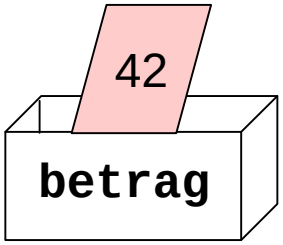
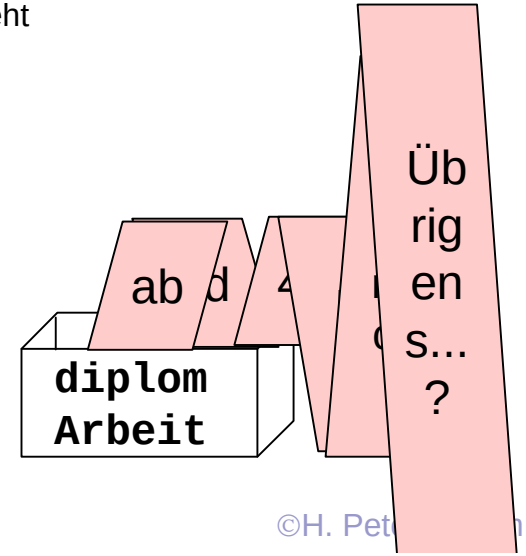
Objekttypen

Referenzen, Objekte, Gleichheit, Wrapper, Arrays, mehr-dimensionale Arrays, Bildbearbeitung, krumme Arrays

Primitive- und Objekt-Datentypen

- Primitive Datentypen benötigen einen vorher genau bekannten Speicherplatz
  - int : 4 Byte, double: 8 Byte, char: 2 Byte
- Manche Objekte können unvorhersehbar viel Speicherplatz verbrauchen
  - String: ?
    - Hängt von der Länge des Textes ab.
      - `String jaNeinAntwort;`
        `String meineDiplomArbeit;`
  - KontenListe: ?
    - Hängt davon ab, wie gut das Geschäft geht

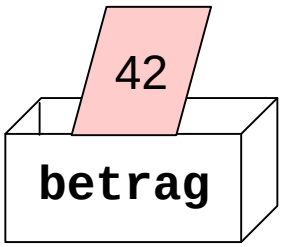




Objekte sind Referenztypen

- In einer Objektvariable wird nur ein Link (*reference*) auf das wirkliche Objekt gespeichert
  - `String diplomArbeit;`
- Die Java Maschine kümmert sich um Platz für die Daten des Objekts.
  - Dazu gehört
    - Besorgung von zusätzlichem Platz bei Bedarf
    - Recyclen von nicht mehr benötigtem Platz
      - Dies heißt: Garbage Collection

- Der Programmierer hat stets einen Link auf das Objekt zur Verfügung



Referenzen

- Beim Zugriff auf Objektfelder folgt Java automatisch dem Link
  - Bei manchen Sprachen muss der Programmierer dies tun
  - In Pascal zum Beispiel:
    `meinKonto^.kontoStand`

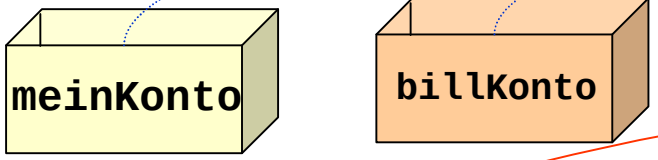
In Java:

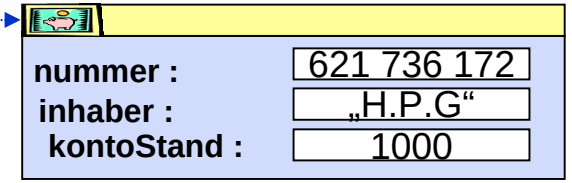
```
meinKonto.kontoStand = billKonto.kontoStand+1 ;
```

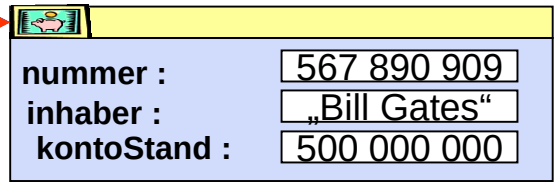
Zuweisung von Objekten

```
meinKonto = billKonto ;
```

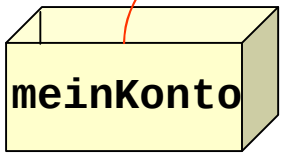
Vorher:

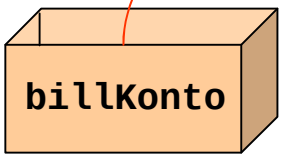






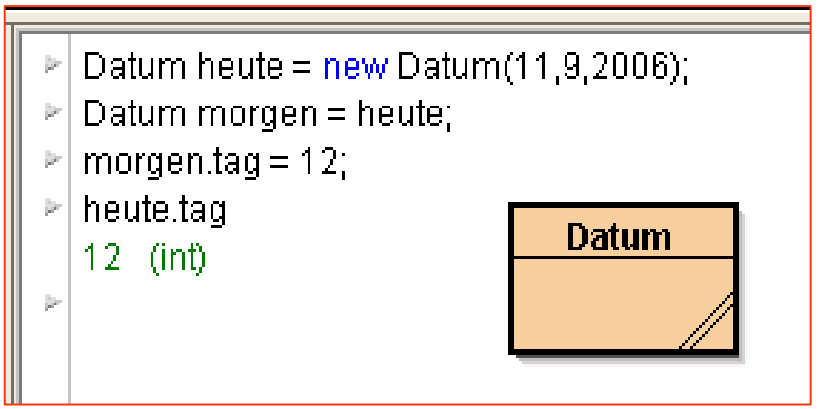
Nachher:





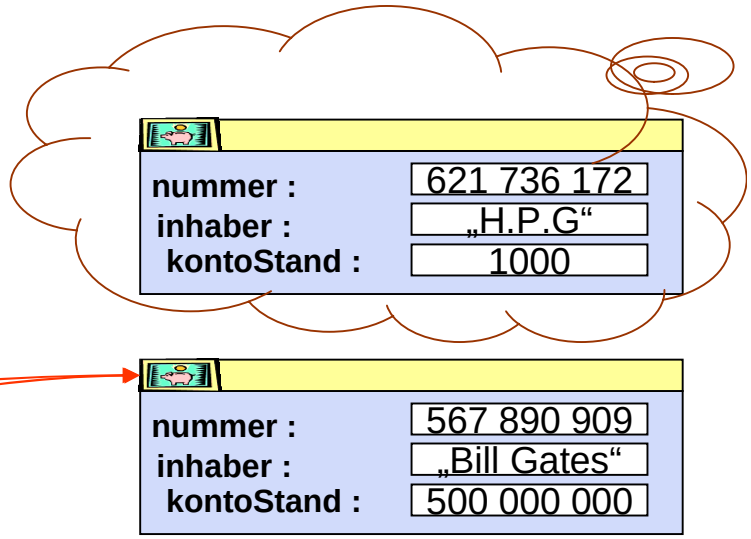
- Bei Zuweisungen ganzer Objekte werden nur die Referenzen übernommen

anderes Beispiel:

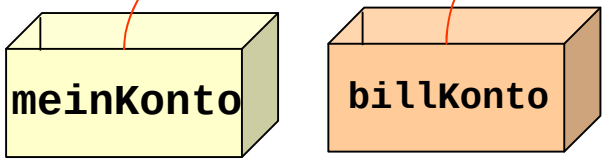


Datenmüll

- Nach der Zuweisung ist ein Objekt unerreichbar geworden
  - Kein Link zeigt mehr auf es
  - Es ist Müll (engl.: garbage)



Nachher:



Java sorgt automatisch für das Recycling des Datenmülls:

Garbage Collection

Beeinflussung, Manipulation

- Wenn zwei Referenzen auf das gleiche Objekt zeigen:
  - Jedes kann die Felder des anderen beeinflussen
  - Ein Name ist dann ein sog. *Alias* für den anderen

```
billKonto.kontoStand = 500000000;

meinKonto.abheben(600000000) ;
   billKonto.getKontoStand()
```

- 100 000 000

Gleichheit

- Vergleich zweier Objekte mit **==** bzw. **!=** prüft nur die Verweise
  - Möglich, dass alle Felder der Objekte übereinstimmen, Vergleich liefert trotzdem **false**.

- Für den inhaltlichen Vergleich ist eine Methode **equals** vorgesehen
  - **equals** mit der Signatur public boolean equals(Object z) wird von **Object** geerbt und ist daher für alle Objekte vorhanden

  - Es sollte für die eigene Klasse geeignet redefiniert werden, z.B.:

```
public boolean equals(Object z){
    return  nummer==(Konto)z.nummer
         && kontoStand == (Konto)z.kontoStand
         && inhaber.equals((Konto)k.inhaber ); }
```

  - **meinKonto.equals(billKonto)**

- **meinKonto == billKonto** ⇒ **false**
- **meinKonto.equals(billKonto)** ⇒ **true**

Wrapper Klassen

- Basis-Datentypen von Java sind **keine** Klassen
  - boolean, char, short, byte,
    int, long, float, double
- Keine Objekte im Sinne des OO-Programmierens
  - Vorteil:
    - direkter Zugriff - ohne Referenz
    - keine explizite Erzeugung (**new**) notwendig
  - Nachteil
    - sind keine Objekte
    - Viele Behälter-Strukturen (**`ArrayLists`**) etc.
      können nur Objekte aufnehmen
- Ausweg
  - Für jede Basisklasse gibt es eine entsprechende *Wrapper-Klasse*
    Boolean, Character, Short, Byte,
    Integer, Long, Float, Double

Wir packen den Typ ein

- Wrapper Klassen verpacken (engl.: to wrap) den Wert des Basistyps

- Eine final deklarierte Klasse kann keine Unterklasse haben
- Ein Objekt der Wrapper-Klasse hat genau ein Feld des entsprechenden Datentyps.Dieses ist unveränderlich final, also konstant
- Hier könenn wir es auspacken
- Jede Wrapper-Klasse ist ähnlich aufgebaut

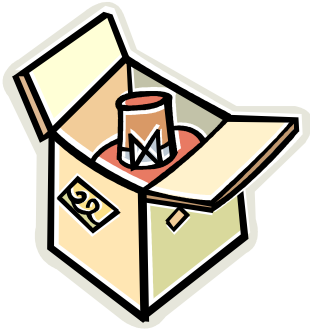
```
public final class Integer {
   // Konstantes Feld des Basistyps
   private final int wert;

   /** Konstruktor */
   Integer(int wert) {this.wert=wert;}

   /** auspacken */
   public int intValue(){ return wert;}

   // Nützliches ...
   public static int parseInt(String lit){
       // ... hier fehlt noch Code ...
   }
}
```

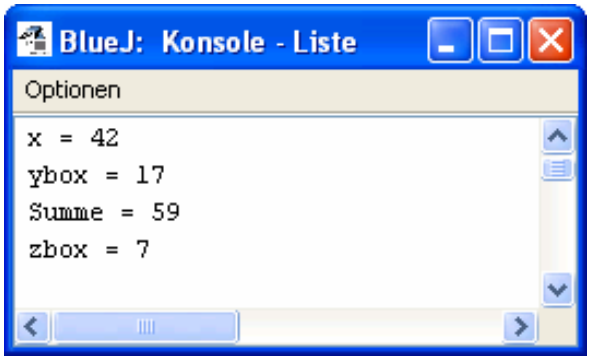
Autoboxing



```
static void test(){
    int x=42;
  // Eigentliche Konstruktion des Wrapper Typs
    Integer ybox = new Integer(17);
  // Autoboxing:
    Integer zbox = 3+4;
  // Auspacken, einpacken, auspacken ...
    int u = new Integer(zbox);
  // Verknüpfen
    System.out.println(
        "x = "+x+"\n"
        +"ybox = " + ybox + "\n"
   // Verknüpfung von Basistyp und Wrappertyp
        +"Summe = "+(x+ybox)+"\n"
        + "zbox = " + zbox);
}
```

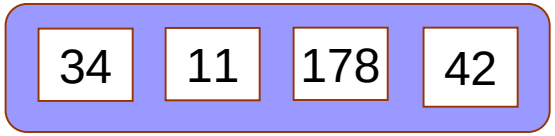
Seit Java 1.5:

- Autoboxing
  - Automatische Umwandlung zwischen Basisklassen und Wrapperklassen



Arrays

- Arrays sind Objekte, die Folgen gleichartiger Variablen enthalten
  - Mathematisch: $x_0, x_1, x_2, \ldots, x_n$
  - Java: `x[0], x[1], x[2], …, x[n]`



- Deklaration:
  - `int [] x;` — **x** ist Folge von ganzen Zahlen
  - `char[] wort ;` — **wort** ist Folge von Zeichen, also eine Zeichenkette
  - `Konto[] konten;` — **konten** ist eine Folge von Konten

- Deklaration mit Initialisierung

```
int[] x = { -13, 27, 42, 128} ;
char[] wort = {'H','a','l','l','o','  ','W','e','l','t'};
```

- Erzeugung

```
int[] x = new int[10];
Konto[] kunden = new Konto[200];
```

Folge: **x[0], x[1], … , x[9]**
Eine Folge von 200 Konten,
kunden[0], kunden[1], ... , kunden[199]

Arrays und for-Schleifen

- Die Länge eines Array-Objekts steht in dem Feld length

```
int[] myArray = new int[x*x+1];
int länge = myArray.length;
```

- for-Schleifen eignen sich gut um Arrays zu durchlaufen

```
Würfel meinWürfel = new Würfel();
for(int k=0; k<länge; k++){
    meinWürfel.würfele();
    myArray[k] = meinWürfel.getAugenZahl();
    }
```

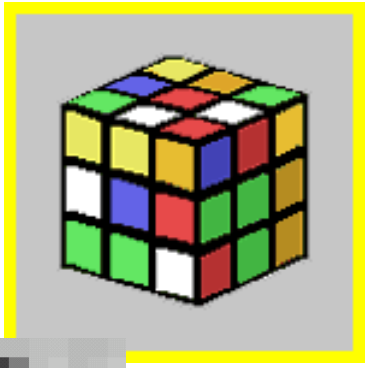
schreiben →

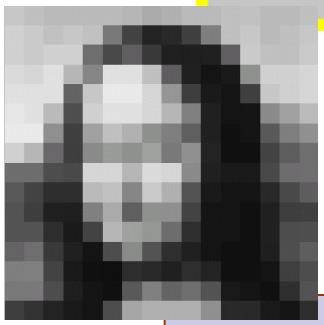
- Typische Suche in einem Array – mit vorzeitigem Verlassen:

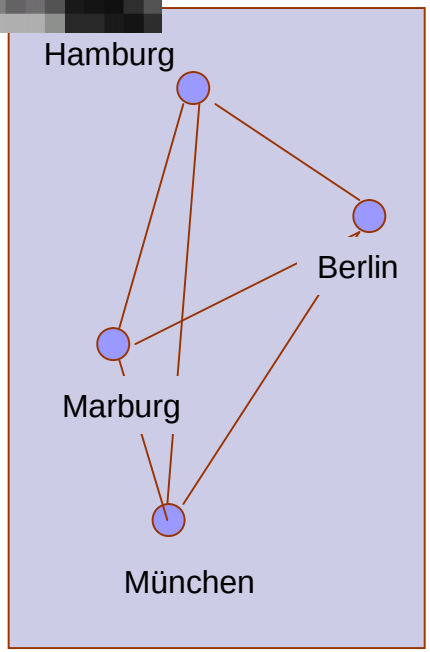
```
for (int k=0; k<länge; k++)
    if (meinArray[k] == 6) {
        gefunden = true; break;
    }
```

lesen →

Mehrdimensionale Arrays

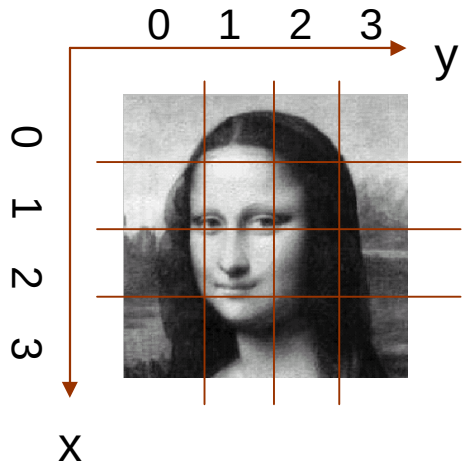






- Matrizen sind mehrdimensionale Arrays
- Man benutzt Matrizen zur Speicherung und Bearbeitung von
  - Bildern
  - Operationstabellen
  - Wetterdaten
  - Graphen
  - Distanztabellen
  - ...
- Deklaration
  - `int [][] greyMonaLisa;`
  - `Color[][][] rubik ;`
- Deklaration mit Erzeugung
  - `Color[][] bildschirm = new Color[1024][748];`
- Deklaration mit Initialisierung
  - `boolean[][] xorTabelle = {{false, true},{true,false}}`
  - ```
    int[][] entfernung = { {  0, 213, 419, 882},
                           {213,   0, 617, 720},
                           {419, 617,   0, 521},
                           {882, 720, 521,   0}};
    ```
    ```

Beispiel: Bildbearbeitung



- Graphik als Matrix von Grauwerten

```
int[][] monaGrey = new int[4][4];
monaGrey = { {21,98,205,23},
             {32,37,126,98},
             {113,47,191,139},
             {107,189,191,96} } ;
```

21	98	205	23
32	37	126	98
113	47	191	139
107	189	191	96

- Aufhellen

```
for (int x = 0; x < höhe; x++)
  for(int y =0; y < breite; y++)
    monaGrey[x][y] = Math.min(monaGrey[x][y]*9/10,255);
```

- Negieren

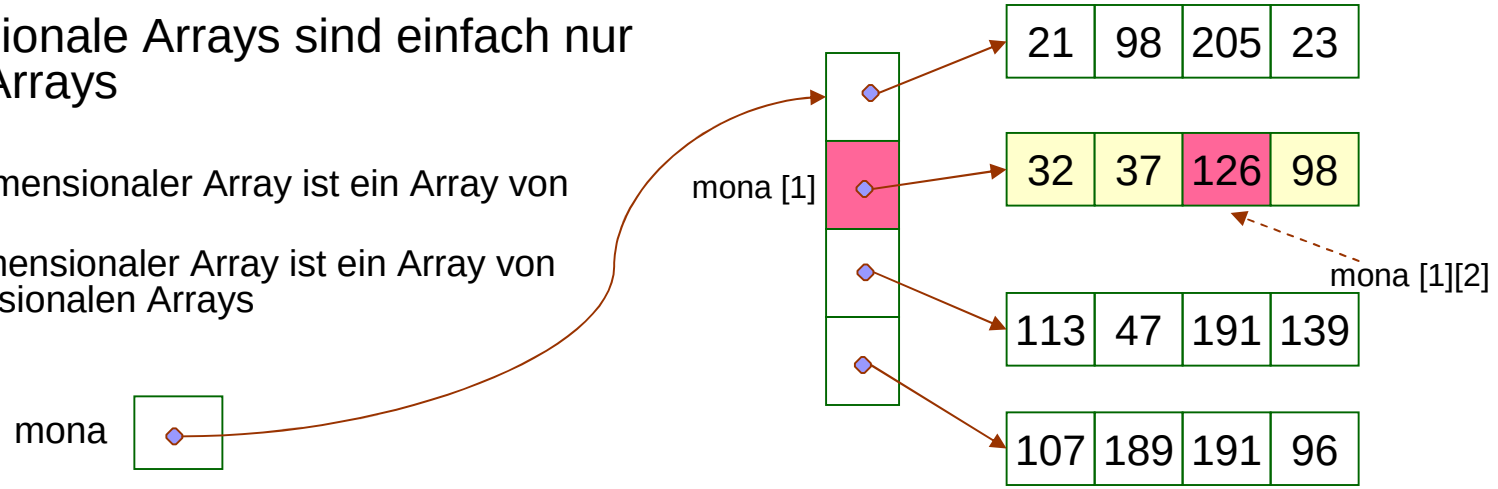
```
for (int x = 0; x < höhe; x++)
  for(int y =0; y < breite; y++)
     monaGrey[x][y] = 255-monaGrey[x][y] ;
```

- Schwarzweiss

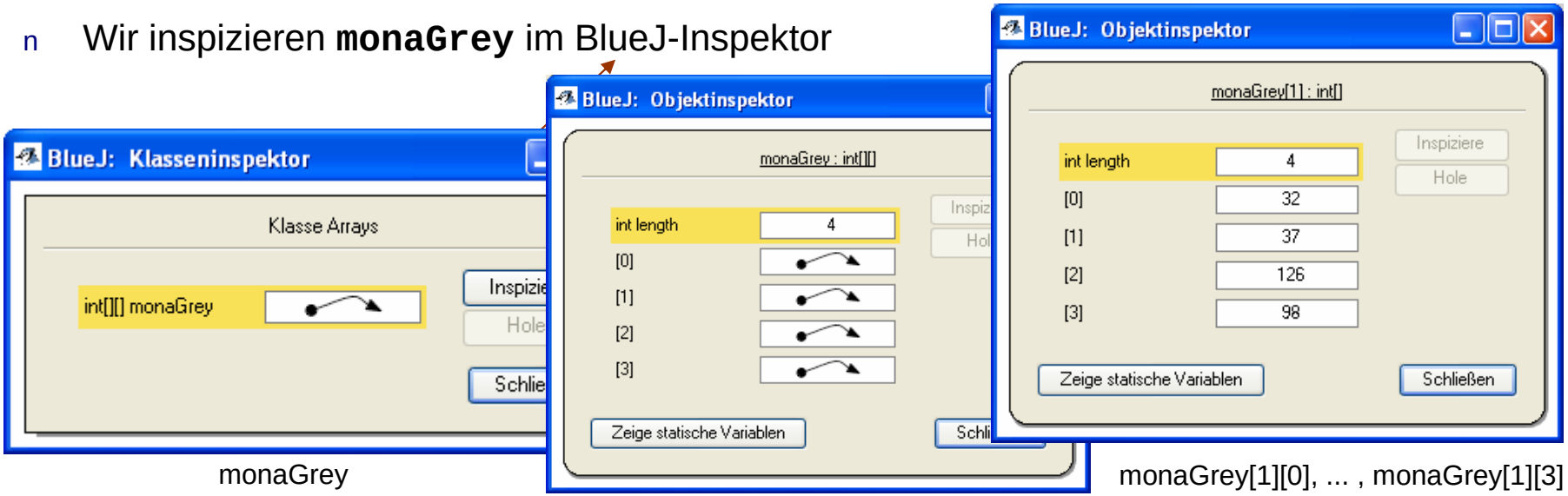
```
for (int x = 0; x < höhe; x++)
  for(int y =0; y < breite; y++)
    if(monaGrey[x][y] < 128) monaGrey[x][y] = 0;
    else monaGrey[x][y] = 255;
```

Mehrdimensionale Arrays – braucht man nicht !

- Mehrdimensionale Arrays sind einfach nur Arrays von Arrays
  - Ein zweidimensionaler Array ist ein Array von Zeilen
  - Ein dreidimensionaler Array ist ein Array von zweidimensionalen Arrays
  - ...



- Wir inspizieren **`monaGrey`** im BlueJ-Inspektor



monaGrey

monaGrey[0], ... , monaGrey[3]

monaGrey[1][0], ... , monaGrey[1][3]

Arrays: Krumm und schief

- Die Dimension eines Arrays ist nicht Teil seines Typs
  - Folglich können verschieden große Arrays gleichen Typ haben

```
int [] alt = new int[3];
int [] neu = new int[17];
alt = neu ;                    // das ist in Java möglich !
```

- Eine Matrix kann verschieden lange Zeilen haben

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

- Wie durchläuft man krumme Arrays ?
  - Die innere for-Schleife muss die Länge der zu durchlaufenden Zeile selber bestimmen
  - Das geht mittels des **`length`**-Feldes

- Durchlauf durch **`schiefArray`**

```
for(int zeile=0; zeile < schiefArray.length; zeile++)
   for(int spalte=0; spalte < schiefArray[zeile].length; spalte++)
     tuWasSinnvollesMit(schiefArray[zeile][spalte]);
```

Arrays kopieren

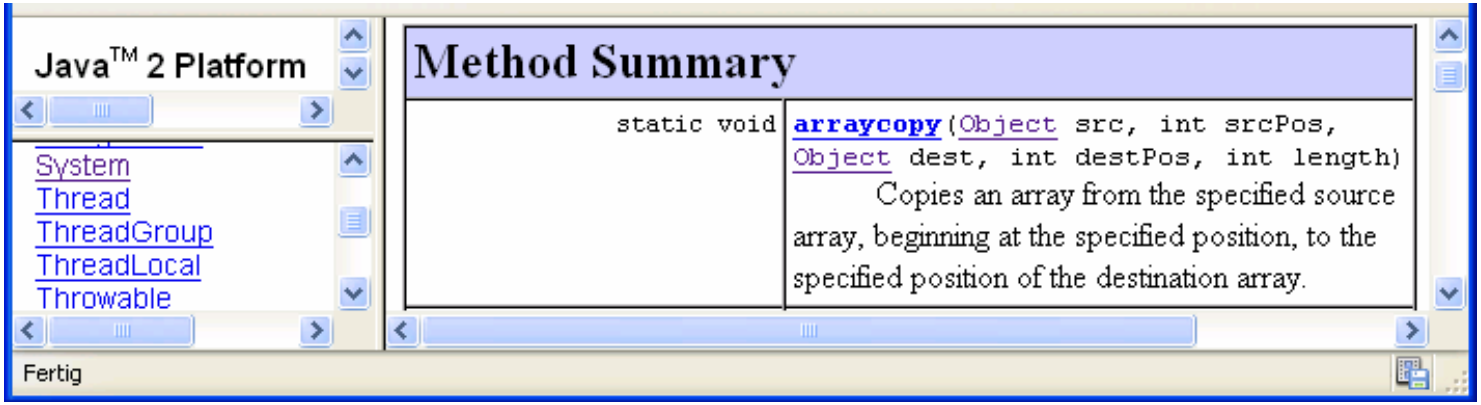
- Arrays sind Objekte
  - Array-Variablen speichern Referenzen
  - Kopieren kopiert Referenzen
    - *shallow copy* (flache Kopie)
  - wie bei jedem Objekt
- Deep Copy :
  - System.arraycopy(…)

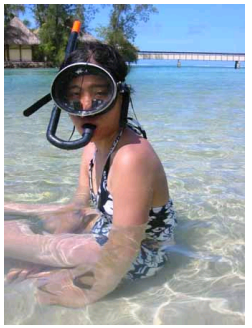
```
char[] latein = { 'a', 'b', 'c' };
char[] griech = latein;
griech[1] = '\u03B2';
latein[1]
β  (char)

char[] deutsch = new char[3];
System.arraycopy(latein,0,deutsch,0,3);
deutsch[0] = 'ä';
latein[0]
a  (char)
```

υυπσ

bässär





shallow