

Programmierung II

Kapitel 1

Vektoren und Matrizen

Literatur

Buch zur Vorlesung

<http://www.springerprofessional.de/978-3-8348-2270-3---java-als-erste-programmiersprache/4893502.html>



Joachim Goll, Cornelia Heinisch

Java als erste Programmiersprache

Ein professioneller Einstieg in die
Objektorientierung mit Java

Verlag: Springer Fachmedien Wiesbaden

1141 Seiten

ISBN: 978-3-8348-2270-3

Neuaufgabe von 2014

Empfehlung



Christian Ullenboom

Java ist auch eine Insel

Das umfassende Handbuch

Verlag: Galileo Press

1294 Seiten

10. aktualisierte Auflage 2011 (Java 7.0)

11. aktualisierte Auflage 2014 (Java 8.0)

Empfehlung



Kathy Sierra, Bert Bates
Java von Kopf bis Fuss
(Behandelt Java 5.0)

O'Reilly Verlag

3. korrigierter Nachdruck (2008)

Überblick Programmierung 2



Software-Engineering

- Testen: Unit-Tests mit JUnit
- Strukturieren: Pakete
- Kommentieren: Javadoc, Annotationen

OO- Programmierung

- Vererbung
- super-Operator
- Polymorphie von Objekten
- Finale Klassen
- Abstrakte Klassen und Schnittstellen
- Wrapper- Klassen
- generische Klassen

Dynamische Datenstrukturen

- Verkettete Listen
- Stack, Queue, Binärbäume

GUI

- GUI-Programmierung mit Swing

Arrays aus Referenzen

Array: Wiederholung

Array (Feld): Objekt bestehend aus Elementen **desselben Datentyps**

Elemente - von einem

- elementaren Datentyp → **eindimensionales** Array
- Referenztyp
 - selbst ein Array → **mehrdimensionales** Array

Array = ein Objekt → zur Laufzeit auf dem Heap angelegt

Länge des Array: Wiederholung

Länge eines Arrays = Anzahl der Elemente

→ Länge immer > 0

mithilfe der Instanzvariable **length** bestimmt

Indizierung der Array-Elemente in Java: **beginnt mit 0**

d.h. Array aus n Elementen (= der Länge n) → Indizes: $0, \dots, n-1$

Arrays aus Referenzen

bisher: Arrays-Elemente – Objekte eines einfachen Datentyps

jetzt: Array-Elemente – **Referenzen** (auf Objekte) eines abstrakten Datentyps

Array anlegen

genauso, wie bei Arrays aus einfachen Datentypen –
in drei Schritten:

- 1. Schritt: **Definition** einer **Referenzvariablen**, die auf das Array-Objekt zeigt

```
Klassenname[] arrayName;
```

- 2. Schritt: **Erzeugen** des Array-Objektes

```
arrayName = new Klassenname[Länge];
```

- 3. Schritt: **Initialisierung** des Arrays (mit **Objekten**)

- a) über Wertezuweisung an jedes Element einzeln

```
arrayName[index] = referenz;
```

- b) Implizites Erzeugen über eine Initialisierungsliste

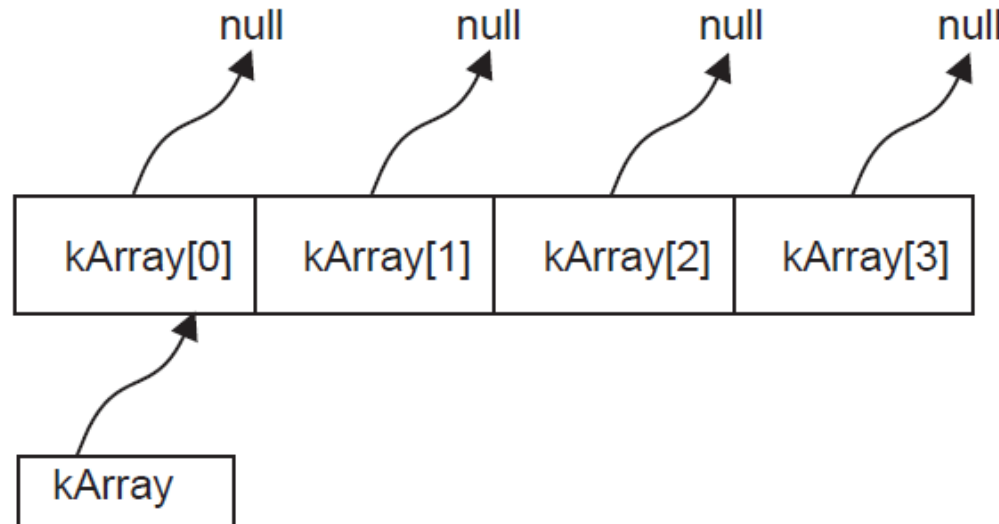
```
Klassenname[] arrayName = {ref1, ref2, ...}
```

Array erzeugen: Beispiel

Beispiel: `Klasse[] kArray = new Klasse[4];`

→ ein neues Array-Objekt aus **Referenzvariablen** vom Typ `Klasse` und der Länge 4 (4 Elemente) erzeugt; jede Referenzvariable mit **null** initialisiert

der (Referenz-)Variable `kArray` – Referenz (Zeiger) auf das Objekt zugewiesen

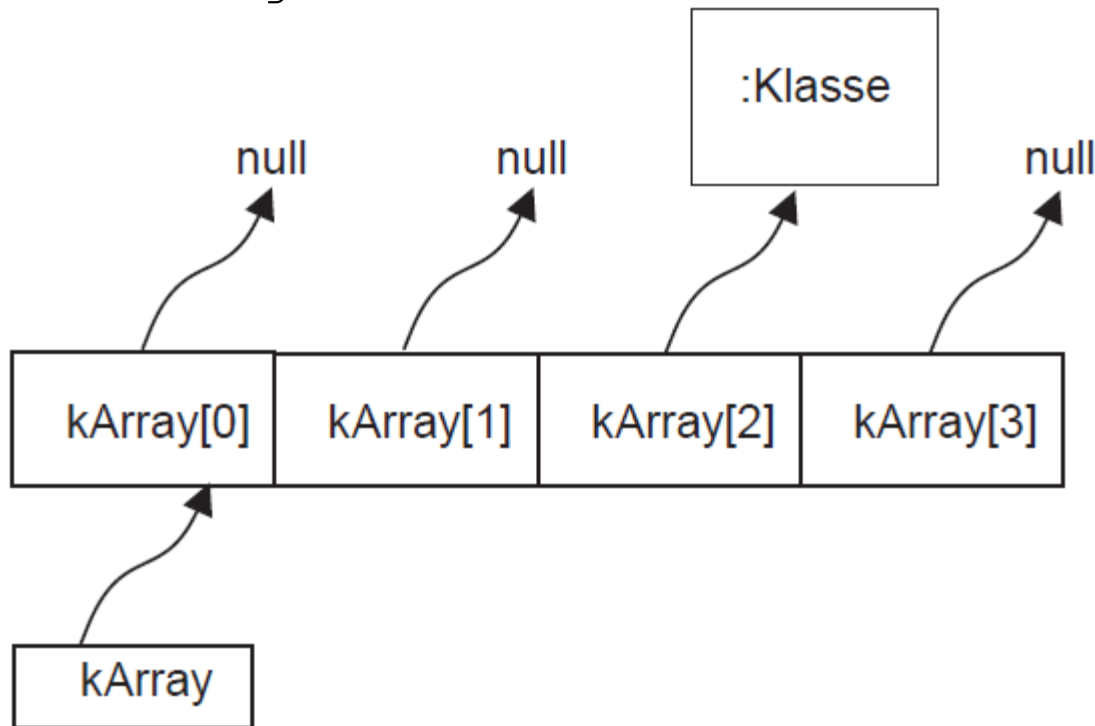


Array initialisieren: Beispiel (1)

a) Wertezuweisung an jedes Element einzeln

Beispiel:

```
Klasse refObj = new Klasse();  
kArray[2] = refObj;
```

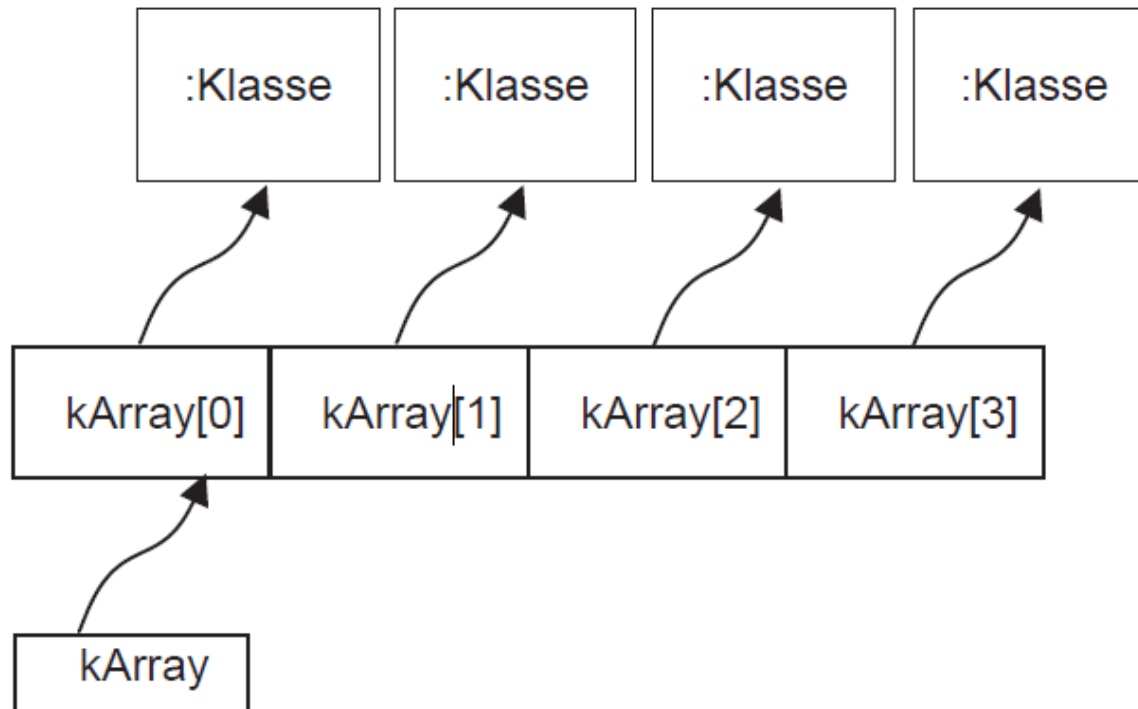


Array initialisieren: Beispiel (2)

a) Wertezuweisung an jedes Element einzeln mit einer Schleife

Beispiel:

```
for (int j = 0; j < kArray.length; j++)  
    kArray[j] = new Klasse();
```



Array initialisieren: Beispiel (3)

b) Implizites Erzeugen über eine Initialisierungsliste

Beispiel:

```
Klasse[] kArray = new Klasse[4];
```

```
Klasse[] kArray = {refK1, refK2, new Klasse(),  
refK3}
```

→ refK1, refK2 und refK3: Referenzen auf **vorhandene** Objekte vom Typ Klasse

Auch möglich: in der Initialisierungsliste ein Objekt eines bestimmten Typs mit Hilfe des new-Operators direkt erzeugen

Beispiel: Punkt-Array

```
public class Punkt {
    private float x;
    private float y;
    public Punkt(float u, float v) { x = u; y = v; }
    public float getX() { return x; }
    public void verschiebe(float vx, float vy) { ... }
    ...
}

public class ArrayPunktTest {
    public static void main (String[] args) {
        Punkt p1 = new Punkt(1.3f, -2);
        Punkt p2 = new Punkt(0, 0);
        Punkt[] arr = { p1, p2, new Punkt(7, 4.5f) };
    }
}
```


for-each-Schleife wiederholt

- über Array-Elemente iterieren

Beispiel: `int[] array = new int [20];`

Array-Elemente ausgeben:

statt

```
for (int i = 0; i < 20; i++) {  
    System.out.println(array[i]);  
}
```

jetzt kürzer

```
for (int elem : array) {  
    System.out.println(elem);  
}
```

Verständnisfragen



- Was ist ein Array?
- Was ist der Unterschied zwischen einem Array aus primitiven und abstrakten Datentypen?
- Geht das:

```
KlasseA[] arr;
```

```
arr = {new KlasseA(), new KlasseA()};
```

- Wieviel Mal wird eine `for-each`-Schleife durchlaufen?

Vektoren

Wiederholung: Objekte

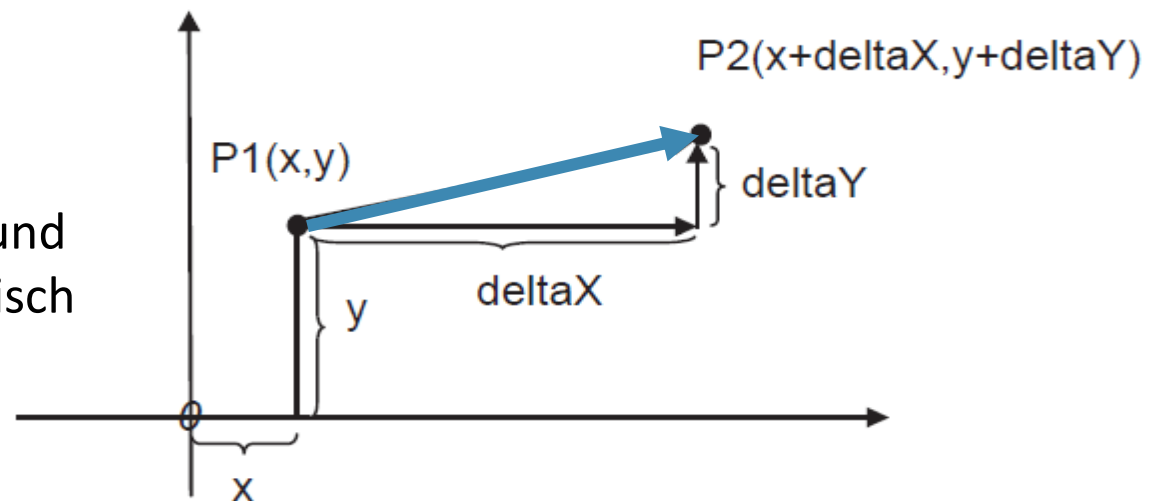
Ebene Vektoren

```
public class Vektor2D {  
    private float delX, delY;  
    private Punkt anker;  
    public Vektor2D(float delX, float delY,  
        Punkt anker) {  
        this.delX = delX;  
        this.delY = delY;  
        this.anker = anker;  
    }  
}
```

Attribute (Datenfelder)

Konstruktor

wenn Namen der
formalen Parameter und
der Datenfelder identisch



Ebene Vektoren erzeugen

```
public class VektorTest {  
    public static void main (String[] args) {
```

Punkt zuerst erzeugen



```
        Punkt p1 = new Punkt(1.3f, 2.0f);  
        Vektor2D v1 = new Vektor2D(3, 5.5f, p1);
```

```
        Vektor2D v2 = new Vektor2D(4, 0, new Punkt(1, 1));
```

```
    }  
}
```



Punkt beim Aufruf des
Vektor2D-Konstruktors
direkt erzeugen

Vektoroperationen

Instanzmethoden
→ Objekt geändert

```
in: public class Vektor2D:
```

```
public void add(Vektor2D vektor) {
```

```
    delX = delX + vektor.delX; direkter Zugriff auf Attribute
```

```
    delY = delY + vektor.delY;
```

```
}
```

Aufruf in main(): `v1.add(v2)` ; wobei v1, v2 2D-Vektoren

```
public float betrag() {
```

```
    return ((float) Math.sqrt(delX*delX + delY*delY));
```

```
}
```

Aufruf in main(): `v1.betrag()` ;

Objekt als formaler Parameter

```
public void verschiebeAnker(Vektor2D schieb) {
```

```
    anker.verschiebe(schieb.delX, schieb.delY);
```

```
}
```

public-Methode der Klasse **Punkt**

Aufruf in main(): `v1.verschiebeAnker(v2)` ;

Vektoroperationen

Klassenmethoden →
Ergebnis zurückgegeben

aus Instanzmethoden → Klassenmethoden machen:

in: public class **Vektor2D**:

statt

```
public void add(Vektor2D vektor) {  
    delX = delX + vektor.delX;  
    delY = delY + vektor.delY;  
}
```

jetzt:

```
public static Vektor2D add1(Vektor2D v1,  
                             Vektor2D v2) {  
    Vektor2D erg = new Vektor2D(v1.delX + v2.delX,  
                                v1.delY + v2.delY, v1.anker);  
    return erg;  
}
```

Aufruf in main(): `Vektor2D.add1 (ve1, ve2) ;`

Instanz- vs. Klassenmethoden

Aufruf: ~~objekt.Klassenmethode()~~

in Java immer möglich...

aber: **schlechter Programmierstil!**



Methodenaufruf:

objekt.instanzmethode()

Klasse.klassenmethode()

Robuste Methode?

```
public static Vektor2D add1(Vektor2D v1,  
                             Vektor2D v2) {  
  
    Vektor2D erg = new Vektor2D(v1.delX + v2.delX,  
                                v1.delY + v2.delY, v1.anker);  
  
    return erg;  
}
```

nicht robust, weil:

v1 == null oder v2 == null → Java wirft
eine NullPointerException →
Programm abgebrochen

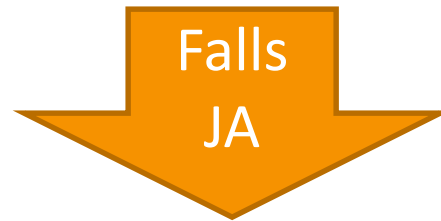
WAS TUN?

robuste Methode: terminiert *normal*
für jede Eingabe und bricht nicht ab

Robust oder nicht robust implementieren?

Die Qual der Wahl...

Soll die gegebene Methode überhaupt robust sein?



einen sinnvollen Dummy-Rückgabewert für den „Fehlerfall“ bestimmen und

- mit `if-else` Fehler „behandeln“ und Dummy-Wert per `return` zurückgeben
- von Java geworfene Exception fangen und Dummy-Wert per `return` zurückgeben

Robuste Implementierung (1)

```
public static Vektor2D add1(Vektor2D v1,  
                             Vektor2D v2) {  
    if(v1 == null || v2 == null)  
        return new Vektor2D(0f, 0f, new Punkt(0,0));  
    else {  
        Vektor2D erg = new Vektor2D(v1.delX + v2.delX,  
                                     v1.delY + v2.delY, v1.anker);  
        return erg;  
    }  
}
```

Robuste Implementierung (2)

```
public static Vektor2D add1(Vektor2D v1,  
                             Vektor2D v2) {  
    try {  
        Vektor2D erg = new Vektor2D(v1.delX + v2.delX,  
                                       v1.delY + v2.delY, v1.anker);  
        return erg;  
    }  
    catch(NullPointerException e) {  
        System.out.println(e.getMessage());  
        return new Vektor2D(0f, 0f, new Punkt(0,0));  
    }  
}
```

Robust oder nicht robust implementieren?

Die Qual der Wahl...

Soll die gegebene Methode überhaupt robust sein?



- im **Kommentar** und/oder im **Methodenkopf** (mit throws) **ankündigen** (unbedingt!), dass Methode (automatisch) eine `NullPointerException` wirft (und die Implementierung *nicht verändern*) → geht in Richtung **vertragsbasierte Programmierung** (design by contract)
- eine passende Exception **selbst werfen** und sie im Kommentar und im Methodenkopf (mit throws) ankündigen → **defensive Programmierung**

Defensive Programmierung



```
public static Vektor2D add1(Vektor2D v1, Vektor2D v2)
    throws IllegalArgumentException {

    if(v1 == null || v2 == null) throw new
        IllegalArgumentException ("Argumente dürfen
                                nicht null sein");

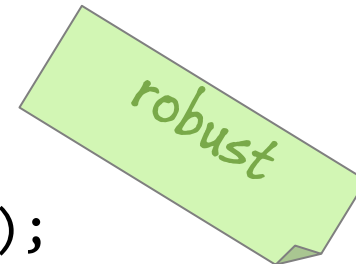
    Vektor2D erg = new Vektor2D(v1.delX + v2.delX,
                                v1.delY + v2.delY, v1.anker);
    return erg;
}
```

defensiv programmierte Methode: terminiert *normal* für alle Eingaben des Definitionsbereichs, für alle anderen Eingaben löst sie eine Ausnahme (Exception) aus.

Aufrufer fängt Exception

```
public static void main (String[] args) {  
    ...  
    Punkt p1 = ...;  
    Vektor2D v1 = null;  
    Vektor2D v2 = new Vektor2D(1.5f,1.5f,p1);  
    try {  
        Vektor2D erg = Vektor2D.add1(v1,v2);  
        System.out.println(erg);  
    }  
    catch (IllegalArgumentException e) {  
        System.out.println(e.getMessage());  
    }  
    ...  
}
```

→ kein Programmabbruch, falls v1 == null




Methode toString()

wiederholt

- toString(): vordefinierte Methode der Klasse *java.lang.Object*
- jede Java-Klasse von *java.lang.Object* abgeleitet → jede Java-Klasse besitzt die Methode toString()

Beispiel:

```
public class VektorTest {  
    public static void main (String[] args) {  
        Vektor2D v2 = new Vektor2D(4,0,new Punkt(1,1));  
        System.out.println(v2.toString());  
        System.out.println(v2);  
    }  
}
```



Methode toString()
aufgerufen

Programmausgabe: Vektor2D@3c291fc2

Wozu toString()

- toString() sollte die String-Repräsentation (d.h. *textuelle* Repräsentation) eines Objektes zurückliefern
 - für den Menschen lesbar
 - zum Testen (für den Entwickler) oder als Rückmeldung (für den Anwender)
- dazu muss toString() in jeder Klasse *sinnvoll überschrieben* werden

Beispiel: toString()

```
public class Vektor2D {  
    ...  
    public String toString() {  
        return "(" + delX + ", " + delY + ")", anker: (" +  
            anker.getX() + ", " + anker.getY() + "));  
    }  
  
    public class VektorTest {  
        public static void main (String[] args) {  
            Vektor2D v2 = new Vektor2D(4,0,new Punkt(1,1));  
            System.out.println(v2);  
        }  
    }  
}
```

Programmausgabe: (4, 0), anker: (1,1)

Räumliche Vektoren

Beispiele:

Zeilenvektoren:

(13.1, -2, -7.5)

(9, 0, 1, 4)

transponieren

Spaltenvektoren:

$$\begin{bmatrix} 9 \\ 0 \\ 1 \\ 4 \end{bmatrix}$$
$$\begin{bmatrix} 2.5 \\ -1 \\ -7.7 \end{bmatrix}$$
$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$


Welche Attribute soll die Klasse Vektor haben?

Klasse Vektor

```
public class Vektor {  
    private int dimension;  
    private float[] komponenten;  
    private boolean istZeilenvektor = true;  
  
    public Vektor(int d, float[] k, boolean z) {  
        dimension = d;  
        komponenten = k;  
        istZeilenvektor = z;  
    }  
}
```

Konstruktor überladen

```
public Vektor(int d, float[] k) {  
    dimension = d;  
    komponenten = k;  
}
```



Was passiert, wenn $d \neq k.length$?
→ behandeln!

Klasse Vektor: Methoden

- `addiere()`, `subtrahiere()`
- `transponiere()`
- `betrag()`
- `skalarMultiplikation()`
- `skalarProdukt()`
- ...

Verständnisfragen



- Muss in jeder Klasse ein Konstruktor definiert werden?
- Welchen Rückgabebetyp kann ein Konstruktor haben?
- Können Datenfelder (Attribute) einer Klasse vom Typ einer anderen Klasse sein?
- Was ist der Unterschied zwischen einer Instanz- und Klassenmethode?
- Kann in Java eine Klassenmethode vom Objekt, d.h. mit `objekt.methode()` aufgerufen werden?
- Wozu ist die Methode `toString()` gut?
- Wenn `toString()` in einer Klasse A (sinnvoll) überschrieben worden ist, wie kann ein Objekt der Klasse A auf dem Bildschirm ausgegeben werden?

Matrizen

Wiederholung: mehrdimensionale Arrays

Matrizen

Beispiele:

(10.5 0 0 3.9)

$$\begin{pmatrix} 2 & 6 & -14 \\ 2 & 7 & -11 \end{pmatrix}$$

$$\begin{array}{c} \text{m} \\ \xrightarrow{\hspace{2cm}} \\ \begin{pmatrix} 2.5 & 1 & 3.9 & 2 \\ -1 & 2 & 3 & 4.5 \\ -7.7 & 5 & -9 & -9 \end{pmatrix} \end{array} \quad \begin{array}{l} \text{n} \uparrow \\ \text{n x m-Matrix} \\ \text{hier: 3 x 4-Matrix} \end{array}$$

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

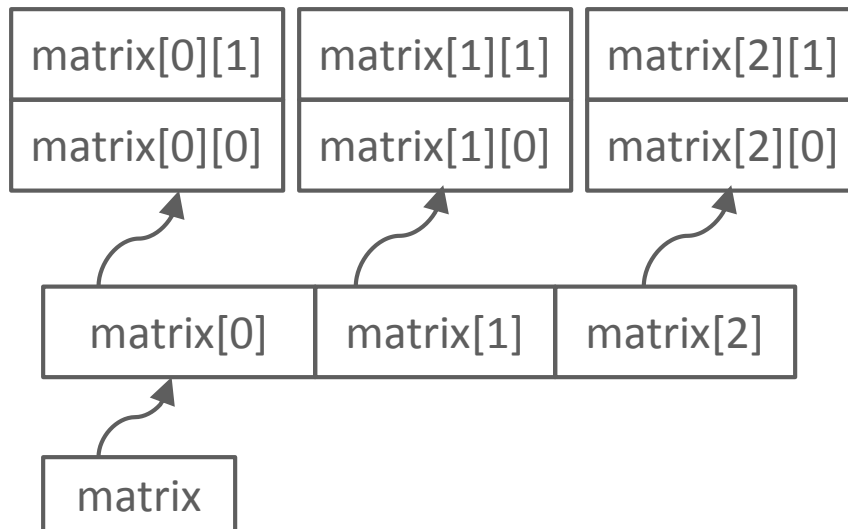


Welche Attribute soll die Klasse `Matrix` haben?

Mehrdim. Arrays erzeugen: Bsp.

- 2-dimensionaler (3x2-)Array aus `int`-Elementen:

```
int[][] matrix = new int[3][2];
```



- 2-dimensionaler Array aus `char`-Elementen:

```
char[][] zweiDArray = new char[3][7];
```

- 3-dimensionaler Array aus `byte`-Elementen:

```
byte[][][] dreiDArray = new byte[10][20][30];
```

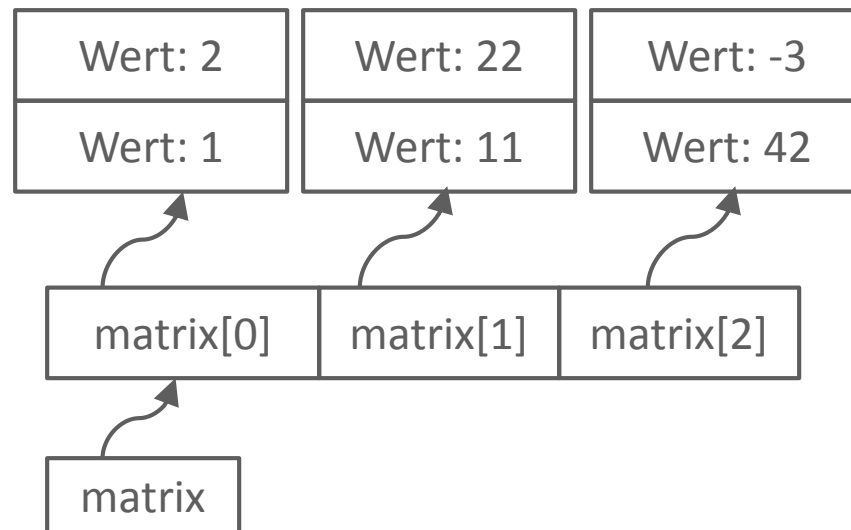
Mehrdim. Arrays initialisier. (1)

```
int[][] matrix = new int[3][2];
```

a) über Wertezuweisung an jedes Element einzeln

Beispiel:

```
matrix[0][0] = 1;  
matrix[0][1] = 2;  
matrix[1][0] = 11;  
matrix[1][1] = 22;  
matrix[2][0] = 42;  
matrix[2][1] = -3;
```



Wie Matrix

```
int[][] matrix = new int[3][2];
```

a) über Wertezuweisung an jedes Element einzeln

Beispiel:

```
matrix[0][0] = 1;
```

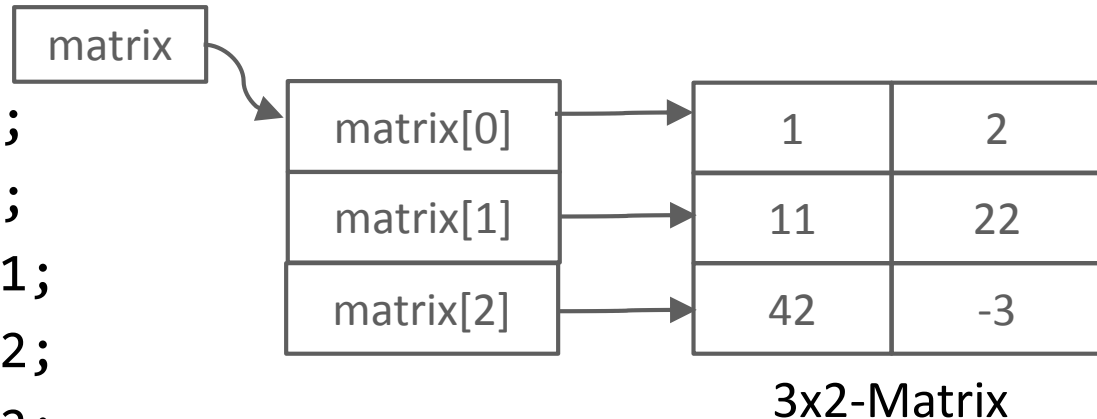
```
matrix[0][1] = 2;
```

```
matrix[1][0] = 11;
```

```
matrix[1][1] = 22;
```

```
matrix[2][0] = 42;
```

```
matrix[2][1] = -3;
```



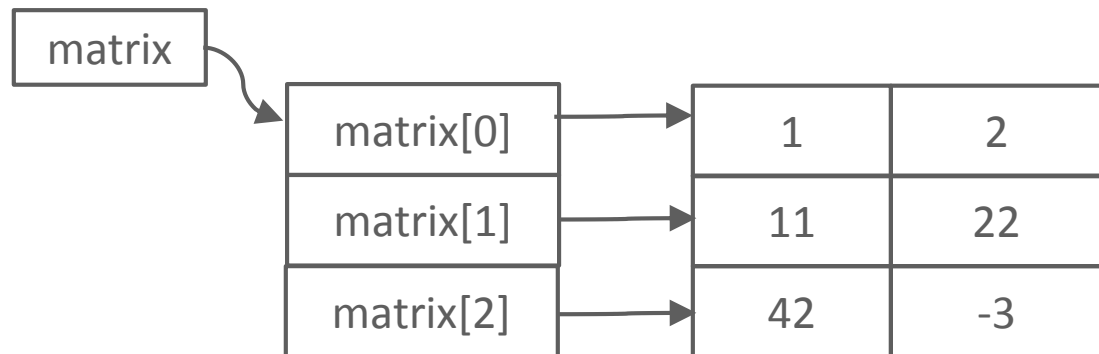
Mehrdim. Arrays initialisier. (2)

~~int[][] matrix = new int[3][2];~~

b) Implizites Erzeugen über eine Initialisierungsliste

Beispiel:

```
int[][] matrix = {{1,2}, {11,22}, {42,-3}};
```



Mehrdimensionale offene Arrays

Nur bei mehrdimensionalen Arrays möglich:

Länge einzelner Dimensionen nicht angegeben → die eckigen Klammern bei der Speicherplatz-Allokierung mit `new` leer gelassen

Beispiel:

```
int[][][][] matrix = new int[5][3][][];
```



Der ersten Dimension eines Arrays muss immer ein Wert zugewiesen werden!

Nicht erlaubt, nach einer leeren eckigen Klammer noch einen Wert in einer der folgenden Klammern anzugeben!

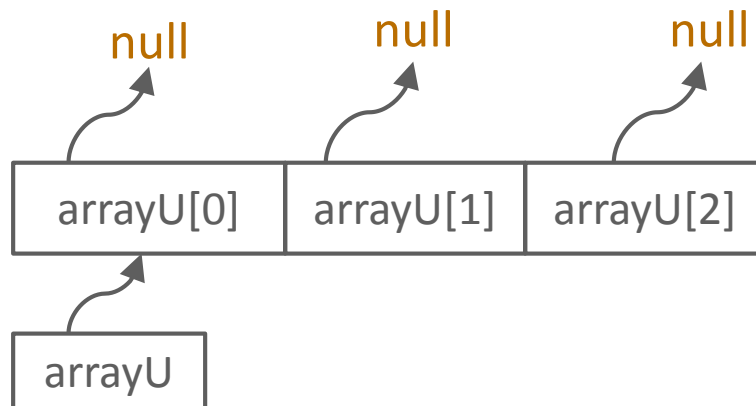
Gegenbeispiele:

```
int[][][][] matrix = new int[][][][];
```

```
int[][][][] matrix = new int[5][][][4];
```

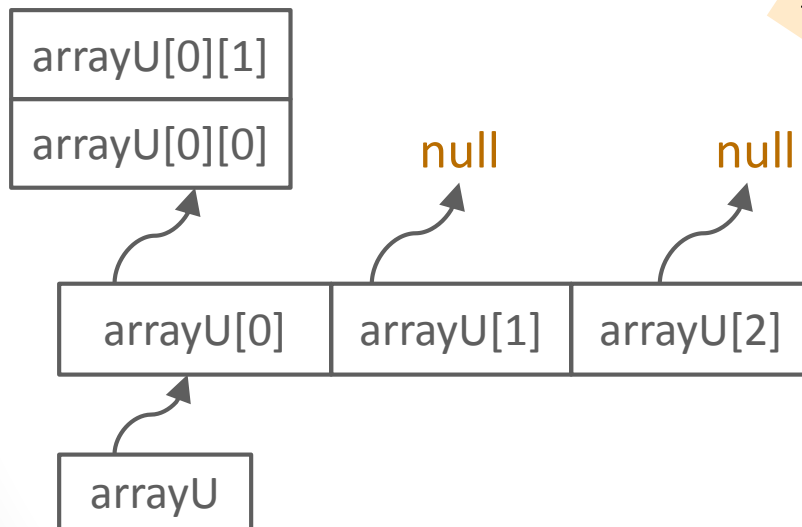
Offene Arrays: Beispiel

```
int[][] arrayU = new int[3][];
```



Offene Arrays: Eigenschaften

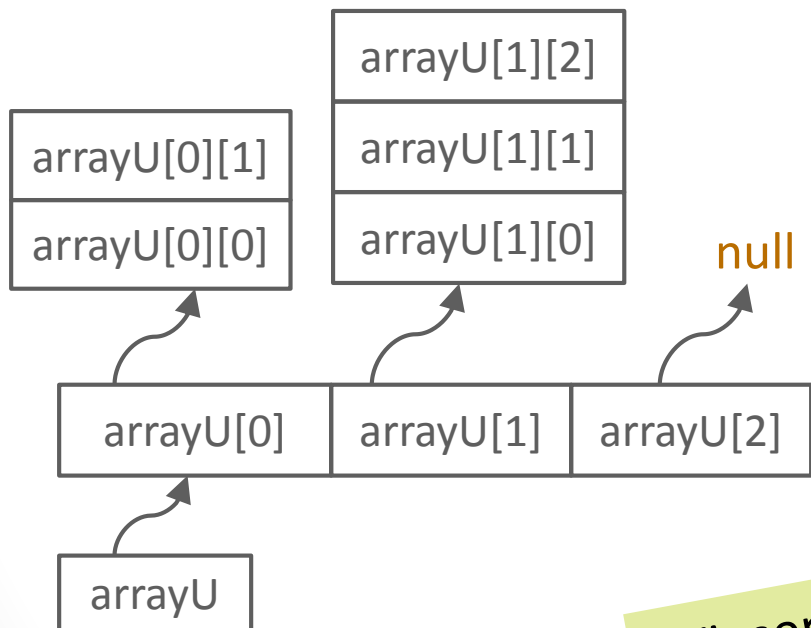
```
int[][] arrayU = new int[3][];  
arrayU[0] = new int[2];
```



können **stufenweise** erzeugt werden

Offene Arrays: Eigenschaften

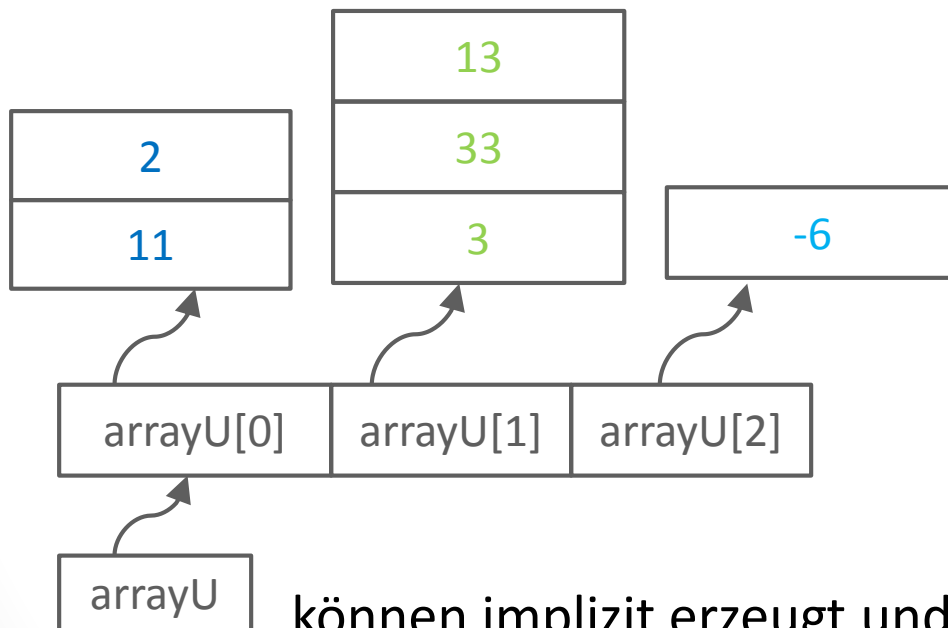
```
int[][] arrayU = new int[3][];  
arrayU[0] = new int[2];  
arrayU[1] = new int[3];
```



müssen **nicht** rechteckig sein

Offene Arrays: Eigenschaften

```
int[][] arrayU = new int[3][];  
arrayU[0] = new int[2];  
arrayU[1] = new int[3];
```



können implizit erzeugt und über **Initialisierungsliste** initialisiert werden:

```
int[][] arrayU = {{11, 2}, {3, 33, 13}, {-6}};
```

Verständnisfragen



- Wie viele Dimensionen kann ein Array besitzen?
- Von welchem Datentyp sind bei einem mehrdimensionalen Array die Elemente der 1. Dimension?
- Ist die Anzahl der Dimensionen bei der Definition eines Arrays (einer Referenz auf ein Array) immer festgelegt?
- Was sind offene Arrays?
- Können mehrdimensionale Arrays Elemente verschiedener elementarer DT enthalten?

Parameter der `main()`-Methode

Parameter von `main()`

`main()`-Methode:

```
public static void main (String[] args)
```

wozu?

→ in Java möglich, Parameter (als String-Objekte) über die Kommandozeile an ein Programm zu übergeben

Parameter von `main()` : Beispiel

// Datei: StringTest.java

```
public class StringTest {
```

```
    public static void main (String[] args) {
```

```
        String a = "HTW";
```

```
        String b = args[0];
```

```
        if (a.equals (b))
```

```
            System.out.println ("OK");
```

```
        else
```

```
            System.out.println ("Nicht OK");
```

```
    }
```

```
}
```

überprüft, ob der 1. per
Kommandozeile angegebener
Argument gleich „HTW“ ist

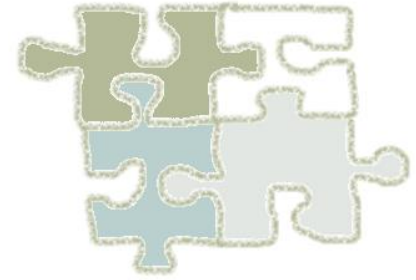
Aufruf des Programms: `java StringTest HTW`

OK

Parameterübergabe in Eclipse:

Run → Run Configurations... → ^(x)=Arguments → Program arguments

Zusammenfassung



- Array = grundlegende Datenstruktur
 - können aus abstrakten oder primitiven Datentypen bestehen (kaum Unterschied in der Handhabung)
 - können mehrdimensional sein
 - können offen sein
- Vektoren und Matrizen sehr elegant mit ein- und zweidimensionalen Arrays modelliert
- Klassenmethoden und Instanzmethoden in einer Klasse unterscheiden sich in ihrer Verwendung und Aufruf (wie?)
- zu textuellen Repräsentation eines Objektes sollte in *jeder* Klasse eine `toString()`-Methode implementiert sein.