Gleitkommazahlen

Die Typen float und double

Variablen

```
float x, y;          // 32 Bit groß
double z;            // 64 Bit groß
```

Konstanten

```
3.14                 // Typ double
3.14f                // Typ float
3.14E0               // 3.14 * 10 0
0.314E1              // 0.314 * 10 1
31.4E- 1             // 31.4 * 10 -1
.23
1. E2                // 100
```

Syntax der Gleitkommakonstanten

```
FloatConstant   = [Digits] "." [Digits] [Exponent] [FloatSuffix].
Digits          = Digit {Digit}.
Exponent        = (" e" | "E") ["+" | "-"] Digits.
FloatSuffix     = "f" | "F" | "d" | "D"..
```

Beispiel: Berechnung der harmonischen Reihe

sum = 1/ 1 + 1/ 2 + 1/ 3 + ... + 1/ n

```
class HarmonicSequence {
    public static void main (String[] arg) {
      float sum = 0;
      int n = Integer.parseInt(arg[0]);
      for (int i = n; i > 0; i--)
         sum += 1f / i ;
       System.out. println(" sum = " + sum);
    }
}
```

Was würden statt $1f/i$ folgende Ausdrücke liefern?

1 / i	0 für alle i>1 (weil ganzzahlige Division)
1.0 / i	einen double- Wert

Zuweisungen und Operationen

Zuweisungskompatibilität

double ⊇ float ⊇ long ⊇ int ⊇ short ⊇ byte

```
float f; int i;
f = i;       // erlaubt
i = f;       // verboten
i = (int) f; // erlaubt: schneidet Nachkommastellen ab; falls zu groß: maxint, minint
f = 1.0;     // verboten, weil 1.0 vom Typ double ist
f = 1.0f;    // ok
```

Erlaubte Operationen

- Arithmetische Operationen (+, -, *, /)
- Vergleiche (==, !=, <, <=, >, >=)
  Achtung: Gleitkommazahlen sollte man nicht auf Gleichheit prüfen.

Typen von Gleitkommaausdrücken

Der "kleinere" Operandentyp wird in den "größeren" konvertiert, zumindest aber in int.

double ⊇ float ⊇ long ⊇ int ⊇ short ⊇ byte

```
double d; float f; int i; short s;
...
d + i      // double
f + i      // float
s + s      // int
```

Ein- / Ausgabe von Gleitkommazahlen

```
double d = Double.parseDouble(arg[0]);
float f = 3.14f;
System.out. println(" d = " + d + ", f = " + f);
```

Methoden

Parameterlose Methoden

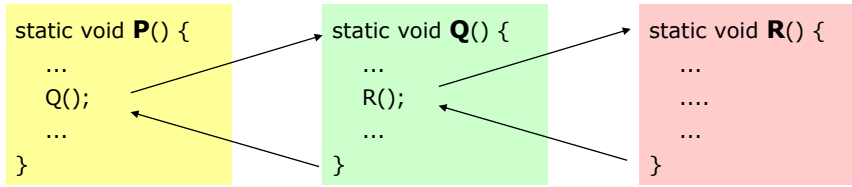
Beispiel: Ausgabe einer Überschrift

```
class Sample {

      static void printHeader() {          // Methodenkopf
        System.out. println("Liste");      // Methodenrumpf
        System.out. println("--------");
      }

      public static void main (String[] arg) {
         printHeader();                    // Methodenaufruf
          ...
         printHeader();
         ...
       }
}
```

Zweck von Methoden

- Wiederverwendung häufig benutzten Codes.
- Definition benutzer-spezifischer Operationen.
- Strukturierung des Programms.

Wie funktioniert ein Methodenaufruf?



Namenskonventionen für Methoden:

Namen sollten mit Verb und Kleinbuchstaben beginnen

Beispiele: *printHeader, findMaximum, traverseList, ...*

Parameter

Werte, die vom Rufer an die Methode übergeben werden

```
class Sample {

    static void printMax(int x, int y) {
        if (x > y)      System.out.print( x);
        else            System.out.print( y);
    }

    public static void main (String[] arg) {
        ...
        printMax( 100, i * 2 );
    }
}
```

formale Parameter
- im Methodenkopf (hier x, y)
- sind Variablen der Methode

aktuelle Parameter
- an der Aufrufstelle (hier 100, 2* i)
- können Ausdrücke sein

Parameterübergabe

Aktuelle Parameter werden den entsprechenden formalen Parametern zugewiesen.

x = 100; y = 2 * i;

Aktuelle Parameter müssen mit formalen Parametern zuweisungskompatibel sein.

Formale Parameter enthalten Kopien der aktuellen Parameter,

Funktionen

Methoden, die Ergebniswerte an den Rufer liefern.

```
class Sample {

    static int Max(int x, int y) {
        if (x > y) return x; else return y;
    }

    public static void main (String[] arg) {
        ...
        System.out.println( Max( 100, i +j ) + 1);
    }
}
```

- Haben Funktionstyp (z. B. *int*) statt void (= kein Typ).
- Liefern Ergebnis mittels return Anweisung an den Rufer.
  (x muß zuweisungskompatibel mit int sein)
- Werden wie Operanden in einem Ausdruck benutzt.

Funktionen Methoden mit Rückgabewert
Prozeduren Methoden ohne Rückgabewert

Weiteres Beispiel

Ganzzahliger Zweierlogarithmus

```
class Sample {

    static int log2 (int x) {     // assert: x >= 0
        int res = 0;
        while (x > 1) {x = x / 2; res++;}
        return res;
    }

    public static void main (String[] arg) {
        int x = log2(17) ;        // x == 4
        .....
    }
}
```

Return in Prozeduren

```
class ReturnDemo {

    static void printLog2 (int x) {
        if (x < 0) return;                // kehrt zum Rufer zurück
        int res = 0;
        while (x > 1) {x = x / 2; res++;}
        System.out.println( res);
    }

    public static void main (String[] arg) {
        int x = In. readInt();
        printLog2( x);
        ...
        if (! In. done()) return;         // beendet das Programm
}
}
```

Funktionen	müssen mit return beendet werden.
Prozeduren	können mit return beendet werden.

Lokale und statische Variablen

```
class C {
   static int a, b;
   static void P() {
      int x, y;
      .....
   }
   ...
}
```

Statische Variablen
auf Klassenebene mit ***static*** deklariert;
auch in Methoden dieser Klasse sichtbar.

Lokale Variablen
in einer Methode deklariert
(lokal zu dieser Methode; nur dort sichtbar).

Reservieren und Freigeben von Speicherplatz

Statische Variablen
am Programmbeginn angelegt,
am Programmende wieder freigegeben.

Lokale Variablen
bei jedem Aufruf der Methode neu angelegt,
am Ende der Methode wieder freigegeben.

Beispiel: Summe einer Zahlenfolge

falsch!

```
class Wrong {

    static void add (int x) {
       int sum = 0;
       sum = sum + x;
    }

    public static void main( String[] arg) {
       add( 1); add( 2); add( 3);
       Out. println(" sum = " + sum);
    }
}
```

richtig!

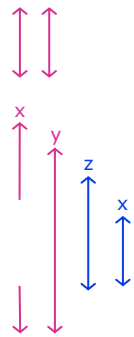
```
class Correct {

   static int sum = 0;

   static void add (int x) {
      sum = sum + x;
   }

   public static void main( String[] arg) {
      add( 1); add( 2); add( 3);
      Out. println(" sum = " + sum);
   }
}
```

- *sum* ist in *main* nicht sichtbar.
- *sum* wird bei jedem Aufruf von *add* neu angelegt (alter Wert geht verloren).

Sichtbarkeitsbereich von Namen

Programmstück, in dem auf diesen Namen zugegriffen werden kann
(auch *Gültigkeitsbereich* oder *Scope* des Namens genannt)

```
class Sample {

   static void P() {
    ...
   }
   static int x;
   static int y;
   static void Q( int z) {
      int x;
      ...
   }
}
```



Regeln

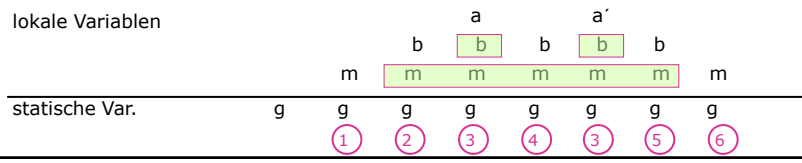
- Ein Name darf in einem Block nicht mehrmals deklariert werden (auch nicht in geschachtelten Anweisungsblöcken).
- Der Sichtbarkeitsbereich eines Namens beginnt bei seiner Deklaration und geht bis zum Ende seines Blocks.
- Lokale Namen verdecken Namen, die auf Klassenebene deklariert sind.
- Auf Klassenebene deklarierte Namen sind in allen Methoden der Klasse sichtbar.

Beispiel zu Sichtbarkeitsregeln

```
class Sample {
   static void P() {
      System.out. println( x);              // gibt 0 aus
   }
   static int x = 0;
   public static void main( String[] arg) {
      System.out. println( x);              // gibt 0 aus
      int x = 1;                            // verdeckt statisches x
      System.out. println( x);              // gibt 1 aus
      P();                                  // gibt 0 aus
      if (x > 0) {
          int x;                            // Fehler: x ist in main bereits deklariert
          int y;
      } else {
          int y;                            // ok, kein Konflikt mit y im then- Zweig
      }
      for (int i = 0; ...) {...}
      for (int i = 1; ...) {...}            // ok, kein Konflikt mit i aus letzter Schleife
   }
}
```

Lebensdauer von Variablen

```
class LifenessDemo {
   static int g;
   static void A() {
        int a;
        3
    }
    static void B() {
         int b;
        2  A(); 4 A(); 5
    }

    public static void main(String[] arg) {
        int m;
        1 B();  6
    }
}
```



Lokalität

Variablen möglichst lokal deklarieren, nicht als statische Variablen.

Vorteile

- Übersichtlichkeit

  Deklaration und Benutzung nahe beisammen.

- Sicherheit

  Lokale Variablen können nicht durch andere Methoden zerstört werden.

- Effizienz

  Zugriff auf lokale Variable ist oft schneller als auf statische Variable.

Überladen von Methoden

Methoden mit gleichem Namen aber verschiedenen Parameterlisten können in derselben Klasse deklariert werden.

```
static void write (int i) {...}
static void write (float f) {...}
static void write (int i, int width) {...}
```

Beim Aufruf wird diejenige Methode gewählt, die am besten zu den aktuellen Parametern paßt.

```
write( 100);          write (int i)
write( 3.14f);        write (float f)
write( 100, 5);       write (int i, int width)
short s = 17;
write( s);            write (int i);
```

Beispiel

Größter gemeinsamer Teiler nach Euklid

```
static int ggt (int x, int y) {
   int rest = x % y;
   while (rest != 0) {
      x = y;   y = rest;    rest = x % y;
   }
   return y;
}
```

Warum funktioniert dieser Algorithmus?

(*ggt* teilt x) & (*ggt* teilt y)
ggt teilt (x - y)
ggt teilt (x - q* y)
ggt teilt rest
ggt(x, y) = *ggt*(y, rest)

Kürzen eines Bruchs

```
static void reduce (int z, int n) {
   int x = ggt( z, n);
   Out. print( z/ x); Out. print("/"); Out. print( n/ x);
}
```

Beispiele

Prüfe, ob x eine Primzahl ist

```
static boolean isPrime (int x) {
   if (x == 1 || x == 2) return true;
   if (x % 2 == 0) return false;
   int i = 3;
   while (i * i <= x) {
      if (x % i == 0) return false;
      i = i + 2;
   }
  /*    i > √x und x ist durch keine Zahl
        kleiner als i teilbar                    */
return true;
}
```

```
for (int i = 3; i * i <= x; i += 2)
      if (x % i == 0) return false;
```

Beispiele

Berechne val^{exp}

```
static long power (int val, int exp) {
    long res = 1;
    for (int i = 1; i <= exp; i++) res = res * val;
 return res;
 }
```

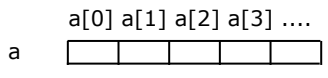
Dasselbe effizienter

```
static long power (int val, int exp) {
   long res = 1;
   while (exp > 0) {
      if (exp % 2 == 0) {
           val = val * val; exp = exp / 2;      // x^2n = (x*x)^n
       } else {
          res = res * val; exp--;               // x^n+1 = x*x^n
       }
   }
   return res;
 }
```

Arrays

Eindimensionale Arrays

Array = Tabelle gleichartiger Elemente



- Name **a** bezeichnet das gesamte Array.
- Elemente werden über Indizes angesprochen (z. B. **a[3]**).
- Indizierung beginnt bei 0.
- Elemente sind namenlose Variablen.

Deklaration

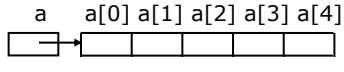
```
int[] a;
float[] b;
```

- Deklariert ein Array namens a (bzw. b).
- Seine Elemente sind vom Typ int (bzw. float).
- Seine Länge ist noch unbekannt.

Erzeugung

```
a = new int[5];
b = new float[10];
```

- Legt ein neues int- Array mit 5 Elementen an (aus dem Heap- Speicher).
- Weist seine Adresse a zu.



Array- Variablen enthalten Zeiger auf Arrays!

(Zeiger = Speicheradresse)

Arbeiten mit Arrays

Zugriff auf Arrayelemente

```
a[3] = 0;
int i =1;
a[ 3* i+ 1] = a[3];
```

- Arrayelemente werden wie Variablen benutzt.
- Index kann ein ganzzahliger Ausdruck sein.
- Laufzeitfehler, falls Array noch nicht erzeugt wurde.
- Laufzeitfehler, falls Index < 0 oder ≥ Arraylänge.

Arraylänge abfragen

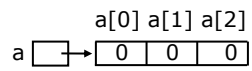
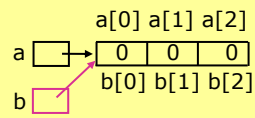
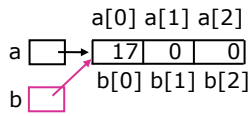
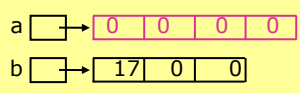
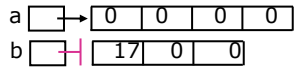
```
int len = a. length;
```

- length ist Standardoperator, der auf alle Arrays angewendet werden kann.
- Liefert Anzahl der Elemente (hier 5).

Beispiele

```
for (int i = 0; i < a. length; i++)      // Array einlesen
   a[ i] = In. readInt();

int sum = 0;                             // Elemente aufaddieren
for (int i = 0; i < a. length; i++)
    sum += a[ i];
```

Arrayzuweisung

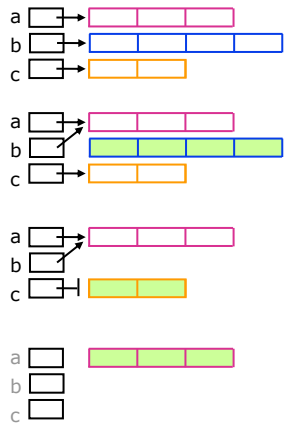
`int[] a, b;` `a = new int[ 3];`		Arrayelemente werden in Java standardmäßig mit 0 initialisiert .
`b = a;`		b bekommt denselben Wert wie a. Arrayzuweisung ist in Java Zeigerzuweisung!
`a[0] = 17;`		Ändert in diesem Fall auch b[0]
`a= new int[4];`		a zeigt jetzt auf neues Array.
`b = null;`		***null***: Spezialwert, der auf kein Objekt zeigt; kann jeder Arrayvariablen zugewiesen werden.

Freigeben von Arrayspeicher

Garbage Collection (Automatische Speicherbereinigung)

Objekte, auf die kein Zeiger mehr verweist, werden automatisch eingesammelt.
Ihr Speicher steht für neue Objekte zur Verfügung

```
static void P() {
   int[] a = new int[ 3];
   int[] b = new int[ 4];
   int[] c = new int[ 2];
   ...
   ...
   b = a;
   ...
   ...
  c = null;
  ...
  ...
  ...
}
```



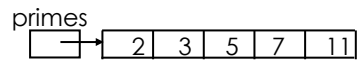
Kein Zeiger mehr auf dieses Objekt wird eingesammelt.

Kein Zeiger mehr auf dieses Objekt wird eingesammelt.

Am Methodenende werden lokale Variablen freigegeben, Zeiger ***a, b, c*** fallen weg, Objekt wird eingesammelt.

Initialisieren von Arrays

```
int[] primes = {2, 3, 5, 7, 11};
```



identisch zu

```
int[] primes = new int[ 5];
primes[ 0] = 2;
primes[ 1] = 3;
primes[ 2] = 5;
primes[ 3] = 7;
primes[ 4] = 11;
```

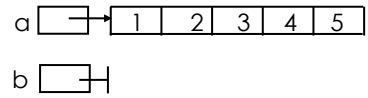
Initialisierung kann auch bei der Erzeugung erfolgen.

```
primes = new int[] {2, 3, 5, 7, 11};
```

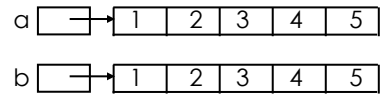
Kopieren von Arrays

int[ ] a = {1,2,3,4,5 };

int[ ] b;



b = (int[ ]) a. clone( ) ;



Typumwandlung nötig, da *clone* etwas vom Typ *Object[ ]* liefert.

Kommandozeilenparameter

Programmaufruf mit Parametern

java Programmname par_1 par_2 ... par_n

Parameter werden als String- Array an main- Methode übergeben

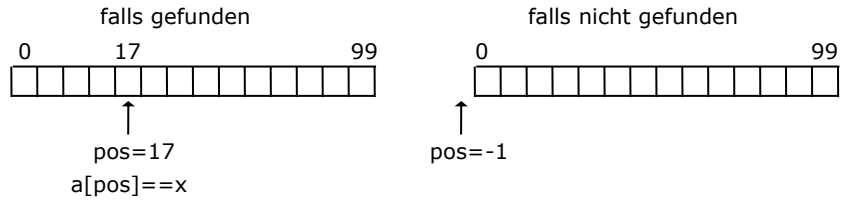
```
class Sample {
    public static void main (String[] arg) {
        for (int i = 0; i < arg. length; i++)
            System.out.println( arg[ i]);
            ...
    }
}
```

Aufruf z. B. : java Sample Anton /a 10

Ausgabe : Anton
/a
10

Beispiel: sequentielles Suchen

Suchen eines Werts x in einem Array mit 100 Einträgen



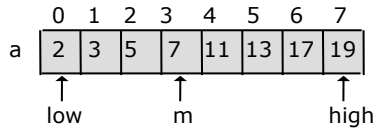

```
static int search (int[] a, int x) {
   int pos = a. length - 1;
   while (pos >= 0 && a[ pos] != x ) pos--;
   return pos;              // pos == -1 || a[ pos] == x
}
```

Achtung: **int[] a** wird nur als Zeiger übergeben.
Würde *search* etwas in a ändern (z. B. a[3] = 0;), würde sich diese Änderung auch auf das Array im Rufer auswirken.

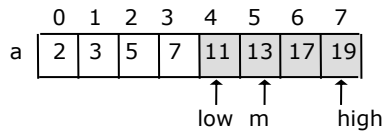
Beispiel: binäres Suchen

- Schneller als sequentielles Suchen.
- Array muß allerdings sortiert sein.

z. B. Suche von 13

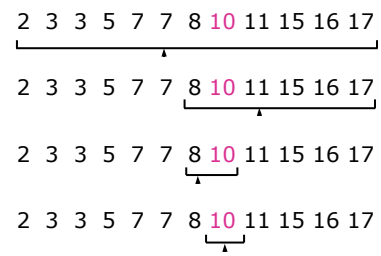


- Index des mittleren Elements bestimmen (m = (low + high) / 2)
- 13 > a[m] zwischen a[m+ 1] und a[high] weitersuchen



Binäres Suchen

```
static int binarySearch (int[] a, int x) {
    int low = 0;
    int high = a. length - 1;
    while (low <= high) {
        int m = (low + high) / 2;
        if (a[ m] == x) return m;
        else if (x > a[ m]) low = m + 1;
             else  high = m - 1; /*   x < a[ m]  */
    } /*  low > high  */
    return -1;
}
```



- Suchraum wird in jedem Schritt halbiert.
- Bei n Arrayelementen sind höchstens log2(n) Schritte nötig, um jedes Element zu finden

n	seq. Suchen	bin. Suchen
10	10	4
100	100	7
1000	1000	10
10000	10000	14

Primzahlenberechnung: Sieb des Erathostenes

1. "Sieb" wird mit den natürlichen Zahlen ab 2 gefüllt.
   2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, ...

2. Erste Zahl im Sieb ist Primzahl. Entferne sie und alle ihre Vielfachen.
   (2) 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, ...
   3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, ...

3. Wiederhole Schritt 2
   (3), 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, ...
   5, 7, 11, 13, 17, 19, 23, 25, ...

... Wiederhole Schritt 2
   (5), 7, 11, 13, 17, 19, 23, 25, ...
   7, 11, 13, 17, 19, 23, ...

Implementierung

Sieb = *boolean*- Array, Zahl *i* im Sieb ⇔ sieve[i] == *true*

0	1	2	3	4	5	6	7	8	9	
false	false	true	true	true	true	true	true	true	true	.....

Zahl *i* entfernen: sieve[i] = *false*

0	1	2	3	4	5	6	7	8	9	
false	false	false	true	false	true	false	true	false	true	.....

```
static void printPrimes (int max) {
   boolean[] sieve = new boolean[ max + 1];
   for (int i = 2; i <= max; i++) sieve[ i] = true;
   for (int i = 2; i <= max; ) {
         System.out. print( i + " ");              // i is prime
         for (int j = i; j <= max; j = j + i) sieve[ j] = false;
         while (i <= max && !sieve[ i]) i++;
    }
}
```

Beispiel: Monatstage berechnen

Bisher mit Switch- Anweisung gelöst

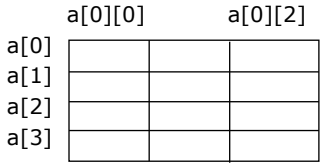
```
switch (month) {
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    days = 31; break;
  case 4: case 6: case 9: case 11:
    days = 30;
  case 2:
   days = 28;
}
```

Parameter werden als String- Array an main- Methode übergeben.

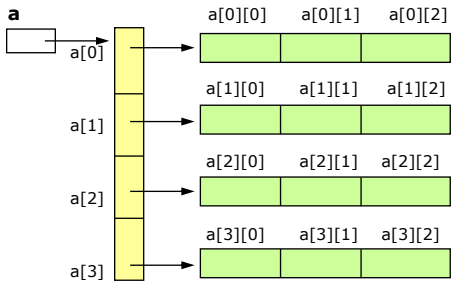
```
int[] days = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
...
int d = days[ month];
```

Mehrdimensionale Arrays

Zweidimensionales Array = Matrix



In Java als Array von Arrays implementiert



Deklaration und Erzeugung

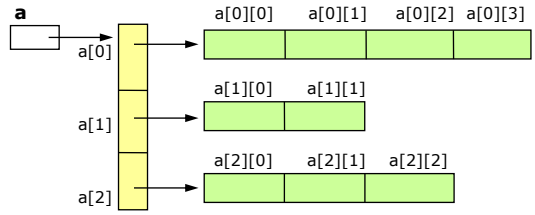
```
int[][] a;
a = new int[ 4][ 3];
```

Zugriff

```
a[ i][ j] = a[ i][ j+ 1];
```

Mehrdimensionale Arrays

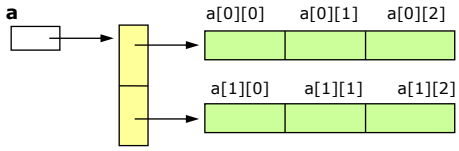
Zeilen können unterschiedlich lang sein (das ist aber selten sinnvoll)



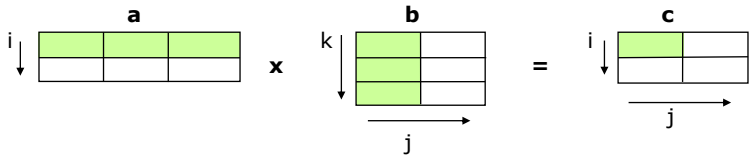

```
int[][] a = new int[ 3][ ];
a[ 0] = new int[ 4];
a[ 1] = new int[ 2];
a[ 2] = new int[ 3];
```

Initialisierung

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



Beispiel: Matrixmultiplikation




```
static float[][] matrixMult (float[][] a, float[][] b) {
    float[][] c = new float[ a. length][ b[ 0]. length];
    for (int i = 0; i < a. length; i++)
        for (int j = 0; j < b[ 0]. length; j++) {
            float sum = 0;
            for (int k = 0; k < b. length; k++)
                sum += a[ i][ k] * b[ k][ j];
            c[ i][ j] = sum;
        }
    return c;
}
```

Iterator-Form der for-Anweisung *Java5*

In einer Schleife sollen nur bestimmte Interger-werte aus einem Array verwendet werden.

```
 ……
int[] primes ={2,3,5,7,11,13,17}
for(int i=0;i<primes.length;i++)   System.out.println( primes[i] );
……
```

Seit Java 1.5
gibt es hierzu eine spezielle äquivalente Form der for-Anweisung.

```
 ……
int[] primes ={2,3,5,7,11,13,17}
for( int p: primes )   System.out.println( p );
……
```

Methoden mit variabler Parameterzahl *Java5*

```
static int sum( int[] values ){
        int result =0;
        for (int i =0; i < values.length; i++) result += values[i];
        return result;
       }
```

```
int res = sum( new int[] {1,2,5,9})    // Anwenden der Methode sum
```

Seit Java 1.5 gibt es hierzu eine äquivalente Form.

```
static int sum2( int... values ){
        int result =0;
        for (int i =0; i < values.length; i++) result += values[i];
        return result;
       }
```

```
int res = sum2(1,2,5,9)                // Anwenden der Methode sum2
```

Wird vom Compiler automatisch in obige Form umgewandelt!