Module 20 of 20

Interview idioms

The fast template and complexity budget, the equals/hashCode/Comparable contracts, the thirty pitfalls, the collections idiom sheet, clean solutions under pressure, and the Java theory drill.

Nineteen modules taught the language. This one teaches the round: how a Java interview is actually conducted, what is actually judged, and the small set of habits that turn knowing Java into passing.

It opens with the template — a fast reader, a flushed writer, solve separated from I/O — and the complexity budget that maps "n ≤ 10^5" to the algorithm class you need before you design. The equals/hashCode/Comparable lesson states the contracts every collection depends on and the mistakes that break them. The pitfalls lesson is the thirty-item review pass — overflow, integer division, boxed ==, split's regex, the sort worst case, recursion depth — that catches most "correct but rejected" solutions. The idiom sheet names the fifteen shapes interview problems take and the two-to-six-line Java for each. Clean solutions under pressure is about legibility: structure, names, guard clauses, saying the complexity unprompted. The theory drill is thirty questions with the two-sentence answers.

The exercises are the problems interviewers set — top-k frequent, sliding-window maximum, balanced brackets, spiral order, min-stack, anagram groups, two-sum — written the way the module says to write them. The final checkpoint closes the plan.

CONTENTS

1	The interview template — input, output, limits	12 min
2	equals, hashCode and Comparable — the contracts that make collections work	14 min
3	The pitfalls that fail interviews	14 min
4	The collections idiom sheet	15 min
5	Writing clean solutions under time pressure	12 min
6	The Java theory drill — thirty questions, thirty answers	16 min
✓	Final checkpoint — Interview idioms	32 min

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

LESSON 1 · 12 MIN

The interview template — input, output, limits

Every coding round starts the same way: a problem statement, an input format, a time limit, and forty minutes. The people who do well have stopped thinking about the mechanics — how to read the input, how to print, what `n ≤ 10⁵` means for the algorithm — because they carry a template and a complexity budget in their heads. This lesson gives you both, plus the edge-case checklist to run before you say "done".

The template

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

public class Main {
    static BufferedReader in = new BufferedReader(new InputStreamReader(Syster
    static StringTokenizer st;
    static PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStre

    static String next() throws IOException {
        while (st == null || !st.hasMoreTokens()) {
            String line = in.readLine();
            if (line == null) return null;                // end of input
            st = new StringTokenizer(line);
        }
        return st.nextToken();
    }
    static int nextInt() throws IOException { return Integer.parseInt(next());
    static long nextLong() throws IOException { return Long.parseLong(next());

    public static void main(String[] args) throws IOException {
        int n = nextInt();
        long[] a = new long[n];
        for (int i = 0; i < n; i++) a[i] = nextLong();
        out.println(solve(a));
        out.flush();
    }

    static long solve(long[] a) { … }
}
```

Type it until you can produce it in ninety seconds. Notes: the `null` check in `next()` makes "read until end of input" work (`while ((t = next()) ≠ null)`); everything numeric that could exceed two billion is a `long` from the start; `solve` is separate from I/O so the logic is testable and readable; and `out.flush()` is the last line, always.

When the problem is small, the whole-input read is even simpler: `String[] tokens = new String(System.in.readAllBytes()).trim().split("\\s+");` — one allocation, then index through tokens. Watch the empty-input case: `"".split("\\s+")` gives `[""]`, so check `tokens[0].isEmpty()`.

Reading tricky formats

Grid of characters: `char[] row = in.readLine().toCharArray()` per row; never `Scanner.next()` (it splits on whitespace and there is none).

Line with a count then values on the same line: the tokenizer handles it; the count is just the first token.

Unknown number of test cases: loop on `next() ≠ null`.

A line of text with spaces: `in.readLine()` directly — do not mix with the tokenizer without draining it first, and remember `Scanner.nextInt()` leaves the newline in the buffer, so a following `nextLine()` returns `""`.

Very large numbers: `BigInteger` / `BigDecimal` — slow, but correct; a problem that says "up to 10^{18}" fits in `long` (9.2×10^{18}); one that says "up to 100 digits" does not.

The complexity budget

A judge does roughly **10^8 simple operations per second** in Java after warm-up (a little less with heavy object allocation). With a typical 1–2 s limit:

Largest `n` in the statement	Complexity that fits	Typical shape
≤ 12	O(n!)	permutations, brute force
≤ 25	$O(2^n)$	subsets, bitmask DP
≤ 100	$O(n^4)$	four nested loops, small DP
≤ 500	$O(n^3)$	Floyd–Warshall, cubic DP
≤ 5 000	$O(n^2)$	pairwise comparison, LIS quadratic
≤ 10^6	O(n log n)	sorting, heaps, binary search, `TreeMap`
≤ 10^8	O(n)	single pass, hashing, two pointers, prefix sums
≥ 10^9	O(log n) or O(1)	math, binary search on the answer, digit tricks

Read the constraints *before* designing: `n ≤ 10⁵` rules out $O(n^2)$ and tells you to think in sorting, hashing or heaps; `n ≤ 20` invites a bitmask. Say your complexity out loud before coding — interviewers mark the reasoning, not just the answer.

Memory budget

A 256 MB limit fits about 60 million `int`s (4 bytes each), 30 million `long`s, or a few million small objects (16–32 bytes each plus references). A `boolean[10_000_000]` is 10 MB; a `List<Integer>` of ten million is ~200 MB and probably dead. Use primitive arrays for big data, `BitSet` for big boolean tables, and `int[][]` rather than `List<List<Integer>>` for grids.

Output discipline

- Buffer output (`PrintWriter` or `StringBuilder`); flush once.
- Match the format exactly: a trailing space is usually tolerated, a missing newline usually is, a stray debugging line never is.

`Locale.ROOT` for decimals; print the exact number of decimals asked for with `%.6f`, not `Math.round`.

For very large answers "modulo $10^9 + 7$": reduce at every multiplication, keep intermediates in `long`, and `Math.floorMod` for anything that could go negative.

The edge-case checklist

Before saying "done", run these in your head against your code:

Empty input / `n = 0`; **single** element / `n = 1`.

All equal elements; **already sorted** and **reverse sorted**.

Negative numbers and **zero** where the statement allows them.

Maximum sizes — does the complexity fit? Does `int` overflow anywhere (`a[i] * a[j]`, sums, `mid = (lo + hi) / 2`)?

Duplicates — does your `Set` / `Map` logic handle them as the problem wants?

Boundaries — off-by-one in loops, `substring(i, j)` end-exclusive, `length - 1`.

Strings: empty, one character, all the same character, case sensitivity, Unicode if the statement hints at it.

Graphs: disconnected, self-loops, a single node, cycles when you assumed a tree.

Saying "let me check edge cases" and naming three of these is worth more than a fourth correct test.

Interview angle

"How many operations can Java do in a second?" About 10^8 simple ones — the number the budget table is built on.

"n is 10^5; what complexity do you need?" O(n log n) or better; $O(n^2)$ is 10^{10} — too slow.

"Why `long` here?" Any sum or product of values up to 10^9 overflows `int`; `long` covers 9×10^{18}.

"Why is your I/O slow?" `Scanner` and unbuffered `println`; switch to the reader template and a flushed `PrintWriter`.

"How do you handle end of input?" `readLine()` returns `null`; the `next()` helper returns `null` past it.

Key takeaways

- Own the template: buffered reader with a refilling tokenizer, `PrintWriter`, `solve` separated from I/O, `flush` last.
- 10^8 operations per second; map `n` to the complexity that fits before designing.
- Memory: primitive arrays, `BitSet`, `int[][]`; boxed collections of millions are dead on arrival.
- `long` early; `Locale.ROOT`; exact format; modulo at every step.
- Run the eight-point edge-case checklist aloud.

EXERCISE 1

Read until the end

Using the template's `next()` helper (which returns `null` at end of input — **no** `Scanner`), read every integer from standard input, however many lines there are. Print `count=<n> sum=<sum> min=<min> max=<max>` using `long`s, or just `count=0` when there are none. Write the answer through a flushed `PrintWriter`.

Example (three lines): `3 9`, `-2`, `7 1` →

```
count=5 sum=18 min=-2 max=9
```

INPUT 1

```
3 9
-2
7 1
```

OUTPUT 1

```
count=5 sum=18 min=-2 max=9
```

INPUT 2

```

```

OUTPUT 2

```
count=0
```

REFERENCE SOLUTION

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

public class Main {
    static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    static StringTokenizer st;

    static String next() throws IOException {
        while (st == null || !st.hasMoreTokens()) {
            String line = in.readLine();
            if (line == null) return null;
            st = new StringTokenizer(line);
        }
        return st.nextToken();
    }

    public static void main(String[] args) throws IOException {
        PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(
        long count = 0, sum = 0, min = Long.MAX_VALUE, max = Long.MIN_VALUE;
        for (String t; (t = next()) != null; ) {
            long x = Long.parseLong(t);
            count++;
            sum += x;
            min = Math.min(min, x);
            max = Math.max(max, x);
        }
        if (count == 0) out.println("count=0");
        else out.println("count=" + count + " sum=" + sum + " min=" + min + " max="
        out.flush();
    }
}
```

EXERCISE 2

The complexity budget

Read an integer `n` (`1 ≤ n ≤ 10⁹`). With a budget of `100000000` operations and `log n` taken as the bit length of `n` (`32 - Integer.numberOfLeadingZeros(n)`), print `n=<n> budget=100000000` and then, for each of `n log n` , `n^2` , `n^3` and `2^n` in that order, either `<name>: <operations> fits` or `<name>: too slow` . Use `Math.multiplyExact` on `long` s and treat an overflow as too slow; `2^n` is too slow for `n ≥ 63` .

Example: `20` →

```
n=20 budget=100000000
n log n: 100 fits
n^2: 400 fits
n^3: 8000 fits
2^n: 1048576 fits
```

INPUT 1

```
20
```

OUTPUT 1

```
n=20 budget=100000000
n log n: 100 fits
n^2: 400 fits
n^3: 8000 fits
2^n: 1048576 fits
```

INPUT 2

```
100000
```

OUTPUT 2

```
n=100000 budget=100000000
n log n: 1700000 fits
n^2: too slow
n^3: too slow
2^n: too slow
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static final long BUDGET = 100_000_000L;

    static String verdict(String name, long ops) {
        return ops ⩽ BUDGET ? name + ": " + ops + " fits" : name + ": too slow";
    }

    static long safeMul(long a, long b) {
        try { return Math.multiplyExact(a, b); } catch (ArithmeticException e) { ret
    }

    public static void main(String[] args) {
        long n = new Scanner(System.in).nextLong();
        System.out.println("n=" + n + " budget=" + BUDGET);
        long log = 64 - Long.numberOfLeadingZeros(n);
        System.out.println(verdict("n log n", safeMul(n, log)));
        long n2 = safeMul(n, n);
        System.out.println(verdict("n^2", n2));
        System.out.println(verdict("n^3", safeMul(n2, n)));
        System.out.println(verdict("2^n", n ⩾ 63 ? Long.MAX_VALUE : 1L << n));
    }
}
```

QUIZ

1. A Java solution can do roughly how many simple operations per second on a judge?

A. 10^4

B. 10^6

C. 10^8

D. 10^{12}

2. The statement says `n ≤ 10⁵`. Which complexity is the target?

A. $O(n^3)$

B. $O(n^2)$

C. O(n log n) or better

D. $O(2^n)$

3. `mid = (lo + hi) / 2` is dangerous because…

A. It rounds up

B. `lo + hi` can overflow `int`; use `lo + (hi - lo) / 2` or `>>> 1`

C. It is slow

D. It ignores `hi`

4. `"".split("\\s+")` returns…

A. An empty array

B. `[""]` — one empty string, so check for it before parsing

C. `null`

D. Throws

5. A 256 MB memory limit comfortably fits…

A. A `List<Integer>` of 50 million

B. An `int[]` of 50 million (200 MB)

C. A `String` per integer for 50 million

D. 50 million `Integer` objects

ANSWER KEY · THE INTERVIEW TEMPLATE — INPUT, OUTPUT, LIMITS

1. C The number the complexity budget is built on; heavy allocation lowers it.

2. C O(n²) is 10^{10} operations — a hundred seconds.

3. B The famous bug in the JDK's own `binarySearch` for years.

4. B The whole-input read pattern must handle empty input explicitly.

5. B Boxed collections cost 4–5× the memory of primitive arrays.

LESSON 2 · 14 MIN

equals, hashCode and Comparable — the contracts that make collections work

Half of the collections framework — `HashMap`, `HashSet`, `contains`, `remove(Object)`, `indexOf`, `distinct()` — depends on `equals` and `hashCode` agreeing; the other half — `TreeMap`, `sort`, `PriorityQueue`, `max` — depends on a consistent ordering. Get the contracts wrong and the symptoms are maddening: a key you just inserted "is not in the map", a set holds two equal objects, a sort throws `IllegalArgumentException: Comparison method violates its general contract`. Interviewers ask about these contracts because every Java developer has been bitten by them. This lesson states them exactly, shows the implementations that satisfy them, and lists the mistakes.

The `equals` contract

For non-null references: **reflexive** (`a.equals(a)`), **symmetric** (`a.equals(b) == b.equals(a)`), **transitive**, **consistent** (repeated calls agree while nothing changes), and `a.equals(null)` is `false`. The default in `Object` is identity (`==`). Override it when instances are *values* — two `Point(1, 2)`s are the same point — and leave it alone for *entities* whose identity is their existence (a connection, a thread, a session).

```
@Override public boolean equals(Object o) {
    if (this == o) return true;                          // fast path
    if (o == null || getClass() ≠ o.getClass()) return false;   // or `!(o i
    Point p = (Point) o;
    return x == p.x && y == p.y;                         // compare every sig
}
```

`getClass() ≠` versus `instanceof`: `getClass()` makes a `Point` never equal a `ColoredPoint`, which keeps symmetry trivially; `instanceof` lets subclass instances be equal to base instances but makes symmetry your problem (`ColoredPoint.equals(Point)` must then ignore colour). Pick `getClass()` unless you have a reason. The parameter type must be `Object` — `equals(Point p)` is an *overload* that `HashMap` never calls; `@Override` catches this mistake.

The `hashCode` contract

Equal objects must have equal hash codes. Unequal objects *may* share one (a collision), but fewer collisions mean faster maps. `hashCode` must be consistent while the object is unchanged. Override it whenever you override `equals` — a `HashMap` finds the bucket by `hashCode` first and only then calls `equals`, so a class with `equals` but the default identity `hashCode` puts equal keys in different buckets and `get` fails.

```
@Override public int hashCode() { return Objects.hash(x, y); }            // j
@Override public int hashCode() { return 31 * Integer.hashCode(x) + Integer.ha
```

`Objects.hash` boxes and allocates an array — negligible almost always, avoidable in a hot key class. For arrays use `Arrays.hashCode` / `Arrays.equals` (a plain `array.hashCode()` is identity). `String`, the boxed primitives, `List`, `Set`, `Map`, `Optional`, records and enums all implement both correctly; your own classes are the risk.

Records generate `equals`, `hashCode` and `toString` from the components — the fastest correct way to a value class, and the answer to "how would you make this a good map key" nine times out of ten.

Mutable keys: the silent disaster

```
Set<List<Integer>> seen = new HashSet<>();
List<Integer> key = new ArrayList<>(List.of(1, 2));
seen.add(key);
key.add(3);
seen.contains(key);        // false — it hashes to a different bucket now
seen.contains(List.of(1, 2)); // also false — the stored key's contents change
```

The set now contains an entry that no lookup can find and `remove` cannot remove: a leak *and* a bug. The rule: **fields that participate in `equals` / `hashCode` must not change while the object is a key** — best enforced by making the key class immutable (`final` fields, records).

`Comparable`: the natural order

```
record Version(int major, int minor) implements Comparable<Version> {
    public int compareTo(Version o) {
        int c = Integer.compare(major, o.major);
        return c != 0 ? c : Integer.compare(minor, o.minor);
    }
}
```

`compareTo` must be **antisymmetric** (`sgn(a.compareTo(b)) == -sgn(b.compareTo(a))`), **transitive**, and consistent; it should be *consistent with* `equals` (`compareTo == 0` iff `equals`) — `TreeSet` and `TreeMap` use only `compareTo`, so two objects that compare equal but are not `equals` collapse into one entry. Use `Integer.compare(a, b)`, never `a - b`: the subtraction overflows for values of opposite sign (`Integer.MIN_VALUE - 1` is positive), which is exactly the "violates its general contract" exception when `TimSort` catches the inconsistency. Compare `double`s with `Double.compare` (handles `NaN` and `-0.0`), strings with `compareTo` or `String.CASE_INSENSITIVE_ORDER`.

`Comparator`: every other order

```
Comparator<Employee> byDeptThenSalaryDescThenName =
    Comparator.comparing(Employee::dept)
              .thenComparing(Employee::salary, Comparator.reverseOrder())
              .thenComparing(Employee::name);
list.sort(byDeptThenSalaryDescThenName);
Comparator.comparingInt(String::length);                  // primitive special
Comparator.nullsFirst(Comparator.naturalOrder());         // nulls are otherwi
Comparator.reverseOrder(); comparator.reversed();
```

A `Comparator` is the ordering for one *use* (this sort, this heap), `Comparable` is the type's *one* natural order. Sorting with `Comparator.comparing` chains beats hand-written `compare` methods on readability and is harder to get wrong. `List.sort` / `Arrays.sort` on objects are **stable** (TimSort): equal elements keep their input order, which is why "sort by score, ties by input order" needs no extra key.

Where each is used

Needs `equals` / `hashCode`	Needs an order
`HashMap`, `HashSet`, `LinkedHash*`, `ConcurrentHashMap`	`TreeMap`, `TreeSet`, `PriorityQueue`
`List.contains`, `indexOf`, `remove(Object)`	`Collections.sort`, `List.sort`, `Arrays.sort(T[])`
`Stream.distinct`, `groupingBy` keys, `toSet`	`Stream.sorted`, `min`, `max`, `Collections.min/max`
`Objects.equals`, `assertEquals` in tests	`Arrays.binarySearch` (must be sorted with the same order)

Common mistakes, in one place

Overriding `equals` without `hashCode` (or vice versa).

`equals(MyType other)` — an overload, not an override; use `Object` and `@Override`.

`a - b` in `compareTo`; `==` on `Double` / `Integer` fields inside `equals`.

Mutating a key after insertion.

`compareTo` inconsistent with `equals`, then wondering why a `TreeSet` lost elements.

Comparing `double` fields with `==` (`NaN ≠ NaN`; `0.0 == -0.0`); use `Double.compare`.

Forgetting that `Arrays.equals` is needed for array fields and `Arrays.deepEquals` for nested arrays.

Interview angle

"What is the `equals` / `hashCode` contract?" Equal objects must have equal hash codes; `equals` is reflexive, symmetric, transitive, consistent, null-false.

"What happens if you override only `equals`?" Equal keys land in different `HashMap` buckets — lookups fail.

"How does `HashMap.get` use them?" Hash to a bucket, then `equals` along the chain.

"`Comparable` versus `Comparator`?" The type's natural order versus an external, per-use order; `TreeMap` and `PriorityQueue` accept either.

"Why not `return a - b` in a comparator?" Overflow for values of opposite sign breaks antisymmetry; `Integer.compare`.

Key takeaways

Override both or neither; parameter type `Object`; compare all significant fields; `Objects.hash` / `Arrays.hashCode`.

Records give correct value semantics for free — reach for them as map keys.

Keys must be immutable in their hashed fields.

`compareTo`: `Integer.compare` / `Double.compare`, transitive, consistent with `equals` for tree collections.

`Comparator.comparing(...).thenComparing(...)` chains; object sorts are stable.

EXERCISE 1

Point as a key

Write a class `Point` with `int x, y`, a correct `equals(Object)` and a consistent `hashCode` (not a record — do it by hand). Read an integer `n` and `n` points, then `m` and `m` query points. Print `distinct=<size of a HashSet of the points>`, then for each query `(x,y): present` or `(x,y): absent`, then `hashAgree=<whether two separately constructed equal points have equal hash codes>`.

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

```
distinct=2
(1,2): present
(2,1): absent
hashAgree=true
```

INPUT 1

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

OUTPUT 1

```
distinct=2
(1,2): present
(2,1): absent
hashAgree=true
```

INPUT 2

```
1
0 0
1
0 0
```

OUTPUT 2

```
distinct=1
(0,0): present
hashAgree=true
```

REFERENCE SOLUTION

```
import java.util.*;

final class Point {
    final int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Point p = (Point) o;
        return x == p.x && y == p.y;
    }
    @Override public int hashCode() { return 31 * Integer.hashCode(x) + Integer.hash
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Set<Point> set = new HashSet<>();
        for (int i = 0; i < n; i++) set.add(new Point(in.nextInt(), in.nextInt()));
        System.out.println("distinct=" + set.size());
        int m = in.nextInt();
        for (int i = 0; i < m; i++) {
            int x = in.nextInt(), y = in.nextInt();
            System.out.println("(" + x + "," + y + "): " + (set.contains(new Point(x
        }
        System.out.println("hashAgree=" + (new Point(1, 2).hashCode() == new Point(1
    }
}
```

EXERCISE 2

Sort by many keys

Write `record Employee(String dept, String name, int salary) implements Comparable<Employee>` whose natural order is department ascending, then salary **descending**, then name ascending — using `Integer.compare`, never subtraction. Read an integer `n` and `n` employees `dept name salary`, sort them, and print `<dept> <name> <salary>` per line. Then print `richest=<name of the highest salary; ties to the earliest in input>` using `Collections.max` with `Comparator.comparingInt`, and `byName=<names sorted alphabetically, comma-separated>` using `Comparator.comparing`.

Example: `3` then `eng ann 100`, `ops bob 90`, `eng cy 120` →

```
eng cy 120
eng ann 100
ops bob 90
richest=cy
byName=ann,bob,cy
```

INPUT 1

```
3
eng ann 100
ops bob 90
eng cy 120
```

OUTPUT 1

```
eng cy 120
eng ann 100
ops bob 90
richest=cy
byName=ann,bob,cy
```

INPUT 2

```
2
x zed 50
x amy 50
```

OUTPUT 2

```
x amy 50
x zed 50
richest=zed
byName=amy,zed
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.stream.*;

record Employee(String dept, String name, int salary) implements Comparable<Employee
    public int compareTo(Employee o) {
        int c = dept.compareTo(o.dept);
        if (c != 0) return c;
        c = Integer.compare(o.salary, salary);
        return c != 0 ? c : name.compareTo(o.name);
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Employee> staff = new ArrayList<>();
        for (int i = 0; i < n; i++) staff.add(new Employee(in.next(), in.next(), in.
        List<Employee> sorted = new ArrayList<>(staff);
        Collections.sort(sorted);
        for (Employee e : sorted) System.out.println(e.dept() + " " + e.name() + " '
        Employee richest = Collections.max(staff, Comparator.comparingInt(Employee::
        System.out.println("richest=" + richest.name());
        System.out.println("byName=" + staff.stream().sorted(Comparator.comparing(Em
    }
}
```

QUIZ

1. A class overrides `equals` but not `hashCode`. In a `HashMap` …

A. Everything works

B. Equal keys may land in different buckets, so `get` misses keys that are present

C. It throws

D. Keys are sorted

2. `public boolean equals(Point other)` is…

A. A correct override

B. An overload — `HashMap` calls `equals(Object)`, which is still the identity default

C. Faster

D. Required for records

3. `return a - b;` in a comparator is wrong because…

A. It is slow

B. It overflows for operands of opposite sign, breaking antisymmetry — `Integer.compare`

C. It returns `long`

D. It is fine

4. A `TreeSet` of objects whose `compareTo` returns 0 for unequal objects…

A. Stores both

B. Keeps only one — tree collections use `compareTo`, not `equals`

C. Throws

D. Sorts them randomly

5. Predict: a `List<Integer>` key `[1, 2]` is added to a `HashSet`, then `3` is appended to that same list. `set.contains(key)` is…

A. `true`

B. `false` — the hash changed, the entry is in the old bucket

C. Throws

D. `true` after rehash

6. Object sorts in Java (`List.sort` , `Arrays.sort(T[])`) are…

A. Unstable quicksort

B. Stable (TimSort): equal elements keep their input order

C. Random

D. Bubble sort

ANSWER KEY · EQUALS, HASHCODE AND COMPARABLE — THE CONTRACTS THAT MAKE COLLECTIONS WORK

1. B Bucket first, `equals` second — both must agree.

2. B Always `@Override public boolean equals(Object o)`.

3. B TimSort detects the inconsistency: 'Comparison method violates its general contract'.

4. B Keep `compareTo` consistent with `equals`, or add a tie-breaker.

5. B Mutable keys leak and break lookups; use immutable keys.

6. B So 'sort by score, ties by input order' needs no extra key. Primitive `Arrays.sort(int[])` is dual-pivot quicksort.

LESSON 3 · 14 MIN

The pitfalls that fail interviews

Most rejected solutions are not wrong algorithms. They are right algorithms with an `int` that overflowed, a `/` that truncated, a `==` that compared references, or a sort that hit its worst case. Interviewers know the list; this lesson is that list, each item with the one-line fix, organised so you can run it as a review pass over your own code before you say "done". Nothing here is new — every item was taught in an earlier module — but seeing them together is what turns knowledge into a reflex.

Arithmetic

`int` overflow. `a[i] * a[j]` with values up to 10^5 is 10^{10}: overflow. Sums of 10^5 values up to 10^5: overflow. `mid = (lo + hi) / 2` with `hi` near `Integer.MAX_VALUE`: overflow — write `lo + (hi - lo) / 2` or `(lo + hi) >>> 1`. Fix: `long` for anything that multiplies or accumulates; `Math.addExact` / `multiplyExact` when you would rather throw than be silently wrong.

Integer division. `7 / 2` is `3`; `1 / 2 * x` is `0`. Make one operand a `double` when you mean a fraction. Division **truncates toward zero**: `-7 / 2` is `-3`, not `-4`; `Math.floorDiv` rounds down.

`%` with negatives. `-7 % 3` is `-1` in Java. For a non-negative remainder (hashing, circular indexes, modular arithmetic) use `Math.floorMod(-7, 3)` → `2`, or `((a % m) + m) % m`.

`Math.abs(Integer.MIN_VALUE)` is `Integer.MIN_VALUE` — still negative. Any "take the absolute value" on unconstrained input needs `long`.

Floating point equality. `0.1 + 0.2 == 0.3` is `false`. Compare with a tolerance (`Math.abs(a - b) < 1e-9`), or avoid doubles: money in `long` cents, exact fractions as numerator/denominator, `BigDecimal` when the statement demands exactness.

Casting truncates. `(int) 3.99` is `3`; `(int) -3.99` is `-3`; `(int) 1e10` saturates to `Integer.MAX_VALUE`. `Math.round` returns `long` for a `double` argument. `(int) 'a'` is `97` and `(char) ('a' + 1)` is `'b'` — `'a' + 1` alone is an `int`, `98`.

Modular arithmetic under $10^9 + 7$. Reduce after every `*`; multiply in `long`; use `floorMod` after subtraction; precompute factorials and inverses when the problem repeats them.

Strings and characters

`==` on strings. Use `equals`. The interned-literal cases that make `==` "work" in tests will not survive input.

Concatenation in a loop is O(n²). `StringBuilder`, or `String.join`, or `Collectors.joining`.

`substring` in a loop is O(n) each (a copy since Java 7u6). Sliding-window and palindrome problems should index into the original string or a `char[]`.

`split` quirks. `"a,b,,".split(",")` drops trailing empties (`[a, b]`); `split(",", -1)` keeps them. `split(".")` splits on *any character* — the argument is a regex; `split("\\.")` or `split(Pattern.quote("."))`.

Characters versus code points. `s.length()` counts UTF-16 units; emoji are two. `s.charAt(i) - '0'` converts a digit; `Character.getNumericValue` handles more. `Character.isLetter` versus `[a-z]` — decide whether the statement means ASCII.

Case. `toLowerCase()` without a `Locale` follows the default locale (the Turkish `İ` problem); `toLowerCase(Locale.ROOT)` in anything that must be deterministic.

Boxing and collections

`==` on `Integer` / `Long`. Cached to 127 and not beyond — `map.get(a) == map.get(b)` lies past 127. `equals`, or unbox to `int`.

`Arrays.asList(int[])` gives a `List<int[]>` with one element — the array. Box first, or `Arrays.stream(a).boxed().toList()`.

`list.remove(int)` versus `remove(Object)` on a `List<Integer>`: `remove(2)` removes *index* 2. `remove(Integer.valueOf(2))` for the value.

Modifying while iterating → `ConcurrentModificationException`. Use the iterator's `remove`, `removeIf`, or collect first then remove.

Mutable keys in a `HashMap` / `HashSet` (previous lesson): unreachable entries.

`Arrays.sort(int[])` worst case. Dual-pivot quicksort on primitives is O(n²) on adversarial input, and judges have anti-quicksort tests. Shuffle first, or sort `Integer[]` /a `List` (TimSort, guaranteed n log n), or use counting/radix when values are bounded.

`PriorityQueue` iteration is not sorted. Only `peek` / `poll` give order; to print sorted, poll repeatedly or sort separately.

`HashMap` **iteration order is arbitrary** and can differ between runs and versions; never rely on it for output. `TreeMap` or `LinkedHashMap`, or sort the keys.

`Collections.unmodifiableList` **is a view** over the original; `List.copyOf` is a snapshot.

`subList`, `Arrays.asList`, `keySet`, `values` are views: changes propagate, and structural changes to the parent invalidate `subList`.

Control flow and recursion

Recursion depth. Java's default stack handles roughly 10^4–2×10^4 frames; a DFS on a path graph of 10^5 nodes overflows. Convert to iteration with an explicit stack, or run the recursion in a `new Thread(null, runnable, "dfs", 1 << 26)` with a bigger stack.

`switch` **fall-through** in the old form; missing `break` merges cases. Use arrow cases.

Off-by-one. `for (i = 0; i ⩽ n; i++)` with an array of `n`; `substring(i, j)` is end-exclusive; `Arrays.fill` / `copyOfRange` are `[from, to)`.

Uninitialised `Integer.MIN_VALUE` / `MAX_VALUE` **sentinels** when the answer can legitimately equal them; use `long` sentinels or a boolean `found`.

Returning inside `try` **with a** `finally` **that also returns** — the `finally` wins and swallows the exception. Never `return` from `finally`.

Reading and output

`Scanner.nextInt()` **then** `nextLine()` returns the rest of the current line (`""`). Consume the newline, or use the reader template.

Forgetting `flush()`, trailing debugging prints, printing with the wrong locale — the output half of the earlier lesson.

How to use this list

Before submitting, scan your code once for each group heading: *arithmetic* (any `int` that grows? any `/` or `%` with negatives?), *strings* (`==`? concatenation in a loop? `split` on a regex character?), *collections* (boxed `==`? sorting a primitive array on adversarial input? iteration order?), *recursion* (depth?), *I/O* (flush?). Thirty seconds, and it catches the majority of "correct but rejected" submissions. In an interview, doing it aloud — "let me check for overflow: the sum is at most 10^{10}, so `long`" — is the visible carefulness that gets people hired.

Interview angle

"Why did `-7 % 3` give `-1`?" Java's `%` takes the sign of the dividend; `Math.floorMod` for a non-negative result.

"Is `Integer a = 1000, b = 1000; a == b` true?" No — outside the cache; `equals`.

"Why is `Arrays.sort(int[])` dangerous on a judge?" Quicksort worst case on crafted input; shuffle or sort objects.

"What does `\"a.b\".split(\".\")` return?" An empty array — `.` is a regex wildcard; escape it.

"How do you avoid `StackOverflowError` in a deep DFS?" Iterate with an explicit stack, or a thread with a larger stack.

Key takeaways

Arithmetic: `long`, `floorMod` / `floorDiv`, `lo + (hi - lo) / 2`, tolerance for doubles, `abs(MIN_VALUE)`.

Strings: `equals`, `StringBuilder`, no `substring` in loops, `split` is a regex with dropped trailing empties.

Collections: boxed `==`, `remove(int)`, CME, mutable keys, `Arrays.sort(int[])` worst case, unordered `HashMap`, views.

Recursion depth and `finally` returns; `Scanner` newline; `flush`.

Run the list as a thirty-second review pass, out loud.

EXERCISE 1

Exact arithmetic

Read an integer `n` and `n` lines `a op b` with `long` operands and `op` one of `+ - * % /` (`b` is never 0). For `+`, `-` and `*` print the exact result via `Math.addExact` / `subtractExact` / `multiplyExact`, or `overflow` if it throws. For `%` print `rem=<a % b> mod=<Math.floorMod(a, b)>`; for `/` print `div=<a / b> floor=<Math.floorDiv(a, b)>`.

Example: `4` then `9223372036854775807 + 1`, `-7 % 3`, `-7 / 2`, `6 * 7` →

```
overflow
rem=-1 mod=2
div=-3 floor=-4
42
```

INPUT 1

```
4
9223372036854775807 + 1
-7 % 3
-7 / 2
6 * 7
```

OUTPUT 1

```
overflow
rem=-1 mod=2
div=-3 floor=-4
42
```

INPUT 2

```
3
-9223372036854775808 - 1
4000000000 * 4000000000
7 % -3
```

OUTPUT 2

```
overflow
overflow
rem=1 mod=-2
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            long a = in.nextLong();
            String op = in.next();
            long b = in.nextLong();
            try {
                switch (op) {
                    case "+" -> System.out.println(Math.addExact(a, b));
                    case "-" -> System.out.println(Math.subtractExact(a, b));
                    case "*" -> System.out.println(Math.multiplyExact(a, b));
                    case "%" -> System.out.println("rem=" + (a % b) + " mod=" + Math
                    default -> System.out.println("div=" + (a / b) + " floor=" + Mat
                }
            } catch (ArithmeticException e) {
                System.out.println("overflow");
            }
        }
    }
}
```

EXERCISE 2

Predict, then verify

Read an integer `k` and `k` expression ids in `1..15`. For each id print `<id>: <value>`, where the value is what Java computes for the expression — write each expression literally, do not hard-code the answers, and **predict every line before running**:

id	expression	id	expression
1	`(int) 3.99`	9	`-7 / 2`
2	`7 / 2`	10	`-7 % 3`
3	`7 / 2.0`	11	`"1" + 2 + 3`
4	`'a' + 1`	12	`1 + 2 + "3"`
5	`(char) ('a' + 1)`	13	`"a.b".split(".").length`
6	`Integer.MAX_VALUE + 1`	14	`"a,b,,".split(",").length`
7	`Math.abs(Integer.MIN_VALUE)`	15	`Integer.valueOf(128) == Integer.valueOf(128)`
8	`0.1 + 0.2 == 0.3`		

Example: `3` then `2 3 11` →

```
2: 3
3: 3.5
11: 123
```

INPUT 1

```
3
2 3 11
```

OUTPUT 1

```
2: 3
3: 3.5
11: 123
```

INPUT 2

```
6
1 4 5 6 7 8
```

OUTPUT 2

```
1: 3
4: 98
5: b
6: -2147483648
7: -2147483648
8: false
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static String eval(int id) {
        return switch (id) {
            case 1 -> String.valueOf((int) 3.99);
            case 2 -> String.valueOf(7 / 2);
            case 3 -> String.valueOf(7 / 2.0);
            case 4 -> String.valueOf('a' + 1);
            case 5 -> String.valueOf((char) ('a' + 1));
            case 6 -> String.valueOf(Integer.MAX_VALUE + 1);
            case 7 -> String.valueOf(Math.abs(Integer.MIN_VALUE));
            case 8 -> String.valueOf(0.1 + 0.2 == 0.3);
            case 9 -> String.valueOf(-7 / 2);
            case 10 -> String.valueOf(-7 % 3);
            case 11 -> "1" + 2 + 3;
            case 12 -> 1 + 2 + "3";
            case 13 -> String.valueOf("a.b".split(".").length);
            case 14 -> String.valueOf("a,b,,".split(",").length);
            case 15 -> String.valueOf(Integer.valueOf(128) == Integer.valueOf(128));
            default -> "?";
        };
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt();
        for (int i = 0; i < k; i++) {
            int id = in.nextInt();
            System.out.println(id + ": " + eval(id));
        }
    }
}
```

QUIZ

1. `-7 % 3` in Java is…

A. `2`

B. `-1` — the remainder takes the sign of the dividend; `Math.floorMod` gives `2`

C. `1`

D. `-2`

2. `"a.b".split(".")` returns…

A. `["a", "b"]`

B. An empty array — `.` is a regex matching every character, and trailing empties are removed

C. `["a.b"]`

D. Throws

3. `Arrays.sort(int[])` on adversarial judge input…

A. Is always O(n log n)

B. Can hit dual-pivot quicksort's O(n²) worst case — shuffle first or sort boxed/`List`

C. Throws

D. Is stable

4. Predict: `List<Integer> l = new ArrayList<>(List.of(5, 6, 7)); l.remove(1); System.out.println(l);`

A. `[5, 7]` — index 1 removed

B. `[5, 6, 7]`

C. `[6, 7]`

D. `[5, 6]`

5. A recursive DFS on a path graph of 10^5 nodes in Java…

A. Works fine

B. Likely throws `StackOverflowError` — iterate with an explicit stack or use a thread with a larger stack

C. Is $O(n^2)$

D. Needs `volatile`

6. `Math.abs(Integer.MIN_VALUE)` is…

A. `2147483648`

B. `Integer.MIN_VALUE` — still negative

C. `0`

D. Throws

ANSWER KEY · THE PITFALLS THAT FAIL INTERVIEWS

1. B Circular indexes and hashing need `floorMod`.

2. B `split("\\.")` or `split(Pattern.quote("."))`.

3. B Object sorts use TimSort, which is guaranteed O(n log n) and stable.

4. A `remove(int)` is by index; `remove(Integer.valueOf(1))` would search for the value 1 (absent).

5. B Default stacks hold roughly 10^4–2×10^4 frames.

6. B The positive value does not fit in `int`; use `long`.

LESSON 4 · 15 MIN

The collections idiom sheet

Interview problems repeat. Under the surface of "find the k most frequent", "longest substring without repeats", "merge intervals" and "next greater element" are perhaps fifteen shapes, and each has a two-to-six-line Java idiom that experienced candidates type without thinking. This lesson is the sheet: the idiom, the structure it uses, the complexity, and the trap. Learn them as vocabulary — you should be able to say "frequency map, then a min-heap of size k" as one phrase.

Counting and grouping

```
Map<String, Integer> freq = new HashMap<>();
for (String w : words) freq.merge(w, 1, Integer::sum);
Map<Integer, List<String>> byLen = new HashMap<>();
for (String w : words) byLen.computeIfAbsent(w.length(), k → new ArrayList<>(
int[] count = new int[26]; for (char c : s.toCharArray()) count[c - 'a']++;
```

Trap: boxed `==` when comparing counts; an alphabet array is both faster and immune.

Top-k and ordering by a key

```
PriorityQueue<Map.Entry<String, Integer>> heap =
    new PriorityQueue<>(Map.Entry.comparingByValue());
for (var e : freq.entrySet()) { heap.offer(e); if (heap.size() > k) heap.poll(
// heap now holds the k largest; poll to list them ascending, reverse for desc
```

Use a **min**-heap to keep the *largest* k (evict the smallest). For ties, put the secondary key in the comparator (`comparingByValue().thenComparing(Map.Entry.comparingByKey(Comparator.reverseOrder()))` so that alphabetically earlier words survive). For k close to n, sorting is simpler and just as fast. `PriorityQueue` has no `decreaseKey`: for Dijkstra, push duplicates and skip stale entries on poll.

Stacks, queues, deques

```
Deque<Integer> stack = new ArrayDeque<>(); stack.push(x); stack.pop(); stack.p
Deque<Integer> queue = new ArrayDeque<>(); queue.offer(x); queue.poll(); queue
```

Never `Stack` (synchronised, a `Vector`) or `LinkedList` (slow, allocation-heavy). `ArrayDeque` rejects `null` — use a sentinel or `Integer` box carefully. **Monotonic stack** for "next greater element" / histogram problems:

```
Deque<Integer> idx = new ArrayDeque<>();                    // indexes with decr
for (int i = 0; i < n; i++) {
    while (!idx.isEmpty() && a[idx.peek()] < a[i]) next[idx.pop()] = i;   // c
    idx.push(i);
}
```

Sliding-window maximum is the same idea with a deque that also drops indexes that left the window from the front — O(n).

Sorted structures: `TreeMap` navigation

```
TreeMap<Integer, String> tm = …;
tm.floorKey(x); tm.ceilingKey(x); tm.lowerKey(x); tm.higherKey(x);    // neare
tm.firstKey(); tm.lastKey(); tm.headMap(x); tm.tailMap(x, true); tm.subMap(lo,
tm.pollFirstEntry();                                                  // sor
```

Interval scheduling ("is this slot free?"), nearest-value queries, "count of elements ≤ x" (via `headMap(x, true).size()` — O(n), so use a Fenwick tree if it is hot), and an ordered multiset via `TreeMap<Integer, Integer>` counts.

Two pointers and sliding windows

```
int lo = 0; Map<Character, Integer> last = new HashMap<>(); int best = 0;
for (int hi = 0; hi < s.length(); hi++) {
    char c = s.charAt(hi);
    if (last.containsKey(c) && last.get(c) ⩾ lo) lo = last.get(c) + 1;     //
    last.put(c, hi);
    best = Math.max(best, hi - lo + 1);
}
```

Windows are for contiguous subarrays with a monotone condition; two pointers from both ends for sorted arrays (pair sums). Both O(n). **Trap:** shrinking with `while` when the condition is not monotone — check that moving `lo` right can only help.

Prefix sums and hashing for subarrays

```
long[] pre = new long[n + 1]; for (int i = 0; i < n; i++) pre[i + 1] = pre[i]
Map<Long, Integer> seen = new HashMap<>(); seen.put(0L, 1); long run = 0; int
for (int x : a) { run += x; count += seen.getOrDefault(run - target, 0); seen
```

`long` prefixes always. 2-D prefix sums for grid rectangles; XOR prefixes for XOR subarrays.

Hash-set tricks

```
Set<Integer> seen = new HashSet<>();
for (int x : a) { if (seen.contains(target - x)) return true; seen.add(x); }
new HashSet<>(list).size() == list.size();
Set<Integer> common = new HashSet<>(aList); common.retainAll(bList);
```

Anagram grouping: key = sorted characters (`new String(sortedChars)`) or a count signature; `groupingBy` on that key.

Graph skeletons

```
List<List<Integer>> adj = new ArrayList<>(); for (int i = 0; i < n; i++) adj.a
// BFS
Deque<Integer> q = new ArrayDeque<>(); int[] dist = new int[n]; Arrays.fill(di
dist[src] = 0; q.offer(src);
while (!q.isEmpty()) { int u = q.poll(); for (int v : adj.get(u)) if (dist[v]
// DFS, iterative (no stack overflow)
Deque<Integer> st = new ArrayDeque<>(); boolean[] seen = new boolean[n]; st.pu
while (!st.isEmpty()) { int u = st.pop(); if (seen[u]) continue; seen[u] = tru
// Dijkstra
PriorityQueue<long[]> pq = new PriorityQueue<>(Comparator.comparingLong(x -> x
```

Grid graphs: `int[] dr = {1, -1, 0, 0}, dc = {0, 0, 1, -1}` and a bounds check. `int[][]` for the grid, `boolean[][]` for visited — not sets of strings.

Memoisation and DP tables

```
Map<Long, Long> memo = new HashMap<>();                    // key: encode (i,
long[][] dp = new long[n + 1][m + 1];  Arrays.fill(dp[0], INF);
```

Prefer arrays to maps for DP over small integer states; encode multi-dimensional keys into one `long`; choose an `INF` that survives addition.

Bits

```
Integer.bitCount(x); Integer.highestOneBit(x); Integer.numberOfTrailingZeros(x
(mask >> i & 1) == 1;  mask | (1 << i);  mask & ~(1 << i);  mask & (mask - 1);
for (int sub = mask; sub > 0; sub = (sub - 1) & mask) { … }
```

`1 << i` is an `int` — `1L << i` for `i ≥ 31`. `>>>` for unsigned shifts.

String building and manipulation

```
new StringBuilder(s).reverse().toString();
sb.setLength(sb.length() - 1);                                  // drop the trail
sb.insert(0, c); sb.deleteCharAt(i); sb.setCharAt(i, c);
String.join(",", list); String.valueOf(chars); new String(chars, from, len);
Character.isLetterOrDigit(c); Character.toLowerCase(c); Character.getNumericVa
```

The sheet, as sentences

Problem shape	Say	Structure	Complexity
count things	frequency map / alphabet array	`HashMap.merge`, `int[26]`	O(n)
k largest / most frequent	min-heap of size k	`PriorityQueue`	O(n log k)
next greater / histogram	monotonic stack	`ArrayDeque`	O(n)
window max/min	monotonic deque	`ArrayDeque`	O(n)
nearest / range queries on sorted keys	tree map navigation	`TreeMap.floorKey`	O(log n)
contiguous subarray with monotone predicate	sliding window	two indexes	O(n)
subarray sums	prefix sums + hash map	`long[]`, `HashMap`	O(n)
pair with property	hash set / two pointers	`HashSet` / sorted array	O(n) / O(n log n)
shortest path unweighted / weighted	BFS / Dijkstra	`ArrayDeque` / `PriorityQueue`	O(V + E) / O(E log V)
overlapping subproblems	memo / DP table	`long[][]`, encoded `long` keys	states × transitions
subsets of ≤ 20	bitmask	`int` masks	$O(2^n \cdot n)$

Interview angle

"k most frequent words?" Frequency map, min-heap of size k keyed on count (ties by word), poll and reverse — O(n log k).

"Longest substring without repeating characters?" Sliding window with last-seen indexes — O(n).

"Count subarrays summing to k?" Running prefix sum and a map of prefix counts — O(n).

"Why `ArrayDeque` and not `Stack`?" `Stack` is a synchronised `Vector` with the wrong API; `ArrayDeque` is faster and idiomatic.

"Deep DFS in Java?" Iterative with an explicit stack to avoid `StackOverflowError`.

Key takeaways

Fifteen shapes; name the shape, then the structure, then the complexity — before writing code.

`merge` / `computeIfAbsent` for counting and grouping; `int[26]` when the alphabet is small.

Min-heap of size k for top-k; monotonic stack/deque for next-greater and window extremes; `TreeMap` navigation for sorted queries.

Prefix sums in `long`; hash sets for pairs; two pointers for sorted input; iterative BFS/DFS on adjacency lists.

Encode DP keys as `long`; pick `INF` that survives addition; `1L << i` past 31.

EXERCISE 1

Top-k frequent words

Read an integer `n`, `n` words, and `k`. Count with a frequency map, then keep the `k` most frequent words with a **min-heap of size k** (`PriorityQueue` ordered by count ascending, ties by word **descending**, so the alphabetically earlier word survives a tie). Print the survivors most frequent first, ties alphabetical, as `<word> <count>`.

Example: `7` then `b a c a b a d` then `2` →

```
a 3
b 2
```

INPUT 1

```
7
b a c a b a d
2
```

OUTPUT 1

```
a 3
b 2
```

INPUT 2

```
4
z y x w
2
```

OUTPUT 2

```
w 1
x 1
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, Integer> freq = new HashMap<>();
        for (int i = 0; i < n; i++) freq.merge(in.next(), 1, Integer::sum);
        int k = in.nextInt();
        Comparator<Map.Entry<String, Integer>> heapOrder =
            Map.Entry.<String, Integer>comparingByValue().thenComparing(Map.Entry.co
        PriorityQueue<Map.Entry<String, Integer>> heap = new PriorityQueue<>(heapOrd
        for (Map.Entry<String, Integer> e : freq.entrySet()) {
            heap.offer(e);
            if (heap.size() > k) heap.poll();
        }
        List<Map.Entry<String, Integer>> top = new ArrayList<>(heap);
        top.sort(heapOrder.reversed());
        for (Map.Entry<String, Integer> e : top) System.out.println(e.getKey() + " '
    }
}
```

EXERCISE 2

Sliding-window maximum

Read an integer `n`, `n` integers, and a window size `w` (`1 ≤ w ≤ n`). Print the maximum of every window of `w` consecutive elements, space-separated, in O(n) using a **monotonic deque of indexes** (values decreasing from front to back; drop indexes that leave the window from the front).

Example: `8` then `1 3 -1 -3 5 3 6 7` then `3` →

```
3 3 5 5 6 7
```

INPUT 1

```
8
1 3 -1 -3 5 3 6 7
3
```

OUTPUT 1

```
3 3 5 5 6 7
```

INPUT 2

```
1
9
1
```

OUTPUT 2

```
9
```

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 w = in.nextInt();
        Deque<Integer> dq = new ArrayDeque<>();
        StringBuilder out = new StringBuilder();
        for (int i = 0; i < n; i++) {
            while (!dq.isEmpty() && dq.peekFirst() ⩽ i - w) dq.pollFirst();
            while (!dq.isEmpty() && a[dq.peekLast()] ⩽ a[i]) dq.pollLast();
            dq.offerLast(i);
            if (i ⩾ w - 1) out.append(a[dq.peekFirst()]).append(' ');
        }
        System.out.println(out.toString().trim());
    }
}
```

QUIZ

1. To keep the k **largest** of n values in a heap you use…

A. A max-heap of size n

B. A min-heap of size k — evict the smallest whenever it grows past k

C. A `TreeSet`

D. A sorted `ArrayList`

2. `counts.merge(word, 1, Integer::sum)` replaces…

A. A `for` loop

B. `counts.put(word, counts.getOrDefault(word, 0) + 1)`

C. `counts.remove(word)`

D. A `TreeMap`

3. The monotonic deque for sliding-window maximum holds…

A. The window's values sorted

B. Indexes whose values decrease front to back; the front is the current maximum

C. All indexes

D. Only the maximum

4. Nearest key ≤ x in a sorted map is…

A. `map.get(x)`

B. `treeMap.floorKey(x)` — O(log n)

C. `headMap(x).size()`

D. A linear scan

5. Counting subarrays with sum `k` in O(n) uses…

A. Two nested loops

B. A running prefix sum and a map from prefix value to how many times it has occurred

C. Sorting

D. A heap

6. For a stack in Java you should use…

A. `java.util.Stack`

B. `ArrayDeque` via `push` / `pop` / `peek`

C. `LinkedList`

D. `Vector`

ANSWER KEY · THE COLLECTIONS IDIOM SHEET

1. B O(n log k); for k near n, just sort.

2. B The frequency-map idiom in one call; atomic on `ConcurrentHashMap` too.

3. B Each index enters and leaves once — O(n) total.

4. B `ceilingKey`, `lowerKey`, `higherKey` are its siblings.

5. B `count += seen.getOrDefault(run - k, 0)`; seed the map with `{0: 1}`; prefixes in `long`.

6. B `Stack` is a synchronised `Vector`; `ArrayDeque` is the idiom (no nulls).

LESSON 5 · 12 MIN

Writing clean solutions under time pressure

Two candidates solve the same problem correctly in the same time. One is hired. The difference is almost always what the interviewer could *see*: a solution structured so its correctness was obvious, names that said what things were, edge cases handled visibly, and a complexity stated without prompting. Clean code in an interview is not about style points — it is how you make forty minutes of thinking legible to someone who has to decide whether they trust you with a codebase. This lesson is the small set of habits that produce that legibility without slowing you down.

Structure: parse, solve, print

```
public static void main(String[] args) throws IOException {
    int n = nextInt();
    int[] a = readArray(n);
    out.println(solve(a));
    out.flush();
}

static long solve(int[] a) { … }          // pure: input in, answer out; no I/
```

Three benefits: `solve` can be tested with a literal array in your head or in a scratch `main`; the interviewer can read the algorithm without wading through parsing; and when the question changes ("now what if the input is a stream?") only `main` changes. Keep helper methods **pure** where you can — a method that both computes and prints is twice as hard to reason about.

Names that carry the invariant

`lo` / `hi` for a binary search, `left` / `right` for two pointers, `count`, `best`, `prefix`, `seen`, `dist` — the conventional names *are* documentation because every reader has met them. Beyond those, a name should tell the reader what is true: `firstUnsortedIndex`, `windowSum`, `remainingBudget`. Avoid `temp`, `data`, `result2`, `flag`, single letters outside loops and mathematics. If a variable needs a comment to explain what it holds, rename it.

Small methods, one job each

`isValid(row, col)`, `swap(a, i, j)`, `charFrequency(s)`, `neighbours(node)` — extracting a five-line helper costs fifteen seconds and removes a nested block from the main logic. In an interview it also lets you *say* the plan: "I'll write a helper that checks whether a placement is valid, then the backtracking." A method longer than a screen is a method the interviewer has stopped following.

Make the edge cases visible

```
if (n == 0) return 0;                    // empty input
if (n == 1) return a[0];                 // single element
```

Guard clauses at the top, one per line, each a sentence. They show the interviewer you thought about the boundary *and* keep the main algorithm free of special cases. Then, while writing the loop, narrate the invariant once: "at the top of each iteration, `windowSum` is the sum of `a[lo..hi-1]`." A stated invariant is how you convince someone (including yourself) that the loop is correct without tracing it.

Prefer the boring construct

A `for` loop over a stream when there is an index or an early exit; a stream when it is a clean filter/map/collect.

`ArrayList`, `HashMap`, `ArrayDeque` — the defaults — unless you can say why not.

Arrays for fixed-size numeric data; `int[]` over `List<Integer>`.

One return type: do not return `-1` from a method that also returns valid negative numbers; use `OptionalInt`, a boolean-and-out-parameter object, or `Long.MIN_VALUE` with a comment.

No premature abstraction: an interface for one implementation, a factory for one class, generics for one type — each costs reading time and buys nothing in forty minutes.

Comments: why, not what

`// shrink the window until the sum fits` explains the *what* of an obvious line and adds noise. `// a[] is sorted, so once a[i] > target no later pair can work` explains *why* the `break` is correct. Comment the non-obvious decision, the invariant, the reason a special case exists. Delete the debugging prints before you say "done"; a stray `System.out.println("here")` is a wrong-answer verdict.

Say the complexity, then the trade-off

"This is O(n log n) for the sort, then O(n) for the sweep, O(n) extra memory for the map. We could get O(n) with counting sort since values are bounded by 10^6, at the cost of a million-element array — worth it only if n is large." That sentence, unprompted, is the single highest-leverage thing you can say. It shows you know what you built and that you considered the alternative — which is the actual job.

Test in your head, in order

- The example from the statement — trace it, not "it looks right".
- The smallest input: empty, one element.
- One case that exercises the loop boundary: window of size exactly `k`, target equal to a single element, `n = k`.
- One adversarial case: all equal, all negative, sorted descending.

Tracing a four-element array through your loop aloud finds most off-by-ones and takes under a minute. If the interviewer offers test input, take it; running their example is never a waste.

When you are stuck

Say so, and say what you know: "I know I need sub-quadratic; sorting gives me order, but I don't yet see how to use it." Then simplify — solve for small n, for the sorted case, for one query — and grow the solution. A brute force written cleanly and *labelled* as the baseline earns partial credit and often reveals the optimisation. Silence earns nothing.

The last thirty seconds

Run the pitfalls list from the previous lesson: overflow, `==`, `split`, sort worst case, recursion depth, `flush`. Remove debug output. Make sure the method signature matches what was asked. Then say "I believe this is complete; the complexity is …; edge cases handled are …" — and stop typing.

Interview angle

"How do you approach a coding problem?" Clarify constraints, state the brute force and its complexity, find the shape (from the idiom sheet), write parse/solve/print with helpers, trace the example and the edge cases, state the final complexity.

"What makes code readable to you?" Names carrying invariants, small pure methods, guard clauses for edge cases, comments that say why.

"What would you do differently in production?" Tests, input validation, error handling, no static state, perhaps the boring loop instead of the clever stream.

"Why separate I/O from logic?" Testability, readability, and adaptability when the input source changes.

Key takeaways

Parse → `solve` (pure) → print; helpers with one job; conventional names, invariant-carrying names elsewhere.

Guard clauses make edge cases visible; state the loop invariant once.

Boring constructs, no speculative abstraction, comments for *why*.

Say the complexity and the trade-off unprompted; trace the example and the smallest inputs aloud.

Stuck: say what you know, write the labelled brute force, grow it. Finish with the pitfalls pass.

EXERCISE 1

Balanced brackets, cleanly

Read lines until end of input. For each line decide whether its brackets `()[]{}` are balanced (other characters are ignored). Print `balanced`, or `unbalanced at <index>` where the index is the position of the first closing bracket that does not match, or — if some opening brackets are never closed —

the position of the **earliest** unclosed opener. Structure the solution with small helpers (`isOpener`, `matches`) and a pure `check(String)` that returns the index or `-1`.

Example input

```
{a[b](c)}
(]
((x)
```

→

```
balanced
unbalanced at 1
unbalanced at 0
```

INPUT 1

```
{a[b](c)}
(]
((x)
```

OUTPUT 1

```
balanced
unbalanced at 1
unbalanced at 0
```

INPUT 2

```
)
```

OUTPUT 2

```
balanced
unbalanced at 0
```

REFERENCE SOLUTION

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

public class Main {
    static boolean isOpener(char c) { return c == '(' || c == '[' || c == '{'; }
    static boolean isCloser(char c) { return c == ')' || c == ']' || c == '}'; }
    static boolean matches(char open, char close) {
        return (open == '(' && close == ')') || (open == '[' && close == ']') || (op
    }

    /** Index of the first offending bracket, or -1 when balanced. */
    static int check(String s) {
        Deque<Integer> openers = new ArrayDeque<>();      // indexes of unmatched op
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (isOpener(c)) openers.push(i);
            else if (isCloser(c)) {
                if (openers.isEmpty() || !matches(s.charAt(openers.peek()), c)) retu
                openers.pop();
            }
        }
        return openers.isEmpty() ? -1 : openers.peekLast();   // the earliest unclos
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) != null) {
            int at = check(line);
            System.out.println(at < 0 ? "balanced" : "unbalanced at " + at);
        }
    }
}
```

EXERCISE 2

Spiral order

Read `r` and `c` and an `r × c` matrix of integers. Print its elements in clockwise spiral order, space-separated, starting from the top-left. Write it with four bounds (`top`, `bottom`, `left`, `right`) and a helper per side, so the loop body reads as the four moves.

Example: `3 4` then `1 2 3 4` / `5 6 7 8` / `9 10 11 12` →

```
1 2 3 4 8 12 11 10 9 5 6 7
```

INPUT 1

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
```

OUTPUT 1

```
1 2 3 4 8 12 11 10 9 5 6 7
```

INPUT 2

```
1 1
7
```

OUTPUT 2

```
7
```

REFERENCE SOLUTION

```
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
        List<Integer> out = new ArrayList<>();
        int top = 0, bottom = r - 1, left = 0, right = c - 1;
        while (top ⩽ bottom && left ⩽ right) {
            for (int j = left; j ⩽ right; j++) out.add(m[top][j]);          // top
            for (int i = top + 1; i ⩽ bottom; i++) out.add(m[i][right]);    // righ
            if (top < bottom) for (int j = right - 1; j ⩾ left; j--) out.add(m[bott
            if (left < right) for (int i = bottom - 1; i > top; i--) out.add(m[i][le
            top++; bottom--; left++; right--;
        }
        StringBuilder sb = new StringBuilder();
        for (int x : out) sb.append(x).append(' ');
        System.out.println(sb.toString().trim());
    }
}
```

QUIZ

1. Separating `solve(int[] a)` from `main`'s I/O mainly buys…

A. Speed

B. Testability, readability, and easy adaptation when the input source changes

C. Fewer imports

D. Nothing

2. A guard clause like `if (n == 0) return 0;` at the top of a method…

A. Is bad style

B. Makes an edge case visible and keeps the main algorithm free of special cases

C. Slows the method

D. Should be in a `finally`

3. The highest-leverage sentence to say unprompted after coding is…

A. "I think it works"

B. The time and space complexity, and the trade-off you considered

C. "Java is verbose"

D. The line count

4. Good comments in a solution explain…

A. What each line does

B. Why a non-obvious decision or invariant holds

C. The language syntax

D. Nothing — no comments

5. When stuck, the right move is…

A. Silence until it clicks

B. Say what you know, write a clean labelled brute force, and grow it

C. Give up

D. Change the problem

ANSWER KEY · WRITING CLEAN SOLUTIONS UNDER TIME PRESSURE

1. B A pure function can be checked in your head and reused when the question changes.

2. B One line per edge case, each a sentence.

3. B It shows you know what you built and that you weighed the alternative.

4. B 'a is sorted, so once a[i] > target no later pair can work' — not 'increment i'.

5. B Partial credit and, often, the optimisation reveals itself from the baseline.

LESSON 6 · 16 MIN

The Java theory drill — thirty questions, thirty answers

Every Java interview has a theory section, and the questions have not changed in a decade. This lesson is the drill: the thirty questions you will be asked, each with the two-sentence answer an interviewer wants to hear and a pointer to the module that explains it in depth. Read it twice a week before interviews. The answers are deliberately compact — the skill being tested is saying the essential thing first, then stopping so the interviewer can ask the follow-up.

Platform and language basics

JDK vs JRE vs JVM? The JVM executes bytecode; the JRE is the JVM plus the standard library; the JDK is the JRE plus compiler and tools. *(Module 1)*

Why is Java "write once, run anywhere"? Source compiles to platform-neutral bytecode; each platform's JVM interprets and JIT-compiles it. *(1)*

Is Java pass-by-value or pass-by-reference? Strictly pass-by-value; for objects the value copied is the reference, so callees can mutate the object but not rebind the caller's variable. *(5, 16)*

Why are strings immutable? Safety (they are everywhere — class names, keys, security), the string pool needs sharing, hash code caching, and thread safety for free. *(3, 16)*

`==` vs `equals`? Reference identity versus logical equality; `equals` for strings and boxed numbers, always. *(3, 20)*

`final`, `finally`, `finalize`? Constant/non-overridable/non-extensible; the always-runs block after `try`; the deprecated GC hook — never use. *(7, 12, 16)*

Checked vs unchecked exceptions? Checked must be declared or caught (recoverable conditions the caller should handle); unchecked (`RuntimeException`) signal programming errors. *(12)*

`String`, `StringBuilder`, `StringBuffer`? Immutable; mutable and fast; mutable and synchronised (legacy). Builder in loops. *(3)*

Autoboxing pitfalls? `Integer` cache −128..127 makes `==` lie; boxing allocates; `remove(int)` vs `remove(Object)`; unboxing `null` is an NPE. *(2, 16)*

`static` means? Belongs to the class, one copy, no `this`; static methods cannot be overridden (they hide). *(7)*

Object orientation

Overloading vs overriding? Same name different parameters, resolved at compile time by static types; same signature in a subclass, dispatched at run time by the object's class. *(5, 8)*

Abstract class vs interface? A class with state and partial implementation, single inheritance; a contract, multiple implementation, `default` methods but no instance state. Prefer interfaces; abstract classes for shared state. *(8, 9)*

Can a constructor be overridden? Called virtually? No — constructors are not inherited; and calling an overridable method from a constructor sees the subclass's fields uninitialised. *(7, 8)*

What is polymorphism in Java? A reference of a supertype invoking the runtime type's override — dynamic dispatch through the vtable. *(8)*

Composition vs inheritance? Has-a versus is-a; prefer composition — inheritance couples to the parent's implementation and breaks encapsulation. *(8)*

What do records give you? Immutable value classes with `equals` / `hashCode` / `toString` /accessors generated; compact constructors for validation. *(10)*

Sealed classes? A fixed set of permitted subtypes, enabling exhaustive switches. *(9, 19)*

Generics and collections

Type erasure? Generics exist only at compile time; at run time `List<String>` is `List` — no `new T[]`, no `instanceof List<String>`, one class for all instantiations. *(13)*

`? extends` vs `? super`? Producer extends (read), consumer super (write) — PECS. *(13)*

How does `HashMap` work? Hash → spread → bucket; chain or tree on collision; resize at 0.75 load; needs consistent `equals` / `hashCode`; O(1) average. *(14)*

`ArrayList` vs `LinkedList`? Array-backed, O(1) index, amortised O(1) append; node-backed, O(n) index, poor cache behaviour — almost always `ArrayList`. *(14)*

`HashMap` vs `TreeMap` vs `LinkedHashMap`? Unordered O(1); sorted O(log n) with navigation; insertion (or access) order O(1). *(14)*

Fail-fast iteration? Modifying a collection while iterating throws `ConcurrentModificationException`; use the iterator's `remove` or `removeIf`. *(14)*

`Comparable` vs `Comparator`? Natural order on the type versus an external order per use; `Integer.compare`, never subtraction. *(9, 20)*

Streams, memory, concurrency

Intermediate vs terminal operations? Lazy, return a stream, do nothing until a terminal; the terminal runs the pipeline once. Streams are single-use. *(15)*

Where do objects and locals live? Heap and thread stack respectively; `StackOverflowError` versus `OutOfMemoryError`. *(16)*

How does garbage collection work? Reachability from roots, generational (young/old), stop-the-world pauses reduced by concurrent collectors (G1 default, ZGC). *(16)*

Thread vs Runnable; `start` vs `run`? Pass a `Runnable` to a `Thread`; `start` spawns a thread that calls `run`; `run` directly is a plain call. *(17)*

`synchronized` vs `volatile`? Mutual exclusion plus visibility, for compound updates; visibility only, for a flag or a published immutable reference. *(17)*

What is a deadlock and how do you avoid it? Threads waiting on each other's locks forever; consistent lock ordering, one lock, or `tryLock` with back-off. *(17)*

The follow-ups they actually ask

After 4: *"Then how does `StringBuilder` exist?"* — a separate mutable class; `String` shares nothing with it.

After 9: *"So is `Integer.valueOf(127) == Integer.valueOf(127)` true?"* — yes; 128, no.

After 11: *"What is printed?"* — a snippet with an overloaded `describe(Animal)` / `describe(Dog)` called with an `Animal` reference to a `Dog`: the `Animal` overload (static type), while `a.speak()` is the `Dog` override (runtime type). Fields are never polymorphic.

After 18: *"Then how does `List<String>.get` return a `String`?"* — the compiler inserts the cast.

After 20: *"What if `hashCode` always returns 1?"* — everything in one bucket; O(n) lookups, or O(log n) once it treeifies (Java 8+, if keys are `Comparable`).

After 25: *"Can you reuse a stream?"* — no; `IllegalStateException`.

After 29: *"Is `volatile int count; count++` safe?"* — no; `AtomicInteger`.

How to answer theory questions

Lead with the one-sentence definition, add the one consequence that matters, stop. If you know a version number or a class name, say it — "`ConcurrentHashMap`, since Java 8 with per-bin locking" beats "there's a concurrent map". Never guess a fact you are unsure of; "I believe it is X, but I would check" is a fine answer and a wrong confident answer is not. When you do not know, connect it to what you do: "I have not used `StampedLock`, but I know `ReadWriteLock` and would expect it to be the faster, non-reentrant variant."

Key takeaways

- Thirty questions cover almost every Java theory round; know the two-sentence answer to each.
- Lead with the definition, add one consequence, stop; name classes and versions.
- Expect the follow-up: the cache boundary, the overload-versus-override snippet, the all-ones `hashCode`.
- Never bluff; connect the unknown to the known.

EXERCISE

Static versus dynamic dispatch

Write `class Animal { String name = "animal"; String speak() { return "..."; } }`, `class Dog extends Animal` (field `name = "dog"`, `speak()` returns `Woof`) and `class Cat extends Animal` (`name = "cat"`, `Meow`). Add three static overloads `describe(Animal)`, `describe(Dog)`, `describe(Cat)` returning `describe(Animal)`, `describe(Dog)` and `describe(Cat)`. Read `n` and `n` words `dog` / `cat`; for each, with `Animal a = …`, print one line:

`<word>: speak=<a.speak()> describe=<describe(a)> cast=<describe(…) after a pattern-instanceof cast to the real type> field=<a.name>`

Predict before running: the override is chosen at run time, the overload at compile time, and the field by the static type.

Example: `2` then `dog cat` →

```
dog: speak=Woof describe=describe(Animal) cast=describe(Dog) field=animal
cat: speak=Meow describe=describe(Animal) cast=describe(Cat) field=animal
```

INPUT 1

```
2
dog cat
```

OUTPUT 1

```
dog: speak=Woof
describe=describe(Animal)
cast=describe(Dog) field=animal
cat: speak=Meow
describe=describe(Animal)
cast=describe(Cat) field=animal
```

INPUT 2

```
1
cat
```

OUTPUT 2

```
cat: speak=Meow
describe=describe(Animal)
cast=describe(Cat) field=animal
```

REFERENCE SOLUTION

```
import java.util.*;

class Animal { String name = "animal"; String speak() { return "..."; } }
class Dog extends Animal { String name = "dog"; @Override String speak() { return "W
class Cat extends Animal { String name = "cat"; @Override String speak() { return "M

public class Main {
    static String describe(Animal a) { return "describe(Animal)"; }
    static String describe(Dog d) { return "describe(Dog)"; }
    static String describe(Cat c) { return "describe(Cat)"; }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String word = in.next();
            Animal a = word.equals("dog") ? new Dog() : new Cat();
            String cast = a instanceof Dog d ? describe(d) : describe((Cat) a);
            System.out.println(word + ": speak=" + a.speak() + " describe=" + descri
        }
    }
}
```

QUIZ

1. JDK, JRE, JVM — in one sentence:

A. Three names for the same thing

B. The JVM runs bytecode; the JRE is JVM + library; the JDK is JRE + compiler and tools

C. The JDK runs bytecode

D. The JRE compiles

2. Java is…

A. Pass-by-reference for objects

B. Strictly pass-by-value; for objects the copied value is the reference

C. Pass-by-name

D. Pass-by-reference for primitives

3. Overloading versus overriding is resolved…

A. Both at run time

B. Overloading at compile time by static argument types; overriding at run time by the object's class

C. Both at compile time

D. By the JIT

4. Abstract class versus interface — the deciding factor is…

A. Speed

B. Whether shared *state* is needed (abstract class) versus a contract with possibly multiple inheritance (interface)

C. The Java version

D. Nothing

5. Type erasure means…

A. Generics are checked at run time

B. Generic type information is removed after compilation; `List<String>` is `List` at run time

C. You can create `new T[]`

D. Generics are slower

6. If `hashCode` returned the same constant for every key, `HashMap.get` would be…

A. O(1)

B. O(n) — one bucket, a chain — or O(log n) once the bin treeifies for `Comparable` keys

C. Impossible

D. O(1) after resize

7. `String` is immutable partly because…

A. It is faster to copy

B. Safety, sharing through the pool, cached hash codes and free thread safety

C. It cannot hold Unicode

D. Java lacks `char`

8. `volatile` versus `synchronized`:

A. Interchangeable

B. Visibility and ordering for one field versus mutual exclusion plus visibility for compound updates

C. `volatile` is a lock

D. `synchronized` only orders reads

ANSWER KEY · THE JAVA THEORY DRILL — THIRTY QUESTIONS, THIRTY ANSWERS

1. B Lead with the definition, add the consequence, stop.

2. B Callees can mutate the object but never rebind the caller's variable.

3. B Hence `describe(Animal)` for an `Animal` reference to a `Dog`, but `Dog.speak()`.

4. B Prefer interfaces; `default` methods cover shared behaviour without state.

5. B So no `instanceof List<String>`, no generic arrays, one class for all instantiations.

6. B Java 8's treeification bounds the damage; a good hash avoids it.

7. B `StringBuilder` is the separate mutable class for building.

8. B A flag or a published immutable reference versus `count++` and check-then-act.

CHECKPOINT · 32 MIN

Final checkpoint — Interview idioms

The last checkpoint of the Java plan. It covers the interview template and complexity budget, the `equals` / `hashCode` / `Comparable` contracts, the thirty pitfalls, the collections idiom sheet, clean-solution habits, and the theory drill — and, because it is the last one, it reaches back across the whole track.

How it works. Fifteen questions and three programs; 70% on the questions and every program accepted clears the module — and, with every other module complete, the plan.

Before you start, make sure you can answer:

- What complexity `n ≤ 10⁵` allows, and where `int` overflows in a typical solution.
- Why overriding `equals` without `hashCode` breaks `HashMap`, and why keys must be immutable.
- `-7 % 3`, `Integer.valueOf(128) == Integer.valueOf(128)`, `"a.b".split(".")`, and the `Arrays.sort(int[])` worst case.
- The idiom for top-k frequent, sliding-window maximum, subarray sums and anagram groups.
- How to structure a solution so its correctness is visible, and what to say about complexity.
- Overload versus override resolution — and why fields are never polymorphic.

The programs are the classics interviewers actually set: two-sum with a hash map and pair counting, a min-stack with O(1) operations, and anagram grouping — each with the idiom sheet's structure and the pitfalls list applied.

EXERCISE 1

Two sum, both ways

Read an integer `n`, `n` integers and a target `t`. Print `pair=<i> <j>` for the first pair found scanning left to right with a hash map from value to its **first** index — that is, the smallest `j` such that `t - a[j]` appeared earlier, with `i` the first index of that value — or `pair=none`. Then print `pairs=<number of index pairs i < j with a[i] + a[j] = t>`, computed in O(n) from a frequency map (handle the case where both values are equal).

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

```
pair=1 4
pairs=3
```

(The 1 at index 4 completes the 5 first seen at index 1. The pairs summing to 6 are (1,4), (3,4) and (2,5).)

INPUT 1

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

OUTPUT 1

```
pair=1 4
pairs=3
```

INPUT 2

```
3
1 2 3
10
```

OUTPUT 2

```
pair=none
pairs=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();
        long[] a = new long[n];
        for (int i = 0; i < n; i++) a[i] = in.nextLong();
        long t = in.nextLong();

        Map<Long, Integer> firstIndex = new HashMap<>();
        String pair = "none";
        for (int j = 0; j < n; j++) {
            Integer i = firstIndex.get(t - a[j]);
            if (i != null) { pair = i + " " + j; break; }
            firstIndex.putIfAbsent(a[j], j);
        }
        System.out.println("pair=" + pair);

        Map<Long, Long> freq = new HashMap<>();
        for (long x : a) freq.merge(x, 1L, Long::sum);
        long pairs = 0;
        for (Map.Entry<Long, Long> e : freq.entrySet()) {
            long x = e.getKey(), f = e.getValue(), y = t - x;
            if (x < y) pairs += f * freq.getOrDefault(y, 0L);
            else if (x == y) pairs += f * (f - 1) / 2;
        }
        System.out.println("pairs=" + pairs);
    }
}
```

EXERCISE 2

Min-stack

Implement a stack with O(1) `push`, `pop`, `top` and `min` using two `ArrayDeque`s (values, and the running minimums). Read an integer `n` and `n` commands: `push <v>`; `pop` (prints nothing, or `empty` if there is nothing to pop); `top` prints the top value or `empty`; `min` prints the current minimum or `empty`.

Example: `7` then `push 5`, `push 2`, `push 7`, `min`, `pop`, `pop`, `min` →

```
2
5
```

INPUT 1

```
7
push 5
push 2
push 7
min
pop
pop
min
```

OUTPUT 1

```
2
5
```

INPUT 2

```
3
pop
top
min
```

OUTPUT 2

```
empty
empty
empty
```

REFERENCE SOLUTION

```java
import java.util.*;

class MinStack {
    private final Deque<Long> values = new ArrayDeque<>();
    private final Deque<Long> mins = new ArrayDeque<>();
    void push(long v) {
        values.push(v);
        mins.push(mins.isEmpty() ? v : Math.min(v, mins.peek()));
    }
    boolean isEmpty() { return values.isEmpty(); }
    void pop() { values.pop(); mins.pop(); }
    long top() { return values.peek(); }
    long min() { return mins.peek(); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        MinStack s = new MinStack();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "push" -> s.push(in.nextLong());
                case "pop" -> { if (s.isEmpty()) System.out.println("empty"); else s
                case "top" -> System.out.println(s.isEmpty() ? "empty" : String.valu
                default -> System.out.println(s.isEmpty() ? "empty" : String.valueOf
            }
        }
    }
}
```

EXERCISE 3

Group anagrams

Read an integer `n` and `n` lower-case words. Group words that are anagrams of each other (key: the word's characters sorted). Print the groups ordered by **size descending, then by their first word alphabetically**, one per line as `<size>: <words in input order, space-separated>`.

Example: `6` then `eat tea tan ate nat bat` →

```
3: eat tea ate
2: tan nat
1: bat
```

INPUT 1

```
6
eat tea tan ate nat bat
```

OUTPUT 1

```
3: eat tea ate
2: tan nat
1: bat
```

INPUT 2

```
1
solo
```

OUTPUT 2

```
1: solo
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static String key(String w) {
        char[] cs = w.toCharArray();
        Arrays.sort(cs);
        return new String(cs);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, List<String>> groups = new LinkedHashMap<>();
        for (int i = 0; i < n; i++) {
            String w = in.next();
            groups.computeIfAbsent(key(w), k -> new ArrayList<>()).add(w);
        }
        List<List<String>> ordered = new ArrayList<>(groups.values());
        ordered.sort(Comparator.<List<String>>comparingInt(List::size).reversed().th
        for (List<String> g : ordered) System.out.println(g.size() + ": " + String.j
    }
}
```

CHECKPOINT QUESTIONS

1. `n ≤ 5 000` in the statement suggests…

A. $O(n^3)$

B. $O(n^2)$ is acceptable (25 million operations)

C. Only $O(n)$

D. $O(2^n)$

2. Predict: `int x = 100000; long y = x * x;` gives…

A. `10000000000`

B. `1410065408` — the multiplication overflowed in `int` before the widening

C. `0`

D. A compile error

3. Both `equals` and `hashCode` must be overridden together because…

A. The compiler requires it

B. Hash collections find the bucket by `hashCode` and only then compare with `equals`

C. `toString` depends on them

D. Records need it

4. `Comparator.comparing(Employee::salary).reversed().thenComparing(Employee::name)` sorts by…

A. Name then salary

B. Salary descending, then name ascending

C. Salary ascending

D. Name descending

5. `Integer.valueOf(128) == Integer.valueOf(128)` and `"ab" == "a" + "b"` are…

A. `false`, `true` — cache ends at 127; constant folding pools the literal

B. `true`, `true`

C. `false`, `false`

D. `true`, `false`

6. `"a,b,,".split(",").length` is…

A. `4`

B. `2` — trailing empty strings are dropped unless the limit is negative

C. `3`

D. `0`

7. Removing elements from an `ArrayList` inside a for-each loop…

A. Works

B. Throws `ConcurrentModificationException` — use `removeIf` or the iterator's `remove`

C. Removes the wrong element silently

D. Is O(1)

8. For k most frequent words the idiom is…

A. Sort everything

B. Frequency map, then a min-heap of size k keyed on count with a word tie-breaker

C. A `TreeMap` of words

D. Two nested loops

9. Sliding-window maximum in O(n) uses…

A. A `PriorityQueue`

B. A monotonic deque of indexes

C. Sorting each window

D. A `TreeMap`

10. The min-stack's second deque stores…

A. The values sorted

B. The minimum of the stack at each depth, so `pop` restores the previous minimum in O(1)

C. Only the global minimum

D. The indexes

11. The key for grouping anagrams is…

A. The word's length

B. The word's characters sorted (or a 26-count signature)

C. The first letter

D. The hash code

12. A deep recursion that might exceed ~10^4 frames in Java should…

A. Be left alone

B. Be made iterative with an explicit stack, or run in a thread with a larger stack size

C. Use `volatile`

D. Use streams

13. With `Animal a = new Dog();` and overloads `f(Animal)` / `f(Dog)`, `f(a)` calls…

A. `f(Dog)`

B. `f(Animal)` — overloads resolve on the static type

C. Both

D. Neither

14. Before saying 'done' you should…

A. Add more features

B. Run the pitfalls pass (overflow, `==`, `split`, sort worst case, recursion depth, flush), remove debug output, and state the complexity

C. Rewrite in streams

D. Ask for the answer

15. The best response to a theory question you are unsure of is…

A. A confident guess

B. "I believe it is X, but I would check" — and connecting it to what you do know

C. Silence

D. Changing the subject

ANSWER KEY · FINAL CHECKPOINT — INTERVIEW IDIOMS

1. B The budget table: $5\,000^2 = 2.5 \times 10^7$ fits under 10^8.

2. B Cast an operand first: `(long) x * x`.

3. B Equal objects in different buckets are invisible to `get` / `contains`.

4. B `reversed()` applies to what precedes it; later `thenComparing` keys are ascending.

5. A Neither result is a reason to use `==`: `equals` for both.

6. B `split(",", -1)` keeps them.

7. B Fail-fast iteration.

8. B O(n log k); for k ≈ n a sort is simpler.

9. B Each index is pushed and popped at most once.

10. B Push `min(v, currentMin)` alongside every value.

11. B Equal keys ⇔ anagrams; `computeIfAbsent` groups them.

12. B `new Thread(null, task, "dfs", 1 << 26).start()` is the pragmatic judge trick.

13. B `a.speak()` would still call `Dog`'s override; fields are never polymorphic either.

14. B Thirty seconds that catches most 'correct but rejected' submissions.

15. B Wrong confident answers cost more than honest uncertainty.