# Agenda

- Fixed-sized collections: Arrays
- Emergence: **Fibonacci, automaton project**
- Multidimensional Array: **Brain project**
- **Class variables & methods**
- **Clean code**

# Fixed-sized collections
# Arrays

# Fixed-size collections

Sometimes the maximum collection size can be pre-determined.

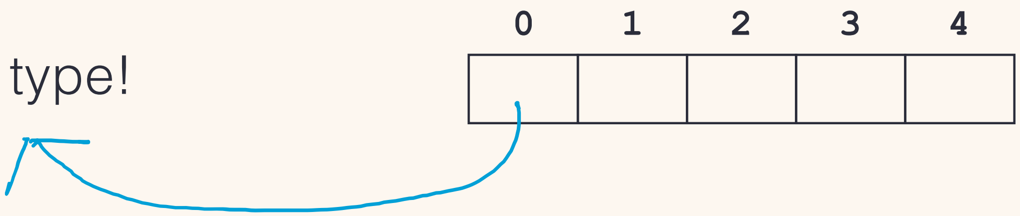
A special fixed-size collection type is available: an *array*.

Unlike the flexible **`List`** collections, arrays can store object references or primitive-type values (without autoboxing).

Index : 0- n-1

Length: number of elements

All elements are of the same type!



# Standard array use

Type

```
private int[] hourCounts;
private String[] names;
```

declaration

```
...

hourCounts = new int[24];
```

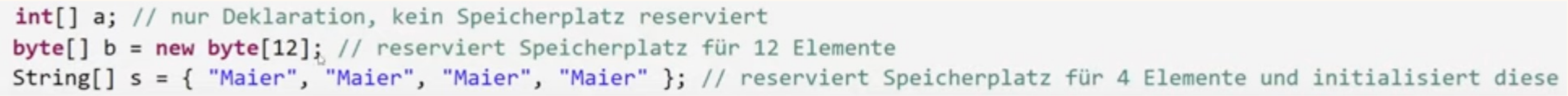
creation

integer Value

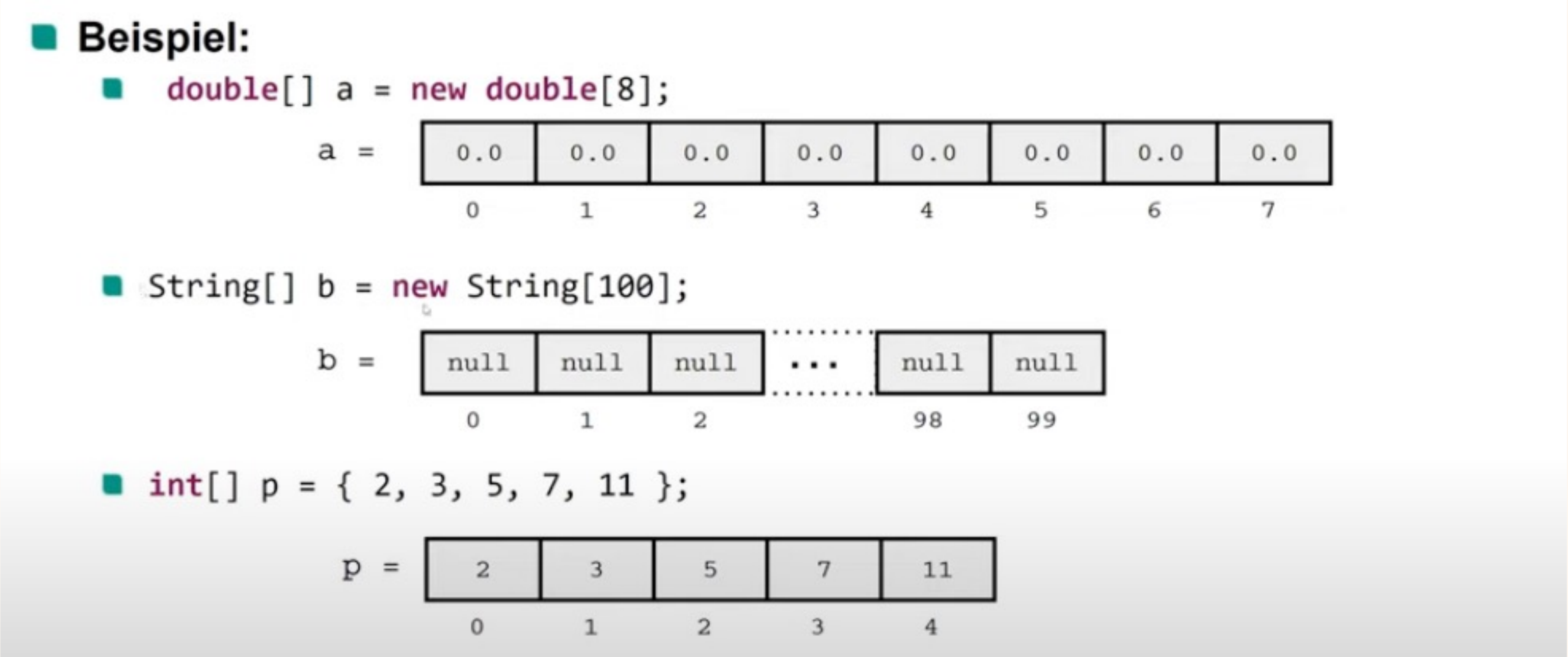
```
...

hourCounts[i] = 0;
hourCounts[i]++;
System.out.println(hourCounts[i]);
```

use

# Beispiele



```java
int[] a; // nur Deklaration, kein Speicherplatz reserviert
byte[] b = new byte[12]; // reserviert Speicherplatz für 12 Elemente
String[] s = { "Maier", "Maier", "Maier", "Maier" }; // reserviert Speicherplatz für 4 Elemente und initialisiert diese
```



- **Beispiel:**
  - `double[] a = new double[8];`

```
a =  | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
        0     1     2     3     4     5     6     7
```

  - `String[] b = new String[100];`

```
b =  | null | null | null | ... | null | null |
        0      1      2            98     99
```

  - `int[] p = { 2, 3, 5, 7, 11 };`

```
p =  | 2 | 3 | 5 | 7 | 11 |
       0   1   2   3   4
```

# Features of arrays

Fixed in length, unlike **`ArrayList`**, **`HashSet`**, **`HashMap`**, etc.

Use a special syntax.

- For historical reasons.

Objects with no methods.

- Methods are provided by other classes; e.g., **`java.util.Arrays`**.
- Methods that are static.

# Array literals

Array literals in this form can only be used in declarations.

Later uses require **`new`**:

- The size is inferred from the data.

```
private int[] numbers = { 3, 15, 4, 5 };
```

declaration, creation and initialization

```
numbers = new int[] {
    3, 15, 4, 5
};
```

# Array length

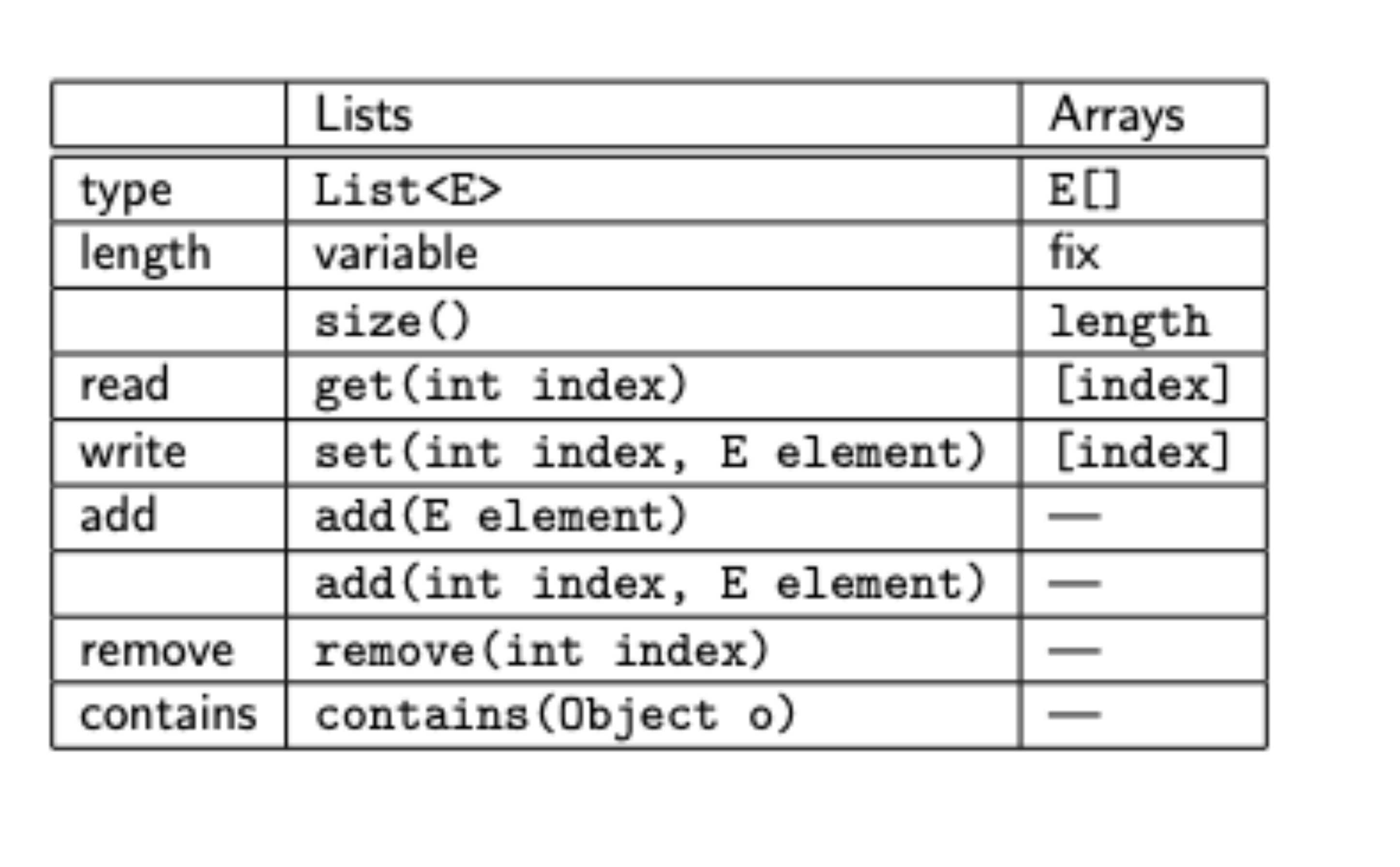
NB: **`length`** is a field rather than a method!

It cannot be changed – ‘fixed size’.

```
private int[] numbers = { 3, 15, 4, 5 };

int n = numbers.length;
```

no brackets!

# ArraList vs Array



| | Lists | Arrays |
|---|---|---|
| type | `List<E>` | `E[]` |
| length | variable | fix |
| | `size()` | `length` |
| read | `get(int index)` | `[index]` |
| write | `set(int index, E element)` | `[index]` |
| add | `add(E element)` | — |
| | `add(int index, E element)` | — |
| remove | `remove(int index)` | — |
| contains | `contains(Object o)` | — |

# Practice

Write a declaration for an array variable people that could be used to refer to an array of Person objects.

Write a declaration for an array variable vacant that could be used to refer to an array of boolean values.

What is wrong with the following array declarations? Correct them.

```
[]int counts;
boolean[5000] occupied;
```

# Creating an array object

```
public class LogAnalyzer
{
    private int[] hourCounts;
    private LogfileReader reader;

    public LogAnalyzer()
    {
        hourCounts = new int[24];
        reader = new LogfileReader();
    }
    ...
}
```

Array type
– does not contain size

Array object creation
– specifies size

# Using an array

Square-bracket notation is used to access an array element:

- **`hourCounts[...]`**

Elements are used like ordinary variables.

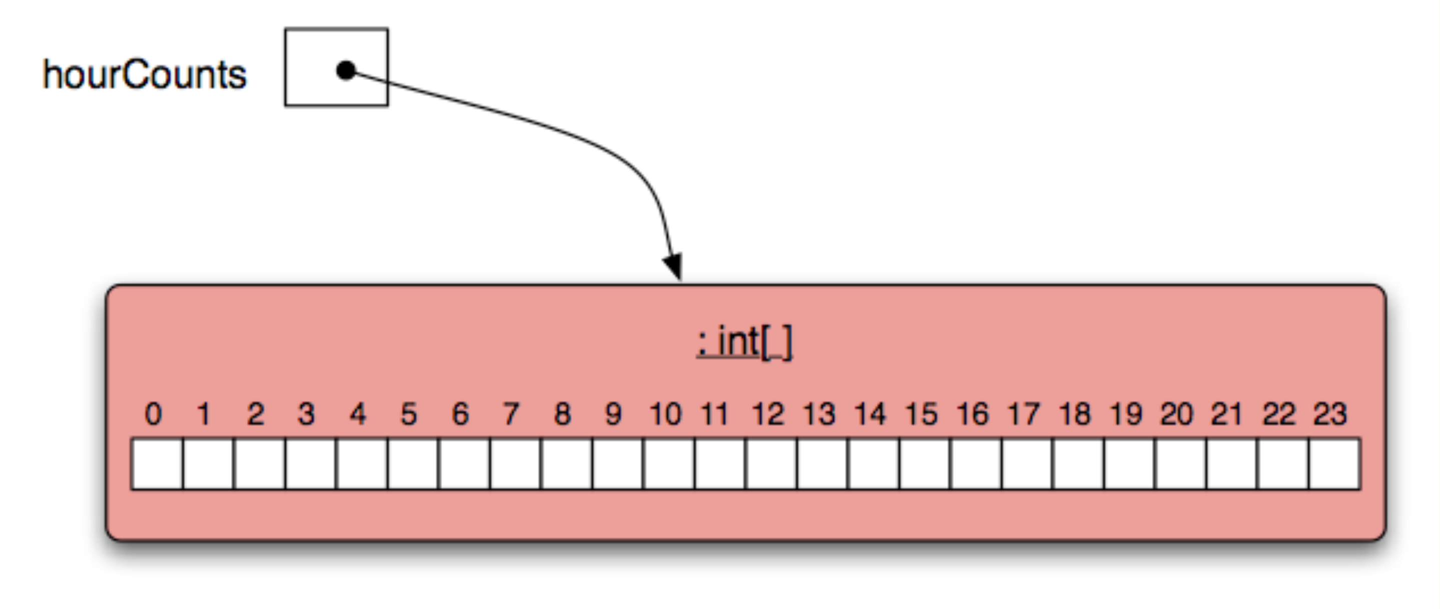
The target of an assignment:

```
hourCounts[hour] = ...;
```

In an expression:

```
hourCounts[hour]++;
if(hourCounts[hour] > 0) ...
```

# The **hourCounts** array



# Array iteration

field
(not method!)

for loop version

```
for(int hour = 0; hour < hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

while loop version

```
int hour = 0;
while(hour < hourCounts.length) {
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

# Practice

Given an array of numbers, print out all the numbers in the array, using a for loop.

```
int[] numbers = { 4, 1, 22, 9, 14, 3, 9};

for ...
```

# for loop with bigger step

```
// Print multiples of 3 that are below 40.
for(int num = 3; num < 40; num = num + 3) {
    System.out.println(num);
}
```

# Array-related methods

**`System`** has static **`arraycopy`**.

**`java.util.Arrays`** contains static utility methods for processing arrays:

- **`binarySearch`**
- **`fill`**
- **`sort`**

**`ArrayList`** has **`toArray`**.

# for loop and **Iterator**

No post-body action required.

```
for(Iterator<Track> it = tracks.iterator(); it.hasNext(); ) {
    Track track = it.next();
    if(track.getArtist().equals(artist)) {
        it.remove();
    }
}
```

# Review

Arrays are appropriate where a fixed-size collection is required.

Arrays use a special syntax.

For loops are used when an index variable is required.

For loops offer an alternative to while loops when the number of repetitions is known.

Used with a regular step size.

# Practice: array

Implement the sum method in ForLoop.

```
----------------------public class ForLoop
{
    /**
     * Your assignment is to implement this method:
     * Sum up every element in the array passed as
     * an argument and return the sum
     *
     * @param  a   an array of integers
     * @return     the sum of of all ints in a
     */
    public int sum(int[] a)
    {
        return 0;
    }
}
```

# Practice: Array

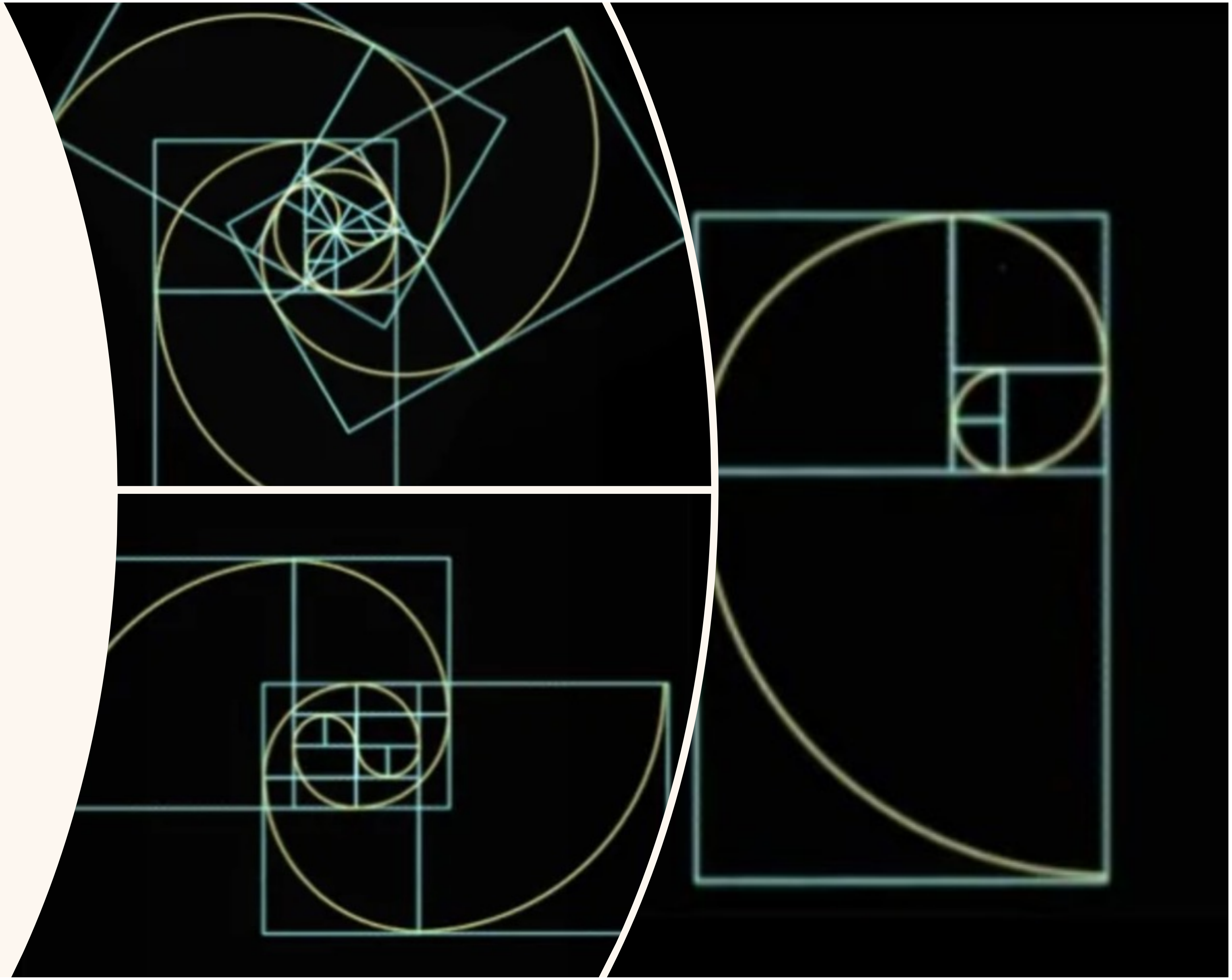
Write a method according to this documentation comment

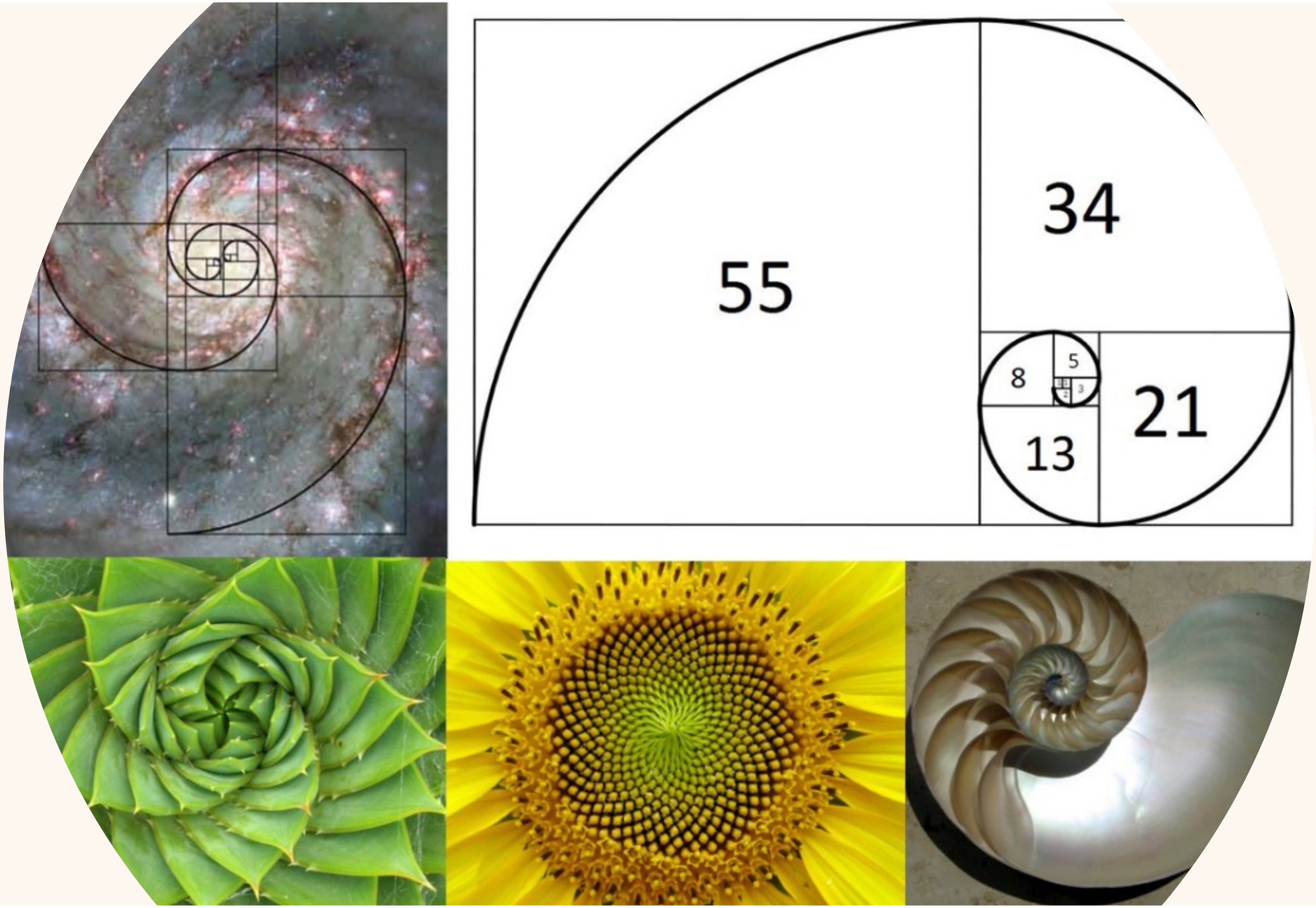
```
/**
 * This method searches for a specific value in an integer array and returns its index.
 * If the value is found, it returns the index of the first occurrence of the value.
 * If the value is not found, it returns -1.
 *
 * @param array The array in which to search for the value.
 * @param value The value to search for in the array.
 * @return The index of the first occurrence of the value in the array, or -1 if the value is not found.
 */
```

# Emergence of spatio-temporal patterns

# Fibonacci

Nature, Mathematics, Art




55
34
21
13
8
5
3
2
1
1

# Practice

Erstellen wir ein array, in dem die Fibonacci-Folge enthalten ist. Diese geht so, dass das erste Element den Wert 0 hat und das zweite Element den Wert 1 und alle nachfolgenden Elemente als Wert die Summe der Werte ihrer beiden Vorgänger hat

```
0 1 1 2 3 5 8 13 21 34 ...
```

```
int howMany = 20;
int[] fib = new int[howMany];

fib[0] = 0;
fib[1] = 1;

for(...) ...
```

# Cellular Automata

“Zelluläre oder auch zellulare Automaten dienen der Modellierung **räumlich diskreter dynamischer** Systeme.

Sie bestehen aus einzelnen Zellen, die zu **diskreten Zeitpunkten** gleichzeitig ihren Zustand ändern.

Die Änderung erfolgt **für alle Zellen nach den gleichen Regeln**.

Sie hängt von den Zellzuständen in einer **vorgegebenen Nachbarschaft** und vom Zustand der Zelle selbst ab.”

https://de.wikipedia.org/wiki/Zellul%C3%A4rer_Automat

# The *automaton* project

An array of ‘cells’.

Each cell maintains a simple state.

- Usually a small numerical value.
- E.g., on/off or alive/dead.

The states change according to simple rules.

Changes affected by neighboring states.

# A simple automaton

```
nextState[i] =
   (state[i-1] + state[i] + state[i+1]) % 2;
```

| Step | Cell states – blank cells are in state 0 | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | + | | | | | | |
| 1 | | | | | | + | + | + | | | | | |
| 2 | | | | | + | | + | | + | | | | |
| 3 | | | | + | + | | + | | + | + | | | |
| 4 | | | + | | | | + | | | | + | | |
| 5 | | + | + | + | | + | + | + | | + | + | + | |
| 6 | + | | + | | | | + | | | | + | | + |

# The conditional operator (ternary operator )

Choose between two values:

*condition* ? *value1* : *value1*

```
for(int cellValue : state) {
    System.out.print(cellValue == 1 ? '+' : ' ');
}
System.out.println();
```

# Multidimensional Arrays

Folge 330 – Manfred Salmhofer über Emergenz

https://www.youtube.com/watch?v=T9Dn1_TUy6E

# Emergenz

# Arrays of more than one dimension

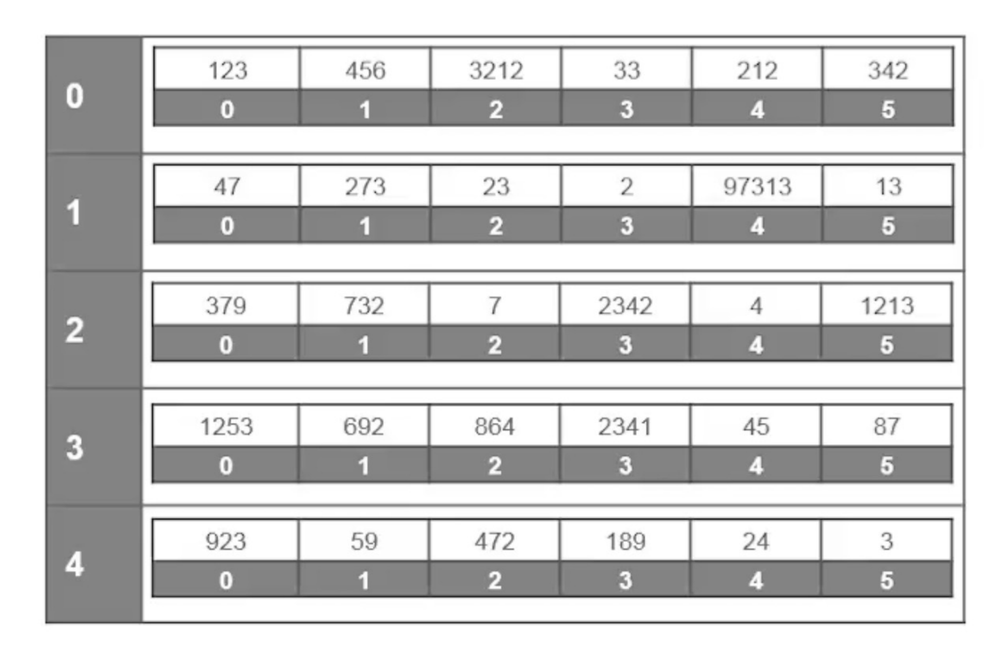
Array syntax supports multiple dimensions.

- E.g., 2D array to represent a game board, or a grid of cells.

Can be thought of as an array of arrays.

Synonyme:

**Array**, Tupel, Matrix, Vektor, Tabelle



| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 123 | 456 | 3212 | 33 | 212 | 342 |
| 1 | 47 | 273 | 23 | 2 | 97313 | 13 |
| 2 | 379 | 732 | 7 | 2342 | 4 | 1213 |
| 3 | 1253 | 692 | 864 | 2341 | 45 | 87 |
| 4 | 923 | 59 | 472 | 189 | 24 | 3 |

# Beispiele

Slido: Welche Beispiele fallen Ihnen für 2D Arrays ein?

# Arrays of more than one dimension

Array syntax supports multiple dimensions.

- E.g., 2D array to represent a game board, or a grid of cells.

Can be thought of as an array of arrays.

Synonyme:

Array, Tupel, Matrix, Vektor, Tabelle

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 2 | -2 | 67 | 2 | 90 |
| 1 | 33 | 3 | -6 | 5 | 2 |
| 2 | 4 | 2 | 2 | 78 | 93 |

- 3 x 5 - Matrix (2-dimensional)
- Elementtyp: **`int`**
- 3 Zeilen, 5 Spalten
- Numerierung beginnt bei 0 (!!)

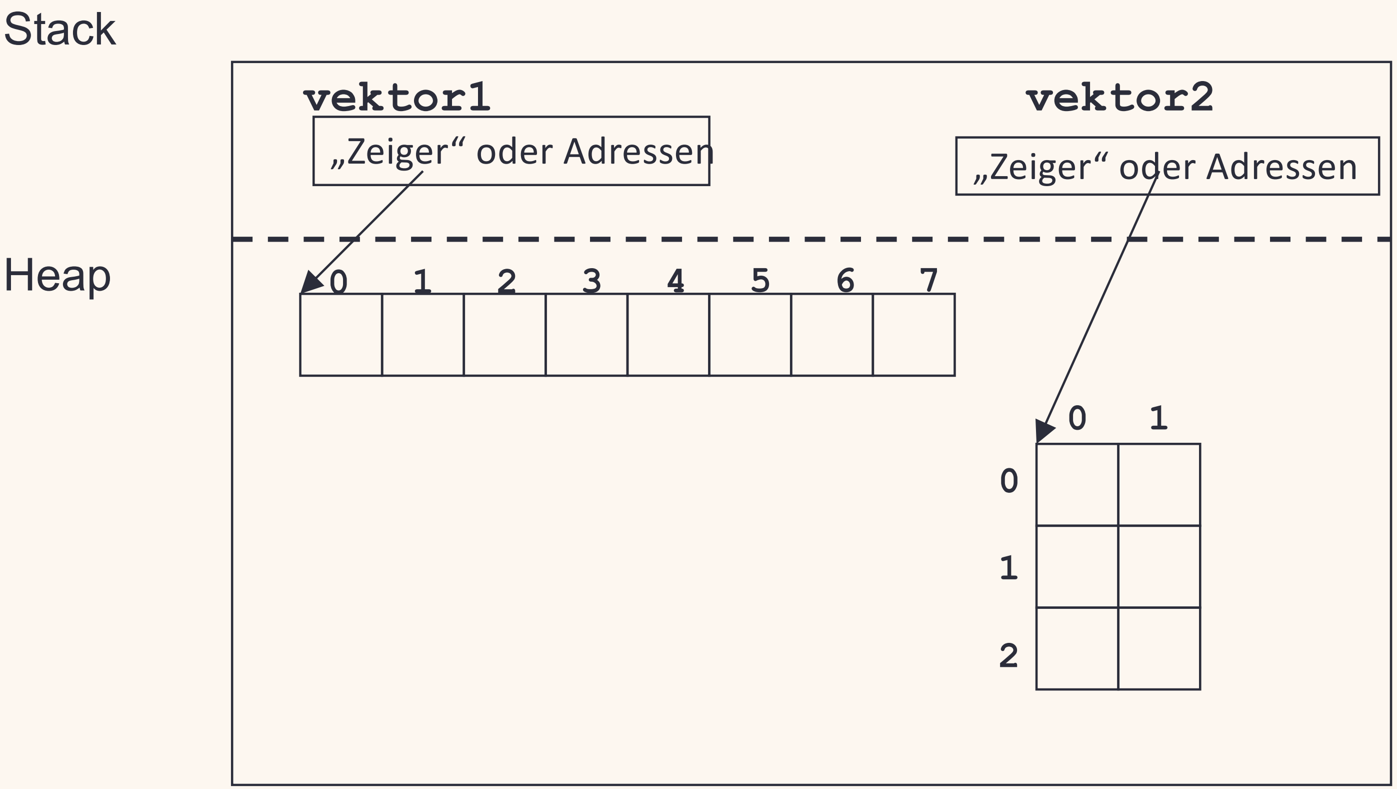
```
m[0]
m[0][2] == 67
m[5][5] → Laufzeitfehler
```

# Practice

```
String[][] arr = { {"hello","there","world"},{"how","are","you"} };

        System.out.println("Rows:");
        // ADD CODE TO PRINT NUMBER OF ROWS HERE //
           System.out.println("Rows:" + arr.length);

        System.out.println("Columns:");
        // ADD CODE TO PRINT NUMBER OF COLUMNS HERE //
            System.out.println("Columns:" + arr[0].length);
```

Schema:

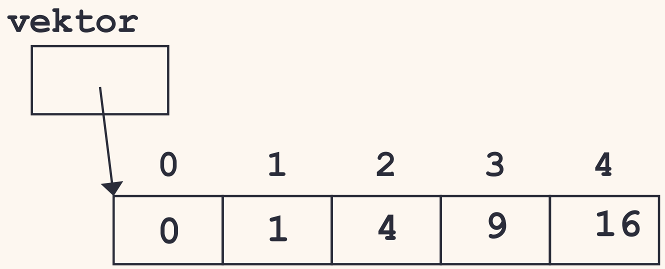
# Felder / Erzeugung eines Feldes



# Felder / Initialisierung der Feldelemente

Implizite Erzeugung und Initialisierung:

```
int i = 3;
int[] vektor = {0, 1, 4, i*i, 16};
```

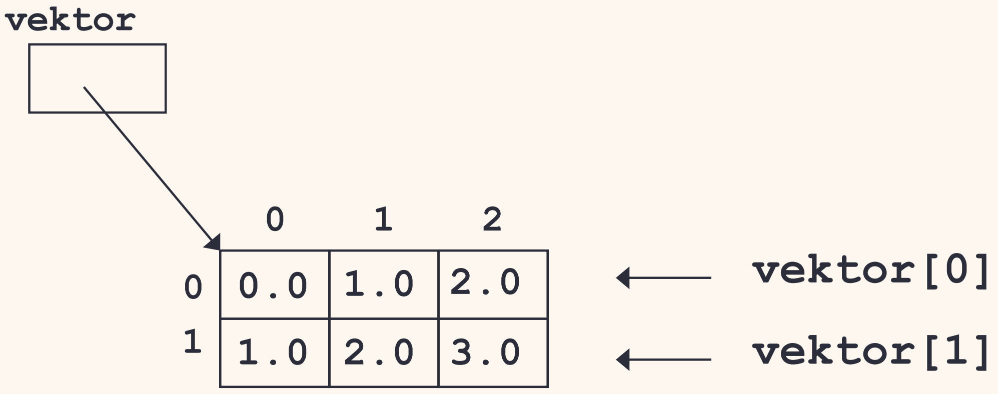


- erzeugt Feld mit entsprechender Elementanzahl
- initialisiert die Elemente (Reihenfolge!)
- Initialisierungswerte können durch Ausdrücke des entsprechenden Elementtyps gebildet werden (i.a. Literale)

# Felder / Mehrdimensionale Felder

Normalfall: Anzahl an Elementen pro Dimension ist identisch

```
  float[][] vektor = new float[2][3];
  for (int z=0; z < vektor.length; z++){
    for (int s=0; s < vektor[z].length; s++){
      vektor[z][s] = z + s;
     }
}
```



|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0.0 | 1.0 | 2.0 |
| 1 | 1.0 | 2.0 | 3.0 |

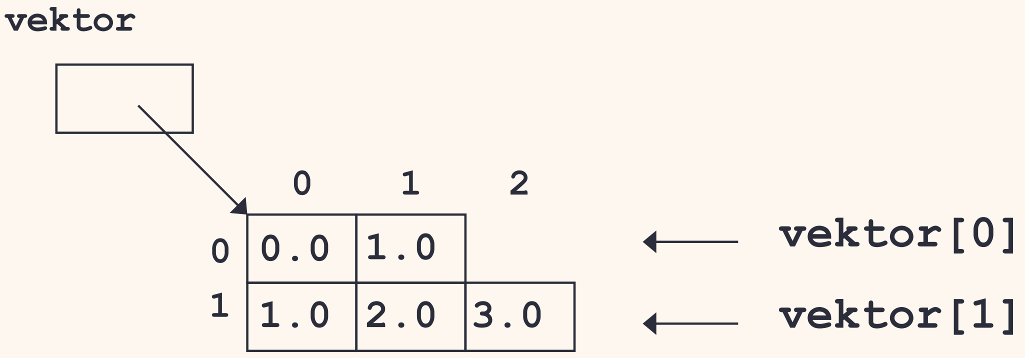
# Mehrdimensionale arrays /Felder

Möglich: Anzahl an Elementen pro Dimension ist unterschiedlich

```
float[][] vektor = new float[2][];
vektor[0] = new float[2];
vektor[1] = new float[3];
```

oder implizite Erzeugung und Initialisierung:

```
float[][] vektor = { {0.0, 1.0}, {1.0, 2.0, 3.0}}
```



vektor

|   | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0.0 | 1.0 | |
| 1 | 1.0 | 2.0 | 3.0 |

← vektor[0]

← vektor[1]

## Felder / Beispiel 2

Summe aller Elemente in einer int X int Matrix:

```
public int sum(int[][] feld){
  int s = 0;
  for (int y=0; y<feld.length; y++){
    for (int x=0; x<feld[y].length; x++){
      s=s+feld[y][x];
    }
  }
  return s;
}
```

# The *brain* project

```
Cell[][] cells;
...
cells = new Cell[numRows][numCols];
...
for(int row = 0; row < numRows; row++) {
    for(int col = 0; col < numCols; col++) {
        cells[row][col] = new Cell();
    }
}
```

# Alternative iteration

‘Array of array’ style.

Requires no access to **`numRows`** and **`numCols`**.

Works with irregular shape arrays, which are supported in Java.

```
for(int row = 0; row < cells.length; row++) {
    Cell[] nextRow = cells[row];
    for(int col = 0; col < nextRow.length; col++) {
        nextRow[col] = new Cell();
    }
}
```

# Practice

Given a 2D array board and a cell board [i][j].

List all of the cells that are its neighbors.

# Practice

Given a 2D array board and a cell board [i][j].

List all of the cells that are its neighbors.

| | | |
|---|---|---|
| [i-1][j-1] | [i-1][j] | [i-1][j+1] |
| [i][j-1] | [i][j] | [i][j+1] |
| [i+1][j-1] | [i+1][j] | [i+1][j+1] |

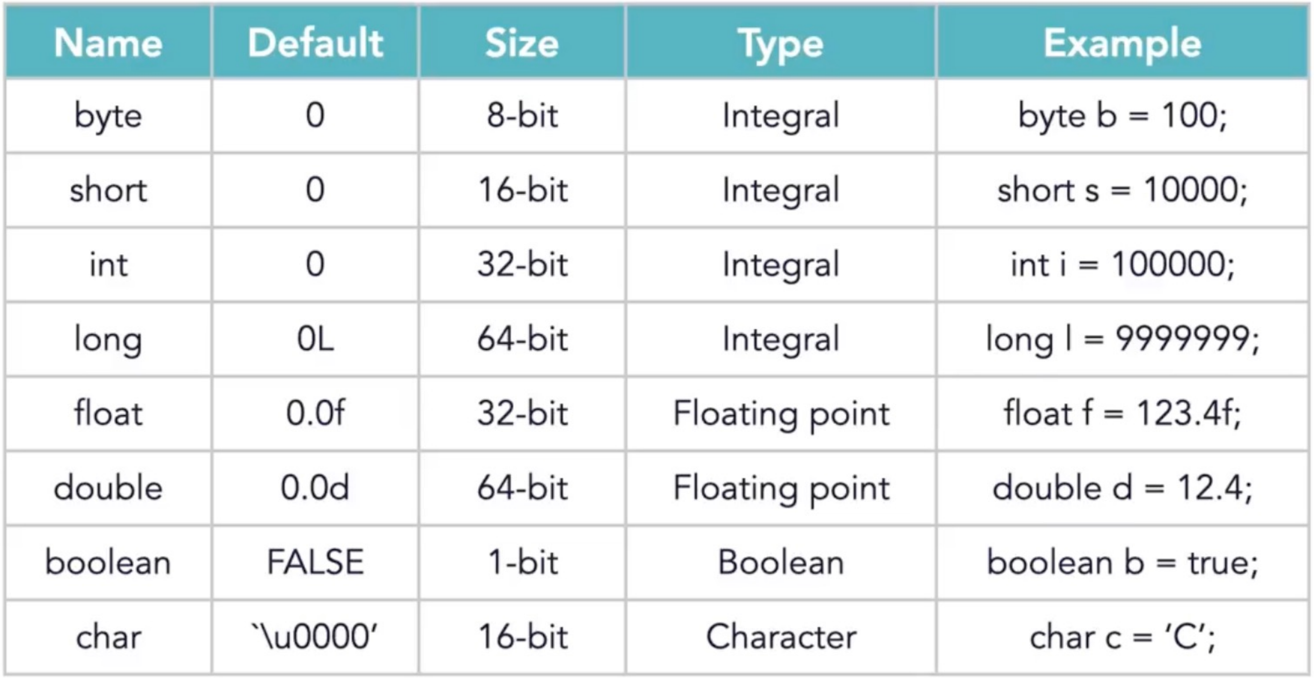
# Moore Neighbourhood

- A cell that is alive changes its state to dying
- A cell that is dying changes its state to dead
- A cell that is dead changes its state to alive if exactly two of its neighbours are alive, otherwise it remains dead

# Read about Java primitive types

https://www.linkedin.com/learning-login/share?account=82533698&forceAccount=false&redirect=https%3A%2F%2Fwww.linkedin.com%2Flearning%2Fjava-essential-training-syntax-and-structure-16025610%2Fprimitive-data-types%3Ftrk%3Dshare_video_url%26shareId%3DuUcKDe%252BFSYCuqDzSe4AvKw%253D%253D

https://freiheit.f4.htw-berlin.de/prog1/variablen/



| Name | Default | Size | Type | Example |
|---|---|---|---|---|
| byte | 0 | 8-bit | Integral | byte b = 100; |
| short | 0 | 16-bit | Integral | short s = 10000; |
| int | 0 | 32-bit | Integral | int i = 100000; |
| long | 0L | 64-bit | Integral | long l = 9999999; |
| float | 0.0f | 32-bit | Floating point | float f = 123.4f; |
| double | 0.0d | 64-bit | Floating point | double d = 12.4; |
| boolean | FALSE | 1-bit | Boolean | boolean b = true; |
| char | `\u0000’ | 16-bit | Character | char c = ‘C’; |

# Clean Code

# Welche Konventionen sollten wir beim Programmieren beachten?

Do not edit

How to change the design

ⓘ The Slido app must be installed on every computer you're presenting from

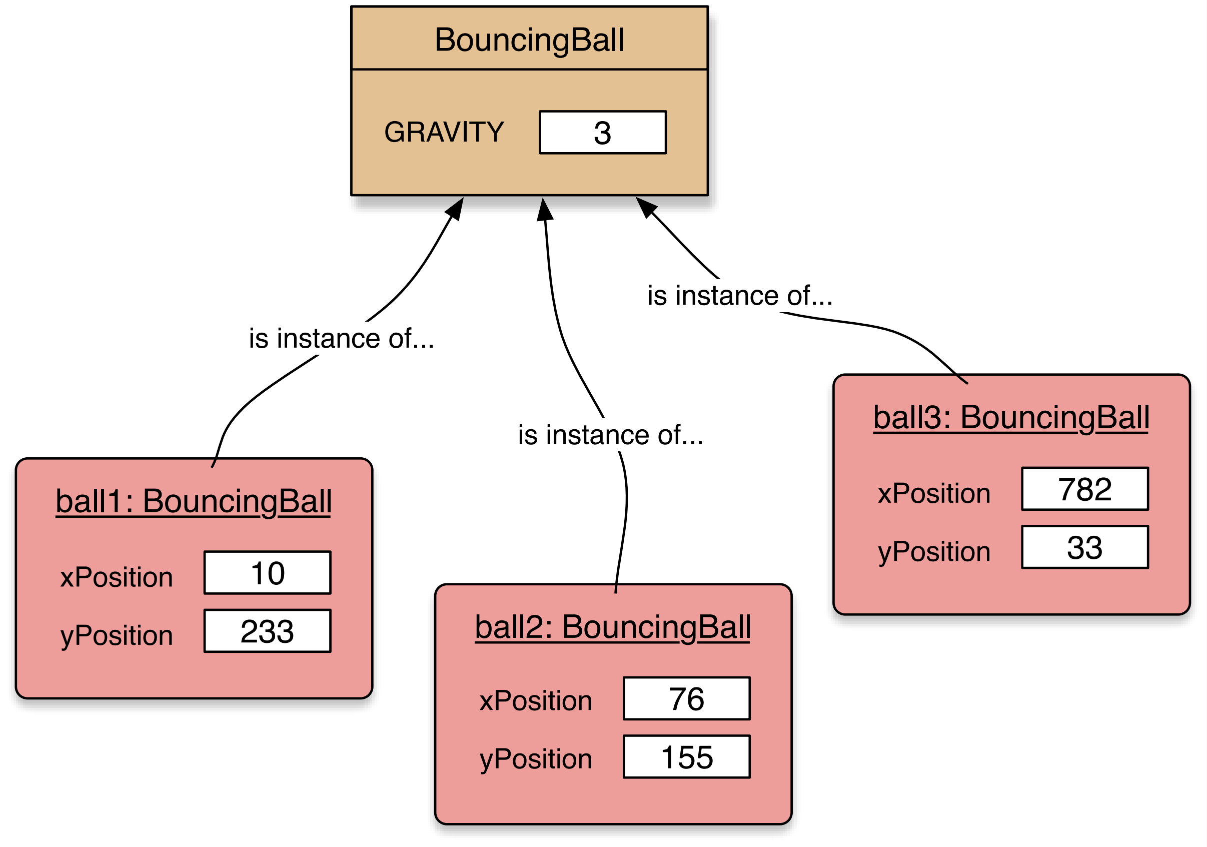
slido

# Class variables and constants

# Class variables

- A class variable is shared between all instances of the class.
- In fact, it belongs to the class and exists independent of any instances.
- Designated by the **`static`** keyword.
- Public static variables are accessed via the class name; e.g.:
  - **`Thermometer.boilingPoint`**

# Class variables



# Constants

- A variable, once set, can have its value fixed.
- Designated by the **`final`** keyword.
  - **`final int SIZE = 10;`**
- Final *fields* must be set in their declaration or the constructor.
- Combing **`static`** and **`final`** is common.

# Class constants

**`static`**: class variable

**`final`**: constant

```
private static final int gravity = 3;
```

Public visibility is less of an issue with **`final`** fields.

Upper-case names often used for class constants:

```
public static final int BOILING_POINT = 100;
```

System.out.println(“whatever”);

## Class methods

A **`static`** method belongs to its class rather than the instances:

```
public static int getDaysThisMonth()
```

Static methods are invoked via their class name:

```
int days = Calendar.getDaysThisMonth();
```

# Limitations of class methods

A static method exists independent of any instances.

Therefore:

- They cannot access instance fields within their class.
- They cannot call instance methods within their class.

# Review

Class variables belong to their class rather than its instances.

Class methods belong to their class rather than its instances.

Class variables are used to share data among instances.

Class methods are prohibited from accessing instance variables and methods.

# Review

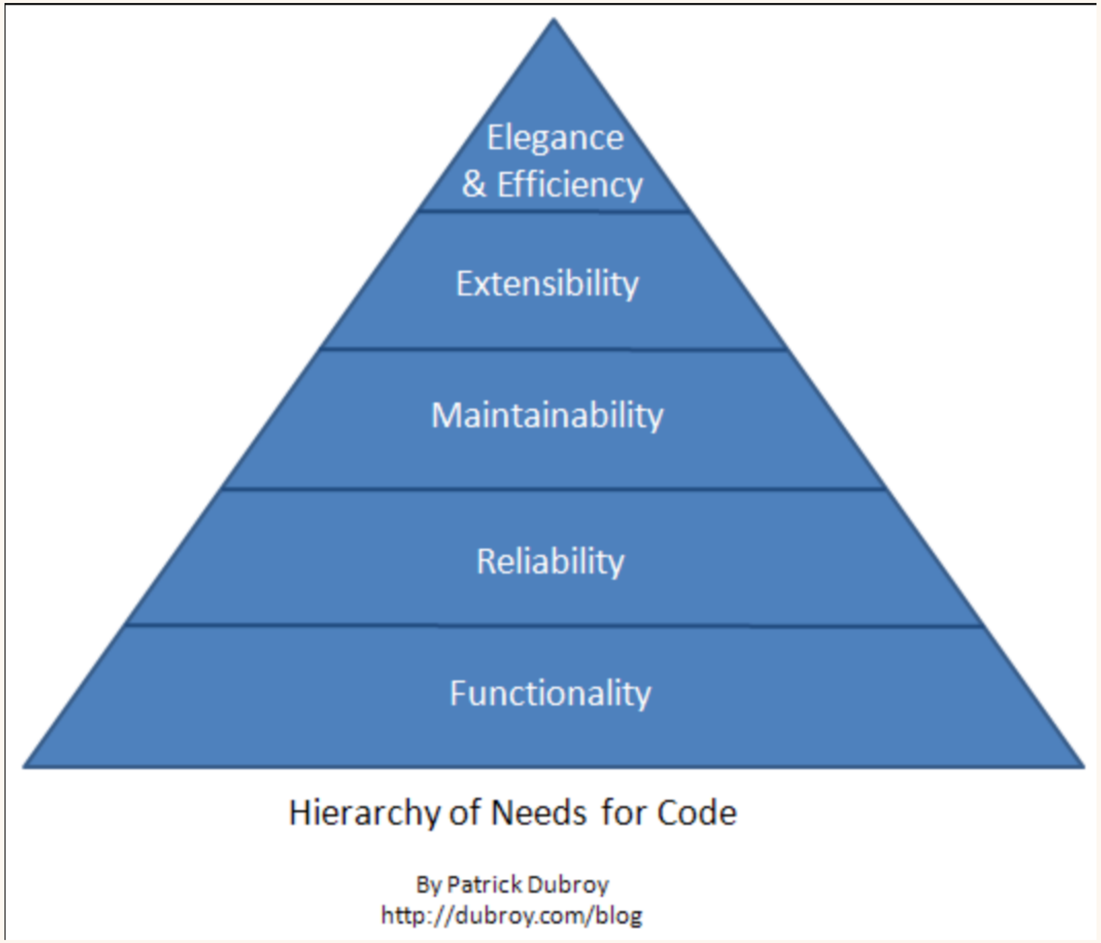
The values of **`final`** variables are fixed.

They must be assigned at declaration or in the constructor (for fields).

**`final`** and **`static`** are unrelated concepts, but they are often used together.

Priorities in Design

## Considering Hierarchy of Needs in Coding



https://dubroy.com/blog/a-hierarchy-of-needs-for-code/