Data Structure and Operations: Day-18(The Knight's Tour Problem, Rat in a Maze, N Queen Problem):

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

```
package wipro.com.assignment13;
public class TwoThreadsPrinting {
    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();
    }

    // Runnable class to print numbers from 1 to 10 with a 1-second delay
    private static class NumberPrinter implements Runnable {
        private final 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) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Output:

```
Thread 2: 1
Thread 1: 1
Thread 2: 2
Thread 1: 2
Thread 2: 3
Thread 1: 3
Thread 1: 4
Thread 2: 4
```

```
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 2: 7
Thread 1: 7
Thread 2: 8
Thread 1: 8
Thread 2: 9
Thread 1: 9
Thread 2: 10
Thread 1: 10
```

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..

```
package wipro.com.assignment13;
public class ThreadLifecycleSimulation {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new MyRunnable());

        // State: NEW
        System.out.println("Thread state after creation (NEW): " +
thread.getState());

        // Start the thread
        thread.start();

        // State: RUNNABLE
        System.out.println("Thread state after starting (RUNNABLE): " +
thread.getState());

        // Sleep to demonstrate TIMED_WAITING state
        Thread.sleep(1000);

        // State: TIMED_WAITING
        System.out.println("Thread state after sleeping (TIMED_WAITING): " +
thread.getState());

        synchronized (thread) {
            // Start waiting to demonstrate WAITING state
            thread.wait();

            // State: WAITING
            System.out.println("Thread state after waiting (WAITING): " +
thread.getState());
        }
```

```
        // Join to demonstrate BLOCKED state (waiting for thread to finish)
        thread.join();

        // State: TERMINATED
        System.out.println("Thread state after joining (TERMINATED): " +
thread.getState());
    }

    // Runnable implementation to simulate thread behavior
    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            synchronized (this) {
                // Notify to exit the waiting state
                notify();
            }
        }
    }
}
```

Output:

```
Thread state after creation (NEW): NEW
Thread state after starting (RUNNABLE): RUNNABLE
Thread state after sleeping (TIMED_WAITING): 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.

```
package wipro.com.assignment13;
import java.util.LinkedList;
import java.util.Queue;

public class ProducerConsumer {
    public static void main(String[] args) {
        Queue<Integer> buffer = new LinkedList<>();
        int capacity = 5; // Capacity of the buffer

        Thread producerThread = new Thread(new Producer(buffer, capacity),
"Producer");
        Thread consumerThread = new Thread(new Consumer(buffer), "Consumer");

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

    // Producer class
    static class Producer implements Runnable {
        private final Queue<Integer> buffer;
        private final int capacity;
```

```
        public Producer(Queue<Integer> buffer, int capacity) {
            this.buffer = buffer;
            this.capacity = capacity;
        }

        @Override
        public void run() {
            int item = 0;
            while (true) {
                synchronized (buffer) {
                    // Wait while buffer is full
                    while (buffer.size() == capacity) {
                        try {
                            System.out.println("Buffer is full. Producer is
waiting...");
                            buffer.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }

                    // Produce an item and add it to the buffer
                    System.out.println("Producing item: " + item);
                    buffer.offer(item++);

                    // Notify consumer that buffer has items to consume
                    buffer.notify();

                    // Introduce a delay between productions
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }

    // Consumer class
    static class Consumer implements Runnable {
        private final Queue<Integer> buffer;

        public Consumer(Queue<Integer> buffer) {
            this.buffer = buffer;
        }

        @Override
        public void run() {
            while (true) {
                synchronized (buffer) {
                    // Wait while buffer is empty
                    while (buffer.isEmpty()) {
                        try {
```

```
                            System.out.println("Buffer is empty. Consumer is
waiting...");
                            buffer.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }

                    // Consume an item from the buffer
                    int consumed = buffer.poll();
                    System.out.println("Consuming item: " + consumed);

                    // Notify producer that buffer has space to produce more items
                    buffer.notify();

                    // Introduce a delay between consumption
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

Output:

```
Producing item: 0
Producing item: 1
Consuming item: 0
Consuming item: 1
Buffer is empty. Consumer is waiting...
Producing item: 2
Consuming item: 2
Buffer is empty. Consumer is waiting...
Producing item: 3
Producing item: 4
Producing item: 5
Producing item: 6
Producing item: 7
Buffer is full. Producer is waiting...
Consuming item: 3
Producing item: 8
Buffer is full. Producer is waiting...
Consuming item: 4
Producing item: 9
Consuming item: 5
Consuming item: 6
Producing item: 10
Producing item: 11
Consuming item: 7
```

```
Consuming item: 8
Consuming item: 9
Producing item: 12
Producing item: 13
Consuming item: 10
Consuming item: 11
Consuming item: 12
Consuming item: 13
Buffer is empty. Consumer is waiting...
Producing item: 14
Producing item: 15
Consuming item: 14
Consuming item: 15
Producing item: 16
Producing item: 17
Producing item: 18
Producing item: 19
Producing item: 20
Buffer is full. Producer is waiting...
Consuming item: 16
Consuming item: 17
Consuming item: 18
Producing item: 21
Producing item: 22
Producing item: 23
Consuming item: 19
Consuming item: 20
Consuming item: 21
Producing item: 24
Producing item: 25
Producing item: 26
Buffer is full. Producer is waiting...
Consuming item: 22
Consuming item: 23
Consuming item: 24
Producing item: 27
Producing item: 28
Consuming item: 25
Consuming item: 26
Producing item: 29
Producing item: 30
Consuming item: 27
Consuming item: 28
Consuming item: 29
Producing item: 31
Producing item: 32
Consuming item: 30
Producing item: 33
Producing item: 34
Consuming item: 31
Consuming item: 32
Consuming item: 33
Producing item: 35
Producing item: 36
Producing item: 37
```

```
Consuming item: 34
Producing item: 38
Producing item: 39
Consuming item: 35
Consuming item: 36
Consuming item: 37
Producing item: 40
Consuming item: 38
Consuming item: 39
Producing item: 41
Producing item: 42
Producing item: 43
Consuming item: 40
Consuming item: 41
Producing item: 44
Producing item: 45
Consuming item: 42
Consuming item: 43
Producing item: 46
Consuming item: 44
Consuming item: 45
Producing item: 47
Producing item: 48
Consuming item: 46
Consuming item: 47
Consuming item: 48
Producing item: 49
Producing item: 50
Producing item: 51
Consuming item: 49
Consuming item: 50
Consuming item: 51
Buffer is empty. Consumer is waiting...
Producing item: 52
Consuming item: 52
Buffer is empty. Consumer is waiting...
Producing item: 53
```

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.

```java
package wipro.com.assignment13;

public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
```

```
        this.balance = initialBalance;
    }

    // Synchronized method to deposit money into the account
    public synchronized void deposit(double amount) {
        System.out.println("Depositing: " + amount);
        balance += amount;
        System.out.println("New Balance after deposit: " + balance);
    }

    // Synchronized method to withdraw money from the account
    public synchronized void withdraw(double amount) {
        if (balance >= amount) {
            System.out.println("Withdrawing: " + amount);
            balance -= amount;
            System.out.println("New Balance after withdrawal: " + balance);
        } else {
            System.out.println("Insufficient funds for withdrawal: " + amount);
        }
    }

    // Getter method to retrieve the current balance
    public synchronized double getBalance() {
        return balance;
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.0);

        // Create multiple threads to simulate transactions
        Thread thread1 = new Thread(() -> {
            account.deposit(500.0);
            account.withdraw(200.0);
        });

        Thread thread2 = new Thread(() -> {
            account.deposit(100.0);
            account.withdraw(300.0);
        });

        // Start the threads
        thread1.start();
        thread2.start();

        // Wait for threads to complete
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

```
}
```

Output:

```
Depositing: 500.0
New Balance after deposit: 1500.0
Withdrawing: 200.0
New Balance after withdrawal: 1300.0
Depositing: 100.0
New Balance after deposit: 1400.0
Withdrawing: 300.0
New Balance after withdrawal: 1100.0
Final Balance: 1100.0
```

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.

```
package wipro.com.assignment13;

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

public class ThreadPoolExample {
    public static void main(String[] args) {
        // Create a fixed-size thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks to the thread pool
        for (int i = 1; i <= 5; i++) {
            executor.submit(new Task("Task " + i));
        }

        // Shutdown the executor
        executor.shutdown();
    }

    // Task class representing a task to be executed by a thread
    static class Task implements Runnable {
        private final String taskName;

        public Task(String taskName) {
            this.taskName = taskName;
        }

        @Override
        public void run() {
            try {
                // Simulate some complex calculation or I/O operation
```

```
                System.out.println(taskName + " is executing on thread: " +
Thread.currentThread().getName());
                Thread.sleep(2000); // Simulate time-consuming task
                System.out.println(taskName + " has completed.");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Output:

```
Task 3 is executing on thread: pool-1-thread-3
Task 1 is executing on thread: pool-1-thread-1
Task 2 is executing on thread: pool-1-thread-2
Task 3 has completed.
Task 1 has completed.
Task 4 is executing on thread: pool-1-thread-1
Task 5 is executing on thread: pool-1-thread-3
Task 2 has completed.
Task 5 has completed.
Task 4 has completed.
```

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 wipro.com.assignment13;
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;

public class PrimeNumberExecutorAsyncIO {
    public static void main(String[] args) {
        int maxNumber = 1000; // Max number up to which prime numbers are calculated

        // Create a fixed-size thread pool with 4 threads
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Use CompletableFuture to handle async file writing
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            List<Integer> primeNumbers = calculatePrimes(maxNumber, executor);
            writeToFile(primeNumbers, "prime_numbers.txt");
```

```
        }, executor);

        // Shutdown the executor
        executor.shutdown();

        // Wait for CompletableFuture to complete
        future.join();
    }

    // Method to calculate prime numbers up to a given number
    private static List<Integer> calculatePrimes(int maxNumber, ExecutorService
executor) {
        List<Integer> primes = new ArrayList<>();

        // Submit tasks to executor to find prime numbers in parallel
        List<CompletableFuture<Void>> futures = new ArrayList<>();
        for (int number = 2; number <= maxNumber; number++) {
            final int num = number;
            CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
                if (isPrime(num)) {
                    synchronized (primes) {
                        primes.add(num);
                    }
                }
            }, executor);
            futures.add(future);
        }

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

        return primes;
    }

    // Method to check if a number is prime
    private static boolean isPrime(int number) {
        if (number <= 1) return false;
        if (number <= 3) return true;
        if (number % 2 == 0 || number % 3 == 0) return false;
        int i = 5;
        while (i * i <= number) {
            if (number % i == 0 || number % (i + 2) == 0)
                return false;
            i += 6;
        }
        return true;
    }

    // Method to write prime numbers to a file
    private static void writeToFile(List<Integer> primeNumbers, String filename) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            for (int prime : primeNumbers) {
                writer.write(String.valueOf(prime));
                writer.newLine();
            }
```

```
            System.out.println("Prime numbers have been written to " + filename);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

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.

```
package wipro.com.assignment13;
public class Counter {
    private int count;

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

    // Synchronized method to increment the counter
    public synchronized void increment() {
        count++;
    }

    // Synchronized method to decrement the counter
    public synchronized void decrement() {
        count--;
    }

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