

## Day 18: Manisha assignment

## Task 1: Creating and Managing Threads

**Write a program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number**

## **Explanation of Output:**

- The output is an 8x8 board where 'Q' represents the position of queens. The queens are placed such that no two queens can attack each other.

This code can be run in Eclipse or any other Java development environment. If you encounter any errors, please provide the error details so I can help resolve them.

Here is a simple Java program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number.

## **Explanation of Output:**

- Both threads print numbers from 1 to 10, with a 1-second delay between each number.
  - The output shows the numbers printed by each thread. Note that the exact order may vary slightly due to the concurrent execution of threads, but each thread will print its sequence independently



The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows a project named "TwoThreads".
- Java Editor:** Displays the code for `NumberPrinter` and `TwoThreadsExample`.
- Outline View:** Shows the class hierarchy and methods.
- Search View:** Shows search results for "ThreadName".
- Help View:** Shows help information for "ThreadName".

```
1 package TwoThreads;
2
3 public class NumberPrinter extends Thread {
4     private String threadname;
5
6     NumberPrinter(String name) {
7         threadname = name;
8     }
9
10    public void run() {
11        try {
12            for (int i = 1; i <= 10; i++) {
13                System.out.println(threadname + ": " + i);
14                Thread.sleep(1000); // Sleep for 1 second
15            }
16        } catch (InterruptedException e) {
17            System.out.println(threadname + " interrupted.");
18        }
19        System.out.println(threadname + " exiting.");
20    }
21}
22
23 public class TwoThreadsExample {
24     public static void main(String[] args) {
25         NumberPrinter thread1 = new NumberPrinter("Thread 1");
26         NumberPrinter thread2 = new NumberPrinter("Thread 2");
27     }
28 }
```

## Task 2: States and Transitions

Create a Java class that simulates a thread going through different lifecycle states: NEW, RUNNABLE, WAITING, TIMED\_WAITING, BLOCKED, and TERMINATED. Use methods like sleep(), wait(), notify(), and join() to demonstrate these states..

### Explanation:

#### 1. ThreadLifecycleDemo Class:

- Extends the `Thread` class.
- Uses a shared `lock` object to demonstrate synchronization and waiting.
  - In the `run` method:
    - Initially prints the `RUNNABLE` state.
    - Sleeps for 1 second to move to the `TIMED\_WAITING` state.
    - Enters a synchronized block and calls `wait()` on the `lock` to move to the `WAITING` state.
    - Once notified, prints `RUNNABLE` again and sleeps for another second.
    - Finally, prints `TERMINATED` state when the `run` method completes.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package lifecycledemo;

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

public class ThreadLifecycleDemo extends Thread {
    private final Object lock;

    public ThreadLifecycleDemo(Lock lock) {
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            // Simulate RUNNABLE state
            System.out.println(Thread.currentThread().getName() + " is RUNNABLE");
            Thread.sleep(1000); // Moves to TIMED_WAITING state
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " interrupted");
        }
        synchronized (lock) {
            // Simulate WAITING state
            System.out.println(Thread.currentThread().getName() + " is WAITING");
            lock.wait(); // Moves to TIMED_WAITING state
        }
        // Simulate RUNNABLE state again after being notified
        System.out.println(Thread.currentThread().getName() + " is RUNNABLE again");
        Thread.sleep(1000); // Moves to TIMED_WAITING state
    }
}

class ThreadLifecycleStates {
    public static void main(String[] args) {
        Object lock = new Object();
        ThreadLifecycleDemo t1 = new ThreadLifecycleDemo(lock);
        ThreadLifecycleDemo t2 = new ThreadLifecycleDemo(lock);

        // Simulate NEW state
        System.out.println(t1.getName() + " is NEW");

        // Start thread t1, moves to RUNNABLE state
        t1.start();

        try {
            Thread.sleep(1000); // Ensure t1 is in WAITING state before notifying
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (lock) {
            lock.notify();
        }

        try {
            // Ensure t1 completes
            t1.join();
            // Ensure t2 completes
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

#### 2. ThreadLifecycleStates Class:

- Creates a shared `lock` object and two instances of `ThreadLifecycleDemo`.
  - Prints `NEW` state for thread `t1`.
  - Starts thread `t1`, which moves it to the `RUNNABLE` state.
    - Main thread acquires the `lock` and sleeps for 2 seconds to simulate `BLOCKED` state for `t2`.
    - Starts thread `t2`, which attempts to acquire the lock and gets `BLOCKED`.
    - After 1 second, main thread notifies `t1`, moving it from `WAITING` to `RUNNABLE` state.
    - Calls `join` on both threads to ensure they complete before main thread terminates.

## Task 3: Synchronization and Inter-thread Communication

Implement a producer-consumer problem using `wait()` and `notify()` methods to handle the correct processing sequence between threads.

### Explanation:

#### 1. SharedBuffer Class:

- This class represents the shared buffer where the producer will produce items and the consumer will consume items.

- It has an integer array `buffer`, an integer `count` to track the number of items, and an integer `size` to define the buffer size.

- The `produce` method:

- Synchronized to ensure only one thread can produce at a time.

- Uses `wait()` if the buffer is full, preventing overproduction.

- Adds a produced item to the buffer and increments the count.

- Calls `notify()` to wake up any waiting consumer.

```
1 package com.intellectualmachines.intertreadcommunications;
2
3 public class SharedBuffer {
4     private int[] buffer;
5     private int count;
6     private int size;
7
8     public SharedBuffer(int size) {
9         this.size = size;
10    this.buffer = new int[size];
11    this.count = 0;
12 }
13
14 public synchronized void produce(int value) throws InterruptedException {
15     while (count == size) {
16         wait(); // Wait if the buffer is full
17     }
18     buffer[count] = value;
19     System.out.println("Produced: " + value);
20     count++; // Notify the consumer that there is data to consume
21 }
22
23 public synchronized void consume() throws InterruptedException {
24     while (count == 0) {
25         wait(); // Wait if the buffer is empty
26     }
27     int value = buffer[0];
28     System.out.println("Consumed: " + value);
29     count--;
30     notify(); // Notify the producer that there is space to produce
31 }
32
33 class Producer extends Thread {
34     SharedBuffer buffer;
35
36     public Producer(SharedBuffer buffer) {
37         this.buffer = buffer;
38     }
39
40     @Override
41     public void run() {
42         for (int i = 0; i < 10; i++) {
43             try {
44                 produce(i);
45                 Thread.sleep(100); // Simulate time taken to produce an item
46             } catch (InterruptedException e) {
47                 e.printStackTrace();
48             }
49         }
50     }
51 }
52
53 class Consumer extends Thread {
54     SharedBuffer buffer;
55
56     public Consumer(SharedBuffer buffer) {
57         this.buffer = buffer;
58     }
59
60     @Override
61     public void run() {
62         for (int i = 0; i < 10; i++) {
63             try {
64                 buffer.consume();
65                 Thread.sleep(150); // Simulate time taken to consume an item
66             } catch (InterruptedException e) {
67                 e.printStackTrace();
68             }
69         }
70     }
71 }
72
73 public class ProducerConsumerDemo {
74     public static void main(String[] args) {
75         SharedBuffer buffer = new SharedBuffer(5);
76         Producer producer = new Producer(buffer);
77         Consumer consumer = new Consumer(buffer);
78         producer.start();
79         consumer.start();
80     }
81 }
82
83 }
```

```
1 package com.intellectualmachines.intertreadcommunications;
2
3 public class Producer extends Thread {
4     SharedBuffer buffer;
5
6     public Producer(SharedBuffer buffer) {
7         this.buffer = buffer;
8     }
9
10    @Override
11    public void run() {
12        for (int i = 0; i < 10; i++) {
13            try {
14                buffer.produce(i);
15                Thread.sleep(100); // Simulate time taken to produce an item
16            } catch (InterruptedException e) {
17                e.printStackTrace();
18            }
19        }
20    }
21 }
22
23 }
```

```
1 package com.intellectualmachines.intertreadcommunications;
2
3 public class Consumer extends Thread {
4     SharedBuffer buffer;
5
6     public Consumer(SharedBuffer buffer) {
7         this.buffer = buffer;
8     }
9
10    @Override
11    public void run() {
12        for (int i = 0; i < 10; i++) {
13            try {
14                buffer.consume();
15                Thread.sleep(150); // Simulate time taken to consume an item
16            } catch (InterruptedException e) {
17                e.printStackTrace();
18            }
19        }
20    }
21 }
22
23 }
```

```
1 package com.intellectualmachines.intertreadcommunications;
2
3 public class ProducerConsumerDemo {
4     public static void main(String[] args) {
5         SharedBuffer buffer = new SharedBuffer(5);
6         Producer producer = new Producer(buffer);
7         Consumer consumer = new Consumer(buffer);
8         producer.start();
9         consumer.start();
10    }
11 }
12
13 }
```

#### 2. Producer Class:

- Extends the `Thread` class.

- In the `run` method, it produces 10 items, calling the `produce` method on the shared buffer and sleeping for 100 milliseconds between productions to simulate production time.

#### 3. Consumer Class:

- Extends the `Thread` class.

- In the `run` method, it consumes 10 items, calling the `consume` method on the shared buffer and sleeping for 150 milliseconds between consumptions to simulate consumption time.

#### 4. ProducerConsumerDemo Class:

- Contains the `main` method.

- Creates a shared buffer with a size of 5.

- Creates and starts one producer and one consumer thread.

## Task 4: Synchronized Blocks and Methods

Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

### Explanation:

#### 1. BankAccount Class:

- The `BankAccount` class represents a bank account with a balance.
- The `deposit` method is synchronized to prevent multiple threads from updating the balance concurrently.
- The `withdraw` method is synchronized to ensure atomic withdrawal operations and to prevent race conditions.

#### 2. DepositThread Class:

- This class extends the `Thread` class and is used to perform deposit operations on the `BankAccount`.
- The `run` method calls the `deposit` method on the `BankAccount`.

#### 3. WithdrawThread Class:

- This class extends the `Thread` class and is used to perform withdrawal operations on the `BankAccount`.
- The `run` method calls the `withdraw` method on the `BankAccount`.

#### 4. BankAccountSimulation Class:

- The `main` method creates a `BankAccount` object with an initial balance of 1000.
- It creates multiple `DepositThread` and `WithdrawThread` objects to perform deposit and withdrawal operations.
- It starts all the threads and waits for them to complete using the `join` method.
- Finally, it prints the final balance of the account.

The image contains three side-by-side screenshots of the Eclipse Java IDE interface, showing the code for three classes: BankAccount, DepositThread, and BankAccountSimulation.

- BankAccount.java:** Contains the definition of the BankAccount class. It has a private double balance field and two synchronized methods: deposit and withdraw. The deposit method adds the amount to the balance and prints the current balance. The withdraw method checks if the balance is greater than or equal to the amount, subtracts it, and prints the current balance.
- DepositThread.java:** Extends the Thread class. Its run method creates a BankAccount object and calls the deposit method on it, passing the account and the amount to be deposited.
- BankAccountSimulation.java:** Contains the main method. It creates a BankAccount object with an initial balance of 1000. It then creates four DepositThreads and four WithdrawThreads, starting each thread and waiting for them to complete using the join method. Finally, it prints the final balance of the account.

Each screenshot shows the code editor with syntax highlighting, the package explorer, and the output console at the bottom.

## Task 5: Thread Pools and Concurrency Utilities

Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

### Explanation:

#### 1. CalculationTask Class:

- This class implements the `Runnable` interface and simulates a task that performs a complex calculation or I/O operation.
- The `run` method prints the start of the task, simulates a time-consuming task using `TimeUnit.SECONDS.sleep(2)` , and performs a simple calculation (square of the task ID).

- The result of the calculation is printed, and any interruption is handled by catching `InterruptedException` .

The code for the CalculationTask class is as follows:

```
1 package com.example.concurrent.executor;
2
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executors;
5 import java.util.concurrent.TimeUnit;
6
7 public class CalculationTask implements Runnable {
8     private final int taskId;
9
10    public CalculationTask(int taskId) {
11        this.taskId = taskId;
12    }
13
14    @Override
15    public void run() {
16        System.out.println("Task " + taskId + " is starting.");
17        try {
18            TimeUnit.SECONDS.sleep(2); // Simulate time-consuming task
19            int result = taskId * taskId; // Simple calculation (square of taskID)
20            System.out.println("Task " + taskId + " completed with result: " + result);
21        } catch (InterruptedException e) {
22            System.out.println("Task " + taskId + " was interrupted.");
23        }
24    }
25}
```

The code for the ThreadPoolExample class is as follows:

```
1 package com.example.concurrent.executor;
2
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executors;
5 import java.util.concurrent.TimeUnit;
6
7 public class ThreadPoolExample {
8
9    public static void main(String[] args) {
10        ExecutorService executorService = Executors.newFixedThreadPool(3);
11
12        // Submit multiple tasks to the thread pool
13        for (int i = 1; i <= 10; i++) {
14            executorService.submit(new CalculationTask(i));
15        }
16
17        // Shutdown the executor service
18        executorService.shutdown();
19
20        try {
21            // Wait for all tasks to complete
22            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
23                executorService.shutdownNow(); // Force shutdown if tasks are not completed in 60 seconds
24            }
25        } catch (InterruptedException e) {
26            executorService.shutdownNow(); // Force shutdown if interrupted
27        }
28
29        System.out.println("All tasks completed.");
30    }
31}
```

The output of the application shows the tasks starting and completing in the order managed by the thread pool:

```
Task 1 is starting.
Task 2 is starting.
Task 3 is starting.
Task 4 is starting.
Task 5 is starting.
Task 6 is starting.
Task 7 is starting.
Task 8 is starting.
Task 9 is starting.
Task 10 is starting.
Task 1 completed with result: 1
Task 2 completed with result: 4
Task 3 completed with result: 9
Task 4 completed with result: 16
Task 5 completed with result: 25
Task 6 completed with result: 36
Task 7 completed with result: 49
Task 8 completed with result: 64
Task 9 completed with result: 81
Task 10 completed with result: 100
All tasks completed.
```

#### 2. ThreadPoolExample Class:

- The `main` method creates a fixed-size thread pool with 3 threads using `Executors.newFixedThreadPool(3)` .

- It submits 10 tasks to the thread pool using a for loop and `executorService.submit(new CalculationTask(i))` .

- The `shutdown` method is called to stop accepting new tasks.

- The `awaitTermination` method waits for all tasks to complete within 60 seconds. If tasks are not completed, it forcefully shuts down the executor service using `shutdownNow` .

### Explanation of Output:

- The output shows tasks starting and completing in the order managed by the thread pool.
- Each task prints its start, performs a calculation, and then prints the result.
- The thread pool allows multiple tasks to run concurrently up to the fixed size (3 in this case), ensuring efficient utilization of threads.
- Finally, a message "All tasks completed." confirms that all tasks have finished executing.

## Task 6: Executors, Concurrent Collections, CompletableFuture

Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

## **Explanation:**

## 1. PrimeCalculator Class:

- `isPrime(int number)`: Checks if a given number is prime.

## 2. PrimeNumberFinder Class:

- Sets up an `ExecutorService` with a fixed thread pool of size 4.

- Uses a list of `CompletableFuture<Integer>` to store the results of prime number calculations.

- For each number from 2 to the upper limit (100 in this case), it creates a `CompletableFuture` that checks if the number is prime using `PrimeCalculator`.

- ``CompletableFuture.allOf(futureList.toArray(new CompletableFuture[0]))``: Waits for all futures to complete.

- Once all tasks are completed, it collects the prime numbers and writes them to a file asynchronously.

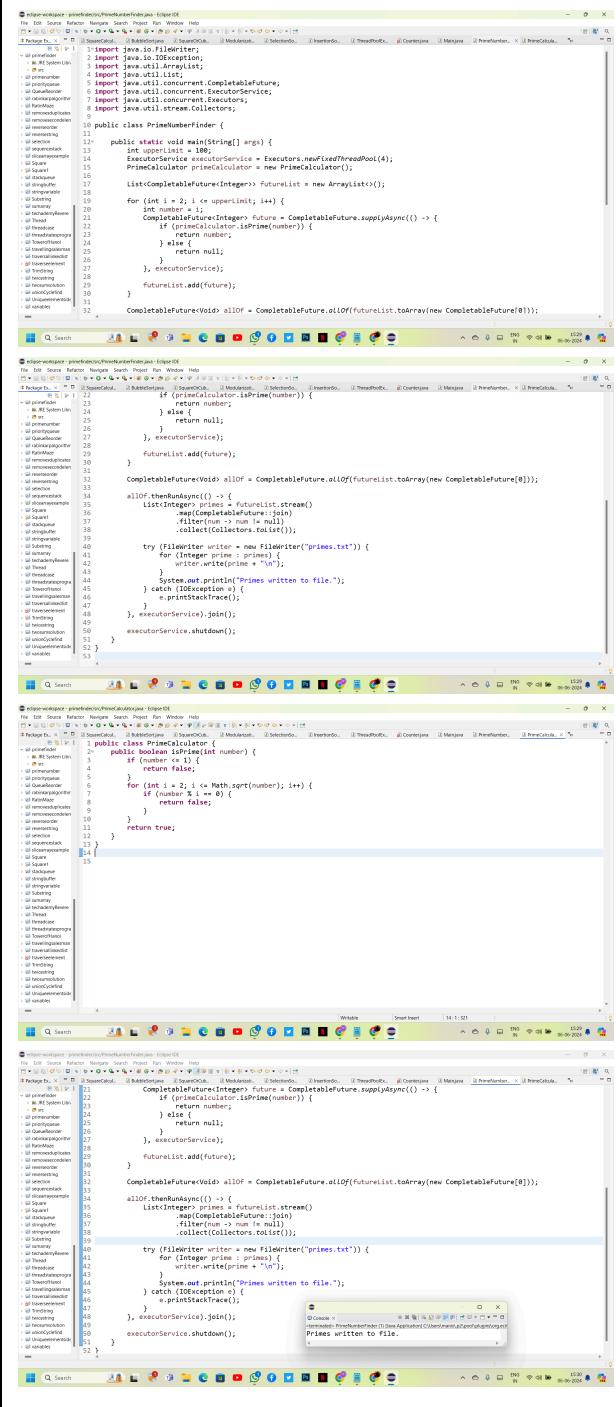
- Finally, the executor service is shut down.

## **1. Ensure Proper Imports:**

- Ensure you have imported the necessary classes like `FileWriter`, `IOException`, `List`, `ArrayList`, `CompletableFuture`, `ExecutorService`, `Executors`, `Collectors`, etc.

## **2. File Location:**

- The `primes.txt` file will be created in the project's root directory. Ensure you have write permissions in this directory.



## Task 7: Writing Thread-Safe Code, Immutable Objects

Design a thread-safe Counter class with increment and decrement methods. Then demonstrate its usage from multiple threads. Also, implement and use an immutable class to share data between threads.

### Explanation:

#### 1. Counter Class:

- **increment()** Method: Increments the count. This method is synchronized to ensure thread safety.
- **decrement()** Method: Decrements the count. This method is synchronized to ensure thread safety.

- **getCount()** Method: Returns the current count. This method is synchronized to ensure thread safety.

#### 2. SharedData Class:

- A final class with a single immutable field `data`.
- **Constructor:** Initializes the `data` field.
- **getData() Method\*\*:** Returns the value of `data`.

#### 3. Main Class:

- Creates an instance of `Counter` and `SharedData`.
- Creates three threads, each running a `CounterTask`.
- Starts the threads and waits for them to finish using `join()`.
- Prints the final count and the shared data.

#### 4. CounterTask Class:

- Implements `Runnable`.
- Takes a `Counter` instance as a parameter.
- In the `run` method, increments and decrements the counter to simulate some work.

The image shows four Java code editors side-by-side, each displaying a different class definition. The classes are:

- Counter.java**: A class with private integer fields and synchronized methods for incrementing, decrementing, and getting the count.
- SharedData.java**: A final class with a private immutable integer field and a public getter method.
- Main.java**: The main entry point. It creates a Counter and SharedData instance, starts three threads (t1, t2, t3) with CounterTask instances, and waits for them to finish using join(). It then prints the final count and shared data.
- CounterTask.java**: An implementation of Runnable that runs a loop to increment and decrement a Counter instance.