TASK 3: SYNCHRONIZATION AND INTER-THREAD COMMUNICATION

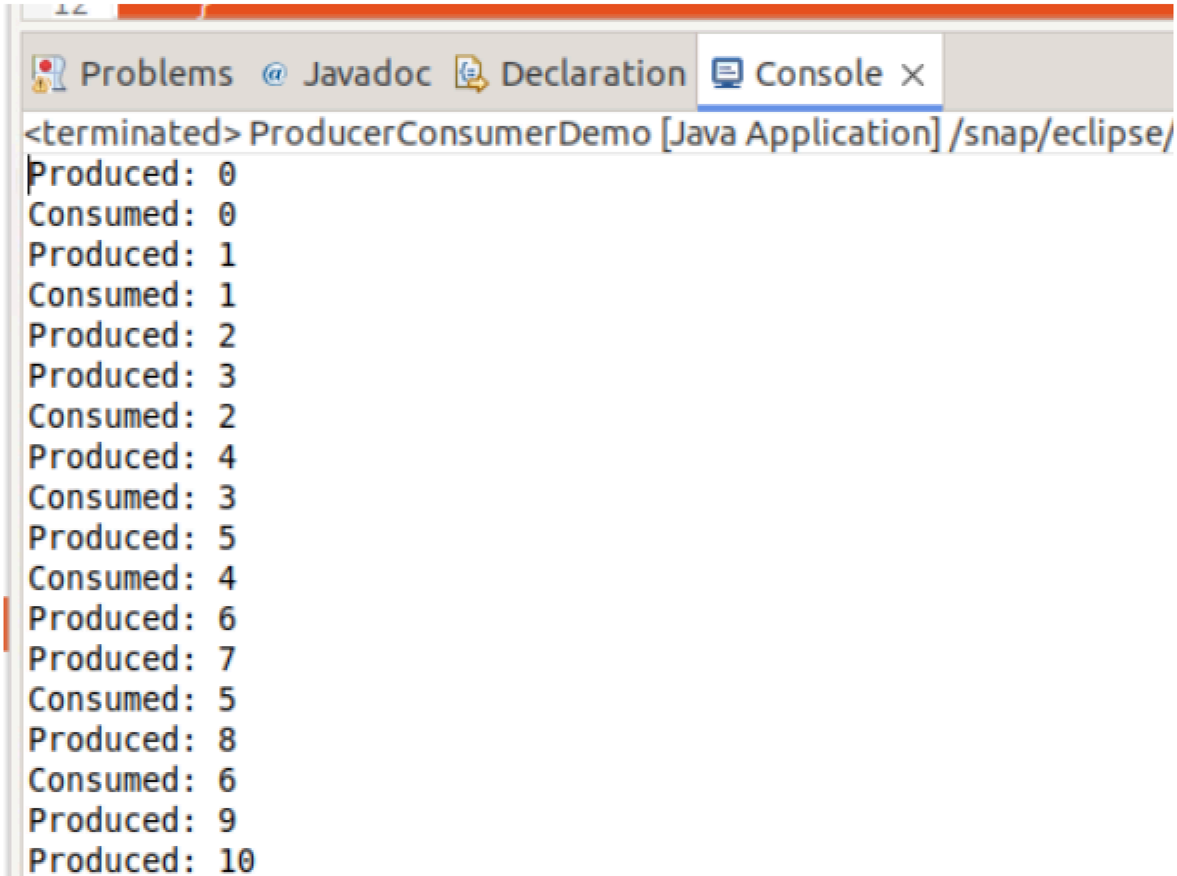
Implement a producer-consumer problem using wait() and notify() methods to handle the correct processing sequence between threads.

Producer-Consumer Implementation

```
package com.wipro.threading;

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

class SharedQueue {
    private final int capacity;
    private final Queue<Integer> queue = new LinkedList<>();

    public SharedQueue(int capacity) {
        this.capacity = capacity;
    }

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

    public synchronized int consume() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        int item = queue.poll();
```

```java
        System.out.println("Consumed: " + item);
        notifyAll();
        return item;
    }
}

class Producer implements Runnable {
    private final SharedQueue sharedQueue;

    public Producer(SharedQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
    }

    @Override
    public void run() {
        int item = 0;
        try {
            while (true) {
                sharedQueue.produce(item++);
                Thread.sleep(1000); // Simulate time taken to produce
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Producer interrupted");
        }
    }
}

class Consumer implements Runnable {
    private final SharedQueue sharedQueue;

    public Consumer(SharedQueue sharedQueue) {
        this.sharedQueue = sharedQueue;
    }
```

```
    @Override
    public void run() {
        try {
            while (true) {
                sharedQueue.consume();
                Thread.sleep(1500); // Simulate time taken to consume
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Consumer interrupted");
        }
    }
}

public class ProducerConsumerDemo {
    public static void main(String[] args) {
        SharedQueue sharedQueue = new SharedQueue(5); // Set buffer
size to 5
        Thread producerThread = new Thread(new
Producer(sharedQueue));
        Thread consumerThread = new Thread(new
Consumer(sharedQueue));

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



How It Works

- The producerThread produces items and adds them to the SharedQueue.
- If the queue is full, the producer thread waits until the consumer consumes an item.
- The consumerThread consumes items from the SharedQueue.
- If the queue is empty, the consumer thread waits until the producer produces an item.
- The wait() method causes the current thread to wait until another thread calls notify() or notifyAll() on the same object.
- The notifyAll() method wakes up all waiting threads on the object.

TASK 4: SYNCHRONIZED BLOCKS AND METHODS

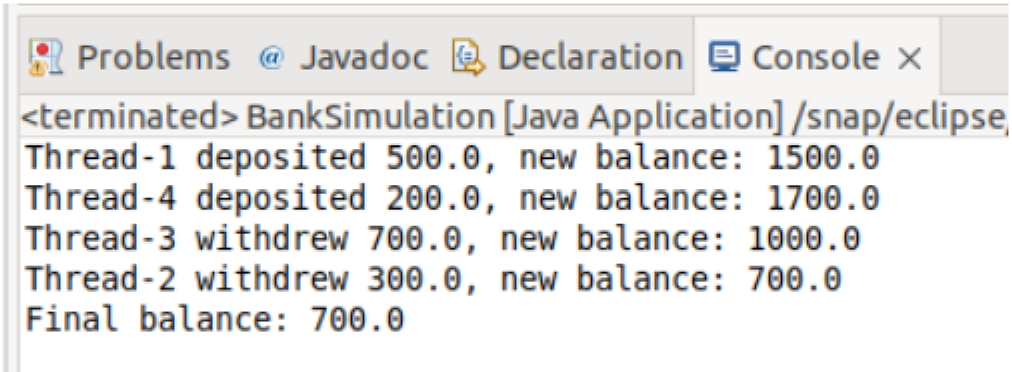
Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

BankAccount Simulation

```
package com.wipro.bank;

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() + " tried to
withdraw " + amount + ", but insufficient funds. Current balance: " +
balance);
        }
```

```java
    }

    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;
    }

    @Override
    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;
    }

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

```
    }
}

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

        Thread t1 = new Thread(new DepositTask(account, 500),
"Thread-1");
        Thread t2 = new Thread(new WithdrawTask(account, 300),
"Thread-2");
        Thread t3 = new Thread(new WithdrawTask(account, 700),
"Thread-3");
        Thread t4 = new Thread(new DepositTask(account, 200),
"Thread-4");

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

        try {
            t1.join();
            t2.join();
            t3.join();
            t4.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final balance: " + account.getBalance());
    }
}
```



How It Works

- Synchronization: The synchronized keyword ensures that only one thread can execute the deposit, withdraw, or getBalance methods at a time, preventing race conditions.
- Thread Operations: Multiple threads perform deposit and withdrawal operations concurrently.
- Thread Safety: The use of synchronized methods ensures that the balance updates correctly, even when accessed by multiple threads simultaneously.

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.

Thread Pool with Multiple Tasks

```
package com.wipro.threading;

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;
    }

    @Override
    public void run() {
        System.out.println("Task " + taskId + " started by " +
Thread.currentThread().getName());
        long result = performComplexCalculation(taskId);
        System.out.println("Task " + taskId + " completed by " +
Thread.currentThread().getName() + " with result: " + result);
    }

    private long performComplexCalculation(int n) {
        long sum = 0;
        for (int i = 0; i < n * 1000; i++) {
            sum += i;
        }
        return sum;
```

```
    }
}

class IOTask implements Runnable {
    private final int taskId;

    public IOTask(int taskId) {
        this.taskId = taskId;
    }

    @Override
    public void run() {
        System.out.println("I/O Task " + taskId + " started by " +
Thread.currentThread().getName());
        simulateIOOperation();
        System.out.println("I/O Task " + taskId + " completed by " +
Thread.currentThread().getName());
    }

    private void simulateIOOperation() {
        try {
            Thread.sleep(2000); // Simulate I/O operation
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("I/O Task interrupted");
        }
    }
}

public class ThreadPoolDemo {
    public static void main(String[] args) {
        ExecutorService executorService =
Executors.newFixedThreadPool(4);
```

```
        // Submit Calculation Tasks
        for (int i = 1; i <= 5; i++) {
            executorService.submit(new CalculationTask(i));
        }

        // Submit I/O Tasks
        for (int i = 1; i <= 3; i++) {
            executorService.submit(new IOTask(i));
        }

        executorService.shutdown();

        try {
            if (!executorService.awaitTermination(10, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }

        System.out.println("All tasks completed");
    }
}
```

```
Problems  @ Javadoc  Declaration  Console ×
<terminated> ThreadPoolDemo [Java Application] /snap/eclipse/87/plu
Task 3 started by pool-1-thread-3
Task 1 started by pool-1-thread-1
Task 4 started by pool-1-thread-4
Task 2 started by pool-1-thread-2
Task 1 completed by pool-1-thread-1 with result: 499500
Task 3 completed by pool-1-thread-3 with result: 4498500
Task 5 started by pool-1-thread-1
Task 5 completed by pool-1-thread-1 with result: 12497500
I/O Task 2 started by pool-1-thread-1
Task 4 completed by pool-1-thread-4 with result: 7998000
I/O Task 3 started by pool-1-thread-4
Task 2 completed by pool-1-thread-2 with result: 1999000
I/O Task 1 started by pool-1-thread-3
I/O Task 2 completed by pool-1-thread-1
I/O Task 3 completed by pool-1-thread-4
I/O Task 1 completed by pool-1-thread-3
All tasks completed
```

How It Works

- Fixed-Size Thread Pool: The thread pool is created with a fixed size of 4 threads, meaning at most 4 tasks will run concurrently.
- Task Submission: A total of 8 tasks (5 calculation tasks and 3 I/O tasks) are submitted to the thread pool.
- Execution: Tasks are executed by the available threads in the pool. If more tasks are submitted than there are available threads, the tasks wait in a queue until a thread becomes available.
- Shutdown and Await Termination: After submitting all tasks, the executor service is shut down, and the program waits for all tasks to complete using awaitTermination.

TASK 6: EXECUTORS, CONCURRENT COLLECTIONS, COMPLETABLEFUTURE

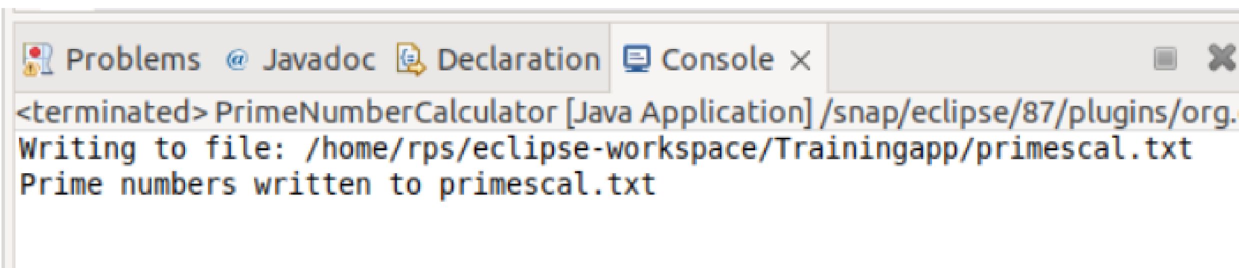
Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

```
package com.wipro.prime;

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;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class PrimeNumberCalculator {

    public static void main(String[] args) {
        int upperLimit = 100000;
        int numberOfThreads = 4;

        ExecutorService executorService =
Executors.newFixedThreadPool(numberOfThreads);
try {
            List<Integer> primeNumbers = findPrimes(upperLimit,
executorService);
            String filename = "primescal.txt";
            System.out.println("Writing to file: " + new
java.io.File(filename).getAbsolutePath());
```

```
            writePrimesToFileAsync(primeNumbers, filename,
executorService);
        }
 finally {
            executorService.shutdown();
            try {
                if (!executorService.awaitTermination(60, TimeUnit.SECONDS))
{
                    executorService.shutdownNow();
                }
            } catch (InterruptedException e) {
                executorService.shutdownNow();
            }
        }
    }

    public static List<Integer> findPrimes(int upperLimit, ExecutorService
executorService) {
        List<CompletableFuture<List<Integer>>> futures = new
ArrayList<>();
        int chunkSize = upperLimit / 4;

        for (int i = 0; i < 4; i++) {
            int start = i * chunkSize + 1;
            int end = (i == 3) ? upperLimit : start + chunkSize - 1;
            futures.add(CompletableFuture.supplyAsync(() ->
calculatePrimes(start, end), executorService));
        }

        List<Integer> primeNumbers = new ArrayList<>();
        futures.forEach(future -> {
            try {
                primeNumbers.addAll(future.get());
            } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    });

    return primeNumbers;
}

public 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;
}

public 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;
}

public static void writePrimesToFileAsync(List<Integer> primes, String
filename, ExecutorService executorService) {
    CompletableFuture.runAsync(() -> {
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(filename))) {
```

```java
                for (int prime : primes) {
                    writer.write(prime + System.lineSeparator());
                }
                System.out.println("Prime numbers written to " + filename);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }, executorService);
    }
}
```



Text file containing prime numbers from 1 to 1000.

primescal.txt
~/eclipse-workspace/Trainingapp

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
```

How It Works

- Thread Pool: A fixed-size thread pool is created with 4 threads.
- Task Parallelization: The range of numbers up to the given limit is divided into chunks, and each chunk is processed in parallel to find prime numbers.
- Asynchronous File Writing: After calculating the primes, the results are written to a file asynchronously using CompletableFuture.
- Graceful Shutdown: The executor service is shut down gracefully, ensuring all tasks are completed before the application exits.

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

To ensure thread safety in the `Counter` class, we'll use the `synchronized` keyword to make the `increment` and `decrement` methods thread-safe.

```
public class Counter {
    private int count;

    public Counter() {
        this.count = 0;
    }

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

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

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

Immutable Class to Share Data Between Threads

Immutable classes are inherently thread-safe because their state cannot be modified after they are created. Here's an example of an immutable class:

```
public final class ImmutableData {
    private final int value;

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

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

Demonstration of Usage from Multiple Threads

We will create multiple threads that will increment and decrement the counter and use the immutable class to share data between threads.

```
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
```

```
        }
    };

    Runnable decrementTask = () -> {
        for (int i = 0; i < 1000; i++) {
            counter.decrement();
        }
    };

    Thread thread1 = new Thread(incrementTask);
    Thread thread2 = new Thread(decrementTask);
    Thread thread3 = new Thread(incrementTask);
    Thread thread4 = new Thread(decrementTask);

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();

    try {
        thread1.join();
        thread2.join();
        thread3.join();
        thread4.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Final Counter Value: " + counter.getCount());

    // Demonstrating usage of ImmutableData
    ImmutableData data = new ImmutableData(42);
    System.out.println("Immutable Data Value: " + data.getValue());
}
```

```
}
```

Output

The output will show the final value of the counter after all increments and decrements are done, and it will print the value of the immutable data.