1. [16 marks] Implement the following using Java streams. Any explicit looping/recur-

sive implementations and data structures are strictly not allowed.

(a) [3 marks] Complete the method myCount that takes in a Stream<T> of finite stream elements of generic type T, and returns the number of such elements. You are not allowed to use the terminal operation count(). For example,

• myCount(Stream.of("abc", "xyz")) returns 2;

• myCount(Stream.of())) returns 0;

ANSWER:

public static <T> long myCount(Stream<T> stream) {

return stream.reduce(0,(x,y)-> x+1);

}

(b) [4 marks] Complete the method countRepeats that takes in a string of lowercase letters and returns the number of occurrences of adjacent repeated letters. For example,

• the string “mississippi” has three occurrences;

• the string “ssss” has one occurrence

Hint: The following String method might be useful:  
char charAt(int index) — Returns the char value at the specified index.

ANSWER:

public static long countRepeats(String str) {

char[] array = str.toCharArray();

return IntStream.range(0,array.length-1).filter(x-> {return (array[x] == array[x+1])&& str.SubString(x).charAt(1) != array[x]);}).count();

}

page2image84242000

page3image20285056

(c) [6 marks] Complete the method variance that takes in an integer array of elements and returns the variance of the elements. The variance of an array of xi elements is defined as

􏰘n−1 (xk − μ)2 σ2= k=0

n−1  
where μ is the average of all n elements. For example,

• variance(IntStream.rangeClosed(1,6).toArray()) returns OptionalDouble[3.5];

• variance(IntStream.of().toArray()) returns OptionalDouble.empty;

ANSWER:

public static OptionalDouble variance(int[] data) {

OptionalDouble mean = OptionalDouble.ofNullable(Arrays.toStream(data).reduce(0,(x,y)->x+y));

OptionalDouble var = OptionalDouble.ofNullable(Arrays.toStream(data).map(x-> x-mean.get()).reduce(0,(x,y)-> x+y)/data.length)

Return var;

}

(d) [3 marks] Complete the method reverse that takes in a String and returns the reverse of the string while parallelizing the process.

public static String reverse(String str) {

return str.chars().parallel().

2. [6 marks] This question relates to the Discrete Event Simulator assignment.  
Before the simulation starts, customers are assigned the inter-arrival time and service

time using the RandomGenerator object rng in the following manner:

List<Customer> customers = new ArrayList<>();

double now = 0;  
for (int i = 0; i < numOfCustomers; i++) {

Customer customer = new Customer();

customer.setArrivalTime(now);

customer.setServiceTime(() -> rng.genServiceTime());

customers.add(customer);

now += rng.genInterArrivalTime();

}

Part of the Customer class is given below.

class Customer {

double serviceTime;

double arrivalTime;

void setArrivalTime(double arrivalTime) {

this.arrivalTime = arrivalTime;

}

void setServiceTime(double serviceTime) {

this.serviceTime = serviceTime;

}

double getArrivalTime() {

return this.arrivalTime;

}

double getServiceTime() {

return this.serviceTime;

}

:

:

It was found that during the simulation, depending on the order in which customers were served, the service times were no longer in the order in which it was generated. Re-write the above program fragments on the following page such that it retains the order of service times generated while the customers are being served.

List<Customer> customers = new ArrayList<>();

double now = 0;  
for (int i = 0; i < numOfCustomers; i++) {

Customer customer = new Customer();

customer.setArrivalTime(now);

customers.add(customer);

now += rng.genInterArrivalTime();

}

for (Customer c : customers) {

c.setServiceTime(() -> rng.genServiceTime());

}

class Customer {

double serviceTime;

double arrivalTime;

void setArrivalTime(double arrivalTime) {

this.arrivalTime = arrivalTime;

}

void setServiceTime(double serviceTime) {

this.serviceTime = arrivalTime + serviceTime;

}

double getArrivalTime() {

return this.arrivalTime;

}

double getServiceTime() {

return this.serviceTime;

}

3. [8 marks] You are given a Java program that implements a question-answer system using two types of question formats:

• MCQ: multiple-choice questions comprising answers: A B C D E • TFQ: true/false questions comprising answers: T F

The classes for MCQ and TFQ are given below:

class MCQ {

String question;

char answer;

public MCQ(String question) {

this.question = question;

}

void getAnswer() {  
System.out.print(question + " ");  
answer = (new Scanner(System.in)).next().charAt(0); if (answer < 'A' || answer > 'E') {

throw new InvalidMCQException("Invalid MCQ answer"); }

} }

class TFQ {

String question;

char answer;

public TFQ(String question) {

this.question = question;

}

void getAnswer() {  
System.out.print(question + " ");  
answer = (new Scanner(System.in)).next().charAt(0); if (answer != 'T' && answer != 'F') {

throw new InvalidTFQException("Invalid TFQ answer"); }

} }

In particular, an invalid answer to any of the questions will cause an exception (either InvalidMCQException or InvalidTFQException) to be thrown. These exceptions are sub-classes of the IllegalArgumentException class. An example is given below.

class InvalidMCQException extends IllegalArgumentException { public InvalidMCQException(String mesg) {

super(mesg);

}

}

The client class Main is provided to illustrate how the question-answer system works.

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

try {  
MCQ mcq = new MCQ("What is the answer to this MCQ?"); TFQ tfq = new TFQ("What is the answer to this TFQ?");

mcq.getAnswer();

tfq.getAnswer();  
} catch (InvalidMCQException ex) {

System.err.println(ex);  
} catch (InvalidTFQException ex) {

System.err.println(ex);

}

} }

A sample run for the above is given below. User input is underlined. Notice that the program terminates once an invalid answer is given.

What is the answer to this MCQ? Q InvalidMCQException: Invalid MCQ answer

To better manage the different types of questions, you are to design a more general question-answer class QA that can take the place of both MCQ and TFQ types of questions (and possibly more in future, each with their own type of exceptions).

You will need to show the following:

* The entire QA class;
* The changes needed for the existing exception classes;
* Any other new classes that are included;
* Modifications to the Main driver class above for the new design. Note that the sample output should still remain the same.

ANSWER:

Abstract Class QA

String question;

Char answer;

Public QA(String qn){

This.question = qn;

}

Public void answer(){

System.out.print(question+” “);

Char Ans =(new Scanner(System.in)).next().charAt(0);

getAnswer(char c);

}

Abstract void getAnswer(char c);

}

Class InvalidQAException extends IllegalArgumentException{

Public InvalidQAException (String msg){

Super(msg);

}

}

Class InvalidMCQException extends InvalidQAException

Class InvalidTFQException extends InvalidQAException

TFQ extends QA{

TFQ(String qn){

Super(qn);

}

Void getAnswer(char c){

if (answer != 'T' && answer != 'F') {

throw new InvalidTFQException("Invalid TFQ answer");

}

}

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

try {

MCQ mcq = new MCQ("What is the answer to this MCQ?");

TFQ tfq = new TFQ("What is the answer to this TFQ?");

mcq.answer();

tfq.answer();  
} catch (InvalidQAException ex) {

System.err.println(ex);  
}

}

}

4. [12 marks] You own a dating agency where your clients gets to indicate which other clients they want to date with the following conditions:

• There are two types of clients Man and Woman;  
• A Man can date another Woman, or vice-versa;  
• A Man can also date another Man, and so does the women;  
• A Man (or Woman) can only date one other Woman (or Man);  
• For simplicity, all man and woman woman have unique names.

Here is an example of a possible setup of two relationships involving three men(M) and one women(W):

• Mickey(M) dates Minnie(W) • Donald(M) dates Goofy(M)

Now suppose, there is a new relationship given by: • Daisy(W) dates Donald(M)

This would cause Donald to “breakup” with Goofy, so that Daisy can proceed to date Donald. Hence, there are still two dating relationships, but three men (including Goofy) and two women as tracked by the system.

A sample output for the above setup is given below:

Number of relationships: 2 Number of men: 3  
Number of women: 2

Mickey(M) and Minnie(W) are in a relationship Daisy(W) and Donald(M) are in a relationship Goofy(M) is not in a relationship

Your task is to design an OOP Java program to support the dating application. Take note of the following:

* This is a design question, so keep in mind the OOP concepts and design principles;
* There are only two import statements:
* import java.util.List;
* import java.util.ArrayList;
* You do not need to handle user input. Just write a test class to set up the above three relationships in sequence, so that it gives the desired output as shown above.

class Person {

Boolean Gender;//false is female true is male

String name;

Boolean relation;

Person(String name , Boolean gender){

This.name = name;

This.gender = gender

This.relation = false;

}

Public Boolean isMale(){

Return gender;

}

Public boolean isRelation (){

Return relation;

}

Public void setInRelation(){

Relation = True;

}

Public void setBrokeup(){

Retation = False;

}

@Override

Public Boolean equals(Object o){

If (o instanceof Person){

Person p = (Person) o;

Return p.name.equals(name) && p.gender== gender;

Else{

Return False;

}

}

@Override

Public String toString(){

Return (gender == true)? Name+”(M)”:Name+”(F)”;

}

}

Class Relationship {

List<Person> relation;

Relationship(Person p1, Person p2){

Relation = new ArrayList<>();

Relation.add(p1);

Relation.add(p2);

}

Public boolean contains(Person p){

Return Relation.contains(p);

}

public void breakup(){

for (Person p : relation){

p.setBrokeup();

}

}

Public String toString(){

Return relation.get(0)+” and “+relation.get(1)+” are in a relationship”;

}

}

Class Application{

Private List<Person> people;

Private List<Relationship> rel;

Application(){

People = new ArrayList<>();

Relationship = new ArrayList<>();

}

Public void AddUser(Person p){

People.add(p);

}

Public void addRelation(Person p1,Person p2){

Person a =People.get(People.indexof(p1));

Person b = People.get(People.indexof(p2));

For (Relationship r : rel){

If (r.contains(p1) || r.contains(p2)){

r.breakup();

Rel.remove(r);

}

}

a.setInRelation();

b.setInRelation();

Rel.add(new Relationship(a,b));

}

Public void print(){  
 int Male = people.stream().filter(x-> x.isMale()).count();

Int Female = people.stream().filter(x-> !x.isMale()).count();

List<Person> alone = people.stream().filter(x-> !x.isRelation()).collect(Arrays.toList());

String aloned = alone.stream().reduce(””,(x,y)-> x+”\n”+y.toString()+” is not in a relationship”);

String s = rel.stream().map(x-> x.toString()).reduce(“”, (x,y)-> x+”\n”+y)

String stat = “Number of relationships: “+ rel.size()+ “\nNumber of men: “+Male+”\nNumber of women: “+Female;

System.out.println(stat+ “\n”+s+aloned);

}

}

Test{

Public static void main(String[] args){

Application a = new Application();

Person p1,p2,p3,p4,p5;

P1 = new Person (“Mickey”, true)

P2 = new Person (“Minnie”, false)

P3 = new Person (“Donald”, true)

P4 = new Person (“Goofy”, true)

P5 = new Person (“Daisy”, false)

a.addUser(p1);

a.addUser(p2);

a.addUser(p3);

a.addUser(p4);

a.addUser(p5);

a.addRelation(p1,p2);

a.addRelation(p3,p4);

a.addRelation(p5,p3);

a.print();

}

}

* \*5. [8 marks] In the lecture, we have seen the SOLID principles as a set of guiding principles for designing OO programs. How are these principles applied in the context of streams and lambda expressions? In particular, you will need to describe the applicability with respect to the following three principles:
* (a) Single Responsibility Principle (b) Liskov-Substitution Principle
* (c) Open-closed Principle  
  Where appropriate, you should use sample program fragments to illustrate.

SOLID principles applied in use of streams and lambda expression could be said to include, the Single Responsibility Principle, where each single lambda expression instanciated has solely one responsibility to do a single task.

Towards the open-closed principle, the functional interfaces are open to extension by virtue of being Generically typed, allowing it to apply functional operations on any valid class, but are closed for modification of the underlying implementation, hence ensuring adherence to the Open Closed Principle.

In terms of the Liskov-Substitution Principle, through the use of Generic typing once again, functional operators can carry out the same kinds of operations on inheriting classes as that of their parent classes, through the use of Generic Types:

Function <? Extends T,R>

This allows the function to act on classes that inherit from the Type T.

For streams, they act as a declarative style of programming that declare what to do with a collection of data rather than how to do it. Such collections follow the L principle similarly through the use of Generic typing.

List<X> l = n

Stream<? Extends T> s(){

Return l.stream().map(x-> x.doSomething());

}

In this snippent of code, l is a list of X objects, of which some are Y objects, Y extends X.

doSomething() prints out a different String than that of the parent class.