Chapter 3: Arrays

Data Structures

The array concept | Arrays in Java | Stack and heap | Array algorithms | Multidimensional arrays | Complexity | Sorting | Caesar cipher

Sercan KÜLCÜ · Giresun University · Computer Engineering

INTRODUCTION

Learning Outcomes

The array model

Explain how arrays are laid out in memory and how indexing works.

Arrays in Java

Declare and use one- and multidimensional arrays correctly.

Algorithms

Search, insert, delete, and sort on arrays.

Cost analysis

Compare the time and space costs of array operations.

BASIC CONCEPTS

Arrays

- Stores many values **of the same type**, in order, under a single name.
- Its size is fixed when the array is created and **cannot change** afterwards.
- The elements sit side by side in **contiguous** memory slots.
- Element values can always be updated; "inserting" or "deleting" means shifting elements.
- Used to represent matrices, images (pixels), sound (samples), and similar data.



arr.length = 5 · first index 0 · last index 4

BASIC CONCEPTS

Using Arrays: Access by Index

- An element is reached by its **index**: `arrayName[index]`.
- The first element has index **0**, the last **length − 1**.
- Because the elements are contiguous and equal in size, each address takes one multiplication and one addition to compute.



Click a cell: the top row is the index, the bottom row the value.

```
int[] arr = {7, 2, 9, 5, 1};
int first = arr[0];      // 7
int last = arr[arr.length - 1];      // 1
```

BASIC CONCEPTS

Kinds of Arrays

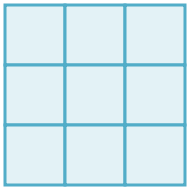
One-dimensional



Elements sit in a single row.

```
int[] a = new int[5];
```

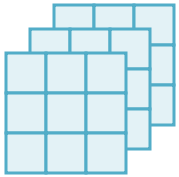
Two-dimensional



Elements sit in rows and columns (a matrix).

```
int[][] m = new int[3][3];
```

Multidimensional



More complex layouts, e.g. time × row × column.

```
int[][][] k = new int[3][3][3];
```

BASIC CONCEPTS

Arrays: Pros and Cons

Advantages

- \+ Fast, direct access by index: **O(1)**.
- \+ Groups data and keeps it in insertion order.
- \+ Holds primitives (`int` , `char` , ...) as well as objects.
- \+ Contiguous memory: no extra references, and cache-friendly.

Disadvantages

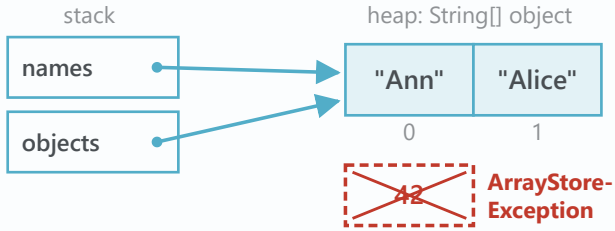
- – Fixed size: a full array cannot grow; you must create a bigger one and copy.
- – Inserting or deleting in the middle means shifting elements: **O(n)**.
- – Unused capacity is wasted.
- – Arrays have no methods for sorting, searching, and the like; `java.util.Arrays` provides them.

Java arrays are **covariant**, which buys flexibility at the cost of a hole in type safety.

Array Covariance and ArrayStoreException

- Because `String` is a subtype of `Object`, `String[]` counts as a subtype of `Object[]`.
- So one method that takes an `Object[]` works with any array of references.
- The price: a store of the wrong type is caught at run time, **not at compile time**.

```
String[] names = {"Ann", "Alice"};
Object[] objects = names; // compiles
objects[0] = 42;   // compiles, but
                   // throws at run time!
```



Flexibility

You can write general-purpose methods.

```
static void print(Object[] a)
```

Contrast

Generic lists are not covariant, so the error is caught at compile time.

```
List<Object> l = new ArrayList<String>
(); // error
```

Declaring Arrays

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

DataType
The type of the values the array stores.
`int, double, String…`

arrayName
The name of the reference variable that points to the array.
`nums`

arraySize
The number of elements; it may be computed at run time.
`5, n, 2 * n`

```
int[] nums = new int[5];
// → [0, 0, 0, 0, 0]
```

Every element of an array created with `new` starts at its type's **default value**.

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

ARRAYS IN JAVA

Access, Initialization, and `length`

Initialization
An initializer list supplies the starting values. You don't write the size: the **number of elements** sets it.

Access
Indices start at 0. Reading and writing use the same syntax.

length
The number of elements. It is a **field**, not a method: no parentheses.

```
int[] nums = {10, 20, 30, 40, 50};   // 5 initializers → size 5
int first    = nums[0];            // 10
int second = nums[1];             // 20
nums[4] = 55;                          // last element updated
int size   = nums.length;     // 5

nums = new int[] {1, 2, 3};          // later assignment needs new int[]
int[] bad = new int[3] {1, 2, 3};   // compile error: size + list
```

Code Trace: Reading and Writing an Array



The initializer list has **5 values** → a 5-element `int[]` is created on the heap; `nums` holds only its **reference**.

Step 1 / 6

```
▶ int[] nums = {10, 20, 30, 40, 50};
2 int first    = nums[0];
3 int second = nums[1];
4 nums[4] = 55;
5 int size   = nums.length;
6 nums = new int[] {1, 2, 3};
```

VARIABLES

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

The last line of the previous slide (`new int[3] {1, 2, 3}`) does not compile.

Code Trace: Reading and Writing an Array



Read: `nums[0]` is the first cell. Address = start + 0 × 4 → one step, O(1).

Step 2 / 6

```
1 int[] nums = {10, 20, 30, 40, 50};
▶ int first    = nums[0];
3 int second = nums[1];
4 nums[4] = 55;
5 int size   = nums.length;
6 nums = new int[] {1, 2, 3};
```

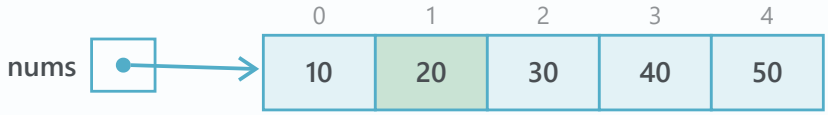
VARIABLES

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

`first = 10`

The last line of the previous slide (`new int[3] {1, 2, 3}`) does not compile.

Code Trace: Reading and Writing an Array



Index 1 is the **second** element (indices start at 0).

Step 3 / 6

```
1 int[] nums = {10, 20, 30, 40, 50};
2 int first    = nums[0];
▶ int second = nums[1];
4 nums[4] = 55;
5 int size   = nums.length;
6 nums = new int[] {1, 2, 3};
```

VARIABLES

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

`first = 10` `second = 20`

The last line of the previous slide (`new int[3] {1, 2, 3}`) does not compile.

Code Trace: Reading and Writing an Array



Write: same syntax, on the left of the assignment. The last cell (index 4 = length − 1) goes from 50 to **55**.

Step 4 / 6

```
1 int[] nums = {10, 20, 30, 40, 50};
2 int first    = nums[0];
3 int second = nums[1];
▶ nums[4] = 55;
5 int size   = nums.length;
6 nums = new int[] {1, 2, 3};
```

VARIABLES

`nums = [10, 20, 30, 40, 55]`

`first = 10` `second = 20`

The last line of the previous slide (`new int[3] {1, 2, 3}`) does not compile.

Code Trace: Reading and Writing an Array



`length` is a **field** (no parentheses) → 5. The size was fixed when the array was created.

Step 5 / 6

```
1 int[] nums = {10, 20, 30, 40, 50};
2 int first    = nums[0];
3 int second = nums[1];
4 nums[4] = 55;
▶ int size   = nums.length;
6 nums = new int[] {1, 2, 3};
```

VARIABLES

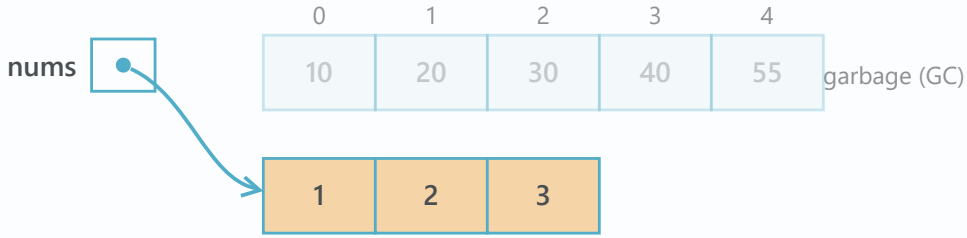
`nums = [10, 20, 30, 40, 55]`

`first = 10` `second = 20`

`size = 5`

The last line of the previous slide (`new int[3] {1, 2, 3}`) does not compile.

Code Trace: Reading and Writing an Array



A later assignment needs `new int[]` : a **new** 3-element array is created and `nums` points to it. The old array is unchanged; with no reference left, the GC may collect it. `size` is still 5: it was a copy.

Step 6 / 6

```
1 int[] nums = {10, 20, 30, 40, 50};
2 int first    = nums[0];
3 int second = nums[1];
4 nums[4] = 55;
5 int size   = nums.length;
▶ nums = new int[] {1, 2, 3};
```

VARIABLES

`nums = [1, 2, 3]` `first = 10` `second = 20` `size = 5`

The last line of the previous slide (`new int[3] {1, 2, 3}`) does not compile.

Example: Declare, Fill, Print

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] nums = new int[5];            // 5-element array
        for (int i = 0; i < nums.length; i++)
            nums[i] = (i + 1) * 10;             // 10, 20, 30, 40, 50

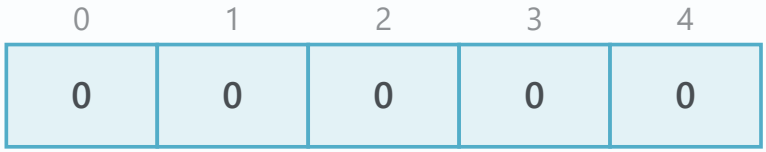
        System.out.println("First element: " + nums[0]);
        System.out.println("Third element: " + nums[2]);

        System.out.print("Elements: ");
        for (int x : nums)                      // for-each: when no index is needed
            System.out.print(x + " ");
    }
}
```

```
First element: 10
Third element: 30
Elements: 10 20 30 40 50
```

A loop, rather than five separate `nums[k] = …` assignments, keeps working if the size changes. The loop bound is always `i < nums.length`.

Code Trace: Fill and Print



`new int[5]`: 5 cells are allocated, all starting at **0**, the default for `int`.

Step 1 / 15

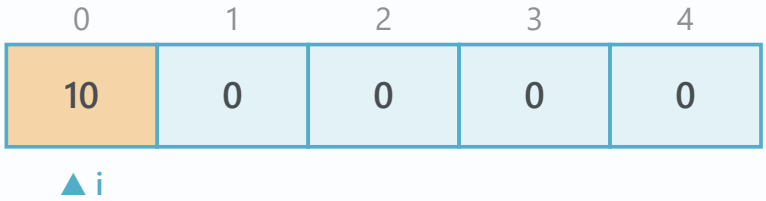
```
▶ int[] nums = new int[5];
2 for (int i = 0; i < nums.length; i++)
3     nums[i] = (i + 1) * 10;
4 System.out.println("First element: "
5         + nums[0]);
6 System.out.println("Third element: "
7         + nums[2]);
8 System.out.print("Elements: ");
9 for (int x : nums)
10    System.out.print(x + " ");
```

VARIABLES

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

OUTPUT

Code Trace: Fill and Print



i = 0 < 5 → **nums[0] = (0 + 1) × 10 = 10**.

Step 2 / 15

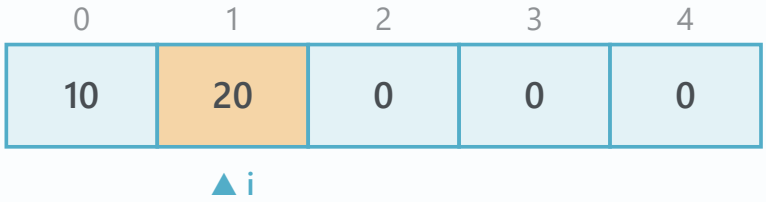
```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
▶      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

VARIABLES

`nums = [10, 0, 0, 0, 0]` `i = 0`

OUTPUT

Code Trace: Fill and Print



i = 1 < 5 → **nums[1] = (1 + 1) × 10 = 20**.

Step 3 / 15

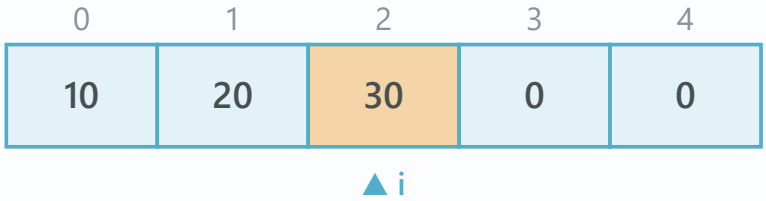
```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
▶      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

VARIABLES

`nums = [10, 20, 0, 0, 0]` `i = 1`

OUTPUT

Code Trace: Fill and Print



i = 2 < 5 → **nums[2] = (2 + 1) × 10 = 30**.

Step 4 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
▶      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

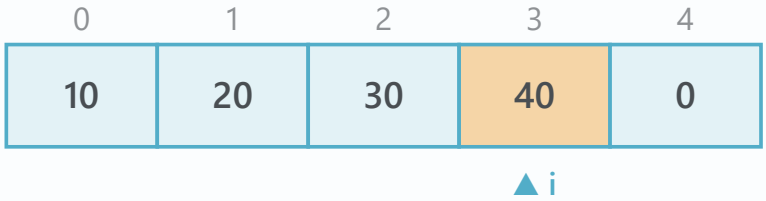
VARIABLES

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

`i = 2`

OUTPUT

Code Trace: Fill and Print



i = 3 < 5 → **nums[3] = (3 + 1) × 10 = 40**.

Step 5 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
▶      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

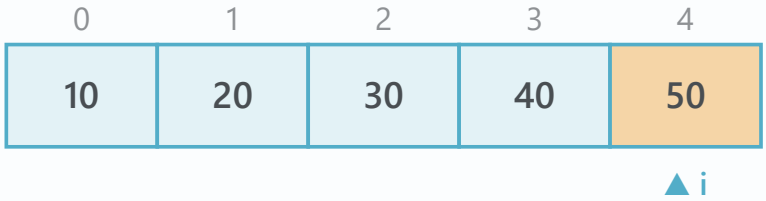
VARIABLES

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

`i = 3`

OUTPUT

Code Trace: Fill and Print



i = 4 < 5 → **nums[4] = (4 + 1) × 10 = 50**.

Step 6 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
▶      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

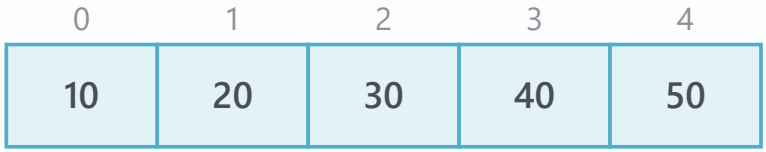
VARIABLES

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

`i = 4`

OUTPUT

Code Trace: Fill and Print



At i = 5, **5 < 5** is false → the loop ends. With `nums.length` as the bound, the code stays correct if the size changes.

Step 7 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
3      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

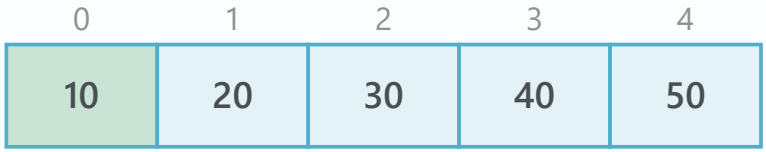
VARIABLES

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

`i = 5`

OUTPUT

Code Trace: Fill and Print



Index 0 is read → 10; `println` adds a newline.

Step 8 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
3      nums[i] = (i + 1) * 10;
▶  System.out.println("First element: "
▶          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

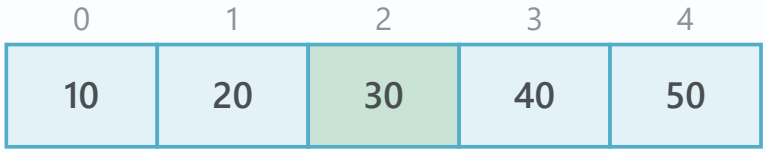
VARIABLES

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

OUTPUT

```
First element: 10
```

Code Trace: Fill and Print



The third element is at **index 2** → 30.

Step 9 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
3      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
▶  System.out.println("Third element: "
▶          + nums[2]);
8  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

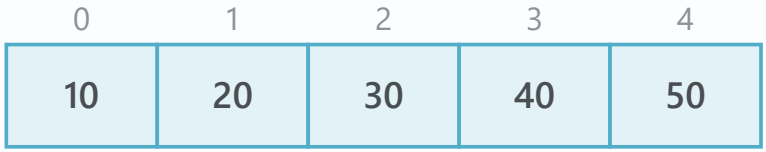
VARIABLES

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

OUTPUT

```
First element: 10
Third element: 30
```

Code Trace: Fill and Print



`print` adds no newline, so the elements will go on the same line.

Step 10 / 15

```
1  int[] nums = new int[5];
2  for (int i = 0; i < nums.length; i++)
3      nums[i] = (i + 1) * 10;
4  System.out.println("First element: "
5          + nums[0]);
6  System.out.println("Third element: "
7          + nums[2]);
▶  System.out.print("Elements: ");
9  for (int x : nums)
10     System.out.print(x + " ");
```

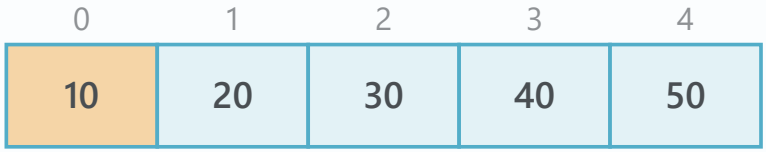
VARIABLES

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

OUTPUT

```
First element: 10
Third element: 30
Elements:
```

Code Trace: Fill and Print



for-each: **x = 10** (index 0). When you don't need an index variable, this reads better.

Step 11 / 15

```
1 int[] nums = new int[5];
2 for (int i = 0; i < nums.length; i++)
3     nums[i] = (i + 1) * 10;
4 System.out.println("First element: "
5         + nums[0]);
6 System.out.println("Third element: "
7         + nums[2]);
8 System.out.print("Elements: ");
▶ for (int x : nums)
▶     System.out.print(x + " ");
```

VARIABLES

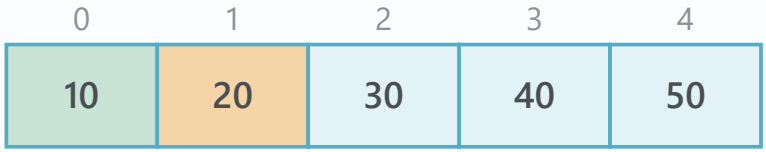
`nums = [10, 20, 30, 40, 50]`

`x = 10`

OUTPUT

```
First element: 10
Third element: 30
Elements: 10
```

Code Trace: Fill and Print



x = 20 is printed.

Step 12 / 15

```
1 int[] nums = new int[5];
2 for (int i = 0; i < nums.length; i++)
3     nums[i] = (i + 1) * 10;
4 System.out.println("First element: "
5         + nums[0]);
6 System.out.println("Third element: "
7         + nums[2]);
8 System.out.print("Elements: ");
▶ for (int x : nums)
▶     System.out.print(x + " ");
```

VARIABLES

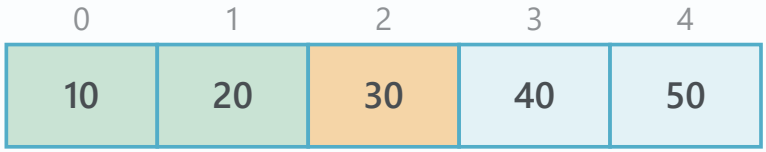
`nums = [10, 20, 30, 40, 50]`

`x = 20`

OUTPUT

```
First element: 10
Third element: 30
Elements: 10 20
```

Code Trace: Fill and Print



x = 30 is printed.

Step 13 / 15

```
1 int[] nums = new int[5];
2 for (int i = 0; i < nums.length; i++)
3     nums[i] = (i + 1) * 10;
4 System.out.println("First element: "
5         + nums[0]);
6 System.out.println("Third element: "
7         + nums[2]);
8 System.out.print("Elements: ");
▶ for (int x : nums)
▶     System.out.print(x + " ");
```

VARIABLES

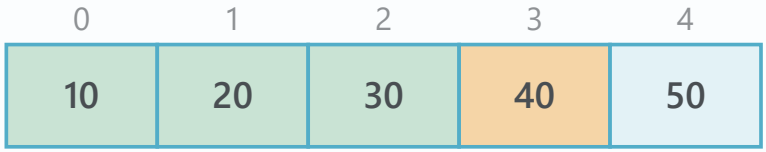
nums = [10, 20, 30, 40, 50]

x = 30

OUTPUT

```
First element: 10
Third element: 30
Elements: 10 20 30
```

Code Trace: Fill and Print



x = 40 is printed.

Step 14 / 15

```
1 int[] nums = new int[5];
2 for (int i = 0; i < nums.length; i++)
3     nums[i] = (i + 1) * 10;
4 System.out.println("First element: "
5         + nums[0]);
6 System.out.println("Third element: "
7         + nums[2]);
8 System.out.print("Elements: ");
▶ for (int x : nums)
▶     System.out.print(x + " ");
```

VARIABLES

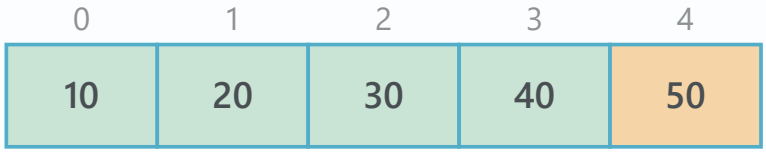
nums = [10, 20, 30, 40, 50]

x = 40

OUTPUT

```
First element: 10
Third element: 30
Elements: 10 20 30 40
```

Code Trace: Fill and Print



x = 50 is printed. Every element has been visited → the loop ends.

Step 15 / 15

```
1 int[] nums = new int[5];
2 for (int i = 0; i < nums.length; i++)
3     nums[i] = (i + 1) * 10;
4 System.out.println("First element: "
5         + nums[0]);
6 System.out.println("Third element: "
7         + nums[2]);
8 System.out.print("Elements: ");
▶ for (int x : nums)
▶     System.out.print(x + " ");
```

VARIABLES

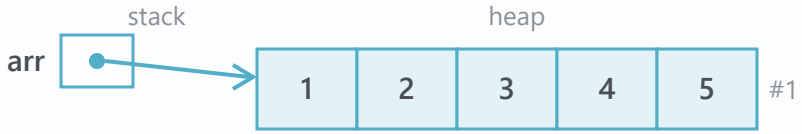
`nums = [10, 20, 30, 40, 50]`

`x = 50`

OUTPUT

```
First element: 10
Third element: 30
Elements: 10 20 30 40 50
```

Copying Arrays: Reference or Copy?



An array is an **object**, created on the heap (#1). The variable `arr` on the stack holds only its address.

Step 1 / 4

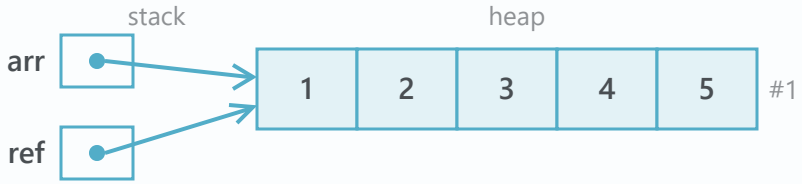
```
▶ int[] arr = {1, 2, 3, 4, 5};
2 int[] ref = arr;   // same object
3 int[] copy =
4     Arrays.copyOf(arr, arr.length);
5 ref[0] = 99;
```

VARIABLES

`arr = →#1`

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

Copying Arrays: Reference or Copy?



`=` copies only the **address**: both variables point to the **same** array (#1). No new array is created.

Step 2 / 4

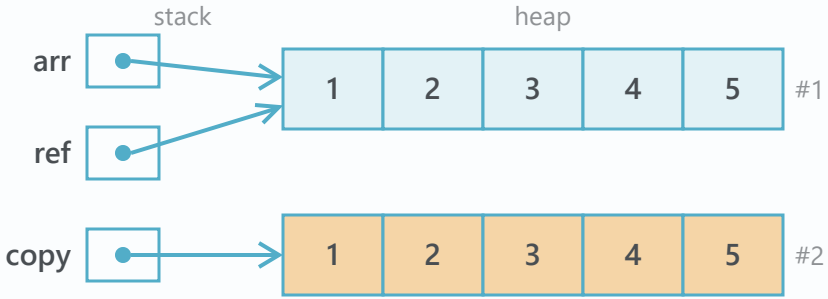
```
1 int[] arr = {1, 2, 3, 4, 5};
▶ int[] ref = arr;   // same object
3 int[] copy =
4     Arrays.copyOf(arr, arr.length);
5 ref[0] = 99;
```

VARIABLES

`arr = →#1` `ref = →#1`

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

Copying Arrays: Reference or Copy?



`Arrays.copyOf` creates a **new** array (#2) on the heap and copies the elements one by one → O(n). (`arr.clone()` and `System.arraycopy` copy elements too.)

Step 3 / 4

```
1 int[] arr = {1, 2, 3, 4, 5};
2 int[] ref = arr;   // same object
▶ int[] copy =
▶     Arrays.copyOf(arr, arr.length);
5 ref[0] = 99;
```

VARIABLES

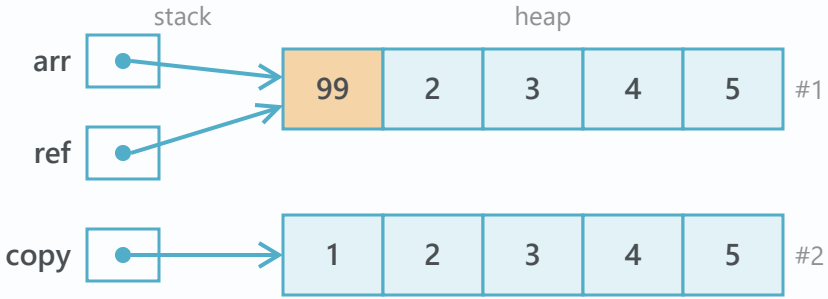
`arr = →#1` `ref = →#1`

`copy = →#2`

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

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

Copying Arrays: Reference or Copy?



`ref[0] = 99` changes #1 → `arr[0]` becomes **99** too. `copy[0]` lives in a separate object and is still **1**.

Step 4 / 4

```
1 int[] arr = {1, 2, 3, 4, 5};
2 int[] ref = arr;   // same object
3 int[] copy =
4     Arrays.copyOf(arr, arr.length);
▶ ref[0] = 99;
```

VARIABLES

`arr = →#1` `ref = →#1`

`copy = →#2`

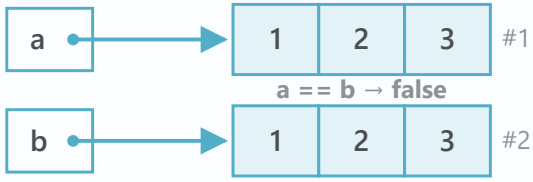
`#1 = [99, 2, 3, 4, 5]`

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

Ready-Made Array Operations: `java.util.Arrays`

Method	What it does	Cost
`Arrays.toString(a)`	Returns a string like `"[1, 2, 3]"`	O(n)
`Arrays.equals(a, b)`	Same length and equal elements, one by one?	O(n)
`Arrays.fill(a, v)`	Writes `v` into every cell	O(n)
`Arrays.copyOf(a, n)`	New array; if `n > a.length`, the extra cells get the default value	O(n)
`Arrays.sort(a)`	Sorts in ascending order (in place)	O(n log n)
`Arrays.binarySearch(a, x)`	Binary search in a **sorted** array; negative result if absent	O(log n)

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
a == b              // false: different objects
Arrays.equals(a, b) // true: same contents
println(a)          // [I@1b6d3586
```



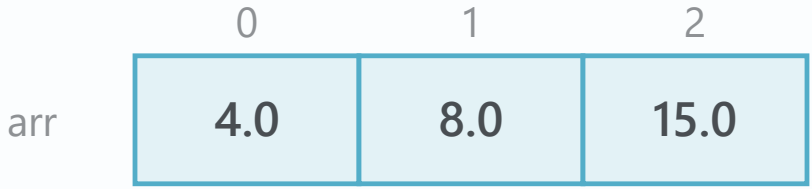
Varargs (Variable-Length Arguments)

- The `type... name` syntax lets a method take any number of arguments.
- Inside the method, the parameter is an ordinary **array**.
- Only the **last** parameter can be varargs, so a method has at most one.
- You can also pass an existing array directly.

```
static double average(double... arr) {
    if (arr.length == 0)
        return Double.NaN;  // undefined
    var sum = 0.0;
    for (var value : arr)
        sum += value;
    return sum / arr.length;
}
```

Call	arr.length	Result
`average(4, 8, 15)`	3	9.0
`average(7.5)`	1	7.5
`average()`	0	NaN
`average(new double[] {1, 2})`	2	1.5

Code Trace: `average(4, 8, 15)`



For the call `average(4, 8, 15)` the compiler packs the arguments into a **double[]** (each int is widened to double).

Step 1 / 10

```
▶ static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
5     for (var value : arr)
6         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

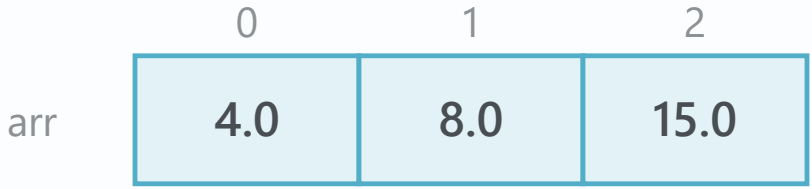
`arr = [4.0, 8.0, 15.0]`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



`arr.length` = 3 ≠ 0 → not the empty case, so carry on.

Step 2 / 10

```
1 static double average(double... arr) {
▶     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
5     for (var value : arr)
6         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

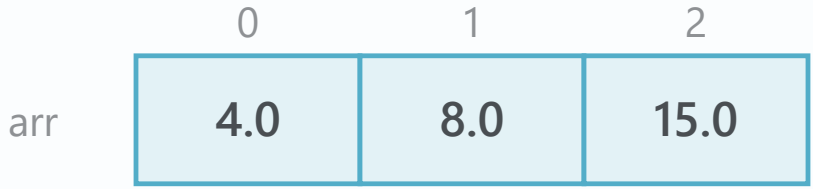
`arr = [4.0, 8.0, 15.0]`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



`var` : the type is inferred from the value on the right → `double` .

Step 3 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
  ▶   var sum = 0.0;
5     for (var value : arr)
6         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

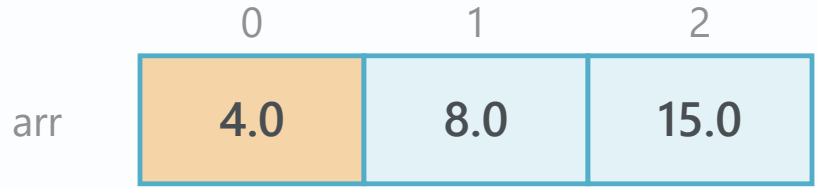
`sum = 0.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



for-each: **value = 4.0** (element 1 of the array).

Step 4 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
▶     for (var value : arr)
6         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

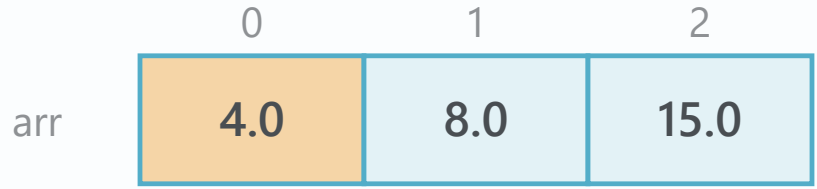
`sum = 0.0` `value = 4.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



sum = 0.0 + 4.0 = **4.0**.

Step 5 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
5     for (var value : arr)
▶         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

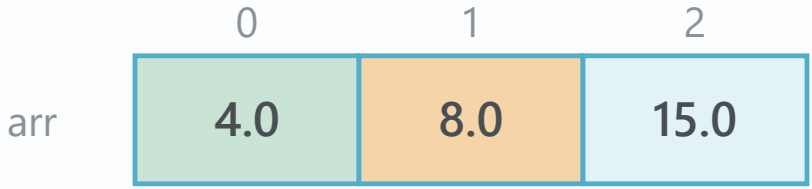
`sum = 4.0` `value = 4.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



for-each: **value = 8.0** (element 2 of the array).

Step 6 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
▶     for (var value : arr)
6         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

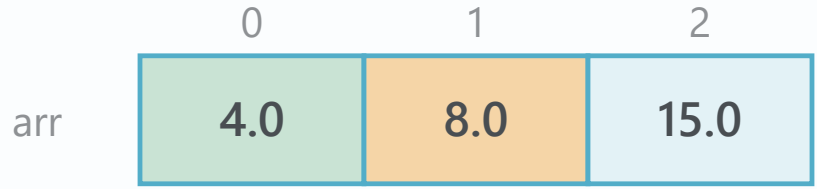
`sum = 4.0` `value = 8.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



sum = 4.0 + 8.0 = **12.0**.

Step 7 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
5     for (var value : arr)
▶         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

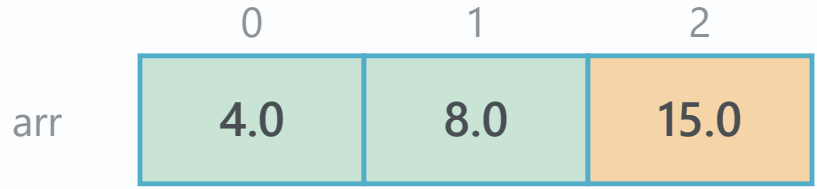
`sum = 12.0` `value = 8.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



for-each: **value = 15.0** (element 3 of the array).

Step 8 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
▶     for (var value : arr)
6         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

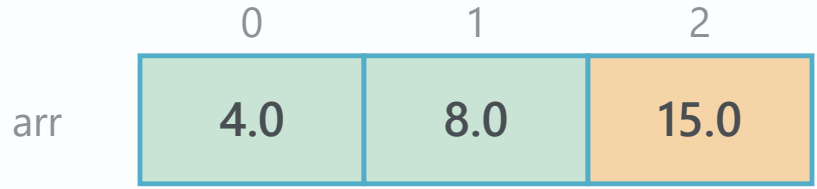
`sum = 12.0` `value = 15.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



sum = 12.0 + 15.0 = **27.0**.

Step 9 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
5     for (var value : arr)
▶         sum += value;
7     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

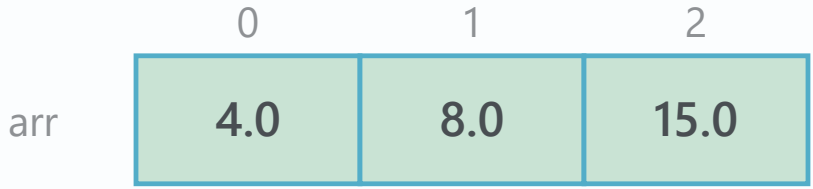
`sum = 27.0` `value = 15.0`

CALL STACK

`average(4, 8, 15)`

`main()`

Code Trace: `average(4, 8, 15)`



The loop is done. **return 27.0 / 3 = 9.0**. A call to `average()` would have length 0 and return `NaN` on line 3.

Step 10 / 10

```
1 static double average(double... arr) {
2     if (arr.length == 0)
3         return Double.NaN;  // undefined
4     var sum = 0.0;
5     for (var value : arr)
6         sum += value;
▶     return sum / arr.length;
8 }
```

VARIABLES

`arr = [4.0, 8.0, 15.0]`

`sum = 27.0`

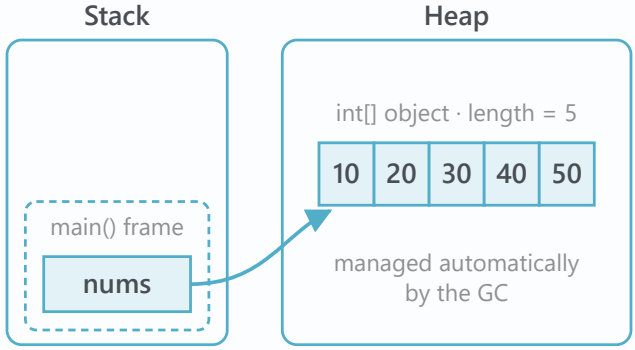
CALL STACK

`average(4, 8, 15)`

`main()`

Arrays in Memory: Stack and Heap

- **Stack**: local variables and method calls. When a method returns, its frame is removed automatically.
- **Heap**: every object created with `new`. In Java **every array** lives on the heap; only its reference sits on the stack.
- The heap is managed by the **garbage collector** (GC), which frees objects that no reference points to.



	Stack	Heap
Holds	local variables, references, call frames	objects, arrays
Lifetime	until the method returns	as long as it is reachable
Managed by	automatic (call / return)	automatic (GC)

ARRAYS IN JAVA

"Static" and "Dynamic" Arrays: C vs. Java

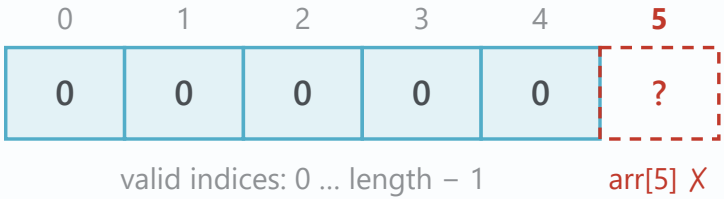
	C / C++	Java
Array on the stack	`int a[10];` — size fixed at compile time; gone when the function returns	None: every array is on the heap
Array on the heap	`malloc` / `new int[n]`	`new int[n]` (initializer lists too)
Size	on the heap, set at run time	set at run time, then fixed
Freeing	the programmer: `free` / `delete[]`	automatic GC; `arr = null` only drops the reference

```
int[] dynArr = new int[n];    // room for n elements on the heap
dynArr = null;                // GC may collect it if no other reference exists
```

`System.gc()` is only a **request** and guarantees nothing; Java has no `delete`. "Dynamic array" also names an array that grows by itself when full (`ArrayList`).

Index Out of Bounds

- Accessing an index outside the valid range makes Java throw an `ArrayIndexOutOfBoundsException` (a subclass of `IndexOutOfBoundsException`).
- If it isn't caught, the program stops. Java checks every access; C doesn't, and silently writes to some other memory.




```
int[] arr = new int[5];
arr[5] = 1;  // Index 5 out of
             // bounds for length 5
arr[-1] = 1; // same error
```

```
if (index >= 0
        && index < arr.length) {
    int value = arr[index];
} else {
    System.out.println("Invalid");
}
```

ARRAY ALGORITHMS

Reading an Array from the User

The mean, median, and mode programs all start by reading input the same way, so the shared code goes into a helper (the `int[]` version for the mode is identical):

```
import java.util.Scanner;

static double[] readArray(Scanner scanner) {
    System.out.print("Enter the array size: ");
    int size = scanner.nextInt();
    double[] arr = new double[size];  // size known only at run time
    for (int i = 0; i < size; i++) {
        System.out.print("Enter element " + (i + 1) + ": "); // count from 1
        arr[i] = scanner.nextDouble();
    }
    return arr;
}
```

Mean
Sum / number of elements
`O(n)`

Median
The middle of the sorted array
`O(n log n)`

Mode
The most frequent value
`O(n log n)`

Code Trace: `readArray(sc)`

no array yet (size unknown)

First we ask for the size. `print` adds no newline, so the cursor waits on the same line.

Step 1 / 12

```
1  static double[] readArray(Scanner scanner) {
▶    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

—

OUTPUT

```
Enter the array size:
```

Code Trace: `readArray(sc)`

no array yet (size unknown)

The user types **4** and `nextInt()` reads it. (What the user types also appears on the console.)

Step 2 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
▶    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

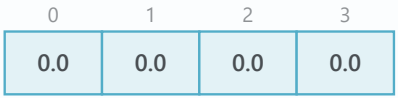
VARIABLES

`size = 4`

OUTPUT

```
Enter the array size: 4
```

Code Trace: `readArray(sc)`



The size became known **at run time** → `new double[4]`, every cell 0.0.

Step 3 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
▶    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

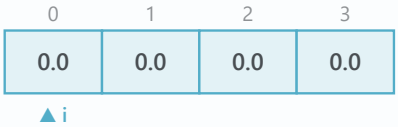
`size = 4`

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

OUTPUT

```
Enter the array size: 4
```

Code Trace: `readArray(sc)`



i = 0, but the prompt says element **1** (people count from 1).

Step 4 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
▶      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

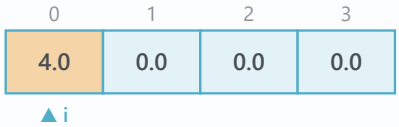
`size = 4`

`arr = [0.0, 0.0, 0.0, 0.0]` `i = 0`

OUTPUT

```
Enter the array size: 4
Enter element 1:
```

Code Trace: `readArray(sc)`



The user types 4 → **arr[0] = 4.0**.

Step 5 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

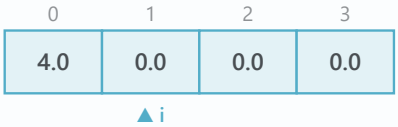
`size = 4`

`arr = [4.0, 0.0, 0.0, 0.0]` `i = 0`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
```

Code Trace: `readArray(sc)`



i = 1, but the prompt says element **2** (people count from 1).

Step 6 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
▶      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

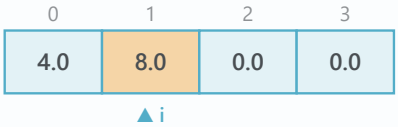
`size = 4`

`arr = [4.0, 0.0, 0.0, 0.0]` `i = 1`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2:
```

Code Trace: `readArray(sc)`



The user types 8 → **arr[1] = 8.0**.

Step 7 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

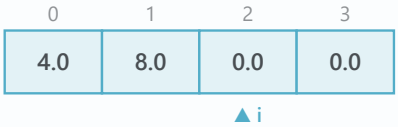
`size = 4`

`arr = [4.0, 8.0, 0.0, 0.0]` `i = 1`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2: 8
```

Code Trace: `readArray(sc)`



i = 2, but the prompt says element **3** (people count from 1).

Step 8 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
▶      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

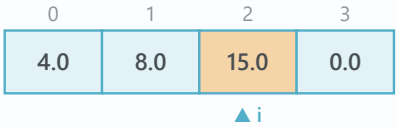
`size = 4`

`arr = [4.0, 8.0, 0.0, 0.0]` `i = 2`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2: 8
Enter element 3:
```

Code Trace: `readArray(sc)`



The user types 15 → **arr[2] = 15.0**.

Step 9 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

`size = 4`

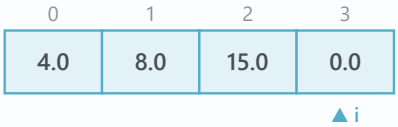
`arr = [4.0, 8.0, 15.0, 0.0]`

`i = 2`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2: 8
Enter element 3: 15
```

Code Trace: `readArray(sc)`



i = 3, but the prompt says element **4** (people count from 1).

Step 10 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
▶      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

`size = 4`

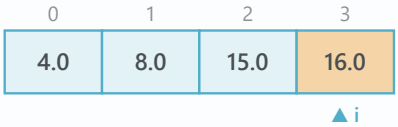
`arr = [4.0, 8.0, 15.0, 0.0]`

`i = 3`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2: 8
Enter element 3: 15
Enter element 4:
```

Code Trace: `readArray(sc)`



The user types 16 → **arr[3] = 16.0**. At i = 4 the loop ends.

Step 11 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
▶      arr[i] = scanner.nextDouble();
8    }
9    return arr;
10 }
```

VARIABLES

`size = 4`

`arr = [4.0, 8.0, 15.0, 16.0]`

`i = 3`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2: 8
Enter element 3: 15
Enter element 4: 16
```

Code Trace: `readArray(sc)`



The method returns the array's **reference**; the array is on the heap, so it outlives the method.

Step 12 / 12

```
1  static double[] readArray(Scanner scanner) {
2    System.out.print("Enter the array size: ");
3    int size = scanner.nextInt();
4    double[] arr = new double[size];
5    for (int i = 0; i < size; i++) {
6      System.out.print("Enter element " + (i + 1) + ": ");
7      arr[i] = scanner.nextDouble();
8    }
▶    return arr;
10 }
```

VARIABLES

`size = 4`

`arr = [4.0, 8.0, 15.0, 16.0]`

OUTPUT

```
Enter the array size: 4
Enter element 1: 4
Enter element 2: 8
Enter element 3: 15
Enter element 4: 16
```

Computing the Mean

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double[] arr = readArray(sc);
    double sum = 0;
    for (int i = 0; i < arr.length; i++)
        sum += arr[i];
    double average = sum / arr.length;
    System.out.println("Array: "
        + Arrays.toString(arr));
    System.out.println("Average: "
        + average);
    sc.close();
}
```

Example

arr = {4, 8, 15, 16, 23}
sum = 66
average = 66 / 5 =
13.2

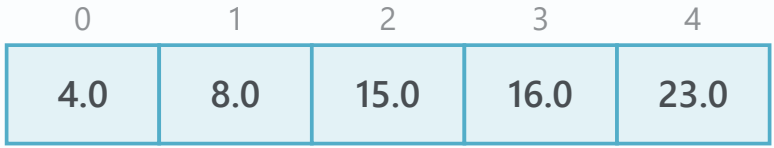
Cost

Each element is added once → **O(n)** time, O(1) extra space.

Edge case

With size 0 we get 0.0 / 0 = `NaN`; better to warn the user.

Code Trace: Computing the Mean



Suppose this time the user enters 5 values: `readArray` returns {4, 8, 15, 16, 23}.

Step 1 / 11

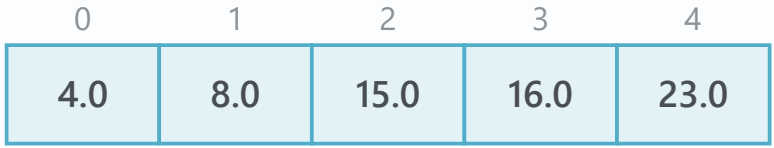
```
▶ double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
4     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

VARIABLES

arr =	[4.0, 8.0, 15.0, 16.0, 23.0]

OUTPUT

Code Trace: Computing the Mean



The accumulator starts at 0 (int 0 → double 0.0).

Step 2 / 11

```
1 double[] arr = readArray(sc);
▶ double sum = 0;
3 for (int i = 0; i < arr.length; i++)
4     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

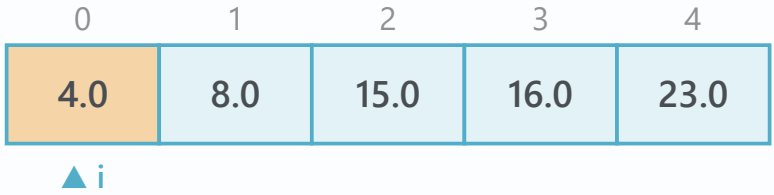
VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 0.0

OUTPUT

Code Trace: Computing the Mean



sum += arr[0] → 0.0 + 4.0 = **4.0**.

Step 3 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
      sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 4.0 i = 0

OUTPUT

Code Trace: Computing the Mean



sum += arr[1] → 4.0 + 8.0 = **12.0**.

Step 4 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
▶     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

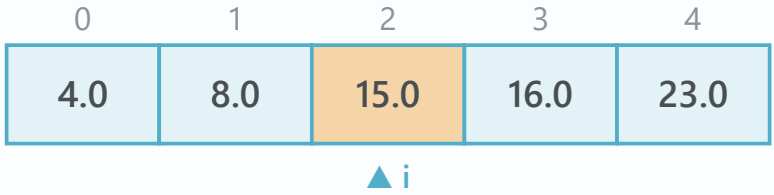
VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 12.0 | i = 1

OUTPUT

Code Trace: Computing the Mean



sum += arr[2] → 12.0 + 15.0 = **27.0**.

Step 5 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
▶     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 27.0 | i = 2

OUTPUT

Code Trace: Computing the Mean



sum += arr[3] → 27.0 + 16.0 = **43.0**.

Step 6 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
▶     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

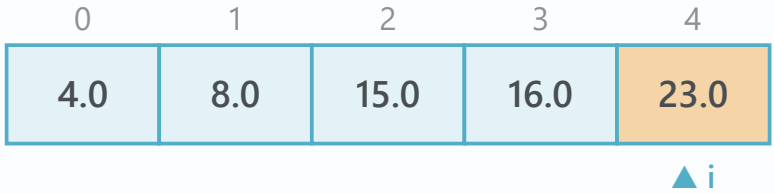
VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 43.0 | i = 3

OUTPUT

Code Trace: Computing the Mean



sum += arr[4] → 43.0 + 23.0 = **66.0**.

Step 7 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
▶     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

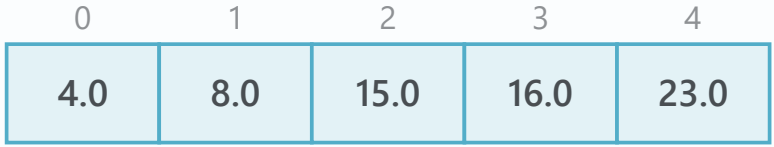
VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 66.0 | i = 4

OUTPUT

Code Trace: Computing the Mean



i = 5, and 5 < 5 is false → the loop ends. Each element was added **once** → O(n).

Step 8 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
▶ for (int i = 0; i < arr.length; i++)
4     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

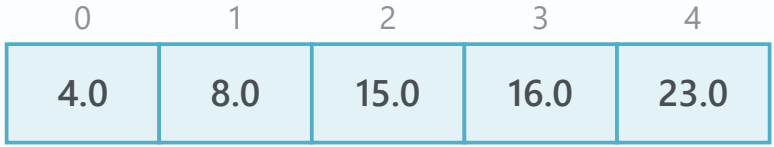
VARIABLES

arr = [4.0, 8.0, 15.0, 16.0, 23.0]

sum = 66.0 | i = 5

OUTPUT

Code Trace: Computing the Mean



average = 66.0 / 5 = **13.2**. With size 0 this would be 0.0 / 0 = NaN.

Step 9 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
4     sum += arr[i];
▶ double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

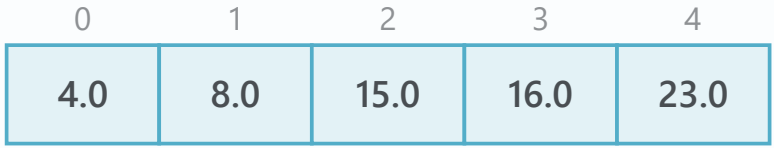
VARIABLES

`arr = [4.0, 8.0, 15.0, 16.0, 23.0]`

`sum = 66.0` `average = 13.2`

OUTPUT

Code Trace: Computing the Mean



`Arrays.toString` turns the array into text of the form `[a, b, …]`. (`println(arr)` would print only type@address.)

Step 10 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
4     sum += arr[i];
5 double average = sum / arr.length;
▶ System.out.println("Array: "
▶     + Arrays.toString(arr));
8 System.out.println("Average: "
9     + average);
```

VARIABLES

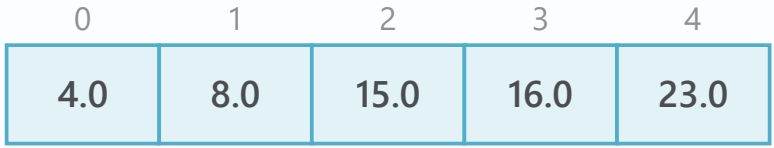
`arr` = `[4.0, 8.0, 15.0, 16.0, 23.0]`

`sum` = `66.0` `average` = `13.2`

OUTPUT

```
Array: [4.0, 8.0, 15.0, 16.0, 23.0]
```

Code Trace: Computing the Mean



The result is printed. Overall: O(n) time, O(1) extra space.

Step 11 / 11

```
1 double[] arr = readArray(sc);
2 double sum = 0;
3 for (int i = 0; i < arr.length; i++)
4     sum += arr[i];
5 double average = sum / arr.length;
6 System.out.println("Array: "
7     + Arrays.toString(arr));
▶ System.out.println("Average: "
▶     + average);
```

VARIABLES

`arr` = `[4.0, 8.0, 15.0, 16.0, 23.0]`

`sum` = `66.0` `average` = `13.2`

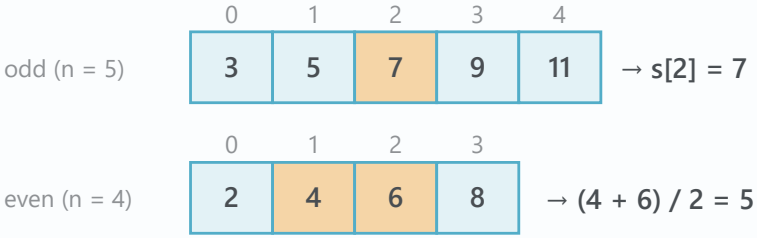
OUTPUT

```
Array: [4.0, 8.0, 15.0, 16.0, 23.0]
Average: 13.2
```

Computing the Median

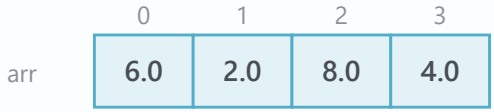
```
static double median(double[] arr) {
    double[] s = arr.clone(); // copy
    Arrays.sort(s);              // O(n log n)
    int size = s.length;
    double median;
    if (size % 2 == 0) {       // even
        int m1 = size/2 - 1, m2 = size/2;
        median = (s[m1] + s[m2]) / 2;
    } else {                     // odd
        median = s[size / 2];
    }
    return median;
}
```

- The median is the middle value of the **sorted** array.
- For an odd size it is the exact middle; for an even size, the mean of the two middle values.
- Unlike the mean, it isn't thrown off by outliers (e.g. 1000).



Code Trace: `median()`

```
▶ static double median(double[] arr) {
2     double[] s = arr.clone(); // copy
3     Arrays.sort(s);              // O(n log n)
4     int size = s.length;
5     double median;
6     if (size % 2 == 0) {       // even
7         int m1 = size/2 - 1, m2 = size/2;
8         median = (s[m1] + s[m2]) / 2;
9     } else {                   // odd
10        median = s[size / 2];
11    }
12    return median;
13 }
```



Call 1 (even size): `median` is called with a 4-element array.

Step 1 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

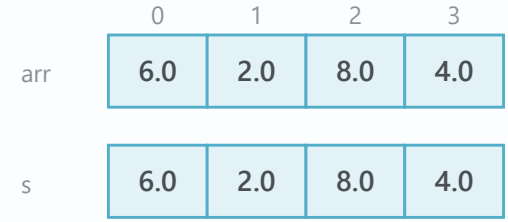
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
▶      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



Make a **copy** first, so the caller's array keeps its order.

Step 2 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [6.0, 2.0, 8.0, 4.0]`

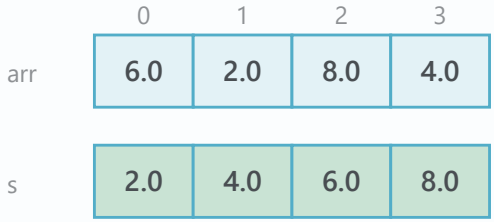
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
▶      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {       // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                   // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



The copy is sorted; the original `arr` is unchanged.

Step 3 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [2.0, 4.0, 6.0, 8.0]`

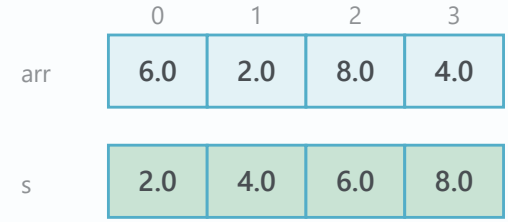
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
▶      int size = s.length;
5      double median;
6      if (size % 2 == 0) {        // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



size = 4.

Step 4 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [2.0, 4.0, 6.0, 8.0]` `size = 4`

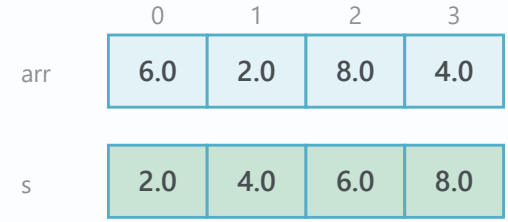
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
▶      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



4 % 2 == 0 → **even**: there is no single middle element, but two.

Step 5 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [2.0, 4.0, 6.0, 8.0]` `size = 4`

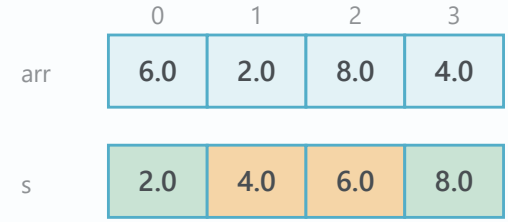
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {         // even
▶          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



The two middle indices: m1 = 1, m2 = 2.

Step 6 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [2.0, 4.0, 6.0, 8.0]` `size = 4`

`m1 = 1` `m2 = 2`

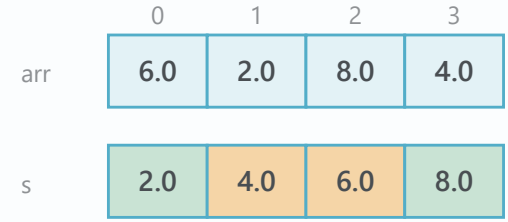
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
▶          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



median = (4.0 + 6.0) / 2 = **5.0**.

Step 7 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [2.0, 4.0, 6.0, 8.0]` `size = 4`

`m1 = 1` `m2 = 2` `median = 5.0`

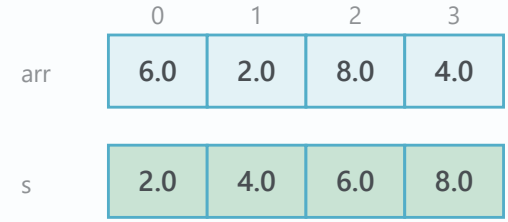
CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
▶      return median;
13 }
```



return 5.0.

Step 8 / 15

VARIABLES

`arr = [6.0, 2.0, 8.0, 4.0]`

`s = [2.0, 4.0, 6.0, 8.0]` `size = 4`

`m1 = 1` `m2 = 2` `median = 5.0`

CALL STACK

`median({6, 2, 8, 4})`

`main()`

Code Trace: `median()`

```
▶ static double median(double[] arr) {
2     double[] s = arr.clone(); // copy
3     Arrays.sort(s);              // O(n log n)
4     int size = s.length;
5     double median;
6     if (size % 2 == 0) {        // even
7         int m1 = size/2 - 1, m2 = size/2;
8         median = (s[m1] + s[m2]) / 2;
9     } else {                    // odd
10        median = s[size / 2];
11    }
12    return median;
13 }
```

	0	1	2	3	4
arr	9.0	3.0	11.0	7.0	5.0

Call 2 (odd size): `median` is called with a 5-element array.

Step 9 / 15

VARIABLES

`arr = [9.0, 3.0, 11.0, 7.0, 5.0]`

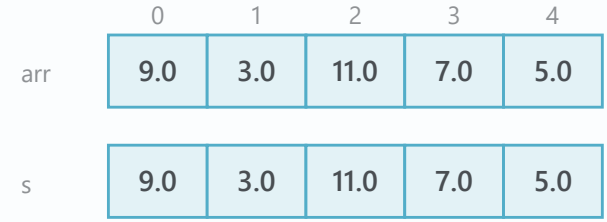
CALL STACK

`median({9, 3, 11, 7, 5})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
▶      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



Make a **copy** first, so the caller's array keeps its order.

Step 10 / 15

VARIABLES

`arr = [9.0, 3.0, 11.0, 7.0, 5.0]`

`s = [9.0, 3.0, 11.0, 7.0, 5.0]`

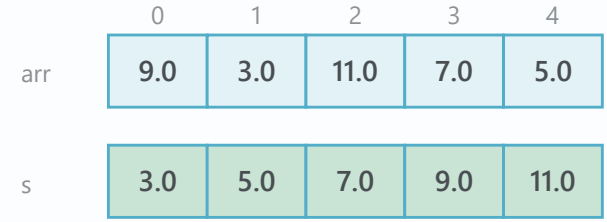
CALL STACK

`median({9, 3, 11, 7, 5})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
▶      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {       // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                   // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



The copy is sorted; the original `arr` is unchanged.

Step 11 / 15

VARIABLES

`arr = [9.0, 3.0, 11.0, 7.0, 5.0]`

`s = [3.0, 5.0, 7.0, 9.0, 11.0]`

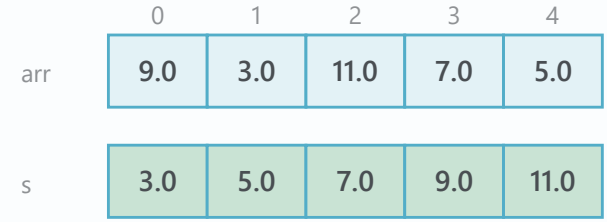
CALL STACK

`median({9, 3, 11, 7, 5})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
▶      int size = s.length;
5      double median;
6      if (size % 2 == 0) {        // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



size = 5.

Step 12 / 15

VARIABLES

`arr = [9.0, 3.0, 11.0, 7.0, 5.0]`

`s = [3.0, 5.0, 7.0, 9.0, 11.0]`

`size = 5`

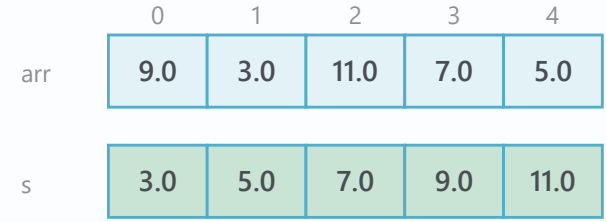
CALL STACK

`median({9, 3, 11, 7, 5})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
▶      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
12     return median;
13 }
```



5 % 2 = 1 → **odd**: take the else branch.

Step 13 / 15

VARIABLES

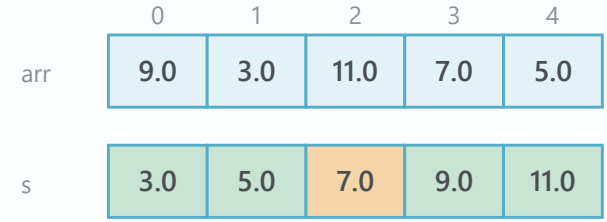
- `arr = [9.0, 3.0, 11.0, 7.0, 5.0]`
- `s = [3.0, 5.0, 7.0, 9.0, 11.0]`
- `size = 5`

CALL STACK

- `median({9, 3, 11, 7, 5})`
- `main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {         // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
▶          median = s[size / 2];
11     }
12     return median;
13 }
```



The exact middle: index 5 / 2 = 2 → **7.0**.

Step 14 / 15

VARIABLES

`arr = [9.0, 3.0, 11.0, 7.0, 5.0]`

`s = [3.0, 5.0, 7.0, 9.0, 11.0]`

`size = 5` `median = 7.0`

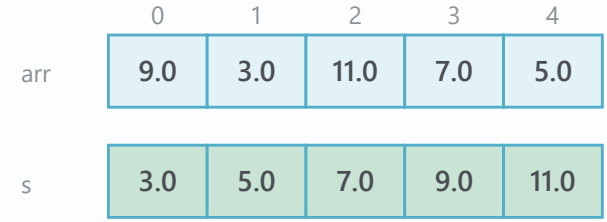
CALL STACK

`median({9, 3, 11, 7, 5})`

`main()`

Code Trace: `median()`

```
1  static double median(double[] arr) {
2      double[] s = arr.clone(); // copy
3      Arrays.sort(s);              // O(n log n)
4      int size = s.length;
5      double median;
6      if (size % 2 == 0) {        // even
7          int m1 = size/2 - 1, m2 = size/2;
8          median = (s[m1] + s[m2]) / 2;
9      } else {                     // odd
10         median = s[size / 2];
11     }
▶      return median;
13 }
```



return 7.0. Sorting dominates → O(n log n).

Step 15 / 15

VARIABLES

`arr = [9.0, 3.0, 11.0, 7.0, 5.0]`

`s = [3.0, 5.0, 7.0, 9.0, 11.0]`

`size = 5` `median = 7.0`

CALL STACK

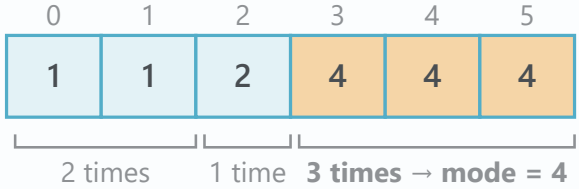
`median({9, 3, 11, 7, 5})`

`main()`

Mode (the Most Frequent Value)

```
static int mode(int[] arr) { // not empty
    int[] s = arr.clone();
    Arrays.sort(s);  // equal values side by side
    int best = s[0], bestCount = 1;
    int count = 1;
    for (int i = 1; i < s.length; i++) {
        if (s[i] == s[i - 1]) count++;
        else count = 1;   // new value
        if (count > bestCount) {
            bestCount = count;
            best = s[i];
        }
    }
    return best;
}
```

- After sorting, equal values are **adjacent**, so one pass finds the longest run.
- On a tie the first (smallest) value wins, because the comparison is `>`.
- Cost: sorting O(n log n) + scanning O(n).



Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
▶ static int mode(int[] arr) { // not empty
2     int[] s = arr.clone();
3     Arrays.sort(s);  // equal values side by side
4     int best = s[0], bestCount = 1;
5     int count = 1;
6     for (int i = 1; i < s.length; i++) {
7         if (s[i] == s[i - 1]) count++;
8         else count = 1;   // new value
9         if (count > bestCount) {
10            bestCount = count;
11            best = s[i];
12        }
13    }
14    return best;
15 }
```



Input: {4, 1, 4, 2, 1, 4}. Equal values are scattered.

Step 1 / 14

VARIABLES

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

CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
▶      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```

s

0	1	2	3	4	5
1	1	2	4	4	4

Once the copy is sorted, equal values sit **side by side**, and one pass is enough.

Step 2 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]`

CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
▶      int best = s[0], bestCount = 1;
▶      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```

	0	1	2	3	4	5
s	1	1	2	4	4	4

Start: the first value (1) is the most frequent so far, seen once.

Step 3 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `best = 1` `bestCount = 1` `count = 1`

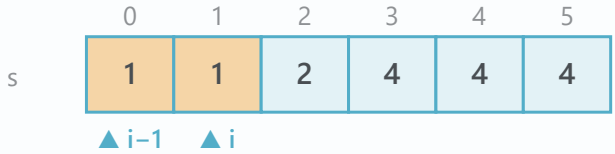
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
▶          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



s[1] = 1 == s[0] → the run continues, **count = 2**.

Step 4 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 1`

`best = 1` `bestCount = 1`

`count = 2`

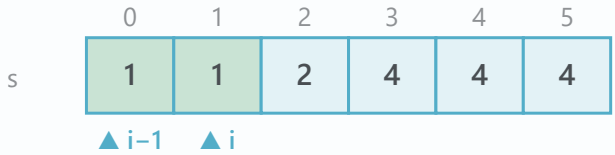
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
▶          if (count > bestCount) {
▶              bestCount = count;
▶              best = s[i];
12         }
13     }
14     return best;
15 }
```



2 > the old record → **new record**: best = 1, 2 times.

Step 5 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 1`

`best = 1` `bestCount = 2`

`count = 2`

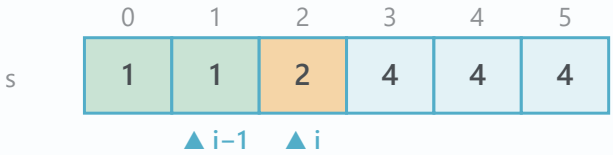
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
▶          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



s[2] = 2 ≠ 1 → **new value**; the count resets to 1.

Step 6 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 2`

`best = 1` `bestCount = 2`

`count = 1`

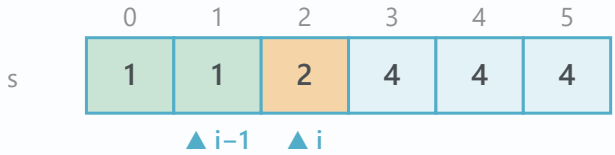
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
▶          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



1 > 2 is false → the record stands.

Step 7 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 2`

`best = 1` `bestCount = 2`

`count = 1`

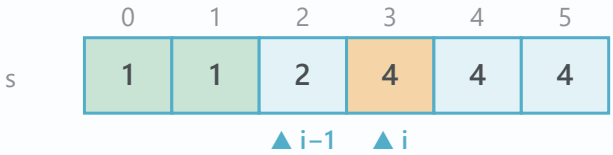
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
▶          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



s[3] = 4 ≠ 2 → **new value**; the count resets to 1.

Step 8 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 3`

`best = 1` `bestCount = 2`

`count = 1`

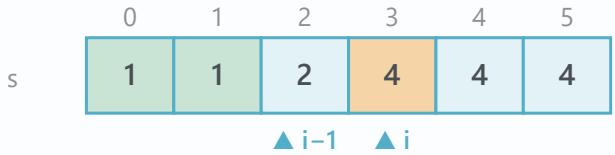
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
▶          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



1 > 2 is false → the record stands.

Step 9 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 3`

`best = 1` `bestCount = 2`

`count = 1`

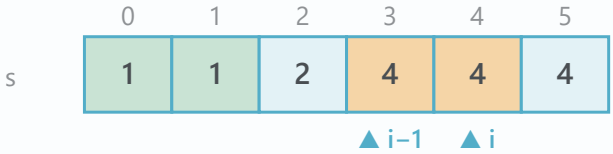
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
▶          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



s[4] = 4 == s[3] → the run continues, **count = 2**.

Step 10 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 4`

`best = 1` `bestCount = 2`

`count = 2`

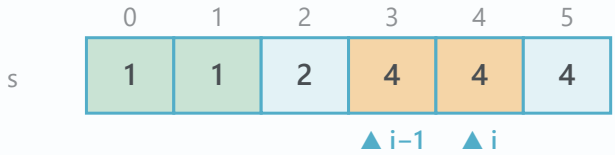
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
▶          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



2 > 2 is false → the record stands. (On a tie the first value wins.)

Step 11 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 4`

`best = 1` `bestCount = 2`

`count = 2`

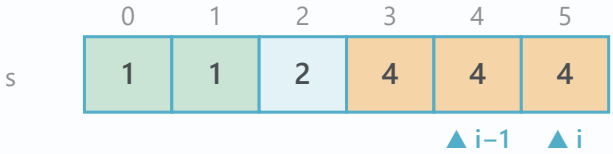
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
▶          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
14     return best;
15 }
```



s[5] = 4 == s[4] → the run continues, **count = 3**.

Step 12 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 5`

`best = 1` `bestCount = 2`

`count = 3`

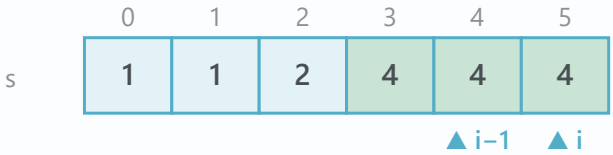
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
▶          if (count > bestCount) {
▶              bestCount = count;
▶              best = s[i];
12         }
13     }
14     return best;
15 }
```



3 > the old record → **new record**: best = 4, 3 times.

Step 13 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `i = 5`

`best = 4` `bestCount = 3`

`count = 3`

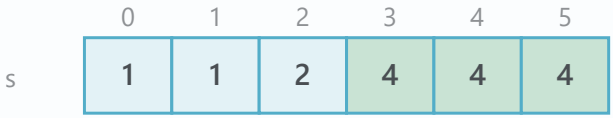
CALL STACK

`mode(arr)`

`main()`

Code Trace: `mode({4, 1, 4, 2, 1, 4})`

```
1  static int mode(int[] arr) { // not empty
2      int[] s = arr.clone();
3      Arrays.sort(s);  // equal values side by side
4      int best = s[0], bestCount = 1;
5      int count = 1;
6      for (int i = 1; i < s.length; i++) {
7          if (s[i] == s[i - 1]) count++;
8          else count = 1;   // new value
9          if (count > bestCount) {
10             bestCount = count;
11             best = s[i];
12         }
13     }
▶      return best;
15 }
```



return 4 (3 times). Sorting O(n log n) + scanning O(n).

Step 14 / 14

VARIABLES

`s = [1, 1, 2, 4, 4, 4]` `best = 4` `bestCount = 3` `count = 3`

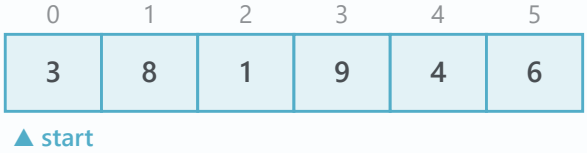
CALL STACK

`mode(arr)`

`main()`

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
▶      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = —

The left pointer starts at the first element.

Step 1 / 18

VARIABLES

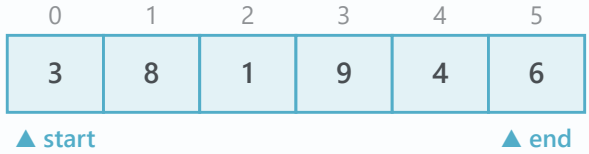
`arr = [3, 8, 1, 9, 4, 6]`

`start = 0`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
▶      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = —

The right pointer starts at the last element (length − 1 = 5).

Step 2 / 18

VARIABLES

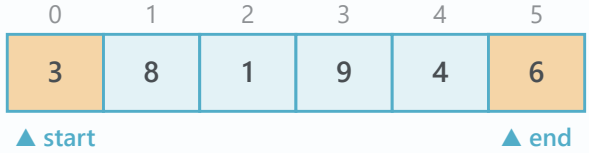
`arr = [3, 8, 1, 9, 4, 6]`

`start = 0` `end = 5`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
▶      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = —

0 < 5 → there is another pair to swap.

Step 3 / 18

VARIABLES

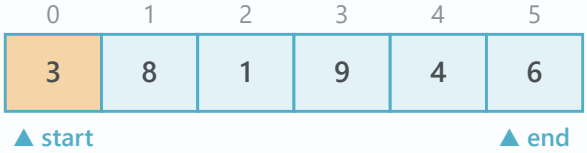
`arr = [3, 8, 1, 9, 4, 6]`

`start = 0` `end = 5`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 3

temp = arr[0] = 3: save the left value so it isn't lost.

Step 4 / 18

VARIABLES

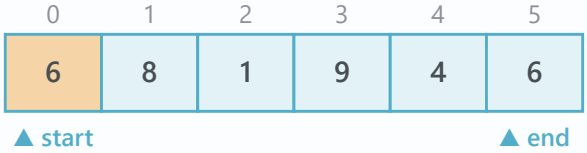
`arr = [3, 8, 1, 9, 4, 6]`

`start = 0` `end = 5` `temp = 3`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
▶          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 3

arr[0] = arr[5] → **6**. (The old 3 now survives only in temp.)

Step 5 / 18

VARIABLES

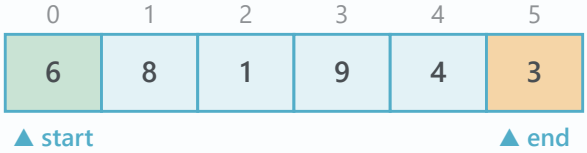
`arr = [6, 8, 1, 9, 4, 6]`

`start = 0` `end = 5` `temp = 3`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
▶          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 3

arr[5] = temp → **3**. The pair is swapped.

Step 6 / 18

VARIABLES

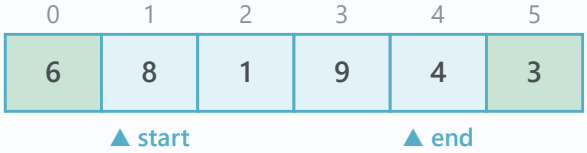
`arr = [6, 8, 1, 9, 4, 3]`

`start = 0` `end = 5` `temp = 3`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
▶          start++;   // move inward
▶          end--;
11     }
12 }
```



temp = 3

The pointers move inward: start = 1, end = 4.

Step 7 / 18

VARIABLES

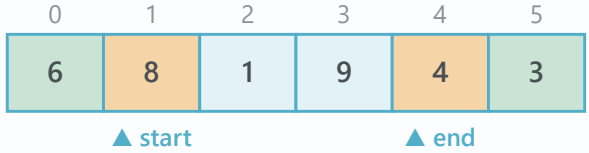
`arr = [6, 8, 1, 9, 4, 3]`

`start = 1` `end = 4` `temp = 3`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
▶      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 3

1 < 4 → there is another pair to swap.

Step 8 / 18

VARIABLES

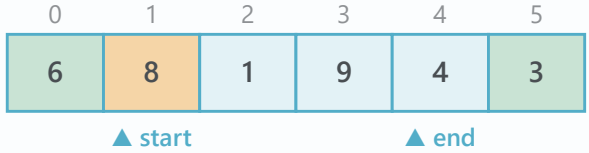
`arr = [6, 8, 1, 9, 4, 3]`

`start = 1` `end = 4` `temp = 3`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
           int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 8

temp = arr[1] = 8: save the left value so it isn't lost.

Step 9 / 18

VARIABLES

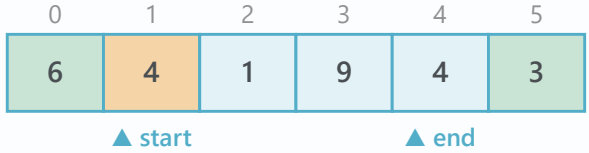
`arr = [6, 8, 1, 9, 4, 3]`

`start = 1` `end = 4` `temp = 8`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
▶          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 8

arr[1] = arr[4] → **4**. (The old 8 now survives only in temp.)

Step 10 / 18

VARIABLES

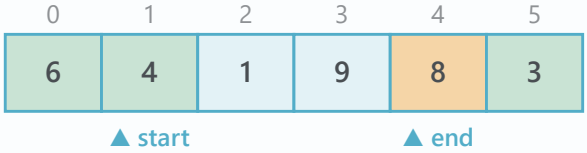
`arr = [6, 4, 1, 9, 4, 3]`

`start = 1` `end = 4` `temp = 8`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
▶          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 8

arr[4] = temp → **8**. The pair is swapped.

Step 11 / 18

VARIABLES

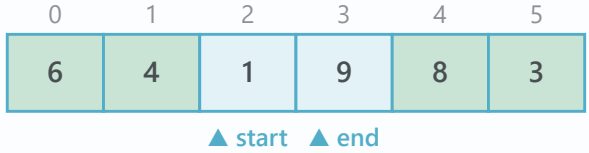
`arr = [6, 4, 1, 9, 8, 3]`

`start = 1` `end = 4` `temp = 8`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
▶          start++;   // move inward
▶          end--;
11     }
12 }
```



temp = 8

The pointers move inward: start = 2, end = 3.

Step 12 / 18

VARIABLES

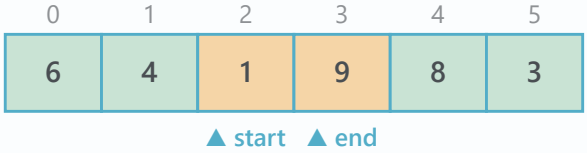
`arr = [6, 4, 1, 9, 8, 3]`

`start = 2` `end = 3` `temp = 8`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
▶      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 8

2 < 3 → there is another pair to swap.

Step 13 / 18

VARIABLES

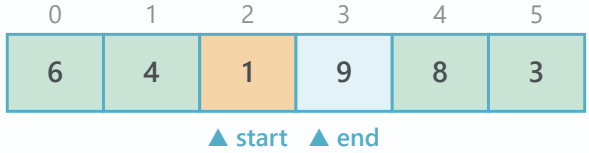
`arr = [6, 4, 1, 9, 8, 3]`

`start = 2` `end = 3` `temp = 8`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
           int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 1

temp = arr[2] = 1: save the left value so it isn't lost.

Step 14 / 18

VARIABLES

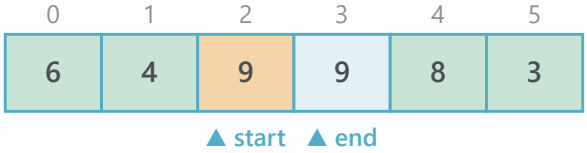
`arr = [6, 4, 1, 9, 8, 3]`

`start = 2` `end = 3` `temp = 1`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
▶          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 1

arr[2] = arr[3] → **9**. (The old 1 now survives only in temp.)

Step 15 / 18

VARIABLES

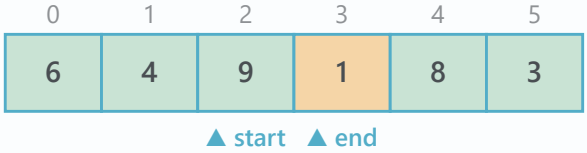
`arr = [6, 4, 9, 9, 8, 3]`

`start = 2` `end = 3` `temp = 1`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
▶          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 1

arr[3] = temp → **1**. The pair is swapped.

Step 16 / 18

VARIABLES

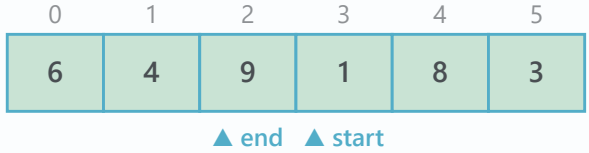
`arr = [6, 4, 9, 1, 8, 3]`

`start = 2` `end = 3` `temp = 1`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
4      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
▶          start++;   // move inward
▶          end--;
11     }
12 }
```



temp = 1

The pointers move inward: start = 3, end = 2.

Step 17 / 18

VARIABLES

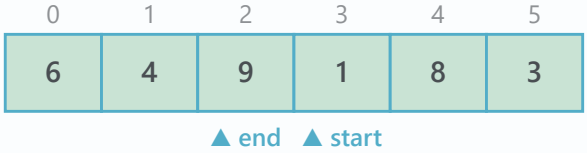
`arr = [6, 4, 9, 1, 8, 3]`

`start = 3` `end = 2` `temp = 1`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Reversing an Array in Place

```
1  static void reverse(int[] arr) {
2      int start = 0;
3      int end = arr.length - 1;
▶      while (start < end) {
5          // swap the two outermost elements
6          int temp = arr[start];
7          arr[start] = arr[end];
8          arr[end] = temp;
9          start++;   // move inward
10         end--;
11     }
12 }
```



temp = 1

3 < 2 is false → the pointers have crossed in the middle and the loop ends. ⌊n/2⌋ = 3 swaps → O(n) time, O(1) extra space.

Step 18 / 18

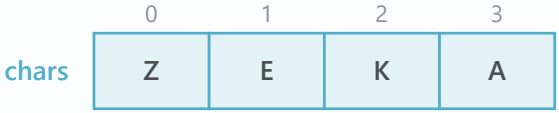
VARIABLES

`arr = [6, 4, 9, 1, 8, 3]`

`start = 3` `end = 2` `temp = 1`

Copying back to front into a new array (`result[i] = arr[n - 1 - i]`) is also O(n) time, but needs O(n) extra space. With an odd number of elements, the middle one stays put.

Application: Caesar Cipher `encrypt("ZEKA", 3)`



`toCharArray()` copies the string's letters into a new **char[]**. A `String` is immutable; the array can be changed in place.

Step 1 / 10

```
1 static String encrypt(String message, int k) {
▶     char[] chars = message.toCharArray();
3     for (int i = 0; i < chars.length; i++) {
4         int pos = chars[i] - 'A';        // 0..25 (A–Z only)
5         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

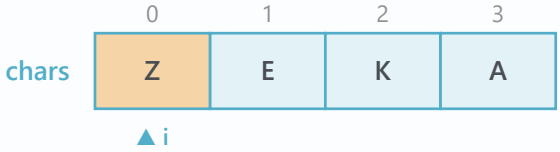
`chars = [Z, E, K, A]`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'Z' − 'A' = 25

`i = 0` : `'Z' − 'A'` gives the letter's position in the alphabet: **25** (char values are integers).

Step 2 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
▶     for (int i = 0; i < chars.length; i++) {
▶         int pos = chars[i] - 'A';        // 0..25 (A-Z only)
5         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [Z, E, K, A]` `i = 0`

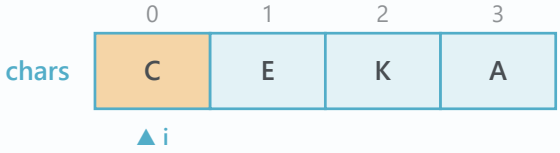
`pos = 25`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'Z' − 'A' = 25 → (25 + 3) % 26 = 2 → 'C'

`chars[0] = 'C'` : shift the position by 3 and turn it back into a letter. A position past the end wraps around with `% 26` (Z → C).

Step 3 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
3     for (int i = 0; i < chars.length; i++) {
4         int pos = chars[i] - 'A';        // 0..25 (A–Z only)
▶         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, E, K, A]` `i = 0`

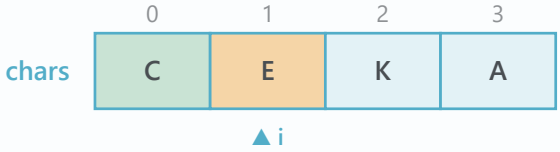
`pos = 25`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'E' − 'A' = 4

`i = 1` : `'E' - 'A'` gives the letter's position in the alphabet: **4** (char values are integers).

Step 4 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
▶     for (int i = 0; i < chars.length; i++) {
▶         int pos = chars[i] - 'A';        // 0..25 (A-Z only)
5         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, E, K, A]` `i = 1`

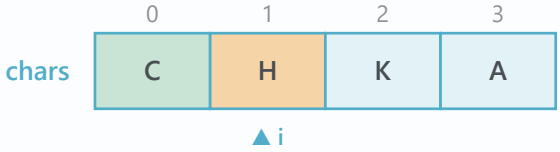
`pos = 4`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'E' − 'A' = 4 → (4 + 3) % 26 = 7 → 'H'

`chars[1] = 'H'` : shift the position by 3 and turn it back into a letter.

Step 5 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
3     for (int i = 0; i < chars.length; i++) {
4         int pos = chars[i] - 'A';        // 0..25 (A–Z only)
▶           chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

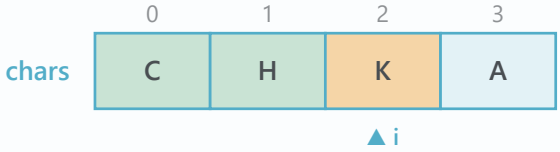
VARIABLES

`message = "ZEKA"` `k = 3`
`chars = [C, H, K, A]` `i = 1`
`pos = 4`

CALL STACK

`encrypt("ZEKA", 3)`
`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'K' − 'A' = 10

`i = 2` : `'K' - 'A'` gives the letter's position in the alphabet: **10** (char values are integers).

Step 6 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
▶     for (int i = 0; i < chars.length; i++) {
▶         int pos = chars[i] - 'A';        // 0..25 (A-Z only)
5         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, H, K, A]` `i = 2`

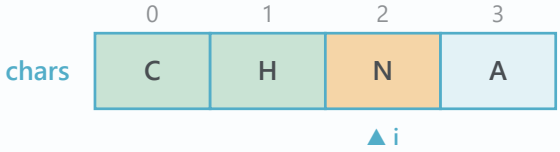
`pos = 10`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'K' − 'A' = 10 → (10 + 3) % 26 = 13 → 'N'

`chars[2] = 'N'` : shift the position by 3 and turn it back into a letter.

Step 7 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
3     for (int i = 0; i < chars.length; i++) {
4         int pos = chars[i] - 'A';        // 0..25 (A–Z only)
▶             chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, H, N, A]` `i = 2`

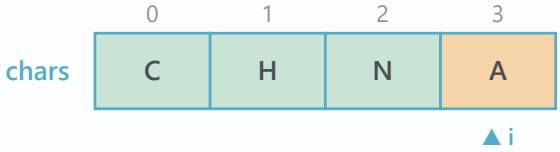
`pos = 10`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'A' − 'A' = 0

`i = 3` : `'A' - 'A'` gives the letter's position in the alphabet: **0** (char values are integers).

Step 8 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
▶     for (int i = 0; i < chars.length; i++) {
▶         int pos = chars[i] - 'A';        // 0..25 (A-Z only)
5         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, H, N, A]` `i = 3`

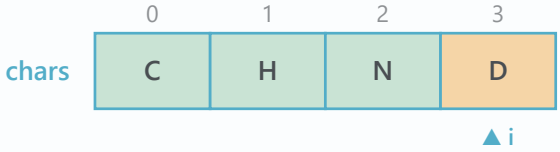
`pos = 0`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



pos = 'A' − 'A' = 0 → (0 + 3) % 26 = 3 → 'D'

`chars[3] = 'D'` : shift the position by 3 and turn it back into a letter.

Step 9 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
3     for (int i = 0; i < chars.length; i++) {
4         int pos = chars[i] - 'A';        // 0..25 (A–Z only)
▶             chars[i] = (char) ('A' + (pos + k) % 26);
6     }
7     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, H, N, D]` `i = 3`

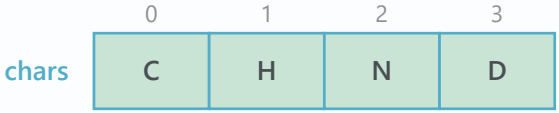
`pos = 0`

CALL STACK

`encrypt("ZEKA", 3)`

`main()`

Application: Caesar Cipher `encrypt("ZEKA", 3)`



`new String(chars)` → **"CHND"**. To decrypt, call the same method with `k = 26 − 3 = 23`. Each letter is handled once: **O(n)**.

Step 10 / 10

```
1 static String encrypt(String message, int k) {
2     char[] chars = message.toCharArray();
3     for (int i = 0; i < chars.length; i++) {
4         int pos = chars[i] - 'A';        // 0..25 (A–Z only)
5         chars[i] = (char) ('A' + (pos + k) % 26);
6     }
▶     return new String(chars);
8 }
```

VARIABLES

`message = "ZEKA"` `k = 3`

`chars = [C, H, N, D]`

CALL STACK

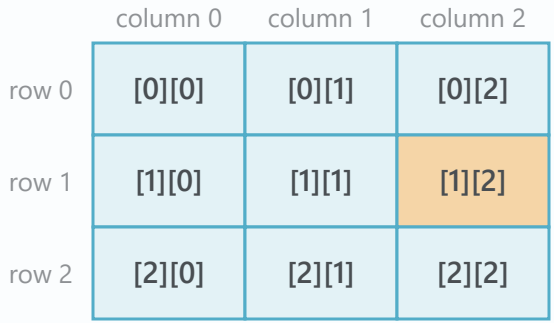
`encrypt("ZEKA", 3)`

`main()`

Two-Dimensional Arrays (Matrices)

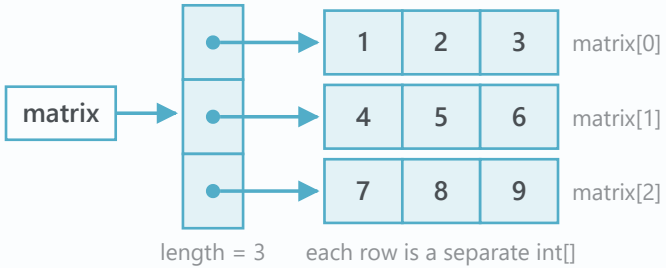
- A table-like structure of rows and columns.
- A natural fit for two-dimensional data: a chessboard, an image (a grid of pixels), a table.
- An element is reached with two indices: `matrix[row][column]`.

```
int[][] matrix = new int[3][3];
matrix[1][2] = 7;   // 2nd row, 3rd column
```



In Java, a Matrix Is an Array of Arrays

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
matrix.length      // 3 (number of rows)
matrix[0].length   // 3 (number of columns)
```



- The initializer gives no sizes: the number of rows comes from the outer list, the number of columns from the inner lists.
- Each row is a separate array, so rows can differ in length (a **jagged array**): `int[][] tri = new int[3][]; tri[0] = new int[1];` …

Jagged Array: Pascal's Triangle



`new int[n][]` creates only the outer array of 4 **references**; the row arrays don't exist yet (`null`).

Step 1 / 12

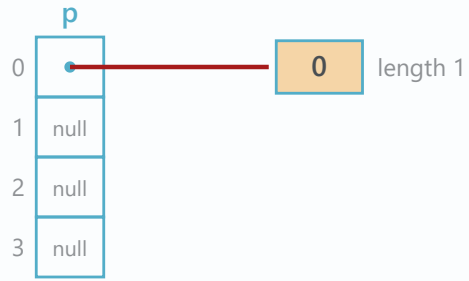
```
▶ int n = 4;
▶ int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4`

`p = [null, null, null, null]`

Jagged Array: Pascal's Triangle



`p[0] = new int[1]` : row 0 is a separate array of length **1**; its cells start at 0.

Step 2 / 12

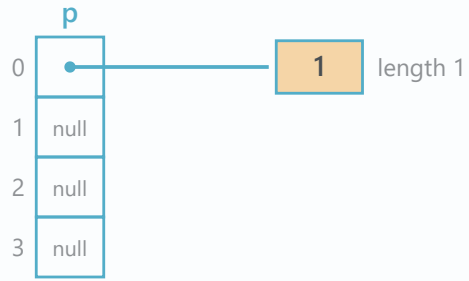
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
▶     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 0`

`p = [[0], null, null, null]`

Jagged Array: Pascal's Triangle



`p[0][0] = 1` : in a one-cell row, the first and last cells are the same.

Step 3 / 12

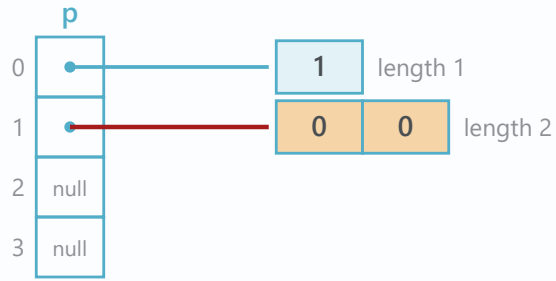
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
▶     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 0`

`p = [[1], null, null, null]`

Jagged Array: Pascal's Triangle



`p[1] = new int[2]` : row 1 is a separate array of length **2**; rows of different lengths make a jagged array.

Step 4 / 12

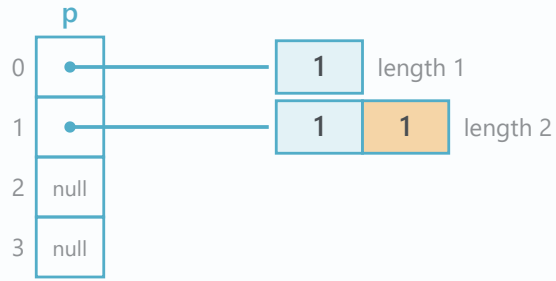
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
▶     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 1`

`p = [[1], [0, 0], null, null]`

Jagged Array: Pascal's Triangle



`p[1][0] = p[1][1] = 1` : both ends of the row are 1 (a chained assignment runs right to left).

Step 5 / 12

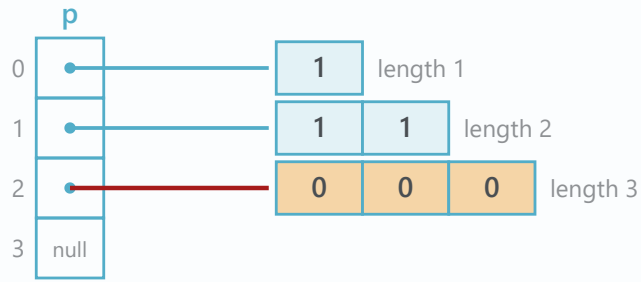
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
▶     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 1`

`p = [[1], [1, 1], null, null]`

Jagged Array: Pascal's Triangle



`p[2] = new int[3]` : a new row of 3 cells.

Step 6 / 12

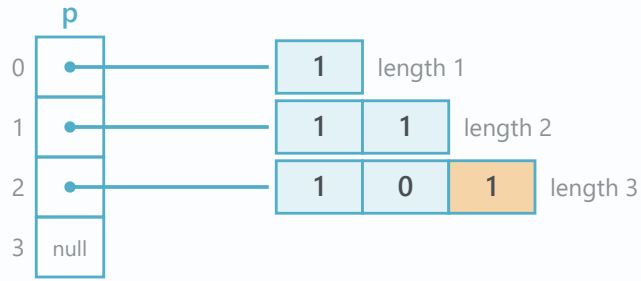
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
▶     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 2`

`p = [[1], [1, 1], [0, 0, 0], null]`

Jagged Array: Pascal's Triangle



`p[2][0] = p[2][2] = 1` : both ends of the row are 1 (a chained assignment runs right to left).

Step 7 / 12

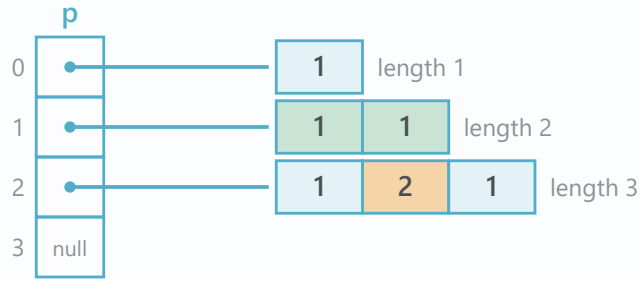
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
▶     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 2`

`p = [[1], [1, 1], [1, 0, 1], null]`

Jagged Array: Pascal's Triangle



`p[2][1] = p[1][0] + p[1][1]` = 1 + 1 = **2**: the sum of the two neighbors above.

Step 8 / 12

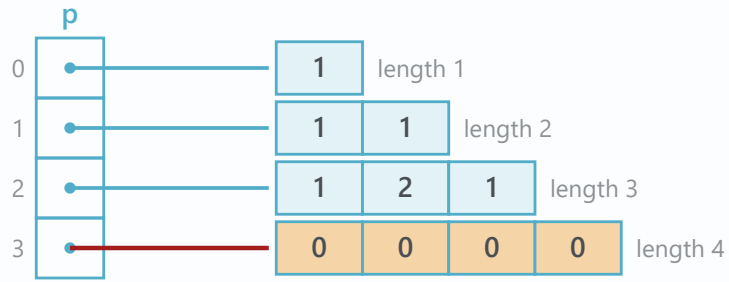
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
▶         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 2` `j = 1`

`p = [[1], [1, 1], [1, 2, 1], null]`

Jagged Array: Pascal's Triangle



`p[3] = new int[4]` : a new row of 4 cells.

Step 9 / 12

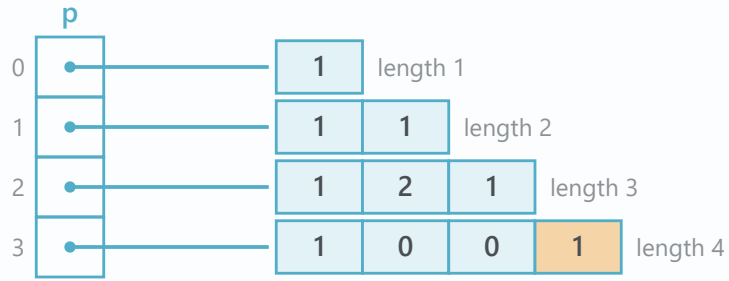
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
▶     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 3`

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

Jagged Array: Pascal's Triangle



`p[3][0] = p[3][3] = 1` : both ends of the row are 1 (a chained assignment runs right to left).

Step 10 / 12

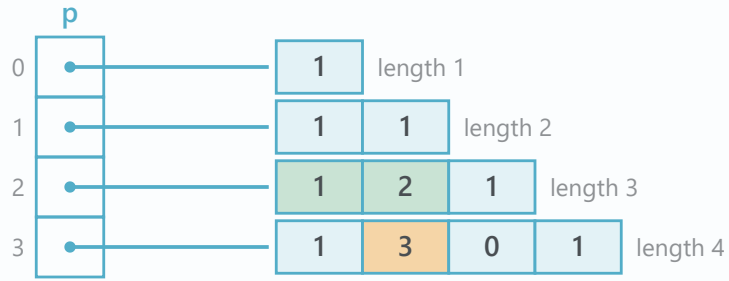
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
▶     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
7         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 3`

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

Jagged Array: Pascal's Triangle



`p[3][1] = p[2][0] + p[2][1]` = 1 + 2 = **3**: the sum of the two neighbors above.

Step 11 / 12

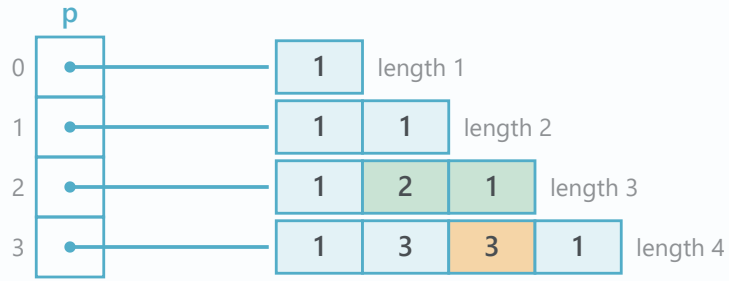
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
          p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 3` `j = 1`

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

Jagged Array: Pascal's Triangle



`p[3][2] = p[2][1] + p[2][2]` = 2 + 1 = **3**: the sum of the two neighbors above. Pascal's triangle is complete: 1+2+3+4 = 10 cells in all (not 4×4 = 16).

Step 12 / 12

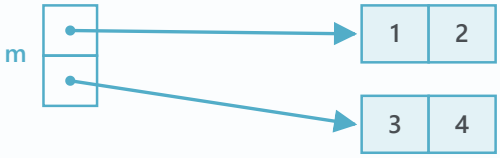
```
1 int n = 4;
2 int[][] p = new int[n][];          // no rows yet
3 for (int i = 0; i < n; i++) {
4     p[i] = new int[i + 1];         // length i + 1
5     p[i][0] = p[i][i] = 1;
6     for (int j = 1; j < i; j++)
▶         p[i][j] = p[i-1][j-1] + p[i-1][j];
8 }
```

VARIABLES

`n = 4` `i = 3` `j = 2`

`p = [[1], [1, 1], [1, 2, 1], [1, 3, 3, 1]]`

Shallow and Deep Copies of a 2D Array



A 2×2 matrix: the outer array `m` points to two row arrays (an array of arrays).

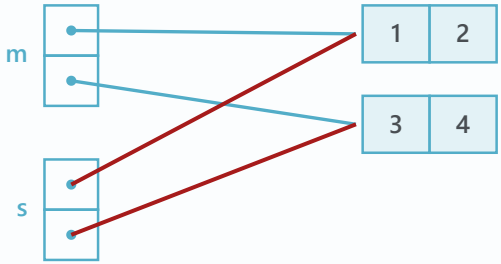
Step 1 / 8

```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
6 for (int i = 0; i < 2; i++)
7     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

VARIABLES

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

Shallow and Deep Copies of a 2D Array



`m.clone()` copies only the **outer** array: `s` is a new outer array, but its references point to the **same** rows → a **shallow copy**.

Step 2 / 8

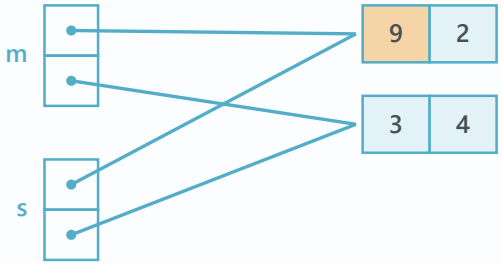
```
1 int[][] m = { {1, 2}, {3, 4} };
▶ int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
6 for (int i = 0; i < 2; i++)
7     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

VARIABLES

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

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

Shallow and Deep Copies of a 2D Array



`s[0][0] = 9` changes the shared row, so `m[0][0]` becomes **9** too. A common mistake!

Step 3 / 8

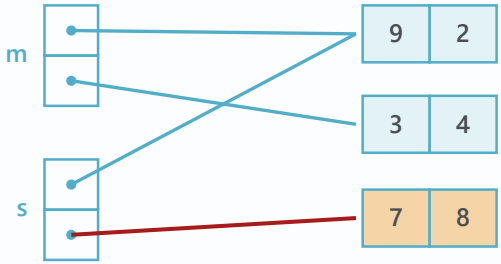
```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
▶ s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
6 for (int i = 0; i < 2; i++)
7     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

VARIABLES

`m = [[9, 2], [3, 4]]`

`s = [[9, 2], [3, 4]]`

Shallow and Deep Copies of a 2D Array



`s[1] = new int[] {7, 8}`, on the other hand, changes a reference in `s`'s own outer array; `m[1]` is still {3, 4}.

Step 4 / 8

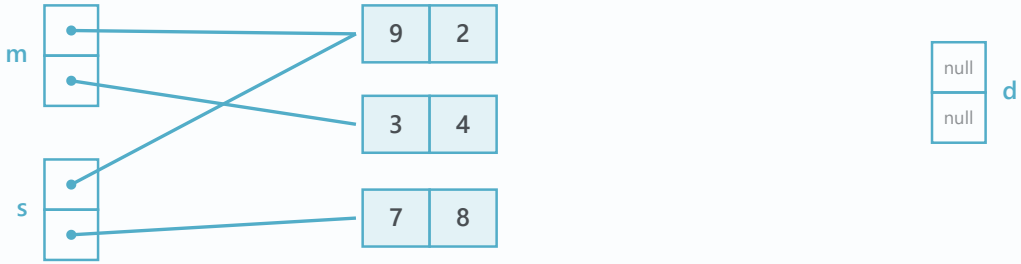
```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
▶ s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
6 for (int i = 0; i < 2; i++)
7     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

VARIABLES

`m = [[9, 2], [3, 4]]`

`s = [[9, 2], [7, 8]]`

Shallow and Deep Copies of a 2D Array



A deep copy starts with a new outer array `d`; it has no rows yet.

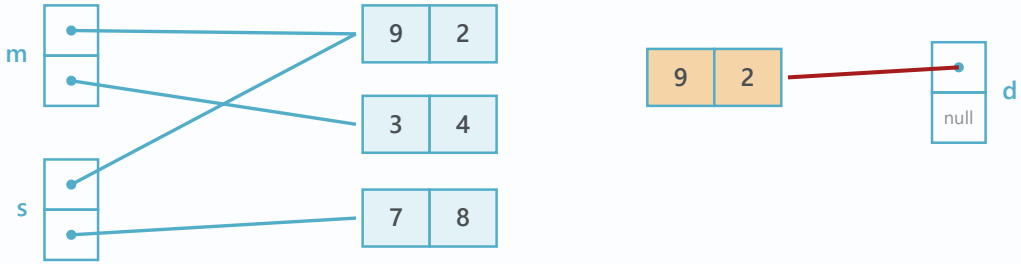
Step 5 / 8

```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
▶ int[][] d = new int[2][];       // deep copy:
6 for (int i = 0; i < 2; i++)
7     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

VARIABLES

- `m = [[9, 2], [3, 4]]`
- `s = [[9, 2], [7, 8]]`
- `d = [null, null]`

Shallow and Deep Copies of a 2D Array



`d[0] = m[0].clone()` : the **elements** of row 0 are copied into a new array.

Step 6 / 8

```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
▶ for (int i = 0; i < 2; i++)
▶     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

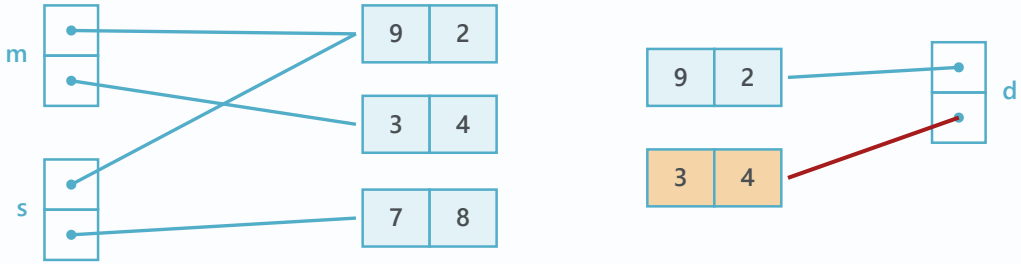
VARIABLES

`m = [[9, 2], [3, 4]]`

`s = [[9, 2], [7, 8]]`

`d = [[9, 2], null]` `i = 0`

Shallow and Deep Copies of a 2D Array



`d[1] = m[1].clone()` : the **elements** of row 1 are copied into a new array.

Step 7 / 8

```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
▶ for (int i = 0; i < 2; i++)
▶     d[i] = m[i].clone();        // rows too
8 d[0][1] = 5;                    // m unaffected
```

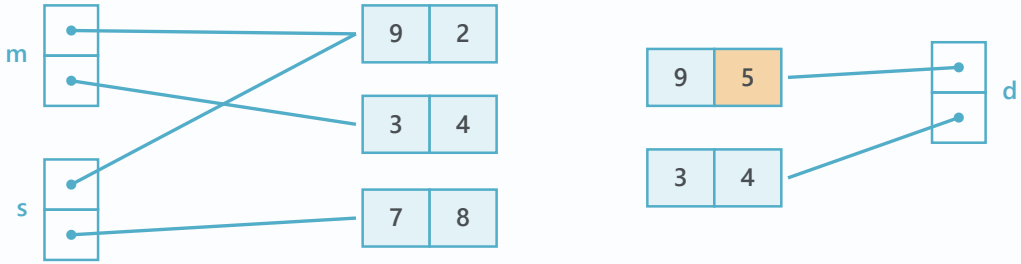
VARIABLES

`m = [[9, 2], [3, 4]]`

`s = [[9, 2], [7, 8]]`

`d = [[9, 2], [3, 4]]` `i = 1`

Shallow and Deep Copies of a 2D Array



`d[0][1] = 5` changes only the copy; `m[0][1]` is still 2. A deep copy takes O(rows × columns).

Step 8 / 8

```
1 int[][] m = { {1, 2}, {3, 4} };
2 int[][] s = m.clone();          // shallow copy
3 s[0][0] = 9;                    // m[0][0] is 9 too!
4 s[1] = new int[] {7, 8};        // only s changes
5 int[][] d = new int[2][];       // deep copy:
6 for (int i = 0; i < 2; i++)
7     d[i] = m[i].clone();        // rows too
▶ d[0][1] = 5;                    // m unaffected
```

VARIABLES

- `m = [[9, 2], [3, 4]]`
- `s = [[9, 2], [7, 8]]`
- `d = [[9, 5], [3, 4]]`

Example: Fill and Print a Matrix

```
public class MultiArrayExample {
  public static void main(String[] a) {
    int[][] matrix = new int[3][3];
    // initial values: 1…9
    for (int i = 0; i < 3; i++)      // row
      for (int j = 0; j < 3; j++)    // column
        matrix[i][j] = i * 3 + j + 1;

    System.out.println("Matrix contents:");
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        System.out.print(matrix[i][j] + " ");
      System.out.println();   // new line
    }
  }
}
```

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

Nested loops

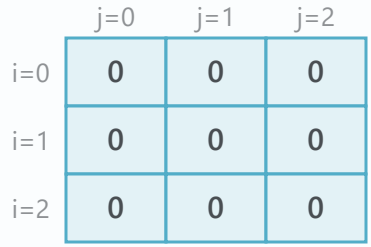
The outer loop walks the rows, the inner loop the columns. 3 × 3 = 9 assignments.

`i * 3 + j + 1`

Writing `matrix.length` and `matrix[i].length` instead of the constant 3 makes the code work for any size.

Code Trace: Filling the Matrix

```
▶ int[][] matrix = new int[3][3];
2 for (int i = 0; i < 3; i++)       // row
3   for (int j = 0; j < 3; j++)     // column
4     matrix[i][j] = i * 3 + j + 1;
5 System.out.println("Matrix contents:");
6 for (int i = 0; i < 3; i++) {
7   for (int j = 0; j < 3; j++)
8     System.out.print(matrix[i][j] + " ");
9   System.out.println();   // new line
10 }
```



An outer array of 3 rows plus 3 separate row arrays are created; every cell is 0.

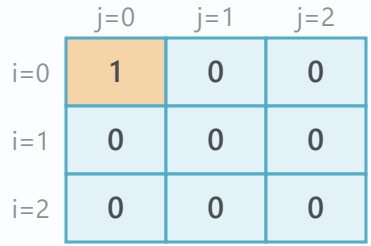
Step 1 / 11

VARIABLES

`matrix = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]`

Code Trace: Filling the Matrix

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



i = 0, j = 0 → **matrix[0][0] = 0·3 + 0 + 1 = 1**. The outer loop picks the row, the inner loop the column.

Step 2 / 11

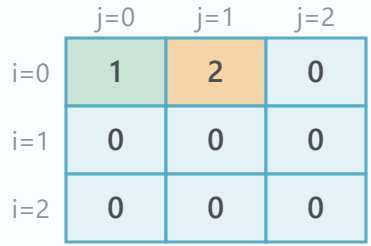
VARIABLES

`matrix = [[1, 0, 0], [0, 0, 0], [0, 0, 0]]`

`i = 0` `j = 0`

Code Trace: Filling the Matrix

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



i = 0, j = 1 → **matrix[0][1] = 0·3 + 1 + 1 = 2**.

Step 3 / 11

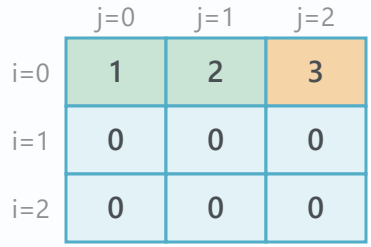
VARIABLES

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

`i = 0` `j = 1`

Code Trace: Filling the Matrix

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



i = 0, j = 2 → **matrix[0][2] = 0·3 + 2 + 1 = 3**.

Step 4 / 11

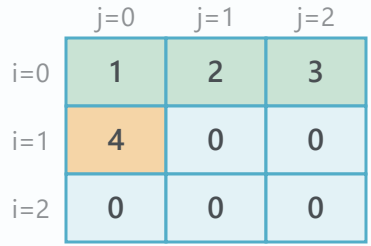
VARIABLES

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

`i = 0` `j = 2`

Code Trace: Filling the Matrix

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



i = 1, j = 0 → **matrix[1][0] = 1·3 + 0 + 1 = 4**. The inner loop finished and the outer loop moved to the next row; j is 0 again.

Step 5 / 11

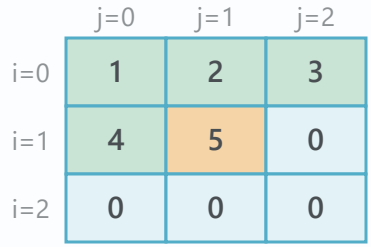
VARIABLES

`matrix = [[1, 2, 3], [4, 0, 0], [0, 0, 0]]`

`i = 1` `j = 0`

Code Trace: Filling the Matrix

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



i = 1, j = 1 → **matrix[1][1] = 1·3 + 1 + 1 = 5**.

Step 6 / 11

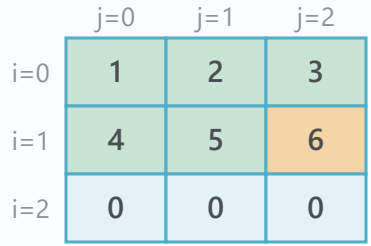
VARIABLES

`matrix = [[1, 2, 3], [4, 5, 0], [0, 0, 0]]`

`i = 1` `j = 1`

Code Trace: Filling the Matrix

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



i = 1, j = 2 → **matrix[1][2] = 1·3 + 2 + 1 = 6**.

Step 7 / 11

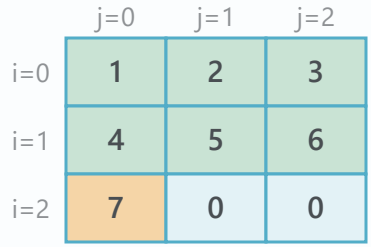
VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [0, 0, 0]]

i = 1 j = 2

Code Trace: Filling the Matrix

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



i = 2, j = 0 → **matrix[2][0] = 2·3 + 0 + 1 = 7**. The inner loop finished and the outer loop moved to the next row; j is 0 again.

Step 8 / 11

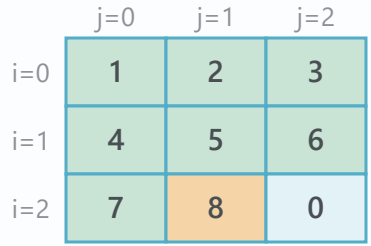
VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6], [7, 0, 0]]`

`i = 2` `j = 0`

Code Trace: Filling the Matrix

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



i = 2, j = 1 → **matrix[2][1] = 2·3 + 1 + 1 = 8**.

Step 9 / 11

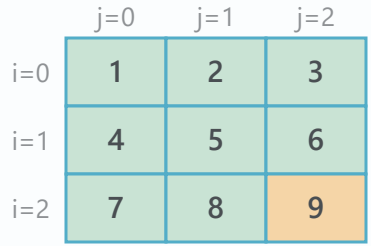
VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

i = 2 j = 1

Code Trace: Filling the Matrix

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



i = 2, j = 2 → **matrix[2][2] = 2·3 + 2 + 1 = 9**.

Step 10 / 11

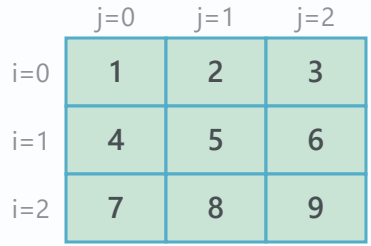
VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 2 j = 2

Code Trace: Filling the Matrix

```
1  int[][] matrix = new int[3][3];
▶ for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
8      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```



i = 3 → the outer loop ends. The inner body ran 3 × 3 = **9 times**.

Step 11 / 11

VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 3`

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
▶  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
8      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

The heading is printed; the matrix is now full.

Step 1 / 13

VARIABLES

matrix	[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

OUTPUT

```
Matrix contents:
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

Row 0: j starts at 0 → 1.

Step 2 / 13

VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 0 j = 0

OUTPUT

```
Matrix contents:
1
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

matrix[0][1] = 2 goes on the same line (`print`).

Step 3 / 13

VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 0 j = 1

OUTPUT

```
Matrix contents:
1 2
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

matrix[0][2] = 3 goes on the same line (`print`).

Step 4 / 13

VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 0 j = 2

OUTPUT

```
Matrix contents:
1 2 3
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
8      System.out.print(matrix[i][j] + " ");
▶    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

Row 0 done → `println()` moves to the next line.

Step 5 / 13

VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 0`

OUTPUT

```
Matrix contents:
1 2 3
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

Row 1: j starts at 0 → 4.

Step 6 / 13

VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 1 j = 0

OUTPUT

```
Matrix contents:
1 2 3
4
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

matrix[1][1] = 5 goes on the same line (`print`).

Step 7 / 13

VARIABLES

matrix = `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

i = 1 j = 1

OUTPUT

```
Matrix contents:
1 2 3
4 5
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

matrix[1][2] = 6 goes on the same line (`print`).

Step 8 / 13

VARIABLES

`matrix` = `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 1` `j = 2`

OUTPUT

```
Matrix contents:
1 2 3
4 5 6
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
8      System.out.print(matrix[i][j] + " ");
▶    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

Row 1 done → `println()` moves to the next line.

Step 9 / 13

VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 1`

OUTPUT

```
Matrix contents:
1 2 3
4 5 6
```

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

Row 2: j starts at 0 → 7.

Step 10 / 13

VARIABLES

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

i = 2 j = 0

OUTPUT

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

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

matrix[2][1] = 8 goes on the same line (`print`).

Step 11 / 13

VARIABLES

`matrix` = `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 2` `j = 1`

OUTPUT

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

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
▶      System.out.print(matrix[i][j] + " ");
9    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

matrix[2][2] = 9 goes on the same line (`print`).

Step 12 / 13

VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 2` `j = 2`

OUTPUT

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

Code Trace: Printing the Matrix (continued)

```
1  int[][] matrix = new int[3][3];
2  for (int i = 0; i < 3; i++)       // row
3    for (int j = 0; j < 3; j++)     // column
4      matrix[i][j] = i * 3 + j + 1;
5  System.out.println("Matrix contents:");
6  for (int i = 0; i < 3; i++) {
7    for (int j = 0; j < 3; j++)
8      System.out.print(matrix[i][j] + " ");
▶    System.out.println();   // new line
10 }
```

	j=0	j=1	j=2
i=0	1	2	3
i=1	4	5	6
i=2	7	8	9

Row 2 done → `println()` moves to the next line. The whole matrix is printed.

Step 13 / 13

VARIABLES

`matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`

`i = 2`

OUTPUT

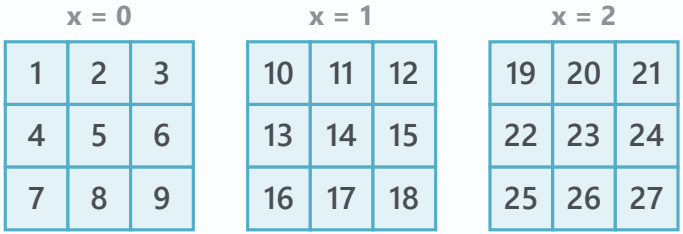
```
Matrix contents:
1 2 3
4 5 6
7 8 9
```

Three-Dimensional Arrays

- Java allows arrays of three or more dimensions (e.g. time × row × column, or RGB layers).
- Each dimension means one more nested loop: a 3 × 3 × 3 array has 27 elements.

```
int[][][] array3D =
        new int[3][3][3];
for (int x = 0; x < 3; x++)
  for (int y = 0; y < 3; y++)
    for (int z = 0; z < 3; z++)
      array3D[x][y][z] =
          x * 9 + y * 3 + z + 1;

// printing: the same three loops;
// println after the z loop (y),
// println after the y loop (x)
```



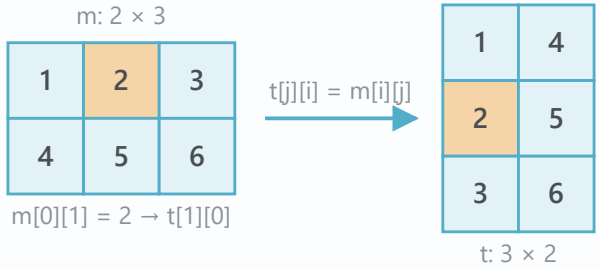
each layer is a 3×3 matrix · row y, column z

Non-ASCII letters in a variable name (`größe`, say) are legal in Java, but ASCII names are preferred for portability.

Matrix Transpose

Rows become columns and columns become rows: the transpose of an m×n matrix is n×m.

```
static int[][] transpose(int[][] m) {
    int rows = m.length;
    int cols = m[0].length;
    int[][] t = new int[cols][rows];
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            t[j][i] = m[i][j];
    return t;
}
static void printMatrix(int[][] m) {
    for (int[] row : m) {
        for (int x : row)
            System.out.print(x + " ");
        System.out.println();
    }
}
```



Cost: each element is copied once → O(m·n).

The helpers must be `static`: `main` is static, and without creating an object it can call only static methods.

Code Trace: `transpose(m)`

m: 2 × 3

1	2	3
4	5	6

m: 2 rows, 3 columns.

Step 1 / 11

```
▶ static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`

m: 2 × 3

1	2	3
4	5	6

`m.length` = the number of **rows** (the length of the outer array).

Step 2 / 11

```
1 static int[][] transpose(int[][] m) {
▶     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`

m: 2 × 3

1	2	3
4	5	6

`m[0].length` = the length of the first row = the number of **columns**.

Step 3 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
▶     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`

m: 2 × 3

1	2	3
4	5	6

t: 3 × 2

0	0
0	0
0	0

The dimensions swap: a new **3 × 2** matrix (all 0).

Step 4 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3`

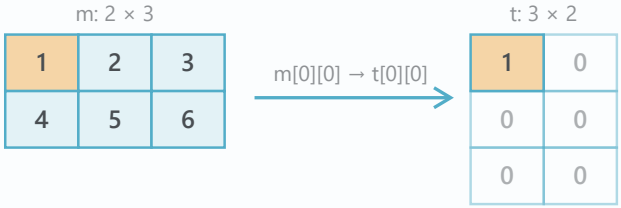
`t = [[0, 0], [0, 0], [0, 0]]`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`



t[0][0] = m[0][0] = 1: row ↔ column.

Step 5 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

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

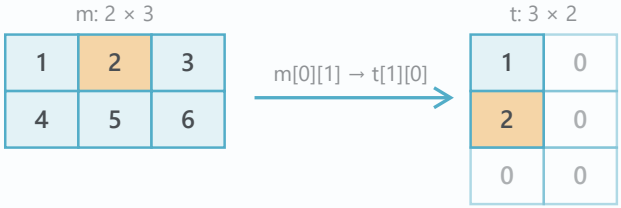
`t = [[1, 0], [0, 0], [0, 0]]`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`



t[1][0] = m[0][1] = 2: row ↔ column.

Step 6 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3` `i = 0` `j = 1`

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

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`



t[2][0] = m[0][2] = 3: row ↔ column. Row 0 of m becomes column 0 of t.

Step 7 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

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

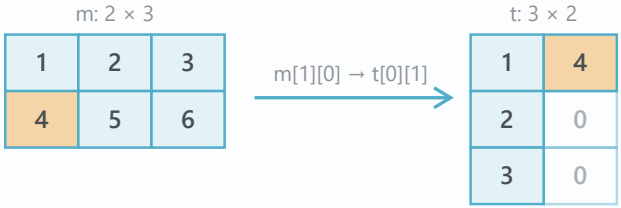
`t = [[1, 0], [2, 0], [3, 0]]`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`



t[0][1] = m[1][0] = 4: row ↔ column.

Step 8 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3` `i = 1` `j = 0`

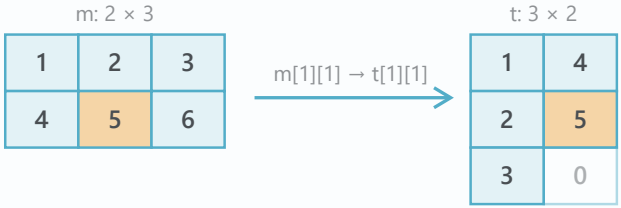
`t = [[1, 4], [2, 0], [3, 0]]`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`



t[1][1] = m[1][1] = 5: row ↔ column.

Step 9 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3` `i = 1` `j = 1`

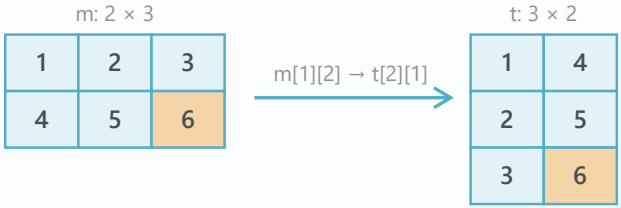
`t = [[1, 4], [2, 5], [3, 0]]`

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`



t[2][1] = m[1][2] = 6: row ↔ column.

Step 10 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
8     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3` `i = 1` `j = 2`

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

CALL STACK

`transpose(m)`

`main()`

Code Trace: `transpose(m)`

m: 2 × 3

1	2	3
4	5	6

t: 3 × 2

1	4
2	5
3	6

return t. 2 × 3 = 6 copies → O(m·n).

Step 11 / 11

```
1 static int[][] transpose(int[][] m) {
2     int rows = m.length;
3     int cols = m[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] = m[i][j];
▶     return t;
9 }
```

VARIABLES

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

`rows = 2` `cols = 3`

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

CALL STACK

`transpose(m)`

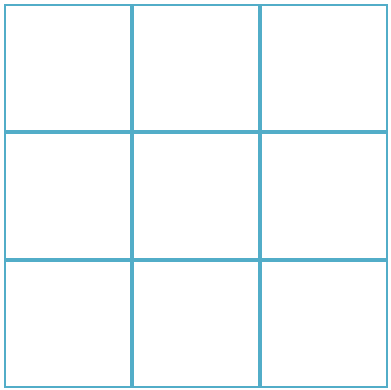
`main()`

Application: Tic Tac Toe

- A 3×3 game board: `char[][] board`.
- Two players, X and O, take turns.
- Three of your marks in a **row**, **column**, or **diagonal** wins.
- If the board fills up first, the game is a draw.

Every move is a single array assignment:

```
board[row][col] = player
```



X to move. Click a cell.

new game

Tic Tac Toe: The Main Loop

```
public class TicTacToe {   // import java.util.Scanner;
    static Scanner scanner = new Scanner(System.in); // single Scanner

    public static void main(String[] args) {
        char[][] board = new char[3][3];   // 3x3 game board
        char player = 'X';                   // X moves first
        boolean gameOn = true;
        initBoard(board);                  // every cell ' '
        while (gameOn) {
            showBoard(board);
            makeMove(board, player);
            gameOn = isGameOn(board, player);
            player = (player == 'X') ? 'O' : 'X';   // switch turns
        }
        showBoard(board);
    }
```

The cells of `new char[3][3]` start as `'\u0000'`; to show empty cells as `' '`, the board is filled first. All helper methods are `static`.

Tic Tac Toe: Board and Moves

```
static void initBoard(char[][] t) {
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      t[i][j] = ' ';
}
static void showBoard(char[][] t) {
  System.out.println("-------------");
  for (int i = 0; i < 3; i++) {
    System.out.print("| ");
    for (int j = 0; j < 3; j++)
      System.out.print(t[i][j]+" | ");
    System.out.println("\n-------------");
  }
}
```

```
static void makeMove(char[][] t,
                     char player) {
  int row, col;
  do {
    System.out.print("Player " + player
      + ": row and column (1-3): ");
    row = scanner.nextInt() - 1;
    col = scanner.nextInt() - 1;
  } while (row < 0 || row > 2
        || col < 0 || col > 2
        || t[row][col] != ' ');
  t[row][col] = player;
}
// the user enters 1-3, the array uses 0-2;
// invalid/occupied cell → ask again
```

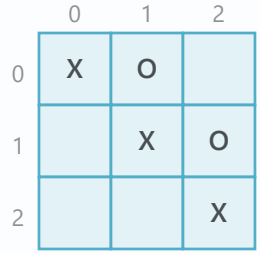
Tic Tac Toe: Win or Draw

The 8 winning lines (3 rows + 3 columns + 2 diagonals) are checked with one loop (rows and columns) and a single return (diagonals).

```
static boolean hasWon(char[][] t, char o) {
    for (int i = 0; i < 3; i++)
        if ((t[i][0] == o && t[i][1] == o && t[i][2] == o)     // row i
         || (t[0][i] == o && t[1][i] == o && t[2][i] == o))    // column i
            return true;
    return (t[0][0] == o && t[1][1] == o && t[2][2] == o)      // main diagonal
        || (t[0][2] == o && t[1][1] == o && t[2][0] == o);     // anti-diagonal
}
static boolean isGameOn(char[][] t, char o) {
    if (hasWon(t, o)) {
        System.out.println("Congratulations, player " + o + " wins!");
        return false;
    }
    for (char[] row : t)
        for (char c : row)
            if (c == ' ') return true;          // an empty cell → continue
    System.out.println("The game is a draw.");
    return false;
}
```

Code Trace: `hasWon(t, 'X')`

```
▶ static boolean hasWon(char[][] t, char o) {
2   for (int i = 0; i < 3; i++)
3     if ((t[i][0] == o && t[i][1] == o
4          && t[i][2] == o)   // row i
5      || (t[0][i] == o && t[1][i] == o
6          && t[2][i] == o))  // column i
7       return true;
8   return (t[0][0] == o && t[1][1] == o
9          && t[2][2] == o)   // main diagonal
10      || (t[0][2] == o && t[1][1] == o
11         && t[2][0] == o);  // anti-diagonal
12 }
```



X made the last move. Has X won? Eight lines to check.

Step 1 / 9

VARIABLES

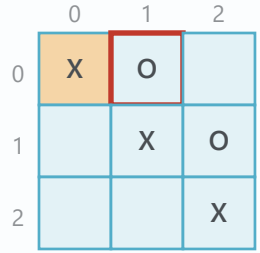
`o = 'X'`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
▶      if ((t[i][0] == o && t[i][1] == o
▶           && t[i][2] == o)   // row i
5       || (t[0][i] == o && t[1][i] == o
6           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 0, **row 0**: t[0][1] = 'O' ≠ 'X' → false. `&&` short-circuits, so the last cell is never checked.

Step 2 / 9

VARIABLES

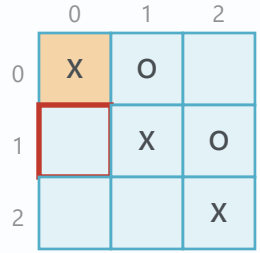
`o = 'X'` `i = 0`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
3      if ((t[i][0] == o && t[i][1] == o
4           && t[i][2] == o)   // row i
▶       || (t[0][i] == o && t[1][i] == o
▶           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 0, **column 0**: t[1][0] = empty ≠ 'X' → false. `&&` short-circuits, so the last cell is never checked.

Step 3 / 9

VARIABLES

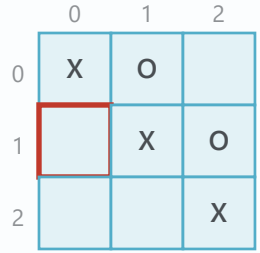
`o = 'X'` `i = 0`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
▶      if ((t[i][0] == o && t[i][1] == o
▶           && t[i][2] == o)   // row i
5       || (t[0][i] == o && t[1][i] == o
6           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 1, **row 1**: t[1][0] = empty ≠ 'X' → false. `&&` short-circuits, so the other 2 cells are never checked.

Step 4 / 9

VARIABLES

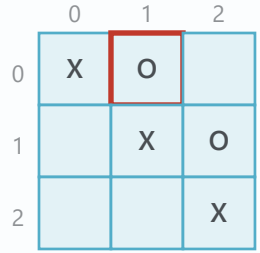
`o = 'X'` `i = 1`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
3      if ((t[i][0] == o && t[i][1] == o
4           && t[i][2] == o)   // row i
▶       || (t[0][i] == o && t[1][i] == o
▶           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 1, **column 1**: t[0][1] = 'O' ≠ 'X' → false. `&&` short-circuits, so the other 2 cells are never checked.

Step 5 / 9

VARIABLES

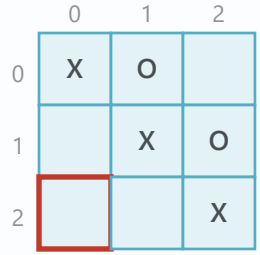
`o = 'X'` `i = 1`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
▶      if ((t[i][0] == o && t[i][1] == o
▶           && t[i][2] == o)   // row i
5       || (t[0][i] == o && t[1][i] == o
6           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 2, **row 2**: t[2][0] = empty ≠ 'X' → false. `&&` short-circuits, so the other 2 cells are never checked.

Step 6 / 9

VARIABLES

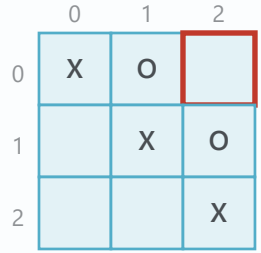
`o = 'X'` `i = 2`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
3      if ((t[i][0] == o && t[i][1] == o
4           && t[i][2] == o)   // row i
▶       || (t[0][i] == o && t[1][i] == o
▶           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 2, **column 2**: t[0][2] = empty ≠ 'X' → false. `&&` short-circuits, so the other 2 cells are never checked.

Step 7 / 9

VARIABLES

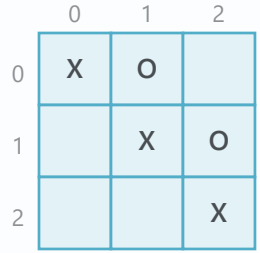
`o = 'X'` `i = 2`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
▶    for (int i = 0; i < 3; i++)
3      if ((t[i][0] == o && t[i][1] == o
4           && t[i][2] == o)   // row i
5       || (t[0][i] == o && t[1][i] == o
6           && t[2][i] == o))  // column i
7        return true;
8    return (t[0][0] == o && t[1][1] == o
9           && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11          && t[2][0] == o);  // anti-diagonal
12 }
```



i = 3 → the loop ends: no row or column is all X.

Step 8 / 9

VARIABLES

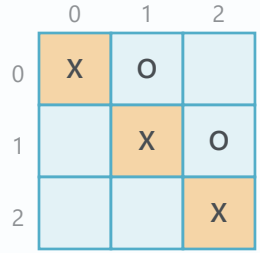
`o = 'X'` `i = 3`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

Code Trace: `hasWon(t, 'X')`

```
1  static boolean hasWon(char[][] t, char o) {
2    for (int i = 0; i < 3; i++)
3      if ((t[i][0] == o && t[i][1] == o
4           && t[i][2] == o)   // row i
5       || (t[0][i] == o && t[1][i] == o
6           && t[2][i] == o))  // column i
7        return true;
▶    return (t[0][0] == o && t[1][1] == o
▶            && t[2][2] == o)   // main diagonal
10       || (t[0][2] == o && t[1][1] == o
11           && t[2][0] == o);  // anti-diagonal
12 }
```



Main diagonal: t[0][0], t[1][1], and t[2][2] are all 'X' → `||` never looks at its right side; **return true**.

Step 9 / 9

VARIABLES

`o = 'X'`

CALL STACK

- `hasWon(t, 'X')`
- `isGameOn(t, 'X')`
- `main()`

COMPLEXITY

Cost of Array Operations

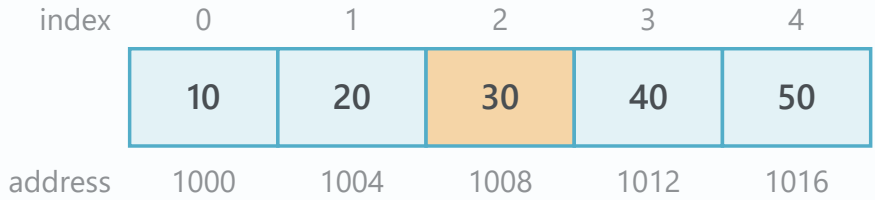
Complexity describes how running time and memory use grow with the input size **n**.

Operation	Time	Why?
Access / update by index `a[i]`	**O(1)**	the address is computed directly
Search for a value (unsorted array)	**O(n)**	may have to look at every element
Search for a value (sorted array, binary search)	**O(log n)**	each step halves the range
Insert / delete at the end (if there is room)	**O(1)**	no shifting
Insert / delete in the middle	**O(n)**	the elements after it shift
Sorting (a good algorithm)	**O(n log n)**	e.g. merge sort
Copying / growing	**O(n)**	every element moves to a new array

COMPLEXITY

Accessing an Element: O(1)

- The time doesn't depend on the array's size: it is the same for 5 elements and for 5 million.
- That's because every element's address can be computed: `address = start + i × element size`.



address(arr[2]) = 1000 + 2 × 4 = 1008

```
int[] arr = {10, 20, 30, 40, 50};
int value = arr[2];      // 30 — in one step, O(1)
```

Searching an Array: Linear Search, O(n)

target = 5

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

Call 1: search for **5**. The array is unsorted, so the only way is to scan it from start to end.

Step 1 / 14

```
▶ static boolean linearSearch(int[] arr,
2                             int target) {
3     for (int x : arr)
4         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

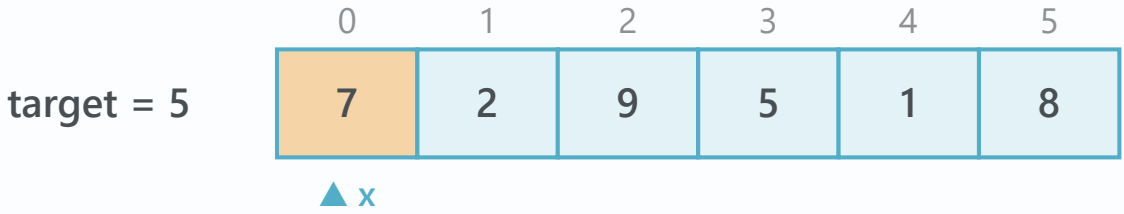
target = 5

CALL STACK

linearSearch(arr, 5)

main()

Searching an Array: Linear Search, O(n)



7 ≠ 5 → next element.

Step 2 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

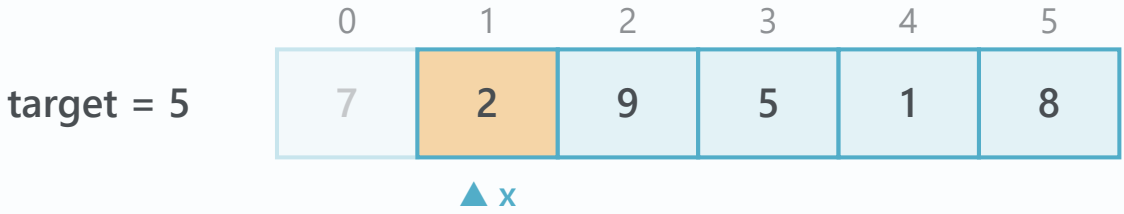
target = 5 | x = 7 | comparisons = 1

CALL STACK

linearSearch(arr, 5)

main()

Searching an Array: Linear Search, O(n)



2 ≠ 5 → next element.

Step 3 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

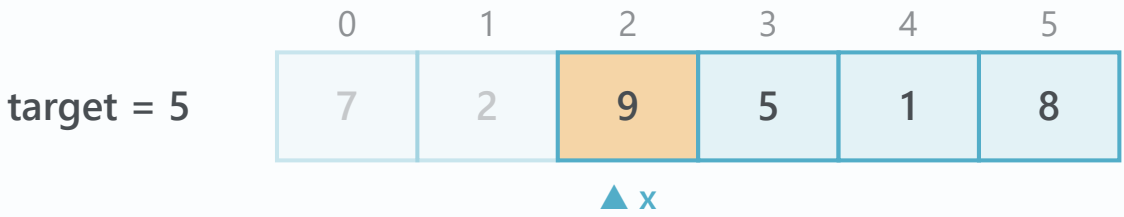
target = 5 | x = 2 | comparisons = 2

CALL STACK

linearSearch(arr, 5)

main()

Searching an Array: Linear Search, O(n)



9 ≠ 5 → next element.

Step 4 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

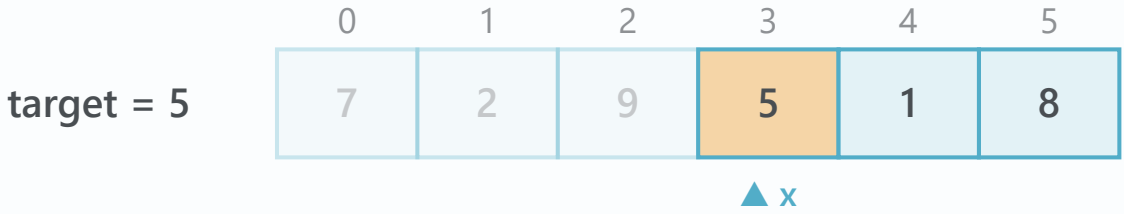
target = 5 | x = 9 | comparisons = 3

CALL STACK

linearSearch(arr, 5)

main()

Searching an Array: Linear Search, O(n)



5 == 5 → **equal**.

Step 5 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

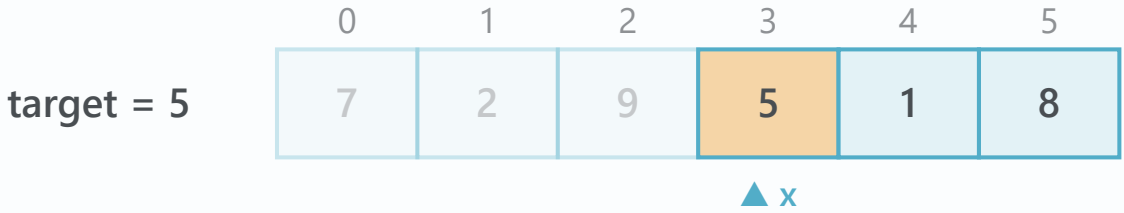
target = 5 | x = 5 | comparisons = 4

CALL STACK

linearSearch(arr, 5)

main()

Searching an Array: Linear Search, O(n)



return true: found after 4 comparisons; the rest are never examined. If the target is first, one comparison suffices → best case O(1).

Step 6 / 14

```
1 static boolean linearSearch(int[] arr,
2                              int target) {
3     for (int x : arr)
4         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

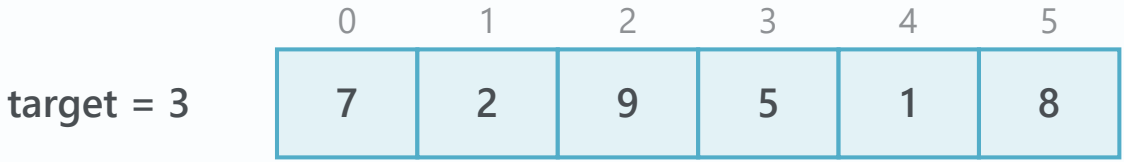
target = 5 | x = 5 | comparisons = 4

CALL STACK

linearSearch(arr, 5)

main()

Searching an Array: Linear Search, O(n)



Call 2: search for **3**. The array is unsorted, so the only way is to scan it from start to end.

Step 7 / 14

```
▶ static boolean linearSearch(int[] arr,
2                             int target) {
3     for (int x : arr)
4         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

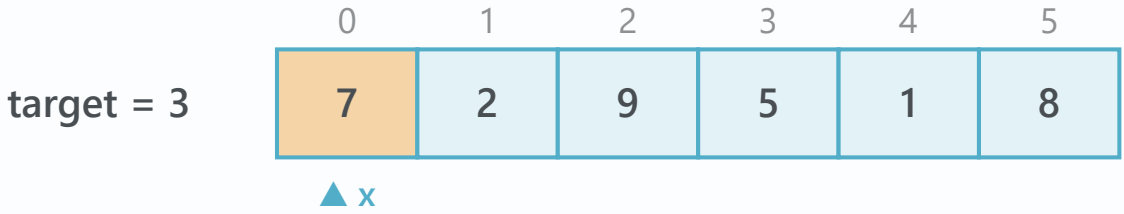
`target = 3`

CALL STACK

`linearSearch(arr, 3)`

`main()`

Searching an Array: Linear Search, O(n)



7 ≠ 3 → next element.

Step 8 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

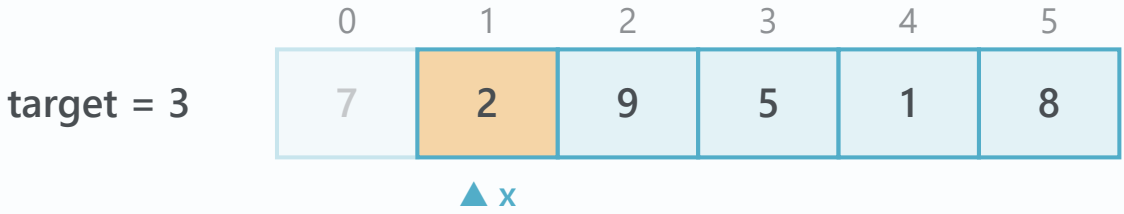
target = 3 | x = 7 | comparisons = 1

CALL STACK

linearSearch(arr, 3)

main()

Searching an Array: Linear Search, O(n)



2 ≠ 3 → next element.

Step 9 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

target = 3 | x = 2 | comparisons = 2

CALL STACK

linearSearch(arr, 3)

main()

Searching an Array: Linear Search, O(n)

target = 3

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

▲ x

9 ≠ 3 → next element.

Step 10 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

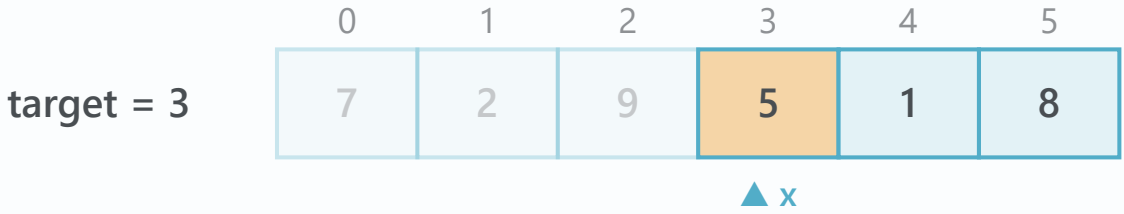
target = 3 | x = 9 | comparisons = 3

CALL STACK

linearSearch(arr, 3)

main()

Searching an Array: Linear Search, O(n)



5 ≠ 3 → next element.

Step 11 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

target = 3 | x = 5 | comparisons = 4

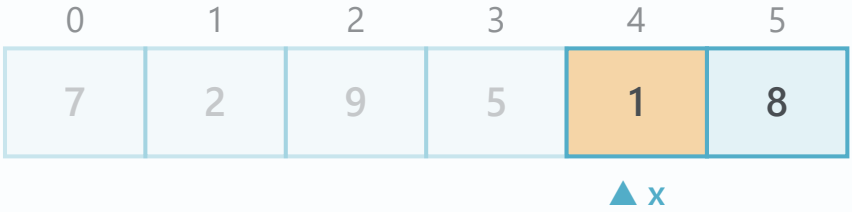
CALL STACK

linearSearch(arr, 3)

main()

Searching an Array: Linear Search, O(n)

target = 3



1 ≠ 3 → next element.

Step 12 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

target = 3 | x = 1 | comparisons = 5

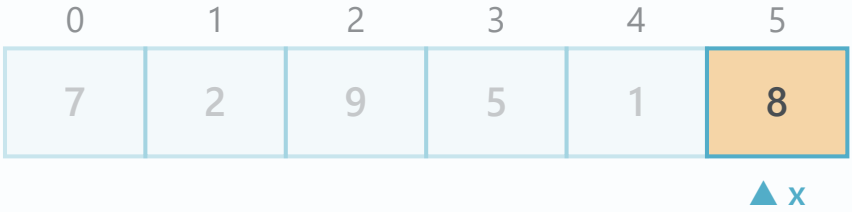
CALL STACK

linearSearch(arr, 3)

main()

Searching an Array: Linear Search, O(n)

target = 3



8 ≠ 3 → next element.

Step 13 / 14

```
1 static boolean linearSearch(int[] arr,
2                             int target) {
▶     for (int x : arr)
▶         if (x == target)
5             return true; // found
6     return false;
7 }
```

VARIABLES

target = 3 | x = 8 | comparisons = 6

CALL STACK

linearSearch(arr, 3)

main()

Searching an Array: Linear Search, O(n)

target = 3

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

End of array, **return false**: all n = 6 elements were examined → worst case **O(n)**.

Step 14 / 14

```
1 static boolean linearSearch(int[] arr,
2                              int target) {
3     for (int x : arr)
4         if (x == target)
5             return true; // found
▶     return false;
7 }
```

VARIABLES

`target = 3` `comparisons = 6`

CALL STACK

`linearSearch(arr, 3)`

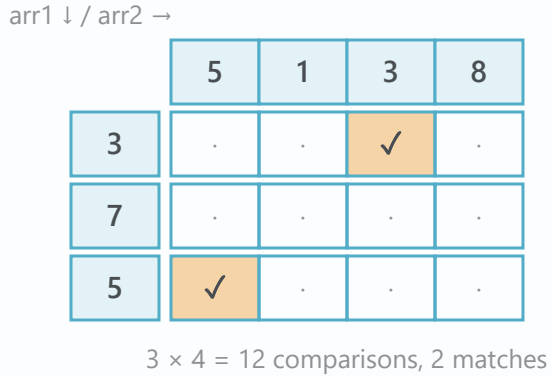
`main()`

COMPLEXITY

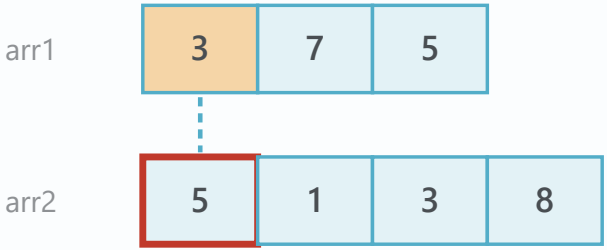
Finding Common Elements of Two Arrays: O(n·m)

```
static void printMatches(int[] arr1, int[] arr2) {
    for (int x : arr1)          // n times
        for (int y : arr2)      // m times
            if (x == y)
                System.out.println(
                    "Match: " + x);
}
```

- Every element of `arr1` is compared with every element of `arr2`: n·m comparisons.
- If both arrays have n elements, that is **$O(n^2)$**: too slow for large data.
- Better: sort one and use binary search, O(n log n), or use a hash set, O(n) on average.



Code Trace: `printMatches()`



comparisons: 1

Outer loop: **x = 3**; the inner loop scans arr2 from the start. 3 ≠ 5.

Step 1 / 15

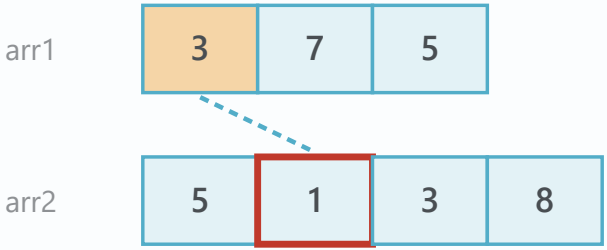
```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

VARIABLES

`x = 3` `y = 5` `comparisons = 1`

OUTPUT

Code Trace: `printMatches()`



comparisons: 2

3 ≠ 1.

Step 2 / 15

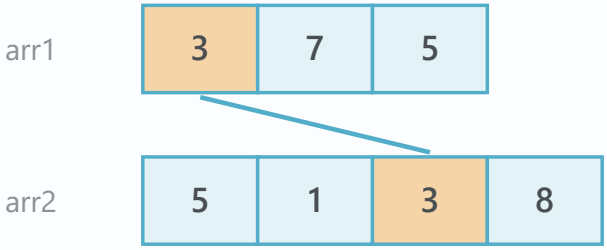
```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

VARIABLES

`x = 3` `y = 1` `comparisons = 2`

OUTPUT

Code Trace: `printMatches()`



comparisons: 3

3 == 3 → **match**.

Step 3 / 15

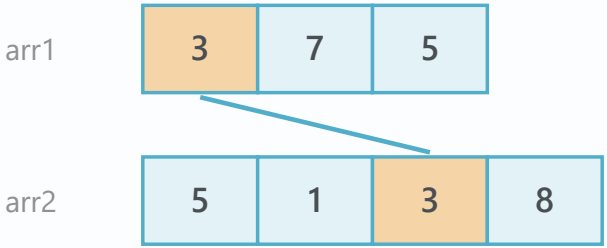
```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

VARIABLES

`x = 3` `y = 3` `comparisons = 3`

OUTPUT

Code Trace: `printMatches()`



comparisons: 3

The match is printed, but the inner loop still runs **to the end**.

Step 4 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
3         for (int y : arr2)      // m times
4             if (x == y)
▶                 System.out.println(
▶                     "Match: " + x);
7 }
```

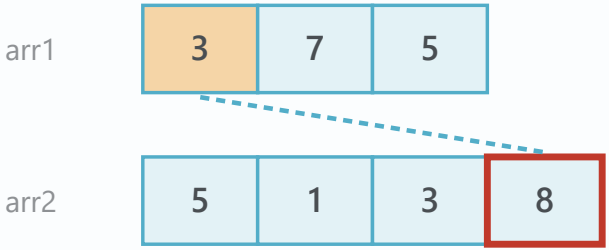
VARIABLES

`x = 3` `y = 3` `comparisons = 3`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 4

3 ≠ 8.

Step 5 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

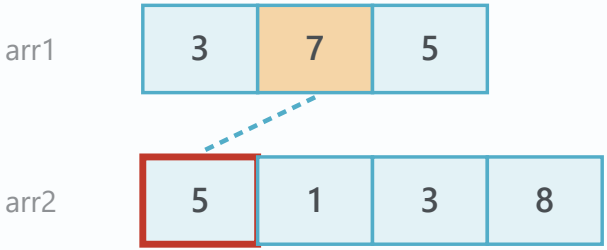
VARIABLES

`x = 3` `y = 8` `comparisons = 4`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 5

Outer loop: **x = 7**; the inner loop scans arr2 from the start. 7 ≠ 5.

Step 6 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

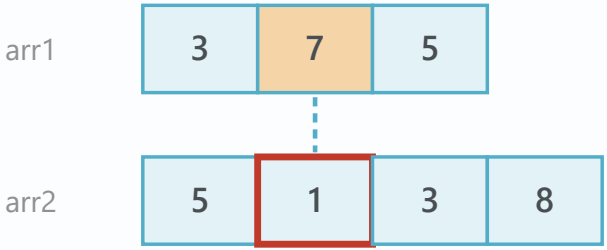
VARIABLES

`x = 7` `y = 5` `comparisons = 5`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 6

7 ≠ 1.

Step 7 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

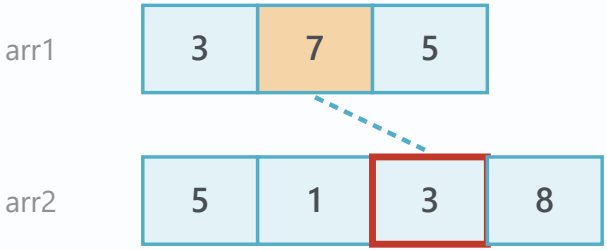
VARIABLES

`x = 7` `y = 1` `comparisons = 6`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 7

7 ≠ 3.

Step 8 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

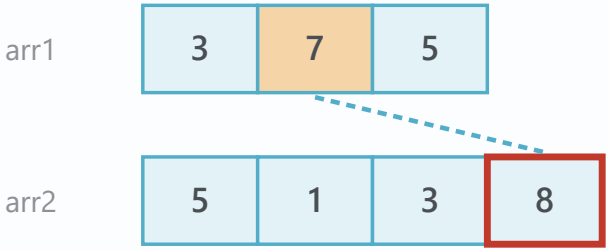
VARIABLES

`x = 7` `y = 3` `comparisons = 7`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 8

7 ≠ 8.

Step 9 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

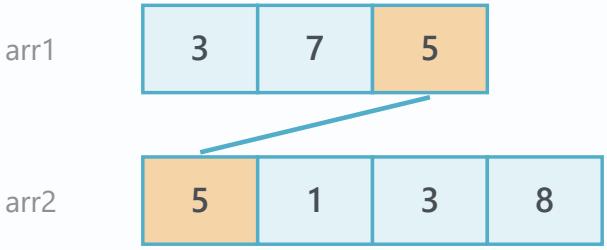
VARIABLES

`x = 7` `y = 8` `comparisons = 8`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 9

5 == 5 → **match**.

Step 10 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

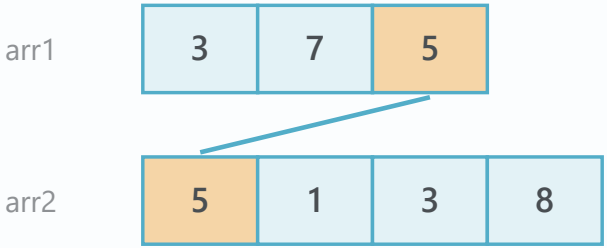
VARIABLES

`x = 5` `y = 5` `comparisons = 9`

OUTPUT

```
Match: 3
```

Code Trace: `printMatches()`



comparisons: 9

The match is printed, but the inner loop still runs **to the end**.

Step 11 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
3         for (int y : arr2)      // m times
4             if (x == y)
▶                 System.out.println(
▶                     "Match: " + x);
7 }
```

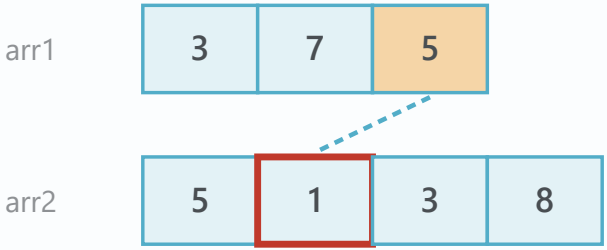
VARIABLES

`x = 5` `y = 5` `comparisons = 9`

OUTPUT

```
Match: 3
Match: 5
```

Code Trace: `printMatches()`



comparisons: 10

5 ≠ 1.

Step 12 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

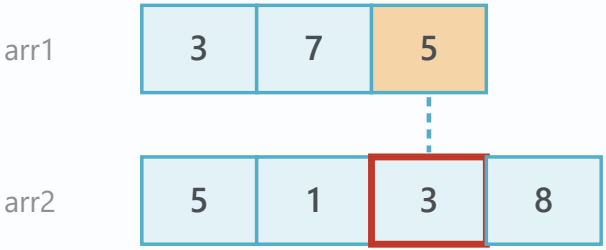
VARIABLES

`x = 5` `y = 1` `comparisons = 10`

OUTPUT

```
Match: 3
Match: 5
```

Code Trace: `printMatches()`



comparisons: 11

5 ≠ 3.

Step 13 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

VARIABLES

`x = 5` `y = 3` `comparisons = 11`

OUTPUT

```
Match: 3
Match: 5
```

Code Trace: `printMatches()`



comparisons: 12

5 ≠ 8.

Step 14 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
▶         for (int y : arr2)      // m times
▶             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
7 }
```

VARIABLES

`x = 5` `y = 8` `comparisons = 12`

OUTPUT

```
Match: 3
Match: 5
```

Code Trace: `printMatches()`



comparisons: 12

Done: 3 × 4 = **12 comparisons** = n·m. If both arrays have n elements, $O(n^2)$.

Step 15 / 15

```
1 static void printMatches(int[] arr1, int[] arr2) {
2     for (int x : arr1)          // n times
3         for (int y : arr2)      // m times
4             if (x == y)
5                 System.out.println(
6                     "Match: " + x);
▶ }
```

VARIABLES

`comparisons = 12`

OUTPUT

```
Match: 3
Match: 5
```

COMPLEXITY · STEP THROUGH THE CODE

Inserting: Shift Right, O(n)



Capacity 6, with 5 cells in use. To insert **4** at index 1, we first have to make room.

Step 1 / 8

```
▶ static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
4   for (int k = size; k > index; k--)
5     a[k] = a[k - 1];   // shift right
6   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

`a = [7, 2, 9, 5, 1, 0]` `size = 5`

`index = 1` `value = 4`

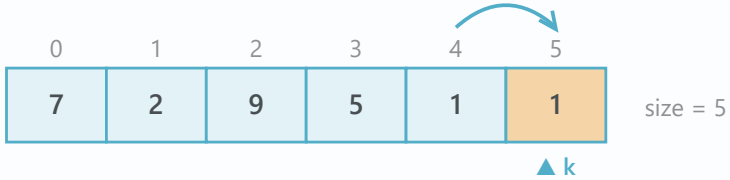
CALL STACK

`insert(a, 5, 1, 4)`

`main()`

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Inserting: Shift Right, O(n)



k = 5: **a[5] = a[4] = 1** (copied one place right). We start from the right: starting from the left, 2 would overwrite 9.

Step 2 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
▶   for (int k = size; k > index; k--)
▶     a[k] = a[k - 1];   // shift right
6   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

`a = [7, 2, 9, 5, 1, 1]` `size = 5` `index = 1` `value = 4` `k = 5`

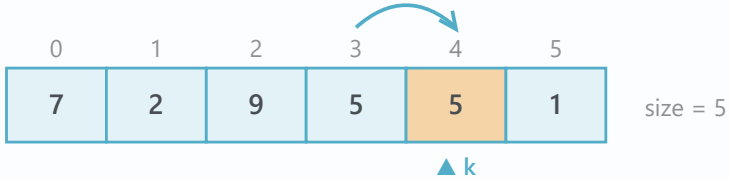
CALL STACK

`insert(a, 5, 1, 4)`

`main()`

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Inserting: Shift Right, O(n)



k = 4: **a[4] = a[3] = 5** (copied one place right).

Step 3 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
▶   for (int k = size; k > index; k--)
▶     a[k] = a[k - 1];   // shift right
6   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

`a = [7, 2, 9, 5, 5, 1]` `size = 5` `index = 1` `value = 4` `k = 4`

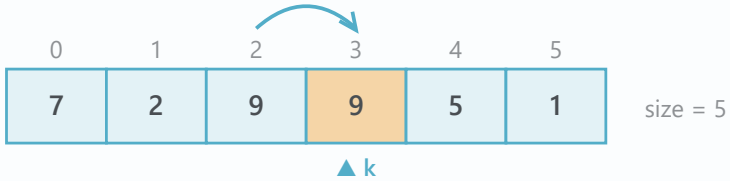
CALL STACK

`insert(a, 5, 1, 4)`

`main()`

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Inserting: Shift Right, O(n)



k = 3: **a[3] = a[2] = 9** (copied one place right).

Step 4 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
▶   for (int k = size; k > index; k--)
▶     a[k] = a[k - 1];   // shift right
6   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

`a = [7, 2, 9, 9, 5, 1]` `size = 5` `index = 1` `value = 4` `k = 3`

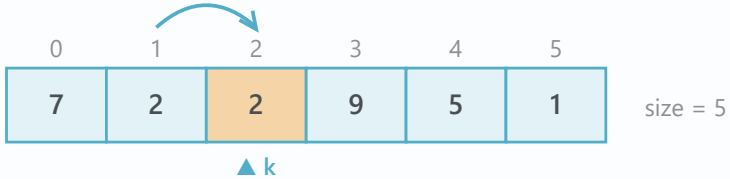
CALL STACK

`insert(a, 5, 1, 4)`

`main()`

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Inserting: Shift Right, O(n)



k = 2: **a[2] = a[1] = 2** (copied one place right).

Step 5 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
▶   for (int k = size; k > index; k--)
▶     a[k] = a[k - 1];   // shift right
6   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

a = [7, 2, 2, 9, 5, 1] | size = 5 | index = 1 | value = 4 | k = 2

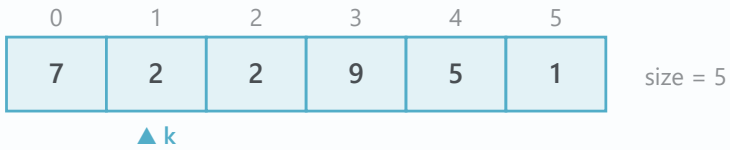
CALL STACK

insert(a, 5, 1, 4)

main()

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Inserting: Shift Right, O(n)



k = 1, and 1 > 1 is false → the loop ends. That took 4 shifts; a[1] and a[2] are now equal (2).

Step 6 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
▶   for (int k = size; k > index; k--)
5     a[k] = a[k - 1];   // shift right
6   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

`a = [7, 2, 2, 9, 5, 1]` `size = 5` `index = 1` `value = 4` `k = 1`

CALL STACK

`insert(a, 5, 1, 4)`

`main()`

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Inserting: Shift Right, O(n)



Write into the gap: **a[1] = 4**.

Step 7 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
4   for (int k = size; k > index; k--)
5     a[k] = a[k - 1];   // shift right
▶   a[index] = value;
7   return size + 1;   // new size
8 }
```

VARIABLES

`a = [7, 4, 2, 9, 5, 1]` `size = 5` `index = 1` `value = 4`

CALL STACK

`insert(a, 5, 1, 4)`

`main()`

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

COMPLEXITY · STEP THROUGH THE CODE

Inserting: Shift Right, O(n)



return 6. Inserting at the front takes n shifts, in the middle ≈ n/2 → **O(n)**; at the end (if there is room), O(1).

Step 8 / 8

```
1 static int insert(int[] a, int size,
2               int index, int value) {
3   // precondition: size < a.length
4   for (int k = size; k > index; k--)
5     a[k] = a[k - 1];   // shift right
6   a[index] = value;
▶   return size + 1;   // new size
8 }
```

VARIABLES

a = [7, 4, 2, 9, 5, 1] size = 6

index = 1 value = 4

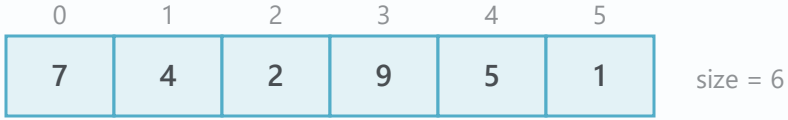
CALL STACK

insert(a, 5, 1, 4)

main()

If the array is full (size == a.length), a bigger array must be created and filled first: O(n).

Deleting: Shift Left, O(n)



After the previous insertion we delete **a[1] = 4**. To leave no gap, the elements to its right shift left.

Step 1 / 8

```
▶ static int remove(int[] a, int size, int index) {
2   for (int k = index; k < size - 1; k++)
3     a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

a = [7, 4, 2, 9, 5, 1]

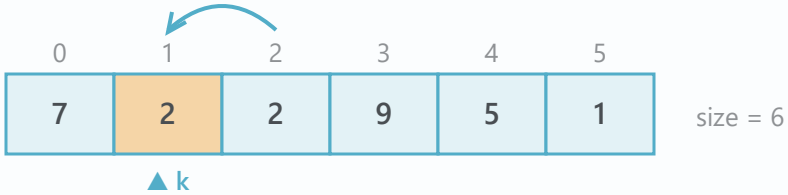
size = 6 index = 1

CALL STACK

remove(a, 6, 1)

main()

Deleting: Shift Left, O(n)



k = 1: **a[1] = a[2] = 2**. This time we start from the **left**, overwriting the deleted value.

Step 2 / 8

```
1 static int remove(int[] a, int size, int index) {
▶   for (int k = index; k < size - 1; k++)
▶     a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

a = [7, 2, 2, 9, 5, 1]

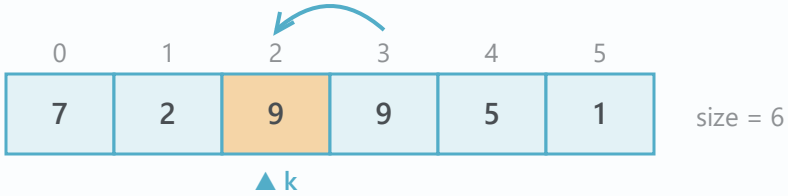
size = 6 | index = 1 | k = 1

CALL STACK

remove(a, 6, 1)

main()

Deleting: Shift Left, O(n)



k = 2: **a[2] = a[3] = 9**.

Step 3 / 8

```
1 static int remove(int[] a, int size, int index) {
▶   for (int k = index; k < size - 1; k++)
▶     a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

`a = [7, 2, 9, 9, 5, 1]`

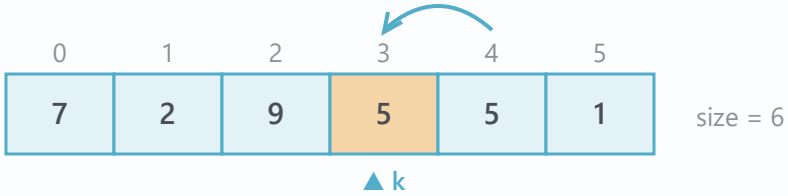
`size = 6` `index = 1` `k = 2`

CALL STACK

`remove(a, 6, 1)`

`main()`

Deleting: Shift Left, O(n)



k = 3: **a[3] = a[4] = 5**.

Step 4 / 8

```
1 static int remove(int[] a, int size, int index) {
▶   for (int k = index; k < size - 1; k++)
▶     a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

`a = [7, 2, 9, 5, 5, 1]`

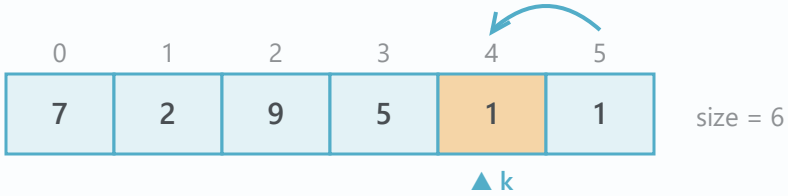
`size = 6` `index = 1` `k = 3`

CALL STACK

`remove(a, 6, 1)`

`main()`

Deleting: Shift Left, O(n)



k = 4: **a[4] = a[5] = 1**.

Step 5 / 8

```
1 static int remove(int[] a, int size, int index) {
    for (int k = index; k < size - 1; k++)
      a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

a = [7, 2, 9, 5, 1, 1]

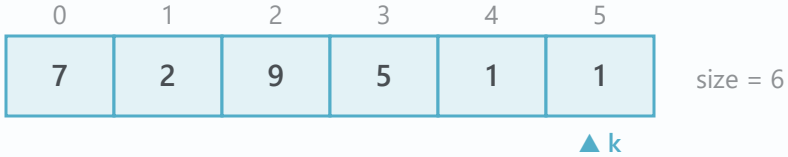
size = 6 | index = 1 | k = 4

CALL STACK

remove(a, 6, 1)

main()

Deleting: Shift Left, O(n)



k = 5 < 5 is false → the loop ends. 4 shifts.

Step 6 / 8

```
1 static int remove(int[] a, int size, int index) {
▶   for (int k = index; k < size - 1; k++)
3     a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

`a = [7, 2, 9, 5, 1, 1]`

`size = 6` `index = 1` `k = 5`

CALL STACK

`remove(a, 6, 1)`

`main()`

Deleting: Shift Left, O(n)



The last cell still held a copy of the old value; write 0 there (`null` in an object array, which matters for the GC).

Step 7 / 8

```
1 static int remove(int[] a, int size, int index) {
2   for (int k = index; k < size - 1; k++)
3     a[k] = a[k + 1];   // shift left
▶   a[size - 1] = 0;     // clear the old last cell
5   return size - 1;     // new size
6 }
```

VARIABLES

a = [7, 2, 9, 5, 1, 0]

size = 6 index = 1

CALL STACK

remove(a, 6, 1)

main()

Deleting: Shift Left, O(n)



return 5. Deleting at the front takes n − 1 shifts → **O(n)**; at the end, O(1).

Step 8 / 8

```
1 static int remove(int[] a, int size, int index) {
2   for (int k = index; k < size - 1; k++)
3     a[k] = a[k + 1];   // shift left
4   a[size - 1] = 0;     // clear the old last cell
▶   return size - 1;     // new size
6 }
```

VARIABLES

a = [7, 2, 9, 5, 1, 0]

size = 5 | index = 1

CALL STACK

remove(a, 6, 1)

main()

COMPLEXITY

Sorting an Array: Time Complexity

The cost of sorting depends on the algorithm:

Algorithm	Best	Average	Worst	Extra space
Bubble sort	O(n)*	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Quick sort	O(n log n)	O(n log n)	**$O(n^2)$**	O(log n)**
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)

* For the version that stops after a pass with no swaps. Quick sort's worst case comes from bad pivots (e.g. sorted input with the first element as pivot). ** Quick sort's extra space is O(log n) on average and O(n) in the worst case (the recursion depth).

`Arrays.sort` uses dual-pivot quick sort for primitive arrays and stable TimSort for object arrays.

Selection Sort

```
▶ int[] arr = {64, 25, 12, 22, 11};
▶ int n = arr.length;
3 for (int i = 0; i < n - 1; i++) {
4     int minIndex = i;
5     for (int j = i + 1; j < n; j++)
6         if (arr[j] < arr[minIndex])
7             minIndex = j;
8     // swap by index, in place
9     int temp = arr[minIndex];
10    arr[minIndex] = arr[i];
11    arr[i] = temp;
12 }
```



An unsorted 5-element array. Each pass will move the smallest element of the unsorted part to the front of that part.

Step 1 / 19

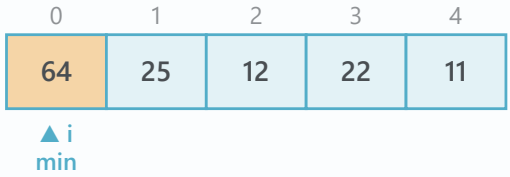
VARIABLES

`arr = [64, 25, 12, 22, 11]` `n = 5`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
▶      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



Pass 1: the unsorted part starts at index 0; the smallest so far is arr[0] = 64.

Step 2 / 19

VARIABLES

`arr = [64, 25, 12, 22, 11]` `i = 0` `minIndex = 0` `comps = 0`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
▶              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



25 < 64 → **minIndex = 1**.

Step 3 / 19

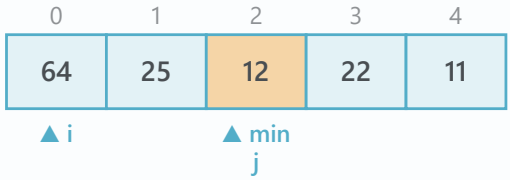
VARIABLES

`arr = [64, 25, 12, 22, 11]` `i = 0` `j = 1` `minIndex = 1` `comps = 1`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
▶              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



12 < 25 → **minIndex = 2**.

Step 4 / 19

VARIABLES

`arr = [64, 25, 12, 22, 11]` `i = 0` `j = 2` `minIndex = 2` `comps = 2`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



22 < 12 is false → no change.

Step 5 / 19

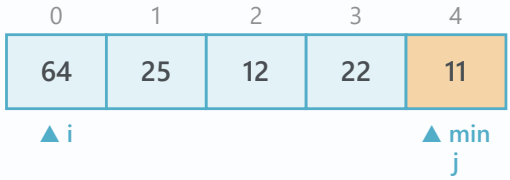
VARIABLES

`arr = [64, 25, 12, 22, 11]` `i = 0` `j = 3` `minIndex = 2` `comps = 3`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
▶              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



11 < 12 → **minIndex = 4**.

Step 6 / 19

VARIABLES

`arr = [64, 25, 12, 22, 11]` `i = 0` `j = 4` `minIndex = 4` `comps = 4`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
▶      int temp = arr[minIndex];
▶      arr[minIndex] = arr[i];
▶      arr[i] = temp;
12 }
```



Swap: arr[0] ↔ arr[4] (64 ↔ 11). Index 0 is now final.

Step 7 / 19

VARIABLES

`arr = [11, 25, 12, 22, 64]` `i = 0` `minIndex = 4` `comps = 4`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
▶      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



Pass 2: the unsorted part starts at index 1; the smallest so far is arr[1] = 25.

Step 8 / 19

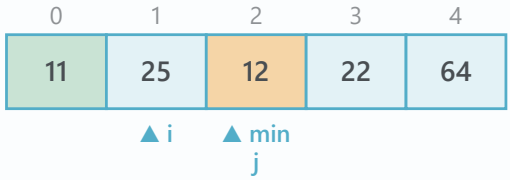
VARIABLES

`arr = [11, 25, 12, 22, 64]` `i = 1` `minIndex = 1` `comps = 4`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
▶              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



12 < 25 → **minIndex = 2**.

Step 9 / 19

VARIABLES

`arr = [11, 25, 12, 22, 64]` `i = 1` `j = 2` `minIndex = 2` `comps = 5`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



22 < 12 is false → no change.

Step 10 / 19

VARIABLES

`arr = [11, 25, 12, 22, 64]` `i = 1` `j = 3` `minIndex = 2` `comps = 6`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



64 < 12 is false → no change.

Step 11 / 19

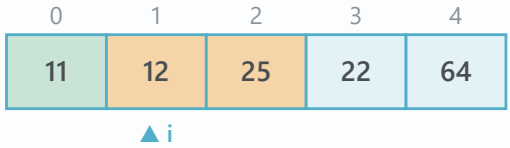
VARIABLES

`arr = [11, 25, 12, 22, 64]` `i = 1` `j = 4` `minIndex = 2` `comps = 7`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
▶      int temp = arr[minIndex];
▶      arr[minIndex] = arr[i];
▶      arr[i] = temp;
12 }
```



Swap: arr[1] ↔ arr[2] (25 ↔ 12). Index 1 is now final.

Step 12 / 19

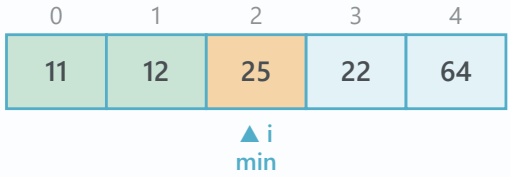
VARIABLES

`arr = [11, 12, 25, 22, 64]` `i = 1` `minIndex = 2` `comps = 7`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
▶      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



Pass 3: the unsorted part starts at index 2; the smallest so far is arr[2] = 25.

Step 13 / 19

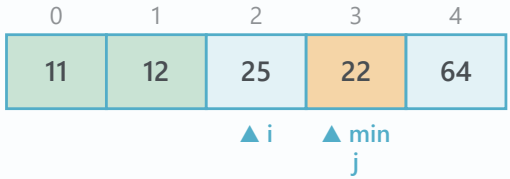
VARIABLES

`arr = [11, 12, 25, 22, 64]` `i = 2` `minIndex = 2` `comps = 7`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
▶              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



22 < 25 → **minIndex = 3**.

Step 14 / 19

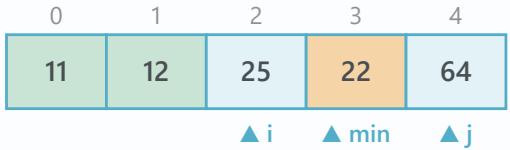
VARIABLES

`arr = [11, 12, 25, 22, 64]` `i = 2` `j = 3` `minIndex = 3` `comps = 8`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



64 < 22 is false → no change.

Step 15 / 19

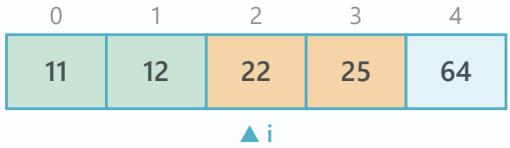
VARIABLES

`arr = [11, 12, 25, 22, 64]` `i = 2` `j = 4` `minIndex = 3` `comps = 9`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
▶      int temp = arr[minIndex];
▶      arr[minIndex] = arr[i];
▶      arr[i] = temp;
12 }
```



Swap: arr[2] ↔ arr[3] (25 ↔ 22). Index 2 is now final.

Step 16 / 19

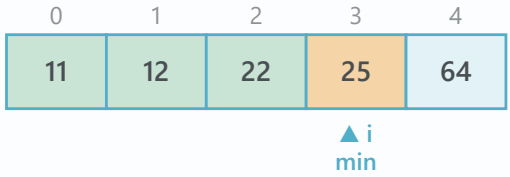
VARIABLES

`arr = [11, 12, 22, 25, 64]` `i = 2` `minIndex = 3` `comps = 9`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
▶      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



Pass 4: the unsorted part starts at index 3; the smallest so far is arr[3] = 25.

Step 17 / 19

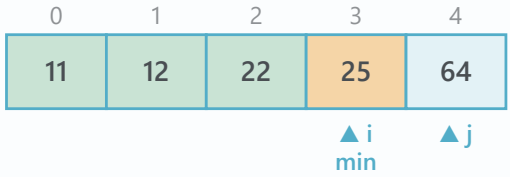
VARIABLES

`arr = [11, 12, 22, 25, 64]` `i = 3` `minIndex = 3` `comps = 9`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
▶          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
9      int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```



64 < 25 is false → no change.

Step 18 / 19

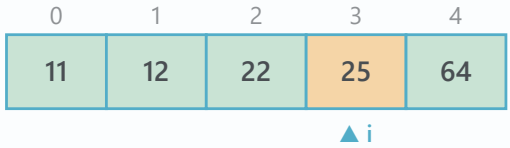
VARIABLES

`arr = [11, 12, 22, 25, 64]` `i = 3` `j = 4` `minIndex = 3` `comps = 10`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Selection Sort

```
1  int[] arr = {64, 25, 12, 22, 11};
2  int n = arr.length;
3  for (int i = 0; i < n - 1; i++) {
4      int minIndex = i;
5      for (int j = i + 1; j < n; j++)
6          if (arr[j] < arr[minIndex])
7              minIndex = j;
8      // swap by index, in place
▶      int temp = arr[minIndex];
▶      arr[minIndex] = arr[i];
▶      arr[i] = temp;
12 }
```



The smallest is already in place (minIndex = i = 3), so it swaps with itself and nothing changes. The last element is then in place too → **sorted**: 4 + 3 + 2 + 1 = 10 = n(n−1)/2 comparisons, O(n^2).

Step 19 / 19

VARIABLES

`arr = [11, 12, 22, 25, 64]` `i = 3` `minIndex = 3` `comps = 10`

Java passes arguments by value: `swap(arr[minIndex], arr[i])` swaps only copies. Swap by index instead (or with `swap(arr, i, minIndex)`).

Insertion Sort



At first the sorted part is just a[0] (a single element is always sorted). Each pass puts the next element into its place in the sorted part by **shifting**.

Step 1 / 16

```
▶ static void insertionSort(int[] a) {
2     for (int k = 1; k < a.length; k++) {
3         int key = a[k];
4         int j = k;
5         while (j > 0 && a[j - 1] > key) {
6             a[j] = a[j - 1];   // shift right
7             j--;
8         }
9         a[j] = key;         // insert
10    }
11 }
```

VARIABLES

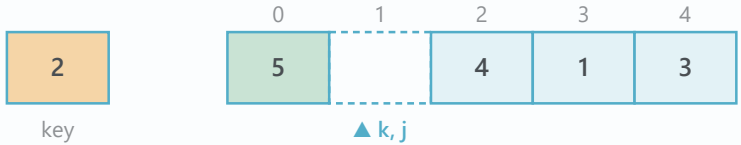
a = [5, 2, 4, 1, 3]

CALL STACK

insertionSort(a)

main()

Insertion Sort



Pass k = 1: set aside `key = a[1] = 2`; its cell now counts as a gap.

Step 2 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
▶          int key = a[k];
▶          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
9          a[j] = key;        // insert
10     }
11 }
```

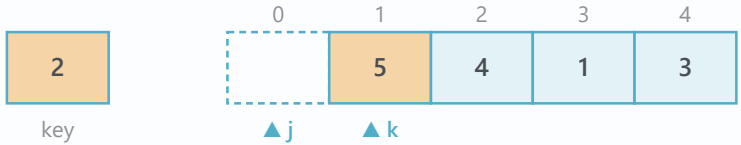
VARIABLES

`a = [5, 2, 4, 1, 3]` `k = 1` `key = 2` `j = 1`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[0] = 5 > 2` → 5 shifts one place right; the gap moves to index 0.

Step 3 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

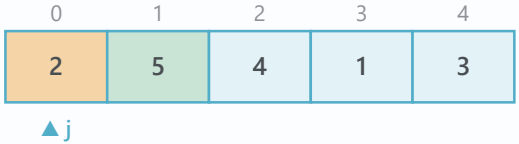
VARIABLES

`a = [5, 5, 4, 1, 3]` `k = 1` `key = 2` `j = 0`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`j == 0` : we have reached the front of the sorted part; `a[0] = 2` . The first 2 elements are now sorted.

Step 4 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
▶          a[j] = key;         // insert
10     }
11 }
```

VARIABLES

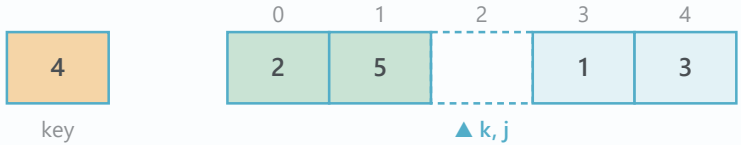
`a = [2, 5, 4, 1, 3]` `k = 1` `key = 2` `j = 0`

CALL STACK

`insertionSort(a)`

`main()`

Insertion Sort



Pass k = 2: set aside `key = a[2] = 4`; its cell now counts as a gap.

Step 5 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
▶          int key = a[k];
▶          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

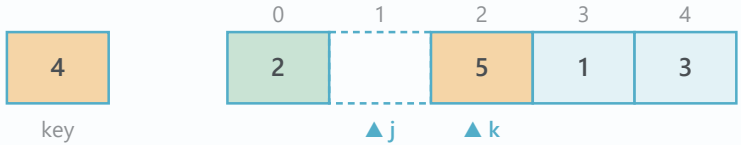
VARIABLES

`a = [2, 5, 4, 1, 3]` `k = 2` `key = 4` `j = 2`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[1] = 5 > 4` → 5 shifts one place right; the gap moves to index 1.

Step 6 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

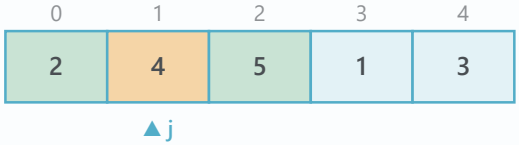
VARIABLES

`a = [2, 5, 5, 1, 3]` `k = 2` `key = 4` `j = 1`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[0] = 2 ≤ 4` : stop; `a[1] = 4` . The first 3 elements are now sorted.

Step 7 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
▶          a[j] = key;         // insert
10     }
11 }
```

VARIABLES

`a = [2, 4, 5, 1, 3]` `k = 2`

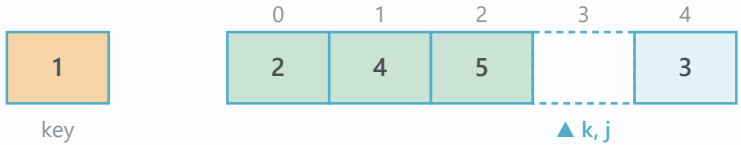
`key = 4` `j = 1`

CALL STACK

`insertionSort(a)`

`main()`

Insertion Sort



Pass k = 3: set aside `key = a[3] = 1`; its cell now counts as a gap.

Step 8 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
▶          int key = a[k];
▶          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

VARIABLES

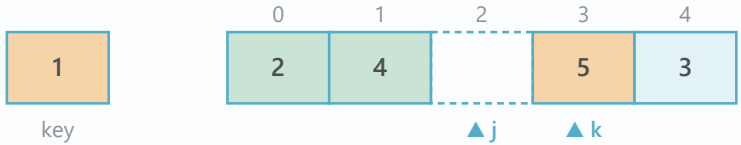
`a = [2, 4, 5, 1, 3]` `k = 3`

`key = 1` `j = 3`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[2] = 5 > 1` → 5 shifts one place right; the gap moves to index 2.

Step 9 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

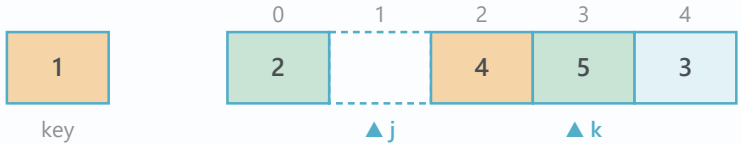
VARIABLES

`a = [2, 4, 5, 5, 3]` `k = 3` `key = 1` `j = 2`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[1] = 4 > 1` → 4 shifts one place right; the gap moves to index 1.

Step 10 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

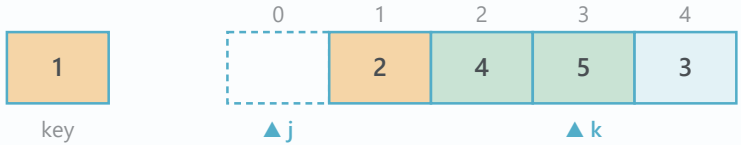
VARIABLES

`a = [2, 4, 4, 5, 3]` `k = 3` `key = 1` `j = 1`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[0] = 2 > 1` → 2 shifts one place right; the gap moves to index 0.

Step 11 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

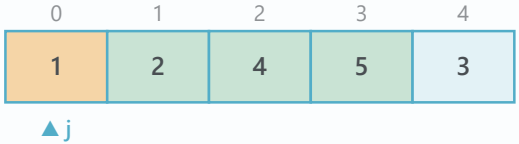
VARIABLES

`a = [2, 2, 4, 5, 3]` `k = 3` `key = 1` `j = 0`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`j == 0` : we have reached the front of the sorted part; `a[0] = 1` . The first 4 elements are now sorted.

Step 12 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
▶          a[j] = key;         // insert
10     }
11 }
```

VARIABLES

a = [1, 2, 4, 5, 3] k = 3

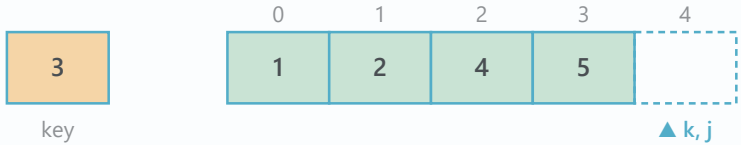
key = 1 j = 0

CALL STACK

insertionSort(a)

main()

Insertion Sort



Pass k = 4: set aside `key = a[4] = 3`; its cell now counts as a gap.

Step 13 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
▶          int key = a[k];
▶          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

VARIABLES

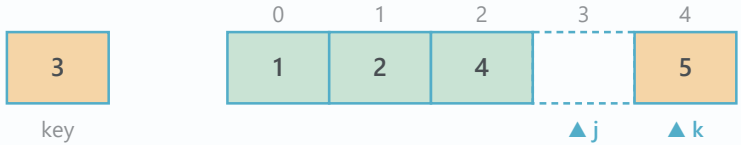
`a = [1, 2, 4, 5, 3]` `k = 4` `key = 3` `j = 4`

CALL STACK

`insertionSort(a)`

`main()`

Insertion Sort



`a[3] = 5 > 3` → 5 shifts one place right; the gap moves to index 3.

Step 14 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

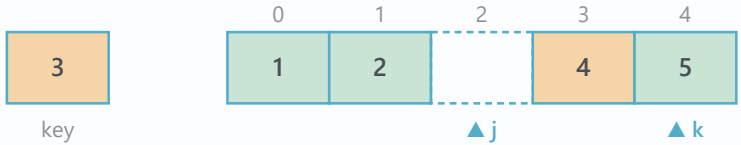
VARIABLES

`a = [1, 2, 4, 5, 5]` `k = 4` `key = 3` `j = 3`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[2] = 4 > 3` → 4 shifts one place right; the gap moves to index 2.

Step 15 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
▶          while (j > 0 && a[j - 1] > key) {
▶              a[j] = a[j - 1];   // shift right
▶              j--;
8          }
9          a[j] = key;         // insert
10     }
11 }
```

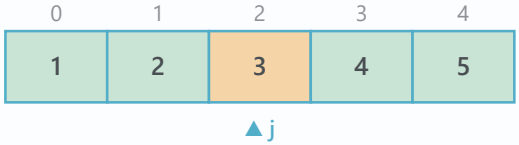
VARIABLES

`a = [1, 2, 4, 4, 5]` `k = 4` `key = 3` `j = 2`

CALL STACK

- `insertionSort(a)`
- `main()`

Insertion Sort



`a[1] = 2 ≤ 3` : stop; `a[2] = 3` . The array is sorted. 7 shifts = the number of inversions in the array. Worst case $O(n^2)$; O(n) on an already sorted array.

Step 16 / 16

```
1  static void insertionSort(int[] a) {
2      for (int k = 1; k < a.length; k++) {
3          int key = a[k];
4          int j = k;
5          while (j > 0 && a[j - 1] > key) {
6              a[j] = a[j - 1];   // shift right
7              j--;
8          }
▶          a[j] = key;         // insert
10     }
11 }
```

VARIABLES

`a = [1, 2, 3, 4, 5]` `k = 4` `key = 3` `j = 2`

CALL STACK

`insertionSort(a)`

`main()`

COMPLEXITY

Space Complexity and Memory Use

- Space complexity describes how the memory an algorithm or data structure needs grows with n.
- An array of n elements takes **O(n)** space; the number of bytes depends on the element type.

Array type	Per element	100 elements (data only)
`int[]`	4 bytes	400 bytes
`char[]` (UTF-16)	2 bytes	200 bytes
`double[]`, `long[]`	8 bytes	800 bytes
`Object[]` (references)	4 / 8 bytes	400 / 800 bytes + the objects

Overhead

Every array object also carries a header of about 16 bytes (type information + `length`).

Arrays of references

Only the references are contiguous; the objects they point to take separate space on the heap.

SUMMARY

Common Mistakes with Arrays

Off-by-one error

With `<=`, the last iteration accesses `a[a.length]` → exception.

```
for (i = 0; i < a.length; i++)
```

`length` / `length()`

A field on arrays, a method on `String`.

```
a.length · s.length()
```

`=` does not copy

`b = a` points to the same array; for a copy use `clone` / `copyOf`.

```
int[] b = a.clone();
```

Comparing contents with `==`

`==` compares references.

```
Arrays.equals(a, b)
```

Printing an array directly

`println(a)` prints type@address (`[I@…`).

```
Arrays.toString(a)
```

Null rows in a jagged array

After `new int[3][]`, `m[0][0]` → `NullPointerException`.

```
m[i] = new int[k];
```

Also: `binarySearch` gives wrong answers on an unsorted array, and in a 2D array use `m[i].length` for each row's column count.

Lab: Sorting

Selection sort: 64, 25, 12, 22, 11

Input (comma-separated numbers) 64, 25, 12, 22, 11

CODE · **step 1 / 38**

```
 1 int[] arr = {64, 25, 12, 22, 11};
 2 int n = arr.length;
 3 for (int i = 0; i < n - 1; i++) {
 4     int minIndex = i;
 5     for (int j = i + 1; j < n; j++)
 6         if (arr[j] < arr[minIndex])
 7             minIndex = j;
 8     // swap by index, in place
 9     int temp = arr[minIndex];
10     arr[minIndex] = arr[i];
11     arr[i] = temp;
12 }
```

arr 64 25 12 22 11

comparisons = 0

OUTPUT —

arr is created with 5 items: {64, 25, 12, 22, 11}.

SUMMARY

Summary

Structure
One type, fixed size, contiguous memory; indices 0 … length − 1.

In Java
An array is an object on the heap; `=` copies the reference, `Arrays.copyOf` the elements.

Multidimensional
An array of arrays; rows can differ in length. `m[i][j]`.

Operation	Cost	Operation	Cost
Access `a[i]`	O(1)	Insert / delete in the middle	O(n)
Linear search	O(n)	Reversing	O(n)
Selection sort	$O(n^2)$	Transpose (m×n)	O(m·n)

Interactive simulator: to try access, insertion, deletion, and search step by step, open the Array Simulator.

SUMMARY

Sample Problems

1. After `int[] a = new int[4];`, what happens on `a[4]`? What is `a.length`?
2. After `int[] b = a; b[0] = 9;`, what is `a[0]`? Why?
3. Write a method that finds the largest element of an n-element array. What is its complexity?
4. What does {5, 1, 4, 2, 8} look like after the first two passes of selection sort?
5. How many bytes does the data of a 1000-element `double[]` take?
6. How many rows and columns does the transpose of a 3×4 matrix have? Which element is `t[2][1]`?

▶ **Show answers**

SUMMARY

Exercises

1. After `int[] a = {3, 1, 2}; int[] b = a; b = new int[] {9};`, what are `a[0]` and `b.length`?
2. Run insertion sort on {7, 3, 5, 1}. What is the array after each outer pass? How many shifts are made?
3. After `int[][] m = new int[3][]; m[1] = new int[4];`, what are `m.length`, `m[1].length`, and `m[0]`?
4. What is the worst-case cost of inserting into a sorted array that has room, keeping it sorted?
5. What does `encrypt("XYZ", 2)` return? What `k` decrypts it?
6. Write a method that rotates an array one step left in place (`a[0]` moves to the end). What does it cost?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Lafore, *Data Structures and Algorithms in Java*, 2nd ed., 2002.
- Oracle, Java SE 17 API documentation for `Arrays` and `System`.

The End

Chapter 3: Arrays

Next deck: Chapter 4: Recursion · Study questions · Simulator: Array Simulator · Glossary