**PSG COLLEGE OF TECHNOLOGY, COIMBATORE DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES**

**20XW57 – JAVA PROGRAMMING LAB**

**PROBLEM SHEET 6 – MULTI THREADED PROGRAMMING,**

1.Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers 90 81 78 95 79 72 85

Output:

The average value is 82

The minimum value is 72

The maximum value is 95

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited.

| import java.lang.\*;  import java.util.\*;  public class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  System.*out*.println("Enter the numbers separated by spaces: ");  ArrayList<String> arr = new ArrayList<>(Arrays.*asList*(sc.nextLine().split(" ")));  Average av = new Average(arr);  Minimum mi = new Minimum(arr);  Maximum ma = new Maximum(arr);  Thread t1 = new Thread(av);  Thread t2 = new Thread(mi);  Thread t3 = new Thread(ma);  t1.start();  t2.start();  t3.start();  System.*out*.println("A line at the end of program.");  }  }  class Average implements Runnable  {  ArrayList<String> arrL = new ArrayList<>();  Average(ArrayList<String> al) {  this.arrL = al;  }  public void run(){  int sum = 0;  for (String s : arrL) {  sum += Integer.*parseInt*(s);  }  System.*out*.println("The Average value is " + sum/arrL.size());  }  }  class Minimum implements Runnable  {  ArrayList<String> arrL = new ArrayList<>();  Minimum(ArrayList<String> al) {  this.arrL = al;  }  public void run(){  int min = Integer.*parseInt*(arrL.get(0));  for (String s : arrL) {  if (min > Integer.*parseInt*(s)) {  min = Integer.*parseInt*(s);  }  }  System.*out*.println("The Minimum value is " + min);  }  }  class Maximum implements Runnable {  ArrayList<String> arrL = new ArrayList<>();  Maximum(ArrayList<String> al) {  this.arrL = al;  }  public void run() {  int max = Integer.*parseInt*(arrL.get(0));  for (String s : arrL) {  if (max < Integer.*parseInt*(s)) {  max = Integer.*parseInt*(s);  }  }  System.*out*.println("The Maximum value is " + max);  }  } |
| --- |
|  |

2.Character Frequency - Multiple Threads

Your English literature friend is very happy with the code you gave him. Now for his research, he used your application to find character frequency in many novels. For larger novels, the application takes a lot of time for computation. So he called you on a fine Sunday to discuss this with you. He wanted to know whether you can improve the speed of the application.

You decided to modify the application by using multiple threads to reduce the computation time. For this, accept the number of counters or threads at the beginning of the problem and get the string for each counter or thread. Create a thread by extending the Thread class and take the user entered string as input. Each thread calculates the character frequency for the word assigned to that thread. All the counts are stored locally in the thread and once all the threads are completed print the character frequency for each of the threads.

Create a class Main.

Input and Output format:

Refer to sample Input and Output for formatting specifications.

Sample input and output:

Enter Number of Counters :

2

Enter text for counter 1 :

FrequencyCounter

Enter text for counter 2 :

JavaTheCompleteReference

Counter 1 Result :

C:1 F:1 c:1 e:3 n:2 o:1 q:1 r:2 t:1 u:2 y:1

Counter 2 Result :

C:1 J:1 R:1 T:1 a:2 c:1 e:7 f:1 h:1 l:1 m:1 n:1 o:1 p:1 r:1 t:1 v:1

| import java.lang.\*;  import java.util.\*;  public class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  System.*out*.println("Enter the no. of counters: ");  Integer noOfCounters = Integer.*parseInt*(sc.nextLine());  ArrayList<String> theList = new ArrayList<>();  for(int i = 0; i < noOfCounters; i++) {  System.*out*.print("Enter a string to freqCount: ");  theList.add(sc.nextLine());  }  System.*out*.println("Displaying the counts: ");  for(String s : theList) {  new Thread(new FrequencyCounter(s)).start();  }  }  }  class FrequencyCounter implements Runnable {  public String theString;  FrequencyCounter(String s) {  this.theString = s;  }  public void run() {  HashMap<Character, Integer> freqMap = new HashMap<>();  for (Character c : theString.toCharArray()) {  synchronized (this) {  if (freqMap.containsKey(c)) {  freqMap.put(c, freqMap.get(c) + 1);  }  else {  freqMap.put(c, 1);  }  }  }  synchronized (this) {  String result = "The freqCount of " + this.theString + " is: ";  for(Map.Entry entry : freqMap.entrySet()) {  result += entry.getKey() + ":" + entry.getValue() + " ";  }  System.*out*.println(result);  }  }  } |
| --- |
|  |

**3.User notification**

It is an important aspect for every application to be as user-friendly as possible. So that means it needs to send mails, messages whenever needed to the users. Write a thread program that sends the users notification after getting each of their names and number.

**Strictly adhere to the Object-Oriented specifications given in the problem statement. All class names, attribute names, and method names should be the same as specified in the problem statement.**

Create a Class **User** with the following attributes

| **Attributes** | **Datatype** |
| --- | --- |
| username | String |
| mobileNumber | String |

Generate getters and setters, create default and parameterized constructors in the following order: User(String userName, String mobileNumber)

Create a class **UserBO** which extends Thread class with the following **public** attributes,

| **Attributes** | **Datatype** |
| --- | --- |
| userList | List<User> |
| **static** message | List<String> |

Override the following methods in the **UserBO** class,

| **Method**  **name** | **Description** |
| --- | --- |
| public void run() | In this method, create a synchronized block and get the user name, and mobile number from the userList and create a notification message and add it to the message list.  The notification message should be like below, The message is sent to the user "userName" at the mobile number "mobileNumber" |

Get the number of users, number of users per thread, and user details and generate a list of notification for the users.

Create multi-threads according to the inputs and add users to the corresponding list in the thread. Create a driver class **Main** to test the above requirements.

**Input Format**

The first line of the input corresponds to the total number of users 'n'.

The second line of the input corresponds to the number of users per thread.

The next 'n' line of input contains user details (user name, mobile number) separated by comma[,].

**Output Format**

The output consists of the notification for all the users.

Refer to sample output for formatting specifications.

**[All text in bold corresponds to input and rest corresponds to output]**

**Sample Input/Output:**

Enter the number of users:

**6**

Enter the number of users per thread:

**2**

Enter the user details

**Sarah,9876543210**

**Richard,4567894564**

**Arya,4569871233**

**Stephanie,879879789**

**Sheldon,7878787987**

**Oliver,8989898987**

The message is sent to the user Sarah at the mobile number 9876543210

The message is sent to the user Richard at the mobile number 4567894564

The message is sent to the user Arya at the mobile number 4569871233

The message is sent to the user Stephanie at the mobile number 879879789 The message is sent to the user Sheldon at the mobile number 7878787987

The message is sent to the user Oliver at the mobile number 8989898987

| import java.lang.\*;  import java.util.\*;  public class Main {  public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  System.*out*.println("Enter the number of users: ");  Integer noOfUsers = Integer.*parseInt*(sc.nextLine());  System.*out*.println("Enter the number of users per thread: ");  Integer noOfUsersPerThread = Integer.*parseInt*(sc.nextLine());  System.*out*.println("Enter the user details: ");  List<String> users = new ArrayList<>();  List<User> userList = new ArrayList<>();  for (int i = 0; i < noOfUsers; i++) {  users.add(sc.nextLine());  }  sc.close();  for (String s : users) {  userList.add(new User(s.split(",")[0], s.split(",")[1]));  }  // Adding to the messages  try {  *addMessages*(noOfUsers, noOfUsersPerThread, userList);  } catch (InterruptedException e) {  e.printStackTrace();  }  // Printing the messages  System.*out*.println();  for (String msg : UserBO.*message*) {  System.*out*.println(msg);  }  }  public synchronized static void addMessages(Integer noOfUsers, Integer noOfUsersPerThread, List<User> userList)  throws InterruptedException {  for (int i = 0; i < noOfUsers; i += noOfUsersPerThread) {  Thread t = new UserBO(userList.subList(i, i + noOfUsersPerThread));  t.start();  t.join();  }  }  }  class User {  private String username;  private String mobileNumber;  public User() {  }  public User(String username, String mobileNumber) {  this.username = username;  this.mobileNumber = mobileNumber;  }  public String getUsername() {  return username;  }  public void setUsername(String username) {  this.username = username;  }  public String getMobileNumber() {  return mobileNumber;  }  public void setMobileNumber(String mobileNumber) {  this.mobileNumber = mobileNumber;  }  @Override  public String toString() {  return "User{" +  "username='" + username + '\'' +  ", mobileNumber='" + mobileNumber + '\'' +  '}';  }  }  class UserBO extends Thread {  public List<User> userList;  public static List<String> *message* = new ArrayList<>();  public UserBO(List<User> userList) {  this.userList = userList;  }  @Override  public void run() {  synchronized (this) {  for (User u : userList) {  *message*.add("The message is sent to the user " + u.getUsername() + " at the mobile number "  + u.getMobileNumber());  }  }  }  } |
| --- |
|  |

4.Three students A, B and C of B.Tech-IT II year contest for the PR election. With the total strength of 240 students in II year, simulate the vote casting by generating 240 random numbers (1 for student A, 2 for B and 3 for C) and store them in an array. Create four threads to equally share the task of counting the number of votes cast for all the three candidates.

| import java.lang.\*;  import java.util.\*;  public class Main {  public static int *counter1* = 0;  public static int *counter2* = 0;  public static int *counter3* = 0;  public static void main(String[] args) {  Integer[] votes = new Integer[240];  for (int i = 0; i < 240; i++)  votes[i] = (int) Math.*floor*(Math.*random*() \* 3 + 1);  List<Integer> voteList = Arrays.*asList*(votes);  // Four threads to do the counting  try {  for (int i = 0; i < 4; i++) {  Thread t = new Thread(new Counter(voteList.subList(i \* 60, (i + 1) \* 60)));  t.start();  t.join();  }  } catch (InterruptedException e) {  e.printStackTrace();  }  // Printing the results  System.*out*.println("Candidate 1: " + *counter1*);  System.*out*.println("Candidate 2: " + *counter2*);  System.*out*.println("Candidate 3: " + *counter3*);  }  }  class Counter implements Runnable {  private List<Integer> voteList;  public Counter(List<Integer> voteList) {  this.voteList = voteList;  }  public synchronized void run() {  for (Integer i : voteList) {  if (i == 1)  Main.*counter1*++;  else if (i == 2)  Main.*counter2*++;  else  Main.*counter3*++;  }  }  } |
| --- |
|  |

Use synchronized method or synchronized block to update the three count variables. The main thread should receive the final vote count for all three contestants and hence decide the PR based on the values received.

5.Write a program that demonstrates thread cooperation. Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.

6.Write a program for interthread communication process. In this they have three classes consumer, producer and stock.



**Additional Programs for Practise**

**Ticket Booking**

In our application, booking and cancelling tickets are some important features. As every user uses it to book/cancel tickets for every event/shows/fair. So we need to make it as quick and safe as possible. We are gonna use the synchronized method in this threading problem to get the First-come First-served basis. When one booking is done another will wait due to the synchronized method. Thus booking/cancellation can be done without much confusion.

**Strictly adhere to the Object-Oriented specifications given in the problem statement. All class names, attribute names and method names should be the same as specified in the problem statement.**

Create a class **Seat** with the following private attributes,

| **Attribute** | **Datatype** |
| --- | --- |
| seatNumber | Integer |
| Booked | Boolean |

Create default constructor and a parameterized constructor with arguments in order **Seat(Integer seatNumber, Boolean booked).**

Create appropriate getters and setters.

Create a class **TicketBookingThread** which extends **Thread** with the following private attributes,

| **Attribute** | **Datatype** |
| --- | --- |
| startingSeat | Integer |
| endingSeat | Integer |
| isBooking | Boolean |

Create appropriate getters and setters.

Create default constructor and a parameterized constructor with arguments in order **TicketBookingThread(Integer startingSeat, Integer endingSeat, Boolean isBooking, Runnable target)**.

Create a class **Booking** which implements **Runnable** with the following attributes,

| **Attribute** | **Datatype** |
| --- | --- |
| private List<Seat> list; | List<Seat> |

Create appropriate getters and setters.

Create the following methods in **Booking** class.

| **Method** | **Description** |
| --- | --- |
| public void run() | This function runs the thread and calls  the **doBooking()**  method each time.  It displays the statements of acknowledgement. |

| book/cancel them.  public synchronized Boolean doBooking(Integer  true,  startingSeat, Integer  endingSeat,Boolean isBooking) it returns false. | This method checks the availability of seats and After successful booking/cancellation, it returns  else if the ticket to be booked is not available or the ticket to be cancelled is not done at all, then |
| --- | --- |

**Note:** Print "The seats (startingSeat)-(endingSeat) are booked" for successful booking and "The seats (startingSeat)-(endingSeat) are cancelled" for successful cancellation. And in case of failure to book/cancel print "The booking/cancellation cannot be done".

Print these acknowledgements in the run() method.

**Input Format:**

The input for Booking tickets should be given as startingSeat,endingSeat, Booking. **Example:**

**45,67,Booking** - for booking all tickets from 45 to 67.

The input for cancellation startingSeat,endingSeat, Cancellation.

**Example:**

56,60,Booking – to cancel all tickets from 56 to 60.

**Output Format:**

The available seats are shown as a tabular form with 10 seats in each row.

Booked seats are represented by "**\***" and for the available seats, the seat numbers are shown. Leave a space between all the seats.

Refer sample Input and Output for formatting specifications.

Create a driver class **Main** and use the main method for getting inputs and displaying the available seats at the last.

**Sample Input and Output 1:**

**[All Texts in bold corresponds to the input and rest are output]**

The available seats are:

1 2 3 4 5 6 7 8 9 10

11 12 13 14 15 16 17 18 19 20

21 22 23 24 25 26 27 28 29 30

31 32 33 34 35 36 37 38 39 40

41 42 43 44 45 46 47 48 49 50

51 52 53 54 55 56 57 58 59 60

61 62 63 64 65 66 67 68 69 70

71 72 73 74 75 76 77 78 79 80

81 82 83 84 85 86 87 88 89 90

91 92 93 94 95 96 97 98 99 100

Enter the number of Bookings:

**4**

**2,96,Booking**

The seats 2-96 are booked

**95,97,Cancellation**

The booking/cancellation cannot be done

**3,7,Cancellation**

The seats 3-7 are cancelled

**4,6,Booking**

The seats 4-6 are booked

The available seats are:

1 \* 3 \* \* \* 7 \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

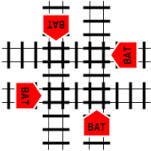
\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* \* \* \* \*

\* \* \* \* \* \* 97 98 99 100

**The BATMAN**

In a heroic effort to meet increasingly tighter shipping deadlines, a company has employed a *Bidirectional Autonomous Trolley* (BAT) system to move products from its warehouses to the delivery trucks. Each BAT is a mobile platform that travels on separate tracks back and forth between a warehouse and a truck. Because the goods are fragile, the tracks are perfectly leveled, which requires the placement of level crossings between warehouses. At most one BAT can cross at a time. Traffic at the crossing arriving from the right has the right of way. But this presents a problem, as the company soon found out when a simultaneous shipment of plastic penguins and black umbrellas caused the system to come to a grinding halt. Two BATs with the shipments and two other BATs returning to the warehouses were deadlocked at a level crossing:



In our system each BAT is controlled by a separate thread. It is your task to create BATMAN: a BAT Manager that prevents deadlock at a crossing. Part of the solution is to use one mutex lock for the crossing, so that at most one BAT at a time can pass. The mutex lock will act as a monitor for the BAT operations at the crossing. Besides the mutex lock, we will also need a set of condition variables to control the synchronization of the BATs.

We need a condition variable per BAT to queue BATs arriving from one direction (NorthQueue, EastQueue, SouthQueue, WestQueue). For example, when a BAT from North is already at the crossing, a second BAT from North will have to wait.

Another type of condition variable is needed to let BATs from the right have precedence to cross (NorthFirst, EastFirst, SouthFirst, WestFirst). However, using this rule can cause starvation. To prevent starvation, when a BAT is waiting to cross but BATs continuously arriving from the right have the right of way, we will let a BAT that just passed the crossing signal a waiting BAT on his left.

When deadlock occurs the BAT Manager must signal one of the BATs to proceed, e.g. the BAT from North. You will need a counter for each direction to keep track of the number of BATs waiting in line.

The program must take a string of 'n', 's', 'e', 'w' from the command line indicating a sequence of arrivals of BATs from the four directions. For example:

$ ./batman nsewwewn

BAT 4 from West arrives at crossing

BAT 2 from South arrives at crossing

BAT 1 from North arrives at crossing

BAT 3 from East arrives at crossing

DEADLOCK: BAT jam detected, signalling North to go

BAT 1 from North leaving crossing

BAT 3 from East leaving crossing

BAT 2 from South leaving crossing

BAT 4 from West leaving crossing

BAT 6 from East arrives at crossing

BAT 5 from West arrives at crossing

BAT 8 from North arrives at crossing

BAT 5 from West leaving crossing

BAT 6 from East leaving crossing

BAT 8 from North leaving crossing

BAT 7 from West arrives at crossing

BAT 7 from West leaving crossing

Note: the ordering of the above arrivals and departures may vary between runs and implementations. You don't need to produce exactly the same output.