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 ProducerConsumer {

  private static final int CAPACITY = 5;
  private final Queue<Integer> queue = new LinkedList<>();

  // Lock object for synchronization (better practice)
  private final Object lock = new Object();

  public void produce() throws InterruptedException {
    synchronized (lock) {
      while (queue.size() == CAPACITY) {
        lock.wait(); // Wait for space in the queue
      }

      int value = (int) (Math.random() * 100);
      queue.offer(value); // Add to queue using offer()
      System.out.println("Producer produced: " + value);
      lock.notifyAll(); // Notify waiting consumers
    }
  }

  public int consume() throws InterruptedException {
    synchronized (lock) {
      while (queue.isEmpty()) {
        lock.wait(); // Wait for items in the queue
      }
```

```
        int value = queue.poll(); // Remove from queue using poll()
        System.out.println("Consumer consumed: " + value);
        lock.notifyAll(); // Notify waiting producers
        return value;
    }
}

public static void main(String[] args) {
    ProducerConsumer pc = new ProducerConsumer();

    Thread producerThread = new Thread(() -> {
        try {
            while (true) {
                pc.produce();
                Thread.sleep(1000); // Simulate production time
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // Proper way to handle interruption
        }
    });

    Thread consumerThread = new Thread(() -> {
        try {
            while (true) {
                pc.consume();
                Thread.sleep(2000); // Simulate consumption time
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    });

    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 + ", current balance: " +
balance);
        }
    }

    public synchronized void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println(Thread.currentThread().getName() + " withdrew " + amount + ", current balance: " +
balance);
        } else {
            System.out.println(Thread.currentThread().getName() + " tried to withdraw " + amount + ", but insufficient
funds. Current balance: " + balance);
        }
```

```
    }

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

class AccountHolder implements Runnable {
    private BankAccount account;

    public AccountHolder(BankAccount account) {
        this.account = account;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            double depositAmount = Math.random() * 100;
            account.deposit(depositAmount);

            double withdrawalAmount = Math.random() * 100;
            account.withdraw(withdrawalAmount);

            try {
                Thread.sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

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

```
        BankAccount account = new BankAccount(1000);

        Thread accountHolder1 = new Thread(new AccountHolder(account), "AccountHolder1");
        Thread accountHolder2 = new Thread(new AccountHolder(account), "AccountHolder2");
        Thread accountHolder3 = new Thread(new AccountHolder(account), "AccountHolder3");

        accountHolder1.start();
        accountHolder2.start();
        accountHolder3.start();
    }
}
```

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.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {

    public static void main(String[] args) {
        int numThreads = 4; // Number of threads in the pool
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);

        // Complex Calculation Task
        Runnable calculationTask = () -> {
            long result = 0;
            for (long i = 0; i < 1000000000L; i++) {
                result += i;
```

```
        }
        System.out.println(Thread.currentThread().getName() + ": Calculation result: " + result);
    };

    // File I/O Task
    Runnable ioTask = () -> {
        try {
            String content = Files.readString(Paths.get("your_file.txt"));
            System.out.println(Thread.currentThread().getName() + ": File content:\n" + content);
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    };

    // Submitting tasks
    for (int i = 0; i < 5; i++) {
        executor.submit(calculationTask);
        executor.submit(ioTask);
    }

    // Shutting down the executor
    executor.shutdown();
  }
}
```

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.IOException;
import java.nio.file.Files;
```

```
import java.nio.file.Paths;
import java.util.Set;
import java.util.concurrent.*;

public class PrimeCalculator {

  // Concurrent collection to store prime numbers
  private static final Set<Integer> primes = ConcurrentHashMap.newKeySet();

  // Method to check if a number is prime
  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;
  }

  public static void main(String[] args) throws IOException, InterruptedException, ExecutionException {
    int upperLimit = 1000000; // Calculate primes up to this number
    int numThreads = Runtime.getRuntime().availableProcessors(); // Use available processors
    ExecutorService executor = Executors.newFixedThreadPool(numThreads);

    List<CompletableFuture<Void>> futures = new ArrayList<>();
    int rangeSize = upperLimit / numThreads; // Divide work into ranges

    for (int i = 0; i < numThreads; i++) {
      final int start = i * rangeSize + 1;
      final int end = (i + 1) * rangeSize;

      futures.add(CompletableFuture.runAsync(() -> {
        for (int num = start; num <= end; num++) {
          if (isPrime(num)) {
            primes.add(num);
```

```
                }
            }
        }, executor));
    }

    // Wait for all calculations to complete
    CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).get();

    // Asynchronously write results to file
    CompletableFuture<Void> writeFuture = CompletableFuture.runAsync(() -> {
        try {
            Files.write(Paths.get("primes.txt"), primes.toString().getBytes());
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }
    });

    System.out.println("Prime numbers calculated and writing to file...");

    // You can do other work here while the file is being written

    writeFuture.get(); // Optional: Wait for file writing to complete
    System.out.println("Primes written to primes.txt");

    executor.shutdown();
  }
}
```

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 Counter {
    private int value = 0;

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

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

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

// Immutable Data Class
final class ImmutableData {
    private final String data;

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

    public String getData() {
        return data;
    }
}

// Demonstration (Main Class)
public class ThreadSafetyDemo {

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

```
        Counter counter = new Counter();
        ImmutableData data = new ImmutableData("Shared data is immutable");

        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
            System.out.println(Thread.currentThread().getName() + ": Counter = " + counter.getValue() + ", Data = " +
data.getData());
        };

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");

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

        t1.join();
        t2.join();

        System.out.println("Final Counter Value: " + counter.getValue());
    }
}
```