Chapter 3.3: Arrays

Object Oriented Programming with Java

Arrays and indices | Declaration and initialization | References and memory

Mean, median, mode | Searching and matching | Selection sort

Multidimensional arrays | Matrix transpose | Tic Tac Toe

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_03_03_Arrays.html

Learning Outcomes

Concept

Explain that an array holds a fixed number of elements of one type in consecutive cells, accessed by an **index** starting at 0.

Syntax

Declare one- and multidimensional arrays, create them with `new` or an initializer list, and traverse them with `length`.

Memory

Predict the effect of assignment, copying and passing to a method, knowing that an array variable holds a **reference** and that the array itself lives on the heap.

Errors

`ArrayIndexOutOfBoundsException` and common compile errors: recognize them and prevent them with bounds checks.

Algorithms

Write mean, median, mode, linear and binary search, matching, reversal, selection and bubble sort; compare O(1), O(n) and $O(n^2)$ costs.

Practice

Fill, print and transpose matrices with nested loops, and build Tic Tac Toe with `char[][]`.

THE ARRAY CONCEPT

What Is an Array?

- **Array**: **a structure that stores data of the same type** in an orderly way under a single name.
- Think of it as one variable that can hold one or more values.
- Elements are accessed by **index** number; indexing makes access very fast.
- **The first index is 0**; in an array of n elements the last index is **n − 1**.
- To access an element, write the array name followed by the index in square brackets: `arr[0]`.

Apartment flats

One address (the array name) and numbered flats (the indices). Saying "number 3" is enough to find the flat.

Egg carton

The number of cells is fixed up front; the carton never grows. Each cell holds the same kind of thing.

Example: `arr = [10, 20, 30, 40, 50]`, so `arr[0]` is the first element, with value **10**; `arr[4]` is the last (50).

Why Arrays?

WITHOUT ARRAYS: 50 VARIABLES FOR 50 STUDENTS

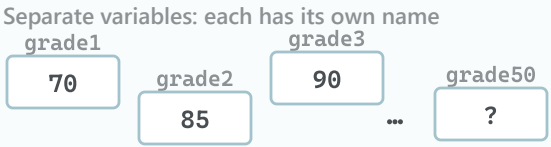
```
int grade1 = 70;
int grade2 = 85;
int grade3 = 90;
// ... up to grade50
int sum = grade1 + grade2 + grade3 /* + ... */;
```

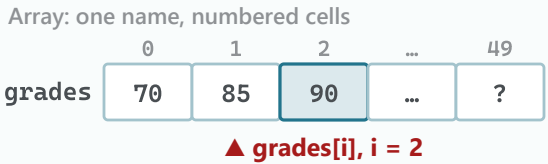
Computing the mean means typing all 50 names; if the class size changes, so does the code.

WITH AN ARRAY: ONE NAME, A LOOP DOES THE WORK

```
int[] grades = new int[50];
// ... fill grades
int sum = 0;
for (int i = 0; i < grades.length; i++) {
    sum += grades[i];
}
```

The same code works for 5 grades or 5000: **the index can be a variable** (`grades[i]`), so the loop visits every element.





Here is the power of arrays: in `grades[i]` the cell to read is decided **at run time by the value of `i`**.

An Array in Memory

The figure shows a one-dimensional array of five elements. Each element sits in a **cell**; the cells are laid out **side by side (contiguously)** in memory.



index	0	1	2	3	4
element	13	−1	4	2	7

`a[1]` = −1, `a[4]` = 7, `a.length` = 5. `a[5]`: there is **no such cell**; valid indices are 0…4.

Pros and Cons

Advantages

+ Stores data **in order, contiguously**.
+ Indexing gives **fast access** to elements: `arr[i]` takes the same time for every i.
+ Groups related data together; easy to process with loops.
+ Lean memory use: only the elements and the length are stored.

Disadvantages

– **Fixed size**: once created, an array cannot be resized.
– To add an element when it is full, you must create a **larger new array** and copy.
– Inserting into or deleting from the middle requires **shifting** elements (extra cost).
– Holds elements of one type only.

For lists that change size, Java offers `ArrayList`; it uses an array internally and grows it when full (`java.util` collections).



Kinds of Arrays and Their Uses

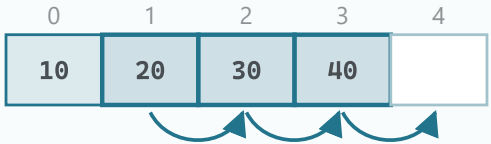
One-dimensional
Elements are stored in a single row.
`int[] grades`

Two-dimensional
Elements are stored in rows and columns (a matrix).
`int[][] table`

Multidimensional
Supports storing and accessing data with more structure.
`int[][][] cube`

USES

- The most common data structure in programming; ideal for organizing and processing data.
- **Measurements**: samples read from a sensor every second, an audio signal (`double[]`).
- **Matrices**: circuit analysis, systems of linear equations.
- **Images**: a grid of pixels (`int[height][width]`).



Inserting 15 at index 1: first 40, 30, 20 each shift right

Declaring Arrays

```
Type[] arrayName = new Type[arraySize];
```

Type

The type of the values stored: `int`, `double`, `char`, `String`…

arrayName

The array's name; follows the usual naming rules.

arraySize

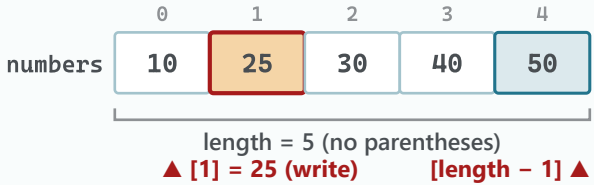
How many elements the array holds: a constant or a run-time `int`.

```
int[] numbers = new int[5];       // 5 ints
double[] readings = new double[100]; // 100 doubles
String[] names = new String[30];
int n = 12;
char[] letters = new char[n];     // size from a variable
int arr2[] = new int[3];          // C style: legal, not recommended
```

Type	Default value
`int`, `long`…	`0`
`double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (code 0)
`String` and objects	`null`

Access and length

- Array elements are accessed by **index**; the index starts at 0.
- `arr[i]` is used both for **reading** and **writing**.
- The size of an array comes from the `length` **field**: the number of elements it holds.
- `length` is not a method: **no parentheses** (`String`'s `length()` is a method).
- In a non-empty array the last element is `arr[arr.length - 1]`.




```java
int[] numbers = {10, 20, 30, 40, 50};
int first = numbers[0];    // first element
int second = numbers[1]; // second element
int size = numbers.length; // 5
System.out.println(first + " " + second
        + " " + size);
numbers[1] = 25;                // write
int last = numbers[numbers.length - 1]; // 50
```

OUTPUT

```
10 20 5
```

Initializing Arrays

- When an array is declared, its elements can be given **initial values** with an **initializer list** (array initializer).
- The size is then **not given**; the initializers determine it: 5 values make a 5-element array.

```
int[] numbers = {10, 20, 30, 40, 50};      // 5 elements
int[] n = {1, 2, 3, 4, 5};   // 5 values, so n has 5 elements
int[] m = new int[]{1, 2, 3}; // long form: works anywhere
String[] days = {"Mon", "Tue", "Wed"};
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
```

Short form only in a declaration

You cannot later write `a = {1, 2, 3};` (`illegal start of expression`). Write `a = new int[]{1, 2, 3};` instead.

Not both at once

`new int[3]{1, 2, 3}` does not compile: give either a size or a list.

Example: ArrayExample

```java
public class ArrayExample {
    public static void main(String[] args) {
        // declare an array of 5 ints
        int[] numbers = new int[5];
        // assign values to the elements
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;
        // access elements
        System.out.println("first element: " + numbers[0]);
        System.out.println("third element: " + numbers[2]);
        // print all elements in a loop
        System.out.print("Elements: ");
        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i] + " ");
        }
    }
}
```

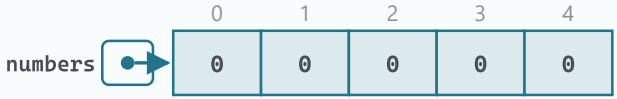
OUTPUT

```
first element: 10
third element: 30
Elements: 10 20 30 40 50
```

Loop pattern

`for (int i = 0; i < arr.length; i++)` visits **every** index of the array, from 0 to length − 1. `<=` in its place is a common mistake.

`ArrayExample`: Trace the Code Step by Step



`new int[5]` creates a 5-cell array on the heap; every cell starts at the **default value 0**. `numbers` holds this array's **reference**.

Step 1 / 15

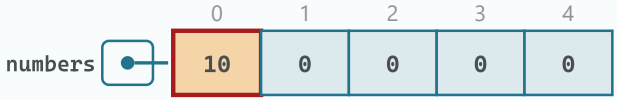
```
▶ int[] numbers = new int[5];
2 numbers[0] = 10;
3 numbers[1] = 20;
4 numbers[2] = 30;
5 numbers[3] = 40;
6 numbers[4] = 50;
7 System.out.println("first element: " + numbers[0]);
8 System.out.println("third element: " + numbers[2]);
9 System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [0, 0, 0, 0, 0]`

OUTPUT

`ArrayExample`: Trace the Code Step by Step



`numbers[0] = 10;` → writes cell 1 (index 0).

Step 2 / 15

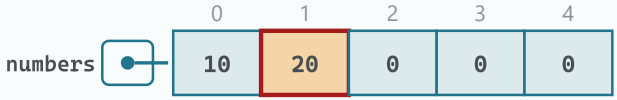
```
1  int[] numbers = new int[5];
▶  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 0, 0, 0, 0]`

OUTPUT

`ArrayExample`: Trace the Code Step by Step



`numbers[1] = 20;` → writes cell 2 (index 1).

Step 3 / 15

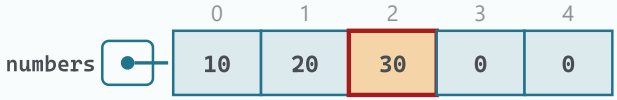
```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
▶ 3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 0, 0, 0]`

OUTPUT

`ArrayExample`: Trace the Code Step by Step



`numbers[2] = 30;` → writes cell 3 (index 2).

Step 4 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
▶  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 30, 0, 0]`

OUTPUT

`ArrayExample`: Trace the Code Step by Step



`numbers[3] = 40;` → writes cell 4 (index 3).

Step 5 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
▶  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 30, 40, 0]`

OUTPUT

`ArrayExample`: Trace the Code Step by Step



`numbers[4] = 50;` → writes cell 5 (index 4).

Step 6 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
▶  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 30, 40, 50]`

OUTPUT

`ArrayExample`: Trace the Code Step by Step



The first element has index **0**: `numbers[0]` is read → 10.

Step 7 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
▶  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

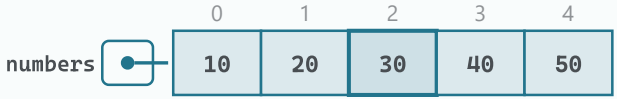
VARIABLES

`numbers = [10, 20, 30, 40, 50]`

OUTPUT

```
first element: 10
```

`ArrayExample`: Trace the Code Step by Step



The third element is `numbers[2]` (index = position − 1) → 30.

Step 8 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
▶  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 30, 40, 50]`

OUTPUT

```
first element: 10
third element: 30
```

`ArrayExample`: Trace the Code Step by Step



`print` does not end the line; the elements go on one line.

Step 9 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
▶ 9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

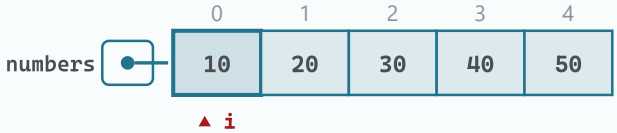
VARIABLES

`numbers = [10, 20, 30, 40, 50]`

OUTPUT

```
first element: 10
third element: 30
Elements:
```

`ArrayExample`: Trace the Code Step by Step



`i = 0` : `0 < 5` true → `numbers[0]` = 10 is printed.

Step 10 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
▶      System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

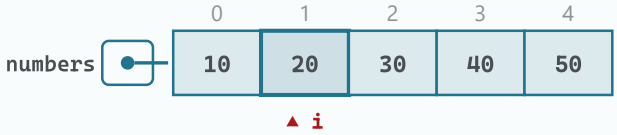
`numbers = [10, 20, 30, 40, 50]`

`i = 0`

OUTPUT

```
first element: 10
third element: 30
Elements: 10
```

`ArrayExample`: Trace the Code Step by Step



`i = 1` : `1 < 5` true → `numbers[1]` = 20 is printed.

Step 11 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
▶      System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

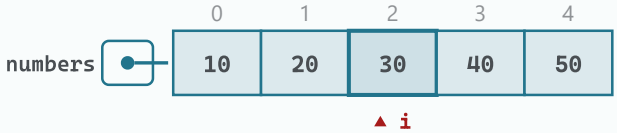
`numbers = [10, 20, 30, 40, 50]`

`i = 1`

OUTPUT

```
first element: 10
third element: 30
Elements: 10 20
```

`ArrayExample`: Trace the Code Step by Step



`i = 2` : `2 < 5` true → `numbers[2]` = 30 is printed.

Step 12 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
▶      System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

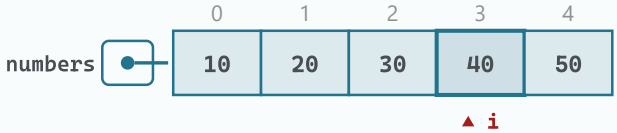
`numbers = [10, 20, 30, 40, 50]`

`i = 2`

OUTPUT

```
first element: 10
third element: 30
Elements: 10 20 30
```

`ArrayExample`: Trace the Code Step by Step



`i = 3` : `3 < 5` true → `numbers[3]` = 40 is printed.

Step 13 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
▶      System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

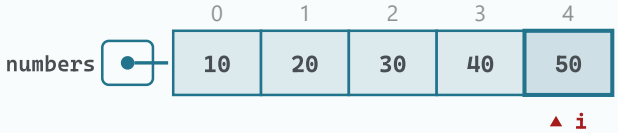
`numbers = [10, 20, 30, 40, 50]`

`i = 3`

OUTPUT

```
first element: 10
third element: 30
Elements: 10 20 30 40
```

`ArrayExample`: Trace the Code Step by Step



`i = 4` : `4 < 5` true → `numbers[4]` = 50 is printed.

Step 14 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
10 for (int i = 0; i < numbers.length; i++) {
▶      System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 30, 40, 50]`

`i = 4`

OUTPUT

```
first element: 10
third element: 30
Elements: 10 20 30 40 50
```

`ArrayExample`: Trace the Code Step by Step



`i = 5` : `5 < numbers.length` is false → the loop ends. The last valid index is **length − 1 = 4**.

Step 15 / 15

```
1  int[] numbers = new int[5];
2  numbers[0] = 10;
3  numbers[1] = 20;
4  numbers[2] = 30;
5  numbers[3] = 40;
6  numbers[4] = 50;
7  System.out.println("first element: " + numbers[0]);
8  System.out.println("third element: " + numbers[2]);
9  System.out.print("Elements: ");
▶  for (int i = 0; i < numbers.length; i++) {
11     System.out.print(numbers[i] + " ");
12 }
```

VARIABLES

`numbers = [10, 20, 30, 40, 50]`

`i = 5`

OUTPUT

```
first element: 10
third element: 30
Elements: 10 20 30 40 50
```

The for-each Loop

```java
int[] grades = {70, 85, 90};
// classic for: we have the index
for (int i = 0; i < grades.length; i++) {
    System.out.println("grade " + i + ": " + grades[i]);
}
// for-each: values only
int sum = 0;
for (int g : grades) {
    sum += g;
}
System.out.println("Sum: " + sum);
```

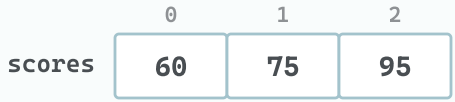
OUTPUT

```
grade 0: 70
grade 1: 85
grade 2: 90
Sum: 245
```

- **Enhanced for (for-each)**: `for (type item : arr)` puts the next element into `item` on each pass.
- No index is written, so it cannot go out of bounds.
- Only for **reading**: `g = 0;` changes only the copy, not the element in the array.
- If you need the index (to write, report a position, compare neighbors), use a classic `for`.

for-each: You Get the Value, Not the Cell

```
▶ int[] scores = {60, 75, 95};
2 int sum = 0;
3 for (int p : scores) {
4     sum += p;
5 }
6 System.out.println("Sum: " + sum);
7 for (int p : scores) {
8     p = 0;
9 }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



A three-cell array lives on the heap; `scores` holds its reference.

Step 1 / 12

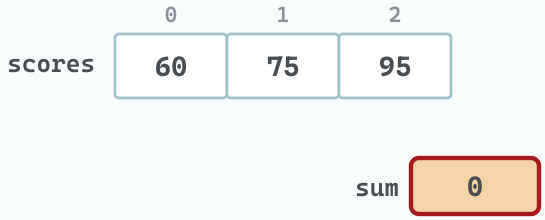
VARIABLES

`scores = [60, 75, 95]`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
▶  int sum = 0;
3  for (int p : scores) {
4      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



`sum` starts at 0.

Step 2 / 12

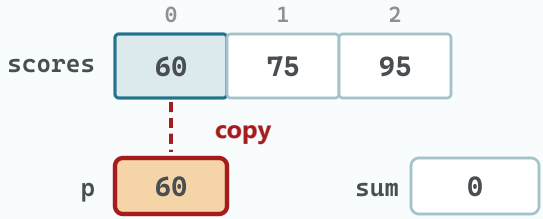
VARIABLES

`scores = [60, 75, 95]` `sum = 0`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
▶ 3  for (int p : scores) {
4      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



First pass: the loop takes `scores[0]` and puts a **copy of it into** `p`, so p = 60. No index is written.

Step 3 / 12

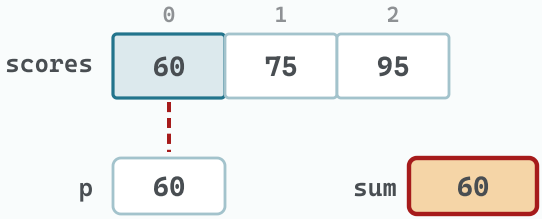
VARIABLES

`scores = [60, 75, 95]` `sum = 0` `p = 60`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
3  for (int p : scores) {
▶      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



`sum += p` → 60.

Step 4 / 12

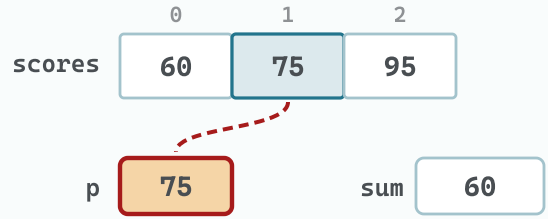
VARIABLES

`scores = [60, 75, 95]` `sum = 60` `p = 60`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
▶ 3  for (int p : scores) {
4      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



Second pass: p = 75 (the loop supplies the next element itself).

Step 5 / 12

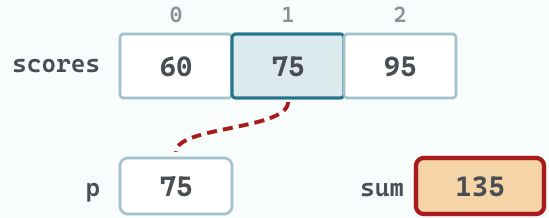
VARIABLES

`scores = [60, 75, 95]` `sum = 60` `p = 75`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
3  for (int p : scores) {
▶      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



sum = 60 + 75 = 135.

Step 6 / 12

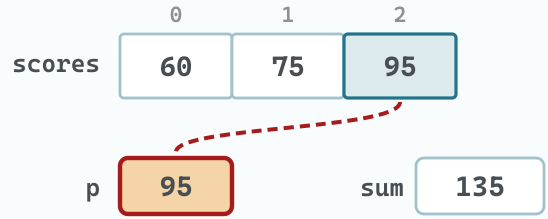
VARIABLES

`scores = [60, 75, 95]` `sum = 135` `p = 75`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
▶ 3  for (int p : scores) {
4      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



Last pass: p = 95.

Step 7 / 12

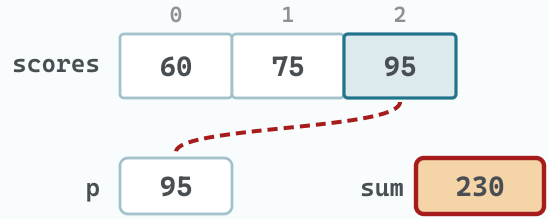
VARIABLES

`scores = [60, 75, 95]` `sum = 135` `p = 95`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
3  for (int p : scores) {
▶      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



sum = 230. No elements left; the loop ends.

Step 8 / 12

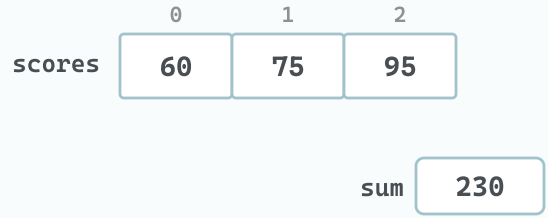
VARIABLES

`scores = [60, 75, 95]` `sum = 230` `p = 95`

OUTPUT

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
3  for (int p : scores) {
4      sum += p;
5  }
▶  System.out.println("Sum: " + sum);
7  for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



`Sum: 230` is printed.

Step 9 / 12

VARIABLES

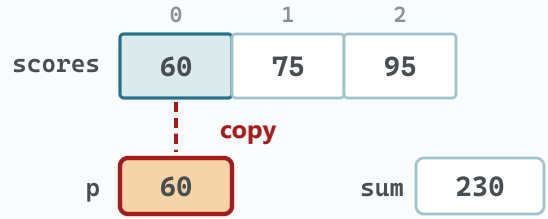
`scores = [60, 75, 95]` `sum = 230`

OUTPUT

```
Sum: 230
```

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
3  for (int p : scores) {
4      sum += p;
5  }
6  System.out.println("Sum: " + sum);
▶ for (int p : scores) {
8      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



Second loop: p is again a copy of `scores[0]`, 60.

Step 10 / 12

VARIABLES

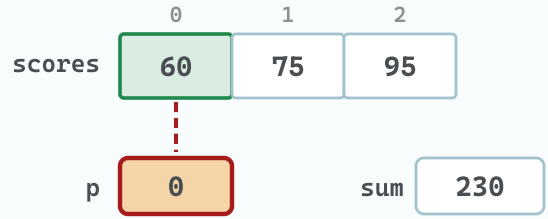
`scores = [60, 75, 95]` `sum = 230` `p = 60`

OUTPUT

```
Sum: 230
```

for-each: You Get the Value, Not the Cell

```
1  int[] scores = {60, 75, 95};
2  int sum = 0;
3  for (int p : scores) {
4      sum += p;
5  }
6  System.out.println("Sum: " + sum);
7  for (int p : scores) {
▶      p = 0;
9  }
10 System.out.println(scores[0] + " " +
   scores[2]);
```



`p = 0` changes **only the copy**; the 60 in the array stays put. The remaining passes are the same.

Step 11 / 12

VARIABLES

`scores = [60, 75, 95]` `sum = 230` `p = 0`

OUTPUT

```
Sum: 230
```

for-each: You Get the Value, Not the Cell

```
1 int[] scores = {60, 75, 95};
2 int sum = 0;
3 for (int p : scores) {
4     sum += p;
5 }
6 System.out.println("Sum: " + sum);
7 for (int p : scores) {
8     p = 0;
9 }
▶ System.out.println(scores[0] + " " +
  scores[2]);
```



60 95: the array never changed. for-each **cannot write** to the array; writing needs a classic for and `scores[i]`.

Step 12 / 12

VARIABLES

`scores = [60, 75, 95]` `sum = 230`

OUTPUT

```
Sum: 230
60 95
```

Temperature Conversion: Arrays and for-each

```java
double[] celsiusValues = {-40, 0, 25, 100};
for (double celsius : celsiusValues) {
    double fahrenheit = (celsius * 1.8) + 32;
    double kelvin = celsius + 273.15;
    System.out.println(celsius + " °C = " + fahrenheit
            + " °F = " + kelvin + " K");
}
```

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
100.0 °C = 212.0 °F = 373.15 K
```



- `double[] celsiusValues = {…};` gathers four values into an **array**.
- **for-each** loop: `for (double celsius : celsiusValues)` means “for each element in the array”; `celsius` takes the next element on each pass.
- The body runs 4 times; the table is produced in one go.
- `233.14999999999998` : floating-point numbers cannot be stored exactly in binary, so small rounding errors appear. When printing, `printf` can limit the digits.

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```



The array holds four Celsius values (indices 0–3). The `for-each` loop will copy them, one by one, into `celsius`.

Step 1 / 14

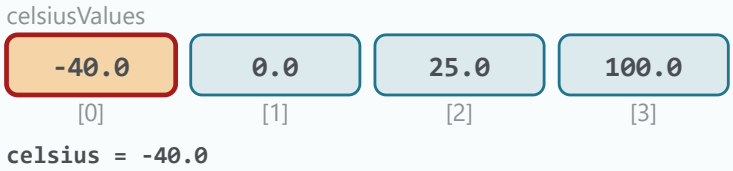
VARIABLES

—

OUTPUT

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25,
  100};
▶ for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius *
  1.8) + 32;
4     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C
  = " + fahrenheit
6             + " °F = " + kelvin + "
  K");
7 }
```



celsius = -40.0

The next element (index 0) is copied into `celsius`: `celsius = -40.0`.

Step 2 / 14

VARIABLES

`celsius = -40.0`

OUTPUT

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25,
  100};
2 for (double celsius : celsiusValues) {
▶     double fahrenheit = (celsius *
  1.8) + 32;
▶     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C
  = " + fahrenheit
6             + " °F = " + kelvin + "
  K");
7 }
```



`celsius = -40.0` `°F = -40.0` `K = 233.14999999999998`

`-40.0 × 1.8 + 32 = -40.0` °F, `-40.0 + 273.15 = 233.14999999999998` K. In Kelvin, `233.14999999999998` : binary floating point stores the value only approximately.

Step 3 / 14

VARIABLES

`celsius = -40.0` `fahrenheit = -40.0` `kelvin = 233.14999999999998`

OUTPUT

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
▶     System.out.println(celsius + " °C = " + fahrenheit
▶             + " °F = " + kelvin + " K");
7 }
```



`celsius = -40.0` `°F = -40.0`

`K = 233.14999999999998`

The line is printed; the body ends and the loop moves to the next element.

Step 4 / 14

VARIABLES

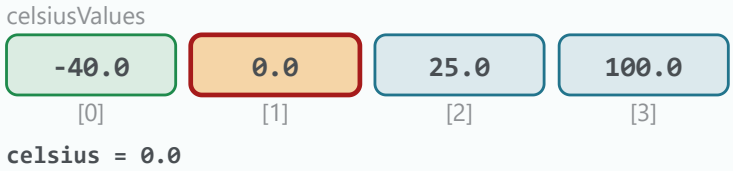
`celsius = -40.0` `fahrenheit = -40.0` `kelvin = 233.14999999999998`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25,
  100};
▶ for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius *
  1.8) + 32;
4     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C
  = " + fahrenheit
6             + " °F = " + kelvin + "
  K");
7 }
```



`celsius = 0.0`

The next element (index 1) is copied into `celsius`: `celsius = 0.0`.

Step 5 / 14

VARIABLES

`celsius = 0.0`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
▶     double fahrenheit = (celsius * 1.8) + 32;
▶     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```

celsiusValues

-40.0	0.0	25.0	100.0
[0]	[1]	[2]	[3]

celsius = 0.0 **°F = 32.0** **K = 273.15**

`0.0 × 1.8 + 32 = 32.0` °F, `0.0 + 273.15 = 273.15` K.

Step 6 / 14

VARIABLES

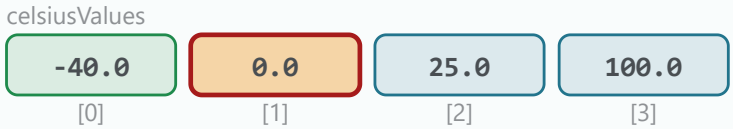
`celsius = 0.0` `fahrenheit = 32.0` `kelvin = 273.15`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
▶     System.out.println(celsius + " °C = " + fahrenheit
▶             + " °F = " + kelvin + " K");
7 }
```



celsius = 0.0 **°F = 32.0**

K = 273.15

The line is printed; the body ends and the loop moves to the next element.

Step 7 / 14

VARIABLES

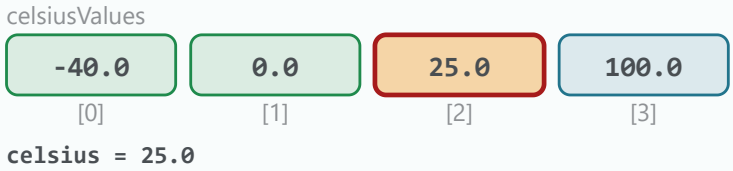
`celsius = 0.0` `fahrenheit = 32.0` `kelvin = 273.15`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
▶ for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```



celsius = 25.0

The next element (index 2) is copied into `celsius`: `celsius = 25.0`.

Step 8 / 14

VARIABLES

`celsius = 25.0`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
▶     double fahrenheit = (celsius * 1.8) + 32;
▶     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```



`celsius = 25.0` `°F = 77.0` `K = 298.15`

`25.0 × 1.8 + 32 = 77.0` °F, `25.0 + 273.15 = 298.15` K.

Step 9 / 14

VARIABLES

`celsius = 25.0` `fahrenheit = 77.0` `kelvin = 298.15`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
▶     System.out.println(celsius + " °C = " + fahrenheit
▶             + " °F = " + kelvin + " K");
7 }
```



celsius = 25.0 **°F = 77.0**

K = 298.15

The line is printed; the body ends and the loop moves to the next element.

Step 10 / 14

VARIABLES

`celsius = 25.0` `fahrenheit = 77.0` `kelvin = 298.15`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
▶ for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```

celsiusValues

-40.0	0.0	25.0	100.0
[0]	[1]	[2]	[3]

celsius = 100.0

The next element (index 3) is copied into `celsius`: `celsius = 100.0`.

Step 11 / 14

VARIABLES

`celsius = 100.0`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
▶     double fahrenheit = (celsius * 1.8) + 32;
▶     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```

celsiusValues

-40.0	0.0	25.0	100.0
[0]	[1]	[2]	[3]

celsius = 100.0 **°F = 212.0** **K = 373.15**

`100.0 × 1.8 + 32 = 212.0` °F, `100.0 + 273.15 = 373.15` K.

Step 12 / 14

VARIABLES

`celsius = 100.0` `fahrenheit = 212.0` `kelvin = 373.15`

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
2 for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
▶     System.out.println(celsius + " °C = " + fahrenheit
▶             + " °F = " + kelvin + " K");
7 }
```



celsius = 100.0 **°F = 212.0**

K = 373.15

The last line is printed.

Step 13 / 14

VARIABLES

`celsius` = **100.0** `fahrenheit` = **212.0** `kelvin` = **373.15**

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
100.0 °C = 212.0 °F = 373.15 K
```

A Temperature Table with for-each

```
1 double[] celsiusValues = {-40, 0, 25, 100};
▶ for (double celsius : celsiusValues) {
3     double fahrenheit = (celsius * 1.8) + 32;
4     double kelvin = celsius + 273.15;
5     System.out.println(celsius + " °C = " + fahrenheit
6             + " °F = " + kelvin + " K");
7 }
```

celsiusValues

-40.0	0.0	25.0	100.0
[0]	[1]	[2]	[3]

No elements left: the loop ends. The body ran 4 times; no index variable was ever written.

Step 14 / 14

VARIABLES

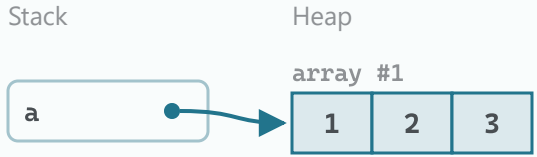
—

OUTPUT

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
100.0 °C = 212.0 °F = 373.15 K
```

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
6 c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



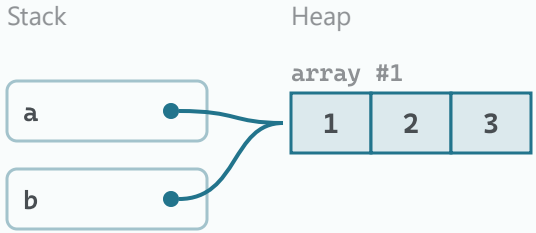
An array is created on the heap; the variable `a` holds not the array itself but its **address (reference)**.

Step 1 / 8

OUTPUT

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
▶ int[] b = a;          // not a copy!
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
6 c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



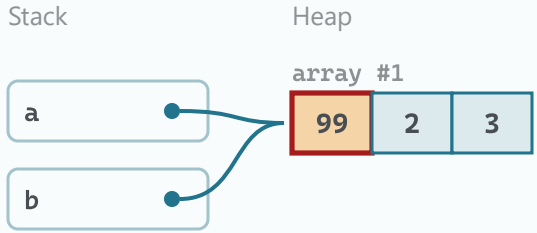
`b = a` only **copies the address**. Now both variables refer to the **same** array (aliasing).

Step 2 / 8

OUTPUT

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
▶ b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
6 c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



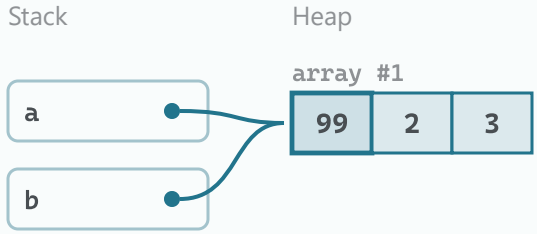
`b[0] = 99` changes the shared array...

Step 3 / 8

OUTPUT

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
3 b[0] = 99;
▶ System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
6 c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



…so `a[0]` also becomes **99**.

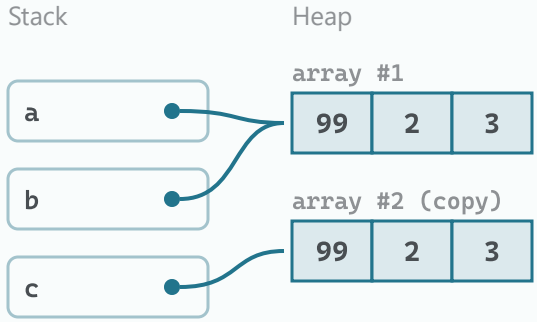
Step 4 / 8

OUTPUT

```
99
```

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
3 b[0] = 99;
4 System.out.println(a[0]);
▶ int[] c = a.clone(); // a real copy
6 c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



`clone()` creates a **new** array on the heap and copies the elements; `c` refers to this new array.

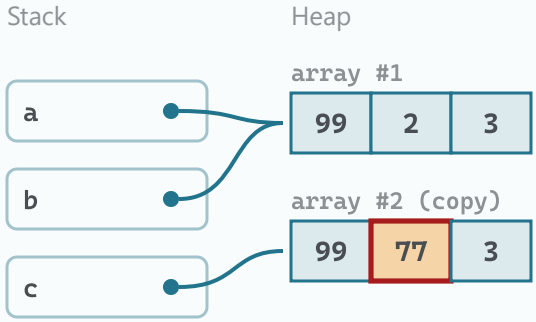
Step 5 / 8

OUTPUT

```
99
```

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
▶ c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



`c[1] = 77` changes only the copy; the original is unaffected.

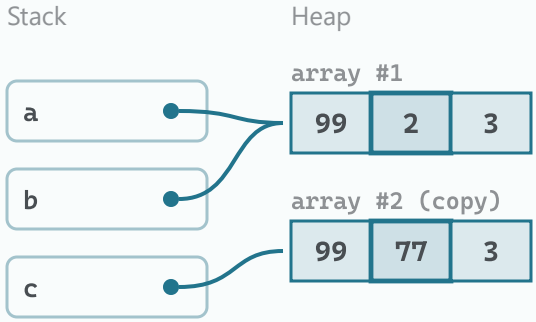
Step 6 / 8

OUTPUT

```
99
```

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
6 c[1] = 77;
▶ System.out.println(a[1] + " " + c[1]);
8 System.out.println(a == b);
9 System.out.println(a == c);
```



`a[1]` is still 2, while `c[1]` is 77.

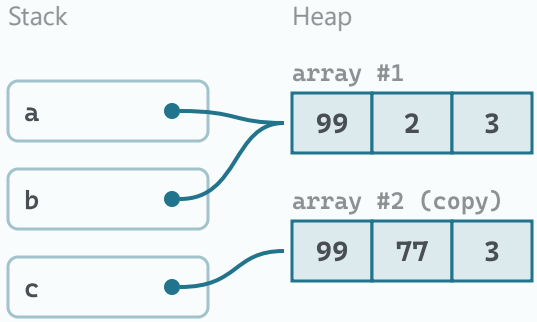
Step 7 / 8

OUTPUT

```
99
2 77
```

An Array Variable Is a Reference

```
1 int[] a = {1, 2, 3};
2 int[] b = a;          // not a copy!
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone(); // a real copy
6 c[1] = 77;
7 System.out.println(a[1] + " " + c[1]);
▶ System.out.println(a == b);
▶ System.out.println(a == c);
```



`==` compares references (addresses): `a == b` is true (same array), `a == c` is false (different arrays).

Step 8 / 8

OUTPUT

```
99
2 77
true
false
```

The java.util.Arrays Class

```
int[] arr = {5, 3, 8, 1};
System.out.println(Arrays.toString(arr));
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
int[] copy = Arrays.copyOf(arr, 6);
System.out.println(Arrays.toString(copy));
int[] filled = new int[4];
Arrays.fill(filled, 7);
System.out.println(Arrays.toString(filled));
int[] x = {1, 2};
int[] y = {1, 2};
System.out.println((x == y) + " " + Arrays.equals(x, y));
System.out.println(Arrays.binarySearch(arr, 5));
```

OUTPUT

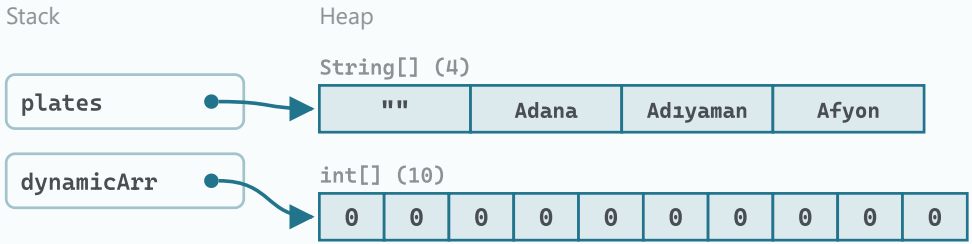
```
[5, 3, 8, 1]
[1, 3, 5, 8]
[1, 3, 5, 8, 0, 0]
[7, 7, 7, 7]
false true
2
```

Method (`import java.util.Arrays;`)	**Purpose**
`Arrays.toString(d)`	Array as text; `println(d)` prints an identity like `[I@7ef20235`
`Arrays.sort(d)` · `binarySearch`	Sorts in place · finds an index in a **sorted** array (5 → 2); arrays have no `indexOf`
`Arrays.copyOf(d, n)` · `d.clone()`	Copies into a new array; extra cells are 0
`Arrays.fill` · `Arrays.equals`	Same value in every cell · compares contents (`==` compares addresses)

Where Do Arrays Live in Memory?

```
String[] plates = {"", "Adana", "Adıyaman", "Afyon"};
System.out.println(plates[1]);
int[] dynamicArr = new int[10]; // allocated on the heap
System.out.println(dynamicArr.length);
dynamicArr = null; // reference dropped
System.out.println(dynamicArr == null);
```

- In **C/C++**, `int a[10];` is on the stack; a raw array from `malloc` / `new[]` is freed **by hand** with `free` / `delete[]` (C++'s `std::vector` does it for you).
- Output: `Adana` , `10` , `true` .
- In **Java**, **all arrays are on the heap**; the size can be set at run time, then stays fixed. Freeing is **automatic**.

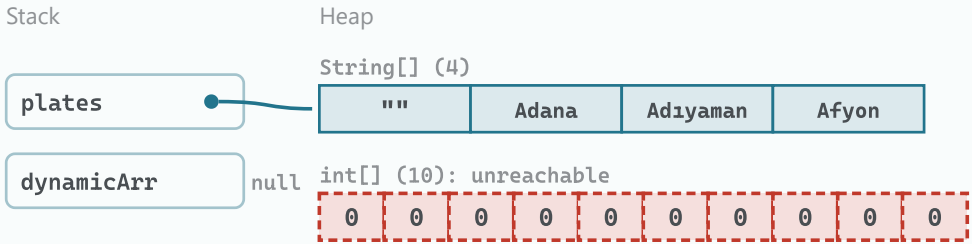


Both arrays are **on the heap**; only the reference variables are on the stack. (A `String[]` 's cells are references to separate `String` objects.)

Where Do Arrays Live in Memory?

```
String[] plates = {"", "Adana", "Adıyaman", "Afyon"};
System.out.println(plates[1]);
int[] dynamicArr = new int[10]; // allocated on the heap
System.out.println(dynamicArr.length);
dynamicArr = null; // reference dropped
System.out.println(dynamicArr == null);
```

- In **C/C++**, `int a[10];` is on the stack; a raw array from `malloc` / `new[]` is freed **by hand** with `free` / `delete[]` (C++'s `std::vector` does it for you).
- Output: `Adana` , `10` , `true` .
- In **Java**, **all arrays are on the heap**; the size can be set at run time, then stays fixed. Freeing is **automatic**.



`dynamicArr = null` **does not delete the array**; it only breaks the link. The array is now unreachable, so the **garbage collector** reclaims it later.

Where Do Arrays Live in Memory?

```
String[] plates = {"", "Adana", "Adıyaman", "Afyon"};
System.out.println(plates[1]);
int[] dynamicArr = new int[10]; // allocated on the heap
System.out.println(dynamicArr.length);
dynamicArr = null; // reference dropped
System.out.println(dynamicArr == null);
```

- In **C/C++**, `int a[10];` is on the stack; a raw array from `malloc` / `new[]` is freed **by hand** with `free` / `delete[]` (C++'s `std::vector` does it for you).
- Output: `Adana` , `10` , `true` .
- In **Java**, **all arrays are on the heap**; the size can be set at run time, then stays fixed. Freeing is **automatic**.



When the method returns, its stack frame and references go **automatically**; unreferenced arrays are collected in time. Java has no `free` / `delete` .

Index Out of Bounds

- Accessing an index outside the array's bounds (0 … length − 1) makes Java throw, at run time, an `ArrayIndexOutOfBoundsException` (a subclass of `IndexOutOfBoundsException`).
- If not caught, the program **crashes** at that line; it does not silently corrupt other memory as C does.

```java
int[] arr = {10, 20, 30, 40, 50};
int index = 5;
// check the array length
if (index >= 0 && index < arr.length) {
    int value = arr[index];
    System.out.println("Value: " + value);
} else {
    System.out.println("Invalid index: " + index);
}
// unchecked access:
System.out.println(arr[index]);
```

OUTPUT

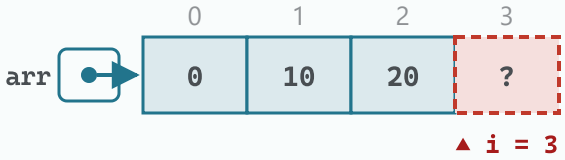
```
Invalid index: 5
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
Index 5 out of bounds for length 5
        at
OutOfBounds.main(OutOfBounds.java:13)
```

The message gives the bad index (5), the array length (5) and the line number: the first place to look.

Classic Bug: Looping with `<=`

```
int[] arr = new int[3];
for (int i = 0; i <= arr.length; i++) {
    arr[i] = i * 10;
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: Index
3 out of bounds for length 3
        at BoundsLoop.main(BoundsLoop.java:5)
```



- `i = 3` : the condition `3 <= 3` is still true, and the loop tries to write `arr[3]` .
- Correct condition: `i < arr.length` ; with a constant (`i < 3`) in place of `length` , the code would break when the size changes.

Looping with <=: Which Pass Fails?

```
▶ int[] sq = new int[4];
2 for (int i = 0; i <= sq.length; i++) {
3     sq[i] = i * i;
4 }
5 System.out.println("done");
```



A 4-cell array; valid indices are 0 … 3, `sq.length` = 4.

Step 1 / 11

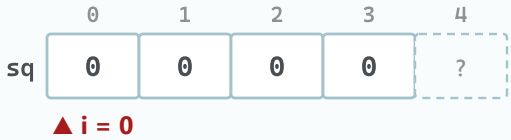
VARIABLES

`sq = [0, 0, 0, 0]`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
▶ for (int i = 0; i <= sq.length; i++) {
3     sq[i] = i * i;
4 }
5 System.out.println("done");
```



i = 0: condition `0 <= 4` true.

Step 2 / 11

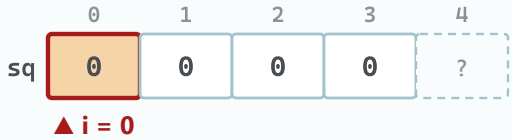
VARIABLES

`sq = [0, 0, 0, 0]` `i = 0`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
2 for (int i = 0; i <= sq.length; i++) {
▶     sq[i] = i * i;
4 }
5 System.out.println("done");
```



`sq[0] = 0` .

Step 3 / 11

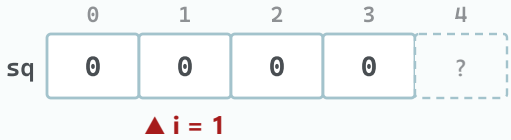
VARIABLES

`sq = [0, 0, 0, 0]` `i = 0`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
▶ for (int i = 0; i <= sq.length; i++) {
3     sq[i] = i * i;
4 }
5 System.out.println("done");
```



i = 1: `1 <= 4` true.

Step 4 / 11

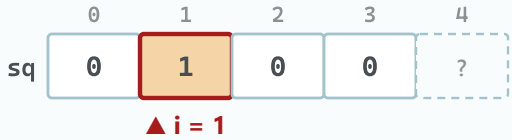
VARIABLES

`sq = [0, 0, 0, 0]` `i = 1`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
2 for (int i = 0; i <= sq.length; i++) {
▶     sq[i] = i * i;
4 }
5 System.out.println("done");
```



`sq[1] = 1` .

Step 5 / 11

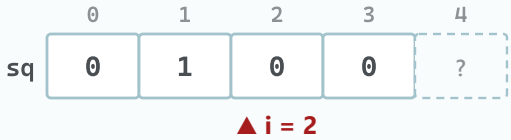
VARIABLES

`sq = [0, 1, 0, 0]` `i = 1`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
▶ for (int i = 0; i <= sq.length; i++) {
3     sq[i] = i * i;
4 }
5 System.out.println("done");
```



i = 2: true.

Step 6 / 11

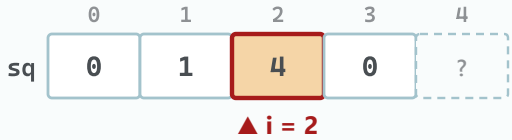
VARIABLES

`sq = [0, 1, 0, 0]` `i = 2`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
2 for (int i = 0; i <= sq.length; i++) {
▶     sq[i] = i * i;
4 }
5 System.out.println("done");
```



`sq[2] = 4` .

Step 7 / 11

VARIABLES

`sq = [0, 1, 4, 0]` `i = 2`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
▶ for (int i = 0; i <= sq.length; i++) {
3     sq[i] = i * i;
4 }
5 System.out.println("done");
```



i = 3: true.

Step 8 / 11

VARIABLES

`sq = [0, 1, 4, 0]` `i = 3`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
2 for (int i = 0; i <= sq.length; i++) {
▶     sq[i] = i * i;
4 }
5 System.out.println("done");
```



`sq[3] = 9` : the last valid cell is filled.

Step 9 / 11

VARIABLES

`sq = [0, 1, 4, 9]` `i = 3`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
▶ for (int i = 0; i <= sq.length; i++) {
3     sq[i] = i * i;
4 }
5 System.out.println("done");
```



i = 4: `4 <= 4` is also **true**! With < the loop would have ended here.

Step 10 / 11

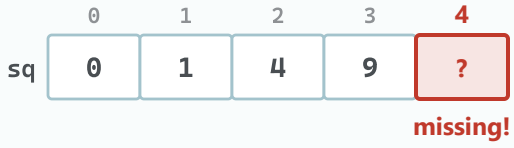
VARIABLES

`sq = [0, 1, 4, 9]` `i = 4`

OUTPUT

Looping with <=: Which Pass Fails?

```
1 int[] sq = new int[4];
2 for (int i = 0; i <= sq.length; i++) {
▶     sq[i] = i * i;
4 }
5 System.out.println("done");
```



ArrayIndexOutOfBoundsException

`sq[4]` does not exist: the JVM throws `ArrayIndexOutOfBoundsException`, and the program stops at this line; “done” is never printed.

Step 11 / 11

VARIABLES

`sq = [0, 1, 4, 9]`

OUTPUT

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
Index 4 out of bounds for length 4
```

Reading an Array from the User

```java
import java.util.Scanner;
public class Average {
    public static void main(String[] args) {
        // read the array size from the user
        Scanner scanner = new Scanner(System.in);
        System.out.print("enter size: ");
        int size = scanner.nextInt();
        // read the elements into a new array
        double[] arr = new double[size];
        for (int i = 0; i < size; i++) {
            System.out.print("element " + (i + 1) + ": ");
            arr[i] = scanner.nextDouble();
        }
        // ... computations on the array
    }
}
```

SAMPLE RUN (INPUT IN BOLD)

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

- The array size comes **from the user**: `new double[size]`. The size is fixed at run time and never changes.
- The prompt prints `i + 1` (users count from 1); the value is stored with `arr[i]` (from 0).
- The mean, median and mode programs all start with this input part.

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
▶  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```

The user enters **4** as the size.

`nextInt()` reads an integer from the keyboard: `size = 4`. Only now, **at run time**, is the array size known.

Step 1 / 13

VARIABLES

`size = 4`

OUTPUT

```
enter size: 4
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
▶  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`new double[size]` : a 4-cell array, all `0.0` .

Step 2 / 13

VARIABLES

`size = 4`

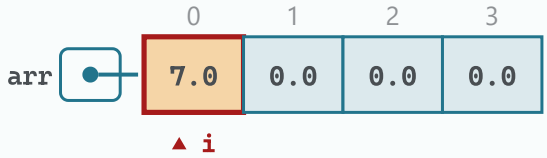
`arr = [0.0, 0.0, 0.0, 0.0]`

OUTPUT

```
enter size: 4
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`i = 0` : the prompt `element 1:` is printed, the user enters **7** → `arr[0] = 7.0` .

Step 3 / 13

VARIABLES

`size = 4` `i = 0`

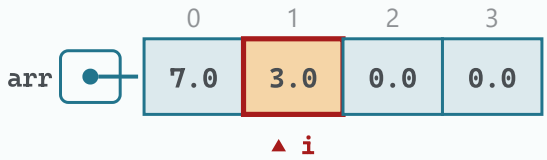
`arr = [7.0, 0.0, 0.0, 0.0]`

OUTPUT

```
enter size: 4
element 1: 7
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`i = 1` : the prompt `element 2:` is printed, the user enters **3** → `arr[1] = 3.0`.

Step 4 / 13

VARIABLES

`size = 4` `i = 1`

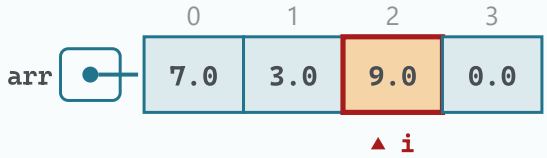
`arr = [7.0, 3.0, 0.0, 0.0]`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`i = 2` : the prompt `element 3:` is printed, the user enters **9** → `arr[2] = 9.0`.

Step 5 / 13

VARIABLES

`size = 4` `i = 2`

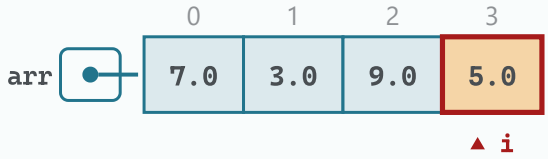
`arr = [7.0, 3.0, 9.0, 0.0]`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`i = 3` : the prompt `element 4:` is printed, the user enters **5** → `arr[3] = 5.0` .

Step 6 / 13

VARIABLES

`size = 4` `i = 3`

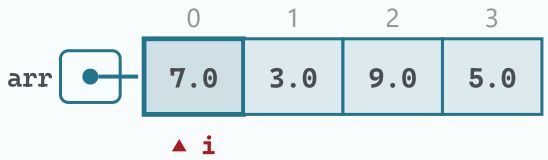
`arr = [7.0, 3.0, 9.0, 5.0]`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
▶      sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`sum += arr[0]` → sum = **7.0**.

Step 7 / 13

VARIABLES

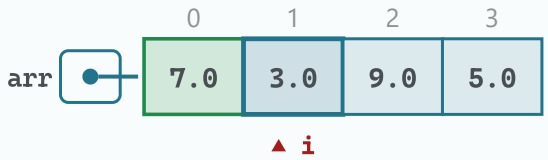
`i = 0` `sum = 7.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
▶      sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`sum += arr[1]` → sum = **10.0**.

Step 8 / 13

VARIABLES

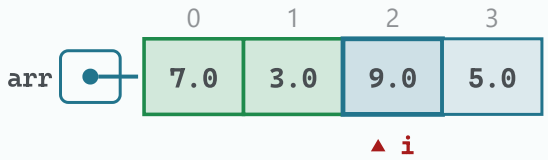
`i = 1` `sum = 10.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
▶      sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`sum += arr[2]` → sum = **19.0**.

Step 9 / 13

VARIABLES

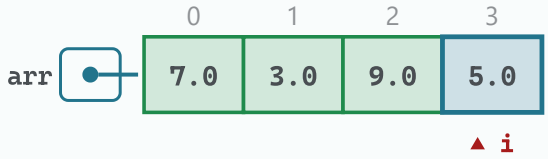
`i = 2` `sum = 19.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
▶      sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`sum += arr[3]` → sum = **24.0**.

Step 10 / 13

VARIABLES

`i = 3` `sum = 24.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
▶ double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
19 System.out.println("Mean: " + mean);
```



`mean = 24.0 / 4` = **6.0**. `sum` is a `double`, so the division is floating-point.

Step 11 / 13

VARIABLES

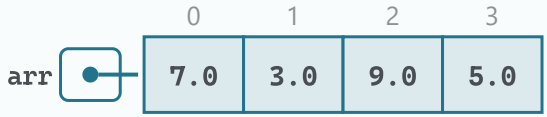
`sum = 24.0` `mean = 6.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
▶  System.out.println();
19 System.out.println("Mean: " + mean);
```



The second loop prints the elements side by side; the values appear as `double` values, like `7.0`. Then `println()` ends the line.

Step 12 / 13

VARIABLES

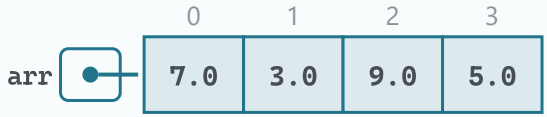
`sum = 24.0` `mean = 6.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
Elements: 7.0 3.0 9.0 5.0
```

Average: Trace the Code Step by Step

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("enter size: ");
3  int size = scanner.nextInt();
4  double[] arr = new double[size];
5  for (int i = 0; i < size; i++) {
6      System.out.print("element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8  }
9  double sum = 0;
10 for (int i = 0; i < size; i++) {
11     sum += arr[i];
12 }
13 double mean = sum / size;
14 System.out.print("Elements: ");
15 for (int i = 0; i < size; i++) {
16     System.out.print(arr[i] + " ");
17 }
18 System.out.println();
▶  System.out.println("Mean: " + mean);
```



The result is printed.

Step 13 / 13

VARIABLES

`sum = 24.0` `mean = 6.0`

OUTPUT

```
enter size: 4
element 1: 7
element 2: 3
element 3: 9
element 4: 5
Elements: 7.0 3.0 9.0 5.0
Mean: 6.0
```

Median

```java
// size and arr are read as in Average
Arrays.sort(arr); // sort the array
double median;
if (size % 2 == 0) { // even number of elements
    int mid1 = size / 2 - 1;
    int mid2 = size / 2;
    median = (arr[mid1] + arr[mid2]) / 2;
} else { // odd number of elements
    median = arr[size / 2];
}
System.out.println("Median: " + median);
```

SAMPLE RUN

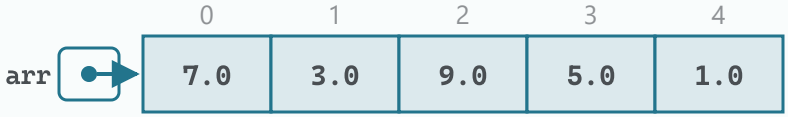
```
input: 7 3 9 5 1
Elements (sorted): 1.0 3.0 5.0 7.0 9.0
Median: 5.0
```

Median

The value in the **middle** of the sorted array. With an even count, it is the mean of the two middle values.

`median` must be declared **outside** the branches; if it is declared only inside them, the compiler reports `error: cannot find symbol … variable median`. `import java.util.Arrays;` is required.

Median: Odd and Even Counts



The input is unsorted: the middle cell (9.0) is **not the median**.

Step 1 / 3

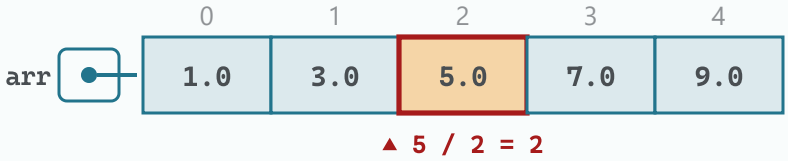
Why sort?

The median is defined by order. The middle element of an unsorted array is just an arbitrary value.

Side effect

`Arrays.sort` changes the array **in place**; the input order is lost. To keep it, first take `arr.clone()`.

Median: Odd and Even Counts



`Arrays.sort(arr)` has run. Size 5 (odd): `arr[5 / 2] = arr[2]` → median **5.0**. Integer division truncates 2.5 to 2.

Step 2 / 3

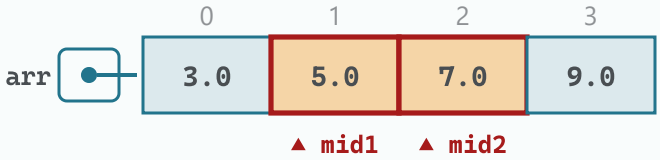
Why sort?

The median is defined by order. The middle element of an unsorted array is just an arbitrary value.

Side effect

`Arrays.sort` changes the array **in place**; the input order is lost. To keep it, first take `arr.clone()`.

Median: Odd and Even Counts



Size 4 (even; input 7 3 9 5): `mid1 = 4/2 - 1 = 1`, `mid2 = 4/2 = 2` → `(5.0 + 7.0) / 2` = **6.0**.

Step 3 / 3

Why sort?

The median is defined by order. The middle element of an unsorted array is just an arbitrary value.

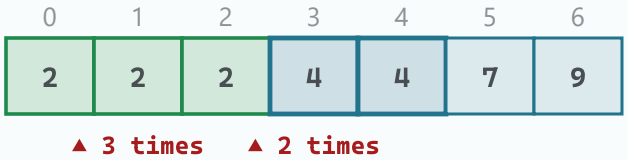
Side effect

`Arrays.sort` changes the array **in place**; the input order is lost. To keep it, first take `arr.clone()`.

Mode

Mode

The value that occurs **most often** in the array. For 4 2 7 2 4 2 9 the mode is 2 (3 times).



- First, `Arrays.sort(arr)`: equal values end up **next to each other**.
- Then a single pass finds the longest "run": each element is compared with the one before it.

PRINTING THE RESULT, AND OUTPUT (INPUT: 7, THEN 4 2 7 2 4 2 9)

```
// print the result
System.out.println("Sorted elements: " + Arrays.toString(arr));
System.out.print("Mode: " + modeValue);
System.out.println(" (count: " + modeCount + ")");
```

```
Sorted elements: [2, 2, 2, 4, 4, 7, 9]
Mode: 2 (count: 3)
```

Mode: Counting Runs Step by Step

```
▶ int modeValue = arr[0];
▶ int modeCount = 1;
▶ int currentValue = arr[0];
▶ int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
7          currentCount++;
8      } else {
9          currentCount = 1;
10         currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
14         modeValue = currentValue;
15     }
16 }
```

0	1	2	3	4	5	6
2	2	2	4	4	7	9

The array has already been **sorted**: equal values are now adjacent. The first element is the candidate for both current and mode (1 time).

Step 1 / 8

VARIABLES

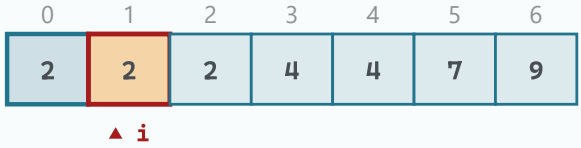
`currentValue = 2`

`currentCount = 1` `modeValue = 2`

`modeCount = 1`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
▶          currentCount++;
8      } else {
9          currentCount = 1;
10         currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
▶          modeValue = currentValue;
15     }
16 }
```



`arr[1] = 2` == `arr[0] = 2` → the run continues, `currentCount = 2`. 2 > previous best → **mode is 2 (2 times)**.

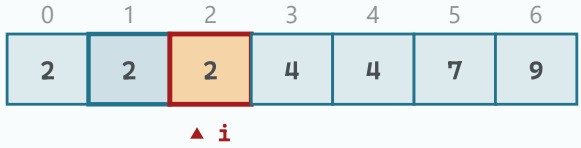
Step 2 / 8

VARIABLES

`i = 1` `currentValue = 2` `currentCount = 2` `modeValue = 2` `modeCount = 2`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
▶          currentCount++;
8      } else {
9          currentCount = 1;
10         currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
▶          modeValue = currentValue;
15     }
16 }
```



`arr[2] = 2` == `arr[1] = 2` → the run continues, `currentCount = 3`. 3 > previous best → **mode is 2 (3 times)**.

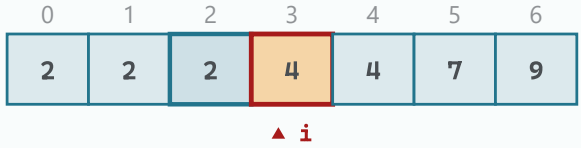
Step 3 / 8

VARIABLES

`i = 2` `currentValue = 2` `currentCount = 3` `modeValue = 2` `modeCount = 3`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
7          currentCount++;
8      } else {
9          currentCount = 1;
▶          currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
14         modeValue = currentValue;
15     }
16 }
```



`arr[3] = 4` ≠ `arr[2] = 2` → a new run starts: `currentValue = 4`, count 1.

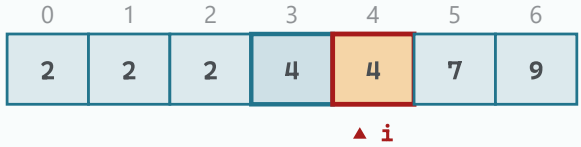
Step 4 / 8

VARIABLES

`i = 3` `currentValue = 4` `currentCount = 1` `modeValue = 2` `modeCount = 3`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
▶          currentCount++;
8      } else {
9          currentCount = 1;
10         currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
14         modeValue = currentValue;
15     }
16 }
```



`arr[4] = 4` == `arr[3] = 4` → the run continues, `currentCount = 2`.

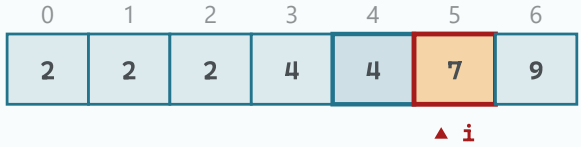
Step 5 / 8

VARIABLES

`i = 4` `currentValue = 4` `currentCount = 2` `modeValue = 2` `modeCount = 3`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
7          currentCount++;
8      } else {
9          currentCount = 1;
▶          currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
14         modeValue = currentValue;
15     }
16 }
```



`arr[5] = 7` ≠ `arr[4] = 4` → a new run starts: `currentValue = 7`, count 1.

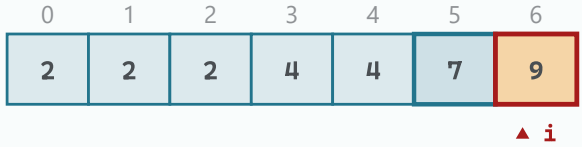
Step 6 / 8

VARIABLES

`i = 5` `currentValue = 7` `currentCount = 1` `modeValue = 2` `modeCount = 3`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
5  for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
7          currentCount++;
8      } else {
9          currentCount = 1;
▶          currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
14         modeValue = currentValue;
15     }
16 }
```



`arr[6] = 9` ≠ `arr[5] = 7` → a new run starts: `currentValue = 9`, count 1.

Step 7 / 8

VARIABLES

`i = 6` `currentValue = 9` `currentCount = 1` `modeValue = 2` `modeCount = 3`

Mode: Counting Runs Step by Step

```
1  int modeValue = arr[0];
2  int modeCount = 1;
3  int currentValue = arr[0];
4  int currentCount = 1;
▶ for (int i = 1; i < size; i++) {
6      if (arr[i] == arr[i - 1]) {
7          currentCount++;
8      } else {
9          currentCount = 1;
10         currentValue = arr[i];
11     }
12     if (currentCount > modeCount) {
13         modeCount = currentCount;
14         modeValue = currentValue;
15     }
16 }
```



`i = 7` : the loop ends. **Mode = 2**, occurring 3 times. With runs of equal length, the `>` comparison would keep the **first** one found.

Step 8 / 8

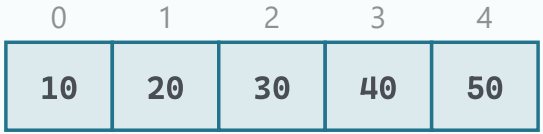
VARIABLES

`i = 7` `currentValue = 9` `currentCount = 1` `modeValue = 2` `modeCount = 3`

Linear Search

The simplest way to find an element in an array: look at every element **in order**, from first to last.

```
1  static boolean linearSearch(int[] arr, int target)
   {
2      for (int item : arr) {
3          if (item == target) {
4              return true;
5          }
6      }
7      return false;
8  }
9
10 // in main:
11 int[] arr = {10, 20, 30, 40, 50};
▶  boolean result = linearSearch(arr, 30);
```



`main` , `linearSearch(arr, 30)` : main calls it. The method receives the array's **reference** and 30.

Step 1 / 5

CALL STACK

`linearSearch(arr, 30)`

`main()`

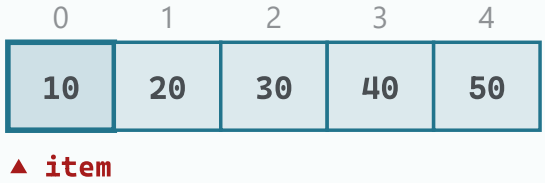
VARIABLES

`target = 30`

Linear Search

The simplest way to find an element in an array: look at every element **in order**, from first to last.

```
1  static boolean linearSearch(int[] arr, int target)
   {
2      for (int item : arr) {
▶          if (item == target) {
4              return true;
5          }
6      }
7      return false;
8  }
9
10 // in main:
11 int[] arr = {10, 20, 30, 40, 50};
12 boolean result = linearSearch(arr, 30);
```



`item = 10` : 10 == 30 is false → go to the next element.

Step 2 / 5

CALL STACK

- `linearSearch(arr, 30)`
- `main()`

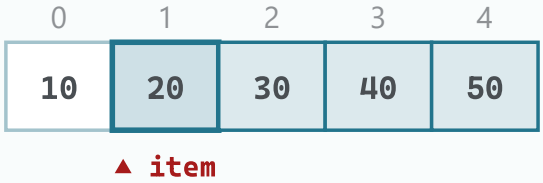
VARIABLES

`target = 30` `item = 10`

Linear Search

The simplest way to find an element in an array: look at every element **in order**, from first to last.

```
1  static boolean linearSearch(int[] arr, int target)
   {
2      for (int item : arr) {
▶          if (item == target) {
4              return true;
5          }
6      }
7      return false;
8  }
9
10 // in main:
11 int[] arr = {10, 20, 30, 40, 50};
12 boolean result = linearSearch(arr, 30);
```



`item = 20` : 20 == 30 is false → go to the next element.

Step 3 / 5

CALL STACK

`linearSearch(arr, 30)`

`main()`

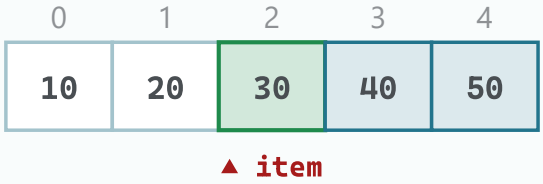
VARIABLES

`target = 30` `item = 20`

Linear Search

The simplest way to find an element in an array: look at every element **in order**, from first to last.

```
1  static boolean linearSearch(int[] arr, int target)
   {
2      for (int item : arr) {
3          if (item == target) {
▶              return true;
5          }
6      }
7      return false;
8  }
9
10 // in main:
11 int[] arr = {10, 20, 30, 40, 50};
12 boolean result = linearSearch(arr, 30);
```



`item = 30` : 30 == 30 → **true**. The loop stops early and returns `true` .

Step 4 / 5

CALL STACK

`linearSearch(arr, 30)`

`main()`

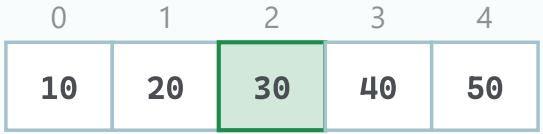
VARIABLES

`target = 30` `item = 30`

Linear Search

The simplest way to find an element in an array: look at every element **in order**, from first to last.

```
1  static boolean linearSearch(int[] arr, int target)
   {
2      for (int item : arr) {
3          if (item == target) {
4              return true;
5          }
6      }
7      return false;
8  }
9
10 // in main:
11 int[] arr = {10, 20, 30, 40, 50};
▶  boolean result = linearSearch(arr, 30);
```



The method's frame is popped off the stack: `result = true`. 3 comparisons were enough; 40 and 50 were never examined. For a target of 35 it would return `false` after 5 comparisons.

Step 5 / 5

CALL STACK

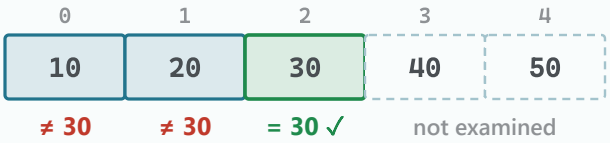
`main()`

VARIABLES

`result = true`

Cost of Linear Search

- In the worst case the **whole array** is scanned: n comparisons for n elements.
- Time complexity is **O(n)** (n: array size): work grows in **direct proportion** to input size.
- **Best case**: the target is first: 1 comparison.
- **Worst case**: the target is last or absent.
- Unsorted, nothing beats this: "absent" needs a look at every element.



key = 30 → 3 comparisons; worst case n = 5

```java
int[] arr = {10, 20, 30, 40, 50};
// search an array for a given element
int key = 30;
boolean found = false;
for (int item : arr) {
    if (item == key) {
        found = true;
        break;
    }
} // this is O(n)
```

`break` exits as soon as the target is found: only 3 elements are checked and `found = true`.

Binary Search

```
static int binarySearch(int[] arr, int target) {
    int lo = 0, hi = arr.length - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (arr[mid] == target) {
            return mid;          // found
        } else if (target < arr[mid]) {
            hi = mid - 1;        // left half
        } else {
            lo = mid + 1;        // right half
        }
    }
    return -1;                   // range is empty
}
// in main:
int[] arr = {3, 9, 14, 27, 42, 48, 55, 61, 70};
System.out.println(binarySearch(arr, 48));
System.out.println(binarySearch(arr, 50));
```

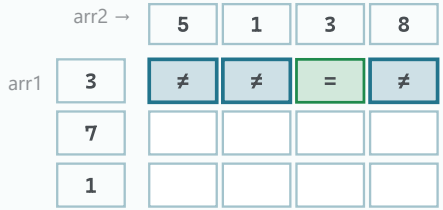
OUTPUT

```
5
-1
```

- Works only on a **sorted** array: look at the middle element; if the target is smaller, search the left half, otherwise the right half.
- For 48 the middle indices are 4 (42), 6 (55), 5 (48): **3 comparisons**. For 50 the range becomes empty once 6 > 5, and the method returns `-1`.
- Each step halves the search range: **O(log n)**. For n = 1000 at most 10 comparisons, for n = 1,000,000 at most 20.
- `Arrays.binarySearch` uses the same method; on an unsorted array its result is meaningless.

Matching: Common Elements

Call: `match(new int[]{3, 7, 1}, new int[]{5, 1, 3, 8})`



Outer loop `e1 = 3` : the inner loop compares it with **all 4 elements** of arr2. A match is found → printed.

Step 1 / 4

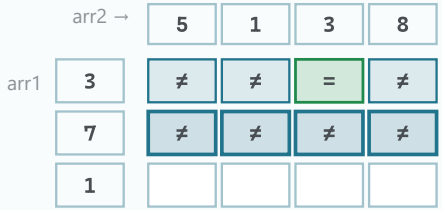
```
1 static void match(int[] arr1, int[] arr2) {
▶     for (int e1 : arr1) {
3         for (int e2 : arr2) {
▶             if (e1 == e2) {
5                 System.out.println("Match: " + e1);
6             }
7         }
8     }
9 }
```

OUTPUT

```
Match: 3
```

Matching: Common Elements

Call: `match(new int[]{3, 7, 1}, new int[]{5, 1, 3, 8})`



Outer loop `e1 = 7` : the inner loop compares it with **all 4 elements** of arr2. No match.

Step 2 / 4

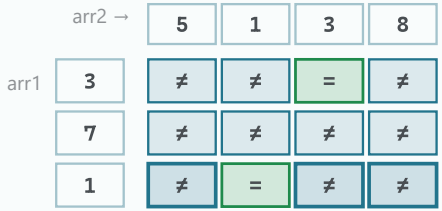
```
1 static void match(int[] arr1, int[] arr2) {
▶       for (int e1 : arr1) {
3           for (int e2 : arr2) {
▶               if (e1 == e2) {
5                   System.out.println("Match: " + e1);
6               }
7           }
8       }
9   }
```

OUTPUT

```
Match: 3
```

Matching: Common Elements

Call: `match(new int[]{3, 7, 1}, new int[]{5, 1, 3, 8})`



Outer loop `e1 = 1` : the inner loop compares it with **all 4 elements** of arr2. A match is found → printed.

Step 3 / 4

```
1 static void match(int[] arr1, int[] arr2) {
▶     for (int e1 : arr1) {
3         for (int e2 : arr2) {
▶             if (e1 == e2) {
5                 System.out.println("Match: " + e1);
6             }
7         }
8     }
9 }
```

OUTPUT

```
Match: 3
Match: 1
```

Matching: Common Elements

Call: `match(new int[]{3, 7, 1}, new int[]{5, 1, 3, 8})`



In total 3 × 4 = **12 comparisons**. For arrays of n elements, that is n × n = n^2 comparisons.

Step 4 / 4

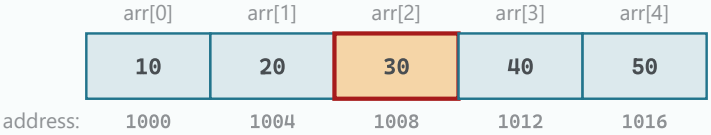
```
1 static void match(int[] arr1, int[] arr2) {
▶     for (int e1 : arr1) {
3         for (int e2 : arr2) {
4             if (e1 == e2) {
5                 System.out.println("Match: " + e1);
6             }
7         }
8     }
9 }
```

OUTPUT

```
Match: 3
Match: 1
```

Complexity: O(1), O(n), O(n^2)

WHY IS ACCESS O(1)?



address(arr[2]) = 1000 + 2 × 4 = 1008

Cells are contiguous and of equal size (`int` = 4 bytes), so the address of element i takes one multiply and one add to **compute**: `address = start + i × 4` . `arr[2]` and `arr[1000000]` take the same time to read.

Operation	Complexity	for n = 1000
Access `arr[i]`	**O(1)** constant	1
Linear search	**O(n)** linear	≤ 1000
Reversal	**O(n)**	500 swaps
Matching (nested)	**O(n^2)** quadratic	1,000,000
Selection sort	**O(n^2)**	≈ 500,000

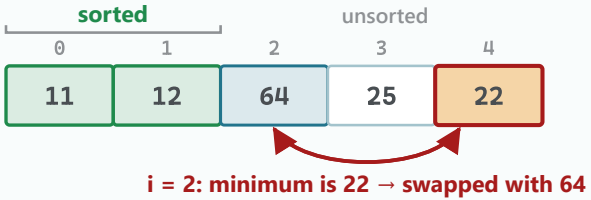
O(n^2) algorithms are **inefficient** for large data sets: if n grows 10×, the time grows 100×. Access, by contrast, does not depend on input size.

Selection Sort

```java
static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        int tmp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = tmp;
    }
}
```

- **Selection sort**: at each step the smallest element is found and placed at the front of the unsorted part.
- The outer loop runs n − 1 times, the inner loop scans the rest → **O(n^2)** comparisons, at most n − 1 swaps.
- Swap **inside the array**, using a temporary variable.

`swap(int x, int y)` as a swap method **does not work**: the method receives only **copies** of the elements. A working version takes the array itself: `swap(int[] a, int i, int j)`.



{64, 25, 12, 22, 11}: Step by Step



Start: no element is in place. Each pass finds the **smallest** element of the remaining part and puts it at the **front** of that part.

Step 1 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

VARIABLES

`i = –` `minIdx = –`

OUTPUT

{64, 25, 12, 22, 11}: Step by Step



Pass `i = 0`: the inner loop searches `j = 1…4` for the smallest value → **11** (index 4).

Step 2 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

VARIABLES

`i = 0` `minIdx = 4`

OUTPUT

{64, 25, 12, 22, 11}: Step by Step



`arr[0]` and `arr[4]` swapped. The first cell is now **in final position**.

Step 3 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

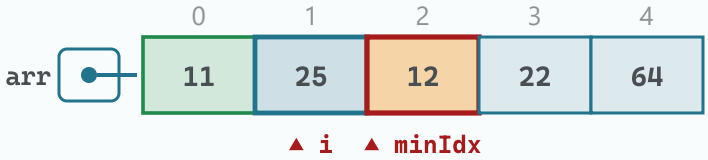
VARIABLES

`i = 0` `minIdx = 4`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
```

{64, 25, 12, 22, 11}: Step by Step



Pass `i = 1`: the inner loop searches `j = 2…4` for the smallest value → **12** (index 2).

Step 4 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

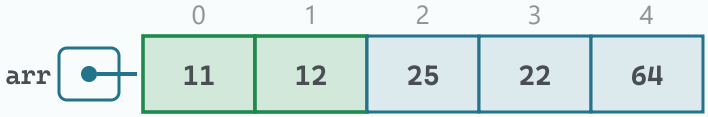
VARIABLES

`i = 1` `minIdx = 2`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
```

{64, 25, 12, 22, 11}: Step by Step



`arr[1]` and `arr[2]` swapped. The first 2 cells are now **in final position**.

Step 5 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

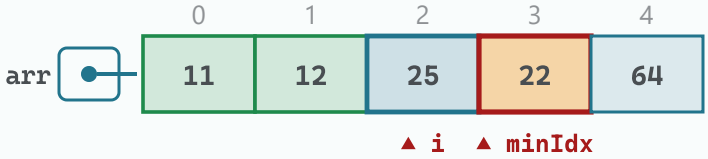
VARIABLES

`i = 1` `minIdx = 2`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
i=1: [11, 12, 25, 22, 64]
```

{64, 25, 12, 22, 11}: Step by Step



Pass `i = 2`: the inner loop searches `j = 3…4` for the smallest value → **22** (index 3).

Step 6 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

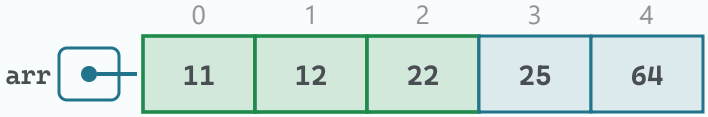
VARIABLES

`i = 2` `minIdx = 3`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
i=1: [11, 12, 25, 22, 64]
```

{64, 25, 12, 22, 11}: Step by Step



`arr[2]` and `arr[3]` swapped. The first 3 cells are now **in final position**.

Step 7 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

VARIABLES

`i = 2` `minIdx = 3`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
i=1: [11, 12, 25, 22, 64]
i=2: [11, 12, 22, 25, 64]
```

{64, 25, 12, 22, 11}: Step by Step



Pass `i = 3` : the inner loop searches `j = 4…4` for the smallest value → **25** (index 3).

Step 8 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

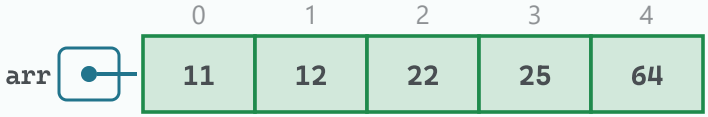
VARIABLES

`i = 3` `minIdx = 3`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
i=1: [11, 12, 25, 22, 64]
i=2: [11, 12, 22, 25, 64]
```

{64, 25, 12, 22, 11}: Step by Step



The minimum is already in place; it is swapped with itself (no change). After 4 passes the last element is in place too: the array is sorted.

Step 9 / 9

```
int[] arr = {64, 25, 12, 22, 11};
selectionSort(arr);
// selectionSort prints after each pass:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(arr));
```

VARIABLES

`i = 3` `minIdx = 3`

OUTPUT

```
i=0: [11, 25, 12, 22, 64]
i=1: [11, 12, 25, 22, 64]
i=2: [11, 12, 22, 25, 64]
i=3: [11, 12, 22, 25, 64]
```

Bubble Sort

```java
static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int pass = 0; pass < n - 1; pass++) {
        boolean swapped = false;
        for (int j = 0; j < n - 1 - pass; j++) {
            if (arr[j] > arr[j + 1]) {  // out of order
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
                swapped = true;
            }
        }
        System.out.println(Arrays.toString(arr));
        if (!swapped) break;  // no swaps: sorted
    }
}
```

OUTPUT: `{64, 25, 12, 22, 11, 36, 48}`

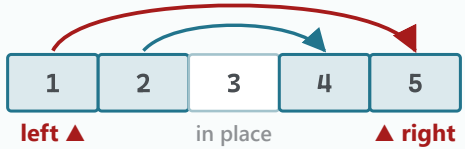
```
[25, 12, 22, 11, 36, 48, 64]
[12, 22, 11, 25, 36, 48, 64]
[12, 11, 22, 25, 36, 48, 64]
[11, 12, 22, 25, 36, 48, 64]
[11, 12, 22, 25, 36, 48, 64]
```

- **Bubble sort**: two neighbors out of order are swapped; on each pass the largest element "bubbles up" to the end.
- Each line is the result of one pass. After each pass the largest element is at the end, so the inner loop only runs up to `n - 1 - pass`.
- `swapped` flag: on pass 5 there are no swaps, so the array is sorted and `break` stops early. A sorted array needs just one pass: O(n).
- In the worst case it is still **$O(n^2)$** comparisons; unlike selection sort, the number of swaps can also be $O(n^2)$.

Reversing an Array

```java
static void reverse(int[] arr) {
    int left = 0;
    int right = arr.length - 1;
    while (left < right) {
        // swap the end elements
        int tmp = arr[left];
        arr[left] = arr[right];
        arr[right] = tmp;
        // move both indices inward
        left++;
        right--;
    }
}
```

- **The two-pointer** technique: one index comes from the front, one from the back; each pass swaps the two end elements.
- About n/2 swaps → **O(n)**; no extra array is needed (in place).
- The method is `void`, but since it receives a copy of the array's reference, **the caller's array** changes.

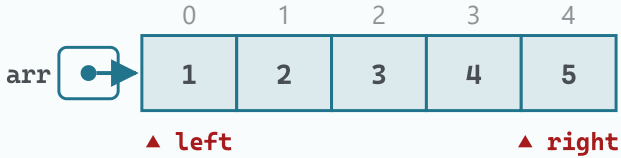


OUTPUT

```
[1, 2, 3, 4, 5]
→ [5, 4, 3, 2, 1]
[1, 2, 3, 4]
→ [4, 3, 2, 1]
```

`reverse`: Step by Step

```
1  static void reverse(int[] arr) {
▶      int left = 0;
▶      int right = arr.length - 1;
4      while (left < right) {
5          int tmp = arr[left];
6          arr[left] = arr[right];
7          arr[right] = tmp;
8          left++;
9          right--;
10     }
11 }
```



Two pointers: `left = 0` starts from the left, `right = length − 1 = 4` from the right.

Step 1 / 4

CALL STACK

- `reverse(arr)`
- `main()`

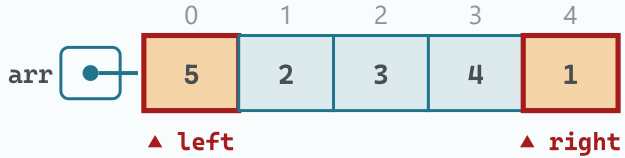
VARIABLES

`left = 0` `right = 4`

`arr = [1, 2, 3, 4, 5]`

`reverse`: Step by Step

```
1  static void reverse(int[] arr) {
2      int left = 0;
3      int right = arr.length - 1;
4      while (left < right) {
▶          int tmp = arr[left];
▶          arr[left] = arr[right];
▶          arr[right] = tmp;
8          left++;
9          right--;
10     }
11 }
```



`0 < 4` → `tmp` is used to swap `arr[0]` and `arr[4]` (`tmp = 1`).

Step 2 / 4

CALL STACK

- `reverse(arr)`
- `main()`

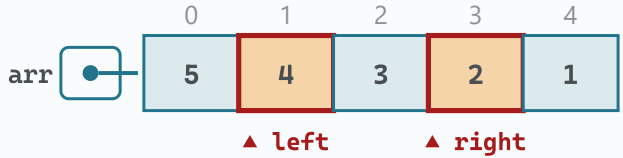
VARIABLES

`left = 0` `right = 4` `tmp = 1`

`arr = [5, 2, 3, 4, 1]`

`reverse`: Step by Step

```
1  static void reverse(int[] arr) {
2      int left = 0;
3      int right = arr.length - 1;
4      while (left < right) {
▶          int tmp = arr[left];
▶          arr[left] = arr[right];
▶          arr[right] = tmp;
8          left++;
9          right--;
10     }
11 }
```



`1 < 3` → `tmp` is used to swap `arr[1]` and `arr[3]` (`tmp = 2`).

Step 3 / 4

CALL STACK

- `reverse(arr)`
- `main()`

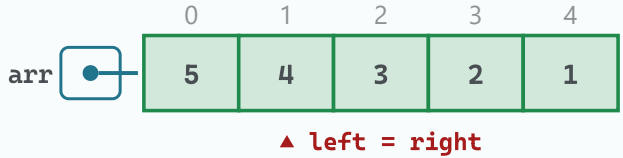
VARIABLES

`left = 1` `right = 3` `tmp = 2`

`arr = [5, 4, 3, 2, 1]`

`reverse`: Step by Step

```
1  static void reverse(int[] arr) {
2      int left = 0;
3      int right = arr.length - 1;
▶      while (left < right) {
5          int tmp = arr[left];
6          arr[left] = arr[right];
7          arr[right] = tmp;
8          left++;
9          right--;
10     }
11 }
```



`left = right = 2` : `2 < 2` is false → the loop ends; the middle element stays put. The method returns nothing, yet **the caller's array** has changed: `[5, 4, 3, 2, 1]` .

Step 4 / 4

CALL STACK

- `reverse(arr)`
- `main()`

VARIABLES

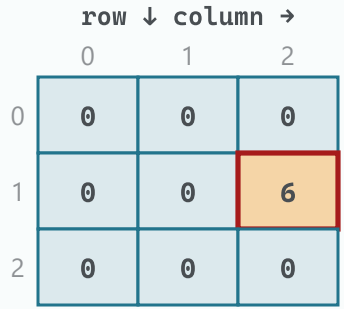
`left = 2` `right = 2`

`arr = [5, 4, 3, 2, 1]`

Two-Dimensional Arrays (Matrices)

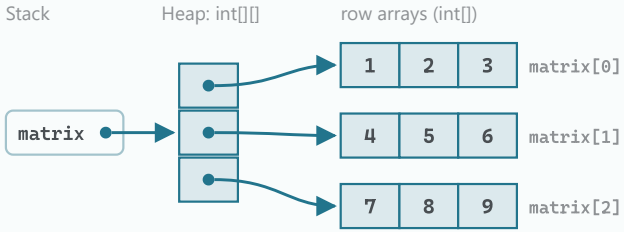
- Multidimensional arrays store data along more than one dimension; Java supports two-dimensional (matrix) and higher-dimensional arrays.
- A two-dimensional array is a table-like structure of **rows and columns**: a chessboard, an image, a grade sheet.
- Access: `matrix[row][col]`; the first index is the **row**, the second the **column**.

```
int[][] matrix = new int[3][3]; // a 3x3 matrix
matrix[1][2] = 6;  // row 2, column 3
```



Initialization and Arrays of Arrays

- Two-dimensional arrays can also take an initializer list; no size is needed.
- In Java a 2D array is really an **array of arrays**: `matrix[i]` is row i itself, that is, an `int[]`.
- `matrix.length` = **row** count, `matrix[0].length` = the first row's **column** count.




```
int[][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
System.out.println(matrix.length + " "
        + matrix[0].length + " " + matrix[1][2]);
```

OUTPUT

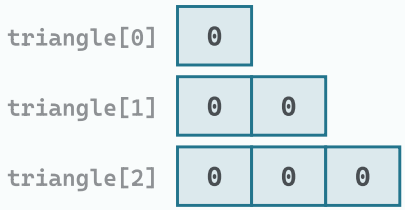
```
3 3 6
```

Ragged Arrays

```
int[][] triangle = new int[3][];
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int[i + 1];
}
System.out.println(triangle[0].length + " " + triangle[1].length
        + " " + triangle[2].length);
int[][] table = {{1, 2}, {3, 4, 5}};
for (int[] row : table) {
    for (int x : row) {
        System.out.print(x + " ");
    }
    System.out.println();
}
```

OUTPUT

```
1 2 3
1 2
3 4 5
```



- Rows are separate arrays, so they **can differ in length**.
- `new int[3][]` creates only the row references (all `null`); the rows are allocated later.
- That is why using `matrix[i].length` in the inner loop is safer than a fixed column count.
- `for-each`: each outer-loop element is an `int[]` row.

Pascal's Triangle: A Ragged Array, Row by Row

```
▶ int[][] p = new int[4][];
2 for (int i = 0; i < p.length; i++) {
3     p[i] = new int[i + 1];
4     p[i][0] = 1;
5     p[i][i] = 1;
6     for (int j = 1; j < i; j++) {
7         p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8     }
9 }
10 System.out.println(Arrays.toString(p[3]));
```



`new int[4][]` reserves only 4 row references; all are `null`.

Step 1 / 13

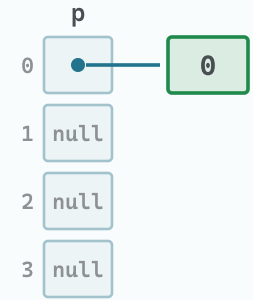
VARIABLES

`i = –`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
▶      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



i = 0: a 1-cell row is created (filled with 0) and `p[0]` points to it.

Step 2 / 13

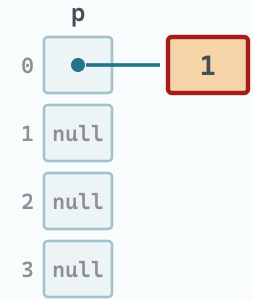
VARIABLES

`i = 0`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
▶      p[i][0] = 1;
▶      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



The row's first and last cell are the same cell: 1.

Step 3 / 13

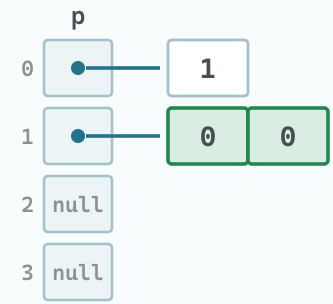
VARIABLES

`i = 0`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
▶      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



i = 1: a 2-cell row.

Step 4 / 13

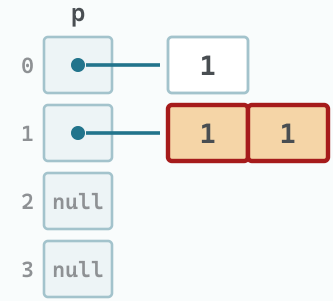
VARIABLES

`i = 1`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
▶      p[i][0] = 1;
▶      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



Ends are 1; no inner cells (`j < 1` is false at once).

Step 5 / 13

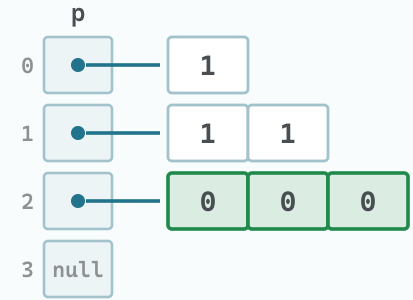
VARIABLES

`i = 1`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
▶      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



i = 2: a 3-cell row.

Step 6 / 13

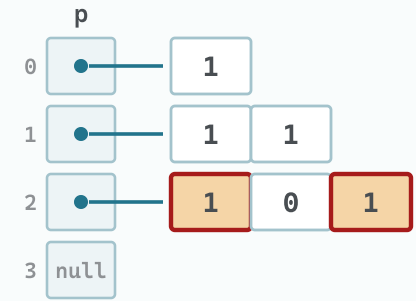
VARIABLES

`i = 2`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
▶      p[i][0] = 1;
▶      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



Ends are 1.

Step 7 / 13

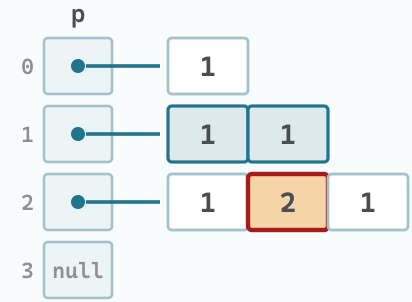
VARIABLES

`i = 2`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
▶          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



p[2][1] = p[1][0] + p[1][1] = 1 + 1 = 2 .

Step 8 / 13

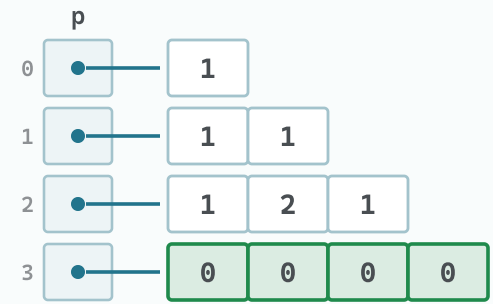
VARIABLES

i = 2 j = 1

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
▶      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



i = 3: a 4-cell row.

Step 9 / 13

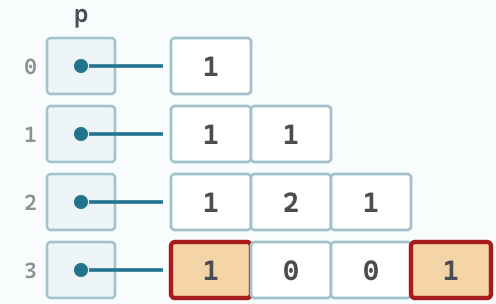
VARIABLES

`i = 3`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
▶      p[i][0] = 1;
▶      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
7          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



Ends are 1.

Step 10 / 13

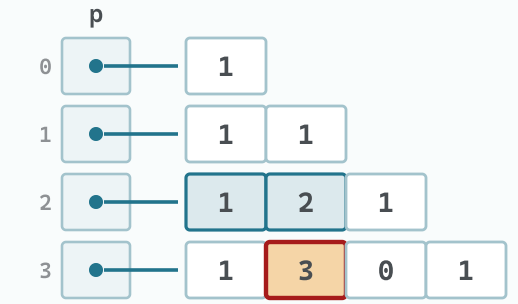
VARIABLES

`i = 3`

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
▶          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



p[3][1] = p[2][0] + p[2][1] = 1 + 2 = 3 .

Step 11 / 13

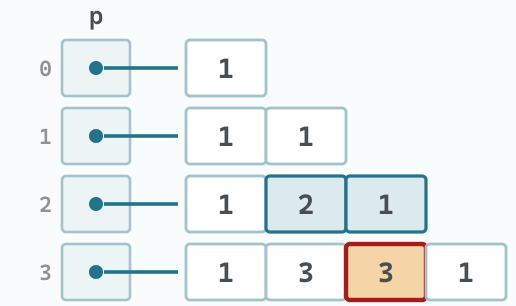
VARIABLES

i = 3 j = 1

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1  int[][] p = new int[4][];
2  for (int i = 0; i < p.length; i++) {
3      p[i] = new int[i + 1];
4      p[i][0] = 1;
5      p[i][i] = 1;
6      for (int j = 1; j < i; j++) {
▶          p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8      }
9  }
10 System.out.println(Arrays.toString(p[3]));
```



p[3][2] = p[2][1] + p[2][2] = 2 + 1 = 3 .

Step 12 / 13

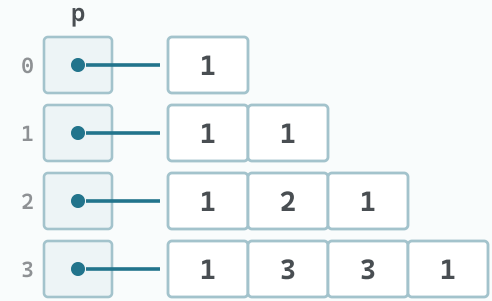
VARIABLES

i = 3 j = 2

OUTPUT

Pascal's Triangle: A Ragged Array, Row by Row

```
1 int[][] p = new int[4][];
2 for (int i = 0; i < p.length; i++) {
3     p[i] = new int[i + 1];
4     p[i][0] = 1;
5     p[i][i] = 1;
6     for (int j = 1; j < i; j++) {
7         p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
8     }
9 }
▶ System.out.println(Arrays.toString(p[3]));
```



Row lengths are 1, 2, 3, 4: a ragged array. The last row is `[1, 3, 3, 1]`.

Step 13 / 13

VARIABLES

—

OUTPUT

```
[1, 3, 3, 1]
```

Example: MultiDimExample

```java
public class MultiDimExample {
    public static void main(String[] args) {
        // a 3x3 two-dimensional array (matrix)
        int[][] matrix = new int[3][3];
        // fill the matrix with initial values
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                matrix[row][col] = row * 3 + col + 1;
            }
        }
        // print the matrix
        System.out.println("Matrix contents:");
        for (int row = 0; row < 3; row++) {
            for (int col = 0; col < 3; col++) {
                System.out.print(matrix[row][col] + " ");
            }
            System.out.println(); // next line
        }
    }
}
```

OUTPUT

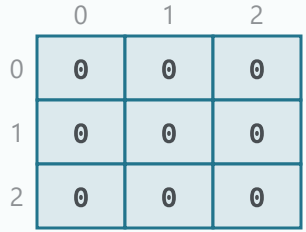
```
Matrix contents:
1 2 3
4 5 6
7 8 9
```

Nested loops

The outer loop walks the **rows**, the inner loop a row's **columns**. `row * 3 + col + 1` numbers the cells from 1 to 9.

Filling and Printing a Matrix: Step by Step

```
1 int[][] matrix = new int[3][3];
2 for (int row = 0; row < 3; row++) {
3     for (int col = 0; col < 3; col++) {
4         matrix[row][col] = row * 3 + col + 1;
5     }
6 }
7 System.out.println("Matrix contents:");
8 for (int row = 0; row < 3; row++) {
9     for (int col = 0; col < 3; col++) {
10        System.out.print(matrix[row][col] + " ");
11    }
12    System.out.println();
13 }
```



`new int[3][3]` : 3 rows × 3 columns, all cells 0.

Step 1 / 11

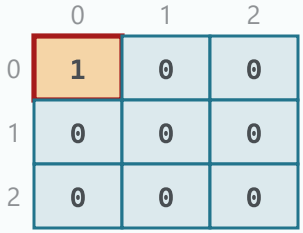
VARIABLES

—

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[0][0] = 0 * 3 + 0 + 1 = 1`

Step 2 / 11

VARIABLES

`row = 0` `col = 0`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[0][1] = 0 * 3 + 1 + 1 = 2`

Step 3 / 11

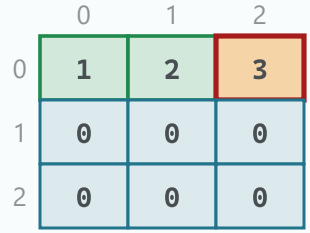
VARIABLES

`row = 0` `col = 1`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[0][2] = 0 * 3 + 2 + 1 = 3`

Step 4 / 11

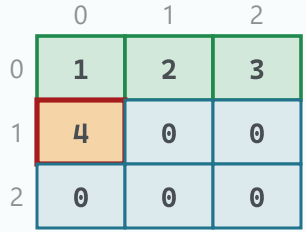
VARIABLES

`row = 0` `col = 2`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[1][0] = 1 * 3 + 0 + 1 = 4` .

The inner loop is done; the outer loop moves to the next **row**.

Step 5 / 11

VARIABLES

`row = 1` `col = 0`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[1][1] = 1 * 3 + 1 + 1 = 5`

Step 6 / 11

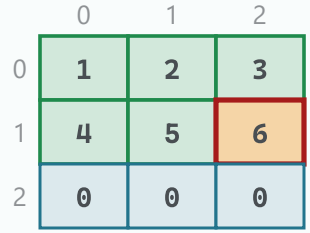
VARIABLES

`row = 1` `col = 1`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[1][2] = 1 * 3 + 2 + 1 = 6`

Step 7 / 11

VARIABLES

`row = 1` `col = 2`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[2][0] = 2 * 3 + 0 + 1 = 7` .

The inner loop is done; the outer loop moves to the next **row**.

Step 8 / 11

VARIABLES

`row = 2` `col = 0`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[2][1] = 2 * 3 + 1 + 1 = 8`

Step 9 / 11

VARIABLES

`row = 2` `col = 1`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
▶          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
10         System.out.print(matrix[row][col] + " ");
11     }
12     System.out.println();
13 }
```



`matrix[2][2] = 2 * 3 + 2 + 1 = 9`

Step 10 / 11

VARIABLES

`row = 2` `col = 2`

OUTPUT

Filling and Printing a Matrix: Step by Step

```
1  int[][] matrix = new int[3][3];
2  for (int row = 0; row < 3; row++) {
3      for (int col = 0; col < 3; col++) {
4          matrix[row][col] = row * 3 + col + 1;
5      }
6  }
7  System.out.println("Matrix contents:");
8  for (int row = 0; row < 3; row++) {
9      for (int col = 0; col < 3; col++) {
▶          System.out.print(matrix[row][col] + " ");
11     }
▶      System.out.println();
13 }
```

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

The print loop visits cells in the same order; at the end of each row `println()` moves to a new line.

Step 11 / 11

VARIABLES

—

OUTPUT

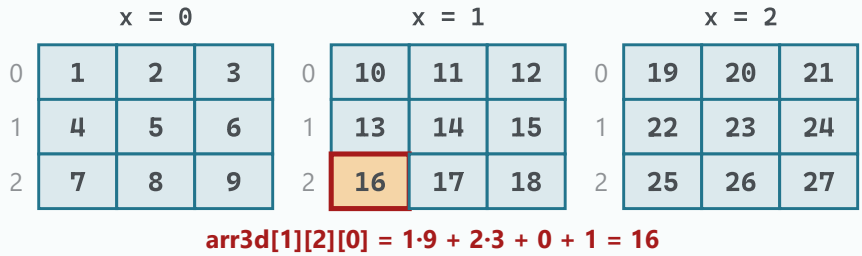
```
Matrix contents:
1 2 3
4 5 6
7 8 9
```

Three or More Dimensions

- Java also allows three- and higher-dimensional arrays: each extra dimension is one more pair of brackets.
- More dimensions allow richer data: a building `[floor][row][col]`, a color image `[y][x][channel]`, a table of measurements over time.
- The element count is the **product**: `new int[3][3][3]` → 3 × 3 × 3 = **27** cells. Memory grows fast as dimensions are added.

```
// a 3x3x3 three-dimensional array
int[][][] arr3d = new int[3][3][3];
arr3d[1][2][0] = 16; // layer 1, row 2, column 0
```

THREE-DIMENSIONAL ARRAY: 3 LAYERS, EACH A 3×3 MATRIX



arr3d[1][2][0] = 1·9 + 2·3 + 0 + 1 = 16

Example: ThreeDimExample

```
int[][][] arr3d = new int[3][3][3];
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        for (int z = 0; z < 3; z++) {
            arr3d[x][y][z] = x * 9 + y * 3 + z + 1;
        }
    }
}
System.out.println("3D array contents:");
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        for (int z = 0; z < 3; z++) {
            System.out.print(arr3d[x][y][z] + " ");
        }
        System.out.println();
    }
    System.out.println();
}
```

OUTPUT

```
3D array contents:
1 2 3
4 5 6
7 8 9

10 11 12
13 14 15
16 17 18

19 20 21
22 23 24
25 26 27
```

Three nested loops: `z` runs along a row, `y` through a layer, `x` through the whole array. A blank line follows each layer.

Matrix Transpose

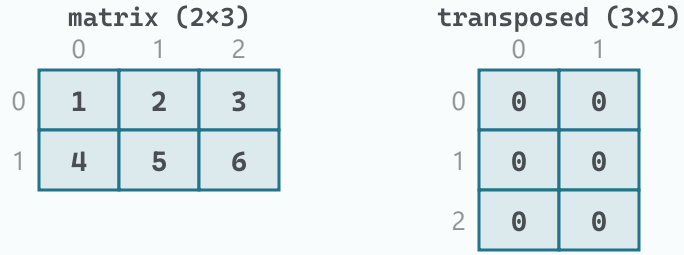
```
public static void main(String[] args) {
    int[][] matrix = {
        {1, 2, 3},
        {4, 5, 6}
    };
    System.out.println("Original matrix:");
    printMatrix(matrix);
    int rows = matrix.length;
    int cols = matrix[0].length;
    int[][] transposed = new int[cols][rows];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            transposed[j][i] = matrix[i][j];
        }
    }
    System.out.println("Transposed matrix:");
    printMatrix(transposed);
}
```

OUTPUT

```
Original matrix:
1 2 3
4 5 6
Transposed matrix:
1 4
2 5
3 6
```

- The transpose turns a matrix's **rows into columns** and its columns into rows: `T[j][i] = M[i][j]`.
- The transpose of a 2×3 matrix is 3×2: `new int[cols][rows]`.

Transpose: Copying Cell by Cell



`rows = 2`, `cols = 3`; a new 3×2 array is created for the transpose (cells 0).

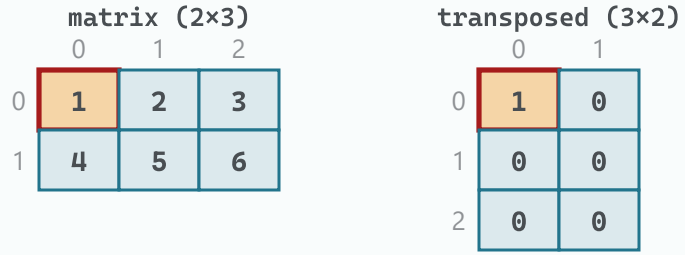
Step 1 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = –` `j = –`

Transpose: Copying Cell by Cell



`transposed[0][0] = matrix[0][0]` = 1 · the matrix's **row** becomes the transpose's **column**.

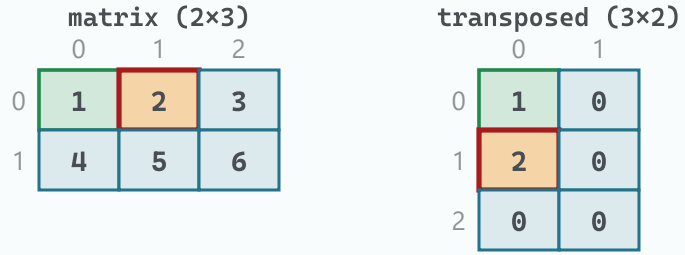
Step 2 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = 0` `j = 0`

Transpose: Copying Cell by Cell



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

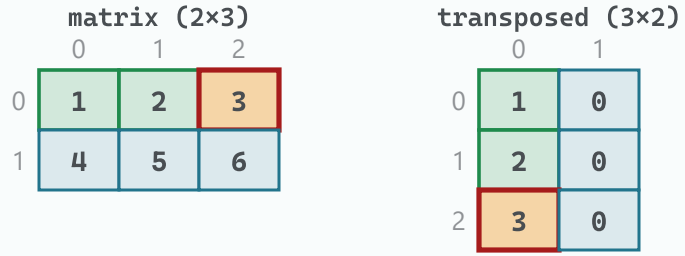
Step 3 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = 0` `j = 1`

Transpose: Copying Cell by Cell



`transposed[2][0] = matrix[0][2]` = 3

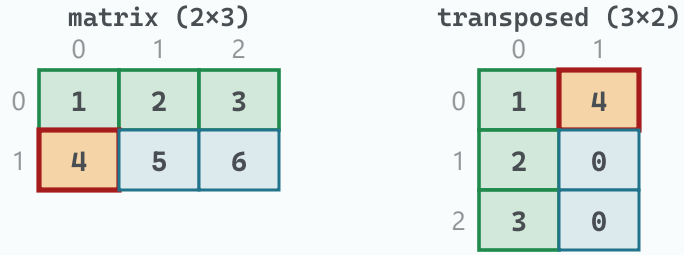
Step 4 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = 0` `j = 2`

Transpose: Copying Cell by Cell



`transposed[0][1] = matrix[1][0]` = 4 · the matrix's **row** becomes the transpose's **column**.

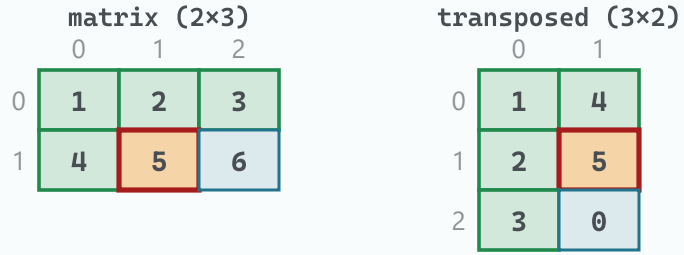
Step 5 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = 1` `j = 0`

Transpose: Copying Cell by Cell



`transposed[1][1] = matrix[1][1]` = 5

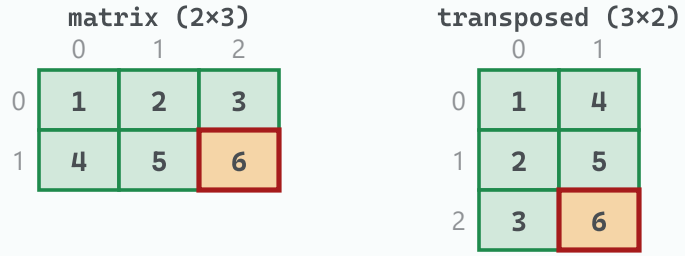
Step 6 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = 1` `j = 1`

Transpose: Copying Cell by Cell



`transposed[2][1] = matrix[1][2]` = 6

Step 7 / 7

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        transposed[j][i] = matrix[i][j];
    }
}
```

VARIABLES

`i = 1` `j = 2`

Helper Method: `printMatrix`

```
public static void printMatrix(int[][] matrix) {
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            System.out.print(matrix[i][j] + " ");
        }
        System.out.println();
    }
}
```

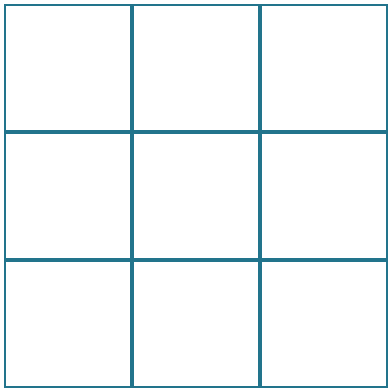
- Printing has been moved into a **helper method**; `main` calls it twice: first for the matrix, then for the transpose.
- The parameter `int[][] matrix`: works with a 2D array of any size, since the row count `matrix.length` and the column count `matrix[i].length` **are read from the array**.
- The method receives not a copy of the array but its **reference**; the call is cheap even for a big matrix.
- With a fixed `3`, one of the 2×3 and 3×2 matrices would print wrong or throw an `ArrayIndexOutOfBoundsException`.

PRACTICE: TIC TAC TOE

Tic Tac Toe

- The game uses a 3×3 board: `char[][] board = new char[3][3];`
- Two players (X and O) take turns; each move is a single assignment: `board[row][col] = player`.
- Whoever places **3 of their marks** in one row, column or diagonal wins.
- If no empty cell is left, the game is a draw.

The board on the right shows the array assignment run by each click.



Player **X** to move. Click a cell.

new game

TicTacToe: main

```java
import java.util.Scanner;

public class TicTacToe {
    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        char[][] board = new char[3][3]; // 3x3 board
        char player = 'X'; // X moves first
        boolean running = true;
        initBoard(board); // all cells empty
        while (running) {
            printBoard(board);
            makeMove(board, player);
            running = isRunning(board, player);
            // switch players
            player = (player == 'X') ? 'O' : 'X';
        }
        printBoard(board);
    }
    // ... methods
}
```

- The work is split into **methods**: fill, print, read a move, check.
- Methods receive a **copy of the board's reference**: the change made by `makeMove` is visible in `main`.
- The methods must be `static`; otherwise:

```
error: non-static method
initBoard(char[][]) cannot be
referenced from a static
context
```

TicTacToe: Filling and Printing the Board

```java
public static void initBoard(char[][] board) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            board[i][j] = ' ';
        }
    }
}

public static void printBoard(char[][] board) {
    System.out.println("-------------");
    for (int i = 0; i < 3; i++) {
        System.out.print("| ");
        for (int j = 0; j < 3; j++) {
            System.out.print(board[i][j] + " | ");
        }
        System.out.println("\n-------------");
    }
}
```

THE EMPTY BOARD

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

- `new char[3][3]` cells start as `'\u0000'`; so that the board prints neatly and empty cells can be checked, they are first filled with `' '` (a space).
- `"\n-------------"`: the line break and the separator in a single `println`.

TicTacToe: Reading a Move

```java
static Scanner scanner = new Scanner(System.in);

public static void makeMove(char[][] board, char player) {
    int row, col;
    do {
        System.out.print("Player " + player
                + ", enter row and column (1-3): ");
        row = scanner.nextInt() - 1;
        col = scanner.nextInt() - 1;
    } while (row < 0 || row > 2 || col < 0 || col > 2
            || board[row][col] != ' ');
    board[row][col] = player;
}
```

0-based index

The user enters 1–3; `- 1` turns it into index 0–2.

do-while

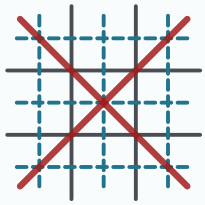
While an invalid or **taken** cell is chosen, the prompt repeats; a valid choice ends the loop.

Short circuit

The bounds checks come **before** `board[row][col]`: `||` stops at the first true condition, so no out-of-bounds access happens.

TicTacToe: Is There a Winner?

```java
public static boolean isRunning(char[][] board, char player) {
    // check the winning lines
    if ((board[0][0] == player && board[0][1] == player && board[0][2] == player) ||
        (board[1][0] == player && board[1][1] == player && board[1][2] == player) ||
        (board[2][0] == player && board[2][1] == player && board[2][2] == player) ||
        (board[0][0] == player && board[1][0] == player && board[2][0] == player) ||
        (board[0][1] == player && board[1][1] == player && board[2][1] == player) ||
        (board[0][2] == player && board[1][2] == player && board[2][2] == player) ||
        (board[0][0] == player && board[1][1] == player && board[2][2] == player) ||
        (board[0][2] == player && board[1][1] == player && board[2][0] == player)) {
        System.out.println("Congratulations, player " + player + " wins!");
        return false;
    }
    // ... draw check
```



3 rows + 3 columns + 2 diagonals

- The 8 winning lines are checked one by one: each needs **all 3 cells** to hold `player` (`&&`), and **one** line is enough (`||`).
- Only the player who just moved is checked: only they can have completed a new line.
- If there is a winner, the method returns `false` : "the game is not running".

TicTacToe: Is It a Draw?

```java
// isRunning, continued
// check for a draw
boolean draw = true;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (board[i][j] == ' ') {
            draw = false;
            break;
        }
    }
    if (!draw) {
        break;
    }
}
if (draw) {
    System.out.println("The game is a draw.");
    return false;
}
return true;
```

- First, a draw is **assumed** (`draw = true`); a single empty cell disproves it.
- The inner `break` leaves only the **inner** loop; the outer `if (!draw) break;` ends the outer loop too.
- With no winner and an empty cell left, it returns `true` : the game goes on.
- Win is checked before draw: the last move may fill the board and win at the same time.

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
▶ boolean draw = true;
2 for (int i = 0; i < 3; i++) {
3     for (int j = 0; j < 3; j++) {
4         if (board[i][j] == ' ') {
5             draw = false;
6             break;
7         }
8     }
9     if (!draw) {
10        break;
11    }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

draw = **true**

A draw is assumed first. The board has one empty cell: (1, 2).

Step 1 / 13

VARIABLES

`draw = true`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
▶          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

draw = **true**

i = 0, j = 0

(0, 0) = 'X': not empty, continue.

Step 2 / 13

VARIABLES

`draw = true` `i = 0` `j = 0`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
▶          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

(0, 1) = 'O': not empty.

draw = **true**

i = 0, j = 1

Step 3 / 13

VARIABLES

`draw = true` `i = 0` `j = 1`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
▶          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

draw = **true**

i = 0, j = 2

(0, 2) = 'X': row 0 is full; the inner loop ends on its own.

Step 4 / 13

VARIABLES

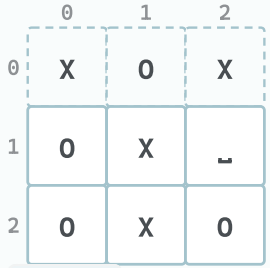
`draw = true` `i = 0` `j = 2`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
▶      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```



draw = **true**

i = 0

`!draw` is false: the outer loop goes on.

Step 5 / 13

VARIABLES

`draw = true` `i = 0`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
▶          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

i = 1: (1, 0) = 'O'.

draw = **true**

i = 1, j = 0

Step 6 / 13

VARIABLES

`draw = true` `i = 1` `j = 0`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
▶          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

(1, 1) = 'X'.

draw = **true**

i = 1, j = 1

Step 7 / 13

VARIABLES

`draw = true` `i = 1` `j = 1`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
▶          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

draw = **true**

i = 1, j = 2

(1, 2) = ' ': **empty cell found**.

Step 8 / 13

VARIABLES

`draw = true` `i = 1` `j = 2`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

draw = **false**

i = 1, j = 2

The assumption fails: `draw = false`.

Step 9 / 13

VARIABLES

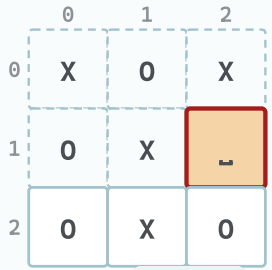
`draw = false` `i = 1` `j = 2`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (board[i][j] == ' ') {
5              draw = false;
▶              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```



draw = **false**

i = 1, j = 2

The inner `break` : it leaves only the **inner** loop; the outer loop is still running.

Step 10 / 13

VARIABLES

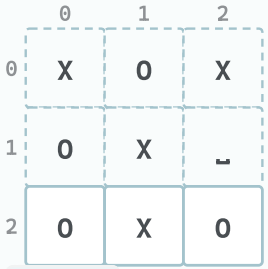
`draw = false` `i = 1` `j = 2`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```



draw = **false**

i = 1

`!draw` true.

Step 11 / 13

VARIABLES

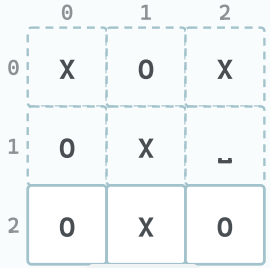
`draw = false` `i = 1`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
▶          break;
11     }
12 }
13 System.out.println(draw ? "draw" : "continue");
```



draw = **false**

i = 1

Outer `break`: row 2 is never checked.

Step 12 / 13

VARIABLES

`draw = false` `i = 1`

OUTPUT

Draw: Two `break`s, Two Loops

Board: `X O X` · `O X ␣` · `O X O` (␣: space character)

```
1  boolean draw = true;
2  for (int i = 0; i < 3; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (board[i][j] == ' ') {
5              draw = false;
6              break;
7          }
8      }
9      if (!draw) {
10         break;
11     }
12 }
▶  System.out.println(draw ? "draw" : "continue");
```

	0	1	2
0	X	O	X
1	O	X	␣
2	O	X	O

draw = **false**

There is an empty cell: `continue` is printed.

Step 13 / 13

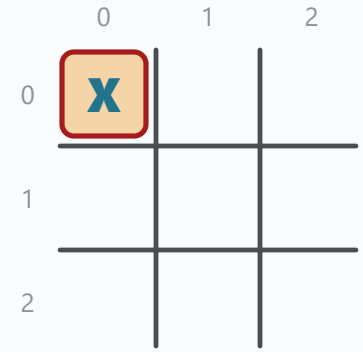
VARIABLES

`draw = false`

OUTPUT

```
continue
```

A Sample Game



X **1 1** → `board[0][0] = 'X'`. No winner, empty cells left → `true`; O's turn.

Step 1 / 6

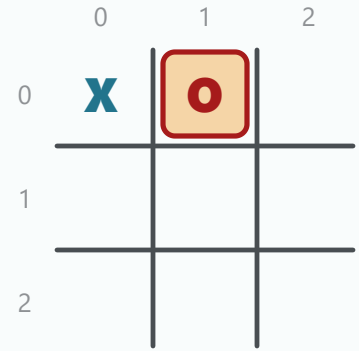
INPUT (ROW COLUMN)

```
1 1   1 2   2 2   2 2   1 3   3 3
```

FINAL OUTPUT OF THE PROGRAM

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

A Sample Game



O **1 2** → `board[0][1] = 'O'`. No winner, empty cells left → `true`; X's turn.

Step 2 / 6

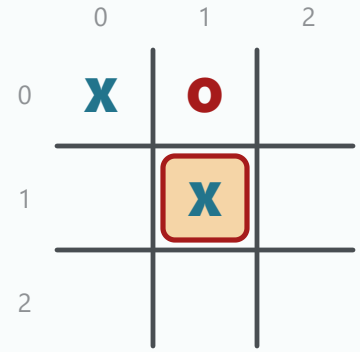
INPUT (ROW COLUMN)

```
1 1   1 2   2 2   2 2   1 3   3 3
```

FINAL OUTPUT OF THE PROGRAM

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

A Sample Game



X **2 2** → `board[1][1] = 'X'`. No winner, empty cells left → `true`; O's turn.

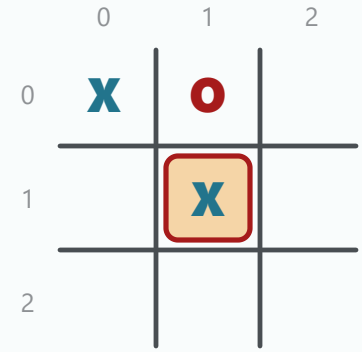
INPUT (ROW COLUMN)

```
1 1   1 2   2 2   2 2   1 3   3 3
```

FINAL OUTPUT OF THE PROGRAM

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

A Sample Game



○ **2 2**: `board[1][1]` is taken (`!= ' '`)
→ `do-while` condition is true, so the same prompt repeats.

Step 4 / 6

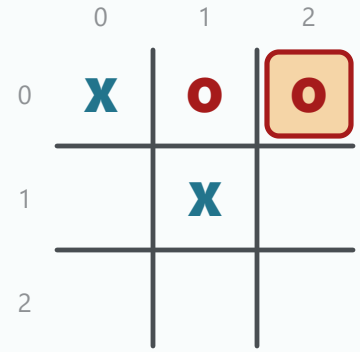
INPUT (ROW COLUMN)

```
1 1   1 2   2 2   2 2   1 3   3 3
```

FINAL OUTPUT OF THE PROGRAM

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

A Sample Game



O **1 3** → `board[0][2] = 'O'`. No winner, empty cells left → `true`; X's turn.

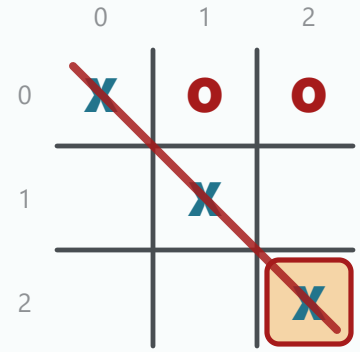
INPUT (ROW COLUMN)

```
1 1   1 2   2 2   2 2   1 3   3 3
```

FINAL OUTPUT OF THE PROGRAM

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

A Sample Game



X **3 3** → `board[2][2] = 'X'`. Main diagonal X → winning message; `isRunning` returns `false` and the loop ends.

INPUT (ROW COLUMN)

```
1 1   1 2   2 2   2 2   1 3   3 3
```

FINAL OUTPUT OF THE PROGRAM

```
Congratulations, player X wins!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

Summary

Topic	Key point
Array	A fixed number of elements of one type; indices 0 … length − 1
Creation	`new int[n]` (default values) · `{1, 2, 3}` (declaration only) · `new int[]{…}`
length	A field, no parentheses: `arr.length`; in 2D, `m.length` rows, `m[i].length` columns
Reference	The variable holds an address; `b = a` same array, `clone()` / `Arrays.copyOf` new array; `==` compares addresses
Memory	All arrays live on the heap, references on the stack; the garbage collector frees unreachable arrays
Errors	Out-of-bounds index → `ArrayIndexOutOfBoundsException`; in loops use `i < length`
Algorithms	Access O(1) · binary search O(log n) · linear search, reversal O(n) · matching, selection and bubble sort $O(n^2)$
Multiple dimensions	Arrays of arrays; nested loops; rows may differ in length

SUMMARY

Common Mistakes (1)

Short initialization after declaration

`a = {1, 2, 3};`
`error: illegal start of expression` → `a = new int[]{1, 2, 3};`

`length()` with parentheses

`a.length()`
`error: cannot find symbol … method length()` → `a.length`

Forgetting the size

`new int[]`
`error: array dimension missing`

Element of the wrong type

`int[] a = {1, 2.5, 3};`
`error: incompatible types: possible lossy conversion from double to int`

Variable declared inside a branch

As in the median example: `error: cannot find symbol … variable median`

`static` missing on a method

`error: non-static method … cannot be referenced from a static context`

SUMMARY

Common Mistakes (2)

Taking length as the last index

`a[a.length]` or `i <= a.length`: `ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3`

Array never created

`int[] a = null; a[0]`: `NullPointerException: Cannot load from int array because "a" is null`

Printing an array directly

`System.out.println(a)` → `[I@7ef20235`. For the contents use `Arrays.toString(a)`.

Mistaking assignment for a copy

`b = a`; then changing `b[0]` changes `a[0]` too. To copy: `a.clone()`.

`==` for comparing contents

Two arrays with equal elements give `false`. For the contents use `Arrays.equals(a, b)`.

Taking the first index as 1

Users see "element 1", but in code it is `arr[0]`: `"element " + (i + 1)`.

SUMMARY

Worked Problems (1)

```
static int mystery(int[] d) {
    int s = 0;
    for (int i = 0; i < d.length; i += 2) {
        s += d[i];
    }
    return s;
}
static void modify(int[] d, int x) {
    d[0] = 100;
    x = 100;
}
// main:
int[] d = {4, 8, 15, 16, 23, 42};
int[] a = {1, 2, 3};
int x = 1;
modify(a, x);
```

1. `mystery(d)` : what does it return? What does the method compute?
2. `d[d.length / 2]` = ?
3. `modify(a, x)` : what are `a[0]` and `x` ? Why do they differ?

▶ **Show answers**

SUMMARY

Worked Problems (2)

```java
int[][] m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int trace = 0;
for (int i = 0; i < m.length; i++) {
    trace += m[i][i];
}
// ---
int[] b = new int[5];
for (int i = 1; i < b.length; i++) {
    b[i] = b[i - 1] + i;
}
System.out.println(Arrays.toString(b));
// ---
int[] p = {3, 1, 2};
int[] q = p;
q[2] = q[0] + q[1];
System.out.println(Arrays.toString(p));
```

4. In the first snippet, what is `trace`? Which cells are added?
5. What does the second snippet print?
6. What does the third snippet print?

▶ **Show answers**

SUMMARY

Exercises

1. `int max(int[] arr)` : return the largest element. `{12, 7, 30, 5, 18}` → ?
2. `int countEven(int[] arr)` : count the even elements (with for-each). Same array → ?
3. `int indexOf(int[] arr, int target)` : index of the target, or −1. For 5 and 99 → ?
4. `int[][] add(int[][] a, int[][] b)` : matrix sum. `{{1, 2}, {3, 4}}` + `{{5, 6}, {7, 8}}` → ?
5. `int[] rowSums(int[][] m)` : each row's sum. `{{1, 2, 3}, {4, 5, 6}}` → ?
6. Add `isBoardFull` to Tic Tac Toe; use it for the draw check.

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: Arrays; The for Statement (enhanced for).
- Oracle, Java SE 21 API: `java.util.Arrays`.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (Chapter 10: Arrays).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 7: Arrays and ArrayLists).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (3.10 Arrays).
- A. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly (Chapter 7: Arrays and References).

The End

Chapter 3.3: Arrays

Previous: Chapter 3.2: Methods · Next: Chapter 3.4: Strings