ASSIGNMENT: Day_18

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

ANS:

```
public class CreatingAndManagingThreads implements Runnable {
   private String threadName;


   public CreatingAndManagingThreads(String threadName) {
      this.threadName = threadName;
   }


   @Override
   public void run() {
      for (int i = 1; i <= 10; i++) {
         System.out.println(threadName + ": " + i);
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println(threadName + " interrupted.");
         }
      }
   }


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

```
        Thread thread1 = new Thread(new CreatingAndManagingThreads("Thread 1"));
        Thread thread2 = new Thread(new CreatingAndManagingThreads("Thread 2"));

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

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..

ANS:

```
public class StatesAndTransition {

    public static void main(String[] args) throws InterruptedException {
        LifecycleTask lifecycleTask = new LifecycleTask();

        Thread thread1 = new Thread(lifecycleTask::doWork);
        Thread thread2 = new Thread(lifecycleTask::doWorkWithWait);

        System.out.println("State after creating thread1: " + thread1.getState());

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

```
        System.out.println("State after starting thread1: " + thread1.getState());
        Thread.sleep(100);
        System.out.println("State of thread1 while running: " + thread1.getState());
        System.out.println("State of thread2 while running: " + thread2.getState());
        synchronized (lifecycleTask) {
            lifecycleTask.notify();
        }

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

        System.out.println("State of thread1 after finishing task: " + thread1.getState());
        System.out.println("State of thread2 after finishing task: " + thread2.getState());
    }

    static class LifecycleTask {
        public void doWork() {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        public void doWorkWithWait() {
```

```
        synchronized (this) {
            try {
                wait(500);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
  }
}
//code_by_RUBY
```

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.

ANS:

```
import java.util.LinkedList;
import java.util.Queue;

public class ProducerConsumer {
  private final Queue<Integer> queue = new LinkedList<>();
  private final int LIMIT = 5;
  private final Object lock = new Object();

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

```
    ProducerConsumer pc = new ProducerConsumer();
    Thread producerThread = new Thread(pc::produce);
    Thread consumerThread = new Thread(pc::consume);

    producerThread.start();
    consumerThread.start();
}

public void produce() {
    int value = 0;
    while (true) {
        synchronized (lock) {
            while (queue.size() == LIMIT) {
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
            queue.add(value++);
            lock.notify();
        }
    }
}
```

```
    public void consume() {
        while (true) {
            synchronized (lock) {
                while (queue.isEmpty()) {
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
                int value = queue.poll();
                System.out.println("Consumed: " + value);
                lock.notify();
            }
        }
    }
}
//code_by_RUBY
```

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.

ANS:

```
public class SynchronisedBlocksAndMethods {
    private int balance = 0;
```

```
public synchronized void deposit(int amount) {
    balance += amount;
    System.out.println("Deposited " + amount + ", Balance: " + balance);
}

public synchronized void withdraw(int amount) {
    if (balance >= amount) {
        balance -= amount;
        System.out.println("Withdrew " + amount + ", Balance: " + balance);
    } else {
        System.out.println("Insufficient balance. Attempted to withdraw: " + amount);
    }
}

public static void main(String[] args) {
    SynchronisedBlocksAndMethods account = new SynchronisedBlocksAndMethods();
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 10; i++) {
            account.deposit(10);
        }
    });

    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 10; i++) {
```

```
            account.withdraw(10);
        }
    });

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

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.

ANS:

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

public class ThreadPoolAndConcurrency {
  public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(3);

    for (int i = 0; i < 10; i++) {
      int taskId = i;
      executorService.submit(() -> {
```

```
            System.out.println("Task " + taskId + " is running on thread " +
Thread.currentThread().getName());
            try {
               Thread.sleep(2000);
            } catch (InterruptedException e) {
               Thread.currentThread().interrupt();
            }
         });
      }

      executorService.shutdown();
   }
}
//code_by_RUBY
```

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.

ANS:

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
```

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

public class Executors_ConcurrentCollections_CompletableFutures {

    public static void main(String[] args) {
        int upperLimit = 100;
        ExecutorService executor = Executors.newFixedThreadPool(4);

        CompletableFuture<List<Integer>> future = CompletableFuture.supplyAsync(() -> findPrimes(upperLimit), executor);

        future.thenAcceptAsync(primes -> {
            try (FileWriter writer = new FileWriter("primes.txt")) {
                for (int prime : primes) {
                    writer.write(prime + "\n");
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }, executor);

        future.join();
        executor.shutdown();
    }

    public static List<Integer> findPrimes(int upperLimit) {
```

```
        List<Integer> primes = new ArrayList<>();
        for (int num = 2; num <= upperLimit; num++) {
            if (isPrime(num)) {
                primes.add(num);
            }
        }
        return primes;
    }

    private static boolean isPrime(int num) {
        if (num <= 1) return false;
        for (int i = 2; i <= Math.sqrt(num); i++) {
            if (num % i == 0) return false;
        }
        return true;
    }
}

#code-by-RUBY
```

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.

ANS:

```
public class ThreadSafe_ImmutableObject {
   private int count = 0;

   public synchronized void increment() {
      count++;
   }

   public synchronized void decrement() {
      count--;
   }

   public synchronized int getCount() {
      return count;
   }

   public static void main(String[] args) {
      ThreadSafe_ImmutableObject counter = new ThreadSafe_ImmutableObject();
      Runnable incrementTask = counter::increment;
      Runnable decrementTask = counter::decrement;
```

```
        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(decrementTask);
        Thread t3 = new Thread(incrementTask);
       // Thread t4 = new Thread(decrementTask);

        t1.start();
        t2.start();
        t3.start();
      //  t4.start();

        try {
            t1.join();
            t2.join();
            t3.join();
            //t4.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("Final count: " + counter.getCount());
    }
}

final class ImmutableData {
```

```
    private final int value;


    public ImmutableData(int value) {
        this.value = value;
    }


    public int getValue() {
        return value;
    }
}
//code_by_RUBY
```