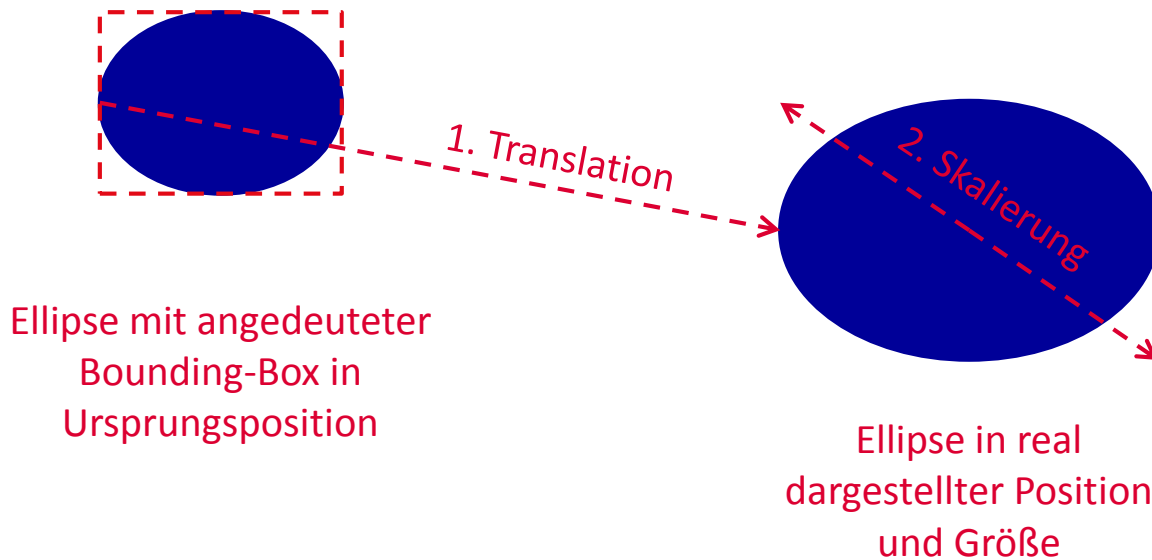
- Darstellung der Figuren:
 - ◆ **void setFill(Paint paint)**: Füllfarbe, Füllmuster, ... festlegen
 - ◆ **void setStroke(Paint paint)**: Rahmenfarbe
- Positionierung der Figuren (ohne dritte Dimension):
 - ◆ **void setLayoutX(double xOffset), void setLayoutY(double yOffset)**: Verschiebung der Figur um einen Offset
 - ◆ **void relocate(double xOffset, double yOffset)**: entspricht den Aufrufen von **setLayoutX(xOffset), setLayoutY(yOffset)**
 - ◆ **void setTranslateX(double x), void setTranslateY(double y)**: Wert, um den die Figur nachträglich (nach der Platzierung) verschoben wird
- Streckung/Stauchung der Figuren:
 - ◆ **void resize(double width, double height)**: Änderung der Größe der Bounding-Box
 - ◆ **void setScaleX(double x), void setScaleY(double y)**: Wert, um den die Bounding-Box vergrößert wird, Zentrum der Skalierung ist die Mitte der Figur



- Rotation der Figuren:
 - ◆ **void setRotationAxis(Point3D axis)**: Rotationsachse, vordefiniert sind **Rotate.X_AXIS**, **Rotate.Y_AXIS**, **Rotate.Z_AXIS** → soll hier nicht besprochen werden
 - ◆ **void setRotate(double angle)**: Winkel, um den gedreht wird → soll hier nicht besprochen werden
- Weitere Methoden für Linienarten, Linienverbindungen, abgerundete Kanten, ...
- Jede spezielle Figur hat auch Methoden, die „intuitivere“ Manipulationen sowie absolute Positionierung erlauben. Beispiele:
 - ◆ Ellipse: **setCenterX(double x)**, **setCenterY(double y)**, **setRadiusX(double x)**, **setRadiusY(double y)**
 - ◆ Line: **setStartX(double x)**, **setStartY(double y)**, **setEndX(double x)**, **setEndY(double y)**

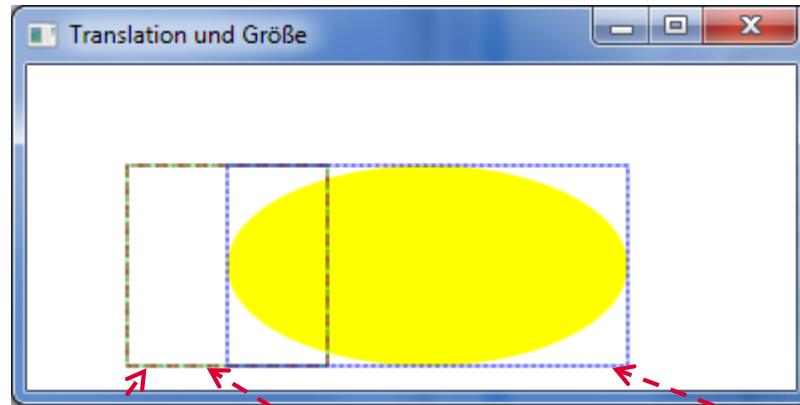


- Übersicht zu Translation und Skalierung:





- Quellcode zum Beispiel (die Bounding-Box wird berechnet und als Rechteck platziert):



Bounding-Box
(getBoundsInLocal(),
getBoundsInParent())
Ursprungsposition
und Originalgröße

Bounding-Box
(getBoundsInLocal())
nach Verschiebung
und Skalierung

Bounding-Box
(getBoundsInParent())
nach Verschiebung
und Skalierung

Grafikwidgets

Bounding-Box



```
Ellipse ellipse = new Ellipse(100, 100, BALL_RADIUS, BALL_RADIUS);
ellipse.setFill(Color.YELLOW);

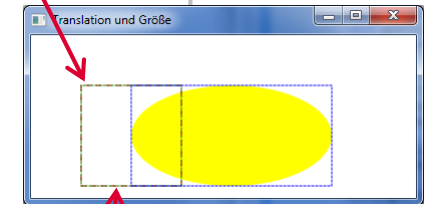
// Ursprungs-Bounding-Box ohne Verschiebung und Skalierung
Bounds bounds = ellipse.getBoundsInLocal();
Rectangle boundsOrig = new Rectangle(bounds.getMinX(), bounds.getMinY(),
                                     bounds.getWidth(), bounds.getHeight());

boundsOrig.setStroke(Color.RED);
boundsOrig.getStrokeDashArray().addAll(5.0, 5.0); // Linienmuster
boundsOrig.setFill(null); // keine Füllung

// Ellipse verschieben und Skalieren
ellipse.setTranslateX(100);
ellipse.setScaleX(2);

// Bounding-Box der Ellipse markieren, ohne dass Verschiebung und Skalierung
// berücksichtigt werden
bounds = ellipse.getBoundsInLocal();
Rectangle boundsOrig = new Rectangle(bounds.getMinX(), bounds.getMinY(),
                                     bounds.getWidth(), bounds.getHeight());

boundsOrig.setStroke(Color.GREEN);
boundsOrig.getStrokeDashArray().addAll(2.0, 2.0);
boundsOrig.setFill(null);
```

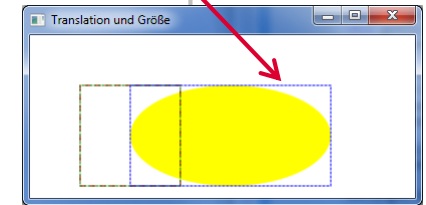




```
bounds = ellipse.getBoundsInParent();
Rectangle boundsVisible = new Rectangle(bounds.getMinX(), bounds.getMinY(),
                                         bounds.getWidth(), bounds.getHeight());

boundsVisible.setStroke(Color.BLUE);
boundsVisible.getStrokeDashArray().addAll(2.0, 2.0);
boundsVisible.setFill(null);

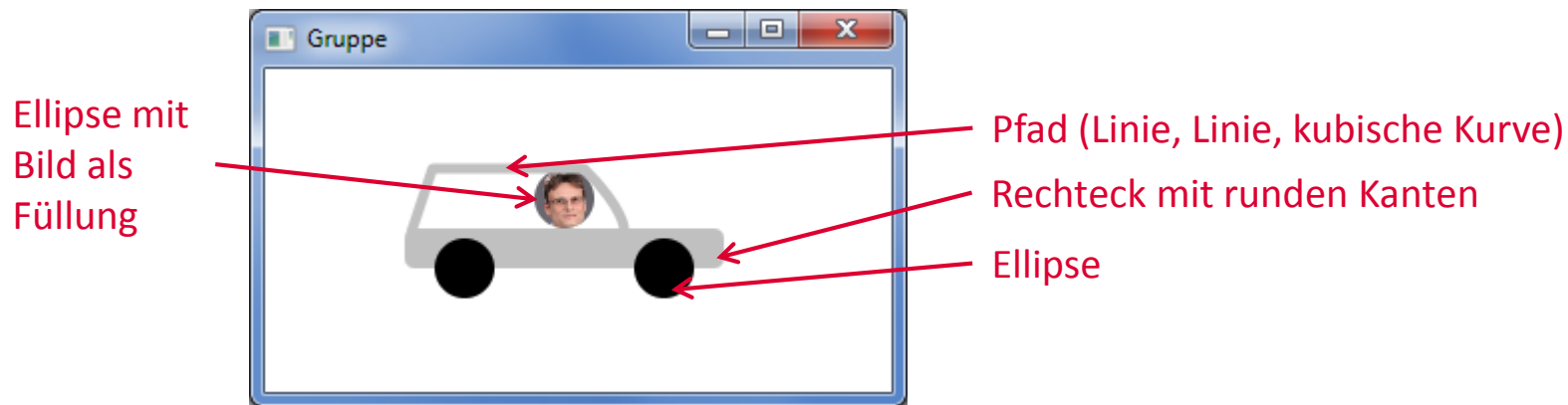
// Alle Figuren zur Zeichenfläche hinzufügen
field.getChildren().add(ellipse);
field.getChildren().add(boundsOrig);
field.getChildren().add(boundsNew);
field.getChildren().add(boundsVisible);
```



- Bounding-Boxen:
 - ◆ **Bounds** `getBoundsInLocal()`: untransformierte Bounding-Box (**ohne** Berücksichtigung von Verschiebung und Skalierung)
 - ◆ **Bounds** `getBoundsInParent()`: transformierte Bounding-Box (**mit** Berücksichtigung von Verschiebung und Skalierung)
 - ◆ **Bounds** `getLayoutBounds()`: Bounding-Box zur Berechnung des Layouts (Größe ist bei **Group** usw. 0)



- Zusammengesetzte Figuren:
 - ◆ Die einzelnen Figuren bieten nur primitive Grafiken wie Ellipsen usw.
 - ◆ Die Klasse **Group** kann mehrere primitive Figuren zu einer Figur gruppieren.
 - ◆ Die Gruppe ist selbst wiederum eine Figur, die ihre Kindfiguren selbst verwaltet.
 - ◆ Die Gruppe berechnet ihre Bounding-Box anhand der Positionen und Größen ihrer Kindern.
 - ◆ Beim Verschieben, Skalieren oder Rotieren einer Gruppe passt die Gruppe ihre Kinder ebenfalls an.
- Beispiel:





- Code zum Beispiel:

```
Group car = new Group();

Ellipse wheel1 = new Ellipse(100, 100, 15, 15);
wheel1.setFill(Color.BLACK);

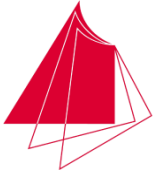
Ellipse wheel2 = new Ellipse(200, 100, 15, 15);
wheel2.setFill(Color.BLACK);

Rectangle body = new Rectangle(70, 80, 160, 20);
body.setFill(Color.SILVER);
body.setArcHeight(10.0);
body.setArcWidth(10.0);

ImagePattern pattern = new ImagePattern(new Image("file:images/hv.gif"));
Ellipse driver = new Ellipse(150, 65, 15, 15);
driver.setFill(pattern);

// usw...
car.getChildren().addAll(body, wheel1, wheel2, driver, roof);

field.getChildren().add(car);
```



- Interaktion mit dem Widgets
 - ◆ Das Prinzip der Ereignisbehandlung wurde bereits vorgestellt.
 - ◆ Es gibt weitere Ereignistypen, die gerade für die primitiven Grafik-Widgets hilfreich sind → funktionieren aber auch bei den „normalen“ Widgets.
- Wichtige Ereignistypen je nach Unterstützung durch das Gerät:
 - ◆ Mausereignisse (klicken, „draggen“, bewegen)
 - ◆ Gesten (drehen, verschieben, zoomen, ...)
 - ◆ Tastaturereignisse
- Die Registrierung als Beobachter erfolgt über die Methode mit den Namen **setOnXYZEvent**.
Beispiel für „draggen“ per Maus:
 - ◆ **setOnMousePressed**: Die Maustaste wurde gedrückt und ist eventuell noch gedrückt → Startposition der Maus merken.
 - ◆ **setOnMouseDragged**: Die Maus wird mit gedrückter Taste bewegt → „draggen“.



- Meldung der Ereignisse durch „Event-Bubbling“:
 - ◆ JavaFX ermittelt die Figur, die sich bei einem Event unter dem Mauszeiger befindet.
 - ◆ Alle Beobachter an dieser Figur werden benachrichtigt.
 - ◆ Danach werden alle Vaterfiguren der Figur ebenfalls benachrichtigt.
 - ◆ Die Ereignis-Objekte steigen wie Blasen an die Oberfläche des Szene-Graphen.
 - ◆ Soll das Ereignis nicht weitergemeldet werden, dann kann die Zustellungskette unterbrochen werden. Beispiel:

```
field.setOnMousePressed(new EventHandler<MouseEvent>() {  
    @Override  
    public void handle(MouseEvent event) {  
        Node node = (Node) event.getTarget();  
        if (node instanceof Shape) {  
            // Mache etwas  
            event.consume(); // Weiterleitung "nach oben" verhindern  
        }  
    }  
});
```



- Es könnte auch ein Lambda-Ausdruck angegeben werden, der eine Methode mit dem eigentlichen Code aufruft:

```
field.setOnMousePressed(this::handleEvent);
```

Die Methode **handleEvent** ist in derselben Klasse implementiert.



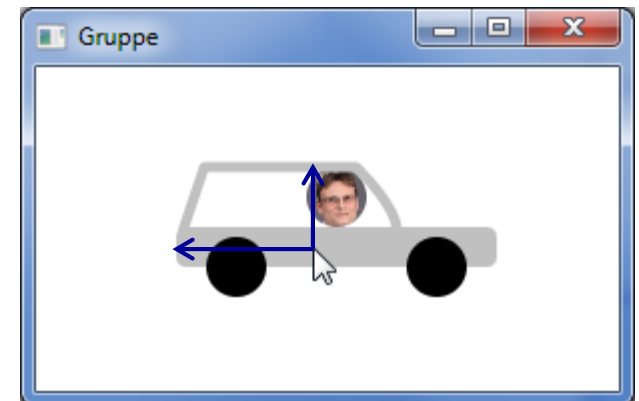
- Beispiel zum „draggen“ des Fahrzeugbildes (der Gruppe), kleines Problem:
 - ◆ Die Gruppe wird nie als Ziel des Ereignisses gemeldet, weil sie unsichtbar ist.
 - ◆ Stattdessen wird immer die unter dem Mauszeiger befindliche Figur zurückgegeben.
 - ◆ Ausweg: Suchen des obersten Gruppen-Elementes, um dieses zu verschieben → es kann sich ja auch um eine Hierarchie aus mehreren geschachtelten Gruppen handeln.

- Suche des obersten Gruppen-Knotens:

```
private Node getBaseParent(Node node) {  
    // Vater suchen  
    while (node.getParent().getClass() == Group.class) {  
        node = node.getParent();  
    }  
    return node;  
}
```

- Startpositionen des Mauszeigers innerhalb der ausgewählten Figur zu Beginn der Aktion:

```
private double dragStartOffsetX;  
private double dragStartOffsetY;
```

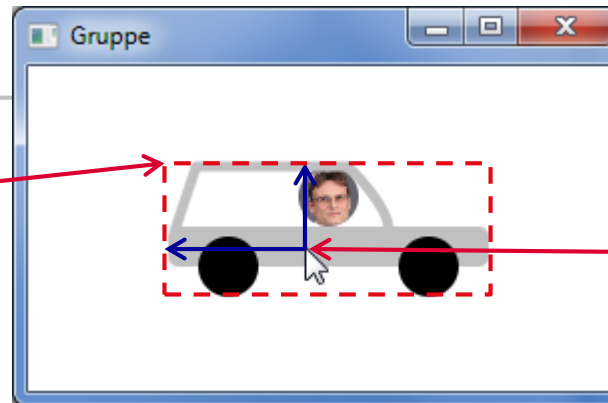




- Beim Klick auf die Figur die Startposition des Mauszeigers relativ zur Figur merken:

```
field.setOnMousePressed(new EventHandler<MouseEvent>() {  
    @Override  
    public void handle(MouseEvent event) {  
        Node node = (Node) event.getTarget(); // angeklickte Figur  
        if (node instanceof Shape) { // Klick auf Pane ignorieren  
            // Vater suchen  
            node = getBaseParent(node);  
            dragStartOffsetX = event.getSceneX() - node.getBoundsInParent().getMinX();  
            dragStartOffsetY = event.getSceneY() - node.getBoundsInParent().getMinY();  
            event.consume();  
        }  
    }  
});
```

`node.getBoundsInParent().getMinX(),`
`node.getBoundsInParent().getMinY():`
linke obere Ecke der
Bounding-Box der Gruppe



`event.getSceneX(),`
`event.getSceneY():`
Mauszeigerposition
innerhalb der Szene



- Verschieben der Figur:

```
field.setOnMouseDragged(new EventHandler<MouseEvent>() {  
    @Override  
    public void handle(MouseEvent event) {  
        Node node = (Node) event.getTarget();  
        if (node instanceof Shape) {  
            // Vater suchen  
            node = getBaseParent(node);  
            node.relocate(event.getSceneX() - dragStartOffsetX,  
                           event.getSceneY() - dragStartOffsetY);  
            event.consume();  
        }  
    }  
});
```



■ Gesten unter Windows:

GESTURE	WINDOWS USAGE	GESTURE ACTION	ACTION (○ = finger down ◯ = finger up)	Single Contact	Multi Contact
Tap / Double Tap	Click / Double Click			★	★
Panning with Inertia	Scrolling	Drag 1 or 2 fingers up and down			★
Selection / Drag (left to right with one finger)	Mouse Drag / Selection	Drag one finger left / right		★	★
Press and Tap	Right-click	Press on target and tap using a second finger			★
Zoom	Zoom (defaults to CTRL key + Scroll wheel)	Move two fingers apart / toward each other			★
Rotate	No system default unless handled by Application (using WM_GESTURE API)	Move two fingers in opposing directions -or- Use one finger to pivot around another			★
Two-Finger Tap	N/A – Exposed through Gesture API, used by Application discretion.	Tap two fingers at the same time (where the target is the midpoint between the fingers)			★
Press and Hold	Right-click	Press, wait for blue ring animation to complete, then release		★	★
Flicks	Default: Pan up/ Pan Down/ Back, and Forward	Make quick drag gestures in the desired direction		★	★



- Beispiel, Geste zur Drehung der Gruppe:

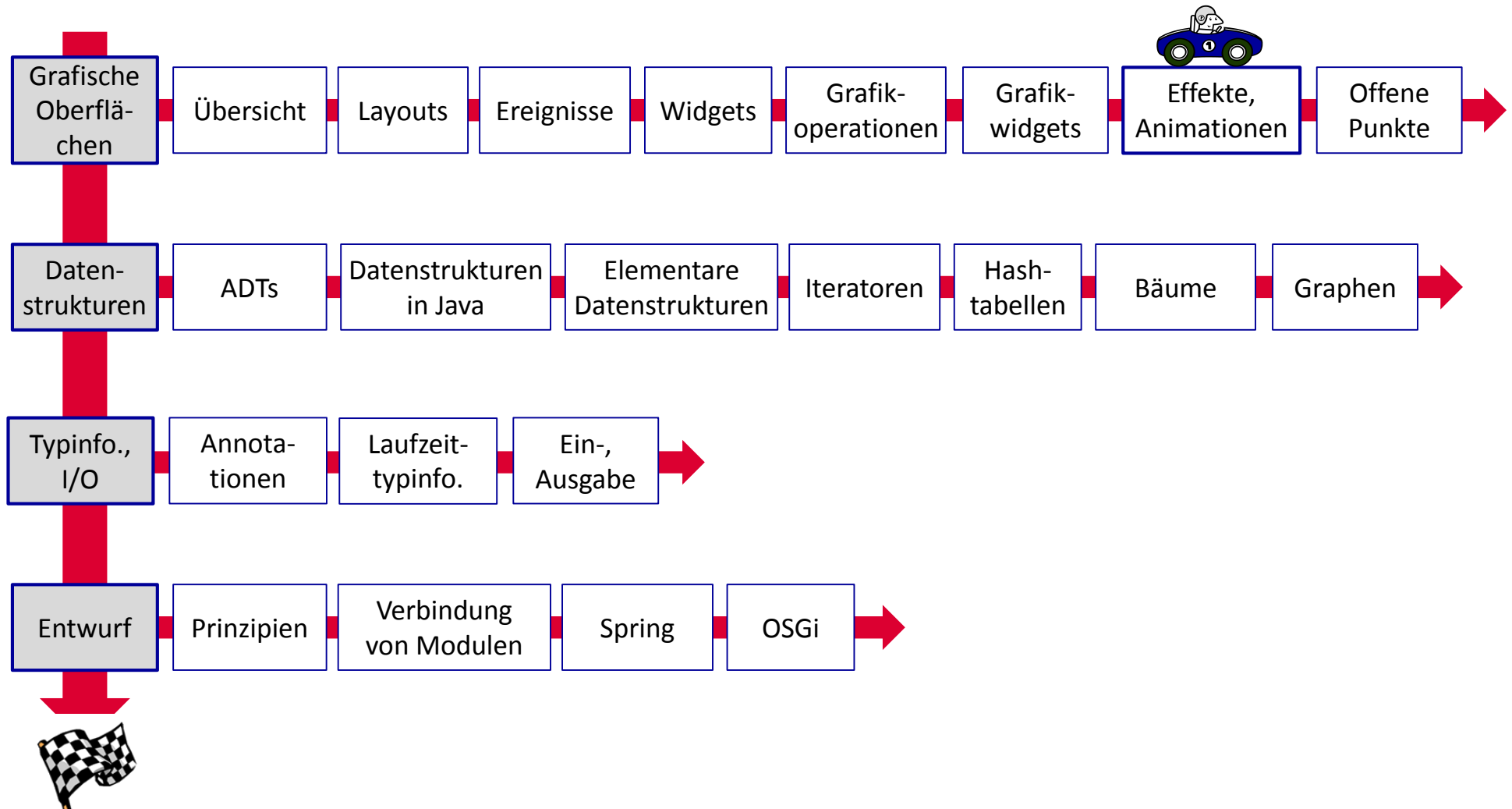
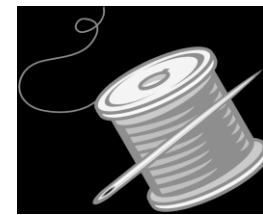
```
field.setOnRotate(new EventHandler<RotateEvent>() {  
    @Override public void handle(RotateEvent event) {  
        Node node = (Node) event.getTarget();  
        if (node instanceof Shape) {  
            node.setRotate(node.getRotate() + event.getAngle());  
            event.consume();  
        }  
    }  
});
```

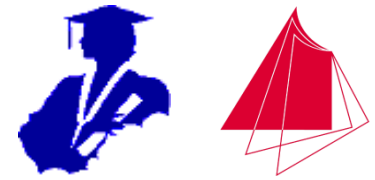
- Beispiel, Geste zur Skalierung der Gruppe:

```
field.setOnZoom(new EventHandler<ZoomEvent>() {  
    @Override public void handle(ZoomEvent event) {  
        Node node = (Node) event.getTarget();  
        if (node instanceof Shape) {  
            node.setScaleX(node.getScaleX() * event.getZoomFactor());  
            node.setScaleY(node.getScaleY() * event.getZoomFactor());  
            event.consume();  
        }  
    }  
});
```

Effekte und Animationen

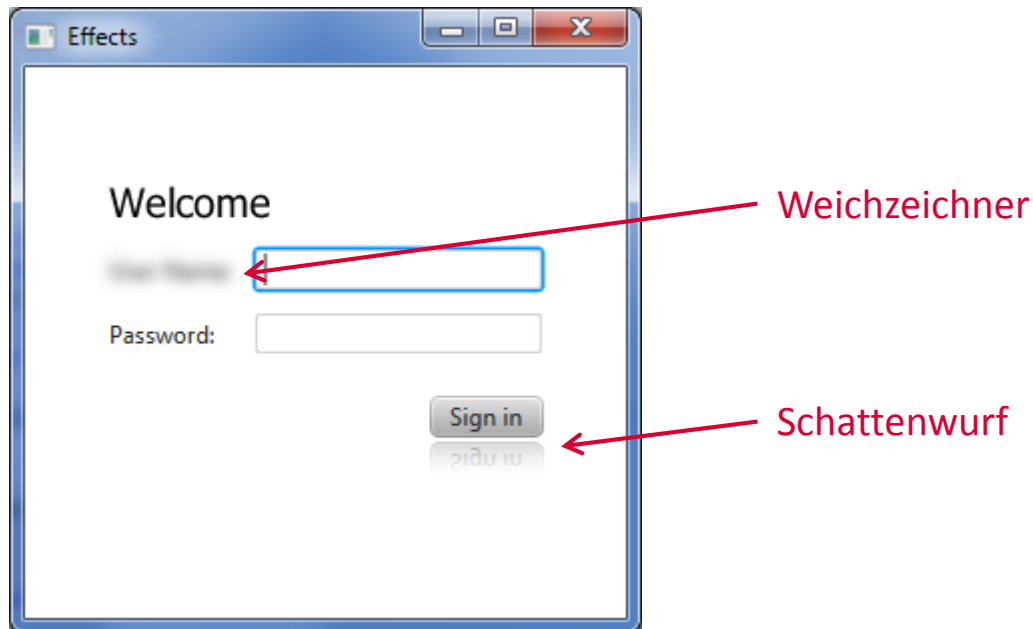
Übersicht

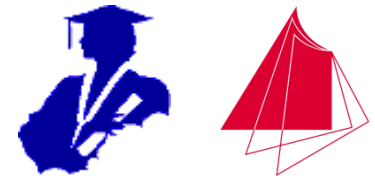




Effekte

- Allen Knoten-Objekten können mit **setEffect(Effect eff)** einzelne oder eine Kette von Effekten zugeordnet werden.
- Beispiel:





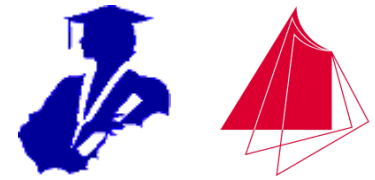
- Weichzeichner, Unschärfe (**BoxBlur**, **MotionBlur**, **GaussianBlur**)

```
Label userName = new Label("User Name:");  
grid.add(userName, 0, 1);  
userName.setEffect(new GaussianBlur());
```

- Reflektion:

```
Button btn = new Button("Sign in");  
btn.setEffect(new Reflection());
```

- Weitere Effekte:
 - ◆ Schatten
 - ◆ Licht-Effekte
 - ◆ Perspektive Verzerrungen
 - ◆ Verkettung von Effekten zu einem neuen Effekt
- Darüber hinaus lässt sich auch festlegen, wie übereinanderliegende Knoten erscheinen sollen („Überblendung“).



Es gibt eine ganze Menge vordefinierter Animationen, um Figuren zu bewegen oder zu verändern:

- **FadeTransition**: sanftes Ein- und Ausblenden einer Figur durch Änderung der Transparenz
- **FillTransition**: Änderung der Füllung einer Figur
- **RotateTransition**: Drehen einer Figur
- **ScaleTransition**: Größenänderung einer Figur
- **StrokeTransition**: Änderung der Rahmenfarbe einer Figur
- **PathTransition**: Bewegung einer Figur entlang eines Pfades
- **PauseTransition**: zeitliche Verzögerung, an deren Ende ein Ereignis ausgelöst wird
- **TranslateTransition**: Verschiebung einer Figur
- **ParallelTransition**: parallele Ausführung mehrerer Transitionen
- **SequentialTransition**: sequentielle Ausführung mehrerer Transitionen
- **TimelineTransition**: mächtigste Transaktion → kommt gleich genauer
- **Transition**: Basisklasse aller Transitionen, auch für eigene Transitionen



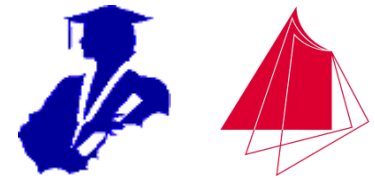
- Beispiel zur **FadeTransition** (aus dem JavaFX-Tutorial von Oracle):

```
// zu animierende Figur
final Rectangle rect1 = new Rectangle(10, 10, 100, 100);
rect1.setArcHeight(20);
rect1.setArcWidth(20);
rect1.setFill(Color.RED);
field.getChildren().add(rect1);

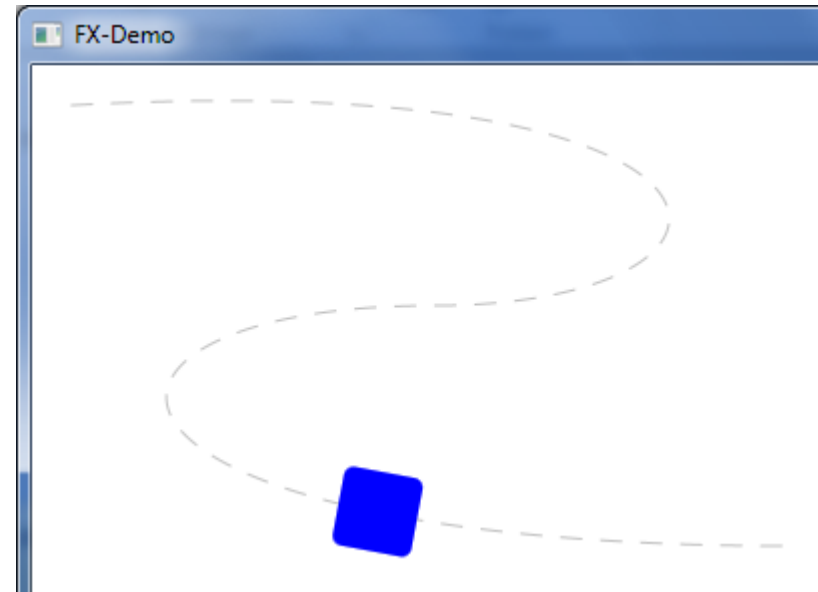
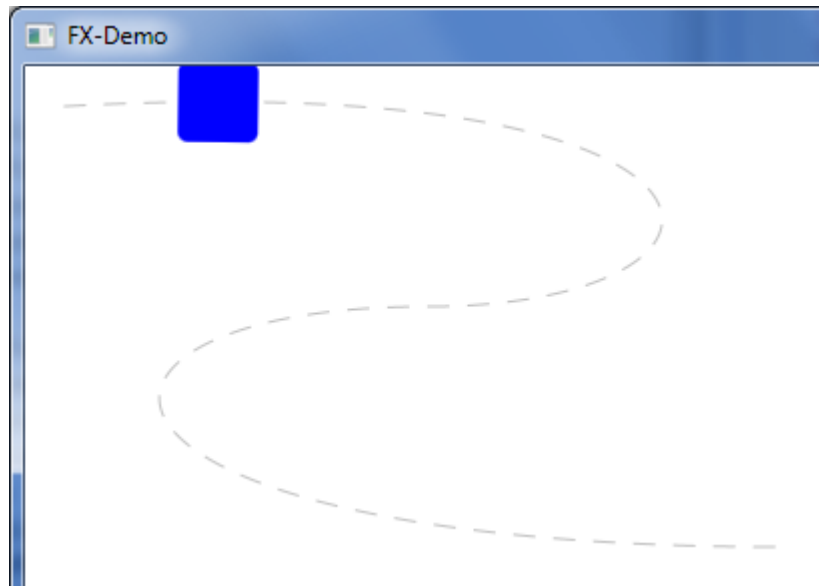
// Dauer eines Animationsschrittes: 3 Sekunden
FadeTransition ft = new FadeTransition(Duration.millis(3000), rect1);

// Transparenz zwischen 100% und 10% verändern
ft.setFromValue(1.0);
ft.setToValue(0.1);
// Animation läuft unendlich (immer Wechsel zwischen 100% und 10% Transparenz)
ft.setCycleCount(Timeline.INDEFINITE);
// Umkehr der Richtung bei Ablauf
ft.setAutoReverse(true);
// Start der Animation
ft.play();
```





- Beispiel zur **PathTransition** (aus dem JavaFX-Tutorial von Oracle), Rechteck bewegt sich entlang eines Pfades aus zwei Kurven und dreht sich mit, der Pfad wird hier angezeigt → ist für die Animation nicht erforderlich):





- Quelltext:

```
// zu animierende Figur
final Rectangle rectPath = new Rectangle (0, 0, 40, 40);
rectPath.setArcHeight(10);
rectPath.setArcWidth(10);
rectPath.setFill(Color.BLUE);

// Pfad, entlang dessen sich das Rechteck bewegen soll
Path path = new Path();
path.getElements().add(new MoveTo(20,20));
path.getElements().add(new CubicCurveTo(380, 0, 380, 120, 200, 120));
path.getElements().add(new CubicCurveTo(0, 120, 0, 240, 380, 240));

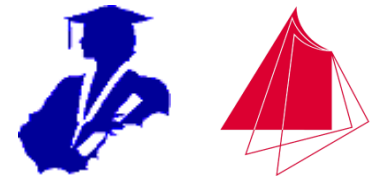
path.setStroke(Color.SILVER);
path.getStrokeDashArray().addAll(10.0, 10.0);

// Pfad zum Feld hinzufügen, damit er sichtbar wird -> nicht erforderlich
// für die Animation
field.getChildren().add(path);

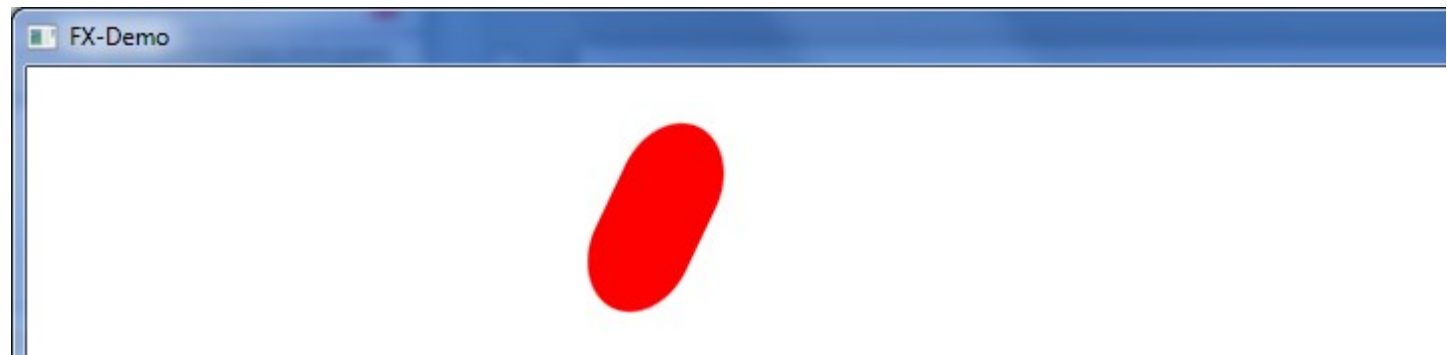
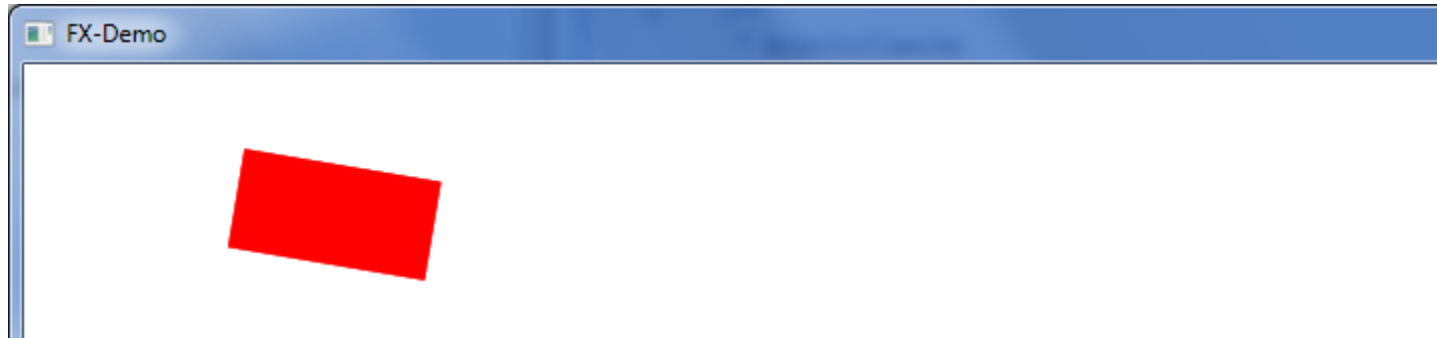
// Rechteck zum Feld hinzufügen
field.getChildren().add(rectPath);
```

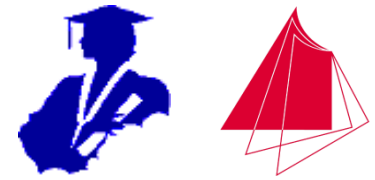



```
// Animation entlang eines Pfades
PathTransition pathTransition = new PathTransition();
// Ein Durchlauf dauert vier Sekunden
pathTransition.setDuration(Duration.millis(4000));
// Pfad, entlang dessen bewegt werden soll
pathTransition.setPath(path);
// Knoten, der bewegt werden soll
pathTransition.setNode(rectPath);
// Der Knoten soll sich mit dem Pfad mitdrehen (-> steht senkrecht darauf),
// Alternative ohne Drehung: NONE).
pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
// Animation soll unendlich laufen
pathTransition.setCycleCount(Timeline.INDEFINITE);
// Nach jedem Durchlauf soll die Richtung der Animation umgekehrt werden.
pathTransition.setAutoReverse(true);
// Animation starten
pathTransition.play();
```



- Zur **TimelineTransition** gehören zwei wichtige Klassen:
 - ◆ **KeyFrame**: Zeitpunkt, zu dem eine Animation geschehen soll
 - ◆ **KeyValue**: Eigenschaften, die zu diesem Zeitpunkt manipuliert werden sollen
- Beispiel (drehendes, sich bewegendes Rechteck, bei dem sich der Radius der Ecken ändert):

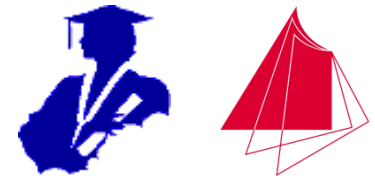




- Quelltext:

```
// zu animierende Figur
final Rectangle rectBasicTimeline = new Rectangle(100, 50, 100, 50);
rectBasicTimeline.setFill(Color.RED);
field.getChildren().add(rectBasicTimeline);

final Timeline timeline = new Timeline();
// unendliche Wiederholung der Animation
timeline.setCycleCount(Timeline.INDEFINITE);
// automatische Richtungsumkehr
timeline.setAutoReverse(true);
// x-Position des Rechtecks auf 300 Pixel ändern
final KeyValue kv1 = new KeyValue(rectBasicTimeline.xProperty(), 300);
// Rechteck um 360 Grad rotieren
final KeyValue kv2 = new KeyValue(rectBasicTimeline.rotateProperty(), 360);
// Rundungen der Ecken bis auf 80 Pixel vergrößern
final KeyValue kv3 = new KeyValue(rectBasicTimeline.arcHeightProperty(), 80);
final KeyValue kv4 = new KeyValue(rectBasicTimeline.arcWidthProperty(), 80);
// Animation läuft 2 Sekunden mit den angegebenen Manipulationen kv1 bis kv4.
final KeyFrame kf = new KeyFrame(Duration.millis(2000), kv1, kv2, kv3, kv4);
timeline.getKeyFrames().add(kf);
timeline.play();
```



- Was fehlt noch?
 - ◆ Animationen laufen in den Beispielen gleichmäßig ab. Durch sogenannte **Interpolator** lässt sich das Verhalten ändern (z.B. stark beschleunigen und sanft abbremsen) → siehe Beispiel mit dem fallenden Ball.
 - ◆ Alle Animationen können Ereignisse auslösen.

Effekte und Animationen

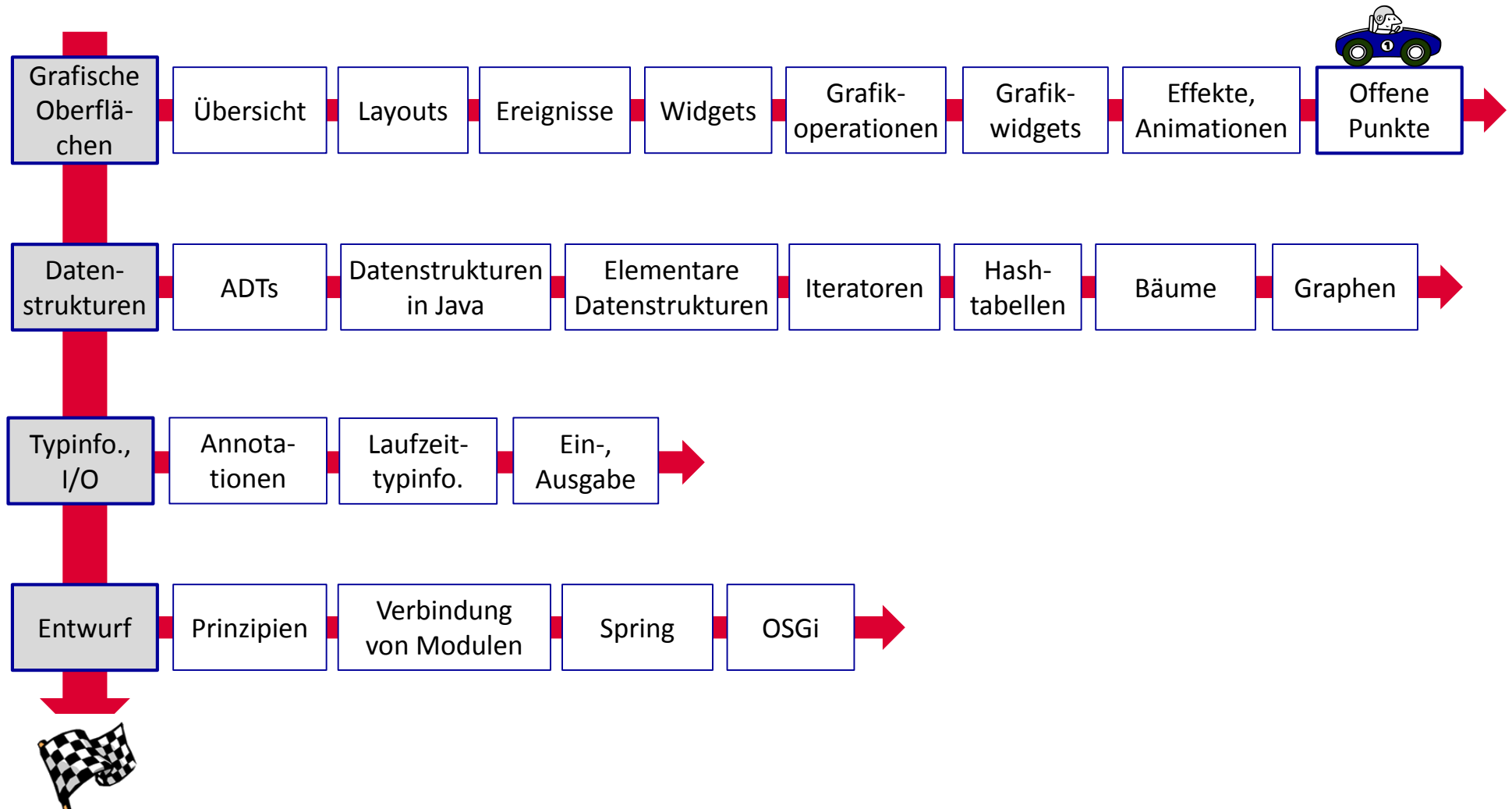
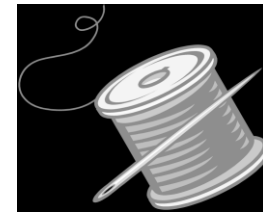
Animationen



- Anwendung des Wissens: Ein kleines Ballerspiel mit allen bisher erlernten Techniken.

Offene Punkte

Übersicht





- Es fehlen noch einige wirklich interessante Eigenschaften von JavaFX:
 - ◆ Properties, Data-Bindings → Java-Beans
 - ◆ Styling der Widgets mit CSS
 - ◆ FXML zur deklarativen Beschreibung der Oberfläche, wird vom SceneBuilder erzeugt
 - ◆ Audio- und Video-Behandlung
 - ◆ 3D-Darstellungen