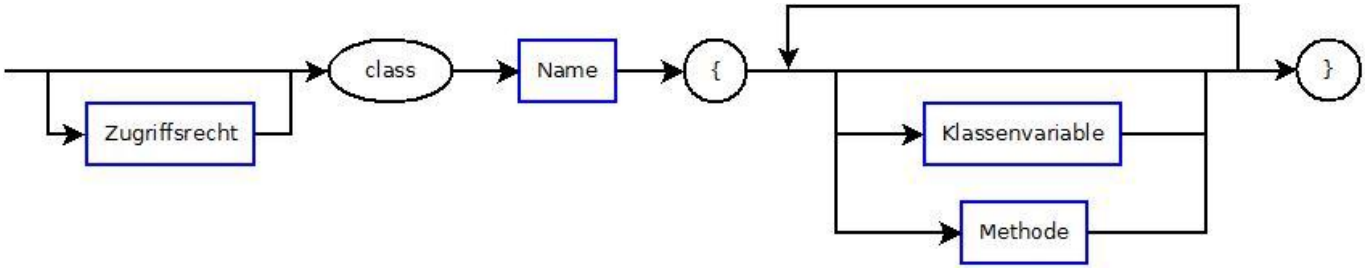
Java-Grundlagen

Hier werden Sie Schritt für Schritt in die Programmierung mit Java eingeführt. Im Anschluss an jedes Kapitel finden sich stets kleine Übungsaufgaben, die zum Verständnis beitragen sollen. Denn man Programmieren nur durch selbstständiges Programmieren lernen!

Klassen

Klassen sind die zentralen Elemente in der Java-Programmierung, und jedes Java-Programm besteht aus mindestens einer Klasse.



Jede Klasse besteht also mindestens aus dem Schlüsselwort **class**, dem Namen der Klasse und einem Paar geschweifter Klammern. In den geschweiften Klammern können dann mehrere Klassenvariablen und Methoden stehen. Zudem kann vor dem Schlüsselwort **class** auch noch ein Zugriffsrecht angegeben werden. Im Quelltext sieht so etwas wie folgt aus.

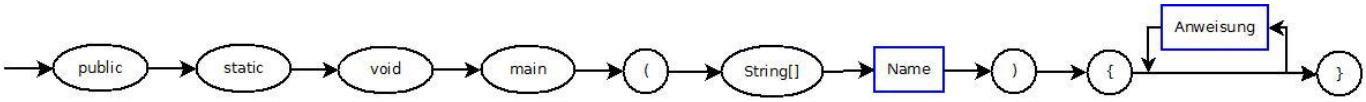
```
public class ErsteKlasse {

  //Klassenvariablen
  static int variableA;
  static double variableB;
  static boolean variableC;

  //Klassenmethoden
  public static void methode1(){
  }

  public static void methode2(int parameter){
  }

  //Hauptmethode!!!!!!!
  public static void main(String[] args) {
  }
}
```

Innerhalb einer Klasse können im Klassenrumpf **Variablen** und **Methoden** deklariert werden. Wir werden uns in diesem Halbjahr immer mit **Klassenvariablen** und **Klassenmethoden** beschäftigen. Diese erkennt man an dem Schlüsselwort **static**.

In unserem Beispiel sind also die Variablen mit den Namen variableA, variableB, und variableC Klassenvariablen. Diese sind innerhalb der Klasse definiert. Die Klassenmethoden heißen methode1, methode2 und main. Die wichtigste Methode in einem Java-Programm ist die main-Methode, denn nur das was dort enthalten ist wird auch ausgeführt.

Die main-Methode – Das Herz eines Programms

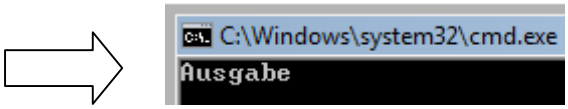
Schauen wir und jetzt einmal die wichtigste Methode an - die **main**-Methode. Diese hat auch einen festgelegten Aufbau.



Wie gesagt: Ohne diese Methode passiert in unserem Quellcode nichts!

Damit wir etwas sehen können, fügen wir eine spezielle Anweisung ein.

```
public class Ausgabe {

  public static void main(String[] args) {
    System.out.println("Ausgabe");
  }
}
```



Die oben angegebene Methode werden wir ab jetzt immer dann verwenden, wenn wir sehen wollen, ob unsere Lösungen auch funktionieren.

Aufgabe

Die Methode `System.out.println()` erzeugt eine Ausgabe mit einem Zeilenumbruch auf der Konsole. Möchte man den Zeilenumbruch nicht haben, so schreibt man hierfür einfach `System.out.print()`.

Leider kann man noch nichts sehen, also müssen wir in die Klammern die Werte hineinschreiben, die wir auf der Konsole sehen möchten. Hierbei ist folgendes zu beachten.

- Wörter bzw. Sätze schreibt man immer in doppelten Anführungszeichen (`"Ausgabe"`).
- Zahlen kann man direkt hineinschreiben, allerdings muss man auf die englische Schreibweise achten, d.h. hier wird das Komma durch einen Punkt ersetzt (`1` bzw. `4.5`).
- Buchstaben werden in einfache Anführungszeichen gesetzt (`'c'`).

Experimentieren Sie ein wenig herum und geben Sie folgende Werte aus:

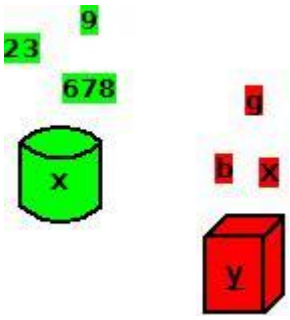
1) 13
2) Mein Name ist
3) B
4) Geben Sie die größte ganze Zahl aus, die möglich ist.

Java kann verschiedene einfache Rechnungen direkt ausführen. Experimentieren Sie mit den Grundrechenarten herum und achten Sie auf das Ergebnis.

Was gibt 17 / 4? Was gibt 17.0 / 4? Was ist b+c? Was ist c+1? Was ist Hallo + Du?

Um diese Ergebnisse genauer zu verstehen, müssen wir uns mit Variablen beschäftigen.

Variablen



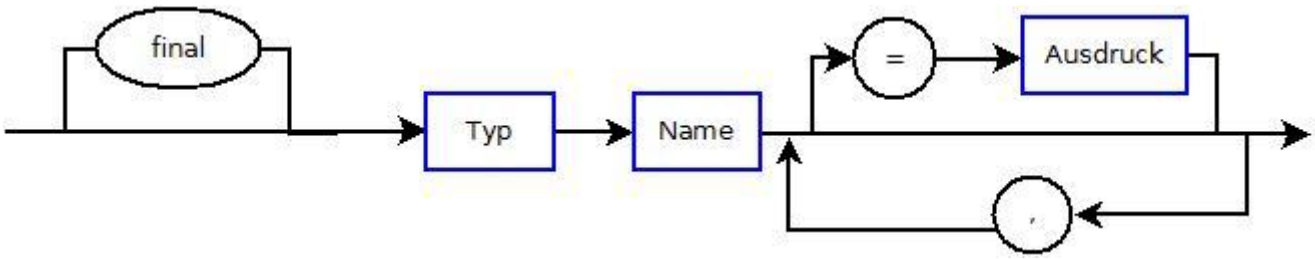
Variablen sind „Gedächtniszellen" eines Programms. Hierfür wird ein bestimmter Speicherplatz reserviert und dieser mit einem ansprechbaren Namen versehen. Im Gegensatz zu andern Programmiersprachen muss man bei der Erzeugung einer Variablen angeben, um welche Art der Information gespeichert werden soll – welchen **Typ** die Variable hat. Die Werte der Variablen könne innerhalb eines Programms jederzeit verändert werden, allerdings muss der Typ dann immer übereinstimmen.

```
// Ganze Zahlen
byte a;
short b;
int c;
long d;

// Gleitpunktzahlen
float e;
double f;

// Wahrheitswert
boolean g;

// Zeichen
char h;
```

Man sagt: Variablen müssen vor ihrer Benutzung **deklariert** werden. Dazu stellt man dem Namen, den man einer Variablen gibt, einen Datentyp voran.

In dem Beispiel links oben haben Sie die acht verschiedene Variablen mit den acht unterschiedlichen **primitiven Datentypen**.

Hierbei gibt es vier verschiedene Typen für ganze Zahlen (**byte**, **short**, **int**, **long**), zwei für Kommazahlen (**float**, **double**), einen für Wahrheitswerte (**boolean**) und einen für Zeichen (**char**).



Nach ihrer Deklarierung haben Variablen einen undefinierten Wert. Vor ihrer ersten Benutzung, d.h. bevor eine Variable erstmals gelesen wird, muss sie **initialisiert** worden sein. Dazu belegt man die Variable mit einem Wert. Dies kann auch schon mit der Deklaration zusammen geschehen.

```
c=5;
f=4.6;
h='k';
int i=8;
short j=4, k=5, l=7;
```

Innerhalb eines Programms kann nun der Wert der Variable jederzeit verändert werden. Steht allerdings **final** vor dem Datentyp, so geht dies nicht mehr. Hierbei handelt es sich jetzt um eine **Konstante**.

Der Compiler überprüft bei Ihrem Quellcode stets, ob die Variable auch deklariert bzw., vor der ersten Benutzung initialisiert wurden. Ansonsten gibt er Fehlermeldungen aus.

```
Ausgabe.java:36:5: cannot find symbol
symbol  : variable m
location: class Ausgabe
    m=5;
    ^
```

Variable m wurde nicht **deklariert**

Primitive Datentypen

In Java sind acht Datentypen für die Bearbeitung elementarer Werte festgelegt. Man muss sich also Gedenken darüber machen, welche Art von Werten man benutzen will und wie groß sie maximal werden sollen.

Ganzzahlige Datentypen

Datentyp	Größe	Wertebereich
byte	1 Byte	-128 bis 127
short	2 Byte	-32.768 bis 32.767
int	4 Byte	-2.147.483.648 bis 2.147.483.647
long	8 Byte	-2^{63} bis $2^{63}-1$

Normalerweise benutzen wir zum Rechnen mit ganzen Zahlen den Typ int.

```
byte a = 5;
short b = 23;
int c = 42;
long d = 6786L;
int e = 0567;
long f = 0X112L;
```

Möchte man den Datentyp long benutzen, muss man l bzw. L an die Zahl anfügen. Mit einer vorangestellten 0 kann man die Zahl als Oktalzahl, mit einem vorangestellten 0X als Hexadezimalzahl darstellen.

Gleitpunktzahlen

Datentyp	Größe	Wertebereich
float	4 Byte	Ca. $-3.4 * 10^{38}$ bis $3.4 * 10^{38}$
double	8 Byte	Ca. $-1.8 * 10^{308}$ bis $1.8 * 10^{308}$

Normalerweise benutzen wir zum Rechnen mit Gleitpunktzahlen den Typ double.

```
double a = 4.5;
double b = 3.9e2;
float c = 3.205F;
float d=7f;
```

Eine Gleitkommazahl kann einen Dezimalpunkt oder einen Exponenten (e,E) enthalten. Sie werden ohne angehängtes f oder F standardmäßig als double-Wert angesehen. Spezielle Werte sind hierbei `NaN`, `NEGATIVE-INFINITY` bzw. `POSITIVE-INFINITY`.

Zeichen

Datentyp	Größe	Wertebereich
char	2 Byte	Alle Zeichen

```
char a = 'f';
char b = '\u0061';
char c = '\n';
```

Zeichen können entweder als einzelnes Zeichen, das in einfachen Anführungszeichen angegeben wird oder aber als Unicode-Wert engegeben werden.

Zeichen

Datentyp	Größe	Wertebereich
boolean	1 Byte	true, false

```
boolean wert1=true;
boolean wert2=false;
```

Es gibt zwei verschiedene Werte für den Typ boolean: entweder wahr(**true**) oder falsch (**false**).

Bezeicher, Namen, Konventionen

Bevor wir jetzt mit dem Rechnen mit Variablen weitermachen, müssen Sie ein klein wenig über die Namensvergabe erfahren. Sie müssen sich stets sinnvolle Namen für Ihre Klassen, Variablen aber auch Methoden überlegen. Hierbei ist vieles erlaubt, man sollte Sie aber **sinnvoll** und im geltenden Bereich **eindeutig** wählen.

Die Bezeichner für diese Namen können aus Buchstaben, Ziffern, dem Unterstrich und dem Dollarzeichen zusammengesetzt werden, allerdings darf man hierbei nicht mit einer Ziffer anfangen. Auch Leerzeichen sind innerhalb von Bezeichnern nicht erlaubt.

Wie Sie im Beispiel unten sehen, hat man nur Bezeichner gewählt, die aus Buchstaben und Ziffern zusammengesetzt sind.

```
public class Bezeichner {

  public static void main(String[] args) {
    double wert;
    int z1, z2;
    boolean meinWahrheitswert;
    char ersterBuchstabe;
  }
}
```

Zur Vergabe gibt es unter den Programmierern eine gemeinsame Vereinbarung, eine Art Kodex, an den sich viele Java-Programmierer halten - die Namenskonventionen.

Konventionen

- Namen von **Variablen** oder **Methoden** beginnen immer mit einem Kleinbuchstaben.
- Namen von **Klassen** beginnen immer mit einem Großbuchstaben.
- Bei zusammengesetzten Wörtern steht an der Wortgrenze ein Großbuchstabe.

Zudem werden alle Elemente, die innerhalb einer geschweiften Klammer stehen um zwei Leerzeichen eingerückt.

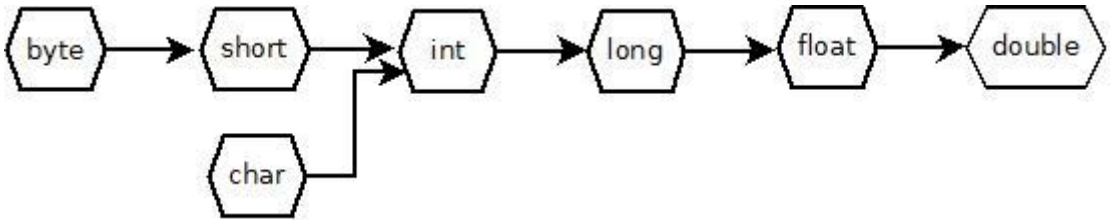
Wir werden später noch weitere Konventionen besprechen, aber dies erst an gegebener Stelle.

Ausdrücke, Anweisungen und Operatoren

Wir werden in diesem Kapitel viele Analogien zur Mathematik erkennen. Aber keine Angst: es sind nur leichte Vergleiche. Ein **Ausdruck** ist eine Verarbeitungsvorschrift, ein Term, die sich aus verschiedenen Operanden (Variablen, Buchstaben, Zahlen,…) und **Operatoren** zusammensetzt. Durch ein **;** wird aus einem Ausdruck eine **Anweisung**, die dem Programm einen Auftrag zur Berechnung übergibt. Da wir dem Programm stets Anweisungen geben müssen, ist des Semikolon äußerst wichtig, wird allerdings am Anfang Ihr häufigster Fehler werden. ☺

```
12+5    //Ausdruck
j=3+4   //Ausdruck
15==3   //Ausdruck

System.out.println("Anweisung"); //Anweisung
variable=13+4-3/2;              //Anweisung
```

Bevor wir uns jetzt aber den einzelnen Operationen zuwenden, muss man zuerst noch einmal wissen, was passiert, wenn man Daten verschiedener Datentypen miteinander vermischt.



Verbindet man verschiedene Datentypen, so wandelt der Compiler diese Typen automatisch in den „größeren" Datentyp um. Dies nennt man einen impliziten **Cast**.

Hier sehen Sie, dass man den „größeren" Datentypen die „kleineren" Datentypen zuweisen kann. Viel interessanter ist dies bei der Operation mit verschiedenen Datentypen.

```
short s =123 ;  //s hat Typ short
int i = 21;     //i hat Typ int
long l;         //l hat Typ long

i=s;            //Umwandlung von short zu int
l=s;            //Umwandlung von short zu long
```

Operatoren in Java

Es gibt eine Vielzahl von Operatoren in Java, die auf unterschiedliche Datentypen angewendet werden können. Diese wind alle in der folgenden Tabelle dargestellt. Wir werden sie später an Beispielen schrittweise erläutern.

Arithmetische Operatoren:	+	-	*	/	%	++	--
Bitoperatoren	&	|	^	~	<<	>>	>>>
Vergleichsoperatoren:	>	>=	==	!=	<=	<	? :
Logische Operatoren:	&	&&	|	||	!		
Zuweisungsoperatoren:	=	+=	-=	*=	/=	%=	
	&=	|=	^=	<<=	>>=	>>>=	

Damit Sie aber schon einen kleinen Vorgeschmack bekommen, gibt es hier ein paar einfache Beispiele, die nur einem kleinen Hinweis bedürfen. Das Gleichheitszeichen ist in Java kein Vergleich, sondern eine Zuweisung. In der dritten Zeile beispielsweise wird die Summe der Werte von `x` und `y` der Variablen `breite` zugewiesen.

```
byte x = 5;
short y = 16;
int breite = x + y;        // Addition
int laenge = breite - 1;   // Subtraktion
double produkt = x * y;    // Multiplikation
int quotient = x / y;      // Division
int rest = 17 % 4;         // Modulo (Rest beim Teilen)
x++;                       // Inkrement: x = x + 1
x--;                       // Dekrement: x = x - 1
```

Die verschiedenen Operatoren haben eine unterschiedliche **Priorität** (Reihenfolge). Ähnlich wie in der Mathematik die Priorität „Punkt- vor Strichrechnung" wird den Operatoren auch in der Informatik eine gewisse Gewichtung mitgegeben.

Priorität	Operator		Beispiele
15	Methodenaufruf Arrayindex Elementzugriff	() [] .	System.out.println() schlumpf[3] Beispiel.meins
14	Inkrement Dekrement Vorzeichen Bitkomplement logische Negation Typumwandlung	++ -- + , - ~ ! (<Typ>)	++i x-- -5 ~i ! true (int) 3.14
13	Multiplikation Division Divisionsrest	* / %	5*7 6/5 67%2
12	Addition Subtraktion Konkatenation	+ - +	a+1 13-4 "Ich "+"und "+"Du"
11	Linksverschiebung Rechtsverschiebung Rechtsverschiebung	<< >> >>>	i<<3 a>>4
10	kleiner, kleiner gleich größer, größer gleich Typüberprüfung	<, <= >, >= instanceof	2<a, a<=b 4>x
9	gleich ungleich	== !=	a==b b!=2
8	bitweises UND	&	0xFF & 223
7	bitweises XOR	^	Variable ^ 0x0
6	bitweises ODER	|	0xFF | 0x20
5	logisches UND	&&	a==b && b>c
4	logisches ODER	||	x<20 || x>10
3	Bedingung	?:	return (a<b)?2:3
2	Zuweisung	= *=, /=, %=, +=, - =, &=, ^=, |=, <<=, >>=, >>>=	a=3 x%=2
1	Kommaoperator	,	int a,b

In Ausdrücken mit Operatoren gleicher Priorität entscheidet die **Assoziativität** über die Reihenfolge der Bewertung. Beispielsweise sind **/** und ***** in der gleichen Prioritätsstufe. Hier wird von links nach rechts ausgewertet (linksassoziativ).

Aufgabe 1

Welchen Wert und welchen Typ haben die in der Tabelle angegebenen Ausdrücke?
Die Auswertung eines einzelnen Ausdrucks soll dabei unabhängig von den jeweils anderen erfolgen.
Sie finden in den ersten Zeilen der Tabelle Beispiele.

a) Geben Sie folgende Werte in die main-Methode ein.
   1. int a, b = 4, c = -2, d=7, e=14, x=1,y=2;
   2. boolean f = false;
   3. String s = "sss";
   4. float g=2.8f;
   5. char h= 'c';
   6. double i=1.0;
   7. byte j=126, k=65;

b) Versuchen Sie zuerst durch Überlegen herauszufinden, welchen Typ und welchen Wert folgende Ausdrücke haben. Geben Sie sie Werte dieser Ausdrücke anschließend auf der Konsole aus (mit `Sytem.out.println(…)`)

Ausdruck	Wert	Typ
`b=c*3`	-6	int
`(b==4)`	true	boolean
`c + b`		
`s + "b"`		
`s + b`		
`a = (int) 0.5*2`		
`(c = 5) < 1`		
`f != true`		
`!(f == true)`		
`!(f = true)`		
`++c * 2 + --b -1`		
`g+d/e`		
`b++ * 2 + (1 + --c) / 2 + b`		
`h++`		
`i=g/b-c`		
`j|k`		
`j&k`		
`j^k`		
`e > d ? d-e : g+e`		
`++x << 2*2+(1+--y) /2+x`		

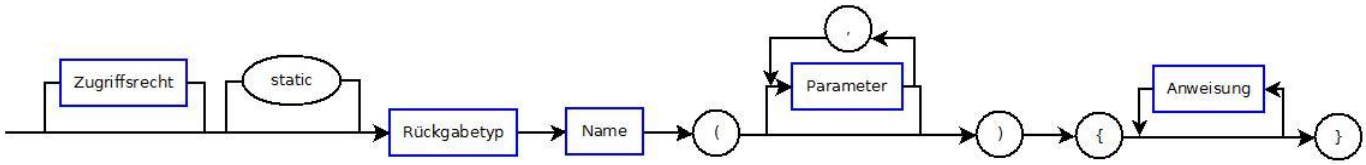
Aufgabe 2

Mit welchen Datentypen würden Sie folgende Größen darstellen?

a) Die Durchschnittstemperatur in einem Jahr.
b) Die Anzahl der Schüler einer Schule.
c) Die Entscheidung, ob eine Wahl gültig ist oder nicht.
d) Die Anzahl der Menschen auf unserem Planeten.
e) Der Buchstabe, der eine Eingabe beenden soll.

Methoden

Neben der main-Methode kann es in einer Klasse wie oben beschrieben auch weitere Methoden geben. Da die main-Methode so eine wichtige Bedeutung hat, werden wir Sie auch nur zum Ausführen anderer Methoden nutzen. Die Syntax von Methoden sieht allgemein wie folgt aus.



Eine Methode besteht mindestens aus einem Rückgabetyp, einem Methodennamen, einem Paar runder Klammern und einem Paar geschweifter Klammern. Da Sie sich in diesem Halbjahr fast ausschließlich mit Klassenmethoden beschäftigen, werden Sie stets vor den Rückgabetyp das Schlüsselwort **`static`** voranstellen und zudem als Zugriffsrecht das Schlüsselwort **`public`**.

Bei jeder Methode muss angegeben werden, welchen Wert man zurückgeben will. Möchte man keinen Wert zurückgeliefert bekommen, so gibt man dies durch das Schlüsselwort **`void`** an. Diese Methode nennt man dann eine **Prozedur**.

```
public class Methoden {

  public static int i=8;

  public static void erhoehen(){
    i=i+10;
  }
}
```

In unserem Beispiel ist die Klassenvariable i mit dem Wert 8 gegeben. Durch die Erhöhung wird der Wert der Variablen um 10 erhöht. Wenn man allerdings das Programm ausführt passiert nichts. Warum???

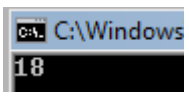
Es wird nur das ausgeführt, was in der main-Methode steht. Also rufen wir diese Methode innerhalb der main-Methode auf…

```
public class Methoden {

  public static int i=8;

  public static void main(String[] args) {
    erhoehen();
  }

  public static void erhoehen(){
    i=i+10;
  }
}
```

… und schon wird die Methode aufgerufen und der Wert erhöht. Allerdings kann man dies nicht erkennen.

Daher bauen wir aus Testzwecken wieder unsere Ausgabemethode ein.

Schreiben Sie nach der Anweisung in der Prozedur erhoehen eine Ausgabe des Werts i (mittels `System.out.println(i)`) und so kann man das Ergebnis betrachten.



Schöner wäre es noch, wenn wir den Wert zum Erhöhen selbst festlegen könnten. Daher kann man einer Methode auch **Parameter** mitgeben. Dazu muss der Typ des Parameters und der Name in die runden Klammern der Methode geschrieben werden.

```
  public static void erhoehen(int n){
    i=i+n;
    System.out.println(i);
  }
```

Der Wert von i wird jetzt um n erhöht. Die gewünschte Zahl muss man in die Klammern beim Aufruf in der main-Methode angeben.

```
    erhoehen(6);
```

Viel öfter kommt es aber vor, dass man eine Methode dazu benutzt etwas zu berechnen und diesen Wert wieder zurückbekommen möchte, um damit weiterzuarbeiten. Methoden, die auf diese Art aufgebaut sind, nennt man **Funktionen**.

In diesem Fall muss man als Rückgabetyp den Datentyp angeben, den man gerne zurückbekommen möchte. In unserem Beispiel ist dies eine ganze Zahl – also der Datentyp int. Innerhalb der Methode gibt man den Wert an, den man zurückgibt. Dies geschieht durch das Schlüsselwort **return**.

```
public class Methoden {

  public static int i=8;

  public static void main(String[] args) {
    int wert=erhoehen(10);
    System.out.println(wert);
  }

  public static int erhoehen(int n){
    i=i+n;
    return i;
  }
}
```

Die Methode erhoehen ist nun eine Funktion, die einen Parameter vom Typ int bekommt und einen int-Wert zurückliefert.

In der main-Methode speichern wir diesen Wert in der Variablen wert und geben anschließend den Inhalt von wert auf der Konsole aus.

Aufgaben

Aufgabe 1: Flächenberechnung

Implementieren Sie eine Funktion, die Radius eines Kreises übergeben bekommt und den Flächeninhalt zurückgibt.

Aufgabe 2: Umfang eines Kreises

Implementieren Sie eine Prozedur, die den Umfang eines Kreises berechnet und ausgibt.

Aufgabe 3: Addieren / Subtrahieren

Implementieren Sie eine Prozedur **addieren** und eine Prozedur **subtrahieren**, die jeweils zwei int-Werte als Parameter übergeben bekommen und als Ausgabe die Summe bzw. die Differenz dieser Werte zurückgibt.

Aufgabe 4: Multiplizieren / Dividieren

Implementieren Sie eine Funktion **multiplizieren** und eine Funktion **dividieren**, die jeweils zwei int-Werte als Parameter übergeben bekommen und als Ausgabe das Produkt bzw. den Quotienten dieser Werte ausgeben. Was passiert, wenn Sie durch 0 teilen???

Aufgabe 5: Methode dokumentieren

Method Summary

static void	addiere(int a, int b) Diese Methode addiert zwei int-Werte
static int	dividiere(int a, int b) Diese Methode dividiert zwei int-Werte

Mithilfe von /** … */ vor den Methoden werden JavaDoc-Kommentare erzeugt. Achten Sie darauf, dass Ihre Methoden immer dokumentiert sind.

Kontrollstrukturen

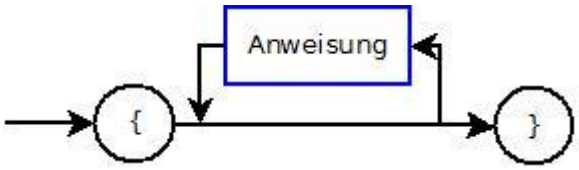
Bislang sind die Java-Programme eher langweiliger Natur. In der main-Methode werden andere Methoden aufgerufen, die Ergebnisse zurückliefern oder aber direkt etwas ausführen. Mithilfe der nächsten Konstrukte sollen neue Anweisungen gebildet werde. Diese Kontrollstrukturen in Algorithmen ermöglichen so viel kreativere Anwendungen. Diese Konstrukte sind:

- Sequentielle Anweisungen (Anweisungsblock)
- Bedingte Anweisungen (if-Anweisung, switch-Anweisung)
- Schleifen (while, for, do-while)

Alle Algorithmen lassen sich prinzipiell aus Zuweisungen und diesen drei Konstrukten aufbauen. Eine Sprache, die sich mit diesen Konstrukten begnügt, nennt man eine Kernsprache. Es ist aber einfacher, zusätzliche Konstrukte zu nutzen.

Der Anweisungsblock

Die einfachste strukturierte Anweisung ist der Anweisungsblock. Man spricht von einer **Sequenz** von Anweisungen. Eine Folge von Anweisungen wird zu einer einzigen Einheit zusammengefasst. Diese ist selbst wieder **eine** Anweisung. Ein Anweisungsblock wird benutzt, um z.B. in einer if- oder while-Anweisung mehrere Anweisungen logisch zusammenzufassen.



Anweisungsblöcke sind an den Stellen im Programm zulässig, an denen auch eine Anweisung stehen darf.

```
public class Kontrollstrukturen {

  public static int a=3;
  public static double b=4;

  /** Diese Methode verdeutlicht Anweisungsblöcke*/
  public static void benutzen(){
    int c=5;
    double d=7.8;
    double e=c+d*a;
    System.out.println(e+b);
  }

  public static void main(String[] args) {
    benutzen();
    System.out.println(a);
  }
}
```

In dem Beispiel links erkennt man zwei Klassenvariablen und innerhalb der Methode `benutzen` drei weitere Variablen. Diese Variablen sind nur innerhalb des Blockes gültig, in denen sie deklariert wurden. Man spricht hierbei von **lokalen Variablen**.

Auf Klassenvariablen hingegen kann man von überall her zugreifen.

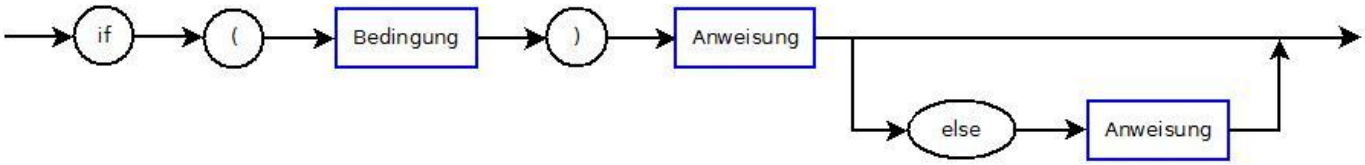
Die Bedeutung der Blockanweisungen wird bei der Betrachtung der nächsten Kontrollstrukturen ersichtlich.

Bedingte Anweisungen

Oftmals müssen in Algorithmen Entscheidungen auf vorgegebenen Kriterien getroffen werden. Hierfür sind die Bedingten Anweisungen (if, switch) zuständig.

If-Anweisung

Bei der if-Anweisung wird zuerst die Bedingung in den Klammern überprüft. Ist dieser boolesche Term **wahr (true)**, wird die folgende Anweisung, bzw. der folgende Anweisungsblock, ausgeführt. Wenn eine Alternative durch das **else** gegeben wird, so wird diese ausgeführt, wenn die Bedingung **falsch (false)** war. Andernfalls wird die Anweisung übergangen und das Programm mit der ersten Anweisung nach der if-Anweisung fortgesetzt.




```
/** Beispielmethode für Bedingte Anweisung*/
public static void eingabe(int wert1, int wert2){
  int differenz;
  int kleiner;
  if(wert1>wert2){
    differenz=wert1-wert2;
    kleiner=wert2;
  }
  else{
    differenz=wert2-wert1;
    kleiner=wert1;
  }
  if(differenz !=0)
    System.out.println(differenz/kleiner);
}
```

In unserer Prozedur werden zwei ganze Zahlen eingelesen. Anschließend wird die Differenz zwischen der größeren und der kleineren Zahl gebildet.

Wenn diese Differenz nicht 0 ist, so wird der Quotient zwischen der Differenz und der kleineren Zahl ausgegeben.

Selbstverständlich ist es auch erlaubt `if`-Anweisungen zu **verschachteln**, indem man im `if`- oder `else`-Teil eine weitere `if`-Anweisung einbaut. Wird allerdings bei geschachtelten `if-else`-Anweisungen nicht geklammert, kann es zum sogenannten **dangling-else-Problem** kommen. In Java gilt dann die Regel, dass „freie" `else`-Zuweisungen immer dem nächsten inneren `if` zugeordnet werden.

```
/** Durch die Einrückung kann man viel erkennen!*/
public static void bedingteAnweisung(int n, int m){
  if(n>100)
    if(m<100)
      System.out.println("m ist kleiner n");
    else
      System.out.println("m ist größer gleich n");
}
```

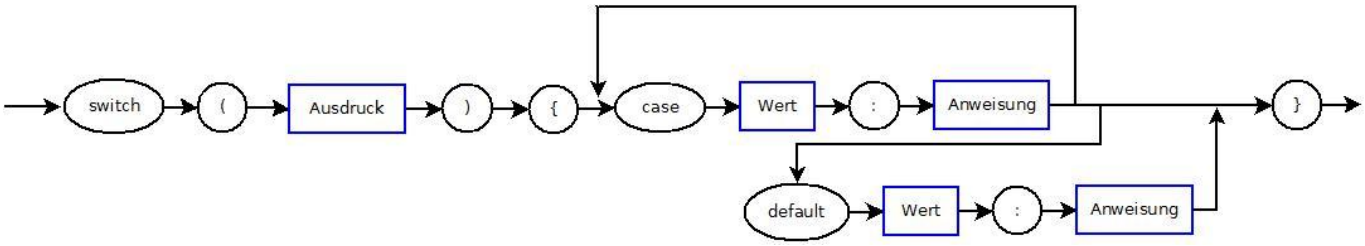
Das letzte `else` gehört zu dem inneren `if`. Dies kann man anhand der Einrückung sehr gut erkennen.

Einfache if-else-Verzweigungen können auch mit Hilfe des Bedingungsoperators (?:) ausgedrückt werden. Diese Anweisung ist sehr kurz, aber für viele auch schwer zu lesen.

```
/** Beispielmethode von oben mit Bedingungsoperator*/
public static void eingabe(int wert1, int wert2){
  int kleiner=(wert1>wert2) ? wert2 : wert1;
```

Switch-Anweisung

Die switch-Anweisung ist eine Mehrfachverzweigung, die eine numerische Variable oder einen Ausdruck mit mehreren Konstanten vergleicht.



Eine übliche Vorgehensweise beim Programmieren in jeder Sprache ist das Testen einer Variablen auf einen bestimmten Wert. Falls sie nicht zu diesem Wert passt, wird sie anhand eines anderen Wertes geprüft, und falls dieser wieder nicht passt, wird wieder mit einem anderen Wert geprüft.

Werden nur if-Anweisungen verwendet, kann das, je nachdem wie der Quelltest formatiert wurde und wie viele verschiedene Optionen geprüft werden müssen, sehr unhandlich sein. Eine Kurzform anstelle verschachtelter if-Anweisungen ermöglicht es Ihnen, in manchen Fällen Tests und Aktionen gemeinsam in einer Anweisung auszuführen.

Bei der `switch`-Anweisung wird der Wert des Ausdrucks mit den Konstanten der `case`-Marken verglichen. Stimmt der Wert überein, wird die folgende Anweisung ausgeführt. Entspricht keine `case`-Marke diesem Ausdruck, so springt die Programmausführung zur `default`-Marke, wenn diese existiert.

```
/** Wandelt Zahl in eine Tagausgabe um*/
public static void Wochentag(int tag){
  switch(tag) {
    case 1:System.out.println("Montag"); break;
    case 2:System.out.println("Dienstag"); break;
    case 3:System.out.println("Mittwoch"); break;
    case 4:System.out.println("Donnerstag"); break;
    case 5:System.out.println("Freitag"); break;
    case 6:System.out.println("Samstag"); break;
    case 7:System.out.println("Sonntag"); break;
    default: System.out.println("Ihr Fehler!");
  }
```

Die `break`-Anweisung nach jedem `case`-Fall bewirkt, dass nach der Abarbeitung der Anweisung die `switch`-Anweisung verlassen wird. Ansonsten würde jeder Fall abgearbeitet. Als `Switch`-Ausdruck können Werte vom Typ byte, short, int oder char verwendet werden.

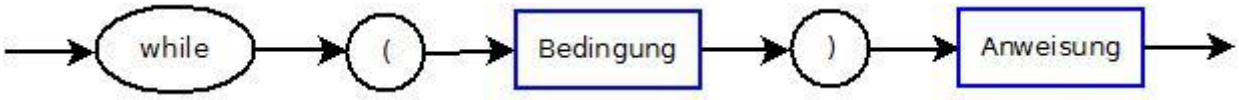
Aufgabe 1: Dreiecksarten
Die Methode dreieck nimmt drei Seitenlängen der Größe nach sortiert entgegen (die kürzeste Seite zuerst) und macht dann eine Aussage zu dem Dreieck (spitz-, recht- oder stumpfwinklig, gleichschenklig, gleichseitig).

Aufgabe 2: Rechnung der Bank
Eine Bank führt die ersten 10 Buchungen auf einem Girokonto kostenlos durch, die nächsten 10 kosten 0,50 Euro und jede weitere 0,40 Euro. Eine Methode soll berechnen, wie viel Geld ein Kunde bei einer bestimmten Anzahl von Buchungen zu Zahlen hat.

Schleifen

Oftmals ist es wünschenswert, dass eine Anweisung nicht nur einmal, sondern beliebig oft ausgeführt wird. (Beispielsweise die Summation aller Zahlen von 1 bis 100, eine zehnmalige Ausgabe einer Leerzeile,…). In diesen Fällen bietet sich die Schleife als Kontrollstruktur an. Hierbei gibt es drei unterschiedliche Schleifen.

Die while-Schleife



In der While-Schleife wird zuerst die Bedingung, ein boolescher Ausdruck, überprüft. Wenn das Ergebnis **true** liefert, wird die Anweisung abgearbeitet und die Bedingung erneut ausgewertet. Dies geschieht solange, bis der Wert false liefert.

Wenn die Bedingung **false** liefert, wird die Anweisung nicht ausgeführt und das Programm verlässt die Schleife. Sollen mehrere Anweisungen wiederholt werden, müssen diese in einen Anweisungsblock zusammengefasst werden.

```
/** gibt die Quersumme einer Zahl zurück*/
public static int quersumme(int zahl){
  int ergebnis=0;
  while(zahl>0){
    ergebnis += zahl%10; // verkürzte Zuweisung
    zahl /=10;           // verkürzte Zuweisung
  }
  return ergebnis;
}
```

Es wird entschieden, ob der Wert der Variablen zahl größer 0 ist. In diesem Fall wird zu dem Wert von `ergebnis` die letzte Ziffer der Zahl hinzuaddiert und die Zahl anschließend durch 10 geteilt (bei int-Zahlen wird hierbei die letzte Ziffer entfernt). Dies wird solange ausgeführt, bis die Zahl den Wert 0 hat. Erst dann erfolgt die Rückgabe des Wertes.

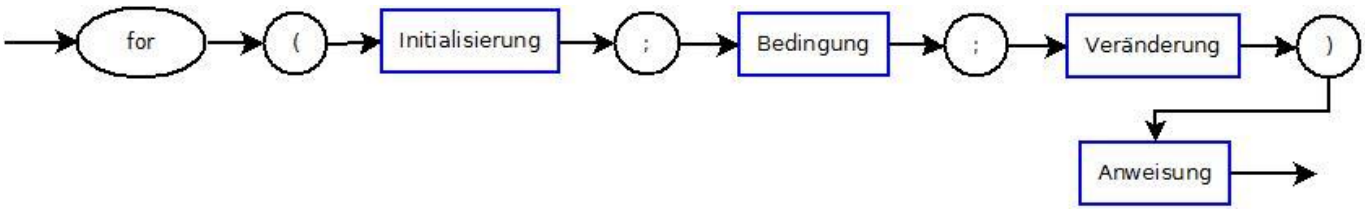
Meist wird für die while-Schleife eine **Schleifenvariable** eingerichtet, die die Ausführung der Schleife steuert. Dabei durchläuft die Schleifenvariable folgende Schritte:

1) Vor der Schleife wird die Variable initialisiert.
2) Am Anfang der Schleife wird mit ihr die Bedingung ausgewertet.
3) In der Schleife wird der Wert verändert (meist inkrementiert oder dekrementiert).

```
public class WhileSchleife {
  /** Das kleine 1*1 einer Zahl wird ausgegeben */
  public static void schleife(int i){
    int n=1;          // 1)Initialisierung
    while(n<=10){     // 2)Auswertung der Bedingung
      System.out.println(n+"*"+i+" ist: "+n*i);
      n++;            // 3)Inkrementieren
    }
  }
}
```

Da man bei der while-Schleife **vor** der Durchführung der Schleife die Bedingung überprüft, sagt man, dass diese Schleife eine **kopfgesteuerte Schleife** ist.

Die for-Schleife



Die for-Schleife ist eine weitere Variantenform, die der while-Schleife sehr ähnelt. In dieser Variante werden drei Anweisungen zur Veränderung und Kontrolle der Schleifenvariablen (Initialisierungsteil, Bedingung, Veränderung) übersichtlich im Schleifenkopf dargestellt. Eingesetzt wird die for-Schleife vor allem dann, wenn die Anzahl der Schleifeniterationen bereits vor dem Schleifendurchlauf feststeht.

Man kann eine while-Schleife in eine for-Schleife umwandeln und umgekehrt.

for-Schleife

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

while-Schleife

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

Zu Beginn der for-Schleife wird die Initialisierungsanweisung aus dem Schleifenkopf ausgeführt. Hierbei wird meist die Variable auch deklariert und ist nur im Kopf und Schleifenrumpf gültig.

Anschließend werden die Anweisungen so lange wiederholt ausgeführt, wie die formulierte Bedingung gültig ist. Nach jedem Schleifendurchlauf wird die Schleifenvariable noch vor der erneuten Überprüfung verändert.

Übung 1

a) Implementieren Sie mit einer for-Schleife eine Methode **nichtDurchDrei()**, die alle Zahlen von 1 bis 99, die nicht durch 3 teilbar sind aufaddiert und ausgibt.
b) Implementieren Sie die eine Methode, die die gleiche Funktionalität hat, aber eine while-Schleife benutzt.

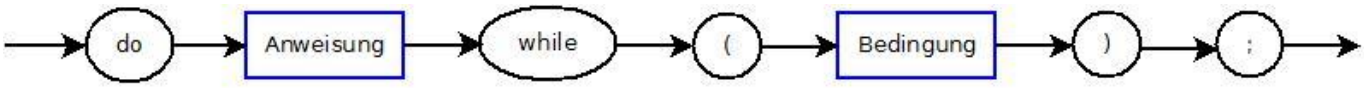
Übung 2

Implementieren Sie eine Methode **summeBis(int n)**, die alle Zahlen bis zu n addiert und das Ergebnis auf der Konsole (mit **System.out.println();**) ausgibt.

Übung 3

a) Implementieren Sie eine Methode kgV, die das *kleinste gemeinsame Vielfache* zweier Zahlen m und n in einer **while-Schleife** berechnet und ausgibt.
b) Implementieren Sie ebenso eine Methode mit einer for-Schleife.

Die do-while-Schleife



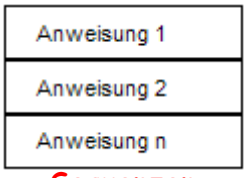
Die do-while-Schleife ist eng mit der while-Schleife verwandt. Der einzige wichtige Unterschied besteht darin, dass bei der while-Schleife ebenso wie bei der for-Schleife die Bedingung **vor** dem Anweisungsblock getestet wird, während die do-while-Schleife die Bedingung erst **nach** der Durchführung der Anweisungen überprüft. Eine do-while-Schleife wird also mindestens einmal ausgeführt. Daher nennt man Sie auch eine **fußgesteuerte Schleife**.

Mit all diesen grundlegenden Kontrollstrukturen kann man jetzt verschiedenartige Probleme lösen. Jetzt fehlt einem nur noch die Logik – und vor allem die Übersicht hierfür!

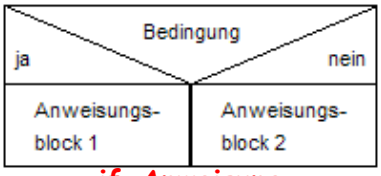
Ein Mittel stets alles im Blick zu behalten ist die Skizzierung mit Hilfe eines Struktogramms.

Struktogramme

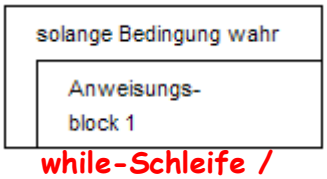
Eine Möglichkeit die Struktur eines Algorithmus zu beschreiben ist durch ein **Struktogramm**. Hier werden die einzelnen Bestandteile des Algorithmus durch verschiedene geometrische Formen repräsentiert.



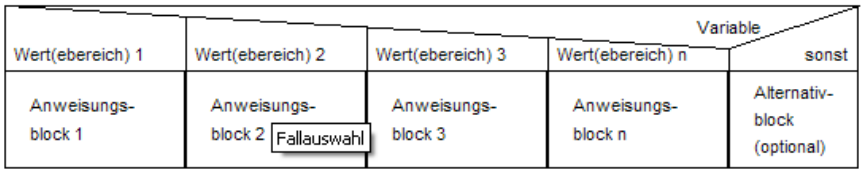
Sequenzen



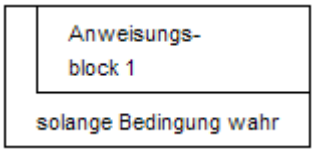
if-Anweisung



while-Schleife / for-Schleife

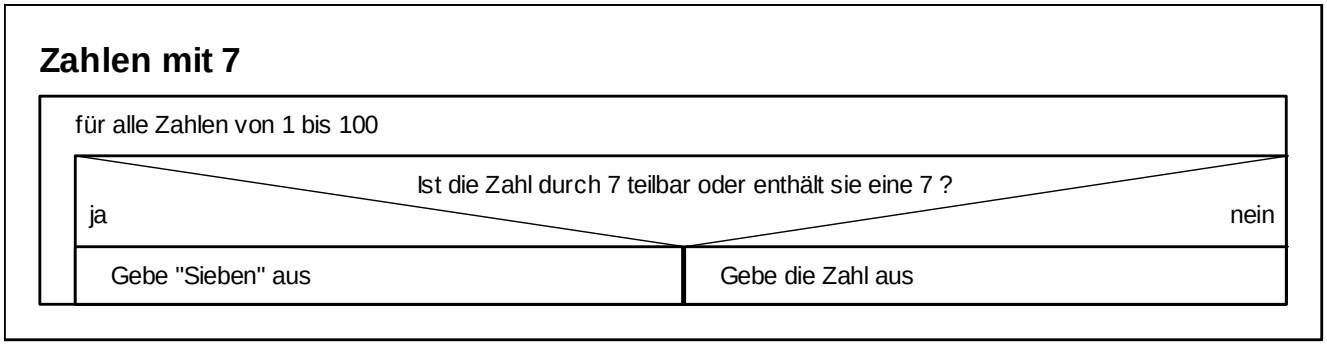


switch-Anweisung



do-while-Schleife

Sie können hier alle erlernten Kontrollstrukturen erkennen. Ein kleines Beispiel hierzu ist das unten dargestellte. Wie Sie erkennen können, achtet man hierbei nicht auf die Java-Syntax, denn diese Darstellungsform soll universell für alle Sprachen einsetzbar sein.



Arrays

Die bisherigen Programmbeispiele, die Sie kennengelernt haben, kamen stets mit einer kleinen Anzahl von Daten / Variablen aus. Aber wie sieht dies bei einer sehr großen Anzahl von Daten (Messergebnissen, Adressen,…) aus?

Die Informatik hat sich schon sehr lange mit der effizienten Verwaltung größerer Datenmengen in einem Programm auseinandergesetzt. Die so entstandenen Grundmodelle sind Listen, Stacks, Queues, Hashes oder eben Arrays.

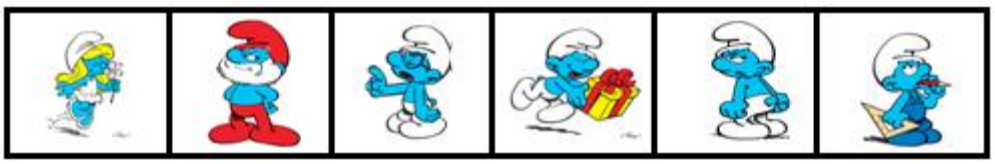
Arrays fassen Werte **gleichen Typs** zu einer Einheit zusammen. Dabei wird ein Array durch das Zeichen **[]** symbolisiert. In dem Beispiel unten ist für Gargamel kein Platz im Array, da er nicht vom Typ **Schlümpfe** ist.

Beispiel:

Erzeugung eines „Schlumpf-Arrays" mit 6 Schlümpfen

```
Schlümpfe [] schlumpf = new Schlümpfe [6];
```

Schlümpfe-Array neuer Schlümpfe-Array mit 6 Plätzen



schlumpf[0] schlumpf[1] schlumpf[2] schlumpf[3] schlumpf[4] schlumpf[5]

Die einzelnen, in einem Array abgelegten Werte können über einen **Index** angesprochen werden. Der Index wird in eckigen Klammern angegeben. Er ist immer eine ganze Zahl und beginnt bei 0. Also gilt: schlumpf[0] =Schlumpfine, schlumpf[1] = Papa Schlumpf, …

Wesentliche Eigenschaften von Arrays:

- Ein Array kann nur Elemente **gleichen Typs** enthalten.
- Jeder Array besitzt die Variable **length**, die die Anzahl der Elemente im Array angibt. Für den Array **schlumpf** gilt daher: **schlumpf.length**=6.
- Der Wertebereich für den Laufindex eines Arrays ist **0** bis **length-1**.

Die Anzahl der Elemente in einem Array wird bei der Erzeugung des Arrays zur Programmlaufzeit festgelegt und kann danach nicht mehr verändert werden. (Man sagt: **Arrays sind statisch**).

```
//deklarieren eines neuen Arrays vom Typ double
      double[] a;
      //am Anfang muss die Länge festgelegt werden
      a=new double[10];
      // oder zusammen:
      double[] a=new double[10];
      // jetzt kann man mit den einzelnen Zellen arbeiten
      a[6]=12.6;
      a[0]=9;

      //Initialisierungsliste
      int[] b={1,2,3,4,5,6,7,8,9,10};
```

Noch ein kleines Beispiel zu den Arrays, an dem auch wiederum schön die Struktur der for-Schleife verständlich wird.

```
/**Erzeugt einen int-Array mit n Feldern und füllt ihn mit
* Zufallszahlen. Anschließend werden alle Zahlen ausgegeben. */
  public static void zufallsArray(int n){
    int[] a=new int[n];
    for(int i=0;i<a.length;i++){
      a[i]=(int)(Math.random()*100);
    }
    for(int i=0;i<a.length;i++)
      System.out.println(a[i]);
  }
```

Zuerst wird der int-Array a mit der Länge n erzeugt. Wenn man nun jedes Element mit einem Wert belegen möchte, so muss man alle Elemente einzeln ansprechen. Dies geschieht hier mit der for-Schleife. (Hierbei haben wir uns eine Methode aus der Klasse Math bedient, die Zufallszahlen erzeugt. Leider liefert diese Funktion einen double-Wert zurück. Um dies aber in einem int-Feld abzuspeichern, müssen wir die Zahl casten – wir schneiden also die Kommastellen ab).

Übung 1

Implementieren Sie eine Methode, die

a) ein Array von 100 ganzen Zahlen erzeugt und dieses mit den Werten 0^2, 1^2, 2^2, ... belegt und

b) danach die Zahlen von 99^2 bis 0^2 ausgibt.

Übung 2

Erstellen Sie eine Klasse, welche die Ziehung von Lottozahlen simuliert. Dazu soll ein Array mit sieben Elementen benutzt werden. Die Werte sollen zufällig ermittelt werden. Dabei kann zur Vereinfachung ignoriert werden, ob eine Zahl schon gezogen wurde. Hinweis: (int) (Math.random() * 5) liefert zufällige Zahlen zwischen 0 und 4.

Übung 3

Für die ganzen Zahlen von 0 bis 100 soll in einem Array *teilbar* gespeichert werden, ob sie durch (mindestens) eine der drei Zahlen 2, 3 oder 5 teilbar sind (**true**) oder nicht (**false**).
Gegeben sei folgender Quellcode-Ausschnitt der Klasse Teilbar:

```
public class Teilbar {
      /** Testmethode */
      public static void main(String[] args) {
        boolean[] teilbar = new boolean[101];
        ausgabe(fuelle(teilbar));
      }

      /** Gibt die einzelnen Felder eines boolean-Arrays aus*/
      public static void ausgabe(boolean[] b){
        for(int i=0;i<b.length;i++)
          System.out.println("teilbar["+i+"]="+b[i]);
      }
}
```

Implementieren Sie die Methode **fuelle**, die den Array *teilbar* nach den oben angegebenen Kriterien mit Werten füllt.