

ESOP - Gleitkommazahlen, Methoden und Arrays

Assoc. Prof. Dr. Mathias Lux
ITEC / AAU

Wiederholung ..



```
/**
 * Check for primes, simple version ...
 */
public class Primes {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime; candidate++) {
            boolean isPrime = true;
            // iterate potential dividers
            for (int divider = 2; divider < candidate; divider++) {
                // check for division without rest
                if (candidate % divider == 0) {
                    isPrime = false;
                }
            }
            if (isPrime)
                System.out.println("prime = " + candidate);
        }
    }
}
```

- Finde Primzahlen < maxPrime

Gleitkommazahlen



- Zwei Datentypen
 - float ... 32 Bit Genauigkeit (24/8 in Java 8)
 - double ... 64 bit Genauigkeit (53/11 in Java 8)

- Syntax

`FloatConstant = [Digits] "." [Digits] [Exponent]
[FloatSuffix].`

`Digits = Digit {Digit}.`

`Exponent = ("e" | "E") ["+" | "-"] Digits.`

`FloatSuffix = "f" | "F" | "d" | "D".`

Gleitkommazahlen



- **Variablen**
 - float x, y;
 - double z;
- **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

Harmonische Reihe



```
public class HarmonicSequence {  
    public static void main (String[] arg) {  
        float sum = 0;  
        int n = 10;  
        for (int i = n; i > 0; i--)  
            sum += 1.0f / i;  
        System.out.println("sum = " + sum);  
    }  
}
```

- Was würden statt $1.0f / i$ folgende Ausdrücke liefern?
 - $1 / i$... 0 (weil ganzzahlige Division)
 - $1.0 / i$... einen double-Wert

Float vs. Double



```
public class HarmonicSequence {  
    public static void main (String[] arg) {  
        float sum = 0;  
        int n = 10;  
        for (int i = n; i > 0; i--)  
            sum += 1.0f / i;  
        System.out.println("sum = " + sum);  
    }  
}
```

D:\Java\JDK\jdk1.6.0_45\bin\java ...
sum = 2.9289684

Process finished with exit code 0

```
public class HarmonicSequence {  
    public static void main (String[] arg) {  
        double sum = 0;  
        int n = 10;  
        for (int i = n; i > 0; i--)  
            sum += 1.0d / i;  
        System.out.println("sum = " + sum);  
    }  
}
```

D:\Java\JDK\jdk1.6.0_45\bin\java ...
sum = 2.9289682539682538

Process finished with exit code 0

Zuweisungen und Operationen



Achtung

- Zuweisungskompatibilität
 - $\text{double} \supseteq \text{float} \supseteq \text{long} \supseteq \text{int} \supseteq \text{short} \supseteq \text{byte}$
- Erlaubte Operationen
 - Arithmetische Operationen (+, -, *, /)
 - Vergleiche (==, !=, <, <=, >, >=)

Achtung! Gleitkommazahlen nicht auf Gleichheit prüfen!

Zuweisungen und Casts



```
float f; int i;  
f = i;           // erlaubt  
i = f;           // verboten  
i = (int) f;     // erlaubt: schneidet Nachkommastellen ab;  
                 // falls zu groß oder zu klein:  
                 // Integer.MAX_VALUE, Integer.MIN_VALUE  
f = 1.0;         // verboten, weil 1.0 vom Typ double ist
```

Review: Datentypen



- Ganzzahlige Typen: `byte`, `char`, `short`, `int`, `long`
- Gleitkommazahlen: `float`, `double`
- Zeichenketten: `String`
- Boolesche Variablen: `boolean`

See also <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Review: Datentypen



- Ganzzahlige Ausdrücke werden zu `int`
- Kommazahlen und „e“-Format werden zu `double`
- Erzwungener Typ mit Suffix
 - „L“ oder „l“ -> `long`
 - „d“ -> `double`
 - „f“ -> `float`

Review: Datentypen



Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Review: Datentypen



- Operatoren
 - Unäre Operatoren + , – bzw. !
 - Binäre Operatoren
 - Achtung bei String und „+“

Beispiele: Datentypen



- IDEA - rote Unterstreichung usw.
- Suffix für erzwungenen Typ

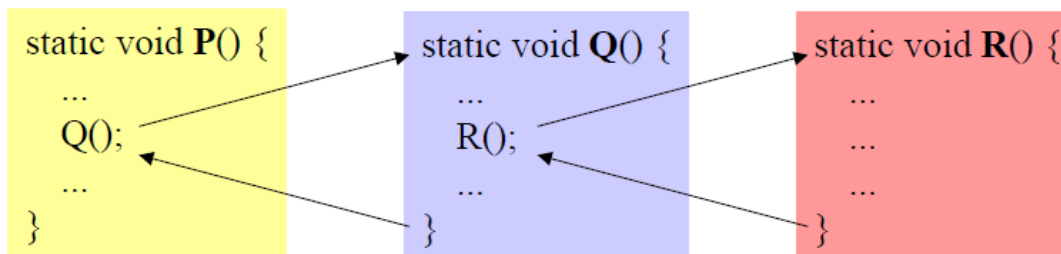


- Vgl. prozedurale & funktionale Sprachen
 - Unterprogramme, Funktionen, ...
- Ziel ist Code wiederzuverwenden
 - Oft genutzte Funktionen / Operationen
- Weniger Zeilen Code
 - weniger Arbeit, weniger Fehler
 - leichter zu warten

Methoden in Java



- Wir betrachten erst Spezialfall von Methoden
 - .. und nutzen sie als Unterprogramme
- Namenskonventionen für Methoden
 - Beginnen mit Verb und Kleinbuchstaben
 - Beispiele:
 - printHeader, findMaximum, traverseList, ...



Methoden in Java



```
public class SubroutineExample {  
    private static void printRule() {           // Methodenkopf  
        System.out.println("-----");        // Methodenrumpf  
    }  
  
    public static void main(String[] args) {  
        printRule();                           // Aufruf  
        System.out.println("Header 1");  
        printRule();  
    }  
  
}
```

D:\Java\JDK\jdk1.6.0_45\bin\java ...

Header 1

Process finished with exit code 0

Parameter



- Es können Werte an die Methode übergeben werden.

```
class Sample {  
  
    static void printMax (int x, int y) {  
        if (x > y) Out.print(x); else Out.print(y);  
    }  
  
    public static void main (String[] arg) {  
        ...  
        printMax(100, 2 * i);  
    }  
}
```

formale Parameter

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

aktuelle Parameter

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

Parameter



- Aktuelle Parameter werden den entsprechenden formalen Parametern zugewiesen
- $x = 100; y = 2 * i;$
 - aktuelle Parameter müssen mit formalen zuweisungskompatibel sein

```
class Sample {  
  
    static void printMax (int x, int y) {  
        if (x > y) Out.print(x); else Out.print(y);  
    }  
  
    public static void main (String[] arg) {  
        ...  
        printMax(100, 2 * i);  
    }  
}
```

Funktionen



- Funktionen sind Methoden, die einen Ergebniswert an den Aufrufenden zurückliefern

```
class Sample {  
    static int max (int x, int y) {  
        if (x > y) return x; else return y;  
    }  
  
    public static void main (String[] arg) {  
        ...  
        int result = 3 * max(100, i + j) + 1;  
        ...  
    }  
}
```

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

Funktionen vs. Prozeduren



- Funktionen
 - Methoden mit Rückgabewert
 - static `int` max (int x, int y) {...}
- Prozeduren
 - Methoden ohne Rückgabewert
 - static `void` printMax (int x, int y) {...}

Beispiel



```
public class BinomialCoefficient {  
    public static void main(String[] args) {  
        int n = 5, k = 3;  
        int result = factorial(n) /  
            (factorial(k) * factorial(n - k));  
        System.out.println("result = " + result);  
    }  
  
    public static int factorial(int k) {  
        int result = 1;  
        for (int i = 2; i <= k; i++) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

$$\binom{n}{k} = \frac{n!}{k! \cdot (n - k)!}.$$

Return & Rekursion



```
public class BinomialCoefficient {
    static int n = 5, k = 3;

    public static void main(String[] args) {
        int result = factorial(n) /
            (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        if (k>1) {
            return factorial(k-1)*k;
        }
        else {
            return 1;
        }
    }
}
```

- Return leitet Rücksprung ein
- Kann an beliebiger Stelle in der Methode stehen
- Methode, die sich selbst aufruft -> direkte Rekursion

Primzahlen



```
/**
 * Primes based on function.
 */
public class PrimesWithMethod {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime;
            candidate++) {
            if (isPrime(candidate))
                System.out.println("prime = " + candidate);
        }
    }
}
```

```
public static boolean isPrime(int candidate) {
    boolean isPrime = true;
    // iterate potential dividers
    for (int divider = 2; divider < candidate; divider++) {
        // check for division without rest
        if (candidate % divider == 0) {
            isPrime = false;
        }
    }
    return isPrime;
}
```

```
/**
 * Check for primes, simple version ...
 */
public class Primes {
    public static void main(String[] args) {
        int maxPrime = 1000;
        // iterate candidates
        for (int candidate = 3; candidate <= maxPrime;
            candidate++) {
            boolean isPrime = true;
            // iterate potential dividers
            for (int divider = 2; divider < candidate; divider++) {
                // check for division without rest
                if (candidate % divider == 0) {
                    isPrime = false;
                }
            }
            if (isPrime)
                System.out.println("prime = " + candidate);
        }
    }
}
```

Gültigkeitsbereiche (Scope) von Variablen



- Innerhalb eines Blocks gültig
 - `{ ... }`,
 - `for (int i; ...) {...}`
- Außerhalb ist Variable nicht bekannt!

Beispiel



```
public class BinomialCoefficient {  
    public static void main(String[] args) {  
        int n = 5, k = 3;   
        int result = factorial(n) /  
            (factorial(k) * factorial(n - k));  
        System.out.println("result = " + result);  
    }  
  
    public static int factorial(int k) {  
        int result = 1;  
        for (int i = 2; i <= k; i++) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

Unterschiedliche
Variablen mit
unterschiedlichen
Gültigkeitsbereichen

Beispiel: Scope



```
public class BinomialCoefficient {  
    static int n = 5, k = 3;   
  
    public static void main(String[] args) {  
        int result = factorial(n) /  
            (factorial(k) * factorial(n - k));  
        System.out.println("result = " + result);  
    }  
  
    public static int factorial(int k) {  
        int result = 1;  
        for (int i = 2; i <= k; i++) {  
            result *= i;  
        }  
        return result;  
    }  
}
```

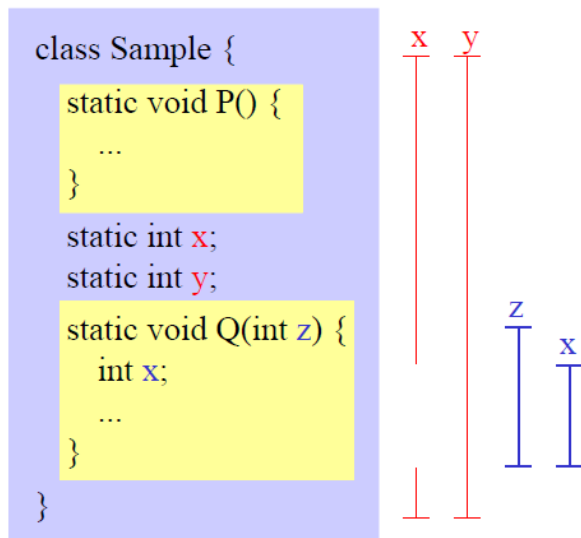
Kleinster
Gültigkeitsbereich
entscheidend!

Sichtbarkeitsbereich von Namen: Lokale Variablen



Regeln

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



Lokale & statische Variablen

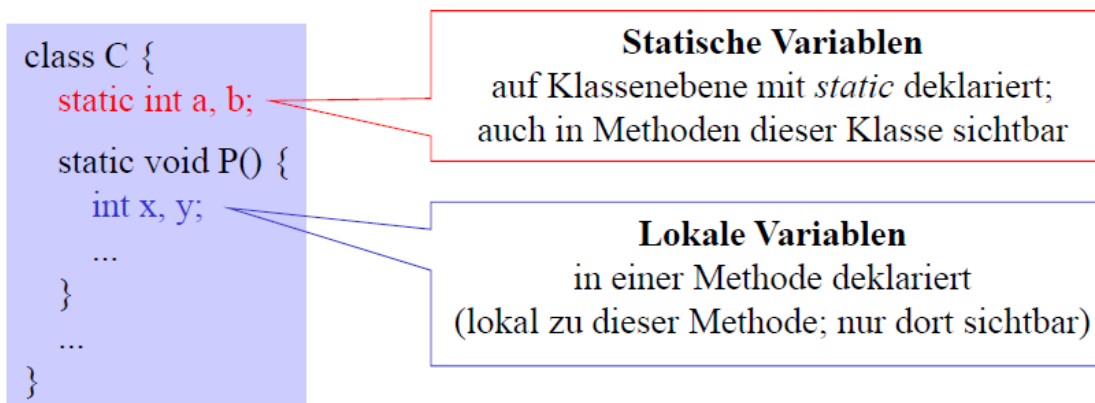


Statische Variablen

- Am Programmbeginn angelegt
- Zu Programmende wieder freigegeben

Lokale Variablen

- Bei jedem Aufruf der Methode angelegt
- Am Ende der Methode wieder freigegeben





Best Practice: 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);
```

Varargs



- In Java können Methoden mit beliebiger Anzahl an Parametern definiert werden.

```
public class VarargExample {  
    public static void main(String[] args) {  
        printList("one", "two", "three");  
    }  
  
    public static void printList(String... list) {  
        System.out.println("list[0] = " + list[0]);  
        System.out.println("list[1] = " + list[1]);  
        System.out.println("list[2] = " + list[2]);  
    }  
}
```



Beispiel: Name Generator mit Prozeduren



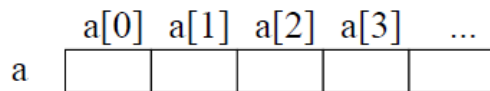
- Vokale vs. Konsonanten

Arrays



- Zusammenfassung Daten gleichen Typs
- Arrays haben fixe Länge
 - Bei Erzeugung festgelegt
- Array Variablen sind Referenz-Variablen
 - In Java! Vgl. int, float, etc. -> Basistypen
- Zugriff erfolgt über Index
 - Erstes Element hat Indexzahl 0.

Eindimensionale Arrays



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

Deklaration

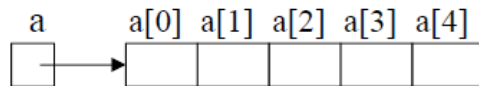
```
int[] a;  
float[] b;
```

- 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 in Java Zeiger auf Arrays!
(Zeiger = Speicheradresse)

Zugriff auf Arrays



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

```
a[3] = 0;  
a[2*i+1] = a[i] * 3;
```

- *length* ist ein Standardoperator
- Liefert Anzahl der Elemente

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

Beispiel



```
public class ArrayExample {  
    public static void main(String[] args) {  
        int[] myArray = new int[5];  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        for (int i = 0; i < myArray.length; i++) {  
            myArray[i] = i+1;  
        }  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int i = 0; i < myArray.length; i++) {  
            sum += myArray[i];  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

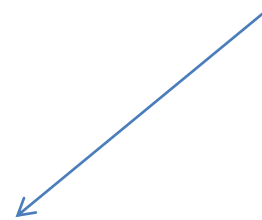
- Berechnet Durchschnitt
- Beachte impliziten Cast auf float!

Beispiel: While, For Each



```
public class ArrayExample {  
    public static void main(String[] args) {  
        int[] myArray = new int[5];  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        int i = 0;  
        while (i < myArray.length) { // while  
            myArray[i] = i+1;  
            i++;  
        }  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int myInt : myArray) { // for each  
            sum += myInt;  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

- Andere Schleifen
- Beachte „for each“



Beispiel: Initialisierung



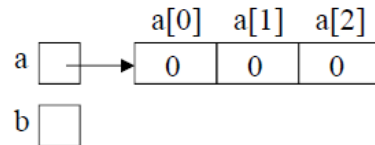
```
public class ArrayExample {  
    public static void main(String[] args) {  
        // initialisiere Werte in Array: {1, 2, 3, 4, 5}  
        int[] myArray = {1, 2, 3, 4, 5};  
        // Berechne Durchschnitt:  
        float sum = 0;  
        for (int myInt : myArray) { // for each  
            sum += myInt;  
        }  
        System.out.println(sum/myArray.length);  
    }  
}
```

- Andere Initialisierung!

Arrayzuweisung

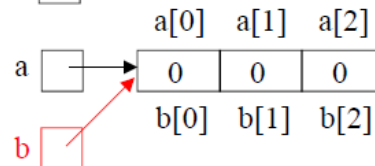


```
int[] a, b;  
a = new int[3];
```



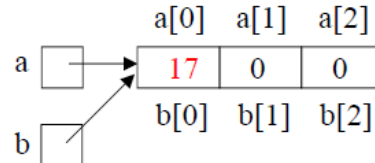
Arrayelemente werden in Java standardmäßig mit 0 initialisiert

```
b = a;
```



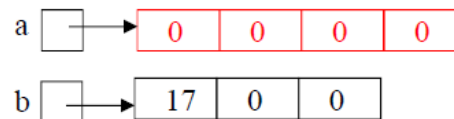
`b` bekommt denselben Wert wie `a`.
Arrayzuweisung ist in Java **Zeigerzuweisung**!

```
a[0] = 17;
```



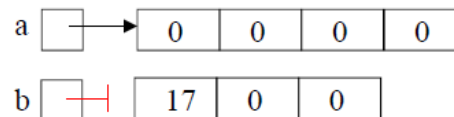
ändert in diesem Fall auch `b[0]`

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



`a` zeigt jetzt auf neues Array.

```
b = null;
```

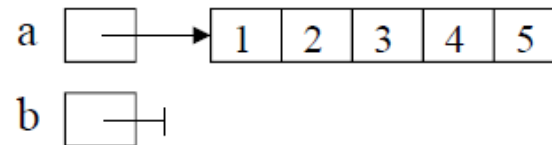


`null`: Spezialwert, der auf kein Objekt zeigt;
kann jeder Arrayvariablen zugewiesen werden

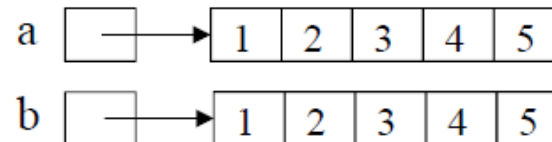
Kopieren von Arrays



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



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



- Typumwandlung nötig, da `a.clone()` Typ `Object[]` liefert

Wiederholung: Datentypen



- Zwei Gleitkommazahlen-Datentypen
 - float ... 32 Bit Genauigkeit (24/8 in Java 8)
 - double ... 64 bit Genauigkeit (53/11 in Java 8)

- Ganzzahlige Typen:

byte	8 bit	$-2^7 \dots 2^7-1$	(-128 .. 127)
short	16 bit	$-2^{15} \dots 2^{15}-1$	(-32.768 .. 32.767)
int	32 bit	$-2^{31} \dots 2^{31}-1$	(-2.147.483.648 ..)
long	64 bit

- Andere: boolean, char und String

Wiederholung: Bedingungen



```
if (<expr>) {  
    // do something  
}  
else {  
    // do something else  
}
```

Wiederholung: Schleifen



- Abbruchbedingung am Beginn
 - `while (<expr>) { ... }`
- Abbruchbedingung am Ende
 - `do { ... } while (<expr>);`
- Zählschleife
 - `for (int i=0; i < 10; i++) { ... }`

Wiederholung: Methoden



Unterprogramme, die beliebig oft aufgerufen werden können

- Prozeduren
 - Haben keinen Rückgabewert
- Funktionen
 - Haben Rückgabewert

```
public static void doSomething(int x) {...} // Prozedur  
public static int doSomething(int x) {...} // Funktion
```

Wiederholung: Gültigkeitsbereich



- Variablen sind nur innerhalb definierter Bereiche gültig
 - Block { ... }
 - Methode
 - Klasse

Wiederholung: Arrays

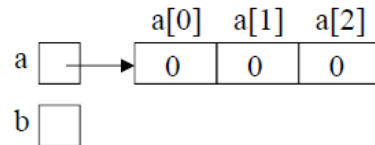


- Arrays sind Zusammenfassung von Werten desselben Typs in einer Variablen
- Index beginnt bei 0
- Arrays sind Referenzvariablen!

Arrayzuweisung

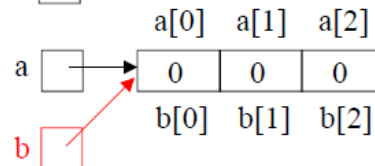


```
int[] a, b;  
a = new int[3];
```



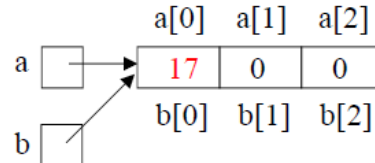
Arrayelemente werden in Java standardmäßig mit 0 initialisiert

```
b = a;
```



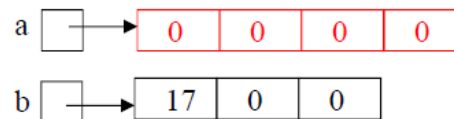
`b` bekommt denselben Wert wie `a`.
Arrayzuweisung ist in Java **Zeigerzuweisung**!

```
a[0] = 17;
```



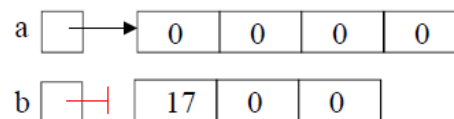
ändert in diesem Fall auch `b[0]`

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



`a` zeigt jetzt auf neues Array.

```
b = null;
```



`null`: Spezialwert, der auf kein Objekt zeigt;
kann jeder Arrayvariablen zugewiesen werden

Beispiel: Namensgenerator mit Arrays



Kommandozeilenparameter



- Programmaufruf mit Parametern
 - `java Programmname par1 par2 par3 ...`
- Parameter als String-Array
 - `main(String[] args)` Methode des Programms

Kommandozeilenparameter



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

```
$> java ArrayExample one two three  
one  
two  
three
```

Beispiel: Sequentielle Suche



```
public class LinearSearch {  
    public static void main(String[] args) {  
        int[] myArray = {12, 2, 32, 74, 26, 42, 53, 22};  
        int query = 22;  
        for (int i = 0; i < myArray.length; i++) {  
            if (query == myArray[i]) {  
                System.out.println("Found at position " + i);  
            }  
        }  
    }  
}
```

- Jedes Element wird untersucht
-> sequentiell
- Braucht n Schritte - Wie groß ist n ?

Beispiel: Sortierung



- Wie sortiert man ein Array a ?
- Einfacher Ansatz:
 1. Erzeuge ein gleich großes Array b
 2. Verschiebe Minimum von a nach b
 3. Falls a nicht leer gehe zu Schritt 2.

Beispiel: Sortierung



```
public class SortExample {
    public static void main(String[] args) {
        // o.b.d.A. a[k] > 0 & a[k] < 100
        int[] a = {12, 2, 32, 74, 26, 42, 53, 22};
        // create result array
        int[] b = new int[a.length];
        for (int i = 0; i < b.length; i++) { // set each item of b
            int minimum = 100;
            int pos = 0;
            for (int j = 0; j < a.length; j++) { // find minimum
                if (a[j] < minimum) {
                    minimum = a[j];
                    pos = j;
                }
            }
            b[i] = minimum;
            a[pos] = 100; // set visited.
        }

        for (int i = 0; i < b.length; i++) {
            System.out.print(b[i] + ", ");
        }
    }
}
```

- Lösbar auf viele Arten
- Vgl. AlgoDat!

Beispiel: Sieb des Eratosthenes



```
public class Sieve {
    public static void main(String[] args) {
        int maxPrime = 200 000;
        boolean[] sieve = new boolean[maxPrime];
        // init array
        for (int i = 0; i < sieve.length; i++) {
            sieve[i] = true;
        }

        // mark the non-primes
        for (int i = 2; i < Math.sqrt(sieve.length); i++) {
            if (sieve[i] == true) { // if it is a prime
                int k = 2;
                while (k*i < sieve.length) {
                    sieve[k*i] = false;
                    k++;
                }
            }
        }

        // print results
        for (int i = 2; i < sieve.length; i++) {
            if (sieve[i]) System.out.println(i);
        }
    }
}
```

