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.

```
class NumberPrinter implements Runnable {
    private String threadName;

    public NumberPrinter(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); // 1-second delay
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted.");
            }
        }
        System.out.println(threadName + " finished.");
    }
}

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

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


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

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 ThreadLifecycleExample {


    private static final Object lock = new Object();


    public static void main(String[] args) {
        // Thread to demonstrate NEW, RUNNABLE, TIMED_WAITING, and TERMINATED states
        Thread thread1 = new Thread(() -> {
            try {
                System.out.println(Thread.currentThread().getName() + " - State: RUNNABLE");
                Thread.sleep(2000); // TIMED_WAITING state
                System.out.println(Thread.currentThread().getName() + " - State: TERMINATED");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
}, "Thread-1");

// Thread to demonstrate WAITING state and BLOCKED state
Thread thread2 = new Thread(() -> {
    synchronized (lock) {
        try {
            System.out.println(Thread.currentThread().getName() + " - State: WAITING");
            lock.wait(); // WAITING state
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println(Thread.currentThread().getName() + " - State: RUNNABLE");
    System.out.println(Thread.currentThread().getName() + " - State: TERMINATED");
}, "Thread-2");

// Thread to demonstrate BLOCKED state
Thread thread3 = new Thread(() -> {
    synchronized (lock) {
        System.out.println(Thread.currentThread().getName() + " - State: RUNNABLE");
        lock.notify(); // Notify thread2 to wake up from WAITING
        System.out.println(Thread.currentThread().getName() + " - State: TERMINATED");
    }
}, "Thread-3");
```

```java
// Start thread1

System.out.println(thread1.getName() + " - State: NEW");

thread1.start();


// Start thread2 and thread3

System.out.println(thread2.getName() + " - State: NEW");

thread2.start();

try {

        Thread.sleep(1000); // Ensure thread2 goes into WAITING state

} catch (InterruptedException e) {

        e.printStackTrace();

}

System.out.println(thread3.getName() + " - State: NEW");

thread3.start();


// Join threads to ensure main thread waits for their completion

try {

        thread1.join();

        thread2.join();

        thread3.join();

} catch (InterruptedException e) {

        e.printStackTrace();

}


System.out.println("All threads have finished execution.");
```

```
    }
}
```

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;

class SharedBuffer {
    private LinkedList<Integer> list = new LinkedList<>();
    private int capacity = 5;

    public void produce() throws InterruptedException {
        int value = 0;
        while (true) {
            synchronized (this) {
                while (list.size() == capacity) {
                    wait();
                }

                System.out.println("Producer produced: " + value);
                list.add(value++);
                notify();
                Thread.sleep(1000); // Simulate time taken to produce an item
```

```java
            }
        }
    }

    public void consume() throws InterruptedException {
        while (true) {
            synchronized (this) {
                while (list.isEmpty()) {
                    wait();
                }

                int value = list.removeFirst();
                System.out.println("Consumer consumed: " + value);
                notify();
                Thread.sleep(1000); // Simulate time taken to consume an item
            }
        }
    }
}

public class ProducerConsumerExample {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();

        Thread producerThread = new Thread(new Runnable() {
```

```java
        @Override
        public void run() {
            try {
                buffer.produce();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    Thread consumerThread = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                buffer.consume();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

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

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 && balance >= amount) {

            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. Current balance: " + balance);
```

```
        }
    }

    public synchronized double getBalance() {
        return balance;
    }
}

class BankingTask implements Runnable {
    private BankAccount account;
    private boolean isDeposit;
    private double amount;

    public BankingTask(BankAccount account, boolean isDeposit, double amount) {
        this.account = account;
        this.isDeposit = isDeposit;
        this.amount = amount;
    }

    @Override
    public void run() {
        if (isDeposit) {
            account.deposit(amount);
        } else {
            account.withdraw(amount);
```

```java
        }
    }
}

public class BankAccountExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000); // Initial balance of 1000

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

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

        // Wait for all threads to finish
        try {
            t1.join();
            t2.join();
            t3.join();
            t4.join();
        } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }

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

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 ComplexCalculationTask implements Runnable {
    private final int taskId;

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

    @Override
    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() {
        // Simulate a complex calculation or I/O operation with sleep
        try {
            Thread.sleep(2000); // Sleep for 2 seconds to simulate work
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Task " + taskId + " interrupted.");
        }
    }
}

public class ThreadPoolExample {
    public static void main(String[] args) {
        // Create a fixed-size thread pool with 4 threads
        ExecutorService executorService = Executors.newFixedThreadPool(4);

        // Submit 10 tasks to the thread pool
        for (int i = 1; i <= 10; i++) {
            executorService.submit(new ComplexCalculationTask(i));
        }

        // Shut down the executor service
        executorService.shutdown();
```

```
        try {
            // Wait for all tasks to complete
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }

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

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

```
public class PrimeNumberCalculator {

    public static void main(String[] args) {
        int limit = 1000; // Calculate primes up to 1000
        int numberOfThreads = 4; // Number of threads in the pool
        String outputFile = "primes.txt";

        // Create a fixed-size thread pool
        ExecutorService executorService = Executors.newFixedThreadPool(numberOfThreads);

        // Calculate primes in parallel
        List<CompletableFuture<List<Integer>>> futures = new ArrayList<>();
        for (int i = 0; i < numberOfThreads; i++) {
            int start = i * (limit / numberOfThreads) + 1;
            int end = (i + 1) * (limit / numberOfThreads);
            futures.add(CompletableFuture.supplyAsync(() -> calculatePrimes(start, end),
executorService));
        }

        // Combine the results from all futures
        CompletableFuture<Void> allOf = CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0]));
        CompletableFuture<List<Integer>> allPrimes = allOf.thenApply(v -> {
            List<Integer> primes = new ArrayList<>();
            futures.forEach(future -> primes.addAll(future.join()));
            return primes;
```

```java
});

// Write results to file asynchronously
CompletableFuture<Void> writeToFile = allPrimes.thenAcceptAsync(primes -> {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile))) {
        for (Integer prime : primes) {
            writer.write(prime + "\n");
        }
        System.out.println("Primes written to file: " + outputFile);
    } catch (IOException e) {
        e.printStackTrace();
    }
}, executorService);

// Wait for all tasks to complete
writeToFile.join();

// Shutdown the executor service
executorService.shutdown();
try {
    if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
        executorService.shutdownNow();
    }
} catch (InterruptedException e) {
    executorService.shutdownNow();
```

```java
        }

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

    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);
            }
        }
        System.out.println(Thread.currentThread().getName() + " calculated primes from " + start + " to " + end);
        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;
    }
}
```

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.

```
// Counter.java
public class Counter {
    private int count = 0;

    // Synchronized increment method
    public synchronized void increment() {
        count++;
        System.out.println(Thread.currentThread().getName() + " incremented to " + count);
    }

    // Synchronized decrement method
    public synchronized void decrement() {
        count--;
        System.out.println(Thread.currentThread().getName() + " decremented to " + count);
    }

    // Synchronized method to get the current count
    public synchronized int getCount() {
        return count;
```

```java
    }
}

// ImmutableData.java
public final class ImmutableData {
    private final int value;

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

    public int getValue() {
        return value;
    }
}

// ThreadSafeCounterExample.java
public class ThreadSafeCounterExample {

    public static void main(String[] args) {
        Counter counter = new Counter();
        ImmutableData sharedData = new ImmutableData(42); // Shared immutable data

        // Create threads to increment and decrement the counter
        Thread t1 = new Thread(new CounterTask(counter, true), "Thread-1");
```

```java
        Thread t2 = new Thread(new CounterTask(counter, false), "Thread-2");
        Thread t3 = new Thread(new CounterTask(counter, true), "Thread-3");
        Thread t4 = new Thread(new CounterTask(counter, false), "Thread-4");

        // Start the threads
        t1.start();
        t2.start();
        t3.start();
        t4.start();

        // Demonstrate usage of immutable data
        System.out.println("Shared immutable data value: " + sharedData.getValue());

        // Wait for all threads to finish
        try {
            t1.join();
            t2.join();
            t3.join();
            t4.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

```
}

class CounterTask implements Runnable {
    private final Counter counter;
    private final boolean increment;

    public CounterTask(Counter counter, boolean increment) {
        this.counter = counter;
        this.increment = increment;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) { // Perform 10 operations
            if (increment) {
                counter.increment();
            } else {
                counter.decrement();
            }
            try {
                Thread.sleep(100); // Simulate some work with sleep
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
```

```
    }
}
```