

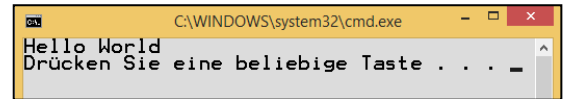
## Inhaltsverzeichnis

	Seite
1. Das erste Programm: "Hello World"	2
2. Elementare Datentypen, Variablen und Konstanten	2
3. Kommentare	3
4. Formatierung von Konsolenausgaben	4
5. Realisierung von Eingaben mittels Tastatur	4
6. Mathematische Operatoren	5
7. Vergleichsoperatoren	6
8. Logische Operatoren	6
9. Programmablaufplan und Struktogramm	7
10. Die bedingte Entscheidung (if-else-Struktur)	8
11. Die Mehrfachentscheidung (Mehrfachfachauswahl, switch-case-Struktur)	11
12. Schleifen (Wiederholung, Iteration)	12
12.1 Die fußgesteuerte Schleife	12
12.2 Die kopfgesteuerte Schleife	13
12.3 Die Zählschleife	14
12.4 Ergänzende Hinweise für alle Schleifentypen	15
13. Eindimensionale Arrays	16
14. Mehrdimensionale Arrays	17
15. Suchalgorithmen (lineare und binäre Suche)	19
16. Sortieralgorithmen	20
17. Bitoperationen	22
18. Funktionen / Methoden	wird demnächst ergänzt

## 1. Das erste Programm: "Hello World"

Die linke Abbildung zeigt ein Programm, welches "Hello World" auf der Konsole (Abbildung rechts) ausgibt.

```
1
2 public class MeinProgramm
3 {
4     public static void main (String args[])
5     {
6         System.out.println("Hello World");
7     }
8 }
```



Ein Javaprogramm besteht aus mindestens einer Klasse. Auf Zeile 2 wird der Klassenname "MeinProgramm" definiert. Der Inhalt dieser Klasse beginnt mit der öffnenden geschweiften Klammer auf Zeile 3 und endet mit der schließenden geschweiften Klammer auf Zeile 8. Jedes Javaprogramm enthält genau eine Main-Methode, welche auf Zeile 4 definiert ist. Der Inhalt dieser Methode befindet sich ebenfalls innerhalb eines geschweiften Klammerpaares (Zeilen 5 bis 7). Auf Zeile 6 wird mit Hilfe der Methode `System.out.println` die Ausgabe des Strings (Zeichenkette) "Hello World" auf der Konsole realisiert. Es handelt sich hier um eine Anweisung, welche stets mit einem Semikolon abgeschlossen werden muss.

## 2. Elementare Datentypen, Variablen und Konstanten

Für die meisten Programme gilt das EVA-Prinzip. EVA steht für Eingabe, Verarbeitung und Ausgabe von Daten. Diese Daten müssen während der Laufzeit des Programms im Arbeitsspeicher (RAM) des Computers zwischengespeichert werden. Das direkte Speichern der Daten als binärer Wert (Folge von Nullen und Einsen, z.B. 10110001) im RAM wäre extrem umständlich. Darum muss sich aber bei Java kein Programmierer kümmern, sondern er verwendet für die Speicherung Variablen. Variablen besitzen einen Datentyp und einen Namen (auch Bezeichner genannt, engl.: Identifier). Möchte man z. B. ganzzahlige numerische Werte (ganze Zahlen) speichern, so kann der Datentyp Integer verwendet werden.

Programmbeispiel mit  
 nachfolgender Erläuterung:

```
2 public class MeinProgramm
3 {
4     public static void main (String args[])
5     {
6         int zahl;
7         zahl = 3;
8         System.out.println(zahl);
9     }
10 }
```

Auf Zeile 6 wird eine Variable mit dem Namen "zahl" deklariert. Das Schlüsselwort `int` legt fest, dass die Variable den Datentyp Integer (Ganzzahl) besitzen soll. Auf Zeile 7 wird der Variablen `zahl` der Wert 3 zugewiesen. Eine Ausgabe des Inhalts der Variablen `zahl` erfolgt auf Zeile 8.

Die Zeilen 6 und 7 können auch zusammengefasst werden:

```
int zahl = 3;
```

Damit wird die Variable `zahl` gleichzeitig deklariert und initialisiert. Diese Zusammenfassung ist auch bei Verwendung anderer Datentypen möglich.

Es folgt eine Übersicht zu einigen wichtigen elementaren (primitiven) Java-Datentypen:

Typ (Schlüsselwort)	Größe in Byte	Wertebereich
byte	1	Ganzzahl -128 bis 127
short	2	Ganzzahl -32.768 bis 32.767
int	4	Ganzzahl -2.147.483.648 bis 2.147.483.647
long	8	Ganzzahl -9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807
float	4	Fließkommazahl 1,40239846E-45 bis 3,40282347E+38
double	8	Fließkommazahl mit doppelter Genauigkeit : 4,94065645841246544E-324 bis 1,79769131486231570E+308
boolean	1	Logischer Wert (Wahrheitswert): <code>true</code> oder <code>false</code>
char	2	Einzelnes Unicode Zeichen (engl.: character)

Es folgen einige Beispiele bzgl. Deklaration von Variablen und Wertzuweisungen:

- Deklarieren von mehreren gleichartigen Variablen: `int zahl1, zahl2, zahl3;`
- Character-Variable deklarieren, anschließend Wert zuweisen:  
(Zeichen zwischen Hochkommata einschließen) `char b;`  
`b = 'k';`
- Boolean-Variable deklarieren, anschließend Wert zuweisen: `boolean c;`  
`c = true;`
- Float-Variable deklarieren, anschließend Wert zuweisen:  
(Wird der Buchstabe `f` nicht an die Zahl angehängt, so entsteht beim Kompilieren eine Fehlermeldung, weil die Zahl automatisch als Double-Wert interpretiert wird.) `float a;`  
`a = 0.815f;`

Möchte man einen String (Zeichenkette) speichern, so steht dafür kein elementarer Datentyp zur Verfügung. Aber man kann die im JDK enthaltene Klasse `String` verwenden (Großschreibung beachten). Beispiel:

Programmcode:  
(Ausschnitt)

```
String str1;  
str1 = "Java";  
System.out.println(str1);  
  
String str2 = "Sumatra";  
System.out.println(str2);
```

Ausgabe:

```
Java  
Sumatra
```

Man kann einer Variablen auch einen Wert zuweisen, welcher in einer anderen Variablen gespeichert ist. Im Beispiel rechts werden die Variablen `a` und `b` deklariert, dabei erhält `a` den Wert 4. Anschließend wird der in `a` gespeicherte Wert in `b` gespeichert.

```
int a = 4, b;  
b = a;
```

Innerhalb eines Programms können Variablen beliebig oft überschrieben werden. Falls das verhindert werden soll, so muss das Schlüsselwort `final` verwendet werden.

Beispiel: `final double PI = 3.14;` Damit ist `PI` keine Variable, sondern eine Konstante.

Die Namen von Klassen, Variablen und Konstanten werden auch Bezeichner (engl.: Identifier) genannt. Für diese Bezeichner gelten Standards, welche unbedingt einzuhalten sind. Entsprechende Informationen sind im Internet leicht zu finden oder/und werden Ihnen von der Lehrkraft zur Verfügung gestellt.

### 3. Kommentare

Kommentare erleichtern die Lesbarkeit von Programmen für andere Personen. Das gilt auch für den Autor selbst, wenn er seinen eigenen Programmcode nach einiger Zeit verändern oder erweitern muss. Im abgebildeten Beispielprogramm sind einzeilige Kommentare (z. B. Zeile 13) und ein mehrzeiliger Kommentar (Zeilen 2 bis 7) enthalten. Einzeilige Kommentare müssen immer mit der Zeichenfolge `/**` eingeleitet werden. Mehrzeilige Kommentare müssen immer mit `/*` begonnen und mit `*/` beendet werden. Ohne diese Zeichen würde der Compiler die Kommentare wie Programmcode behandeln, was natürlich eine Fehlermeldung erzeugt.

```
2  /*  
3  Das illegale Kopieren dieses Programms kann mit  
4  bis zu 20 Jahren Gefängnis ohne Bewährung bei  
5  Wasser und Brot sowie anschließender Sicherheits-  
6  verwahrung bestraft werden.  
7  */  
8  
9  public class TollesProgramm  
10 {  
11     public static void main (String args[])  
12     {  
13         //Variable deklarieren und initialisieren  
14         double var = 0.815;  
15  
16         //Variableninhalt (Wert) ausgeben  
17         System.out.println(var);
```

## 4. Formatierung von Konsolenausgaben

Um die Lesbarkeit von Konsolenausgaben zu erleichtern, sollte auf eine geeignete Formatierung geachtet werden. Die bereits erwähnte Methode `System.out.println` erzeugt nach der Ausgabe einen Zeilenumbruch. Möchte man diesen vermeiden, so muss `System.out.print` verwendet werden.

Beispiel mit Ausgabe:

```
System.out.println("1");  
System.out.print("2");  
System.out.println("3");  
System.out.println(" 4");
```

```
1  
23  
4
```

Anmerkung: Eine zusätzliche Leerzeile kann man mit `System.out.println()` (leere Klammer) erzeugen.

Man kann auch die Ausgabe von Zeichenketten und Variableninhalten mit Hilfe des Operators `+` kombinieren, was hier gut zu sehen ist:

```
int minAlter = 16;  
System.out.println("Dieser Film ist für Zuschauer unter " + minAlter + " Jahren nicht geeignet.");
```

```
Dieser Film ist für Zuschauer unter 16 Jahren nicht geeignet.  
Drücken Sie eine beliebige Taste . . . _
```

Hilfreich für formatierte Ausgaben sind Escape-Sequenzen (auch Steuerzeichen genannt), welche sich innerhalb eines Strings befinden. Betrachten wir das folgende Beispiel:

```
6 int ja = 5, nein = 3, ent = 1;  
7 System.out.println("Abstimmungsergebnis:\n");  
8 System.out.println("Ja\tNein\tEnthaltungen");  
9 System.out.println(ja + "\t" + nein + "\t" + ent);
```

```
Abstimmungsergebnis:  
Ja      Nein      Enthaltungen  
5       3       1
```

Auf Zeile 7 befindet sich die Sequenz `\n`. Das Zeichen `\` bewirkt, dass das nachfolgende Zeichen (und nur dieses) nicht als Zeichen sondern als Kommando interpretiert wird. Das `n` steht für das Kommando "new line", es wird also ein Zeilenumbruch erzeugt. Weil die Methode `System.out.println` ebenfalls einen Zeilenumbruch bewirkt, entsteht nach der Ausgabe eine Leerzeile.

Auf den Zeilen 8 und 9 befindet sich die Sequenz `\t`. Das `t` steht für Tabulator (horizontal). Der Cursor springt dann 8 Stellen nach rechts. Man kann auch mehrere Escape-Sequenzen aufeinander folgen lassen, die Sequenz `\n\n\n\n\n\n\n` würde jede Menge Zeilenumbrüche und damit Leerzeilen erzeugen. Es gibt noch weitere Escape-Sequenzen, auf welche hier aber nicht näher eingegangen wird.

## 5. Realisierung von Eingaben mittels Tastatur

Das Einlesen von Daten über die Tastatur ist in Java nicht ganz einfach. Deshalb verwenden wir eine nicht zum JDK gehörende Klasse mit Namen `Tastatur`, welche Ihnen als Datei mit Namen `Tastatur.java` von der Lehrkraft zur Verfügung gestellt wird. Um diese Klasse verwenden zu können, muss sich eine Kopie der Datei `Tastatur.java` im selben Ordner befinden, in der auch Ihre Klasse liegt. Für jeden Datentyp muss eine eigene Methode verwendet werden, beispielsweise `intInput()` für Integer-Zahlen. Das folgende Programmbeispiel soll dies verdeutlichen:

```
//Variablen deklarieren  
int a;  
double b;  
String c;  
  
//Variablen mittels Tastatur füllen  
a = Tastatur.intInput(); // ganze Zahl eingeben  
b = Tastatur.doubleInput(); // Fließkommazahl eingeben  
c = Tastatur.stringInput(); // String eingeben  
  
// Ausgaben auf der Konsole.  
System.out.println(a);  
System.out.println(b);  
System.out.println(c);
```

Wie die Methoden für alle andere elementaren Datentypen heißen, ist jetzt sicherlich selbsterklärend.

## 6. Mathematische Operatoren

Für Berechnungen stehen folgende mathematische (arithmetische) Standardoperatoren zur Verfügung:

Operator	Bedeutung	Beispiel
+	Addition	$y = 3 + 4$
-	Subtraktion	$m = 7 - x$
*	Multiplikation	$n = a * b$
/	Division	$y = 10 / 2$
%	Modulo (liefert ganzzahligen Rest einer Division)	$a = 20 \% 7$
++	Inkrement	$i++$ (i wird um eins erhöht)
--	Dekrement	$i--$ (i wird um eins erniedrigt)

### 1. Beispiel

Auf den Zeilen 9 und 10 wird jeweils ein Wert berechnet und in den Variablen `a` bzw. `b` gespeichert. Dabei finden folgende Formeln Anwendung:

$$a = 4x - 5 \quad b = 2(x^2 + 1)$$

```

8      int x = 3, a, b;
9      a = 4 * x - 5;
10     b = 2 * (x * x + 1);
11     System.out.println(a);
12     System.out.println(b);
    
```

### 2. Beispiel

```

7      int zahl, teiler = 3, rest;
8      System.out.print("Geben Sie eine ganze Zahl ein: ");
9      zahl = Tastatur.readInt();
10     rest = zahl % teiler;
11     System.out.println("Rest: " + rest);
    
```

Möchte man 5 Autos unter 3 Personen gerecht aufteilen, so erhält jede Person 1 Auto und es bleibt ein Rest von 2 Autos übrig. Diese Art Division wird Modulo-Rechnung genannt. Auf der Zeile 10 wird diese Modulo-Rechnung durchgeführt, das Ergebnis (Rest) wird in der Variablen `rest` gespeichert.

```

C:\
Geben Sie eine ganze Zahl ein: 5
Rest: 2
Drücken Sie eine beliebige Taste
    
```

### 3. Beispiel

```

7      int a = 7, b = 2;
8      double y1 = a / b;
9      double y2 = (double) a / b;
10     System.out.println("y1 hat den Wert: " + y1);
11     System.out.println("y2 hat den Wert: " + y2);
    
```

```

C:\
y1 hat den Wert: 3.0
y2 hat den Wert: 3.5
    
```

Wie auf der Konsolenausgabe zu sehen ist, liefert die Division auf Zeile 8 ein falsches Ergebnis. Der Grund ist, dass die Formel Integer-Zahlen (Variablen `a` und `b`) enthält. Damit wird das Ergebnis automatisch auch eine Integer-Zahl und alles hinter dem Komma wird "abgeschnitten". Dabei spielt es keine Rolle, dass das Ergebnis in einer Variable von Typ `Double` gespeichert wird. Um dieses Problem zu lösen, muss vor der Formel "`(double)`" eingefügt werden, siehe Zeile 9. Dieses Verfahren wird Datentypumwandlung (Typecast) genannt.

### 4. Beispiel

```

7      int k = 1, m = 1, n = 1;
8      k = k + 3;
9      m = m + 1;
10     n++;
11     System.out.println("k=" + k + " m=" + m + " n=" + n);
    
```

```

C:\
k=4 m=2 n=2
    
```

Auf Zeile 7 werden die Variablen `k`, `m`, `n` deklariert und erhalten den Wert 1. Anschließend wird auf Zeile 8 zu dem in `k` gespeicherten Wert die Zahl 3 addiert, das Ergebnis wird wieder in `k` gespeichert (`k` wird also überschrieben). Etwas Ähnliches passiert auf Zeile 9, der Inhalt von `m` wird um 1 erhöht. Die gleiche Wirkung hat die Verwendung des Operators `++` auf Zeile 10, damit wird `n` um 1 erhöht.

Theoretisch können alle Berechnungen mit Hilfe der Standardoperatoren ausgeführt werden, allerdings wäre das sehr umständlich. Deshalb enthält das JDK eine Klasse mit Namen `Math`. Diese Klasse verfügt über eine Vielzahl von Methoden, von denen hier nur die Methode `sqrt` vorgestellt werden soll, welche zur Berechnung der Quadratwurzel dient. Andere Methoden können bei Bedarf der Java-Dokumentation entnommen werden, es existieren auch genügend Beschreibungen im Internet. Beispiel für die Anwendung der Methode `sqrt`:

```
double y, x = 9;
y = Math.sqrt(x);
System.out.println("Die Wurzel aus " + x + " lautet: " + y);
```

```
Die Wurzel aus 9.0 lautet: 3.0
Drücken Sie eine beliebige Taste . .
```

## 7. Vergleichsoperatoren

Operator	Bedeutung	Beispiel
<code>==</code>	gleich	<code>x == 5</code>
<code>!=</code>	ungleich	<code>x != 5</code>
<code>&lt;</code>	kleiner als	<code>x &lt; 5</code>
<code>&gt;</code>	größer als	<code>x &gt; 5</code>
<code>&lt;=</code>	kleiner als oder gleich	<code>x &lt;= 5</code>
<code>&gt;=</code>	größer als oder gleich	<code>x &gt;= 5</code>

Mit Hilfe von Vergleichsoperatoren können die Inhalte von je 2 Variablen (oder der Inhalt einer Variable mit einer Zahl) verglichen werden. Das Ergebnis eines Vergleichs ist immer ein logischer Wert (Wahrheitswert), also `true` oder `false`. Dieser Wert kann in einer Boolean-Variable gespeichert und auch ausgegeben werden, was das nachfolgende Beispiel verdeutlichen soll. Hier wird geprüft, ob eine eingegebene Zahl größer oder gleich 5 ist. Das Programm wird zweimal ausgeführt.

```
double zahl;
boolean erg;
System.out.print("Gib eine Zahl ein: ");
zahl = Tastatur.doubleInput();
erg = (zahl >= 5);
System.out.println("Ergebnis: " + erg);
```

```
Gib eine Zahl ein: 4.99
Ergebnis: false
```

```
Gib eine Zahl ein: 5
Ergebnis: true
```

Man kann auch prüfen, ob eine Variable vom Typ `char` ein bestimmtes Zeichen enthält. Im folgenden Beispiel ist das Ergebnis nur dann wahr (`true`), wenn die Variable `z` das Zeichen 'n' (Kleinschreibung beachten) enthält:

```
char z = Tastatur.charInput();
boolean erg = (z == 'n');
```

## 8. Logische Operatoren

Operator	Bedeutung	Beispiel
<code>!</code>	Negation	<code>!(a &gt; b)</code>
<code>&amp;&amp;</code>	Logisches UND	<code>(a &gt; b) &amp;&amp; (c &lt; 5)</code>
<code>  </code>	Logisches ODER	<code>(a &gt; b)    (c &lt; 5)</code>

Zusammenhänge:	false UND false = false	false ODER false = false
	false UND true = false	false ODER true = true
	true UND false = false	true ODER false = true
	true UND true = true	true ODER true = true

Häufig müssen mehrere Vergleiche kombiniert werden, dann kommen logische Operatoren zum Einsatz. Wenn zwei Bedingungen gleichzeitig erfüllt sein sollen, so verknüpft man beide Vergleiche mit einem UND. Möchte man prüfen, ob wenigstens eine von zwei Bedingungen erfüllt ist, so verwendet man das ODER.

Beispiele:

```
boolean ergebnis = ((a > 1) && (a < 5));
```

Nur wenn `a` größer als 1 und kleiner als 5 ist, liefert der Vergleich den Wert `true`. Ansonsten immer `false`.  
Für `a = 6` würde gelten: Der 1. Vergleich (`a > 1`) liefert `true` und der 2. Vergleich (`a < 5`) liefert `false`.  
Die Verknüpfung `true` UND `false` liefert `false`.

```
boolean ergebnis = ((a == 1) || (b != 1));
```

Wenn `a` gleich 1 oder `b` ungleich 1 ist, liefert der Vergleich den Wert `true`. Das gilt auch dann, wenn beide Bedingungen erfüllt sind. Der Wert `false` wird nur dann erzeugt, wenn beide Bedingungen nicht erfüllt sind.

Hinweis zur Klammersetzung:

Es ist zu bemerken, dass verschiedene Operatoren eine unterschiedliche Wertigkeit haben. Das heißt, manche Operatoren werden in einer Codezeile vor anderen ausgeführt. Um dies zu vermeiden, sind Klammern so zu setzen, dass einerseits die Befehlszeile ihren Zweck erfüllt und andererseits der Programmcode für den Programmierer lesbar bleibt. (Also lieber ein Klammerpaar zuviel, als eins zu wenig.)

Möchte man das Ergebnis eines Vergleichs bzw. den Wert einer Boolean-Variable negieren (aus `true` wird `false` und umgekehrt), so verwendet man den Operator `!`. Beispiele:

**1. Beispiel**

```
boolean b = true;           // b wird auf true gesetzt
b = !b;                     // jetzt enthält b den Wert false
```

**2. Beispiel**

```
int a = 3;
boolean b = !((a > 1) && (a < 7));
```

Beide Vergleiche liefern `true`, das Ergebnis der UND-Verknüpfung ist damit auch `true`.  
Dieses Ergebnis wird schließlich zu `false` negiert, weil sich der entsprechende Operator vor der äußeren Klammer befindet. Im folgenden Beispiel wird die äußere Klammer "vergessen":

**3. Beispiel**

```
int a = 3;
boolean b = !(a > 1) && (a < 7);
```

Der 1. Vergleich (`a > 1`) liefert `true` und wird danach zu `false` negiert. Der 2. Vergleich (`a < 7`) liefert `true`. Die Verknüpfung `false` UND `true` liefert als Ergebnis ein `false`.

## 9. Programmablaufplan und Struktogramm

Programmablaufpläne und Struktogramme dienen zur graphischen Darstellung von Programmen, ohne das dafür Kenntnisse einer bestimmten Programmiersprache erforderlich sind. Sie finden Anwendung bei der Entwicklung von Programmen.

Im weiteren Verlauf dieses Unterrichtsskripts werden parallel zu den Beispielprogrammen auch Beispiele für Programmablaufpläne und Struktogramme vorgestellt. Eine Zusammenstellung der verwendeten Symbole mit Hinweisen erhalten Sie gesondert als PDF-Dokument von Ihrer Lehrkraft.



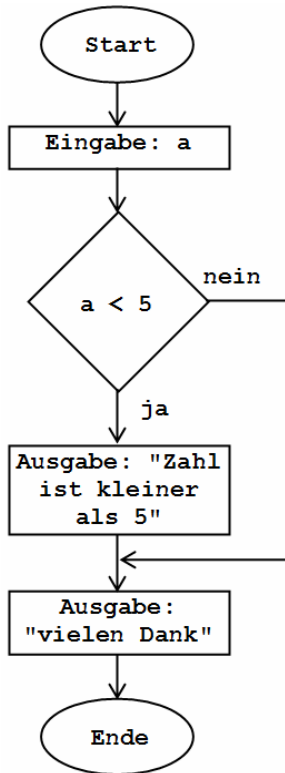
## 10. Die bedingte Entscheidung (if-else-Struktur)

Oft ist es erforderlich, dass einige Anweisungen im Programm nur unter einer bestimmten Bedingung ausgeführt werden sollen. Dafür wird die bedingte Entscheidung (if-else-Struktur) verwendet.

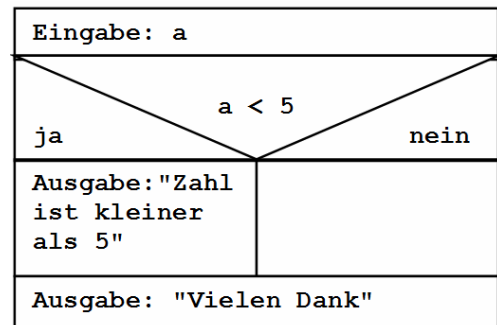
### 1. Beispiel

Nach Eingabe einer Zahl soll ein Programm feststellen, ob die Zahl kleiner als 5 ist. Falls ja, so soll ein entsprechender Text ausgegeben werden. Anschließend verabschiedet sich das Programm mit der Ausgabe "Vielen Dank". Sollte die Zahl nicht kleiner als 5 sein, so darf nur der Text "Vielen Dank" ausgegeben werden.

Programmablaufplan:



Struktogramm:



Javaprogramm:

```
3 public class Beispiell
4 {
5     public static void main (String args[])
6     {
7         int a;
8         a = Tastatur.intInput();
9
10        if(a < 5)
11        {
12            System.out.println("Zahl ist kleiner als 5");
13        }
14
15        System.out.println("Vielen Dank !");
16    }
17 }
```

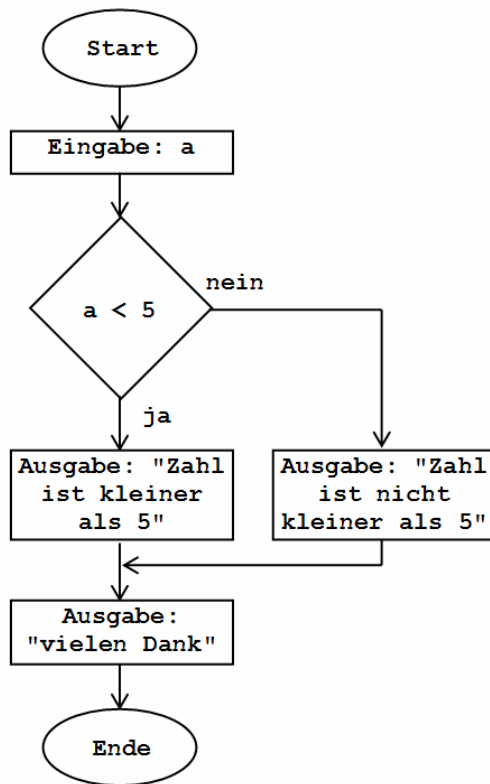
Nach Eingabe einer Zahl (Zeile 8) wird auf Zeile 10 geprüft, ob diese Zahl kleiner als 5 ist. Dafür muss hinter dem Schlüsselwort `if` innerhalb der runden Klammern eine entsprechende Bedingung formuliert sein. Ist diese Bedingung wahr, so wird ein Befehlsblock ausgeführt, welcher durch das geschweifte Klammerpaar (Zeilen 11 und 13) begrenzt ist. In diesem Beispiel besteht der Block aus einer einzigen Anweisung (Zeile 12), es können aber beliebig viele sein. Falls die Bedingung nicht erfüllt ist, wird der Block nicht ausgeführt. Die Anweisung auf Zeile 15 befindet sich nicht mehr im Block und wird deshalb immer ausgeführt.



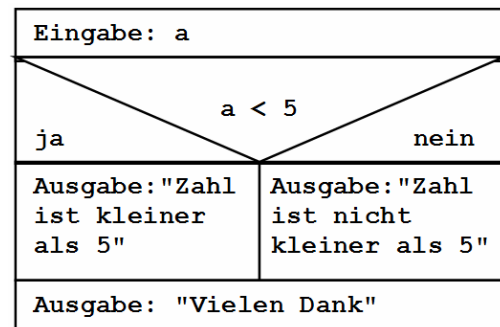
## 2. Beispiel

Das vorherige Beispiel wird jetzt um folgende Zusatzfunktion erweitert: Falls die eingegebene Zahl nicht kleiner als 5 ist, so soll auch hier ein entsprechender Text ausgegeben werden.

### Programmablaufplan:



### Struktogramm:



### Javaprogramm:

```

3 public class Beispiel2
4 {
5     public static void main (String args[])
6     {
7         int a;
8         a = Tastatur.intInput();
9
10        if(a < 5)
11        {
12            System.out.println("Zahl ist kleiner als 5");
13        }
14        else
15        {
16            System.out.println("Zahl ist nicht kleiner als 5");
17        }
18
19        System.out.println("Vielen Dank !");
20    }
21 }
    
```

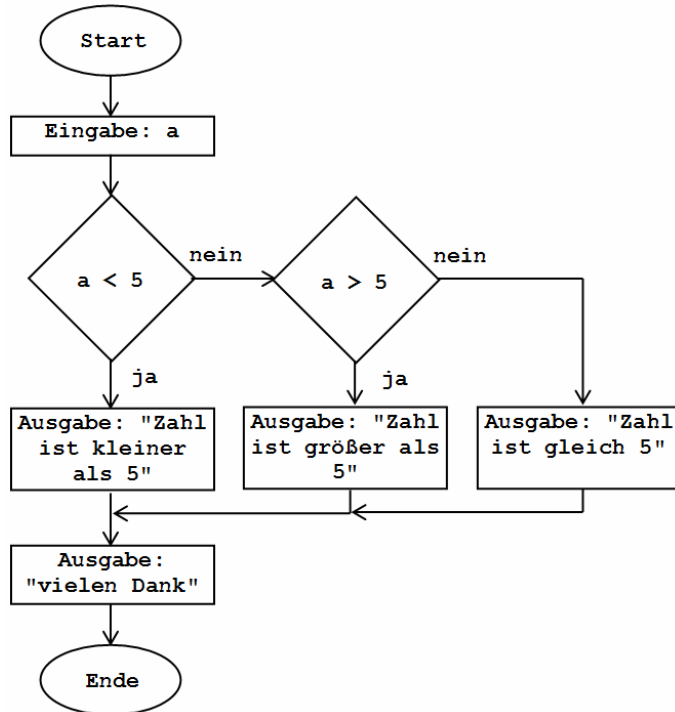
Falls die Bedingung hinter `if` nicht erfüllt wird, muss ein anderer Befehlsblock ausgeführt werden. Dieser befindet sich hinter dem Schlüsselwort `else` (Zeile 14) und wird ebenfalls durch ein geschweiftes Klammerpaar begrenzt.

Die Anweisung auf Zeile 19 gehört nicht mehr zur `if-else`-Konstruktion und wird deshalb immer ausgeführt.

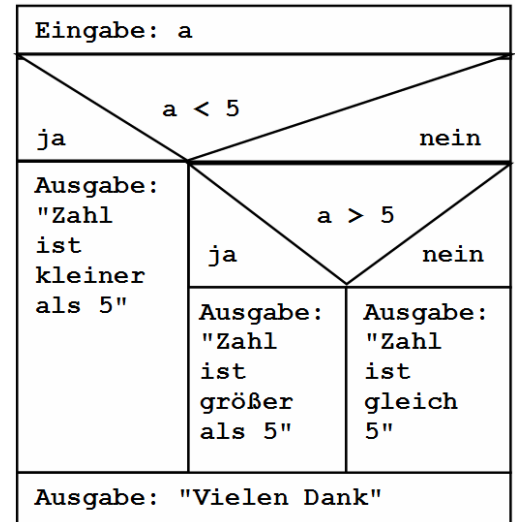
### 3. Beispiel

Das vorherige Beispiel wird jetzt um folgende Zusatzfunktion erweitert: Falls die eingegebene Zahl gleich 5 ist, so soll auch ein entsprechender Text ausgegeben werden. Damit können also 3 verschiedene Fälle auftreten. Dies erfordert mehrere bedingte Verzweigungen, welche z. B. auch ineinander geschachtelt werden können.

**Programmablaufplan:**



**Struktogramm:**



**Javaprogramm:**

```

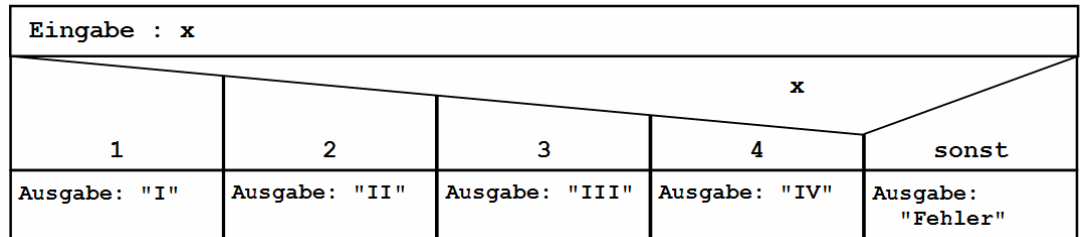
3 public class Beispiel3
4 {
5     public static void main (String args[])
6     {
7         int a;
8         a = Tastatur.intInput();
9
10        if(a < 5)
11        {
12            System.out.println("Zahl ist kleiner als 5");
13        }
14        else if(a > 5)
15        {
16            System.out.println("Zahl ist größer als 5");
17        }
18        else
19        {
20            System.out.println("Zahl ist gleich 5");
21        }
22
23        System.out.println("Vielen Dank !");
24    }
25 }
    
```

Falls die Bedingung hinter `if` (Zeile 10) nicht erfüllt ist, erfolgt hinter `else` (Zeile 14) eine zweite Prüfung mit einer anderen Bedingung. Ist diese erfüllt wird der Block hinter `if` (Zeilen 15-17) ausgeführt, ansonsten der Block hinter `else` (Zeilen 19-21).

## 11. Die Mehrfachentscheidung (Mehrfachauswahl, switch-case-Struktur)

Manchmal müssen beim Prüfen einer Variable sehr viele Fälle berücksichtigt werden. Realisiert man dies mit `if-else`-Strukturen, so wird das Programm schnell unübersichtlich. Hier hilft eine `switch-case`-Struktur, welche mit Hilfe eines Programmbeispiels erläutern werden soll. Das Programm soll nach Eingabe der Zahlen 1, 2, 3 oder 4 die jeweilige römische Zahl (I, II, III oder IV) und ansonsten den String "Fehler" ausgeben.

**Struktogramm:**



**Programmablaufplan:** Hier gibt es kein genormtes Symbol, deshalb wird auf die Darstellung verzichtet.

**Javacode:**

```

7  int x = Tastatur.intInput();
8
9  switch(x)
10 {
11     case 1: System.out.println("I"); break;
12     case 2: System.out.println("II"); break;
13     case 3: System.out.println("III"); break;
14     case 4: System.out.println("IV"); break;
15     default: System.out.println("Fehler"); break;
16 }
    
```

Auf Zeile 7 wird eine Variable `x` definiert und mittels `Tastatur` gefüllt. Die `switch-case`-Struktur beginnt auf Zeile 9 mit der Festlegung, dass die Variable `x` geprüft werden soll. Auf den Zeilen 11-14 werden nun 4 Fälle geprüft, nämlich ob in `x` die Zahlen 1, 2, 3 oder 4 enthalten sind. Wurde z. B. die Zahl 3 eingegeben, so werden alle Anweisungen welche hinter `case 3` (Zeile 13) stehen ausgeführt. Es würde dann also der String "III" ausgegeben werden. Anschließend bewirkt die Anweisung (bzw. das Schlüsselwort) `break`, dass die `switch-case`-Struktur sofort verlassen wird. (Das Programm wird dann unterhalb der Zeile 16 fortgesetzt.) Sollte eine Zahl eingegeben werden, welche nicht mittels `case` als gültiger Wert definiert wurde (z. B. die 0, welche als römische Zahl nicht existiert), so werden alle Anweisungen hinter `default` (Zeile 15) ausgeführt. Falls es den Anforderungen an das Programm genügt, kann `default` auch weggelassen werden.

Falls `break` hinter dem zutreffendem `case` fehlen sollte, wird `default` auch ausgeführt. Dies kann zu Widersprüchen führen.

Mit einer `switch-case`-Struktur kann man auch prüfen, ob eine Variable vom Typ `char` ein bestimmtes Zeichen enthält. Zum Beispiel den Großbuchstaben W:

```
case 'W': System.out.println("...") ; break;
```

Zwischen `case` und `break` dürfen beliebig viele Anweisungen stehen, immer mit Semikolon getrennt. Allerdings sollte man den Code dann vernünftig formatieren, also alle zusammengehörende Anweisungen untereinander schreiben und mit geschweiften Klammern zu einem Block zusammenfassen.

Mit einer `switch-case`-Struktur kann nur geprüft werden, ob eine Variable genau definierte Werte enthält. Es ist beispielsweise nicht möglich zu prüfen, ob der Wert innerhalb verschiedener Bereiche (z. B. 0...9, 10...99) liegt. Falls dies aber erforderlich sein sollte, müssen `if-else`-Strukturen verwendet werden.

Festlegung: Die oben beschriebene Einschränkung ist auch beim Erstellen von Struktogrammen zu berücksichtigen, damit diese direkt in Javacode umgesetzt werden können.

## 12. Schleifen (Wiederholung, Iteration)

Häufig müssen in einem Programm bestimmte Anweisungen mehrfach ausgeführt werden, dafür sind Schleifen zu verwenden. Die zu wiederholenden Anweisungen bilden den sogenannten **Schleifenkörper**. Weiterhin wird eine **Schleifenbedingung** benötigt. Befindet sich diese Schleifenbedingung unterhalb des Schleifenkörpers, so spricht man von einer **fußgesteuerten Schleife**. Das Gegenteil davon ist die **kopfgesteuerte Schleife**, dort befindet sich die Schleifenbedingung oberhalb des Schleifenkörpers.

### 12.1 Die fußgesteuerte Schleife

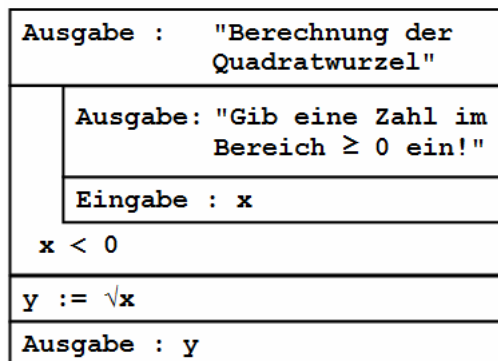
In jedem Fall wird zunächst der Schleifenkörper komplett durchlaufen. Anschließend wird geprüft, ob die Schleifenbedingung erfüllt ist. Falls ja, so wird der Schleifenkörper nochmals durchlaufen und anschließend wird wieder geprüft. Erst wenn die Schleifenbedingung nicht erfüllt ist, wird die Schleife verlassen.

#### Beispiel:

Ein Programm berechnet die Quadratwurzel einer einzugebenden Zahl und gibt diese aus. Dabei wird vermieden, dass aus einer negativen Zahl die Wurzel gezogen wird. Dies ist nämlich nicht möglich und würde zu einem Abbruch des Programms führen. Es folgt eine detaillierte Beschreibung des Programms:  
 Nach dem Start des Programms gibt es den 1. Text "Berechnung der Quadratwurzel" aus. Anschließend wird der 2. Text "Gib eine Zahl im Bereich  $\geq 0$  ein!" ausgegeben. Nach Eingabe dieser Zahl wird geprüft, ob diese negativ ist. Falls ja, so wird die Ausgabe des 2. Textes und die Eingabe der Zahl wiederholt. Falls nein, so wird die Wurzel berechnet und ausgegeben.

#### Struktogramm:

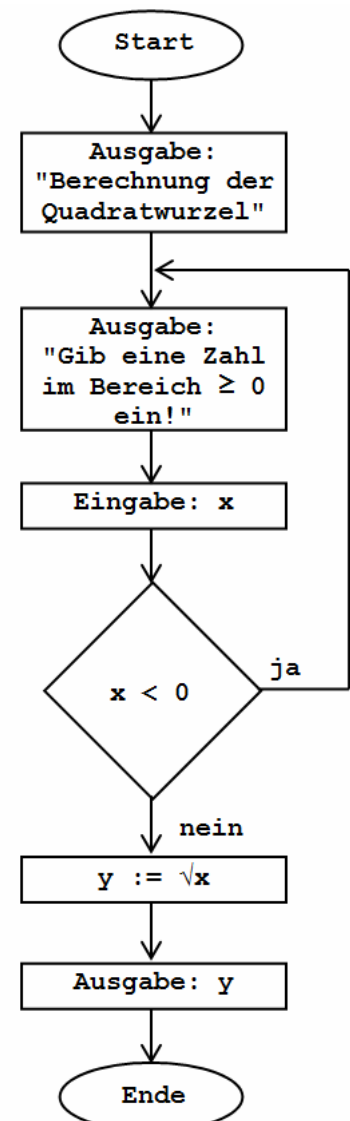
Beschreibung der Variablen		
Variable	Datentyp	Beschreibung
x	double	speichert die eingegebene Zahl
y	double	speichert das Ergebnis (Wurzel aus x)



Schleifenkörper

Schleifenbedingung

#### Programmablaufplan:



Es folgt die Umsetzung des Struktogramms bzw. Programmblaufplans in den Javacode. Eine fußgesteuerte Schleife wird immer mit dem Schlüsselwort `do` (Zeile 10) eingeleitet. Innerhalb eines geschweiften Klammerpaares folgt dann der Schleifenkörper (Zeilen 11-14). Anschließend wird unterhalb des Schleifenkörpers hinter dem Schlüsselwort `while` die Schleifenbedingung formuliert. Erst wenn diese nicht mehr erfüllt ist, wird die Schleife verlassen und die Anweisungen auf den Zeilen 17 und 18 werden ausgeführt.

```

5 public static void main (String args[])
6 {
7     double x, y;
8     System.out.println("Berechnung der Quadratwurzel");
9
10    do
11    {
12        System.out.println("Gib eine Zahl im Bereich >= 0 ein!");
13        x = Tastatur.doubleInput();
14    }
15    while(x < 0);
16
17    y = Math.sqrt(x);
18    System.out.println(y);
19 }
    
```

## 12.2 Die kopfgesteuerte Schleife

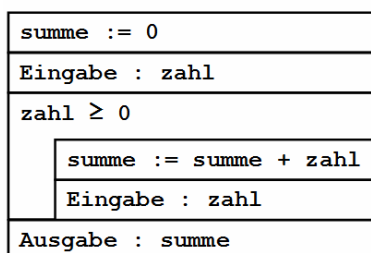
Zuerst wird die Schleifenbedingung geprüft. Ist diese erfüllt, so wird der Schleifenkörper komplett durchlaufen. Anschließend wird wieder die Schleifenbedingung geprüft. Wenn die Schleifenbedingung nicht erfüllt ist, wird die Schleife verlassen. Im Gegensatz zur fußgesteuerten Schleife ist es also möglich, dass die kopfgesteuerte Schleife kein einziges Mal ausgeführt wird.

### Beispiel:

Ein Programm bildet die Summe von beliebig vielen ganzen Zahlen im Bereich 0...99999, welche der Anwender nacheinander eingibt. Nach Eingabe einer negativer Zahl wird die berechnete Summe ausgegeben. Anschließend ist das Programm beendet.

### Struktogramm:

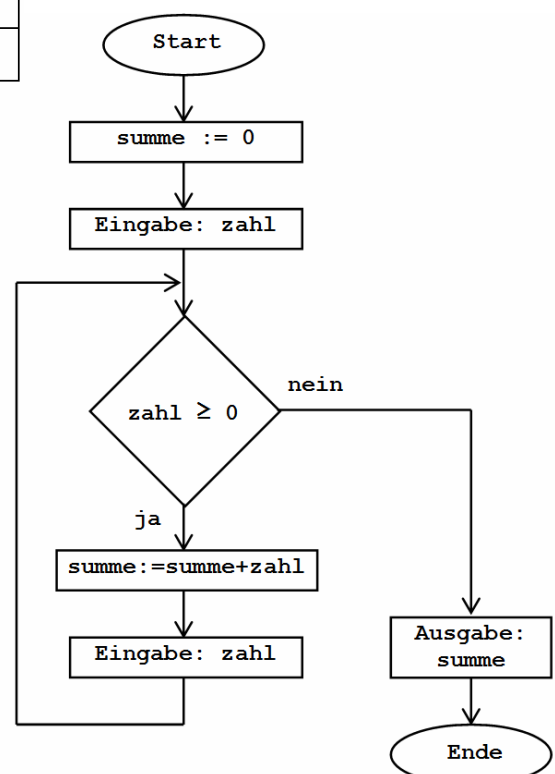
Beschreibung der Variablen		
Variable	Datentyp	Beschreibung
zahl	int	speichert die aktuell eingegebene Zahl
summe	int	speichert die Summe aller eingegebenen Zahlen



← **Schleifenbedingung**

**Schleifenkörper**

### Programmblaufplan:



Es folgt die Umsetzung des Struktogramms bzw. Programmblaufplans in den Javacode. Eine kopfgesteuerte Schleife wird immer mit dem Schlüsselwort `while` und der Schleifenbedingung (Zeile 10) eingeleitet. Innerhalb eines geschweiften Klammerpaares folgt dann der Schleifenkörper (Zeilen 11-14).

```
5 public static void main (String args[])
6 {
7     int zahl, summe = 0;
8     zahl = Tastatur.intInput();
9
10    while(zahl >= 0)
11    {
12        summe = summe + zahl;
13        zahl = Tastatur.intInput();
14    }
15
16    System.out.println(summe);
17 }
```

### 12.3 Die Zählschleife

Die Zählschleife ist eine Sonderform der kopfgesteuerten Schleife und wird verwendet, wenn eine definierte Anzahl von Durchläufen bekannt ist.

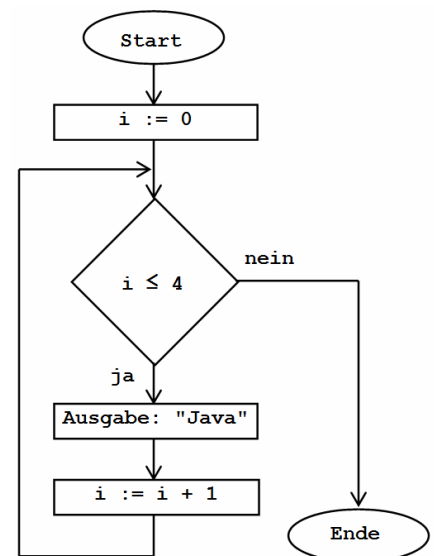
#### Beispiel:

Eine Zählschleife soll 5 mal den String "Java" ausgeben. Eine Integer-Variable "i" dient als Zählvariable.

#### Struktogramm:

zähle i von 0 bis 4, Schrittweite 1
Ausgabe : "Java"

#### Programmblaufplan:



#### Javacode mit Erläuterung der Zeile 7:

```
5 public static void main (String args[])
6 {
7     for(int i=0; i<=4; i++)
8     {
9         System.out.println("Java");
10    }
11 }
```

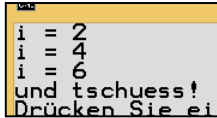
**for :** Dieses Schlüsselwort leitet die Zählschleife ein. Innerhalb eines geschweiften Klammerpaares folgt dann der Schleifenkörper.

**int i=0 :** Die Integervariable `i` (Zählvariable) wird definiert und auf den Startwert 0 gesetzt. Prinzipiell kann der Startwert auch eine andere Integer-Zahl sein. Innerhalb der Schleife kann auf diese Variable lesend und schreibend zugegriffen werden. Außerhalb der Schleife existiert diese Variable nicht, ein Zugriff würde dort eine Fehlermeldung des Compilers bewirken.

**i<=4 :** Diese Bedingung legt fest, dass der Schleifenkörper nur dann durchlaufen wird, wenn `i` maximal den Wert 4 hat.

**i++ :** Diese Anweisung bewirkt, dass nach jedem Durchlauf `i` um 1 erhöht wird, die Schrittweite ist also 1. Damit kann `i` also 5 Werte annehmen, nämlich 0, 1, 2, 3 und 4. Es ist auch möglich, dass `i` heruntergezählt wird (`i--`) oder die Schrittweite einen anderen Wert hat, z. B. 3: `i=i+3`

In den Anweisungen bzw. Bedingungen können auch Variablen enthalten sein, welche außerhalb der Zählschleife definiert und gefüllt wurden.  
Beispielcode mit Ausgabe:



```
i = 2
i = 4
i = 6
und tschuess!
Drücken Sie ei
```

```
5 public static void main (String args[])
6 {
7     int min = 2, max = 7, sw = 2;
8
9     for(int i = min; i <= max; i = i + sw)
10    {
11        System.out.println("i = " + i);
12    }
13
14    System.out.println("und tschuess!");
15 }
```

## 12.4 Ergänzende Hinweise für alle Schleifentypen

Jede Art Schleife kann gewollt oder ungewollt zu einer Endlosschleife werden, Beispiel: `while(1 == 1)`  
Die Bedingung hinter `while` ist immer erfüllt, also wird der Schleifenkörper immer wieder durchlaufen.  
Das Ergebnis des Vergleichs `1 == 1` ist immer der Wahrheitswert `true`, also kann eine Endlosschleife auch so realisiert werden: `while(true)`

Bei allen Schleifentypen kann der Schleifenkörper alle möglichen Anweisungen oder Strukturen enthalten, z. B. eine `if-else`-Struktur. Oder eine Schleife. Deren Schleifenkörper kann dann ebenfalls eine Schleife enthalten ... (Schleifen können ineinander geschachtelt werden.)

Mit Hilfe des Schlüsselworts `break` kann eine Schleife (Typ egal) jederzeit sofort abgebrochen werden. In diesem Fall werden alle nachfolgenden Anweisungen im Schleifenkörper ignoriert und das Programm wird unterhalb der Schleife fortgesetzt.

Im abgebildeten Beispiel werden in einer Endlosschleife Zahlen eingegeben und aufsummiert. Das Eingeben der Zahl "-1" bewirkt dann auf Zeile 15 den Abbruch der Schleife. Das Programm wird dann ab Zeile 20 fortgesetzt.

```
5 public static void main (String args[])
6 {
7     int zahl, summe = 0;
8
9     while(true)
10    {
11        zahl = Tastatur.intInput();
12
13        if(zahl == -1)
14        {
15            break;
16        }
17
18        summe = summe + zahl;
19    }
20
21    System.out.println(summe);
22 }
```

Mit Hilfe des Schlüsselwortes `continue` kann an den Anfang einer Schleife gesprungen werden, was der abgebildete Programmcode beispielhaft verdeutlichen soll: Das Programm summiert eingegebene Zahlen, solange die Summe kleiner als 20 ist. Falls eine negative Zahl eingegeben wurde, wird von Zeile 16 zum Anfang der Schleife gesprungen. Damit sind die Zeilen 19 und 20 nicht wirksam und es geht weiter auf Zeile 11.

```
7     int zahl, summe = 0;
8
9     while (summe < 20)
10    {
11        System.out.print("Zahl >= 0 eingeben!");
12        zahl = Tastatur.intInput();
13
14        if(zahl < 0)
15        {
16            continue;
17        }
18
19        summe = summe + zahl;
20        System.out.println("aktuelle Summe: " + summe);
21    }
```



## 1. Eindimensionale Arrays

Mal angenommen, ein Programm soll 1000 Temperaturwerte einlesen und anschließend verschiedene statistische Berechnungen durchführen. Mit einfachen Datentypen (z.B. double) wird das problematisch. Zunächst müssten 1000 Variablen deklariert werden, anschließend erfolgen 1000 Wertzuweisungen. Bei einer Durchschnittsberechnung müssten 1000 Variablen zunächst addiert werden, man stelle sich die Länge der Formel vor! Einen deutlich kürzeren und vor allem übersichtlicheren Code erhält man, wenn der komplexe Datentyp **Array** (manchmal auch Feld oder Reihung genannt) verwendet wird. Variablen, welche eine logische Einheit (z.B. Temperaturwerte einer bestimmten Wetterstation) bilden und vom selben Datentyp sind, werden zu einem Array zusammengefasst.

Es wird jetzt ein Array betrachtet, welches 5 Werte (ganze Zahlen) aufnehmen kann:

zahlen						← Name des Arrays
0	1	2	3	4		← Index
8	7	9	5	2		← Elemente mit Inhalt (Werte)

Die Größe (Länge) dieses Arrays beträgt 5, der Index beginnt immer bei 0. Über den Namen des Arrays und den Index wird auf die Elemente zugegriffen. Beispiel: Das 2. Element des Arrays "zahlen" enthält den Wert "9".

Das oben beschriebene Array (für Integer-Werte) wird in JAVA so angelegt:

```
int[] zahlen = new int[5];
```

Es folgen einige Beispiele, welche den Zugriff auf die Elemente des Arrays demonstrieren:

Das 2. Element des Arrays soll mittels Tastatureingabe gefüllt werden:

```
zahlen[2] = Tastatur.intInput();
```

Der Inhalt des 2. Elements soll ausgelesen und in der Variablen a gespeichert werden:

```
int a = zahlen[2];
```

Der Inhalt des 2. Elements soll ausgegeben werden:

```
System.out.println(zahlen[2]);
```

Soll ein Array bereits beim Programmstart Werte enthalten, kann es auch so angelegt werden: (Dies ist z. B. während der Testphase vorteilhaft, weil es das mehrfache Wiederholen von Eingaben erspart.)

```
int[] zahlen = {8,7,9,5,2};
```

Die Größe des Arrays wird dabei vom Compiler automatisch ermittelt.

Richtig sinnvoll ist bei größeren Arrays die Verwendung von Schleifen. Die folgende Schleife gibt die ersten 4 im Array gespeicherten Werte aus:

```
for (int i=0; i<=3; i++)  
{  
    System.out.println(zahlen[i]);  
}
```

Achtung: Über die Feldlänge hinaus darf nicht gelesen oder geschrieben werden. JAVA erzeugt dann eine entsprechende Exception und beendet das Programm. Manche Programmiersprachen tolerieren diesen Fehler, was u. U. bis zum Rechnerabsturz führen kann.

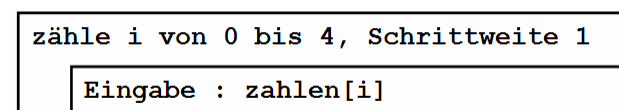
Selbstverständlich können auch Arrays für andere Datentypen angelegt werden. Zum Beispiel ein String-Array für 10 Schüler. Anschließend werden 2 Schüler eingetragen:

```
String[] schueler = new String[10];
schueler[0] = "Johannes Burg";
schueler[1] = "Marie Niert";
```

Es ist auch möglich, die Größe eines Arrays erst während der Laufzeit eines Programms festzulegen:

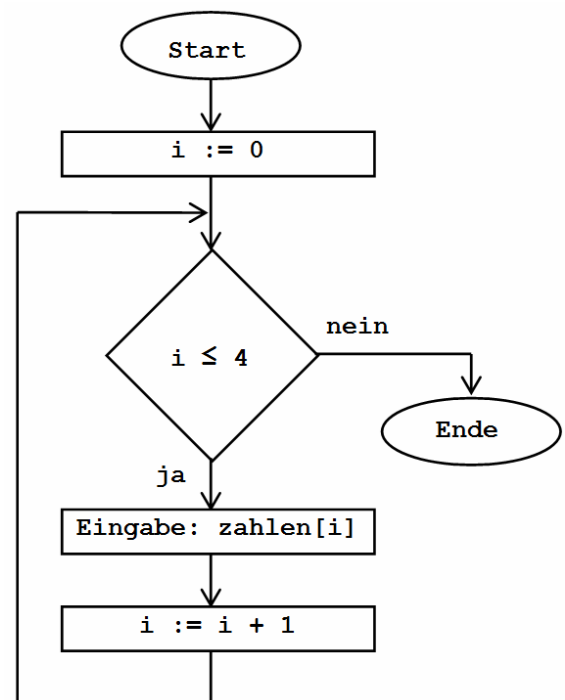
```
int b = Tastatur.intInput();
int[] zahlen = new int[b];
```

Die Schreibweise für Arrays in Struktogrammen und Programmlaufplänen ist nicht genormt, deshalb wird sie hier von JAVA übernommen. Es folgen beispielhaft die Darstellungen eines Programms, welches ein Array (Name: zahlen) mittels Tastatureingabe füllt:



Struktogramm

Programmlaufplan



## 2. Mehrdimensionale Arrays

Manchmal ist die Verwendung von mehrdimensionalen Arrays sinnvoll. Ein zweidimensionales Array kann man sich als Tabelle mit Zeilen und Spalten vorstellen.

Es wird jetzt ein Array betrachtet, welches aus 3 Zeilen und 5 Spalten besteht. Möchte man auf ein Element zugreifen, so müssen Zeilen- und Spaltenindex angegeben werden. Beispiel: Das Element auf der Zeile 0 und Spalte 3 enthält den Wert 18,5.

	0	1	2	3	4	← Spaltenindex
0	17,3	17,9	18,8	18,5	18,1	
1	22,7	22,0	22,5	23,1	24,1	
2	19,9	20,2	19,8	19,7	19,6	
↑ Zeilenindex						

Die folgende Programmzeile legt das oben abgebildete Array an, als Bezeichner wurde "zahlen" gewählt:

```
double[][] zahlen = new double[3][5]; //3 Zeilen, 5 Spalten
```

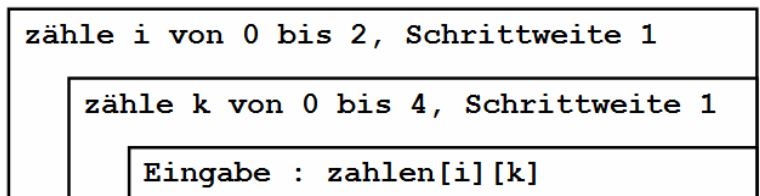
Ausgabe eines einzelnen Elements:

```
System.out.println(zahlen[0][3]);           //Zeile 0, Spalte 3 (gibt 18,5 aus)
```

Mit Hilfe von zwei ineinander geschachtelten Zählschleifen kann sehr einfach auf alle Elemente des Arrays zugegriffen werden. Das Beispiel zeigt das Füllen des Arrays mittels Tastatureingabe:

```
for(int i = 0; i <= 2; i++)                //Zeilenindex zählen
{
    for(int k = 0; k <= 4; k++)              //Spaltenindex zählen
    {
        zahlen[i][k] = Tastatur.doubleInput(); //Eingabe mittels Tastatur
    }
}
```

Und so sieht das zugehörige Struktogramm aus:



Wie bereits im letzten Kapitel erwähnt, gibt es bzgl. der Schreibweise von Arrays in Struktogrammen bzw. Programmablaufplänen keine Norm. Deshalb wird hier die Schreibweise von JAVA übernommen. In der Literatur werden auch andere Varianten verwendet, z. B. `zahlen[i,k]` oder `zahlen(i,k)`.

Auch bei zweidimensionalen Arrays ist es möglich, bei Bedarf (z. B. in der Testphase) das Array durch feste Vorgaben im Programmcode anzulegen und zu füllen. Die Anzahl der Zeilen und Spalten wird dabei vom Compiler automatisch ermittelt. Beispiel:

```
double[][] zahlen = {
    {17.3, 17.9, 18.8, 18.5, 18.1},
    {22.7, 22.0, 22.5, 23.1, 24.1},
    {19.9, 20.2, 19.8, 19.7, 19.6},
};
```

Prinzipiell können auch Arrays mit mehr als zwei Dimensionen verwendet werden, Beispiele:

Dreidimensionales Array anlegen: `int[][][] zahlen = new int[5][5][4];`

Vierdimensionales Array anlegen: `int[][][][] zahlen = new int[8][5][4][3];`

Allerdings dürfte spätestens ab Einbeziehung der 4. Dimension das menschliche Vorstellungsvermögen erheblich überstrapaziert werden.

### 3. Suchalgorithmen (lineare und binäre Suche)

Möchte man ermitteln, ob in einem Array ein bestimmter Wert existiert und unter welchem Index er zu finden ist, so verwendet man einen Suchalgorithmus.

#### Was ist ein Algorithmus?

Unter einem Algorithmus versteht man eine eindeutige Handlungsvorschrift zur Lösung eines Problems. Das können beispielsweise Berechnungsvorschriften in der Mathematik, Montageanleitungen für Möbel oder auch Kochrezepte sein. Bei der Problemlösung wird eine bestimmte Eingabe in eine bestimmte Ausgabe überführt. So kann zum Beispiel aus den "Eingaben" Hopfen, Malz, Hefe und Wasser die "Ausgabe" Bier erzeugt werden.

#### Welche Eigenschaften muss ein Algorithmus in der Datenverarbeitungstechnik besitzen?

Damit ein Algorithmus nicht nur in der menschlichen Sprache formuliert, sondern auch in einem Computerprogramm implementiert werden kann, muss er folgende Eigenschaften besitzen:

- \*Algorithmen bestehen aus definierten Einzelschritten, deren Anzahl endlich sein muss.
- \*Jeder Einzelschritt muss tatsächlich ausführbar sein.
- \*Nach der Ausführung eines Einzelschritts muss eindeutig definiert sein, welcher Schritt anschließend erfolgt.
- \*Unter gleichen Voraussetzungen muss ein Algorithmus immer das gleiche Ergebnis bringen.
- \*Der zur Ausführung des Algorithmus erforderliche Speicherplatz muss zu jedem Zeitpunkt begrenzt sein.

In der Regel gilt, dass ein Algorithmus nach einer endlichen Zeit beendet sein muss (Terminierung). Es gibt allerdings Ausnahmen, ein typisches Beispiel ist eine Temperaturregelung.

#### Lineare Suche

Bei der linearen Suche (auch sequentielle Suche genannt) wird ein Array innerhalb einer Schleife durchlaufen. Dabei werden die im Array gespeicherten Werte mit dem gesuchten Wert verglichen. Bei Übereinstimmung findet dann eine definierte Aktion statt, zum Beispiel die Ausgabe des Index. Je nach Aufgabenstellung ist die lineare Suche beendet, wenn entweder das Array komplett durchlaufen oder ein erster "Treffer" gefunden wurde.

#### Binäre Suche

Für die binäre Suche ist es zwingend erforderlich, dass das Array **sortiert** ist. Das Prinzip der binären Suche wird anhand eines Beispiels erläutert: Gegeben ist das in **Bild 1** dargestellte Array, in welchem der Wert 666 gesucht werden soll.

<b>Bild 1</b>															
								Mitte							
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	112	131	140	199	201	209	332	345	588	666	723	799	810	889	899
									</						

Zuerst wird geprüft, ob das mittlere Element des Arrays (Index 7) den gesuchten Wert enthält. Weil dem nicht so ist, wird anschließend geprüft, ob der gesuchte Wert größer oder kleiner als der Wert im mittleren Element ist. Weil in diesem Fall der gesuchte Wert größer ist, kann er sich nur rechts von der Mitte (Indizes 8 bis 14) befinden. Deshalb wird jetzt nur noch dieser Teil des Arrays untersucht, siehe **Bild 2**. Hier wird wieder der Wert des mittleren Elements (Index 11) mit dem gesuchten Wert verglichen. In diesem Fall ist der gesuchte Wert kleiner, also kann er sich nur links von der Mitte (Indizes 8 bis 10) befinden. Prüft man in diesem Teil-Array (**Bild 3**) das mittlere Element, so erhält man einen "Treffer" und die Suche wird beendet.

Anmerkungen: Ist der gesuchte Wert tatsächlich im Array enthalten, so wird der Suchalgorithmus i. d. R. beendet, wenn der Wert gefunden wurde. Sollte das Array mehrere gleiche Werte enthalten, welche auch alle gefunden werden sollen, so ist der Suchalgorithmus entsprechend zu modifizieren. Sehr häufig tritt der Fall ein, dass ein Wert im Array nicht vorhanden ist. In diesem Fall muss dafür gesorgt werden, dass der Suchalgorithmus trotzdem rechtzeitig beendet wird.

## 4. Sortialgorithmen

Häufig ist es erforderlich, dass ein Array sortiert werden muss. Bei numerischen Werten wird i. d. R. aufsteigend sortiert, beim Index 0 ist also der kleinste Wert zu finden. Bei Strings wird i. d. R. aufsteigend nach dem Alphabet sortiert. Es existieren viele Sortialgorithmen, welche unterschiedlich effizient und häufig an bestimmte Anforderungen angepasst sind. Weiterführende Erläuterungen sind bei Bedarf im Internet zu finden. In diesem Skript wird nur auf den relativ einfachen Sortialgorithmus **Bubblesort** eingegangen:

Betrachtet wird ein Array mit 6 Elementen:

0	1	2	3	4	5	Index
47	42	77	45	55	52	Inhalt

**Beginnend beim Index 0 wird jeder Wert des Arrays mit seinem rechten Nachbar verglichen. Sollte der rechte Wert kleiner sein, so werden beide Werte vertauscht, ansonsten nicht.**

Im ersten Durchlauf wird also zuerst die 47 (Index 0) mit der 42 (Index 1) verglichen und anschließend müssen beide Werte vertauscht werden. Danach wird die 47 (jetzt auf Index 1) mit der 77 (Index 2) verglichen, hier darf keine Vertauschung stattfinden. Die folgende Abbildung zeigt den kompletten 1. Durchlauf, die zu vergleichenden Werte sind jeweils unterstrichen:

<b>1. Durchlauf</b>											
<u>47</u>	<u>42</u>	77	45	55	52	1. Vergleich , anschließend Vertauschung erforderlich					
42	<u>47</u>	<u>77</u>	45	55	52	2. Vergleich , keine Vertauschung					
42	47	<u>77</u>	<u>45</u>	55	52	3. Vergleich , anschließend Vertauschung erforderlich					
42	47	45	<u>77</u>	<u>55</u>	52	4. Vergleich , anschließend Vertauschung erforderlich					
42	47	45	55	<u>77</u>	<u>52</u>	5. Vergleich , anschließend Vertauschung erforderlich					
42	47	45	55	52	<b>77</b>	<b>Nach dem 1. Durchlauf ist die größte Zahl auf der richtigen Position.</b>					

Weil bis jetzt nur die größte Zahl auf der richtigen Position steht, muss ein zweiter Durchlauf stattfinden. Allerdings genügt es jetzt, wenn die Sortierung nur noch innerhalb der ersten 5 Elementen des Arrays erfolgt:

<b>2. Durchlauf</b>											
<u>42</u>	<u>47</u>	45	55	52	<b>77</b>	1. Vergleich , keine Vertauschung					
42	<u>47</u>	<u>45</u>	55	52	<b>77</b>	2. Vergleich , anschließend Vertauschung erforderlich					
42	45	<u>47</u>	<u>55</u>	52	<b>77</b>	3. Vergleich , keine Vertauschung					
42	47	45	<u>55</u>	<u>52</u>	<b>77</b>	4. Vergleich , anschließend Vertauschung erforderlich					
42	47	45	52	<b>55</b>	<b>77</b>	<b>Nach dem 2. Durchlauf ist die zweitgrößte Zahl auf der richtigen Position.</b>					

Jetzt der dritte Durchlauf, die Sortierung erfolgt nur noch innerhalb der ersten 4 Elemente des Arrays:

<b>3. Durchlauf</b>															
<u>42</u>	<u>47</u>	45	52	<b>55</b>	<b>77</b>	1. Vergleich , keine Vertauschung									
42	<u>47</u>	<u>45</u>	52	<b>55</b>	<b>77</b>	2. Vergleich , anschließend Vertauschung erforderlich									
42	45	<u>47</u>	<u>52</u>	<b>55</b>	<b>77</b>	3. Vergleich , keine Vertauschung									
42	45	47	<b>52</b>	<b>55</b>	<b>77</b>	<b>Nach dem 3. Durchlauf ist die drittgrößte Zahl auf der richtigen Position.</b>									

Wie man bereits erkennen kann, wird es bei einem vierten Durchlauf keinerlei Vertauschungen mehr geben, weil das Array bereits sortiert ist. In diesem Fall kann der Sortiervorgang vorzeitig beendet werden.

**Wendet man den Bubblesort auf ein beliebiges Array an, welches n Elemente enthält, so gelten folgende Aussagen:**

Es sind maximal (n-1) Durchläufe erforderlich. Im ersten Durchlauf erfolgen genau (n-1) Vergleiche bzw. maximal (n-1) Vertauschungen. In jedem weiteren Durchlauf reduziert sich die Anzahl der Vergleiche bzw. der maximaler Vertauschungen um 1.

1. Grenzfalle (minimaler Zeitbedarf): Das Array ist bereits sortiert. In diesem Fall ist nur ein Durchlauf erforderlich, durch den die Sortierung festgestellt werden kann, weil keine einzige Vertauschung notwendig ist.

2. Grenzfalle (maximaler Zeitbedarf): Das Array ist zwar sortiert, allerdings in der umgekehrten Reihenfolge. Hier muss die maximale Anzahl der Durchläufe absolviert werden und jeder Vergleich führt zu einer Vertauschung.

$$\text{Anzahl der Vertauschungen} = \frac{n \cdot (n - 1)}{2}$$

