Module 14 of 20

Collections framework

The framework's map and complexities, lists, sets, maps and HashMap internals, queues and priority queues, sorting with comparators, iteration and fail-fast, immutable collections.

The collections framework is the part of Java you use in every program and the part interviewers probe most: how HashMap works, ArrayList versus LinkedList, which structure for a stack, why ConcurrentModificationException happens, how to sort by two fields.

This module covers the whole framework methodically: the interface/implementation split and the complexity table; ArrayList and LinkedList with the List API traps; the three sets and their membership rules; HashMap's buckets, resizing and treeification with the modern merge/computeIfAbsent API, plus TreeMap and LinkedHashMap; ArrayDeque and PriorityQueue for stacks, queues and heaps; declarative sorting with Comparator; iterators and the fail-fast contract with the safe ways to modify; and immutable collections with the Collections and Arrays toolkits.

By the end you pick the right structure by reflex, use the modern map API fluently, and answer the HashMap question with confidence.

CONTENTS

1	The collections framework — the map and how to choose	14 min
2	Lists — ArrayList, LinkedList and the List API	14 min
3	Sets — HashSet, LinkedHashSet and TreeSet	12 min
4	Maps — HashMap internals, TreeMap, LinkedHashMap and the modern API	16 min
5	Queues, deques and PriorityQueue	12 min
6	Sorting, Comparable and Comparator in depth	13 min
7	Iterators, fail-fast behaviour and safe modification	11 min
8	Immutable collections and the utility toolkit	12 min
✓	Checkpoint — Collections framework	26 min

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

LESSON 1 · 14 MIN

The collections framework — the map and how to choose

`java.util` holds the data structures every Java program is built from: lists, sets, maps, queues. They share one design — interfaces that describe *what* a collection does, classes that decide *how*, and a small set of conventions (`equals` / `hashCode`, `Iterable`, fail-fast iterators, optional operations) that let them interoperate. This lesson is the map of the framework and the decision procedure for picking a structure; the following lessons take each family in depth.

The hierarchy

```
Iterable<E>
└── Collection<E>            — size, isEmpty, contains, add, remove, iterator,
    ├── List<E>              — ordered by index, duplicates allowed
    │   ├── ArrayList        — growable array; the default list
    │   ├── LinkedList       — doubly linked; also a Deque
    │   └── (Vector, Stack)  — legacy, synchronised; do not use
    ├── Set<E>               — no duplicates (by equals)
    │   ├── HashSet          — hash table; no order
    │   ├── LinkedHashSet    — hash table + insertion order
    │   └── TreeSet          — red-black tree; sorted (SortedSet, NavigableSet
    └── Queue<E>             — head-first processing
        ├── PriorityQueue    — heap; smallest first
        └── Deque<E>         — both ends
            ├── ArrayDeque   — the stack and queue of choice
            └── LinkedList

Map<K, V>                    — NOT a Collection; keys → values, unique keys
├── HashMap                  — hash table; no order
├── LinkedHashMap            — + insertion (or access) order
├── TreeMap                  — sorted by key (SortedMap, NavigableMap)
├── EnumMap                  — array indexed by enum ordinal (Module 10)
├── ConcurrentHashMap        — thread-safe (Module 17)
└── (Hashtable)              — legacy, synchronised; do not use
```

`Map` is separate from `Collection` because a map is not a bag of elements — but its **views** are collections: `keySet()` is a `Set<K>`, `values()` a `Collection<V>`, `entrySet()` a

`Set<Map.Entry<K, V>>`.

Interfaces first

Declare with the interface, construct with the class:

```
List<String> names = new ArrayList<>();
Set<Integer> seen = new HashSet<>();
Map<String, List<Order>> byCustomer = new HashMap<>();
Deque<Character> stack = new ArrayDeque<>();
Queue<Task> queue = new ArrayDeque<>();
```

Method parameters and return types use the interfaces (`List<String>`, `Collection<? extends T>`); a public method returning `ArrayList<String>` promises an implementation detail forever. The one common exception is a private field where the concrete type's extra methods are needed (`LinkedHashMap.removeEldestEntry`, `TreeMap.firstKey`) — declare it as the narrowest interface that has them (`NavigableMap`).

Choosing a structure

Need	Use	Why
A sequence, index access, appends	`ArrayList`	O(1) get/append; cache-friendly
A stack or queue	`ArrayDeque`	O(1) at both ends; faster than `LinkedList`, and `Stack` is legacy
Unique elements, fast membership	`HashSet`	O(1) contains
Unique, remember insertion order	`LinkedHashSet`	+ predictable iteration
Unique, sorted, range queries	`TreeSet`	O(log n), `first` / `floor` / `subSet`
Key → value lookup	`HashMap`	O(1)
Key → value, insertion order or LRU	`LinkedHashMap`	order + `removeEldestEntry`
Key → value, sorted / nearest key	`TreeMap`	`floorKey`, `headMap`, ordered iteration
Enum keys or elements	`EnumMap` / `EnumSet`	array/bit-vector speed
Always-smallest-first processing	`PriorityQueue`	O(log n) insert and poll
Frequent inserts/removes in the middle by iterator	`LinkedList`	O(1) at the iterator — rarely the actual bottleneck
Shared across threads	`ConcurrentHashMap`, `CopyOnWriteArrayList`, `ConcurrentLinkedQueue`	Module 17

Default choices: `ArrayList`, `HashMap`, `HashSet`, `ArrayDeque`. Switch only when a measured or structural need appears (order, sorting, both-ends access).

Complexities to know cold

Operation	ArrayList	LinkedList	HashSet/HashMap	TreeSet/TreeMap	ArrayDeque	Pr
get(i) / by key	O(1)	O(n)	O(1) avg	O(log n)	—	—
add at end / put	O(1) amortised	O(1)	O(1) avg	O(log n)	O(1)	O
add at front	O(n)	O(1)	—	—	O(1)	—
remove by index / key	O(n)	O(n) to find, O(1) to unlink	O(1) avg	O(log n)	O(1) at ends	O
contains	O(n)	O(n)	O(1) avg	O(log n)	O(n)	O
iteration order	index	index	none	sorted	FIFO/LIFO	no

"avg" for hash structures assumes a decent `hashCode`; the worst case degrades to O(n) (or O(log n) once buckets treeify, Java 8+).

The conventions every collection follows

`equals` / `hashCode` **on elements** drive `contains`, `remove(Object)`, set membership and map keys — override them on element/key classes (Module 8) or use records.

`Iterable` — every collection works with for-each and `forEach`; maps via their views.

Fail-fast iterators — modifying a collection while iterating it (other than through the iterator) throws `ConcurrentModificationException` (lesson 7).

Optional operations — immutable collections throw `UnsupportedOperationException` from `add` / `remove`; the interface documents which methods are optional.

Null policy varies: `ArrayList` / `HashMap` / `HashSet` allow nulls; `TreeMap` / `TreeSet` reject null keys (cannot compare); `List.of` / `Map.of` / `ConcurrentHashMap` reject nulls entirely.

`toString` prints contents (`[a, b]`, `{k=v}`) — unlike arrays.

`equals` **between collections** is by contents: two lists with the same elements in order are equal; two sets with the same elements are equal, regardless of implementation class.

The utility classes

`Collections` (sort, reverse, shuffle, max, min, frequency, unmodifiableX, synchronizedX, emptyX, singletonX, nCopies) and `Arrays` (asList, sort, binarySearch, fill, copyOf, stream) hold the static helpers; `List.of`, `Set.of`, `Map.of`, `List.copyOf` (Java 9–10) create immutable collections (lesson 8).

Reading the API docs

Each interface's Javadoc specifies contracts (ordering, null policy, optional operations); each class's documents complexity and iteration order. `ArrayList`'s says "constant amortized time" for `add`; `HashMap`'s explains load factor; `TreeMap`'s says "log(n) time cost for containsKey, get, put and remove". When choosing, read the class summary — it is written to answer exactly this question.

Interview angle

"`ArrayList` vs `LinkedList`?" Array-backed O(1) index access and cache locality vs linked nodes with O(1) insert/remove at a known position; `ArrayList` wins almost always in practice.

"`HashMap` vs `TreeMap`?" O(1) unordered vs O(log n) sorted with navigation methods.

"Why is `Map` not a `Collection`?" It maps keys to values rather than holding elements; its views are collections.

"Which structures reject nulls?" `TreeMap` / `TreeSet` (keys/elements), `List.of` / `Map.of` / `Set.of`, `ConcurrentHashMap`, `ArrayDeque`.

"What should you declare a variable as?" The interface — `List`, not `ArrayList`.

Key takeaways

Interfaces (`List`, `Set`, `Map`, `Queue`, `Deque`) describe behaviour; classes (`ArrayList`, `HashMap`, …) decide performance and order.

Defaults: `ArrayList`, `HashMap`, `HashSet`, `ArrayDeque`; `Linked*` for order, `Tree*` for sorting, `PriorityQueue` for smallest-first.

Know the complexity table and the null policies.

Elements need correct `equals` / `hashCode`; collections compare by contents and print their contents.

EXERCISE 1

Pick the structure

Read an integer `n` and `n` words. Using the right structure for each job, print: `first=<first word>` and `last=<last word>` (from a `List`), `distinct=<count>` (a `HashSet`), `sorted=<distinct words in natural order, space-separated>` (a `TreeSet`), `inOrder=<distinct words in first-seen order>` (a `LinkedHashSet`), and `top=<the most frequent word; ties broken by natural order>` (a `HashMap` of counts).

Example: `5` then `b a b c a` →

```
first=b last=a
distinct=3
sorted=a b c
inOrder=b a c
top=a
```

INPUT 1

```
5
b a b c a
```

OUTPUT 1

```
first=b last=a
distinct=3
sorted=a b c
inOrder=b a c
top=a
```

INPUT 2

```
1
z
```

OUTPUT 2

```
first=z last=z
distinct=1
sorted=z
inOrder=z
top=z
```

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();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());

        System.out.println("first=" + words.get(0) + " last=" + words.get(words.size
        System.out.println("distinct=" + new HashSet<>(words).size());
        System.out.println("sorted=" + String.join(" ", new TreeSet<>(words)));
        System.out.println("inOrder=" + String.join(" ", new LinkedHashSet<>(words))

        Map<String, Integer> counts = new HashMap<>();
        for (String w : words) counts.merge(w, 1, Integer::sum);
        String top = null;
        for (Map.Entry<String, Integer> e : counts.entrySet()) {
            if (top == null || e.getValue() > counts.get(top) || (e.getValue().equal
                top = e.getKey();
            }
        }
        System.out.println("top=" + top);
    }
}
```

EXERCISE 2

Choose by workload

Encode the framework's decision table. Read `n` lines, each a workload described by flags: `kind=<list|set|map|queue>` plus zero or more of `sorted`, `ordered` (insertion order), `lifo`, `priority`. Print the default class for each: list → `ArrayList`; set → `HashSet`, `LinkedHashSet` if ordered, `TreeSet` if sorted; map → `HashMap` / `LinkedHashMap` / `TreeMap` likewise; queue → `ArrayDeque`, or `PriorityQueue` if priority (`lifo` is still `ArrayDeque`, used as a stack). `sorted` beats `ordered` when both appear.

Example: `3` then `kind=set ordered`, `kind=map sorted`, `kind=queue priority` →

```
LinkedHashSet
TreeMap
PriorityQueue
```

INPUT 1

```
3
kind=set ordered
kind=map sorted
kind=queue priority
```

OUTPUT 1

```
LinkedHashSet
TreeMap
PriorityQueue
```

INPUT 2

```
3
kind=list sorted
kind=queue lifo
kind=map
```

OUTPUT 2

```
ArrayList
ArrayDeque
HashMap
```

REFERENCE SOLUTION

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

public class Main {
    static String choose(String kind, Set<String> flags) {
        boolean sorted = flags.contains("sorted"), ordered = flags.contains("ordered
        return switch (kind) {
            case "list" -> "ArrayList";
            case "set" -> sorted ? "TreeSet" : ordered ? "LinkedHashSet" : "HashSet'
            case "map" -> sorted ? "TreeMap" : ordered ? "LinkedHashMap" : "HashMap'
            default -> flags.contains("priority") ? "PriorityQueue" : "ArrayDeque";
        };
    }

    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(in.readLine().trim());
        for (int i = 0; i < n; i++) {
            String[] parts = in.readLine().trim().split("\\s+");
            String kind = parts[0].substring("kind=".length());
            Set<String> flags = new HashSet<>(Arrays.asList(parts).subList(1, parts.
            System.out.println(choose(kind, flags));
        }
    }
}
```

QUIZ

1. Why is `Map` not a subinterface of `Collection`?

A. Historical accident

B. A map holds key→value pairs, not elements; its `keySet` / `values` / `entrySet` are the collections

C. Maps are not iterable

D. Maps are immutable

2. The default choices for list, set, map and stack are…

A. `LinkedList`, `TreeSet`, `TreeMap`, `Stack`

B. `ArrayList`, `HashSet`, `HashMap`, `ArrayDeque`

C. `Vector`, `HashSet`, `Hashtable`, `Stack`

D. `ArrayList`, `LinkedHashSet`, `LinkedHashMap`, `LinkedList`

3. `contains` on an `ArrayList` of a million elements is…

A. O(1)

B. O(log n)

C. O(n) — a linear scan

D. O(n log n)

4. Which collections reject `null` elements or keys?

A. `ArrayList` and `HashMap`

B. `TreeMap`, `TreeSet`, `ArrayDeque`, `List.of` / `Set.of` / `Map.of`, `ConcurrentHashMap`

C. None

D. Only `Set.of`

5. Two `List`s with the same elements in the same order, one an `ArrayList` and one a `LinkedList`, are…

A. Never equal

B. `equals` — list equality is by contents, not class

C. Equal only if both are `ArrayList`

D. Equal only if empty

ANSWER KEY · THE COLLECTIONS FRAMEWORK — THE MAP AND HOW TO CHOOSE

1. B The views bridge the two worlds; the map itself is a different abstraction.

2. B Switch to the ordered or sorted variants only when order or sorting is required.

3. C Lists have no index by value; use a `HashSet` for repeated membership tests.

4. B Tree structures cannot compare null; `ArrayDeque` uses null for "empty"; the immutable factories and concurrent maps are null-hostile by design.

5. B `List.equals` is specified in terms of elements and order.

LESSON 2 · 14 MIN

Lists — ArrayList, LinkedList and the List API

A `List` is an ordered sequence with index access and duplicates allowed — the collection you reach for first. `ArrayList` implements it on a growable array and is the right choice almost always; `LinkedList` implements it on linked nodes and is the right choice in a narrow set of cases that interviewers like to ask about. This lesson covers how each works, the full `List` API with its traps (`remove(int)` versus `remove(Object)`, `subList` views, `Arrays.asList`), and how to use lists efficiently.

`ArrayList` internals

An `ArrayList` holds an `Object[]` and a `size`. `add(x)` writes at `size++`; when the array is full it allocates a new one 1.5× larger and copies (`Arrays.copyOf`). Amortised over many adds, each costs O(1). `get(i)` is a bounds check and an array read — O(1). `add(0, x)` or `remove(0)` shifts every element — O(n). Capacity is separate from size: `new ArrayList<>(10_000)` pre-sizes the array when the count is known and avoids the regrowth copies; `trimToSize()` releases slack.

`LinkedList` internals

A doubly linked chain of `Node{item, prev, next}` with head and tail. `addFirst` / `addLast` / `removeFirst` / `removeLast` are O(1); `get(i)` walks from the nearer end — O(n); every node is a separate heap object (three references plus the header), so memory and cache behaviour are far worse than an array. Its one genuine advantage — O(1) insert/remove *at a position you already hold via an iterator* — is rarely needed. For a stack or queue, `ArrayDeque` beats it; for a list, `ArrayList` does. Joshua Bloch, who wrote it, has said he never uses it.

The `List` API

```
List<String> xs = new ArrayList<>(List.of("a", "b", "c"));
xs.add("d");                // append
xs.add(1, "z");             // insert at index (shifts)
xs.get(0);                  // "a"
xs.set(0, "A");             // replace, returns the old element
xs.remove(1);               // by INDEX: removes "z", returns it
xs.remove("d");             // by OBJECT: removes the first equal element, re
xs.indexOf("b");            // first index or -1; lastIndexOf too
xs.contains("b");           // linear scan
xs.size(); xs.isEmpty(); xs.clear();
xs.addAll(other); xs.removeAll(other); xs.retainAll(other);
xs.subList(1, 3);           // a VIEW of [1, 3)
xs.sort(null);              // natural order (Comparable); or a Comparator
xs.replaceAll(String::toUpperCase);
xs.removeIf(s -> s.isEmpty());
xs.toArray(new String[0]);  // or xs.toArray(String[]::new)
Collections.reverse(xs); Collections.shuffle(xs); Collections.swap(xs, 0, 1);
```

The `remove` trap

```
List<Integer> nums = new ArrayList<>(List.of(10, 20, 30));
nums.remove(1);                       // remove(int index): removes 20
nums.remove(Integer.valueOf(10));     // remove(Object): removes the element
nums.remove((Integer) 30);            // same, by cast
```

With `List<Integer>`, an `int` argument selects `remove(int index)` by exact match (Module 5) — passing the *value* 1 removes the element at index 1. Box explicitly to remove by value.

`subList` is a view

```
List<Integer> big = new ArrayList<>(List.of(1, 2, 3, 4, 5));
List<Integer> mid = big.subList(1, 4);     // [2, 3, 4], backed by big
mid.set(0, 99);                            // big is now [1, 99, 3, 4, 5]
mid.clear();                               // big is now [1, 5] — the idiom fo
big.add(6);                                // structural change to big…
mid.size();                                // …invalidates mid: ConcurrentModi
```

`subList` shares storage with the parent. It is the efficient way to clear or operate on a range, and a trap if you keep it around while modifying the parent. Copy with `new ArrayList<>(big.subList(1, 4))` when you want independence.

Building lists

```
List<String> a = new ArrayList<>();                 // growable, empty
List<String> b = new ArrayList<>(existing);         // a growable COPY
List<String> c = List.of("x", "y");                 // immutable, no nulls (Ja
List<String> d = List.copyOf(existing);             // immutable copy (returns
List<String> e = Arrays.asList("x", "y");           // fixed-size VIEW over ar
List<String> f = Collections.nCopies(3, "-");       // immutable [-, -, -]
List<String> g = Collections.emptyList();           // immutable, typed empty
List<String> h = stream.toList();                   // Java 16: unmodifiable;
```

The two traps: `List.of` throws on `null` elements and on any mutation; `Arrays.asList` is backed by the array (writes go through both ways) and throws on size changes. Wrap either in `new ArrayList<>(...)` when you need a mutable list.

Iterating and modifying

```
for (String s : xs) if (s.isEmpty()) xs.remove(s);          // ConcurrentModi
xs.removeIf(String::isEmpty);                                 // right
for (Iterator<String> it = xs.iterator(); it.hasNext(); ) { if (it.next().isEr
for (int i = xs.size() - 1; i ⩾ 0; i--) if (xs.get(i).isEmpty()) xs.remove(i)
```

`ListIterator` adds `hasPrevious` / `previous`, `set`, `add` and `nextIndex` — the tool for editing a list in place while walking it.

Equality and hashing

Two lists are `equals` when they have the same elements in the same order, whatever their classes (`new ArrayList<>(List.of(1)).equals(new LinkedList<>(List.of(1)))` is true). `hashCode` is defined from the elements, so a list can be a map key — as long as it is not mutated afterwards. Prefer immutable `List.of` for keys.

Performance habits

Pre-size when you know the count; avoid `add(0, x)` in loops (use `ArrayDeque` or build reversed and `Collections.reverse`).

`contains` / `indexOf` are linear: for many membership tests, put the elements in a `HashSet`.

Iterate `LinkedList` with for-each, never `get(i)` in a loop (O(n²)).

`toArray(new T[0])` is as fast as pre-sizing on modern JVMs and simpler.

`List<Integer>` boxes; for numeric bulk data use `int[]` or a primitive-specialised library.

Interview angle

"When would you use `LinkedList`?" Almost never; when you need O(1) insert/remove via an iterator in the middle, or as a `Deque` — but `ArrayDeque` is faster for that.

"How does `ArrayList` grow?" By 1.5× when full, copying the array; amortised O(1) per add.

"`list.remove(1)` on a `List<Integer>`?" Removes index 1 — the `int` overload wins.

"Is `subList` a copy?" No — a view; structural changes to the parent invalidate it.

"`Arrays.asList` vs `List.of`?" Fixed-size view over the array (writes through, allows null) vs immutable, null-hostile copy.

Key takeaways

`ArrayList`: array-backed, O(1) get/append, O(n) insert at front; pre-size when you can. `LinkedList`: rarely the right list.

`remove(int)` vs `remove(Object)`; `subList` is a live view; `Arrays.asList` is fixed-size; `List.of` is immutable and rejects nulls.

Remove during iteration with `removeIf`, an `Iterator`, or a backwards index loop.

Lists compare and hash by contents in order.

EXERCISE 1

List surgery

Read an integer `n` and `n` integers into an `ArrayList<Integer>`. Then read an integer `v` and do, in order: remove the element **at index** 1 (if it exists); remove the first occurrence of the **value** `v` (using `Integer.valueOf`); replace every remaining element with its double via `replaceAll`; clear the sublist `[0, 1)` if the list has at least 2 elements; finally `removeIf` every element greater than 100. Print the list with `toString` after each of the five steps.

Example: `4` then `5 6 7 60`, `v = 7` →

```
[5, 7, 60]
[5, 60]
[10, 120]
[120]
[]
```

INPUT 1

```
4
5 6 7 60
7
```

OUTPUT 1

```
[5, 7, 60]
[5, 60]
[10, 120]
[120]
[]
```

INPUT 2

```
1
9
9
```

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();
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) list.add(in.nextInt());
        int v = in.nextInt();

        if (list.size() > 1) list.remove(1);
        System.out.println(list);
        list.remove(Integer.valueOf(v));
        System.out.println(list);
        list.replaceAll(x -> x * 2);
        System.out.println(list);
        if (list.size() >= 2) list.subList(0, 1).clear();
        System.out.println(list);
        list.removeIf(x -> x > 100);
        System.out.println(list);
    }
}
```

EXERCISE 2

Merge two sorted lists

Read two sorted integer lists (`n` values, then `m` values). Merge them into one sorted `ArrayList` in O(n + m) with two indexes — no sorting call. Print `merged=<list>`, then `middle=<merged.subList(1, size - 1)>` when the merged list has at least 3 elements (else `middle=[]`), and `indexOf7=<merged.indexOf(7)>`.

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

```
merged=[1, 3, 4, 7, 9]
middle=[3, 4, 7]
indexOf7=3
```

INPUT 1

```
3
1 4 9
2
3 7
```

OUTPUT 1

```
merged=[1, 3, 4, 7, 9]
middle=[3, 4, 7]
indexOf7=3
```

INPUT 2

```
1
5
1
5
```

OUTPUT 2

```
merged=[5, 5]
middle=[]
indexOf7=-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();
        List<Integer> a = new ArrayList<>();
        for (int i = 0; i < n; i++) a.add(in.nextInt());
        int m = in.nextInt();
        List<Integer> b = new ArrayList<>();
        for (int i = 0; i < m; i++) b.add(in.nextInt());
        List<Integer> merged = new ArrayList<>(n + m);
        int i = 0, j = 0;
        while (i < n && j < m) merged.add(a.get(i) ⩽ b.get(j) ? a.get(i++) : b.get(
        while (i < n) merged.add(a.get(i++));
        while (j < m) merged.add(b.get(j++));
        System.out.println("merged=" + merged);
        System.out.println("middle=" + (merged.size() ⩾ 3 ? merged.subList(1, merge
        System.out.println("indexOf7=" + merged.indexOf(7));
    }
}
```

QUIZ

1. `ArrayList.add` at the end is O(1) *amortised* because…

A. It never resizes

B. Occasional 1.5× resizes are spread over many cheap appends

C. It uses a linked list

D. It pre-allocates a million slots

2. `List<Integer> l; l.remove(2);` removes…

A. The value 2

B. The element at index 2

C. Both

D. Nothing

3. `list.subList(1, 3)` returns…

A. A copy

B. A live view backed by the list

C. An immutable list

D. An array

4. `Arrays.asList("a", "b").add("c")` throws…

A. Nothing

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

C. `IndexOutOfBoundsException`

D. `NullPointerException`

5. The practical reason to avoid `LinkedList` for random access is…

A. It is deprecated

B. `get(i)` walks nodes — O(n) — and each node is a separate object with poor cache behaviour

C. It cannot hold nulls

D. It is not a `List`

ANSWER KEY · LISTS — ARRAYLIST, LINKEDLIST AND THE LIST API

1. B Each resize copies the array, but the total copying over n appends is O(n).

2. B An `int` argument matches `remove(int index)` exactly. Use `remove(Integer.valueOf(2))` for the value.

3. B Writes go through; structural changes to the parent invalidate the view.

4. B `set` is allowed; size changes are not. Wrap in `new ArrayList<>(...)` to grow.

5. B `ArrayList` wins on almost every workload; `ArrayDeque` wins for deque use.

LESSON 3 · 12 MIN

Sets — HashSet, LinkedHashSet and TreeSet

A `Set` holds each element at most once, where "once" is decided by `equals` (and `hashCode`, or `compareTo` for tree sets). Three implementations cover almost every need: `HashSet` for speed, `LinkedHashSet` when order of insertion matters, `TreeSet` when elements must be sorted or you need "the smallest element ≥ x". This lesson covers how each decides membership, the set-algebra operations, and the navigation API that makes `TreeSet` worth its logarithm.

Membership is `equals` / `hashCode` (or `compareTo`)

```
Set<Point> seen = new HashSet<>();
seen.add(new Point(1, 2));
seen.contains(new Point(1, 2));          // true only if Point overrides equals
```

`HashSet` is a `HashMap` with dummy values: `add` hashes the element to a bucket and compares with `equals` against what is there. A class without a proper `hashCode` puts equal objects in different buckets and the set fills with duplicates. Records and strings are safe; hand-written classes must override both (Module 8).

`TreeSet` never calls `equals` or `hashCode` — it uses `compareTo` (or its `Comparator`): two elements comparing as 0 are the *same* element. `new TreeSet<>(List.of(new BigDecimal("2.0"), new BigDecimal("2.00"))).size()` is 1, while a `HashSet` holds 2.

The three implementations

	`HashSet`	`LinkedHashSet`	`TreeSet`
add/contains/remove	O(1) avg	O(1) avg	O(log n)
Iteration order	unspecified, may change on resize	insertion order	sorted
Nulls	one allowed	one allowed	not allowed
Needs	`equals` / `hashCode`	`equals` / `hashCode`	`Comparable` or a `Comparator`
Extra	—	predictable output, LRU-ish with access order via `LinkedHashMap`	`first`, `last`, `floor`, `ceiling`, `headSet`, `subSet` …

Use `HashSet` by default. Use `LinkedHashSet` when the set is shown to a user or compared in tests — unordered output is a source of flaky tests. Use `TreeSet` when you need sorting or range queries; if you only need the elements sorted *once* at the end, a `HashSet` plus one sort is cheaper.

Set algebra

```
Set<String> a = new HashSet<>(List.of("x", "y", "z"));
Set<String> b = Set.of("y", "z", "w");

Set<String> union = new HashSet<>(a); union.addAll(b);           // {x, y, z,
Set<String> inter = new HashSet<>(a); inter.retainAll(b);        // {y, z}
Set<String> diff  = new HashSet<>(a); diff.removeAll(b);         // {x}
boolean subset = b.containsAll(inter);                           // true
boolean disjoint = Collections.disjoint(a, Set.of("q"));         // true
```

The bulk operations mutate their receiver, so copy first when the original must survive. Note `add` and `remove` return `boolean` — whether the set changed — which is the idiom for "have I seen this before":

```
if (!seen.add(item)) System.out.println("duplicate: " + item);
```

`TreeSet` navigation

```
TreeSet<Integer> t = new TreeSet<>(List.of(10, 20, 30, 40));
t.first(); t.last();                  // 10, 40
t.floor(25); t.ceiling(25);           // 20, 30   — greatest ≤ / least ≥
t.lower(20); t.higher(20);            // 10, 30   — strictly below / above
t.headSet(30);                        // [10, 20]          — view, exclusive
t.tailSet(30);                        // [30, 40]          — view, inclusive
t.subSet(15, 35);                     // [20, 30]          — view [from, to)
t.pollFirst(); t.pollLast();          // remove and return the ends
t.descendingSet();                    // a reversed view
new TreeSet<>(Comparator.reverseOrder());          // custom order
new TreeSet<>(String.CASE_INSENSITIVE_ORDER);      // "a" and "A" are one ele
```

`floor` / `ceiling` in O(log n) are the reason to choose a tree: nearest-neighbour lookups, interval scheduling, "next available slot". The range views are live — modifying them modifies the set, and adding outside the range throws.

Common patterns

```
// dedupe preserving first occurrence
List<String> unique = new ArrayList<>(new LinkedHashSet<>(items));

// fast membership for a filter
Set<String> stop = Set.of("the", "a", "an");
words.removeIf(stop::contains);

// count distinct
long distinct = new HashSet<>(items).size();

// first duplicate
Set<Integer> seen = new HashSet<>();
for (int x : nums) if (!seen.add(x)) return x;

// sorted unique output
new TreeSet<>(words)     // iterate in order
```

Mutating elements — do not

A `HashSet` locates elements by the hash computed at insertion; mutate a field that participates in `hashCode` and the element is in the wrong bucket — `contains` says false, `remove` fails, and the set holds a ghost. Same for `TreeSet` with fields that affect `compareTo`. Set elements (and map keys) should be immutable: strings, numbers, records, enums.

Immutable sets

`Set.of(...)` (Java 9): immutable, no nulls, no duplicates (throws `IllegalArgumentException` on a duplicate literal), iteration order deliberately randomised per JVM run so nobody depends on it. `Set.copyOf(collection)` copies; `Collections.unmodifiableSet(s)` is a read-only *view* of a live set.

Interview angle

"How does `HashSet` check membership?" `hashCode` picks the bucket, `equals` confirms — so both must be overridden consistently.

"`HashSet` vs `TreeSet`?" O(1) unordered vs O(log n) sorted with navigation; `TreeSet` needs comparability and rejects null.

"How do you dedupe a list while keeping order?" `new ArrayList<>(new LinkedHashSet<>(list))`.

"What does `add` return?" Whether the element was absent — the idiom for duplicate detection.

"Why is `Set.of` iteration order unpredictable?" Deliberately randomised so code cannot depend on it.

Key takeaways

`HashSet` (fast, unordered), `LinkedHashSet` (insertion order), `TreeSet` (sorted, navigable, no nulls).

Hash sets need `equals` + `hashCode`; tree sets use `compareTo` only — and disagree with hash sets when it is inconsistent with `equals`.

`addAll` / `retainAll` / `removeAll` on a copy for union/intersection/difference; `add` returns whether it was new.

`floor` / `ceiling` / `subSet` are the tree's payoff. Never mutate elements while they are in a set.

EXERCISE 1

Set algebra and navigation

Read two lines of integers (space-separated; the first token of each is the count). Compute with sets: `union`, `intersection` and `difference` (A minus B), printing each as a sorted, space-separated list (use `TreeSet` for the output order; print an empty line for an empty set). Then read an integer `x` and print `floor=<greatest element of the union ≤ x or none> ceiling=<least element ≥ x or none>`.

Example: `3 1 5 9` / `2 5 7`, `x = 6` →

```
1 5 7 9
5
1 9
floor=5 ceiling=7
```

INPUT 1

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

OUTPUT 1

```
1 5 7 9
5
1 9
floor=5 ceiling=7
```

INPUT 2

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

OUTPUT 2

```
1 2
1 2

floor=none ceiling=1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static List<Integer> readSet(Scanner in) {
        int n = in.nextInt();
        List<Integer> xs = new ArrayList<>();
        for (int i = 0; i < n; i++) xs.add(in.nextInt());
        return xs;
    }

    static String show(Set<Integer> s) {
        StringBuilder sb = new StringBuilder();
        for (int v : new TreeSet<>(s)) { if (sb.length() > 0) sb.append(' '); sb.app
        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Set<Integer> a = new HashSet<>(readSet(in));
        Set<Integer> b = new HashSet<>(readSet(in));
        int x = in.nextInt();

        Set<Integer> union = new HashSet<>(a); union.addAll(b);
        Set<Integer> inter = new HashSet<>(a); inter.retainAll(b);
        Set<Integer> diff = new HashSet<>(a); diff.removeAll(b);
        System.out.println(show(union));
        System.out.println(show(inter));
        System.out.println(show(diff));

        TreeSet<Integer> t = new TreeSet<>(union);
        Integer floor = t.floor(x), ceiling = t.ceiling(x);
        System.out.println("floor=" + (floor == null ? "none" : floor) + " ceiling="
    }
}
```

EXERCISE 2

Visitors, three ways

Read `n` lines `<day> <visitor>`. Print `firstSeen=<visitors in first-seen order>` (a `LinkedHashSet`), `sorted=<visitors in natural order>` (a `TreeSet`), `everyDay=<visitors present on every distinct day>` in natural order (intersect per-day `HashSet`s with `retainAll`), and `days=<number of distinct days>`.

Example: `4` then `mon zed`, `mon amy`, `tue amy`, `tue bob` →

```
firstSeen=[zed, amy, bob]
sorted=[amy, bob, zed]
everyDay=[amy]
days=2
```

INPUT 1

```
4
mon zed
mon amy
tue amy
tue bob
```

OUTPUT 1

```
firstSeen=[zed, amy, bob]
sorted=[amy, bob, zed]
everyDay=[amy]
days=2
```

INPUT 2

```
2
d1 x
d1 x
```

OUTPUT 2

```
firstSeen=[x]
sorted=[x]
everyDay=[x]
days=1
```

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();
        Set<String> firstSeen = new LinkedHashSet<>();
        Map<String, Set<String>> byDay = new LinkedHashMap<>();
        for (int i = 0; i < n; i++) {
            String day = in.next(), visitor = in.next();
            firstSeen.add(visitor);
            byDay.computeIfAbsent(day, d -> new HashSet<>()).add(visitor);
        }
        System.out.println("firstSeen=" + firstSeen);
        System.out.println("sorted=" + new TreeSet<>(firstSeen));
        Set<String> every = null;
        for (Set<String> s : byDay.values()) {
            if (every == null) every = new TreeSet<>(s);
            else every.retainAll(s);
        }
        System.out.println("everyDay=" + (every == null ? "[]" : every));
        System.out.println("days=" + byDay.size());
    }
}
```

QUIZ

1. `HashSet` decides whether an element is already present using…

A. `compareTo`

B. `hashCode` to find the bucket, then `equals`

C. `==`

D. `toString`

2. `TreeSet` decides duplicates using…

A. `equals`

B. `compareTo` (or its comparator): a result of 0 means the same element

C. `hashCode`

D. Identity

3. `set.add(x)` returns `false` when…

A. The set is full

B. `x` was already present

C. `x` is null

D. Never

4. To dedupe a list while keeping first-seen order…

A. `new HashSet<>(list)`

B. `new ArrayList<>(new LinkedHashSet<>(list))`

C. `new TreeSet<>(list)`

D. `list.distinct()`

5. `TreeSet.floor(25)` on `{10, 20, 30}` returns…

A. 30

B. 20

C. 10

D. null

ANSWER KEY · SETS — HASHSET, LINKEDHASHSET AND TREESET

1. B Elements without a consistent `hashCode` / `equals` pair produce duplicates.

2. B This is why `TreeSet` and `HashSet` can disagree when `compareTo` is inconsistent with `equals`.

3. B The boolean return is the idiom for detecting duplicates in one call.

4. B `LinkedHashSet` remembers insertion order; `HashSet` does not, `TreeSet` sorts.

5. B `floor` is the greatest element ≤ the argument; `ceiling(25)` would be 30.

LESSON 4 · 16 MIN

Maps — HashMap internals, TreeMap, LinkedHashMap and the modern API

`Map<K, V>` is the most used data structure after `List`, and `HashMap` is the one every interviewer asks about: how it stores entries, what happens on collision, why it resizes, why `hashCode` matters. This lesson covers that machinery, the three main implementations, and the Java 8 methods — `getOrDefault`, `merge`, `computeIfAbsent`, `putIfAbsent` — that replaced most of the null-checking boilerplate maps used to require.

`HashMap` internals

A `HashMap` is an array of **buckets** (`Node<K,V>[] table`, length a power of two, default 16). To `put(key, value)`:

1. Compute `h = key.hashCode()`, spread its high bits into the low bits (`h ^ (h >>> 16)`), and take the bucket index `h & (table.length - 1)`.
2. If the bucket is empty, store a new node there.
3. Otherwise walk the bucket's chain comparing `hash` first and then `equals`; replace the value if the key is found, else append a node.
4. If `size` exceeds `capacity × loadFactor` (0.75 by default), **resize**: double the table and redistribute every node.

`get(key)` is the same hash-and-walk, without the insert. With well-distributed hash codes each bucket holds about one node and both are O(1). When many keys share a bucket (bad `hashCode`, or a deliberate collision attack), a chain longer than 8 is converted into a red-black **tree** (Java 8+), bounding the worst case at O(log n) — provided the keys are `Comparable`.

Consequences you must know:

- **`hashCode` and `equals` must agree** (Module 8), or `get` looks in the wrong bucket.
- **Keys must not change** while in the map — the bucket was chosen from the old hash.
- **Iteration order is unspecified** and changes after a resize. Never rely on it.
- **Initial capacity** (`new HashMap<>(expectedSize / 0.75 + 1)`) avoids resizes when the count is known; the default is fine otherwise.

One `null` key is allowed (bucket 0); null values are allowed — which is why `get` returning null is ambiguous (`containsKey` disambiguates).

The three implementations

	`HashMap`	`LinkedHashMap`	`TreeMap`
get/put/remove	O(1) avg	O(1) avg	O(log n)
Order	none	insertion (or access, with a flag)	sorted by key
Null keys	one	one	none
Needs	`equals` / `hashCode` on keys	same	`Comparable` keys or a `Comparator`
Extra	—	`removeEldestEntry` → LRU cache	`firstKey` , `floorKey` , `headMap` , `subMap` , `descendingMap`

`LinkedHashMap` threads a doubly linked list through the entries; with `accessOrder = true` every `get` moves the entry to the end, and overriding `removeEldestEntry` gives a bounded LRU cache in five lines (Module 13's exercise). `TreeMap` is a red-black tree keyed by `compareTo` — the navigation methods are its reason to exist.

The modern API

```
Map<String, Integer> counts = new HashMap<>();

// counting — the old way, with a null check
Integer c = counts.get(word);
counts.put(word, c == null ? 1 : c + 1);

// counting — the modern way
counts.merge(word, 1, Integer::sum);                    // put 1 if absent, e

// reading with a default
int n = counts.getOrDefault(word, 0);                   // no null, no unboxi

// multimap: a list per key, created on first use
Map<String, List<String>> byInitial = new HashMap<>();
byInitial.computeIfAbsent(word.substring(0, 1), k -> new ArrayList<>()).add(wo

// insert only if absent
counts.putIfAbsent("x", 0);                             // returns the existi

// recompute in place
counts.compute("x", (k, v) -> v == null ? 1 : v * 2);
counts.computeIfPresent("x", (k, v) -> v > 100 ? null : v);   // returning nul

// bulk
counts.forEach((k, v) -> System.out.println(k + "=" + v));
counts.replaceAll((k, v) -> v * 10);
counts.entrySet().removeIf(e -> e.getValue() == 0);
```

`merge(key, value, fn)`: if absent, store `value`; else store `fn(old, value)`; a `null` result removes the key. `computeIfAbsent(key, fn)`: if absent (or mapped to null), store and return `fn(key)`; else return the existing value — and the mapping function must not modify the map. These two replace the majority of `if (map.containsKey(...))` code.

Iterating

```
for (Map.Entry<String, Integer> e : counts.entrySet()) {     // the efficient
    e.getKey(); e.getValue(); e.setValue(e.getValue() + 1);    // setValue wri
}
for (String k : counts.keySet()) counts.get(k);             // two lookups pe
for (int v : counts.values()) …
counts.forEach((k, v) -> …);
```

`keySet()`, `values()` and `entrySet()` are **views**: removing from them removes from the map (`counts.keySet().removeIf(...)`, `counts.values().remove(0)` removes one entry with value 0), and adding to them is unsupported. Modifying the map while iterating a view throws `ConcurrentModificationException` — except through the iterator's own `remove` or `entry.setValue`.

`TreeMap` navigation

```
TreeMap<Integer, String> t = new TreeMap<>();
t.firstKey(); t.lastKey(); t.firstEntry();
t.floorKey(25); t.ceilingKey(25); t.lowerKey(20); t.higherKey(20);
t.headMap(30); t.tailMap(30, true); t.subMap(10, 30);       // views
t.pollFirstEntry(); t.descendingMap();
```

Same navigation as `TreeSet`, with entries. Interval lookups, "the price valid at date d" (`floorEntry(d)`), leaderboards and ordered reports are `TreeMap` jobs.

Choosing a key type

Keys must be immutable with correct `equals` / `hashCode` (hash maps) or a consistent `compareTo` (tree maps): `String`, `Integer`, `Long`, enums, records, `UUID`, `LocalDate`. Composite keys are records (`record Cell(int r, int c)`). Never arrays (identity equality), never mutable objects.

`Map.of`, `Map.entry`, copies

`Map.of("a", 1, "b", 2)` (up to 10 pairs) and `Map.ofEntries(Map.entry("a", 1), …)` are immutable, null-hostile, order-randomised. `Map.copyOf` copies; `Collections.unmodifiableMap` wraps. `new HashMap<>(other)` copies into a mutable map.

Interview angle

> *"How does `HashMap` work?"* Hash → bucket index; chain (then tree) of nodes; `equals` to confirm; resize at 0.75 load; O(1) average.

"What happens with a bad `hashCode`?" Everything lands in one bucket: O(n) lookups, or O(log n) once treeified.

"Why is 0.75 the load factor?" A time/space compromise: fewer collisions than 1.0, less waste than 0.5.

"`HashMap` vs `Hashtable` vs `ConcurrentHashMap`?" Unsynchronised / legacy fully synchronised / modern concurrent with striped locking (Module 17).

"How do you count words?" `merge(word, 1, Integer::sum)`.

"Can a `HashMap` have a null key?" One. `TreeMap` and `ConcurrentHashMap` cannot.

Key takeaways

`HashMap`: buckets by spread `hashCode`, chains that treeify, resize at 0.75 — O(1) with good hashes; immutable keys with consistent `equals` / `hashCode`.

`LinkedHashMap` for order and LRU; `TreeMap` for sorted keys and `floorKey`-style navigation; none of the three is thread-safe.

`getOrDefault`, `merge`, `computeIfAbsent`, `putIfAbsent`, `compute` replace null-check boilerplate; a null result from `compute*` / `merge` removes.

Iterate `entrySet()`; the views write through and are fail-fast.

EXERCISE 1

Grouping and counting with the modern API

Read an integer `n` and `n` lines `<city> <name>`. Build (1) a `Map<String, Integer>` of how many people per city using `merge`, (2) a `Map<String, List<String>>` of names per city using `computeIfAbsent`, and (3) a `TreeMap` copy of the counts. Print, for every city in sorted order, `<city> <count>: <names in input order joined by ,>`, then `largest=<the city with the most people; ties by name via TreeMap iteration>`, then `floor=<the TreeMap's floorKey for the string "p", or none>`.

Example: `3` then `paris ada`, `oslo bob`, `paris cy` →

```
oslo 1: bob
paris 2: ada,cy
largest=paris
floor=oslo
```

INPUT 1

```
3
paris ada
oslo bob
paris cy
```

OUTPUT 1

```
oslo 1: bob
paris 2: ada,cy
largest=paris
floor=oslo
```

INPUT 2

```
2
zurich a
zurich b
```

OUTPUT 2

```
zurich 2: a,b
largest=zurich
floor=none
```

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> counts = new HashMap<>();
        Map<String, List<String>> names = new HashMap<>();
        for (int i = 0; i < n; i++) {
            String city = in.next(), name = in.next();
            counts.merge(city, 1, Integer::sum);
            names.computeIfAbsent(city, k -> new ArrayList<>()).add(name);
        }
        TreeMap<String, Integer> sorted = new TreeMap<>(counts);
        String largest = null;
        for (Map.Entry<String, Integer> e : sorted.entrySet()) {
            System.out.println(e.getKey() + " " + e.getValue() + ": " + String.join(
            if (largest == null || e.getValue() > sorted.get(largest)) largest = e.g
        }
        System.out.println("largest=" + largest);
        String floor = sorted.floorKey("p");
        System.out.println("floor=" + (floor == null ? "none" : floor));
    }
}
```

EXERCISE 2

An inverted index

Read lines of text until end of input (line numbers start at 1). Build an inverted index: `TreeMap<String, TreeSet<Integer>>` from each lower-cased word to the set of lines it appears on,

using `computeIfAbsent`. Print `<word>: <lines>` for every word, then `vocabulary=<size>` and `mostSpread=<the word on the most lines; ties by natural order>`.

Example input

```
the cat
The dog
a cat
```

→

```
a: [3]
cat: [1, 3]
dog: [2]
the: [1, 2]
vocabulary=4
mostSpread=cat
```

INPUT 1

```
the cat
The dog
a cat
```

OUTPUT 1

```
a: [3]
cat: [1, 3]
dog: [2]
the: [1, 2]
vocabulary=4
mostSpread=cat
```

INPUT 2

```
solo
```

OUTPUT 2

```
solo: [1]
vocabulary=1
mostSpread=solo
```

REFERENCE SOLUTION

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

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        TreeMap<String, TreeSet<Integer>> index = new TreeMap<>();
        String line;
        int lineNo = 0;
        while ((line = in.readLine()) != null) {
            lineNo++;
            for (String w : line.trim().split("\\s+")) {
                if (w.isEmpty()) continue;
                index.computeIfAbsent(w.toLowerCase(Locale.ROOT), k -> new TreeSet<>
            }
        }
        String best = null;
        for (Map.Entry<String, TreeSet<Integer>> e : index.entrySet()) {
            System.out.println(e.getKey() + ": " + e.getValue());
            if (best == null || e.getValue().size() > index.get(best).size()) best =
        }
        System.out.println("vocabulary=" + index.size());
        System.out.println("mostSpread=" + best);
    }
}
```

QUIZ

1. `HashMap.put` chooses a bucket by…

A. The key's `toString`

B. Spreading `key.hashCode()` and masking with `table.length - 1`

C. The insertion order

D. The key's `compareTo`

2. When a `HashMap` bucket's chain exceeds 8 nodes (Java 8+)…

A. The map throws

B. The chain becomes a red-black tree, bounding lookups at O(log n)

C. The map resizes immediately

D. Nothing changes

3. `counts.merge(word, 1, Integer::sum)` does…

A. Replaces the value with 1

B. Stores 1 if absent, otherwise stores old + 1

C. Removes the key

D. Throws if absent

4. `map.computeIfAbsent(k, key → new ArrayList<>()).add(v)` …

A. Always creates a new list

B. Creates the list only when `k` is absent, then adds to whichever list is mapped

C. Removes `k`

D. Is a compile error

5. Which iteration is most efficient over a `HashMap`?

A. `for (K k : map.keySet()) map.get(k)`

B. `for (Map.Entry<K,V> e : map.entrySet())`

C. `for (V v : map.values()) map.get(v)`

D. Index-based

ANSWER KEY · MAPS — HASHMAP INTERNALS, TREEMAP, LINKEDHASHMAP AND THE MODERN API

1. B The table length is a power of two, so the mask is a cheap modulo.

2. B Treeification protects against bad hash codes and collision attacks.

3. B One call replaces the get-null-check-put sequence for counting.

4. B It is the multimap idiom; the mapping function runs only on a miss.

5. B `entrySet` gives each key and value with no extra lookup; the `keySet` + `get` form does two hashes per entry.

LESSON 5 · 12 MIN

Queues, deques and PriorityQueue

Queues process elements in an order decided by the structure: first-in-first-out for a plain queue, last-in-first-out for a stack, smallest-first for a priority queue. Java's `Queue` and `Deque` interfaces, `ArrayDeque` and `PriorityQueue` cover all three — and the legacy `Stack` class is the one you must know exists so you can avoid it. This lesson covers the interfaces' two method families, the implementations, and the algorithms (BFS, balanced brackets, top-k, scheduling) each is built for.

`Queue`: two method families

Every `Queue` operation comes in a throwing form and a returning form:

Purpose	Throws on failure	Returns special value
Insert	`add(e)` → `IllegalStateException` if full	`offer(e)` → `false`
Remove head	`remove()` → `NoSuchElementException` if empty	`poll()` → `null`
Examine head	`element()` → `NoSuchElementException`	`peek()` → `null`

Use `offer` / `poll` / `peek` for ordinary code — an empty queue is a normal condition, not an exception. The throwing forms are for bounded queues where "full" is a programming error. `poll` returning `null` means queues should not hold null elements (`ArrayDeque` forbids them; `LinkedList` allows and confuses).

`Deque`: both ends

`Deque` (double-ended queue) adds `addFirst` / `addLast`, `offerFirst` / `offerLast`, `pollFirst` / `pollLast`, `peekFirst` / `peekLast`, plus stack aliases `push` (= `addFirst`), `pop` (= `removeFirst`), `peek` (= `peekFirst`).

```
Deque<Integer> stack = new ArrayDeque<>();
stack.push(1); stack.push(2);
stack.peek();            // 2
stack.pop();             // 2

Deque<String> queue = new ArrayDeque<>();
queue.offer("a"); queue.offer("b");    // offer = offerLast
queue.poll();                          // "a" — poll = pollFirst
```

`ArrayDeque` is a circular array: O(1) at both ends, no nulls, no capacity limit (it doubles), faster than `LinkedList` for every deque operation and than `Stack` for every stack operation. It is the answer to "which class for a stack?" and "which class for a queue?".

Why not `java.util.Stack`

`Stack extends Vector`: every method synchronised (slow), inherits `get(i)` / `insertElementAt` that break the stack abstraction, and iterates bottom-to-top (surprising). It survives for compatibility; the Javadoc itself says to use `Deque`. Interviewers ask about it precisely to hear this.

Note the iteration difference: iterating an `ArrayDeque` used as a stack yields top-first (`push` adds at the front); iterating a `Stack` yields bottom-first.

Stack patterns

```
// balanced brackets
Deque<Character> st = new ArrayDeque<>();
for (char c : s.toCharArray()) {
    if ("([{".indexOf(c) >= 0) st.push(c);
    else {
        if (st.isEmpty()) return false;
        char open = st.pop();
        if ("([{".indexOf(open) != ")]}".indexOf(c)) return false;
    }
}
return st.isEmpty();
```

Also: evaluating postfix expressions, undo stacks, DFS without recursion, "next greater element" (a monotonic stack), matching tags. Any recursion can be made iterative with an explicit `Deque` of frames.

Queue patterns

```
// breadth-first search over a grid
Deque<int[]> q = new ArrayDeque<>();
boolean[][] seen = new boolean[rows][cols];
q.offer(new int[] {sr, sc}); seen[sr][sc] = true;
while (!q.isEmpty()) {
    int[] cur = q.poll();
    for (int[] d : DIRS) {
        int nr = cur[0] + d[0], nc = cur[1] + d[1];
        if (nr < 0 || nr ⩾ rows || nc < 0 || nc ⩾ cols || seen[nr][nc] || gr
        seen[nr][nc] = true;
        q.offer(new int[] {nr, nc});
    }
}
```

BFS, level-order tree traversal, task pipelines, sliding-window maximum (a monotonic deque), producer–consumer (with the concurrent `BlockingQueue` — Module 17).

`PriorityQueue`

A binary **heap** over an array: `offer` and `poll` are O(log n), `peek` O(1), and `poll` always returns the **smallest** element by natural order or a supplied comparator.

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder()
PriorityQueue<Task> byDeadline = new PriorityQueue<>(Comparator.comparing(Task

minHeap.offer(5); minHeap.offer(1); minHeap.offer(3);
minHeap.poll();          // 1
minHeap.peek();          // 3
```

Facts that trip people:

Iteration is not sorted. `for (x : pq)` and `pq.toString()` show heap order, not ascending order. To get elements in order, `poll` repeatedly (destructive) or copy and sort.

No nulls; elements must be `Comparable` or a comparator must be given.

`remove(Object)` and `contains` are O(n).

Not thread-safe; `PriorityBlockingQueue` is the concurrent version.

Changing an element's priority while it is in the queue corrupts the heap — remove, mutate, re-add.

Patterns: **top-k** (keep a min-heap of size k: offer each element, poll when size exceeds k — O(n log k)), merging k sorted streams, Dijkstra's algorithm, event simulation ordered by time, scheduling by deadline.

Choosing

Need	Use
Stack (LIFO)	`ArrayDeque` via `push` / `pop` / `peek`
Queue (FIFO)	`ArrayDeque` via `offer` / `poll` / `peek`
Both ends	`ArrayDeque`
Smallest/largest first	`PriorityQueue` (+ `reverseOrder` for largest)
Blocking, between threads	`ArrayBlockingQueue`, `LinkedBlockingQueue` (Module 17)
Legacy code	`Stack`, `LinkedList` — read them, do not write them

Interview angle

"Which class would you use for a stack in Java?" `ArrayDeque` — `Stack` is a synchronised `Vector` with a broken abstraction.

"`add` vs `offer`?" `add` throws when it cannot insert; `offer` returns false. Same for `remove` / `poll` and `element` / `peek`.

"Is a `PriorityQueue` sorted?" Only its head is guaranteed; iteration order is heap order.

"How do you get the k largest elements efficiently?" A min-heap of size k.

"Why does `ArrayDeque` reject null?" `poll` / `peek` use null to mean empty.

Key takeaways

`offer` / `poll` / `peek` for everyday use; `add` / `remove` / `element` throw.

`ArrayDeque` is the stack *and* the queue; `Stack` and `LinkedList` are legacy for these roles.

`PriorityQueue` is a heap: O(log n) offer/poll, smallest first, unsorted iteration, no nulls, do not mutate priorities in place.

Stacks for brackets/DFS/undo; queues for BFS/pipelines; heaps for top-k/scheduling/Dijkstra.

EXERCISE 1

Brackets and top-k

Read an integer `n` and `n` lines. Lines of the form `check <text>` are answered `balanced` or `unbalanced` for `()[]{}` using an `ArrayDeque` stack. Lines of the form `topk <k> <m> <m numbers>` are answered with the `k` largest numbers in **descending** order, space-separated, found with a min-heap `PriorityQueue` of size at most `k`.

Example: `2` then `check ([]{})` and `topk 2 5 4 9 1 7 3` →

```
balanced
9 7
```

INPUT 1

```
2
check ([]{})
topk 2 5 4 9 1 7 3
```

OUTPUT 1

```
balanced
9 7
```

INPUT 2

```
3
check ([)]
check (
topk 3 2 5 1
```

OUTPUT 2

```
unbalanced
unbalanced
5 1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static boolean balanced(String s) {
        Deque<Character> st = new ArrayDeque<>();
        String open = "([{", close = ")]}";
        for (char c : s.toCharArray()) {
            if (open.indexOf(c) >= 0) st.push(c);
            else if (close.indexOf(c) >= 0) {
                if (st.isEmpty() || open.indexOf(st.pop()) != close.indexOf(c)) retu
            }
        }
        return st.isEmpty();
    }

    static List<Integer> topK(int k, int[] xs) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int x : xs) {
            minHeap.offer(x);
            if (minHeap.size() > k) minHeap.poll();
        }
        List<Integer> out = new ArrayList<>();
        while (!minHeap.isEmpty()) out.add(minHeap.poll());
        Collections.reverse(out);
        return out;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = Integer.parseInt(in.nextLine().trim());
        for (int i = 0; i < n; i++) {
            String[] t = in.nextLine().trim().split("\\s+");
            if (t[0].equals("check")) {
                System.out.println(balanced(t.length > 1 ? t[1] : "") ? "balanced" :
            } else {
                int k = Integer.parseInt(t[1]);
                int m = Integer.parseInt(t[2]);
                int[] xs = new int[m];
                for (int j = 0; j < m; j++) xs[j] = Integer.parseInt(t[3 + j]);
                StringBuilder sb = new StringBuilder();
                for (int v : topK(k, xs)) { if (sb.length() > 0) sb.append(' '); sb.
                System.out.println(sb);
            }
        }
    }
}
```

EXERCISE 2

Merge k sorted lists with a heap

Read `k`, then `k` sorted lists (each a length followed by its values). Merge them into one sorted sequence with a `PriorityQueue` of `(value, listIndex, position)` entries — push the head of each list, then repeatedly poll the smallest and push that list's next element. Print the merged values space-separated, then `polls=<total polls>`.

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

```
1 2 3 4 5 6
polls=6
```

INPUT 1

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

OUTPUT 1

```
1 2 3 4 5 6
polls=6
```

INPUT 2

```
2
0
2 4 4
```

OUTPUT 2

```
4 4
polls=2
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    record Entry(int value, int list, int pos) { }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt();
        int[][] lists = new int[k][];
        for (int i = 0; i < k; i++) {
            int len = in.nextInt();
            lists[i] = new int[len];
            for (int j = 0; j < len; j++) lists[i][j] = in.nextInt();
        }
        PriorityQueue<Entry> heap = new PriorityQueue<>(Comparator.comparingInt(Entr
        for (int i = 0; i < k; i++) if (lists[i].length > 0) heap.offer(new Entry(li
        StringBuilder out = new StringBuilder();
        int polls = 0;
        while (!heap.isEmpty()) {
            Entry e = heap.poll();
            polls++;
            out.append(out.length() > 0 ? " " : "").append(e.value());
            if (e.pos() + 1 < lists[e.list()].length) heap.offer(new Entry(lists[e.l
        }
        System.out.println(out.length() == 0 ? "-" : out);
        System.out.println("polls=" + polls);
    }
}
```

QUIZ

1. The recommended class for a stack in modern Java is…

A. `Stack`

B. `ArrayDeque`

C. `LinkedList`

D. `Vector`

2. `poll()` on an empty queue…

A. Throws `NoSuchElementException`

B. Returns `null`

C. Blocks

D. Returns 0

3. Iterating a `PriorityQueue` with for-each gives…

A. Ascending order

B. Heap (array) order — not sorted

C. Descending order

D. Insertion order

4. To find the k largest of n elements in O(n log k)…

A. Sort everything

B. Keep a min-heap of size k, polling the smallest when it grows past k

C. Use a max-heap of size n

D. Use a `HashSet`

5. `ArrayDeque` versus `LinkedList` as a deque…

A. `LinkedList` is faster

B. `ArrayDeque` is faster and uses less memory; it rejects nulls

C. Identical

D. `ArrayDeque` cannot be a queue

ANSWER KEY · QUEUES, DEQUES AND PRIORITYQUEUE

1. B `Stack` is a synchronised `Vector` with a leaky abstraction; `ArrayDeque` is faster and cleaner.

2. B `remove()` is the throwing form. This is also why `ArrayDeque` rejects null elements.

3. B Only `peek` / `poll` guarantee the smallest; drain with `poll` for sorted output.

4. B The heap's root is the k-th largest so far; anything smaller than it can be discarded.

5. B A circular array beats per-element nodes for every deque operation.

LESSON 6 · 13 MIN

Sorting, Comparable and Comparator in depth

Sorting a list of objects in Java is one line once you know the comparator vocabulary — `comparing`, `thenComparing`, `reversed`, `nullsFirst` — and a page of bugs if you do not. This lesson covers the sort entry points, how to build any ordering declaratively, stability, the contract a comparator must satisfy, and binary search on sorted data.

The entry points

```
Collections.sort(list);                       // natural order — elements must b
Collections.sort(list, comparator);
list.sort(null);                              // natural order (Java 8; what Col
list.sort(comparator);
Arrays.sort(array);                           // primitives: dual-pivot quicksor
Arrays.sort(objects, comparator);
list.stream().sorted(comparator).toList();  // a new sorted list; the source i
new TreeSet<>(comparator); new TreeMap<>(comparator); new PriorityQueue<>(comp
```

Object sorts use **TimSort**: O(n log n), **stable** (equal elements keep their relative order), and fast on partially sorted input. Primitive array sorts use quicksort — unstable, which is irrelevant for values that are just numbers.

Natural order: `Comparable`

Implement `Comparable<T>` on types with one obvious order (Module 9): compare the most significant field first with `Integer.compare` / `Double.compare` / `compareTo`, and keep it consistent with `equals`. `String`, the wrappers, `LocalDate`, `BigDecimal`, enums (by ordinal) and `UUID` are all `Comparable`.

Building comparators

```
Comparator<Person> byAge   = Comparator.comparingInt(Person::age);        // p
Comparator<Person> byName  = Comparator.comparing(Person::lastName);
Comparator<Person> byNameCi = Comparator.comparing(Person::lastName, String.CA
Comparator<Person> byAgeDesc = byAge.reversed();
Comparator<Person> full = byNameCi.thenComparing(Person::firstName).thenCompar
Comparator<Person> nicknameLast = Comparator.comparing(Person::nickname, Compa

people.sort(full);
people.sort(Comparator.comparing(Person::score).reversed().thenComparing(Perso
```

- `comparing(keyExtractor)` orders by a `Comparable` key; `comparing(keyExtractor, keyComparator)` supplies how to compare the keys.
- `comparingInt` / `comparingLong` / `comparingDouble` avoid boxing.
- `thenComparing` chains tie-breakers; `reversed()` flips the comparator built so far.
- `nullsFirst` / `nullsLast` wrap a comparator to tolerate null *elements*; for null *keys*, wrap the key comparator as above.
- `naturalOrder()` and `reverseOrder()` are the `Comparable`-based ones.

The inference trap from Module 13: `Comparator.comparing(Person::lastName).reversed()` sometimes fails to infer `Person`; write `Comparator.comparing((Person p) -> p.lastName()).reversed()` or assign the first comparator to a typed variable.

Writing a comparator by hand

```
Comparator<Person> byAgeThenName = (a, b) -> {
    int c = Integer.compare(a.age(), b.age());
    return c != 0 ? c : a.name().compareTo(b.name());
};
```

Rules a comparator must obey (the `Comparator` contract): antisymmetric, transitive, and consistent — `compare(a, b)` must give the same sign every time. Violations produce `IllegalArgumentException: Comparison method violates its general contract!` from TimSort, or silently wrong orders. The classic violation is `a.age() - b.age()` on values that can overflow; use `Integer.compare`. Another is comparing doubles with `<` / `>` and returning 0 for `NaN` inconsistently; use `Double.compare`.

Stability, and why it matters

Stable sorts preserve the input order of equal elements. Sorting by last name a list already sorted by first name yields "last name, then first name" for free — the basis of multi-pass sorting. `List.sort` and `Arrays.sort(Object[])` are stable; `Arrays.sort(int[])` is not (and need not be); `Stream.sorted()` is stable for ordered streams.

Sorting maps

Maps have no sort; sort their entries:

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(counts.entrySet());
entries.sort(Map.Entry.<String, Integer>comparingByValue().reversed().thenComp
// or keep a TreeMap for key order; or stream: counts.entrySet().stream().sort
```

`Map.Entry.comparingByKey()` / `comparingByValue()` (with optional comparators) are the helpers.

Sorting in reverse and by multiple keys with mixed directions

```
list.sort(Comparator.comparingInt(Person::age).reversed().thenComparing(Perso
list.sort(Comparator.comparing(Person::name).thenComparing(Comparator.compari
```

`reversed()` applies to everything chained *before* it; to reverse only a later key, pass a reversed comparator into `thenComparing`.

Binary search on sorted data

```
Collections.sort(list);
int i = Collections.binarySearch(list, target);                // natural orde
int j = Collections.binarySearch(list, target, comparator);    // must be the 
int k = Arrays.binarySearch(sortedArray, target);
```

Binary search is O(log n) and correct only on data sorted by the same ordering. A negative result encodes the insertion point — useful for "first element ≥ target".

Performance notes

Sorting is O(n log n); do it once, not inside a loop. Keep a `TreeMap` / `TreeSet` / `PriorityQueue` if the data must *stay* ordered under inserts.

`comparing(Person::name)` calls the key extractor O(n log n) times; if the key is expensive to compute, precompute it (decorate-sort-undecorate) or cache it.

Prefer `comparingInt` over `comparing` for primitive keys.

`Collections.sort` on a `LinkedList` copies to an array, sorts, and writes back — fine, but another reason not to use `LinkedList`.

Interview angle

"How do you sort by two fields?" `Comparator.comparing(a).thenComparing(b)`, or a hand-written two-step compare.

"What algorithm does Java use?" TimSort for objects (stable), dual-pivot quicksort for primitives.

"What is a stable sort?" Equal elements keep their input order — enables multi-key sorting by successive passes.

"Why not `return a - b` in a comparator?" Overflow flips the sign for large values of opposite sign.

"What does `reversed()` reverse?" Everything chained before it.

Key takeaways

`list.sort(cmp)`, `Arrays.sort`, `stream().sorted` — stable TimSort for objects.

Build orderings with `comparing` / `comparingInt`, `thenComparing`, `reversed`, `nullsFirst/Last`, `naturalOrder/reverseOrder`; type the first lambda when inference fails.

Comparators must be consistent and transitive: `Integer.compare` / `Double.compare`, never subtraction.

Sort map entries with `Map.Entry.comparingByValue/Key`; binary search only with the same ordering.

EXERCISE 1

Sort three ways

Read an integer `n` and `n` lines `<name> <dept> <salary>`. Print the names in three orders, each on one line space-separated: (1) by salary descending then name ascending; (2) by department ascending (case-insensitive) then salary ascending; (3) natural order of a record `Employee implements Comparable<Employee>` defined as name ascending. Use `Comparator.comparing` / `thenComparing` / `reversed` for the first two and `Collections.sort` for the third.

Example: `3` then `bob ENG 100`, `ada eng 100`, `cy ops 90` →

```
ada bob cy
bob ada cy
ada bob cy
```

INPUT 1

```
3
bob ENG 100
ada eng 100
cy ops 90
```

OUTPUT 1

```
ada bob cy
bob ada cy
ada bob cy
```

INPUT 2

```
2
z a 1
y b 2
```

OUTPUT 2

```
y z
z y
y z
```

REFERENCE SOLUTION

```java
import java.util.*;

record Employee(String name, String dept, int salary) implements Comparable<Employee
    @Override public int compareTo(Employee o) { return name.compareTo(o.name); }
}

public class Main {
    static String names(List<Employee> es) {
        StringBuilder sb = new StringBuilder();
        for (Employee e : es) { if (sb.length() > 0) sb.append(' '); sb.append(e.nam
        return sb.toString();
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Employee> es = new ArrayList<>();
        for (int i = 0; i < n; i++) es.add(new Employee(in.next(), in.next(), in.nex

        List<Employee> a = new ArrayList<>(es);
        a.sort(Comparator.comparingInt(Employee::salary).reversed().thenComparing(Em
        System.out.println(names(a));

        List<Employee> b = new ArrayList<>(es);
        b.sort(Comparator.comparing(Employee::dept, String.CASE_INSENSITIVE_ORDER).t
        System.out.println(names(b));

        List<Employee> c = new ArrayList<>(es);
        Collections.sort(c);
        System.out.println(names(c));
    }
}
```

EXERCISE 2

Stable, and null-safe

Read `n` lines `<name> <score>` where the score may be `-` meaning unknown (`null`). Sort by score **descending** with unknown scores last (`Comparator.comparing(Player::score, Comparator.nullsLast(Comparator.reverseOrder()))`) — and because `List.sort` is stable, players with equal scores stay in input order. Print `<name> <score or ->` per line, then `unknown=<count>`.

Example: `4` then `ann 80`, `bob -`, `cy 90`, `dee 80` →

```
cy 90
ann 80
dee 80
bob -
unknown=1
```

INPUT 1

```
4
ann 80
bob -
cy 90
dee 80
```

OUTPUT 1

```
cy 90
ann 80
dee 80
bob -
unknown=1
```

INPUT 2

```
2
x -
y -
```

OUTPUT 2

```
x -
y -
unknown=2
```

REFERENCE SOLUTION

```java
import java.util.*;

record Player(String name, Integer score) { }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Player> players = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String name = in.next(), s = in.next();
            players.add(new Player(name, s.equals("-") ? null : Integer.valueOf(s)))
        }
        players.sort(Comparator.comparing(Player::score, Comparator.nullsLast(Compar
        int unknown = 0;
        for (Player p : players) {
            if (p.score() == null) unknown++;
            System.out.println(p.name() + " " + (p.score() == null ? "-" : p.score()
        }
        System.out.println("unknown=" + unknown);
    }
}
```

QUIZ

1. Java's sort for object arrays and lists is…

A. Quicksort, unstable

B. TimSort, stable, O(n log n)

C. Bubble sort

D. Heapsort

2. `Comparator.comparingInt(Person::age).reversed().thenComparing(Person::name)` sorts by…

A. Name desc, then age

B. Age descending, then name ascending

C. Age ascending, name descending

D. Name only

3. `(a, b) → a.age() - b.age()` is a poor comparator because…

A. It is slow

B. Subtraction can overflow and flip the sign for large values of opposite sign

C. It is not a lambda

D. Ages are strings

4. `Collections.binarySearch(list, key, cmp)` is valid only if…

A. The list is an `ArrayList`

B. The list is sorted by the same comparator

C. The list has no duplicates

D. `key` is a `String`

5. To sort a map's entries by value…

A. Call `map.sort()`

B. Copy `entrySet()` into a list and sort with `Map.Entry.comparingByValue()`

C. Use a `HashMap`

D. Sort the keys

ANSWER KEY · SORTING, COMPARABLE AND COMPARATOR IN DEPTH

1. B Stability preserves the input order of equal elements — the basis of multi-key sorting.

2. B `reversed` flips only what precedes it; the tie-breaker added afterwards is ascending.

3. B `Integer.compare(a.age(), b.age())` is correct for all inputs.

4. B Binary search on data ordered differently returns garbage.

5. B Maps have no sort; `TreeMap` orders by key only.

LESSON 7 · 11 MIN

Iterators, fail-fast behaviour and safe modification

`ConcurrentModificationException` is the exception most Java developers meet in their first month: they remove an element from a list while looping over it. Understanding *why* it happens — the fail-fast iterator and its `modCount` — makes the fix obvious and reveals the four correct ways to modify a collection you are walking. This lesson also covers `Iterator` / `ListIterator` directly, and the map views' iteration rules.

`Iterator`

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    String s = it.next();
    if (s.isBlank()) it.remove();          // remove the element last returned
}
```

For-each is sugar over exactly this loop (minus `remove`). `remove()` is optional (immutable collections throw `UnsupportedOperationException`), may be called once per `next()`, and throws `IllegalStateException` otherwise. `Iterator` is single-pass; `iterator()` gives a fresh one.

`ListIterator` (lists only) adds bidirectional movement (`hasPrevious` / `previous`), positional info (`nextIndex`), `set(e)` to replace the last element returned, and `add(e)` to insert at the cursor — the tool for in-place edits during a walk.

Fail-fast: `modCount`

`ArrayList`, `HashMap`, `HashSet` and friends keep a `modCount` incremented on every **structural modification** (add, remove, clear — anything that changes size; `set` does not count). An iterator records the count when created and checks it on every `next()`; a mismatch throws `ConcurrentModificationException`:

```
for (String s : list) {
    if (s.isBlank()) list.remove(s);      // modCount changes; the NEXT next()
}
```

Two facts that surprise:

The exception comes from the *iterator*, on the *following* `next()` — so removing the second-to-last element and then ending the loop may *not* throw. The behaviour is "best-effort": never rely on it not throwing, and never rely on it throwing.

"Concurrent" is misleading: this is single-threaded. Genuine multi-thread modification also triggers it (and worse, since the check is not synchronised); that is Module 17's problem.

Fail-fast exists so that a bug — iterating stale structure — fails loudly instead of skipping or repeating elements silently.

The four safe ways to modify while iterating

`removeIf(predicate)` — the clearest for removals: `list.removeIf(String::isBlank)`, `map.entrySet().removeIf(e → e.getValue() == 0)`.

The iterator's own `remove()` (or `ListIterator.set` / `add`) — the iterator updates its expected count.

A backwards index loop on a list: `for (int i = list.size() - 1; i ⩾ 0; i--) if (…) list.remove(i);` — removing at `i` does not disturb indices below it.

Collect then apply: gather the elements to remove (or add) into a separate collection and modify after the loop — the only option when you need to *add* while iterating.

For replacements without size change: `list.set(i, x)` inside an index loop, or `replaceAll(fn)`; these are not structural and do not trip the check.

Iterating maps safely

```java
for (Map.Entry<K, V> e : map.entrySet()) {
    e.setValue(newValue);                // fine: not structural
    map.put(e.getKey(), newValue);       // fine for an EXISTING key in HashMap
    map.put(newKey, v);                  // ConcurrentModificationException
    map.remove(e.getKey());              // ConcurrentModificationException
}
map.entrySet().removeIf(e → …);         // the safe removal
map.keySet().removeIf(k → …);
map.replaceAll((k, v) → …);             // safe bulk replacement
```

Fail-safe and weakly consistent iterators

The concurrent collections (`ConcurrentHashMap`, `CopyOnWriteArrayList`, `ConcurrentLinkedQueue`) never throw `ConcurrentModificationException`: their iterators are **weakly consistent** — they reflect the state at some point during iteration and may or may not show later changes — or, for `CopyOnWriteArrayList`, iterate a snapshot. This is the right choice for multi-threaded reads with rare writes, and the wrong choice as a way to "fix" a single-threaded modification bug (use `removeIf`).

`Iterable` and `forEach`

`Iterable.forEach(Consumer)` and `Map.forEach(BiConsumer)` are the lambda forms of iteration; they carry the same fail-fast behaviour and the same rule: do not structurally modify the collection inside the consumer. Streams over collections likewise must not modify the source (the pipeline's behaviour is then undefined).

Iteration order recap

Collection	Order
`List`, `ArrayDeque` (as queue)	index / FIFO
`ArrayDeque` as stack	top first
`LinkedHashSet` / `LinkedHashMap`	insertion (or access)
`TreeSet` / `TreeMap`	sorted
`EnumSet` / `EnumMap`	declaration order
`HashSet` / `HashMap`	unspecified; changes on resize
`PriorityQueue`	heap order — not sorted
`Set.of` / `Map.of`	deliberately randomised per run

Tests that assert on the iteration order of a hash collection are flaky by construction; use ordered collections or sort before asserting.

Interview angle

"What causes `ConcurrentModificationException`?" Structural modification of a collection while an iterator over it is active, detected via `modCount`; usually single-threaded.

"How do you remove elements while iterating?" `removeIf`, the iterator's `remove`, a backwards index loop, or collect-then-remove.

"Is fail-fast guaranteed?" No — best-effort; it may miss a modification near the end.

"Which collections have fail-safe iterators?" The concurrent ones (`ConcurrentHashMap`, `CopyOnWriteArrayList`) — weakly consistent or snapshot iterators.

"Does `set` trigger it?" No — only structural (size-changing) modifications.

Key takeaways

For-each is an `Iterator`; `iterator.remove()` is the only removal the iterator tolerates.

Fail-fast iterators check `modCount` on `next()` — best-effort detection of structural modification, single-threaded included.

Modify safely with `removeIf`, the iterator, a backwards index loop, or collect-then-apply; `set` / `replaceAll` / `entry.setValue` are non-structural.

Concurrent collections are weakly consistent instead; hash-collection order is never something to assert on.

EXERCISE 1

Remove safely, four ways

Read an integer `n` and `n` integers. Remove every **even** number four times from four independent copies of the list, using: (1) `removeIf`; (2) an explicit `Iterator` with `it.remove()`; (3) a backwards index loop; (4) collect-then-`removeAll`. Print the four resulting lists; then demonstrate the failure: attempt the removal in a for-each loop inside a `try`, catching `ConcurrentModificationException`, and print `cme=<true|false>` (whether it was thrown).

Example: `5` then `1 2 3 4 5` →

```
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
cme=true
```

INPUT 1

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

OUTPUT 1

```
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
cme=true
```

INPUT 2

```
3
1 3 5
```

OUTPUT 2

```
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
[1, 3, 5]
cme=false
```

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();
        List<Integer> base = new ArrayList<>();
        for (int i = 0; i < n; i++) base.add(in.nextInt());

        List<Integer> a = new ArrayList<>(base);
        a.removeIf(x -> x % 2 == 0);
        System.out.println(a);

        List<Integer> b = new ArrayList<>(base);
        for (Iterator<Integer> it = b.iterator(); it.hasNext(); ) {
            if (it.next() % 2 == 0) it.remove();
        }
        System.out.println(b);

        List<Integer> c = new ArrayList<>(base);
        for (int i = c.size() - 1; i >= 0; i--) {
            if (c.get(i) % 2 == 0) c.remove(i);
        }
        System.out.println(c);

        List<Integer> d = new ArrayList<>(base);
        List<Integer> evens = new ArrayList<>();
        for (int x : d) if (x % 2 == 0) evens.add(x);
        d.removeAll(evens);
        System.out.println(d);

        List<Integer> e = new ArrayList<>(base);
        boolean cme = false;
        try {
            for (Integer x : e) {
                if (x % 2 == 0) e.remove(x);
            }
        } catch (ConcurrentModificationException ex) {
            cme = true;
        }
        System.out.println("cme=" + cme);
    }
}
```

EXERCISE 2

Edit while iterating, with a ListIterator

Read `n` integers into an `ArrayList`. Walk it once with a `ListIterator<Integer>`: replace every negative value with 0 (`set`), insert `x / 2` **after** every even positive value (`add`), and remove every value greater than 100 (`remove`). Print the resulting list and `size=<n>`. This is the one way to change a list while iterating it without a `ConcurrentModificationException`.

Example: `5` then `-3 8 5 200 2` →

```
[0, 8, 4, 5, 2, 1]
size=6
```

INPUT 1

```
5
-3 8 5 200 2
```

OUTPUT 1

```
[0, 8, 4, 5, 2, 1]
size=6
```

INPUT 2

```
3
101 102 103
```

OUTPUT 2

```
[]
size=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();
        List<Integer> xs = new ArrayList<>();
        for (int i = 0; i < n; i++) xs.add(in.nextInt());
        ListIterator<Integer> it = xs.listIterator();
        while (it.hasNext()) {
            int x = it.next();
            if (x > 100) it.remove();
            else if (x < 0) it.set(0);
            else if (x > 0 && x % 2 == 0) it.add(x / 2);   // inserted after x; the
        }
        System.out.println(xs);
        System.out.println("size=" + xs.size());
    }
}
```

QUIZ

1. `ConcurrentModificationException` in single-threaded code means…

A. Two threads collided

B. The collection was structurally modified while an iterator over it was active

C. The list is full

D. A null was inserted

2. Which modification does **not** trip a fail-fast iterator?

A. `list.add(x)`

B. `list.set(i, x)`

C. `list.remove(0)`

D. `list.clear()`

3. Fail-fast detection is…

A. Guaranteed

B. Best-effort — a modification near the end of iteration may go unnoticed

C. Only for maps

D. Disabled by default

4. `map.entrySet().removeIf(e → e.getValue() == 0)` …

A. Throws

B. Safely removes entries with value 0 from the map through the view

C. Only removes from the view

D. Requires a `TreeMap`

5. `CopyOnWriteArrayList` iterators…

A. Throw on modification

B. Iterate a snapshot and never throw `ConcurrentModificationException`

C. Are faster than `ArrayList`'s

D. Reflect every concurrent change

ANSWER KEY · ITERATORS, FAIL-FAST BEHAVIOUR AND SAFE MODIFICATION

1. B The iterator compares the collection's `modCount` with the value it recorded at creation.

2. B `set` is not structural — the size is unchanged, so `modCount` is not incremented.

3. B Never write code that depends on the exception being thrown, or on it not being thrown.

4. B The views write through, and `removeIf` uses the iterator's own removal.

5. B Right for many readers and rare writers across threads; wrong as a fix for a single-threaded removal bug.

LESSON 8 · 12 MIN

Immutable collections and the utility toolkit

Most collections a program creates are never modified after being built — a set of allowed values, a map of configuration, the result of a query. Making them **immutable** turns that intention into a guarantee: no accidental `add`, safe sharing across threads, safe use as keys, and a clear API contract ("this list will not change under you"). Java offers three mechanisms with different semantics, plus a toolkit of utility methods in `Collections` and `Arrays` that this lesson gathers in one place.

The three kinds of "unmodifiable"

Mechanism	What it is	Nulls	Notes
`List.of`, `Set.of`, `Map.of`, `Map.ofEntries` (Java 9)	A new, truly immutable collection	rejected	`Set.of` / `Map.of` reject duplicate keys; iteration order randomised
`List.copyOf`, `Set.copyOf`, `Map.copyOf` (Java 10)	An immutable *copy* of a collection	rejected	Returns the same instance if the source is already one of these
`Collections.unmodifiableList/Set/Map(x)`	A read-only **view** of a live collection	allowed	Changes to `x` show through the view
`Stream.toList()` (Java 16)	An unmodifiable list	allowed	vs `collect(toList())`, which is mutable in practice but unspecified
`Arrays.asList(arr)`	A fixed-size view over an array	allowed	`set` works (writes to the array); `add` / `remove` throw

All of them throw `UnsupportedOperationException` from mutators; the type is still `List` / `Set` / `Map`, so the compiler cannot warn you — read the source of a collection before assuming you may modify it, and prefer to return immutable collections from APIs so callers cannot corrupt your state.

Choosing

Building a constant: `List.of(...)`, `Set.of(...)`, `Map.of(...)` (up to ten pairs; `Map.ofEntries(Map.entry(k, v), …)` beyond).

Returning a snapshot of internal state: `List.copyOf(items)` — cheap when items is already immutable, safe otherwise.

Exposing internal state as read-only without copying (large, frequently read): `Collections.unmodifiableList(items)` — remember the caller sees later changes.

Need nulls in an immutable collection: `Collections.unmodifiableList(new ArrayList<>(withNulls))`.

Shallow, again

Immutable collections hold *references*; the elements are as mutable as they were. `List.of(new StringBuilder())` is an immutable list of one mutable builder. Immutable elements (strings, records, numbers) make the whole structure immutable in practice.

The `Collections` toolkit

```
Collections.sort(list); Collections.sort(list, cmp);
Collections.reverse(list); Collections.shuffle(list); Collections.shuffle(list
Collections.swap(list, i, j); Collections.rotate(list, k);
Collections.max(coll); Collections.min(coll); Collections.max(coll, cmp);
Collections.frequency(coll, x);          // how many equal to x
Collections.disjoint(a, b);              // no elements in common?
Collections.nCopies(n, x);               // immutable list of n copies
Collections.fill(list, x);               // overwrite every element
Collections.addAll(coll, a, b, c);       // varargs add
Collections.emptyList(); emptySet(); emptyMap();           // typed immutable
Collections.singletonList(x); singleton(x); singletonMap(k, v);   // immutable
Collections.unmodifiableList(x); …Set; …Map; …Collection;
Collections.synchronizedList(x); …Map;   // coarse-locked wrappers (Module 17
Collections.binarySearch(sortedList, key);
Collections.reverseOrder(); reverseOrder(cmp);
```

`Collections.emptyList()` returns one shared immutable instance — the idiomatic "nothing" return before `List.of()` existed; both are fine. `singletonList` remains useful for passing

one element to an API that wants a list without allocating an `ArrayList`.

The `Arrays` bridge

`Arrays.asList(T...)` (fixed-size view), `Arrays.stream(array)`, `Arrays.sort`, `Arrays.fill`, `Arrays.copyOf`, `Arrays.equals`, `Arrays.hashCode`, `Arrays.toString` — Module 6. `list.toArray(new T[0])` / `toArray(T[]::new)` go the other way; `Arrays.asList(intArray)` does not box (one element).

Collection equality and hashing

`List.equals`: same elements in order, any implementations. `Set.equals`: same elements, any implementations. `Map.equals`: same entries. `hashCode` follows: `List` from elements in order, `Set` as the sum of element hashes, `Map` as the sum of entry hashes. So `List.of(1, 2).equals(Arrays.asList(1, 2))` is true, `Set.of(1, 2).equals(new TreeSet<>(List.of(2, 1)))` is true, and collections can be map keys — if they are immutable.

Copies and defensive copies

```
List<String> mutableCopy = new ArrayList<>(source);          // independent, g
List<String> frozen = List.copyOf(source);                   // independent, i
Map<K, V> mapCopy = new HashMap<>(source);
```

A constructor or setter that receives a collection and stores it should copy it (`List.copyOf` when nulls are impossible); a getter that returns internal state should return a copy or an unmodifiable view (Module 7). "Whose list is this?" is the question; copying at boundaries is the answer.

Wrappers and views, summarised

Views are cheap and live: `subList`, `keySet` / `values` / `entrySet`, `headMap` / `tailSet`, `unmodifiableX`, `Arrays.asList`, `descendingMap`. Copies are independent: constructors (`new ArrayList<>(x)`), `copyOf`, `clone`, `toArray`, `stream().toList()`. Knowing which you hold is what prevents both aliasing bugs and needless copying.

Interview angle

"`List.of` vs `Arrays.asList` vs `Collections.unmodifiableList`?" Immutable null-hostile copy / fixed-size array view / read-only view of a live collection.

"How do you return internal state safely?" `List.copyOf` or an unmodifiable view.

"Is an immutable list deeply immutable?" No — elements may still be mutable.

"What does `Collections.emptyList()` return?" A shared immutable empty list; `List.of()` is the modern equivalent.

"Can two different implementations be equal?" Yes — `List` / `Set` / `Map` equality is by contents.

Key takeaways

`List.of` / `Set.of` / `Map.of` and `copyOf`: immutable, no nulls.
`Collections.unmodifiableX`: a read-only view. `Arrays.asList`: fixed-size, writes through.

Return immutable snapshots or views from APIs; copy on the way in.

Immutability is shallow — prefer immutable elements.

`Collections` holds the algorithms (sort, reverse, shuffle, max, frequency, nCopies…); `Arrays` bridges arrays.

Collections compare and hash by contents across implementations.

EXERCISE 1

Views versus copies

Read an integer `n` and `n` words into a mutable `ArrayList`. Create `view = Collections.unmodifiableList(list)`, `copy = List.copyOf(list)` and `fixed = Arrays.asList(list.toArray(new String[0]))`. Then add the word `extra` to the original list and try `view.add("x")`, `copy.add("x")` and `fixed.set(0, "first")`, catching `UnsupportedOperationException` where it occurs. Print: `view=<view.toString()>`, `copy=<copy>`, `fixed=<fixed>`, and `unsupported=<how many of the three attempts threw>`. Also print `frequency=<Collections.frequency(list, first word)>` and `max=<Collections.max(list)>`.

Example: `2` then `b a` →

```
view=[b, a, extra]
copy=[b, a]
fixed=[first, a]
unsupported=2
frequency=1
max=extra
```

INPUT 1

```
2
b a
```

OUTPUT 1

```
view=[b, a, extra]
copy=[b, a]
fixed=[first, a]
unsupported=2
frequency=1
max=extra
```

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();
        List<String> list = new ArrayList<>();
        for (int i = 0; i < n; i++) list.add(in.next());
        String first = list.get(0);

        List<String> view = Collections.unmodifiableList(list);
        List<String> copy = List.copyOf(list);
        List<String> fixed = Arrays.asList(list.toArray(new String[0]));

        list.add("extra");
        int unsupported = 0;
        try { view.add("x"); } catch (UnsupportedOperationException e) { unsupported
        try { copy.add("x"); } catch (UnsupportedOperationException e) { unsupported
        try { fixed.set(0, "first"); } catch (UnsupportedOperationException e) { uns

        System.out.println("view=" + view);
        System.out.println("copy=" + copy);
        System.out.println("fixed=" + fixed);
        System.out.println("unsupported=" + unsupported);
        System.out.println("frequency=" + Collections.frequency(list, first));
        System.out.println("max=" + Collections.max(list));
    }
}
```

EXERCISE 2

The Collections toolkit tour

Read `n` words. Print `frequency(first)=<Collections.frequency of the first word>`, `nCopies=<Collections.nCopies(3, first word)>`, then rotate a copy of the list by 1 with `Collections.rotate` and print `rotated=<…>`, shuffle a copy with `Collections.shuffle(copy, new Random(42))` and print `shuffled=<…>` (a seeded `Random` makes it reproducible), and finally try to `add` to `Collections.unmodifiableList(words)` and print `unmodifiable=<true if it threw>`.

Example: `4` then `a b a c` →

```
frequency(first)=2
nCopies=[a, a, a]
rotated=[c, a, b, a]
shuffled=[c, b, a, a]
unmodifiable=true
```

INPUT 1

```
4
a b a c
```

OUTPUT 1

```
frequency(first)=2
nCopies=[a, a, a]
rotated=[c, a, b, a]
shuffled=[c, b, a, a]
unmodifiable=true
```

INPUT 2

```
1
q
```

OUTPUT 2

```
frequency(first)=1
nCopies=[q, q, q]
rotated=[q]
shuffled=[q]
unmodifiable=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();
        List<String> words = new ArrayList<>();
        for (int i = 0; i < n; i++) words.add(in.next());
        String first = words.get(0);
        System.out.println("frequency(first)=" + Collections.frequency(words, first));
        System.out.println("nCopies=" + Collections.nCopies(3, first));
        List<String> rotated = new ArrayList<>(words);
        Collections.rotate(rotated, 1);
        System.out.println("rotated=" + rotated);
        List<String> shuffled = new ArrayList<>(words);
        Collections.shuffle(shuffled, new Random(42));
        System.out.println("shuffled=" + shuffled);
        boolean threw = false;
        try {
            Collections.unmodifiableList(words).add("z");
        } catch (UnsupportedOperationException e) {
            threw = true;
        }
        System.out.println("unmodifiable=" + threw);
    }
}
```

QUIZ

1. `Collections.unmodifiableList(list)` returns…

A. An immutable copy

B. A read-only view that reflects later changes to `list`

C. A sorted list

D. A synchronised list

2. `List.of("a", null)` …

A. Creates a two-element list

B. Throws `NullPointerException` — the immutable factories reject nulls

C. Creates `[a]`

D. Compiles only with a cast

3. `List.copyOf(x)` when `x` is already a `List.of` list…

A. Copies anyway

B. Returns the same instance — no copy needed for an immutable source

C. Throws

D. Returns null

4. An immutable list of `StringBuilder`s…

A. Prevents changing the builders

B. Prevents changing which builders it holds, not the builders' contents — immutability is shallow

C. Is a compile error

D. Copies the builders

5. `Collections.frequency(list, x)` returns…

A. The index of `x`

B. How many elements equal `x`

C. Whether `x` is present

D. The most common element

ANSWER KEY · IMMUTABLE COLLECTIONS AND THE UTILITY TOOLKIT

1. B For an independent snapshot use `List.copyOf(list)`.

2. B Use `Collections.unmodifiableList(new ArrayList<>(...))` when nulls are required.

3. B This makes defensive `copyOf` calls cheap in the common case.

4. B Immutable elements (strings, records) are what make a structure immutable in practice.

5. B A linear count using `equals`.

CHECKPOINT · 26 MIN

Checkpoint — Collections framework

This checkpoint covers the framework's structure and complexities, lists, sets, maps and `HashMap` internals, queues and `PriorityQueue`, sorting and comparators, iteration and fail-fast behaviour, and immutable collections and utilities.

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

Before you start, make sure you can answer:

- The complexity of `get`, `add` and `contains` on `ArrayList`, `HashSet` and `TreeMap`.
- How `HashMap.put` finds a bucket and what happens on collision and resize.
- Which class to use for a stack, and why not `Stack`.
- Whether iterating a `PriorityQueue` gives sorted order.
- The four safe ways to remove while iterating.
- The difference between `List.of`, `Arrays.asList` and `Collections.unmodifiableList`.

The programs are a word-frequency report using the modern map API and comparators, and a small task scheduler built on a priority queue and a deque.

EXERCISE 1

Word frequency report

Read all of standard input as text. Split into words on non-letters (`[^a-zA-Z]+`), lower-case them, ignore empties. Count with a `HashMap` and `merge`. Print the **top 3** words as `<word> <count>` per line, ordered by count descending then word ascending (sort the entry set with `Map.Entry.comparingByValue().reversed()` and a tie-breaker — or a hand-written comparator), then `distinct=<number of distinct words>`, then `longest=<the longest word; ties by natural order>`.

Example input `the cat and the hat. The end` →

```
the 3
and 1
cat 1
distinct=5
longest=and
```

INPUT 1

```
the cat and the hat. The end
```

OUTPUT 1

```
the 3
and 1
cat 1
distinct=5
longest=and
```

INPUT 2

```
a b b c c c
```

OUTPUT 2

```
c 3
b 2
a 1
distinct=3
longest=a
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) throws java.io.IOException {
        String text = new String(System.in.readAllBytes());
        Map<String, Integer> counts = new HashMap<>();
        for (String w : text.toLowerCase(Locale.ROOT).split("[^a-zA-Z]+")) {
            if (!w.isEmpty()) counts.merge(w, 1, Integer::sum);
        }
        List<Map.Entry<String, Integer>> entries = new ArrayList<>(counts.entrySet()
        entries.sort((a, b) -> {
            int c = Integer.compare(b.getValue(), a.getValue());
            return c != 0 ? c : a.getKey().compareTo(b.getKey());
        });
        for (int i = 0; i < Math.min(3, entries.size()); i++) {
            System.out.println(entries.get(i).getKey() + " " + entries.get(i).getVal
        }
        System.out.println("distinct=" + counts.size());
        String longest = null;
        for (String w : counts.keySet()) {
            if (longest == null || w.length() > longest.length() || (w.length() == l
        }
        System.out.println("longest=" + longest);
    }
}
```

EXERCISE 2

A tiny scheduler

Tasks have a name and a priority (lower number = more urgent). Read an integer `n` and `n` commands: `add <name> <priority>` (into a `PriorityQueue` ordered by priority, ties by name); `run` (poll the most urgent and print `run <name>`, or `idle` if empty); `defer` (poll the most urgent and push it onto an `ArrayDeque` stack of deferred tasks, printing `deferred <name>`, or `idle`); `resume` (pop the last deferred task back into the queue, printing `resumed <name>`, or `none`). After all commands print `pending=<queue size> deferred=<stack size>`.

Example: `5` then `add b 2`, `add a 1`, `defer`, `run`, `resume` →

```
deferred a
run b
resumed a
pending=1 deferred=0
```

INPUT 1

```
5
add b 2
add a 1
defer
run
resume
```

OUTPUT 1

```
deferred a
run b
resumed a
pending=1 deferred=0
```

INPUT 2

```
3
run
defer
resume
```

OUTPUT 2

```
idle
idle
none
pending=0 deferred=0
```

REFERENCE SOLUTION

```
import java.util.*;

record Task(String name, int priority) { }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        PriorityQueue<Task> queue = new PriorityQueue<>(Comparator.comparingInt(Task
        Deque<Task> deferred = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "add" -> queue.offer(new Task(in.next(), in.nextInt()));
                case "run" -> {
                    Task t = queue.poll();
                    System.out.println(t == null ? "idle" : "run " + t.name());
                }
                case "defer" -> {
                    Task t = queue.poll();
                    if (t == null) System.out.println("idle");
                    else { deferred.push(t); System.out.println("deferred " + t.name
                }
                default -> {
                    Task t = deferred.poll();
                    if (t == null) System.out.println("none");
                    else { queue.offer(t); System.out.println("resumed " + t.name())
                }
            }
        }
        System.out.println("pending=" + queue.size() + " deferred=" + deferred.size(
    }
}
```

EXERCISE 3

Meeting rooms

Read `n` meetings `<start> <end>` (integers, end exclusive). Compute the minimum number of rooms so that no two overlapping meetings share a room: sort by start, keep a `PriorityQueue<Integer>` of end times of meetings in progress, free a room when the earliest end is `≤` the next start. Print `rooms=<k>`, then assign rooms greedily in that order — each meeting takes the lowest-numbered free room (a `TreeSet<Integer>` of free rooms) — printing `<start>-<end> room <r>` in start order.

Example: `3` then `0 30`, `5 10`, `15 20` →

```
rooms=2
0-30 room 1
5-10 room 2
15-20 room 2
```

INPUT 1

```
3
0 30
5 10
15 20
```

OUTPUT 1

```
rooms=2
0-30 room 1
5-10 room 2
15-20 room 2
```

INPUT 2

```
2
7 10
2 4
```

OUTPUT 2

```
rooms=1
2-4 room 1
7-10 room 1
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    record Meeting(int start, int end) { }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<Meeting> meetings = new ArrayList<>();
        for (int i = 0; i < n; i++) meetings.add(new Meeting(in.nextInt(), in.nextIr
        meetings.sort(Comparator.comparingInt(Meeting::start).thenComparingInt(Meeti

        // Pass 1: how many rooms.
        PriorityQueue<Integer> ends = new PriorityQueue<>();
        int rooms = 0;
        for (Meeting m : meetings) {
            while (!ends.isEmpty() && ends.peek() <= m.start()) ends.poll();
            ends.offer(m.end());
            rooms = Math.max(rooms, ends.size());
        }
        System.out.println("rooms=" + rooms);

        // Pass 2: assign the lowest free room; a heap of (end, room) frees rooms as
        TreeSet<Integer> free = new TreeSet<>();
        for (int r = 1; r <= rooms; r++) free.add(r);
        PriorityQueue<int[]> busy = new PriorityQueue<>(Comparator.comparingInt(a ->
        for (Meeting m : meetings) {
            while (!busy.isEmpty() && busy.peek()[0] <= m.start()) free.add(busy.pol
            int room = free.pollFirst();
            busy.offer(new int[] { m.end(), room });
            System.out.println(m.start() + "-" + m.end() + " room " + room);
        }
    }
}
```

CHECKPOINT QUESTIONS

1. Which operation is O(1) on an `ArrayList`?

A. `add(0, x)`

B. `get(i)`

C. `contains(x)`

D. `remove(Object)`

2. `HashMap` resizes when…

A. Any bucket has two nodes

B. `size` exceeds `capacity × loadFactor` (0.75 by default)

C. A key is removed

D. Never

3. A key whose `hashCode` changes after insertion into a `HashMap`…

A. Is rehashed automatically

B. Becomes unreachable via `get` because it sits in the bucket of its old hash

C. Throws

D. Is removed

4. `TreeMap` requires keys to be…

A. Hashable

B. `Comparable` (or a `Comparator` supplied) and non-null

C. Strings

D. Immutable records

5. `LinkedHashMap` with `accessOrder = true` and `removeEldestEntry` overridden gives…

A. A sorted map

B. An LRU cache

C. A thread-safe map

D. A multimap

6. `Deque.push` and `Deque.pop` operate at…

A. The tail

B. The head — `push` is `addFirst`, `pop` is `removeFirst`

C. Random positions

D. Both ends

7. Changing an element's priority while it sits in a `PriorityQueue`…

A. Re-heaps automatically

B. Corrupts the heap ordering — remove, modify, re-add instead

C. Is fine

D. Throws

8. A stable sort guarantees…

A. O(n) time

B. Equal elements keep their relative input order

C. No duplicates

D. Ascending order only

9. `Comparator.nullsFirst(Comparator.naturalOrder())`…

A. Throws on null

B. Orders nulls before non-nulls, then by natural order

C. Removes nulls

D. Is a compile error

10. `for (Integer x : list) if (x < 0) list.remove(x);` most likely…

A. Works

B. Throws `ConcurrentModificationException` on the next iteration

C. Removes by index

D. Is a compile error

11. Which iterator kind never throws `ConcurrentModificationException`?

A. `ArrayList`'s

B. `HashMap`'s

C. `ConcurrentHashMap`'s (weakly consistent)

D. `TreeSet`'s

12. `Arrays.asList(1, 2, 3).set(0, 9)`…

A. Throws

B. Works — `set` writes through to the backing array

C. Creates a new list

D. Is a compile error

13. Returning `Collections.unmodifiableList(items)` from a getter…

A. Copies the list

B. Exposes a read-only view that still reflects internal changes — acceptable when callers only read

C. Makes `items` immutable

D. Is deprecated

14. `Set.of(1, 2).equals(new HashSet<>(List.of(2, 1)))` is…

A. `false`

B. `true` — set equality is by contents

C. A compile error

D. `true` only for `TreeSet`

15. `HashMap`, `Hashtable` and `ConcurrentHashMap` differ in that…

A. Only `HashMap` is generic

B. `HashMap` is unsynchronised, `Hashtable` is a legacy fully synchronised map, `ConcurrentHashMap` is the modern concurrent one

C. They are identical

D. `ConcurrentHashMap` allows null keys

ANSWER KEY · CHECKPOINT — COLLECTIONS FRAMEWORK

1. B Index access is an array read; the others shift or scan.

2. B The table doubles and every entry is rehashed into the new buckets.

3. B Keys must be immutable (or at least their hashed fields).

4. B It is a red-black tree ordered by comparison; null cannot be compared.

5. B Every `get` moves the entry to the end; the eldest is evicted past capacity.

6. B Stack semantics on the front of the deque; `offer` / `poll` give FIFO by adding at the tail.

7. B The heap invariant is maintained only through `offer` / `poll`.

8. B TimSort is stable; this enables sorting by successive keys.

9. B The wrapper handles null elements that a raw natural-order comparator would reject.

10. B Use `removeIf`, an iterator, or a backwards index loop.

11. C Concurrent collections trade fail-fast detection for weakly consistent or snapshot iteration.

12. B Only size-changing operations are unsupported on the `asList` view.

13. B `List.copyOf` gives an independent snapshot when that is what callers should have.

14. B Order and implementation class do not matter for sets.

15. B Prefer `HashMap` single-threaded and `ConcurrentHashMap` shared; never `Hashtable` in new code.