CSW2 ASSIGNMENT 8
(CHAPTER 19 – MULTITHREADING)
SOLUTIONS

---

1. Write a Java program to demonstrate performing multiple tasks concurrently using multiple threads. Create two separate thread classes:
   - The first thread should calculate and print the sum of the first 100 natural numbers.
   - The second thread should display the multiplication table of a given number Start both threads from the **main()** method and show that the tasks run concurrently.

```
package q1;
class SumThread extends Thread {
    public void run() {
        int sum = 0;
        for (int i = 1; i <= 100; i++) {
            sum += i;
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                System.out.println("SumThread interrupted");
            }
        }
        System.out.println("Sum of first 100 natural numbers: " + sum);
    }
}
class MultiplicationThread extends Thread {
    private int number;
    public MultiplicationThread(int number) {
        this.number = number;
    }
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(number + " x " + i + " = " + (number * i));
            try {
                Thread.sleep(15);
            } catch (InterruptedException e) {
                System.out.println("MultiplicationThread interrupted");
            }
        }
    }
}
public class MultiTask {
    public static void main(String[] args) {
        SumThread sumThread = new SumThread();
        MultiplicationThread multiplicationThread = new MultiplicationThread(5);
        sumThread.start();
        multiplicationThread.start();
    }
}
```

OUTPUT:

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
Sum of first 100 natural numbers: 5050
```

2. 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.

```
package q2;
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: " + a + " + " + b + " = " + (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: " + a + " - " + b + " = " + (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: " + a + " * " + b + " = " + (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: " + a + " / " + b + " = " + ((double)
a / b));
        } else {
            System.out.println("Division: Cannot divide by zero.");
        }
    }
}
public class Calculator {
    public static void main(String[] args) {
        int num1 = 20, num2 = 5;
        AdditionThread additionThread = new AdditionThread(num1, num2);
        SubtractionThread subtractionThread = new SubtractionThread(num1, num2);
        MultiplicationThread multiplicationThread = new
MultiplicationThread(num1, num2);
        DivisionThread divisionThread = new DivisionThread(num1, num2);
        additionThread.start();
        subtractionThread.start();
        multiplicationThread.start();
        divisionThread.start();
    }
}
```

OUTPUT:

```
Addition: 20 + 5 = 25
Multiplication: 20 * 5 = 100
Subtraction: 20 - 5 = 15
Division: 20 / 5 = 4.0
```

3. Rewrite the multithreading calculator program from Q1 using **lambda expressions**. Each arithmetic operation (addition, subtraction, multiplication, division) should still be handled by a separate thread, but this time, define the behavior of each thread using Java lambda expressions.

```
package q3;
public class CalculatorUsingLambda {
    public static void main(String[] args) {
        int num1 = 20;
        int num2 = 5;
        Thread addThread = new Thread(() -> {
            System.out.println("Addition: " + num1 + " + " + num2 + " = " +
(num1 + num2));
        });
        Thread subtractThread = new Thread(() -> {
            System.out.println("Subtraction: " + num1 + " - " + num2 + " = " +
(num1 - num2));
        });
        Thread multiplyThread = new Thread(() -> {
            System.out.println("Multiplication: " + num1 + " * " + num2 + " = "
+ (num1 * num2));
        });
        Thread divideThread = new Thread(() -> {
            if (num2 != 0) {
                System.out.println("Division: " + num1 + " / " + num2 + " = " +
(num1 / num2));
            } else {
                System.out.println("Division by zero is not allowed.");
            }
        });
        addThread.start();
        subtractThread.start();
        multiplyThread.start();
        divideThread.start();
    }
}
```

OUTPUT:

```
Addition: 20 + 5 = 25
Division: 20 / 5 = 4
Multiplication: 20 * 5 = 100
Subtraction: 20 - 5 = 15
```

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

```
package q4;
class MatrixMultiplierThread extends Thread {
    private final int[][] A, B, result;
    private final int row;
    public MatrixMultiplierThread(int[][] A, int[][] B, int[][] result, int row)
{
        this.A = A; this.B = B; this.result = result; this.row = row;
    }
    public void run() {
        int colsB = B[0].length; int colsA = A[0].length;
        for (int j = 0; j < colsB; j++) {
            result[row][j] = 0;
            for (int k = 0; k < colsA; k++) {
                result[row][j] += A[row][k] * B[k][j];
            }
        }
    }
}
public class MatrixMultiplication {
    public static void main(String[] args) {
        int[][] A = {
            {1, 2, 3},
            {4, 5, 6}
        };
        int[][] B = {
            {7, 8},
            {9, 10},
            {11, 12}
        };
        int rowsA = A.length;
        int colsB = B[0].length;
        int[][] result = new int[rowsA][colsB];
        MatrixMultiplierThread[] threads = new MatrixMultiplierThread[rowsA];
        for (int i = 0; i < rowsA; i++) {
            threads[i] = new MatrixMultiplierThread(A, B, result, i);
            threads[i].start();
        }
        for (int i = 0; i < rowsA; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Resultant Matrix:");
        for (int[] row : result) {
            for (int value : row) {
                System.out.print(value + " ");
            }
            System.out.println();
        }
    }
}
```

OUTPUT:

```
Resultant Matrix:
58 64
139 154
```

5. 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.

```
package q5;
class Number {
    private int num = 1; private final int limit;
    public Number(int limit) {
        this.limit = limit;
    }
    public synchronized void printOdd() {
        while (num < limit) {
            while (num % 2 == 0) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Odd: " + num);
            num++;
            notify();
        }
    }
    public synchronized void printEven() {
        while (num <= limit) {
            while (num % 2 != 0) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Even: " + num);
            num++;
            notify();
        }
    }
}
public class OddEven {
    public static void main(String[] args) {
        int limit = 10;
        Number num = new Number(limit);
        Thread oddThread = new Thread(num::printOdd);
        Thread evenThread = new Thread(num::printEven);
        oddThread.start();
        evenThread.start();
    }
}
```

OUTPUT:

```
Odd: 1
Even: 2
Odd: 3
Even: 4
Odd: 5
Even: 6
Odd: 7
Even: 8
Odd: 9
Even: 10
```

6. Implement a Java program that demonstrates thread synchronization using the **synchronized block**.
Create a scenario where multiple threads try to book seats from a limited pool of available seats. Use a synchronized block to ensure that only one thread can access and modify the shared resource at a time, preventing race conditions during seat booking.

```
package q6;
class SeatBookingSystem {
    private int availableSeats;
    public SeatBookingSystem(int totalSeats) {
        this.availableSeats = totalSeats;
    }
    public void bookSeats(String customerName, int seatsRequested) {
        System.out.println(customerName + " is trying to book " + seatsRequested
+ " seat(s).");
        synchronized (this) {
            if (availableSeats >= seatsRequested) {
                availableSeats -= seatsRequested;
                System.out.println(customerName + " successfully booked " +
seatsRequested +
                        " seat(s). Remaining seats: " + availableSeats);
            } else {
                System.out.println("Booking failed for " + customerName +
                        ". Not enough seats available. Remaining seats: " +
availableSeats);
            }
        }
    }
}
public class SeatBookingDemo {
    public static void main(String[] args) {
        SeatBookingSystem bookingSystem = new SeatBookingSystem(5);
        Object[][] customers = {
            {"Cust 1", 2},
            {"Cust 2", 1},
            {"Cust 3", 2},
            {"Cust 4", 1}
        };
        for (Object[] customer : customers) {
            String name = (String) customer[0];
            int seats = (int) customer[1];
            Thread thread = new Thread(() -> bookingSystem.bookSeats(name,
seats), name);
            thread.start();
        }
    }
}
```

OUTPUT:

```
Cust 1 is trying to book 2 seat(s).
Cust 2 is trying to book 1 seat(s).
Cust 3 is trying to book 2 seat(s).
Cust 4 is trying to book 1 seat(s).
Cust 1 successfully booked 2 seat(s). Remaining seats: 3
Cust 4 successfully booked 1 seat(s). Remaining seats: 2
Cust 3 successfully booked 2 seat(s). Remaining seats: 0
Booking failed for Cust 2. Not enough seats available. Remaining seats: 0
```

7. 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.

```
package q7;
import java.util.ArrayList;
import java.util.List;
class PrimeCalculator extends Thread {
    private final int start, end;
    private final List<Integer> primes;
    public PrimeCalculator(int start, int end, List<Integer> primes) {
        this.start = start;
        this.end = end;
        this.primes = primes;
    }
    private boolean isPrime(int num) {
        if (num < 2) return false;
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) return false;
        }
        return true;
    }
    public void run() {
        for (int num = start; num <= end; num++) {
            if (isPrime(num)) {
                synchronized (primes) {
                    primes.add(num);
                }
            }
        }
    }
}
public class PrimeNumber {
    public static void main(String[] args) {
        int limit = 50;
        int numThreads = 4;
        List<Integer> primes = new ArrayList<>();
        Thread[] threads = new Thread[numThreads];
        int range = limit / numThreads;
        for (int i = 0; i < numThreads; i++) {
            int start = i * range + 1;
            int end = (i == numThreads - 1) ? limit : (i + 1) * range;
            threads[i] = new PrimeCalculator(start, end, primes);
            threads[i].start();
        }
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        primes.sort(Integer::compareTo);
        System.out.println("Prime numbers up to " + limit + ": " + primes);
    }
}
```

OUTPUT:

```
Prime numbers up to 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

8. Write a Java program to demonstrate the classic Producer-Consumer problem using multithreading and inter-thread communication. In this program, create a shared buffer class with a fixed capacity to store integer values. Implement synchronized **put()** and **get()** methods in the buffer to manage data insertion and removal. Use **wait()** to pause the producer when the buffer is full and the consumer when the buffer is empty. Use **notify()** to wake up waiting threads when conditions change. The producer thread should generate and insert five integer values into the buffer, while the consumer thread should retrieve and process five items from it. Include **Thread.sleep()** to simulate the time taken to produce and consume items. Ensure that the producer and consumer threads run concurrently and terminate gracefully after completing their respective tasks.

```
package q8;
import java.util.LinkedList;
import java.util.Queue;
class SharedBuffer {
    private final Queue<Integer> buffer = new LinkedList<>();
    private final int capacity;
    public SharedBuffer(int capacity) {
        this.capacity = capacity;
    }
    public synchronized void put(int value) {
        while (buffer.size() == capacity) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        buffer.offer(value);
        System.out.println("Produced: " + value);
        notify();
    }
    public synchronized int get() {
        while (buffer.isEmpty()) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        int value = buffer.poll();
        System.out.println("Consumed: " + value);
        notify(); return value;
    }
}
class Producer extends Thread {
    private final SharedBuffer buffer;
    public Producer(SharedBuffer buffer) {
        this.buffer = buffer;
    }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            buffer.put(i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
class Consumer extends Thread {
    private final SharedBuffer buffer;
    public Consumer(SharedBuffer buffer) {
        this.buffer = buffer;
    }
    public void run() {
        for (int i = 1; i <= 5; i++) {
            buffer.get();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public class ProducerConsumer {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer(3);
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);
        producer.start();
        consumer.start();
        try {
            producer.join();
            consumer.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Producer and Consumer have completed their tasks.");
    }
}
```

OUTPUT:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Produced: 4
Consumed: 3
Produced: 5
Consumed: 4
Consumed: 5
Producer and Consumer have completed their tasks.
```