

## Wiederholende Übungen zu den Datentypen

Wir wollen nun unsere Kenntnisse festigen und erweitern. Legen Sie im Projekt **Datentypen** eine neue Klasse **TestDTK** an, in der Sie alle folgenden Beispiele als öffentliche Methoden

```
+test*(): void
```

ohne Parameterliste und Rückgabewert implementieren! (\* wird durch die jeweilige Beispielnnummer ersetzt!)

Klasse TestDTK:

```
/**
 * Klasse zum Testen von Datentypen und Kontrollstrukturen
 *
 * @author (Ihr Name)
 * @version (eine Versionsnummer oder ein Datum)
 */
public class TestDTK{
    /**
     * Konstruktor für Objekte der Klasse Test
     */
    public TestDTK(){

    }
    /**
     * Beispiel test1, Deklaration und Wertzuweisung Integerzahl,
     * Multiplikation
     */
    public void test1(){
        int n;
        n = 3*4;
        System.out.println(n);
    }
}
```

Alle benötigten Variablen werden lokal in den Methoden vereinbart und belegt! Als Ausgabe dient der Befehl `System.out.println(String)`, der im Konsolenfenster von BlueJ die Ausgaben protokolliert:

<code>System.out.println("Das ist ein Text");</code>	<p><b>Aufruf</b> des Unterprogramms <code>println</code> im Object <code>System.out</code>. Dieses Unterprogramm kann den als Parameter übergebenen <b>String</b> "Hello World" auf der Kommandozeile ausgeben.</p> <p>Die Zeile ist ein Beispiel für eine <b>Anweisung</b>. Jede Anweisung wird mit einem <code>;</code> abgeschlossen.</p>
--	--

Das Programm berechnet das Produkt der Zahlen 3 und 4, also 12, speichert das Ergebnis auf einer Variablen ab, und gibt den Wert der Variablen aus.

Beispiel 1 –Methode `test1()`:

```
/**
 * Beispiel test1, Deklaration und Zuweisung Integerzahl, Multiplikation
 */
public void test1(){
    int n;
    n = 3*4;
    System.out.println(n);
}
```

Es gibt in diesem Beispiel folgende Sprachelemente:

## Wiederholende Übungen zu den Datentypen

<code>int n;</code>	Deklaration einer <b>Variablen</b> mit Namen <code>n</code> . Die Variable wird auf dem Stack gespeichert und steht nur in dem Unterprogramm <code>main</code> lokal zur Verfügung. Nach Beendigung des Unterprogramms steht sie nicht mehr zur Verfügung. Da unser Programm dann ebenfalls beendet ist, ist dies nicht weiter relevant. Die Variable ist vom Typ <code>int</code> (integer, d.h. ganze Zahl). Sie kann Werte von $-2^{147} \cdot 483 \cdot 648$ bis $2^{147} \cdot 483 \cdot 647$ annehmen. Die allgemeine Form der Variablendeklaration lautet <code>datentyp variablenname;</code>
<code>n=3*4;</code>	In diesem Fall ist der Datentyp <code>int</code> und der Variablenname <code>n</code> . Dies ist eine <b>Zuweisung</b> . An <code>n</code> wird der Wert zugewiesen, der sich aus der Berechnung des <b>Ausdrucks</b> <code>3*4</code> ergibt. Dieser Ausdruck ist vom Typ <code>int</code> , da er eine <b>Operation</b> mit zwei <code>int</code> - <b>Konstanten</b> ist.

Natürlich ergibt sich der Typ eines Ausdrucks aus dem Typ der Operanden. In diesem Fall sind beide Konstanten, 3 und 4, vom Typ `int` und damit auch das Ergebnis `3*4`. Diese allgemeine Regel gilt immer.

**Regel:** Der Typ der Operanden bestimmt den Typ des Ergebnisses.

## Datentypen

Es gibt folgende elementare Datentypen. In der Tabelle sind jeweils der Java-Name des Datentyps, Beispiele für Konstanten des Datentyps und die möglichen Werte wiedergegeben, die Variablen dieses Datentyps annehmen können.

Datentyp	Konstanten	Wertemenge
<code>short</code>		$-2^{15} \dots 2^{15} - 1$
<code>int</code>	3, -3, 0, 0x3FF (hexadezimal), 0377 (octal)	$-2^{31} \dots 2^{31} - 1$
<code>long</code>	31, -3000000000001 (1, keine 1- l...kleiner Buchstabe l)	$-2^{63} \dots 2^{63} - 1$
<code>float</code>		ca 6-stellige Mantisse und zweistelliger Exponent.
<code>double</code>	3.1415, 1e20, 1e-10, -1E10	ca. 16-stellige Dezimal-Mantisse und 3-stelliger Exponent.
<code>byte</code>		$0 \dots 2^8 - 1 = 255$ . Normale Speichereinheit.
<code>char</code>	'A', 'a', ' ', '\t', '\n', '\xFE', '\u0121' (Unicode)	Zwei byte-Charakter in Unicode. Wenn das oberste Byte 0 ist, so entspricht dies dem normalen Zeichensatz (ASCII). In anderen Sprachen ist dieser Datentyp nur ein Byte lang.
<code>boolean</code>	true, false	Wahr oder falsch.

Der Typ von Konstanten, wie 3 und 4, ergibt sich aus Ihrer Schreibweise. Bei Zahlen ohne Dezimalpunkt ist er **int**, bei Zahlen mit Dezimalpunkt **double**, z.B. 3.4, -1.55 oder auch 1.5e10. **long**-Konstanten haben ein angehängtes **l** als kleinen Buchstaben.

Ein Ausdruck bleibt solange vom Typ **int** als keine **double**-Werte in ihm enthalten sind.

**Achtung!** Der Ausdruck **1/3** ergibt 0, da er ganzzahlig berechnet wird.

## Wiederholende Übungen zu den Datentypen

Will man 0.333... als Ergebnis erhalten, so kann man **1.0/3** schreiben. In diesem Fall ist der erste Operand **double**, und daher auch das Ergebnis. Natürlich geht auch **1.0/3.0**, oder wie wir später sehen werden **(double)1/3**.

Nun können wir etwa einen komplizierteren Ausdruck berechnen.

Beispiel 1 –Methode **test2()**:

```
double x=1.5,y;  
y=1+x+x*x/2+x*x*x/6+x*x*x*x/24;  
System.out.println("y = "+y);
```

Es ist also möglich mehrere Variablen in einer Zeile zu deklarieren (**int x, y;**) und ihnen auch gleich Werte zuzuweisen (**double x=1.5;**). Wird einer Variablen kein Wert zugewiesen, kann sie nicht verwendet werden. Der Compiler wird dies monieren.

Außerdem wird hier die Variable **x** als Operand in Ausdrücken eingesetzt. Bisher haben wir nur Konstanten in Ausdrücken verwendet.

Interessant ist weiter, wie  $1+x+\dots$  berechnet wird. 1 ist nämlich eine Konstante vom Typ **int**, aber **x** ist eine Variable vom Typ **double**. Hier ergibt sich das Ergebnis aus dem höherwertigem Typ der Operanden, in diesem Fall **double**. Natürlich wird die Summe aus mehreren Summanden von links nach rechts berechnet. Java beherrscht dabei die Regel "Punkt vor Strich" und berücksichtigt auch Klammersetzungen mit runden Klammern ().

Man beachte die Addition eines Strings und einer **double**-Variablen bei der Ausgabe (**"y = "+y**).

Hier wird zuerst der **double**-Wert zu einem String umgewandelt und danach werden beide Strings aneinandergehängt.

Das Programm ergibt folgende Ausgabe: **y = 4.3984375**

Was passiert, wenn ein Ausdruck eines niederen Typs auf eine Variable eines höheren Typs gespeichert wird? In diesem Fall wird automatisch umgewandelt. Umgekehrt gibt es eine Fehlermeldung des Compilers.

**Regel:** Automatische Umwandlungen sind nur erlaubt, wenn das Ziel den Wert in jedem Fall aufnehmen kann.

**Test3**

```
int n = 3;  
long l = n; // automatische Umwandlung (
```

Umgekehrt ist ein **type cast** notwendig.

**Test4**

```
long l = 20000;  
int n = (int) l; // type cast von long nach int
```

Damit lassen sich Datentypen beliebig ineinander umwandeln. Für Fehler trägt der Programmierer die Verantwortung.

## Wiederholende Übungen zu den Datentypen

### Test5

```
long c=1000000000000000L // das L ist notwendig!  
System.out.println(c); // 5.1  
System.out.println((int)c); // 5.2
```

Bei Überlauf wird kein Laufzeitfehler erzeugt. Division durch 0 oder ähnliche Fehler erzeugen allerdings eine Fehlermeldung (*arithmetic exception*).

## Arithmetische Operatoren

Für die arithmetischen Datentypen (alle außer **boolean**) sind die üblichen Operatoren **+, -, \*, /** und **%** (modulo) definiert. Das Ergebnis ist jeweils vom gleichen Typ. Deswegen

### Test6

```
int n=3;  
System.out.println(n/4); // 6.1
```

Die Ausführungsreihenfolge von Operatoren ist streng festgelegt, entspricht aber der üblichen Konvention ("Punkt vor Strich", und Berücksichtigung von Klammern).

### Test7

```
System.out.println((3.0+7.0)/(4.0*5.0+8.0)); //7.1  
System.out.println((3.0+7)/(4*5+8)); // 7.2  
System.out.println((double)(3+7)/(4*5+8)); // 7.3  
System.out.println((3+7)/(4*5+8)); // 7.4
```

Vergleiche von Zahlen ergeben **true** oder **false** vom Typ **boolean**. Es gibt die Vergleiche **<, <=** (kleiner oder gleich), **>, >=**, **==** (gleich), **!=** (ungleich). Beispiel:

### Test8

```
double x=3;  
System.out.println(1/x>=0.3); // 8.1
```

Ausdrücke vom Typ **boolean** werden mit **&&** (und) und **||** (oder) verknüpft. Sie werden ins Gegenteil verkehrt mit **!** (nicht).

Solche *boolschen Ausdrücke* werden wir später besonders für die Ablaufsteuerung von Programmen benötigen, zum Beispiel für bedingte Sprünge und Anweisungen.

Es gibt auch die Operatoren **++** und **--**, und zwar nachgestellt und vorausgestellt. Diese Operatoren erhöhen, bzw. erniedrigen, einen ganzzahligen Wert um 1. **++n** tut dies vor der Verwendung von **n**, und **n++** danach. Man kann diese Operatoren auch als Anweisungen verwenden. Beispiel:

### Test9

```
int i=1;  
i++;  
System.out.println(i); // 9.1.
```

Es ist fraglich, ob es nicht besser ist, **i=i+1** zu schreiben, was deutlicher ist, und damit weniger fehlerträchtig. Jedoch wurde **i++** von früheren Compilern in Code übersetzt, der schneller ausgeführt wird, so dass sich diese Schreibweise eingebürgert hat.

Außerdem existieren Kombinationen zwischen Operatoren und **=**.

## Wiederholende Übungen zu den Datentypen

Beispiel: **Test10**

```
int i=1;
i+=2;
System.out.println(i); // 10.1.
```

Hier ist die Frage, ob **i=i+2** nicht deutlicher ist, und dennoch genauso schnell ausgeführt wird.

## Strings

Strings sind eigentlich keine elementaren Datentypen sondern verhalten sich wie Klassen. Da sie aber für nicht-mathematische Beispiele entscheidend wichtig sind, behandeln wir sie schon hier. Später werden wir uns noch einmal detaillierter um Strings kümmern.

String-Konstanten haben wir schon benutzt. Solche Konstanten werden durch Gänsefüßchen eingeschlossen. Beispiel

**Test11**

```
String s = "Dies ist ein Teststring.";
```

Die wichtigste Rechenoperation mit Strings ist die Verkettung (Addition) **+**. Diese Addition hängt beide Strings hintereinander. Man kann auch elementare Datentypen zu Strings addieren. Diese Werte werden dann zuerst in eine String-Representation umgewandelt. Dadurch kann man numerische Ausgaben kommentieren.

**Test12**

```
double pi = Math.PI;
System.out.println("Pi zum Quadrat ist " +(pi*pi));
```

Hier wird zuerst **pi\*pi** berechnet, dann wird dieser Wert in einen String umgewandelt, und schließlich wird

```
Pi zum Quadrat ist 9.869604401089358
```

ausgegeben. Wie viele Ziffern nach dem Komma ausgegeben werden, hängt von der Java-Version ab. Im Allgemeinen gibt Java alle relevanten Stellen aus und enthüllt damit Rechenungenauigkeiten, die bei der Ausgabe mit anderen Sprachen weggerundet werden, obwohl sie intern dort genauso vorhanden sind. Will man auf eine bestimmte Stellenzahl formatieren, so muss man die Formatierungsroutinen von Java verwenden, die wir später behandeln.

Hier ist ein anderes Beispiel, dass den Buchstaben ausgibt, der den Code dezimal von 67 hat.

**Test13**

```
int n = 67;
System.out.println("Der Buchstabe Nummer " + n + " ist " +(char)n);
```

n wird einmal als **int**-Wert und einmal als **char**-Wert behandelt, und dementsprechend verschieden in einen String umgewandelt. Das Ergebnis ist

```
Der Buchstabe Nummer 67 ist C
```

(Vergleich ASCII-Tabelle im Tafelwerk!)

Hier haben wir noch mal eine Anwendung zum expliziten Type-Cast!

## Verzweigungen (Bedingte Anweisungen) und Schleifen

### if ...

Bisher haben wir in der Wiederholung nur Programme betrachtet, bei denen alle Anweisungen nacheinander ausgeführt werden. Solche Programme sind aber in der Funktionalität stark eingeschränkt. Die wahre Kraft erreicht ein Programm erst dadurch, dass Anweisungen mehrfach, oder nur unter gewissen Bedingungen ausgeführt werden. Nur dadurch kann es nicht-triviale Aufgaben erledigen.

Will man eine Anweisung nur unter einer Bedingung ausführen lassen, so verwendet man `if`. Ein Beispiel wäre etwa

```
if (x<0) x=-x; // entspricht x=Math.abs(x);
```

Die Anweisung `x=-x` wird also nur ausgeführt, wenn `x` negativ ist. Allgemein lautet die Syntax

```
if (bedingung) anweisungsblock
```

Man beachte, dass die Bedingung in runden Klammern stehen muss. *anweisungsblock* entweder eine einzelne Anweisung (mit `;`) oder ein *Block* von Anweisungen, der durch `{...}` geklammert ist. Beispiel:

**Test14**

```
if (x<0)
{
    System.out.println("x war negativ");
    x=-x;
}
```

Man kann auch die Alternative festlegen.

**Test15**

```
if (x<0)
{
    System.out.println("x war negativ");
    x=-x;
}
else
{
    System.out.println("x war positiv");
}
```

Der `else`-Teil wird nur ausgeführt, wenn die Bedingung falsch ist. In diesen Beispiel kann man sich die Klammern nach `else` sparen. Allgemein lautet die Syntax also

```
if (bedingung) anweisungsblock
else anweisungsblock
```

Wie vorher, kann *anweisungsblock* auch eine einzelne Anweisung mit `;` sein. Falls mehrere `if` geschachtelt werden, so bezieht sich `else` auf das vorhergehende `if`, unabhängig von der Einrückung.

**Regel:** *else bezieht sich immer auf das vorangegangene if.*

Beispiel:

**Test16**

```
if (needpositive)
    if (x<0) x=-x;
    else System.out.println("x war schon positiv");
```

Dabei ist `needpositive` eine Variable vom Typ `boolean`, die ein Flag enthalten soll, ob der Test durchzuführen ist oder nicht (`boolean needpositive = true/false` ist vor der Verzweigung einzubauen!) Die Einrückung von `else` ist unwichtig, es bezieht sich in jedem Fall auf

```
if (x<0) ...
```

Vermutlich ist es bei geschachtelten `if` besser, die Anweisungen mit `{...}` zu klammern auch wenn sie formal nicht notwendig sind.

*Man sollte so und durch das Einrücken immer auf einwandfreie Lesbarkeit seiner Programme achten!*


### while ...

Schleifen dienen dazu, Programnteile mehrfach mit veränderten Werten ausführen zu lassen. Schleifen werden abgebrochen, wenn eine vorgegebene Bedingung erfüllt ist, bzw. nicht mehr erfüllt ist.

Die einfachste *Schleife* ist die while-Schleife. Ein Block von Anweisungen wird so lange ausgeführt, wie die Abbruch-Bedingung wahr ist. Das folgende Beispiel zählt zum Beispiel bis 10.

```
Test17
int i=1;
while (i<=10)
{
    System.out.println(i);
    i++; // oder ausführlicher: i=i+1;
}
```

Diese Schleife kann man so formulieren:

1. Setze den Zähler *i* auf 1.
  2. Gib *i* aus.
  3. Erhöhe *i* um 1.
  4. Falls *i* noch kleiner oder gleich 10 ist, springe nach 2.
  5. Ansonsten: Beende die Schleife.
- 

Allgemein lautet die Syntax der while-Schleife

```
while (bedingung) anweisungsblock
```

Wie bei *if* muss die Bedingung in runden Klammern stehen.

Die *Abbruchsbedingung* wird jeweils überprüft, *bevor* der Anweisungsblock durchlaufen wird.

Falls sie etwa von vornherein *false* ist, so wird die Schleife nie durchlaufen.

Man muss sicher stellen, dass die Abbruchsbedingung auch irgendwann erfüllt wird. Sonst entsteht eine *Dauerschleife*, die nur durch einen Programmabbruch beendet werden kann. Bei Java kann man dazu in der Kommandozeile CNTRL-C drücken. Eine Dauerschleife ist etwa

```
while (true)
{
    System.out.println("Bitte CNTRL-C drücken");
}
```

Diese Schleife druckt andauernd den gleichen Text. Das Programm muss vom Benutzer abgebrochen werden.

### do ... while

Man kann die Bedingung auch ans Ende stellen.

```
do anweisungsblock while (bedingung);
```

Hier wird die Schleife zuerst durchlaufen und dann die Bedingung getestet. Falls sie *false* ist, bricht die Schleife ab. Die Schleife wird also in jedem Fall mindestens einmal durchlaufen.

Ansonsten unterscheidet sie sich nicht von der while-Schleife.

Unser Beispiel, das von 1 bis 10 zählt, sieht dann so aus.

```
Test18
int i=1;
do
{
    System.out.println(i);
    i++; // oder ausführlicher: i=i+1;
}
while (i<=10);
```

## Wiederholende Übungen zu den Kontrollstrukturen

### for ...

Eine andere, kompaktere Form der Schleife ist die for-Schleife, die hauptsächlich eine Vereinfachung für einfache Zählschleifen ist. Zuerst ein Beispiel, das wieder von 1 bis 10 zählt.

**Test19**

```
int i;
for (i=0; i<=10; i++) System.out.println(i);
```

Allgemein lautet die Syntax

```
for (anfangs-anweisung; bedingung; schleifen-anweisung)
    anweisungsblock
```

<i>anfangs-anweisung</i>	Wird vor Beginn des ersten Schleifendurchlaufs ausgeführt.  Im obigen Beispiel wird <i>i</i> auf den Wert 0 gesetzt.
<i>schleifen-anweisung</i>	Diese Anweisung wird am Ende jedes Schleifendurchlaufs ausgeführt.  Im obigen Beispiel wird <i>i</i> um 1 erhöht.
<i>bedingung</i>	Dieser Ausdruck wird vor jedem Durchlauf (auch vor dem ersten) ausgewertet. Der Ausdruck muss vom Wert boolean sein. Fällt die Auswertung positiv aus, so wird ein neuer Durchlauf gestartet, ansonsten wird die Schleife beendet.  Im obigen Beispiel wird die Schleife beendet, wenn <i>i</i> größer als 10 wird (d.h. sie wird ausgeführt, solange <i>i</i> ≤10 wahr ist.

Das folgende Beispiel zählt rückwärts von 10 bis 1.

**Test20**

```
int i;
for (i=10; i>=1; i--)
    System.out.println(i);
```

Es ist wahrscheinlich günstig, die for-Schleife nur für einfache Zählschleifen einzusetzen, da sonst die Syntax verwirrend wird.

Man kann übrigens auch die Variable gleich in der Schleife deklarieren. Damit sieht das letzte Beispiel so aus:

**Test21**

```
for (int i=10; i>=1; i--)
    System.out.println(i);
```

Man kann auch mehrere einfache Befehle, durch Komma getrennt, als Anfangsanweisung verwenden. Beispiel

**Test22**

```
for (i=0, k=n; i<n; i++, k--)
    System.out.println(i+", "+k);
```

Dies zählt *i* hoch und *k* herunter. Hier ist schon unklar, ob eine solche Kompaktheit nicht der Übersichtlichkeit schadet.

Man kann den gleichen Effekt auch deutlicher erreichen.

**Test23**

```
k=n;
for (i=0; i<n; i++)
{
    System.out.println(i+", "+k);
    k--;
}
```



## Wiederholende Übungen zu den Kontrollstrukturen

**Wichtig!** Die folgende Schleife ist inkorrekt:

**Test24**

```
int i;
for (i=0; i<=10; i++){
    System.out.println(i);}
```

Das Semikolon vor dem Schleifenblock ist falsch. Es wird als Leerbefehl interpretiert, der dann 10-mal ausgeführt wird. Danach wird der Schleifenblock einmal ausgeführt mit dem Wert 10 für i.

### Beispiele

Als erstes Beispiel erzeugen wir 100 Zufallszahlen zwischen 0 und 1. Die Routine `Math.random()` liefert eine (Pseudo-)Zufallszahl vom Type `double` zwischen 0 und 1. Die beiden Endwerte kommen nicht vor, und die Zahlen sind ansonsten gleichmäßig im Intervall verteilt. Unser Ziel ist, die größte der erzeugten Zahlen auszugeben. Dazu verwenden wir folgenden Algorithmus:

1. Erzeuge die erste Zufallszahl und merke diese Zahl als Maximum.
2. Erzeuge die restlichen Zufallszahlen und teste, ob eine von ihnen größer als das Maximum ist.  
Wenn ja, aktualisiere das Maximum.
3. Gib das Maximum aus.

Das Programm dazu sieht folgendermaßen aus:

**Test25**

```
int i;
double max;
max=Math.random(); // Erste Zufallszahl
for (i=2; i<=100; i++)
{
    double x=Math.random(); // nächste Zahl
    if (x>max) max=x; // Aktualisiere Maximum
}
System.out.println("Maximum :"+max);
```

Als zweites Beispiel schachteln wir zwei Schleifen ineinander.

**Test26**

```
for (i=1; i<10; i++)
{
    for (j=1; j<10; j++) System.out.print(i*j+" ");
    // gib eine Zeile des Einmaleins aus
    System.out.println(""); // neue Zeile anfangen.
}
```

Das Programm druckt das kleine Einmaleins.

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

Das Unterprogramm `System.out.print` druckt *ohne* Zeilenumbruch. Dadurch lassen sich alle 10 Werte in einer Zeile ausgeben, bevor ein Zeilenumbruch die nächste Zeile beginnt.

## Wiederholende Übungen zu den Kontrollstrukturen

Die Formatierung lässt noch zu wünschen übrig, da die Zahlen verschieden breit ausgegeben werden. Als einfachen Fix bietet sich folgende Lösung an:

Der Trick besteht darin, ein zusätzliches Leerzeichen auszugeben, wenn die Zahl kleiner als 10 ist. Dadurch werden alle Zahlen auf zwei Stellen ausgegeben.

Man beachte die Deklaration einer lokalen Variablen im Schleifenblock. Diese Variable gilt nur innerhalb der `for`-Schleife. Die Ausgabe ist nun wesentlich schöner.

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

Setzen Sie diese Lösung um! **Test27**

### **break, continue**

Eine Schleife kann auch unterbrochen werden. Dazu dient die `break`-Anweisung. Normalerweise wird sie in einer Bedingung aufgerufen, das heißt innerhalb einer `if`-Anweisung. Z.B. zählt die folgende Schleife nur bis 5, weil sie an diesem Punkt durch `break` unterbrochen wird.

**Test28**

```
for (int i=1; i<=10; i++)
{
    System.out.println(i);
    if (i==5) break;}
}
```

Es kann natürlich mehrere `break`-Anweisungen in einer Schleife geben.

Analog funktioniert der Befehl `continue`. Er bricht die Schleife allerdings nicht ab, sondern überspringt nur den Rest dieses Schleifendurchlaufs. Die Abbruchbedingung der Schleife wird danach überprüft, und bei `for`-Schleifen wird vorher noch die Schleifen-Anweisung ausgeführt.

Bei geschachtelten Schleifen kann auch zu einer äußeren Schleife gesprungen werden. Dazu dienen Labels.

```
loop: while (bedingung)
{
    ...
    break loop;
    ...
}
```

In diesem Beispiel springt der `break`-Befehl aus der angegebenen `while`-Schleife, auch wenn er in einer inneren Schleife enthalten ist. Dadurch lässt sich der Effekt eines absoluten Sprunges erreichen.

```
loop: while (true)
{
    ...
    if (error) break loop;
    ...
    break;
}
```

Dies ist eigentlich gar keine Schleife, da am Ende ja ein `break` steht. Falls aber ein Fehler auftritt, wird jedesmal der Rest der Klammer übersprungen.

Es ist fraglich, ob ein solcher absoluter Sprung nützlich ist. Der Programmablauf wird dadurch verschleiert. Dies führt dazu, dass das Programm schwerer zu durchschauen ist, und damit fehleranfälliger.