Module 6 of 20

Arrays

Fixed-size typed arrays, 2D and jagged arrays, the Arrays utility class, aliasing versus copying, and the in-place patterns.

Arrays are Java's most primitive data structure and the one under every other collection: fixed length, one element type, zero-based, bounds-checked, contiguous. Everything about them follows from one fact — an array is an object and an array variable is a reference.

This module builds up from that fact: creation and indexing, arrays of arrays (and why 2D copies are shallow), the Arrays class that supplies sorting, searching, copying and printing, the aliasing and defensive-copy rules that keep state safe, and the in-place patterns — reverse, rotate, count, prefix sums, two pointers — that interview problems are made of.

By the end you handle arrays fluently and safely, and recognise the standard patterns inside larger problems.

CONTENTS

1	Arrays — fixed-size, typed, zero-based	13 min
2	Two-dimensional and jagged arrays	12 min
3	The Arrays class and System.arraycopy	13 min
4	Aliasing, shallow copies and defensive copies	12 min
5	Array patterns — reverse, rotate, count, prefix sums	15 min
✓	Checkpoint — Arrays	22 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Arrays — fixed-size, typed, zero-based

An array is the simplest data structure Java has and the one closest to the machine: a fixed-length block of elements of one type, indexed from zero, laid out contiguously. Every other collection is built on it. This lesson covers how arrays are declared, created and indexed, what an array *is* at run time, and the errors that come from forgetting any of that.

Declaring and creating

```
int[] counts;                    // declares a variable of type "array of int
counts = new int[10];            // creates an array of 10 ints, all 0
int[] primes = {2, 3, 5, 7};     // array initialiser — length 4, only in a d
int[] more = new int[] {11, 13}; // the same, usable anywhere (arguments, ret
String[] names = new String[3];  // three nulls
double[] empty = new double[0];  // a legal, empty array
```

Two things to notice. First, the type is `int[]` — "array of int" — and the variable is a **reference**; `counts` is `null` until an array is assigned. Second, the length is fixed at creation: `new int[10]` is ten slots forever. To "grow" an array you create a bigger one and copy (that is exactly what `ArrayList` does internally).

`int counts[]` (C style) is also accepted by the compiler and rejected by every style guide. Put the brackets on the type.

Default values

`new T[n]` fills every slot with the default for `T`: `0` for numeric types, `'\0'` for `char`, `false` for `boolean`, `null` for any reference type. This is guaranteed — unlike C, an array is never uninitialised memory. The null defaults in a `String[]` are the source of many early `NullPointerException`s: `names[0].length()` before any assignment.

Indexing and length

```
int[] a = {10, 20, 30};
a[0]          // 10 — the first element
a[2]          // 30 — the last: index length - 1
a.length      // 3 — a final FIELD, no parentheses (String.length() is a metho
a[3]          // ArrayIndexOutOfBoundsException: Index 3 out of bounds for len
a[-1]         // ArrayIndexOutOfBoundsException: Index -1 out of bounds for le
a[1] = 25;    // assignment
a[1] += 5;    // compound assignment works on elements
```

Indices run from `0` to `length - 1`. Every access is **bounds-checked** at run time; an out-of-range index throws immediately, with a message naming the index and the length. This costs a comparison per access (the JIT eliminates most of them in loops) and buys the memory safety C lacks.

The canonical loop is `for (int i = 0; i < a.length; i++)`, and when the index is not needed, `for (int v : a)`.

Arrays are objects

Every array is an object on the heap, even `int[]`:

- It has a length and a runtime type (`int[].class`, `String[].class`).
- Array variables hold references; `int[] b = a;` makes `b` and `a` the **same array** (next lessons).
- `a == b` compares references, not contents; `a.equals(b)` is the same as `==` (arrays do not override it). Use `Arrays.equals`.
- `a.toString()` gives `[I@1b6d3586` (type code and hash), not the contents. Use `Arrays.toString`.
- An array's superclass is `Object`; it implements `Cloneable` and `Serializable`; `a.clone()` makes a shallow copy.
- `Object[] o = new String[3];` compiles — arrays are *covariant* — and `o[0] = 1;` then throws `ArrayStoreException` at run time. Generics were designed the opposite way to avoid exactly this.

Arrays of primitives versus arrays of objects

`int[]` holds the numbers themselves, contiguously: 1000 ints are 4000 bytes plus a header. `Integer[]` holds 1000 *references* to `Integer` objects scattered on the heap. For numeric work, primitive arrays are several times faster and smaller — the reason they exist alongside `List<Integer>`.

Multidimensional preview

`int[][] grid = new int[3][4];` — an array of 3 arrays of 4 ints. The next lesson covers these.

Common errors

Error	Cause
`NullPointerException` at `a.length`	Declared but never created: `int[] a;`
`ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`	`<=` instead of `<` in the loop, or `a[a.length]`
`NegativeArraySizeException`	`new int[n]` with `n < 0` — a bad computed size
`ArrayStoreException`	Storing the wrong type through a covariant reference
`incompatible types: int[] cannot be converted to int`	`int x = a;` — the array, not an element
`array required, but int found`	Indexing a non-array: `x[0]` where `x` is an `int`

Arrays and `main`

`String[] args` in `main` is an ordinary array: `args.length` is the number of command-line arguments and `args[0]` the first. It is never null — with no arguments it is an empty array.

When to use an array

Fixed size known up front; numeric or character data; performance matters; multi-dimensional grids.

Otherwise, `ArrayList` (Module 14) — it grows, has `add` / `remove` / `contains`, and its `toString` is readable. The switch costs little; arrays remain the right tool for algorithms and hot loops.

Interview angle

"Are arrays objects in Java?" Yes — heap-allocated, with a `length` field and a runtime type; variables hold references.

"Can you change an array's size?" No; create a new one and copy (`Arrays.copyOf`).

"`a.length` vs `s.length()` vs `list.size()`?" Field, method, method — a classic trick question.

"What is stored in `new String[5]`?" Five nulls.

Key takeaways

`T[] a = new T[n]` — fixed length, zero-based, every slot defaulted (0/false/null).

`a.length` is a field; valid indices are `0 … length - 1`; every access is bounds-checked.

Arrays are objects: variables are references, `==` is identity, `toString` is unreadable — use the `Arrays` class.

Primitive arrays are contiguous values; object arrays are references.

EXERCISE 1

Fill and report

Read an integer `n`, then `n` integers into an `int[]`. Print three lines: the array reversed (space-separated), `sum=<total>`, and `evens=<count of even values>`. Use the array's `length` and index loops.

For `n = 0` the first line is empty.

Example: `4` then `1 2 3 4` →

```
4 3 2 1
sum=10
evens=2
```

INPUT 1

```
4
1 2 3 4
```

OUTPUT 1

```
4 3 2 1
sum=10
evens=2
```

INPUT 2

```
0
```

OUTPUT 2

```
sum=0
evens=0
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();

        StringBuilder out = new StringBuilder();
        for (int i = a.length - 1; i ⩾ 0; i--) {
            if (i < a.length - 1) out.append(' ');
            out.append(a[i]);
        }
        out.append('\n');
        long sum = 0;
        int evens = 0;
        for (int v : a) {
            sum += v;
            if (v % 2 == 0) evens++;
        }
        out.append("sum=").append(sum).append('\n');
        out.append("evens=").append(evens).append('\n');
        System.out.print(out);
    }
}
```

EXERCISE 2

Dice histogram

Read an integer `n` and `n` dice rolls (each 1–6). Count them in an `int[7]` indexed by face (index 0 unused), then print one line per face `<face>: <count> <stars>` with one `*` per roll, and finally `mode=<the most rolled face, lowest on ties>`.

Example: `6` then `3 1 3 6 3 1` →

```
1: 2 **
2: 0
3: 3 ***
4: 0
5: 0
6: 1 *
mode=3
```

INPUT 1

```
6
3 1 3 6 3 1
```

OUTPUT 1

```
1: 2 **
2: 0
3: 3 ***
4: 0
5: 0
6: 1 *
mode=3
```

INPUT 2

```
0
```

OUTPUT 2

```
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
mode=1
```

REFERENCE SOLUTION

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] counts = new int[7];
        for (int i = 0; i < n; i++) counts[in.nextInt()]++;
        int mode = 1;
        for (int face = 1; face ⩽ 6; face++) {
            System.out.println(face + ": " + counts[face] + " " + "*".repeat(counts[
            if (counts[face] > counts[mode]) mode = face;
        }
        System.out.println("mode=" + mode);
    }
}
```

QUIZ

1. What does `new String[3]` contain immediately after creation?

A. Three empty strings

B. Three nulls

C. Nothing — it must be initialised first

D. Three spaces

2. `int[] a = {1, 2, 3}; a[3]` results in…

A. 0

B. `ArrayIndexOutOfBoundsException`

C. 3

D. null

3. Which is correct?

A. `a.length()`

B. `a.length`

C. `a.size()`

D. `length(a)`

4. `int[] a = {1}; int[] b = {1}; a == b` is…

A. `true`

B. `false` — different array objects; use `Arrays.equals`

C. Compile error

D. `true` only for primitives

5. Can an array's length change after creation?

A. Yes, with `a.length = 10`

B. Yes, with `a.resize(10)`

C. No — create a new array and copy

D. Only for object arrays

ANSWER KEY · ARRAYS — FIXED-SIZE, TYPED, ZERO-BASED

1. B Reference-type arrays default every slot to `null`. Calling a method on an element before assigning it throws `NullPointerException`.

2. B Valid indices are 0 through `length - 1`. Every access is bounds-checked at run time.

3. B An array's length is a `final` field. `String.length()` and `List.size()` are methods — a favourite trick question.

4. B Arrays are objects and `==` compares references. `a.equals(b)` is the same identity test; `Arrays.equals(a, b)` compares contents.

5. C Length is fixed at `new`. `Arrays.copyOf(a, newLength)` creates the bigger array; `ArrayList` automates this.

LESSON 2 · 12 MIN

Two-dimensional and jagged arrays

Java has no true multi-dimensional arrays. `int[][]` is an *array of arrays*: an outer array whose elements are references to inner arrays, each of which can be any length — or null. That model explains everything about how grids are created, iterated, copied and sized, and it is what makes "jagged" arrays possible.

Creating a grid

```
int[][] grid = new int[3][4];        // 3 rows, each a new int[4] of zeros
grid[1][2] = 7;                      // row 1, column 2
grid.length                          // 3 — the number of rows
grid[0].length                       // 4 — the length of row 0
int[][] literal = {
    {1, 2, 3},
    {4, 5, 6},
};                                   // 2 × 3, trailing comma allowed
```

`new int[3][4]` allocates **four** objects: the outer `int[3]` and three `int[4]`s. Row-major is the convention — `grid[row][col]` — but nothing enforces it; it is simply the order of the indices.

Iterating

```
for (int r = 0; r < grid.length; r++) {
    for (int c = 0; c < grid[r].length; c++) {     // grid[r].length, not gri
        System.out.print(grid[r][c] + " ");
    }
    System.out.println();
}

for (int[] row : grid) {          // for-each over rows
    for (int v : row) { … }
}
```

Iterating row by row (outer loop over rows) walks memory in the order it is laid out and is cache-friendly; column-first loops over a large grid are measurably slower.

Jagged arrays

Because rows are independent arrays, they need not be the same length:

```
int[][] triangle = new int[4][];        // 4 rows, all NULL
for (int r = 0; r < 4; r++) {
    triangle[r] = new int[r + 1];        // row r has r + 1 slots
}
// triangle[3].length == 4, triangle[0].length == 1

int[][] ragged = {{1}, {2, 3}, {}, {4, 5, 6}};   // literal jagged array; row
```

`new int[4][]` with the second dimension omitted creates only the outer array; each row is `null` until assigned. Pascal's triangle, adjacency lists, and "rows read from input of varying width" are natural jagged arrays.

Consequences: `grid[0].length` is not "the width" unless you know the grid is rectangular; a row may be `null` (NPE on `grid[r].length`); and `grid[r] = grid[0]` makes two rows the *same* array.

Rows are references

```
int[][] a = new int[2][2];
int[] row = a[0];
row[1] = 9;                          // a[0][1] is now 9 — row IS a[0]

int[][] b = a.clone();               // shallow: b is a new outer array, but b[0]
b[0][0] = 5;                         // a[0][0] is also 5
```

`clone()`, `Arrays.copyOf` and `System.arraycopy` on the outer array copy the **row references**, not the rows. A deep copy is a loop: `for (r …) b[r] = a[r].clone();` (or `Arrays.stream(a).map(int[]::clone).toArray(int[][]::new)`).

The same applies to printing and comparing: `Arrays.toString(grid)` prints `[[I@…, [I@…]`; use `Arrays.deepToString(grid)`. `Arrays.equals(a, b)` compares row references;

`Arrays.deepEquals(a, b)` compares contents.

Three and more dimensions

`int[][][] cube = new int[2][3][4];` — an array of 2 arrays of 3 arrays of 4 ints. Same rules, one more level. Beyond three dimensions, consider a flat array with computed indices.

Flat arrays as grids

For performance-critical grids (image processing, dynamic programming tables), a single `int[rows * cols]` indexed as `a[r * cols + c]` avoids the per-row indirection and the extra objects, and keeps everything in one contiguous block. It is less readable; use it when measurement says so.

Reading a grid from input

```
int rows = in.nextInt(), cols = in.nextInt();
int[][] g = new int[rows][cols];
for (int r = 0; r < rows; r++)
    for (int c = 0; c < cols; c++)
        g[r][c] = in.nextInt();
```

For a character grid (mazes, boards): `char[][] board = new char[rows][]; for (r …) board[r] = in.next().toCharArray();` — each line becomes a row.

Common patterns

- **Transpose** (rectangular): `t[c][r] = g[r][c]` with `t = new int[cols][rows]`.
- **Neighbours**: check bounds `0 ⩽ nr && nr < rows && 0 ⩽ nc && nc < cols` before `g[nr][nc]`; direction arrays `int[] dr = {-1, 1, 0, 0}, dc = {0, 0, -1, 1}` avoid four copies of the check.
- **Row sums** / **column sums**: two nested loops; column sums accumulate into `colSum[c]` while walking rows.
- **Diagonals**: main diagonal `g[i][i]`; anti-diagonal `g[i][n - 1 - i]`.

Interview angle

"Is `int[][]` a matrix?" An array of arrays; rows are separate objects and may differ in length.

"How many objects does `new int[3][4]` create?" Four.

"Why does `clone()` on a 2D array not copy the contents?" It copies the outer array's references to the same rows.

"How do you print a 2D array?" `Arrays.deepToString`.

Key takeaways

`int[][]` is an array of row arrays; `grid.length` is rows, `grid[r].length` is that row's length.

`new int[n][]` leaves rows null; rows may be jagged and may be shared.

Copying, comparing and printing the outer array is shallow — use `deepToString`, `deepEquals`, and clone each row.

Iterate row-major; bounds-check neighbours with direction arrays.

EXERCISE 1

Row and column sums

Read integers `R` and `C`, then an `R × C` grid of integers. Print `R` lines `row <i>=<sum>`, then `C` lines `col <j>=<sum>`, then `diag=<sum of cells where row == col>`.

Example: `2 3` then `1 2 3` / `4 5 6` →

```
row 0=6
row 1=15
col 0=5
col 1=7
col 2=9
diag=6
```

INPUT 1

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

OUTPUT 1

```
row 0=6
row 1=15
col 0=5
col 1=7
col 2=9
diag=6
```

INPUT 2

```
3 1
5
-5
10
```

OUTPUT 2

```
row 0=5
row 1=-5
row 2=10
col 0=10
diag=5
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int r = in.nextInt(), c = in.nextInt();
        long[][] g = new long[r][c];
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++)
                g[i][j] = in.nextLong();

        StringBuilder out = new StringBuilder();
        long[] col = new long[c];
        long diag = 0;
        for (int i = 0; i < r; i++) {
            long row = 0;
            for (int j = 0; j < c; j++) {
                row += g[i][j];
                col[j] += g[i][j];
                if (i == j) diag += g[i][j];
            }
            out.append("row ").append(i).append('=').append(row).append('\n');
        }
        for (int j = 0; j < c; j++) out.append("col ").append(j).append('=').append(
        out.append("diag=").append(diag).append('\n');
        System.out.print(out);
    }
}
```

EXERCISE 2

Transpose and trace

Read `r` and `c` and an `r × c` matrix. Print its transpose (`c` rows of `r` values, space-separated), then `trace=<sum of m[i][i] for i < min(r, c)>` and `symmetric=<true if r == c and the matrix equals its transpose>`.

Example: `2 3` then `1 2 3` / `4 5 6` →

```
1 4
2 5
3 6
trace=6
symmetric=false
```

INPUT 1

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

OUTPUT 1

```
1 4
2 5
3 6
trace=6
symmetric=false
```

INPUT 2

```
2 2
1 7
7 1
```

OUTPUT 2

```
1 7
7 1
trace=2
symmetric=true
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int r = in.nextInt(), c = in.nextInt();
        int[][] m = new int[r][c];
        for (int i = 0; i < r; i++) for (int j = 0; j < c; j++) m[i][j] = in.nextInt
        int[][] t = new int[c][r];
        for (int i = 0; i < r; i++) for (int j = 0; j < c; j++) t[j][i] = m[i][j];
        for (int[] row : t) {
            StringBuilder sb = new StringBuilder();
            for (int x : row) sb.append(sb.length() > 0 ? " " : "").append(x);
            System.out.println(sb);
        }
        long trace = 0;
        for (int i = 0; i < Math.min(r, c); i++) trace += m[i][i];
        System.out.println("trace=" + trace);
        System.out.println("symmetric=" + (r == c && Arrays.deepEquals(m, t)));
    }
}
```

QUIZ

1. How many array objects does `new int[3][4]` allocate?

A. 1

B. 3

C. 4

D. 12

2. `int[][] j = new int[3][];` — what is `j[0]`?

A. An empty array

B. `null`

C. `new int[3]`

D. Compile error

3. `int[][] b = a.clone(); b[0][0] = 9;` — effect on `a`?

A. None

B. `a[0][0]` is also 9 — the rows are shared

C. `a` is null

D. Compile error

4. How do you print a 2D array's contents?

A. `System.out.println(grid)`

B. `Arrays.toString(grid)`

C. `Arrays.deepToString(grid)`

D. `grid.toString()`

5. The width of row `r` in a possibly jagged `int[][] g` is…

A. `g.length`

B. `g[0].length`

C. `g[r].length`

D. `g.width`

ANSWER KEY · TWO-DIMENSIONAL AND JAGGED ARRAYS

1. C One outer `int[3]` plus three inner `int[4]` rows. There is no single 12-element block.

2. B Omitting the inner dimension creates only the outer array; each row must be created separately, and rows may have different lengths.

3. B `clone` copies the outer array's references. Deep-copy by cloning each row.

4. C `toString` on the outer array prints row references like `[I@1b6d`; `deepToString` recurses into the rows.

5. C Each row is its own array with its own length. `g[0].length` is only the width if the grid is rectangular.

LESSON 3 · 13 MIN

The Arrays class and System.arraycopy

Arrays themselves have one field and no useful methods. Everything you actually do with them — sort, search, fill, copy, compare, print, convert — lives in `java.util.Arrays`, a class of static helpers, plus one native method in `System`. Knowing this toolkit saves writing loops and, more importantly, saves writing *wrong* loops.

Printing and comparing

```
int[] a = {3, 1, 2};
Arrays.toString(a)            // "[3, 1, 2]"
Arrays.deepToString(grid)     // nested arrays
Arrays.equals(a, b)           // same length and equal elements, in order
Arrays.deepEquals(g1, g2)     // nested
Arrays.hashCode(a)            // content-based hash (for using arrays as map k
Arrays.compare(a, b)          // lexicographic, Java 9+
Arrays.mismatch(a, b)         // index of the first difference, or -1
```

`a.equals(b)` and `a.hashCode()` are `Object`'s — identity. This is why an `int[]` as a `HashMap` key never matches another array with the same contents; use a `List` or a record instead.

Sorting

```
Arrays.sort(a);                          // ascending, in place; primitives use q
Arrays.sort(a, 1, 4);                    // a range [from, to)
Arrays.sort(strings);                    // natural order (compareTo); objects us
Arrays.sort(strings, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(people, Comparator.comparing(Person::age));      // any Comparator
Arrays.sort(boxed, Collections.reverseOrder());              // Integer[] desc
Arrays.parallelSort(big);                // multi-threaded for large arrays
```

There is **no descending sort for a primitive array** — `Arrays.sort(int[], Comparator)` does not exist because comparators take objects. Sort ascending and read backwards, negate

the values, or box to `Integer[]`. Sorting is stable for objects (equal elements keep their order) and, being in place on primitives, unstable in the sense that does not matter for values.

For a 2D array by a column: `Arrays.sort(rows, (x, y) → Integer.compare(x[0], y[0]));` — `int[][]` is an array of objects, so a comparator applies.

Searching

```
int i = Arrays.binarySearch(sorted, 7);     // index, or -(insertionPoint) - 1
```

On an unsorted array the result is meaningless. A negative return encodes where the value *would* go: `-(r + 1)` is the insertion point — useful for "first element ≥ x". For a linear search on an unsorted array there is no helper; write the loop or `Arrays.asList(objs).indexOf(x)` for object arrays.

Filling and creating

```
Arrays.fill(a, -1);                  // every element
Arrays.fill(a, 2, 5, 0);             // a range
Arrays.setAll(a, i → i * i);         // compute each element from its index (
int[] sq = IntStream.range(0, 5).map(i → i * i).toArray();   // 0 1 4 9 16
```

`Arrays.fill` on a 2D array fills the *outer* array with the same row reference — `Arrays.fill(grid, new int[4])` makes every row the same object. Fill each row: `for (int[] row : grid) Arrays.fill(row, -1);`.

Copying

```
int[] copy = Arrays.copyOf(a, a.length);      // a full copy
int[] longer = Arrays.copyOf(a, 10);          // copied and padded with zeros
int[] part = Arrays.copyOfRange(a, 1, 3);     // [from, to)
int[] c = a.clone();                          // also a full copy
System.arraycopy(a, 0, dest, 2, 3);           // (src, srcPos, dest, destPos,
```

`System.arraycopy` is the primitive the others are built on — a native, bulk memory copy that handles overlapping ranges correctly (so `System.arraycopy(a, 0, a, 1, n - 1)` shifts right in place). It throws `ArrayIndexOutOfBoundsException` if either range is out of bounds and `ArrayStoreException` for a type mismatch. All of these are **shallow**: for object arrays they copy references.

Converting

```
List<Integer> list = Arrays.asList(boxed);    // FIXED-SIZE view over the arra
List<String> l2 = Arrays.asList("a", "b");    // varargs — same fixed-size vie
List<Integer> real = new ArrayList<>(Arrays.asList(boxed));   // a growable co
List<int[]> oops = Arrays.asList(intArray);   // an int[] is ONE element — pri
List<Integer> ok = Arrays.stream(intArray).boxed().toList();  // the right way
Integer[] back = list.toArray(new Integer[0]);
IntStream s = Arrays.stream(a);               // sum(), max(), filter(), … (Ma
int sum = Arrays.stream(a).sum();
int max = Arrays.stream(a).max().getAsInt();
```

Two traps in `asList`: it is a *view* backed by the array (changes go both ways; size cannot change), and it does not box primitive arrays — `Arrays.asList(new int[]{1,2})` is a list with one element, the array. `List.of(...)` (Java 9) is the immutable alternative and rejects nulls.

Streams over arrays (a preview)

`Arrays.stream(a)` opens the door to `sum`, `average`, `max`, `filter`, `map`, `sorted`, `distinct`, `anyMatch` — one-liners for many loop patterns. Module 15 covers streams; until then, `Arrays.stream(a).sum()` and `.max().getAsInt()` are safe to use as vocabulary.

Utility methods you will write yourself

The JDK has no `Arrays.reverse`, `Arrays.max` for `int[]` (outside streams), `Arrays.indexOf`, `Arrays.contains` or `Arrays.swap`. They are five-line helpers; in interviews write them, in projects keep them in a small util class or use a library like Guava/Commons.

Interview angle

"How do you sort an `int[]` descending?" No comparator on primitives: sort ascending and reverse, or box.

"`Arrays.asList(new int[]{1,2}).size()`?" 1.

"What does `Arrays.binarySearch` return when the value is absent?" `-(insertionPoint) - 1`.

"Difference between `Arrays.copyOf` and `System.arraycopy`?" `copyOf` allocates a new array of a given length; `arraycopy` copies into an existing one and can overlap.

"Can you `add` to `Arrays.asList`?" No — `UnsupportedOperationException`; wrap in `new ArrayList<>(…)`.

Key takeaways

`Arrays.toString/equals/hashCode` for contents; the array's own methods are identity-based.

`Arrays.sort` (no descending for primitives), `binarySearch` (sorted only, negative = insertion point), `fill`, `setAll`.

`copyOf` / `copyOfRange` / `clone` / `System.arraycopy` — all shallow; `arraycopy` handles overlap.

`Arrays.asList` is a fixed-size view and does not box primitives; `Arrays.stream` for `int[]`.

EXERCISE 1

Sort, search, slice

Read an integer `n`, then `n` integers, then an integer `q` and `q` query values. Sort the array with `Arrays.sort` and print it with `Arrays.toString`. For each query print `<value> at <index>` if `Arrays.binarySearch` finds it, otherwise `<value> insert at <position>` where `position` is decoded from the negative return value. Finally print the middle third of the sorted array (`Arrays.copyOfRange` from `n/3` to `2n/3`) with `Arrays.toString`.

Example: `6` then `5 3 9 1 7 3`, `2` then `7 4` →

```
[1, 3, 3, 5, 7, 9]
7 at 4
4 insert at 3
[3, 5]
```

INPUT 1

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

OUTPUT 1

```
[1, 3, 3, 5, 7, 9]
7 at 4
4 insert at 3
[3, 5]
```

INPUT 2

```
3
2 2 2
1
0
```

OUTPUT 2

```
[2, 2, 2]
0 insert at 0
[2]
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        Arrays.sort(a);
        StringBuilder out = new StringBuilder();
        out.append(Arrays.toString(a)).append('\n');
        int q = in.nextInt();
        for (int i = 0; i < q; i++) {
            int v = in.nextInt();
            int r = Arrays.binarySearch(a, v);
            if (r >= 0) out.append(v).append(" at ").append(r).append('\n');
            else out.append(v).append(" insert at ").append(-(r + 1)).append('\n');
        }
        int[] middle = Arrays.copyOfRange(a, n / 3, 2 * n / 3);
        out.append(Arrays.toString(middle)).append('\n');
        System.out.print(out);
    }
}
```

EXERCISE 2

Binary search report

Read an integer `n`, `n` integers, then `q` and `q` query values. Sort the array with `Arrays.sort` and print `sorted=<Arrays.toString>`. For each query print `<v>: found at <index>` when

`Arrays.binarySearch` finds it, or `<v>: insert at <position>` decoded from the negative result (`-(insertion point) - 1`). Finally print `equalToCopy=<Arrays.equals(sorted, Arrays.copyOf(sorted, n))>`.

Example: `4` then `9 2 7 4`, `3` then `7 5 10` →

```
sorted=[2, 4, 7, 9]
7: found at 2
5: insert at 2
10: insert at 4
equalToCopy=true
```

INPUT 1

```
4
9 2 7 4
3
7 5 10
```

OUTPUT 1

```
sorted=[2, 4, 7, 9]
7: found at 2
5: insert at 2
10: insert at 4
equalToCopy=true
```

INPUT 2

```
1
5
2
5 0
```

OUTPUT 2

```
sorted=[5]
5: found at 0
0: insert at 0
equalToCopy=true
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        Arrays.sort(a);
        System.out.println("sorted=" + Arrays.toString(a));
        int q = in.nextInt();
        for (int i = 0; i < q; i++) {
            int v = in.nextInt();
            int at = Arrays.binarySearch(a, v);
            if (at >= 0) System.out.println(v + ": found at " + at);
            else System.out.println(v + ": insert at " + (-at - 1));
        }
        System.out.println("equalToCopy=" + Arrays.equals(a, Arrays.copyOf(a, n)));
    }
}
```

QUIZ

1. `Arrays.binarySearch(a, x)` on an **unsorted** array returns…

A. The correct index

B. −1

C. An unspecified result

D. Throws

2. How do you sort an `int[]` in descending order?

A. `Arrays.sort(a, Collections.reverseOrder())`

B. `Arrays.sort(a, Comparator.reverseOrder())`

C. Sort ascending and read it backwards (or box to `Integer[]`)

D. `Arrays.sortDesc(a)`

3. `Arrays.asList(new int[]{1, 2, 3}).size()` is…

A. 3

B. 1 — the `int[]` is a single element

C. 0

D. Compile error

4. `List<String> l = Arrays.asList("a", "b"); l.add("c");` throws…

A. Nothing

B. `UnsupportedOperationException` — the list is a fixed-size view

C. `IndexOutOfBoundsException`

D. `NullPointerException`

5. `System.arraycopy(a, 0, a, 1, a.length - 1)` does what?

A. Throws because source and destination are the same

B. Shifts every element one position to the right, in place, correctly

C. Corrupts the array

D. Copies nothing

ANSWER KEY · THE ARRAYS CLASS AND SYSTEM.ARRAYCOPY

1. C Binary search assumes sortedness; on unsorted input the answer is meaningless. Sort first.

2. C Comparators apply to object arrays only. For primitives, reverse after sorting, negate values, or use `Integer[]`.

3. B `asList` takes `T...`; a primitive array is not boxed, so it becomes one element of type `int[]`. Use `Arrays.stream(a).boxed()`.

4. B `asList` wraps the array; `set` works but the size cannot change. Wrap in `new ArrayList<>(…)` to get a growable list.

5. B `arraycopy` handles overlapping ranges as if through a temporary buffer. The last element is overwritten; the first is left as is.

LESSON 4 · 12 MIN

Aliasing, shallow copies and defensive copies

Because an array variable is a reference, two variables can name one array — *aliasing* — and a "copy" can be a copy of the references rather than of the things they point to — a *shallow copy*. Most array bugs that are not off-by-one are one of these two. This lesson makes the distinction concrete and shows the defensive habits that keep other code from changing your arrays behind your back.

Aliasing

```
int[] a = {1, 2, 3};
int[] b = a;              // b is ANOTHER NAME for the same array
b[0] = 99;
System.out.println(a[0]); // 99
a == b                    // true
```

Assignment copies the reference. No new array exists; `a` and `b` are two arrows to one object. Passing an array to a method (Module 5), storing it in a field, or putting it in a collection all create aliases. Sometimes that is the point (a method that fills a buffer); often it is a surprise (a "saved snapshot" that changes when the original does).

Shallow copy

```
int[] c = a.clone();      // a NEW array with the same values
c[0] = 0;
a[0]                      // still 99 — independent
```

For an array of **primitives**, a copy of the values is a complete copy: there is nothing deeper. For an array of **references**, a copy of the values is a copy of the *references* — the elements themselves are shared:

```
StringBuilder[] sbs = { new StringBuilder("x"), new StringBuilder("y") };
StringBuilder[] copy = sbs.clone();
copy[0].append("!");      // modifies the SAME StringBuilder that sbs[0] poin
sbs[0]                    // "x!"
copy[0] = new StringBuilder("z");   // rebinding a slot in copy does NOT affe
```

`clone()`, `Arrays.copyOf`, `Arrays.copyOfRange`, `System.arraycopy` and `List.toArray` are all shallow. This is not a defect — it is what "copy the array" means — but you must know which level you copied.

Deep copy

A deep copy duplicates the elements too. Java has no general deep-copy operation because it cannot know how deep to go or how to copy arbitrary objects; you write it for the shape you have:

```
int[][] deep = new int[grid.length][];
for (int r = 0; r < grid.length; r++) deep[r] = grid[r].clone();

Point[] copy = new Point[pts.length];
for (int i = 0; i < pts.length; i++) copy[i] = new Point(pts[i].x, pts[i].y);
```

If the elements are **immutable** (`String`, `Integer`, records with immutable fields), a shallow copy is as good as a deep one: nobody can change the shared elements, so sharing them is harmless. This is a major practical benefit of immutability and the reason `String[]` copies are never a worry.

Defensive copies

A class that stores an array a caller handed it, or returns an internal array, has given away control of its state:

```
public class Scores {
    private final int[] values;

    public Scores(int[] values) {
        this.values = values;               // ALIAS: the caller can change our
    }
    public int[] getValues() {
        return values;                      // ALIAS: any caller can change our
    }
}
```

The fix is to copy on the way in and on the way out:

```
    public Scores(int[] values) {
        this.values = values.clone();       // our own copy
    }
    public int[] getValues() {
        return values.clone();              // a copy they may do anything with
    }
```

Effective Java calls this "make defensive copies when needed" and it applies to any mutable object received or exposed — arrays, `Date`, collections, `StringBuilder`. For collections the alternative is an unmodifiable view (`Collections.unmodifiableList`, `List.copyOf`); arrays have no read-only view, so a copy is the only option. When an array is large and copies would hurt, document the aliasing explicitly and keep the class package-private.

`final int[] values` does not help: `final` freezes the reference, not the elements. `values[0] = 5` is legal on a final array.

Returning arrays

Prefer returning a fresh array (or a `List`, or a stream) over exposing a field. Returning an **empty array** rather than `null` for "no results" lets callers loop without a null check: `return new int[0];` — the JVM shares zero-length arrays cheaply, and `EMPTY = new int[0]` as a constant is a common idiom.

Comparing after copying

`Arrays.equals(a, a.clone())` is true; `a == a.clone()` is false. `Arrays.equals` on object arrays calls `equals` on each element pair — deep enough for strings and records, one level only for nested arrays (`deepEquals` for those).

Where aliasing is intended

A method that fills or sorts a caller's array in place (`Arrays.sort(a)`, `System.arraycopy`, your `reverse(a)`).

A buffer shared between a producer and a consumer.

A cache of rows in a jagged structure.

Name such methods so the mutation is obvious (`sortInPlace`, `fillWith`), document it, and never *also* return the array as if it were new.

Interview angle

"What does `clone()` do on an array of objects?" A new array whose slots point at the same objects.

"Why return `values.clone()` from a getter?" To keep callers from mutating internal state.

"Does `final int[] a` prevent `a[0] = 1`?" No.

"How do you deep-copy a 2D array?" Clone each row.

Key takeaways

Assignment aliases; `clone` / `copyOf` / `arraycopy` make shallow copies — elements of object arrays are shared.

Deep copies are written by hand per shape; immutable elements make shallow copies safe.

Copy on the way in and out of a class that holds an array (`final` does not protect contents).

Return empty arrays, not null; name in-place mutators honestly.

EXERCISE 1

Snapshot versus alias

Read an integer `n` and `n` integers into an array `live`. Make `alias = live` and `snapshot = live.clone()`. Then read an integer `m` and `m` updates `index value`, applying each as `live[index] = value`. Finally print three lines: `live=<Arrays.toString(live)>`, `alias=<…>`, `snapshot=<…>`, and a fourth line `same=<alias == live> equal=<Arrays.equals(live, snapshot)>`.

Example: `3` then `1 2 3`, `1` then `0 9` →

```
live=[9, 2, 3]
alias=[9, 2, 3]
snapshot=[1, 2, 3]
same=true equal=false
```

INPUT 1

```
3
1 2 3
1
0 9
```

OUTPUT 1

```
live=[9, 2, 3]
alias=[9, 2, 3]
snapshot=[1, 2, 3]
same=true equal=false
```

INPUT 2

```
2
4 4
0
```

OUTPUT 2

```
live=[4, 4]
alias=[4, 4]
snapshot=[4, 4]
same=true equal=true
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] live = new int[n];
        for (int i = 0; i < n; i++) live[i] = in.nextInt();
        int[] alias = live;
        int[] snapshot = live.clone();
        int m = in.nextInt();
        for (int i = 0; i < m; i++) {
            int index = in.nextInt();
            int value = in.nextInt();
            live[index] = value;
        }
        System.out.println("live=" + Arrays.toString(live));
        System.out.println("alias=" + Arrays.toString(alias));
        System.out.println("snapshot=" + Arrays.toString(snapshot));
        System.out.println("same=" + (alias == live) + " equal=" + Arrays.equals(liv
    }
}
```

EXERCISE 2

Shallow grid, deep grid

Read `r`, `c` and an `r × c` grid. Make `shallow = grid.clone()` (a new outer array whose rows are the **same** row arrays) and `deep` — a fresh array with every row cloned. Set `grid[0][0] = 99`. Print `shallow[0][0]=<…>` and `deep[0][0]=<…>`, then `sharesRows=<shallow[0] == grid[0]>` and `deepSharesRows=<deep[0] == grid[0]>`, then `deepEquals=<Arrays.deepEquals(grid, deep)>`.

Example: `2 2` then `1 2` / `3 4` →

```
shallow[0][0]=99
deep[0][0]=1
sharesRows=true
deepSharesRows=false
deepEquals=false
```

INPUT 1

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

OUTPUT 1

```
shallow[0][0]=99
deep[0][0]=1
sharesRows=true
deepSharesRows=false
deepEquals=false
```

INPUT 2

```
1 1
99
```

OUTPUT 2

```
shallow[0][0]=99
deep[0][0]=99
sharesRows=true
deepSharesRows=false
deepEquals=true
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int r = in.nextInt(), c = in.nextInt();
        int[][] grid = new int[r][c];
        for (int i = 0; i < r; i++) for (int j = 0; j < c; j++) grid[i][j] = in.next
        int[][] shallow = grid.clone();
        int[][] deep = new int[r][];
        for (int i = 0; i < r; i++) deep[i] = grid[i].clone();
        grid[0][0] = 99;
        System.out.println("shallow[0][0]=" + shallow[0][0]);
        System.out.println("deep[0][0]=" + deep[0][0]);
        System.out.println("sharesRows=" + (shallow[0] == grid[0]));
        System.out.println("deepSharesRows=" + (deep[0] == grid[0]));
        System.out.println("deepEquals=" + Arrays.deepEquals(grid, deep));
    }
}
```

QUIZ

1. `int[] b = a;` followed by `b[0] = 5` changes `a[0]` because…

A. Arrays are copied lazily

B. `a` and `b` are two references to one array

C. `int[]` is a primitive

D. It does not change `a[0]`

2. `StringBuilder[] c = sbs.clone(); c[0].append("!");` affects `sbs[0]`?

A. No

B. Yes — the copy is shallow; both slots refer to the same builder

C. Only if `sbs` is final

D. Compile error

3. Why does a getter return `values.clone()` instead of `values`?

A. Performance

B. So callers cannot mutate the object's internal array

C. Because arrays cannot be returned

D. To sort it

4. Does `final int[] a` prevent `a[0] = 1`?

A. Yes

B. No — `final` fixes the reference, not the elements

C. Only for primitives

D. Only outside the class

5. A shallow copy of a `String[]` is safe to share because…

A. Strings are primitives

B. Strings are immutable, so shared elements cannot be changed

C. `String[]` is always deep-copied

D. It is not safe

ANSWER KEY · ALIASING, SHALLOW COPIES AND DEFENSIVE COPIES

1. B Assignment copies the reference. Only `clone` / `copyOf` create a second array.

2. B Copying an object array copies references. The elements are shared until you copy them individually.

3. B This is a defensive copy. Returning the field would let any caller change the object's state from outside.

4. B `final` means `a` cannot be reassigned to another array. The array's contents remain mutable.

5. B Sharing immutable elements is harmless. Shallow copies are a problem only for mutable element types.

LESSON 5 · 15 MIN

Array patterns — reverse, rotate, count, prefix sums

A handful of array manipulations appear in an outsized share of interview problems and real code: reversing in place, rotating, counting frequencies with an index-as-key array, prefix sums for range queries, two-pointer partitioning, and matrix walks. Each has a canonical, boundary-safe form. This lesson gives you those forms so you write them without thinking — and recognise them inside bigger problems.

Swap and reverse in place

```
static void swap(int[] a, int i, int j) { int t = a[i]; a[i] = a[j]; a[j] = t;

static void reverse(int[] a) {
    for (int i = 0, j = a.length - 1; i < j; i++, j--) swap(a, i, j);
}
```

Two indices walking inward; the loop ends when they meet or cross (`i < j`). Works for odd and even lengths and for empty arrays. Reversing a *range* `[from, to)` is the same with `i = from, j = to - 1` — the building block of rotation.

Rotate by k

Rotating right by `k` (the last `k` elements move to the front) in O(n) time and O(1) space is the three-reversal trick:

```
static void rotateRight(int[] a, int k) {
    int n = a.length;
    if (n == 0) return;
    k = ((k % n) + n) % n;          // normalise: handles k > n and negative
    reverse(a, 0, n);               // whole
    reverse(a, 0, k);               // first k
    reverse(a, k, n);               // rest
}
```

`[1,2,3,4,5]` right by 2: reverse all → `[5,4,3,2,1]`, reverse first 2 → `[4,5,3,2,1]`, reverse rest → `[4,5,1,2,3]`. The alternative — a temporary array and `a[(i + k) % n] = old[i]` — is O(n) space and often clearer; know both.

Counting with an index-as-key array

When the values are small integers or characters, an array indexed by value is the fastest frequency table:

```
int[] freq = new int[26];
for (char c : word.toCharArray()) freq[c - 'a']++;      // lowercase letters

int[] counts = new int[101];
for (int score : scores) counts[score]++;               // scores 0..100
```

Anagram checks, "first non-repeating character", counting sort, histogram — all this pattern. Guard the index range: `c - 'a'` for an uppercase letter is negative and throws. For arbitrary values use a `HashMap` (Module 14).

Prefix sums

Precompute running totals once; then any range sum is one subtraction:

```
long[] prefix = new long[n + 1];                    // prefix[0] = 0
for (int i = 0; i < n; i++) prefix[i + 1] = prefix[i] + a[i];

long sum(int from, int to) {                        // sum of a[from..to), O(1)
    return prefix[to] - prefix[from];
}
```

The extra leading zero is what makes `from = 0` work without a special case. Use `long` — sums overflow before individual values do. Prefix sums answer many range queries in O(1) after O(n) setup, and the 2D version (`P[r][c]` = sum of the rectangle above-left) answers rectangle sums the same way. Difference arrays are the inverse trick for range *updates*.

Two-pointer partition

Move elements matching a condition to the front, keeping order, in one pass:

```
int write = 0;
for (int read = 0; read < a.length; read++) {
    if (a[read] != 0) a[write++] = a[read];     // keep non-zeros
}
while (write < a.length) a[write++] = 0;        // fill the tail
```

"Remove duplicates from a sorted array", "move zeros to the end", and "filter in place" are all this read/write two-pointer. The write index is also the new logical length.

Max / min / second max in one pass

```
int max = Integer.MIN_VALUE, second = Integer.MIN_VALUE;
for (int v : a) {
    if (v > max) { second = max; max = v; }
    else if (v > second && v != max) second = v;
}
```

Seeded from the extreme; "second largest distinct" needs the `v != max` guard. For the index of the max, track it alongside (Module 4's best-so-far).

Kadane: maximum subarray sum

```
long best = a[0], current = a[0];
for (int i = 1; i < a.length; i++) {
    current = Math.max(a[i], current + a[i]);    // extend the run or start fr
    best = Math.max(best, current);
}
```

Linear time, constant space, and the canonical "dynamic programming in one variable" example. Seeded from `a[0]` so all-negative arrays give the largest single element rather than 0.

Matrix walks

```
// transpose (square, in place): swap above the diagonal
for (int r = 0; r < n; r++)
    for (int c = r + 1; c < n; c++) swap(m, r, c);   // m[r][c] ⟷ m[c][r]

// rotate 90° clockwise: transpose, then reverse each row
// spiral: four bounds (top, bottom, left, right) shrinking inward
// neighbours: dr/dc direction arrays with a bounds check
```

`c = r + 1` in the transpose is the pairs pattern from Module 4 — each pair swapped once.

Searching a sorted array

```
int lo = 0, hi = a.length - 1;
while (lo ⩽ hi) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == target) return mid;
    if (a[mid] < target) lo = mid + 1; else hi = mid - 1;
}
return -1;                      // or: lo is the insertion point
```

The iterative form of Module 5's recursive search; `lo` after the loop is where the target would be inserted — the basis of "first element ≥ x" (lower bound) variants. `Arrays.binarySearch` does this for you when a plain "is it there" suffices.

Boundary checklist for array code

- Empty array: does the code handle `length == 0` (seeding from `a[0]` throws)?
- One element: two-pointer loops with `i < j` should do nothing.
- All equal, all negative, already sorted, reversed: the usual edge inputs.
- `k` larger than `n` or negative in rotations: normalise with `((k % n) + n) % n`.
- Sums: `long`.

Interview angle

"Rotate an array in place in O(1) extra space?" Three reversals.

"Count character frequencies without a map?" `int[26]` indexed by `c - 'a'`.

"Range sum queries fast?" Prefix sums with a leading zero.

"Move zeros to the end preserving order?" Read/write two pointers.

Key takeaways

Reverse with two indices (`i < j`); rotate with three reversals after normalising `k` .

Small-integer values → index-as-key counting arrays.

Prefix sums (`long` , size `n + 1`) turn range sums into subtractions.

Read/write pointers filter in place; Kadane runs in one variable.

Test empty, single, all-equal and all-negative inputs.

EXERCISE 1

Rotate, count, prefix

Read an integer `n` , then `n` integers in the range 0–9, then an integer `k` . Print three lines:

The array rotated **right** by `k` (k may exceed n), using the three-reversal method, space-separated.

`counts=<c0 c1 … c9>` — how many times each digit 0–9 appears in the (original) array, using a counting array.

`prefix=<p1 p2 … pn>` — the running totals of the original array.

Example: `5` then `1 2 3 4 5` , `k = 2` →

```
4 5 1 2 3
counts=0 1 1 1 1 1 0 0 0 0
prefix=1 3 6 10 15
```

INPUT 1

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

OUTPUT 1

```
4 5 1 2 3
counts=0 1 1 1 1 1 0 0 0 0
prefix=1 3 6 10 15
```

INPUT 2

```
4
9 9 0 9
7
```

OUTPUT 2

```
9 0 9 9
counts=1 0 0 0 0 0 0 0 0 3
prefix=9 18 18 27
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static void reverse(int[] a, int from, int to) {
        for (int i = from, j = to - 1; i < j; i++, j--) {
            int t = a[i]; a[i] = a[j]; a[j] = t;
        }
    }

    static String join(int[] a) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < a.length; i++) {
            if (i > 0) sb.append(' ');
            sb.append(a[i]);
        }
        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int k = in.nextInt();

        int[] counts = new int[10];
        long[] prefix = new long[n];
        long run = 0;
        for (int i = 0; i < n; i++) {
            counts[a[i]]++;
            run += a[i];
            prefix[i] = run;
        }

        int[] rotated = a.clone();
        if (n > 0) {
            int r = ((k % n) + n) % n;
            reverse(rotated, 0, n);
            reverse(rotated, 0, r);
            reverse(rotated, r, n);
        }

        System.out.println(join(rotated));
        System.out.println("counts=" + join(counts));
        StringBuilder p = new StringBuilder("prefix=");
        for (int i = 0; i < n; i++) {
            if (i > 0) p.append(' ');
            p.append(prefix[i]);
        }
```

```
        System.out.println(p);
    }
}
```

EXERCISE 2

Two pointers, in place

Read an integer `n` and `n` integers **sorted ascending**. In place, with two indexes and no extra array: (1) remove duplicates so the first `k` slots hold the distinct values — print `distinct=<k> <those k values>`; then read another `m` and `m` integers and (2) move every zero to the end while keeping the order of the non-zeros — print `zerosMoved=<array>`.

Example: `6` then `1 1 2 3 3 3`, then `5` then `0 4 0 5 6` →

```
distinct=3 1 2 3
zerosMoved=[4, 5, 6, 0, 0]
```

INPUT 1

```
6
1 1 2 3 3 3
5
0 4 0 5 6
```

OUTPUT 1

```
distinct=3 1 2 3
zerosMoved=[4, 5, 6, 0, 0]
```

INPUT 2

```
0
3
0 0 0
```

OUTPUT 2

```
distinct=0
zerosMoved=[0, 0, 0]
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int k = 0;
        for (int i = 0; i < n; i++) {
            if (k == 0 || a[i] != a[k - 1]) a[k++] = a[i];
        }
        StringBuilder sb = new StringBuilder("distinct=" + k);
        for (int i = 0; i < k; i++) sb.append(' ').append(a[i]);
        System.out.println(sb);

        int m = in.nextInt();
        int[] b = new int[m];
        for (int i = 0; i < m; i++) b[i] = in.nextInt();
        int write = 0;
        for (int i = 0; i < m; i++) if (b[i] != 0) b[write++] = b[i];
        while (write < m) b[write++] = 0;
        System.out.println("zerosMoved=" + Arrays.toString(b));
    }
}
```

QUIZ

1. Rotating an array right by `k` with three reversals is…

A. O(n) time, O(n) space

B. O(n) time, O(1) extra space

C. O(k·n) time

D. O(n log n)

2. Why does a prefix-sum array usually have `n + 1` entries?

A. To store the total twice

B. A leading 0 makes `prefix[to] - prefix[from]` work when `from == 0`

C. Arrays must be even-length

D. For 1-based indexing only

3. A counting array `int[26]` indexed by `c - 'a'` is appropriate when…

A. Values are arbitrary strings

B. Values are a small, known range such as lowercase letters

C. Values are doubles

D. Never — use a map

4. In the read/write two-pointer filter, the write index after the loop is…

A. Always n

B. The number of kept elements — the new logical length

C. The index of the last removed element

D. Undefined

5. Kadane's algorithm is seeded with `a[0]` rather than 0 so that…

A. It runs faster

B. An all-negative array yields its largest element, not 0

C. It handles empty arrays

D. The loop starts at index 0

ANSWER KEY · ARRAY PATTERNS — REVERSE, ROTATE, COUNT, PREFIX SUMS

1. B Reverse the whole array, then the first k, then the rest — each element moves a constant number of times, in place.

2. B With `prefix[0] = 0`, any range sum is one subtraction with no special case at the start.

3. B Index-as-key counting is the fastest frequency table when the key space is small and dense; a `HashMap` handles the general case.

4. B Every kept element is written at `write++`, so `write` counts the kept elements and marks where the tail begins.

5. B With a 0 seed, `best` would never go below 0 and an all-negative input would report a subarray sum of 0 — an empty subarray, which is usually not allowed.

CHECKPOINT · 22 MIN

Checkpoint — Arrays

This checkpoint covers array creation and indexing, two-dimensional and jagged arrays, the `Arrays` class and `System.arraycopy`, aliasing and copying, and the standard in-place patterns.

How it works. Twelve questions and two programs; 70% on the questions and both programs accepted clears the module.

Before you start, make sure you can answer:

- What `new String[3]` contains, and what `a.length` is (field or method).
- How many objects `new int[3][4]` creates, and what `clone()` copies on it.
- What `Arrays.asList(new int[]{1, 2}).size()` returns.
- Why a getter should return `values.clone()`.
- How to rotate an array in place, and why prefix arrays have `n + 1` entries.

The programs are a matrix rotation and a prefix-sum range query — both classics that reward knowing the pattern.

EXERCISE 1

Rotate a square matrix

Read an integer `n`, then an `n × n` grid of integers. Rotate it 90° clockwise **in place** — transpose (swap across the main diagonal, each pair once), then reverse each row — and print the result, one row per line, values space-separated.

Example: `3` then `1 2 3` / `4 5 6` / `7 8 9` →

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

INPUT 1

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

OUTPUT 1

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

INPUT 2

```
1
5
```

OUTPUT 2

```
5
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[][] m = new int[n][n];
        for (int r = 0; r < n; r++)
            for (int c = 0; c < n; c++)
                m[r][c] = in.nextInt();

        for (int r = 0; r < n; r++) {
            for (int c = r + 1; c < n; c++) {
                int t = m[r][c]; m[r][c] = m[c][r]; m[c][r] = t;
            }
        }
        for (int[] row : m) {
            for (int i = 0, j = n - 1; i < j; i++, j--) {
                int t = row[i]; row[i] = row[j]; row[j] = t;
            }
        }

        StringBuilder out = new StringBuilder();
        for (int[] row : m) {
            for (int c = 0; c < n; c++) {
                if (c > 0) out.append(' ');
                out.append(row[c]);
            }
            out.append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Range sums with a prefix array

Read an integer `n`, then `n` integers (which may be negative and large), then an integer `q` and `q` queries `l r` (0-based, inclusive on both ends). Build a prefix-sum array of `n + 1` `long`s once and answer every query in O(1). Print one sum per line.

Example: `5` then `1 2 3 4 5`, `3` then `0 4`, `1 3`, `2 2` →

```
15
9
3
```

INPUT 1

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

OUTPUT 1

```
15
9
3
```

INPUT 2

```
3
2000000000 2000000000 -1
2
0 1
0 2
```

OUTPUT 2

```
4000000000
3999999999
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        String[] t = new String(System.in.readAllBytes()).trim().split("\\s+");
        int p = 0;
        int n = Integer.parseInt(t[p++]);
        long[] prefix = new long[n + 1];
        for (int i = 0; i < n; i++) prefix[i + 1] = prefix[i] + Long.parseLong(t[p++
        int q = Integer.parseInt(t[p++]);
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < q; i++) {
            int l = Integer.parseInt(t[p++]);
            int r = Integer.parseInt(t[p++]);
            out.append(prefix[r + 1] - prefix[l]).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 3

Pascal's triangle

Read an integer `n` (`1 ≤ n ≤ 30`). Build the first `n` rows of Pascal's triangle as a **jagged** `long[][]` — row `i` has `i + 1` entries, each the sum of the two above it — and print each row space-separated. Then print `rowSum=<sum of the last row>` (which is `2^(n-1)`) and `middle=<the largest value in the last row>`.

Example: `4` →

```
1
1 1
1 2 1
1 3 3 1
rowSum=8
middle=3
```

INPUT 1

```
4
```

OUTPUT 1

```
1
1 1
1 2 1
1 3 3 1
rowSum=8
middle=3
```

INPUT 2

```
1
```

OUTPUT 2

```
1
rowSum=1
middle=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        int n = new Scanner(System.in).nextInt();
        long[][] t = new long[n][];
        for (int i = 0; i < n; i++) {
            t[i] = new long[i + 1];
            t[i][0] = t[i][i] = 1;
            for (int j = 1; j < i; j++) t[i][j] = t[i - 1][j - 1] + t[i - 1][j];
        }
        for (long[] row : t) {
            StringBuilder sb = new StringBuilder();
            for (long x : row) sb.append(sb.length() > 0 ? " " : "").append(x);
            System.out.println(sb);
        }
        long sum = 0, max = 0;
        for (long x : t[n - 1]) { sum += x; max = Math.max(max, x); }
        System.out.println("rowSum=" + sum);
        System.out.println("middle=" + max);
    }
}
```

CHECKPOINT QUESTIONS

1. `int[] a = new int[5];` — the value of `a[4]` is…

A. Undefined

B. 0

C. null

D. 5

2. `for (int i = 0; i ⩽ a.length; i++) sum += a[i];` fails with…

A. A compile error

B. `ArrayIndexOutOfBoundsException` at the last iteration

C. Nothing — it works

D. `NullPointerException`

3. `Object[] o = new String[2]; o[0] = 42;` results in…

A. Compile error

B. `ArrayStoreException` at run time — arrays are covariant but checked on store

C. Success

D. `ClassCastException`

4. Which creates a jagged array with rows of lengths 1, 2 and 3?

A. `new int[3][3]`

B. `new int[3][]` then assigning `new int[r + 1]` to each row

C. `new int[1][2][3]`

D. It is impossible

5. `Arrays.deepEquals` is needed for…

A. Comparing two `int[]`

B. Comparing nested arrays such as `int[][]` by contents

C. Comparing strings

D. Sorting

6. `Arrays.fill(grid, new int[4])` on an `int[4][4]` results in…

A. A zeroed grid of independent rows

B. Every row being the **same** array object

C. A compile error

D. Only row 0 filled

7. `Arrays.binarySearch(sorted, x)` returns −4. Where would `x` be inserted?

A. Index 4

B. Index 3 — the insertion point is `-(r + 1)`

C. Index −4

D. It is present at 4

8. Which produces a growable list from `Integer[] boxed`?

A. `Arrays.asList(boxed)`

B. `new ArrayList<>(Arrays.asList(boxed))`

C. `List.of(boxed)`

D. `boxed.toList()`

9. A constructor stores a caller's `int[]` directly in a field. The risk is…

A. Memory leak

B. The caller can later change the object's state through their own reference

C. Compile error

D. The array is copied twice

10. To sum ranges quickly after O(n) preprocessing, use…

A. A sorted copy

B. A prefix-sum array

C. Binary search

D. A counting array

11. `reverse(a, 0, n); reverse(a, 0, k); reverse(a, k, n);` performs…

A. A left rotation by k

B. A right rotation by k

C. A full reversal

D. Nothing

12. Why prefer `int[]` over `Integer[]` for numeric work?

A. `Integer[]` cannot be sorted

B. Primitives are stored contiguously without per-element objects — smaller and faster

C. `int[]` can grow

D. No difference

ANSWER KEY · CHECKPOINT — ARRAYS

1. B Numeric arrays are zero-filled on creation. Only reference arrays hold nulls.

2. B `a[a.length]` is one past the end. The condition must be `i < a.length`.

3. B The assignment compiles because `String[]` is an `Object[]`; the JVM checks the element's runtime type on every store.

4. B Leaving the inner dimension open and creating each row separately gives rows of any lengths.

5. B `Arrays.equals` on `int[][]` compares the row references. `deepEquals` recurses into the rows.

6. B `fill` stores the one reference in every outer slot. Fill each row separately: `for (int[] row : grid) Arrays.fill(row, v)`.

7. B `-(-4 + 1)` = 3. The encoding lets a negative result carry the position too.

8. B `asList` and `List.of` give fixed-size/immutable lists. Copying into an `ArrayList` gives one you can add to.

9. B The field aliases the caller's array. Store `values.clone()` to own the data.

10. B Prefix sums answer any `[l, r]` sum with one subtraction.

11. B Reverse all, then the first k, then the remainder — the last k elements end up at the front, in order.

12. B An `Integer[]` holds references to heap objects; an `int[]` holds the values. Cache behaviour and memory both favour the primitive array.