Manjula Nannuri

Day 19_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

```
public class Main {
    public static void main(String[] args) {
        Runnable task1 = new PrintNumbersTask("Thread 1");
        Runnable task2 = new PrintNumbersTask("Thread 2");
        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);
        thread1.start();
        thread2.start();
    }
}
class PrintNumbersTask implements Runnable {
    private String threadName;
   public PrintNumbersTask(String threadName) {
        this.threadName = threadName;
    }
    public void run() {
```

```
        for (int i = 1; i <= 10; i++) {
            System.out.println(threadName + ": " + i);
            try {
                Thread.sleep(1000); // 1-second delay
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Output:

```
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
```

Thread 2: 7

Thread 1: 8

Thread 2: 8

Thread 1: 9

Thread 2: 9

Thread 1: 10

Thread 2: 10

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..

```
class ThreadLifecycleDemo {
    public static void main(String[] args) {
        Object lock = new Object();
        Thread thread = new Thread(new LifecycleTask(lock));
        System.out.println("State after creating thread (NEW): " + thread.getState());
        thread.start();
        System.out.println("State after starting thread (RUNNABLE): " + thread.getState());
        try {
            Thread.sleep(100);
            synchronized (lock) {
                lock.notify();
            }
            Thread.sleep(100);
```

```
            System.out.println("State after notifying (TIMED_WAITING or WAITING): " +
thread.getState());
            Thread.sleep(5000);
            System.out.println("State after completion (TERMINATED): " + thread.getState());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class LifecycleTask implements Runnable {
    private final Object lock;
   public LifecycleTask(Object lock) {
        this.lock = lock;
    }
    public void run() {
        try {
            System.out.println("Thread is RUNNABLE now.");
            synchronized (lock) {
                System.out.println("Thread is WAITING.");
                lock.wait();
            }
            System.out.println("Thread is TIMED_WAITING (sleeping).");
            Thread.sleep(2000);
```

```
            System.out.println("Thread is RUNNABLE after sleep.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (lock) {
            System.out.println("Thread is BLOCKED (attempting to re-acquire lock).");
        }

        System.out.println("Thread execution finished (TERMINATED).");
    }
}
```

Output :

```
State after creating thread (NEW): NEW
Thread is RUNNABLE now.
Thread is WAITING.
State after starting thread (RUNNABLE): RUNNABLE
Thread is TIMED_WAITING (sleeping).
State after notifying (TIMED_WAITING or WAITING): TIMED_WAITING
Thread is RUNNABLE after sleep.
Thread is BLOCKED (attempting to re-acquire lock).
Thread execution finished (TERMINATED).
State after completion (TERMINATED): TERMINATED
```

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.

```
import java.util.LinkedList;

import java.util.Queue;


class ProducerConsumerDemo {

    public static void main(String[] args) {

        SharedQueue sharedQueue = new SharedQueue(5);


        Thread producerThread = new Thread(new Producer(sharedQueue));

        Thread consumerThread = new Thread(new Consumer(sharedQueue));


        producerThread.start();

        consumerThread.start();

    }

}


class SharedQueue {

    private final Queue<Integer> queue;

    private final int capacity;


    public SharedQueue(int capacity) {

        this.queue = new LinkedList<>();

        this.capacity = capacity;
```

```
    }

    public void produce(int value) throws InterruptedException {
        synchronized (this) {
            while (queue.size() == capacity) {
                wait();
            }
            queue.add(value);
            System.out.println("Produced: " + value);
            notify();
        }
    }

    public void consume() throws InterruptedException {
        synchronized (this) {
            while (queue.isEmpty()) {
                wait();
            }
            int value = queue.poll();
            System.out.println("Consumed: " + value);
            notify();
        }
    }
}
```

```
class Producer implements Runnable {
    private final SharedQueue sharedQueue;
    public Producer(SharedQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
    public void run() {
        int value = 0;
        while (true) {
            try {
                sharedQueue.produce(value++);
                Thread.sleep(500);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
    }
}

class Consumer implements Runnable {
    private final SharedQueue sharedQueue;
    public Consumer(SharedQueue(SharedQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
```

```
    }
    public void run() {
        while (true) {
            try {
                sharedQueue.consume();
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }
    }
}
```

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.

```
class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public synchronized void deposit(double amount) {
        if (amount > 0) {
```

```
            balance += amount;
            System.out.println(Thread.currentThread().getName() + " deposited: " + amount + ", New balance: " + balance);
        }
    }
    public synchronized void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew: " + amount + ", New balance: " + balance);
        } else {
            System.out.println(Thread.currentThread().getName() + " attempted to withdraw: " + amount + " but insufficient funds.");
        }
    }
    public synchronized double getBalance() {
        return balance;
    }
}

class DepositTask implements Runnable {
    private final BankAccount account;
    private final double amount;
    public DepositTask(BankAccount account, double amount) {
        this.account = account;
```

```
        this.amount = amount;
    }
    public void run() {
        account.deposit(amount);
    }
}
class WithdrawTask implements Runnable {
    private final BankAccount account;
    private final double amount;
    public WithdrawTask(BankAccount account, double amount) {
        this.account = account;
        this.amount = amount;
    }
    public void run() {
        account.withdraw(amount);
    }
}
public class BankAccountSimulation {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.0);
        Thread t1 = new Thread(new DepositTask(account, 200), "Thread 1");
        Thread t2 = new Thread(new WithdrawTask(account, 150), "Thread 2");
        Thread t3 = new Thread(new DepositTask(account, 300), "Thread 3");
        Thread t4 = new Thread(new WithdrawTask(account, 500), "Thread 4");
```

```
        Thread t5 = new Thread(new WithdrawTask(account, 1000), "Thread 5");
        t1.start();
        t2.start();
        t3.start();
        t4.start();
        t5.start();
        try {
            t1.join();
            t2.join();
            t3.join();
            t4.join();
            t5.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final balance: " + account.getBalance());
    }
}
```

Output :

```
Thread 1 deposited: 200.0, New balance: 1200.0
Thread 5 withdrew: 1000.0, New balance: 200.0
Thread 3 deposited: 300.0, New balance: 500.0
Thread 4 withdrew: 500.0, New balance: 0.0
Thread 2 attempted to withdraw: 150.0 but insufficient funds.
```

Final balance: 0.0

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.

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

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

class CalculationTask implements Runnable {

   private final int taskId;

   public CalculationTask(int taskId) {

      this.taskId = taskId;

   }

   public void run() {

      System.out.println("Task " + taskId + " started by " + Thread.currentThread().getName());

      performComplexCalculation();

      System.out.println("Task " + taskId + " completed by " +
Thread.currentThread().getName());

   }

   private void performComplexCalculation() {

      try {

         Thread.sleep(2000); // Simulate time taken to perform complex calculation

      } catch (InterruptedException e) {

         Thread.currentThread().interrupt();

      }

   }
```

```
}

class IOTask implements Runnable {

    private final int taskId;

    public IOTask(int taskId) {

        this.taskId = taskId;

    }

    public void run() {

        System.out.println("I/O Task " + taskId + " started by " +
Thread.currentThread().getName());

        performIOOperation();

        System.out.println("I/O Task " + taskId + " completed by " +
Thread.currentThread().getName());

    }

    private void performIOOperation() {

        try {

            Thread.sleep(3000); // Simulate time taken to perform I/O operation

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

}

public class ThreadPoolExample {

    public static void main(String[] args) {

        int numberOfTasks = 10;

        ExecutorService executor = Executors.newFixedThreadPool(3);
```

```
        for (int i = 0; i < numberOfTasks; i++) {
            executor.submit(new CalculationTask(i));
        }
        for (int i = 0; i < numberOfTasks; i++) {
            executor.submit(new IOTask(i));
        }
        executor.shutdown();
        try {
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
            Thread.currentThread().interrupt();
        }
        System.out.println("All tasks completed.");
    }
}
```

Output:

```
Task 1 started by pool-1-thread-2
Task 0 started by pool-1-thread-1
Task 2 started by pool-1-thread-3
Task 0 completed by pool-1-thread-1
Task 1 completed by pool-1-thread-2
```

Task 3 started by pool-1-thread-2

Task 4 started by pool-1-thread-1

Task 2 completed by pool-1-thread-3

Task 5 started by pool-1-thread-3

Task 3 completed by pool-1-thread-2

Task 4 completed by pool-1-thread-1

Task 6 started by pool-1-thread-2

Task 7 started by pool-1-thread-1

Task 5 completed by pool-1-thread-3

Task 8 started by pool-1-thread-3

Task 6 completed by pool-1-thread-2

Task 7 completed by pool-1-thread-1

Task 9 started by pool-1-thread-2

I/O Task 0 started by pool-1-thread-1

Task 8 completed by pool-1-thread-3

I/O Task 1 started by pool-1-thread-3

Task 9 completed by pool-1-thread-2

I/O Task 2 started by pool-1-thread-2

I/O Task 0 completed by pool-1-thread-1

I/O Task 3 started by pool-1-thread-1

I/O Task 1 completed by pool-1-thread-3

I/O Task 4 started by pool-1-thread-3

I/O Task 2 completed by pool-1-thread-2

I/O Task 5 started by pool-1-thread-2

I/O Task 3 completed by pool-1-thread-1

I/O Task 6 started by pool-1-thread-1

I/O Task 4 completed by pool-1-thread-3

I/O Task 7 started by pool-1-thread-3

I/O Task 5 completed by pool-1-thread-2

I/O Task 8 started by pool-1-thread-2

I/O Task 6 completed by pool-1-thread-1

I/O Task 9 started by pool-1-thread-1

I/O Task 7 completed by pool-1-thread-3

I/O Task 8 completed by pool-1-thread-2

I/O Task 9 completed by pool-1-thread-1

All tasks completed.

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.

```
import java.io.IOException;

import java.io.PrintWriter;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutorService;

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

```
import java.util.concurrent.Future;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
public class PrimeNumberCalculator {
   private static booleanisPrime(int number) {
      if (number <= 1) {
         return false;
      }
      for (inti = 2; i<= Math.sqrt(number); i++) {
         if (number % i == 0) {
            return false;
         }
      }
      return true;
   }
   public static List<Integer>calculatePrimes(int limit, ExecutorServiceexecutorService) throws
Exception {
      List<Future<Integer>> futures = new ArrayList<>();
      for (inti = 2; i<= limit; i++) {
         final int number = i;
futures.add(executorService.submit(() ->isPrime(number) ? number : null));
      }
      List<Integer> primes = new ArrayList<>();
      for (Future<Integer>future : futures) {
```

```
            Integer prime = future.get();
            if (prime != null) {
primes.add(prime);
            }
        }
        return primes;
    }
    public static CompletableFuture<Void>writePrimesToFile(List<Integer> primes, String
filePath) {
        return CompletableFuture.runAsync(() -> {
            try (PrintWriter writer = new PrintWriter(Files.newBufferedWriter(Paths.get(filePath)))) {
                for (intprime : primes) {
writer.println(prime);
                }
            } catch (IOException e) {
e.printStackTrace();
            }
        });
    }
    public static void main(String[] args) {
int limit = 1000;
        String filePath = "primes.txt";
ExecutorServiceexecutorService = Executors.newFixedThreadPool(10);
        try {
```

```
            List<Integer> primes = calculatePrimes(limit, executorService);
System.out.println("Prime numbers calculated: " + primes);
CompletableFuture<Void>writeFuture = writePrimesToFile(primes, filePath);
writeFuture.thenRun(() ->System.out.println("Prime numbers written to file: " + filePath));
writeFuture.join();
        } catch (Exception e) {
e.printStackTrace();
        } finally {
executorService.shutdown();
        }
    }
}
```

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.

```
class Counter {
    private int count;
    public Counter(int initialCount) {
        this.count = initialCount;
    }
    public synchronized void increment() {
        count++;
```

```
    }
    public synchronized void decrement() {
        count--;
    }
    public synchronized int getCount() {
        return count;
    }
}
final class SharedData {
    private final String data;
    public SharedData(String data) {
        this.data = data;
    }
    public String getData() {
        return data;
    }
}
class CounterTask implements Runnable {
    private final Counter counter;
    private final SharedData sharedData;
    public CounterTask(Counter counter, SharedData sharedData) {
        this.counter = counter;
        this.sharedData = sharedData;
    }
```

```
    public void run() {
        for (int i = 0; i < 10; i++) {
            counter.increment();
            System.out.println(Thread.currentThread().getName() + " - Incremented: " +
counter.getCount() + ", SharedData: " + sharedData.getData());
            try {
                Thread.sleep(100); // Simulate some work
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            counter.decrement();
            System.out.println(Thread.currentThread().getName() + " - Decremented: " +
counter.getCount() + ", SharedData: " + sharedData.getData());
        }
    }
}
public class ThreadSafeCounterDemo {
    public static void main(String[] args) {
        Counter counter = new Counter(0);
        SharedData sharedData = new SharedData("Immutable Data");
        Thread t1 = new Thread(new CounterTask(counter, sharedData), "Thread 1");
        Thread t2 = new Thread(new CounterTask(counter, sharedData), "Thread 2");
        Thread t3 = new Thread(new CounterTask(counter, sharedData), "Thread 3");
        t1.start();
        t2.start();
```

```
        t3.start();
        try {
            t1.join();
            t2.join();
            t3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final counter value: " + counter.getCount());
    }
}
```

Output:

```
Thread 1 - Incremented: 1, SharedData: Immutable Data
Thread 2 - Incremented: 2, SharedData: Immutable Data
Thread 3 - Incremented: 3, SharedData: Immutable Data
Thread 1 - Decremented: 2, SharedData: Immutable Data
Thread 3 - Decremented: 0, SharedData: Immutable Data
Thread 3 - Incremented: 2, SharedData: Immutable Data
Thread 2 - Decremented: 1, SharedData: Immutable Data
Thread 2 - Incremented: 3, SharedData: Immutable Data
Thread 1 - Incremented: 1, SharedData: Immutable Data
Thread 3 - Decremented: 2, SharedData: Immutable Data
Thread 3 - Incremented: 3, SharedData: Immutable Data
Thread 2 - Decremented: 2, SharedData: Immutable Data
```

Thread 2 - Incremented: 3, SharedData: Immutable Data

Thread 1 - Decremented: 2, SharedData: Immutable Data

Thread 1 - Incremented: 3, SharedData: Immutable Data

Thread 3 - Decremented: 2, SharedData: Immutable Data

Thread 3 - Incremented: 3, SharedData: Immutable Data

Thread 2 - Decremented: 2, SharedData: Immutable Data

Thread 2 - Incremented: 2, SharedData: Immutable Data

Thread 1 - Decremented: 1, SharedData: Immutable Data

Thread 1 - Incremented: 3, SharedData: Immutable Data

Thread 3 - Decremented: 2, SharedData: Immutable Data

Thread 3 - Incremented: 3, SharedData: Immutable Data

Thread 2 - Decremented: 2, SharedData: Immutable Data

Thread 2 - Incremented: 3, SharedData: Immutable Data

Thread 1 - Decremented: 2, SharedData: Immutable Data

Thread 1 - Incremented: 3, SharedData: Immutable Data

Thread 3 - Decremented: 2, SharedData: Immutable Data

Thread 3 - Incremented: 2, SharedData: Immutable Data

Thread 2 - Decremented: 2, SharedData: Immutable Data

Thread 2 - Incremented: 2, SharedData: Immutable Data

Thread 1 - Decremented: 1, SharedData: Immutable Data

Thread 1 - Incremented: 3, SharedData: Immutable Data

Thread 3 - Decremented: 2, SharedData: Immutable Data

Thread 1 - Decremented: 0, SharedData: Immutable Data

Thread 1 - Incremented: 1, SharedData: Immutable Data

Thread 2 - Decremented: 1, SharedData: Immutable Data

Thread 2 - Incremented: 2, SharedData: Immutable Data

Thread 3 - Incremented: 3, SharedData: Immutable Data

Thread 1 - Decremented: 2, SharedData: Immutable Data