Practise Questions with Answers

Collection Framework

Q1. You need to store a list of customer IDs where:

- ○ Order of insertion must be preserved
- ○ Duplicate IDs are allowed.
- ○ Which Java collection would you use and why?

Answer : I use ArrayList (or List interface) because:

- ○ ArrayList maintains insertion order
- ○ It allows duplicate elements
- ○ Provides fast access using index
- ○ Most commonly used when order + duplicates are required
- ○ Example :
- ○ List<Integer> customerIds = new ArrayList<>();
- ○ customerIds.add(101);
- ○ customerIds.add(102);
- ○ customerIds.add(101); // duplicate allowed
- ○ Output order: 101, 102, 101

- **What would change if duplicates were NOT allowed?**

Answer : If duplicates are NOT allowed, the main change is that you would use a Set instead of a List.

I would use a LinkedHashSet because it maintains insertion order while ensuring uniqueness of elements.

Correct Choice : LinkedHashSet

- LinkedHashSet maintains insertion order
- Does NOT allow duplicates
- Internally uses a hash table + linked list

Example - Set<Integer> customerIds = new LinkedHashSet<>();

customerIds.add(101);

customerIds.add(102);

customerIds.add(101); // ignored

Output order: 101, 102

Q2. In a multi-threaded application, multiple threads update a shared collection.

- Why are normal collections like ArrayList or HashMap not thread-safe?

Answer : Normal collections are not thread-safe because:

- They do not use synchronization
- Multiple threads can modify the collection at the same time
- This can lead to:
- Data inconsistency
    - Race conditions
    - Lost updates
    - ConcurrentModificationException

Example problem:

One thread is adding elements while another is reading → internal structure becomes corrupted.

- Name **one thread-safe collection** in Java.

Answer: One thread-safe collection in Java is:

ConcurrentHashMap

- is designed for concurrent access
- Allows multiple threads to read and write simultaneously
- Uses fine-grained locking, so performance is better than synchronized collections

Example :

Map<Integer, String> map = new ConcurrentHashMap<>();

map.put(1, "A");

map.put(2, "B");

Safe to use in multi-threaded applications

- When would you prefer a **Concurrent collection** over Collections.synchronizedList()?

Answer : I prefer Concurrent collections when:

- The application has high concurrency
- There are many read and write operations
- You want better performance and scalability

Why Concurrent collections are better:

- Use fine-grained locking (not one lock for entire collection)
- Allow multiple threads to read/write simultaneously
- No need to manually synchronize during iteration
- Avoid performance bottlenecks

Q3. If an ArrayList is initialized with a size of 25 and a 26th element is added, what happens internally?

Answer : When an ArrayList is created with an initial capacity of 25 and you try to add the 26th element, ArrayList automatically grows its internal array.

Internal Working (Step by Step)

1. ArrayList uses an internal array (Object[]) to store elements.

2. Initial capacity = 25
3. When the 26th element is added:

    - Current capacity is not sufficient
    - ArrayList creates a new, larger array
    - Existing elements are copied to the new array
    - Old array is discarded

By default, ArrayList grows by:

New Capacity = Old Capacity + (Old Capacity / 2)

So:

25 + (25 / 2) = 25 + 12 = 37

New capacity becomes 37

Q4. You are given a list of employee names where:

• Names may repeat

• Case should be treated as same ("John" and "john")

Your task is to:

1. Remove duplicates

2. Preserve the original insertion order

3. Print the unique employee names

Input: ["John", "Alice", "john", "Bob", "Alice", "BOB"]

Output:

John

Alice

Bob

Answer : Using a LinkedHashMap (or LinkedHashSet with normalization)

We track:

- Key → lowercase name (for comparison)
- Value → original name (to preserve original format)

Code : import java.util.*;

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

      List<String> names = Arrays.asList(
            "John", "Alice", "john", "Bob", "Alice", "BOB"
      );

      Map<String, String> uniqueNames = new LinkedHashMap<>();

      for (String name : names) {
         String key = name.toLowerCase();
         uniqueNames.putIfAbsent(key, name);
      }

      // Print unique employee names
```

```
        for (String name : uniqueNames.values()) {
            System.out.println(name);
        }
    }
}
```

```
Output: John
            Alice
            Bob
```

Why LinkedHashMap because:

- Preserves insertion order
- Ensures uniqueness using keys
- Handles case-insensitive comparison
- Keeps first occurrence formatting