⍰ JAVA COLLECTIONS — CONCEPT NOTES (INTERVIEW + DEVELOPER EDITION)

□ 1. Why Collections?

- Arrays are fixed-size → Collections are dynamic
- Arrays can't remove duplicates / sort / search → Collections provide ready-made algorithms
- Collections give powerful data structures: `List`, `Set`, `Map`, `Queue`
- Unified API → faster development, cleaner code
- Used everywhere: backend, Spring Boot, microservices, interview rounds

---

□ 2. Core Interfaces (Super Important)

List

- Ordered
- Allows duplicates
- Index-based access
- Best for: histories, sequences, frequent reads
- Examples: `ArrayList`, `LinkedList`, `Vector`

Set

- Unique elements
- No duplicates
- Order not guaranteed (HashSet)
- Best for: uniqueness, membership checking
- Examples: `HashSet`, `LinkedHashSet`, `TreeSet`

Map

- Key–value pairs
- Keys unique
- Values duplicated
- Best for: lookups, caching, grouping
- Examples: `HashMap`, `TreeMap`, `LinkedHashMap`, `ConcurrentHashMap`

□ 3. ArrayList vs LinkedList — REAL Understanding

ArrayList

- Backed by **dynamic array**
- Fast read (O(1))
- Slow insert/delete in middle (O(n)) due to shifting
- Great for frequent reads and end-appends

LinkedList

- Backed by **doubly linked list**
- Slow read (O(n))
- Fast insert/delete at start/middle (O(1) if node known)
- Best for queues, schedulers, heavy structural modifications

---

□ 4. HashMap Internal Working (MOST IMPORTANT)

✔ Hashing Process

```
hashCode() → spread → index = (n - 1) & hash
```

✔ Storage structure

- Bucket array: `Node<K,V>[] table`
- Each bucket stores **linked list** OR **Red-Black Tree**

✔ Collision handling

- Same bucket index → store in linked list
- If size > 8 → convert to TreeNode (treeify)
- Improves worst-case to O(log n)

✔ get(key)

1. Compute hash → index
2. Go to bucket
3. Compare hash
4. Compare equals()
5. Return value

✔ Why equals + hashCode needed?

- hashCode → finds bucket
- equals → finds correct key
- BOTH required for correctness

□ 5. HashSet Internal Working

- HashSet = wrapper over HashMap
- Elements stored as **Keys**
- Dummy value = `PRESENT`
- Uniqueness handled via:
  - hashCode() → bucket
  - equals() → duplicate check

□ 6. LinkedHashSet & LinkedHashMap

Maintain order using Doubly Linked List

✔ Use case:

- LRU Cache
- Maintaining insertion order
- Maintaining access order

□ 7. TreeMap / TreeSet (Sorted Collections)

- Implemented using **Red-Black Tree**
- Guarantees **sorted order**
- O(log n) insertion, deletion, search
- Use-case:
  - Leaderboards
  - Range queries
  - Sorted dictionaries

□ 8. PriorityQueue (Min/Max Heap)

- Default: Min-Heap
- O(log n) insert/remove
- Great for:
  - Top K
  - Shortest path algorithms
  - Job scheduling

□ 9. Fail-Fast vs Fail-Safe Iterators

Fail-Fast

- Collections: ArrayList, HashMap, HashSet
- Detect concurrent modification via `modCount`
- Throw `ConcurrentModificationException`

Fail-Safe

- Concurrent collections: CopyOnWriteArrayList, ConcurrentHashMap
- Iterate on cloned snapshot
- No exception
- Changes not reflected in iterator

□ 10. ConcurrentHashMap (Thread-Safe Map)

JDK 7

- Segment-based locking

JDK 8

- No segments
- CAS + bucket-level locks
- No blocking entire map
- Weakly consistent iterator
- Best for high-concurrency read/write environments

□ 11. Load Factor & Capacity

Default:

- initial capacity = 16
- load factor = 0.75

Why 0.75?

- Balanced compromise
- Prevents too many collisions
- Avoids excessive resizing

Resizing:

- When size > capacity * load factor
- Doubles table size
- Rehash all keys (expensive)

□ 12. Comparable vs Comparator (Sorting)

Comparable (Natural Ordering)

- Override `compareTo()`
- Only ONE sort order

Comparator (Custom ordering)

- Can create multiple comparators
- Using `Comparator.comparing()` + `thenComparing()`

□ 13. Common Real Interview Use-Cases

✔ Frequency Counting

Use → `HashMap`

✔ Remove duplicates but preserve order

Use → `LinkedHashSet`

✔ Sort by multiple fields

Use → `Comparator.thenComparing()`

✔ Grouping items

Use → `Map<K,List<V>>` + `computeIfAbsent()`

✔ Top K frequent

Use → `PriorityQueue` + `HashMap`

✔ LRU Cache

Use → `LinkedHashMap` (accessOrder = true)

□ 14. MUST-KNOW Complexity Cheatsheet

DS	Insert	Search	Delete	Notes
ArrayList	O(1)/O(n)	O(1)	O(n)	slow in middle
LinkedList	O(1)	O(n)	O(1)	good for queues
HashSet	O(1)	O(1)	O(1)	unique values
HashMap	O(1)	O(1)	O(1)	collisions → tree
TreeMap	O(log n)	O(log n)	O(log n)	sorted
PriorityQueue	O(log n)	O(1)	O(log n)	heap

□ 15. Real-World Use Cases

HashMap

- caching
- counting
- lookups
- database indexing

HashSet

- duplicate removal
- membership testing
- unique tags/categories

LinkedHashMap

- LRU Cache

- logs
- sequential auditing

TreeMap

- leaderboard
- schedules
- sorted search

PriorityQueue

- scheduling
- Top-K
- nearest tasks first

□ □ 10 Practical Coding Examples

□ 1. First Non-Repeating Character using LinkedHashMap

```
public static Character firstNonRepeating(String str) {

    Map<Character, Integer> map = new LinkedHashMap<>();

    for (char c : str.toCharArray()) {
        map.put(c, map.getOrDefault(c, 0) + 1);
    }

    for (Map.Entry<Character, Integer> entry : map.entrySet()) {
        if (entry.getValue() == 1) return entry.getKey();
    }

    return null;
}

public static void main(String[] args) {
    System.out.println(firstNonRepeating("teeter")); // r
}
```

Why LinkedHashMap?
Preserves order + allows counting → best for "first unique".

□ 2. Remove Duplicates but Keep Order (LinkedHashSet)

```
public static List<Integer> removeDuplicates(List<Integer> list) {
    return new ArrayList<>(new LinkedHashSet<>(list));
}

public static void main(String[] args) {
    System.out.println(removeDuplicates(Arrays.asList(2,3,2,5,3,7)));
}
```

Output:

```
[2, 3, 5, 7]
```

□ 3. Frequency Counter (HashMap)

```
public static Map<Integer, Integer> frequency(int[] arr) {
    Map<Integer, Integer> map = new HashMap<>();

    for (int num : arr)
        map.put(num, map.getOrDefault(num, 0) + 1);

    return map;
}
```

Output:

```
{1=1, 2=2, 3=3}
```

□ 4. Two Sum (HashMap)

```
public static int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < nums.length; i++) {
        int need = target - nums[i];

        if (map.containsKey(need))
            return new int[] { map.get(need), i };

        map.put(nums[i], i);
    }
    return new int[] {-1, -1};
}
```

Time: O(n)

□ 5. Group Anagrams (HashMap of Lists)

```
public static List<List<String>> groupAnagrams(String[] arr) {
    Map<String, List<String>> map = new HashMap<>();

    for (String s : arr) {
        char[] c = s.toCharArray();
        Arrays.sort(c);
        String key = new String(c);

        map.computeIfAbsent(key, k -> new ArrayList<>()).add(s);
    }
    return new ArrayList<>(map.values());
}
```

□ 6. Sort Students using Comparator (Marks DESC → Name ASC)

```
class Student {
    int id;
    String name;
    int marks;

    Student(int i, String n, int m) { id=i; name=n; marks=m; }

    public String toString() { return name + " " + marks; }
}

public static void main(String[] args) {
    List<Student> list = Arrays.asList(
            new Student(1,"Riya",90),
            new Student(2,"Neha",95),
            new Student(3,"Rupali",95)
    );

    list.sort(Comparator
            .comparingInt((Student s) -> s.marks).reversed()
            .thenComparing(s -> s.name));

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

□ 7. PriorityQueue → Top K Frequent

```
public static List<Integer> topK(int[] nums, int k) {
    Map<Integer, Integer> freq = new HashMap<>();

    for (int n : nums)
        freq.put(n, freq.getOrDefault(n, 0) + 1);

    PriorityQueue<Map.Entry<Integer,Integer>> pq =
        new PriorityQueue<>(Comparator.comparingInt(Map.Entry::getValue));

    for (Map.Entry<Integer,Integer> e : freq.entrySet()) {
        pq.add(e);
        if (pq.size() > k) pq.poll();
    }

    List<Integer> ans = new ArrayList<>();
    while (!pq.isEmpty()) ans.add(pq.poll().getKey());

    return ans;
}
```

□ 8. TreeMap → Sorted Key Map

```
public static void main(String[] args) {
    Map<String, Integer> map = new TreeMap<>();

    map.put("Banana", 40);
    map.put("Apple", 60);
    map.put("Cherry", 20);

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

Output:

```
{Apple=60, Banana=40, Cherry=20}
```

(TreeMap sorts keys automatically)

□ 9. Fail-Fast Iterator Example

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("A");
    list.add("B");

    for (String s : list) {
        list.add("C"); // throws CME
    }
}
```

Output:

```
ConcurrentModificationException
```