



# **Modul 404**



## **Objektbasiert programmieren nach Vorgabe**

# Allgemeine Informationen

## Modulbeschreibung

-  [Modulidentifikation](#)
-  [LBV](#)

## Faktenblätter


-  [Faktenblatt 404](#)
-  [Faktenblatt 403](#)



### TIP

Wenn Ihr die Faktenblätter **aktiv** verwendet, sind sie euch eine bessere Hilfe bei der Prüfung!

# Quartalsübersicht

	Woche	Datum	Thema	Leistungsbewertung
	Woche 1	26.01.2024	Repetition / Klassen / Objekte	
	Woche 2	02.02.2024	Fachklassen / Static / Swing	
	Woche 3	09.02.2024	Klassenstruktur / Java Interface - ActionListener	
		16.02.2024 / 23.02.2024	☆ Fasnachtsferien ☆	
	Woche 4	01.03.2024	Konstruktor / Formatierung / UML	
	<b>Woche 5</b>	<b>08.03.2024</b>	<b>LB 1</b> / Projektstart	<b>LB1</b>
	Woche 6	15.03.2024	🌸 <b>Lehrprobe</b> Kickoff Projekt / Arbeiten am Projekt	
	Woche 7	22.03.2024	Arbeiten am Projekt	
		29.03.2024 / 05.04.2024	☆ Osterferien ☆	
	Woche 8	12.04.2024	Arbeiten am Projekt	
	<b>Woche 9</b>	<b>19.04.2024</b>	<b>Abgabe / Präsentation des Projekts</b>	<b>LB2</b>

# Woche 1



## Präsentation

 [Open in Browser](#) |  [download PDF](#)



[Ablauf im Detail](#)




## Input

- [Modulidentifikation](#)
- [Fachklasse](#)



## Aufgaben

### Grundlagen

- [Repetition Modul 319/403](#) (Lernkarten)
- [Account Applikation](#) ( **Live-Coding**)
- [Analyse Account Applikation](#)
- [Refactoring Account Analyse](#)



### Swing

- Startet nächste Woche!

# Woche 2



## Präsentation

 [Open in Browser](#) |  [download PDF](#)



[Ablauf im Detail](#)



## Input

- [Repetition Fachklassen](#)
- [Static](#)
- [Einstieg in Swing](#)



## Aufgaben

### Grundlagen

- [Starterklasse](#)
- [Fahrenheit-Celsius Konverter](#)



### Swing

- [Fenster \(JFrame\)](#)
- [JFrame Komponenten](#)
- [:superhero: Optional Repetition Arrays \(PDF\)](#)

# Woche 3



## Präsentation

 [Open in Browser](#) |  [download PDF](#)



[Ablauf im Detail](#)



## Input

- [Projekt Klassenstruktur](#)
- [Java Interface](#)



## Aufgaben



### Swing

- [Buttons aktivieren](#)
- [Strings in Zahlen umwandeln](#)
- [Account Applikation in Swing](#)



### Optional

- [Weitere Swing-Aufgaben](#)
- vertieftes Wissen zu Interfaces **in den Folien am Schluss!**

### Grundlagen

- nächste Woche wieder

# Woche 4



## Präsentation

 [Open in Browser](#) |  [download PDF](#)



[Ablauf im Detail](#)



## Input


- Kontoverwaltung in Swing (Erläutern)
- [Formatierung](#)
- [UML Klassendiagramm](#)
- [Konstruktor](#)



## Aufgaben



### Swing

- [Konstruktor](#)
- [DiceGame](#) ( **Live-Coding**)



### Optional

- [Weitere Swing-Aufgaben](#)

### Grundlagen

- [Formatierung](#)
- [Account UML](#)

# Woche 5



## Präsentation

 [Open in Browser](#) |  [download PDF](#)



## Input

- [UML](#) kurze Repetition



## Aufgaben

-  [LB1](#)
-  [Projektstart](#)



# Woche 6



## Präsentation

 [Open in Browser](#) |  [download PDF](#)






## Input

- Klassenstruktur Beispiel wiederholen
- Beispielprojekt zeigen



## Aufgaben

-  [LB1](#)
-  Arbeiten am [Projekt](#)
-  [JPanel](#), wichtige Aufgabe um den Punktestand anzuzeigen

# Woche 7



## Präsentation



 [Open in Browser](#) |  [download PDF](#)



[Ablauf im Detail](#)



## Aufgaben

-  Arbeiten am [Projekt](#)
-  [PlayerPanel](#)

# Woche 8




## Präsentation

 [Open in Browser](#) |  [download PDF](#)



## Aufgaben

-  Arbeiten am [Projekt](#)

# Woche 9




## Präsentation

 [Open in Browser](#) |  [download PDF](#)



## Aufgaben

-  Fertigstellung vom [Projekt](#)
- Modulende!

## **LB 1**

| Dauer | Gewichtung |

## **LB 2 - Projekt**

| Dauer | Gewichtung |



# LB 1

Dauer	Gewichtung
90 Minuten	50% der Modulnote



## Mögliche Aufgaben

- Multiple-Choice-Fragen beantworten
- **Klassen** schreiben oder ergänzen
- Konsolen-Programm schreiben (ähnlich der **Account Applikation**)
- **Starterklasse** schreiben
- **Swing-Programm** schreiben (Erweitern von `JFrame`)
- **Swing-Komponenten** anwenden (`JLabel`, `TextField`, `Button`)
- **ActionListener** implementieren
- **Klassendiagramm** zeichnen
  - Programmieren gemäss einem Klassendiagramm
- **Konstruktor** anwenden

### TIP

-  Vergesst nicht wie ein Konsolen-Programm, **ohne Swing**, geschrieben wird!
-  Vergewissert euch, dass ihr eine Fachklasse auch alleine in einer `main` Methode verwenden könnt!

## Hilfsmittel

-  [Faktenblatt 404](#)
-  [Faktenblatt 403](#)

### DANGER

- Das Faktenblatt, **doppelseitig** ausdrucken, es sind nur **handgeschriebene Notizen** darauf erlaubt!

# LB 2 - Projekt

Dauer	Gewichtung
15.03.2024 - 19.04.2024	50% der Modulnote

## **PLAGIAT**

- Sollten Plagiate (auch Teilplagiate) abgegeben werden, so erhalten alle Beteiligten für das Projekt die **Note 1**
- Es wird darauf verzichtet, Nachforschungen anzustellen, wer Quelle und wer Empfänger war

## **PRIVATES GIT REPOSITORY**

- Wenn Ihr Euren Code mit **git** versioniert ist das super!
- Aber bitte verwendet ein **privates Repository** auf GitHub und Co., damit nicht zukünftige Kameraden von Ihnen klauen können!

## Einleitung

Im Modul 403 wurden nur konsolenbasierte Programme realisiert. Mittlerweile haben Sie die Fähigkeit, Programme mit einer grafischen Oberfläche zu schreiben.

## Ziele

- Die Anforderungen an eine Fachklasse erforschen und formulieren.
- Eine grafische Benutzeroberfläche programmieren.
- Die eigene Arbeit mit Klassendiagrammen [dokumentieren](#).
- Sauberen Quellcode schreiben.
- Programmcode und **Dokumentation** im eigenen Subordner des **Google Drive Abgabeordner** hochgeladen

## Aufgabenstellung

Es geht darum, ein Würfelspiel für 2 Mitspieler zu programmieren. Das Ziel des Spieles ist es, in einer gegebenen Anzahl Runden möglichst viele Punkte zu erzielen.

Pro Runde darf ein Spieler maximal 5 Mal würfeln. Die gewürfelten Augenzahlen ergeben die Rundenpunkte. Sie werden zu der Gesamtpunktzahl des Spielers addiert.

**Aber Achtung:** es werden nur die geraden Zahlen aufaddiert. Sobald der Spieler eine ungerade Zahl würfelt, verliert er die Rundenpunkte und der andere Spieler ist an der Reihe. Der Spieler muss die Möglichkeit haben, die Runde abubrechen und weniger als fünfmal in seiner Runde zu würfeln.

Hier sehen Sie einen möglichen Spielverlauf mit einer Visualisierung:

 Spielverlauf

Wer am Ende der festgelegten Rundenzahl am meisten Punkte hat, gewinnt das Spiel.

Ihr Programm muss am Ende nicht genau so aussehen wie oben gezeigt. So eine Visualisierung des Spielverlaufs mit Würfelaugen und mehreren Runden ist aufwändig, ergibt dafür zusätzliche Punkte.

Es sind auch Spielvarianten denkbar. So könnte zum Beispiel zu Beginn festgelegt werden, dass die ungeraden Zahlen aufaddiert werden und die geraden Zahlen die Spielverderber sind.

Oder man könnte eine Augenzahl als Spielverderber festlegen und alle anderen Augenzahlen addieren.

## Produkt

Sie erstellen selbständig ein Programm gemäss der obigen Spielbeschreibung. Das Programm muss eine grafische Oberfläche besitzen.

Ausserdem besteht Ihr Produkt aus (mindestens) drei Klassen:

- grafische Darstellung des Spiels



- Starterklasse
- Fachklasse(n), die den Spielstand verwalten und logische Funktionen anbieten

Der Quellcode ist sauber formatiert. Alle Klassen, Methoden und Attribute haben aussagekräftige Namen.

Die von Ihnen erstellten Klassen dokumentieren Sie mit einem UML-Klassendiagramm, das alle Klassen enthält. Auch die Beziehungen zwischen den Klassen müssen dargestellt sein. In einer [Dokumentation](#) beschreiben Sie Ihr Programm. Notieren Sie, was Sie wie herausgefunden haben. Wenn zur Zeit der Abgabe noch nicht alles funktioniert, dann beschreiben Sie was nicht funktioniert und was das Problem ist.

Das Klassendiagramm gehört in die [Dokumentation](#).

## Bewertung

	Punkte	Kriterium
	1	Die Starterklasse funktioniert.
	3	Das Programm läuft und lässt das beschriebene Spiel zu ( <i>Spielregeln</i> ).
	2	Die Spieler können Ihre Namen selber eingeben.
	2	Der aktuelle Punktestand, wird pro Spieler angezeigt.
	2	Der aktuelle Spieler kann eine Runde beenden.
	2	Die Benutzerführung ( <i>UX</i> ) macht Sinn ( <i>Buttons werden ein/ausgeblendet resp. de/aktiviert</i> ).
	2	Die gewürfelten Punkte pro Runde werden als Text angezeigt.
	3	Der Würfelverlauf wird über mehrere Runden angezeigt ( <i>nicht nur die aktuelle</i> ).
	3	Der Quellcode ist korrekt formatiert ( <i>Einrückungen</i> ).
	3	Die Namen der Klassen, Methoden und Attribute sind aussagekräftig und einheitlich.
	3	<i>Dokumentation:</i> die Beschreibung des Programms ist vollständig.
	3	<i>Dokumentation:</i> die Reflexion ist tiefgründig und geht auf die Probleme ein.
😊	(29)	<b>Volle Punktzahl bis hier ergeben eine <u>5.0</u>.</b>

	Punkte	Kriterium
	2	Der <b>Verlauf des Punktestands</b> aller bereits gespielten Runden wird pro Spieler angezeigt.
	2	Die gewürfelten Punkte pro Runde werden <b>bildlich</b> dargestellt.
	3	Die <b>Fachklassen</b> beinhalten die <b>Logik</b> / verwalten den <b>Spielstand</b> (ohne UI-Elemente).
	<b>36</b>	<b>TOTAL</b>
		<b>Bonus</b>
	2	<i>Das Programm übersteigt die oben beschriebenen Minimalanforderungen an das Spiel <b>wesentlich</b> und <b>ist auch vollumfänglich lauffähig</b>.</i>

## Termin

Das a Produkt (inklusive der [Dokumentation](#)) muss bis am **Freitag, 19.04.2024, 16:15** über den eigenen Unterordner im [Google Drive Abgabeordner](#) abgegeben werden.

### KURZES VERSTÄNDNISGESPRÄCH

Nach der Pause, **um 14:40**, werde ich kurz bei jedem vorbeischaun, den Code durchsehen und kurze Verständnisfragen stellen! Diese werden nicht benotet.

# Strukturidee

 Projektstruktur

## **Lernkarten**

Um Ihr Wissen aus Modul 403 etwas aufzufrischen, existieren Lernkarten in

## **Arrays**

Arrays sind ein besonderer Datentyp, da sie es erlauben, mehrere Werte in einer

## **Basics**

Hier werden nochmals in Kürze alle Konzepte vom Modul 319 aufgeführt

# Lernkarten

Um Ihr Wissen aus Modul 403 etwas aufzufrischen, existieren Lernkarten in [Card2Brain](#).

- Auf [Card2Brain](#) können Sie sich mit Ihrem GIBM Schul-Account (**eXXX** ...) anmelden.
- Unter [Card2Brain](#) können Sie über **Jetzt lernen → Frage zuerst** eine Karte nach der anderen bearbeiten.

## Aufgabe

**Sozialform:** Partnerarbeit (Zufällig zugeteilt)

**Vorgehen:**

- Start über "Jetzt lernen" → "Frage zuerst"
- Frage lesen
- Kurznotiz zur Antwort, danach mit Partner besprechen
- Vergleich der eigenen Antwort mit der "Lösung"

! **DIE KARTEN WERDEN AUCH IM UNTERRICHT IN PAPIERFORM AUSGETEILT! :::**

# Arrays

Arrays sind ein besonderer Datentyp, da sie es erlauben, mehrere Werte in einer Variablen zu speichern. Diese Variable kann dazu mit einem `int` Wert angesprochen werden, der als Index benutzt wird, und den Zugriff auf die unterschiedlichen Werte ermöglicht.

Anschliessend findet Ihr Aufgaben für die wichtigsten Anwendungszwecke von Arrays.



## TIP

Diese Aufgaben werden euch im Projekt sicher was bringen!

## Array Zeichen

Verwenden Sie eine `for`-Schleife, um den folgenden Array auszugeben:

```
char[] text = {'h','a','l','l','o',' ','w','e','l','t'};
```

- Geben Sie danach den Array in **umgekehrter Reihenfolge** aus, also von hinten nach vorne.
- Verwenden Sie dafür auch wieder eine `for`-Schleife.

## String sortieren

1. Kopieren Sie den untenstehenden Code in ein Eclipse-Projekt und lassen Sie ihn laufen.
2. Erweitern Sie das Programm so, dass der Array sortiert und danach wieder ausgegeben wird.

```
public class SortString {

    public static void main (String[] args) {
        String beruf[] = {
            "Wurstmacher",
            "Holzbearbeiter",
            "Laufbursche",
            "Taxifahrer",
            "Velokurier",
            "Wasserfahrer",
            "Zitronenfalter",
            "Fensterputzer",
            "Halsabschneider",
        }
        //Ausgabe aller Werte des Arrays
        for (int i = 0; i < beruf.length; i++){
            System.out.println(i + ". " + beruf[i]);
        }
        // Erweitern Sie hier das Programm.
    }

}
```

## Array negativ

Gegeben ist Array **a**. Geben Sie alle Elemente mit einem negativen Wert untereinander auf der Konsole aus. Verwenden Sie auch wieder eine **for**-Schleife.

```
int[] a = { 1, -2, -25, 6, -3, 5 };
```

## Addiere Arrays

Gegeben sind drei Arrays **a**, **b** und **c**. Verändern Sie durch Java-Code den Array **c** so, dass im Element **c[0]** die Summe **a[0]** plus **b[0]** steht und entsprechend für die Elemente 1, 2, 3.

```
int[] a = { 1, 2, 25, 6 }; // { 1, 2, 25, 6 }
int[] b = { 9, 18, 5, 34 }; // + { 9, 18, 5, 34 }
int[] c = new int[4];      // = { 10, 20, 30, 40 }
```

## Kombiniere Arrays

Gegeben sind zwei Arrays **a** und **b**. Erzeugen Sie einen neuen Array **c**, der so lang ist wie **a** und **b** zusammengenommen und auch die Werte von **a** und **b** (in dieser Reihenfolge) enthält.

```
int[] a = { 1, 2, 25 };  
int[] b = { 9, 18 };
```

## Kombiniere Arrays mit Methode

Verwende den Code von [Kombiniere Arrays](#).

Nun versuche die Aufgabe so zu lösen, dass das Kopieren und Ausgeben in zwei Methoden gemacht wird. Die Signaturen der Methoden sind folgende:

- `public static void copyArray(int[] source, int[] destination, int destination_index)`
- `public static void showArray(String name, int[] array)`

## Noten

Erstellen Sie ein Programm, welches Noten einlesen kann und in einem Array speichert.

- Zuerst soll es nach der Anzahl Noten fragen.
  - ... und dann mit diesem Wert einen neuen Array anlegen.
- Dann sollen alle Werte vom Benutzer eingegeben werden, wobei.
  - ... nach jedem Wert der Durchschnitt der bisher eingegebenen Noten berechnet und angezeigt wird.



### TIP

Die Arbeit mit Methoden kann Ihr Programm übersichtlicher machen.

## Würfel

1. Erstellen Sie einen Array mit sechs 0en.
2. Würfeln Sie 100 Mal mit der Methode `rollDice()`.
  - Bei jedem Wurf zählen Sie für die gewürfelte Zahl der entsprechende Index im Array hoch.
3. Geben Sie aus wie häufig die jeweiligen Zahlen gewürfelt wurden.



### TIP

Diese Aufgabe ist besonders interessant für das Projekt



```
public class Dice {  
  
    public static void main(String[] args) {  
        // die Aufgabe hier implementieren  
    }  
  
    public static int rollDice() {  
        return (int)(Math.random() * (6 - 1 + 1)) + 1;  
    }  
}
```

# Basics

Hier werden nochmals in Kürze alle Konzepte vom Modul 319 aufgeführt

## Variablen

Variablen sind Platzhalter um Werte zu Speichern.

- Jede Variable reserviert einen Speicherplatz im Computerspeicher.
- Durch den eindeutigen Namen kann auf diesen Speicherplatz zugegriffen werden.

Um eine Variable zu definieren, muss ein **Typ**, einen **Namen** und einen **Wert** angegeben werden.

Variable syntax

```
type variableName = value;
```

## Primitive Datentypen

In Java gibt es eine beachtliche Anzahl an Datentypen gemäss folgender Tabelle. Vorerst nutzen wir die *primitiven* Datentypen. Diese sind erkennbar, indem der **Datentyp kleingeschrieben** ist.

Datentyp	Grösse	Beschreibung	Spezifika
<code>boolean</code>	1 Bit	speichert <code>true</code> ( <i>wahr</i> ) oder <code>false</code> ( <i>falsch</i> ) Werte	
<code>byte</code>	1 Byte	speichert ganze Zahlen von <code>-128</code> bis <code>127</code>	
<code>short</code>	2 Bytes	speichert ganze Zahlen von <code>-32'768</code> bis <code>32'767</code>	
<code>char</code>	2 Bytes	speichert ein einzelnes Zeichen oder <a href="#">ASCII</a>	<code>' '</code>
<code>int</code>	4 Bytes	speichert ganze Zahlen <code>-2'147'483'648</code> bis <code>2'147'483'647</code>	
<code>float</code>	4 Bytes	speichert Gleitkommazahlen von <code>6</code> bis <code>7</code> Dezimalstellen	<code>f</code>
<code>long</code>	8 Bytes	speichert ganze Zahlen von <code>-9'223'372'036'854'775'8081</code> bis <code>9'223'372'036'854'775'8071</code>	<code>l</code>
<code>double</code>	8 Bytes	speichert Gleitkommazahlen von <code>15</code> Dezimalstellen	<code>d</code>

## Initialisierung

Beispiele, wie Variablen initialisiert werden können. Die Leerzeichen dienen nur der Übersichtlichkeit.

### Deklaration von Variablen

```
// Datentyp    Variablenname    Semikolon
int           number          ;
char          sign            ;
...
```

Die Initialisierung der Werte verwendet spezifische Zeichen für die verschiedenen Datentypen. So endet ein `float`-Wert immer mit `f` oder `long` mit `l`. Die Spezifika pro Datentyp finden Sie in der Tabelle oberhalb unter "Spezifika".

### Deklaration & Initialisierung von Variablen

```
// Datentyp    Variablenname    Zuweiseoperator    Wert    Semikolon
int           number          =          5          ;
char          sign            =          'c'         ;
//           ^ ^ spezifisch für char sind (')
long          longNumber      =          123l        ;
//           ^ spezifisch für long (l)
...
```

## Der Datentyp `String`

Der Datentyp `String` **dient zur Speicherung von Zeichenfolgen**, also allgemeiner Text. Der Text muss immer zwischen zwei **"doppelten Anführungszeichen"** gestellt werden.

Datentyp	Grösse	Beschreibung	Spezifika
String	2 byte pro Zeichen	speichert beliebigen Text	<code>" "</code>

## Strings initialisieren

```
// Datentyp    Variablenname    Zuweiseoperator    Wert    Semikolon
String        zeichenFolge    =          "abc"    ;
//           ^ ^
//           "doppelten Anführungszeichen"
```

## Strings zusammensetzen

Mehrere Variablen vom Datentyp `String` können durch ein Plus-Zeichen `+` zusammengesetzt werden. Dabei sollte man darauf achten, dass der erste Text mit einem Leerzeichen enden soll. Wieso, sieht ihr im Beispiel:

Mit `+` Strings zusammensetzen

```
String name = "Mr Robot";
System.out.println("Your name is " + name);
//                      ^ Leerzeichen!
//                Your name is Mr Robot

System.out.println("Your name is" + name);
//                Your name isMr Robot
```

### HÖHERE DATENTYPEN SIND GROSSGESCHRIEBEN

- `String` ist grossgeschrieben, da es sich um einen *höheren* Datentyp handelt.
- Ein `String` baut auf dem *primitiven* Datentyp `char` auf (💡 *daher höher*)
  - Evtl. Hilft die Analogie von "Atome (primitiv)" zu "Moleküle (höher)".
- *Höhere* Datentypen sind auch Objekte.
  - Was Objekte genau sind, erfahrt ihr im Folgemodul 404 und ist noch nicht relevant!

## Strings mit Zahlen zusammensetzen

Strings können auch mit allen *primitiven* Datentypen, also auch mit numerischen Werten, durch das Plus-Zeichen `+` zu einer Zeichenfolge kombiniert werden. Der *primitive* Datentyp wird dadurch automatisch zu einem String!

String mit Zahlen kombinieren

```
System.out.println("Ihre Geschwindigkeit lautet " + 21);
//                Ihre Geschwindigkeit lautet 21
```

### EINE ZAHL IN EINEN STRING UMWANDELN

Werden Zahlen mit einem **leeren String** `""` konkateniert, wird die Zahl alleine in einen String umgewandelt. Dies ist ein gängiger Java "Hack".

Zahl in String umwandeln

```
String zahl = "" + 21;
// zahl ist nun "21"
```

# Operatoren

## Arithmetische `+`, `-`, `/`, `*`, `%`

Arithmetische Operatoren kennt Ihr bereits von der Mathematik. Damit lassen sich die gängigen mathematischen Operationen durchführen. Neu ist einzig der Rest Operator `%`. Dieser dividiert eine Zahl und gibt den Rest zurück.

`+` `-` `*` `%` Arithmetische Operatoren

```
int result;  
int number = 9;  
int anotherNumber = 3;  
  
result = number + anotherNumber; // Addition  
result = number - anotherNumber; // Subtraktion  
result = number / anotherNumber; // Division  
result = number * anotherNumber; // Multiplikation  
result = number % anotherNumber; // Rest der Division
```

### **GERADE/UNGERADE BERECHNEN MIT `%`**

Der Rest-Operator `%` gibt bei einer Division immer den Rest zurück. Wenn man nun eine Division durch 2 durchführt lässt sich herausfinden, ob eine Zahl gerade oder ungerade ist.

```
9 % 2 // ergibt 4 * 2 Rest 1 also ungerade  
10 % 2 // ergibt 5 * 2 Rest 0 also gerade  
  
public boolean even(int number) {  
    return number % 2 == 0;  
}
```

## Verkürzte arithmetische Operation mit sich selbst `+=`, `-=`, `*=`, `/=`

Oft möchte man den Wert einer Variablen direkt verändern. Das Resultat also nicht in eine neue Variable, sondern in sich selber speichern. Gegeben ist z.B. die Variable `number` vom Typ `int` mit dem Initialwert `3`.

```
int number = 3;
```

Möchte man dieser Variable `4` hinzuaddieren geht das folgendermassen:

```
number = number + 4; // Addition und Zuweisung zu sich selbst
```

Da dies sehr oft vorkommt, ist in allen gängigen Programmiersprachen dafür ein kombinierter Operator vorgesehen. Es wird dem Zuweisungsoperator, den arithmetischen Operator **vorangestellt**.

```
number += 4; // Verkürzte Addition und Zuweisung zu sich selbst
```

Aus `number = number + 4;` wird somit `number += 4;` und erspart uns die Dopplung der Variable. Dies geht natürlich auch für alle anderen arithmetischen Operatoren.

```
number -= 7; // Subtraktion und Zuweisung zu sich selbst  
number *= 9; // Multiplikation und Zuweisung zu sich selbst  
number /= 2; // Division und Zuweisung zu sich selbst
```

## Unäre (einstellige) Operatoren ++, --

Noch häufiger als die verkürzte arithmetische Operation mit sich selbst wird im Programmieren schrittweise hoch und runtergezählt **was auch Iteration genannt wird**.

Möchte man also von 0 bis 3 hochzählen geht dies so:

```
int number = 0;  
number += 1;  
number += 1;  
number += 1;
```

Der Unäre Operator `++` zählt der links vorangestellten Variable eines numerischen Typs eine 1 hinzu. Der obere Code ist somit identisch zu diesem:

```
int number = 0;  
number++;  
number++;  
number++;
```

### ❗ UNÄR → EINSTELLIG

Unär bedeutet einstellig, es braucht daher **nur der linke** und nicht auch einen rechten Teil beim Operator.

## 💡 ITERIEREN DURCH ARRAYS

Der unäre Operator `++` wird insbesondere beim **Iterieren durch Arrays** wie im folgenden Beispiel verwendet. Was gibt der obere Code wohl aus?

```
char[] text = {'h','a','l','l','o',' ','w','e','l','l','t'};

for (int i = 0; i < text.length; i++) {
    System.out.print(text[i]);
}
```

- Es wird Schrittweise jede Stelle vom Array `char[] text` in einem `for`-Loop ausgegeben.
- Die Variable `i`, Iterator, wird durch `i++`, für jeden Schritt +1 hochgezählt
- `i++` könnte auch mit `i += 1` oder `i = i + 1` ersetzt werden.
- `i++` ist jedoch viel kürzer.

## 📄 FUNFACT

C++ erweitert die Programmiersprache C. Um diese Verwandtschaft ein bisschen "nerdisch" auszudrücken wurde das Wortspiel C++ gewählt. C++ ist eine weitere Iteration von C.

## Vergleichsoperatoren `==`, `!=`, `!`

Vergleichsoperatoren ergeben immer einen **boolean** (`true`/`false`). Sie werden in Kontrollstrukturen als Bedingungen verwendet.

`==` Gleichheit

```
true == true; // true
1    == 1    // true

false == true; // false
1     == 2    // false
```

`!=` Ungleichheit

```
false != true; // true
1      != 2    // true

true  != true; // false
1     != 1     // false
```

`!` Negation

```
!false // true
!(1 == 2) // true

!true // false
!(1 == 1) // false
```

## Vergleichen von Strings `equals`

Strings und andere höhere Datentypen, können **nicht durch Vergleichsoperatoren verglichen werden**.

- Höhere Datentypen sind Datentypen die durch **Klassen** definiert werden.
- Diese sind **immer grossgeschrieben**.
- Höhere Datentypen besitzen die **Methode** `equals` (was auf Deutsch "gleicht" heisst)

```
String text = new String("Hallo Welt");  
String text2 = new String("Hallo Welt");  
  
text == text2;    // false! Die Speicherorte der Objekte `text` und `text2` sind  
                  unterschiedlich  
text.equals(text2) // true! Der Inhalt von beiden Objekte ist jedoch gleich.
```

## Wieso ist dies nun so?

### Technisch:

1. Durch `String text = new String("Hallo Welt");` wird ein **Objekt** erzeugt. Dieses besitzt einen **eigenen Speicherbereich** in der Variable `text`.
2. Durch `String text2 = new String("Hallo Welt");` wird ebenfalls ein **neues Objekt** erzeugt. Dieses besitzt wiederum einen **eigenen Speicherbereich** in der Variable `text2`:
3. Werden jetzt die zwei Objekte durch `==` verglichen, wird nicht deren Inhalt, sondern deren **Speicherort verglichen!**
4. Mit `equals` wird jedoch **aber der Inhalt verglichen**. Zwei Pakete mit dem gleichen Inhalt sind zwar eigenständig, aber sie sind sich gleich.

### Verständlich:

1. Stellen wir uns **ein Paket**, Paket1, vor, in welches die Druckletter `H,a,l,l,o, ,W,e,l` und `t` gelegt werden
2. Stellen wir uns **ein zweites Paket**, Paket2, vor, in welches **ebenfalls** die Druckletter `H,a,l,l,o, ,W,e,l` und `t` gelegt werden
3. Zwei Pakete mit gleichem Inhalt sind auch in der uns bekannten Welt immer noch zwei unterschiedliche Pakete, oder?
  - Mit `==` werden die Pakete und nicht deren Inhalt "Hallo Welt" verglichen.
4. Vergleicht man jedoch der Inhalt befinden sich in beiden Paketen die gleichen Druckletter in der gleichen Reihenfolge.
  - Mit `equals` wird der Inhalt und nicht die Pakete verglichen.

## Und wieso darf man nun primitiven Datentypen mit == vergleichen?

Primitive Datentypen sind in Java **keine Objekte**, sie werden somit nicht in ein Paket verpackt.

- Die Nummer `4` ist die Nummer `4`. Punkt.
- Das Zeichen `c` ist das Zeichen `c`. Punkt.
- Aber der Satz "Hallo Welt" könnte man
  - alphabetisch sortieren
  - grossschreiben
  - kleinschreiben
  - ...



### ❗ STRING IST EINE KLASSE UND SOMIT EIN OBJEKT

Eigentlich müsste man einen `String` wie folgt initialisieren:

```
String text = new String("String ist eine Klasse und kann somit auch mit new  
initialisiert werden");
```

Da jedoch so oft Strings verwendet werden und die Klasse somit fest in die Sprache verankert ist, **wurde ihr durch den Compiler ein paar Sonderheiten verliehen**. So kann das `new String()` weggelassen werden.

```
String text = "String ist eine Klasse und kann somit auch mit new initialisiert  
werden";
```

Beim Kompilieren wird das `new String()` automatisch hinzugefügt!

### ❗ COMPILER SIND SCHLAU!

`"Hallo Welt" == "Hallo Welt";` ist `true`. Dies ist eine Ausnahme, da der Compiler schlau ist und merkt, dass hier nicht zwei Objekte benötigt werden. Der Compiler versucht Speicher zu sparen. Sobald aber zwei Strings dynamisch erstellt werden (`new`) geht dies nicht mehr!

Nach dem compilieren sieht das in etwa so aus:

```
String text = new String("Hallo Welt");  
text == text; // true da gleiches Objekt und Speicherort!
```

### ❗ NICHT ALLE PROGRAMMIERSPRACHEN SIND GLEICH

In Ruby ist z.B. alles ein Objekt. Auch Zahlen. Da gehen lustige Dinge wie:

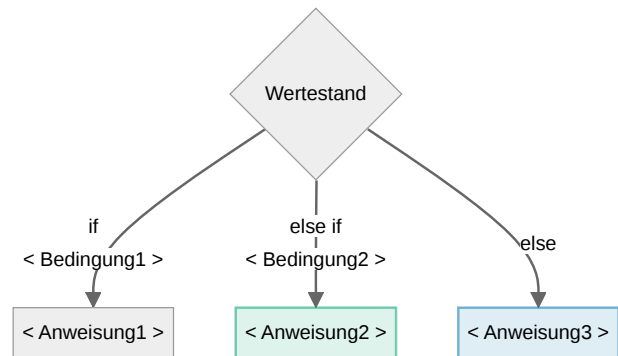
- `print 3.month.from_now` gibt das Datum von in drei Monaten zurück
- oder `3.times { print "Hallo Welt" }` gibt 3 Mal "Hallo Welt" aus

# Kontrollstrukturen

## if-Kontrollstruktur

### Pseudo-Code

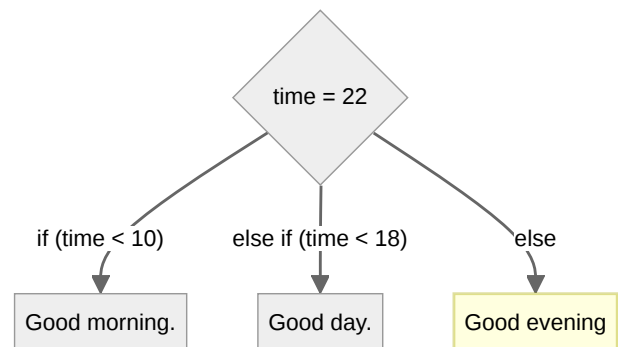
```
if (<Bedingung1>) {  
    // Codeblock bei wahrer bedingung1  
    <Anweisung1>  
}  
else if (<Bedingung2>) {  
    // Codeblock bei  
    // - unwahrer bedingung1  
    // - aber wahrer bedingung2  
    <Anweisung2>  
}  
else {  
    // Codeblock bei  
    // - unwahrer bedingung1  
    // - unwahrer bedingung2  
    <Anweisung3>  
}
```



## if-Beispiel

### if Beispiel

```
int time = 22;  
if (time < 10) {  
    System.out.println("Good morning.");  
} else if (time < 18) {  
    System.out.println("Good day.");  
} else {  
    System.out.println("Good evening.");  
}  
// Outputs "Good evening." (else).
```



- Es wird der Code-Block von `else` ausgeführt, da die Variable `time` sowohl grösser als 10, als auch grösser als 18 ist.

## switch

### Pseudo-Code

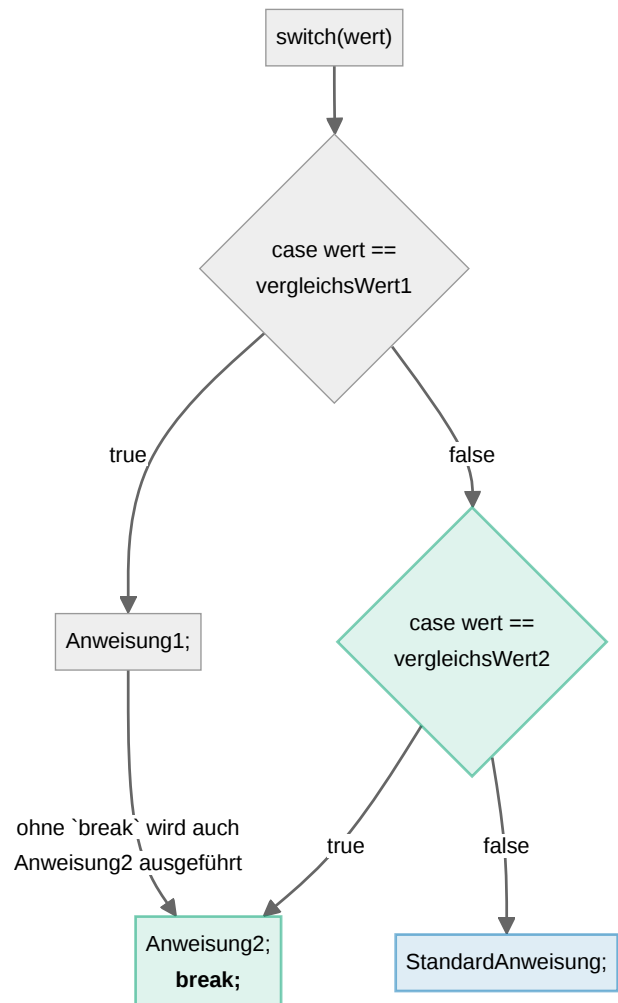
```
switch(wert) {  
  case vergleichsWert1:  
    <Anweisung1>;  
    // ohne `break` wird auch  
    <Anweisung2>  
    // bis zum `break` ausgeführt.  
  case vergleichsWert2:  
    <Anweisung2>;  
    break;  
  default:  
    <StandardAnweisung>;  
}
```

### ❗ WERTEVERGLEICH IMMER MIT `==`, `equals`

Der `wert` wird bei einer `switch-case`-Kontrollstruktur pro `case` mit dem Vergleichswert verglichen. Dabei gilt immer `==` resp. für höhere Datentypen `equals`

### ❗ `break`

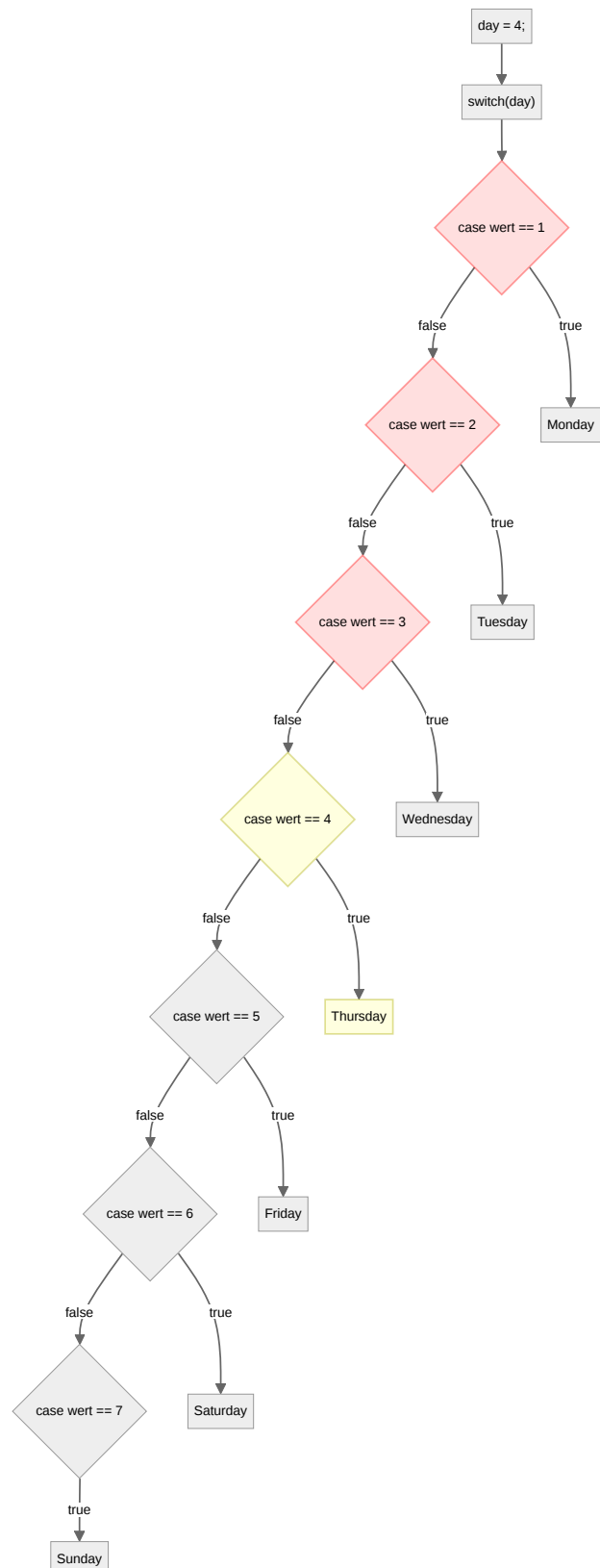
Wird eine Anweisung eines `case` nicht mit `break` beendet, wird ebenfalls die Anweisung des nächsten `case` ausgeführt. Solange bis ein `break` erscheint. Dies gilt auch für die Standardanweisung (`default`)!



## Switch-Beispiel

switch Beispiel

```
int day = 4;  
switch (day) {  
    case 1:  
        System.out.println("Monday");  
        break;  
    case 2:  
        System.out.println("Tuesday");  
        break;  
    case 3:  
        System.out.println("Wednesday");  
        break;  
    case 4:  
        System.out.println("Thursday");  
        break;  
    case 5:  
        System.out.println("Friday");  
        break;  
    case 6:  
        System.out.println("Saturday");  
        break;  
    case 7:  
        System.out.println("Sunday");  
        break;  
}  
// Outputs "Thursday" (day 4)
```



### ! **break** WIRD EIGENTLICH IMMER VERWENDET

Es gibt fast keine realen Szenarien, in denen es Sinn macht mehrere **case** Fälle auszuführen. Deswegen sind **switch-case** Statements nicht allzu oft anzutreffen.

Eine **switch-case**-Kontrollstruktur die für jede Anweisung ein **break** verwendet, kann immer mit einer **if-else**-Kontrollstruktur mit **==** Bedingungen ersetzt werden. Damit lassen sich die **break** Statements sparen.

switch-case als if-else

```
if (wert == 1) {  
    <Anweisung1>  
} else if (wert == 2) {  
    <Anweisung2>  
} else {  
    <StandardAnweisung>  
}
```

### 💡 **ARRAYS SIND TOLL!**

📁 Mit einem **Array** kann teilweise sogar eine Kontrollstruktur vermieden werden.

```
String[] days = {"Monday",  
    "Tuesday",  
    "Wednesday", "Thursday",  
    "Friday",  
    "Saturday", "Sunday"};  
int day = 4;  
System.out.println(days[day-1]); //  
Arrays starten bei 0! deswegen day-1  
// Outputs "Thursday"
```

## Methoden

## 1. Account Applikation

Sicherzustellen, dass alle die Entwicklungsumgebung zum Laufen haben und auch

## 2. Analyse Account Applikation

| Material | Abbildung unterhalb |

## 3. Refactoring Account Applikation

Machen Sie sich mit dem Konzept der

## 4. Account - Klassendiagramm

Machen Sie sich mit den UML Klassendiagrammen

## 5. Starterklasse

- Es ist eine gute Praxis in der main Methode nicht viel Logik zu

## 6. Fahrenheit-Celsius Konverter

Machen Sie sich mit dem Konzept der Static


## 7. Formatierung

Machen Sie sich mit dem Konzept der

# 1. Account Applikation

## ❗ LIVE CODING!

Sicherzustellen, dass alle die Entwicklungsumgebung zum Laufen haben und auch wieder in Schwung gekommen ist, werden wir diese Aufgabe zusammen lösen!

<b>Material</b>	<b>Wissen und Faktenblatt Modul 403</b>
<b>Richtzeit</b>	ca. 45 Minuten
<b>Sozialform</b>	 <b>Live Coding</b> mit Lehrer!

## Auftrag

Erstellen Sie anhand des Wissens und Könnens aus dem Modul 403 ein Programm, welches einen einfachen Dialog für ein Bankkonto realisiert. Es soll möglich sein Geld:

- **einzuzahlen**
- **abzuheben**
- den Kontostand **anzuzeigen**

## ❗ WICHTIG!

Die Operationen zum `einzuzahlen` und `abzuheben` sollen jeweils **durch eine Methode realisiert** sein.

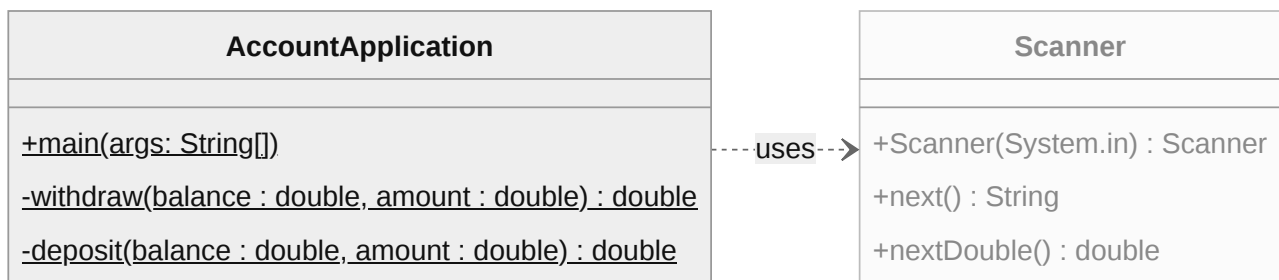
**Die Interaktion mit dem Benutzer soll so aussehen, wie nachfolgend gezeigt.**

### AccountApplication Beispiel Terminal Ausgabe

```
Welcome to the account application
Please enter the amount, 0 (zero) to terminate
10
To deposit, press +, to withdraw, press -
+
Please enter the amount, 0 (zero) to terminate
30
To deposit, press +, to withdraw, press -
+
Please enter the amount, 0 (zero) to terminate
5
To deposit, press +, to withdraw, press -
-
Please enter the amount, 0 (zero) to terminate
0
Final balance: 35.0
```

## UML der Musterlösung

Verwenden Sie das UML um die Struktur der Aufgabe zu verstehen.



### ! INFO

Das UML von verwendeten Java Standard Klassen, welche Sie also nicht selber implementieren müssen, sind ausgebleicht dargestellt. Hier also der `Scanner`. Es werden immer nur die Methoden angegeben, welche verwendet werden, auch wenn die eigentliche Klasse mehr Methoden besitzt.

## Erste Hilfe

## Zusatzaufgaben für Schnelle

- Ermöglichen Sie zusätzlich zu `+` und `-` die Menüauswahl `=` um jederzeit den Kontostand abzufragen.
- Geben Sie am Schluss vor dem Schlusssaldo eine Auflistung aller Transaktionen (Ein- und Auszahlungen) aus.



# Musterlösung

## 2. Analyse Account Applikation

Material	Abbildung unterhalb
Richtzeit	ca. 15 Minuten
Sozialform	Zweier Teams

### Auftrag

Sie haben nun ein kleines Programm geschrieben, das vom Benutzer Eingaben über Ein- und Auszahlungen einliest und damit einen Kontostand verwaltet. Wie vom Modul 403 her gewohnt, haben wir alles in einer Klasse erledigt. Wir wollen jetzt **analysieren, was diese Klasse alles für Aufgaben übernimmt**. Dazu ist der Code der **Musterlösung** mit Nummern versehen worden

1. Überlegen Sie **zu zweit**, was die jeweilige nummerierte Zeile genau macht
2. **Notieren Sie sich die Antworten in elektronischer Art oder auf einem Blatt Papier**

## Analyse Account Applikation

# Lösung

# 3. Refactoring Account Applikation

## WICHTIG

 **Machen Sie sich mit dem Konzept der Fachklassen bekannt bevor Sie weiterfahren!**

## REFACTORING

Bezeichnung im Programmieren, dass man den **vorhandenen Code neu strukturiert, ohne neue Funktionalität hinzuzufügen**. Refactoring dient dazu, dass die Applikation/Software auf lange Zeit besser wartbar und erweiterbar ist.

## Ausgangslage

1. Das Konto-Programm bearbeitet zu viele Aufgaben (Verantwortlichkeiten) in einer Klasse.
  2. Die Arbeit wollen wir **in zwei Klassen aufteilen**
- `AccountApplication` (Beinhaltet die Benutzerinteraktion und `main` Methode)
  - `Account` resp. `Konto` (Beinhaltet die Fachlogik)

## Einführung der Klasse `Account/Konto`

Die **Fachlogik** der `AccountApplication` kann in eine eigene Klasse `Account` ausgelagert werden.

Account.java

```
public class Account {  
    private double balance;           // englisch für "kontostand"  
  
    public void deposit(double value) { // englisch für "einzahlen"  
        balance += value;  
    }  
  
    public void withdraw(double value) { // englisch für "auszahlen"  
        balance -= value;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
}
```

### ! INFO

Dazu muss man folgendes beachten: Die Variable `private double balance;` ist nicht mehr in einer Methode deklariert, sondern gleich zu Beginn, **vor** den einzelnen Methoden (im *Klassen-Body*). Dadurch ist die Variable **überall in der Klasse sichtbar**.

Dies hat nun auch den Vorteil, dass wir daraus viele Konto-Objekte erstellen können, die komplett eigenständig einen Kontostand verwalten können. Somit wird ermöglicht, theoretisch mehrere Konten anzulegen.

Beispiel: Mehrere Objekte der Klasse 'Account'

```
Account sparkonto = new Account(); // neues `Account` Objekt gespeichert in der
Variable `sparkonto`
Account girokonto = new Account(); // neues `Account` Objekt gespeichert in der
Variable `girokonto`

sparkonto.deposit(10); // dem Sparkonto 10 Franken einzahlen
sparkonto.deposit(20); // dem Sparkonto 20 Franken einzahlen

girokonto.withdraw(20); // dem Girokonto 20 Franken abheben

System.out.println(sparkonto.getBalance()); // => 30;
System.out.println(girokonto.getBalance()); // => -20;
```

### 💡 OBJEKTE HABEN EIGENEN SPEICHERBEREICH!

- Das Objekt `sparkonto` und `girokonto` **teilen sich den Code** der Klasse `Account`.
- Die **Werte der Instanz-Variable** `private double balance;` sind jedoch **unabhängig**!

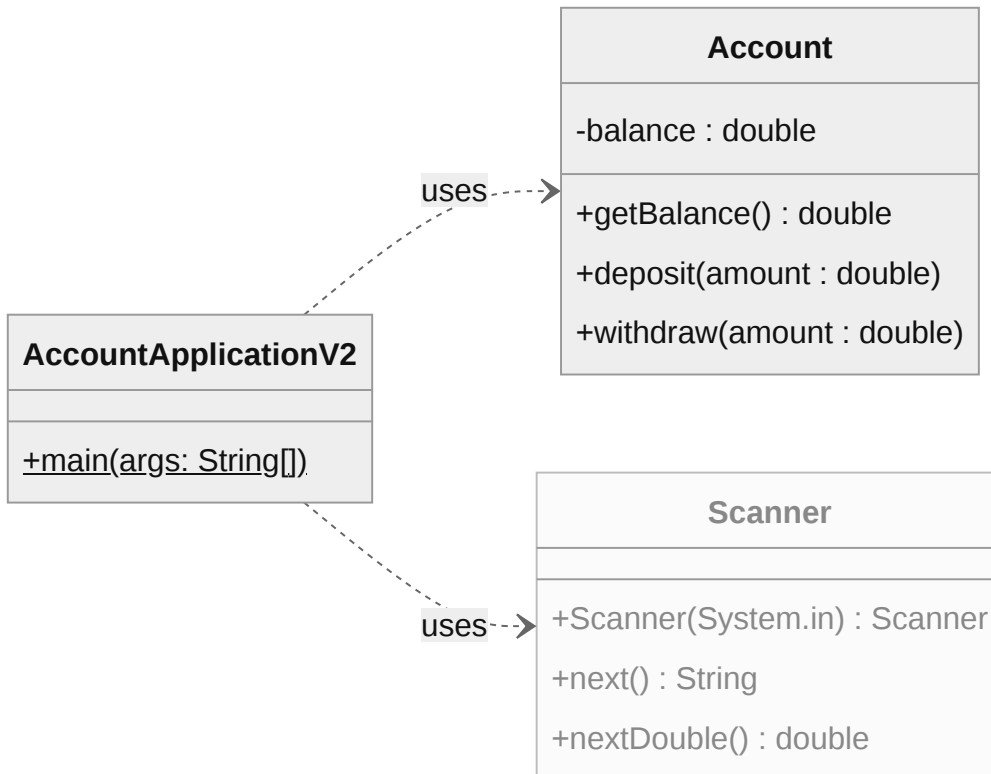
## Aufgabe

Bauen Sie Ihr Programm jetzt so um, dass es aus zwei Klassen besteht (die ursprüngliche Klasse und die Klasse `Account`).

- Kopieren Sie die Klasse `AccountApplication` und benennen Sie die neue `AccountApplicationV2`.
- Erstellen und implementieren Sie die Klasse `Account`.
- Löschen Sie in der neuen Klasse (`AccountApplicationV2`) die Variable `double balance;`
- Legen Sie dafür ein **Objekt** der Klasse `Account` an.
- Jetzt erscheinen Fehler im Quellcode.
  - Überall dort müssen Sie das Programm anpassen und mit dem **Objekt** der Klasse `Account` arbeiten.
- Die Methoden `deposit` und `withdraw` müssen nun in der Klasse `AccountApplicationV2` gelöscht werden.

## UML der Musterlösung

Verwenden Sie das UML um die Struktur der Aufgabe zu verstehen.



### 💡 TIP

Für jeden **gepunkteten Pfeil der mit `uses` beschriftet** ist, muss in der Klasse, von der der Pfeil abgeht (hier **AccountApplicationV2**) **ein Objekt der Klasse auf die gezeigt wird** vorhanden sein (hier **Account** und **Scanner**).

In der Klasse **AccountApplicationV2** muss somit irgendwo `new Account()` und `new Scanner(System.in)` stehen! Da die Klasse **AccountApplicationV2** nur die `main`-Methode beinhaltet, wird es wohl darin sein.

📝 Theoretisch könnte das Objekt auch über einen Parameter der Klasse übergeben werden. Das nennt sich *Dependency-Injection*.

## Musterlösung

# 4. Account - Klassendiagramm

## WICHTIG

 **Machen Sie sich mit den UML Klassendiagrammen bekannt bevor Sie weiterfahren!**

## Aufgabe

1. Skizzieren Sie das UML Klassendiagramm der folgenden Klasse `Account`

```
public class Account {  
    private double balance = 0;  
  
    public double getBalance() {  
        return balance;  
    }  
    public void deposit(double amount) {  
        balance = balance + amount;  
    }  
    public void withdraw(double amount) {  
        balance = balance - amount;  
    }  
}
```

# 5. Starterklasse

- Es ist eine gute Praxis in der `main` Methode nicht viel Logik zu implementieren.
- Bestenfalls besteht die `main` Methode nur aus der **Instanziierung einer Applikationsklasse** welche die eigentliche App verwaltet.

Starter.java als Beispiel

```
public class Starter {  
  
    public static void main(String[] args) { // Startpunkt des Programms, ist immer  
        static!  
        MyNewShinyApp app = new MyNewShinyApp(); // `new` ist innerhalb von `static`  
        erlaubt  
        app.start(); // starten der eigentlichen App  
    }  
  
}
```

## Aufgabe

### 1. Umbau der `AccountApplicationV2` Klasse

- Kopieren Sie die Klasse und benennen Sie dies neue `AccountApplicationV3`
- Ändern Sie die Definition der Methode `public static void main(String[] args)` um in `public void start()`

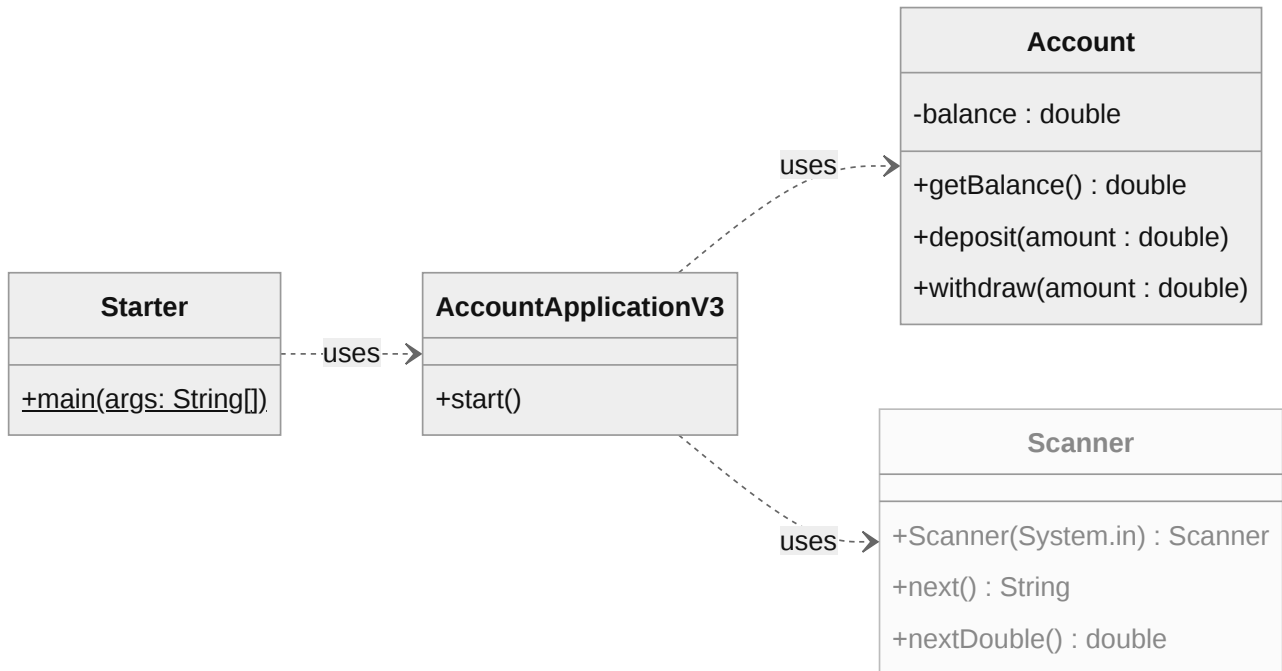
### 2. Erstellen der Starterklasse

- Erstellen Sie eine neue Klasse `Starter`.
- Diese Klasse muss die Methode `public static void main(String[] args)` besitzen.
- Erstellen Sie in der `main` Methode ein **Objekt** Ihrer Applikationsklasse (`AccountApplicationV3`)
- Rufen Sie dann die Methode `start()` des Objekts auf

## UML der Musterlösung

Verwenden Sie das UML um die Struktur der Aufgabe zu verstehen.





## Musterlösung

# 6. Fahrenheit-Celsius Konverter

## WICHTIG

 **Machen Sie sich mit dem Konzept der Static bekannt bevor Sie weiterfahren!**

## Ausgangslage

Gegeben sind folgende Klassen welche diesen Dialog ausgeben:

### Der Dialog-Ablauf

```
Möchten Sie (1) Fahrenheit nach Celsius oder (2) Celsius nach Fahrenheit umrechnen?  
Bitte Geben Sie Ihre Wahl, 1 oder 2, ein: 1  
Bitte geben Sie die Temperatur ein: 32  
Die umgerechnete Temperatur beträgt: 0.0  
Möchten Sie noch eine Temperatur umrechnen? Dann 1 eingeben
```

### Starter.java

```
package converter;  
public class Starter {  
    public static void main(String[] args) {  
        Converter ui = new Converter();  
        ui.dialog();  
    }  
}
```

### Converter.java

```
package converter;
import java.util.Scanner;

public class Converter {
    public void dialog() {
        double convtemp;
        // Geht es ohne `new`, rsp. ohne Objekt `converter`?
        DegreesConverter converter = new DegreesConverter();
        int userEntry = 0;
        try(Scanner scanner = new Scanner(System.in)) {
            do {
                System.out.println("Möchten Sie (1) Fahrenheit nach Celsius oder (2) Celsius
nach Fahrenheit umrechnen?");
                System.out.print("Bitte geben Sie Ihre Wahl 1 oder 2 ein: ");
                userEntry = scanner.nextInt();
                System.out.print("Bitte geben Sie die Temperatur ein: ");
                double temp = scanner.nextDouble();
                if (userEntry == 1) {
                    // Was muss hier geändert werden wenn es kein Objekt `converter` mehr gibt?
                    convtemp = converter.toCelsius(temp);
                } else {
                    // Was muss hier geändert werden wenn es kein Objekt `converter` mehr gibt?
                    convtemp = converter.toFahrenheit(temp);
                }
                System.out.println("Die umgerechnete Temperatur beträgt: " + convtemp);
                System.out.print("Möchten Sie noch eine Temperatur umrechnen? Dann 1 eingeben.
");
                userEntry = scanner.nextInt();
            } while (userEntry == 1);
        }
    }
}
```

### DegreesConverter.java

```
package converter;
public class DegreesConverter {

    public double toFahrenheit(double celsius) { // was muss hier hinzugefügt werden?
        return (celsius * 9.0/5.0) + 32.0;
    }

    public double toCelsius(double fahrenheit) { // was muss hier hinzugefügt werden?
        return (fahrenheit - 32.0) * 5.0/9.0;
    }
}
```

# Aufgabe

1. Kopieren Sie den Code der Klassen `Starter`, `Converter` und `DegreesConverter` von oben
2. Bringen Sie die Applikation in Eclipse zum Laufen.
3. Ändern Sie die Methoden `toFahrenheit` und `toCelsius` sodass, die Klasse `DegreesConverter` **statisch, also ohne** `new` verwendet werden kann.

# Musterlösung

# 7. Formatierung

🔔 Machen Sie sich mit dem Konzept der **Formatierung** bekannt bevor Sie weiterfahren!

## Aufgabe

Gehen Sie Ihre bisherigen Programme durch und prüfen Sie, ob Sie diese Formatierung jeweils eingehalten haben.

- Wenn nicht, passen Sie in **einer** der alten Aufgaben den Code an
- Achten Sie ebenfalls darauf, dass Sie **möglichst gute Namen** für die Klassen, Variablen und Methoden einsetzen.
- "Gut" heisst, dass die **Namen möglichst "sprechend"** sind, also **Auskunft über die Funktion** einer Methode oder den Inhalt einer Variablen geben.



**TIP**



Je besser die Namen, desto weniger Kommentare sind nötig!

## 1. Einstieg in Swing

Eine Bibliothek für grafische Oberflächen

## 2. Fenster (JFrame)

Das folgende Beispiel zeigt die Vorkehrungen, die notwendig sind, um ein Fenster

## 3. JFrame Komponenten

Neben JFrame werden hier drei für unser Modul unverzichtbare Komponenten

## 4. JButton Aktivieren

Machen Sie sich mit dem Konzept des

## 5. Strings in Zahlen umwandeln

Das nachfolgende Programm zeigt ein Swing-Programm, bei dem eine Zahl eingelesen

## 6. Account Applikation

In der ersten Woche haben wir eine Konsolenapplikation implementiert, wobei man

## 7. Konstruktor

Machen Sie sich mit dem Konzept der

## **8. Easy Dice Game**

- Machen Sie sich mit den

## **9. JPanel**

Für diejenigen, die mit JPanels arbeiten wollen, dient folgendes

## **10. PlayerPanel**

Hier gibt es nun noch ein JPanel Beispiel welches es durch folgende Methoden

## **Zusatzaufgaben**

Versucht alle Aufgaben mit einer Starter, Gui und Fachklasse zu

# 1. Einstieg in Swing

Eine Bibliothek für grafische Oberflächen

Die bisherigen Programme im **Modul 403** sowie die ersten Aufgaben im **Modul 404** waren zwar in der Lage mit dem Benutzer zu kommunizieren oder Ausgaben vorzunehmen, aber all dies erfolgte lediglich **über die Konsole**.

Benutzer sind sich gewohnt, dass Programme entweder in Fenstern oder im Browser ablaufen. Java bietet mit **Swing** eine sogenannte Bibliothek (eine Gruppe von Klassen) an, um Programmfenster mit Eingabefeldern und Befehlsschaltflächen zu erstellen.



## **MACHT MÖGLICHST VIELE DER SWING-AUFGABEN! JE BESSER IHR DIESE**

beherrscht, desto einfacher fällt euch das Projekt!

- **Ihr werdet nur besser beim Machen!** 🧐.
- Programmieren kann man nicht lesend lernen 🤖.
- Lest trotzdem alle Seiten vorsichtig und dann MACHT die Aufgaben 😊.
- Kopieren der Aufgaben ist nicht klug, es bringt euch nix! 💡.
  - Tippt die Aufgaben ab, wenn ihr nicht weiter kommt! :technologist: :::



## 2. Fenster (JFrame)

Das folgende Beispiel zeigt die Vorkehrungen, die notwendig sind, um ein Fenster anzuzeigen:

### Starter.java

In der `main`-Methode der Klasse `Starter` wird ein Objekt der Klasse `PureWindow` erzeugt und in der Variable `pureWindow` gespeichert. Die Variable `pureWindow` wird dann verwendet, um die Methode `showDialog()` aufzurufen.

Starter.java

```
public class Starter {  
  
    public static void main(String[] args) {  
        PureWindow pureWindow = new PureWindow(); // Erstellt ein `PureWindow` Objekt und  
        speichert es in der Variable `pureWindow`  
        pureWindow.showDialog(); // Führt die Methode `showDialog()` aus  
    }  
  
}
```

#### TIP

- `PureWindow` ist die Klasse sowie der Datentyp
- `pureWindow` (klein) ist die Variable, die das Objekt beinhaltet.

### PureWindow.java

- Die Klasse `PureWindow` muss von der Klasse `JFrame` alle Fähigkeiten übernehmen. Dies geschieht durch die Anweisung `extends JFrame`. Damit wird die Klasse `PureWindow` zu einem `JFrame`.
- Die Methode `showDialog()` führt die **grundlegenden Konfigurationsschritte** aus. Diese werden bei allen Fenstern benötigt.
- Die Methode `showDialog()` muss über das Objekt `pureWindow` aufgerufen werden.  
`pureWindow.showDialog()`

#### PureWindow.java

```
import javax.swing.JFrame;

public class PureWindow extends JFrame {

    public void showDialog() { // Beliebiger Name, kann auch, "start" oder nur "show"
                              // heissen.
        setLayout(null); // Deaktiviert Layout-Automatismen von Swing
        setDefaultCloseOperation(EXIT_ON_CLOSE); // Beendet beim Schliessen des Fensters
                              // ebenfalls den Prozess
        setSize(300, 300); // Bestimmt die grösse des Fensters
        setTitle("Mein toller Titel"); // Setzt den Titel des Fensters
        setVisible(true); // Muss am Ende stehen! Ohne das wird nichts angezeigt!
    }

}
```

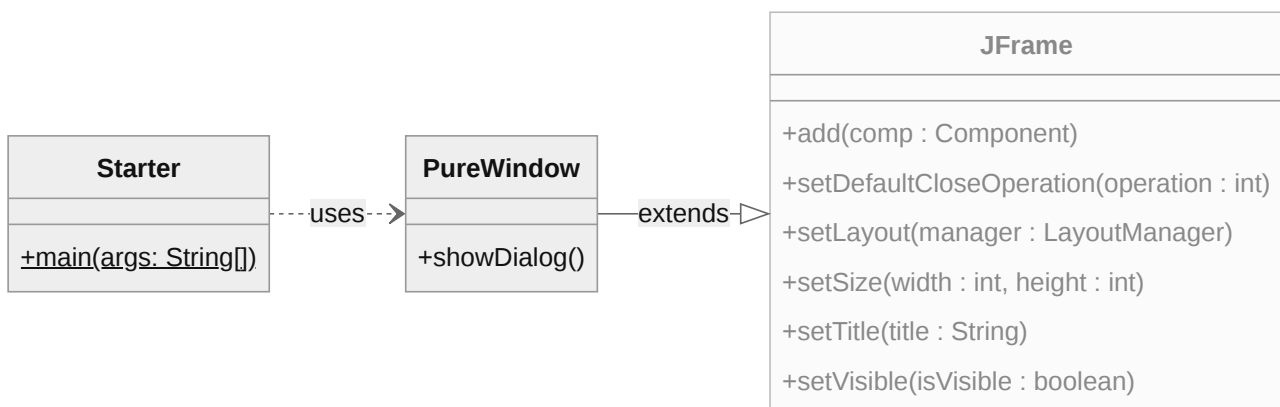
#### **i** NOTE

Die Klasse haben wir `PureWindow` genannt. Deutsch heisst das soviel wie "reines Fenster". Die Klasse kann beliebig benannt werden. `PureWindow` eignet sich, da es sich um ein Beispiel für das Grundgerüst handelt, ohne weiteren Inhalt.

#### **💡** TIP

Lesen Sie auch die Kommentare im Code, löschen Sie die einzelnen Zeilen und schauen was passiert!

## UML des ganzen Programms



### ! EXTENDS

`extends` bedeutet, dass die Klasse von der, der solide Pfeil mit **Dreieckspitze** (UML oben) ausgeht alle Methoden und Instanzvariablen der Klasse, auf welche der Pfeil zeigt, übernimmt/erbt/sich erweitert.

- Die Klasse `PureWindow` erbt somit alle Methoden und Instanzvariablen der Klasse `JFrame`.
- Die Signatur der Klasse `PureWindow` lautet `class PureWindow extends JFrame`

## Aufgabe - Programm starten

- Erstellt ein neues Java Project (z.B. `SwingPureWindow`):
- Kopiert den Code von Oben (`Starter.java`, `PureWindow.java`).
- Startet das Programm und genießt den Blick auf ein Fenster ähnlich dem Bild.
- Löscht einzelne Zeilen und analysiert den Effekt.



### 💡 ANSTATT COPY/PASTE DEN TEXT SELBER ABTIPPEN!

1. Lernt Ihr so besser
2. Merkt Ihr, dass der Editor Vorschläge macht!
3. Erhält Ihr ein besseres "Gefühl" wie es ist zu programmieren 🧐

# Cheat Sheet

```
setLayout(null); // Standard Layout deaktivieren
setDefaultCloseOperation(EXIT_ON_CLOSE); // Beim schliessen des Fensters, das ganze
Programm beenden
setSize(300, 300); // Grösse vom Fenster festlegen
setTitle("Ich bin der Fenster Titel"); // Titel des Fensters festlegen
setVisible(true); // Fenster sichtbar machen

JLabel label = new JLabel("Beschriftung"); // Ein Label
label.setBounds(x, y, width, height); // Bestimmen wo sich das Label befindet
add(label); // Label hinzufügen

JTextField textfield = new JTextField(); // Ein Textfeld
textfield.setBounds(x, y, width, height); // Bestimmen wo sich das Textfeld befindet
add(textfield); // Textfeld hinzufügen

JButton button = new JButton("press me"); // Ein Button
button.setBounds(x, y, width, height); // Bestimmen wo sich der Button befindet
add(button); // Textfeld hinzufügen
```

## UML

JFrame
<ul style="list-style-type: none"><li>+add(comp : Component)</li><li>+setDefaultCloseOperation(operation : int)</li><li>+setLayout(manager : LayoutManager)</li><li>+setSize(width : int, height : int)</li><li>+setTitle(title : String)</li><li>+setVisible(isVisible : boolean)</li></ul>

## 3. JFrame Komponenten

Neben JFrame werden hier drei für unser Modul unverzichtbare Komponenten vorgestellt. Der dazu nötige Quelltext wird zwischen die beiden Blöcke der Konfiguration des JFrame geschrieben.

### JLabel

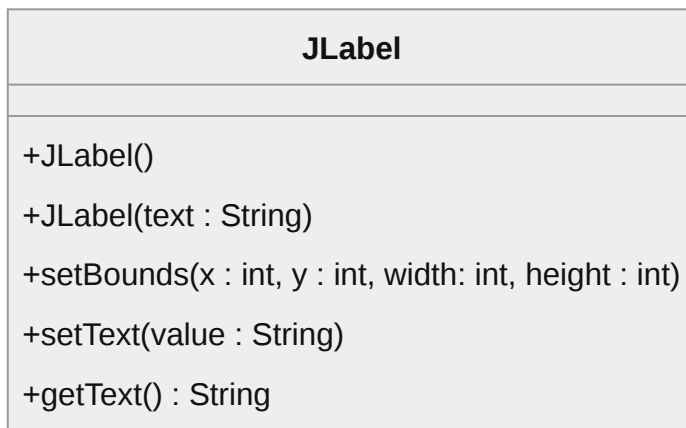
Eine Komponente zur Beschriftung

```
JLabel label = new JLabel("Beschriftung");

// Wichtige Methoden
label.setBounds(x, y, width, height); // setzt die Koordinaten der Komponente auf dem
Fenster
label.setText("Eine neue Beschriftung"); // Setzt einen neuen Text
String text = label.getText(); // gibt den Text des Labels zurück
```

- Der Parameter dient zur Initialisierung eines Textes

### UML



### Wann wird das JLabel eingesetzt?

- Zur **Beschriftung** von Textfelder
- Zur Ausgabe von berechnetem Text
  - z.B. Bei einer Umwandlung von Einheiten
- Labels sind vom GUI schreibgeschützt

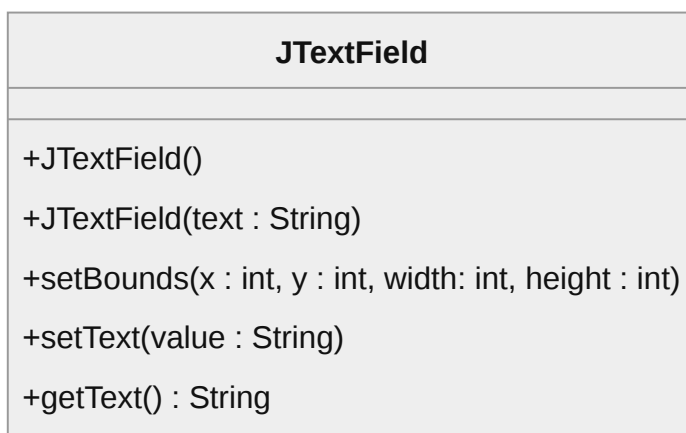


## JTextField

Eine Komponente zur Texteingabe (Alternative zu Scanner)

```
JTextField textfield = new JTextField();  
  
// Wichtige Methoden  
textfield.setBounds(x, y, width, height); // setzt die Koordinaten der Komponente auf  
dem Fenster  
String text = textfield.getText(); // gibt den Text des Textfeldes zurück  
textfield.setText("setzt ein Text"); // Setzt einen neuen Text
```

## UML



## Wann wird das JTextField eingesetzt?

- Zur **Eingabe von Text** durch den Benutzer (z.B. Spielernamen Eingabe)
- Zur Ausgabe von Text (Hierzu eignen sich JLabel meistens besser!)



### TIP

Der **Scanner** wird in einer GUI Applikation **nicht mehr verwendet**. Dafür existiert nun das **JTextField**.

## JButton

Eine Komponente zur Benutzerinteraktion

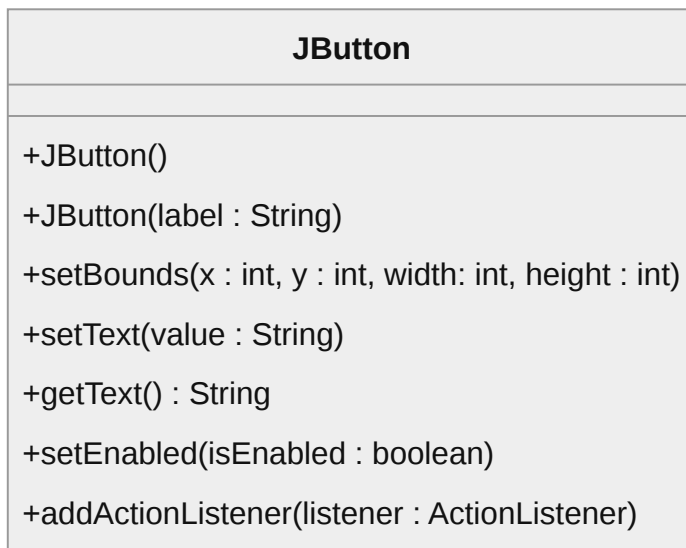
```
 JButton button = new JButton("press me");

// Wichtige Methoden
button.setBounds(x, y, width, height); // setzt die Koordinaten der Komponente auf dem
Fenster
button.setText("Neue Beschriftung");
button.setEnabled(true); // aktivieren (true) / deaktivieren (false) vom Button

// Special
button.addActionListener(this); // Wird zu einem späteren Zeitpunkt genauer erläutert!
```

- Ohne `button.addActionListener(this)` und der Methode `actionPerformed(ActionEvent event)` passiert noch nichts wenn man auf den Button drückt. Dies wird später eingeführt.

## UML



## Wann wird der JButton eingesetzt?

- Zur Aktionsausführung durch den Benutzer
  - (z.B. würfeln, Formular abschicken, usw...)





#### TIP

Der Elementare `do/while`-Loop einer Konsolenapplikation wird in einer GUI Applikation nicht mehr benötigt. Hier wird durch Buttons einen **Aktions-Event** erstellt und abgearbeitet. Wenn der Benutzer den Button mehrmals drückt, wird der Aktions-Event auch mehrmals abgearbeitet. **Dies macht den Loop überflüssig!**

## Komponenten in ein Fenster einfügen

Um eine Swing-Komponente in einem Fenster sichtbar zu machen, sind noch **zwei Anweisungen** notwendig. Dafür nehmen wir eine Komponente in der Variable `component` an. Die Variable `component` darf also auch anders heissen! Muss jedoch ein Objekt einer der oben vorgestellten Komponenten beinhalten.

### `component.setBounds(x, y, width, height)`

Der Aufruf von `component.setBounds(x, y, width, height)` **positioniert die Komponente im Fenster und legt die Grösse fest:**

- In einem Fenster liegt die Koordinate `(0, 0)` oben links
- `component.setBounds(10, 10, 100, 15);` bedeutet somit:
  - 10 Pixel von oben
  - 10 Pixel von links
  - 100 Pixel lang und
  - 15 Pixel hoch.

### `add(component)`

Schliesslich muss die Komponente auch noch wirklich dem Fenster hinzugefügt werden. Der Aufruf dazu lautet `add(component);` wobei `this` ein `JFrame` sein muss:



## ComponentWindow als Beispiel mit JLabel

ComponentWindow.java

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class ComponentWindow extends JFrame {
    JLabel label = new JLabel("Ich bin ein Label"); // Hier werden die Komponenten
    // initialisiert

    public void showDialog() { // Beliebiger Name, kann auch, "start" oder nur "show"
    // heissen.
        setLayout(null); // Deaktiviert Layout-Automatismen von Swing

        label.setBounds(10, 10, 100, 15); // setzt die Position und Grösse vom Label
        add(label); // fügt das Label ins Fenster ein

        setDefaultCloseOperation(EXIT_ON_CLOSE); // Beendet beim Schliessen des Fensters
        // ebenfalls den Prozess
        setSize(300, 300); // Bestimmt die grösse des Fensters
        setTitle("Ein Fenster mit Komponenten"); // Setzt den Titel des Fensters
        setVisible(true); // Muss am Ende stehen! Ohne das wird nichts angezeigt!
    }
}
```

### TIP

Komponenten sollte immer als **Instanz-Variablen** initialisiert werden! So hat man von der gesamten Klasse aus darauf Zugriff! Darum wurde hier das `JLabel label = new JLabel("Ich bin ein Label");` **nicht** in der Methode `showDialog()` initialisiert, sondern im Klassen-Body.

## Aufgaben

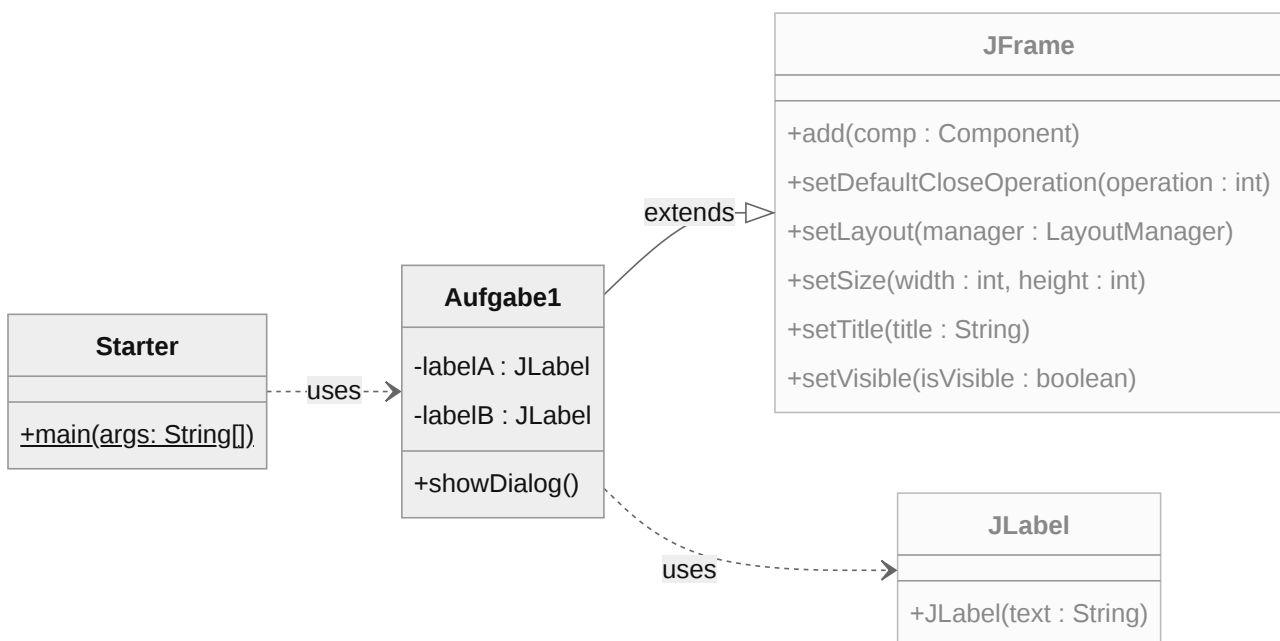
### Aufgabe 1

Ein Fenster mit zwei Label

- Erstellen Sie ein Swing-Fenster-Programm gemäss der Vorgabe rechts.
- Es sollen zwei Labels angezeigt werden.
- Verwenden Sie das folgende UML zur Hilfe.



## UML der Musterlösung



## Musterlösung

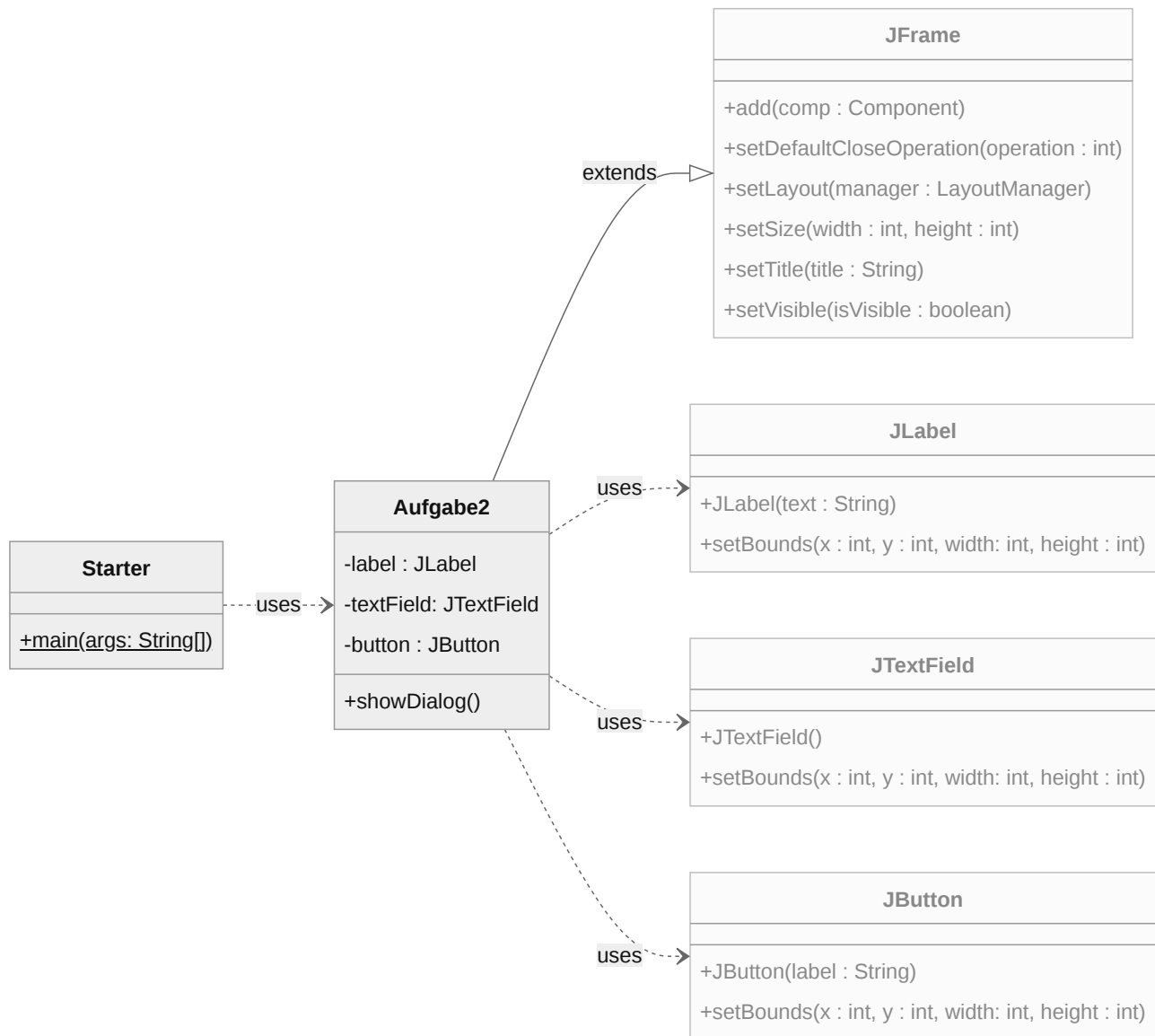
## Aufgabe 2

ein Fenster mit je einem Label, TextField und Button

- Erstellen Sie ein Swing-Fenster-Programm gemäss der Vorgabe rechts.
- Es soll eine Komponente jeder Art auf dem JFrame angezeigt werden, so wie rechts gezeigt.
- Die Befehlsschaltfläche reagiert noch nicht auf Mausklicks.
- Verwenden Sie das folgende UML als Hilfe



## UML der Musterlösung



## Musterlösung

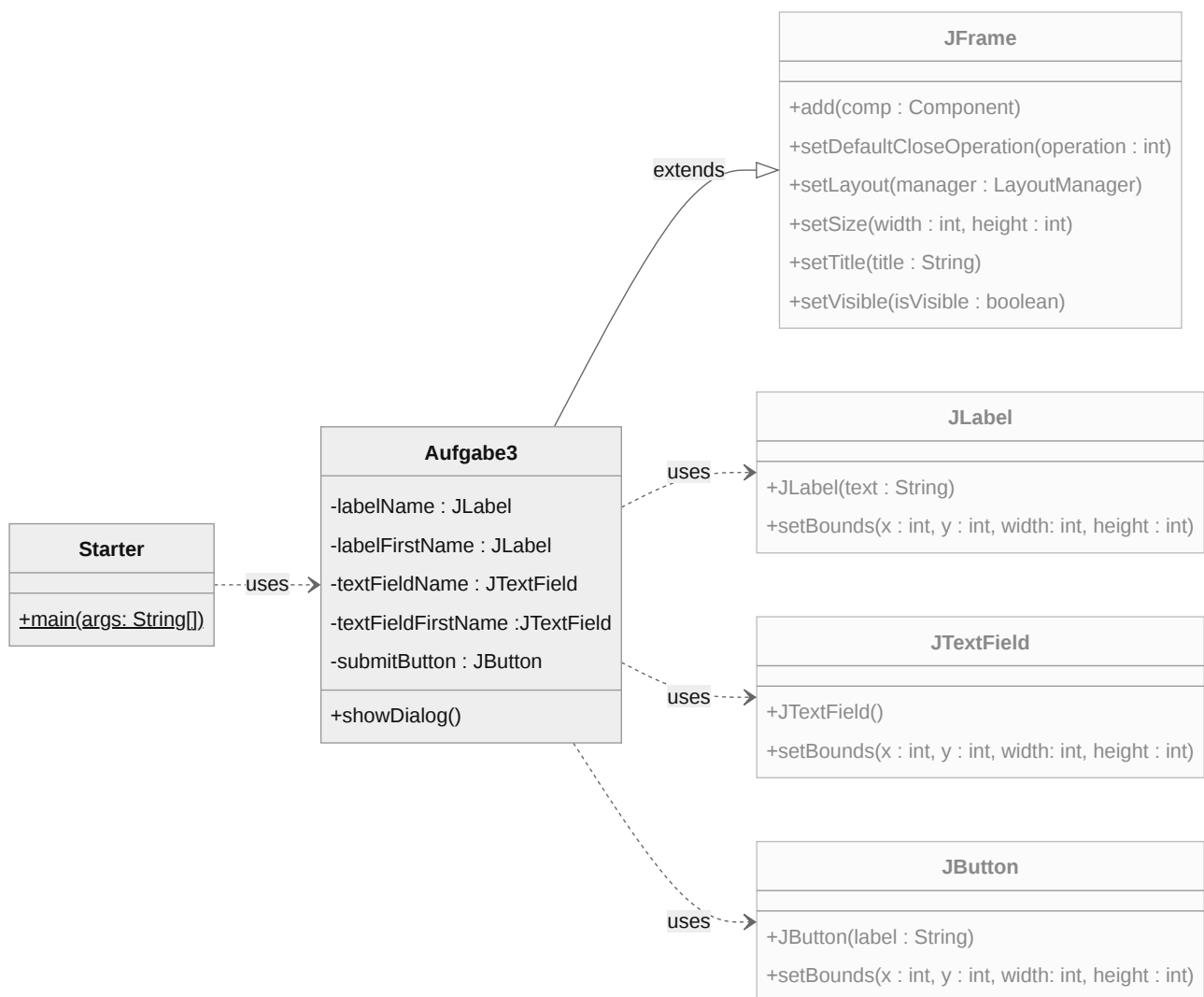
### Aufgabe 3 (Optional)

Beschriftete Textfelder

Programmieren Sie das Fenster auf der rechten Seite. Es enthält links zwei JLabel, rechts zwei JTextField, unterhalb ein JButton. Der JButton reagiert noch nicht auf Mausklicks.



## UML der Musterlösung



## Musterlösung

# 4. JButton Aktivieren

🔗 Machen Sie sich mit dem Konzept des [ActionListener](#) bekannt bevor Sie weiterfahren!

## Aufgabe 1: Einen Wert kopieren

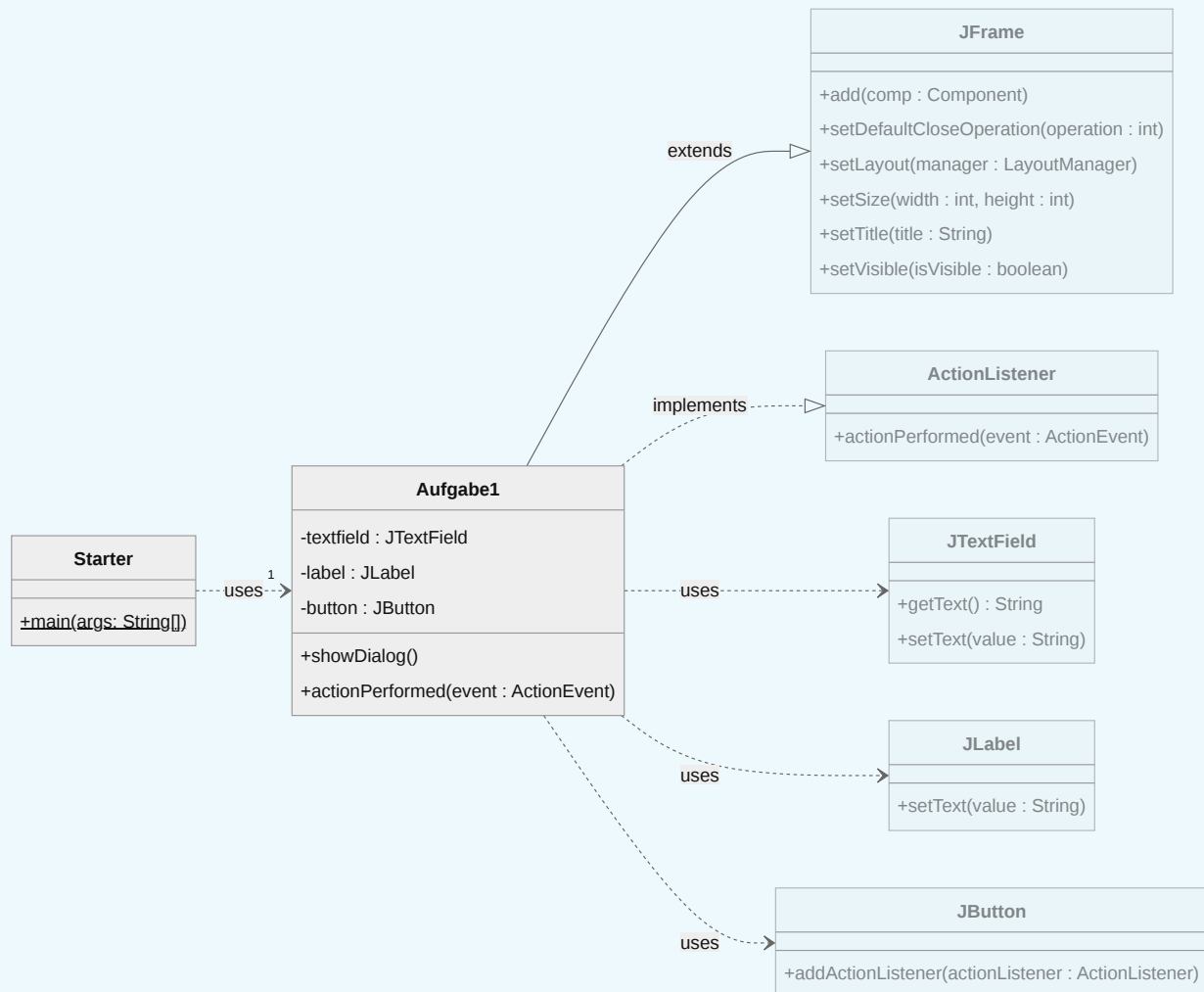
Erstellen Sie das Programm unten. Das Ziel ist, dass bei Anklicken der Schaltfläche der Wert aus dem Textfeld in das Label unterhalb kopiert wird, während das obere Textfeld geleert wird.



# UML

## ❗ JFRAME, ACTIONLISTENER, JLABEL, JBUTTON UND JTEXTFIELD WIRD IM UML

beschrieben, ist jedoch direkt in Java vorhanden. Es werden die verwendeten Methoden aufgelistet! :::



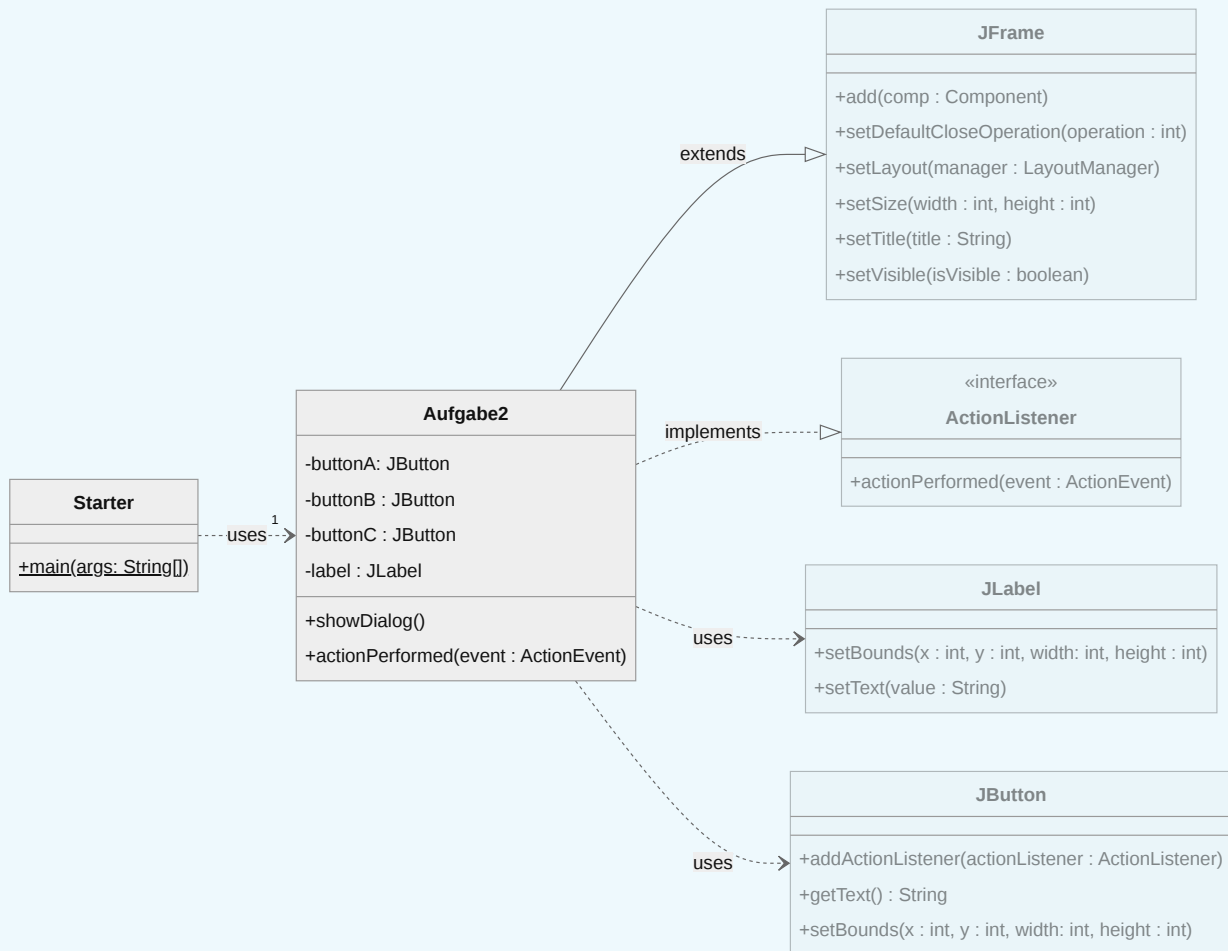
## Aufgabe 2: Mehrere Buttons auswerten

Erstellen Sie das Programm unten. Das Ziel ist, dass beim Anklicken der Schaltfläche der Wert aus dem Textfeld in das Label unterhalb kopiert wird, während das obere Textfeld geleert wird.





## UML



## Cheat Sheet

### ❗ CODE SNIPPETS FUNKTIONIEREN NUR IM RICHTIGEN KONTEXT ;) :::

```

// Die Klasse definieren
class MyClass extends JFrame implements ActionListener

// ActionListener registrieren
button.addActionListener(this);

// Method actionPerformed implementieren
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == button) {
        // do this
    } elseif (e.getSource() == button2) {
        // do that
    }
}
}

```

# 5. Strings in Zahlen umwandeln

Das nachfolgende Programm zeigt ein Swing-Programm, bei dem eine Zahl eingelesen wird, damit eine einfache Rechnung angestellt und das Ergebnis wieder ausgegeben wird.

## Beispiel: Umwandeln und 5 hinzufügen



### CastingExample.java

```
public class CastingExample extends JFrame implements ActionListener {
    JButton commandButton = new JButton("add 5 and display");
    JLabel outputLabel = new JLabel();
    JTextField entryField = new JTextField();

    public void showDialog() {
        setLayout(null);
        entryField.setBounds(10, 10, 150, 15);
        outputLabel.setBounds(10, 40, 150, 15);
        commandButton.setBounds(10, 60, 250, 20);
        add(entryField);
        add(outputLabel);
        add(commandButton);
        commandButton.addActionListener(this);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 300);
        setTitle("String umwandeln");
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        // Damit mit der Eingabe gerechnet werden kann, muss der String
        // in einen double umgerechnet werden. Dies geht mit `Double.parseDouble(String
        string)`
        double number = Double.parseDouble(entryField.getText()); // String -> double
        number = number + 5; // double wird mathematisch mutiert
        // Nach der Rechnung muss der double wieder in einen String umgewandelt werden
        outputLabel.setText("" + number); // double -> String
        entryField.setText("");
    }
}
```

## Erläuterung zum Code:

- Neu ist hier `Double.parseDouble(String string)`, welches demnach einen String als Parameter entgegen nimmt. Diese Methode verwandelt den String, welcher durch `entryField.getText()` zurück gegeben wird in eine Zahl vom Typ `double`, mit der dann gerechnet wird. Anschliessend wird das Resultat im outputLabel ausgegeben.
- Da das `outputLabel` aber einen String-Wert erwartet, wird der Datentyp durch die Methode `Double.toString(double value)` in einen String umgewandelt.

💡 **SOLCHE METHODEN GIBT ES NICHT NUR FÜR DEN DATENTYP `double`, SONDERN AUCH** für `float` und `int`. Da heissen die Methoden dann entsprechend `Float.parseFloat(String value)` und `Integer.parseInt(String value)`.

- 💡 Nach `String` kann **jeder Datentyp** mit `" " + variable` umgewandelt werden! 💡 :::

## Aufgabe

- Erweitern Sie das Programm mit einem weiteren Textfeld `entryField2`
- Benennen Sie den Button nach "Sum" um
- Bei einem Klick auf den Button "Sum" sollen die Eingaben vom `entryField` und `entryField2` aufsummiert werden und im `outputLabel` ausgegeben werden.

❗ **ZU DIESER AUFGABE GIBT ES KEINE MUSTERLÖSUNG :::**

## 6. Account Applikation

In der ersten Woche haben wir eine Konsolenapplikation implementiert, wobei man einen Betrag auf ein Konto einzahlen und abheben konnte, zusätzlich wurde zum Schluss der Kontostand angezeigt.

Sie haben sich mittlerweile schon ein grosses Wissen angeeignet, wie wir mit Swing Programme mit einer grafischen Benutzeroberfläche schreiben können. Sie wissen auch, wie Sie vom Benutzer Eingaben erhalten und Werte ausgeben können.

Jetzt geht es darum, unser Einstiegsbeispiel einer einfachen Kontoverwaltung zu überarbeiten. Versuchen Sie alles Wissen anzuwenden, das Sie sich angeeignet haben. Arbeiten Sie mit dem Faktenblatt zusammen, wenn Sie unsicher sind. Es enthält sehr viele Informationen, aber man muss wissen, wo sie stehen. Wenn Sie sich jetzt an dieses Blatt gewöhnen, kann es Ihnen bei einer Leistungsbeurteilung eine Hilfe sein.



Das Programm muss nicht genau so aussehen. Aber vielleicht möchten Sie etwas ausprobieren.

- Die Applikation startet mit Kontostand 0.
- Im Textfeld kann man einen Betrag eingeben.
- Durch Betätigen des Buttons «Deposit!» wird der eingegebene Betrag dem Konto gutgeschrieben und die Anzeige oberhalb (Balance) aktualisiert. Anschliessend wird das Textfeld mit der Eingabe geleert.
- Durch Betätigen des Buttons «Withdraw!» wird der eingegebene Betrag dem Konto abgezogen und die Anzeige oberhalb (Balance) aktualisiert. Anschliessend wird das Textfeld mit der Eingabe geleert.

## Musterlösung

### ! INFO

Die Musterlösung ist mit einem Konstruktor erstellt, Ihr müsst es jedoch nicht so machen!

# 7. Konstruktor

🔗 Machen Sie sich mit dem Konzept der **Konstruktor** bekannt bevor Sie weiterfahren!

## Aufgabe

Ändern Sie **eines** Ihrer Swing-Programme ab, indem Sie

1. In der Fensterklasse die Methode `showDialog()` (heisst vielleicht bei Ihnen anders) **umbenennen**, **so dass diese Methode zum Konstruktor wird**. Vergessen Sie nicht den **Datentyp des Rückgabewertes zu entfernen**, ein Konstruktor hat keinen Rückgabewert
2. In der Starterklasse entfernen Sie die Zeile mit dem Aufruf der Methode `showDialog()`

# 8. Easy Dice Game


-  Machen Sie sich mit den **UML Klassendiagrammen** bekannt bevor Sie weiterfahren!
-  Für diese Aufgabe müsst Ihr den **ActionListener** verstanden haben!

## Aufgabe

- Es soll ein **Würfelspiel** realisiert werden, wobei der **Spieler in eine Fachklasse ausgelagert** wird.
- Es soll das untenstehende UML-Klassendiagramm als Struktur verwendet werden.
- Die Fachklasse `GamePlayer` (Würfelspieler), kann würfeln und verwaltet die total Punkte.
- Es soll ein GUI in der Klasse `GameGui` mit zwei Spielern programmiert werden

### UML-Klassendiagramm



 **IN DER METHODE `rollTheDice()` IST FOLGENDE BERECHNUNG NÖTIG, UM EINE**  
zufällige Zahl zwischen 1 und 6 zu erzeugen:

Zufallszahl zwischen 1 und 6

```
(int) (Math.random() * 6 + 1);
```

## Arbeitsschritte

1. Programmieren Sie die Fachklasse `GamePlayer`. Untersuchen Sie, was `Math.random()` macht, und warum die weiteren Anweisungen notwendig sind.
2. Um sich mit der Klasse vertraut zu machen, erstellen Sie eine Instanz der Fachklasse und rufen die Methode `rollTheDice()` wiederholt auf, z.B. mittels einer for-Schleife.
3. Entwerfen Sie ein mögliches GUI. Überlegen Sie: welche Interaktionselemente (Schaltflächen, Textanzeige, etc.) sind nötig für das Spiel?
4. Implementieren Sie nun das GUI gemäss Ihrer Skizze.



## 9. JPanel

Für diejenigen, die mit JPanels arbeiten wollen, dient folgendes Beispielprogramm als Inspiration.

Dadurch, dass wir alles mit `setBounds` fix positionieren, ist es zusammen mit der Scrollbar nicht die schönste Variante, aber zumindest eine Verbesserung. Die Arbeit mit sogenannten Layouts wäre hier schöner, dies behandeln wir aber in einem Folgemodul.



## PanelFrame.java

```
package panelExample;

import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class PanelFrame extends JFrame implements ActionListener {

    private JButton btnNewRound = new JButton("new round");
    private JPanel pnl1 = new JPanel(); //Panel, welches andere Panels aufnimmt
    private JScrollPane scrollpane = new JScrollPane(pnl1,
        JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

    // Würde hier ein Array Sinn machen?
    // Wie kann man alle Panels aller Runden speichern?
    private RoundPanel panel = new RoundPanel();
    // Wie löse ich es für den Player zwei?

    private int y = 0; // Vertikale verschiebung der Panels

    // Ersetzt Starter.java für dieses Beispiel
    public static void main(String[] args) {
        new PanelFrame();
    }

    public PanelFrame() {
        this.setLayout(null); // fixe Positionierungen
        pnl1.setLayout(null);
        pnl1.setPreferredSize(new Dimension(350, 1000)); //PreferredSize für scrollbar

        scrollpane.setBounds(10, 10, 400, 350);
        this.add(scrollpane);

        btnNewRound.setBounds(170, 370, 100, 30);
        this.add(btnNewRound);
        this.btnNewRound.addActionListener(this);

        this.setSize(450, 450);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == this.btnNewRound) {
            // `this.` darf auch weggelassen werden
        }
    }
}
```

```
        this.panel = new JPanel() // neues Panel Objekt pro Runde
        this.panel.setBounds(0, y, 350, 60);
        this.y += 61; // y-Position des nächsten Runden Panels
        this.pnl1.add(panel); // zum scrollbaren Panel hinzufügen
        this.repaint(); // Alles neu zeichnen
    }
}
```

#### RoundPanel.java

```
package panelExample;

import java.awt.Color;

import javax.swing.JLabel;
import javax.swing.JPanel;

public class RoundPanel extends JPanel {

    // Könnte man hier auch einen Array machen?
    // Was würde ein Array für Vorteile bringen?
    private JLabel lbl1 = new JLabel("6");
    private JLabel lbl2 = new JLabel("2");
    private JLabel lbl3 = new JLabel("4");
    private JLabel lbl4 = new JLabel("4");
    private JLabel lbl5 = new JLabel("2");

    public RoundPanel() {
        this.setLayout(null);

        lbl1.setBounds(10, 10, 30, 30);
        lbl2.setBounds(40, 10, 30, 30);
        lbl3.setBounds(70, 10, 30, 30);
        lbl4.setBounds(100, 10, 30, 30);
        lbl5.setBounds(130, 10, 30, 30);

        add(lbl1);
        add(lbl2);
        add(lbl3);
        add(lbl4);
        add(lbl5);
        this.setBackground(Color.LIGHT_GRAY);
    }

    // Was fehlt, damit die einzelnen Labels von aussen geändert werden können?
}
```

## Aufgabe

- Schreiben Sie den oben bestehenden Code ab, sodass das Bild oben reproduzierbar ist.

- Versuchen Sie die einzelnen Label **im nachhinein** zu ändern
- Versuchen Sie einen Button `diceButton` hinzuzufügen, welcher würfelt und das Resultat in das entsprechende `RoundPanel` `label` schreibt
- Nach 5 `würfen` soll ein neues `RoundPanel` erstellt werden.

# 10. PlayerPanel

Hier gibt es nun noch ein JPanel Beispiel welches es durch folgende Methoden ermöglicht von aussen die neue Runden zu erstellen und auch die Würfelwerte der aktuellen Runde zu setzen.

- `playerPanel.startNewRound()`
- `playerPanel.setDiceValue(index, value)`

So ist es möglich für zwei Spieler je eine eigene Instanz der gleichen Klasse `PlayerPanel` zu erstellen. Dies ermöglicht die Darstellung der Runden und den Punkteverlauf.



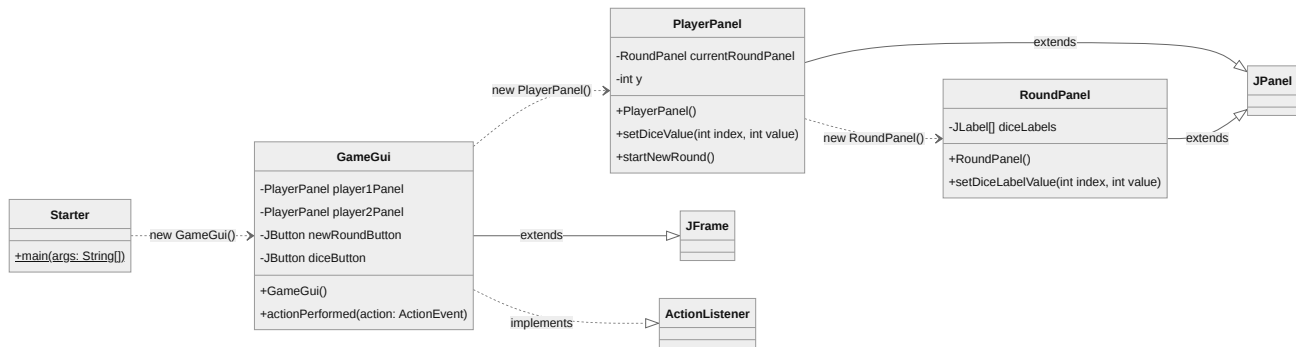
## **JPanel WIE `<div></div>` IN HTML**

- Ein `JPanel` ist ein Container, der Swing Componenten Gruppieren kann.
- `JPanel`'s können beliebig geschachtelt werden.
- Sie entsprechen damit dem `<div></div>` Tag in HTML.

## Darstellung



# UML



## Java Code

Starter.java

```
public class Starter {

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

}
```

## GameGui.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

/**
 * Das GameGui ist die Hauptklasse. Sie zeichnet das Spiel-Fenster.
 *
 * Es werden zwei `PlayerPanel`'s erstellt. Momentan wird nur das PlayerPanel
 * vom Spieler 1 aktiv verwendet! Auch werden immer automatisch 5 Würfe gemacht.
 * Versuchen Sie nun darauf aufbauend das GUI zu erweitern und auch die
 * Spiellogik zu erstellen.
 */
public class GameGui extends JFrame implements ActionListener {

    private PlayerPanel player1Panel = new PlayerPanel();
    private PlayerPanel player2Panel = new PlayerPanel();

    private JButton newRoundButton = new JButton("new round");
    private JButton diceButton = new JButton("dice 5 times");

    public GameGui() {
        this.setLayout(null);

        player1Panel.setBounds(10, 10, 350, 367);
        this.add(player1Panel);

        player2Panel.setBounds(400, 10, 350, 367);
        this.add(player2Panel);

        newRoundButton.setBounds(10, 410, 100, 30);
        this.add(newRoundButton);
        this.newRoundButton.addActionListener(this);

        diceButton.setBounds(120, 410, 140, 30);
        this.add(diceButton);
        this.diceButton.addActionListener(this);

        this.setSize(800, 550);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == this.newRoundButton) {
            // hier wird nur ein Spieler beachtet
            // natürlich sollte das Spiel mit 2 Spieler gehen
            this.player1Panel.startNewRound();
        } else if (e.getSource() == this.diceButton) {
            // 5 mal würfeln hintereinander,

```

```
// Natürlich muss dass dan schritt für schritt passieren  
// und auch abgebrochen werden können  
for (int i = 0; i < 5; i++) {  
    int rand = (int) (Math.random() * 6 + 1);  
    this.player1Panel.setDiceValue(i, rand);  
}  
}  
}  
}
```



## PlayerPanel.java

```
import java.awt.Color;

import javax.swing.BorderFactory;
import javax.swing.JPanel;

/**
 * PlayerPanel koordiniert und zeichnet RoundPanel's für einen Spieler. Die
 * Klasse dient nur zur Darstellung und beinhaltet keine Spiellogik.
 */
public class PlayerPanel extends JPanel {

    // private JPanel parentPanel = new JPanel();
    private RoundPanel currentRoundPanel;
    private int y = 1; // Vertikale Verschiebung der Panels

    public PlayerPanel() {
        this.setLayout(null);
        this.setBorder(BorderFactory.createLineBorder(Color.black));
        this.setVisible(true);
    }

    /**
     * Ermöglicht das Setzen eines Würfelwertes. Delegiert an das RoundPanel der
     * aktiven Runde.
     *
     * @param index - Der index vom Wurf, startet bei 0
     * @param value - Der Wert vom Wurf als int (1-6)
     */
    public void setDiceValue(int index, int value) {
        if (this.currentRoundPanel == null) {
            return; // Schützt vor NullPointerException!
        }
        // Hier werden die Werte delegiert, also weitergereicht
        this.currentRoundPanel.setDiceLabelValue(index, value);
    }

    /**
     * Startet eine neue Runde. Erstellt ein neues RoundPanel und speichert es als
     * aktives RoundPanel in der Instanz-Variable "currentRoundPanel". Sobald eine
     * neue Runde gestartet wurde, kann auf die vorherigen Runden nicht mehr
     * zugegriffen werden!
     */
    public void startNewRound() {
        this.currentRoundPanel = new RoundPanel(); // neues Panel Objekt pro
Runde

        // mit `this.getBounds().width` wird garantiert, dass das RoundPanel
        // gleich breit ist wie das PlayerPanel.
        this.currentRoundPanel.setBounds(1, y, this.getBounds().width - 2, 60);
        this.y += 61; // y-Position des nächsten Runden Panels
        this.add(currentRoundPanel); // zum parentPanel hinzufügen
        this.repaint(); // Alles neu zeichnen
    }
}
```

```
}  
}
```

#### RoundPanel.java

```
import java.awt.Color;  
  
import javax.swing.JLabel;  
import javax.swing.JPanel;  
  
/**  
 * Das RoundPanel dient dazu die fünf Würfe einer Runde nebeneinander  
 * darzustellen. Die Werte der Würfe können durch die Methode  
 * `setDiceLabelValue` gesetzt werden.  
 */  
public class RoundPanel extends JPanel {  
  
    // Ein Array für 5 Würfe!  
    private JLabel[] diceLabels = new JLabel[5];  
  
    // Könnte man hier noch Ergänzungen machen um auch das Total und die  
    // Rundensumme  
    // darzustellen? Es fehlt auch noch die Rundenummer.  
  
    public RoundPanel() {  
        this.setLayout(null);  
  
        for (int i = 0; i < diceLabels.length; i++) {  
            diceLabels[i] = new JLabel();  
            diceLabels[i].setBounds(10 + (i * 30), 10, 30, 30);  
            add(diceLabels[i]);  
        }  
  
        this.setBackground(Color.LIGHT_GRAY);  
    }  
  
    /**  
     * Ermöglicht das Setzen eines Würfelwertes  
     *  
     * @param index - Der index vom Wurf startet bei 0  
     * @param value - Der Wert vom Wurf als int  
     */  
    public void setDiceLabelValue(int index, int value) {  
        diceLabels[index].setText("" + value);  
    }  
}
```

## Aufgabe

- Schreiben Sie den oben bestehenden Code ab, sodass das Bild oben reproduzierbar ist.

- Dies darf als Grundlage fürs Projekt genommen werden!
- Lesen Sie den [Projektbeschreibung](#) gut durch und ergänzen Sie das Spiel.
- Schauen Sie dass die Logik in einer eigenen Klasse geschrieben wird!



# Zusatzaufgaben

## ! IMPORTANT

Versucht alle Aufgaben mit einer **Starter**, **Gui** und **Fachklasse** zu lösen!

## Zusatzaufgabe 1 - einfach

Dieser Auftrag besteht darin, einfache Eingaben für ein Benutzer-Profil zu erstellen und dann diese Eingaben als eine zusammengefasste Profil-Beschreibung anzuzeigen.

- Erstellen Sie ein GUI, in welchem man ein einfaches Profil, mit Namen, bevorzugter Farbe und Sportart eintragen kann.
- Erstellen Sie eine **Fachklasse** `Profil` für die Logik
  - Darin sollen die im GUI eingegebenen Daten gespeichert und verarbeitet werden.
- Nach Click auf einen Button wird das **kombinierte Profil** in einem `JLabel` angezeigt.



## Zusatzaufgabe 2 - mittel

Dieser Auftrag besteht in zwei Varianten:

- a) das Programm simuliert ein einmaliges würfeln, indem eine der Augenzahlen 1 bis 6 entsprechende Zahl zufällig erzeugt wird.
- b) es werden 100 Würfeldurchgänge simuliert und die zufällige Verteilung angezeigt.

### i WIE ERZEUGT MAN AM COMPUTER EINE ZUFÄLLIG GEWÜRFELTE AUGENZAHL?

```
return (int) (Math.random() * 6) + 1;
```

## Variante 1

1. einmal würfeln
2. Das GUI zeigt bei Click auf den Button jeweils eine neue zufällige Zahl an.



## Variante 2

1. hundert Mal würfeln
2. Das GUI zeigt bei Click auf den Button jeweils die Zufallsverteilung der in diesem Durchgang gewürfelten Zahlen an.



## **ActionListener**

- Dies sollen alle vor dem LB1 verstanden und verinnerlicht haben!

## **Fachklassen**

Eine Fachklasse ist eine Klasse, die nur "im Hintergrund" arbeitet und

## **Formatierung**

Wir sind keine Maschinen, sondern Menschen. Um einen Text gut lesen zu

## **Konstruktor**

Konstruktoren sind spezielle Methoden einer Klasse, die von aussen nicht als

## **Static**

Wenn wir ein Java-Programm starten, gibt es noch kein Objekt, das wir ausführen

## **UML**

Mit dem Aufkommen der Programmierung wurde auch die Frage der Kommunikation über

# ActionListener

## 🔥 ALLERWESENTLICHSTER PART!

- Dies sollen alle **vor dem LB1 verstanden und verinnerlicht haben!**
- Ohne dieses Wissen ist das Projekt und auch die Prüfung nicht zu bestehen!

## Das Interface ActionListener

- `ActionListener` ist ein Interface, welches von Java mitgeliefert wird.
- Es **definiert** die Methode `public void actionPerformed(ActionEvent e);`.
- Alle Klassen, die den ActionListener implementieren (`implements ActionListener`) **müssen** auch die Methode `public void actionPerformed(ActionEvent e);` implementieren:
- Die JavaDoc findet man [hier](#)

```
java.awt.event.ActionListener
```

```
package java.awt.event;

import java.util.EventListener;

public interface ActionListener extends EventListener {

    public void actionPerformed(ActionEvent e);
}
```

## ActionListener Beispiel: TimeButton Klasse

Das nachfolgende Programm zeigt ein Swing-Programm, das in einfacher Art interaktiv ist. Bei jedem Klick auf den Button wird in einem `JLabel` das aktuelle Datum mit Uhrzeit angezeigt. (Die Starterklasse ist weggelassen.)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Date;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

// `implements ActionListener` wird neu hinzugefügt.
// Damit wird die Methode `actionPerformed(ActionEvent e)` der Klasse hinzugefügt.
public class TimeButton extends JFrame implements ActionListener {

    private JButton commandButton = new JButton("show time");
    private JLabel outputLabel = new JLabel();

    public void showDialog() {
        setLayout(null);

        outputLabel.setBounds(10, 40, 250, 15);
        commandButton.setBounds(10, 60, 250, 20);
        add(outputLabel);
        add(commandButton);

        // Hier wird die Klasse beim Button `commandButton` registriert
        // Ohne diese Anweisung macht der `commandButton` nix!
        commandButton.addActionListener(this);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(300, 300);
        setTitle("JFrame TimeButton");
        setVisible(true);
    }

    // Diese Methode wird immer ausgeführt, wenn der `commandButton` gedrückt wird
    @Override // dies signalisiert, dass wir die Methode definieren müssen!
    public void actionPerformed(ActionEvent e) {
        // Wenn der `commandButton` gedrückt wird, wird ein neues Datum ins `outputLabel`
        // geschrieben
        outputLabel.setText("" + new Date());
    }
}
```

## Erläuterung zum Code

- Die Komponenten sind im Klassen-Body initialisiert. Sie sind dadurch **Instanz-Variablen**, das heisst sie sind überall in der Klasse sichtbar und nicht nur in dem Codeblock, in welchem sie erzeugt wurden. Statt von Variablen spricht man bei Deklarationen an dieser Stelle von Attributen, Instanz-Variablen oder Felder.



- `implements ActionListener`: Dies bewirkt, dass die Klasse in der Lage ist, Meldungen zu verarbeiten. `JButtons` können zum Beispiel solche Meldungen verschicken. Unsere Klasse ist nun **gezwungen** die Methode `actionPerformed(ActionEvent e)` zu implementieren.
- In `showDialog()` wird die Methode `commandButton.addActionListener(this);` aufgerufen. Mit diesem Aufruf kann sich eine Klasse bei einem `JButton` **registrieren** und wird anschliessend immer benachrichtigt, wenn die Schaltfläche betätigt wurde. Mit dem Schlüsselwort `this` wird ausgesagt, dass sich die Klasse selbst als `Listener` (also Zuhörer) hinzufügt.
- Wird die Schaltfläche betätigt, so benachrichtigt der `JButton` die Klasse, indem er die Methode `actionPerformed(ActionEvent e)` aufruft. Der Parameter beim Aufruf ist vom Typ `ActionEvent`. Mit diesem Event kann über die Methode `getSource()` herausgefunden werden, welche Komponente den Aufruf ausgelöst hat. So kann mittels verschiedenen if-Abfragen entschieden werden, welcher `JButton` betätigt wurde (Unten folgt ein Beispiel dazu.).
- Zudem zeigt das Beispiel wie Werte in Komponenten mit der Methode `setText(String message)` gesetzt werden.

## Feststellen, welcher Button gedrückt wurde

Falls ein Fenster mehrere Buttons hat, muss sich die Fensterklasse bei jedem Button registrieren, damit sie bei einem Klick benachrichtigt wird. Jedes Mal, wenn ein Button gedrückt wird, erfolgt ein Aufruf der Methode `actionPerformed`. In dieser Methode muss nun herausgefunden werden, wer der Urheber des Aufrufs ist.

Dies lässt sich wie folgt feststellen:

Mehrere Buttons unterscheiden mit e.getSource()

```
public class TimeButton extends JFrame implements ActionListener {
    private JButton commandButton = new JButton("show time");
    private JButton otherButton = new JButton("do something different");

    // ... nachfolgender Code ausgeblendet ...

    public void showDialog() {
        // ... vorausgehender Code ausgeblendet ...

        commandButton.addActionListener(this);
        otherButton.addActionListener(this);

        // ... nachfolgender Code ausgeblendet ...
    }

    @Override // dies signalisiert, dass wir die Methode definieren müssen!
    public void actionPerformed(ActionEvent e) {

        if (e.getSource() == commandButton) { //
            // wird ausgeführt, wenn der `commandButton` gedrückt wurde
        } else if (e.getSource() == otherButton) {
            // wird ausgeführt, wenn der `otherButton` gedrückt wurde
        }

    }
}
```

# Fachklassen

Eine **Fachklasse** ist eine Klasse, die nur "im Hintergrund" arbeitet und **nicht für die Interaktion mit dem Benutzer zuständig ist**.

## Klassenstruktur

Gemäss aktuellem Wissensstand folgen Klassen folgendem **Schema**:

Klassen- Deklaration	Klassen- Body	Instanz- Variablen	Instanz- Methoden	Methoden- Body
-------------------------	------------------	-----------------------	----------------------	-------------------

```
public class ClassName { // Klassendeklaration Start

    private int myInstanceVariable;           // Instanzvariablen

    public void setMyInstanceVariable(int value) { // Methode (setter)
        myInstanceVariable = value;
    }

    public int getMyInstanceVariable() {       // Methode (getter)
        return myInstanceVariable;
    }

} // Klassendeklaration Ende
```

- Die **Klassen-Deklaration** definiert den **Namen** der Klasse.
- Der **Namen** der Klasse definiert automatisch auch einen **Datentyp**!

### 💡 KEINE **main** METHODE IN EINER FACHKLASSE

- Es gibt **keine** Methode `public static void main(String[] args)`.
- Diese sollte **nur** in der **Starter** Klasse existieren.
- Es ist theoretisch möglich mehrere **main** Methoden zu haben, dies ist jedoch **schlechter Stil**

### 🔥 KLICKT DURCH DIE TABS!

Bitte alle Tabs einmal durchgeben und **versucht zu verstehen**, wie eine Klasse genau aufgebaut ist!

# Instanziierung und Verwendung eines Objekts/Instanz

Objekte lassen sich im Code wie folgt erstellen:

```
// Datentyp      Variable      Objektzuweisung  Objekterstellung
ClassName      variablenName      =      new ClassName();

// Es können mehrere Variablen mit Objekte der selben Klasse definiert werden
ClassName      otherClassName      =      new ClassName();

//      Mit einem Punkt "." wird auf die Instanz-Methoden zugegriffen!
variablenName.setMyInstanceVariable(12);

// Der Rückgabewert einer Methode kann in einer Variablen gespeichert werden
int value = variablenName.getMyInstanceVariable();

// Der Rückgabewert einer Methode kann auch direkt wiederverwendet werden
otherObject.setMyInstanceVariable(variablenName.getMyInstanceVariable());
```

💡 **JE BESSER DIE NAMEN, DESTO LESERLICHER WIRD DER CODE!**

`ClassName` ist in dem oberen Beispiel generisch gewählt da es sich um ein generelles Beispiel handelt. Anstatt `ClassName` sollte später ein spezifischer Name gewählt werden, wie z.B. `Account`. Der Name der Variable kann beliebig sein. Das Gleiche gilt für `Variablen` und `Methoden`

```
Account savingAccount = new Account(); // Toll
Xyz b = new Xzy();           // Evt. nicht ganz so toll ;)
```

## ❗ INSTANZ ODER OBJEKT?

Die Wörter Objekt und Instanz sind **Synonyme**, können also beliebig vertauscht werden.

## 🧠 Unterschied von einem Objekt und einer Variable

- Ein Objekt ist **immer in einer Variable** gespeichert.
- Eine Variable muss aber nicht immer ein Objekt beinhalten.

```
Account accountObjektVariable = new Account();  
int intValue = 1;  
  
// `accountObjektVariable` beinhaltet ein Objekt der Klasse Account  
// `intValue` beinhaltet den Wert 1 vom `primitiven` Datentyp int  
  
accountObjektVariable.getClass(); // Ein Objekt besitzt Methoden welche ausgeführt  
werden können  
intValue.getClass(); // führt zu einem ERROR. (versuche es in `eclipse`!)
```







### FAUSTREGEL

- Wenn der **Datentyp** mit einem **Grossbuchstaben** anfängt (`String`, `Account`, ...) handelt es sich um ein **Objekt**.
- Wenn der **Datentyp** mit einem **Kleinbuchstaben** anfängt (`int`, `double`, `char`, ...) ist es **kein Objekt**.
- Ein **Objekt besitzt Methoden**, welche man ausführen kann, ein **primitiver Datentyp nicht**.
  - **Jedes Objekt** besitzt die Methode `getClass()`;

# Formatierung

Wir sind keine Maschinen, sondern **Menschen**. Um einen Text gut lesen zu können, brauchen wir gute Struktur! Um eine Konvention zu haben, haben wir in diesem Unterricht folgende Regeln definiert!

## Unsere Regeln

- Wir verwenden **keine Umlaute** im Code (Ausnahmen sind Kommentare).
- Jede **Klasse**
  - beginnt mit einem **Grossbuchstaben**
  - hat einen `AusdrucksstarkenNamen` in  `UpperCamelCase` 
- Jede **Methode**
  - beginnt mit einem **Kleinbuchstaben**
  - hat einen `ausdrucksstarkenNamen` in  `lowerCamelCase` 
- Blöcke `{ }` werden eingerückt ( **Ctrl-Shift-F**  **Command-Shift-F**)
- **Standard-Encoding UTF-8**: Preferences > General > Workspace → UTF-8

## Unformatiert

- Dieser Quellcode unten ist für uns nicht gut lesbar.
- Eclipse hat damit keine Probleme und meldet auch zu Recht keinen Fehler.

```
import java.util.Scanner; public class XYZ { public static void main(String[] args) {
System.out.println("Welcome to the account application"); double k = 0; double a = 0;
String c = ""; do { Scanner sc = new Scanner(System.in); System.out.println("Please
enter the a, 0 (zero) to terminate"); a = sc.nextDouble(); if (a != 0) {
System.out.println("To deposit, press +, to withdraw press -"); c = sc.next(); if
(c.equals("+")) { k = e(k, a); } else if (c.equals("-")) { k = ab(k, a); } } } while (a
!= 0); System.out.println("Final balance: " + ak(k)); } public static double e(double
ks, double b) { return ks + b; } public static double ab(double ks, double bt) { return
ks - bt; } public static double ak(double ks) { return ks; } }
```

### TIP

- Damit andere Entwickler unseren Code gut lesen können (und wir selbst auch), werden wir uns an **einige Regeln** halten.
- Viele Firmen haben übrigens eine ganze Reihe definierter Regeln, wie Quellcode aussehen soll. Hier finden Sie zum Beispiel die Regeln, wie bei Google Java Code zu formatieren ist.

# Formatiert, mit schlechten Namen

- Dieser Quellcode unten ist für uns gut lesbar.
- Die Namen sind jedoch schlecht!
- Es ist nicht erkennbar, was der code wirklich macht, ohne ihn zu analysieren! 🤖

```
import java.util.Scanner;

public class Xyz {
    public static void main(String[] args) {
        System.out.println("Welcome to the account application");
        double k = 0;
        double a = 0;
        String c = "";
        try(Scanner sc = new Scanner(System.in)) {
            do {
                System.out.println("Please enter the a, 0 (zero) to terminate");
                a = sc.nextDouble();
                if (a != 0) {
                    System.out.println("To deposit, press +, to withdraw press -");
                    c = sc.next();
                    if ("+".equals(c)) {
                        k = e(k, a);
                    } else if ("-".equals(c)) {
                        k = ab(k, a);
                    }
                }
            } while (a != 0);
        };
        System.out.println("Final balance: " + ak(k));
    }

    public static double e(double ks, double bt) {
        return ks + bt;
    }

    public static double ab(double ks, double bt) {
        return ks - bt;
    }

    public static double ak(double ks) {
        return ks;
    }
}
```

# Formatiert und gut benannt

```
import java.util.Scanner;

public class AccountApplication {
    public static void main(String[] args) {
        System.out.println("Welcome to the account application");
        double balance = 0;
        double amount = 0;
        String command = "";
        try(Scanner sc = new Scanner(System.in)) {
            do {
                System.out.println("Please enter the amount, 0 (zero) to terminate");
                amount = sc.nextDouble();
                if (amount != 0) {
                    System.out.println("To deposit, press +, to withdraw press -");
                    command = sc.next();
                    if ("+".equals(command)) {
                        balance = deposit(balance, amount);
                    } else if ("-".equals(command)) {
                        balance = withdraw(balance, amount);
                    }
                }
            } while (amount != 0);
        }
        System.out.println("Final balance: " + getBalance(balance));
    }

    public static double deposit(double balance, double amount) {
        return balance + amount;
    }

    public static double withdraw(double balance, double amount) {
        return balance - amount;
    }

    public static double getBalance(double balance) {
        return balance;
    }
}
```



# Konstruktor

Konstrukturen sind spezielle Methoden einer Klasse, die von aussen nicht als Methode aufgerufen werden können, aber bei der Instanziierung eines Objektes aufgerufen werden können.

Ein Konstruktor:

- Methodenname ist **immer gleich** wie die Klasse
- ist **nicht explizit aufrufbar**
- wird **ausgeführt, wenn ein Objekt erstellt wird** (💡 in Verbindung mit `new`)
- hat **keinen Rückgabewert**
- es können mehrere Konstrukturen bestehen (💡 andere Anzahl Parameter)
- werden **keine Parameter** angegeben, nennt man ihn **Standardkonstruktor**
- dient dazu das **Objekt** mit gültigen Werten zu **initialisieren**

## Deklaration

MyClass.java Konstruktor Beispiele

```
public class MyClass {  
    private String name; // Instanzvariable die Inizialisiert werden muss!  
    private int year = 2000; // Instanzvariable mit Standardwert  
  
    public MyClass() { // Standardkonstruktor (ohne Parameter)  
        this.name = "Startwert"; // `name = "Startwert"` ohne `this` ist auch gültig  
    }  
  
    public MyClass(String name) { // Konstruktor mit gleichnamigem Parameter  
        this.name = name; // `this` ist notwendig da gleichnamig  
    }  
  
    public MyClass(String aName, int year) { // Konstruktor mit zwei Variablen  
        name = aName; // `this` darf weggelassen werden (muss aber nicht!)  
        this.year = year; // `this` ist notwendig da gleichnamig  
    }  
}
```

# Verwendung

Starter.java verwendet MyClass

```
public class Starter {  
    public static void main(String[] args) {  
        // Standardkonstruktor wird ausgeführt!  
        MyClass myClass = new MyClass();  
  
        // Konstruktor mit einem Parameter wird ausgeführt  
        MyClass myClass2 = new MyClass("Neuer Startwert");  
  
        // Konstruktor mit zwei Parameter wird ausgeführt  
        MyClass myClass3 = new MyClass("Neuer Startwert", 2022);  
    }  
}
```

# Erläuterung

- **Jede** Klasse besitzt einen Defaultkonstruktor,
  - wenn wir ihn nicht explizit hinschreiben, dann erzeugt der Java-Compiler einfach selbst einen, der Nichts macht
  - der Defaultkonstruktor hat **keine Parameter**.
- Wenn wir einem **Konstruktor mit Parametern** Werte übergeben, kann er diese als Startwerte für das Objekt verwenden.
  - Das Objekt im zweiten Aufruf in `main` ist also gleich mit dem Startwert "Neuer Startwert" initialisiert.
- Wenn der **Name von Parametern gleich ist wie der Name einer Variablen**, dann muss mit dem Schlüsselwort `this` gearbeitet werden.
  - Der Einsatz dieses Wortes bedeutet, dass damit die Instanzvariable gemeint ist und nicht der Parameter

# Static

Wenn wir ein Java-Programm starten, gibt es noch kein Objekt, das wir ausführen könnten. Das Java-Schlüsselwort `static` ist die Lösung für dieses Problem. Elemente einer Klasse, die mit `static` markiert werden, sind nicht abhängig davon, ob es ein Objekt der Klasse gibt oder nicht. Diese Elemente existieren immer.


Wenn wir also eine Methode `public static void main(String[] args)` geschrieben haben, dann existiert diese Methode beim Programmstart im Speicher und ist ausführbar. Auf diese Weise können wir unsere Programme starten.

In der Regel erstellt man eine **Startklasse**, welche die Methode `public static void main(String[] args)` enthält. In dieser Methode erstellt man dann ein Objekt des eigentlichen Programms und ruft die Methode auf, die den Programmfluss steuert.

## Was kann `static`?

- Kann ohne `new` aufgerufen werden.
- Kann wiederum andere `static` Methoden aufrufen.
- Kann `static` Variablen verwenden.
- Kann mit `new` ein `Objekt/Instanz` einer beliebigen Klasse erstellen.

## Wofür sind `static` Methoden gut?

- Die Java `public static void main(String[] args)` Methode ist immer `static` (Programmanfang).
- Generelle/Universelle Helfermethoden  **ohne Datenstand**
  - Z.B. die Java Klasse `Math` ist komplett statisch `Math.sqrt(64);`

## Static vs. Instanz-Methoden

Eine `static` Methode einer `Klasse` kann direkt aufgerufen werden, ohne dass ein `Object/Instanz` der Klasse erstellt werden muss.

```
public class MixedExample {
    private static final double PI = 3.14; // Konstante, kann nicht geändert werden!
    private String greeting = "Hello";      // Instanz-Variablen, kann geändert werden

    public static double staticCircle(double radiant) {
        return radiant * radiant * PI; // Kann auf `PI` zugreifen nicht aber auf
`greeting`
    }

    public String instanceGreeting(String name) {
        return greeting + " " + name; // Kann auf `greeting` zugreifen
// Könnte theoretisch auch auf `PI` zugreifen
    }

    public void setGreeting(String greeting) { this.greeting = greeting; }
}
```

## Verwenden von `MixedExample`

```
public class Starter {
    // Startpunkt des Programms, ist immer static!
    public static void main(String[] args) {

        // Statische Methoden können ohne new ausgeführt werden!
        double circle = MixedExample.staticCircle(1.5d);

        // Um instanceMethoden aufzurufen, muss zuerst eine Instanz erstellt werden
        MixedExample mixedExampleInstance = new MixedExample();
        String greeting = mixedExampleInstance.instanceGreeting("Lukas");
        // Wert ist "Hallo Lukas";

        mixedExampleInstance.setGreeting("Ciao") // Objekt ändern
        greeting = mixedExampleInstance.instanceGreeting("Lukas");
        // Wert ist "Ciao Lukas";
    }
}
```

# UML

Mit dem Aufkommen der Programmierung wurde auch die Frage der Kommunikation über Programme immer wichtiger. Bei Einführung der Objektorientierung kam man bald nicht mehr mit Flussdiagrammen weiter und behalf sich mit einer neuen Notation, der **UML, Unified-Modelling-Language**. Sie hat sich inzwischen breit durchgesetzt, beispielsweise bis in den Geschäftsbereich und damit der **Vermittlung zwischen Fachabteilung und Softwareentwicklung**, und für komplexe Projekte mit Tools, die eine automatische Umwandlung zwischen bestimmten Diagrammtypen und Quelltext beherrschen.

Es gibt viele UML Diagrammtypen, wirklich Verwendung finden vor allem folgende:

- [Klassendiagramm](#)
- [Sequenzdiagramm](#)
- ERM, Entity-Relationship-Diagram (Für Datenbanken)

## Klassendiagramm

- Eine Klasse ist ein Rechteck
- Klassenname ist zentrierter Titel
- Sichtbarkeit
  - - ist `private`
  - + ist `public`
- Oberhalb: Instanz**variablen**
- Unterhalb: Instanz**methoden**
- Unterstrichen: `static`



## Methoden

UML	Java Signatur
+setName(name : String)	public void setName(String name)
+getName() : String	public String getName()
+sum(a : int, b: int) : int	public static int sum(int a, int b)
-secret(key : String) : String	private String secret(String key)

+|-methodennamen( variablenNamen: Datentyp ) : returnDatentyp

## Variablen

+|- variablenNamen : Datentyp

UML	Java
-name : String	private String name;

UML	Java
+year : int	public int year;
+PI : double	public static double PI;

## Abhängigkeiten

**Klasse verwendet ein `new` Objekt**

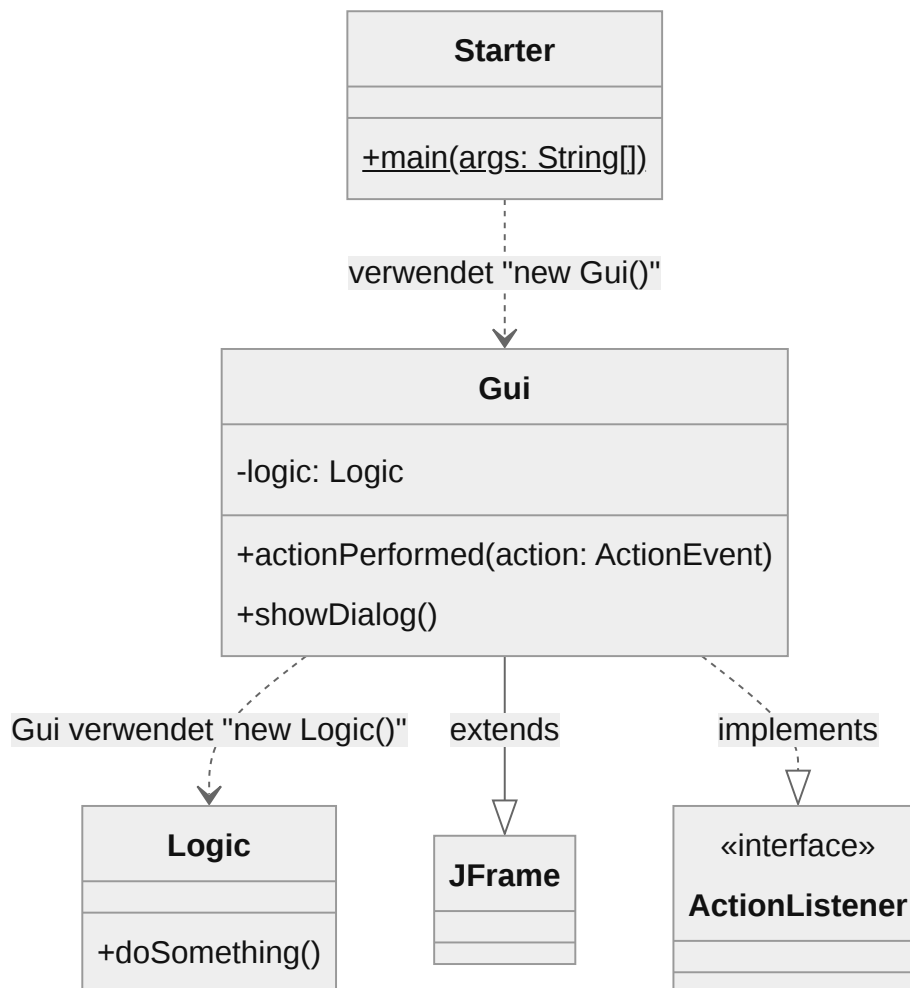
- gestrichelter Pfeil

**Klasse `implements` ein Interface**

- gestrichelter Pfeil mit **Dreiecksspitze**


**Klasse `extends` eine Klasse**

- durchgezogener Pfeil mit **Dreiecksspitze**



# UML Tools

## ! DIE EINFACHSTE ART!

- [IntelliJ Diagrams](#) *jedoch nicht 100% UML Standard!*
  - IntelliJ finden Sie auf unseren **Windows VMs**
- [Mermaid](#) 
  - wird auf dieser Seite verwendet
  - [Mermaid Dokumentation](#)
  - [Mermaid Live im Browser](#)
- ObjektAid funktioniert nur bis **Eclipse 2022-06 (4.24.0)**
  - [ObjektAid for Eclipse](#)
  - [Video mit Installationsanleitung](#)