

Vorlesung Programmieren

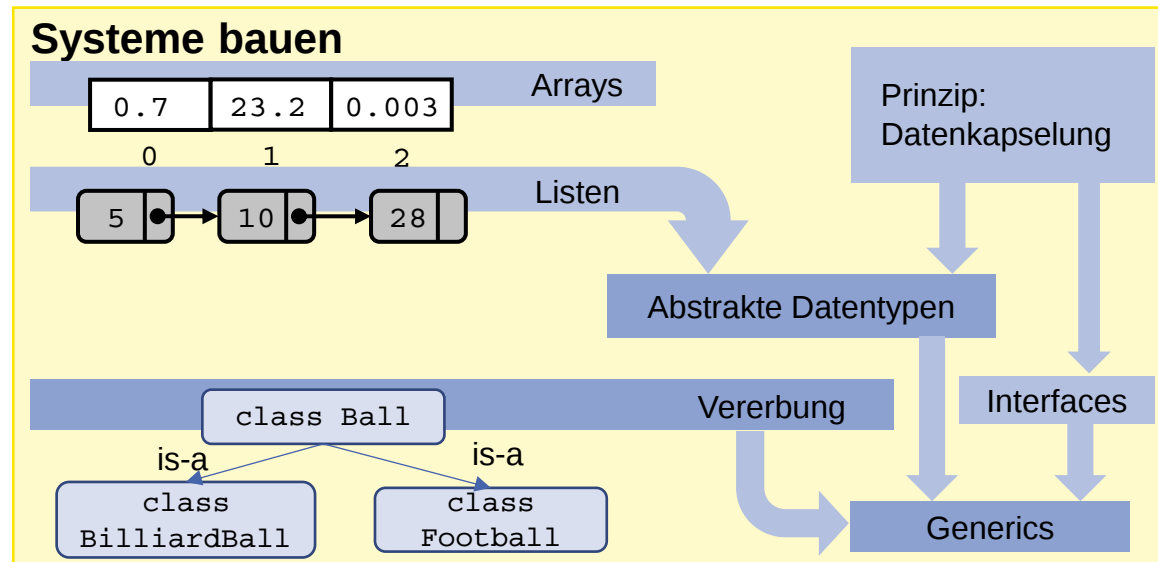
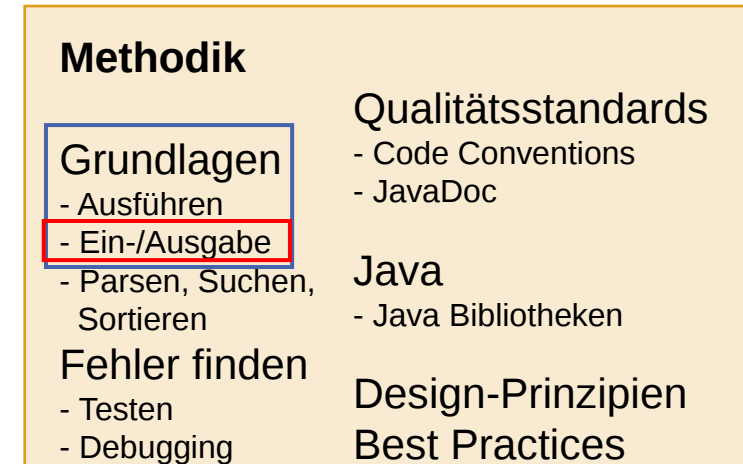
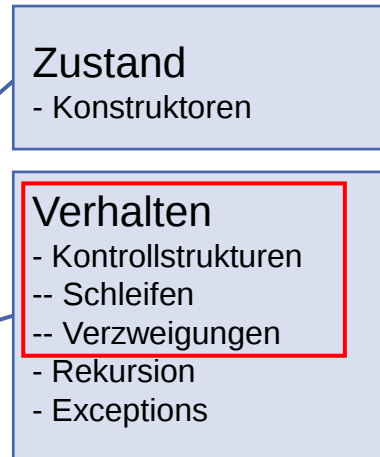
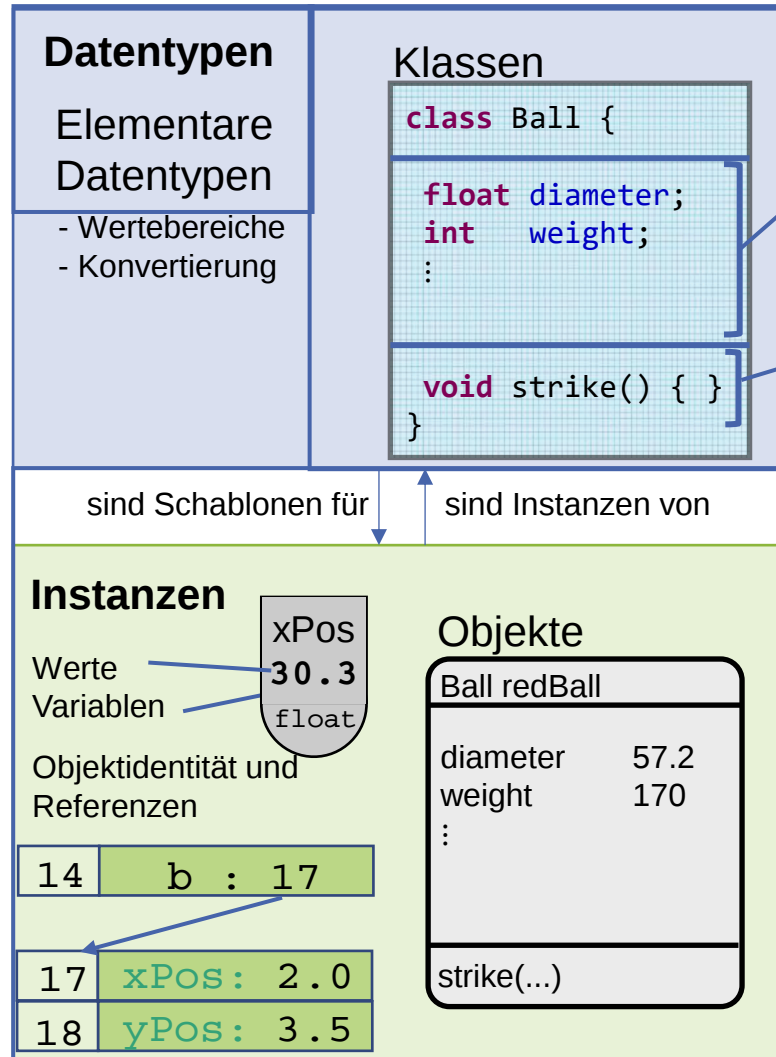
3. Kontrollstrukturen

Prof. Dr.-Ing. Anne Koziolk



Vorlesungsüberblick

Objektorientierte Programmierung in Java



Vorlesungsüberblick: Vorläufiger Semesterplan

21.10.2024	Erstsemesterbegrüßung: Einführung
23.10.2024	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
30.10.2024	Typen und Variablen
06.11.2024	Kontrollstrukturen (+Scanner)
13.11.2024	Konstruktoren und Methoden
20.11.2024	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
27.11.2024	Listen und Abstrakte Datentypen
04.12.2024	Vererbung Audimax 50.34 Raum -101 und Raum -102
11.12.2024	Exceptions; Interfaces
18.12.2024	Generics; Rekursion
08.01.2025	Java-API; Objektorientierte Design-Prinzipien
15.01.2025	Best Practices; Finden und Beheben von Fehlern
22.01.2025	Testen und Assertions
29.01.2025	JUnit; Parsen, Suchen, Sortieren
05.02.2025	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen
12.02.2025	Wrap-Up

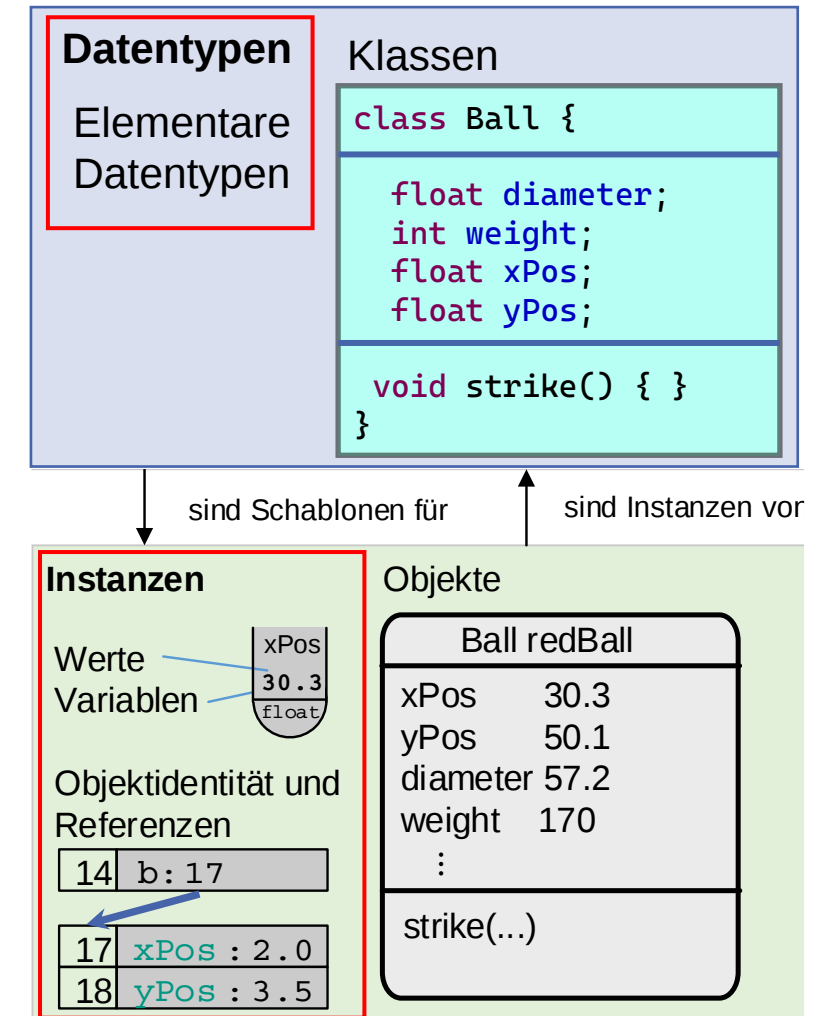
Zusammenfassung letzte Woche

Typen:

- Java kennt 8 elementare Datentypen (**int**, **float**, ...)
- Klassen sind auch Datentypen
- Mit jedem Datentyp sind Operationen bzw. Methoden verbunden

Variablen:

- Variablen sind Namen für Speicherplätze
- Variablen können durch Zuweisung verändert werden
- Objekt-Variablen sind **Referenzen**, d.h. Speicheradressen von Objekt-Identitäten
- Der spezielle Objekt-Referenzwert **null** steht für „kein Objekt“



Lernziele heute

Kontrollstrukturen

- Sie kennen den Unterschied zwischen **Ausdrücken** und **Anweisungen**
 - Sie können Ausdrücke schreiben, die Werte berechnen
 - Sie können Anweisungen schreiben, die den Zustand Ihres Programms ändern
- Sie kennen die **Kontrollfluss-Anweisungen in Java** und können diese in Ihren Programmen verwenden
 - Sie können Fallunterscheidungen mit **if** und **switch** programmieren
 - Sie können Wiederholungen mit den verschiedenen Schleifen-Anweisungen **while**, **do-while** und **for** programmieren und kennen die Unterschiede
 - Sie können Schleifen-Ausführungen mit **break** und **continue** modifizieren

Eingaben:

- Sie können über die Konsole Werte eingeben

Verhalten

- Kontrollstrukturen
- Schleifen
- Verzweigungen

Methodik

Grundlagen

- Ausführen
- Ein-/Ausgabe

Ausdrücke

- **Ausdrücke** (*expressions*) werden in einem Programm verwendet, um vorzunehmende Auswertungen zu beschreiben:
 - auf der rechten Seite von Zuweisungen: `x = Ausdruck;`
 - als Argumente von Methoden: `f(Ausdruck, ..., Ausdruck);`
 - als Rückgabewerte von Funktionen: `return Ausdruck;`
- **Jeder Ausdruck hat einen Typ**

Beispiele:

<code>(3.0 + y) * 4.0</code>	ist ein Ausdruck vom Typ <code>double</code>
<code>new Vector2D(1.0, 2.0)</code>	ist ein Ausdruck vom Typ <code>Vector2D</code>
<code>x == 2</code>	ist ein Ausdruck vom Typ <code>boolean</code>

- Ausdrücke vom Typ **`boolean`** nennt man auch **Bedingungen**

Weitere Ausdrücke in Java

■ Ausdrücke setzen sich wie folgt zusammen:

- Variable (bzw. Attribut), Konstante
- $f(\text{Ausdruck}_1, \dots, \text{Ausdruck}_n)$
- $\text{Ausdruck}_1 \otimes \text{Ausdruck}_2$
- $\ominus \text{Ausdruck}$
- `new Klasse(...)`
- `Variable = Ausdruck`

(Methodenaufruf)

(hier bezeichnet $\otimes$ einen beliebigen binären Operator)

(hier bezeichnet $\ominus$ einen beliebigen unären Operator)

(Zuweisung)

Es gibt in Java noch drei weitere Ausdrücke:

<code>_ instanceof _</code>	Klassen-Zugehörigkeits-Test
<code>_ ? _ : _</code>	Bedingter Ausdruck (wenn-dann-sonst)
<code>(Typ) _</code>	Typumwandlung (<i>type cast</i>)

Beispiele:

<code>p instanceof Vector2D</code>	vom Typ boolean
<code>(x == 3 ? 5 : x+2)</code>	vom Typ int
<code>(int) 2.4</code>	vom Typ int
<code>x = y</code>	vom selben Typ wie x

Anweisungen

■ Anweisungen (statements)

- sind Einheiten der Ausführung („Sätze in der Programmiersprache“)
- verursachen eine Zustandsänderung (in der Regel, bis auf Deklaration)

■ Anweisungen in Java

- Deklarationen lokaler Variablen: `int x;`
- Block-Anweisungen: `{ Anweisung1; ...; Anweisungn; }`
- Return-Anweisungen: `return Ausdruck;      bzw.    return;`
- Expression-Statements: Das sind ausgewählte **Ausdrücke**, die gleichzeitig auch als **Anweisungen** verwendet werden können.

- Inkrement-/Dekrement-Operator

`i++;` vs. `x = y + (i++);`

- Zuweisungen

`x = y;` vs. `x = (y = z);`

- Methodenaufrufe

`f(x,y);` vs. `x = f(x,y);`

- new-Operator

`new Vector2D();` vs. `p = new Vector2D();`

++/-- nur
direkt
vor/hinter
Variablen
erlaubt

- **Control-flow-statements**: heutige Vorlesung

Ausdruck vs. Anweisung

- Unterschiede zwischen **Ausdrücken** und **Anweisungen**:

	Ausdruck	Anweisung
Typisiert	ja	nein
Zweck	„Berechne ...“	„Mache ...“
Effekt	liefert Wert	ändert Zustand
Syntax	ohne „;“	mit „;“

- Ein **Ausdruck** ist immer **Teil einer Anweisung**
- Der Rumpf jeder Methode ist immer eine **Folge von Anweisungen**

Jetzt sind Sie gefragt

■ Bei welchen Java-Fragmenten handelt es sich um gültige Anweisungen?

1. `p = new Vector2D().shift(1.0, 2.0);`
2. `(int)foo(x);`
3. `x = y = z;`
4. `(i++)++;`

Eingaben über die Konsole

```
import java.util.Scanner;
public class TerminalInput{
    public static void main(String[] args) {
        System.out.print("Gib etwas ein:");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        System.out.println("Du hast folgendes eingegeben: "+ input);
    }
}
```

```
...$javac TerminalInput.java
...$java TerminalInput
Gib etwas ein:_
```

■ Vorgefertigte Klasse Scanner

- Importieren mit `import java.util.Scanner;`
- Objekt erstellen mit `new Scanner(System.in);`
- Zeile einlesen mit `scanner.nextLine();`



Eingaben über die Konsole

```
import java.util.Scanner;
public class TerminalInput{
    public static void main(String[] args) {
        System.out.print("Gib etwas ein:");
        Scanner scanner = new Scanner(System.in);
        String input = scanner.nextLine();
        System.out.println("Du hast folgendes eingegeben: "+ input);
    }
}
```

```
...$javac TerminalInput.java
...$java TerminalInput
Gib etwas ein:5
Du hast folgendes eingegeben:5
...$
```

■ Vorgefertigte Klasse Scanner

- Importieren mit `import java.util.Scanner;`
- Objekt erstellen mit `new Scanner(System.in);`
- Zeile einlesen mit `scanner.nextLine();`

Zahlen über die Konsole eingeben

```
import java.util.Scanner;
public class TerminalInput{
    public static void main(String[] args) {
        System.out.print("Gib eine Zahl ein:");
        Scanner scanner = new Scanner(System.in);
        int myNumber = scanner.nextInt();
        System.out.println("Du hast folgendes eingegeben: "+ myNumber);
        System.out.println("Die darauf folgende Zahl ist: "+ (myNumber+1));
    }
}
```

```
...$javac TerminalInput.java
...$java TerminalInput
Gib etwas ein:8
Du hast folgendes eingegeben: 8
Die darauf folgende Zahl ist: 9
...$
```

- `scanner.nextInt();` liest einen Integer-Wert ein
- Achtung: Noch keine Fehlerbehandlung wenn keine Zahl eingegeben
(→ Vorlesung zum Thema Exceptions)

Kontrollfluß-Anweisungen (*control-flow-statements*)

- **if**-Anweisung
- **switch**-Anweisung
- **while**-Anweisung / **do-while**-Anweisung
- **for**-Anweisung
- **break**-Anweisung
- **continue**-Anweisung

Die if-Anweisung

■ Fallunterscheidung: wenn-dann-Prinzip

- Fallunterscheidung ist eine fundamentale Technik der Mathematik und des Programmierens

■ if-Anweisung (*if statement*): Realisierung der Fallunterscheidung in Java

- `if (Bedingung) {
 Anweisungen
}`

- `if (Bedingung) {
 Anweisungen1
} else {
 Anweisungen2
}`

Beispiel:

```
int a = scanner.nextInt();  
int b = scanner.nextInt();  
int max = 0;  
if (a >= b) {  
    max = a;  
} else {  
    max = b;  
}
```

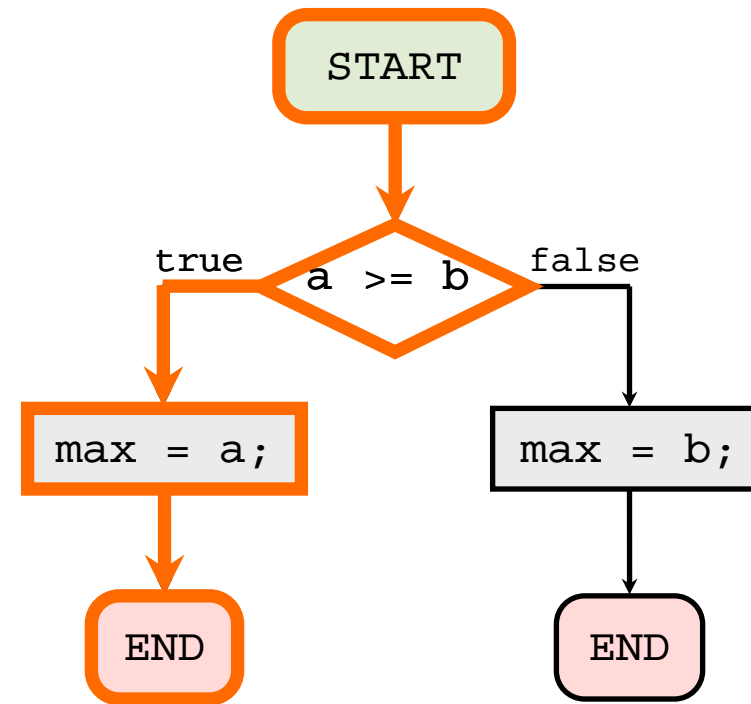
if-Anweisung: Kontrollfluss-Diagramm

Beispielanweisungen:

```
if (a >= b) {  
    max = a;  
} else {  
    max = b;  
}
```

Variablenbelegung:

```
int a = 3;  
int b = 2;
```



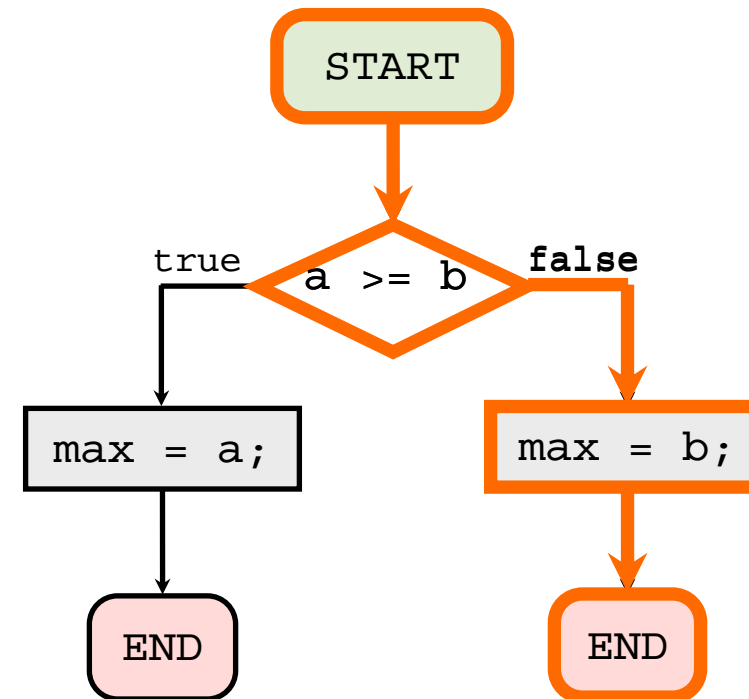
if-Anweisung: Kontrollfluss-Diagramm

Beispielanweisungen:

```
if (a >= b) {  
    max = a;  
} else {  
    max = b;  
}
```

Andere Variablenbelegung:

```
int b = 3;  
int a = 2;
```



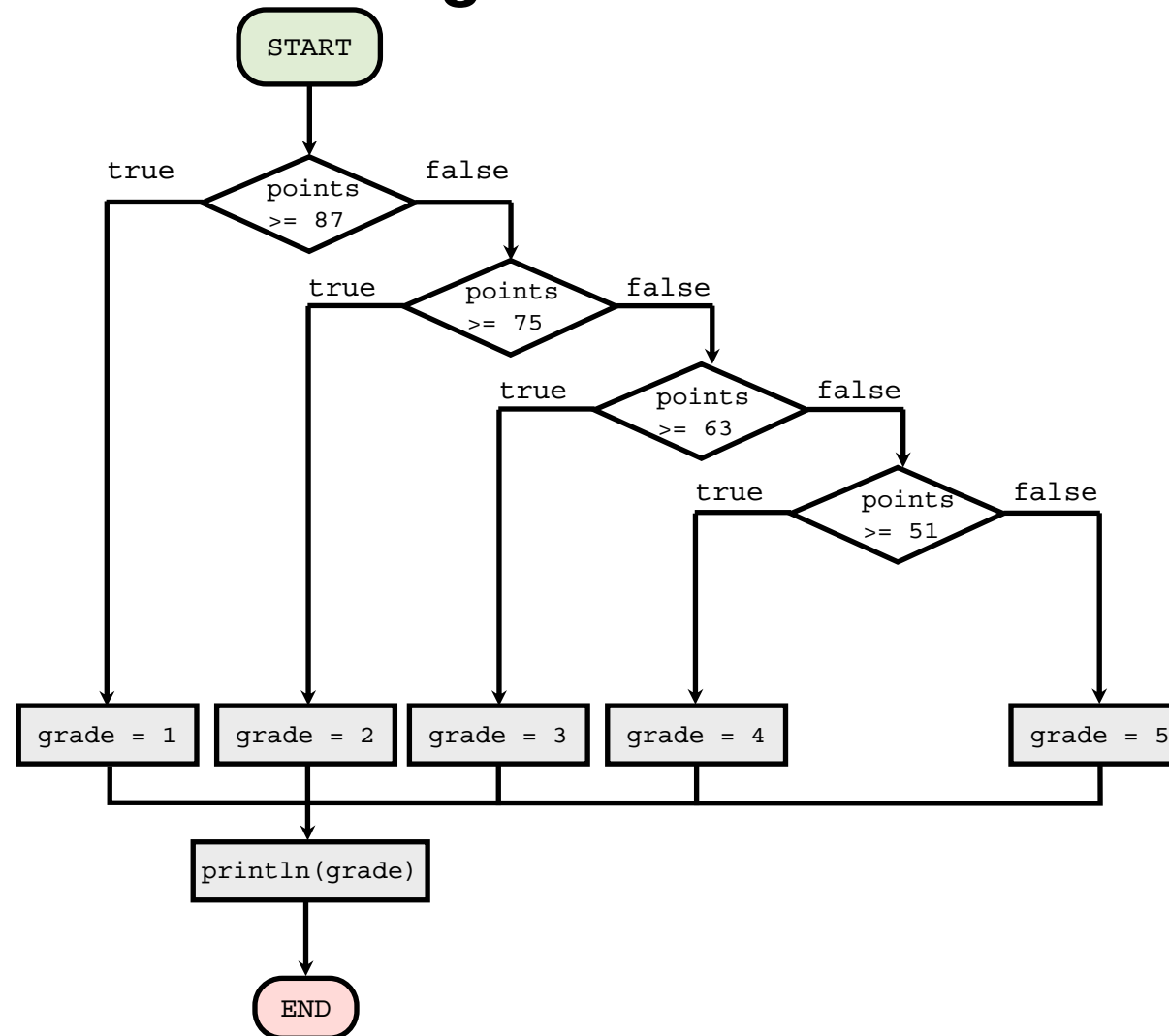
Geschachtelte if-Anweisungen

- Man kann die if-Anweisung auch mehrfach schachteln:

```
int points = 50;
int grade = 0;
if (points >= 87) {
    grade = 1;
} else if (points >= 75) {
    grade = 2;
} else if (points >= 63) {
    grade = 3;
} else if (points >= 51) {
    grade = 4;
} else {
    grade = 5;
}
System.out.println("Note: " + grade);
```

- Nach Möglichkeit am häufigsten auftretende Fälle zuerst behandeln
 - In jedem **else**-Zweig gilt die Negation aller vorangegangenen Bedingungen
- **Auf die Reihenfolge achten!**

Geschachtelte if-Anweisung: Kontrollfluss



Jetzt sind Sie gefragt: if-Anweisung

- Schreiben Sie ein Programm, das für zwei Integer-Eingaben a und b ausgibt, ob a größer, kleiner, oder gleich b ist.

```
Scanner scanner = new Scanner(System.in);
int a = scanner.nextInt();
int b = scanner.nextInt();
if (a > b) {
    System.out.println("a ist größer b");
} else if (a < b) {
    System.out.println("a ist kleiner b");
} else {
    System.out.println("a ist gleich b");
}
```

System.out.println(..) zum Ausgeben eines Strings mit darauffolgendem Zeilenumbruch

Die switch-Anweisung

- Die switch-Anweisung (switch statement) realisiert eine weitere Form der Verzweigung

```
// Konstanten der Klasse
static final char NEW = 'n';
static final char OPEN = 'o';
static final char SAVE = 's';
static final char QUIT = 'q';
...
// Innerhalb einer Methode
char command = ...;
switch (command) {
    case NEW: createNewFile();
    break;
    case OPEN: openFile();
    break;
    case SAVE: saveFile();
    break;
    case QUIT: exitProgram();
    break;
    default: System.out.println("Unbekanntes Kommando: " + command);
    break;
}
```

Die `switch`-Anweisung

■ Syntax:

```
switch (Ausdruck) {  
    case Wert_1: Anweisungen_1;  
    ...  
    case Wert_n: Anweisungen_n;  
    default: Anweisungen;  
}
```

- `Ausdruck` muss dabei den Typ `char`, `byte`, `short`, `int`, `enum` oder `String` haben
- Die Werte nach `case` müssen konstant sein (keine Variablen!)
- Ein „`case Wert`“ legt nur den Einstiegspunkt innerhalb des `switch`-Blocks fest
- Die `break`-Anweisung veranlasst das (sofortige) Verlassen des gesamten `switch`-Blocks
- Ohne `break` werden auch alle Anweisungen der nachfolgenden `case`-Blöcke abgearbeitet
- Nimmt `Ausdruck` keinen der Werte `Wert_1`, ..., `Wert_n` an, so wird der (optionale) `default`-Block abgearbeitet

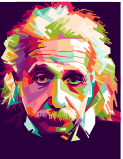
Die switch-Anweisung: Beispiel

Beispiele:

```
int month = 5;
int days = 0;
switch (month) {
    case 1: days = 31; break;
    case 2: days = 28; break;
    case 3: days = 31; break;
    case 4: days = 30; break;
    case 5: days = 31; break;
    case 6: days = 30; break;
    case 7: days = 31; break;
    case 8: days = 31; break;
    case 9: days = 30; break;
    case 10: days = 31; break;
    case 11: days = 30; break;
    case 12: days = 31; break;
}
```

```
int month = 5;
int days = 0;
switch (month) {
    case 2: days = 28; break;
    case 4:
    case 6:
    case 9:
    case 11: days = 30; break;
    default: days = 31; break;
}
```

- Die rechte Variante ist zwar kürzer, aber schlechter lesbar.
Es werden auch Monate größer 12 und kleiner 1 akzeptiert.
→ Fehlerfindung erschwert



Neue switch-
Ausdrücke sind ab
Java 12 ebenfalls
verfügbar

Programmieren von Schleifen

- Schleifen erlauben die wiederholte Ausführung von Anweisungen
- Die Programmierung von Schleifen erfolgt nach dem Muster:
 - Vorbereitung
 - Schleife
 - Nachbereitung

Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;          // Vorbereitung
int sum = 0;
while (i <= b) {    // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3	11	27



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3	11	27
4		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3	11	27
4		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3	11	27
4	12	38



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;
int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3	11	27
4	12	38
5		



Die while-Schleife

■ Die **while**-Anweisung (*while statement*) realisiert eine sich wiederholende Ausführung.

■ **Beispiel:** Berechne

$$\sum_{i=a}^b i$$

```
int a = 8;
int b = 11;

int i = a;           // Vorbereitung
int sum = 0;
while (i <= b) {     // Schleife
    sum = sum + i;
    i = i + 1;
}
```

Iteration	i	sum
-	8	0
1	9	8
2	10	17
3	11	27
4	12	38
5		

→ Ergebnis: 38



Jetzt sind Sie gefragt: while-Schleife

- Schreiben Sie ein Programm, das für eine Integer-Eingabe p die Zahlen von 1 bis p ausgibt
- Gewünschte Ausgabe für p = 5:

1 2 3 4 5

```
Scanner scanner = new Scanner(System.in);
int p = scanner.nextInt();
int i = 1; // Vorbereitung
while (i <= p){ // Schleife
    System.out.print(i + " ");
    i++;
}
```

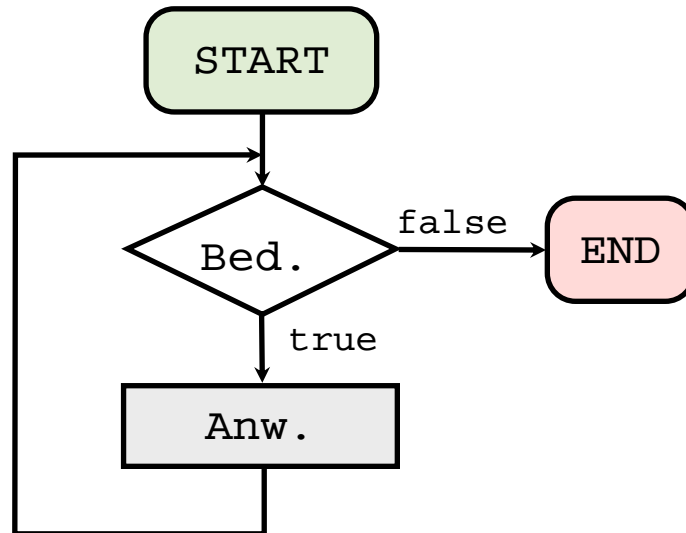
System.out.print(..) zum Ausgeben eines Strings ohne neue Zeile

(do-)while-Schleife: Syntax

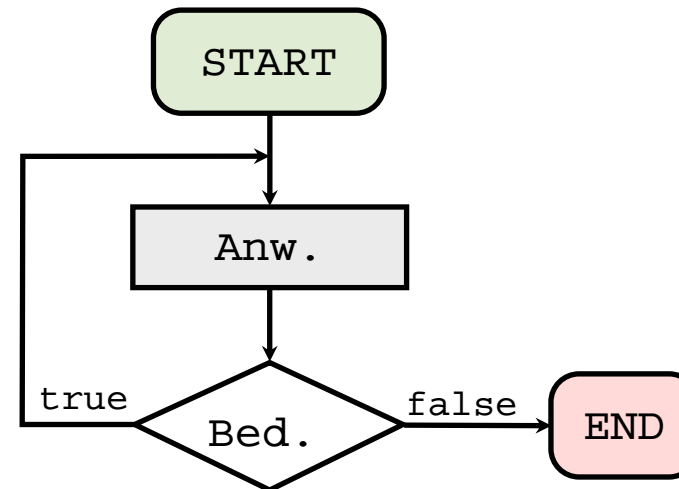
```
■ while (Bedingung) { Anweisungen }  
do { Anweisungen } while (Bedingung);
```

■ Kontrollfluss:

while:



do-while:



Die for-Schleife

- Die **for**-Schleife (*for statement*) ist eine Zählschleife

- **Syntax:**

```
for (Initialisierung; Bedingung; Schritt) { Anweisungen }
```

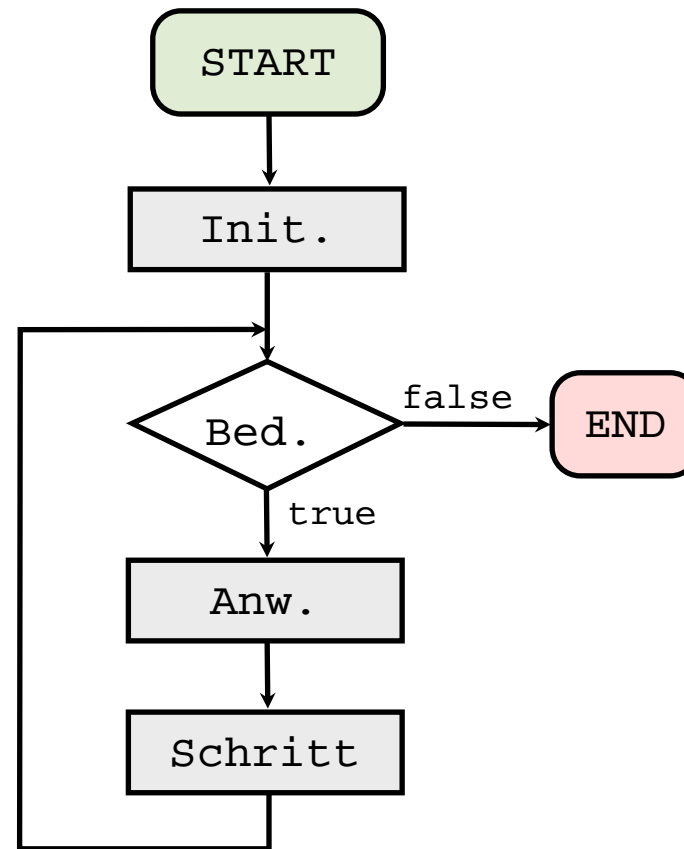
- **Initialisierung** ist eine Anweisung, die einmalig am Anfang der Schleife ausgeführt wird
- Vor jedem Schleifendurchlauf wird geprüft, ob **Bedingung** wahr ist
- **Schritt** ist eine Anweisung, die am Ende jedes Schleifendurchlaufs ausgeführt wird (nach den *Anweisungen*)

Beispiel

```
int a = 8;  
int b = 11;  
int sum = 0; // Vorbereitung  
for (int i = a; i <= b; i++) { // Schleife  
    sum = sum + i;  
}
```

Die for-Schleife: Kontrollfluss

```
for (Initialisierung; Bedingung; Schritt) { Anweisungen }
```



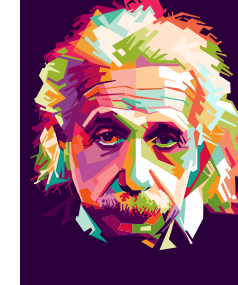
Unterschied for- und while-Schleife

■ Eigentlich:

```
for (ich weiss wie oft) {...}  
while (Anzahl Wiederholungen hängt ab von Ausführung des Schleifenrumpfes) {...}
```

■ In C (und dann C++ und Java) schiefgelaufen:

```
initialisierung;  
while (abbruchbedingung) {  
    anweisungen_1; anweisungen_2;  
}  
  
for (initialisierung; abbruchbedingung; anweisungen_2) {  
    anweisungen_1;  
}
```



Warum ist dieser Unterschied wichtig?

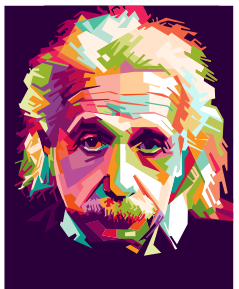
- **for** (ich weiss wie oft (n-mal)) {Schleifenrumpf}

- Kurzform für n-mal: Schleifenrumpf;
also:
Schleifenrumpf; Schleifenrumpf; ... Schleifenrumpf;
- Ich weiß sicher, wie oft es ausgeführt wird, da eigentlich wie sequenzielles Programm ohne Schleife.

Vorschau: Ab Java 8
verfügbar mit sog.
for-each-Schleifen,
→ Vorlesung zu Arrays

- **while** (Anzahl Wiederholungen hängt ab von Ausführung des Schleifenrumpfes) {...}

- Ich weiß vor Beginn der Ausführung der Schleifenrumpfe nicht unbedingt, wie oft es ausgeführt wird.
- Halteproblem ist unentscheidbar (Kurt Gödel, Alan Turing)
- Es gibt keinen Algorithmus, der entscheidet, ob ein Programm mit while-Schleifen terminiert.



Beispiel: Collatz-Funktion

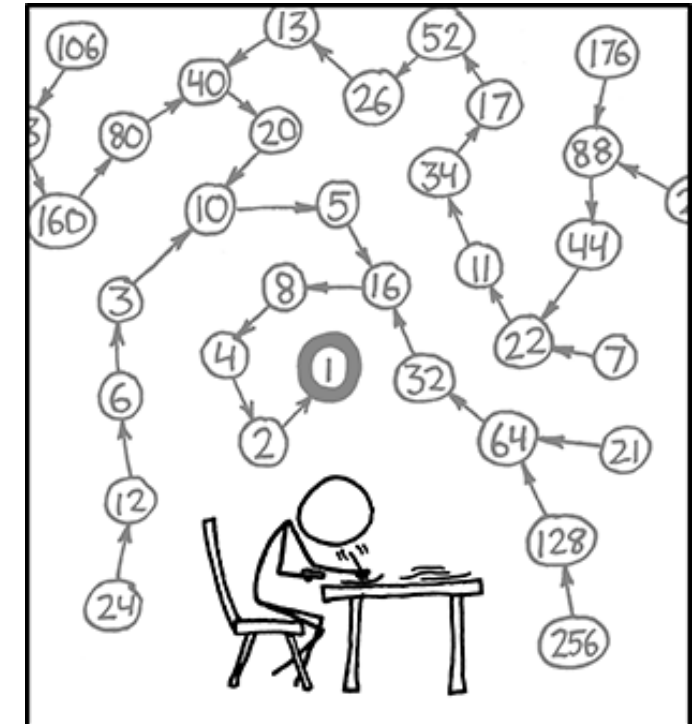
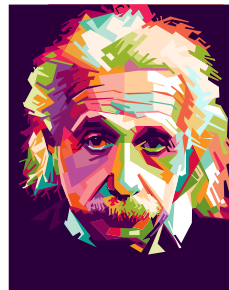
vgl. <https://de.wikipedia.org/wiki/Collatz-Problem>

```
n = beliebige natürlich Zahl;
while (n != 1) {
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = n * 3 + 1;
    }
}
```

→ Hier ist im Voraus unbekannt, wie häufig die Schleife ausgeführt wird

Lothar Collatz (1910-1990),

- dt. Mathematiker,
- Habilitation 1937 in Karlsruhe



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

[XKCD.COM]

Geschachtelte Schleife

- Gesucht: Ein Programm, das für zwei Integer-Eingaben p und m dann p Zeilen auf der Konsole ausgibt, die jeweils die Zahlen von 1 bis m enthalten.
- Gewünschte Ausgabe für $p = 2$ und $m = 5$:

```
1 2 3 4 5
1 2 3 4 5
```

- Tipp
 - Zu Beginn natürlich Werte einlesen.
 - Dann zunächst eine Schleife schreiben, die eine solche Zeile ausgibt
 - Dann überlegen: Was „drumherum“ schreiben, damit diese Zeile p -mal ausgegeben wird?

Geschachtelte Schleife

- Gesucht: Ein Programm, das für zwei Integer-Eingaben p und m dann p Zeilen auf der Konsole ausgibt, die jeweils die Zahlen von 1 bis m enthalten.
- Gewünschte Ausgabe für p = 2 und m = 5:

```
1 2 3 4 5
1 2 3 4 5
```

```
int p = scanner.nextInt();
int m = scanner.nextInt();

for (int i = 1; i <= p; i++){           // Vorbereitung, Bedingung, Schritt außen

    for (int j = 1; j <= m; j++){       // Vorbereitung, Bedingung, Schritt innen
        System.out.print(j + " ");     // Anweisungen innen
    }

    System.out.println("");            // Anweisungen außen
}
```

Die break-Anweisung

- Manchmal möchte man eine Schleife verlassen, *bevor* alle Schleifendurchläufe abgearbeitet wurden
- „**break**;“ veranlasst das sofortige Verlassen der innersten Schleife

Beispiel: Berechne die Summe eingelesener Zahlen bis zur Eingabe „0“

```
int i, sum = 0;
while (true) {
    i = scanner.nextInt();
    if (i == 0) {
        break;
    }
    sum += i;
}
System.out.println("sum = " + sum);
```

- **break** sollte nur sparsam und gezielt eingesetzt werden, so dass der Programmcode übersichtlich und verständlich bleibt

Die `continue`-Anweisung

- `continue` bricht die Ausführung der aktuellen Schleifeniteration ab und springt direkt zur nächsten Iteration
 - Die Schleifenbedingung wird dabei geprüft
 - Bei einer `for`-Schleife wird zuvor auch noch die *Schritt*-Anweisung ausgeführt

Beispiel: Zähle die Anzahl der Einsen in der Eingabe

```
int i, numOnes = 0;
while (true) {
    i = scanner.nextInt();
    if (i == 0) {
        break;
    } else if (i != 1) {
        continue;
    }
    numOnes++;
}
System.out.println("number of ones = " + numOnes);
```

Jetzt sind Sie gefragt

- **Aufgabe:** Lese Zahlen ein, solange diese nicht negativ sind, und berechne von jeder Zahl die Quadratwurzel.
- Welche der drei angegebenen Lösungen würden Sie bevorzugen?

Lösung 1:

```
do {  
    a = scanner.nextDouble();  
    if (a >= 0) {  
        System.out.println(Math.sqrt(a));  
    }  
} while (a >= 0);
```

Lösung 2:

```
while (true) {  
    a = scanner.nextDouble();  
    if (a < 0) {  
        break;  
    }  
    System.out.println(Math.sqrt(a));  
}
```

Lösung 3:

```
a = scanner.nextDouble();  
while (a >= 0) {  
    System.out.println(Math.sqrt(a));  
    a = scanner.nextDouble();  
}
```

Wie könnte man die Schleife noch aufbauen?

Schleifenorganisation

- **Aufgabe:** Lese Zahlen ein, solange diese nicht negativ sind, und berechne von jeder Zahl die Quadratwurzel

- **Lösung 1:**

```
do {  
    a = scanner.nextDouble();  
    if (a >= 0) {  
        System.out.println(Math.sqrt(a));  
    }  
} while (a >= 0);
```

- **Weniger gut:**

- Redundante Kopie der Abbruchbedingung $a \geq 0$

Schleifenorganisation

- **Aufgabe:** Lese Zahlen ein, solange diese nicht negativ sind, und berechne von jeder Zahl die Quadratwurzel
- **Lösung 2:**

```
while (true) {  
    a = scanner.nextDouble();  
    if (a < 0) {  
        break;  
    }  
    System.out.println(Math.sqrt(a));  
}
```

- **Vorteil:** Kein redundanter Code
- **Weniger gut:**
 - **break** / Endlosschleife

Schleifenorganisation

- **Aufgabe:** Lese Zahlen ein, solange diese nicht negativ sind, und berechne von jeder Zahl die Quadratwurzel

- **Lösung 3:**

```
a = scanner.nextDouble();  
while (a >= 0) {  
    System.out.println(Math.sqrt(a));  
    a = scanner.nextDouble();  
}
```

- **Vorteil:** Kein **break**
- **Weniger gut:**
 - Erste Eingabe außerhalb der Schleife, Einlesen an zwei Stellen
- Wie könnte man die Schleife noch aufbauen?

Schleifenorganisation

- **Aufgabe:** Lese Zahlen ein, solange diese nicht negativ sind, und berechne von jeder Zahl die Quadratwurzel

- **Lösung 4:**

```
while ((a = scanner.nextDouble()) >= 0) {  
    System.out.println(Math.sqrt(a));  
}
```

- **Vorteil:** Kein redundanter Code und kein **break**
- **Weniger gut:**
 - Schwieriger zu lesen: Womöglich nicht gleich ersichtlich, dass Bedingung auch Zuweisung enthält

Zusammenfassung

Ausdrücke, Anweisungen:

- Ausdrücke berechnen einen Wert
- Anweisungen ändern den **Zustand** des Programms

Eingaben:

- Klasse Scanner erlaubt Eingaben über die Konsole

Kontrollfluss-Anweisungen:

- Fallunterscheidung: **if**, **switch**
- Schleifen: **while**, **do-while**, **for**
 - Wiederholtes Ausführen von Anweisungen
 - Schleifen-Ausführung kann mit **break** und **continue** modifiziert werden

Literaturhinweis – Weiterlesen

- Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger „Grundkurs Programmieren in Java“, 7. Auflage, 2014 (mit Java 8), Hansa-Verlag
 - Abschnitt „Anweisungen und Ablaufsteuerungen“