HSLU Hochschule Luzern

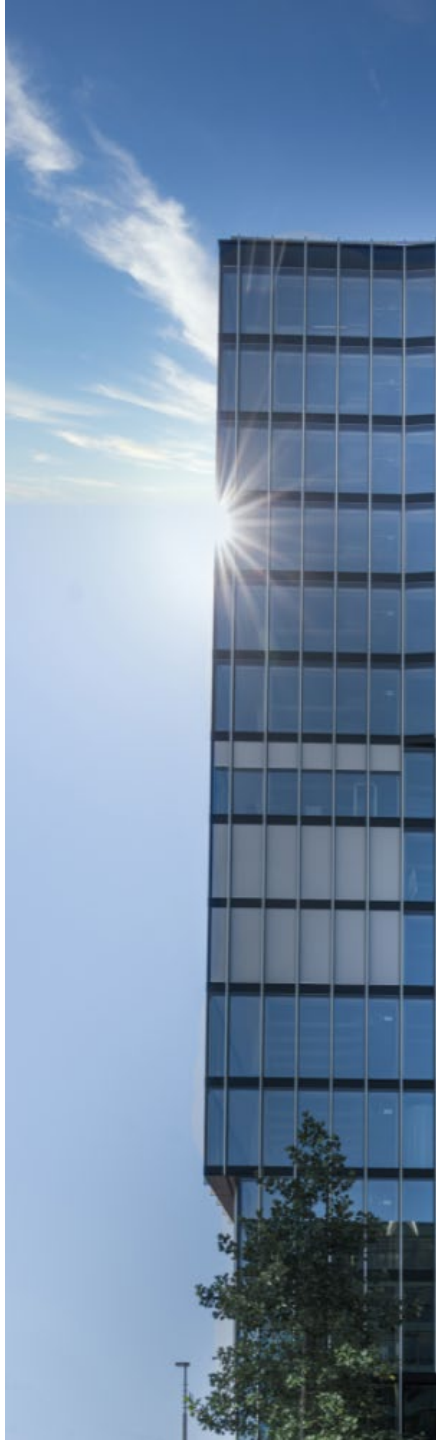
Objektorientierte Programmierung

Iterationen

Wiederholungen mit Schleifen

Roland Gisler

FH Zentralschweiz



Inhalt

- Was sind Iterationen – Einführung
- Schleife mit Eintrittstest: `while`
- Schleife mit Austrittstest: `do-while`
- `for`-Schleife
- foreach-Schleife
- Empfehlungen und Hinweise
- Zusammenfassung

Lernziele

- Sie kennen die verschiedenen Schleifen-Typen von Java mit ihren individuellen Eigenschaften.
- Sie können für die jeweilige Situation den geeigneten Schleifentyp auswählen und implementieren.
- Sie sind in der Lage, sichere Abbruchbedingungen zu Formulieren.

Iterationen (Schleifen) - Einführung

Einführung - Iterationen

- Viele Aufgaben sind repetitiver (wiederholender) Natur:
  - Eine Menge von Objekten bearbeiten / einlesen / ausgeben.
  - In (zeitlichen) Intervallen bestimmte Aufgaben wiederholen.
- Dafür sind Computer prädestiniert (vgl. "Arbeitstier")!
- Die Umsetzung solcher Wiederholungen nennt man **Iteration** oder auch einfach (Programm-)Schleife.
- Im Normalfall wird eine Schleife nicht endlos, sondern abhängig von einer logischen Bedingung ausgeführt.
- Man unterscheidet grundlegend zwischen Schleifenkonstrukten
  - mit **Eingang**s- oder **Ausgang**stest.
  - mit **bekannter** oder **unbekannter** Anzahl Iterationen.

Einführung Schleifen / Iterationen

- Java kennt drei verschiedene Schleifen-Anweisungen für unterschiedliche Zwecke und Nutzungen:
  - **`while`**-Schleife – Schleife mit Eingangstest.
  - **`do-while`**-Schleife – Schleife mit Ausgangstest.
  - **`for`**-Schleife – Schleife mit Eingangstest.
- Grundsätzlich kann **jede** Iteration mit **jeder** Art von Schleife formuliert werden!
  - Die drei Schleifentypen können sich gegenseitig ersetzen.
  - Sogar eine rekursive Lösung ist äquivalent!
- Die wohlüberlegte Auswahl des Schleifentyps erhöht aber einerseits die Lesbarkeit und Verständlichkeit, und reduziert andererseits den Programmieraufwand.

`while`-Schleife

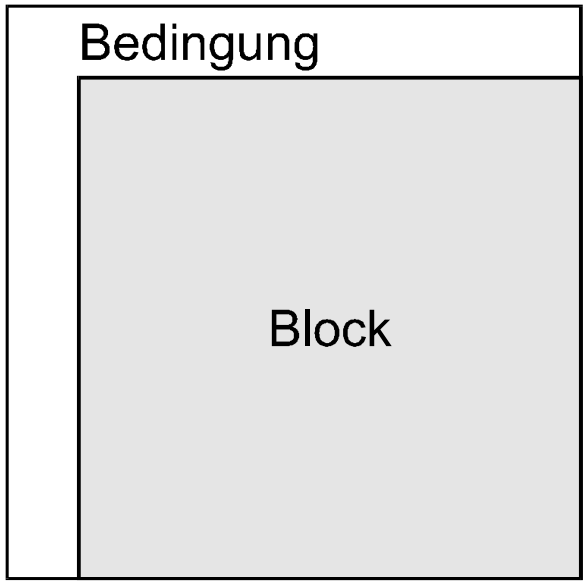
`while`-Schleife: Syntax

```
while (<expression>) {

    <statements> // Schleifenkörper:
                 // Ausführung so lange Bedingung true.
}
```

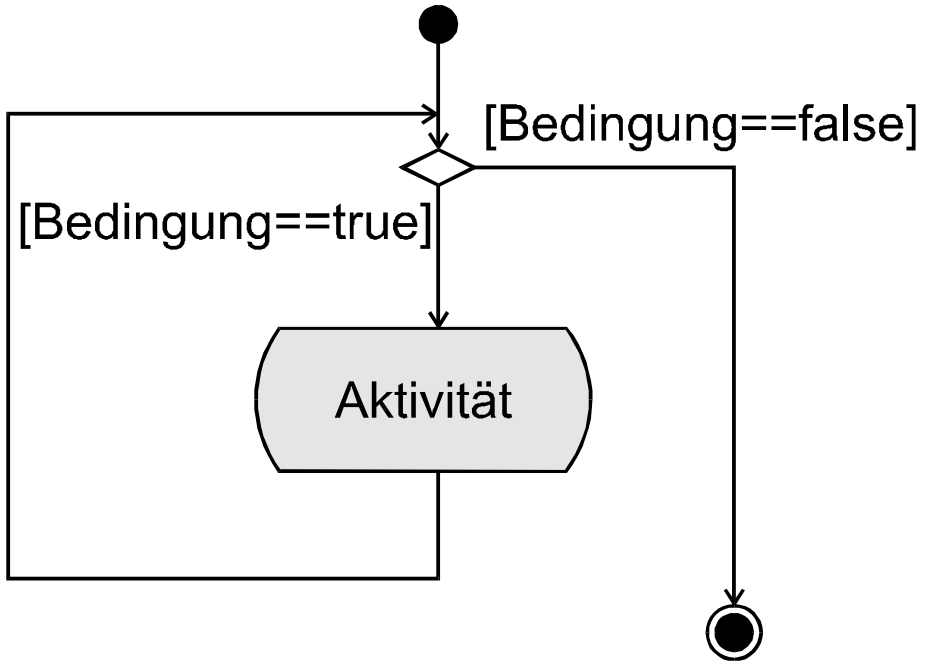
- Die Bedingung (expression) muss ein bool'sches Resultat ergeben (analog zur Bedingung bei `if`-Statements).
- Häufig werden in der Bedingung Variablen genutzt, deren Inhalt innerhalb des Schleifenkörpers verändert wird.
- Evaluiert die Bedingung beim Eintritt in die Schleife bereits auf `false`, wird der Schleifenkörper **nie** ausgeführt.

`while`-Schleife: Formal

- Ablauf einer **`while`**-Schleife:



Struktogramm



Aktivitätsdiagramm (UML)

`while`-Schleife: Beispiel

- Beispiel: "Zahlen von der Konsole (Terminal) einlesen und diese laufend summieren bis die Zahl '`0`' eingegeben wird."

```
int input = scan.nextInt();

int summe = 0;
while (input != 0) {            // Bedingung
    summe = summe + input;      // Schleifenkörper
    input = scan.nextInt();
}

System.out.println(summe);
```

`while`-Schleife: Vereinfachtes Beispiel

- Das Beispiel von oben lässt sich auch kürzer formulieren:

```
int input = 0;
int summe = 0;
while ((input = scan.nextInt()) != 0) {
    summe += input;
}
System.out.println(summe);
```

- Erkenntnis: Bedingungen dürfen auch Zuweisungen enthalten!
  - Eher verpönt, weil schnell mit **==** (Gleichheit) verwechselt.
- Nebenbei: Verkürzte Schreibweise der Addition (**+=**).
- **Hinweis**: Es braucht einiges an Erfahrung, um einen guten Kompromiss zwischen kompaktem Code und trotzdem leichter Verständlichkeit zu erreichen.

`while`-Schleife: Charakteristik

- Die Anzahl Schleifendurchläufe ist in der Regel (mindestens zum Zeitpunkt der Implementation) **nicht** bekannt,
  - z.B. abhängig von Benutzereingaben, Datenmenge etc.
- Schleifenkörper wird ausgeführt, solange die Bedingung `true` ist.
- Die Bedingung wir **vor** jeder weiteren Ausführung des Schleifenkörpers geprüft.
- Da es sich also um eine Schleife mit Eintrittstest handelt, gibt es eventuell **keinen** Durchlauf des Schleifenkörpers.

`do-while`-Schleife

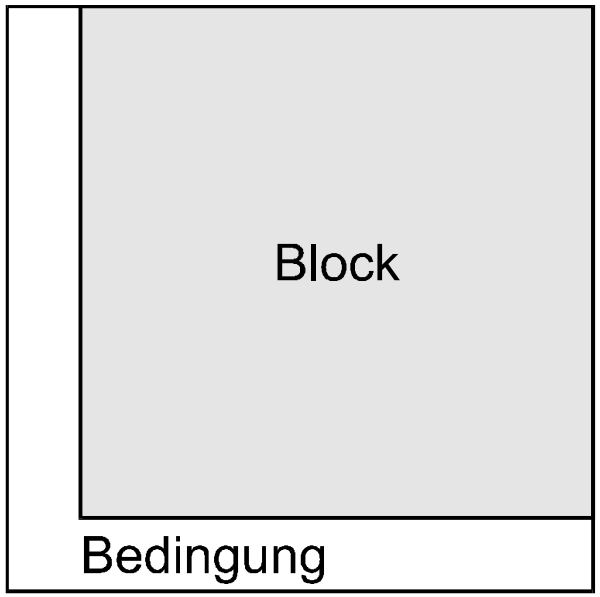
do-while-Schleife: Syntax

```
do {

    <statements> // Schleifenkörper:
                 // Ausführung so lange Bedingung true.
} while (<expression>);
```

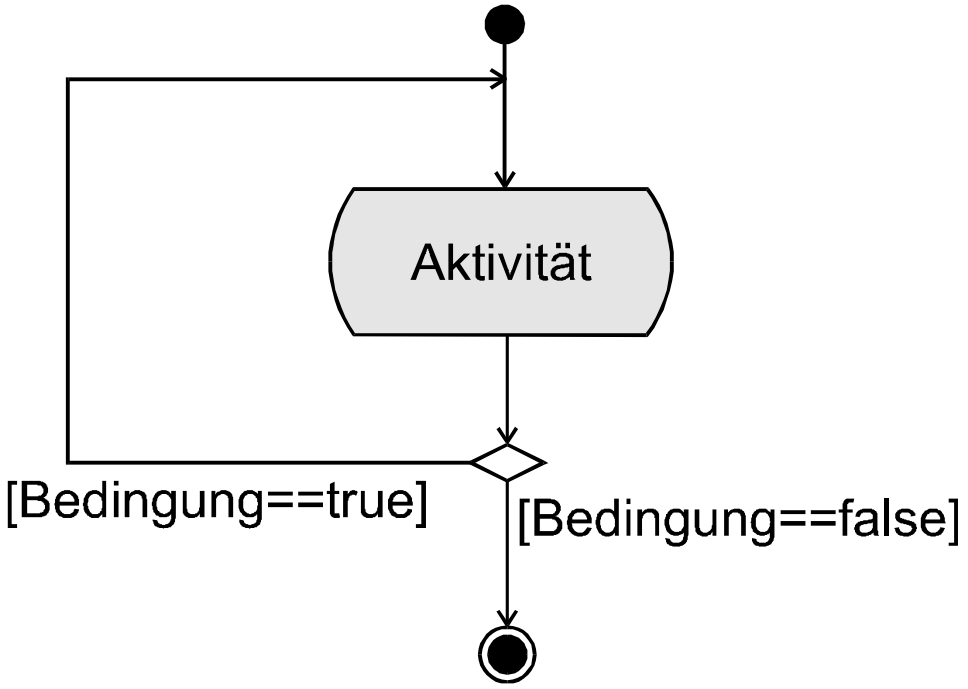
- Die Bedingung (expression) muss ein bool'sches Resultat ergeben (analog zur Bedingung bei **if**-Statements).
- Häufig werden in der Bedingung Variablen genutzt, deren Inhalt innerhalb des Schleifenkörpers verändert werden.
- Vor der ersten Evaluation der Bedingung wird der Schleifenkörper auf jeden Fall **einmal** ausgeführt!

`do-while`-Schleife: Formal

- Ablauf einer **`do-while`**-Schleife:



Struktogramm



Aktivitätsdiagramm (UML)

do-while-Schleife: Beispiel – Variante 1

- Beispiel: Mit drei Würfel so lange würfeln bis alle **6** Punkte zeigen.

```
int tryCount = 0;
int dice1, dice2, dice3;

do {
    dice1 = (int) (Math.random() * 6) + 1;
    dice2 = (int) (Math.random() * 6) + 1;
    dice3 = (int) (Math.random() * 6) + 1;
    tryCount++;
} while (!((dice1 == 6) && (dice2 == 6) && (dice3 == 6)));

System.out.println("Required " + tryCount + " tries.");
```

- Nebenbei: Ist diese Bedingung gut lesbar bzw. leicht verständlich?

do-while-Schleife: Beispiel – Variante 2

- Bedingung nach dem «Gesetz von De Morgan» umgeformt:

```
int tryCount = 0;
int dice1, dice2, dice3;

do {
    dice1 = (int) (Math.random() * 6) + 1;
    dice2 = (int) (Math.random() * 6) + 1;
    dice3 = (int) (Math.random() * 6) + 1;
    tryCount++;
} while ((dice1 != 6) || (dice2 != 6) || (dice3 != 6));

System.out.println("Required " + tryCount + " tries.");
```

- Definitiv die Bessere, weil viel leichter verständliche Variante!
- Nebenbei: Gäbe es vielleicht noch einfachere Varianten?

`do-while`-Schleife: Charakteristik

- Die Anzahl Schleifendurchläufe ist in der Regel (mindestens zum Zeitpunkt der Implementation) nicht bekannt,
  - z.B. abhängig von Benutzereingaben, Datenmenge etc.
- Schleifenkörper wird ausgeführt, solange die Bedingung `true` ist.
- Die Bedingung wir erst **nach** jeder Ausführung des Schleifenkörpers geprüft, somit erfolgt immer mindestens **ein** Durchlauf!

`for`-Schleife

`for`-Schleife: Syntax

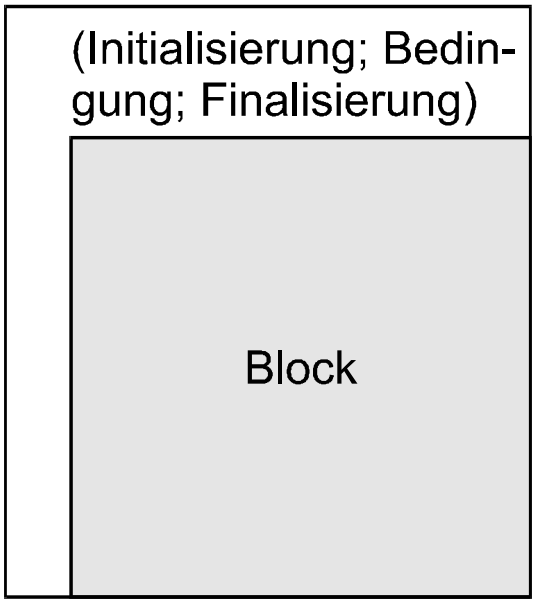
```
for (<initialisierung>; <expression>; <finalisierung>) {

    <statements>    // Schleifenkörper:
                    // Ausführung so lange Bedingung true.
}
```

- Die Initialisierung dient zur Deklaration (optional, aber empfohlen) und Initialisierung der Schleifenvariable(n):
  - Diese sind, bei Deklaration an Ort, nur **innerhalb** des Schleifenblocks sichtbar!
- Die Bedingung muss ein bool'sches Resultat ergeben.
- Die Finalisierung dient zur Veränderung der Schleifenvariable(n).

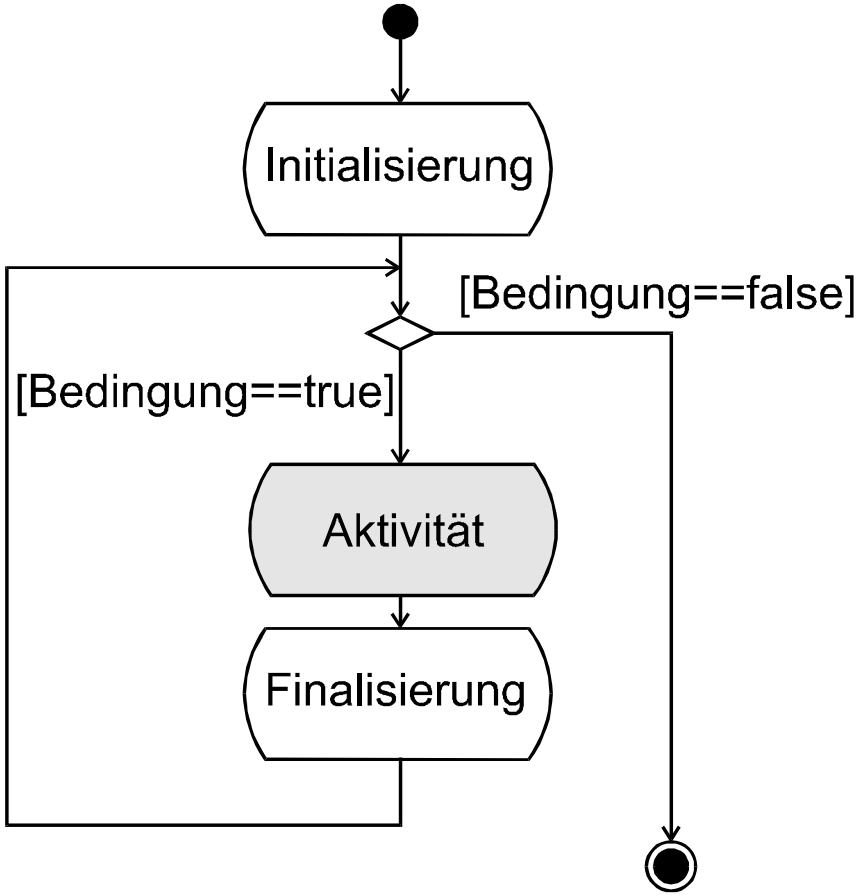
➔Die `for`-Schleife ist ideal für einfache, zählende Schleifen!

`for`-Schleife: Formal

- Ablauf einer `for`-Schleife:



Struktogramm



Aktivitätsdiagramm (UML)

`for`-Schleife: Beispiel

- Beispiel: "Ausgabe der Zahlen von **`1`** bis **`10`**."

```
for (int i = 1; i <= 10; i++) {

    System.out.println("Wert: " + i);
}
// Variable i ab hier nicht mehr existent!
```

- Die **`for`**-Schleifen erlauben speziell für einfache, zählende Schleifen eine sehr kompakte und einfache Schreibweise.
- Da die Schleifenvariablen nur innerhalb des Blockes sichtbar sind (scope), können die Bezeichner problemlos wiederverwendet werden.

`for`-Schleife: Charakteristik

- Anzahl Schleifendurchläufe bekannt oder im Voraus (eventuell auch erst zur Laufzeit) berechenbar.
  - Schleifensteuerung meist durch eine einfache Zählvariable.
  - Zählvariable(n) nur innerhalb der Schleife sichtbar.
  - Sonst verpönte kurze Namen wie **`i`**, **`j`**, **`k`** etc. sind hier für die Zählvariablen durchaus erlaubt und üblich!
- Schleife mit Eingangstest:
  Es gibt eventuell gar keinen Durchlauf des Schleifenkörpers!
- Sowohl Initialisierung als auch Finalisierung dürfen auch mehrere Anweisungen beinhalten.
  - Spezielle Syntax: Anweisungen dann durch **Kommas** getrennt!
- Die Finalisierung ist sehr häufig eine einfache Zählanweisung.

foreach-Schleife

foreach-Schleife – kurzer Ausblick

- Seit Version 5 kennt Java auch eine foreach-Schleife.
- Das ist aber «nur» eine syntaktische Vereinfachung, welche z.B. bei ➔Collections (Datensammlungen) und ➔Arrays eingesetzt werden kann.
- Dafür gibt es in Java aber **kein** explizites neues Schlüsselwort (wie z.B. **`foreach`** in C#), sondern es wird auch **`for`** verwendet.
- Beispiel:

```
for (final Temperatur t : list) {
    t.doSomething();
}
```

➔ Werden wir später im Rahmen von Datenstrukturen behandeln.

Hinweise zu allen Schleifen

Robuste Formulierung der Schleifenbedingung

- Speziell bei Schleifenbedingungen sollte darauf geachtet werden, dass diese **robust** formuliert werden, denn tritt eine Bedingung **nicht** ein, resultiert eine Endlos-Schleife!
- Wenn möglich somit **nicht** auf absolute Werte (z.B. `x == 10`) vergleichen, **sondern** auf Bereiche (z.B. `x >= 10`).
- Darauf ist speziell bei Fliesskommatypen (`float` und `double`) zu achten, da **immer** Rundungsfehler auftreten können!
- Beispiel: Statt des erwarteten Resultates `2.0f`, könnte durch Rundungsfehler auch `1.99999f` auch `2.000001f` auftreten:
  - Eine Bedingung wie `(x == 2.0f)` liefert dann aber `false`!
  - Eine robust implementierte Bedingung wie `(x >= 1.999999f)` liefert hingegen korrekt ein `true`.

Empfehlungen - Verwendung von Schleifen

- In Java sind alle drei Schleifen-Anweisungen gleich mächtig!
  - Grundsätzlich austauschbar.
- Guter Programmierstil: Ziel ist einfache und gute Verständlichkeit:
  - Fallabhängig die adäquate Schleifen-Anweisung wählen.
  - Schleifen-Rumpf immer als Block **{…}** implementieren, auch im Falle einer einzelnen Anweisung (analog **`if-else`**).
  - Ggf. grössere (Teil-)Bedingungen in Methoden auslagern.
- Laufzeit beachten: Beim Aufruf von Methoden im Block oder der Bedingung werden diese **n**-fach ausgeführt!
  - Gegebenenfalls Werte zwischenspeichern (caching).
- **Wichtig**: Schleifen sollten nicht nur eine korrekte, sondern auch eine **robuste** Abbruchbedingung besitzen (siehe vorherige Folie)!

Zusätzliche Möglichkeiten: `break` und `continue`

- **`break`**-Anweisung:
  - Innerhalb eines Schleifen-Blocks ausgeführt, bewirkt sie das **sofortige** Beenden der Schleife.
  - Ungeachtet der Schleifenbedingung!
- **`continue`**-Anweisung:
  - Innerhalb eines Schleifen-Blocks ausgeführt, bewirkt Sie den Sprung an das Ende des Schleifenblockes.
  - Es folgt die sofortige Prüfung der Bedingung, abhängig davon wird die Scheife fortgesetzt oder nicht.
- **Hinweis**: Sowohl **`break`** als auch **`continue`** Anweisungen können (und sollten) meist ohne grossen Aufwand mit einem **`if`**-Statement ersetzt und somit vermieden werden.

➔ Selten empfohlen, entsprechen quasi einem **`goto`**!

Zusammenfassung

- Schleifen mit Eingangstest: `while` und `for`
- Schleife mit Ausgangstest: `do-while`
- `for` Schleife vorwiegend für zählende Schleifen, sehr kompakt.
- Bei Bedingungen auf Bereiche (statt absoluter Werte) prüfen, damit bei unerwarteten Fehlern trotzdem ein Abbruch, und keine Endlosschleife resultiert!
  - bei Ganzahlen, aber speziell auch bei Fliesskommatypen.
- Bei Schleifen mit vielen Durchläufen darauf achten, dass alle Aktivitäten (in Bedingung, Finalisierung und Schleifenkörper) nicht unnötig Ressourcen verschwenden ➔Laufzeit!
- `break` und `continue` Anweisungen wenn möglich vermeiden!

Ach ja – bitte dran denken…

- Auch wenn im Internet sehr viele danach fragen, oder auch davon geschrieben wird - es ändert nichts an der Tatsache:

Es gibt keine `if`-Schleife,
sondern nur `if`-Abfragen!

http://if-schleife.de/

Fragen?