



## Struktur des Datenspeichers

Der Datenspeicher eines Java-Programms besteht aus zwei Teilen:

- der **Keller** für die lokalen Variablen und ihre Werte
- die **Halde** („heap“) für die Reihungen (und die Objekte – siehe später)

## Zugriff auf Reihungen

- `a[i]` bezeichnet den Zugriff auf die *i*-te Komponente der Reihung `a`

### Beispiel:

Mit `a[0]`, `a[1]`, ..., `a[5]` kann man auf die Komponenten der Beispielreihung `a` zugreifen.

- `a.length` gibt die Länge der Reihung an.

Im **Beispiel** hat `a.length` den Wert 6.

## Deklaration von Reihungstypen und -variablen

In Java haben **Reihungstypen** die Form

`type[]... []`

### Beispiel:

`int[], int[][] , boolean[]`

Typ einer 1-dim. Reihung

Typ einer 2-dim. Reihung (siehe später)

## Eindimensionale Reihungen

**Deklaration** einer Reihung mit Elementen vom Typ `type` :

`type[] var = new type[n];`

Die Deklaration deklariert lokale Variable `var` vom Type `type[]` und reserviert Speicherplatz für eindimensionale Reihung der Länge `n`

Erzeugt neue Reihung der Länge `n`, bei der jede Komponente mit dem Standardwert von `type` initialisiert wird.

- Durch die Deklaration werden außerdem implizit `n` zusammengesetzte Variablen `var[0]`, ..., `var[n-1]` erzeugt, mit denen man auf die Werte der Komponenten von `var` zugreifen und diese Werte verändern kann.
- Standardwert (Defaultwert) von `int` : 0  
von `double` : 0.0  
von `boolean` : `false`

## Eindimensionale Reihungen: Initialisierung

**Sofort Anfangswerte zuweisen** („Initialisierung“)

- `type[] var = {v0, ..., vn-1}` // sofortige Zuweisung

vom Typ `type`

! Diese Art der Initialisierung ist aber nur in einer Deklaration zulässig.

- oder Zuweisung an Komponenten:

```
type[] var = new type[n];
var[0] = v0;
⋮
var[n-1] = vn-1;
```

## Eindimensionale Reihungen: Initialisierung

**Beispiel:**

```
char[] a = {'V', 'E', 'R', 'L', 'A', 'G'}
```

Typ von `a` ist `char[]`, d.h. der Typ einer einstufigen Reihung mit Elementen aus `char`.

- Man kann die Reihung initialisieren durch Einzelzuweisungen der Werte:

```
char[] a = new char[6];
a[0] = 'V';    a[3] = 'L';
a[1] = 'E';    a[4] = 'A';
a[2] = 'R';    a[5] = 'G';
```

## Eindimensionale Reihungen

- Man kann beliebige einzelne Buchstaben ändern:

```
a[3] = 'R';
a[5] = 'T';
```

- Das ergibt 'V' 'E' 'R' 'R' 'A' 'T' als neuen Wert der Reihung. Außerdem hat `a[3]` nun den Wert 'R'.

## Direkte Zuweisung

```
char[] c = {'L', 'M', 'U'};
a = c;
```

- Bemerkung:** Da in Java die Länge der Reihung aber nicht Bestandteil des Typs ist, kann einer Feldvariablen eine Reihung mit einer anderen als der initial angegebenen Länge zugewiesen werden.

## Reihungen und for-Schleifen

- Die Länge einer Reihung steht in dem Attribut `length`

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

- for-Schleifen eignen sich gut um Reihungen zu durchlaufen

```
for (int k=1; k<länge; k++)
{ myArray[k] = myArray[k-1] + myArray[k]
}
```



- Typische Suche nach einem Element in einer Reihung - mit vorzeitigem Verlassen:

```
int k=0; int element = 6;
while (k<länge && myArray[k] != element) k++;
gefunden = (k<länge);
```

## Suche nach dem Index eines minimalen Elements einer Reihung

Gegeben sei folgendes Feld:

3	-1	15	1	-1
---	----	----	---	----

**Algorithmus:**

- Bezeichne *minIndex* den Index des kleinsten Elements
- Initialisierung *minIndex* = 0
- Durchlaufe die ganze Reihung. In jedem Schritt *i* vergleiche den Wert von *minIndex* (d.h.  $a[\text{minIndex}]$ ) mit dem Wert des aktuellen Elements (d.h.  $a[i]$ ). Falls  $a[i] < a[\text{minIndex}]$  setze *minIndex* = *i*

## Suche nach dem Index eines minimalen Elements einer Reihung

Java Implementierung

Sei die Reihung `int[] a` gegeben.

```
int minIndex = 0;
for (int i = 1; i < a.length; i++) // Optimierung, da
// a[0] < a[0] falsch ist
{
    if (a[i] < a[minIndex])
        minIndex = i;
}
int minElem = a[minIndex]; // minElem ist der Wert
// des kleinsten Elements;
// minIndex ist der am weitesten links
// stehende Index
// eines kleinsten Elements
```

## Binäre Suche eines Elements *e* in einer geordneten Reihung

Sei *a* ein geordnete Reihung mit den Grenzen *j* und *k*, d.h.  $a[i] \leq a[i+1]$  für  $i=j, \dots, k$ ; also z.B.:

a:	3	7	13	15	20	25	28	29
	<i>j</i>	<i>j+1</i>	...					<i>k</i>

**Algorithmus:**

Um den Wert *e* in *a* zu suchen, teilt man die Reihung in der Mitte und vergleicht *e* mit dem Element in der Mitte:

- Ist  $e < a[\text{mid}]$ , so sucht man weiter im linken Teil  $a[j], \dots, a[\text{mid}-1]$ .
- Ist  $e = a[\text{mid}]$ , hat man das Element gefunden.
- Ist  $e > a[\text{mid}]$ , so sucht man weiter im rechten Teil  $a[\text{mid}+1], \dots, a[k]$ .

## Binäre Suche eines Elements $e$ in einer geordneten Reihung

Implementierung in Java: Sei  $int\ e$  das Element, das in der Reihung  $int[]\ a$  gesucht wird.

```
int j = 0; //linke Grenze
int k = a.length-1; //rechte Grenze
int mid; //Index der Mitte
boolean found = false;
while (j <= k & !found) //solange Reihung nicht leer und e nicht gefunden
{
    mid = (j + k) / 2; //Berechnung der Mitte
    if (e < a[mid]) //falls e kleiner als das mittl. Element
        k = mid - 1; //setze rechte Grenze unterhalb die Mitte
    else //sonst
    {
        if (e == a[mid]) //falls e gleich mittl. Element
            found = true; // e ist in a gefunden
        else //sonst
            j = mid + 1; //setze linke Grenze oberhalb die mitte
    }
}
boolean result = found; //e gefunden genau dann, wenn found == true
```

## Mehrdimensionale Reihungen

- Matrizen sind mehrdimensionale Reihungen
- Man benutzt Matrizen zur Speicherung und Bearbeitung von
  - Bildern
  - Operationstabellen
  - Wetterdaten
  - Graphen
  - Distanztabelle
  - ...
- Deklaration
  - `int [][] greyMonaLisa;`
  - `boolean [][] xorTabelle`
- Deklaration mit Erzeugung
  - `int [][] entfernung = new int[4][4];`

## Mehrdimensionale Reihungen

- 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}};`

## Mehrdimensionale Reihungen

### Allgemein

```
type[]...[] var = new type[n1]...[ni][...] (i > 0)
```

deklariert eine Variable  $var$  vom Typ  $type[]\dots[]$ , reserviert Speicherplatz für ein mehrstufiges Feld.

- Mindestens die Länge  $n_1$  des ersten Indexbereiches muß angegeben werden.

### Initialisierung:

```
type[][] var = {f0, ..., fn1-1};
```

Reihungen von Werten vom Typ  $type[]$

Eine mehrdim. Reihung ist eine Reihung von Reihungen

- Dabei können die Längen von  $f_0, \dots, f_{n_1-1}$  unterschiedlich sein.
- Analog für höherdimensionale Reihungen.

## Mehrdimensionale Reihungen

### Ausdrücke

- Zusammengesetzte Variablen

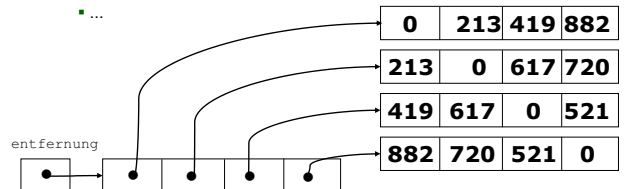
```
var [i1]...[in]
```

- Initialisierungsausdrücke der Form

```
new type[n1]...[ni][...] bzw. {v0, ..., vn1-1}
```

## Mehrdimensionale Reihungen - gibt's gar nicht!

- Mehrdimensionale Reihungen braucht man **eigentlich** nicht
  - Eine zweidimensionale Reihung ist eine Reihung von Zeilen
  - Eine dreidimensionale Reihung ist eine Reihung von zweidimensionalen Reihungen
  - ...



## Mehrdimensionale Reihungen

### Beispiel: Tastentelefon

Ein Tastentelefon besteht aus 4 Zeilen und 3 Spalten:

	0	1	2
3	1	2	3
2	4	5	6
1	7	8	9
0	*	0	#

```
char[][] tastenwert = { {'*', '0', '#'},
                        {'7', '8', '9'},
                        {'4', '5', '6'},
                        {'1', '2', '3'} };
```

z.B.

```
tastenwert[3][0] = '*'
tastenwert[0][2] = '#'
```

M. Wirsing: Reihungen

## Reihungen: Krumm und schief

### Die Dimension einer Reihung ist nicht Teil ihres Typs

- Folglich können verschieden große Reihungen den 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 Reihungen?

- Die innere for-Schleife muss die Länge der zu durchlaufenden Zeilen selber bestimmen
- Das geht mittels des `length`-Attributs

M. Wirsing: Reihungen

## Reihungen: Krumm und schief

### Durchlauf durch schiefArray

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

### Beispiel Pascal-Dreieck

```
for (int zeile=0; zeile < pascalDreieck.length; zeile++)
{
    pascalDreieck[zeile][0] = 1; pascalDreieck[zeile][zeile] = 1;
    for (int spalte=1; spalte < pascalDreieck[zeile].length-1; spalte++)
        pascalDreieck[zeile][spalte] =
            pascalDreieck[zeile-1][spalte-1] + pascalDreieck[zeile-1][spalte];
}
```

M. Wirsing: Reihungen

## Zusammenfassung

- Reihungen sind mathematisch gesehen endliche Abbildungen von einem Indexbereich auf einen Elementbereich.
- Im Speicher werden Reihungen repräsentiert als Zeiger auf Vektoren (vgl. später Objekte)
- Klassische Suchalgorithmen sind
  - die binäre Suche in einer geordneten Reihung und
  - die Suche nach dem Index mit dem kleinsten Element in einer ungeordneten Reihung.

M. Wirsing: Reihungen