Q: Java Comparator & Custom Sorting — Beginner → Advance

1. Problem Understanding

- **Why Sorting Needs a Comparator?**
  - Java can sort primitives automatically (int[], double[]...).
  - But Java does NOT know how to sort:
    - Complex objects (Student, Pair, Node, Employee)
    - 2D arrays (int[][])
    - Lists with custom rules
    - Sorting by multiple keys
    - Descending order
  - So Java allows you to teach it how to compare two items.
  - This "teaching" is done using a Comparator.

- **What Comparator Actually Does (MOST IMPORTANT RULE)**
  - A Comparator must return an int:
    - Negative → a < b
    - Zero → a == b
    - Positive → a > b
  - ✔ What does "a < b" mean here?
    - It means a should come before b in the sorted order.
  - ✔ What does "a > b" mean?
    - It means b should come before a.
  - ✔ So final rule:
    - If return < 0 → a is placed before b
    - If return > 0 → b is placed before a
    - If return = 0 → order does not change

- **The MOST CONFUSING PART (cleared in simple words)**
  - When you write:
    - (a, b) -> a - b
  - This means:
    - If a < b
    - a - b is negative → a goes before b
    - If a > b
    - a - b is positive → b goes before a
    - If equal → order remains
  - Thus:
    - ✔ Negative return means: keep current order (a first)
    - ✔ Positive return means: swap them (put b first)

- **How Java Uses Comparator? (Internal Logic)**
    - You provide a function:
        - compare(a, b)
    - The sorting algorithm internally checks:
        - if(compare(a, b) > 0) {
            - swap(a, b);
        - }
    - This is the secret.
- **Simple Examples**
    - Sort ascending
        - (a, b) -> a - b
    - Sort descending
        - (a, b) -> b - a
    - Sort by first element of 2D array
        - (a, b) -> a[0] - b[0]
    - Sort by second element
        - (a, b) -> a[1] - b[1]
    - Sort by two keys (primary & secondary)
        - (a, b) -> {
            - if(a[0] != b[0]) return a[0] - b[0];
            - return a[1] - b[1];
        - }
- **Java Comparator — Full Theory**
    - ✔ Comparator is used when:
        - You cannot modify the class
        - You want different ways of sorting
        - You want sorting logic outside the class
    - ✔ Defined using lambda:
        - Comparator comp = (a, b) -> a - b;
    - ✔ Or use directly inside sort:
        - Arrays.sort(arr, (a, b) -> a - b);
- **Java Custom Sorting (Theory)**
    - Custom sorting means:
        - You define your own comparison logic
        - Sorting happens according to your business rule
    - Examples:
        - Sort students by marks
        - Sort employees by salary then by age
        - Sort intervals by ending time
        - Sort pairs by second element
        - Sorting is ALWAYS controlled by your comparator.

- **Comparator vs Comparable (MOST ASKED INTERVIEW Q)**
    - ✔ Comparable
        - Used for natural sorting (default sort order)
        - Implemented inside the class itself
        - Class implements:
            - class Student implements Comparable {
                - public int compareTo(Student other) {
                    - return this.marks - other.marks;
                - }
            - }
        - Used when the class decides how it should be sorted
    - ✔ Comparator * Sorting logic written outside the class * You can write multiple comparators for multiple sorting patterns * Flexibility is high
    - ✔ Simple difference summary (in bullets): * Comparable → sorting rule inside class (compareTo()) * Comparator → sorting rule outside class (compare()) * Comparable → only one rule * Comparator → infinite rules * Comparable → used by Collections.sort(list) or Arrays.sort() automatically * Comparator → must be explicitly passed
- **Java Arrays.sort with Comparator (Theory)**
    - Works for:
        - int[][]
        - Integer[]
        - custom objects
    - Signature:
        - Arrays.sort(arr, comparator);
    - Sorting 2D array example:
        - Arrays.sort(arr, (a, b) -> a[0] - b[0]);
    - Internal algorithm:
        - Uses TimSort for object arrays
        - Uses Dual Pivot QuickSort for primitive arrays
- **Java Collections.sort with Comparator (Theory)**
    - Used only for:
        - List (ArrayList, LinkedList)
    - Syntax:
        - Collections.sort(list, comparator);
    - Example:
        - Collections.sort(students, (a, b) -> a.marks - b.marks);
    - Internally:
        - Uses TimSort
        - Fast and stable
- **Java Lambda Comparator (Theory)**
    - Lambda is just a shorter syntax for Comparator.
        - Without lambda (old style):

        - Collections.sort(list, new Comparator() {
            - public int compare(Integer a, Integer b) {
                - return a - b;
            - }
        - });
    - With lambda:
        - Collections.sort(list, (a, b) -> a - b);
    - Lambda Rules:
        - (a, b) are parameters
        - -> means "using this rule"
        - Right side is return statement
    - Much cleaner and preferred.

- **Comparators & Custom Sorting — Overall Theory (High-Level)**
    - Comparator tells Java how to compare two items
    - Sorting algorithm repeatedly calls your comparator
    - You control sort order completely
    - Used heavily in:
        - DSA problems
        - sorting intervals
        - sorting pairs
        - sorting objects
        - greedy algorithms
        - scheduling problems
        - minimum platforms
        - merge intervals
        - activity selection
        - graph edges (Kruskal)
    - Custom sorting is a critical DSA skill.

- **The Full Logic Behind Comparison (Deep Theory)**
    - Every comparison function defines a strict weak ordering:
        - Transitive
        - Consistent
        - Antisymmetric
    - Meaning:
        - If compare(a, b) < 0 and compare(b, c) < 0, then compare(a, c) MUST also be < 0.
        - Otherwise sorting becomes unstable or incorrect.
    - This is why the comparator must be deterministic and consistent.

- **Real DSA Problem Examples**
    - You use custom comparators in:
        - Sort intervals by start
        - Sort intervals by end
        - Sort jobs by profit

        - Sort edges by weight (Kruskal)
        - Sort students by multiple criteria
        - Sort arrays of pairs
        - Sort frequency maps (descending order of frequency)
        - Sort characters by frequency in strings
        - Sort points by distance from origin
    - Sorting is core to many greedy and DP problems.

- **Conclusion (Everything You Need to Know)**

    - Comparator → flexible external custom sorting
    - Comparable → one fixed sorting inside the class
    - Arrays.sort → for arrays
    - Collections.sort → for lists
    - Lambda comparator → modern, cleaner syntax
    - Return negative → keep order
    - Return positive → swap
    - Custom sorting is essential in DSA

Q: Map Interface

---

1. Problem Understanding

- Represents a collection of key-value pairs.
- Each key is unique; values can be duplicate.
- Keys cannot be null in some implementations (TreeMap), but values can be null.
- Common implementations: HashMap, TreeMap, LinkedHashMap.
- Main methods:

    - V put(K key, V value) → Inserts a key-value pair. If the key exists, updates the value.
    - V get(Object key) → Returns the value associated with the key. Returns null if key not found.
    - V remove(Object key) → Removes the key-value pair by key. Returns the value removed.
    - boolean containsKey(Object key) → Checks if the key exists. Returns true/false.
    - boolean containsValue(Object value) → Checks if the value exists. Returns true/false.
    - Set keySet() → Returns all keys as a Set.
    - Collection values() → Returns all values.
    - Set<Map.Entry<K,V>> entrySet() → Returns all key-value pairs as a set of entries.

- **HashMap (java.util.HashMap)**

    - Stores key-value pairs in hash table.
    - Allows one null key and multiple null values.
    - Not ordered (insertion order is not guaranteed).

- O(1) average time complexity for get(), put(), remove().
- Not synchronized (thread-unsafe).
- Example:
  - HashMap<String, Integer> map = new HashMap<>();
  - map.put("A", 1);
  - int val = map.get("A"); // returns 1
  - boolean hasKey = map.containsKey("A"); // true
  - boolean hasVal = map.containsValue(1); // true

- **TreeMap (java.util.TreeMap)**
  - Implements SortedMap → keys are sorted in natural order or by a comparator.
  - No null keys allowed (throws NullPointerException), but null values are allowed.
  - Internally uses a Red-Black Tree.
  - Time complexity: O(log n) for get(), put(), remove().
  - Example:
    - TreeMap<String, Integer> treeMap = new TreeMap<>();
    - treeMap.put("C", 3);
    - treeMap.put("A", 1);
    - treeMap.put("B", 2);
    - // Keys will be sorted: A, B, C
    - int val = treeMap.get("B"); // 2

- **Key Functions: .get() vs .containsKey() vs .containsValue()**
  - .get(key) → returns value for the given key. Returns null if key is not present.
  - .containsKey(key) → checks if a key exists. Returns true or false.
  - .containsValue(value) → checks if a value exists. Returns true or false.
  - Example:
    - HashMap<String, Integer> map = new HashMap<>();
    - map.put("X", 10);
    - map.get("X"); // returns 10
    - map.containsKey("X"); // true
    - map.containsValue(10); // true
    - map.containsKey("Y"); // false
    - map.containsValue(20); // false

- **Map Interface Methods (java.util.Map)**
  - Adding / Updating
    - V put(K key, V value) → Adds key-value pair. Updates if key exists.
    - void putAll(Map<? extends K, ? extends V> m) → Copies all mappings from another map.
    - V putIfAbsent(K key, V value) → Adds key-value pair only if key is absent.
  - Retrieving
    - V get(Object key) → Returns value for the key, or null if key not found.
    - V getOrDefault(Object key, V defaultValue) → Returns value if key exists, else returns defaultValue.
  - Removing

        - V remove(Object key) → Removes entry by key, returns removed value or null.
        - boolean remove(Object key, Object value) → Removes entry only if key maps to value. Returns true if removed.
    - Checking existence
        - boolean containsKey(Object key) → Checks if key exists.
        - boolean containsValue(Object value) → Checks if value exists.
    - Size / Emptiness
        - int size() → Number of key-value mappings.
        - boolean isEmpty() → Checks if map is empty.
    - Iteration / Views
        - Set keySet() → Returns all keys.
        - Collection values() → Returns all values.
        - Set<Map.Entry<K, V>> entrySet() → Returns all key-value pairs as Map.Entry.
    - Replacement / Compute
        - V replace(K key, V value) → Replaces value for key if it exists.
        - boolean replace(K key, V oldValue, V newValue) → Replaces only if current value matches oldValue.
        - V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) → Computes new value.
        - V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) → Computes and inserts value if key is absent.
        - V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) → Computes new value only if key is present.
    - Merge / Other Utilities
        - V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction) → Merges value with existing value.
        - void clear() → Removes all entries.
        - default void forEach(BiConsumer<? super K, ? super V> action) → Performs action on each entry.

---

2. Approaches

Approach 1:

Java Code:

```
import java.util.*;
import java.util.function.*;

public class MapMethodsExample {
    public static void main(String[] args) {

        // Create a HashMap
        Map<String, Integer> map = new HashMap<>();

        // --- Adding / Updating ---
        map.put("A", 10); // Add
```

```
        map.put("B", 20);
        map.put("A", 15); // Update
        map.putIfAbsent("C", 30); // Add only if key is absent
        map.putIfAbsent("B", 50); // Won't update because B exists
        System.out.println("After put & putIfAbsent: " + map);

        // --- Retrieving ---
        System.out.println("get(\"A\"): " + map.get("A")); // 15
        System.out.println("getOrDefault(\"D\", 100): " + map.getOrDefault("D",
100)); // 100

        // --- Checking existence ---
        System.out.println("containsKey(\"B\"): " + map.containsKey("B")); // true
        System.out.println("containsValue(30): " + map.containsValue(30)); // true

        // --- Removing ---
        map.remove("C"); // remove by key
        System.out.println("After remove(\"C\"): " + map);
        boolean removed = map.remove("B", 25); // remove only if value matches
        System.out.println("Attempt to remove B with value 25: " + removed + " |
Map: " + map);

        // --- Size / Emptiness ---
        System.out.println("size(): " + map.size()); // 2
        System.out.println("isEmpty(): " + map.isEmpty()); // false

        // --- Iteration / Views ---
        System.out.println("Keys: " + map.keySet()); // [A, B]
        System.out.println("Values: " + map.values()); // [15, 20]
        System.out.println("Entries: " + map.entrySet()); // [A=15, B=20]

        // Iterate using forEach
        map.forEach((k, v) -> System.out.println(k + " -> " + v));

        // --- Replacement / Compute ---
        map.replace("A", 50); // replace value
        map.replace("B", 20, 40); // replace only if old value matches
        System.out.println("After replace: " + map);

        map.compute("A", (k, v) -> v + 5); // 50 + 5
        map.computeIfAbsent("D", k -> 100); // add only if absent
        map.computeIfPresent("B", (k, v) -> v * 2); // 40 * 2
        System.out.println("After compute methods: " + map);

        // --- Merge / Utilities ---
        map.merge("A", 20, (oldVal, newVal) -> oldVal + newVal); // 55 + 20 = 75
        System.out.println("After merge: " + map);

        map.clear();
        System.out.println("After clear: " + map + " | isEmpty: " +
map.isEmpty());
    }
}
```

Q: printf() Formatting

1. Problem Understanding

- Syntax: System.out.printf("format_string", var1, var2, ...);
- Each % symbol in the format string corresponds to one variable after the comma.
- The order of placeholders and variables must match.
- If the type doesn't match the format specifier, Java throws an IllegalFormatConversionException.
- **Common Format Specifiers**
  - %d → integer
  - %f → floating-point (double/float)
  - %s → string
  - %c → character
  - %b → boolean
  - %n → newline (platform independent, better than \n)
- **Integer Formatting Rules (%d)**
  - %d → normal integer printing
  - %5d → prints integer in width 5 (right-aligned)
  - %-5d → prints integer in width 5 (left-aligned)
  - %05d → width 5, padded with zeros on the left
  - %,d → prints number with commas (e.g., 12,345)
  - %+d → shows sign (e.g., +45)
  - %(d → encloses negative numbers in parentheses (e.g., (45))
  - %x → print integer in hexadecimal
  - %o → print integer in octal
    - 🧩 Example:
      - System.out.printf("|%5d|%-5d|%05d|", 42, 42, 42);
      - Output:
        - | 42|42 |00042|
- **Floating-Point Formatting (%f, %e, %g)**
  - %f → prints floating-point numbers (default 6 digits after decimal)
  - %.2f → 2 digits after decimal
  - %8.3f → width 8, 3 digits after decimal
  - %e → scientific notation (e.g., 3.14e+00)
  - %g → automatically picks shortest representation
  - 🧩 Example:
    - System.out.printf("%.2f %8.3f %e", 3.14159, 3.14159, 3.14159);

        - Output:
        - 3.14 3.142 3.141590e+00

- **String Formatting (%s)**
    - %s → normal string
    - %20s → right-aligned in width 20
    - %-20s → left-aligned in width 20
    - %.5s → prints only first 5 characters of string
        - 🧩 Example:
        - System.out.printf("|%10s|%-10s|%.3s|", "Java", "Code", "Learning");
        - Output:
        - | Java|Code |Lea|

- **Character Formatting (%c)**
    - Prints a single character
    - You can use integer values (ASCII codes) to print characters
        - 🧩 Example:
        - System.out.printf("%c %c", 'A', 66);
        - Output:
        - A B

- **Boolean Formatting (%b)**
    - %b → prints true or false
    - If the variable is null, it prints false
        - 🧩 Example:
        - System.out.printf("%b %b", true, null);
        - Output:
        - true false

- **Date and Time (%t)**
    - %t or %T → used for date/time formatting
        - Example placeholders:
            - %tY → year (e.g., 2025)
            - %tm → month (e.g., 10)
            - %td → day (e.g., 18)
            - %tH:%tM:%tS → hour:minute:second

- **Combining Multiple Placeholders**
    - You can print multiple variables in a single statement.
        - Example:
        - int hour = 5;
        - String minutes = "09";
        - String seconds = "45";
        - System.out.printf("%02d:%s:%s", hour, minutes, seconds);
        - Output:

        - 05:09:45
    - %02d → width 2, padded with zeros (ensures double-digit hours like 05)

---

2. Tips & Observations

- %n is better than \n because it works across all operating systems.
- Always match the data type with the format specifier.
- You can combine alignment, width, and precision in one format specifier.
- Avoid mixing System.out.print() and printf() for same-line formatting (they handle buffers differently).
- You can use String.format() with the same rules to store formatted output in a string.
