E.Tutorial
www.et.ethz.ch

Programmieren mit Java Modul 2

Kontrollstrukturen und Logik

Theorieteil

Inhaltsverzeichnis

Begriffe

Anweisungsblock	Wahrheitswert	while-Schleife
Anweisungskopf	relationale Operatoren	do-while-Schleife
Anweisungskörper	Verzweigung	
logische Operatoren	for-Schleife	geschachtelte Schleife

Autoren:

Lukas Fässler, Barbara Scheuner, David Sichau

E-Mail:

et@ethz.ch

Datum:

9 July 2025

Version: 1.1

Hash: 040665b

Trotz sorgfältiger Arbeit schleichen sich manchmal Fehler ein. Die Autoren sind Ihnen für Anregungen und Hinweise dankbar!

1 Modulübersicht

Ein Algorithmus, der als Programm formuliert ist, besteht in der Regel aus mehreren Anweisungen. Diese Anweisungen werden in einer von der Programmiererin oder dem Programmierer festgelegten Reihenfolge abgearbeitet. Diese Abfolge verläuft selten linear. Oft kommt es vor, dass sich eine Programmsequenz (Folge von Anweisungen) in zwei oder mehrere Programmsequenzen verzweigt, wobei jede nur unter bestimmten Bedingungen ausgeführt wird (**Verzweigung**). Um einen Algorithmus zu vereinfachen, werden oft bestimmte Programmsequenzen wiederholt ausgeführt (**Schleifen**). Mit Hilfe von **Kontrollstrukturen**, die in den meisten Programmiersprachen vorkommen, kann der Programmablauf beeinflusst werden. Die Entscheidung, wie der Ablauf gesteuert wird, muss in **Bedingungen** formuliert werden.

1.1 Anweisungen und Blöcke

Wie bereits in Modul 0 erwähnt, werden einzelne Anweisungen durch ein Semikolon abgeschlossen. Mehrere Anweisungen können in einem **Anweisungsblock** zusammengefasst werden. In Java werden zur Markierung von Anweisungsblöcken **geschweifte Klammern** `{}` verwendet.

```
{ \\ öffnet den Block
    anweisung1;
    anweisung2;
    ...
} \\ schliesst den Block
```

Die Ausführung von Blöcken kann durch Kontrollstrukturen (z.B. Verzweigungen oder Schleifen) gesteuert werden. Diese Kontrollstrukturen bestehen aus einem **Kopf** (*head*) und **Körper** (*body*).

```
    \\ Kopf (head)
{
    \\ Körper (body)
}
```

Bei folgendem Programm wird der Anweisungsblock 1 durch einen Anweisungsblock 2 unterbrochen:

```
\\ Beginn Anweisungsblock 1
Kopf 1 {

\\ Körper 1

  \\ Beginn Anweisungsblock 2
  Kopf 2 {

  \\ Körper 2

  } \\ Ende Anweisungsblock 2

\\ Fortsetzung Körper 1

} \\ Ende Anweisungsblock 1
```

2 Operatoren (Teil II)

Die **arithmetischen Operatoren** sind bereits in Modul 1 beschrieben worden. Im Zusammenhang mit Kontrollstrukturen kommen **logische und relationale Operatoren** zum Einsatz.

2.1 Relationale Operatoren

Relationale Operatoren werden gebraucht, um Werte (Operanden) miteinander zu vergleichen. Sie liefern ein logisches Ergebnis **wahr** (*true*) oder **falsch** (*false*). Werte, die mit relationalen Operatoren verknüpft sind, nennt man in der Aussagenlogik auch **Elementaraussagen**.

Die relationalen Operatoren in Java sind in Tabelle 1 zusammengefasst.

2.2 Logische Operatoren

Logische Operatoren verknüpfen Elementaraussagen miteinander. Dabei werden **Wahrheitswerte** miteinander verglichen. Das Ergebnis ist ebenfalls ein **Wahrheitswert**, also **wahr** (*true*) oder **falsch** (*false*). Da dies die Operanden und Operatoren der Boolschen Aussagenlogik sind, heisst der Datentyp **Boolean**. Die in Java verwendeten logischen Operatoren sind in Tabelle 2 dargestellt.

Operator	Ausdruck	Beschreibung	Liefert wahr (true), wenn...
>	a > b	grösser als	a grösser ist als b.
<	a < b	kleiner als	a kleiner ist als b.
==	a == b	gleich	a und b denselben Wert haben.
!=	a != b	ungleich	a und b ungleiche Werte haben.
>=	a >= b	grösser oder gleich	a grösser oder gleich b ist.
<=	a <= b	kleiner oder gleich	a kleiner oder gleich b ist.

Tabelle 1: Relationale Operatoren in Java.

Operator	Ausdruck	Liefert wahr (true), wenn...
!	!a	a falsch ist (NOT).
&&	a && b	sowohl a als auch b wahr sind (AND).
		Ist a falsch, wird b nicht ausgewertet.
||	a || b	mindestens a oder b wahr sind (OR).
		Ist a wahr, wird b nicht mehr ausgewertet.
^	a^b	a und b unterschiedliche Wahrheitswerte haben

Tabelle 2: Logische Operatoren in Java.

3 Verzweigungen

Verzweigungen überprüfen einen Zustand des Programms. Je nachdem, ob eine bestimmte Bedingung erfüllt ist oder nicht, fährt das Programm mit unterschiedlichen Blöcken von Anweisungen fort. Verzweigungen werden in Java, so wie in vielen anderen Programmiersprachen auch, mit dem Schlüsselwort `if` eingeleitet. Die Bedeutung des `if` ist analog zur englischen Sprache.

If it is raining, then I will take the bus, otherwise I will walk.

Dies könnte in Java wie folgt geschrieben werden:

```
if (rain) {bus} else {walk};
```

Falls die Bedingung `rain` wahr (`true`) ist, wird der Block mit der Anweisung `bus` ausgeführt, andernfalls wird der Block mit der Anweisung `walk` ausgeführt.

Allgemein kann mit einer if-Anweisung zur Laufzeit entschieden werden, ob eine Anweisung oder ein Anweisungsblock ausgeführt werden soll oder nicht. Um Bedingungen zu formulieren, können sowohl Boolsche Variablen, Relationen wie Gleichheit, grösser oder kleiner als auch logische Operatoren verwendet werden.

Je nachdem wie viele Fälle zu unterscheiden sind, ist eine *einseitige* (3.1), *zweiseitige* (3.2) oder *mehrstufige Verzeigung* (3.3) zu wählen.

3.1 Einseitige Verzweigung: bedingte Programmausführung

Eine **einseitige Verzweigung** besteht aus einer Bedingungsabfrage und einem Anweisungsblock, welcher ausgeführt wird oder nicht.

Schreibweise:

```
if (Bedingung) {
    Anweisungsblock;
}
```

Beispiel:

```
if (rain == true) {
    System.out.println("Es regnet.");
}
```

Der Satz `"Es regnet."` wird nur ausgegeben, wenn die Variable `rain` den Wert `true` hat.

3.2 Zweiseitige Verzweigung

Bei einer **zweiseitigen Verzeigung** kann zusätzlich angegeben werden, was im anderen Fall (`else`), wenn also die Bedingung nicht zutrifft, ausgeführt werden soll.

Schreibweise:

```
if (Bedingung) {
    Anweisungsblock1;
}
else {
    Anweisungsblock2;
}
```

Beispiel:

```
if (rain == true) {
    System.out.println("Es regnet.");
}
else {
    System.out.println("Es regnet nicht.");
}
```

Hat die Variable `rain` den Wert `true`, wird der Satz `"Es regnet."` ausgegeben, im anderen Fall (`false`) wird der Satz `"Es regnet nicht."` ausgegeben.

3.3 Mehrstufige Verzweigungen

Mit einer **mehrstufigen Verzweigung** können mehrere Vergleiche gemacht werden. Das kann nötig sein, wenn Sie unterschiedliche Möglichkeiten in einer bestimmten Reihenfolge prüfen möchten.

Schreibweise:

```
if (Bedingung1) {
    Anweisungsblock1;
}
else if (Bedingung2) {
    Anweisungsblock2;
}
else if (Bedingung3) {
    Anweisungsblock3;
}
...
```

Beispiel:

```
if (rain == true) {
    System.out.println("Es regnet.");
}
else if (snow == true) {
    System.out.println("Es schneit.");
}
else if (sun == true) {
    System.out.println("Es scheint die Sonne.");
}
else {
    System.out.println("Die Wetterlage ist unklar.");
}
```

Hat die Variable `rain` den Wert `true`, wird wieder der Satz `"Es regnet."` ausgegeben. Hat sie hingegen den Wert `false`, wird als nächstes die Variable `snow` geprüft. Hat `snow` den Wert `true`, wird der Satz `"Es schneit."` ausgegeben. Hat `snow` den Wert `false`, wird als nächstes die Variable `sun` geprüft. Hat `sun` den Wert `true`, wird der Satz `"Es scheint die Sonne."` ausgegeben. Hat `sun` auch den Wert `false`, wird der Satz `"Die Wetterlage ist unklar."` ausgegeben.

3.4 Fallauswahl (Switch)

Eine andere Möglichkeit, während des Programmablaufs zwischen unterschiedlichen Möglichkeiten auszuwählen, ist die **switch-Anweisung**. Dabei wird der Wert einer Variablen mit unterschiedlichen Werten verglichen.

Schreibweise:

```
switch (ausdruck){
    case constant:
        Anweisungsblock;
    default:
        Anweisungsblock;
}
```

Im Gegensatz zur if-Verzweigung kann mit dem switch-Statement nur auf Gleichheit geprüft werden. Vergleiche auf grösser oder kleiner sind nicht möglich. Als Ausdruck im switch-Statement sind alle ganzzahligen Datentypen und, seit Java 7, auch String-Typen zugelassen. Zusätzlich werden bei einem switch-Statement alle Anweisungen ab dem Einstiegspunkt abgearbeitet. Ist dies nicht erwünscht, sollte der Anweisungsblock mit einem `break` abgeschlossen werden.

Beispiel:

```
switch (test)
{
    case 1:
        System.out.println("Ich wurde ausgewählt.");
        break;
    case 2:
        System.out.println("Du wurdest ausgewählt.");
        break;
    case 3:
        System.out.println("Wir wurden ausgewählt.");
        break;
    default:
        System.out.println("Keiner wurde ausgewählt.");
}
```

4 Schleifen (Loops)

Mit Hilfe von **Schleifen** (*loops*) können dieselben Anweisungen wiederholt ausgeführt werden. Wie in anderen Programmiersprachen gibt es auch in Java verschiedene Schleifenarten. Eine Schleife besteht aus einem **Schleifenkopf** und einem **Schleifenkörper**. Der Schleifenkörper enthält den zu wiederholenden Anweisungsblock. Der Schleifenkopf steuert die Schleife. Er gibt an, wie oft oder unter welchen Bedingungen die Anweisungen des Schleifenkörpers wiederholt werden sollen.

4.1 for-Schleife

Bei der zählergesteuerten **for-Schleife** wird die Anzahl der Schleifendurchläufe durch eine **Laufvariable** von einem Startwert- bis zu einem Endwert durchgezählt. Bei jedem Schleifendurchgang wird der Zähler verändert.

Schreibweise:

```
for (init; test; update){
    Anweisungsblock
}
```

- **Initialisierung** (`init`): Deklarieren der Laufvariable und setzen des Startwerts.
- **Logischer Ausdruck** (`test`): Es wird bei jedem Durchlaufen geprüft, ob die Schleife weiterlaufen muss oder der Endwert schon erreicht worden ist.
- **Aktualisierung** (`update`): Die Laufvariable wird nach jedem Durchlaufen der Schleife verändert.

Beispiel: Folgende Anweisung gibt die Werte 0 bis 4 am Bildschirm aus:

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

Zunächst wird die Laufvariable `i` deklariert (Datentyp Integer) und auf den Anfangswert 0 gesetzt. Danach wird geprüft, ob `i` kleiner ist als 5. Ist dies der Fall, werden die Anweisungen des Schleifenkörpers durchlaufen und dann der Wert von `i` um 1 erhöht.

4.2 while-Schleife

Es ist nicht immer vorhersehbar, wie oft Anweisungen wiederholt werden müssen, da die Anzahl der Wiederholungen von dem abhängen kann, was im Schleifenkörper passiert. Hier geraten wir bei zählergesteuerten Schleifen an eine Grenze. Bei **bedingungsabhängigen Schleifen** wird die Anzahl der Wiederholungen nicht von einem Zähler, sondern von einer **Bedingung** abhängig gemacht. Diese Bedingung wird bei jedem Schleifendurchgang überprüft. **While-** und **do-while-Schleifen** unterscheiden sich dadurch, ob diese Bedingung vor oder nach dem Anweisungsblock überprüft wird.

Schreibweise:

```
Initialisierung der Variablen
while (Bedingung){
    Anweisungsblock
    Aktualisierung
}
```

- **Initialisierung**: Deklarieren einer oder mehrerer Variablen und initialisieren der Startwerte.
- **Bedingung**: Die Bedingung wird geprüft, sobald die while-Schleife erreicht wird. Ist die Bedingung wahr (`true`), wird der Schleifenkörper ausgeführt. Ist die Bedingung falsch (`false`), wird die Schleife abgebrochen und die Anweisungen des Schleifenkörpers werden nicht mehr ausgeführt. Nach jedem Durchlaufen der Schleife wird die Bedingung erneut geprüft.
- **Aktualisierung**: Innerhalb des Schleifenkörpers müssen sich Werte so verändern, dass die Bedingung irgendwann erreicht wird, sonst droht eine Endlosschleife, was der Definition eines Algorithmus widerspricht (ein Algorithmus muss seine Arbeit immer beenden).

Beispiel: Folgende Anweisung gibt die Werte 0 bis 4 am Bildschirm aus:

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

Zunächst wird eine Variable `i` initialisiert und auf 0 gesetzt. Zu Beginn der Schleife wird geprüft, ob i kleiner ist als 5. Ist dies der Fall (`true`), wird der Schleifenkörper ausgeführt. Ist dies nicht der Fall (`false`), wird die Schleife abgebrochen. Die Variable `i` wird innerhalb des Schleifenkörpers jedes Mal um 1 erhöht.

4.3 do-while Schleife

Der Schleifenkörper einer **do-while-Schleife** wird im Gegensatz zur while-Schleife mindestens einmal ausgeführt, da die Bedingungsprüfung zur Wiederholung jeweils am Ende des Schleifenkopfs erfolgt.

Schreibweise:

```
Initialisierung der Variablen
do {
    Anweisungsblock
    Aktualisierung
} while (Bedingung);
```

- **Initialisierung**: Deklarieren einer Variable und setzen des Startwerts.
- **Aktualisierung**: Innerhalb des Schleifenkörpers müssen sich Werte so verändern, dass die Bedingung irgendwann erreicht wird, sonst droht eine Endlosschleife.
- **Bedingung**: Die Bedingungsprüfung findet erst statt, nachdem der Schleifenkörper durchlaufen ist. Sie enthält die Bedingung zum Wiederholen der Schleife. Trifft diese Bedingung zu, wird die Schleife erneut durchlaufen, sonst wird sie abgebrochen.

Beispiel: Folgende Anweisung gibt die Werte 0 bis 4 am Bildschirm aus:

```
int i=0;
do {
    System.out.println(i);
    i++;
} while (i<5);
```

Es wird eine Variable `i` initialisiert und auf 0 gesetzt. Im Schleifenkörper wird die Variable `i` um 1 erhöht. Erst jetzt wird geprüft, ob `i` kleiner ist als 5. Sobald `i` den Wert 5 erreicht, wird die Schleife abgebrochen.

4.4 Geschachtelte Schleifen

Beim Programmieren kommt es oft vor, dass zwei Schleifen ineinander **geschachtelt** werden (*nested loops*). Das hat zur Folge, dass eine äussere Schleife eine innere steuert.

Dies kann wie folgt dargestellt werden:

```
Äussere Schleife {
    Innere Schleife {
        Anweisungsblock
    }
}
```

Eine Analogie zu den geschachtelten Schleifen findet man bei unserer Erde, die sich um die Sonne dreht. Eine Umkreisung in einem Jahr wäre mit der äusseren Schleife vergleichbar, und eine Drehung der Erde um die eigene Achse innerhalb eines Tages wäre mit der inneren Schleife vergleichbar.

In Java könnte ein Programm zur Anzeige von Tagen und Stunden eines Jahres (das kein Schaltjahr ist) mit folgender geschachtelten Schleife geschrieben werden:

```
for (int tage=0; tage<365; tage++){
    for (int stunden=0; stunden<24; stunden++) {
        System.out.println("Tag " + tage);
        System.out.println("Stunde " + stunden);
    }
}
```

Die ersten drei Ausgaben lauten:

```
Tag 0: Stunde 0
Tag 0: Stunde 1
Tag 0: Stunde 2
```

Die letzten drei Ausgaben lauten:

```
Tag 364: Stunde 21
Tag 364: Stunde 22
Tag 364: Stunde 23
```