Chapter 10: Arrays

Computer Fundamentals and Introduction to Programming

Definition | Index | Loops | Search | Matrices

Sercan KÜLCÜ · Giresun University

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/computer/slides/Chapter_10_Arrays.html

INTRODUCTION

Learning Outcomes

Creating

Declare and initialize arrays.

Access

Use indices and length.

Processing

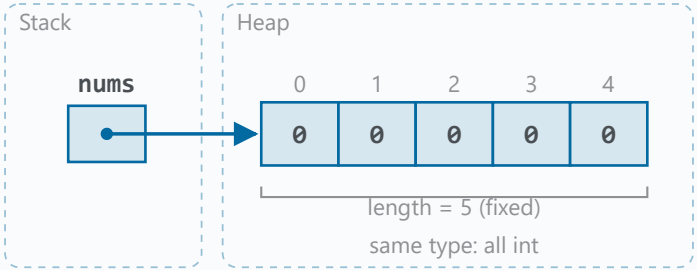
Traverse and transform arrays with loops.

Matrices

Work with rows and columns of two-dimensional arrays.

What Is an Array?

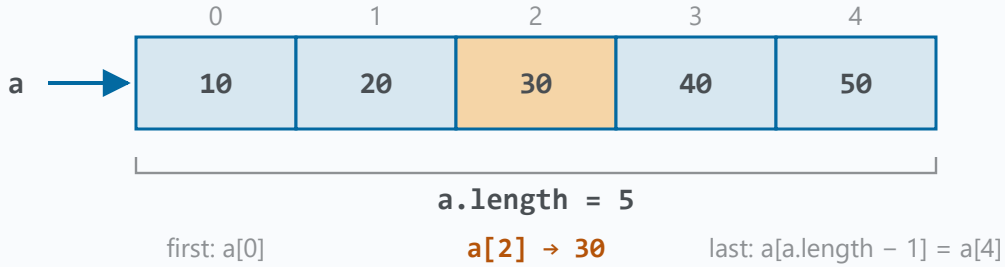
- Stores a fixed number of elements of one type in consecutive cells of **a single object**.
- Elements are accessed by a zero-based **index**: `nums[0]` … `nums[4]`.
- The length (`length`) cannot change after creation.
- An array variable holds not the elements but a **reference** (address) to the array object on the heap.



Variable → array object (reference arrow)

BASICS

Arrays and Indices

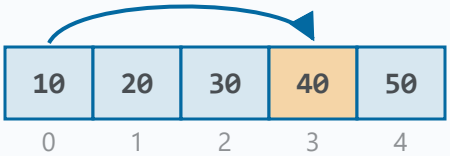


- The first index is **0**, the last is **length − 1** (5 elements → 0…4).
- The address of `a[i]` is computed directly (start + i × cell size), so access takes **constant time** (O(1)) without walking the array.

Strengths and Limits

Strength	Limit
Fast access by index	Fixed size
Consistent element type	Inserting in the middle is costly
Easy to process with loops	Out-of-bounds access is a run-time error
Regular layout in memory	Holds a single element type

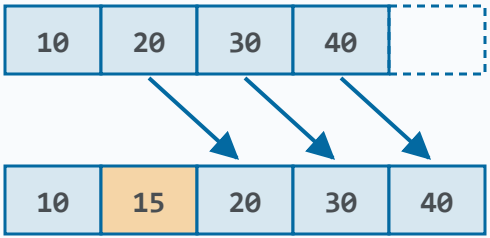
Index access: a direct jump



address = start + 3 × cell size

`a[3] → 40: one step, O(1)`

Insertion: 15 → index 1



the 3 on the right shift by one

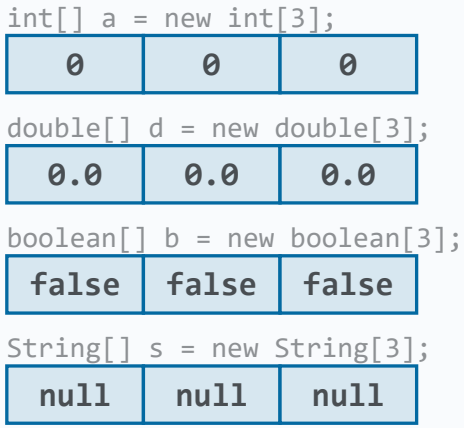
If full: new, larger array + copy (size is fixed)

DECLARATION

Creating Arrays

```
int[] nums = new int[5];
String[] names = {"Ada", "Can", "Ece"};
double[] readings;
readings = new double[10];
```

- Cells of an array created with `new` get the type's default value (right).
- With an initializer `{…}` the length comes from the number of values (names.length = 3).



Default initial values

Code Trace: Creating an Array



Declaration: only the reference variable `nums` appears on the stack; there is no array yet.

Step 1 / 6

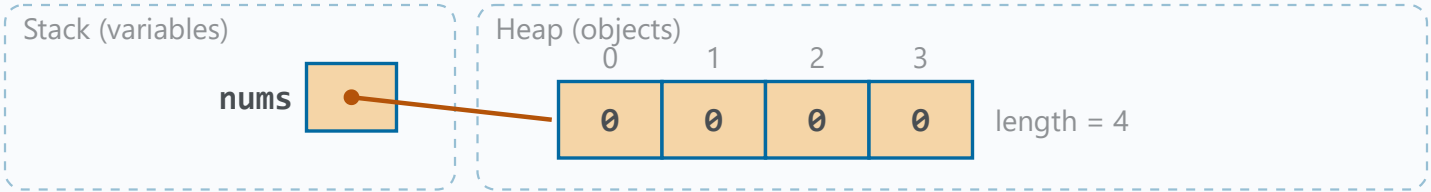
```
▶ int[] nums;
2 nums = new int[4];
3 nums[0] = 7;
4 nums[3] = 2;
5 String[] names = {"Ada", "Can", "Ece"};
6 System.out.println(nums.length + " "
7                    + names.length);
```

VARIABLES

`nums = ? (unassigned)`

OUTPUT

Code Trace: Creating an Array



`new int[4]` creates a 4-cell array on the heap, all **0**. The variable gets the array's **reference** (arrow).

Step 2 / 6

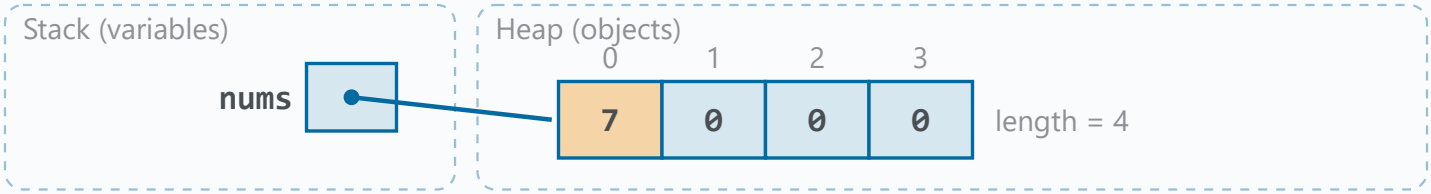
```
1 int[] nums;
▶ nums = new int[4];
3 nums[0] = 7;
4 nums[3] = 2;
5 String[] names = {"Ada", "Can", "Ece"};
6 System.out.println(nums.length + " "
7                    + names.length);
```

VARIABLES

`nums = [0, 0, 0, 0]`

OUTPUT

Code Trace: Creating an Array



`nums[0] = 7` : follow the arrow and write 7 into **cell 0**.

Step 3 / 6

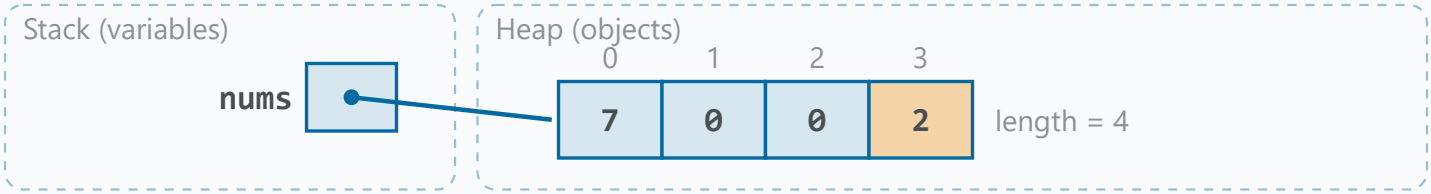
```
1 int[] nums;
2 nums = new int[4];
▶ nums[0] = 7;
4 nums[3] = 2;
5 String[] names = {"Ada", "Can", "Ece"};
6 System.out.println(nums.length + " "
7                    + names.length);
```

VARIABLES

`nums = [7, 0, 0, 0]`

OUTPUT

Code Trace: Creating an Array



`nums[3] = 2` : the last cell (index **length − 1 = 3**) is updated.

Step 4 / 6

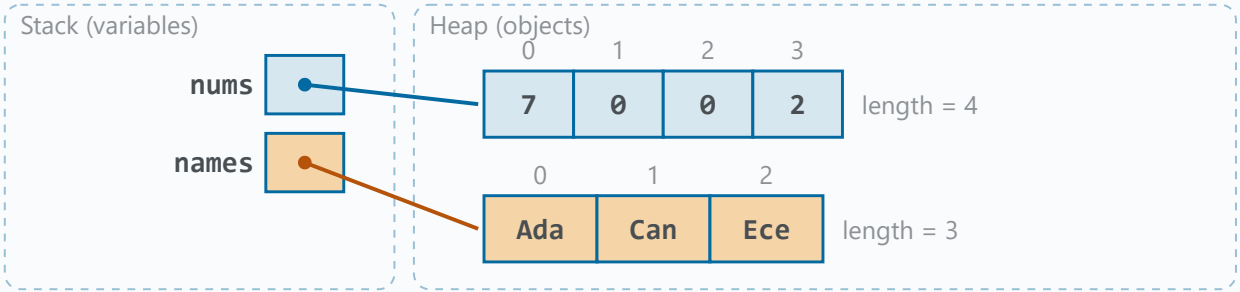
```
1 int[] nums;
2 nums = new int[4];
3 nums[0] = 7;
▶ nums[3] = 2;
5 String[] names = {"Ada", "Can", "Ece"};
6 System.out.println(nums.length + " "
7                    + names.length);
```

VARIABLES

`nums = [7, 0, 0, 2]`

OUTPUT

Code Trace: Creating an Array



Initializer `{…}` : 3 values → an array of length **3** is built and `names` refers to it.

Step 5 / 6

```
1 int[] nums;
2 nums = new int[4];
3 nums[0] = 7;
4 nums[3] = 2;
▶ String[] names = {"Ada", "Can", "Ece"};
6 System.out.println(nums.length + " "
7                    + names.length);
```

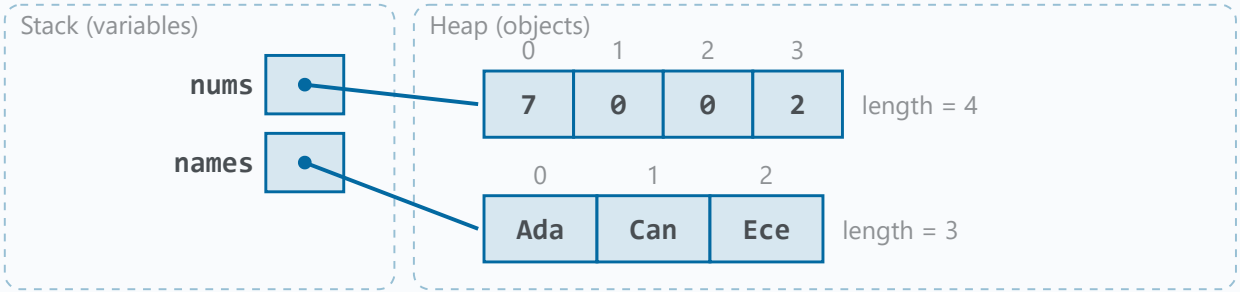
VARIABLES

`nums = [7, 0, 0, 2]`

`names = [Ada, Can, Ece]`

OUTPUT

Code Trace: Creating an Array



`length` is a field (no parentheses): **4** and **3** are printed.

Step 6 / 6

```
1 int[] nums;
2 nums = new int[4];
3 nums[0] = 7;
4 nums[3] = 2;
5 String[] names = {"Ada", "Can", "Ece"};
▶ System.out.println(nums.length + " "
▶                    + names.length);
```

VARIABLES

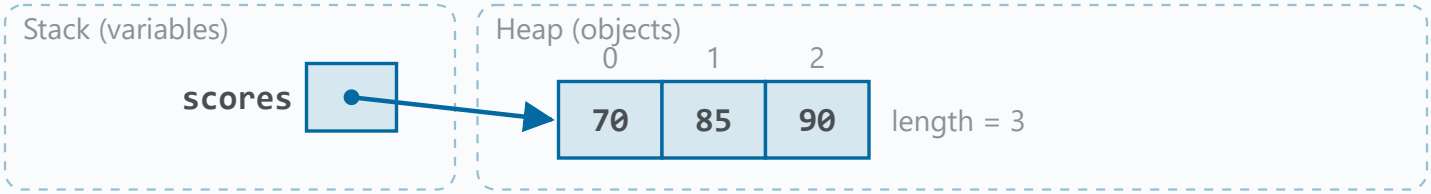
`nums = [7, 0, 0, 2]`

`names = [Ada, Can, Ece]`

OUTPUT

```
4 3
```

Reading and Writing Elements



A three-element array; indices **0, 1, 2**.

Step 1 / 5

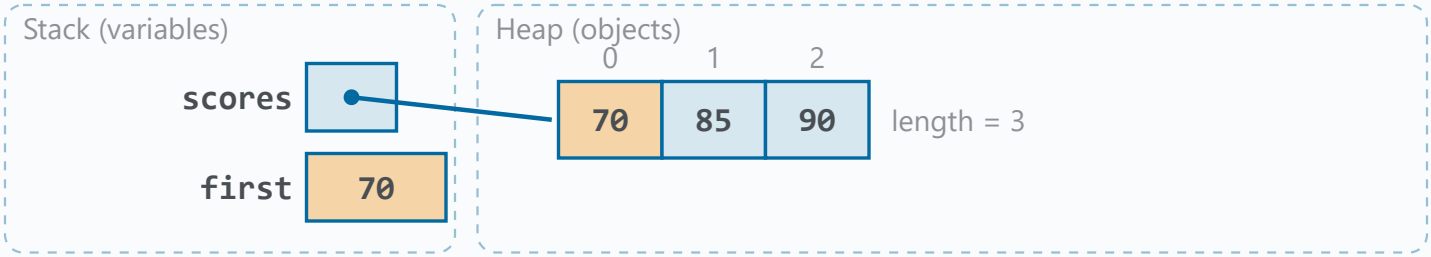
```
▶ int[] scores = {70, 85, 90};
2 int first = scores[0];
3 scores[1] = 88;
4 int last = scores[scores.length - 1];
5 System.out.println(first + " " + last);
```

VARIABLES

`scores = [70, 85, 90]`

OUTPUT

Reading and Writing Elements



Read: `scores[0]` gives the value in the first cell → `first = 70`.

Step 2 / 5

```
1 int[] scores = {70, 85, 90};
▶ int first = scores[0];
3 scores[1] = 88;
4 int last = scores[scores.length - 1];
5 System.out.println(first + " " + last);
```

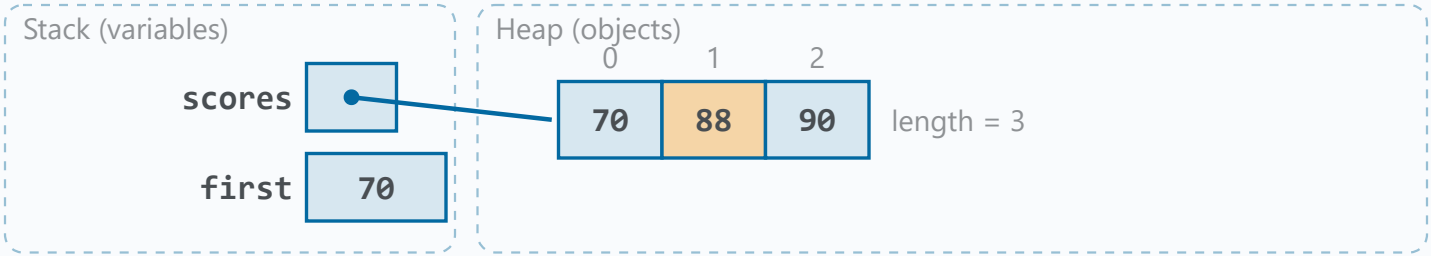
VARIABLES

`scores = [70, 85, 90]`

`first = 70`

OUTPUT

Reading and Writing Elements



Write: `scores[1]` on the left of an assignment is the cell itself; 85 is overwritten with **88**.

Step 3 / 5

```
1 int[] scores = {70, 85, 90};
2 int first = scores[0];
▶ scores[1] = 88;
4 int last = scores[scores.length - 1];
5 System.out.println(first + " " + last);
```

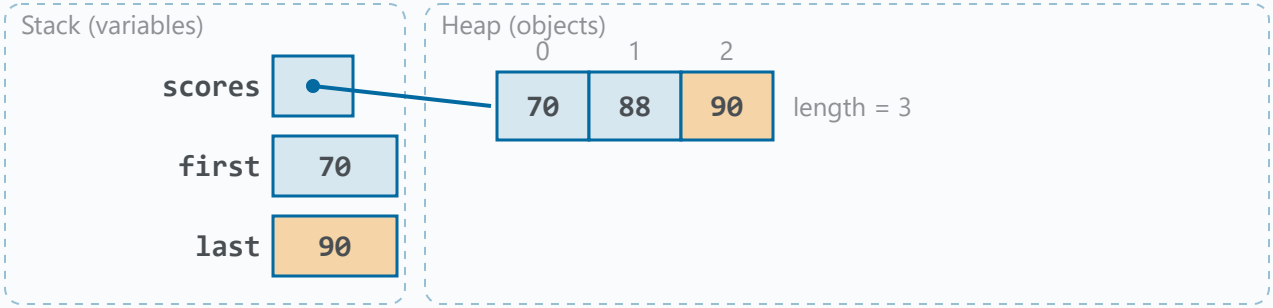
VARIABLES

`scores = [70, 88, 90]`

`first = 70`

OUTPUT

Reading and Writing Elements



`scores.length - 1` = 3 − 1 = **2** → the last element is **90**. The code stays correct if the size changes.

Step 4 / 5

```
1 int[] scores = {70, 85, 90};
2 int first = scores[0];
3 scores[1] = 88;
▶ int last = scores[scores.length - 1];
5 System.out.println(first + " " + last);
```

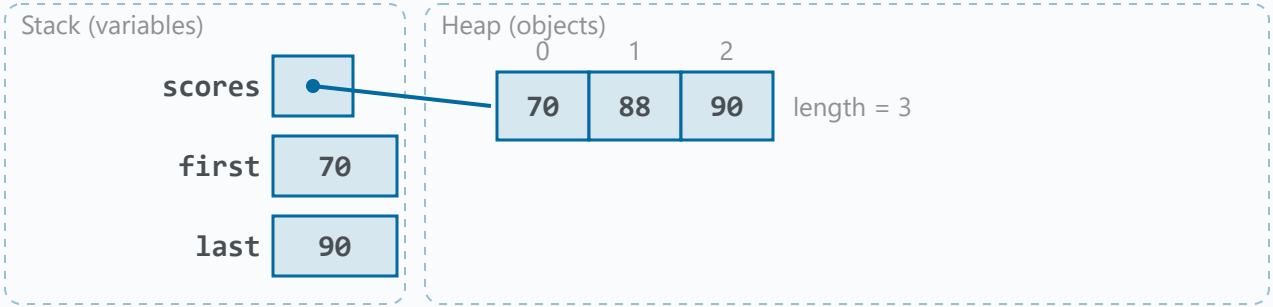
VARIABLES

`scores = [70, 88, 90]`

`first = 70` `last = 90`

OUTPUT

Reading and Writing Elements



The two values are printed with a space between them.

Step 5 / 5

```
1 int[] scores = {70, 85, 90};
2 int first = scores[0];
3 scores[1] = 88;
4 int last = scores[scores.length - 1];
▶ System.out.println(first + " " + last);
```

VARIABLES

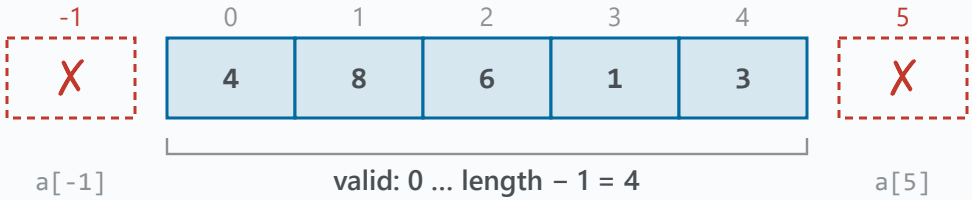
`scores = [70, 88, 90]`

`first = 70` `last = 90`

OUTPUT

```
70 90
```

Out-of-Bounds Errors



a.length = 5 but NO a[5] → ArrayIndexOutOfBoundsException

- Valid indices run from **0** to **length − 1**; negative indices and `length` itself are invalid.
- The compiler doesn't catch this; the program stops **at run time** with `ArrayIndexOutOfBoundsException`.

The loop condition must be `i < arr.length`; `<=` runs one pass too many.

Code Trace: One Pass Too Many (`<=`)



An array of length **3**: valid indices 0, 1, 2.

Step 1 / 9

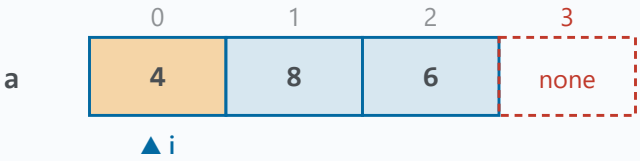
```
▶ int[] a = {4, 8, 6};
2 for (int i = 0; i <= a.length; i++) {
3     System.out.println(a[i]);
4 }
```

VARIABLES

`a = [4, 8, 6]`

OUTPUT

Code Trace: One Pass Too Many (`<=`)



0 <= 3 → **true**, enter the body.

Step 2 / 9

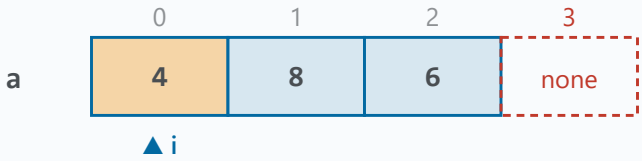
```
1 int[] a = {4, 8, 6};
▶ for (int i = 0; i <= a.length; i++) {
3     System.out.println(a[i]);
4 }
```

VARIABLES

`a = [4, 8, 6]` `i = 0`

OUTPUT

Code Trace: One Pass Too Many (`<=`)



`a[0]` = 4 is printed.

Step 3 / 9

```
1 int[] a = {4, 8, 6};
2 for (int i = 0; i <= a.length; i++) {
      System.out.println(a[i]);
4 }
```

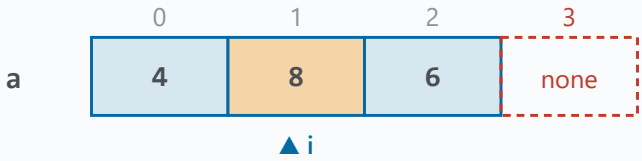
VARIABLES

`a = [4, 8, 6]` `i = 0`

OUTPUT

```
4
```

Code Trace: One Pass Too Many (`<=`)



`i++` → **1 <= 3** → **true**, enter the body.

Step 4 / 9

```
1 int[] a = {4, 8, 6};
▶ for (int i = 0; i <= a.length; i++) {
3     System.out.println(a[i]);
4 }
```

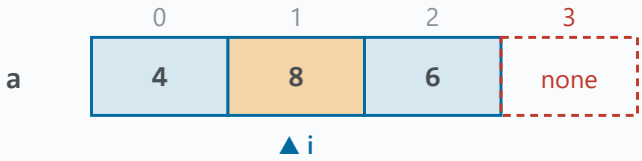
VARIABLES

`a = [4, 8, 6]` `i = 1`

OUTPUT

```
4
```

Code Trace: One Pass Too Many (`<=`)



`a[1]` = 8 is printed.

Step 5 / 9

```
1 int[] a = {4, 8, 6};
2 for (int i = 0; i <= a.length; i++) {
      System.out.println(a[i]);
4 }
```

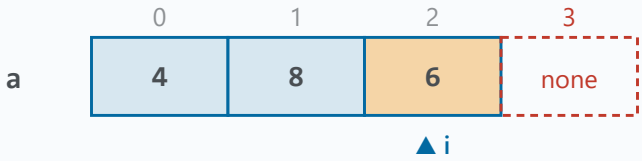
VARIABLES

`a = [4, 8, 6]` `i = 1`

OUTPUT

```
4
8
```

Code Trace: One Pass Too Many (`<=`)



`i++` → **2 <= 3** → **true**, enter the body.

Step 6 / 9

```
1 int[] a = {4, 8, 6};
▶ for (int i = 0; i <= a.length; i++) {
3     System.out.println(a[i]);
4 }
```

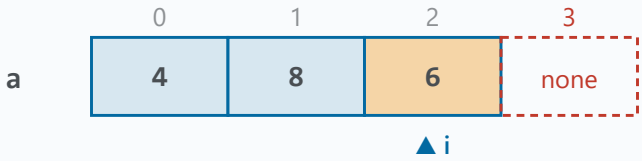
VARIABLES

`a = [4, 8, 6]` `i = 2`

OUTPUT

```
4
8
```

Code Trace: One Pass Too Many (`<=`)



`a[2]` = 6 is printed.

Step 7 / 9

```
1 int[] a = {4, 8, 6};
2 for (int i = 0; i <= a.length; i++) {
▶     System.out.println(a[i]);
4 }
```

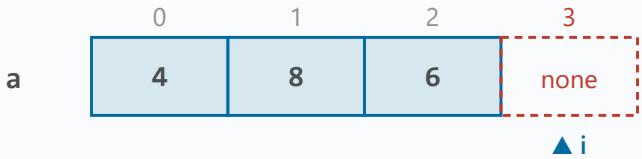
VARIABLES

`a = [4, 8, 6]` `i = 2`

OUTPUT

```
4
8
6
```

Code Trace: One Pass Too Many (`<=`)



`i++` → **3 <= 3** → **true**. Careful: 3 is an invalid index, yet the condition holds!

Step 8 / 9

```
1 int[] a = {4, 8, 6};
▶ for (int i = 0; i <= a.length; i++) {
3     System.out.println(a[i]);
4 }
```

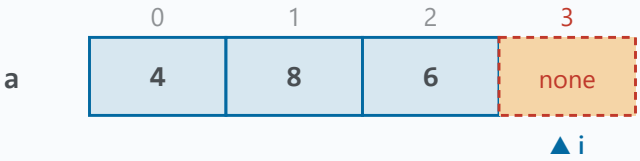
VARIABLES

`a = [4, 8, 6]` `i = 3`

OUTPUT

```
4
8
6
```

Code Trace: One Pass Too Many (`<=`)



`a[3]` : no such cell → the program **stops with an error**. Fix: make the condition `i < a.length` .

Step 9 / 9

```
1 int[] a = {4, 8, 6};
2 for (int i = 0; i <= a.length; i++) {
▶     System.out.println(a[i]);
4 }
```

VARIABLES

`a = [4, 8, 6]` `i = 3`

OUTPUT

```
4
8
6
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsEx
ception: Index 3 out of bounds
for length 3
```

Traversal with a Classic `for`



The array is ready; the index `i` will start at 0.

Step 1 / 8

```
▶ int[] scores = {70, 88, 90};
2 for (int i = 0; i < scores.length; i++) {
3   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

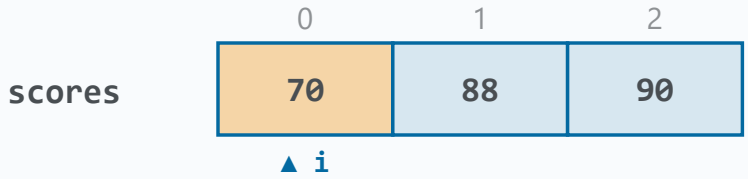
VARIABLES

`scores = [70, 88, 90]`

OUTPUT

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`int i = 0` ; **0 < 3 → true**

Step 2 / 8

```
1 int[] scores = {70, 88, 90};
▶ for (int i = 0; i < scores.length; i++) {
3   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

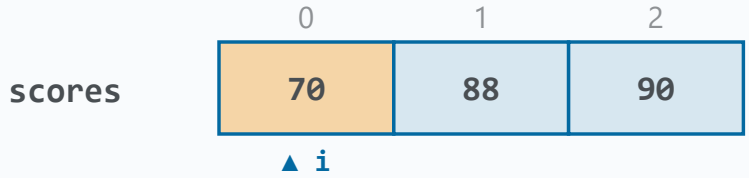
VARIABLES

`scores = [70, 88, 90]` `i = 0`

OUTPUT

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`scores[0]` = 70; `%d: %d` format gives **0: 70**.

Step 3 / 8

```
1 int[] scores = {70, 88, 90};
2 for (int i = 0; i < scores.length; i++) {
▶   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

VARIABLES

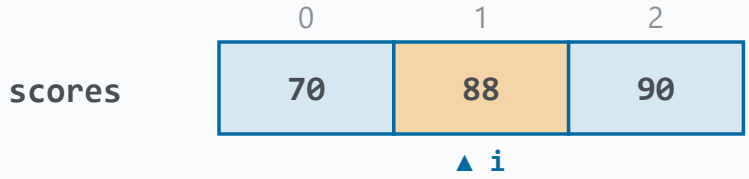
`scores = [70, 88, 90]` `i = 0`

OUTPUT

```
0: 70
```

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`i++` → **1 < 3 → true**

Step 4 / 8

```
1 int[] scores = {70, 88, 90};
▶ for (int i = 0; i < scores.length; i++) {
3   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

VARIABLES

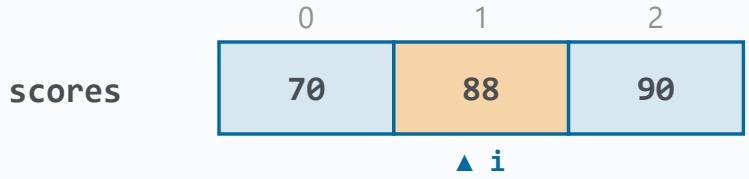
`scores = [70, 88, 90]` `i = 1`

OUTPUT

```
0: 70
```

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`scores[1]` = 88; `%d: %d` format gives **1: 88**.

Step 5 / 8

```
1 int[] scores = {70, 88, 90};
2 for (int i = 0; i < scores.length; i++) {
▶   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

VARIABLES

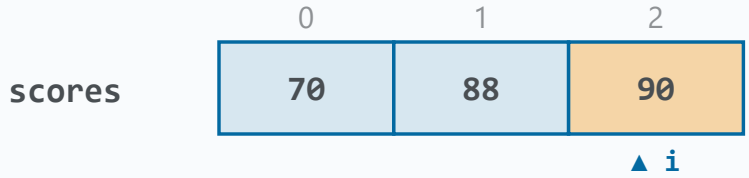
`scores = [70, 88, 90]` `i = 1`

OUTPUT

```
0: 70
1: 88
```

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`i++` → **2 < 3 → true**

Step 6 / 8

```
1 int[] scores = {70, 88, 90};
▶ for (int i = 0; i < scores.length; i++) {
3   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

VARIABLES

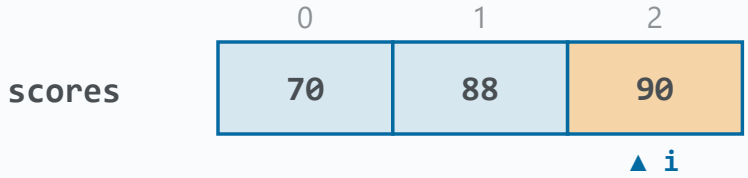
`scores = [70, 88, 90]` `i = 2`

OUTPUT

```
0: 70
1: 88
```

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`scores[2]` = 90; `%d: %d` format gives **2: 90**.

Step 7 / 8

```
1 int[] scores = {70, 88, 90};
2 for (int i = 0; i < scores.length; i++) {
▶   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

VARIABLES

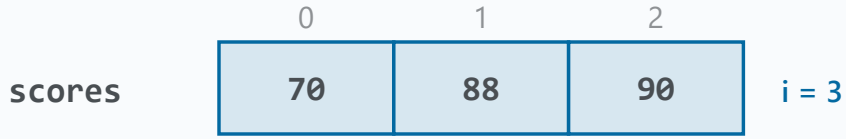
`scores = [70, 88, 90]` `i = 2`

OUTPUT

```
0: 70
1: 88
2: 90
```

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

Traversal with a Classic `for`



`i++` → **3 < 3** → **false** → the loop ends; since i = length, scores[i] is never accessed.

Step 8 / 8

```
1 int[] scores = {70, 88, 90};
▶ for (int i = 0; i < scores.length; i++) {
3   System.out.printf("%d: %d%n", i, scores[i]);
4 }
```

VARIABLES

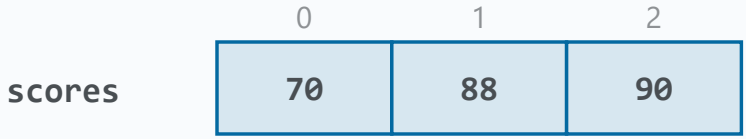
`scores = [70, 88, 90]` `i = 3`

OUTPUT

```
0: 70
1: 88
2: 90
```

Use a classic `for` when you need the index (printing positions, looking at a neighbor, writing to a cell).

The Enhanced `for` (for-each)



The enhanced for hands out the elements in order; there is no index.

Step 1 / 12

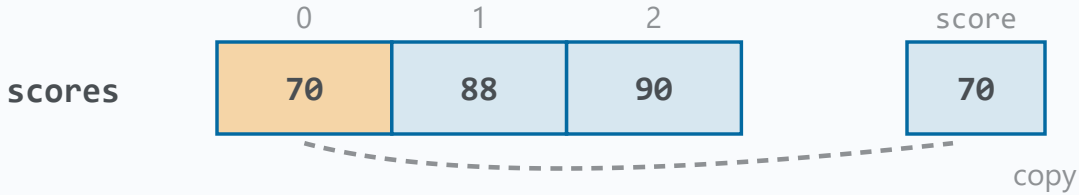
```
▶ int[] scores = {70, 88, 90};
2 for (int score : scores) {
3     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

`scores = [70, 88, 90]`

OUTPUT

The Enhanced `for` (for-each)



Pass 1: `score` gets a **copy** of `scores[0]` (70).

Step 2 / 12

```
1 int[] scores = {70, 88, 90};
▶ for (int score : scores) {
3     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

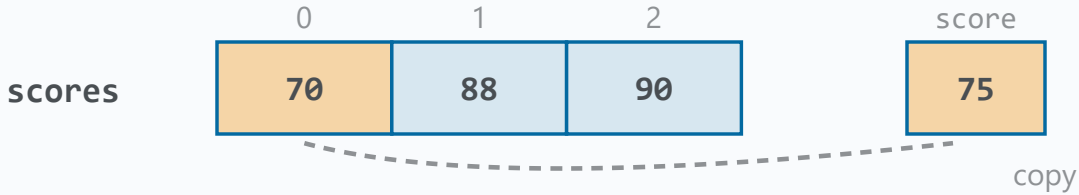
VARIABLES

`scores = [70, 88, 90]`

`score = 70`

OUTPUT

The Enhanced `for` (for-each)



`score = 75` ; the array cell stays **70**: only the copy changed.

Step 3 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
▶     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

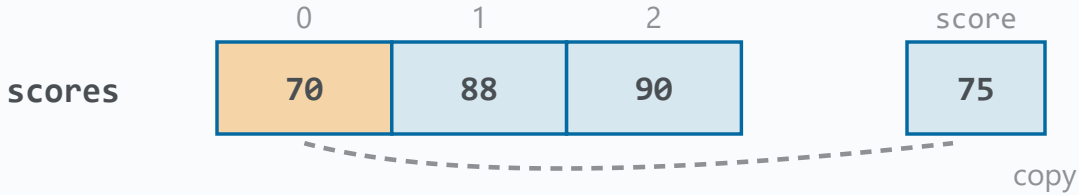
VARIABLES

`scores = [70, 88, 90]`

`score = 75`

OUTPUT

The Enhanced `for` (for-each)



75 is printed.

Step 4 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
3     score = score + 5;
  ▶   System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

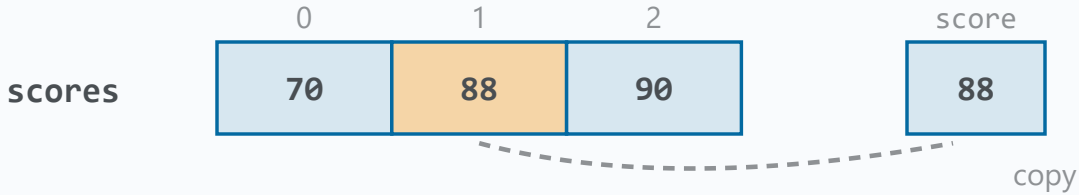
`scores = [70, 88, 90]`

`score = 75`

OUTPUT

```
75
```

The Enhanced `for` (for-each)



Pass 2: `score` gets a **copy** of `scores[1]` (88).

Step 5 / 12

```
1 int[] scores = {70, 88, 90};
▶ for (int score : scores) {
3     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

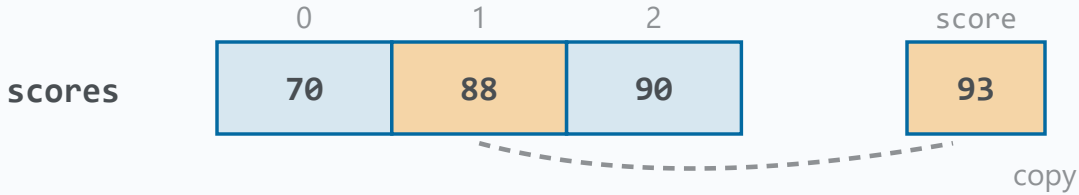
`scores = [70, 88, 90]`

`score = 88`

OUTPUT

```
75
```

The Enhanced `for` (for-each)



`score = 93` ; the array cell stays **88**: only the copy changed.

Step 6 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
▶     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

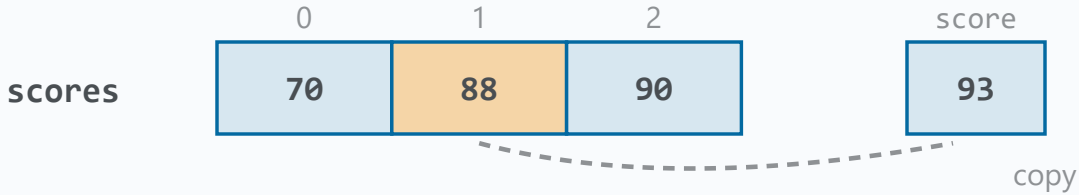
`scores = [70, 88, 90]`

`score = 93`

OUTPUT

```
75
```

The Enhanced `for` (for-each)



93 is printed.

Step 7 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
3     score = score + 5;
▶     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

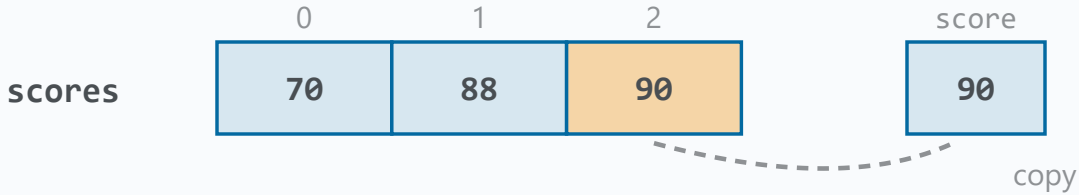
`scores = [70, 88, 90]`

`score = 93`

OUTPUT

```
75
93
```

The Enhanced `for` (for-each)



Pass 3: `score` gets a **copy** of `scores[2]` (90).

Step 8 / 12

```
1 int[] scores = {70, 88, 90};
▶ for (int score : scores) {
3     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

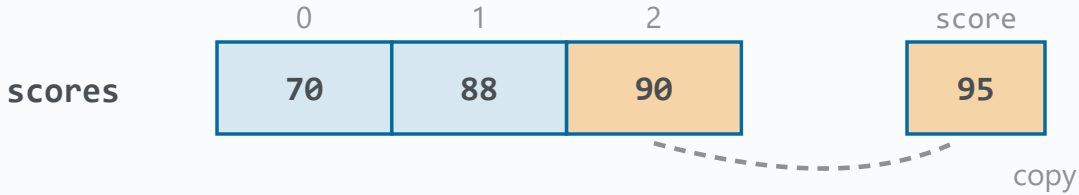
`scores = [70, 88, 90]`

`score = 90`

OUTPUT

```
75
93
```

The Enhanced `for` (for-each)



`score = 95` ; the array cell stays **90**: only the copy changed.

Step 9 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
▶     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

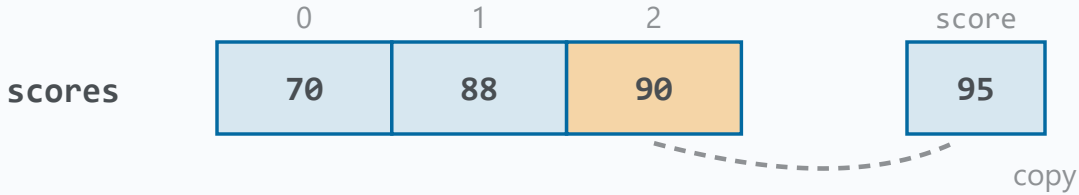
`scores = [70, 88, 90]`

`score = 95`

OUTPUT

```
75
93
```

The Enhanced `for` (for-each)



95 is printed.

Step 10 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
3     score = score + 5;
  ▶   System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

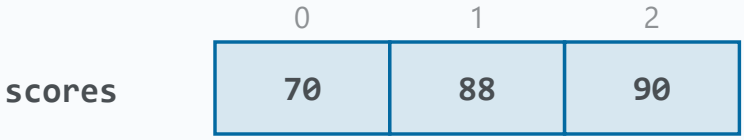
`scores = [70, 88, 90]`

`score = 95`

OUTPUT

```
75
93
95
```

The Enhanced `for` (for-each)



No elements left → the loop ends; `score` goes out of scope.

Step 11 / 12

```
1 int[] scores = {70, 88, 90};
▶ for (int score : scores) {
3     score = score + 5;
4     System.out.println(score);
5 }
6 System.out.println(scores[0]);
```

VARIABLES

`scores = [70, 88, 90]`

OUTPUT

```
75
93
95
```

The Enhanced `for` (for-each)



`scores[0]` is still **70**: in an enhanced for, assigning to the loop variable doesn't change the array.

Step 12 / 12

```
1 int[] scores = {70, 88, 90};
2 for (int score : scores) {
3     score = score + 5;
4     System.out.println(score);
5 }
▶ System.out.println(scores[0]);
```

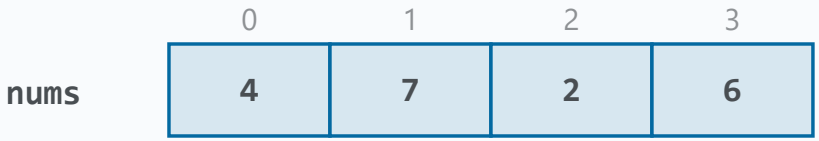
VARIABLES

`scores = [70, 88, 90]`

OUTPUT

```
75
93
95
70
```

Sum and Average



A four-element array.

Step 1 / 10

```
▶ int[] nums = {4, 7, 2, 6};
2 int sum = 0;
3 for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

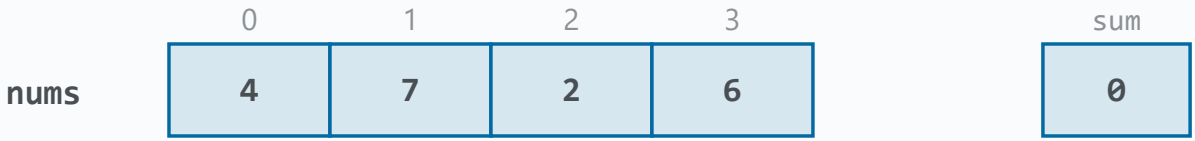
VARIABLES

`nums = [4, 7, 2, 6]`

OUTPUT

OPERATIONS · STEP THROUGH THE CODE

Sum and Average



The accumulator starts at `sum = 0`.

Step 2 / 10

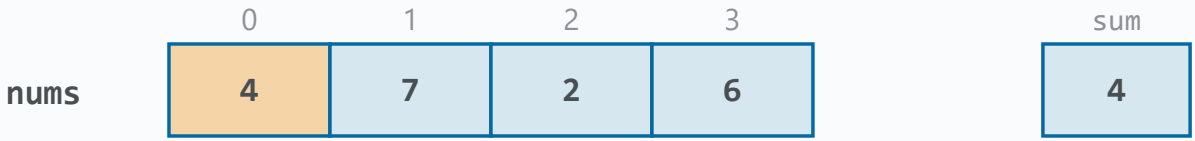
```
1 int[] nums = {4, 7, 2, 6};
▶ int sum = 0;
3 for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 0`

OUTPUT

Sum and Average



`value = 4` → `sum += value` : 0 + 4 = **4**.

Step 3 / 10

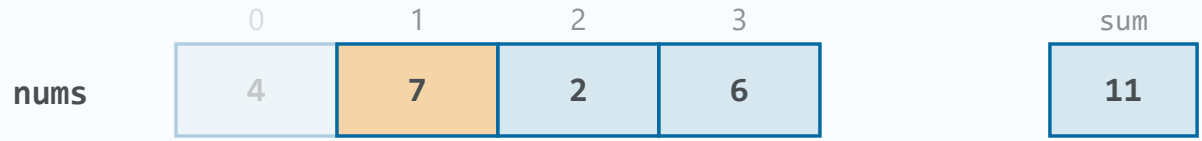
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
▶ for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 4` `value = 4`

OUTPUT

Sum and Average



`value = 7` → `sum += value` : 4 + 7 = **11**.

Step 4 / 10

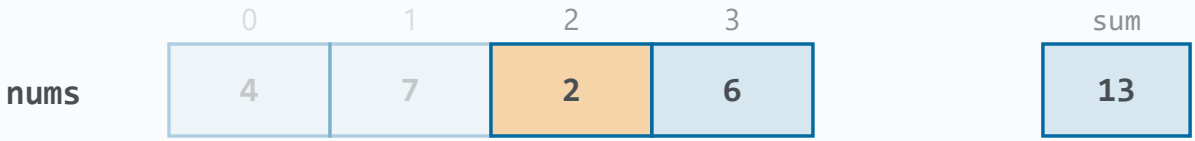
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
▶ for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 11` `value = 7`

OUTPUT

Sum and Average



`value = 2` → `sum += value` : 11 + 2 = **13**.

Step 5 / 10

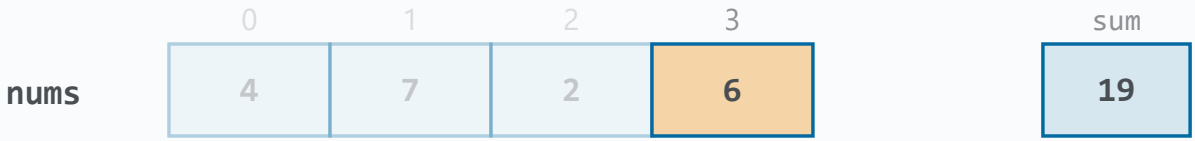
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
▶ for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 13` `value = 2`

OUTPUT

Sum and Average



`value = 6` → `sum += value` : 13 + 6 = **19**.

Step 6 / 10

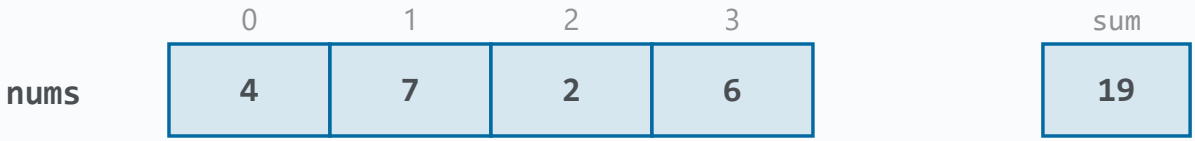
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
▶ for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 19` `value = 6`

OUTPUT

Sum and Average



The loop is done. `avg` starts at 0 (a safe value for an empty array).

Step 7 / 10

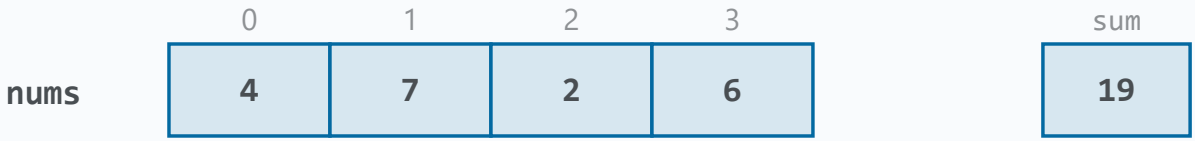
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
3 for (int value : nums) sum += value;
▶ double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 19` `avg = 0.0`

OUTPUT

Sum and Average



4 > 0 → **true**: the array isn't empty, so dividing is safe.

Step 8 / 10

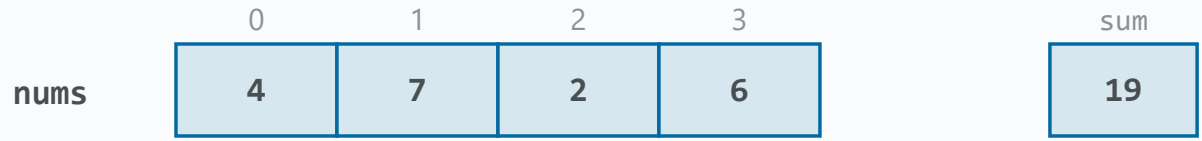
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
3 for (int value : nums) sum += value;
4 double avg = 0;
▶ if (nums.length > 0)
6     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 19` `avg = 0.0`

OUTPUT

Sum and Average



`(double) 19 / 4` = **4.75**. Without the cast, integer division would give 4.

Step 9 / 10

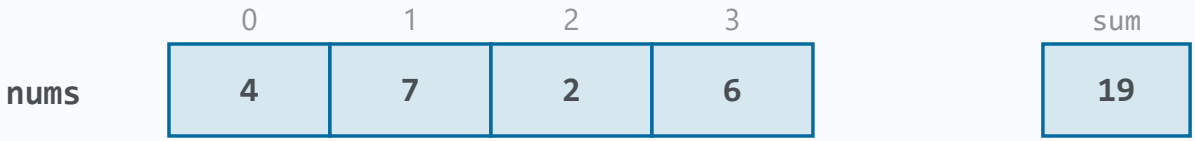
```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
3 for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
▶     avg = (double) sum / nums.length;
7 System.out.println(sum + " " + avg);
```

VARIABLES

`nums = [4, 7, 2, 6]` `sum = 19` `avg = 4.75`

OUTPUT

Sum and Average



The results are printed.

Step 10 / 10

```
1 int[] nums = {4, 7, 2, 6};
2 int sum = 0;
3 for (int value : nums) sum += value;
4 double avg = 0;
5 if (nums.length > 0)
6     avg = (double) sum / nums.length;
▶ System.out.println(sum + " " + avg);
```

VARIABLES

nums = [4, 7, 2, 6] sum = 19 avg = 4.75

OUTPUT

```
19 4.75
```

Minimum and Maximum



Array: 5, 3, 9, 1.

Step 1 / 15

```
▶ int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

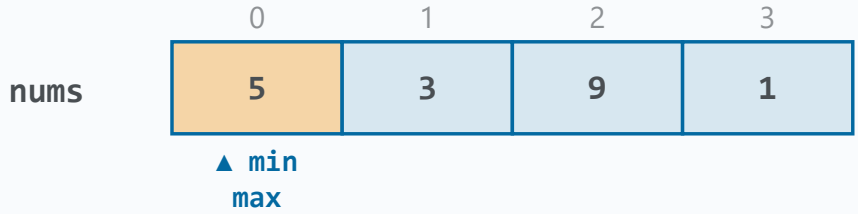
VARIABLES

`nums = [5, 3, 9, 1]`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



The first element is the candidate for both **min** and **max** (starting at 0 would fail for all-negative or all-positive arrays).

Step 2 / 15

```
1 int[] nums = {5, 3, 9, 1};
▶ int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

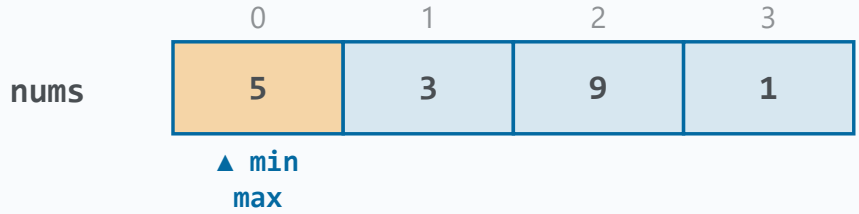
VARIABLES

`nums = [5, 3, 9, 1]` `min = 5` `max = 5`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



Pass 1: `value = 5`.

Step 3 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
▶ for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

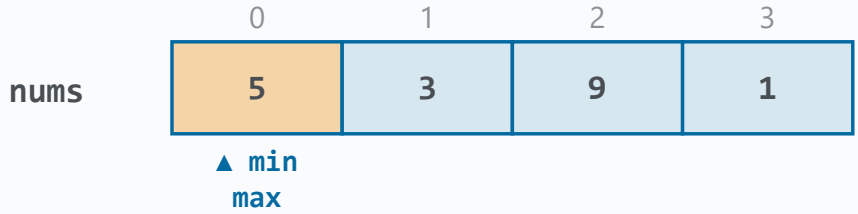
VARIABLES

`nums = [5, 3, 9, 1]` `min = 5` `max = 5` `value = 5`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



5 < 5 → **false**; min unchanged.

Step 4 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
▶     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

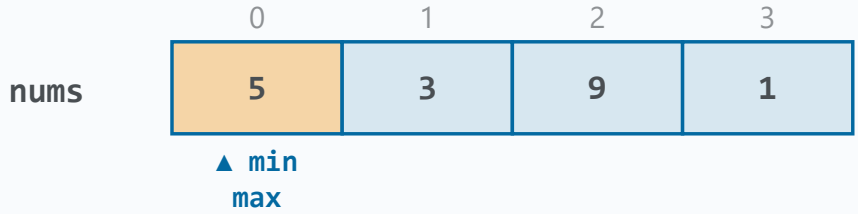
VARIABLES

nums = [5, 3, 9, 1] | min = 5 | max = 5 | value = 5

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



5 > 5 → **false**; max unchanged.

Step 5 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
▶     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

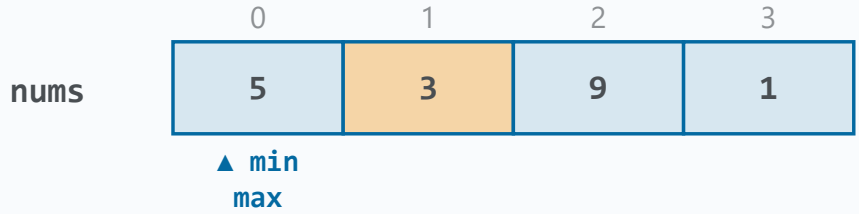
VARIABLES

`nums = [5, 3, 9, 1]` `min = 5` `max = 5` `value = 5`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



Pass 2: `value = 3`.

Step 6 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
▶ for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

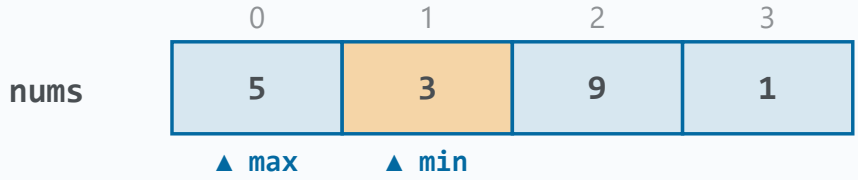
VARIABLES

`nums = [5, 3, 9, 1]` `min = 5` `max = 5` `value = 3`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



3 < 5 → **true**: min = 3.

Step 7 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
▶     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

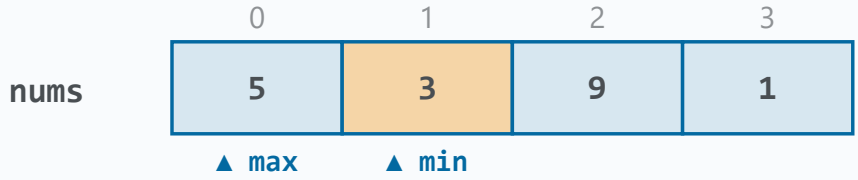
VARIABLES

nums = [5, 3, 9, 1] min = 3 max = 5 value = 3

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



3 > 5 → **false**; max unchanged.

Step 8 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
▶     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

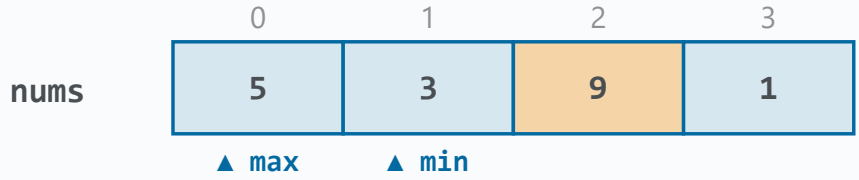
VARIABLES

`nums = [5, 3, 9, 1]` `min = 3` `max = 5` `value = 3`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



Pass 3: `value = 9`.

Step 9 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
▶ for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

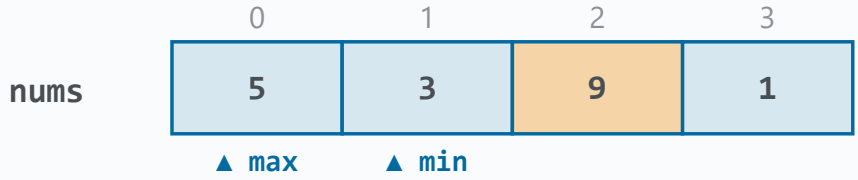
VARIABLES

`nums = [5, 3, 9, 1]` `min = 3` `max = 5` `value = 9`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



9 < 3 → **false**; min unchanged.

Step 10 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
▶     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

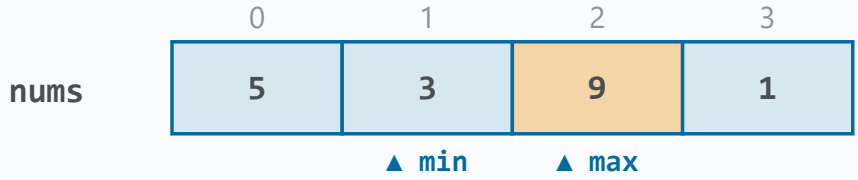
VARIABLES

`nums = [5, 3, 9, 1]` `min = 3` `max = 5` `value = 9`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



9 > 5 → **true**: max = 9.

Step 11 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
▶     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

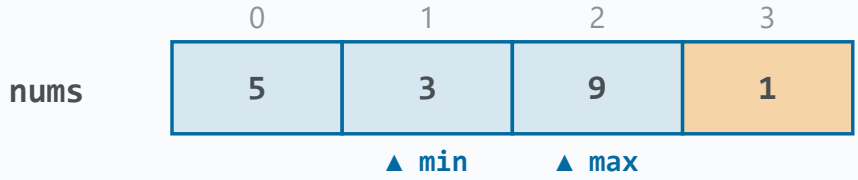
VARIABLES

nums = [5, 3, 9, 1] | min = 3 | max = 9 | value = 9

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



Pass 4: `value = 1`.

Step 12 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
▶ for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

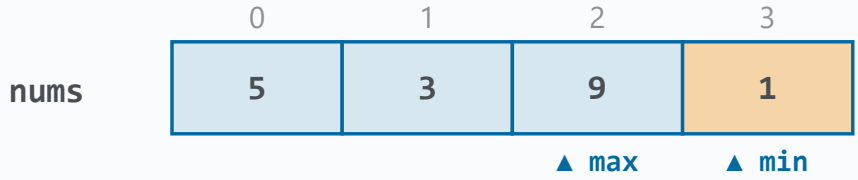
VARIABLES

`nums = [5, 3, 9, 1]` `min = 3` `max = 9` `value = 1`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



1 < 3 → **true**: min = 1.

Step 13 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
▶     if (value < min) min = value;
5     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

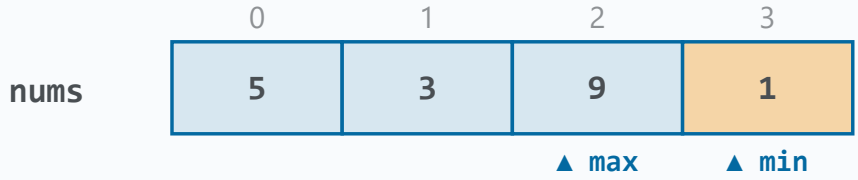
VARIABLES

nums = [5, 3, 9, 1] | min = 1 | max = 9 | value = 1

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



1 > 9 → **false**; max unchanged.

Step 14 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
▶     if (value > max) max = value;
6 }
7 System.out.println(min + " " + max);
```

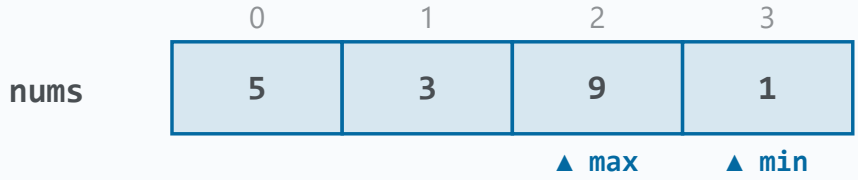
VARIABLES

`nums = [5, 3, 9, 1]` `min = 1` `max = 9` `value = 1`

OUTPUT

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Minimum and Maximum



Every element checked (n = 4 passes) → **1 9**.

Step 15 / 15

```
1 int[] nums = {5, 3, 9, 1};
2 int min = nums[0], max = nums[0];
3 for (int value : nums) {
4     if (value < min) min = value;
5     if (value > max) max = value;
6 }
▶ System.out.println(min + " " + max);
```

VARIABLES

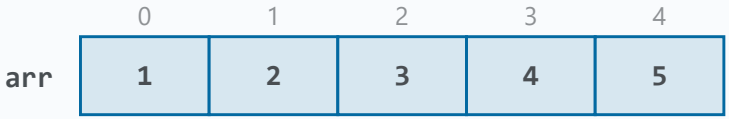
`nums = [5, 3, 9, 1]` `min = 1` `max = 9`

OUTPUT

```
1 9
```

`nums[0]` fails on an empty array; real code should handle `nums.length == 0` first.

Reversing an Array (two pointers)



The array to reverse.

Step 1 / 13

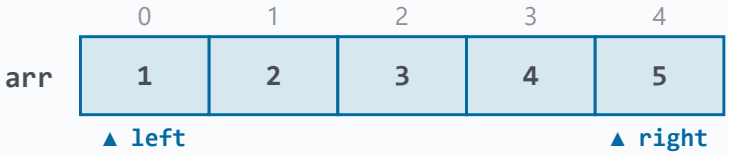
```
▶ int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

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

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



Two pointers: **left = 0** and **right = length − 1 = 4**.

Step 2 / 13

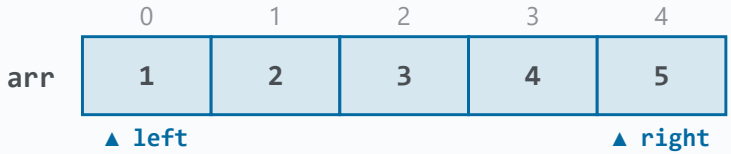
```
1 int[] arr = {1, 2, 3, 4, 5};
▶ int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [1, 2, 3, 4, 5]` `left = 0` `right = 4`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



0 < 4 → **true**, so the ends get swapped.

Step 3 / 13

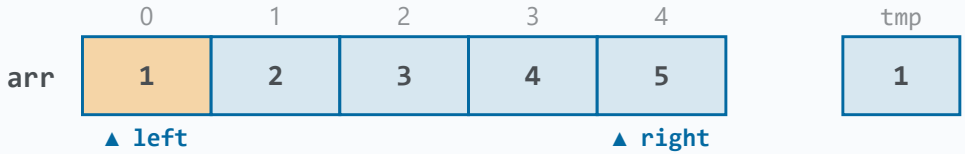
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
▶ while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [1, 2, 3, 4, 5]` `left = 0` `right = 4`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



`tmp = arr[0] = 1` : the left value is saved so it isn't lost.

Step 4 / 13

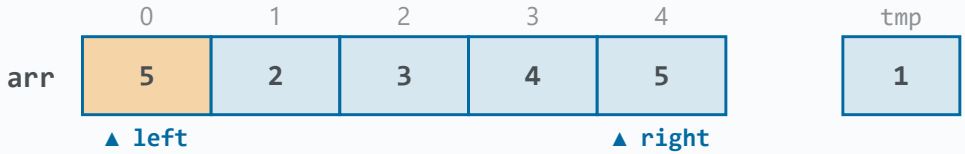
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
▶     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [1, 2, 3, 4, 5]` `left = 0` `right = 4` `tmp = 1`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



`arr[0] = arr[4]` → **5** goes into the left cell.

Step 5 / 13

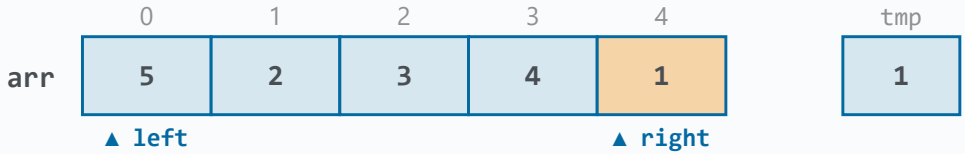
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
▶     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [5, 2, 3, 4, 5]` `left = 0` `right = 4` `tmp = 1`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



`arr[4] = tmp` → **1** goes into the right cell. Swap done.

Step 6 / 13

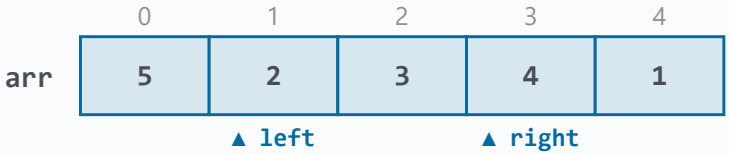
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
▶     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [5, 2, 3, 4, 1]` `left = 0` `right = 4` `tmp = 1`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



The pointers move inward: **left = 1**, **right = 3**.

Step 7 / 13

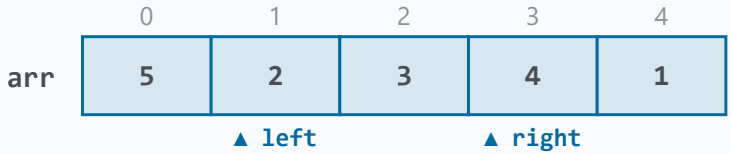
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
▶     left++;
▶     right--;
9 }
```

VARIABLES

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

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



1 < 3 → **true**, so the ends get swapped.

Step 8 / 13

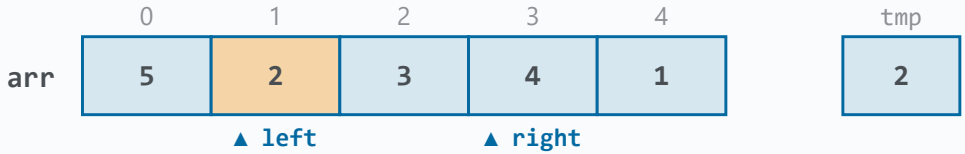
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
▶ while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

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

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



`tmp = arr[1] = 2` : the left value is saved so it isn't lost.

Step 9 / 13

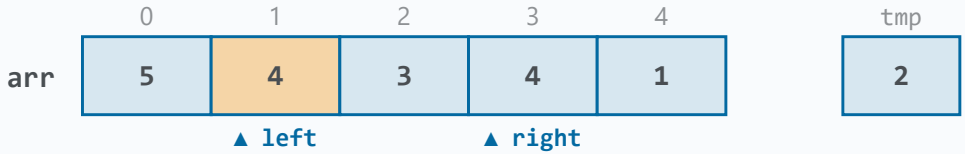
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
▶     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [5, 2, 3, 4, 1]` `left = 1` `right = 3` `tmp = 2`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



`arr[1] = arr[3]` → **4** goes into the left cell.

Step 10 / 13

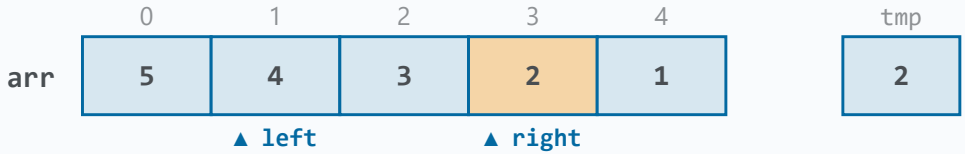
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
▶     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [5, 4, 3, 4, 1]` `left = 1` `right = 3` `tmp = 2`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



`arr[3] = tmp` → **2** goes into the right cell. Swap done.

Step 11 / 13

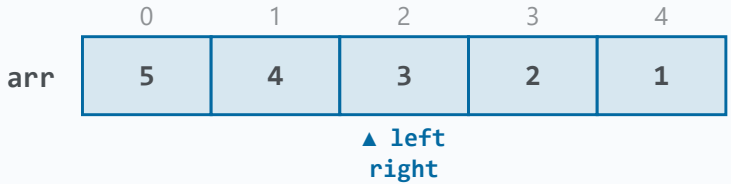
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
▶     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

`arr = [5, 4, 3, 2, 1]` `left = 1` `right = 3` `tmp = 2`

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



The pointers move inward: **left = 2**, **right = 2**.

Step 12 / 13

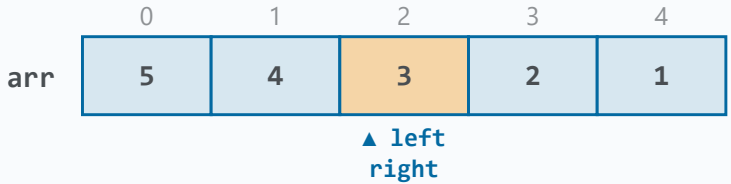
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
3 while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
▶     left++;
▶     right--;
9 }
```

VARIABLES

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

n/2 swaps; no extra array needed (in place, O(1) extra space).

Reversing an Array (two pointers)



2 < 2 → false: the pointers met in the middle → the array is reversed.

Step 13 / 13

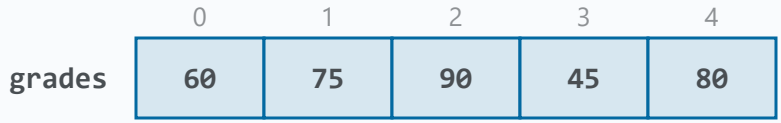
```
1 int[] arr = {1, 2, 3, 4, 5};
2 int left = 0, right = arr.length - 1;
▶ while (left < right) {
4     int tmp = arr[left];
5     arr[left] = arr[right];
6     arr[right] = tmp;
7     left++;
8     right--;
9 }
```

VARIABLES

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

n/2 swaps; no extra array needed (in place, O(1) extra space).

Counting Values Above the Average



Five students' grades. First the average, then those above it: **two passes** are needed.

Step 1 / 15

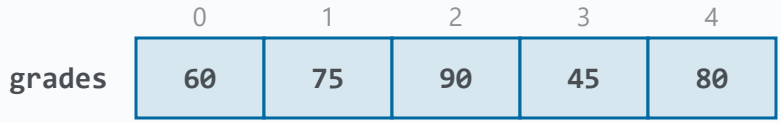
```
▶ int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

VARIABLES

`grades = [60, 75, 90, 45, 80]`

OUTPUT

Counting Values Above the Average



Pass 1: accumulate the sum.

Step 2 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
▶ int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

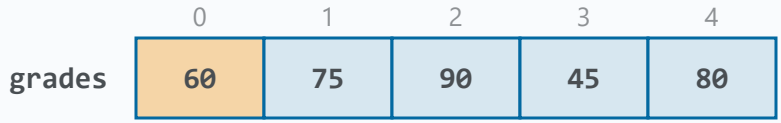
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`sum = 0`

OUTPUT

Counting Values Above the Average



x = 60 → sum = **60**.

Step 3 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
▶ for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

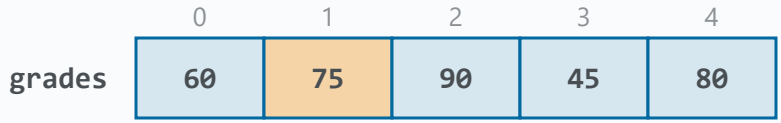
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`sum = 60` `x = 60`

OUTPUT

Counting Values Above the Average



x = 75 → sum = **135**.

Step 4 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
▶ for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

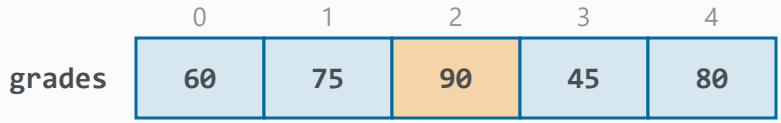
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`sum = 135` `x = 75`

OUTPUT

Counting Values Above the Average



x = 90 → sum = **225**.

Step 5 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
▶ for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

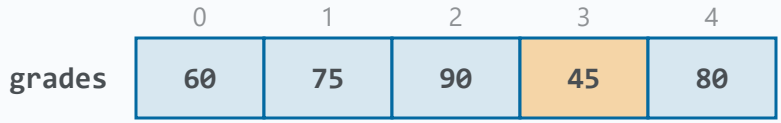
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`sum = 225` `x = 90`

OUTPUT

Counting Values Above the Average



x = 45 → sum = **270**.

Step 6 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
▶ for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

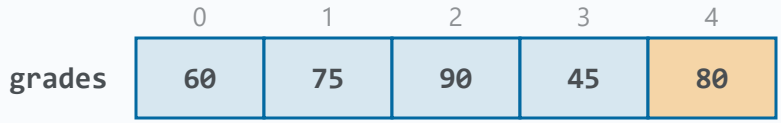
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`sum = 270` `x = 45`

OUTPUT

Counting Values Above the Average



x = 80 → sum = **350**.

Step 7 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
▶ for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

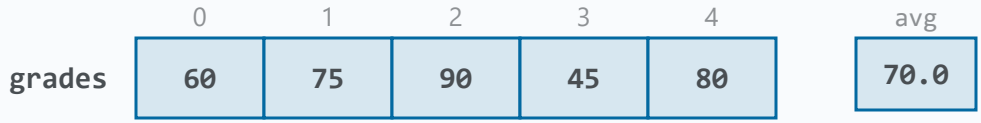
VARIABLES

grades = [60, 75, 90, 45, 80]

sum = 350 x = 80

OUTPUT

Counting Values Above the Average



`(double) 350 / 5` = **70.0**.

Step 8 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
▶ double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

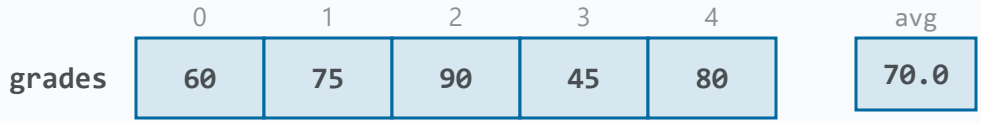
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`sum = 350` `avg = 70.0`

OUTPUT

Counting Values Above the Average



Counter for pass 2: above = 0.

Step 9 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
▶ int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

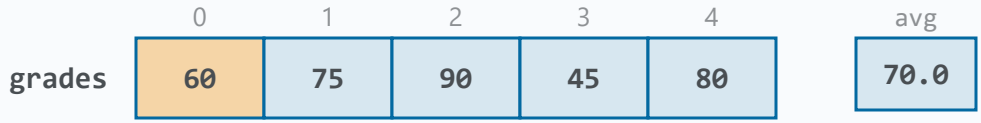
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 0`

OUTPUT

Counting Values Above the Average



60 > 70.0 → false.

Step 10 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
▶ for (int x : grades)
▶   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

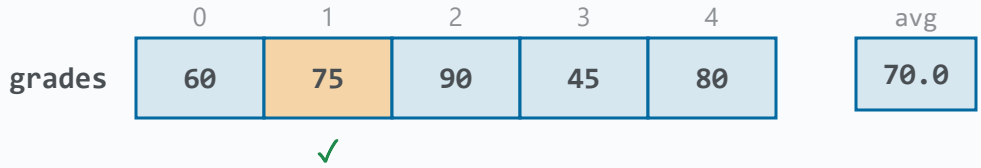
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 0` `x = 60`

OUTPUT

Counting Values Above the Average



75 > 70.0 → **true**: above = 1.

Step 11 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
▶ for (int x : grades)
▶   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

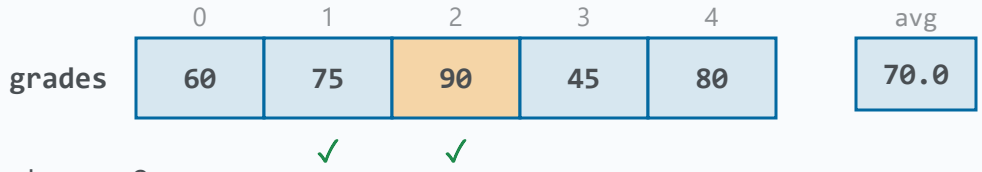
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 1` `x = 75`

OUTPUT

Counting Values Above the Average



90 > 70.0 → **true**: above = 2.

Step 12 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
▶ for (int x : grades)
▶   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

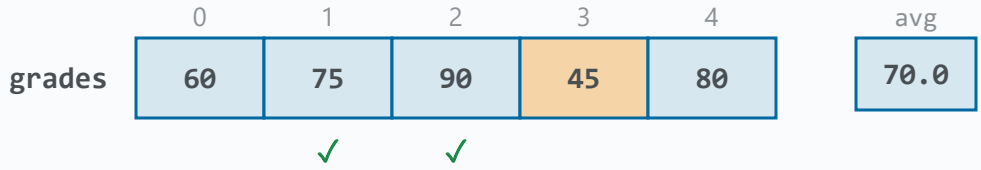
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 2` `x = 90`

OUTPUT

Counting Values Above the Average



45 > 70.0 → false.

Step 13 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
▶ for (int x : grades)
▶   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

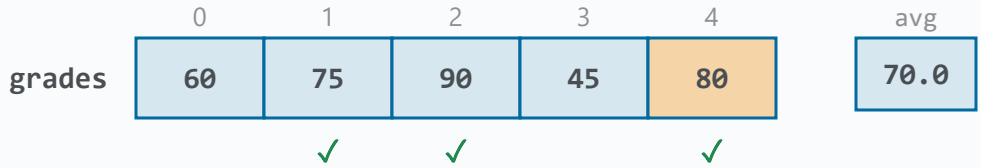
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 2` `x = 45`

OUTPUT

Counting Values Above the Average



80 > 70.0 → **true**: above = 3.

Step 14 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
▶ for (int x : grades)
▶   if (x > avg) above++;
8 System.out.println(avg + " " + above);
```

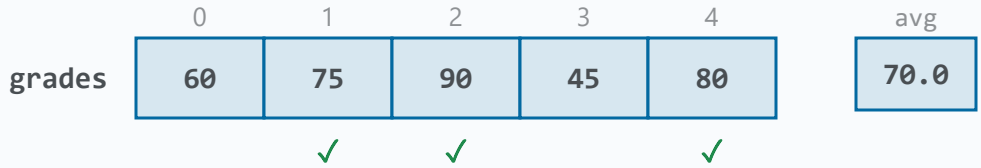
VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 3` `x = 80`

OUTPUT

Counting Values Above the Average



The average is 70.0; **3** grades are above it.

Step 15 / 15

```
1 int[] grades = {60, 75, 90, 45, 80};
2 int sum = 0;
3 for (int x : grades) sum += x;
4 double avg = (double) sum / grades.length;
5 int above = 0;
6 for (int x : grades)
7   if (x > avg) above++;
▶ System.out.println(avg + " " + above);
```

VARIABLES

`grades = [60, 75, 90, 45, 80]`

`avg = 70.0` `above = 3`

OUTPUT

```
70.0 3
```

The Second-Largest Element of an Array



Both candidates start at the smallest int (Integer.MIN_VALUE); each element is examined once.

Step 1 / 7

```
int[] a = {7, 12, 5, 12, 9};
int largest = Integer.MIN_VALUE;
int second = Integer.MIN_VALUE;
for (int x : a) {
  if (x > largest) {
    second = largest;
    largest = x;
  } else if (x > second && x != largest) {
    second = x;
  }
}
System.out.println(second);
```

VARIABLES

`a = [7, 12, 5, 12, 9]`

`largest = MIN` `second = MIN`

OUTPUT

The Second-Largest Element of an Array



7 > MIN → new largest: the old largest (MIN) drops to second, largest = 7.

Step 2 / 7

```
1  int[] a = {7, 12, 5, 12, 9};
2  int largest = Integer.MIN_VALUE;
3  int second = Integer.MIN_VALUE;
4  for (int x : a) {
▶    if (x > largest) {
▶      second = largest;
▶      largest = x;
8    } else if (x > second && x != largest) {
9      second = x;
10   }
11 }
12 System.out.println(second);
```

VARIABLES

a = [7, 12, 5, 12, 9] x = 7

largest = 7 second = MIN

OUTPUT

The Second-Largest Element of an Array



12 > 7 → new largest: the old largest (7) drops to second, largest = 12.

Step 3 / 7

```
1  int[] a = {7, 12, 5, 12, 9};
2  int largest = Integer.MIN_VALUE;
3  int second = Integer.MIN_VALUE;
4  for (int x : a) {
▶    if (x > largest) {
▶      second = largest;
▶      largest = x;
8    } else if (x > second && x != largest) {
9      second = x;
10   }
11 }
12 System.out.println(second);
```

VARIABLES

`a = [7, 12, 5, 12, 9]` `x = 12` `largest = 12` `second = 7`

OUTPUT

The Second-Largest Element of an Array



No change: 5 is not greater than second (7).

Step 4 / 7

```
1  int[] a = {7, 12, 5, 12, 9};
2  int largest = Integer.MIN_VALUE;
3  int second = Integer.MIN_VALUE;
4  for (int x : a) {
5    if (x > largest) {
6      second = largest;
7      largest = x;
   } else if (x > second && x != largest) {
9      second = x;
10   }
11 }
12 System.out.println(second);
```

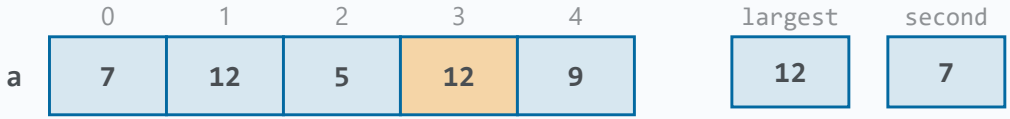
VARIABLES

a = [7, 12, 5, 12, 9] x = 5

largest = 12 second = 7

OUTPUT

The Second-Largest Element of an Array



No change: 12 == largest (a repeat) doesn't count.

Step 5 / 7

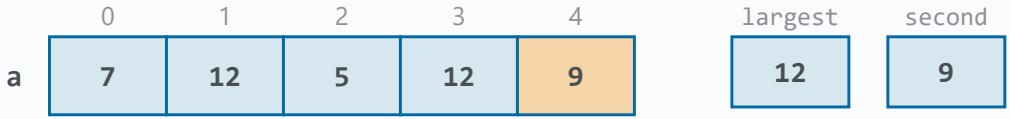
```
1  int[] a = {7, 12, 5, 12, 9};
2  int largest = Integer.MIN_VALUE;
3  int second = Integer.MIN_VALUE;
4  for (int x : a) {
5    if (x > largest) {
6      second = largest;
7      largest = x;
▶    } else if (x > second && x != largest) {
9      second = x;
10   }
11 }
12 System.out.println(second);
```

VARIABLES

`a = [7, 12, 5, 12, 9]` `x = 12` `largest = 12` `second = 7`

OUTPUT

The Second-Largest Element of an Array



9 is smaller than the largest but **greater than second**, and 9 != 12 → second = 9.

Step 6 / 7

```
1  int[] a = {7, 12, 5, 12, 9};
2  int largest = Integer.MIN_VALUE;
3  int second = Integer.MIN_VALUE;
4  for (int x : a) {
5    if (x > largest) {
6      second = largest;
7      largest = x;
▶    } else if (x > second && x != largest) {
▶      second = x;
10   }
11 }
12 System.out.println(second);
```

VARIABLES

`a = [7, 12, 5, 12, 9]` `x = 9`

`largest = 12` `second = 9`

OUTPUT

The Second-Largest Element of an Array



One pass gives **9** (12 appears twice, but the second largest is 9). If `second` is still `Integer.MIN_VALUE`, there is no second distinct value (empty, single-element or all-equal array).

Step 7 / 7

```
1  int[] a = {7, 12, 5, 12, 9};
2  int largest = Integer.MIN_VALUE;
3  int second = Integer.MIN_VALUE;
4  for (int x : a) {
5    if (x > largest) {
6      second = largest;
7      largest = x;
8    } else if (x > second && x != largest) {
9      second = x;
10   }
11 }
▶ System.out.println(second);
```

VARIABLES

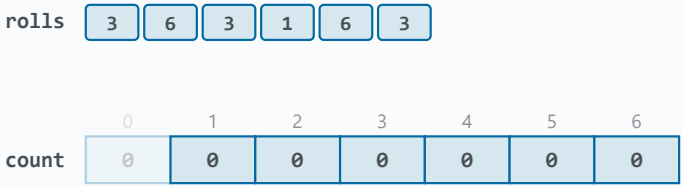
`a = [7, 12, 5, 12, 9]`

`largest = 12` `second = 9`

OUTPUT

```
9
```

Counting Frequencies: A Die Histogram



The rolls are fixed here (really `rnd.nextInt(6) + 1`). A 7-cell counter: index = die value (0 unused). This is the **value as index** trick.

Step 1 / 13

```
▶ int[] rolls = {3, 6, 3, 1, 6, 3};
▶ int[] count = new int[7];
3 for (int die : rolls)
4   count[die]++;
5 for (int y = 1; y <= 6; y++) {
6   System.out.print(y + ": ");
7   for (int k = 0; k < count[y]; k++)
8     System.out.print("*");
9   System.out.println();
10 }
```

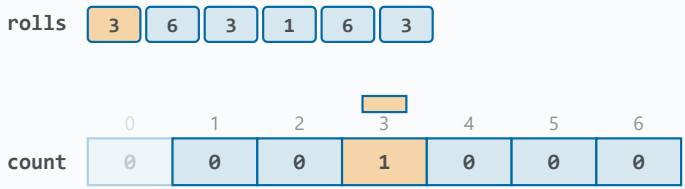
VARIABLES

rolls = [3, 6, 3, 1, 6, 3]

count = [0, 0, 0, 0, 0, 0, 0]

OUTPUT

Counting Frequencies: A Die Histogram



die = 3 → **count[3]++** → 1. No if-else chain needed!

Step 2 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
▶  for (int die : rolls)
▶    count[die]++;
5  for (int y = 1; y <= 6; y++) {
6    System.out.print(y + ": ");
7    for (int k = 0; k < count[y]; k++)
8      System.out.print("*");
9    System.out.println();
10 }
```

VARIABLES

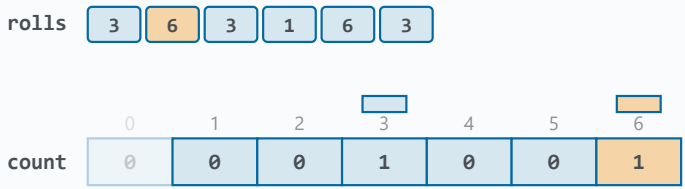
rolls = [3, 6, 3, 1, 6, 3]

count = [0, 0, 0, 1, 0, 0, 0]

die = 3

OUTPUT

Counting Frequencies: A Die Histogram



die = 6 → **count[6]++** → 1. No if-else chain needed!

Step 3 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
▶  for (int die : rolls)
▶    count[die]++;
5  for (int y = 1; y <= 6; y++) {
6    System.out.print(y + ": ");
7    for (int k = 0; k < count[y]; k++)
8      System.out.print("*");
9    System.out.println();
10 }
```

VARIABLES

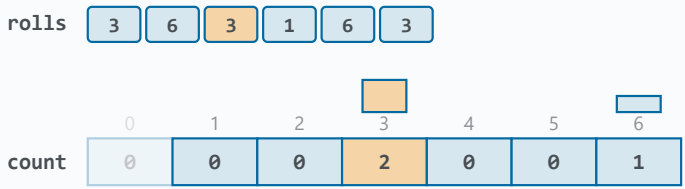
rolls = [3, 6, 3, 1, 6, 3]

count = [0, 0, 0, 1, 0, 0, 1]

die = 6

OUTPUT

Counting Frequencies: A Die Histogram



die = 3 → **count[3]++** → 2. No if-else chain needed!

Step 4 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
▶  for (int die : rolls)
▶    count[die]++;
5  for (int y = 1; y <= 6; y++) {
6    System.out.print(y + ": ");
7    for (int k = 0; k < count[y]; k++)
8      System.out.print("*");
9    System.out.println();
10 }
```

VARIABLES

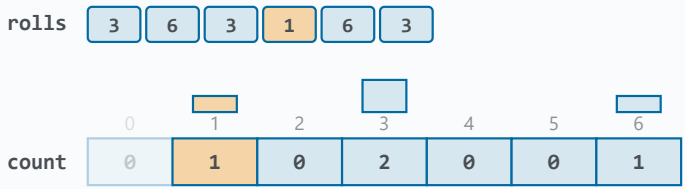
rolls = [3, 6, 3, 1, 6, 3]

count = [0, 0, 0, 2, 0, 0, 1]

die = 3

OUTPUT

Counting Frequencies: A Die Histogram



die = 1 → **count[1]++** → 1. No if-else chain needed!

Step 5 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
▶  for (int die : rolls)
▶    count[die]++;
5  for (int y = 1; y <= 6; y++) {
6    System.out.print(y + ": ");
7    for (int k = 0; k < count[y]; k++)
8      System.out.print("*");
9    System.out.println();
10 }
```

VARIABLES

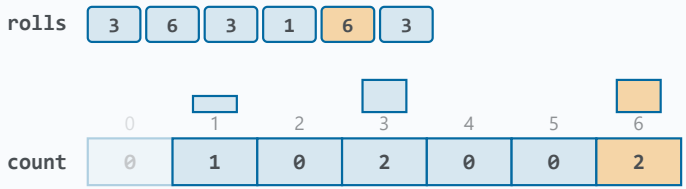
rolls = [3, 6, 3, 1, 6, 3]

count = [0, 1, 0, 2, 0, 0, 1]

die = 1

OUTPUT

Counting Frequencies: A Die Histogram



die = 6 → **count[6]++** → 2. No if-else chain needed!

Step 6 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
▶  for (int die : rolls)
▶    count[die]++;
5  for (int y = 1; y <= 6; y++) {
6    System.out.print(y + ": ");
7    for (int k = 0; k < count[y]; k++)
8      System.out.print("*");
9    System.out.println();
10 }
```

VARIABLES

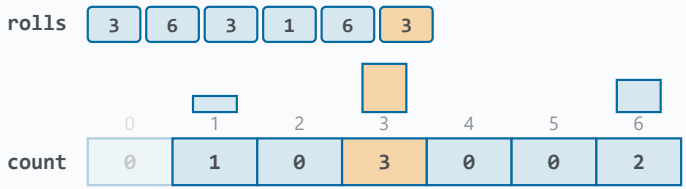
`rolls = [3, 6, 3, 1, 6, 3]`

`count = [0, 1, 0, 2, 0, 0, 2]`

`die = 6`

OUTPUT

Counting Frequencies: A Die Histogram



die = 3 → **count[3]++** → 3. No if-else chain needed!

Step 7 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
▶  for (int die : rolls)
▶    count[die]++;
5  for (int y = 1; y <= 6; y++) {
6    System.out.print(y + ": ");
7    for (int k = 0; k < count[y]; k++)
8      System.out.print("*");
9    System.out.println();
10 }
```

VARIABLES

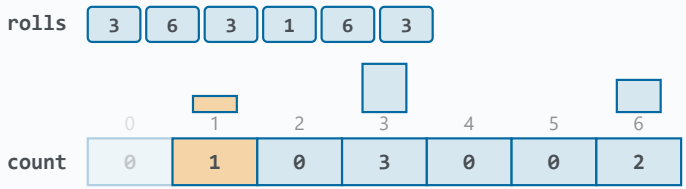
`rolls = [3, 6, 3, 1, 6, 3]`

`count = [0, 1, 0, 3, 0, 0, 2]`

`die = 3`

OUTPUT

Counting Frequencies: A Die Histogram



y = 1: count[1] = 1 star.

Step 8 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
3  for (int die : rolls)
4    count[die]++;
5  for (int y = 1; y <= 6; y++) {
▶    System.out.print(y + ": ");
▶    for (int k = 0; k < count[y]; k++)
▶      System.out.print("*");
▶    System.out.println();
10 }
```

VARIABLES

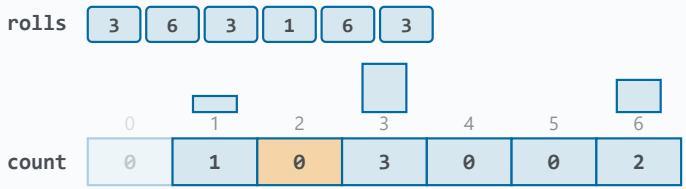
count = [0, 1, 0, 3, 0, 0, 2]

y = 1

OUTPUT

```
1: *
```

Counting Frequencies: A Die Histogram



y = 2: count[2] = 0 stars (none).

Step 9 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
3  for (int die : rolls)
4    count[die]++;
5  for (int y = 1; y <= 6; y++) {
▶    System.out.print(y + ": ");
▶    for (int k = 0; k < count[y]; k++)
▶      System.out.print("*");
▶    System.out.println();
10 }
```

VARIABLES

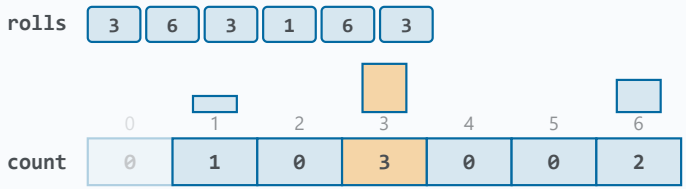
`count = [0, 1, 0, 3, 0, 0, 2]`

`y = 2`

OUTPUT

```
1: *
2:
```

Counting Frequencies: A Die Histogram



y = 3: count[3] = 3 stars.

Step 10 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
3  for (int die : rolls)
4    count[die]++;
5  for (int y = 1; y <= 6; y++) {
▶    System.out.print(y + ": ");
▶    for (int k = 0; k < count[y]; k++)
▶      System.out.print("*");
▶    System.out.println();
10 }
```

VARIABLES

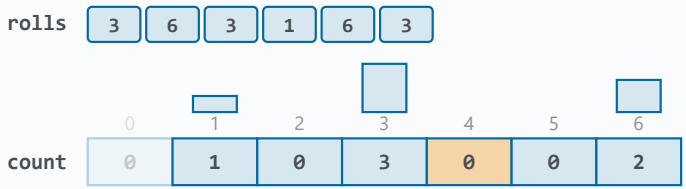
`count = [0, 1, 0, 3, 0, 0, 2]`

`y = 3`

OUTPUT

```
1: *
2:
3: ***
```

Counting Frequencies: A Die Histogram



y = 4: count[4] = 0 stars (none).

Step 11 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
3  for (int die : rolls)
4    count[die]++;
5  for (int y = 1; y <= 6; y++) {
▶    System.out.print(y + ": ");
▶    for (int k = 0; k < count[y]; k++)
▶      System.out.print("*");
▶    System.out.println();
10 }
```

VARIABLES

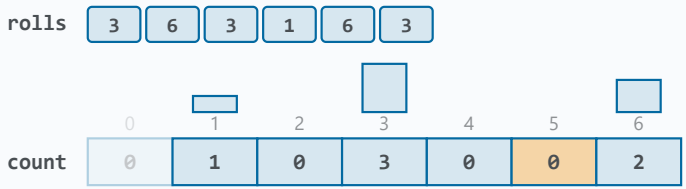
`count = [0, 1, 0, 3, 0, 0, 2]`

`y = 4`

OUTPUT

```
1: *
2:
3: ***
4:
```

Counting Frequencies: A Die Histogram



y = 5: count[5] = 0 stars (none).

Step 12 / 13

```
1 int[] rolls = {3, 6, 3, 1, 6, 3};
2 int[] count = new int[7];
3 for (int die : rolls)
4   count[die]++;
5 for (int y = 1; y <= 6; y++) {
▶   System.out.print(y + ": ");
▶   for (int k = 0; k < count[y]; k++)
▶     System.out.print("*");
▶   System.out.println();
10 }
```

VARIABLES

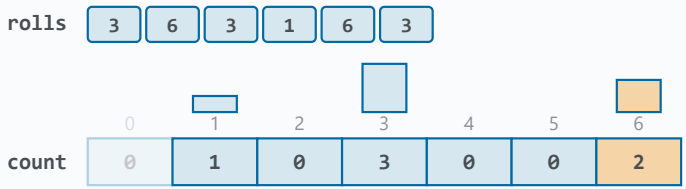
count = [0, 1, 0, 3, 0, 0, 2]

y = 5

OUTPUT

```
1: *
2:
3: ***
4:
5:
```

Counting Frequencies: A Die Histogram



y = 6: count[6] = 2 stars.

Step 13 / 13

```
1  int[] rolls = {3, 6, 3, 1, 6, 3};
2  int[] count = new int[7];
3  for (int die : rolls)
4    count[die]++;
5  for (int y = 1; y <= 6; y++) {
▶    System.out.print(y + ": ");
▶    for (int k = 0; k < count[y]; k++)
▶      System.out.print("*");
▶    System.out.println();
10 }
```

VARIABLES

`count = [0, 1, 0, 3, 0, 0, 2]`

`y = 6`

OUTPUT

```
1: *
2:
3: ***
4:
5:
6: **
```

Sieve of Eratosthenes (`boolean` array)

crossed out: composite[i] = true



All cells `false`: initially 2…20 are all prime candidates.

Step 1 / 9

```
▶ int n = 20;
▶ boolean[] composite = new boolean[n + 1];
3 for (int i = 2; i * i <= n; i++)
4   if (!composite[i])
5     for (int k = i * i; k <= n; k += i)
6       composite[k] = true;
7 for (int i = 2; i <= n; i++)
8   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20`

OUTPUT

Sieve of Eratosthenes (`boolean` array)

crossed out: composite[i] = true



2 × 2 = 4 <= 20 → true.

Step 2 / 9

```
1  int n = 20;
2  boolean[] composite = new boolean[n + 1];
▶  for (int i = 2; i * i <= n; i++)
4    if (!composite[i])
5      for (int k = i * i; k <= n; k += i)
6        composite[k] = true;
7  for (int i = 2; i <= n; i++)
8    if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20` `i = 2`

OUTPUT

Sieve of Eratosthenes (`boolean` array)

crossed out: composite[i] = true



2 is not crossed out → prime. Its multiples are marked starting from 2·2: k = 4, 6, 8, 10, 12, 14, 16, 18, 20.

Step 3 / 9

```
1 int n = 20;
2 boolean[] composite = new boolean[n + 1];
3 for (int i = 2; i * i <= n; i++)
▶   if (!composite[i])
▶     for (int k = i * i; k <= n; k += i)
▶       composite[k] = true;
7 for (int i = 2; i <= n; i++)
8   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20` `i = 2`

OUTPUT

Sieve of Eratosthenes (`boolean` array)



3 × 3 = 9 <= 20 → true.

Step 4 / 9

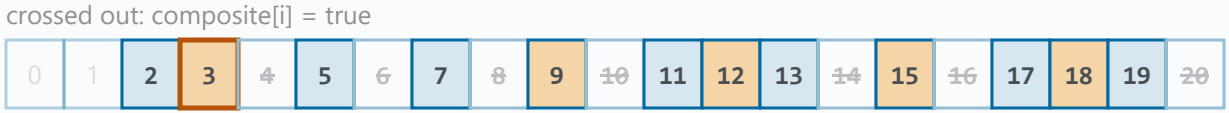
```
1 int n = 20;
2 boolean[] composite = new boolean[n + 1];
▶ for (int i = 2; i * i <= n; i++)
4   if (!composite[i])
5     for (int k = i * i; k <= n; k += i)
6       composite[k] = true;
7 for (int i = 2; i <= n; i++)
8   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20` `i = 3`

OUTPUT

Sieve of Eratosthenes (`boolean` array)



3 is not crossed out → prime. Its multiples are marked starting from 3·3: k = 9, 12, 15, 18.

Step 5 / 9

```
1 int n = 20;
2 boolean[] composite = new boolean[n + 1];
3 for (int i = 2; i * i <= n; i++)
▶   if (!composite[i])
▶     for (int k = i * i; k <= n; k += i)
▶       composite[k] = true;
7 for (int i = 2; i <= n; i++)
8   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20` `i = 3`

OUTPUT

Sieve of Eratosthenes (`boolean` array)

crossed out: composite[i] = true



4 × 4 = 16 <= 20 → true.

Step 6 / 9

```
1  int n = 20;
2  boolean[] composite = new boolean[n + 1];
▶  for (int i = 2; i * i <= n; i++)
4    if (!composite[i])
5      for (int k = i * i; k <= n; k += i)
6        composite[k] = true;
7  for (int i = 2; i <= n; i++)
8    if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20` `i = 4`

OUTPUT

Sieve of Eratosthenes (`boolean` array)



composite[4] = true → 4 is already crossed out; no need to visit its multiples.

Step 7 / 9

```
1 int n = 20;
2 boolean[] composite = new boolean[n + 1];
3 for (int i = 2; i * i <= n; i++)
▶   if (!composite[i])
5     for (int k = i * i; k <= n; k += i)
6       composite[k] = true;
7 for (int i = 2; i <= n; i++)
8   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

`n = 20` `i = 4`

OUTPUT

Sieve of Eratosthenes (`boolean` array)

crossed out: composite[i] = true



5 × 5 = 25 <= 20 → **false**: no need to look for divisors above √20.

Step 8 / 9

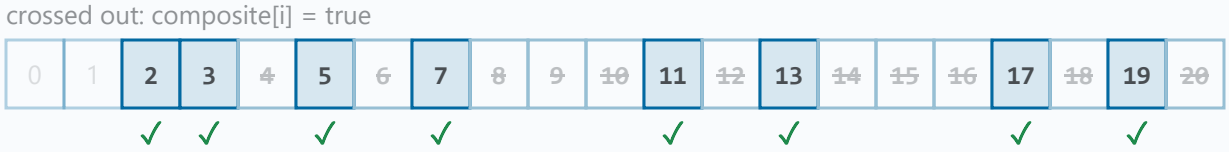
```
1 int n = 20;
2 boolean[] composite = new boolean[n + 1];
▶ for (int i = 2; i * i <= n; i++)
4   if (!composite[i])
5     for (int k = i * i; k <= n; k += i)
6       composite[k] = true;
7 for (int i = 2; i <= n; i++)
8   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

n = 20 i = 5

OUTPUT

Sieve of Eratosthenes (`boolean` array)



The numbers not crossed out are printed: 8 primes.

Step 9 / 9

```
1  int n = 20;
2  boolean[] composite = new boolean[n + 1];
3  for (int i = 2; i * i <= n; i++)
4    if (!composite[i])
5      for (int k = i * i; k <= n; k += i)
6        composite[k] = true;
▶ for (int i = 2; i <= n; i++)
▶   if (!composite[i]) System.out.print(i + " ");
```

VARIABLES

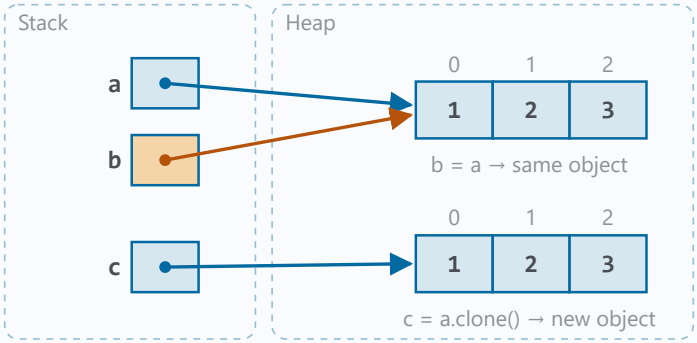
`n = 20`

OUTPUT

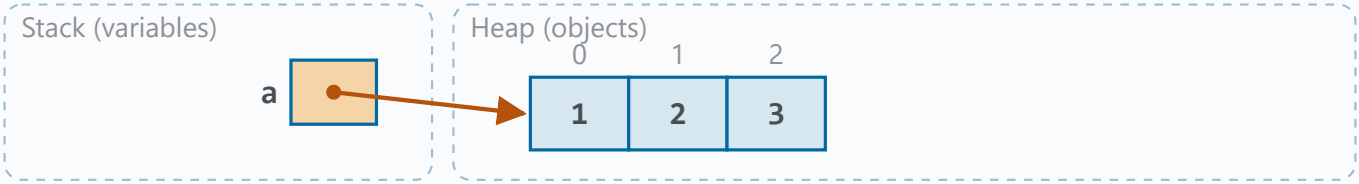
```
2 3 5 7 11 13 17 19
```

Copying Arrays

- `b = a` copies the **reference**, not the elements: two arrows point to the same object.
- For an independent copy use `a.clone()`, `Arrays.copyOf(a, a.length)` or `System.arraycopy`.
- For arrays of objects these copy the element references; the objects themselves are not duplicated (shallow copy).



Code Trace: Copying a Reference vs. Copying Elements



One array on the heap; `a` refers to it.

Step 1 / 7

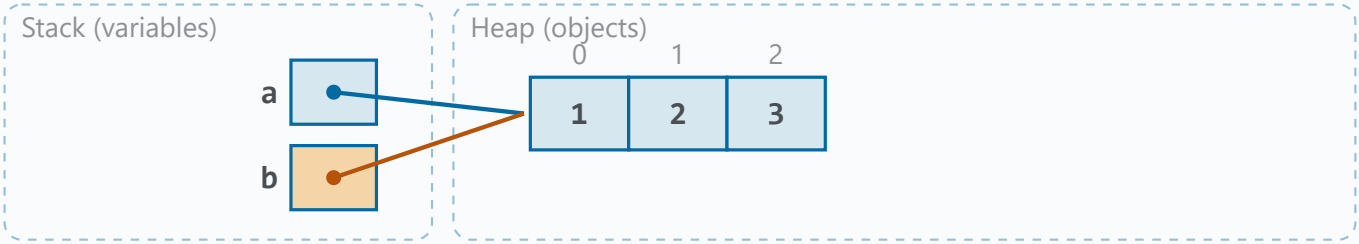
```
▶ int[] a = {1, 2, 3};
2 int[] b = a;
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone();
6 c[1] = 50;
7 System.out.println(a[1] + " " + c[1]);
```

VARIABLES

`a = [1, 2, 3]`

OUTPUT

Code Trace: Copying a Reference vs. Copying Elements



`b = a` : **no** new array is created; b gets the same reference → two arrows on one object.

Step 2 / 7

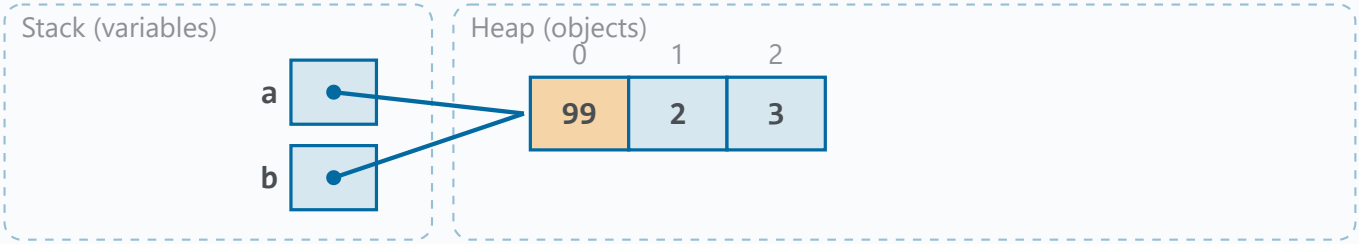
```
1 int[] a = {1, 2, 3};
▶ int[] b = a;
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone();
6 c[1] = 50;
7 System.out.println(a[1] + " " + c[1]);
```

VARIABLES

`a = [1, 2, 3]` `b = [1, 2, 3]`

OUTPUT

Code Trace: Copying a Reference vs. Copying Elements



`b[0] = 99` : the object b refers to changed, and it is a's object too!

Step 3 / 7

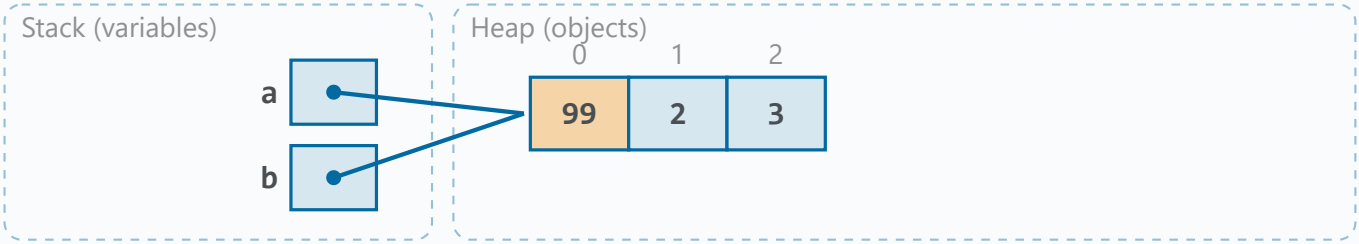
```
1 int[] a = {1, 2, 3};
2 int[] b = a;
▶ b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone();
6 c[1] = 50;
7 System.out.println(a[1] + " " + c[1]);
```

VARIABLES

`a = [99, 2, 3]` `b = [99, 2, 3]`

OUTPUT

Code Trace: Copying a Reference vs. Copying Elements



`a[0]` is now **99**. A copied reference is not an independent array.

Step 4 / 7

```
1 int[] a = {1, 2, 3};
2 int[] b = a;
3 b[0] = 99;
▶ System.out.println(a[0]);
5 int[] c = a.clone();
6 c[1] = 50;
7 System.out.println(a[1] + " " + c[1]);
```

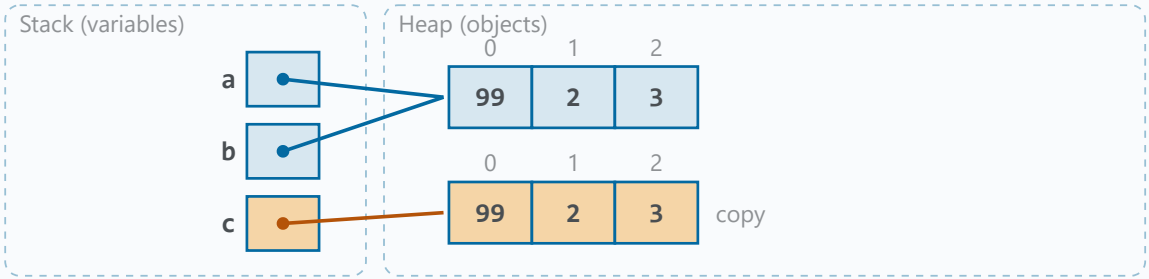
VARIABLES

`a = [99, 2, 3]` `b = [99, 2, 3]`

OUTPUT

```
99
```

Code Trace: Copying a Reference vs. Copying Elements



`a.clone()` creates a **new** array on the heap and copies the elements one by one.

Step 5 / 7

```
1 int[] a = {1, 2, 3};
2 int[] b = a;
3 b[0] = 99;
4 System.out.println(a[0]);
▶ int[] c = a.clone();
6 c[1] = 50;
7 System.out.println(a[1] + " " + c[1]);
```

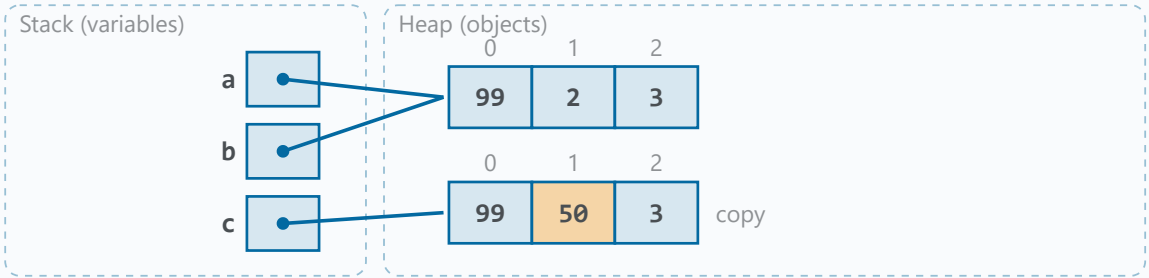
VARIABLES

`a = [99, 2, 3]` `b = [99, 2, 3]` `c = [99, 2, 3]`

OUTPUT

```
99
```

Code Trace: Copying a Reference vs. Copying Elements



`c[1] = 50` changes only the copy.

Step 6 / 7

```
1 int[] a = {1, 2, 3};
2 int[] b = a;
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone();
▶ c[1] = 50;
7 System.out.println(a[1] + " " + c[1]);
```

VARIABLES

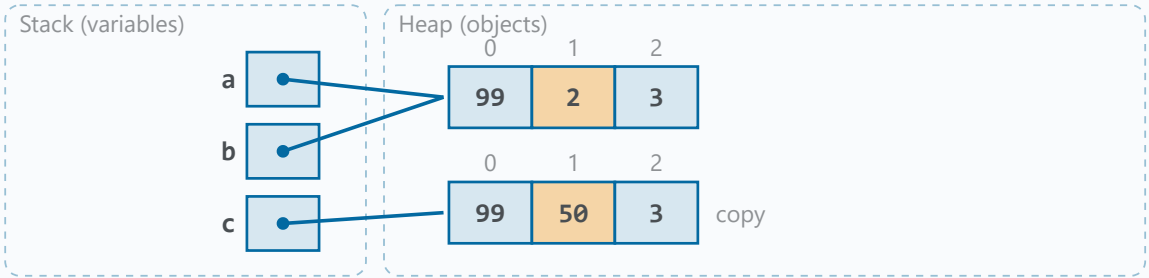
`a = [99, 2, 3]` `b = [99, 2, 3]`

`c = [99, 50, 3]`

OUTPUT

```
99
```

Code Trace: Copying a Reference vs. Copying Elements



`a[1]` = 2 (unchanged), `c[1]` = 50.

Step 7 / 7

```
1 int[] a = {1, 2, 3};
2 int[] b = a;
3 b[0] = 99;
4 System.out.println(a[0]);
5 int[] c = a.clone();
6 c[1] = 50;
▶ System.out.println(a[1] + " " + c[1]);
```

VARIABLES

a = [99, 2, 3] b = [99, 2, 3] c = [99, 50, 3]

OUTPUT

```
99
2 50
```

OPERATIONS

Arrays Utilities

```
import java.util.Arrays;
int[] a = {5, 2, 9}, r = {10, 20, 30, 40, 50}, p = {1, 2, 3}, q = {1, 2, 3}, f = new int[4];
int[][] m = {{1, 2}, {3, 4}};
```

Call	Result
`Arrays.toString(a)`	`[5, 2, 9]`
`Arrays.deepToString(m)`	`[[1, 2], [3, 4]]`
`Arrays.toString(m)`	`[[I@…` (address)
`Arrays.fill(f, 7)` → f	`[7, 7, 7, 7]`
`p == q` / `p.equals(q)`	`false` / `false`
`Arrays.equals(p, q)`	`true`

Copy (Arrays.)	Result
`copyOf(a, 5)`	`[5, 2, 9, 0, 0]`
`copyOf(a, 2)`	`[5, 2]`
`copyOfRange(r, 1, 4)`	`[20, 30, 40]`

- `==` and `equals` compare references; for contents use `Arrays.equals` (`deepEquals` for 2D).
- `copyOf` pads with zeros if the length is larger and truncates if smaller; in `copyOfRange` the end index is exclusive.

Linear Search: `search(d, 6)`

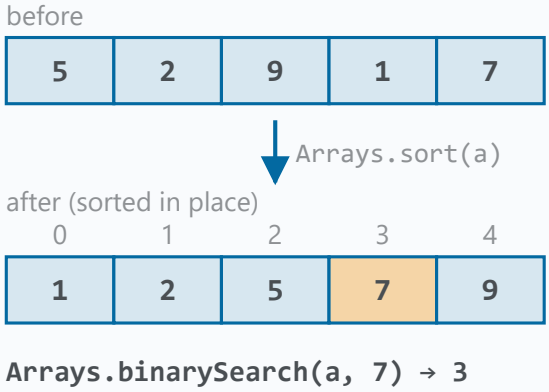
```java
static int search(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++)
        if (arr[i] == target) return i;
    return -1;
}
// in main:
int[] d = {8, 3, 6, 2};
System.out.println(search(d, 6));
```

i	arr[i]	== 6 ?
0	8	false
1	3	false
2	6	**true** → `return 2`

- Output: **2**. The loop ends early at the first match.
- The parameter `arr` refers to **the same array** as `d` (the reference is passed); no copy is made.
- If the target is absent, −1 comes back after n comparisons → **O(n)**.

Sorting and Binary Search

- `Arrays.sort(a)` sorts the array in ascending order **in place** (the original order is lost).
- `Arrays.binarySearch(a, x)` works only on a **sorted** array; it returns the index if found, otherwise a negative value, `-(insertion point) - 1`.
- It halves the range at each step, so it makes far fewer comparisons than linear search.
- Both need `import java.util.Arrays;`.

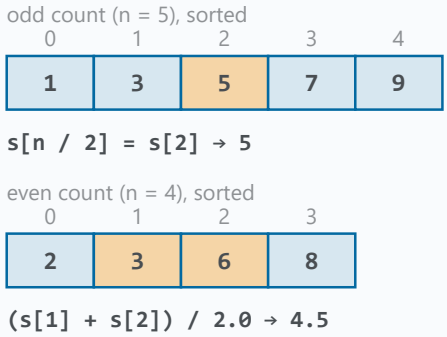


`Arrays.binarySearch(a, 7)` → `3`

The Median

- **Median**: the value in the **middle** of the sorted array; with an even count, the average of the middle two.
- Unaffected by outliers: for {1, 2, 3, 4, 100} the average is 22.0, the median 3.
- `Arrays.sort` changes the array in place, so a **copy** is sorted.
- The `(double)` cast comes before the addition: two large ints can't overflow and the division keeps the fraction.

```
int[] a = {7, 3, 9, 1, 5};
int[] s = Arrays.copyOf(a, a.length);
Arrays.sort(s);    // a is untouched
int n = s.length;
double m = (n % 2 == 1) ? s[n / 2]
    : (s[n/2 - 1] + (double) s[n/2]) / 2;
System.out.println(m);   // 5.0
```



Bubble Sort

```
int[] a = {5, 1, 4, 2};
int n = a.length;
for (int pass = 0; pass < n - 1; pass++) {
  for (int j = 0; j < n - 1 - pass; j++) {
    if (a[j] > a[j + 1]) {
      int tmp = a[j];
      a[j] = a[j + 1];
      a[j + 1] = tmp;
    }
  }
}
```

pass	array after pass	comp.	swap
1	[1, 4, 2, 5]	3	3
2	[1, 2, 4, 5]	2	1
3	[1, 2, 4, 5]	1	0

- Each pass moves the largest remaining element to the end; the inner loop gets one shorter each pass.
- 3 + 2 + 1 = 6 comparisons in all → **$O(n^2)$**.
- No swaps in pass 3: the array was already sorted after pass 2.

Insertion Sort

```
int[] a = {4, 3, 5, 1};
for (int i = 1; i < a.length; i++) {
  int key = a[i];
  int j = i - 1;
  while (j >= 0 && a[j] > key) {
    a[j + 1] = a[j];
    j--;
  }
  a[j + 1] = key;
}
```

i	key	shifted right	array after pass
1	3	4	[3, 4, 5, 1]
2	5	—	[3, 4, 5, 1]
3	1	5, 4, 3	[1, 3, 4, 5]

- The left part is always sorted; elements larger than the key shift one place right, and the key drops into the gap.
- Very fast on nearly sorted arrays; **O(n^2)** in the worst case (reverse order).

SEARCH

Binary Search: `target = 23`

```
int[] a = {2, 5, 8, 12, 16, 23, 38, 56};
int target = 23, lo = 0, hi = a.length - 1;
int result = -1;
while (lo <= hi) {
  int mid = lo + (hi - lo) / 2;
  if (a[mid] == target) { result = mid; break; }
  if (a[mid] < target) lo = mid + 1;
  else hi = mid - 1;
}
System.out.println(result);
```

pass	lo	hi	mid	a[mid]	decision
1	0	7	3	12	12 < 23 → lo = 4
2	4	7	5	23	found: result = 5

- Output: **5**. The array must be sorted.
- Found in 2 passes; linear search would need 6 comparisons → **O(log n)**.

Unsuccessful Search: `Arrays.binarySearch(a, 5)` → -3

```
static int search(int[] a, int x) {
  int lo = 0, hi = a.length - 1;
  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == x) return mid;
    if (a[mid] < x) lo = mid + 1; else hi = mid - 1;
  }
  return -(lo + 1);
}
System.out.println(search(new int[]{2, 4, 6, 8}, 5));
```

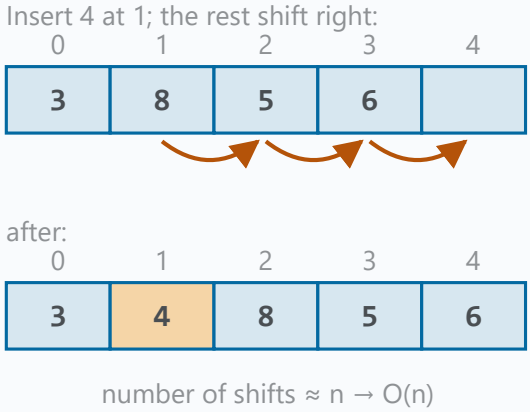
pass	lo	hi	mid	a[mid]	decision
1	0	3	1	4	4 < 5 → lo = 2
2	2	3	2	6	6 > 5 → hi = 1
—	2	1			lo > hi: the loop ends

- The result `-(lo + 1)` = **-3**; a negative value means "not found".
- Insertion point = -(-3) - 1 = 2: 5 would go at position 2. `Arrays.binarySearch` returns the same value.

ANALYSIS

Basic Complexities

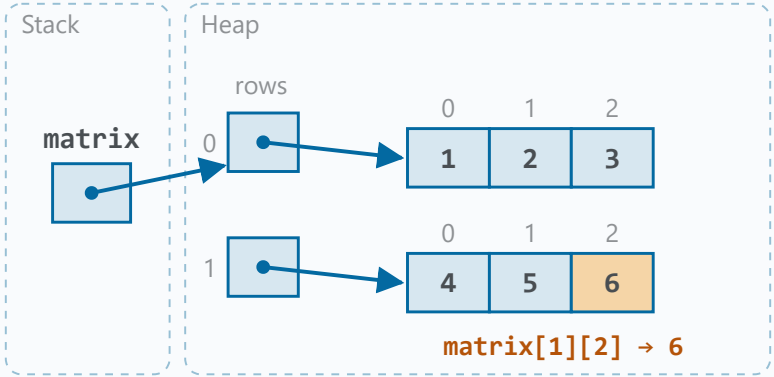
Operation	Time
Index access	O(1)
Linear search	O(n)
Traverse to the end	O(n)
Insert / delete in the middle	O(n)
Two nested full traversals	$O(n^2)$



number of shifts ≈ n → O(n)

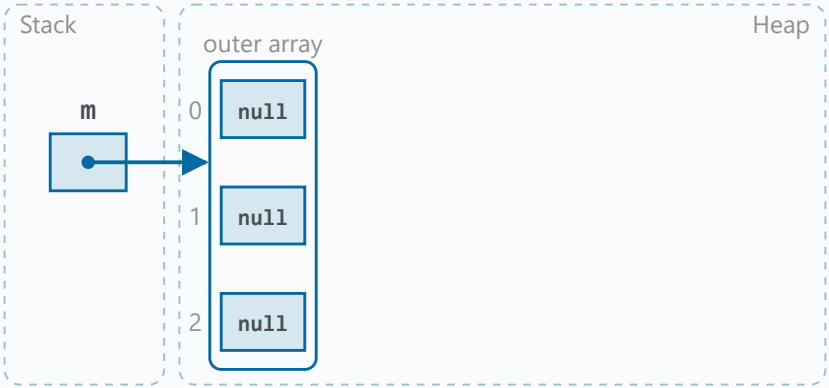
Two-Dimensional Arrays

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6}
};
int value = matrix[1][2]; // 6
```



- The first index is the **row**, the second the **column**.
- In Java a multidimensional array is an array of arrays: each cell of the outer array points to a row array, and rows may differ in length (a jagged array).
- `matrix.length` = number of rows, `matrix[0].length` = number of columns.

A Jagged Array: `new int[3][]`



`new int[3][]` creates only the **outer array** (3 cells). The rows don't exist yet: all three cells are **null**.

Step 1 / 7

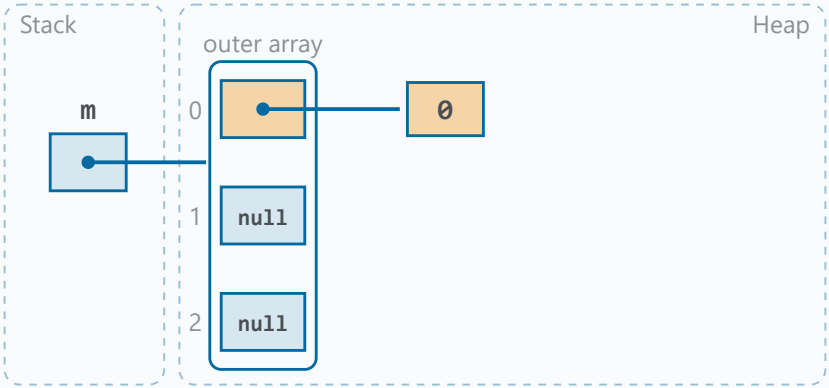
```
▶ int[][] m = new int[3][];
2 m[0] = new int[1];
3 m[1] = new int[2];
4 m[2] = new int[3];
5 for (int i = 0; i < m.length; i++)
6     System.out.print(m[i].length + " ");
```

VARIABLES

`m = →int[3][]`

OUTPUT

A Jagged Array: `new int[3][]`



m[0] = new int[1]: the first cell points to a one-element row. m[1] and m[2] are still null.

Step 2 / 7

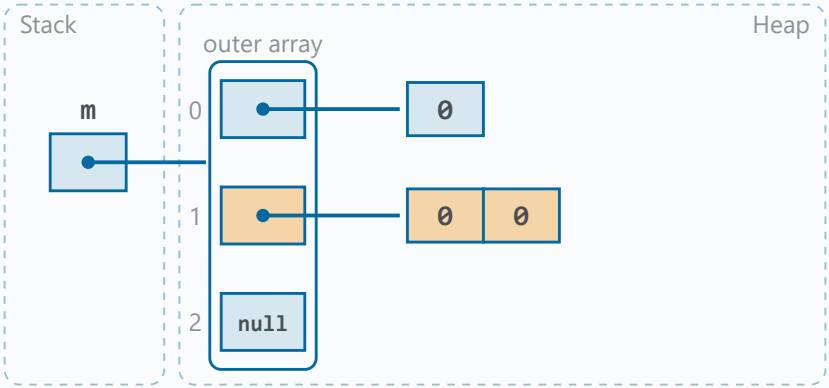
```
1 int[][] m = new int[3][];
▶ m[0] = new int[1];
3 m[1] = new int[2];
4 m[2] = new int[3];
5 for (int i = 0; i < m.length; i++)
6     System.out.print(m[i].length + " ");
```

VARIABLES

`m = →int[3][]`

OUTPUT

A Jagged Array: `new int[3][]`



m[1] = new int[2]: the second row has 2 elements.

Step 3 / 7

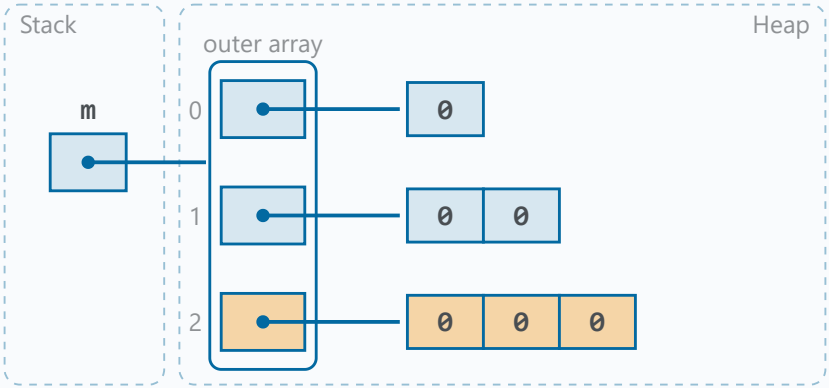
```
1 int[][] m = new int[3][];
2 m[0] = new int[1];
▶ m[1] = new int[2];
4 m[2] = new int[3];
5 for (int i = 0; i < m.length; i++)
6     System.out.print(m[i].length + " ");
```

VARIABLES

`m = →int[3][]`

OUTPUT

A Jagged Array: `new int[3][]`



m[2] = new int[3]: the rows differ in length, which makes a **jagged array**. Each row is a separate object.

Step 4 / 7

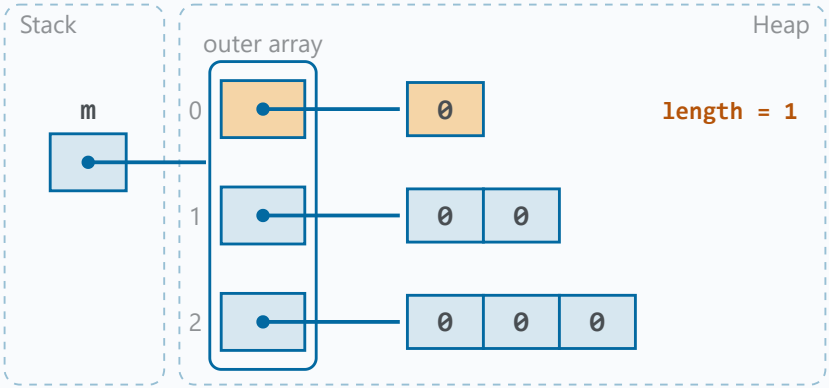
```
1 int[][] m = new int[3][];
2 m[0] = new int[1];
3 m[1] = new int[2];
▶ m[2] = new int[3];
5 for (int i = 0; i < m.length; i++)
6     System.out.print(m[i].length + " ");
```

VARIABLES

`m = →int[3][]`

OUTPUT

A Jagged Array: `new int[3][]`



i = 0: `m[0].length` = 1 is printed.

Step 5 / 7

```
1 int[][] m = new int[3][];
2 m[0] = new int[1];
3 m[1] = new int[2];
4 m[2] = new int[3];
▶ for (int i = 0; i < m.length; i++)
▶     System.out.print(m[i].length + " ");
```

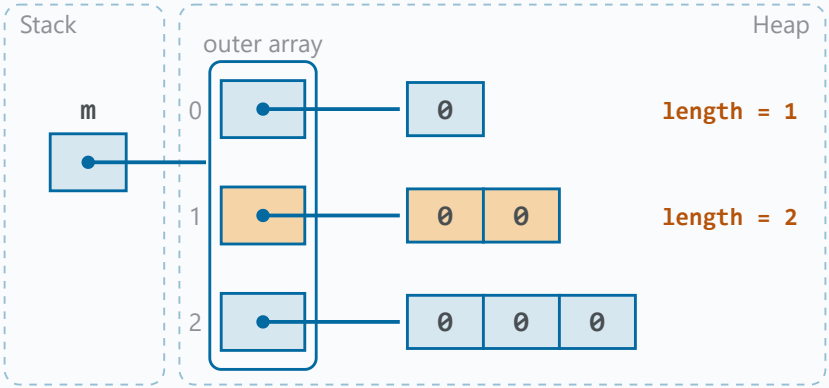
VARIABLES

`m = →int[3][]` `i = 0`

OUTPUT

```
1
```

A Jagged Array: `new int[3][]`



i = 1: `m[1].length` = 2 is printed.

Step 6 / 7

```
1 int[][] m = new int[3][];
2 m[0] = new int[1];
3 m[1] = new int[2];
4 m[2] = new int[3];
▶ for (int i = 0; i < m.length; i++)
▶     System.out.print(m[i].length + " ");
```

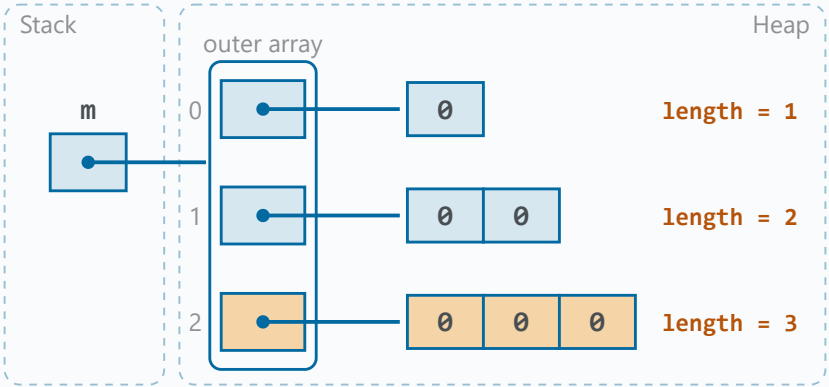
VARIABLES

`m = →int[3][]` `i = 1`

OUTPUT

```
1 2
```

A Jagged Array: `new int[3][]`



i = 2: `m[2].length` = 3. Output: **1 2 3**. Reading `m[i].length` while a row is null would throw NullPointerException.

Step 7 / 7

```
1 int[][] m = new int[3][];
2 m[0] = new int[1];
3 m[1] = new int[2];
4 m[2] = new int[3];
▶ for (int i = 0; i < m.length; i++)
▶     System.out.print(m[i].length + " ");
```

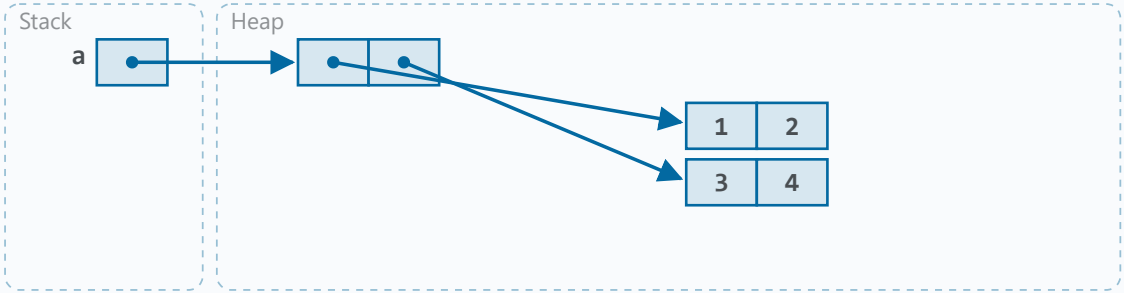
VARIABLES

`m = →int[3][]` `i = 2`

OUTPUT

```
1 2 3
```

The Shallow-Copy Trap in 2D Arrays



`a` is an **outer array** pointing to two row arrays.

Step 1 / 6

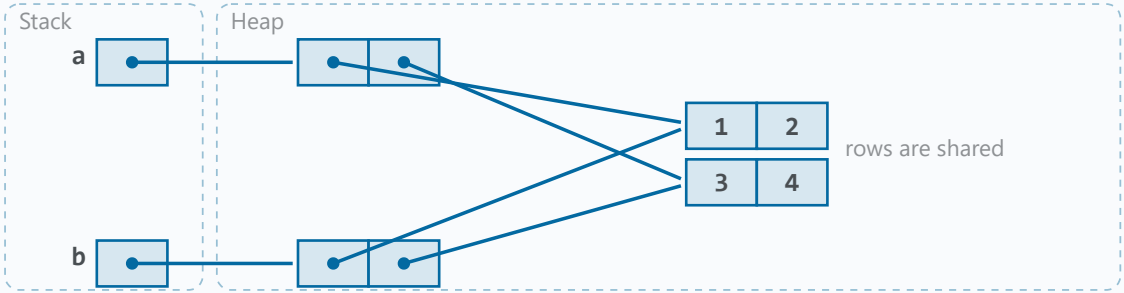
```
▶ int[][] a = {{1, 2}, {3, 4}};
2 int[][] b = a.clone();
3 b[0][0] = 99;
4 b[1] = new int[]{7, 8};
5 System.out.println(Arrays.deepToString(a));
6 System.out.println(Arrays.deepToString(b));
```

VARIABLES

`a = [[1, 2], [3, 4]]`

OUTPUT

The Shallow-Copy Trap in 2D Arrays



`a.clone()` copies only the outer array: both outer arrays point to **the same row objects** (shallow copy).

Step 2 / 6

```
1 int[][] a = {{1, 2}, {3, 4}};
▶ int[][] b = a.clone();
3 b[0][0] = 99;
4 b[1] = new int[]{7, 8};
5 System.out.println(Arrays.deepToString(a));
6 System.out.println(Arrays.deepToString(b));
```

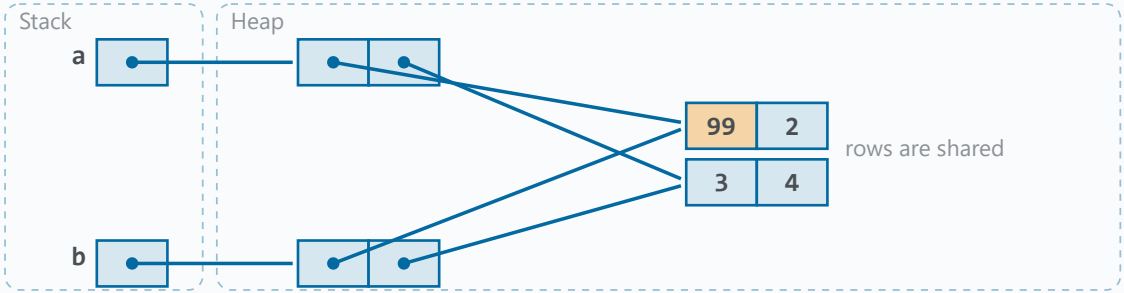
VARIABLES

a = [[1, 2], [3, 4]]

b = [[1, 2], [3, 4]]

OUTPUT

The Shallow-Copy Trap in 2D Arrays



`b[0][0] = 99` : b[0] and a[0] are the same row → **a[0][0] is 99 too**.

Step 3 / 6

```
1 int[][] a = {{1, 2}, {3, 4}};
2 int[][] b = a.clone();
▶ b[0][0] = 99;
4 b[1] = new int[]{7, 8};
5 System.out.println(Arrays.deepToString(a));
6 System.out.println(Arrays.deepToString(b));
```

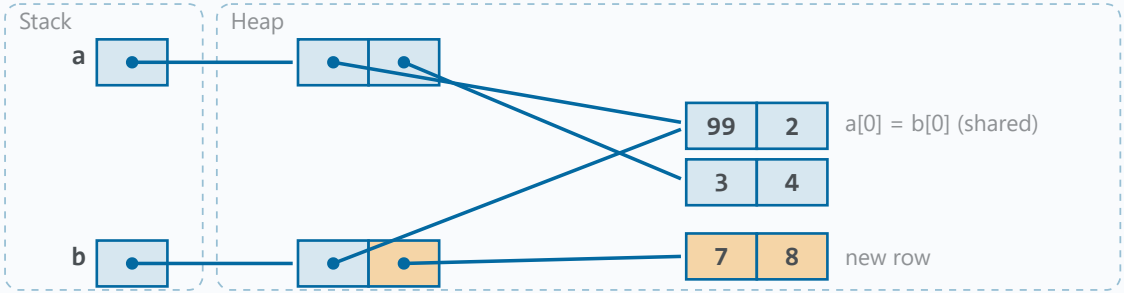
VARIABLES

a = [[99, 2], [3, 4]]

b = [[99, 2], [3, 4]]

OUTPUT

The Shallow-Copy Trap in 2D Arrays



`b[1] = new int[]{7, 8}` only points b's cell at a new row; a[1] is unchanged.

Step 4 / 6

```
1 int[][] a = {{1, 2}, {3, 4}};
2 int[][] b = a.clone();
3 b[0][0] = 99;
▶ b[1] = new int[]{7, 8};
5 System.out.println(Arrays.deepToString(a));
6 System.out.println(Arrays.deepToString(b));
```

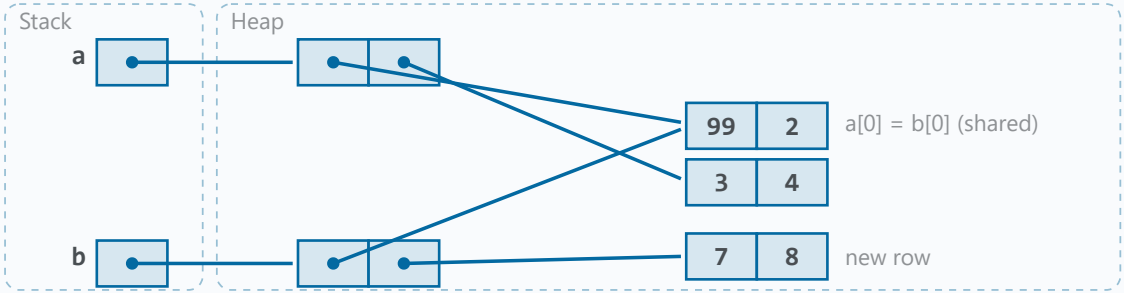
VARIABLES

a = `[[99, 2], [3, 4]]`

b = `[[99, 2], [7, 8]]`

OUTPUT

The Shallow-Copy Trap in 2D Arrays



a: shared row 0 changed (99); row 1 is still {3, 4}.

Step 5 / 6

```
1 int[][] a = {{1, 2}, {3, 4}};
2 int[][] b = a.clone();
3 b[0][0] = 99;
4 b[1] = new int[]{7, 8};
▶ System.out.println(Arrays.deepToString(a));
6 System.out.println(Arrays.deepToString(b));
```

VARIABLES

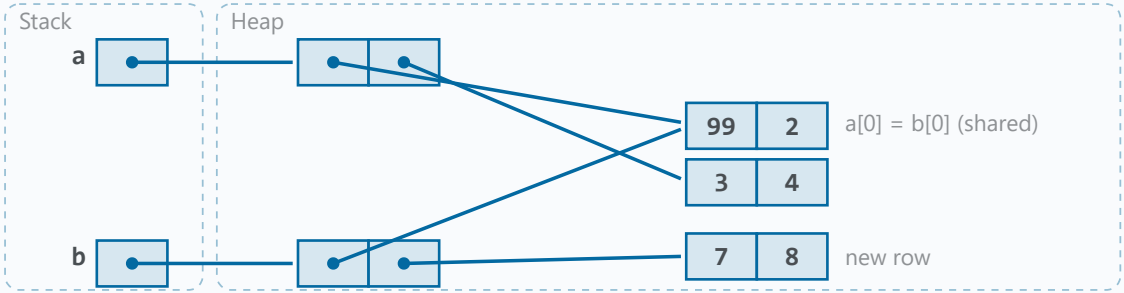
a = `[[99, 2], [3, 4]]`

b = `[[99, 2], [7, 8]]`

OUTPUT

```
[[99, 2], [3, 4]]
```

The Shallow-Copy Trap in 2D Arrays



For a real copy, copy each row as well: `b[i] = a[i].clone()` .

Step 6 / 6

```
1 int[][] a = {{1, 2}, {3, 4}};
2 int[][] b = a.clone();
3 b[0][0] = 99;
4 b[1] = new int[]{7, 8};
5 System.out.println(Arrays.deepToString(a));
▶ System.out.println(Arrays.deepToString(b));
```

VARIABLES

a = **[[99, 2], [3, 4]]**

b = **[[99, 2], [7, 8]]**

OUTPUT

```
[[99, 2], [3, 4]]
[[99, 2], [7, 8]]
```

Traversing a Matrix Row by Row

i ↓ j →	0	1	2
0	1	2	3
1	4	5	6

A matrix of 2 rows × 3 columns; `matrix.length = 2`.

Step 1 / 14

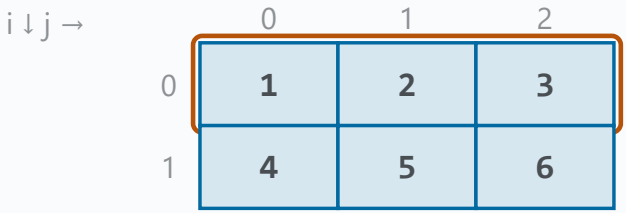
```
▶ int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

matrix =	[[1, 2, 3], [4, 5, 6]]

OUTPUT

Traversing a Matrix Row by Row



Outer loop: **0 < 2** → **true** → process row 0.

Step 2 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
▶ for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

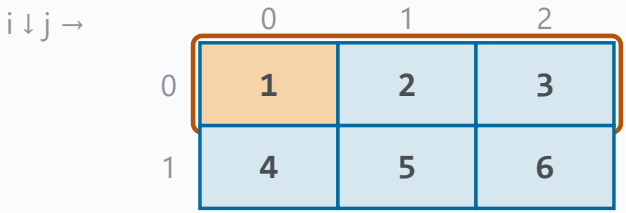
VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 0`

OUTPUT

Traversing a Matrix Row by Row



Inner loop: **0 < 3** → **true**; prints `matrix[0][0] = 1` (print stays on the line).

Step 3 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
▶     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

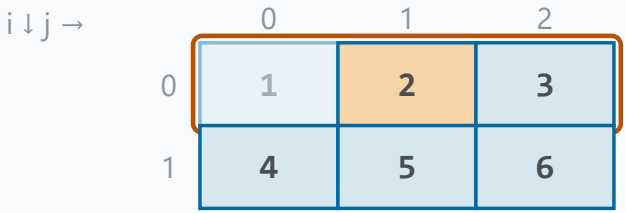
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 0` `j = 0`

OUTPUT

```
1
```

Traversing a Matrix Row by Row



Inner loop: **1 < 3** → **true**; prints `matrix[0][1] = 2` (print stays on the line).

Step 4 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
▶       System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

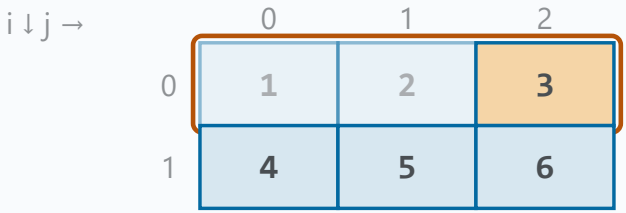
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 0` `j = 1`

OUTPUT

```
1 2
```

Traversing a Matrix Row by Row



Inner loop: **2 < 3** → **true**; prints `matrix[0][2] = 3` (print stays on the line).

Step 5 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
▶     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

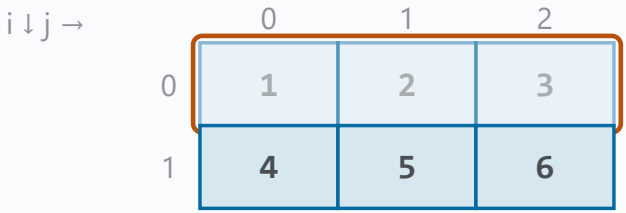
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 0` `j = 2`

OUTPUT

```
1 2 3
```

Traversing a Matrix Row by Row



3 < 3 → false: the row is done; leave the inner loop.

Step 6 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
▶   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

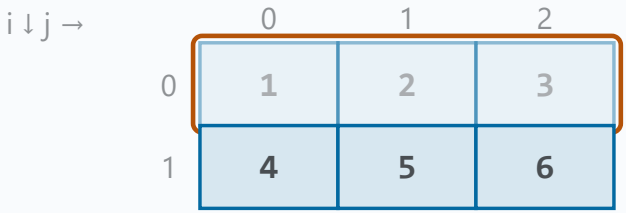
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 0` `j = 3`

OUTPUT

```
1 2 3
```

Traversing a Matrix Row by Row



`println()` moves to the next line; then `i++`.

Step 7 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
▶   System.out.println();
7 }
```

VARIABLES

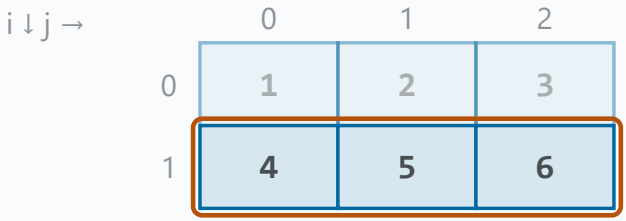
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 0`

OUTPUT

```
1 2 3
```

Traversing a Matrix Row by Row



Outer loop: **1 < 2** → **true** → process row 1.

Step 8 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
▶ for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

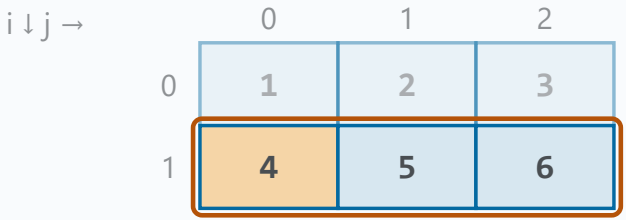
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 1`

OUTPUT

```
1 2 3
```

Traversing a Matrix Row by Row



Inner loop: **0 < 3** → **true**; prints `matrix[1][0] = 4` (print stays on the line).

Step 9 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
▶     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

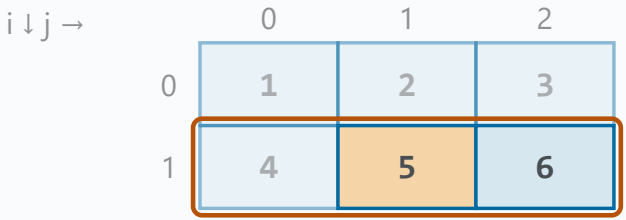
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 1` `j = 0`

OUTPUT

```
1 2 3
4
```

Traversing a Matrix Row by Row



Inner loop: **1 < 3** → **true**; prints `matrix[1][1] = 5` (print stays on the line).

Step 10 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
▶     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

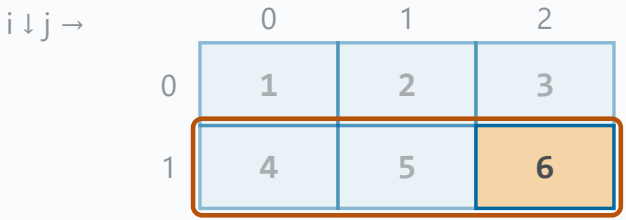
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 1` `j = 1`

OUTPUT

```
1 2 3
4 5
```

Traversing a Matrix Row by Row



Inner loop: **2 < 3** → **true**; prints `matrix[1][2] = 6` (print stays on the line).

Step 11 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
▶     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

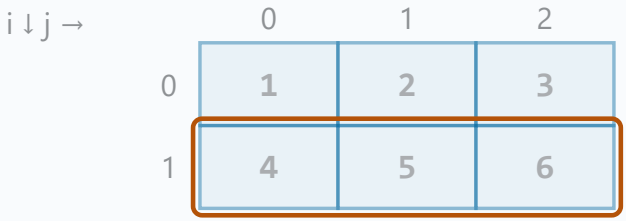
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 1` `j = 2`

OUTPUT

```
1 2 3
4 5 6
```

Traversing a Matrix Row by Row



3 < 3 → false: the row is done; leave the inner loop.

Step 12 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
▶   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

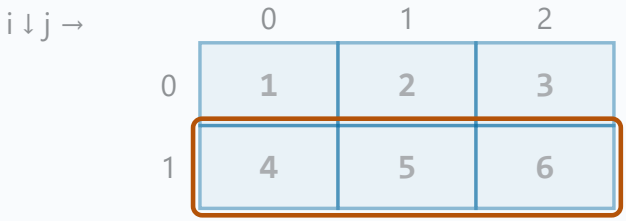
matrix = [[1, 2, 3], [4, 5, 6]]

i = 1 j = 3

OUTPUT

```
1 2 3
4 5 6
```

Traversing a Matrix Row by Row



`println()` moves to the next line; then `i++`.

Step 13 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
▶   System.out.println();
7 }
```

VARIABLES

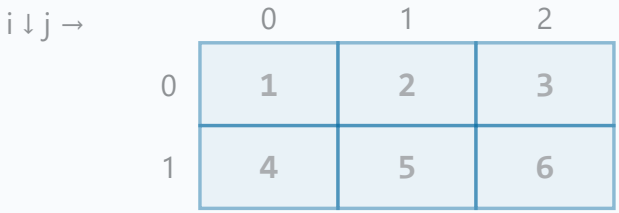
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 1`

OUTPUT

```
1 2 3
4 5 6
```

Traversing a Matrix Row by Row



Outer loop: **2 < 2** → **false** → done. 2 × 3 = 6 cells visited in all.

Step 14 / 14

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
▶ for (int i = 0; i < matrix.length; i++) {
3   for (int j = 0; j < matrix[i].length; j++) {
4     System.out.print(matrix[i][j] + " ");
5   }
6   System.out.println();
7 }
```

VARIABLES

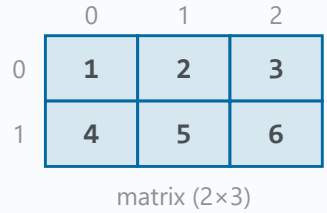
`matrix = [[1, 2, 3], [4, 5, 6]]`

`i = 2`

OUTPUT

```
1 2 3
4 5 6
```

Matrix Transpose



matrix (2×3)

The source matrix is 2 × 3.

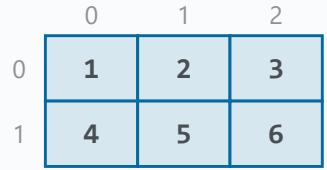
Step 1 / 11

```
▶ int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
7         t[j][i] = matrix[i][j];
```

VARIABLES

matrix =	[[1, 2, 3], [4, 5, 6]]

Matrix Transpose



matrix (2×3)

`matrix.length` = number of rows = **2**.

Step 2 / 11

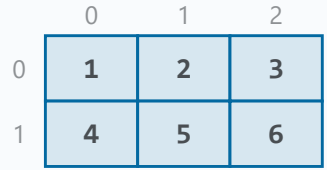
```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
▶ int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
7         t[j][i] = matrix[i][j];
```

VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6]]`

`rows = 2`

Matrix Transpose



matrix (2×3)

`matrix[0].length` = length of the first row = **3**.

Step 3 / 11

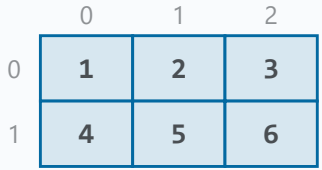
```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
▶ int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
7         t[j][i] = matrix[i][j];
```

VARIABLES

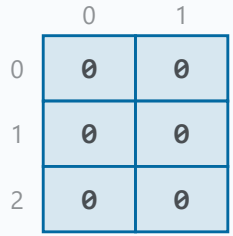
`matrix` = `[[1, 2, 3], [4, 5, 6]]`

`rows` = **2** `cols` = **3**

Matrix Transpose



matrix (2×3)



t (3×2)

The target `t` has the dimensions swapped: **3 × 2**, all cells 0.

Step 4 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
▶ int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
7         t[j][i] = matrix[i][j];
```

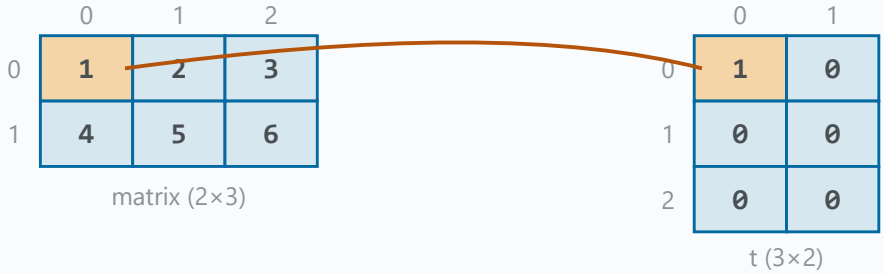
VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6]]`

`rows = 2` `cols = 3`

`t = [[0, 0], [0, 0], [0, 0]]`

Matrix Transpose



`t[0][0] = matrix[0][0]` = **1**: rows ↔ columns swap.

Step 5 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
▶         t[j][i] = matrix[i][j];
```

VARIABLES

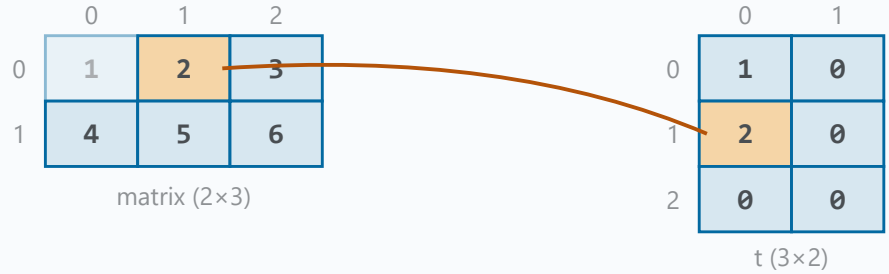
`matrix = [[1, 2, 3], [4, 5, 6]]`

`rows = 2` `cols = 3` `i = 0`

`j = 0`

`t = [[1, 0], [0, 0], [0, 0]]`

Matrix Transpose



`t[1][0] = matrix[0][1]` = **2**: rows ↔ columns swap.

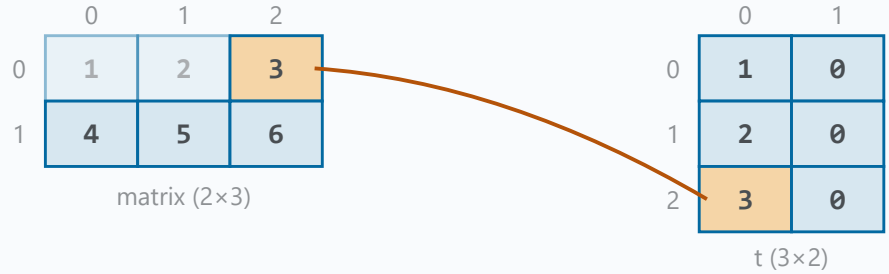
Step 6 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
▶         t[j][i] = matrix[i][j];
```

VARIABLES

- `matrix = [[1, 2, 3], [4, 5, 6]]`
- `rows = 2`
- `cols = 3`
- `i = 0`
- `j = 1`
- `t = [[1, 0], [2, 0], [0, 0]]`

Matrix Transpose



`t[2][0] = matrix[0][2]` = **3**: rows ↔ columns swap.

Step 7 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
▶         t[j][i] = matrix[i][j];
```

VARIABLES

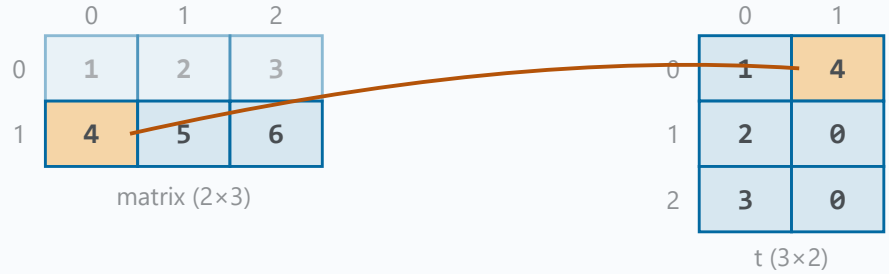
`matrix = [[1, 2, 3], [4, 5, 6]]`

`rows = 2` `cols = 3` `i = 0`

`j = 2`

`t = [[1, 0], [2, 0], [3, 0]]`

Matrix Transpose



`t[0][1] = matrix[1][0]` = **4**: rows ↔ columns swap.

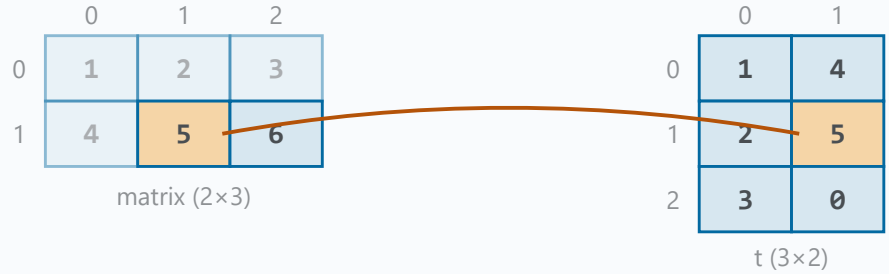
Step 8 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
▶         t[j][i] = matrix[i][j];
```

VARIABLES

- `matrix = [[1, 2, 3], [4, 5, 6]]`
- `rows = 2`
- `cols = 3`
- `i = 1`
- `j = 0`
- `t = [[1, 4], [2, 0], [3, 0]]`

Matrix Transpose



`t[1][1] = matrix[1][1]` = **5**: rows ↔ columns swap.

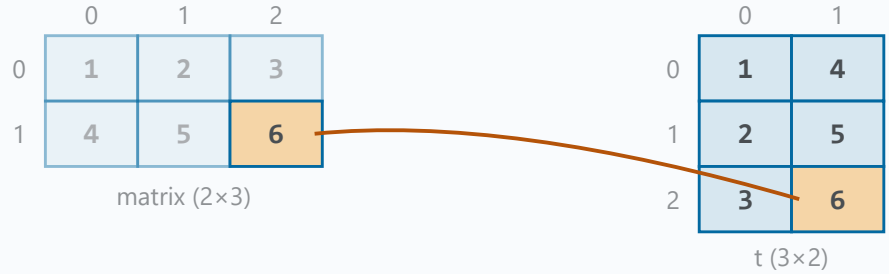
Step 9 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
▶         t[j][i] = matrix[i][j];
```

VARIABLES

- `matrix = [[1, 2, 3], [4, 5, 6]]`
- `rows = 2`
- `cols = 3`
- `i = 1`
- `j = 1`
- `t = [[1, 4], [2, 5], [3, 0]]`

Matrix Transpose



`t[2][1] = matrix[1][2]` = **6**: rows ↔ columns swap.

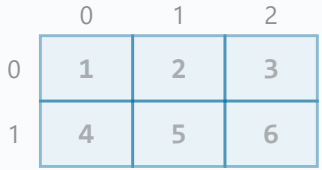
Step 10 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
5 for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
▶         t[j][i] = matrix[i][j];
```

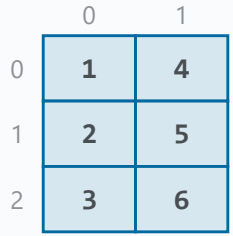
VARIABLES

- `matrix = [[1, 2, 3], [4, 5, 6]]`
- `rows = 2`
- `cols = 3`
- `i = 1`
- `j = 2`
- `t = [[1, 4], [2, 5], [3, 6]]`

Matrix Transpose



matrix (2×3)



t (3×2)

The loops are done: the **rows** of matrix became the **columns** of t.

Step 11 / 11

```
1 int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
2 int rows = matrix.length;
3 int cols = matrix[0].length;
4 int[][] t = new int[cols][rows];
▶ for (int i = 0; i < rows; i++)
6     for (int j = 0; j < cols; j++)
7         t[j][i] = matrix[i][j];
```

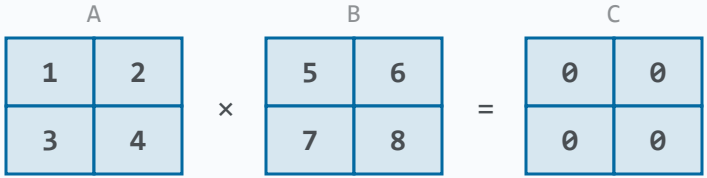
VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6]]`

`rows = 2` `cols = 3`

`t = [[1, 4], [2, 5], [3, 6]]`

Matrix Multiplication (2×2)



C[i][j] = the sum of products of **row i** of A and **column j** of B. C starts as zeros.

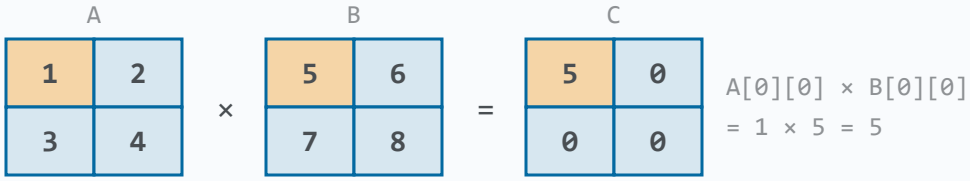
Step 1 / 10

```
▶ int[][] A = {{1, 2}, {3, 4}};
▶ int[][] B = {{5, 6}, {7, 8}};
▶ int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
7       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

C = [[0, 0], [0, 0]]

Matrix Multiplication (2×2)



C[0][0] += 1 × 5 → 0 + 5 = **5**

Step 2 / 10

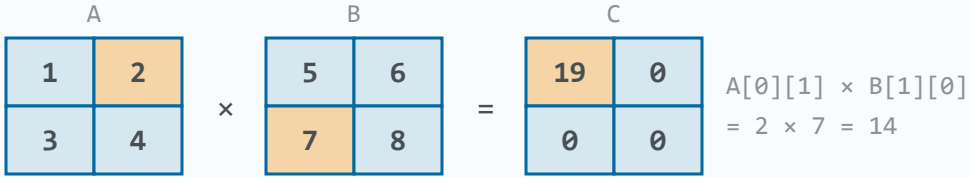
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 0` `j = 0` `k = 0`

`C = [[5, 0], [0, 0]]`

Matrix Multiplication (2×2)



C[0][0] += 2 × 7 → 5 + 14 = **19** (cell done)

Step 3 / 10

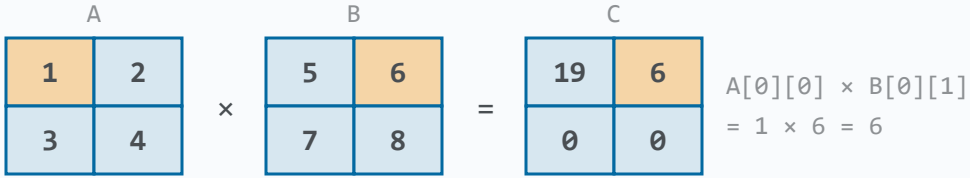
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 0` `j = 0` `k = 1`

`C = [[19, 0], [0, 0]]`

Matrix Multiplication (2×2)



C[0][1] += 1 × 6 → 0 + 6 = **6**

Step 4 / 10

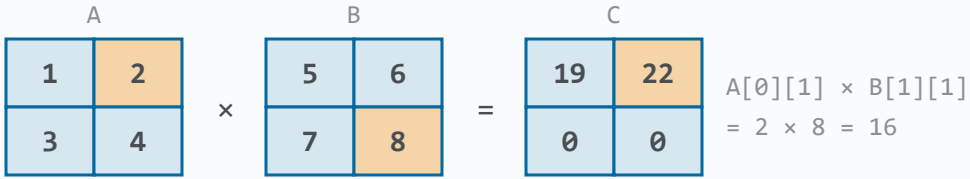
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 0` `j = 1` `k = 0`

`C = [[19, 6], [0, 0]]`

Matrix Multiplication (2×2)



C[0][1] += 2 × 8 → 6 + 16 = **22** (cell done)

Step 5 / 10

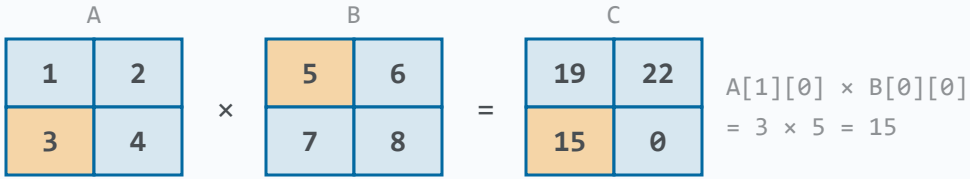
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 0` `j = 1` `k = 1`

`C = [[19, 22], [0, 0]]`

Matrix Multiplication (2×2)



C[1][0] += 3 × 5 → 0 + 15 = **15**

Step 6 / 10

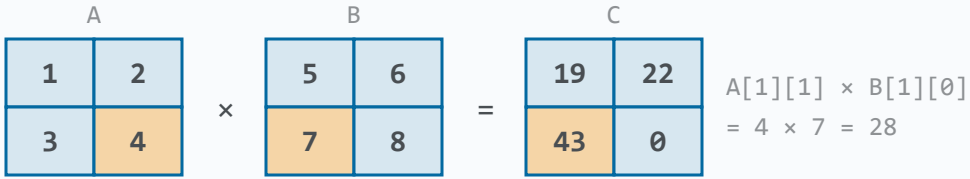
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 1` `j = 0` `k = 0`

`C = [[19, 22], [15, 0]]`

Matrix Multiplication (2×2)



C[1][0] += 4 × 7 → 15 + 28 = **43** (cell done)

Step 7 / 10

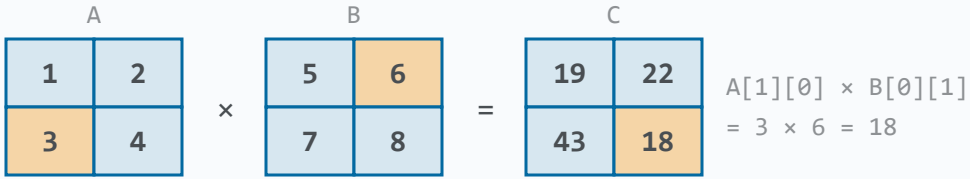
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 1` `j = 0` `k = 1`

`C = [[19, 22], [43, 0]]`

Matrix Multiplication (2×2)



C[1][1] += 3 × 6 → 0 + 18 = **18**

Step 8 / 10

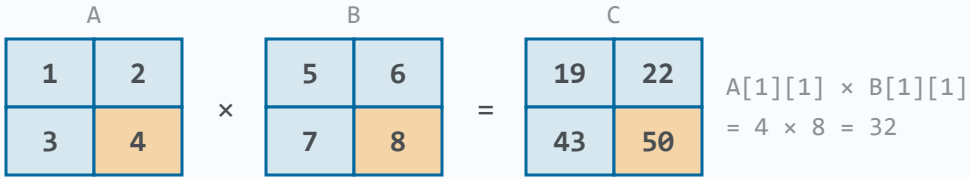
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 1` `j = 1` `k = 0`

`C = [[19, 22], [43, 18]]`

Matrix Multiplication (2×2)



C[1][1] += 4 × 8 → 18 + 32 = **50** (cell done)

Step 9 / 10

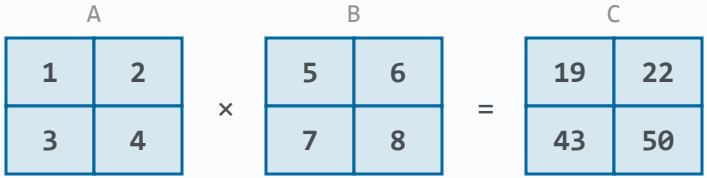
```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
4 for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
▶       C[i][j] += A[i][k] * B[k][j];
```

VARIABLES

`i = 1` `j = 1` `k = 1`

`C = [[19, 22], [43, 50]]`

Matrix Multiplication (2×2)



Three nested loops: n^3 multiplications for an n×n matrix → **$O(n^3)$**. The column count of A must equal the row count of B.

Step 10 / 10

```
1 int[][] A = {{1, 2}, {3, 4}};
2 int[][] B = {{5, 6}, {7, 8}};
3 int[][] C = new int[2][2];
▶ for (int i = 0; i < 2; i++)
5   for (int j = 0; j < 2; j++)
6     for (int k = 0; k < 2; k++)
7       C[i][j] += A[i][k] * B[k][j];
```

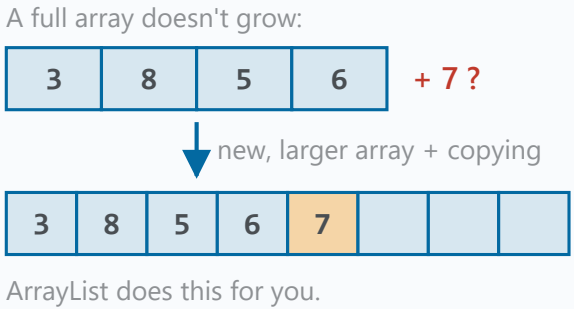
VARIABLES

C = [[19, 22], [43, 50]]

DESIGN

Array or List?

- If the number of elements is known and fixed, an array is simple and fast.
- If the count will change, collections such as `ArrayList` are more convenient.
- `int[]` stores the numbers directly; `ArrayList<Integer>` wraps each number in an object.
- Choose based on usage, readability and measured performance.



VERIFICATION

Common Mistakes

One pass too many

`i <= a.length` reaches `a[length]` on the last pass → `ArrayIndexOutOfBoundsException`. Use `i < a.length`.

First element of an empty array

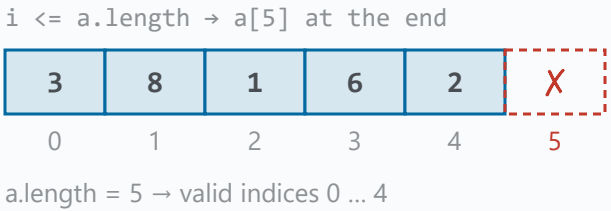
`int min = a[0];` fails on an empty array; check `a.length == 0` first.

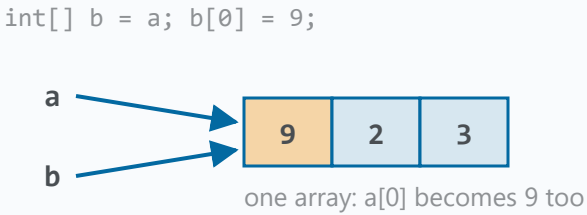
Copying the reference

`int[] b = a;` is not a new array; `b[0] = 9` changes a too. Use `a.clone()` for a copy.

Jagged matrices

Not every row has length `matrix[0].length`; use `matrix[i].length` in the inner loop.





SUMMARY

Chapter Summary

Structure

A fixed number of elements of one type; indices 0 … length − 1; the variable refers to the array.

Access

`a[i]` reads and writes in constant time; an out-of-bounds index is a run-time error.

Traversal

Classic for (when you need the index) and enhanced for (assigning to the loop variable leaves the array alone); sum, min-max, reversing, searching.

Matrices

`int[][]` is an array of arrays: `matrix.length` rows, `matrix[i].length` columns.

SUMMARY

Review Questions

1. What is the last index of a five-element array?
2. When is the enhanced for not enough?
3. What is the difference between assigning an array and copying its elements?
4. What precondition does binary search require?
5. What do `matrix.length` and `matrix[0].length` give?

▶ **Answers**

SUMMARY

Exercises

1. Write code that finds the count and sum of the even numbers in an `int[]`.
2. Write code that **rotates** an array one position right (last element first) with no extra array: {1,2,3,4} → {4,1,2,3}.
3. Merge two sorted arrays into a single sorted array (`{1,4,7}` + `{2,3,9}`).
4. Add a `boolean` flag to bubble sort that stops early when a pass makes no swaps.
5. Use the sieve of Eratosthenes to count the primes up to 100 (answer: 25).
6. Compute the main-diagonal sum and the transpose of a 3×3 matrix; check whether it is symmetric.

References

- Deitel and Deitel, Java: How to Program, Early Objects, 11th ed., Pearson, 2018.
- Liang, Introduction to Java Programming and Data Structures, 13th ed., Pearson, 2024.
- Oracle, The Java Tutorials: **Arrays**.
- Oracle, Java SE 21 API: **java.util.Arrays**; Java Language Specification SE 21: **Chapter 10, Arrays**.
- Sedgewick and Wayne, Computer Science: An Interdisciplinary Approach, Addison-Wesley, 2016: **1.4 Arrays**.
- Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms, 4th ed., MIT Press, 2022.

The End

Chapter 10: Arrays

Previous: Chapter 9: Methods · Next: Chapter 11: Strings