Prajwalhonashetti143@gmail.com

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

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

```
public class ThreadsDemo extends Thread {
    public static void main(String[] args) {
        ThreadsDemo t1 = new ThreadsDemo();
        ThreadsDemo t2 = new ThreadsDemo();

        t1.setPriority(5);
        t2.setPriority(5);

        t1.setName("child");
        t2.setName("parent");

        System.out.println(t1);
        System.out.println(t2);

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }

        System.out.println("Both threads have finished execution.");
    }

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

Output:

```
Thread[child,5,main]
Thread[parent,5,main]
child 1
parent 1
child 2
parent 2
child 3
parent 3
child 4
```

parent 4

child 5

parent 5

child 6

parent 6

child 7

parent 7

child 8

parent 8

child 9

parent 9

child 10

parent 10

Both threads have finished execution.

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..

```
public class ThreadLifecycleDemo {

    public static void main(String[] args) {

        ThreadLifecycleDemo demo = new ThreadLifecycleDemo();

        demo.runDemo();

    }


    private final Object monitor = new Object();
```

```
public void runDemo() {
  Thread thread = new Thread(new Worker());
  System.out.println("Thread State: " + thread.getState());

  thread.start();
  System.out.println("Thread State: " + thread.getState());

  try {
    Thread.sleep(100);
    System.out.println("Thread State (main thread sleeping): " + thread.getState());
    thread.join();
    System.out.println("Thread State: " + thread.getState());
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}

private class Worker implements Runnable {
  @Override
  public void run() {
    try {
      Thread.sleep(500);
 System.out.println("Thread State after sleep: " + Thread.currentThread().getState());

      synchronized (monitor)
        monitor.wait(500);
 System.out.println("Thread State after wait: " + Thread.currentThread().getState());
      }
```

```
            for (int i = 0; i < 3; i++) {
                System.out.println("Working... " + (i + 1));
                Thread.sleep(1000);
            }
            synchronized (monitor) {
                System.out.println("Thread in synchronized block");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation of the Code:

NEW State: The thread is created but not yet started.

System.out.println("Thread State: " + thread.getState());

RUNNABLE State: The thread is started and eligible to run.

- **thread.start();**
- System.out.println("Thread State: " + thread.getState());

TIMED_WAITING State: The thread is put to sleep using Thread.sleep().

- Thread.sleep(500); in run()
- System.out.println("Thread State after sleep: " + Thread.currentThread().getState());

WAITING State: The thread waits on a monitor object using monitor.wait().

- **monitor.wait(500);** in run()
- System.out.println("Thread State after wait: " + Thread.currentThread().getState());

BLOCKED State: Demonstrated by attempting to re-enter a synchronized block on the monitor object.

- The synchronized block in run(): synchronized (monitor) { ... }
- System.out.println("Thread in synchronized block");

TERMINATED State: The thread completes its execution.

- thread.join();
- System.out.println("Thread State: " + thread.getState());

Output:

- **Thread State: NEW**
- **Thread State: RUNNABLE**
- **Thread State (main thread sleeping): TIMED_WAITING**
- **Thread State after sleep: TIMED_WAITING**
- **Thread State after wait: TIMED_WAITING**
- **Working... 1**
- **Working... 2**
- **Working... 3**
- **Thread in synchronized block**
- **Thread State: 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;

public class ProducerConsumerExample {

    public static void main(String[] args) {

        Buffer buffer = new Buffer(5);


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

        Thread consumerThread = new Thread(new Consumer(buffer));
```

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

class Buffer {
    private final Queue<Integer> queue;
    private final int maxSize;

    public Buffer(int maxSize) {
        this.queue = new LinkedList<>();
        this.maxSize = maxSize;
    }

    public synchronized void produce(int value) throws InterruptedException {
        while (queue.size() == maxSize) {
            System.out.println("Buffer is full, producer is waiting...");
            wait();
        }
        queue.add(value);
        System.out.println("Produced " + value);
        notifyAll();
    }

    public synchronized int consume() throws InterruptedException {
        while (queue.isEmpty()) {
            System.out.println("Buffer is empty, consumer is waiting...");
            wait();
        }
```

```
        int value = queue.poll();
        System.out.println("Consumed " + value);
        notifyAll();
        return value;
    }
}

class Producer implements Runnable {
    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        int value = 0;
        try {
            while (true) {
                buffer.produce(value++);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

class Consumer implements Runnable {
```

```
    private final Buffer buffer;


    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }


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

Explanation

Buffer Class:

- The Buffer class manages the shared buffer (a queue) and its maximum size.
- produce(int value): Adds an item to the buffer. If the buffer is full, it waits until there is space.
- consume(): Removes and returns an item from the buffer. If the buffer is empty, it waits until there is an item to consume.

Producer Class:

- Implements the Runnable interface.
- The run method continuously produces items and places them into the buffer.

- It uses Thread.sleep(500) to simulate the time taken to produce an item.

Consumer Class:

- Implements the Runnable interface.
- The run method continuously consumes items from the buffer.
- It uses Thread.sleep(1000) to simulate the time taken to consume an item.

Main Class:

- Creates a Buffer object with a maximum size of 5.
- Creates and starts the producer and consumer threads.

Output:

- **Produced 0**
- **Consumed 0**
- **Produced 1**
- **Buffer is empty, consumer is waiting...**
- **Produced 2**
- **Consumed 1**
- **Produced 3**
- **Produced 4**
- **Produced 5**
- **Consumed 2**
- **Produced 6**
- **Consumed 3**
- **Produced 7**
- **Buffer is full, producer is waiting...**
- **Consumed 4**
- **Produced 8**
- **Consumed 5**
- **Produced 9**
- **Buffer is full, producer is waiting...**
- **Consumed 6**
- **Produced 10**

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.

To simulate a bank account being accessed by multiple threads for deposits and withdrawals, we can create a BankAccount class with synchronized methods to ensure thread safety. This will prevent race conditions by making sure that only one thread can execute a deposit or withdrawal at a time.

```
public 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 if (amount > balance) {
            System.out.println(Thread.currentThread().getName() + " attempted to withdraw " + amount + " but insufficient balance. Current balance: " + balance);
        }
    }
```

```
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.00);

        Thread t1 = new Thread(new DepositTask(account, 200.00), "T1");
        Thread t2 = new Thread(new WithdrawTask(account, 500.00), "T2");
        t1.start();
        t2.start();

    }
}

class DepositTask implements Runnable {
    private BankAccount account;
    private double amount;

    public DepositTask(BankAccount account, double amount) {
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run() {
        account.deposit(amount);
    }
}

class WithdrawTask implements Runnable {
    private BankAccount account;
    private double amount;
```

```java
    public WithdrawTask(BankAccount account, double amount) {
        this.account = account;
        this.amount = amount;
    }

    @Override
    public void run() {
        account.withdraw(amount);
    }
}
```

Explanation

BankAccount Class:

- Holds the balance of the account.
- synchronized deposit method: Adds money to the account if the deposit amount is positive. Synchronization ensures that only one thread can perform a deposit at a time.
- synchronized withdraw method: Withdraws money from the account if the withdrawal amount is positive and does not exceed the current balance. Synchronization ensures that only one thread can perform a withdrawal at a time.

DepositTask Class:

- Implements the Runnable interface.
- Takes a BankAccount instance and an amount to deposit.
- Calls the deposit method of the BankAccount within the run method.

WithdrawTask Class:

- Implements the Runnable interface.
- Takes a BankAccount instance and an amount to withdraw.
- Calls the withdraw method of the BankAccount within the run method.

Main Method:

- Creates a BankAccount instance with an initial balance.
- Creates multiple threads to perform deposit and withdrawal tasks.
- Starts the threads.

Output

- **T1 deposited 200.0. New balance: 1200.0**
- **T2 withdrew 500.0. New balance: 700.0**

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;


public class ThreadPoolExample {

  public static void main(String[] args) {


    ExecutorService executorService = Executors.newFixedThreadPool(4);


    for (int i = 1; i <= 4; i++) {

      int taskId = i;

      executorService.submit(() -> performComplexCalculation(taskId));

    }

    executorService.shutdown();


    try {
```

```
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }

        System.out.println("All tasks have finished execution.");
    }

    private static void performComplexCalculation(int taskId) {
  System.out.println("Task " + taskId + " started by " + Thread.currentThread().getName());
        try {
            Thread.sleep((long) (Math.random() * 5000));
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
                        System.out.println("Task " + taskId + " completed by " +
                                Thread.currentThread().getName());
                                                    }
}
```

Explanation

Creating a Fixed-Size Thread Pool:

- We use Executors.newFixedThreadPool(4) to create a thread pool with a fixed number of 4 threads.

Submitting Tasks to the Thread Pool:

- We submit exactly 4 tasks to the thread pool using a for loop. Each task calls the performComplexCalculation method, which simulates a complex calculation or I/O operation.

Shutting Down the Executor Service:

- We call executorService.shutdown() to stop accepting new tasks and to gracefully shut down the executor service.
- We use executorService.awaitTermination(60, TimeUnit.SECONDS) to wait for all tasks to complete. If the tasks do not complete within 60 seconds, we call executorService.shutdownNow() to forcibly shut down the executor service.

Task Method:

- The performComplexCalculation method prints the start and end of each task.
- It uses Thread.sleep((long) (Math.random() * 5000)) to simulate a time-consuming operation.

Sample Output

Task 1 started by pool-1-thread-1

Task 2 started by pool-1-thread-2

Task 3 started by pool-1-thread-3

Task 4 started by pool-1-thread-4

Task 1 completed by pool-1-thread-1

Task 2 completed by pool-1-thread-2

Task 3 completed by pool-1-thread-3

Task 4 completed by pool-1-thread-4

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.BufferedWriter;

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;
```

```java
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;

public class PrimeNumberCalculator {
    public static void main(String[] args) {
        int limit = 10000;
        int numberOfThreads = 4;

        ExecutorService executorService = Executors.newFixedThreadPool(numberOfThreads);


        List<CompletableFuture<List<Integer>>> futures = new ArrayList<>();
        int range = limit / numberOfThreads;
        for (int i = 0; i < numberOfThreads; i++) {
            int start = i * range + 1;
            int end = (i == numberOfThreads - 1) ? limit : (i + 1) * range;
            CompletableFuture<List<Integer>> future = CompletableFuture.supplyAsync(
                    () -> calculatePrimes(start, end), executorService);
            futures.add(future);
        }
        CompletableFuture<List<Integer>> allPrimesFuture = CompletableFuture.allOf(
                futures.toArray(new CompletableFuture[0]))
                .thenApply(v -> futures.stream()
                        .map(CompletableFuture::join)
                        .flatMap(List::stream)
                        .collect(Collectors.toList()));

        allPrimesFuture.thenAcceptAsync(primes -> writeToFile("primes.txt", primes))
```

```java
        .thenRun(() -> {
            System.out.println("Prime numbers have been written to the file.");

            executorService.shutdown();
            try {
                executorService.awaitTermination(10, TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        })
        .exceptionally(ex -> {
            ex.printStackTrace();
            return null;
        });
}

private static List<Integer> calculatePrimes(int start, int end) {
    List<Integer> primes = new ArrayList<>();
    for (int i = start; i <= end; i++) {
        if (isPrime(i)) {
            primes.add(i);
        }
    }
    return primes;
}

private static boolean isPrime(int number) {
    if (number <= 1) {
        return false;
```

```
        }
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }

    private static void writeToFile(String filename, List<Integer> data) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            for (int prime : data) {
                writer.write(prime + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation

Improved CompletableFuture Usage:

- CompletableFuture.supplyAsync(): Submits tasks to calculate prime numbers asynchronously.
- CompletableFuture.allOf().thenApply(): Combines results from all CompletableFutures into a single list of prime numbers.
- CompletableFuture.thenAcceptAsync(): Writes the prime numbers to a file asynchronously.
- CompletableFuture.thenRun(): Shuts down the executor service after writing to file.

Exception Handling:

- Uses exceptionally() to handle exceptions and print stack traces.

ExecutorService Management:

- Properly shuts down the executor service and waits for termination after writing to file.

Output

Prime numbers have been written to the file

3

5

7

11

13

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.

thread-safe Counter class with increment and decrement methods. Additionally, we will implement an immutable class to share data between threads and demonstrate their usage from multiple threads.

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

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;


public class ThreadSafeDemo {

  public static void main(String[] args) {


    Counter counter = new Counter();
```

```
        ImmutableData immutableData = new ImmutableData("Alice", 30);
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 4; i++) {
            executorService.execute(() -> {

                for (int j = 0; j < 1000; j++) {
                    counter.increment();
                    counter.decrement();
                }

                System.out.println("Immutable Data: " + immutableData.getName() + ", " +
immutableData.getAge());
            });
        }
        executorService.shutdown();

        try {
            executorService.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final Counter value: " + counter.getCount());
    }
}

class Counter {
    private int count = 0;

    public synchronized void increment() {
```

```
        count++;
    }

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

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

final class ImmutableData {
    private final String name;
    private final int age;

    public ImmutableData(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

The Counter class uses synchronized methods to ensure thread safety:

- **increment():** Increments the count variable in a thread-safe manner.
- **decrement():** Decrements the count variable in a thread-safe manner.
- **getCount():** Retrieves the current value of the count variable in a thread-safe manner.

Explanation:

- The ImmutableData class is immutable:
- It has final fields name and age which are set via the constructor.
- It provides only getters and no setters, ensuring that once an instance is created, its state cannot be modified.

Explanation:

Counter Usage:

- Creates a Counter instance and uses ExecutorService to run two threads concurrently.
- Thread 1 increments the counter 1000 times using counter.increment().
- Thread 2 decrements the counter 1000 times using counter.decrement().
- After threads complete, it prints the final counter value using counter.getCount()

ImmutableData Usage:

- Creates an ImmutableData instance with name "Alice" and age 30.
- Uses ExecutorService to run a thread that prints the immutable data.
- The ImmutableData instance is thread-safe because its state cannot be changed after creation.

ExecutorService:

- Uses Executors.newCachedThreadPool() to create an executor service that dynamically reuses threads.
- Shuts down the executor service after all tasks have completed.

Expected Output

Immutable Data: Alice, 30

Immutable Data: Alice, 30

Immutable Data: Alice, 30

Immutable Data: Alice, 30

Final Counter value: 0