```
catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("JVM will exit");
}
```

Output

Main thread starts working...
Daemon thread is running...
Daemon thread is running...
Daemon thread is running...
JVM will exit now

Dt- 12/05/2025

Q1) (NotifyExample) (ProducerConsumerProblem)

```
class SharedResource {
private boolean available = false;
public synchronized void produce() InterruptedException
  while (available) { wait(); }
System.out.println("Produced an item");
available = true;
notify();
}
public synchronized void consume() throws InterruptedException
  while (!available) { wait(); }
System.out.println("Consumed an item");
available = false;
notify(); }
public class NotifyExample {
```

```
public static void main (String [] args) {
SharedResource resource = new SharedResource();
Thread producer = new Thread (() -> {
  try {
    for (int i=0; i<5; i++) {
      resource.produce(); } }
  catch (InterruptedException e) {
      Thread.currentThread.interrupt(); }
  });
Thread consumer = new Thread (() -> {
  try {
    for (int i=0; i<5; i++) {
      resource.consume(); } }
  catch (InterruptedException e) {
      Thread.currentThread().interrupt();
  }
});
producer.start();
consumer.start();
```

Output

Produced an item

Consumed an item

Produced an item

Consumed an item

Produced an item

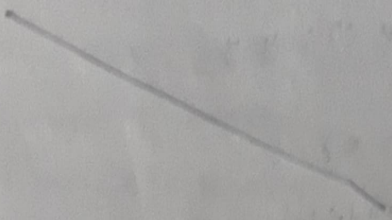
Consumed an item

Produced an item

Consumed an item

Produced an item

Consumed an item.



(3a) (Executor Service)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i=0; i<5; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker); }
        executor.shutdown();
        while (!executor.isTerminated()) { }
        System.out.println("Finished all threads");
    }
}
class WorkerThread implements Runnable {
    private String command;
    public WorkerThread(String s) {
        this.command = s;
    @Override
    public void run() {
```

```
system.out.println (Thread.currentThread().getName() + " Start.
                    Command = " + command);
processCommand();
System.out.println(Thread.currentThread().getName() +
                    " End"); }
private void processCommand() {
  try {
    Thread.sleep(1000); }
  catch (InterruptedException e) { e.printStackTrace();
  }
}
```

Output

```
pool-1-thread-1 Start. command = 0
pool-1 thread-2 Start. Command = 1
pool-1 thread-3 Start. Command = 2
pool-1 thread-1 End
pool-1 thread-3 End
pool-1 thread-2 End
pool-1-thread-1 Start command = 3
pool-1-thread-3 Start. command = 4
pool-1 thread-1 End.
pool-1-thread-3-End
Finished all threads.
```

(Q3) (Synchronizing Code Block)

```
class SychronizedCounter {
  private int count = 0;
  public void increment() {
    synchronized (this) { count++; } }
```

```
Public int getCount () { return count; } }

Public class SyncExample extends Thread {

    Private SynchronizedCounter counter;
    Public SyncExample (SynchronizedCounter counter) {
        this. counter = counter; }
    Public void run () {
        for (int i = 0; i < 1000; i++) {
            counter. increment (); } }
Public static void main (String [] args) throws InterruptedException
    SynchronizedCounter counter = new SynchronizedCounter ();
    Thread t1 = new SyncExample (counter);
    Thread t2 = new SyncExample (counter);
        t1. start ();
        t2. start ();
        t1. join ();
        t2. join ();
    System. out. println ("Final Count:" + counter. getCount ());
}
}
```

Output

Final Count : 2000

(Q4) (Deadlock)

```
Public class Deadlock {
    Public static void main (String [] args) {
        final String resource1 = "Resource 1";
        final String resource2 = "Resource 2";
        Thread t1 = new Thread (() -> {
```

```
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");
    }
}
});

Thread t2 = new Thread(() -> {
    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();
```

Output

Thread 1: locked resource 1
Thread 2: locked resource 2

(ii) (Deadlock General Solution)

```
public class Deadlock {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();
    public void method1() {
        synchronized (lock1) {
            try { Thread.sleep(50); } catch (InterruptedException e)
            { }
            synchronized (lock2) {
```

```
        System.out.println("Method 1");
    }
}

public void method2() {
    synchronized (lock1) {
        try { Thread.sleep(50); } catch (InterruptedException
                                                    e) {
        synchronized (lock2) {
            System.out.println("Method 2"); } } }
public static void main (String[] args) {
    Dead example = new Dead();
    Thread thread1 = new Thread(new Runnable() {
        public void run() { example.method1(); } });
    Thread thread2 = new Thread(new Runnable() {
        public void run() { example.method2(); } });
    thread1.start();
    thread2.start(); } }
```

Output

Method 2

Method 1

(06) (DeadLock using Reentrant Lock)

```
import java.util.concurrent.locks.*;
public class Dead1 {
    private final Lock lock = new ReentrantLock Dead1();
    public void print() {
        lock.lock();
        try {
```

```
System.out.println(Thread.currentThread().getName() + " locked");
Thread.sleep(1000);
catch (InterruptedException e) { e.printStackTrace(); }
finally { lock.unlock();
        System.out.println(Thread.currentThread().getName() +
                           " unlocked."); } }
public static void main(String[] args) {
  Deadl example = new Deadl();
  Runnable task = example::print;
  Thread t1 = new Thread(task);
  Thread t2 = new Thread(task);
  t1.start();
  t2.start(); } }
```

Output

```
Thread-0 locked
Thread-1 locked
Thread-0 unlocked
Thread-1 unlocked.
```

8) (Starvation)

```
import java.util.concurrent.locks.*;
public class StarV {
private final Lock lock = new ReentrantLock(true);
public void accessResource() {
    lock.lock();
  try {
    System.out.println(Thread.currentThread().getName() +
                       " accessed resource.");
    Thread.sleep(100); }
```

```
catch (InterruptedException e) { e.printStackTrace(); }
finally { lock.unlock(); } }
public static void main (String[] args) {
    Starv example = new Starv();
    Runnable task = example::accessResource;
    Thread t1 = new Thread (task);
    Thread t2 = new Thread (task);
    Thread t3 = new Thread (task);
    Thread t4 = new Thread (task);
    t1.setPriority (Thread.MAX_PRIORITY);
    t2.setPriority (Thread.MIN_PRIORITY);
    t3.setPriority (Thread.MIN_PRIORITY);
    t4.setPriority (Thread.MIN_PRIORITY);
    t1.start();
    t2.start();
    t3.start();
    t4.start(); } }
```

Output

```
Thread-0 accessed resource
Thread-1 accessed resource
Thread-2 accessed resource
Thread-3 accessed resource
```

Dt- 13/05/2025

(Q1) (RunnableExample)

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
public class RunnableExample implements Runnable {
    private final long counter;
    RunnableExample (long counter) {
```

```
this.counter = counter; }
@Override
public void run () {
    long total = 0;
    for (long i = 1; i < counter; i++) { total += i; }
    System.out.println (total); }
public static void main (String [] args) {
    Executor executor = Executors.newSingleThreadExecutor ();
    executor.execute (new RunnableExample (1000));
}
}
```

Output

```
499500
```

(2) (Matrix Multiplication using Sequential)

```
import java.util.Arrays;
import java.util.Random;
public class Matrix {
    public static void main (String [] args) {
        int [][] firstMatrix = generateMatrix (3,3);
        int [][] secondMatrix = generateMatrix (3,3);
        System.out.println (Arrays.deepToString (multiplyMatrix (firstMatrix,
                                                  secondMatrix)
    }

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 [][] matrixMultiplication (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++) {
            result[i][j] = 0;
            for (int k=0; k<column1; k++) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
    return result;
}
```

Output

```
[[81, 87, 126], [20, 36, 56], [40, 56, 61]]
```

(Q3) (Matrix Multiplier with Time example)

```
import java.util.Arrays;
import java.util.Date;
import java.util.Random;
public class Matrix {
    public static void main (String [] args) {
        int [][] firstMatrix = generateMatrix (3,3);
        int [][] secondMatrix = generateMatrix (3,3);
        Date start = new Date();
```

```
System.out.println (Arrays.deepToString (multiplyMatrix (
                                firstMatrix,
                                secondMatrix)));

Date end = new Date ();

System.out.println ("Total time taken : " + (end.getTime () - start.getTime ()) + " milliseconds");
}

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 [][] mautiplyMatrix [int [][] (int [][]
                                matrix2) {

int row1 = matrix1.length;
int column1 = matrix1 [0].length
int column2 = matrix2 [0].length;
int [][] result = new int [row] [column1].

for (int i = 0 ; j < row1 ; i++) {
for (int j = 0; j < column2 ; j++) {

    result [i] [j] = 0;

    for (int k = 0; k < column1 ; k++) {
result [i] [j] += matrix1 [i] [k] * Matrix2 [k] [j];
```

```
}}};
return result; } }
```

Output

```
[[81, 87, 126], [20, 36, 56], [40, 56, 61]]
```

Total Time Taken : 1 millisecond

(Matrix Parallel Multiplier Example)

```
import java.util.Arrays;
Public class Matrix3 {
public static void main (String[] args) {
int [][] firstMatrix = generateMatrix(3,3);
int [][] secondMatrix = generateMatrix(3,3);
int [][] result = new int[3][3];
Thread [] threads = new Thread[3];
for (int i=0; i<3; i++) {
final int row = i;
threads[i] = new Thread(() -> {
for (int j=0; j<3; j++) {
for (int k=0; k<3; k++) {
result[row][j] += firstMatrix[row][k] * secondMatrix
[k][j];
}
}});
threads[i].start(); }
for (Thread thread : threads) {
try { thread.join(); }
catch (InterruptedException e) { e.printStackTrace(); }
```

```
System.out.println(Arrays.deepToString(result));
}
public static int[][] generateMatrix(int rows, int columns){
    int[][] matrix = new int[rows][columns];
    for(int i=0; i<rows; i++){
        for(int j=0; j<rows; j++){
            matrix[i][j] = (int)(Math.random()*10);
        }
    }
    return matrix;
}
```

Output

[[5, 76, 61], [19, 52, 33], [86, 100, 92]]

04-14/05/2025

Implementing reactive programming using Eclipse.

Step 1: Open ~~eclipse~~ ~~IDE~~ eclipse IDE (Integrated development environment).

Step 2: File → New → Maven Project

Step 3: create a simple project.

Step 4: fill up group id, artifact ID & name.

Step 5: Modify Pom.xml add dependencies.

For RxJava (version 3):

```
<dependencies>
<dependency>
<groupId> io.reactivex.rxjava3 </groupId>
<artifactId> rxjava </artifactId>
<version> 3.1.8 </version>
</dependency>
</dependencies>
```

For Project Reactor (Core)

```
<dependencies>
<dependency>
<groupId> io.projectreactor </groupId>
<artifactId> reactor-core </artifactId>
<version> 3.6.2 </version>
</dependency>
</dependencies>
```

Step 6: Then Save it.

Step 7: Write the Java Programming code for RxJava & Reactor in Eclipse.

Create a class RxJavaExample.java:

```
import io.reactivex.rxjava2.core.Observable;
public class RxJavaExample {
public static void main (String [] args) {
Observable <String> observable = Observable.just ("Book",
                                 "Lab Manual", "Lab
                                 Practice Copy")
observable.subscribe (
item -> System.out.println ("I came with:" + item),
Throwable::printStackTrace,
() -> System.out.println ("I am ready for the
                          class")
);
} }
```

Create Reactor Example Program.

```
import reactor.core.publisher.Flux;
public class ReactorExample {
```

public static void main (String [] args) {

Flux <String> bookflux = Flux just ("Book", "Lab Manual",
"Lab Pratice copy");

bookflux.subscribe (

item → System.out.println (" I have" + item);

error → System.err.println ("Error" + error);

() → System.out.println ("Attended the class");

); }}

Q) (Java Program on Reactive ATM System)

import java.util.concurrent.Flow;

import java.util.concurrent.SubmissionPublisher;

class BankPublisher extends SubmissionPublisher <Transcation> {

private double accountBalance = 1000.0;

~~public static void main (String [] args) {~~

public void processRequest (Transcation transaction) {

if (transcation.getAmount() > accountBalance) {

transcation.setStatus ("Failed : Insufficient Balance");

} else {

accountBalance -= transaction.getAmount();

transcation.setStatus ("Succes : withdrawn" + transcation.
getAmount());

}

this.submit (transcation); }

public double getAccountBalance () { return accountBalance;

} }

class Transaction {

private double amount;

Private String status;

```
Public Transaction (double amount) {
        this.amount = amount;
        this.status = "pending"; }
Public double getAmount() { return Amount; }
public String getStatus() { return status; }
public void setStatus (String status) { this.status = status;
@ override
Public String toString() {
return "Transaction of Rs." + amount + " - " + status; };
Class ATMSubscriber implements Flow.Subscriber<Transaction> {
    private Flow.Subscription subscription;
    private String atmId;
    public ATMSubscriber (String atmId) { this.atmId = atmId;
@ override
Public void onSubscribe (Flow.Subscription subscription) {
    this.subscription = subscription;
    System.out.println ("ATM" + atmId + "subscribed");
subscription.request (1); }
@ override
public void onError (Throwable throwable) {
System.out.println ("ATM" + atmId + "encountered error" +
                    throwable.getMessage());
}
@ override
Public void onComplete () {
    System.out.println ("ATM" + atmId + "transaction
                                    stream completed")
```

```
public class ReactiveATM {
public static void main (String[] args) throws InterruptedException {
BankPublisher bank = new BankPublisher();
ATMSubscriber atm1 = new ATMSubscriber("ATM-101");
ATMSubscriber atm2 = new ATMSubscriber("ATM-102");
bank.subscribe(atm1);
bank.subscribe(atm2);
bank.processRequest(new Transcation(2000));
bank.processRequest(new Transcation(5000));
bank.processRequest(new Transcation(4000));
Thread.sleep(1000);
System.out.println("Final Balance in Bank: Rs." + bank.
getAccountBalance());
bank.close(); }}
```

Output

ATM-102 subscribed

ATM-101 subscribed

ATM-102 received: Transcation { Amount = 2000.0 status = "success" }

ATM-101 received: Transcation { Amount = 2000.0, Status = 'success' }

ATM-102 recived: Transcation { Amount = 5000.0, Status = 'failed. Insufficient Balance' }

ATM-101 received: Transcation { Amount = 5000.0, Status = 'Failed: Insufficient Balance' }

ATM-102 received: Transcation { Amount = 4000.0, Status = 'failed: Insufficient Balance' }

ATM-101 received : Transcation { Amount = 4000.0, Status = 'failed. Insufficient Balance' }

Final bank balance : 3400.0

ATM-101 transcation stream complete

ATM-102 transcation stream complete