Day 26 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

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.

```
package com.multi.threading;

public class PrintTheNumbers implements Runnable{

      private int start;

      public PrintTheNumbers(int start) {
       this.start = start;
      }

      @Override
      public void run() {
       for (int i = start; i <= 10; i++) {
        System.out.println(Thread.currentThread().getName()
+ ": " + i);
        try {
         Thread.sleep(1000); // 1 second delay
        } catch (InterruptedException e) {
         e.printStackTrace();
        }
       }
      }

      public static void main(String[] args) {
       Thread thread1 = new Thread(new PrintTheNumbers(1));
       Thread thread2 = new Thread(new PrintTheNumbers(1));
// Both threads start from 1

       thread1.setName("Thread-1");
       thread2.setName("Thread-2");

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

Output:

```
Thread-2: 1
Thread-1: 1
Thread-2: 2
Thread-1: 2
```

```
Thread-2: 3
Thread-1: 3
Thread-2: 4
Thread-1: 4
Thread-2: 5
Thread-1: 5
Thread-2: 6
Thread-1: 6
Thread-2: 7
Thread-1: 7
Thread-2: 8
Thread-1: 8
Thread-2: 9
Thread-1: 9
Thread-2: 10
Thread-1: 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.

```java
package com.multi.threading;

public class ThreadStateSimulator implements Runnable {
           private final String ThreadName;

           public ThreadStateSimulator(String threadName) {
                 this.ThreadName = threadName;
           }
       @Override
       public void run() {
          System.out.println(ThreadName + ": NEW");  // Initially in
NEW state

          try {
                 synchronized(this){
                       System.out.println(ThreadName + ": RUNNABLE");
                       wait(2000); // Simulate waiting state
                       System.out.println(ThreadName + ": WAITING");
                 }

                 Thread.sleep(1000); // Simulate timed waiting state
                 System.out.println(ThreadName + ": TIMED_WAITING");

                 synchronized (this) {
                      System.out.println(ThreadName + ": BLOCKED
(waiting on notify)");
                      notify(); // Simulate release from blocked
state
                }

                      System.out.println(ThreadName + ": RUNNABLE");
                } catch (InterruptedException e) {
                           e.printStackTrace();
                }
                      System.out.println(ThreadName + ":
TERMINATED");
       }


       public static void main(String[] args) throws
InterruptedException {
             ThreadStateSimulator simulate = new
ThreadStateSimulator("Sample Thread");
             Thread thread = new Thread(simulate);
         thread.start();

         synchronized (simulate) {
           System.out.println("Main thread: RUNNABLE");
           simulate.notify(); // Simulate notification to waiting
thread
           System.out.println("Main thread: WAITING");
           simulate.wait(); // Simulate waiting on thread completion
           System.out.println("Main thread: RUNNABLE");
           }
           thread.join(); // Wait for the thread to finish
       }
     }
```

Output:

```
Main thread: RUNNABLE
Main thread: WAITING
Sample Thread: NEW
Sample Thread: RUNNABLE
Sample Thread: WAITING
Sample Thread: TIMED_WAITING
Sample Thread: BLOCKED (waiting on notify)
Main thread: RUNNABLE
Sample Thread: RUNNABLE
Sample Thread: 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.

```java
package com.multi.threading;

public class ProducerConsumer {
        private final Object lock = new Object();
        private Integer item = null; // Buffer to hold the item

        public void produce() throws InterruptedException {
          synchronized (lock) {
            while (item != null) { // Wait if buffer is full
              lock.wait();
            }
            item = 1; // Produce an item (replace 1 with your actual data)
            System.out.println("Produced: " + item);
            lock.notify(); // Notify the consumer
          }
        }

        public void consume() throws InterruptedException {
          synchronized (lock) {
            while (item == null) { // Wait if buffer is empty
              lock.wait();
            }
            System.out.println("Consumed: " + item);
            item = null; // Consume the item
            lock.notify(); // Notify the producer
          }
        }

        public static void main(String[] args) {
          ProducerConsumer pc = new ProducerConsumer();
          Thread producerThread = new Thread(() -> {
            try {
              for (int i = 0; i < 5; i++) {
                System.out.println("Product No: " + (i + 1));
                pc.produce();
                Thread.sleep(1000); // Simulate production time
              }
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
          });
          Thread consumerThread = new Thread(() -> {
            try {
              for (int i = 0; i < 5; i++) {
                pc.consume();
                Thread.sleep(500); // Simulate consumption time
              }
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
          });
          producerThread.start(); // Start producer first
          consumerThread.start();
          try {
            producerThread.join();
            consumerThread.join();
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
```

Output:

```
Product No: 1
Produced: 1
Consumed: 1
Product No: 2
Produced: 1
Consumed: 1
Product No: 3
Produced: 1
Consumed: 1
Product No: 4
Produced: 1
Consumed: 1
Product No: 5
Produced: 1
Consumed: 1
```

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.

```
package com.multi.threading;

public class BankAccount {
```

```java
        private final Object lock = new Object(); // Shared lock for the account
        private int balance;

        public BankAccount(int initialBalance) {
          this.balance = initialBalance;
          System.out.println("Initial Balance: " + initialBalance);
        }

        public void deposit(int amount) {
          synchronized (lock) { // Acquire lock on the account object
            balance += amount;
            System.out.println("Deposited: " + amount + ", New balance: " + balance);
          }
        }

        public void withdraw(int amount) {
          synchronized (lock) { // Acquire lock on the account object
            if (balance >= amount) {
              balance -= amount;
              System.out.println("Withdrew: " + amount + ", New balance: " + balance);
            } else {
              System.out.println("Insufficient funds.");
            }
          }
        }

        public static void main(String[] args) {
          BankAccount account = new BankAccount(1000);
          // Anonymous inner class for deposit thread
          Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
              account.deposit(500);
            }
          });
          // Anonymous inner class for withdraw thread
          Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
              account.withdraw(750);
            }
          });
          thread1.start();
          thread2.start();
          try {
            thread1.join();
            thread2.join();
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
```

Output:

```
Initial Balance: 1000
Deposited: 500, New balance: 1500
Withdrew: 750, New balance: 750
```

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.

```
package com.concurrency;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;



public class ComplexTaskExecutor {

     public static void main(String[] args) throws 
InterruptedException {

         // Create a thread pool with 2 threads

         ExecutorService processor = 
Executors.newFixedThreadPool(2);



         // Define tasks with different processing times and inputs

         Runnable process1 = () -> {

           try {
             System.out.println("Processing Unit 1 started...
(Data: Input A)");

             Thread.sleep(4000); // Simulate complex operation
(e.g., data analysis)

             System.out.println("Processing Unit 1 completed.");

           } catch (InterruptedException e) {

             e.printStackTrace();

           }

         };

         Runnable process2 = () -> {

           try {

             System.out.println("Processing Unit 2 started...
(Data: Input B)");

             Thread.sleep(2000); // Simulate complex operation
(e.g., network call)

             System.out.println("Processing Unit 2 completed.");

           } catch (InterruptedException e) {

             e.printStackTrace();

           }

         };

         Runnable process3 = () -> {

           try {

             System.out.println("Processing Unit 3 started...
(Data: Input C)");

             Thread.sleep(1000); // Simulate complex operation
(e.g., file processing)

             System.out.println("Processing Unit 3 completed.");

           } catch (InterruptedException e) {

             e.printStackTrace();
           }
         };
         // Submit tasks to the thread pool
         processor.submit(process1);
         processor.submit(process2);
         processor.submit(process3);
         // Wait for all tasks to finish
         processor.shutdown();
         if (processor.awaitTermination(10, TimeUnit.SECONDS)) {
           System.out.println("All processing units completed successfully.");
         } else {
           System.out.println("Timeout waiting for processing units to finish.");
         }
       }
   }
```

Output:

```
Processing Unit 2 started... (Data: Input B)
Processing Unit 1 started... (Data: Input A)
Processing Unit 2 completed.
Processing Unit 3 started... (Data: Input C)
Processing Unit 3 completed.
Processing Unit 1 completed.
All processing units completed successfully.
```

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.

```
package com.concurrency;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;


public class ConcurrentPrimeSearcher {
  public static boolean isPrime(int num) {
    if (num <= 1) return false;
    for (int i = 2; i * i <= num; i++) {
      if (num % i == 0) return false;
    }
    return true;
  }


  public static void main(String[] args) throws InterruptedException
{
    int searchLimit = 50;
    // Use ExecutorService to parallelize prime number search
    ExecutorService searcherPool = Executors.newFixedThreadPool(4);
    CompletableFuture<Void> searchFuture =
CompletableFuture.runAsync(() -> {
      IntStream.rangeClosed(2, searchLimit)
```

```
          .filter(ConcurrentPrimeSearcher::isPrime)
          .forEach(System.out::println);
    }, searcherPool);

    // Use CompletableFuture to write results asynchronously
(simulated)
    searchFuture.thenRunAsync(() -> {
      System.out.println("Writing prime numbers to a file...");
      try {
        Thread.sleep(1000); // Simulate writing time
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      System.out.println("Prime numbers written successfully.");
    }, searcherPool);
    // Wait for all tasks to finish
    searcherPool.shutdown();
    searcherPool.awaitTermination(10, TimeUnit.SECONDS);
  }
}
```

Output:

```
2
3
5
7
11
13
17
19
23
29
31
```

```
37
41
43
47
```

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.

```java
package com.concurrency;

class Counter {
  private int count;

  public synchronized void increment(int amount) {
    count += amount;
  }

  public synchronized void decrement(int amount) {
    count -= amount;
  }

  public synchronized int getCount() {
    return count;
  }
}

final class ImmutableData {
  private final String data;

  public ImmutableData(String data) {
    this.data = data;
  }

  public String getData() {
    return data;
  }
}


public class Main{
public static void main(String[] args) throws InterruptedException {
    Counter counter = new Counter();
  ImmutableData sharedData = new ImmutableData("Shared
information");

  Runnable incrementTask = () -> {
   for (int i = 0; i < 1000; i++) {
    counter.increment(1);
   }
   System.out.println("Increment task done");
  };

  Runnable decrementTask = () -> {
   for (int i = 0; i < 1000; i++) {
    counter.decrement(1);
   }
   System.out.println("Decrement task done");
  };

  Thread thread1 = new Thread(incrementTask);
  Thread thread2 = new Thread(decrementTask);
  Thread thread3 = new Thread(incrementTask);

  thread1.start();
  thread2.start();
  thread3.start();

  thread1.join();
  thread2.join();
  thread3.join();

  System.out.println("Final count: " + counter.getCount());
  System.out.println("Shared data: " + sharedData.getData());
 }
}
```

Output:

```
Increment task done
Decrement task done
Increment task done
Final count: 1000
Shared data: Shared information
```