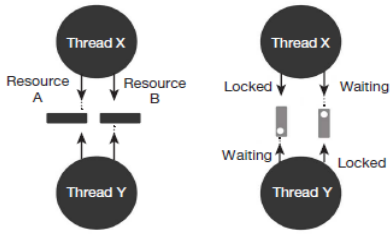
Multithreading

Deadlock:

What?
If two threads are waiting for each other forever.
Why?
Misusing of "synchronized" keyword
Prevent?
Don't use synchronized unnecessarily
Starvation?
High priority thread holds the resource for a long time (may be released in the future).



Example 1:
Thread t1 cannot proceed until it gets resource2, and thread t2 cannot proceed until it gets resource1, both threads are stuck waiting for each other indefinitely, causing a deadlock.

```
public class DeadLockExample {
        public static void main(String[] args) {
          final String resource1 = "First Resource";
          final String resource2 = "Second Resource";
          // t1 tries to lock resource1 then resource2
          Thread t1 = new Thread() {
            public void run() {
                synchronized (resource1) {
                 System.out.println("Thread 1: locked resource 1");

                 try { Thread.sleep(100);} catch (Exception e) {}

                 synchronized (resource2) {
                  System.out.println("Thread 1: locked resource 2");
                 }
               }
            }
          };

          // t2 tries to lock resource2 then resource1
          Thread t2 = new Thread() {
            public void run() {
              synchronized (resource2) {
                System.out.println("Thread 2: locked resource 2");

                try { Thread.sleep(100);} catch (Exception e) {}

                synchronized (resource1) {
                  System.out.println("Thread 2: locked resource 1");
                }
              }
```

```
            }
          };

          t1.start();
          t2.start();
        }
      }
```

Resolve:

If thread t2 holds resource1 and then inside the synchronized block holds resource2, then the deadlock can be removed. In this way, the thread t1 first executes and releases both the resources (resource1 and resource2), and then after thread t2 executes its own block.

Example 2:

The deadlock occurs because both methods depend on each other to call notify() to proceed. However, they both end up calling wait() before either gets a chance to call notify().

```
public class DeadLockInWaitNotify
{
    volatile boolean isNotified = false;

    public synchronized void method1() {
        try
        {
            isNotified = false;
            while (!isNotified)
                wait();
            System.out.println("Method 1");
            isNotified = true;
            notify();
        } catch (InterruptedException e) { }
    }

    public synchronized void method2() {
        try {
            isNotified = true;
            while (isNotified)
                wait();
            System.out.println("Method 2");
            isNotified = false;
            notify();
        } catch (InterruptedException e) { }

    }

    public static void main(String[] args)
    {
      DeadLockInWaitNotify example = new DeadLockInWaitNotify();

        Thread thread1 = new Thread()
        {

            public void run()
            {
```

```
                example.method1();
            }
        };

        Thread thread2 = new Thread()
        {

            public void run()
            {
                example.method2();
            }
        };

        thread1.start();
        thread2.start();
    }
}
```

i. Thread1 waits for isNotified to become true but cannot proceed because thread2 hasn't called notify().

ii. Thread2 waits for isNotified to become false but cannot proceed because thread1 hasn't called notify().

Resolve:

- Initially, both isNotified flags are set to false.
- When method1() executes, it waits until isNotified becomes true, which happens when method2() sets it to true before waiting.
- Similarly, when method2() executes, it waits until isNotified becomes false, which happens when method1() sets it to true after execution.
- This sequential execution ensures that each method waits for the other to finish before proceeding, preventing deadlock.
- The solution effectively resolves the deadlock by ensuring that both methods follow a sequential execution pattern where each method sets isNotified to true before waiting for the other method to set it to a complementary value.

Q1. 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();

    }
}
```

How it Works:

- Initially, isFlag is false, indicating that the even thread should print first.
- The odd thread waits until isFlag becomes true, indicating that it's its turn to print.
- Similarly, the even thread waits until isFlag becomes false.
- After printing, each thread toggles the value of isFlag, notifying the other thread to print.
- This alternation continues until the specified range is reached.
- This program demonstrates how to achieve synchronization between two threads to print even and odd numbers in alternating order. By using a shared flag (isFlag) and synchronized methods

Basic Matrix Multiplication:

```
public class SequentialMatrixMultiplication {
    public static void main(String[] args) {
        int[][] firstMatrix = generateMatrix(3, 3);
        int[][] secondMatrix = generateMatrix(3, 3);
        System.out.println("First Matrix: " + Arrays.deepToString(firstMatrix));
        System.out.println("Second Matrix: " + Arrays.deepToString(secondMatrix));

        int[][] resultMatrix = multiplyMatrix(firstMatrix, secondMatrix);
        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;
    }

    public static int[][] multiplyMatrix(int[][] matrix1, int[][] matrix2) {
        int row1 = matrix1.length;
        int column1 = matrix1[0].length;
        int column2 = matrix2[0].length;
        int[][] result = new int[row1][column2];
        for (int i = 0; i < row1; i++) {
            for (int j = 0; j < column2; j++) {
                for (int k = 0; k < column1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
        return result;
    }
}
```

Q2. 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.

```
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];
            }
        }
    }
}
```

Q3. 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();
    }
}
```

Q4. Write a Java program using Lambdas Expression 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.

```
public class MultiThreadCalculatorLambda {
      public static void main(String[] args) {
        // Define operands
        int a = 10;
        int b = 5;

        // Perform arithmetic operations using lambda expression
        Runnable run1=() -> System.out.println("Addition result: " + (a + b));
        Runnable run2=() -> System.out.println("Subtraction result: " + (a - b));
        Runnable run3=()-> System.out.println("Multiplication result: " + (a * b));
        Runnable run4=() -> {
            if (b != 0) {
                System.out.println("Division result: " + ((double) a / b));
            } else {
                System.out.println("Cannot divide by zero");
            }
        };
        Thread add = new Thread (run1);
        Thread sub = new Thread (run2);
        Thread mul = new Thread (run3);
        Thread div = new Thread (run4);

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