

Informatik 1

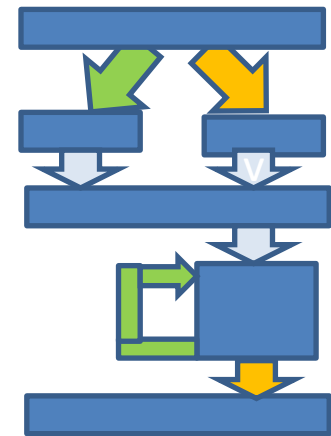
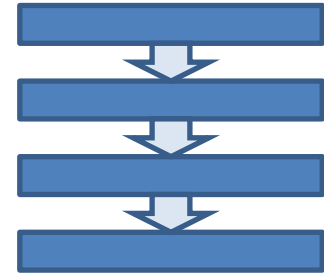
Kontrollanweisungen

Inhalt

- Fallunterscheidungen
 - if-else
 - switch
- Schleifen
 - while
 - do – while
 - for

Kontrollanweisungen

- Bisher
 - Nur streng sequentielle Programmabläufe möglich
 - Ausnahmen: Kurzschlussoperator und ternärer Vergleichsoperator in Ausdrücken
- Kontrollanweisungen
 - steuern den Programmablauf, in Abhängigkeit einer **Bedingung**, **true** oder **false** werden kann
 - ermöglichen nicht sequentielle Abläufe



Fallunterscheidung if

- Einfachauswahl
- Syntax

```
if (Bedingung) {  
    Anweisungen-if  
}
```

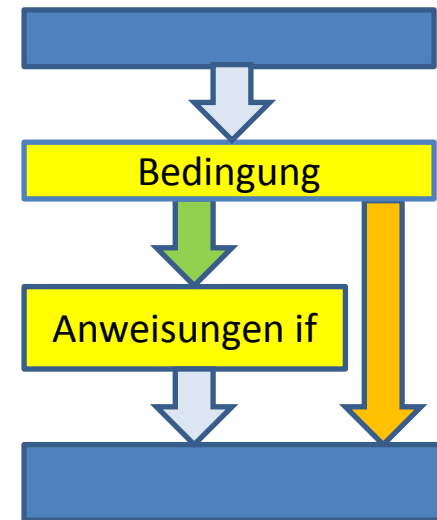
Boolescher Ausdruck

Keine, ein oder mehrere
Anweisungen
Immer { .. } verwenden

- Semantik

1. Die Bedingung wird einmal ausgewertet
2. Falls Ergebnis **true** ist, dann wird Anweisungen-if ausgeführt

Falls Ergebnis **false** ist, dann wird mit den Anweisungen nach der Kontrollanweisung fortgefahren



Fallunterscheidung if-else

- Zweifachauswahl

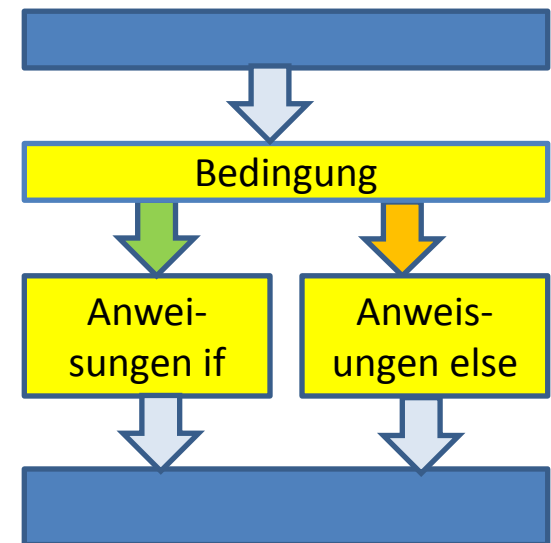
- Syntax

```
if (Bedingung) {  
    Anweisungen-if  
} else {  
    Anweisungen-else  
}
```

- Semantik

1. Die Bedingung wird einmal ausgewertet
2. Falls Ergebnis **true** ist, dann wird Anweisungen-if ausgeführt

Falls Ergebnis **false** ist, dann wird Anweisungen-else ausgeführt



Aufgabe if-else

Gegeben:	Ein Gebrauchtwagen mit Preis in Euro, dem Kilometerstand und die Informationen, ob es sich um einen Sportwagen handelt.
Gesucht:	Ausgabe, die mit Gründen beschreibt, ob der Wagen gekauft wird oder nicht.
Der Wagen wird gekauft, wenn der Preis weniger als 1000 Euro beträgt oder der Kilometerstand kleiner als 10000 km ist.	

Exakte Beispielausgaben	
Wagen kaufen. Sehr günstig. Wenig gefahren.	<- eine der beiden Gründe <- kann wegfallen
Nicht kaufen. Obwohl es ein Sportwagen ist.	<- nur bei einem Sportwagen

Geeignete Variablen und Datentypen wählen.

Ggf. Schrittweise Verfeinerung anwenden.

Nicht den ternären Vergleichsoperator `?:` verwenden.

Wie kann die Redundanz vermieden werden?

Fallunterscheidung switch

- Mehrfachauswahl
- Syntax (beispielhaft)

Arithmetischer Ausdruck
(Datentyp int)

```
switch (Ausdruck) {  
  case KonstanterAusdruck1: Anweisung-1  
                           Anweisung-2  
  case KonstanterAusdruck2: Anweisung-3  
                           Anweisung-4  
  default:                 Anweisung-5  
                           Anweisung-6  
  case KonstanterAusdruck3: Anweisung-7  
                           Anweisung-8  
}
```

Arithmetischer Ausdruck,
der nur konstante Werte
enthält (int)
Werte müssen
unterschiedlich sein

Schlüsselwort
optional, darf nur
einmal im switch
verwendet werden

Einzelne Anweisung,
auch Kontrollanweisung
(optional)

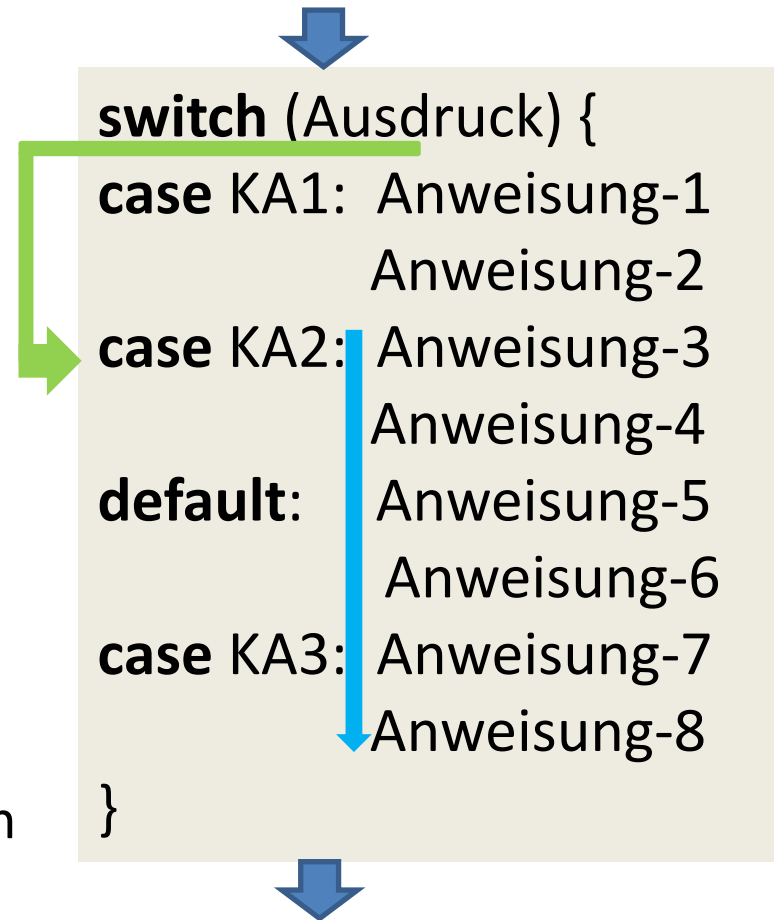
Fallunterscheidung switch

- Semantik (beispielhaft)
 1. Ausdruck wird einmal ausgewertet
 2. Falls ein case mit identischem Wert vorhanden ist, dann werden **alle Anweisungen nach dem case von oben nach unten ausgeführt**

Programmkontrolle **springt** zum passendem case (Sprungmarke)

Falls kein passender case existiert, aber ein **default**, dann wird zum default gesprungen

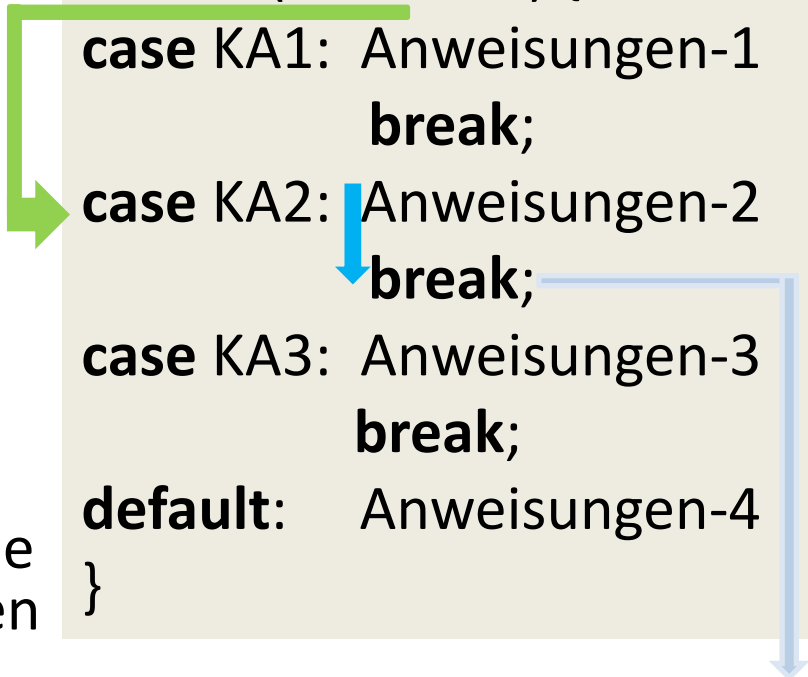
Ansonsten werden die Anweisungen nach dem switch ausgeführt



Fallunterscheidung

- switch wird meist verwendet, um mehrere Fälle voneinander zu trennen
- default am Ende
- Anweisung: **break;**
 - Bricht die umgebende Kontrollanweisung ab
 - Ausnahme: if-else
 - Bei verschachtelten Kontrollanweisung werden die äusseren Kontrollanweisungen nicht mit abgebrochen

```
switch (Ausdruck) {  
  case KA1: Anweisungen-1  
            break;  
  case KA2: Anweisungen-2  
            break;  
  case KA3: Anweisungen-3  
            break;  
  default:  Anweisungen-4  
}
```



Fallunterscheidung

- Beispiel:
 - break bricht nur inneres switch ab
 - “Welt” wird ausgegeben

```
int a = 1;
int b = 2;

switch (a + 1) {
case 2:
case 3:  switch (b - 1) {
        case 1: System.out.print("Hallo ");
                break;
        }
        System.out.println("Welt");
}
```

Fallunterscheidung

- Vor einer Kontrollanweisung darf eine **Markierung** (label) gesetzt werden
- Syntax:
 Bezeichner: Kontrollanweisung
- **break** Bezeichner;
 - Bricht die markierte Kontrollanweisung ab
 - vermeiden

```
int a = 1;
int b = 2;
mark: switch (a + 1) {
case 2:
case 3:  switch (b - 1) {
        case 1: System.out.print("Hallo ");
                break mark;
        }
        System.out.println("Welt");
}
```

Aufgabe switch

Pärmie einer Hausratsversicherung berechnen

Gegeben:	Wohnfläche in Quadratmetern, Anzahl Personen des Haushalts
----------	--

Gesucht:	Zu zahlende Jahresprämie in Euro
----------	----------------------------------

Berechnung (f = Wohnfläche):	
---------------------------------	--

Bei 1 bis 2 Personen:	$10 + f : 2$
-----------------------	--------------

Bei 3 Personen:	$15 + f$
-----------------	----------

Bei 4 Personen:	$15 + 1,5 f$
-----------------	--------------

Mehr als 4:	$25 + 2f$
-------------	-----------

Redundanz vermeiden: Identische, gleiche oder ähnliche Berechnungen und Programmteile sollen nur einmal im Quelltext vorkommen

while Schleife

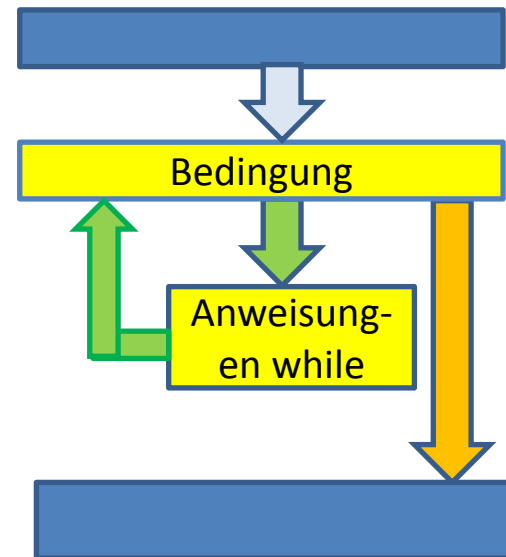
- Kopfgesteuerte Schleife
- Syntax

```
while (Bedingung) {  
    Anweisungen-while  
}
```

Schleifenrumpf:
{ Anweisung-while }

- Semantik
 1. Die Bedingung wird ausgewertet
 2. Falls Ergebnis **true** ist, dann wird Anweisungen-while ausgeführt. Die Steuerung kehrt wieder zum Kopf der Schleife zurück (zu 1.)

Falls Ergebnis **false** ist, dann wird mit den Anweisungen nach der Kontrollanweisung fortgefahren (Schleife bricht ab)



while Schleife

- Beispiel

Größte gemeinsame Teiler zweier Zahlen bestimmen

Gegeben:	Zwei ganze positiven Zahl $a > 0$ und $b > 0$
Gesucht:	Der größte gemeinsame Teiler von a und b

- Euklidischer Algorithmus:
 - Solange a und b verschieden sind, ziehe die kleiner von der größeren Zahl ab
- Behauptung
 - Das Verfahren bricht ab, am Ende sind beide Werte gleich und entsprechen den größten gemeinsamen Teiler der vorherigen Zahlen a und b

while-Schleife

```
public static int  
groessteGemeinsamerTeilerBerechnen(int a, int b) {  
    while (a != b) {  
        if (a > b) {  
            a = a - b;  
        } else {  
            b = b - a;  
        }  
    }  
  
    return a;  
}
```

while-Schleife

- Korrektheit
 - Wenn das Programm zu ende ist, dann ist für eine gegebenen Eingabe das berechnete Ergebnis immer die zugehörige gesuchte Ausgabe
- Terminierung
 - Für jede gesuchte Eingabe bricht das Programm am Ende immer ab
- Falls die Eingabe nicht den geforderten Wertebereich hat:
 - dann muss die Ausgabe nicht korrekt sein
 - oder das Programm muss nicht terminieren
 - es ist nicht definiert, was das Programm in diesen Fällen machen soll
 - In der Praxis: Eingabe auf Korrektheit prüfen und ggf. dem Benutzer einen Fehlertext anzeigen

while-Schleife

- Ziel
 - *Korrekte* Programme für Probleme entwickeln, die *terminieren*
- Problem
 - Durch Testen ist Korrektheit und Terminierung nicht nachweisbar
 - Beweisen ist theoretisch möglich, aber aufwendig
- In der Praxis
 - Programm ausgiebig testen!
 - Programmierer muss sich von Korrektheit des Programms selbst überzeugen (skeptisch sein)

do-while Schleife

- Fußgesteuerte Schleife

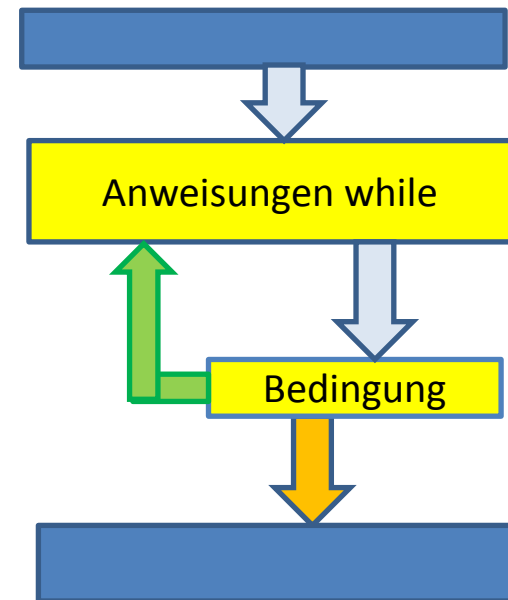
- Syntax

```
do {  
    Anweisungen-while  
} while (Bedingung);
```

- Semantik

← Semikolon wichtig!

1. Schleifenrumpf wird ausgeführt
2. Bedingung wird ausgewertet
 - a) Falls Ergebnis **true** ist, dann kehrt die Steuerung zum Kopf der Schleife zurück (zu 1.)
 - b) Falls Ergebnis **false** ist, bricht die Schleife ab



do-while Schleife

- Beispiel
 - Benutzer soll eine Zahl zwischen 0 und 4 eingeben
 - Z.B. Zeilennummer für das Setzen eines Spielsteins bei einem Spiel
- Lösung umgangssprachlich
 - Gebe eine Zahl ein solange, bis sie nicht zwischen 0 und 4 liegt

do-while Schleife

```
do {  
    A  
} while (B);
```



```
A  
while (B){  
    A  
}
```

- Beide Varianten sind gleich
- Wieso existiert diese spezielle Schleife?
- do-while Schleife wird selten verwendet

for Schleife

- Syntax

Deklaration lokaler Variable(n) eines Datentyps (optional)

```
for (Deklaration; Bedingung; Anweisungen) {  
    Anweisungen-for  
}
```

Mit **Komma** separierte elementare Anweisungen (optional)

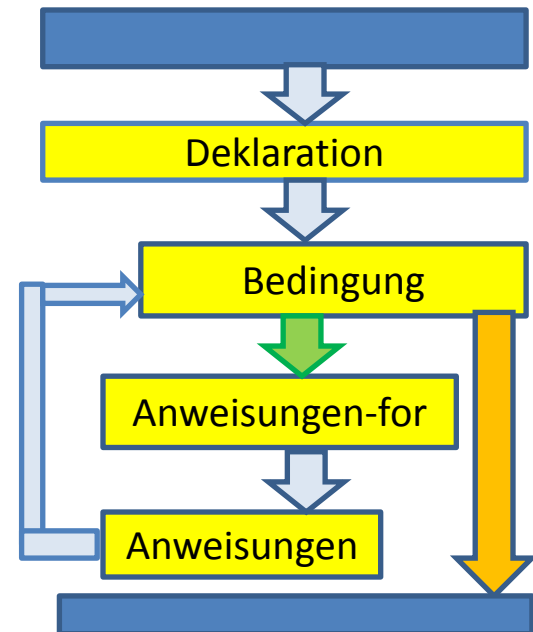
optional (true falls weggelassen)

- Semantik

wie **while**, aber

Deklaration wird genau einmal am Anfang ausgeführt

Anweisungen werden nach Rumpf und vor Bedingung ausgeführt



for Schleife

- Zähl- oder Laufvariable:
 - Variable, die nur zum Aufzählen von Werten eines bestimmte Wertebereichs dient
 - Bei Zahltypen Laufvariablen, die keine weitere Bedeutung haben, i, j, k, l nennen
 - Sie werden von einem **Startwert** ausgehend bis zu einer **Grenze** mit einer **Schrittweite** hoch oder heruntergezählt
- for-Schleife verwenden
 - Deklaration: Laufvariable mit Startwert initialisieren
 - Bedingung: Schleife bricht bei Erreichen der Grenze ab
 - Anweisungen: Hoch- oder Runterzählen
- Beispiel
 - Kleine Ein-Mal-Eins als Tabelle auf dem Bildschirm ausgeben

continue

- Syntax:
continue;
continue Bezeichner;
- Bricht die Ausführung des Schleifenrumpfs ab und führt die Schleife am Anfang wieder fort
- Ausführungsreihenfolge für den Menschen ist dadurch nur schwer durchschaubar

for Schleife

- for nicht verwenden, wenn
 - in der Bedingung die deklarierte(n) Variablen(n) überhaupt nicht vorkommen oder
 - die deklarierten Variable(n) nicht am Ende jeden Durchlaufs geändert werden
- Generell:
 - Schleifen immer über die Abbruchbedingung beenden
 - Insbesondere kein break im Rumpf verwenden
 - Schleifen nicht mit continue fortführen

Aufgaben Schleifen

Fakultät berechnen

Gegeben: Natürliche Zahl $n \geq 0$

Gesucht: $n!$ ($= 1 * 2 * 3 * 4 * \dots * n$)
 $0! = 1$

Näherungslösung von π (Wallissche Produkt)

Gegeben: Ein Wert $n > 0$

Gesucht: $2 \left(\frac{2}{1} \frac{2}{3} \frac{4}{3} \frac{4}{5} \frac{6}{5} \frac{6}{7} \dots \frac{2n}{2n-1} \frac{2n}{2n+1} \right)$