

### Konditionen (if, else, if-else)

Konditionen sind zum "Überprüfen der Wahrheit" von einer Abfrage.

Primitive Abfrage-Möglichkeiten:

> größer als

>= größer oder gleich als

< kleiner als

<= kleiner oder gleich als

== ist gleich

!= Ist nicht gleich

**Aber Achtung:** Bei Strings sowie anderen Objekten sollte die `.equals(x)` Methode genutzt werden. Weil `==` und `!=` schauen, ob es die gleiche Instanz ist. Sobald es eine andere Variabler-Halter ist, ist es nicht mehr die gleiche Instanz.

Und diese Abfragen kommen innerhalb von Abfrage-Blöcken, welche so aussehen:

```
if([Abfrage]){
    // Code falls [Abfrage] true
}
```

An das ende eines Abfrage-Blocks, können weitere Abfragen mit `{ else if([Abfrage]){}` oder ein Block der beim Gegenteil ausgeführt wird `{ else {}` angehängt werden.

Dies sieht dann wie folgt aus:

```
if([Abfrage]){
    // Code falls [Abfrage] true
} else if([Abfrage2]){
    // Code falls [Abfrage2] true
} else {
    // Code falls alles was an mir dran ist falsch war
}
```

Abfragen können auch verknüpft werden. (**Logische Operatoren**)

**&&** Beides muss true ergeben

**||** Eines der beiden muss true ergeben

Beispiel:

```
if(age > 18 && money > 500){
    // Code falls "age" > 18 ist
    und "money" größer als 500 ist
}
```

### Konditionen (if, else, if-else) (cont)

```
}
```

### Switch-Case

Switch-Case Statements testen eine Variable auf einen gewissen Wert gegenüber einer Liste von Möglichkeiten (cases).

Syntax:

```
switch(expression){
    case value1:
        //Code
        break;
    case value2:
        //Code
        break;
    default:
        //Code falls kein case zugesc-
        hlagen hat
        break;
}
```

Oft genutzt für **Enums** oder eine IDs.

### While

Eine While-Schleife führt ihren Code-Block solange aus, bis ihr Statement false ergibt.

Syntax:

```
while(Statement){
    //Code
}
```

Beispiel, welches "321" ausgibt

```
int i = 3;
while(i > 0){
    System.out.print(i);
    i--;
}
```

Ein nutzen dieser Schleifen sind: Iteratoren, Game-Schleifen.

Bei jeder Schleife gibt es 2 Stichwörter:

**break;** Geht aus der Schleife raus.

**continue;** Geht zur nächsten Schleifen-Iteration über. (Überspringt den folgenden Code)

### Primitive Arrays

Eine Array ist eine **Ansammlung** von Objekten, auch genannt eine Liste.

Syntax:

```
Typ[] name = {...};
```

Beispiel:

```
int[] zahlen = {1, 2, 3} //
Erstellt eine int[] mit der
Größe von 3 sowie den angege-
benen Zahlen
int[] nichtGesetzteZahlen = new
int[5]; // Erstellt eine int[]
mit der grÖße 5, welche keine
gesetzten Zahlen hat
```

Operationen:

**arr[index]** Gibt den Wert an der Stelle *index*

**arr[index] = x;** Setzt den Wert an der Stelle *index* auf *x*

**arr.length** Gibt die Länge der Array

Arrays starten bei 0!

### For-Schleife

Es gibt 2 Typen for For-Schleifen:

**"For-Each"** Geht eine Array durch

Syntax:

```
for([Typ] [Name] : [Array]){
    // Code
}
```

Bsp:

```
for(int i : zahlen){
    // Code
}
```

**"For-i"** Schleift solange bis eine Zahl einen Wert erreicht hatSyntax:

```
for([Intialisierung]; [Abfrage];
[Mathel]){
    // Code
}
```

Bsp:

```
for(int i = 0; i != 3; i++){
    // Code
}
```

### 2D Primitive Array

2D Arrays sind Arrays, welche Arrays halten. Das klingt erst einmal kompliziert, ist es aber nicht, wenn man es sich wie **reihen** und **spalten** vorstellt. (x und y)

Syntax:

```
[Typ] [] [] [Name] = { {...}, {...} };
```

Bsp:

```
int [] [] zweiDee = {  
    {1, 2, 3} // 1 "Reihe"  
    {4, 5, 6} // 2 "Reihe"  
}
```

Bei diesem Beispiel wäre **'zweiDee[0][1]' == 2**. Weil man bei Arrays mit 0 anfängt, ist die [0] die erste Reihe, und die darauffolgende [1] die 2. Spalte.

Wird manchmal für 2D Spielfelder wie in TicTacToe verwendet (oA.)

C

By [deleted]  
[cheatography.com/deleted-69240/](https://cheatography.com/deleted-69240/)

Published 15th October, 2018.  
Last updated 15th October, 2018.  
Page 2 of 2.

Sponsored by **Readable.com**  
Measure your website readability!  
<https://readable.com>