

Exercise 11

Hariesh R - 23110344

Aim:

1. The aim of this program is to create a multithreaded application by subclassing the Thread class. Two thread objects are initialized and started, which execute concurrently. Each thread displays the message "Today is hot, humid, and sunny."
2. The program creates two threads using the Runnable interface. One thread prints "Hi" and the other prints "AI," each with a time interval of 300 milliseconds. These threads run concurrently, continuously printing their respective messages with the specified delay.
3. This Java program simulates a bank account where multiple threads concurrently perform deposits and withdrawals. Inter-thread communication methods like wait() and notify() are used to synchronize the operations, ensuring that withdrawals only occur when there is enough balance, and deposits notify waiting threads when funds become available.
4. This Java program defines a generic method that takes a list of numbers and calculates the sum of all the even and odd numbers separately. The method iterates through the list, checks whether each number is even or odd, and then returns the sums of both types of numbers.

Algorithm:

1. Q1)
 1. Define the Thread Class:
 2. Create a class myThread extending Thread.
 3. Override the run() method to print:
 4. "Today is hot"
 5. "Today is humid"
 6. "Today is sunny"
 7. Create the Main Class:

8. Define the Main class with the main method.
9. Instantiate and Start Threads:
 10. Create two myThread instances (thread1 and thread2).
 11. Call start() on both threads to begin execution.
 12. Wait for Threads to Complete:
 13. Use a try-catch block to call join() on both threads to ensure they finish before proceeding.
 14. Print any exception message if an error occurs.

2. Q2)

1. Define the Runnable Class:
 2. Create a class myThread that implements the Runnable interface.
 3. Declare a private final String message to hold the message.
 4. Define a constructor to initialize the message.
 5. Override the run() method to continuously print the message in a loop.
 6. Add Sleep and Interrupt Handling:
 7. Inside the loop, use Thread.sleep(300) to pause for 300 milliseconds after printing the message.
 8. Catch InterruptedException, interrupt the current thread, and break the loop if the thread is interrupted.
 9. Create the Main Class:
 10. Define the Main class with the main method.
 11. Instantiate and Start Threads:
 12. Create two Thread objects (thread1 and thread2), passing myThread instances with "HI" and "AI" as messages.
 13. Call start() on both threads to begin execution.
 14. Wait for 30 Seconds:
 15. Use a try-catch block to call Thread.sleep(30000) in the main thread to allow the child threads to run for 30 seconds.
 16. Interrupt Threads:
 17. After 30 seconds, call interrupt() on both threads to stop their execution.

3. Q3)

1. Initialize Components:
2. Create instances for Bank, RandomGenerator, List<Thread>, and Scanner.
3. Get User Input:
4. Prompt the user for the initial balance, number of transactions, and waiting time (in seconds).
5. Create Bank Instance:
6. Initialize a Bank object with the user-defined initial balance and wait time.
7. Generate Transactions:
8. Loop for the specified number of transactions:
9. Use RandomGenerator to determine the transaction type (deposit or withdraw) and amount.
10. Print the transaction details.
11. Call createAndStartThread() to create and start a new thread for each transaction.
12. Wait for Threads to Complete:
13. After starting all threads, loop through the threads list and call join() on each to wait for their completion.
14. Check for Failed Transactions:
15. After all threads finish, check if there are any failed transactions in the bank account.
16. Print "No Failed Transactions" if the list is empty.
17. Otherwise, print the list of failed transactions.
18. Display Final Balance:
19. Print the final account balance after all transactions are processed.
20. Define Thread Classes:
21. DepositTask: Inherits from Thread, calls the deposit() method on the Bank object and simulates a delay.
22. WithdrawTask: Inherits from Thread, calls the withdraw() method on the Bank object, handles exceptions, and simulates a delay.
23. Define Bank Class:
24. Implement methods for depositing, withdrawing, checking balance, and handling failed transactions using synchronized blocks to ensure thread safety.
25. Define RandomGenerator Class:

26. Create methods for generating random transaction types and amounts.
27. Define WithdrawalTimeoutException Class:
28. Create a custom exception class for handling withdrawal timeouts.

4. Q4)

1. Initialize Main Class:
2. Define the Main class with the main method.
3. Create a List:
4. Initialize an ArrayList<Integer> and populate it with integers from 0 to 20.
5. Call Generic Method:
6. Invoke myClass.myMethod() and pass the list of numbers as an argument.
7. Define the Generic Method:
8. Create a static method myMethod in myClass that accepts a List<T> where T extends Number.
9. Initialize Sum Variables:
10. Declare and initialize oddSum and evenSum to 0.
11. Iterate Through the List:
12. Loop through each number in the list:
13. Check if the number is even or odd using the modulus operator (%).
14. Add the number to evenSum if it's even, or to oddSum if it's odd.
15. Print the Results:
16. After processing all numbers, print the sums of odd and even numbers.

Source Code:

1. Q1)

```
package Exercise11.Q1;  
  
class myThread extends Thread {
```

```

@Override
public void run (){

    System.out.println("Today is hot");
    System.out.println("Today is humid");
    System.out.println("Today is sunny");
}
}

public class Main {

    public static void main(String[] args) {

        myThread thread1 = new myThread();
        myThread thread2 = new myThread();

        thread1.start();
        thread2.start();

        // Wait for the threads to complete
        try {

            thread1.join();
            thread2.join();
        } catch (Exception exception){

            System.out.println(exception.getMessage());
        }
    }
}

```

2. Q2)

```

package Exercise11.Q2;

class myThread implements Runnable {

    private final String message;

    public myThread(String message){

        this.message = message;
    }
}

```

```

@Override
public void run(){

    while (true){

        System.out.println(message);

        try {

            Thread.sleep(300);
        } catch (InterruptedException exception){

            Thread.currentThread().interrupt();
            break;
        }
    }
}

public class Main {

    public static void main(String[] args) {

        Thread thread1 = new Thread(new myThread("HI"));
        Thread thread2 = new Thread(new myThread("AI"));

        thread1.start();
        thread2.start();

        // Wait Main Thread For 3 seconds
        try {

            Thread.sleep(3000);
        } catch (InterruptedException exception){

            System.out.println(exception.getMessage());
        }

        // Kill the threads
        thread1.interrupt();
        thread2.interrupt();
    }
}

```

```
}
```

3. Q3)

```
package Exercise11.Q3;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Bank account;
        RandomGenerator rand;
        List<Thread> threads;
        Scanner scanner;

        String transactionType;
        double amount;
        int transactionCount, initialBalance;
        long waitTime;

        {
            scanner = new Scanner(System.in);
            rand = new RandomGenerator();
            threads = new ArrayList<>();
        }

        System.out.print("Enter Initial Balance: ");
        initialBalance = scanner.nextInt();

        System.out.print("Enter No Of Transactions: ");
        transactionCount = scanner.nextInt();

        System.out.print("Enter Waiting Time In Seconds: ");
        waitTime = scanner.nextLong() * 1000;

        {
            account = new Bank(initialBalance, waitTime);
```

```

    }

    for (int i = 1; i < transactionCount + 1; i++){

        transactionType = rand.deposit_withdraw();
        amount = rand.getAmount();

        System.out.println("Starting Transaction " + i + " " + transactionType + " " +
"Amount: " + amount);

        createAndStartThread(account, amount, transactionType + " id:" + i, threads,
transactionType);
    }

    // Wait for all the threads to complete
    for (Thread thread : threads){

        try {

            thread.join();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
            Thread.currentThread().interrupt();
        }
    }

    if (account.getFailedTransactions().isEmpty()) {

        System.out.println("No Failed Transactions.");
    }
    else {

        System.out.println("Failed Transactions:");
        account.getFailedTransactions().forEach(Main::log);
    }

    System.out.println("\nFinal Account Balance: " + account.getAmount());
}

private static void createAndStartThread (Bank bank, double amount, String name,
List<Thread> threads, String type){

```

```

    Thread transactionThread;

```



```

        if (type.equals("deposit")){
            transactionThread = new DepositTask(bank, amount, name);

            // If current balance is low, prioritize deposits to avoid failed withdrawals
            if (bank.getAmount() < 1000)
                transactionThread.setPriority(Thread.MAX_PRIORITY);

            else
                transactionThread.setPriority(Thread.NORM_PRIORITY);
        }

        else {

            transactionThread = new WithdrawTask(bank, amount, name);

            // If balance is sufficient, prioritize withdrawals to avoid long wait times
            if (bank.getAmount() >= amount)
                transactionThread.setPriority(Thread.MAX_PRIORITY);

            else
                transactionThread.setPriority(Thread.NORM_PRIORITY);
        }

        transactionThread.start();
        threads.add(transactionThread);
    }

    private static void log (String message){

        System.out.println("[ " + java.time.LocalDateTime.now() + " ] " + message);
    }
}

class Bank {

    private double amount;
    private final long waitTime;
    private final List<String> failedTransactions;

    {
        failedTransactions = new ArrayList<>();
    }
}

```

```

public Bank(){

    this.amount = 0.0;
    waitTime = 5000;
}

public Bank(double amount, long waitTime){

    this.amount = amount;
    this.waitTime = waitTime;
}

public Bank(int amount, long waitTime){

    this((double) amount, waitTime);
}

public synchronized void deposit (double amount){

    this.amount += amount;

    System.out.println("Deposited: " + amount);
    this.printBalance();

    notifyAll();
}

public synchronized void withdraw (double amount) throws InterruptedException,
WithdrawalTimeoutException{

    long startTime = System.currentTimeMillis();

    while (amount > this.amount){

        long elapsedTime = System.currentTimeMillis() - startTime;
        long remainingTime = waitTime - elapsedTime;

        if (remainingTime <= 0){

            String errorMessage = "Timeout waiting for deposits for withdrawal of
amount: " + amount;

```

```

        failedTransactions.add(Thread.currentThread().getName());

        throw new WithdrawalTimeoutException(errorMessage);
    }

    System.out.println("Insufficient Funds! Waiting For Deposits...");
    wait(remainingTime);
}

this.amount -= amount;
System.out.println("Withdrawn: " + amount);
this.printBalance();
}

public synchronized double getAmount(){

    return amount;
}

private void printBalance(){

    System.out.println("Balance: " + Math.round(this.amount * 100.0) / 100.0);
}

public List<String> getFailedTransactions(){
    return failedTransactions;
}
}

class DepositTask extends Thread {

    private final Bank account;
    private final double amount;

    public DepositTask(Bank account, double amount, String name){

        super(name);
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run (){

```

```

        account.deposit(amount);

        try {

            // Depositing Delay
            Thread.sleep(1000);

        } catch (InterruptedException exception){

            Thread.currentThread().interrupt();
        }
    }
}

```

```

class WithdrawTask extends Thread {

    private final Bank account;
    private final double amount;

    public WithdrawTask(Bank account, double amount, String name){

        super(name);
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run (){

        try {

            account.withdraw(amount);

            // Withdrawing Delay
            Thread.sleep(2000);

        } catch (InterruptedException | WithdrawalTimeoutException exception){

            System.out.println("Withdrawal failed: " + exception.getMessage());
            Thread.currentThread().interrupt();
        }
    }
}

```

```
}  
}
```

```
class RandomGenerator {
```

```
    private int min_amount;  
    private int max_amount;  
    private final Random random;
```

```
    {  
        min_amount = 200;  
        max_amount = 500;  
        random = new Random();  
    }
```

```
    public RandomGenerator(){  
  
    }
```

```
    public RandomGenerator(int min_amount, int max_amount){  
        this.min_amount = min_amount;  
        this.max_amount = max_amount;  
    }
```

```
    public String deposit_withdraw(){  
  
        return (random.nextBoolean()) ? "deposit" : "withdraw";  
    }
```

```
    public double getAmount(){  
  
        double amount = min_amount + random.nextDouble() * (max_amount -  
min_amount);  
  
        return Math.round(amount * 100.0) / 100.0;  
    }
```

```
    public int getMin_amount() {  
        return min_amount;  
    }
```

```
    public void setMin_amount(int min_amount) {  
        this.min_amount = min_amount;
```

```

    }

    public int getMax_amount() {
        return max_amount;
    }

    public void setMax_amount(int max_amount) {
        this.max_amount = max_amount;
    }
}

class WithdrawalTimeoutException extends Exception {
    public WithdrawalTimeoutException(String message) {
        super(message);
    }
}

```

4. Q4)

```

package Exercise11.Q4;

import java.util.ArrayList;
import java.util.List;

public class Main {

    public static void main(String[] args) {

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

        for (int i = 0; i < 21; i++) list.add(i);

        myClass.myMethod(list);
    }
}

class myClass {

    public static <T extends Number> void myMethod(List<T> numbers) {

        int oddSum, evenSum;
    }
}

```

```
{
    oddSum = 0;
    evenSum = 0;
}

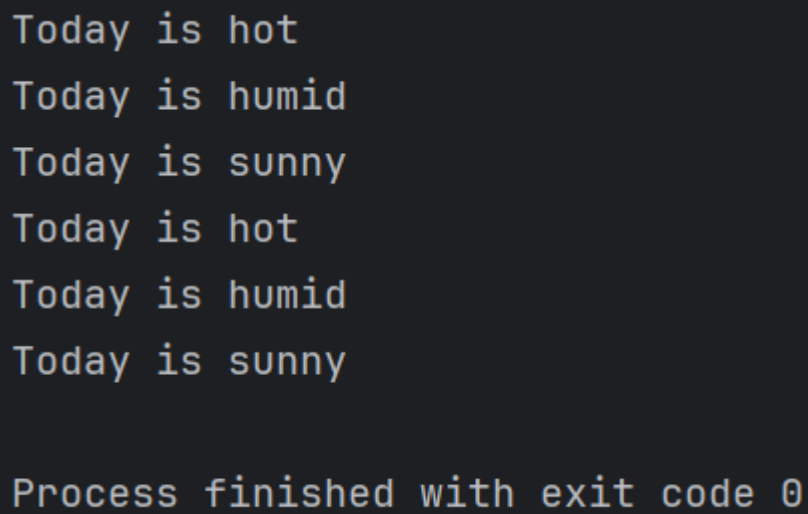
for (T number: numbers){

    if ((int) number % 2 == 0) evenSum += (int) number;
    else oddSum += (int) number;
}

System.out.println("Odd Number Sum: " + oddSum);
System.out.println("Even Number Sum: " + evenSum);
}
}
```

Output:

1.

A terminal window with a dark background and light-colored text. It displays six lines of weather-related strings: "Today is hot", "Today is humid", "Today is sunny", "Today is hot", "Today is humid", and "Today is sunny". At the bottom, it shows "Process finished with exit code 0".

```
Today is hot
Today is humid
Today is sunny
Today is hot
Today is humid
Today is sunny

Process finished with exit code 0
```

2.

```
HI
AI
AI
HI
AI
HI
HI
AI
HI
AI
AI
HI

Process finished with exit code 0
```

3.


```
Enter Initial Balance: 1000
Enter No Of Transactions: 10
Enter Waiting Time In Seconds: 3
Starting Transaction 1 withdraw Amount: 493.4
Starting Transaction 2 withdraw Amount: 326.7
Starting Transaction 3 deposit Amount: 273.31
Starting Transaction 4 deposit Amount: 497.53
Starting Transaction 5 deposit Amount: 447.56
Starting Transaction 6 withdraw Amount: 240.81
Starting Transaction 7 deposit Amount: 309.66
```

```
Withdrawn: 326.7
Balance: 673.3
Deposited: 447.56
Balance: 1120.86
Withdrawn: 240.81
Balance: 880.05
Deposited: 497.53
Balance: 1377.58
Deposited: 273.31
Balance: 1650.89
Withdrawn: 493.4
Balance: 1157.49
```

```
Starting Transaction 8 withdraw Amount: 361.0
Starting Transaction 9 withdraw Amount: 474.77
Starting Transaction 10 deposit Amount: 203.45
Deposited: 309.66
Balance: 1467.15
Withdrawn: 361.0
Balance: 1106.15
Deposited: 203.45
Balance: 1309.6
Withdrawn: 474.77
Balance: 834.83
No Failed Transactions.
```

```
Final Account Balance: 834.8299999999999
```

```
Process finished with exit code 0
```

4.

```
Odd Number Sum: 100
```

```
Even Number Sum: 110
```

```
Process finished with exit code 0
```

Result:

1. The result of the multithreaded Java program will display the messages "Today is hot," "Today is humid," and "Today is sunny" from two concurrently executing threads. The output may vary in order due to thread scheduling, but it will consist

of a total of six printed statements—three from each thread—resulting in the appearance of these phrases twice, potentially interleaved in different runs.

2. The result of the Java program that creates two threads using the Runnable interface will display the messages "Hi" and "AI" alternately, with each thread printing its message every 300 milliseconds. The output will consist of these two phrases appearing in quick succession, potentially interleaved, depending on the thread scheduling. The program will continue printing "Hi" and "AI" until the main thread completes or the program is terminated.
3. The result of the Java program that simulates a bank account with concurrent deposits and withdrawals will display the results of various transactions processed by multiple threads. It will handle deposits and withdrawals while ensuring thread safety through inter-thread communication methods. The output will include messages indicating successful deposits or withdrawals, along with any failed transactions due to insufficient funds. Additionally, the program will print the final account balance after all transactions are completed, and if there are any failed transactions, those will also be displayed. The exact output may vary based on the timing of thread execution.
4. The result of the Java program that defines a generic method for calculating the sums of even and odd numbers from a list will display the total sums for each category. After processing the provided list of integers, the output will include two lines: one showing the sum of odd numbers and the other showing the sum of even numbers. For example, if the input list contains integers from 0 to 20, the output will specify the calculated sums, with the exact values depending on the numbers in the list.