Multithreading and Reactive Programming

Q1. Write a Java program to create a simple calculator that performs arithmetic operations (addition, subtraction, multiplication, division) using **multiple threads**. Each arithmetic operation should be handled by a separate thread.

```
class AdditionThread extends Thread {
    private int a, b;

    public AdditionThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void run() {
        System.out.println("Addition result: " + (a + b));
    }
}

class SubtractionThread extends Thread {
    private int a, b;

    public SubtractionThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void run() {
        System.out.println("Subtraction result: " + (a - b));
    }
}

class MultiplicationThread extends Thread {
    private int a, b;

    public MultiplicationThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void run() {
        System.out.println("Multiplication result: " + (a * b));
    }
}

class DivisionThread extends Thread {
    private int a, b;

    public DivisionThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    public void run() {
        if (b != 0) {
            System.out.println("Division result: " + ((double) a / b));
        } else {
            System.out.println("Cannot divide by zero");
        }
    }
}

public class MultiThreadCalculator {
    public static void main(String[] args) {
        // Create threads for each arithmetic operation
      AdditionThread add = new AdditionThread(10, 5);
      SubtractionThread sub = new SubtractionThread(10, 5);
      MultiplicationThread mul = new MultiplicationThread(10, 5);
      DivisionThread div = new DivisionThread(10, 5);

        // Start threads
        add.start();
        sub.start();
        mul.start();
        div.start();
    }
}
```

Q2. Write a Java program that uses reactive programming to read a file asynchronously. Use RxJava or another reactive library to handle the **file reading** and processing.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;

public class FileReadingReactiveExample {

    public static void main(String[] args) throws InterruptedException {
        // Publisher: File Reader
        SubmissionPublisher<String> filePublisher = new SubmissionPublisher<>();

        // Subscriber: File Processor
        Flow.Subscriber<String> fileSubscriber = new Flow.Subscriber<>() {
            private Flow.Subscription subscription;

            @Override
            public void onSubscribe(Flow.Subscription subscription) {
                this.subscription = subscription;
                System.out.println("File Processor: Subscribed to File Reader.");
                this.subscription.request(1); // Requesting one line from the file
            }

            @Override
            public void onNext(String line) {
                System.out.println("File Processor: Processing line: " + line);
                // Process the line here (in this example, simply printing)
                // In a real scenario, you would perform actual processing here
                this.subscription.request(1); // Request the next line from the
file
            }

            @Override
            public void onError(Throwable throwable) {
                System.err.println("File Processor: Error occurred: " + throwa-
ble.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("File Processor: File processing complete.");
            }
        };

        // Subscribing file processor to the file reader publisher
        filePublisher.subscribe(fileSubscriber);

        // Simulating file reading events
        String filePath = "D:\\info@.txt";
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath)))
{
            String line;
            while ((line = reader.readLine()) != null) {
                filePublisher.submit(line); // Publish each line from the file
            }
        } catch (IOException e) {
           System.out.println(e);
        }

        // Simulating a short delay before closing the publisher
        TimeUnit.SECONDS.sleep(1);

        // Closing the publisher after all lines are submitted
        filePublisher.close();
    }
}
```

Q3. Write a Java program to **multiply two matrices** using multithreading. Divide the task of multiply-ing rows of the matrices among multiple threads to improve performance.

```
import java.util.*;

public class MultithreadedMatrixMultiplication {
    public static void main(String[] args) {
        int[][] firstMatrix = generateMatrix(3, 3);
        int[][] secondMatrix = generateMatrix(3, 3);
        int[][] resultMatrix = new int[firstMatrix.length][secondMatrix[0].length];

        System.out.println("First Matrix: " + Arrays.deepToString(firstMatrix));
        System.out.println("Second Matrix: " + Arrays.deepToString(secondMatrix));

        int numThreads = firstMatrix.length; // One thread per row
        Thread[] threads = new Thread[numThreads];

        for (int i = 0; i < numThreads; i++) {
            threads[i] = new Thread(new MatrixMultiplier(firstMatrix, secondMatrix,
resultMatrix, i));
            threads[i].start();
        }

        // Wait for all threads to finish
        for (int i = 0; i < numThreads; i++) {
            try {
                threads[i].join();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        System.out.println("Result Matrix: " + Arrays.deepToString(resultMatrix));
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt(10);
            }
        }
        return matrix;
    }
}

class MatrixMultiplier implements Runnable {
    private int[][] matrixA;
    private int[][] matrixB;
    private int[][] result;
    private int row;

    public MatrixMultiplier(int[][] matrixA, int[][] matrixB, int[][] result, int
row) {
        this.matrixA = matrixA;
        this.matrixB = matrixB;
        this.result = result;
        this.row = row;
    }

    @Override
    public void run() {
        int columnsB = matrixB[0].length;
        int columnsA = matrixA[0].length;
        for (int j = 0; j < columnsB; j++) {
            result[row][j] = 0;
            for (int k = 0; k < columnsA; k++) {
                result[row][j] += matrixA[row][k] * matrixB[k][j];
            }
        }
    }
}
```

Q4. Implement a program where two **threads communicate** with each other using wait() and notify() methods. One thread should print even numbers, and the other should print odd numbers in sequence.

```
class SharedPrinter {
    private boolean isFlag = false;

    public synchronized void printEven(int number) {
        while (!isFlag) {
            try {
                wait();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
        System.out.println(number);
        isFlag = false;
        notify();
    }

    public synchronized void printOdd(int number) {
        while (isFlag) {
            try {
                wait();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
        System.out.println(number);
        isFlag = true;
        notify();
    }
}

class EvenThread implements Runnable {
    private SharedPrinter printer;
    private int range;

    public EvenThread(SharedPrinter printer, int range) {
        this.printer = printer;
        this.range = range;
    }

    @Override
    public void run() {
        for (int i = 1; i <= range; i++) {
            if(i % 2 == 0)
            printer.printEven(i);
        }
    }
}

class OddThread implements Runnable {
    private SharedPrinter printer;
    private int range;

    public OddThread(SharedPrinter printer, int range) {
        this.printer = printer;
        this.range = range;
    }

    @Override
    public void run() {
        for (int i = 1; i <= range; i++) {
            if(i % 2 != 0)
            printer.printOdd(i);
        }
    }
}
```

```
public class EvenOdd {

      public static void main(String[] args) {
            SharedPrinter printer = new SharedPrinter();
        int range = 10;

        Thread even = new Thread(new EvenThread(printer, range));
        Thread odd = new Thread(new OddThread(printer, range));

        odd.start();
        even.start();

      }
}
```

Q5. Implement a program that demonstrates the **use of locks** (e.g., ReentrantLock) for thread synchronization. Create a scenario where multiple threads access a shared resource, and use locks to ensure that only one thread can access the resource at a time.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class BookSeat {
    private int totalSeats = 10;
    private final Lock lock = new ReentrantLock();

    void bookSeat(int seats) {
        System.out.println("Thread: " + Thread.currentThread().getName() + " trying 
to book " + seats + " seats.");
        lock.lock();
        try {
            if (totalSeats >= seats) {
                System.out.println(seats + " seats booked successfully by " + 
Thread.currentThread().getName());
                totalSeats -= seats;
                System.out.println("Seats left: " + totalSeats);
            } else {
                System.out.println("Seats cannot be booked by " + 
Thread.currentThread().getName());
                System.out.println("Seats left: " + totalSeats);
            }
        } finally {
            lock.unlock();
        }
    }
}

public class SynchronizedBlockWithLock extends Thread {
    private static BookSeat bookSeat;
    private int seats;

    public SynchronizedBlockWithLock(int seats) {
        this.seats = seats;
    }

    public void run() {
        bookSeat.bookSeat(seats);
    }
    public static void main(String[] args) {
        bookSeat = new BookSeat();

        SynchronizedBlockWithLock thread1 = new SynchronizedBlockWithLock(7);
        thread1.setName("Thread 1");
        thread1.start();

        SynchronizedBlockWithLock thread2 = new SynchronizedBlockWithLock(6);
        thread2.setName("Thread 2");
        thread2.start();
    }
}
```

Output:

```
Thread: Thread 1 trying to book 7 seats.
Thread: Thread 2 trying to book 6 seats.
6 seats booked successfully by Thread 2
Seats left: 4
Seats cannot be booked by Thread 1
Seats left: 4
```

N.B.:

We replace the synchronized block with a ReentrantLock object named lock. We acquire the lock using lock.lock() before accessing the shared resource and release the lock using lock.unlock() after accessing the resource. The ReentrantLock ensures that only one thread can access the shared resource (book seats) at a time.

Q6. Write a Java program that **generates prime numbers** up to a given limit using multiple threads. Each thread should generate a subset of the prime numbers.

```
import java.util.*;

class PrimeNumberChecker implements Runnable {
    private List<Integer> primes;
    private int start;
    private int end;

    public PrimeNumberChecker(List<Integer> primes, int start, int end) {
        this.primes = primes;
        this.start = start;
        this.end = end;
    }

    @Override
    public void run() {
        for (int i = start; i <= end; i++) {
            if (isPrime(i)) {
                synchronized (primes) {
                    primes.add(i);
                }
                System.out.println("Prime: " + i);
            }
        }
    }

    private boolean isPrime(int number) {
        if (number <= 1) return false;
        for (int i = 2; i <= Math.sqrt(number); i++) {
```

```java
            if (number % i == 0) return false;
        }
        return true;
    }
}

public class PrimeGenerator {
    public static void main(String[] args) {
        List<Integer> primes = new ArrayList<>();
        int limit = 50;

        // Create a thread to check for primes from 1 to 25
        Thread thread1 = new Thread(new PrimeNumberChecker(primes, 1, 25));

        // Create a thread to check for primes from 26 to 50
        Thread thread2 = new Thread(new PrimeNumberChecker(primes, 26, 50));

        // Start the threads
        thread1.start();
        thread2.start();

        // Wait for both threads to finish
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Sort and print primes
        Collections.sort(primes);
        System.out.println("Prime numbers up to " + limit + ": " + primes);
    }
}
```

Q8. Develop a Java program that **analyzes real-time weather data** using reactive programming. The program should fetch weather data from a weather API asynchronously and perform analysis (e.g., temperature trends, rainfall predictions). Use a reactive approach to handle the asynchronous nature of weather data updates. Use reactive operators (e.g., map, filter) to process and analyze the weather data stream.

```java
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;

// WeatherData class representing the data structure of weather information
class WeatherData {
    private String city;
    private double temperature;
    private double rainfall;

    public WeatherData(String city, double temperature, double rainfall) {
        this.city = city;
        this.temperature = temperature;
        this.rainfall = rainfall;
    }

    // Getters for weather attributes
    public String getCity() {
        return city;
    }

    public double getTemperature() {
        return temperature;
    }

    public double getRainfall() {
        return rainfall;
    }
}

public class WeatherAnalysis {

    public static void main(String[] args) throws InterruptedException {
        // Create a publisher for weather data updates
        SubmissionPublisher<WeatherData> weatherPublisher = new SubmissionPublish-
er<>();

        // Create a subscriber for weather data analysis
        Flow.Subscriber<WeatherData> weatherAnalyzer = new
Flow.Subscriber<WeatherData>() {
            private Flow.Subscription subscription;

            @Override
            public void onSubscribe(Flow.Subscription subscription) {
                this.subscription = subscription;
                System.out.println("Weather Analyzer: Subscribed to weather data
updates.");
                this.subscription.request(1); // Requesting one data update
            }

            @Override
            public void onNext(WeatherData item) {
                // Perform analysis on weather data
                analyzeWeather(item);
                this.subscription.request(1); // Request the next data update
            }

            @Override
            public void onError(Throwable throwable) {
                System.err.println("Weather Analyzer: Error occurred: " + throwa-
ble.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("Weather Analyzer: Weather analysis complete.");
            }
        };

        // Subscribe the weather analyzer to the weather data publisher
        weatherPublisher.subscribe(weatherAnalyzer);

        // Simulate receiving weather data updates
        System.out.println("Weather API: Sending weather updates...");
        sendWeatherUpdates(weatherPublisher);
        // Simulate a delay to observe the updates
        TimeUnit.SECONDS.sleep(5);

        // Close the weather publisher after processing
        weatherPublisher.close();
    }

    // Method to simulate receiving weather data updates
    private static void sendWeatherUpdates(SubmissionPublisher<WeatherData> weath-
erPublisher) {
        // Simulate receiving weather updates for multiple cities
        weatherPublisher.submit(new WeatherData("New York", 25.6, 0.0));
        weatherPublisher.submit(new WeatherData("London", 18.2, 1.5));
        weatherPublisher.submit(new WeatherData("Tokyo", 30.5, 0.8));
        weatherPublisher.submit(new WeatherData("Sydney", 22.8, 3.2));
    }

    // Method to analyze weather data
    private static void analyzeWeather(WeatherData data) {
        // Example analysis: Display city and temperature if temperature is above
25°C
        if (data.getTemperature() > 25.0) {
            System.out.println("Weather Analyzer: High temperature alert in " + da-
ta.getCity() + ". Temperature: " + data.getTemperature() + "°C");
        }
    }
}
```

Output:

```
Weather API: Sending weather updates...
Weather Analyzer: Subscribed to weather data updates.
Weather Analyzer: High temperature alert in New York. Temperature: 25.6°C
Weather Analyzer: High temperature alert in Tokyo. Temperature: 30.5°C
Weather Analyzer: Weather analysis complete.
```

N.B.: The program uses reactive programming with a publisher-subscriber model to handle real-time weather data updates asynchronously.