# Inhaltsverzeichnis



# 1. Kontrollstrukturen

## 1.1. Konditionalsausdrücke

### 1.1.1. `if - else-if - else`

Java kennt folgende bedingte Anweisungen, um bestimmten Code auszuführen:

- `if`, um einen Codeblock anzugeben, der ausgeführt werden soll, wenn eine angegebene Bedingung *wahr* ist
- `else`, um einen Codeblock anzugeben, der ausgeführt werden soll, wenn dieselbe Bedingung *falsch* ist
- `else if`, um eine neue Bedingung anzugeben, die getestet werden soll, wenn die erste Bedingung *falsch* ist

**Beispiel**:

```
if (index == 0) {
  // wenn index == 0 ist wahr
}
else if (index == 1) {
  // wenn index == 0 falsch ist und index == 1 ist wahr
}
else {
  // alle anderen Fälle, also beide zuvor falsch sind
}
```

Für einfache `if-else` Ausdrücke gibt es auch eine Kurzschreibweise, die sogenannten `ternären` Operatoren:

```
var = (condition) ? expressionTrue :  expressionFalse;
```

Zum Beispiel:

```
String result = (index == 0) ? "passt scho'" : "nix da";
```

### 1.1.2. `switch case`

Das Schlüsselwort `switch` wird benutzt, um viele alternative Codeblöcke anzugeben, die ausgeführt werden sollen

Anstatt also viele `if..else`-Anweisungen zu schreiben, können `switch`-Anweisung verwendet werden. Die switch-Anweisung wählt einen von vielen auszuführenden Codeblöcken aus:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default: ①
    // code block
}
```

① Die Schlüsselwörter `break` und `default` sind optional.

Mit der Version Java SE 17 wurde `pattern matching` für Switch-Ausdrücke und -Anweisungen (JEP 406) als Vorschaufunktion eingeführt. Dieses bietet mehr Flexibilität bei der Definition von Bedingungen für Switch-Fälle.

Zusätzlich zu den `case` Labels, die jetzt Pattern enthalten können, ist der Ausdruck nicht mehr auf nur wenige Typen beschränkt. Bei Java < 17 unterstützten Switch-Cases nur das einfache Testen eines Case-Selektors, der genau mit einem konstanten Wert übereinstimmen musste (und dies mussten entweder Number, String oder eine Konstante sein).

Siehe auch → https://www.baeldung.com/java-switch-pattern-matching

```
final String b = "B";
switch (args[0]) {
    case "A" -> System.out.println("Parameter is A");
    case b   -> System.out.println("Parameter is b");
    default  -> System.out.println("Parameter is unknown");
};
```

## 1.2. Schleifen

### 1.2.1. for (each)

Eine for-Schleife ist eine Kontrollstruktur, die es ermöglicht, bestimmte Vorgänge durch Inkrementieren und Auswerten eines Schleifenzählers zu wiederholen.

Vor der ersten Iteration wird der Schleifenzähler initialisiert, dann wird die Bedingungsauswertung durchgeführt, gefolgt von der Schrittdefinition (normalerweise eine einfache Inkrementierung).

Die **Syntax** der for-Schleife ist:

```
for (Initialization; Boolean-Expression; Step) {
   statement;
}
```

Am **Beispiel**:

```
for (int i = 0; i < 5; i++) {
   System.out.println("simple for loop: i = " + i);
}
```

Die in for-Anweisungen verwendete Initialisierung, der boolesche Ausdruck und der Inkrement-Schritt sind optional.

Hier ist ein Beispiel für eine unendliche for-Schleife:

```
for ( ; ; ) {
    // infinite for loop
}
```

**Erweitert für Schleife**

Seit Java 5 gibt es eine zweite Art von for-Schleife namens **enhanced for**. Sie macht es einfacher, alle Elemente in einem Array oder einer Collection zu durchlaufen.

Die Syntax der erweiterten for-Schleife ist:

```
for(Type item : items) {
    statement;
}
```

Da diese Schleife im Vergleich zur Standard-for-Schleife vereinfacht ist, müssen wir beim Initialisieren einer Schleife nur zwei Dinge deklarieren:

1. Das **Handle** für ein Element, über das wir gerade iterieren
2. Das **Source** Array/Collections

*Enhanced Loop* Beispiel:

```
int[] intArr = { 0,1,2,3,4 };

for (int num : intArr) {
    System.out.println(
        "Enhanced for-each loop: i = " + num);
}
```

Es kann auch gut verwendet werden, um über verschiedene Java-Datenstrukturen zu iterieren:

Gegeben eine List von Strings:

```
List<String> list = List.of("A","B","C");

for (String item : list) {
    System.out.println(item);
}
```

Zu guter Letzt, können auch **Lamdba** Ausdrücke (funktionale Interfaces, dazu gibt es ein eigenes Modul) genutzt werden:

```
List<String> letters = new ArrayList<>();

letters.add("A");
letters.add("B");
letters.add("C");

letters.forEach(letter -> System.out.println(letter));
```

### 1.2.2. (do) while

**while**

Die While-Schleife ist eine der grundlegenden Schleifenanweisungen von Java. Es wiederholt eine Anweisung oder einen Anweisungsblock, während sein steuernder `boolescher` Ausdruck wahr ist.

Die **Syntax** der while-Schleife ist:

```
while (Boolean-expression) {
    statement;
}
```

Der boolesche Ausdruck der Schleife wird vor der ersten Iteration der Schleife ausgewertet – was bedeutet, dass die Schleife möglicherweise nicht einmal ausgeführt wird, wenn die Bedingung mit `falsch` ausgewertet wird.

Ein einfaches Beispiel:

```
int i = 0;
while (i < 5) {
    System.out.println("While loop: i = " + i++);
}
```

**do-while**

Die do-while-Schleife funktioniert genauso wie die while-Schleife, mit der Ausnahme, dass die erste Bedingungsauswertung nach der ersten Iteration der Schleife erfolgt:

```
do {
   statement;
} while (Boolean-expression);
```

Ein einfaches Beispiel:

```
int i = 0;
do {
   System.out.println("Do-While loop: i = " + i++);
} while (i < 5);
```

**Demo**:

Die Unit-Tests zur **Demonstration** finden sich hier:

```
→ src/test/java/de/dhbw/loops/demo/LoopTests.java
```

**Übungen**:

**Übung 1**:

Gegeben eine Liste mit Zahlen. Finde den größten Zahlenwert dieser Liste mithilfe eine (alten) `for` (IntelliJ `fori`) Schleife und teste das Ergebnis mithilfe von `assertEquals`.

*Optional*: Wandele die for Schleife in eine `enhanced for` Schleife um.

**Übung 2**:

Gegeben ein Startwert von `10`. Implementiere einen *CountDown* mithilfe einer `while` Schleife, die den CountDown solange runterzählt, bis die `0` erreicht ist und dann abbricht.

# 2. References

See also:

- For Loop
- While Loop