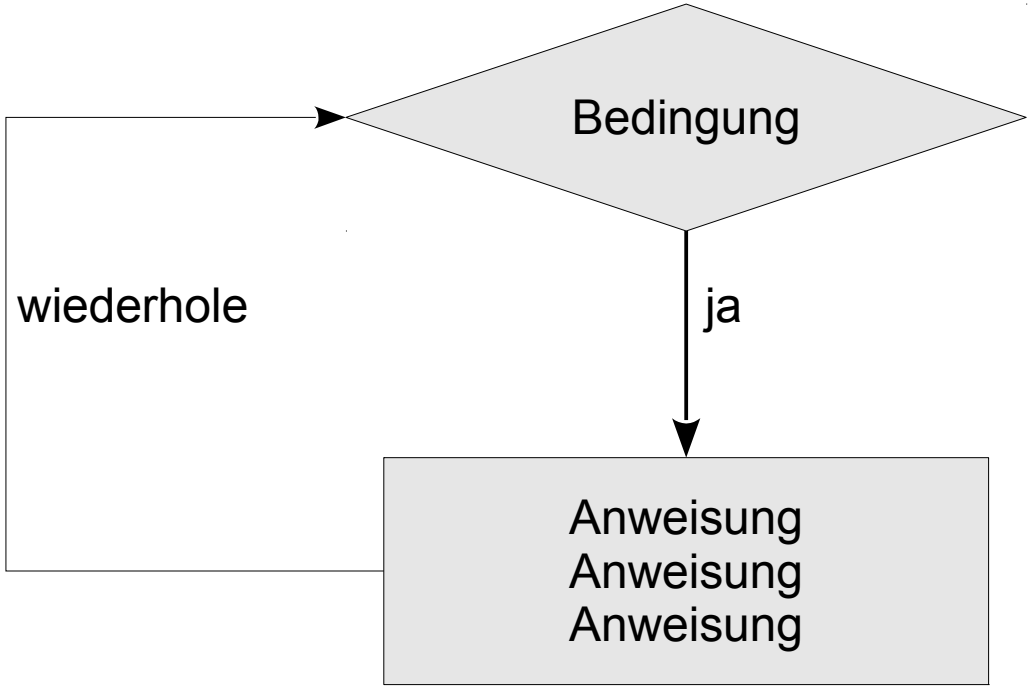
Java - Schleifen



Anweisungen

```
    double dblErgebnis;

    intZahl = 4;
    dblErgebnis = intZahl * DBL_ZAHL;
    System.out.println(dblErgebnis);
```

- Deklaration von Variablen und Konstanten.
- Zuweisungen an Variablen.
- Berechnung von variablen Werten.
- Aufruf von Methoden.

Leerzeichen in Anweisungen

```
    double dblErgebnis;

    intZahl = 4;
    dblErgebnis = intZahl * DBL_ZAHL;
```

- Zwischen Operanden und Schlüsselwörtern muss ein Leerzeichen stehen.
- Zwischen Operanden und Operatoren können Leerzeichen stehen, müssen aber nicht.
- Mit Hilfe von Leerzeichen wird die Lesbarkeit einer Anweisung erhöht.

Anweisungen „beenden“

```
    double dblErgebnis;

    intZahl = 4;
    dblErgebnis = intZahl * DBL_ZAHL;
```

- Jede Anweisung endet mit dem Semikolon.
- Pro Zeile können beliebig viele Anweisungen stehen. Aber es sollte nie mehr als eine Anweisung stehen.

Leere Anweisung

```
    double dblErgebnis;

    intZahl = 4;
    ;
```

- Ein Semikolon kennzeichnet eine leere Anweisung.
- Eine leere Anweisung an den falschen Stellen kann zu Fehlern im Programmablauf führen.

Block von Anweisungen

```
public class Java03_BedingteAnweisung {

    public static void main(String[] args) {
        final double DBL_ZAHL = 0.5;
        int intZahl;
        double dblErgebnis;

        intZahl = 4;
        dblErgebnis = intZahl * DBL_ZAHL;

    }

}
```

Erläuterung

- Beginn und Ende mit den geschweiften Klammern.
- Zusammenfassung von Anweisungen, die in Abhängigkeit einer Bedingung ausgeführt werden.
- Eine Methode fasst Anweisungen zur Beschreibung einer Aktivität zusammen.

Operanden

- Variablen, die einen dynamischen Wert speichern.
- Konstanten, die einen festen Wert speichern. Der Wert einer Konstanten ist nicht durch Anweisungen veränderbar.
- Literale. Statische Werte, die direkt in einer Anweisung stehen. Zahlen wie 4 oder 0.5. Einzelne Zeichen wie ´a´.

Beispiel

```
    public static void main(String[] args) {
       final double dlbZahl = 0.5;   // Konstante
       int intZahl;                  // Variable

       intZahl = 4;                  // Literal 4

    }
```

Operatoren

- Regeln zur Berechnung von Werten.
- Verknüpfung von Ausdrücken und Werten.
- Vorschriften zur Bildung von Ausdrücken aus mehreren Operanden.

Operatoren in Java

- Arithmetische Operatoren berechnen mit Hilfe eines Ausdrucks einen Wert. Der Wert wird einer Variablen zugewiesen, in einer Tabelle gespeichert etc.
- Operatoren vergleichen zwei Werte und geben einen boolschen Wert zurück. Falls der Vergleich stimmt, wird true (wahr) zurückgegeben. Andernfalls wird false (falsch) zurückgegeben. Bedingte Anweisungen und Schleifen in Java nutzen Vergleichsoperatoren.
- Logische Operatoren verknüpfen Ausdrücke, die einen boolschen Wert zurückgeben. Logische Operatoren wie AND, OR und NOT verknüpfen Bedingungen in Anweisungen.

Ausdruck

- Operatoren und Operanden werden in Abhängigkeit bestimmter Regeln miteinander verbunden.
- Ein Ausdruck gibt immer ein Wert zurück. Zum Beispiel geben arithmetische Ausdrücke einen berechneten Wert zurück.

Arithmetische Operatoren

Operator	Berechnung	Beispiel
+	Addition	3 + 4 = 7
-	Subtraktion	3 – 4 = -1
*	Multiplikation	3 * 4 = 12
/	Division	3 / 4 = 0.75
%	Division mit Rest	3 % 4 = 3

Beispiel

```
final double DBL_ZAHL = 0.5;
int intZahl;
double dblErgebnis;

intZahl = 4;
dblErgebnis = intZahl * DBL_ZAHL;
dblErgebnis = intZahl / DBL_ZAHL;
dblErgebnis = intZahl + DBL_ZAHL;
dblErgebnis = intZahl - DBL_ZAHL;
```

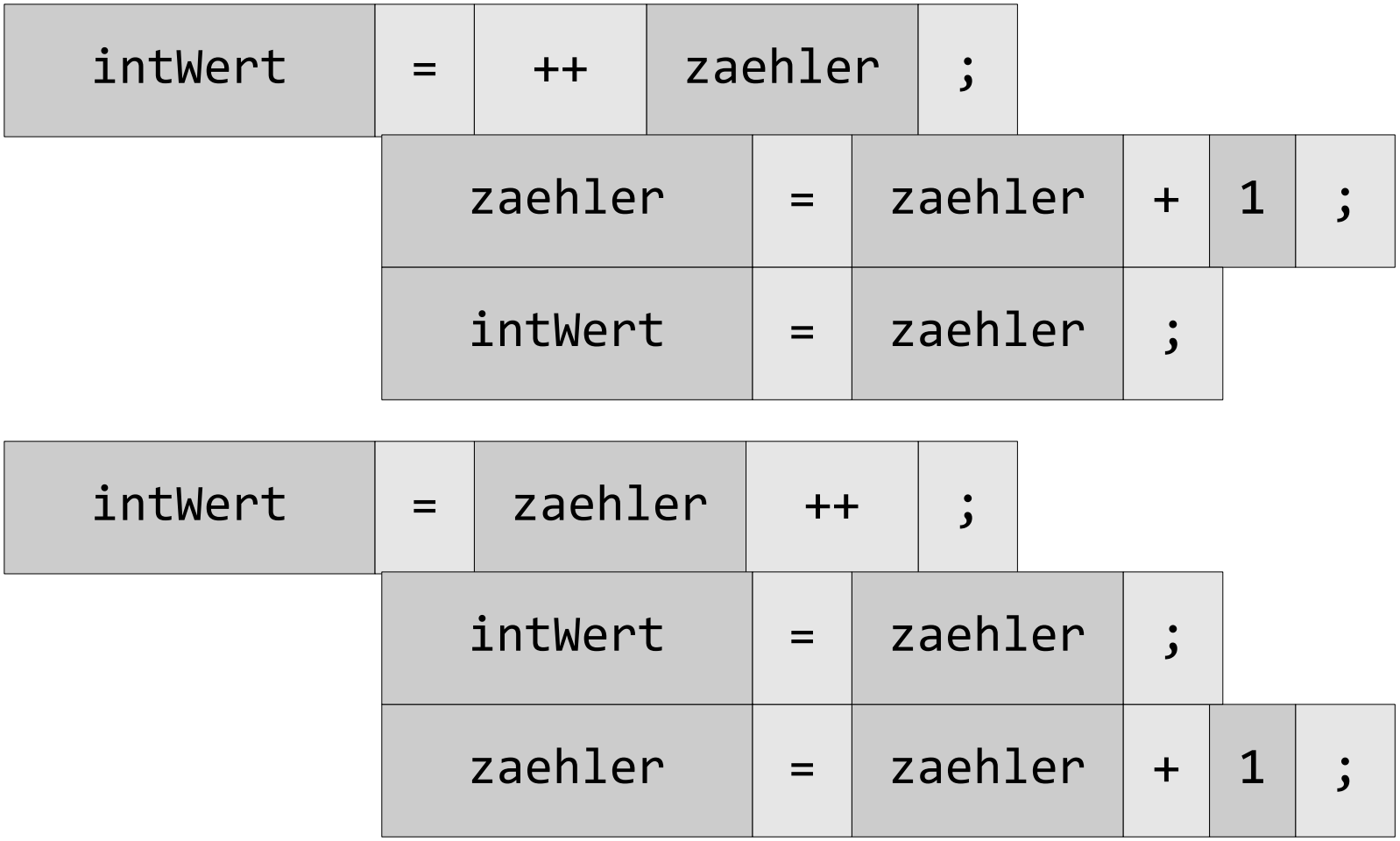
Inkrement-Operator

```
zaehler = 0;
intWert = 0;

intWert = ++zaehler;

zaehler = 0;
intWert = 0;

intWert = zaehler++;
```

Inkrementieren von Variablen

- Zwei Pluszeichen bilden den Inkrement-Operator.
- Der Operator ++ ist eine Kurzschreibweise von „+ 1“.
- Der Operator kann vor oder nach der Variablen stehen. Die Position beeinflusst die Zuweisung.

Präfix- und Postfix-Inkrement



Dekrement-Operator

```
zaehler = 10;
intWert = 10;

intWert = --zaehler;

zaehler = 10;
intWert = 10;

intWert = zaehler--;
```

Dekrementieren von Variablen

- Zwei Minuszeichen bilden den Dekrement-Operator.
- Der Operator -- ist eine Kurzschreibweise von „- 1“.
- Der Operator kann vor oder nach der Variablen stehen. Die Position beeinflusst die Zuweisung.

Präfix- und Postfix-Dekrement

```
intWert = --zaehler;
```

```
zaehler = zaehler - 1;
intWert = zaehler;
```

```
intWert = zaehler--;
```

```
intWert = zaehler;
zaehler = zaehler - 1;
```

Nutzung von relationalen Operatoren

- Vergleiche von zwei Operanden.
- Beantwortung von Fragen, auf die mit Ja oder Nein geantwortet werden kann.
- Haben Eigenschaften die Ausprägung? Zum Beispiel: Die hat Katze eine graue Fellfarbe. Die Aussage kann durch Sehen überprüft werden. Falls die Katze eine schwarze Fellfarbe hat, ist die die Aussage falsch.

Relationale Operatoren (Vergleichsoperatoren)

Operator	Berechnung	Beispiel
==	ist gleich	4 == 5 → falsch
!=	ungleich	4 != 5 → true
<	kleiner als	4 <5 → true
<=	kleiner gleich als	4 <= 5 → true
>	größer als	4 > 5 → false
>=	größer gleich als	4 >= 5 → false

Beispiel

```
int intZahlL = 3;
int intZahlR = 5;
boolean blnErgebnis;

blnErgebnis = (intZahlL == intZahlR);
blnErgebnis = (intZahlL != intZahlR);
blnErgebnis = (intZahlL < intZahlR);
blnErgebnis = (intZahlL <= intZahlR);
blnErgebnis = (intZahlL > intZahlR);
blnErgebnis = (intZahlL >= intZahlR);
```

Hinweise

- Zusammengesetzte Operatoren wie ==, <= und so weiter dürfen nicht durch ein Leerzeichen getrennt werden.
- Der Zuweisungsoperator = darf nicht mit dem Operator „ist gleich“ == verwechselt werden.
- Mit Hilfe von Klammern kann die Lesbarkeit des Ausdrucks erhöht werden.
- Gleitkommazahlen nähren sich einem Wert an. Aus diesen Grund sollte eine Überprüfung auf Gleichheit vermieden werden.

Verknüpfungsoperatoren

Operator	Erläuterung
x || y	x ODER y. Einer der beiden Ausdrücke muss wahr sein. Wenn x true ist, wird y nicht mehr ausgewertet.
x && y	x UND y Beide Ausdrücke müssen wahr sein. Wenn x false ist, wird y nicht mehr ausgewertet.
!x	NICHT x Wahr wird zu falsch und umgekehrt.

Beispiel

```
char cZeichen;
int intZahl;
boolean blnErgebnis;

intZahl = 5;
blnErgebnis = (!(intZahl > 0));
blnErgebnis = ((intZahl >= 0) && (intZahl <= 10));

cZeichen = 'Q';
blnErgebnis = ((cZeichen == 'q') ||
               (cZeichen == 'Q'));
```

Negation

```
int intZahl;
boolean blnErgebnis;

intZahl = 5;
blnErgebnis = (!(intZahl > 0));
blnErgebnis = (intZahl < 0);
```

- Falsch wird zu wahr und umgekehrt.
- Die Negation kann häufig durch eine „Umkehrung“ der Operatoren ersetzt werden. In diesem Beispiel wird der „ist größer“-Operator durch den „ist kleiner“-Operator ersetzt.

UND-Verknüpfung

```
int intZahl;
boolean blnErgebnis;

intZahl = 5;
blnErgebnis = ((intZahl >= 0) && (intZahl <= 10));
```

- Der linke sowohl als auch der rechte Ausdruck müssen wahr sein.
- Sobald einer der Ausdrücke falsch ist, ist auch der Gesamtausdruck falsch.

ODER-Verknüpfung

```
char cZeichen;
boolean blnErgebnis;

cZeichen = 'Q';
blnErgebnis = ((cZeichen == 'q') ||
               (cZeichen == 'Q'));
```

- Einer der beiden Ausdrücke muss wahr sein.

Prioritäten (Rangfolge)

Priorität	Operator
1	Nicht ! Inkrement ++, Dekrement --
2	Multiplikation * Division / Modulo %
3	Addition + Subtraktion -
4	Kleiner <, Kleiner Gleich <= Größer >, Größer gleich >=
5	ist Gleich ==, ist ungleich !=
6	Und &&
7	Oder ||

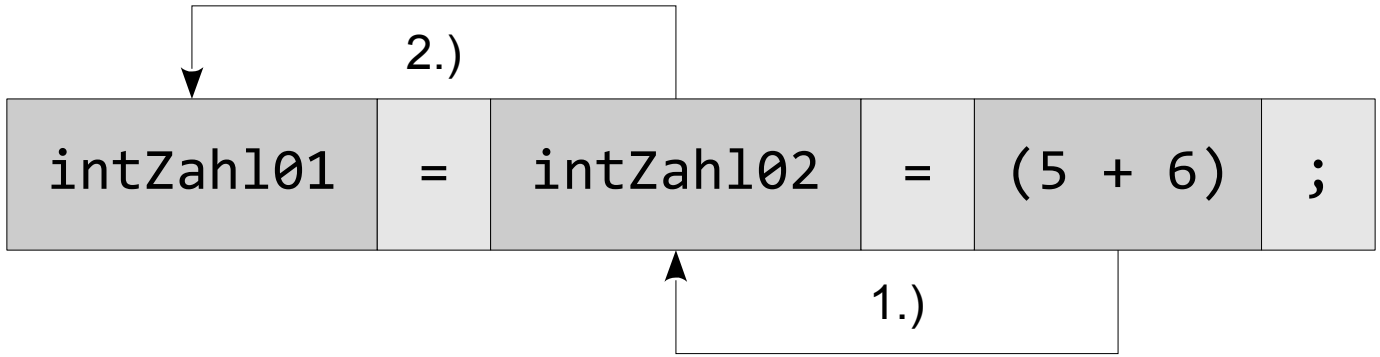
Beispiel

```
double dblZahl = 0.3;
int intZahl = 4;
double ergebnis = 0;

ergebnis = dblZahl * dblZahl + 2 * dblZahl * intZahl +
           intZahl * intZahl;
ergebnis = (dblZahl * dblZahl) + (2 * dblZahl * intZahl) +
           (intZahl * intZahl);

ergebnis = (dblZahl * (dblZahl + 2) * dblZahl * intZahl) +
           (intZahl * intZahl);
```

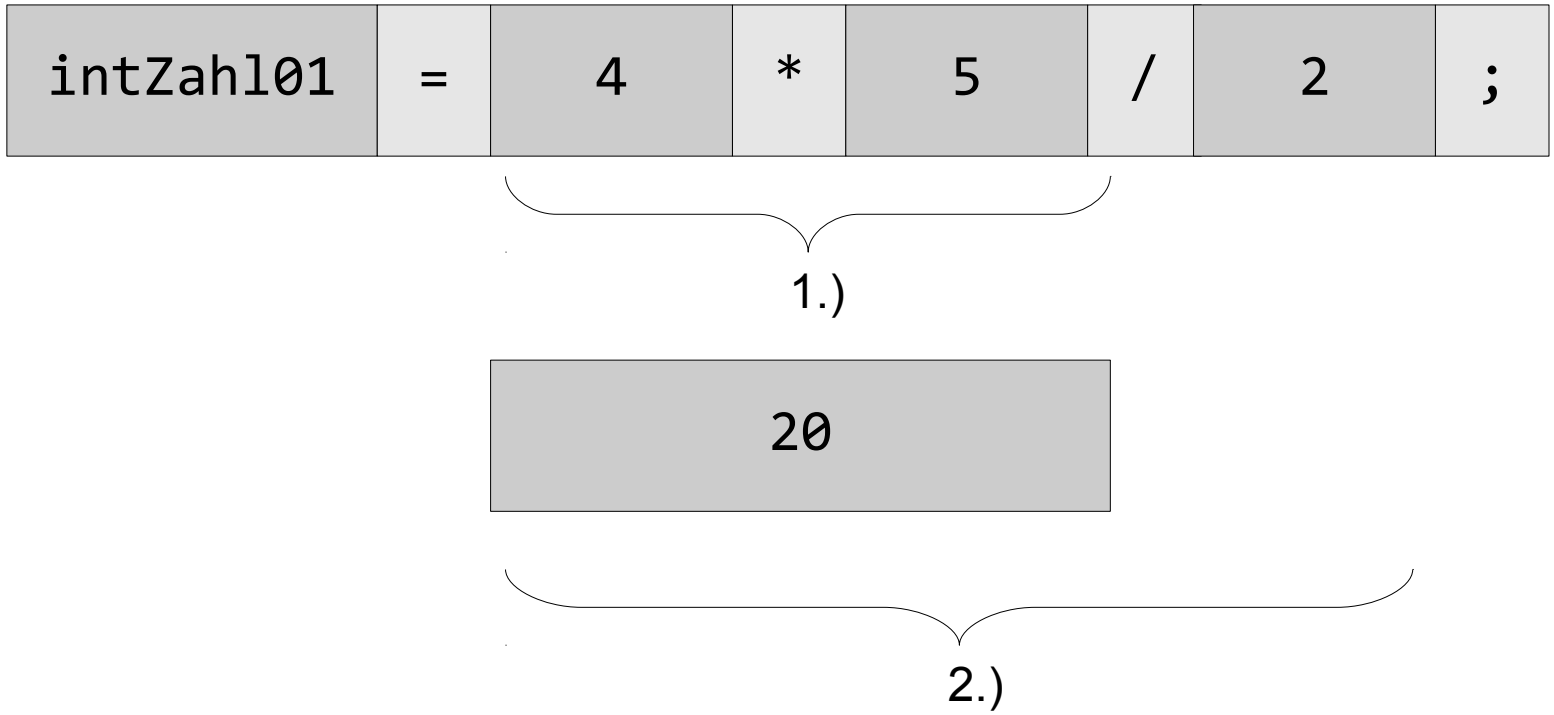
Assoziativität

Assoziativität	Operator
→	Inkrement ++, Dekrement – Multiplikation * Division Modulo % Addition + Subtraktion - Kleiner <, Kleiner Gleich <= Größer >, Größer gleich >= ist gleich ==, ist nicht gleich != Und &&, Oder ||
←	Nicht ! Inkrement ++, Dekrement -- Zuweisung =

Rechts-assoziative Verknüpfung



Links-assoziative Verknüpfung



Schleifen

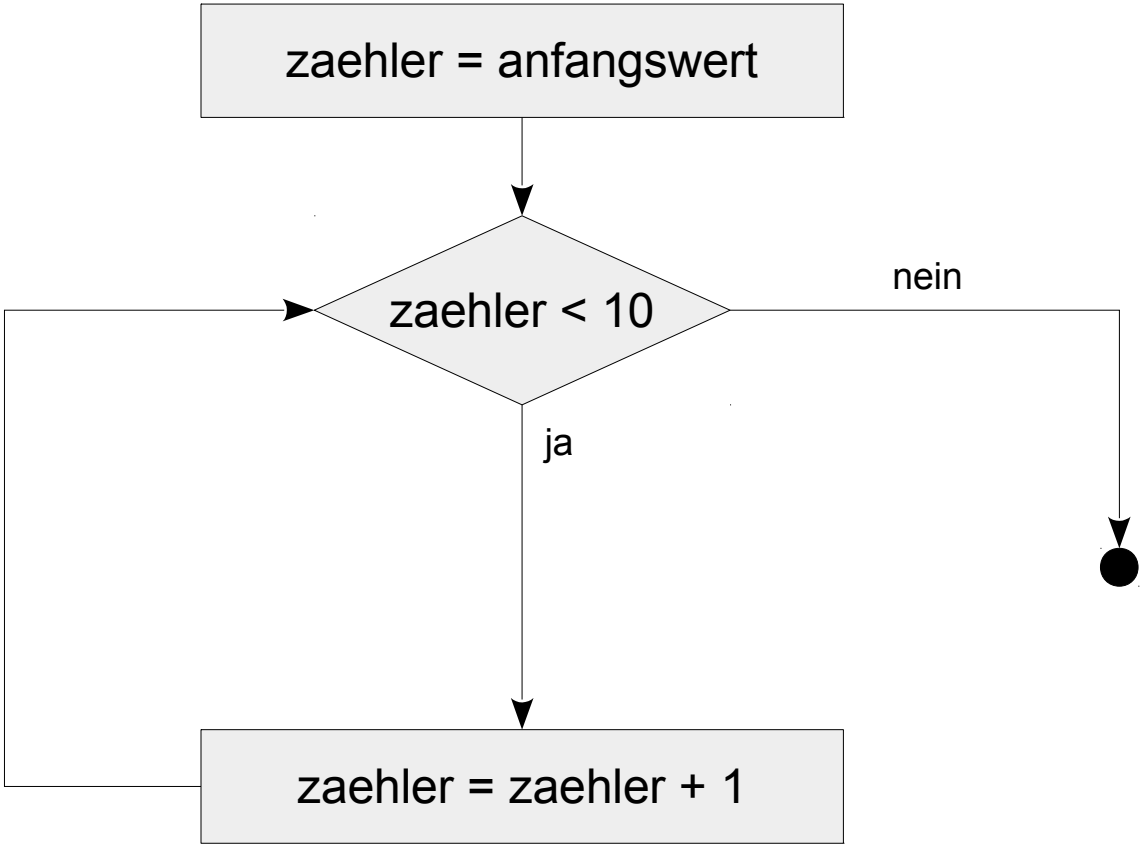
- Iterationsanweisungen.
- Kopf- oder fußgesteuerte Schleifen. Die Anzahl der Schleifendurchläufe ist abhängig von einer Bedingung.
- Zählschleifen. Die Anzahl der Durchläufe ist exakt festgelegt.
- Schleifen können vorzeitig abgebrochen werden.
- Schleifen können verschachtelt werden.

Kopfgesteuerte Schleife

```
int zaehler;
zaehler = 0;

while (zaehler < 10)
{
    System.out.println(zaehler);
    zaehler++;
}
```

Aufbau

Ablauf

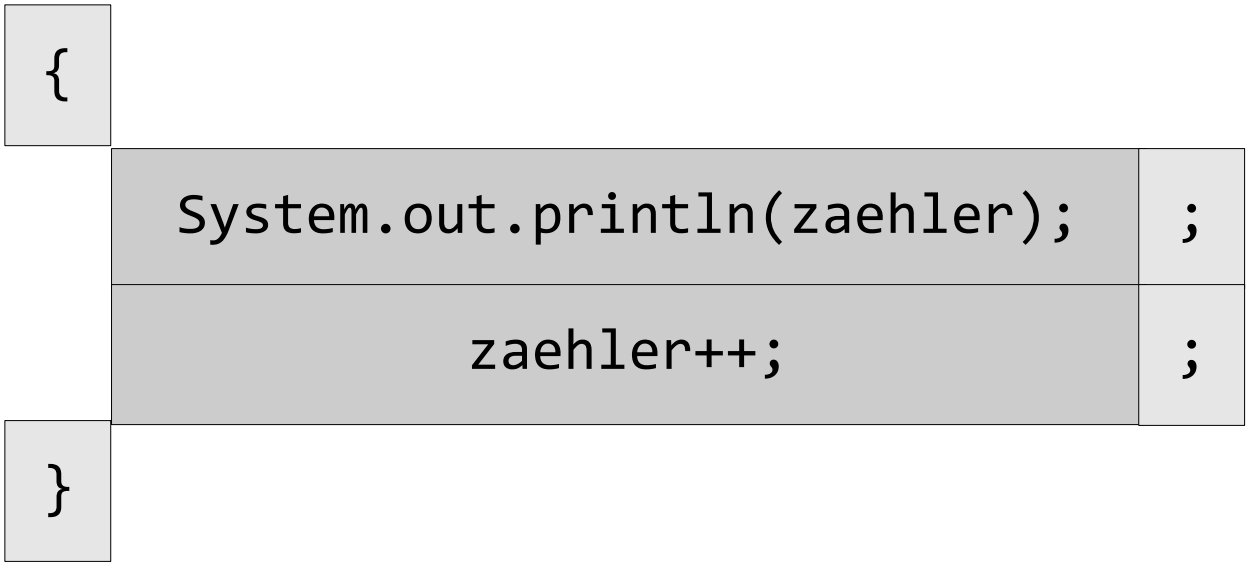


Kopf der Schleife

`while`	`(`	`Bedingung`	`)`
`while`	`(`	`zaehler < 10`	`)`

- Der Schleifenkopf beginnt mit dem Schlüsselwort `while`.
- Dem Schlüsselwort folgt in runden Klammern die Bedingung.
- Bevor der Schleifenrumpf durchlaufen wird, wird die Bedingung überprüft. Solange die Bedingung wahr ist, solange werden die Anweisungen im Rumpf der Schleife ausgeführt.

Schleifenrumpf



Erläuterung

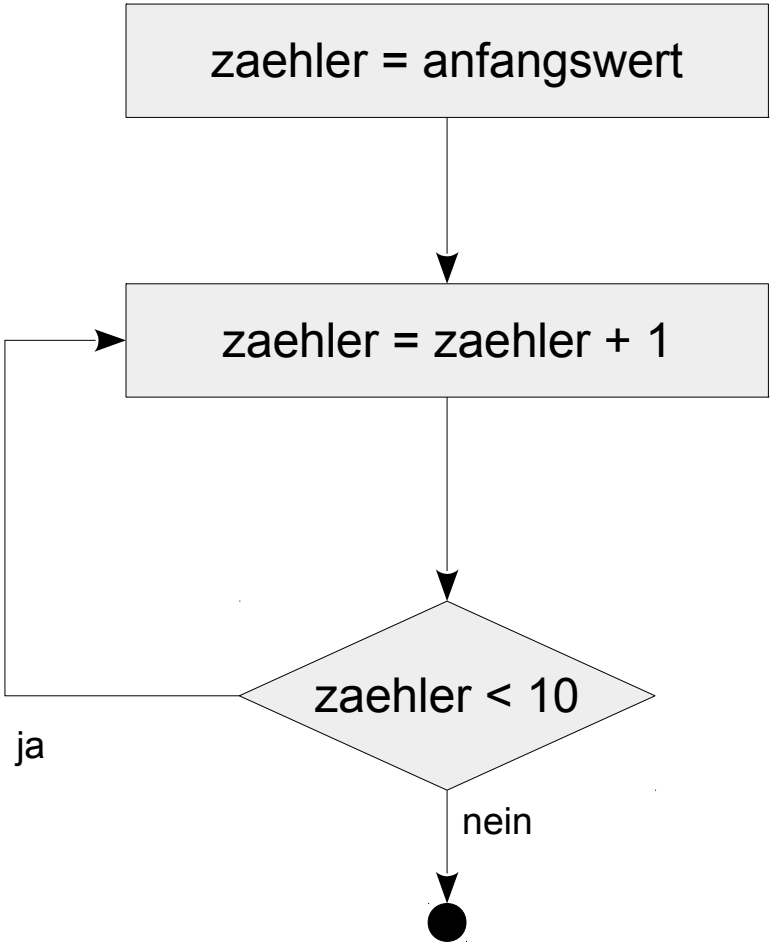
- Der Schleifenrumpf beginnt und endet mit den geschweiften Klammern. Dem Schleifenkopf folgt ein Anweisungsblock.
- Der Anweisungsblock wird solange ausgeführt, wie die Bedingung wahr ist.
- In dem Anweisungsblock muss mindestens eine der Variablen aus der Bedingung so manipuliert werden, dass die Schleife abbricht. Andernfalls wird eine Endlosschleife erzeugt.

Fußgesteuerte Schleife

```
int zaehler;
zaehler = 0;

do
{
    System.out.println(zaehler);
    zaehler++;
}
while (zaehler > 10);
```

Aufbau

Ablauf



Kopf der Schleife

do	{

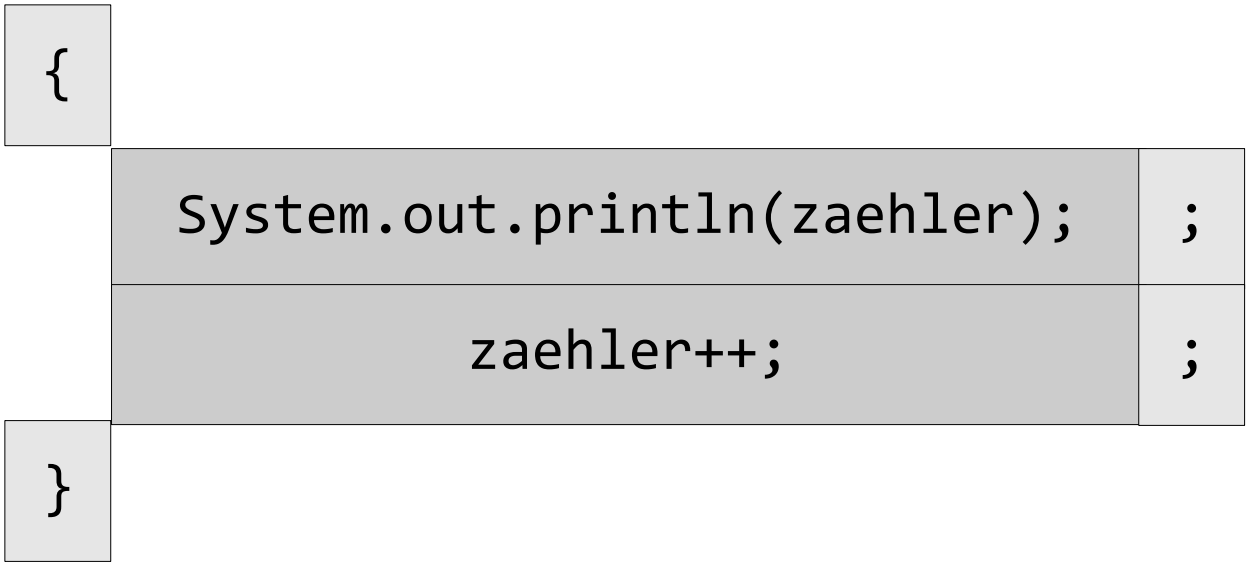
- Der Schleifenkopf beginnt mit dem Schlüsselwort do.
- Dem Schlüsselwort folgt direkt der Schleifenrumpf.
- Der Schleifenrumpf wird mindestens einmal durchlaufen.

Fuß der Schleife

`while`	`(`	`Bedingung`	`)`	`;`
`while`	`(`	`zaehler < 10`	`)`	`;`

- Der Schleifenfuß beginnt mit dem Schlüsselwort `while`.
- Dem Schlüsselwort folgt in runden Klammern die Bedingung.
- Bevor der Schleifenrumpf nochmals durchlaufen wird, wird die Bedingung überprüft. Wenn die Bedingung wahr, wird der Schleifenrumpf durchlaufen.
- Die Anweisung im Fuß der Schleife muss mit einem Semikolon beendet werden.

Schleifenrumpf



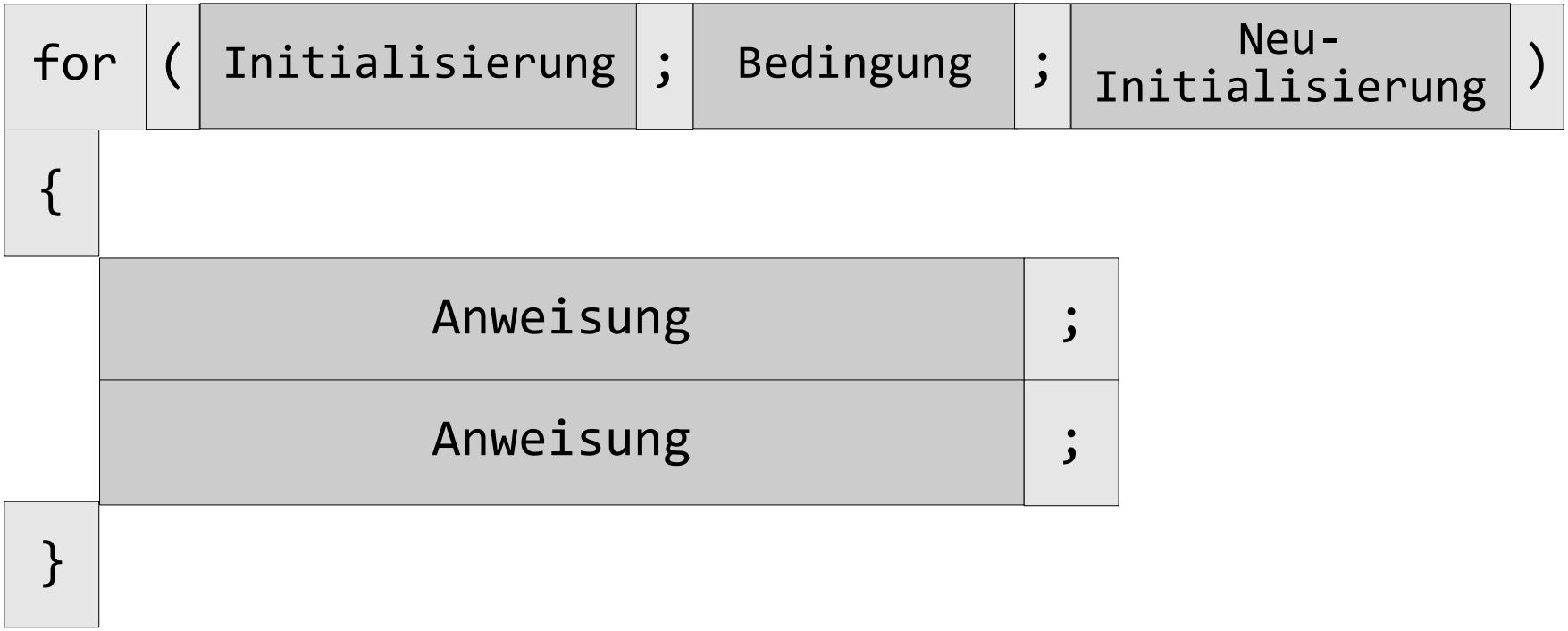
Erläuterung

- Der Schleifenrumpf beginnt und endet mit den geschweiften Klammern. Dem Schleifenkopf folgt ein Anweisungsblock.
- Der Anweisungsblock wird mindestens einmal ausgeführt.
- In dem Anweisungsblock muss mindestens eine der Variablen aus der Bedingung so manipuliert werden, dass die Schleife abbricht. Andernfalls wird eine Endlosschleife erzeugt.

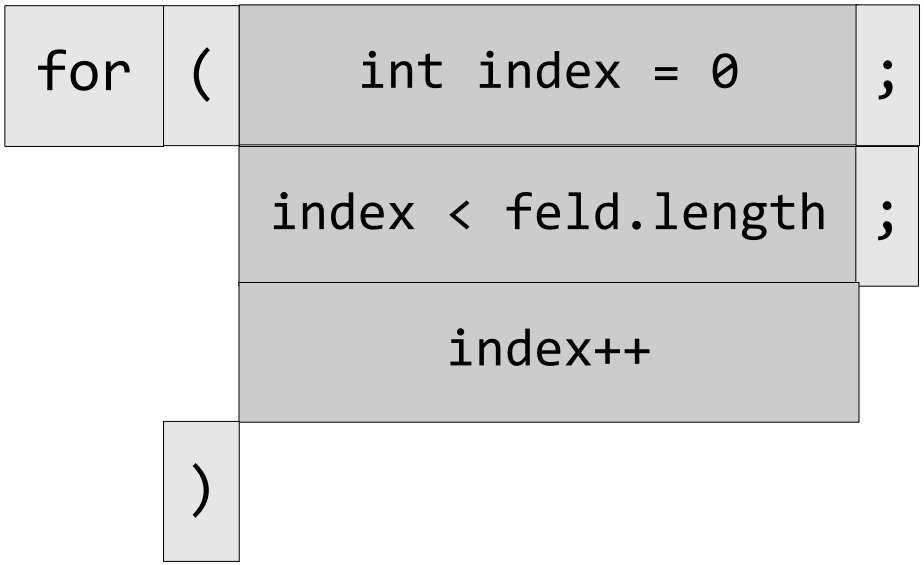
Zählschleife

```
int[] feld;

feld = new int[10];

for(int index = 0; index < feld.length; index++)
{
   feld[index] = index;
}
```

Aufbau


for
(
Initialisierung
;
Bedingung
;
Neu-
Initialisierung
)
{
Anweisung
;
Anweisung
;
}

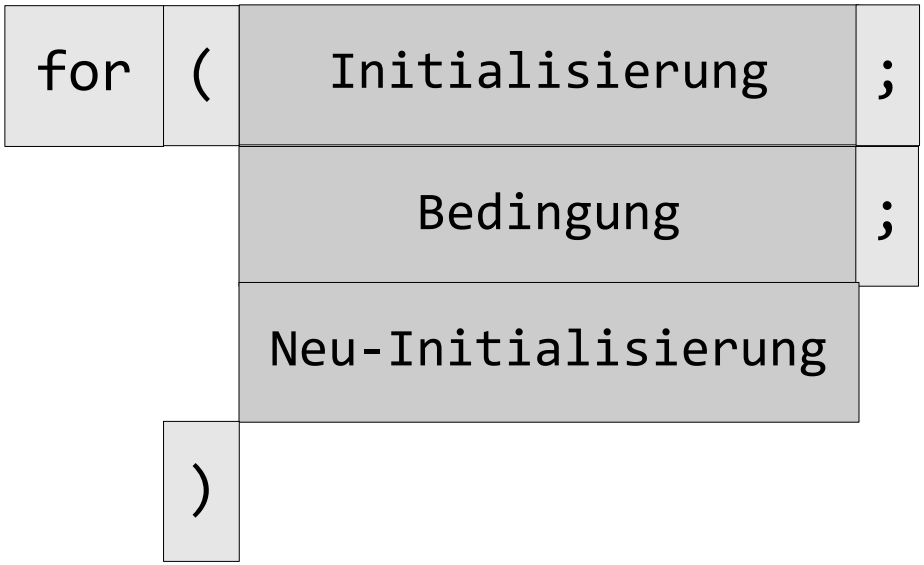
Schleifenkopf



Erläuterung

- Der Schleifenkopf beginnt mit dem Schlüsselwort `for`.
- Dem Schlüsselwort folgen drei Anweisungen. Die drei Anweisungen werden mit Hilfe der runden Klammern zusammengefasst.
- Jede der Anweisungen endet mit einem Semikolon. Die letzte Anweisung wird automatisch „beendet.“

Anweisungen im Schleifenkopf



- Die erste Anweisung kann eine Zählvariable deklarieren und gleichzeitig initialisieren.
- Die zweite Anweisung definiert eine Bedingung zum Abbruch der Schleife.
- Die dritte Anweisung berechnet einen neuen Wert für die Zählvariablen.

1. Anweisung: Deklarieren und initialisieren

`for`	`(`	`Initialisierung`	`;`
`for`	`(`	`int index = 0`	`;`

- Die Zählvariable wird im Kopf der Schleife gleichzeitig deklariert und initialisiert.
- Die Variable `index` ist nur im Rumpf der Schleife gültig. Die Variable kann nicht außerhalb des Schleifenrumpfs genutzt werden.
- Die Zählvariable hat innerhalb der Schleife einen eindeutigen Namen.

1. Anweisung: Initialisieren

`for`	`(`	`Initialisierung`	`;`
`for`	`(`	`index = 0`	`;`

- Die Zählvariable wird im Kopf der Schleife initialisiert.
- Der Variablen `index` wird ein Anfangswert zugewiesen.
- Die Variable `index` ist am Anfang der Methode deklariert. Die Variable kann in der Methode genutzt werden.

1. Anweisung: Leere Anweisung

`for`	`(`	`Initialisierung`	`;`
`for`	`(`		`;`

- Die Variable `index` ist am Anfang der Methode deklariert. Die Variable kann in der Methode genutzt werden.
- Die Variable bekommt einen eindeutigen Wert oberhalb des Schleifenkopfs zugewiesen.

2. Anweisung: Bedingung im Kopf der Schleife

for	(	Initialisierung	;	Bedingung	;
for	(	int index = 0	;	index < feld.length	;

- Die Bedingung zum Abbruch der Schleife wird als zweite Anweisung angegeben.
- In dieser Anweisungen können auch Bedingungen miteinander verknüpft werden.
- Solange die Bedingung erfüllt ist, wird der Schleifenrumpf durchlaufen.

Weitere Möglichkeit

```
int[] feld = new int[10];
int zaehler;

zaehler = feld.length - 1;

for(;;zaehler--)
{
    if(zaehler >= 0){
        System.out.println(feld[zaehler]);
    }
}
```

3. Anweisung: Neu-Initialisierung

for	(	Initialisierung	;	Bedingung	;	Neu-Initial.	)
for	(	int index = 0	;	index < 10	;	index++	)

- Nach jedem Schleifendurchlauf wird der Wert der Zählvariablen entsprechend der 3. Anweisung neu berechnet.
- Die 3. Anweisung kann komplizierte arithmetische Ausdrücke enthalten.

Weitere Möglichkeit

```
int[] feld = new int[10];
int zaehler;

zaehler = feld.length - 1;

for(;;)
{
    if(zaehler >= 0){
        System.out.println(feld[zaehler]);
        zaehler--;
    }
}
```

Schachtelung von Schleifen

```
int[][] feld = new int[10][];

for(int zeile = 0; zeile < feld.length; zeile++)
{

  for(int spalte = 0; spalte < feld[zeile].length; spalte++)
  {
    feld[zeile][spalte] = zeile + spalte;
    System.out.println(feld[zeile][spalte]);
  }

}
```

Länge eines Arrays

- `feldname.length` gibt die Länge der ersten Dimension eines Arrays zurück.
- `feldname[zeile].length` gibt die Länge der aktuellen Dimension eines Arrays zurück.

Vorzeitiger Abbruch von Schleifen

```
int teilbarOhneRest = 0;

for(int divident = 101;divident < 200; divident++)
{
    teilbarOhneRest = divident % 2;

    if (teilbarOhneRest == 0)
    {
        System.out.println(divident);
        break;
    }
}
```

Schlüsselwort break

- Beendigung eines Falls in einer switch-Anweisung.
- In einem Schleifenrumpf. Die dazugehörige Schleife wird abgebrochen.

Beispiel

```
int teilbarOhneRest = 0;

for(int divident = 101; divident < 200; divident++ )
{
   for(int divisor = 2; divisor < 10; divisor++ )
  {
    teilbarOhneRest = divident % divisor;

    if (teilbarOhneRest == 0){
        break;
    }
  }
}
```

Abbruch und Beginn neuer Schleifendurchlauf

```
int teilbarOhneRest = 0;

for(int divident = 1; divident < 10; divident++ )
{
  teilbarOhneRest = divident % 2;

  if (teilbarOhneRest == 0){
       continue;
  }

  System.out.println(divident);
}
```

Schlüsselwort continue

- Der aktuelle Schleifendurchlauf wird beendet. Alle nachfolgenden Anweisungen werden nicht ausgeführt.
- Die Zählvariable wird neu berechnet. Ein neuer Schleifendurchlauf beginnt.

Für jedes Element ...

```
char feld[] = {'a', 'b', 'c', 'd'};

for(int index = 0; index < feld.length; index++)
{
   System.out.println(feld[index]);
}

for(char zeichen : feld)
{
  System.out.println(zeichen);
}
```

Kopf der foreach-Schleife

`for`	`(`	`Datentyp`	`variablenname`	`:`	`feldname`	`)`
`for`	`(`	`char`	`zeichen`	`:`	`feld`	`)`

- Für jedes Element in einem Array ...
- Die Variable links vom Doppelpunkt symbolisiert ein Element aus dem Array rechts vom Doppelpunkt.
- Die Variable und das Array sind vom gleichen Datentyp.

… entspricht der for-Schleife

```
char zeichen = ' ';

for(int index = 0; index < feld.length; index++)
{
    zeichen = feld[index];
    System.out.println(zeichen);
}
```

Hinweise

- Der Variable links vom Doppelpunkt wird der Wert in dem aktuellen Feldelement zugewiesen.
- Durch die Änderung des Variablenwertes wird nicht der Wert des aktuellen Feldelements geändert.
- Durch die foreach-Schleife wird lesend auf ein Feldelement zugegriffen. Ein Schreibzugriff über die Variablen ist nicht möglich.

Mehrdimensionale Arrays

```
int[][] feld = {{1,2,3}, {4,5,6}};

for(int index = 0; index < feld.length; index++){

     for(int zahl : feld[index]){
         System.out.println(zahl);
     }
}
```

Erläuterung

- Die foreach-Schleife kann nur auf eindimensionale Arrays zugreifen.
- In dem Beispiel wird mit Hilfe einer for-Schleife die erste Dimension durchlaufen. Die zweite Dimension wird als „eindimensionales“ Array betrachtet.
- Die aktuelle Dimension kann als eindimensionales Array mit Hilfe einer foreach-Schleife durchlaufen werden.