Question 1

Consider the following program to insert values into HashSet.

class Student

{

public int id;

public String name;

public Student(String name, int id)

{

this.name = name;

this.id = id;

}

public int hashCode()

{

return this.id;

}

public String toString()

{

return "Student: " + this.name + "@" + Integer.toHexString(hashCode());

}

public boolean equals(Object o)

{

if (o instanceof Student)

{

Student s = (Student) o;

return s.id == this.id ? true : false;

} return false;

}

}

public class UpdateHashSet

{

public static void main(String[] args)

{

HashSet<Student> studentList = new HashSet<>();

Student st1 = new Student("Nimit", 1);

Student st2 = new Student("Rahul", 3);

Student st3 = new Student("Nimit", 2);

studentList.add(st1);

studentList.add(st2);

studentList.add(st3);

System.out.println(studentList.size());

st1.id = 3;

System.out.println(studentList.size());

}

}

What will be the output?

' Why the size would be 3 or 2 ?

solution to overcome the issue of 'Updating the value of key field

**Understanding hashCode and HashSet**

1. **Hashing Principle**:
   * When you add an object (Student in this case) to a HashSet, the HashSet calculates the hash code of the object using the hashCode() method.
   * The hash code determines the bucket (or slot) in which the object will be stored within the HashSet.
2. **Modifying Object State**:
   * If you modify the state of an object after it has been added to a HashSet, such as changing the id field of a Student object, the hash code of that object can change if hashCode() depends on the id.
   * This change in hash code means that the HashSet might not be able to locate the object correctly within its internal structure because it’s still looking for the object in the old bucket (based on the old hash code).
3. **HashSet Integrity**:
   * HashSet relies on the correct implementation of hashCode() and equals() to ensure that objects are stored and retrieved correctly.
   * If you modify an object's fields that are used in hashCode() and equals() after adding it to a HashSet, the HashSet might not be able to find the object when you try to access it or remove it, because the object's position in the HashSet is based on its original hash code.

**Example Scenario**

Let's consider your original code snippet where st1 is modified after being added to the HashSet:

java

Copy code

HashSet<Student> studentList = new HashSet<>();

Student st1 = new Student("Nimit", 1);

studentList.add(st1);

System.out.println(studentList.contains(st1)); // true, st1 is in the set

// Now modify st1

st1.id = 3;

// Attempt to find st1 in the set

System.out.println(studentList.contains(st1)); // false, might not find st1 anymore

In this scenario:

* Initially, st1 is added to studentList, and studentList.contains(st1) correctly returns true.
* After modifying st1.id to 3, the hash code of st1 changes (assuming hashCode() depends on id).
* When you call studentList.contains(st1) again, the HashSet calculates the new hash code of st1 based on the modified id (which is 3 now).
* Since HashSet is structured based on the original hash code (1), it might not find st1 in the correct bucket anymore, leading to contains() returning false.

**Conclusion**

To maintain the integrity of a HashSet:

* **Immutable Key Fields**: Ideally, fields used in hashCode() and equals() should be immutable after adding objects to a HashSet.
* **Remove-Modify-Add Pattern**: If you need to modify such fields, remove the object from the HashSet, modify it, and then add it back to ensure the HashSet can correctly manage its internal data structure.
* **Stable Hash Code**: Ensure that hashCode() provides a stable hash code based on fields used in equals() to avoid inconsistencies.

By following these practices, you can ensure that your HashSet behaves correctly and can find objects based on their current state even after modifications.

Question 2

We have a csv file

student\_id, student\_name, student\_dob

1, Bob, 2020-01-05

2, Alice, 2020-01-31

Parse this file store in memory (any data structure you want)

As you cover various scenarios below, ask candidate to suggest the collection / data structure.

Do not prompt them or give choices.  
Reject the records in cases like ID is null, Name is null, DOB is null. These are DQ checks.

Which Data structure will be used when we want to fetch the list of students by dob range (start date, end end)? TreeSet or TreeMap or Binary Search Tree (Balanced vs normal).

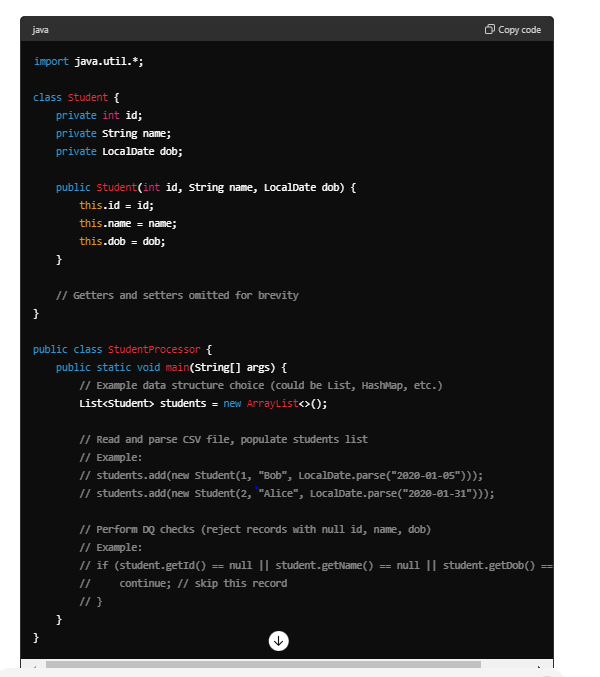
Which DataStructure should be used to search effectivity by student id and also to print the list whenever required in the same order we read from the file. LinkedHashMap or combination of multiple data structures like List+HashMap  
 What will happen when multiple students have same dob

**Solution :**

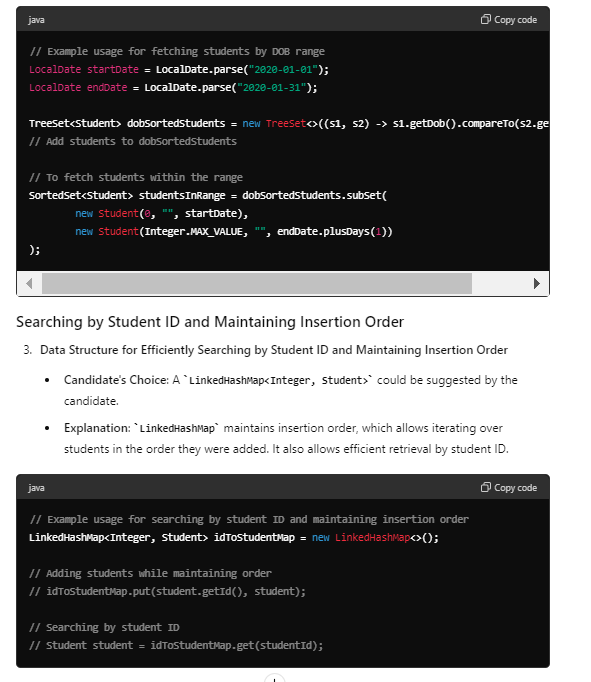
Let's break down the questions and scenarios related to handling student data from a CSV file and discuss the appropriate data structures for each case:

**Parsing and Storing CSV Data**

1. **Data Structure for Storing Students**
   * **Candidate's Choice**: Typically, candidates might choose List<Student> or HashMap<Integer, Student> for storing students, where Student is a custom class representing student details.
   * **Rejecting Records**: Before storing, ensure to perform Data Quality (DQ) checks to reject records where essential fields (student\_id, student\_name, student\_dob) are null.

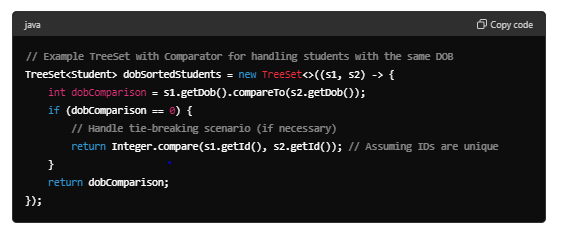
**Fetching Students by DOB Range**

1. **Data Structure for Efficiently Fetching by DOB Range**
   * **Candidate's Choice**: For fetching students efficiently by DOB range (start date to end date), TreeSet or TreeMap can be suggested by the candidate.
   * **Recommendation**: TreeSet is appropriate because it automatically sorts elements (here, students by DOB) and allows efficient range queries.

**Handling Students with the Same DOB**

1. **Handling Students with the Same DOB**
   * **Explanation**: If multiple students have the same DOB and you're using a TreeSet, they will be considered equal according to the Comparator used (in this case, comparing by DOB). However, in practice, it's crucial to have a unique identifier (like student ID) to differentiate between such students.

**Conclusion**



By understanding and correctly applying appropriate data structures (List, TreeSet, LinkedHashMap) based on the specific requirements (such as fetching by DOB range, searching by student ID, maintaining insertion order), candidates demonstrate a solid grasp of data structures and their implications for various scenarios in software development. This knowledge ensures efficient and effective handling of data as per business requirements.

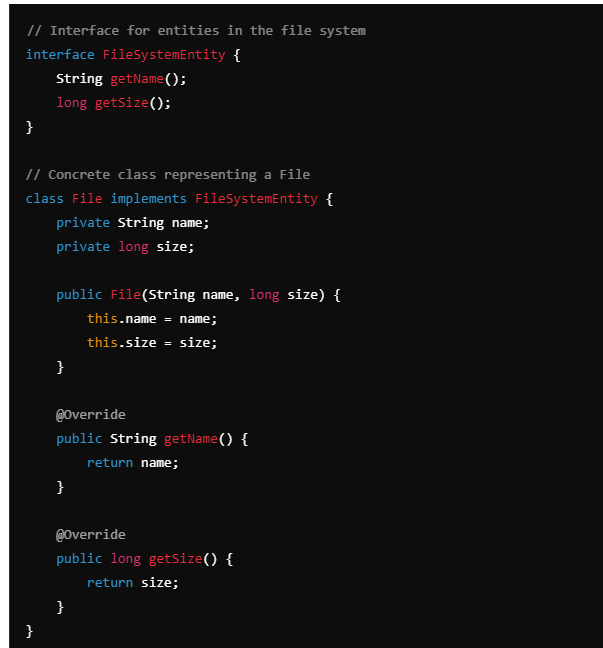
Question 3

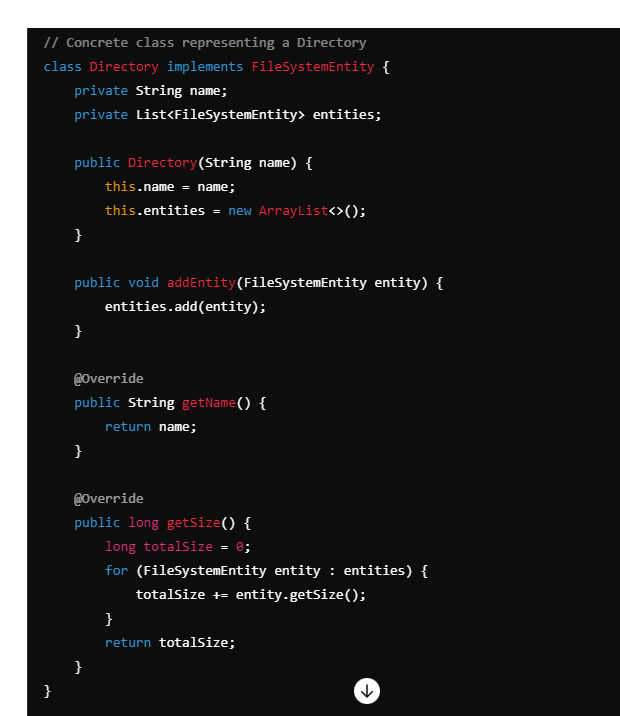
OOPS Concepts: Design entities and behavior for File System. Expose a method to calculate the size of the any entity in the file system. You are expected to give interfaces, classes, hierarchy between classes.

Identified the right entities/interfaces and relation between them?  
 the isA and hasA relationship?  
about Association and Delegation

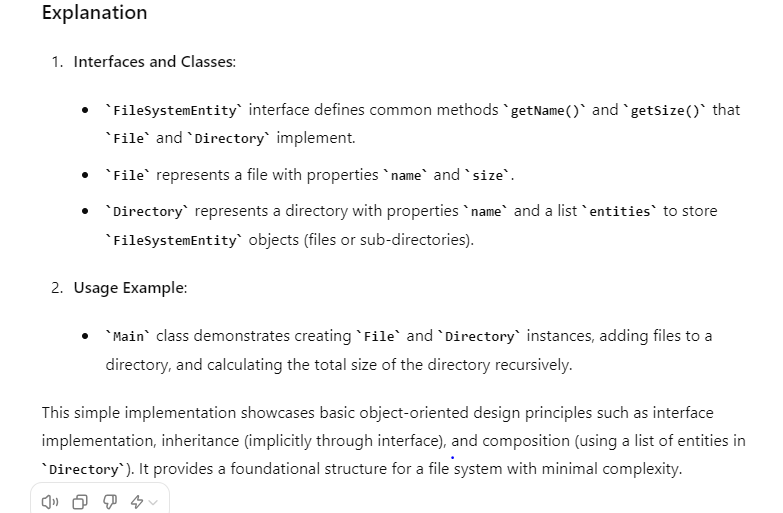
Solution

Certainly! Here's a simplified and concise Java implementation that demonstrates the design of a file system using basic object-oriented principles:









Question 4:-

Table Name : Student Column :

Name, Subject, Marks

Rama Math 50

Rama Physics 60

Rama English 45

Hari Math 70

Hari Physics 65

Hari English 85

Gita Math 90

Gita Physics 55

Gita English 80

Expected Output:

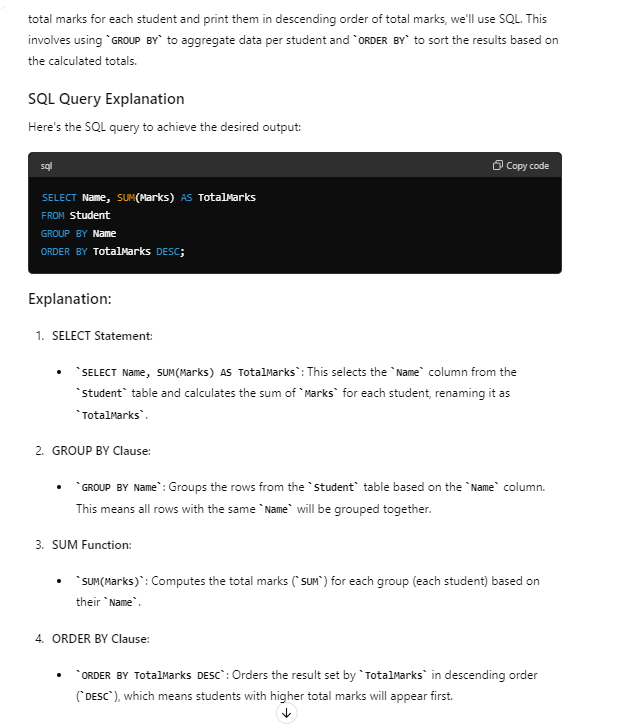
Gita 225

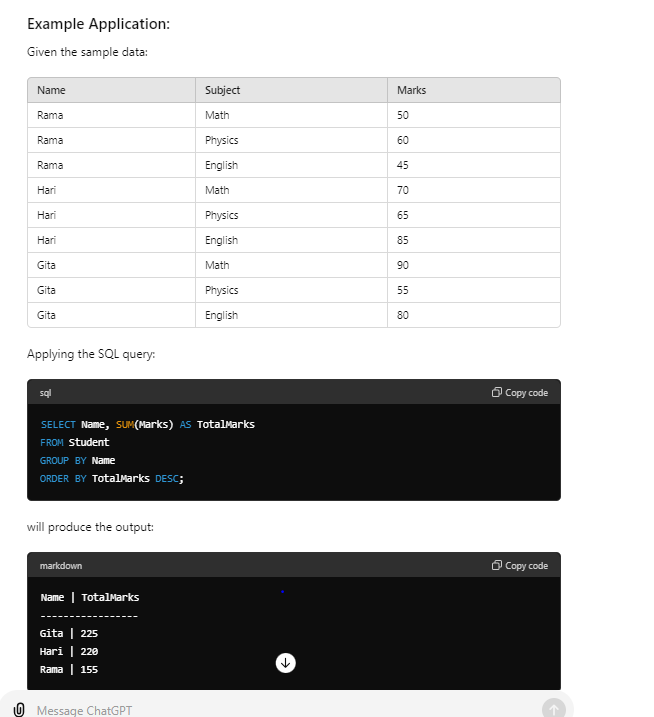
Hari 220

Rama 155

Print the Students In Notice board format. (i.e Calculate the Toal mark for an individual student, and print in descending order of total marks.)  
how Group by and Order by works  
SQL query/statement

Solution



****

**Question 5 :-**

Write a Java program to print a sequence of numbers up to N using 3 threads. For example, if we want to print a sequence of numbers up to 10 then it’ll look like this: THREAD-1 : 1 THREAD-2 : 2 THREAD-3 : 3 THREAD-1 : 4 THREAD-2 : 5 THREAD-3 : 6 THREAD-1 : 7 THREAD-2 : 8 THREAD-3 : 9 THREAD-1 : 10

**package** com;

**public** **class** ThreadSequence {

**private** **static** **final** **int** ***N*** = 10;

**private** **static** **int** *currentNumber* = 1;

**private** **static** **final** Object ***lock*** = **new** Object();

**private** **static** **int** *threadTurn* = 1;

**public** **static** **void** main(String[] args) {

// Create three threads with instances of PrintTask

Thread t1 = **new** Thread(**new** PrintTask(1), "THREAD-1");

Thread t2 = **new** Thread(**new** PrintTask(2), "THREAD-2");

Thread t3 = **new** Thread(**new** PrintTask(3), "THREAD-3");

// Start all three threads

t1.start();

t2.start();

t3.start();

}

// PrintTask class implementing Runnable for each thread

**static** **class** PrintTask **implements** Runnable {

**private** **final** **int** threadId;

**public** PrintTask(**int** threadId) {

**this**.threadId = threadId;

}

@Override

**public** **void** run() {

**while** (*currentNumber* <= ***N***) {

**synchronized** (***lock***) {

// Check if it's this thread's turn to print

**while** (*threadTurn* != threadId) {

**try** {

***lock***.wait(); // Wait if it's not this thread's turn

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

// Print the current number and thread name

System.***out***.println(Thread.*currentThread*().getName() + " : " + *currentNumber*);

*currentNumber*++;

// Update threadTurn to the next thread in round-robin fashion

*threadTurn* = (*threadTurn* % 3) + 1;

// Notify all threads waiting on the lock

***lock***.notifyAll();

}

}

}

}

}

Question 6: even and odd using thread

**package** com;

**public** **class** EvenOddDemo {

**private** **static** **int** *n* = 10;

**private** **static** Object *lock* = **new** Object();

**private** **static** **int** *count* = 1;

**public** **static** **void** main(String[] args) {

Thread even = **new** Thread(**new** EvenCheck());

Thread odd = **new** Thread(**new** OddCheck());

even.start();

odd.start();

}

**static** **class** EvenCheck **implements** Runnable {

@Override

**public** **void** run() {

**while** (*count* <= *n*) {

**synchronized** (*lock*) {

**while** (*count* % 2 != 0) {

**try** {

*lock*.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

System.***out***.println("Even Thread " + *count*);

*count*++;

*lock*.notifyAll();

}

}

}

}

**static** **class** OddCheck **implements** Runnable {

@Override

**public** **void** run() {

**while** (*count* <= *n*) {

**synchronized** (*lock*) {

**while** (*count* % 2 == 0) {

**try** {

*lock*.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

System.***out***.println("Odd Thread " + *count*);

*count*++;

*lock*.notifyAll();

}

}

}

}

}

Question 7

Explained the difference between Runnable and Callable. What are the pros and cons?

write the pseudo code

Runnable and Callable are both interfaces in Java used to represent tasks that can be executed by a thread or an executor. However, they have some key differences:

**Runnable**

* **Definition**: An interface representing a task that can be run by a thread.
* **Method**: void run()
* **Return Type**: Does not return a result and cannot throw checked exceptions.
* **Usage**: Suitable for tasks that do not return a result or do not need to handle checked exceptions.

**Pros**:

1. **Simple to use**: The interface is straightforward with a single run method.
2. **Compatibility**: Can be used with older Java APIs and frameworks.

**Cons**:

1. **No return value**: Cannot return a result.
2. **No checked exceptions**: Cannot throw checked exceptions.

**Callable**

* **Definition**: An interface representing a task that can be run by a thread and return a result.
* **Method**: V call() throws Exception
* **Return Type**: Returns a result of type V and can throw checked exceptions.
* **Usage**: Suitable for tasks that need to return a result or handle checked exceptions.

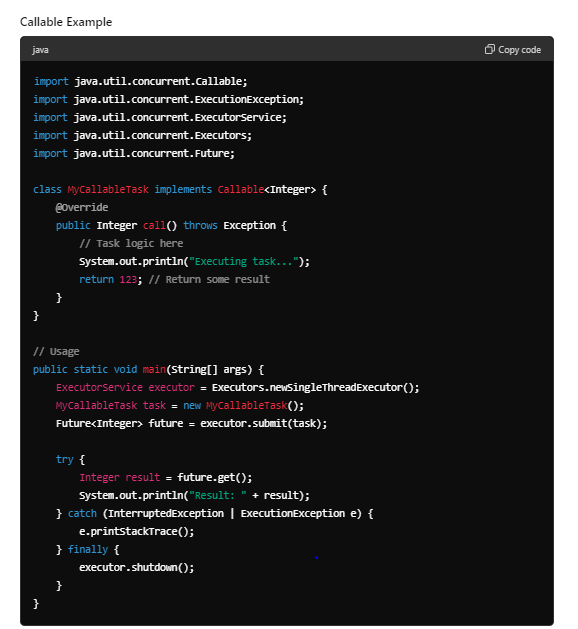
**Pros**:

1. **Return value**: Can return a result.
2. **Checked exceptions**: Can throw checked exceptions.

**Cons**:

1. **Slightly more complex**: Requires handling of Future objects when used with executors.
2. **Compatibility**: Not compatible with some older APIs and frameworks.





**Summary**

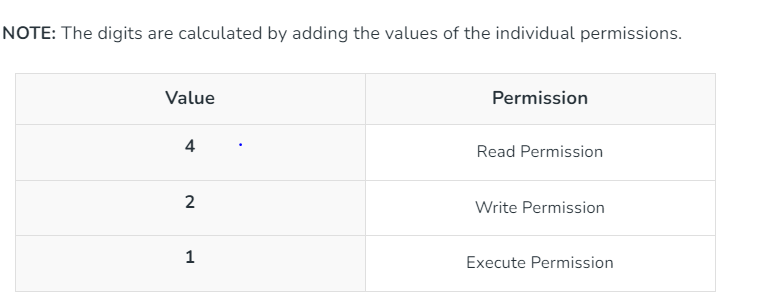
* **Runnable**: Simpler, no return value, no checked exceptions.
* **Callable**: More flexible, returns a value, can throw checked exceptions.

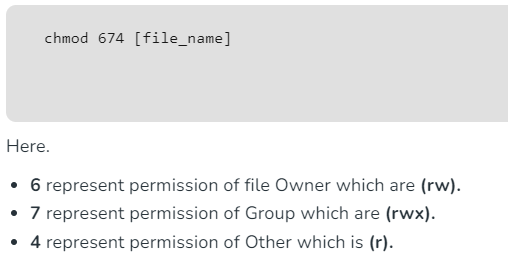
Question 8

Give me the command to find a specific file under a folder and its subfolders.

**find /home/user/documents -type f -name "example.txt"**

chmod 475 <FILE\_NAME> : what this command will do.

****



**Give me the command to find a specific file under a folder and its subfolders.**

find /path/to/folder -name "filename"

**Question 9 :) remove duplicate**

**package** com;

**public** **class** Demo {

**static** **int** *k*;

**public** **static** **void** main(String[] args) {

**int**[] array = { 1, 1, 2, 3, 4, 4, 5 };

**int**[] removeDuplicate = *removeDuplicate*(array);

**for** (**int** i = 0; i < *k*; i++) {

System.***out***.print(removeDuplicate[i] + " ");

}

}

**private** **static** **int**[] removeDuplicate(**int**[] array) {

**if** (array == **null** || array.length == 0) {

**return** array;

}

**int**[] newArray = **new** **int**[array.length];

**for** (**int** i = 0; i < array.length - 1; i++) {

**if** (array[i] != array[i + 1]) {

newArray[*k*++] = array[i];

}

}

newArray[*k*++] = array[array.length - 1];

**return** newArray;

}

}

======================= with single array

**public** **class** Demo

{

**public** **static** **void** main(String[] args) {

**int** arr[] = { 1, 2, 2, 3, 4, 4, 4, 5, 5 };

**int** n = arr.length;

// removeDuplicates() returns new size of array

n = *removeDuplicates*(arr, n);

// Print updated array

**for** (**int** i = 0; i < n; i++)

System.***out***.print(arr[i] + " ");

}

**static** **int** removeDuplicates(**int** arr[], **int** n){

**int** k=0;

**for**(**int** i=0;i<arr.length-1;i++){

**if**(arr[i]!=arr[i+1]){

arr[k++]= arr[i];

}

}

arr[k++]=arr[arr.length-1];

**return** k;

}

}

**Question 10:**

Follow up – Remove some duplicates in-place such that each unique element appears at most twice Example - Input: nums = [1,1,1,2,2,3] Output: 5, nums = [1,1,2,2,3,null]

**package** com;

**public** **class** Demo {

**static** **int** *k*;

**public** **static** **void** main(String[] args) {

**int**[] array = { 5, 5, 1, 1, 1, 2, 3 };

**int**[] removeDuplicate = *removeDuplicate*(array);

**for** (**int** i = 0; i < *k*; i++) {

System.***out***.print(removeDuplicate[i] + " ");

}

}

**private** **static** **int**[] removeDuplicate(**int**[] array) {

**if** (array.length < 2) {

**return** array;

}

**int**[] newArray = **new** **int**[array.length];

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

**if** (i < array.length - 2 && array[i] == array[i + 1] && array[i + 1] == array[i + 2]) {

**continue**;

} **else** {

newArray[*k*++] = array[i];

}

}

**return** newArray;

}

}

Using with one array

**public** **class** Demo

{

**public** **static** **void** main(String[] args) {

**int** arr[] = { 1, 2, 2, 2, 4, 4, 4, 5, 5 };

**int** n = arr.length;

// removeDuplicates() returns new size of array

n = *removeDuplicates*(arr, n);

// Print updated array

**for** (**int** i = 0; i < n; i++)

System.***out***.print(arr[i] + " ");

}

**static** **int** removeDuplicates(**int** arr[], **int** n){

**if**(arr.length<2){

**return** 0;

}

**int** k=0;

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

**if**(i<arr.length-2 &&arr[i]==arr[i+1] &&arr[i+1]==arr[i+2]) {

**continue**;

}**else** {

arr[k++]=arr[i];

}

}

**return** k;

}

}

Question 11 :--- flatten array of object

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

Object[] array = { 1, 2, **new** Object[] { 3, 4, **new** Object[] { 5 }, 6, 7 }, 8, 9, 10 };

List<Object> flattenArray = *flattenArray*(array);

System.***out***.println(flattenArray);

}

**private** **static** List<Object> flattenArray(Object[] array) {

**return** Arrays.*stream*(array)

.flatMap(o -> o **instanceof** Object[] ? *flattenArray*((Object[]) o).stream() : Arrays.*asList*(o).stream())

.collect(Collectors.*toList*());

}

}

Question 12

// write a program to print the sum of all odd numbers, double the number,

// square it and then get the sum.

List<Integer> list =Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Integer sum = list.stream().filter(x -> x % 2 != 0).map(x -> x \* 2).map(x -> x \* x).reduce(0,

(x, y) -> (x + y));

System.***out***.println(sum);

Question 13

Q: You are given an array, Print the output as below:

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

[9]

Solution

**public** **class** Demo {

**public** **static** **void** main(String[] args) {

**int**[] array = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

**int** elementsPerLine = 3;

**int** lines = array.length / elementsPerLine;

**int** remainder = array.length % elementsPerLine;

**int** index = 0;

**for** (**int** i = 0; i < lines; i++) {

System.***out***.print("[");

**for** (**int** j = 0; j < elementsPerLine; j++) {

System.***out***.print(array[index++]);

**if** (j < elementsPerLine - 1) {

System.***out***.print(", ");

}

}

System.***out***.println("]");

}

**if** (remainder > 0) {

System.***out***.print("[");

**for** (**int** i = 0; i < remainder; i++) {

System.***out***.print(array[index++]);

**if** (i < remainder - 1) {

System.***out***.print(", ");

}

}

System.***out***.println("]");

}

}

}

Question 14 Q: Write a program to print the characters which are matching in both the strings

**package** dom;

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** Test

{

**public** **static** **void** main(String[] args) {

String str1 = "programming";

String str2 = "language";

// Step 1: Count frequency of characters in str1 using HashMap

Set<Character> set = **new** HashSet<>();

**for**(**char** ch :str1.toCharArray()) {

set.add(ch);

}

**for**(**char** ch : str2.toCharArray()) {

**if**(set.contains(ch)) {

System.***out***.println(ch);

set.remove(ch);

}

}

}

}

**public** **class** TestNew {

**public** **static** **void** main(String[] args) {

String str1 = "programming";

String str2 = "language";

**boolean**[] visited = **new** **boolean**[256];

**for** (**char** ch : str1.toCharArray()) {

visited[ch] = **true**;

}

**for** (**char** ch : str2.toCharArray()) {

**if** (visited[ch]) {

System.***out***.println(ch);

visited[ch] = **false**;

}

}

}

}