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) â†’ Inserts a key-value pair. If the key exists, updates the value.
  - V get(Object key) â†’ Returns the value associated with the key. Returns null if key not found.
  - V remove(Object key) â†’ Removes the key-value pair by key. Returns the value removed.
  - boolean containsKey(Object key) â†’ Checks if the key exists. Returns true/false.
  - boolean containsValue(Object value) â†’ Checks if the value exists. Returns true/false.
  - Set keySet() â†’ Returns all keys as a Set.
  - Collection values() â†’ Returns all values.
  - Set<Map.Entry<K,V>> entrySet() â†’ 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 â†’ 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) â†’ returns value for the given key. Returns null if key is not present.
  - .containsKey(key) â†’ checks if a key exists. Returns true or false.
  - .containsValue(value) â†’ 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) â†’ Adds key-value pair. Updates if key exists.
    - void putAll(Map<? extends K, ? extends V> m) â†’ Copies all mappings from another map.
    - V putIfAbsent(K key, V value) â†’ Adds key-value pair only if key is absent.
  - Retrieving
    - V get(Object key) â†’ Returns value for the key, or null if key not found.
    - V getOrDefault(Object key, V defaultValue) â†’ Returns value if key exists, else returns defaultValue.
  - Removing
    - V remove(Object key) â†’ Removes entry by key, returns removed value or null.
    - boolean remove(Object key, Object value) â†’ Removes entry only if key maps to value. Returns true if removed.
  - Checking existence
    - boolean containsKey(Object key) â†’ Checks if key exists.
    - boolean containsValue(Object value) â†’ Checks if value exists.
  - Size / Emptiness
    - int size() â†’ Number of key-value mappings.
    - boolean isEmpty() â†’ Checks if map is empty.
  - Iteration / Views
    - Set keySet() â†’ Returns all keys.
    - Collection values() â†’ Returns all values.
    - Set<Map.Entry<K, V>> entrySet() â†’ Returns all key-value pairs as Map.Entry.
  - Replacement / Compute
    - V replace(K key, V value) â†’ Replaces value for key if it exists.
    - boolean replace(K key, V oldValue, V newValue) â†’ Replaces only if current value matches oldValue.
    - V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) â†’ Computes new value.

        - V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) â†’ Computes and inserts value if key is absent.
        - V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) â†’ 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) â†’ Merges value with existing value.
        - void clear() â†’ Removes all entries.
        - default void forEach(BiConsumer<? super K, ? super V> action) â†’ Performs action on each entry.

---

2. Approaches

Approach 1:

Java Code:

```java
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, â€¦);
- Each % symbol in the format string corresponds to one variable after the comma.
- The order of placeholders and variables must match.
- If the type doesnâ€™t match the format specifier, Java throws an IllegalFormatConversionException.
- **Common Format Specifiers**
    - %d â†’ integer
    - %f â†’ floating-point (double/float)
    - %s â†’ string

  - %c â†’ character
  - %b â†’ boolean
  - %n â†’ newline (platform independent, better than \n)

- **Integer Formatting Rules (%d)**

  - %d â†’ normal integer printing
  - %5d â†’ prints integer in width 5 (right-aligned)
  - %-5d â†’ prints integer in width 5 (left-aligned)
  - %05d â†’ width 5, padded with zeros on the left
  - %,d â†’ prints number with commas (e.g., 12,345)
  - %+d â†’ shows sign (e.g., +45)
  - %(d â†’ encloses negative numbers in parentheses (e.g., (45))
  - %x â†’ print integer in hexadecimal
  - %o â†’ print integer in octal
    - ðŸ§© Example:
      - System.out.printf("|%5d|%-5d|%05d|", 42, 42, 42);
      - Output:
        - | 42|42 |00042|

- **Floating-Point Formatting (%f, %e, %g)**

  - %f â†’ prints floating-point numbers (default 6 digits after decimal)
  - %.2f â†’ 2 digits after decimal
  - %8.3f â†’ width 8, 3 digits after decimal
  - %e â†’ scientific notation (e.g., 3.14e+00)
  - %g â†’ automatically picks shortest representation
  - ðŸ§© 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 â†’ normal string
  - %20s â†’ right-aligned in width 20
  - %-20s â†’ left-aligned in width 20
  - %.5s â†’ prints only first 5 characters of string
    - ðŸ§© 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
    - ðŸ§© Example:
    - System.out.printf("%c %c", 'A', 66);

        - Output:
        - A B

- **Boolean Formatting (%b)**
    - %b â†’ prints true or false
    - If the variable is null, it prints false
        - ðŸ§© Example:
        - System.out.printf("%b %b", true, null);
        - Output:
        - true false

- **Date and Time (%t)**
    - %t or %T â†’ used for date/time formatting
        - Example placeholders:
            - %tY â†’ year (e.g., 2025)
            - %tm â†’ month (e.g., 10)
            - %td â†’ day (e.g., 18)
            - %tH:%tM:%tS â†’ 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 â†’ 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.
