**SDET Interview Questions – 12th July, 2025**

**==============================================**

**Java**

**===================**

1.Given a list of Employee objects:  
   class Employee {  
   String name;  
   String department;  
   double salary;  
   }  
   Write a method to:  
   Filter employees with salary > 70,000  
   Group them by department  
   Sort each department group by descending salary  
   Return type: Map<String, List<Employee>>

import java.util.\*;

import java.util.stream.\*;

class Employee {

String name;

String department;

double salary;

// Constructor

public Employee(String name, String department, double salary) {

this.name = name;

this.department = department;

this.salary = salary;

}

// Getters (required for clean stream access)

public String getName() { return name; }

public String getDepartment() { return department; }

public double getSalary() { return salary; }

@Override

public String toString() {

return name + " - " + salary;

}

}

public class EmployeeProcessor {

public static Map<String, List<Employee>> filterGroupAndSort(List<Employee> employees) {

return employees.stream()

.filter(emp -> emp.getSalary() > 70000)

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.collectingAndThen(

Collectors.toList(),

list -> list.stream()

.sorted(Comparator.comparingDouble(Employee::getSalary).reversed())

.collect(Collectors.toList())

)

));

}

// Example usage

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", "HR", 72000),

new Employee("Bob", "Engineering", 85000),

new Employee("Charlie", "HR", 68000),

new Employee("David", "Engineering", 91000),

new Employee("Eve", "Marketing", 71000)

);

Map<String, List<Employee>> result = filterGroupAndSort(employees);

result.forEach((dept, emps) -> {

System.out.println(dept + ": " + emps);

});

}

}

2. Given an array of integers and an index currentIndex, your task is to move the element at currentIndex to a new position k in the array.

import java.util.\*;

public class ArrayElementMover {

public static int[] moveElement(int[] arr, int currentIndex, int newIndex) {

if (arr == null || currentIndex < 0 || currentIndex >= arr.length ||

newIndex < 0 || newIndex >= arr.length) {

throw new IllegalArgumentException("Invalid indices or null array.");

}

int element = arr[currentIndex];

List<Integer> list = new ArrayList<>();

// Copy all elements except the one to move

for (int i = 0; i < arr.length; i++) {

if (i != currentIndex) {

list.add(arr[i]);

}

}

// Insert the element at the new index

list.add(newIndex, element);

// Convert back to array

for (int i = 0; i < arr.length; i++) {

arr[i] = list.get(i);

}

return arr;

}

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40, 50};

int currentIndex = 1; // Move element 20

int newIndex = 3;

System.out.println("Before: " + Arrays.toString(arr));

arr = moveElement(arr, currentIndex, newIndex);

System.out.println("After: " + Arrays.toString(arr));

}

}

3. Write a method that takes a list of log lines and returns the top 3 most frequent error messages. Each line format:  
   Eg: [ERROR] [timestamp] [message]  
   Ignore lines that don’t start with [ERROR]. Ignore timestamps and normalize messages to lowercase before counting.  
   Example Input:  
   List<String> logs = Arrays.asList(  
    "[INFO] 2024-10-01 Startup",  
    "[ERROR] 2024-10-01 NullPointerException at line 22",  
    "[ERROR] 2024-10-01 ArrayIndexOutOfBoundsException",  
    "[ERROR] 2024-10-01 NullPointerException at line 22",  
    "[ERROR] 2024-10-01 NullPointerException at line 22"

import java.util.\*;

import java.util.stream.\*;

public class LogAnalyzer {

public static List<String> getTop3Errors(List<String> logs) {

Map<String, Long> frequencyMap = logs.stream()

.filter(line -> line.startsWith("[ERROR]"))

.map(line -> {

// Remove prefix and timestamp (assumes fixed format)

String[] parts = line.split(" ", 3);

return parts.length >= 3 ? parts[2].toLowerCase() : "";

})

.filter(msg -> !msg.isEmpty())

.collect(Collectors.groupingBy(msg -> msg, Collectors.counting()));

return frequencyMap.entrySet().stream()

.sorted((e1, e2) -> Long.compare(e2.getValue(), e1.getValue())) // descending count

.limit(3)

.map(Map.Entry::getKey)

.collect(Collectors.toList());

}

public static void main(String[] args) {

List<String> logs = Arrays.asList(

"[INFO] 2024-10-01 Startup",

"[ERROR] 2024-10-01 NullPointerException at line 22",

"[ERROR] 2024-10-01 ArrayIndexOutOfBoundsException",

"[ERROR] 2024-10-01 NullPointerException at line 22",

"[ERROR] 2024-10-01 NullPointerException at line 22"

);

List<String> topErrors = getTop3Errors(logs);

System.out.println("Top 3 errors:");

topErrors.forEach(System.out::println);

}

}

4. Write a utility to read a test case data from a nested JSON file and return it as a Map<String, Object> using Jackson or Gson.

import com.fasterxml.jackson.core.type.TypeReference;

import com.fasterxml.jackson.databind.ObjectMapper;

import java.io.File;

import java.io.IOException;

import java.util.Map;

public class JsonTestCaseReader {

private static final ObjectMapper objectMapper = new ObjectMapper();

public static Map<String, Object> readTestCaseFromJson(File file) throws IOException {

return objectMapper.readValue(file, new TypeReference<Map<String, Object>>() {});

}

// Overloaded method for reading from String path

public static Map<String, Object> readTestCaseFromJson(String filePath) throws IOException {

return readTestCaseFromJson(new File(filePath));

}

// Optional: pretty print for debugging

public static void printJson(Map<String, Object> map) throws IOException {

String pretty = objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(map);

System.out.println(pretty);

}

public static void main(String[] args) throws IOException {

// Example usage

Map<String, Object> testData = readTestCaseFromJson("test-data.json");

printJson(testData);

}

}

Example test-data.json

{

"testCaseId": "TC\_001",

"input": {

"username": "admin",

"password": "pass123"

},

"expected": {

"status": "success",

"code": 200

}

}

Sample Output

{

"testCaseId" : "TC\_001",

"input" : {

"username" : "admin",

"password" : "pass123"

},

"expected" : {

"status" : "success",

"code" : 200

}

}

5. Find Top N Frequent Words Using Streams  
   Logic:  
     a) Use frequency Map via Collectors.groupingBy and counting.  
     b) Sort by count descending and limit N.

import java.util.\*;

import java.util.function.Function;

import java.util.stream.Collectors;

public class WordFrequency {

public static List<String> topNFrequentWords(List<String> words, int n) {

return words.stream()

.collect(Collectors.groupingBy(

Function.identity(), Collectors.counting()))

.entrySet().stream()

.sorted((e1, e2) -> Long.compare(e2.getValue(), e1.getValue())) // Descending

.limit(n)

.map(Map.Entry::getKey)

.collect(Collectors.toList());

}

public static void main(String[] args) {

List<String> words = Arrays.asList(

"apple", "banana", "apple", "orange", "banana", "apple", "kiwi", "banana", "kiwi"

);

List<String> topWords = topNFrequentWords(words, 2);

System.out.println("Top 2 words: " + topWords);

}

}

Output:

Top 2 words: [apple, banana]

6. **Find Duplicate Elements in a List**

**Input:** [1, 2, 3, 2, 4, 5, 3]  
 **Output:** [2, 3]  
 **Logic:** Use Set to track seen elements; use another set for duplicates.

### **7. Remove Duplicates and Preserve Order**

### **Input:** [4, 5, 4, 6, 5]

### **Output:** [4, 5, 6] **Logic:** Use LinkedHashSet.

8. **Count Character Frequency in a String**

**Input:** "banana"  
 **Output:** {b=1, a=3, n=2}  
 **Logic:** Use Map<Character, Integer> and loop through characters.

======================================================================

**REST Assured**

====================================================================== 1) Imagine you have to test an API that sometimes returns inconsistent response times and intermittent failures. How would you design your REST Assured tests to handle and report these flaky behaviors?  
   Logical Expectation: Look for retry mechanisms, logging, graceful failure handling, alerts, etc.

2) You need to validate a deeply nested JSON response with arrays and objects. How would you write your REST Assured test to extract and verify multiple values efficiently?  
   Logical Expectation: Expect explanation of JSON path usage, custom POJOs, or JSON schema validation.)

3) If an API requires OAuth2 token-based authentication which expires frequently, how would you automate the process of acquiring and refreshing the token in your REST Assured tests?  
   Logical Expectation: Candidate should explain token fetching, caching, refreshing, and integrating it in test setup.)

4) Suppose the API you are testing is still under development and not always stable. How would you design your REST Assured tests and framework to accommodate ongoing API changes?  
   Logical Expectation: Look for ideas around abstraction layers, versioning, flexible validations, mocking, etc.)

5) You have multiple environments (Dev, QA, Prod) with different base URLs and credentials. How do you organize your REST Assured tests and configuration to run seamlessly across these environments?  
   Logical Expectation: Expect environment-specific config files, profiles, use of property files or environment variables.)