

# Felder

- Felder
  - Eindimensional
  - Mehrdimensional

# Felder

- Anwendungszweck
  - Speicherung von beliebig vielen Werten mit einer Variablen
- Beispiele
  - Alle Wohnadressen einer Person
  - 52 Karten eines Kartenspiels
  - 3 x 3 Spielfelder von Tic-Tac-Toe
  - Vektoren und Matrizen

# Felder

- Ein Feld (engl. array) ist eine endliche Folge von Werten gleichen Datentyps (Feldtyp)

- Konzeptionelle Darstellung eines Felds
  - Anfang meist links oder oben

5	7	-2	0	11	5
---	---	----	---	----	---

1.0
3.4
0.5

- Feldvariable: Variable deren Datentyp ein Feld bezeichnet
- **Felder sind Objekte**
  - Feldvariablen speichern nur einen Verweis auf ein Feld
  - Standardwert von Feldvariablen ist null
- Index: Stelle eines Feldwerts, Fortlaufende Nummerierung beginnend mit 0

	0	1	2	3	4	5
zahlenfolge →	5	7	-2	0	11	5

# Eindimensionale Felder

- **Deklaration** Feldvariable mit einer Dimension:

`int a [ ];` // C-Stil, vermeiden

Feldtyp

`int [ ] a;` // Java-Stil

Datentyp von a

- **Erzeugen** eines Felds mit Konstruktoraufruf:

`a = new int[ 5 ];` // erzeugt ein Feld mit fünf int-Werten

Ausdruck mit Ergebnistyp int

- Die Feldwerte werden mit dem Standardwert des Feldtyps initialisiert



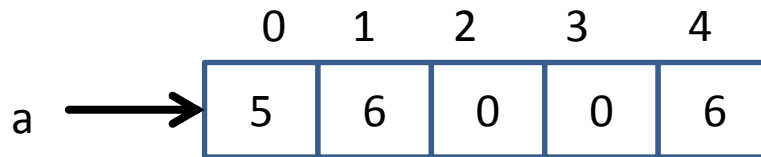
# Eindimensionale Felder

- Zugriff auf Feldwerte mit Indexoperator [ ]

`a[0] = 5;` // weise erstes Element den Wert 5 zu

`a[1] = a[0] + 1;`

`a[ a[0] - 1] = a[1];`



- `a[Ausdruck]`
  - Datentyp von Ausdruck muss int sein
  - verhält sich wie eine Variable des Feldtyps
- *Programmabbruch* bei Zugriff außerhalb des gültigen Indexbereichs
  - `a[6] = 2;` // Abbruch mit `IndexOutOfBoundsException`
  - `a[-2] = 5;` // ebenso
- *Programmabbruch* bei Zugriff auf Feldvariable mit Wert null
  - `Person [] personen = null;`
  - `personen[2] = new Person(25, "Müller");` // `NullPointerException`

# Eindimensionale Felder

- Alternative Deklaration mit Angabe konkreter Werte in { } - Klammern
  - Bei Deklaration

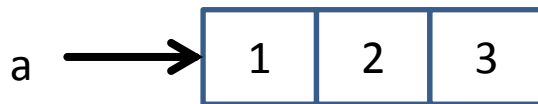
```
int [] a = {5, 2, 7};           // nur bei
Variableninitialisierung
```

```
Person [] Ehepaar = { new Person(25, "Müller"),
                      new Person(72, "Meier") };
```
  - Bei Konstruktoraufruf (Anzahl wird weggelassen)

```
a = new int[ ]{5, 2, 7};    // Konstruktoraufruf mit Angabe
der Werte
```

# Eindimensionale Felder

- *Beachte:* Das Schlüsselwort `final` bezieht sich nur auf den Wert der Variablen nicht den Wert auf den verwiesen wird:  
`final int [] a = {1, 2, 3};`  
`a = new int[3];` // nicht möglich  
`a[0] = 5;` // in Ordnung, da Verweis sich nicht ändert
- Felder besitzen ein `final int` Objektattribut **length**. Es gibt die Anzahl Elemente des Feldes an.



`a.length` hat den Wert 3

- Die Anzahl Elemente kann nach Erzeugung des Felds nicht mehr geändert werden.

# Eindimensionale Felder

## Quadrate ganzer Zahlen berechnen

Gegeben	Eine ganze Zahl $n \geq 1$
Gesucht	Alle Quadrate $1^2, 2^2, 3^2, \dots, n^2$

- Feld anlegen mit Feldtyp int und n Werten
- Werte von 1 bis n mit einer Zählvariablen aufzählen
  - Bei jedem Durchlauf das Quadrat berechnen und dem zugehörigen Feldwert zuweisen
- Zum Testen die Werte auf dem Bildschirm ausgeben



# Eindimensionale Felder

## Primzahlen von 2 bis n berechnen

Gegeben

Eine ganze Zahl  $n \geq 2$

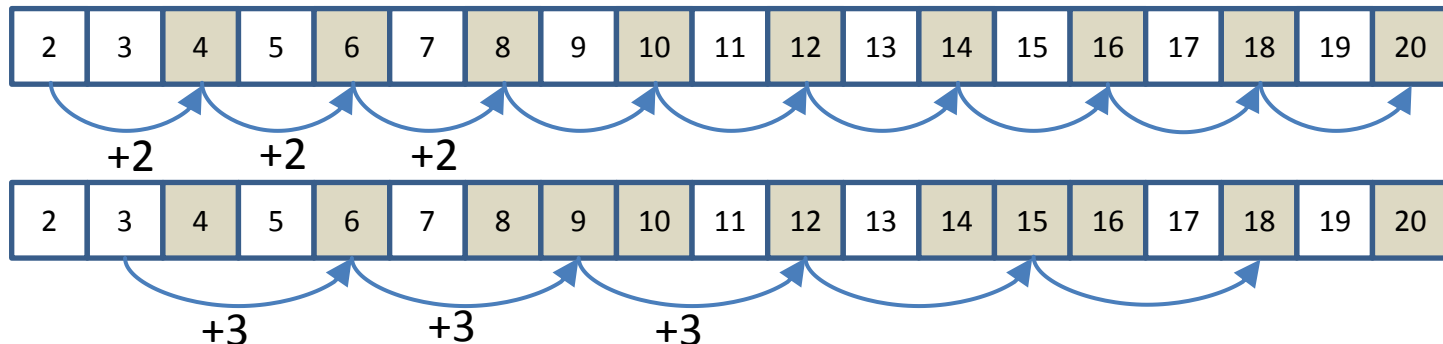
Gesucht

Alle Primzahlen von 2 bis n

- **Sieb des Eratosthenes**

Ausgehend von der kleinsten Primzahl  $p = 2$ :

- alle Vielfachen von  $p$  bis  $n$  streichen
- mit der nächsten nicht gestrichenen Zahl wiederholen, solange bis  $n$  erreicht ist



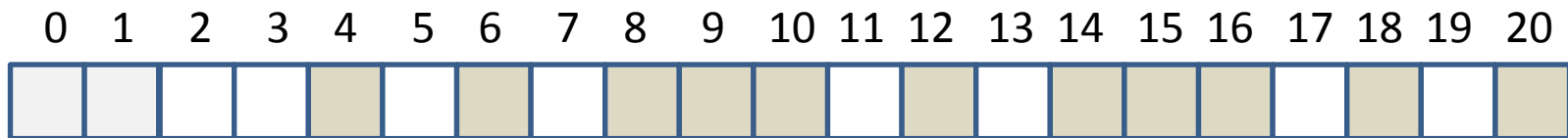
# Eindimensionale Felder

- Nur das Streichen codieren

`true` = Zahl ist gestrichen

`false` = Zahl ist nicht gestrichen (Primzahl)

- Index = Wert der gestrichenen Zahl

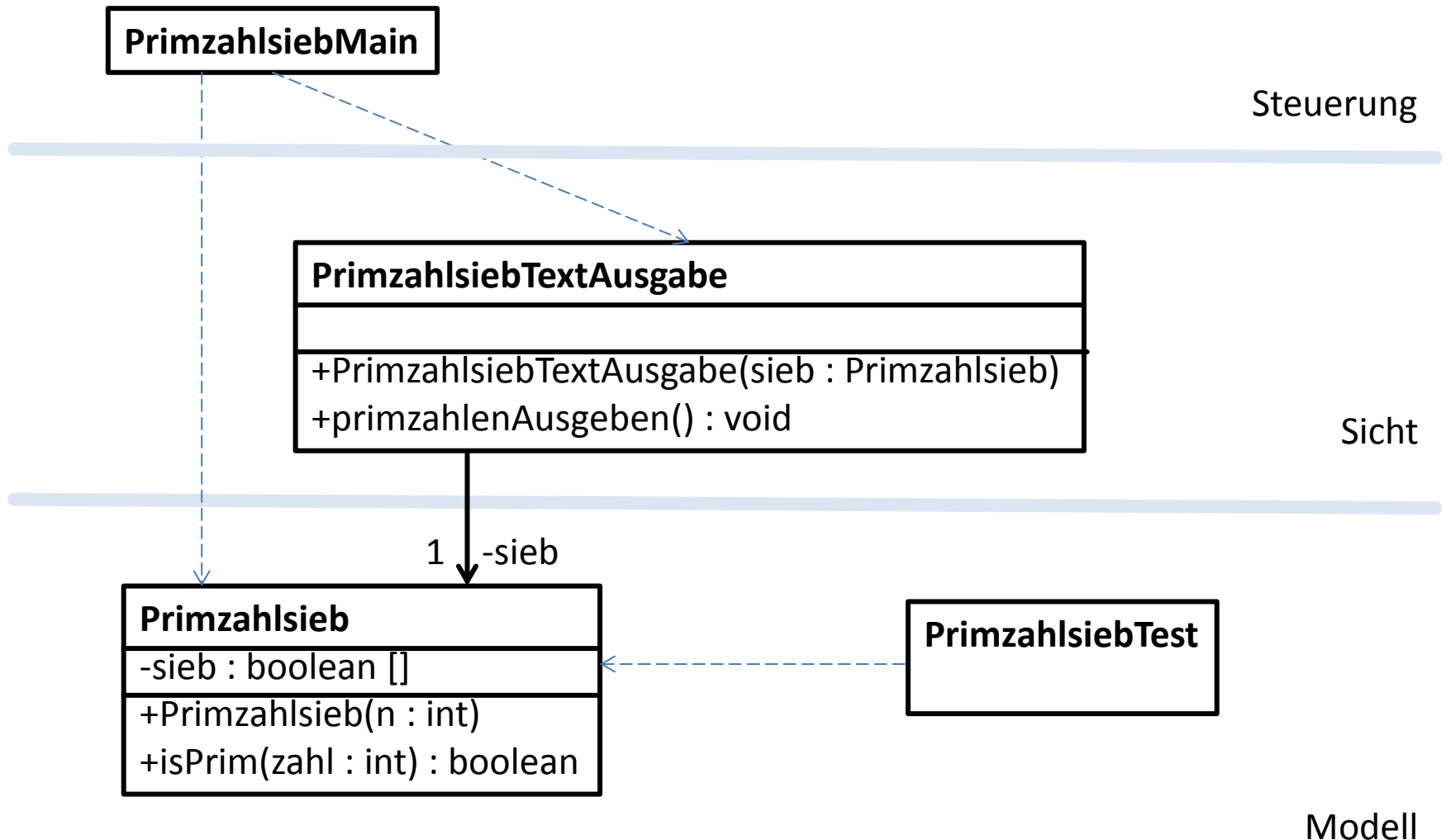


Nicht  
verwendet

# Eindimensionale Felder

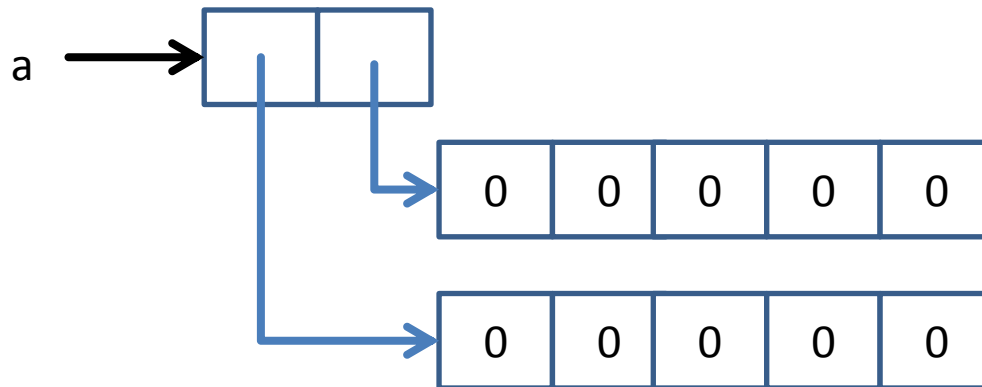
- Entwurfsidee
  - Eine Methode nötig, die für eine Zahl überprüft, ob sie eine Primzahl ist oder nicht
  - Das Primzahlsieb ist lediglich, der in der Klasse verwendete Algorithmus für die Überprüfung (es gibt andere Verfahren)
- Javadoc vor der Implementierung schreiben
- JUnit-Tests **vor** der Implementierung programmieren
  - „**black box testing**“: Tests werden **ohne** Kenntnis des zu testenden Quelltexts erstellt
  - „**white box testing**“: Tests werden mit Kenntnis des zu testenden Quelltextes erstellt
  - Programme sollten sowohl mit black box als auch white box testing überprüft werden
- Die Implementierung ist fertig, wenn alle JUnit-Tests erfolgreich waren
  - Viele Tests nötig

# Eindimensionale Felder



# Mehrdimensionale Felder

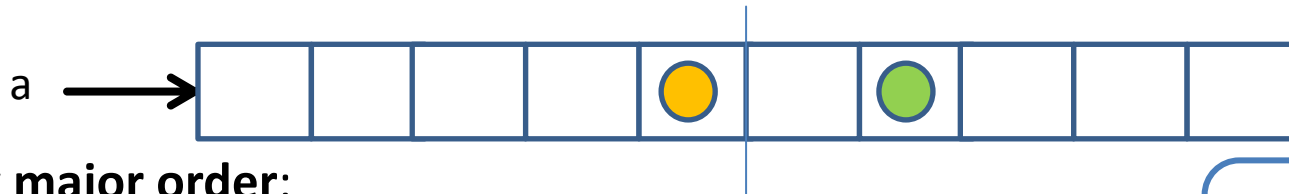
- **Deklaration:** ein Klammerpaar pro Dimension  
`int [ ] [ ] a; // zwei Dimensionen`  
`int [ ] [ ] [ ] b; // drei Dimensionen`
- **Erzeugen** mit Konstruktoraufruf: pro Dimension Anzahl Werte angeben  
`a = new int[2][5];`
- a ist ein Verweis auf ein Feld mit 2 Werte und Feldtyp `int [ ]`
- Die zwei Werte enthalten deswegen Verweise auf eindimensionale Felder mit Feldtyp `int`



# Mehrdimensionale Felder

- Die Werte mehrdimensionaler Felder sind in Java **nicht** hintereinander im Speicher angeordnet
- In systemnahen Sprachen wie C aber schon:

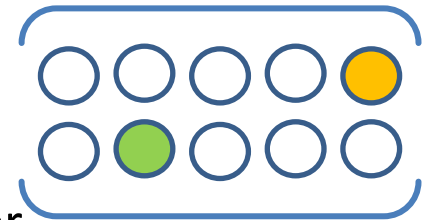
```
int a[2][5];
```



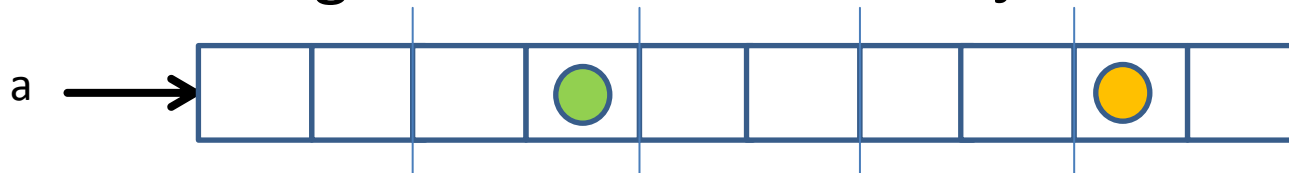
**row major order:**

erster Index wird als Zeileindex ( 2 ) und  
zweiter als Spaltenindex (5) interpretiert

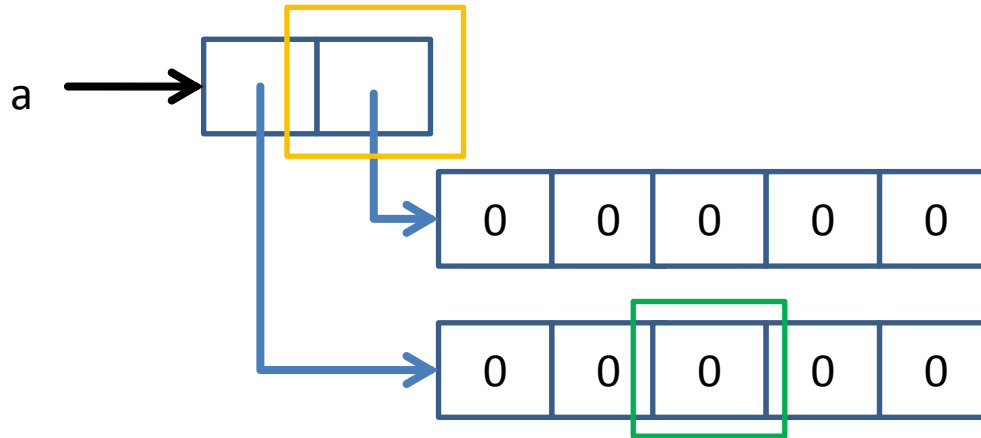
Die Werte der Zeilen stehen hintereinander im Speicher



- Anordnung in Fortran: **column major order**



# Mehrdimensionale Felder



- Zugriff via [ ] wie bei eindimensionalen Feldern
- Ein Indexzugriff folgt einem Verweis
- Feldtyp der Indizierten Werte beachten

`a[1]`

Verweis auf Feld mit Feldtyp int []

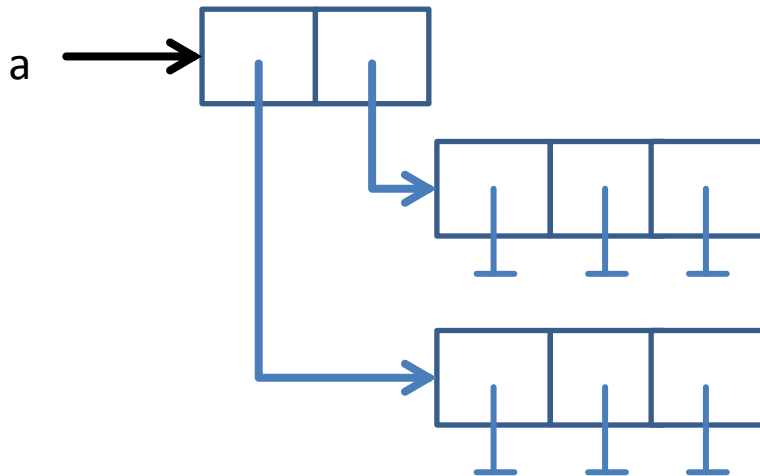
`a[1][2]`

Zweiter Index bezieht sich auf Wert von `a[1]`

# Mehrdimensionale Felder

- Konstruktoraufruf:
  - Mindestens eine Dimension muss angegeben werden
  - Hintere Dimensionen können weggelassen werden
  - Werte der letzten Dimension sind dann null

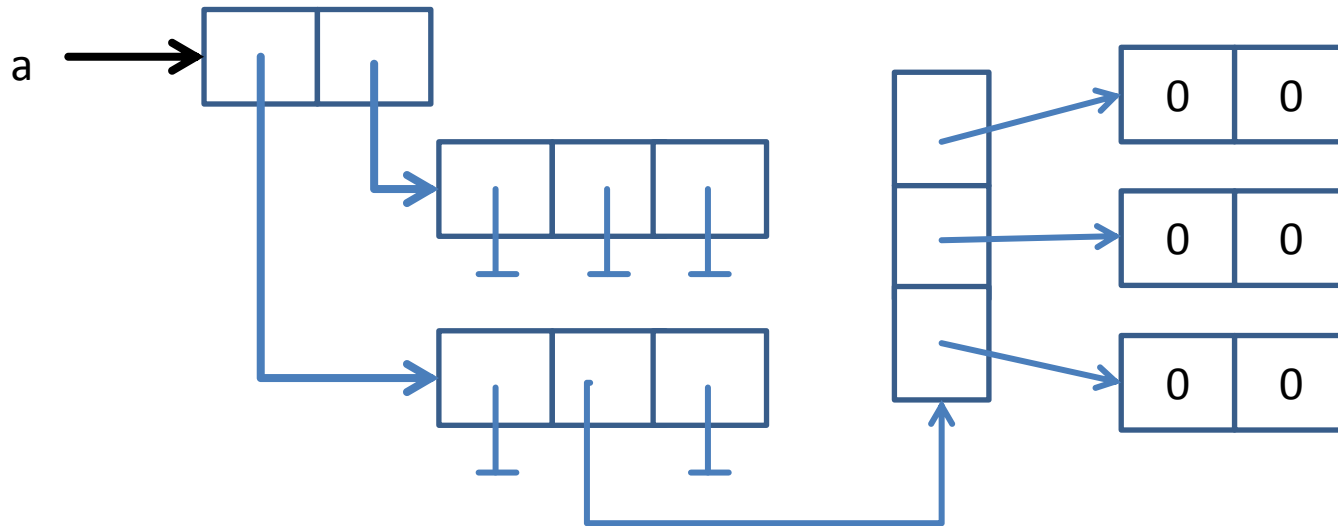
```
int [] [] [] a = new int[2][3][ ] [ ] ;
```





# Mehrdimensionale Felder

```
int [] [] [] [] a = new int[2][3][ ][ ] ;  
a[0][1][2] = ... ; // NullPointerException  
a[0][1] = new int[3][2];
```

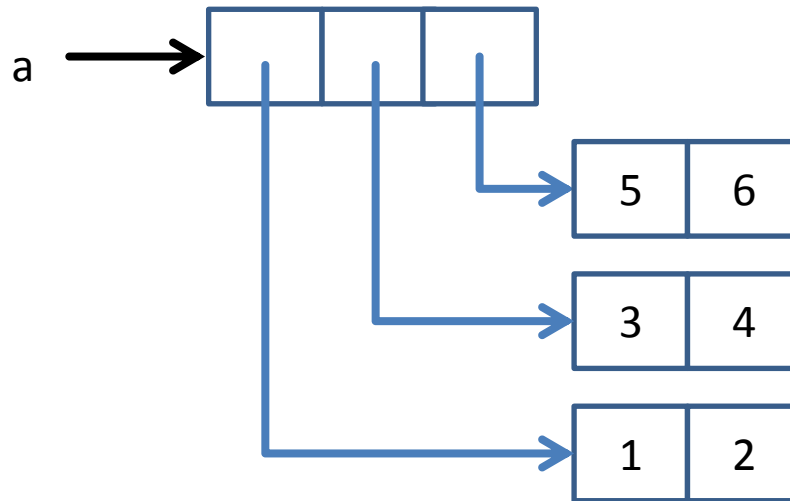


Verwendung: z.B. sehr grosse, aber fast leere Matrizen; spart Speicher

# Mehrdimensionale Felder

- Konstruktoraufruf mit Angabe von Werten
  - Geschweifte Klammern um Werte ergeben ein Feldwert

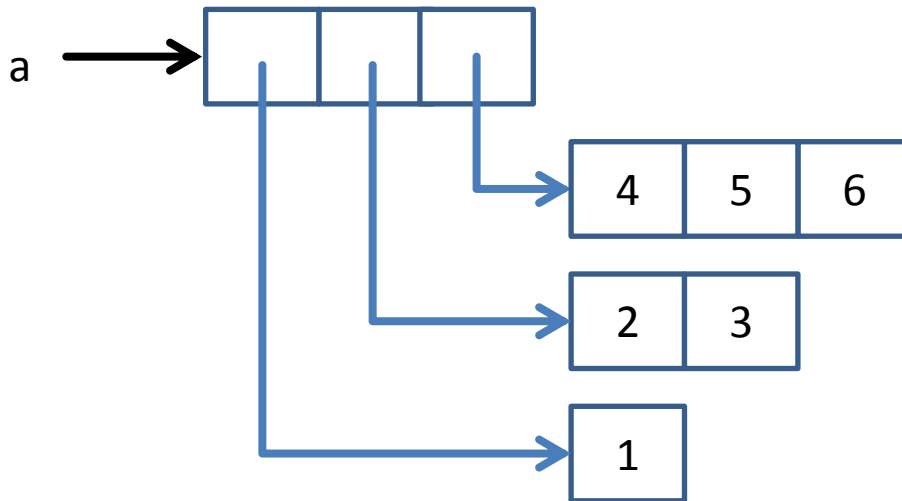
```
int [ ] [ ] a = { {1, 2}, {3, 4}, {5, 6} };  
a = new int[ ] [ ] { {1, 2}, {3, 4}, {5, 6} };
```



# Mehrdimensionale Felder

- Bisher: symmetrische Felder
- Asymmetrisches Feld (*ragged, jagged* array)
  - Dimension der Felder sind verschieden
  - Vermeiden, da fehleranfällig;
  - Spart Speicher

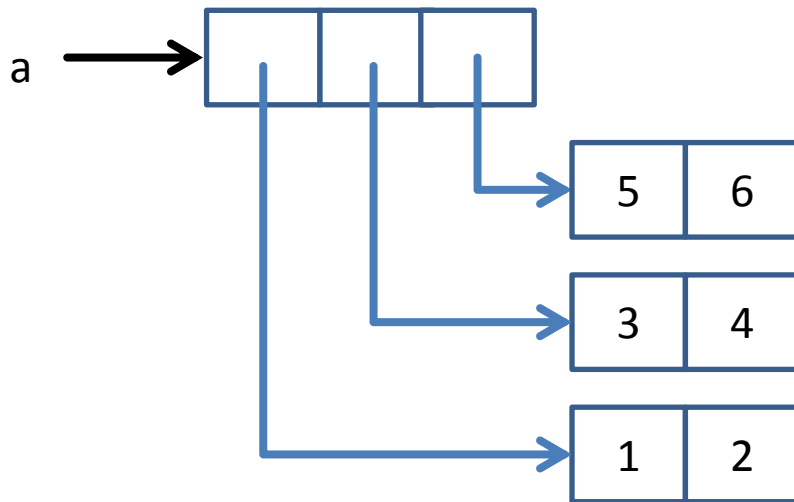
```
int [] [] a = { {1}, {2, 3}, {4, 5, 6} };
```



# Mehrdimensionale Felder

- Die **Interpretation** der Daten eines Felds ist durch das Programm und damit vorab durch den Programmierer gegeben
- Diese Interpretation ist nicht offensichtlich und sollte mit einem Kommentar dokumentiert werden

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



Viele mögliche Interpretationen  
als 2-dimensionale Matrix


$$\begin{pmatrix} 5 & 6 \\ 3 & 4 \\ 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

# Mehrdimensionale Felder

- Eine Matrix im Zweifelsfall so codieren:

`int [ ] [ ] a = { {1, 2},  
                  {3, 4},  
                  {5, 6} };`           $\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$

- Vorteile
  - Struktur der Matrix bei Angabe konkreter Werte direkt sichtbar
  - Erster Index ist der Zeilenindex wie bei Matrizen

# Mehrdimensionale Felder

## Kleine Ein-Mal-Eins berechnen

Gegeben	
Gesucht	Feld mit den Werten des kleinen Ein-Mal-Eins

$$\begin{pmatrix} 1 & \dots & 10 \\ \vdots & \ddots & \vdots \\ 10 & \dots & 100 \end{pmatrix}$$

- Zeilenindex zuerst, wie bei einer Matrix

# Mehrdimensionale Felder

## Binomialkoeffizienten berechnen

Gegeben	$0 \leq k \leq n$
Gesucht	Binomialkoeffizient $\binom{n}{k}$

$$\binom{n}{k} := \frac{n!}{(n-k)!k!} \quad \text{für } 0 \leq k \leq n \quad (n \text{ über } k)$$

- Beispiel für asymmetrisches Feld
- Wieso keine direkte Berechnung mit diese Definition?

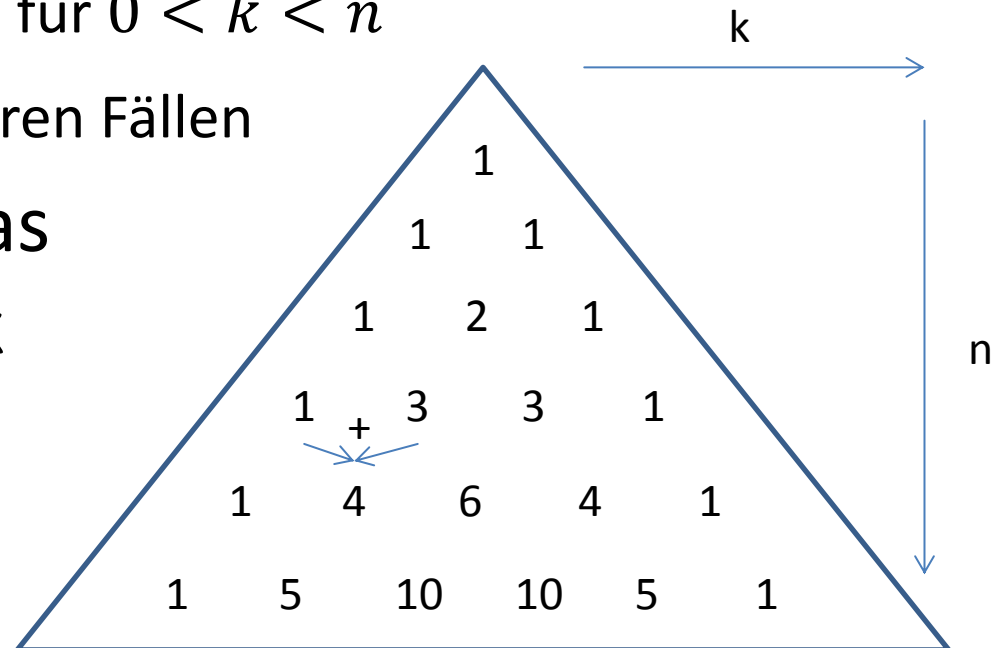
# Mehrdimensionale Felder

- Folgende Eigenschaft des Binomialkoeffizient kann als Definition verwendet werden

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ für } 0 < k < n$$

Der Wert ist 1 in den anderen Fällen

- Berechnung über das **Pascalsche-Dreieck**





# Mehrdimensionale Felder

- Berechnung mit Hilfe des Pascalschen Dreiecks ist ein Beispiel des Dynamisches Programmierens:
  1. Lösungen von Teilproblemen direkt berechnen
  2. Verwende diese Lösungen, um nächst größere Teilprobleme zu lösen.
    - Führe dies fort, bis das ursprüngliche Problem gelöst wurde.
- Ca. 1940 von Richard Bellman für Optimierungsprobleme beschrieben.
- Weiteres Beispiel:
  - Cocke-Kasami-Younger-Algorithmus (Theoretische Informatik 1)

# Dynamisches Programmieren

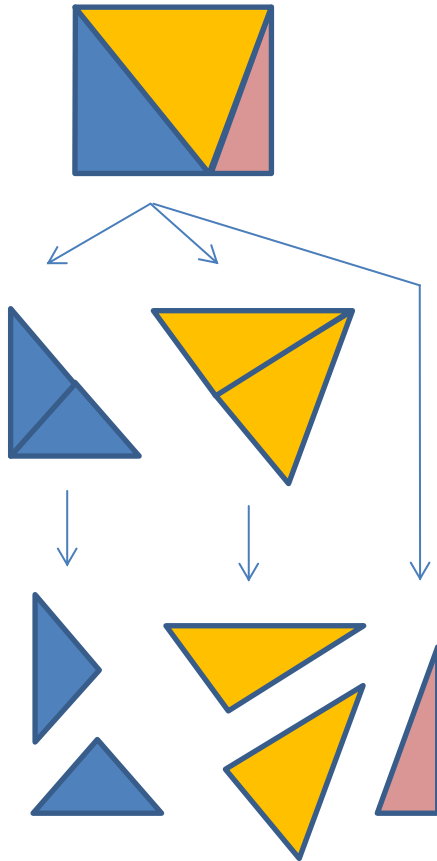
## Algorithmusprinzip

Problem in gleichartige

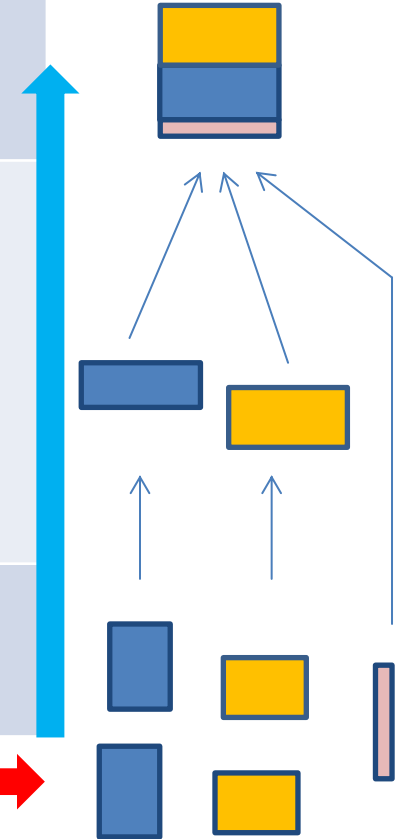
**Teilprobleme unterteilen**

- alle **Lösungen der Teilprobleme** bestimmen
- auf bestehende Teillösungen zurückgreifen
- alle Teillösungen speichern

Aus den Teillösungen die **Gesamtlösung konstruieren**



Unterteilen „top-down“



Zusammenfügen „bottom-up“