Day 18: **Manisha assignment**

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

Explanation of Output:

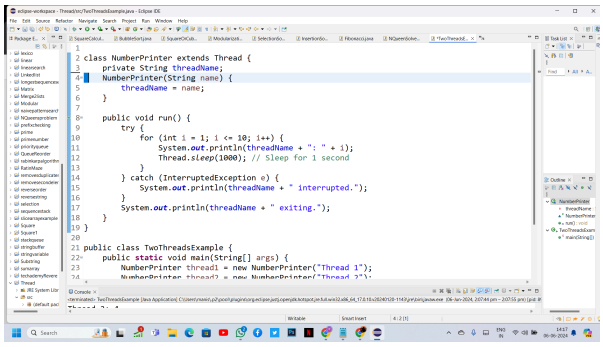
- The output is an 8x8 board where 'Q' represents the position of queens. The queens are placed such that no two queens can attack each other.

This code can be run in Eclipse or any other Java development environment. If you encounter any errors, please provide the error details so I can help resolve them.

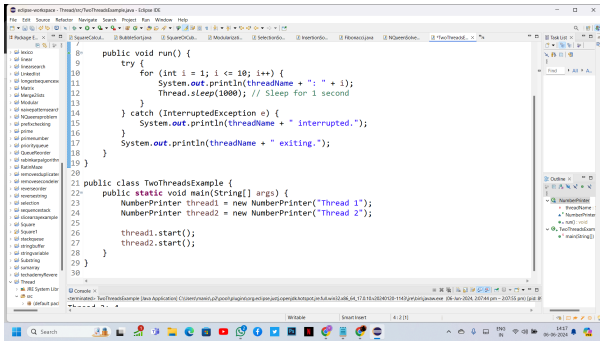
Here is a simple Java program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number.

Explanation of Output:

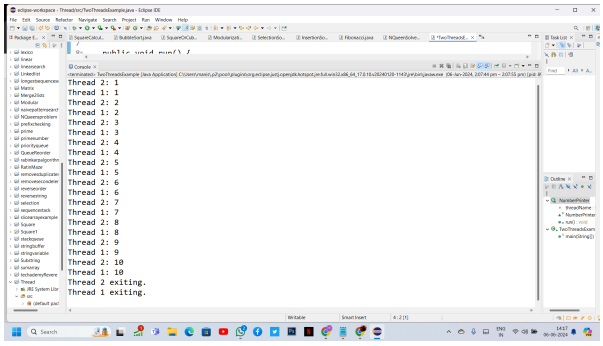
- Both threads print numbers from 1 to 10, with a 1-second delay between each number.
- The output shows the numbers printed by each thread. Note that the exact order may vary slightly due to the concurrent execution of threads, but each thread will print its sequence independently



```
class NumberPrinter extends Thread {
    private String threadName;
    NumberPrinter(String name) {
        threadName = name;
    }

    public void run() {
        try {
            for (int i = 1; i <= 10; i++) {
                System.out.println(threadName + ": " + i);
                Thread.sleep(1000); // Sleep for 1 second
            }
        } catch (InterruptedException e) {
            System.out.println(threadName + " interrupted.");
        }
        System.out.println(threadName + " exiting.");
    }
}

public class TwoThreadsExample {
    public static void main(String[] args) {
        NumberPrinter thread1 = new NumberPrinter("Thread 1");
```



```
    public void run() {
        try {
            for (int i = 1; i <= 10; i++) {
                System.out.println(threadName + ": " + i);
                Thread.sleep(1000); // Sleep for 1 second
            }
        } catch (InterruptedException e) {
            System.out.println(threadName + " interrupted.");
        }
        System.out.println(threadName + " exiting.");
    }
}

public class TwoThreadsExample {
    public static void main(String[] args) {
        NumberPrinter thread1 = new NumberPrinter("Thread 1");
        NumberPrinter thread2 = new NumberPrinter("Thread 2");

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



```
Thread 2: 1
Thread 1: 1
Thread 2: 2
Thread 1: 2
Thread 2: 3
Thread 1: 3
Thread 2: 4
Thread 1: 4
Thread 2: 5
Thread 1: 5
Thread 2: 6
Thread 1: 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
Thread 2 exiting.
Thread 1 exiting.
```

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..

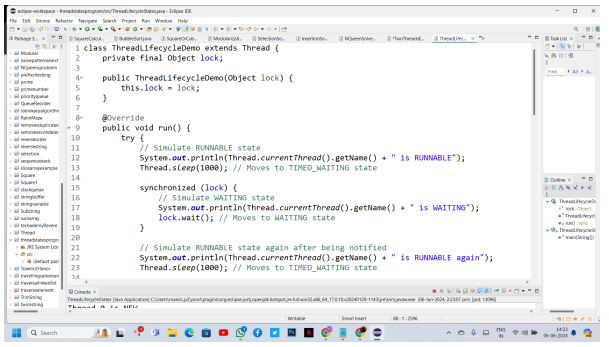
Explanation:

1. ThreadLifecycleDemo Class:

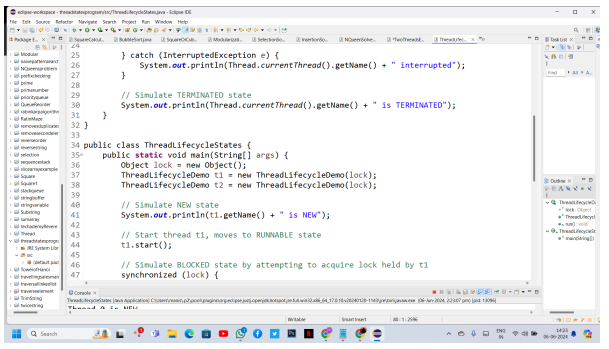
- **Extends the `Thread` class.**
- **Uses a shared `lock` object to demonstrate synchronization and waiting.**
- **In the `run` method:**
  - **Initially prints the `RUNNABLE` state.**
  - **Sleeps for 1 second to move to the `TIMED_WAITING` state.**
  - **Enters a synchronized block and calls `wait()` on the `lock` to move to the `WAITING` state.**
  - **Once notified, prints `RUNNABLE` again and sleeps for another second.**
  - **Finally, prints `TERMINATED` state when the `run` method completes.**

2. ThreadLifecycleStates Class:

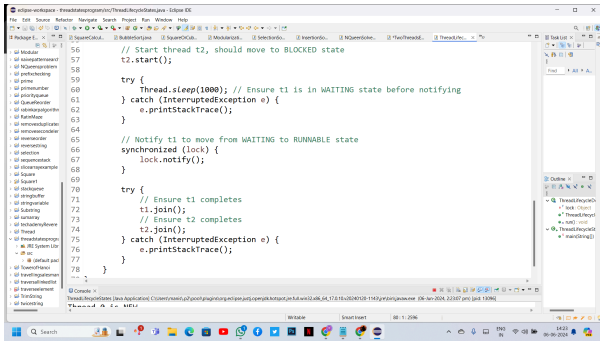
- **Creates a shared `lock` object and two instances of `ThreadLifecycleDemo`.**
- **Prints `NEW` state for thread `t1`.**
- **Starts thread `t1`, which moves it to the `RUNNABLE` state.**
- **Main thread acquires the `lock` and sleeps for 2 seconds to simulate `BLOCKED` state for `t2`.**
- **Starts thread `t2`, which attempts to acquire the lock and gets `BLOCKED`.**
- **After 1 second, main thread notifies `t1`, moving it from `WAITING` to `RUNNABLE` state.**
- **Calls `join` on both threads to ensure they complete before main thread terminates.**



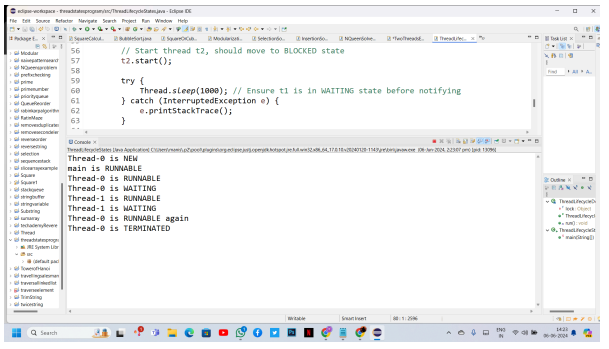
```
class ThreadLifecycleDemo extends Thread {
    private final Object lock;

    public ThreadLifecycleDemo(Object lock) {
        this.lock = lock;
    }

    @Override
    public void run() {
        try {
            // Simulate RUNNABLE state
            System.out.println(Thread.currentThread().getName() + " is RUNNABLE");
            Thread.sleep(1000); // Moves to TIMED_WAITING state

            synchronized (lock) {
                // Simulate WAITING state
                System.out.println(Thread.currentThread().getName() + " is WAITING");
                lock.wait(); // Moves to WAITING state
            }

            // Simulate RUNNABLE state again after being notified
            System.out.println(Thread.currentThread().getName() + " is RUNNABLE again");
            Thread.sleep(1000); // Moves to TIMED_WAITING state
```



```
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " interrupted");
        }

        // Simulate TERMINATED state
        System.out.println(Thread.currentThread().getName() + " is TERMINATED");
    }
}

public class ThreadLifecycleStates {
    public static void main(String[] args) {
        Object lock = new Object();
        ThreadLifecycleDemo t1 = new ThreadLifecycleDemo(lock);
        ThreadLifecycleDemo t2 = new ThreadLifecycleDemo(lock);

        // Simulate NEW state
        System.out.println(t1.getName() + " is NEW");

        // Start thread t1, moves to RUNNABLE state
        t1.start();

        // Simulate BLOCKED state by attempting to acquire lock held by t1
        synchronized (lock) {
```



```
        // Start thread t2, should move to BLOCKED state
        t2.start();

        try {
            Thread.sleep(1000); // Ensure t1 is in WAITING state before notifying
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Notify t1 to move from WAITING to RUNNABLE state
        synchronized (lock) {
            lock.notify();
        }

        try {
            // Ensure t1 completes
            t1.join();
            // Ensure t2 completes
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```



```
        // Start thread t2, should move to BLOCKED state
        t2.start();

        try {
            Thread.sleep(1000); // Ensure t1 is in WAITING state before notifying
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

Thread-0 is NEW
main is RUNNABLE
Thread-0 is RUNNABLE
Thread-0 is WAITING
Thread-1 is RUNNABLE
Thread-1 is WAITING
Thread-0 is RUNNABLE again
Thread-0 is 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.

Explanation:

1. SharedBuffer Class:

- This class represents the shared buffer where the producer will produce items and the consumer will consume items.
- It has an integer array `buffer`, an integer `count` to track the number of items, and an integer `size` to define the buffer size.
- The `produce` method:
  - Synchronized to ensure only one thread can produce at a time.
  - Uses `wait()` if the buffer is full, preventing overproduction.
  - Adds a produced item to the buffer and increments the count.
  - Calls `notify()` to wake up any waiting consumer.

2. Producer Class:

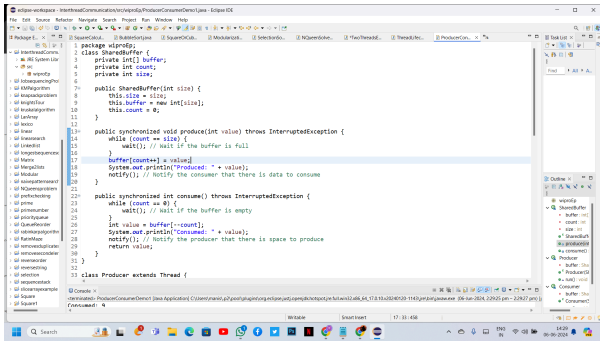
- Extends the `Thread` class.
- In the `run` method, it produces 10 items, calling the `produce` method on the shared buffer and sleeping for 100 milliseconds between productions to simulate production time.

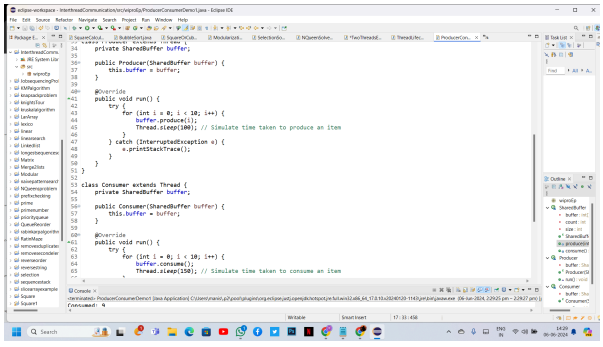
3. Consumer Class:

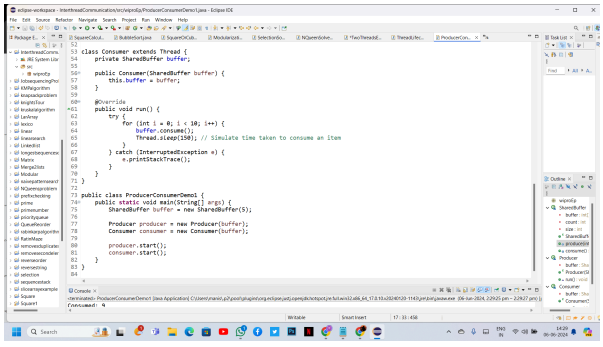
- Extends the `Thread` class.
- In the `run` method, it consumes 10 items, calling the `consume` method on the shared buffer and sleeping for 150 milliseconds between consumptions to simulate consumption time.

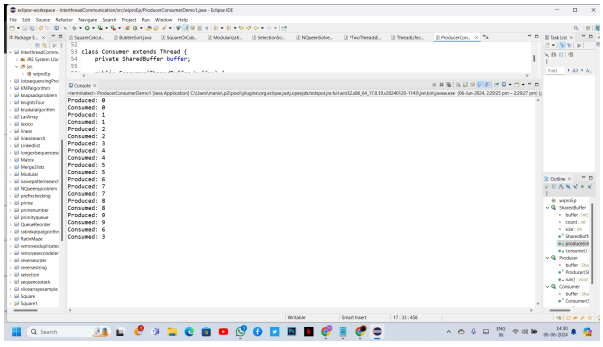
4. ProducerConsumerDemo Class:

- Contains the `main` method.
- Creates a shared buffer with a size of 5.
- Creates and starts one producer and one consumer thread.









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.

Explanation:

1. BankAccount Class:

- The `BankAccount` class represents a bank account with a balance.
- The `deposit` method is synchronized to prevent multiple threads from updating the balance concurrently.
- The `withdraw` method is synchronized to ensure atomic withdrawal operations and to prevent race conditions.

2. DepositThread Class:

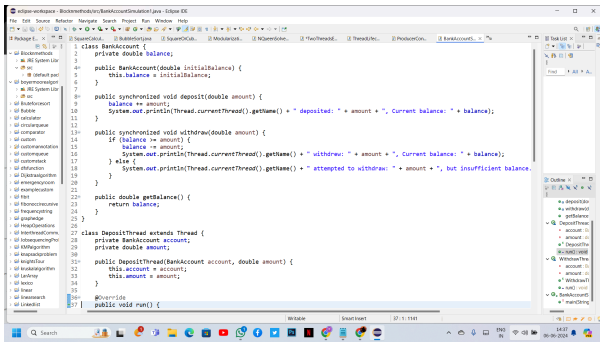
- This class extends the `Thread` class and is used to perform deposit operations on the `BankAccount`.
- The `run` method calls the `deposit` method on the `BankAccount`.

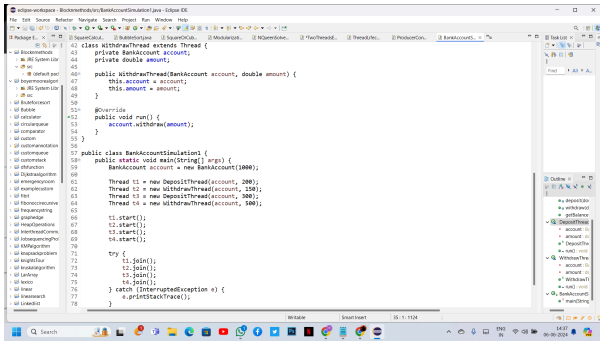
3. WithdrawThread Class:

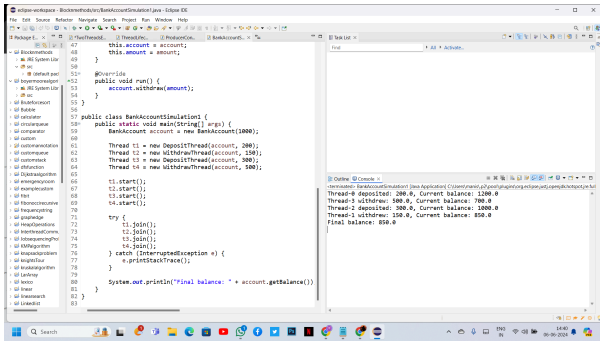
- This class extends the `Thread` class and is used to perform withdrawal operations on the `BankAccount`.
- The `run` method calls the `withdraw` method on the `BankAccount`.

4. BankAccountSimulation Class:

- The `main` method creates a `BankAccount` object with an initial balance of 1000.
- It creates multiple `DepositThread` and `WithdrawThread` objects to perform deposit and withdrawal operations.
- It starts all the threads and waits for them to complete using the `join` method.
- Finally, it prints the final balance of the account.







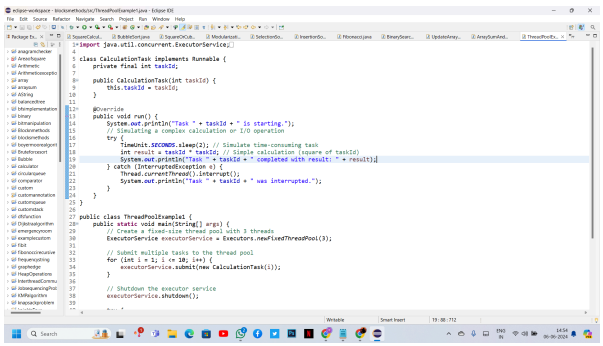
Task 5: Thread Pools and Concurrency Utilities

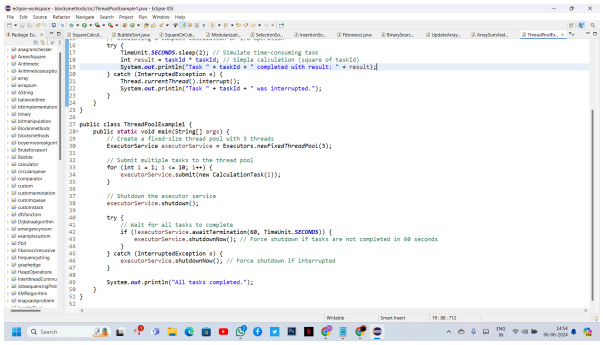
Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

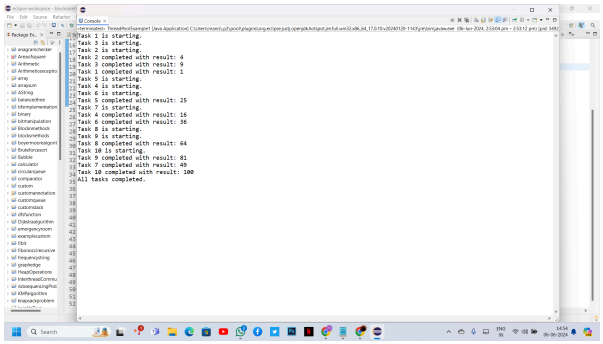
Explanation:

1. CalculationTask Class:
- This class implements the `Runnable` interface and simulates a task that performs a complex calculation or I/O operation.
- The `run` method prints the start of the task, simulates a time-consuming task using `TimeUnit.SECONDS.sleep(2)`, and performs a simple calculation (square of the task ID).
- The result of the calculation is printed, and any interruption is handled by catching `InterruptedException`.

2. ThreadPoolExample Class:
- The `main` method creates a fixed-size thread pool with 3 threads using `Executors.newFixedThreadPool(3)`.
- It submits 10 tasks to the thread pool using a for loop and `executorService.submit(new CalculationTask(i))`.
- The `shutdown` method is called to stop accepting new tasks.
- The `awaitTermination` method waits for all tasks to complete within 60 seconds. If tasks are not completed, it forcefully shuts down the executor service using `shutdownNow`.







Explanation of Output:

- The output shows tasks starting and completing in the order managed by the thread pool.
- Each task prints its start, performs a calculation, and then prints the result.
- The thread pool allows multiple tasks to run concurrently up to the fixed size (3 in this case), ensuring efficient utilization of threads.
- Finally, a message "All tasks completed." confirms that all tasks have finished executing.

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.

Explanation:

1. PrimeCalculator Class:

- `isPrime(int number)`: Checks if a given number is prime.

2. PrimeNumberFinder Class:

- Sets up an `ExecutorService` with a fixed thread pool of size 4.

- Uses a list of `CompletableFuture<Integer>` to store the results of prime number calculations.

- For each number from 2 to the upper limit (100 in this case), it creates a `CompletableFuture` that checks if the number is prime using `PrimeCalculator`.

- `CompletableFuture.allOf(futureList.toArray(new CompletableFuture[0]))`: Waits for all futures to complete.

- Once all tasks are completed, it collects the prime numbers and writes them to a file asynchronously.

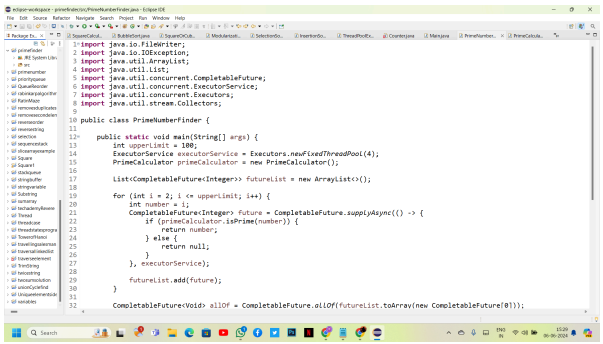
- Finally, the executor service is shut down.

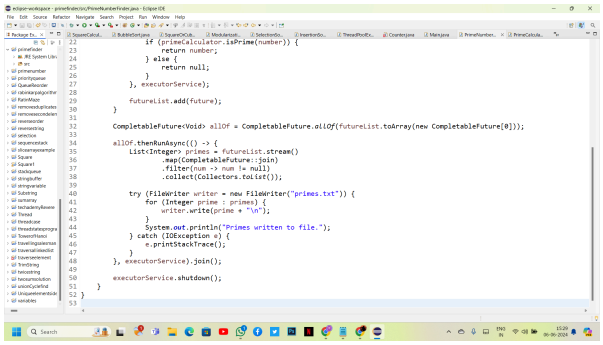
1. Ensure Proper Imports:

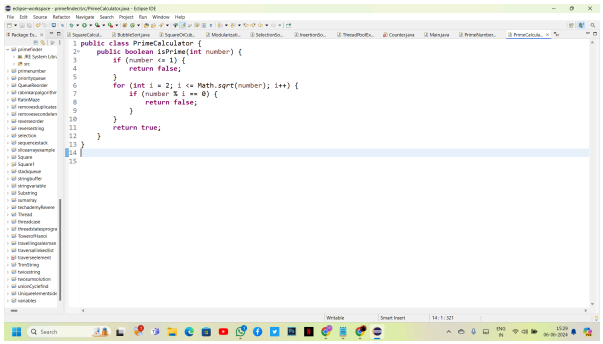
- Ensure you have imported the necessary classes like `FileWriter`, `IOException`, `List`, `ArrayList`, `CompletableFuture`, `ExecutorService`, `Executors`, `Collectors`, etc.

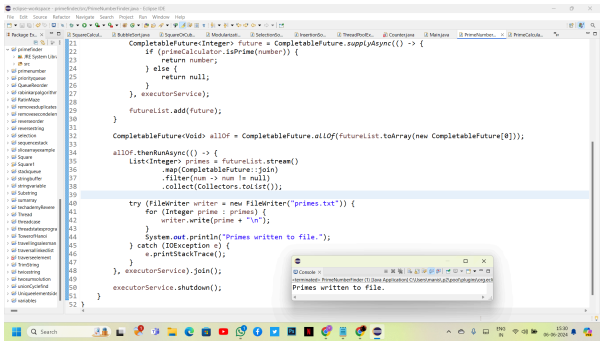
2. File Location:

- The `primes.txt` file will be created in the project's root directory. Ensure you have write permissions in this directory.









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.

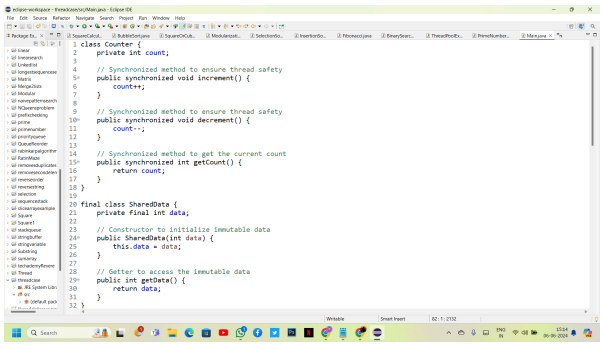
Explanation:

1. Counter Class:
- **increment() Method: Increments the count. This method is synchronized to ensure thread safety.**
- **decrement() Method: Decrements the count. This method is synchronized to ensure thread safety.**
- **getCount() Method: Returns the current count. This method is synchronized to ensure thread safety.**

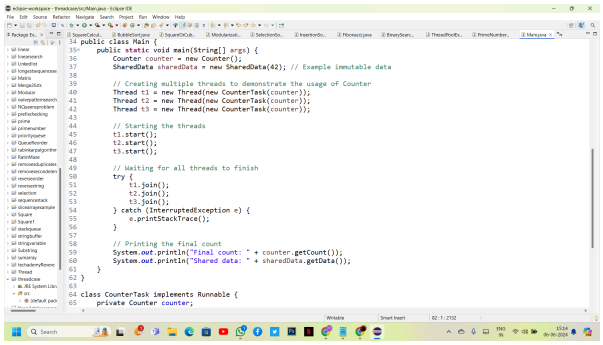
2. SharedData Class:
- **A final class with a single immutable field `data`.**
- **Constructor: Initializes the `data` field.**
- **getData() Method**: Returns the value of `data`.**

3. Main Class:
- **Creates an instance of `Counter` and `SharedData`.**
- **Creates three threads, each running a `CounterTask`.**
- **Starts the threads and waits for them to finish using `join()`.**
- **Prints the final count and the shared data.**

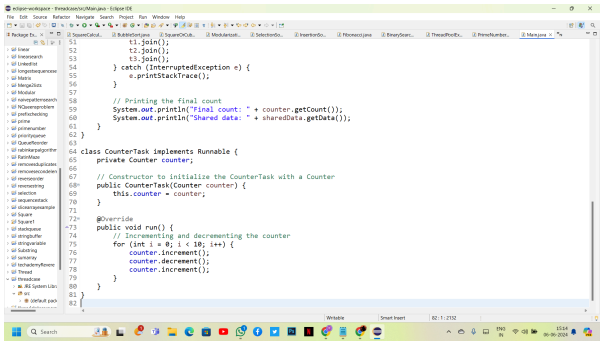
4. CounterTask Class:
- **Implements `Runnable`.**
- **Takes a `Counter` instance as a parameter.**
- **In the `run` method, increments and decrements the counter to simulate some work.**



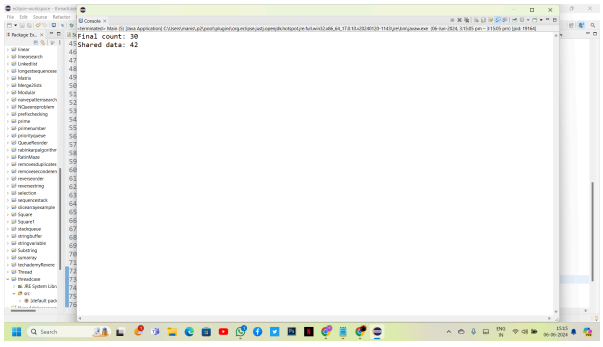
```
class Counter {
    private int count;

    // Synchronized method to ensure thread safety
    public synchronized void increment() {
        count++;
    }

    // Synchronized method to ensure thread safety
    public synchronized void decrement() {
        count--;
    }

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

final class SharedData {
    private final int data;

    // Constructor to initialize immutable data
    public SharedData(int data) {
        this.data = data;
    }

    // Getter to access the immutable data
    public int getData() {
        return data;
    }
}
```



```
public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        SharedData sharedData = new SharedData(42); // Example immutable data

        // Creating multiple threads to demonstrate the usage of Counter
        Thread t1 = new Thread(new CounterTask(counter));
        Thread t2 = new Thread(new CounterTask(counter));
        Thread t3 = new Thread(new CounterTask(counter));

        // Starting the threads
        t1.start();
        t2.start();
        t3.start();

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

        // Printing the final count
        System.out.println("Final count: " + counter.getCount());
        System.out.println("Shared data: " + sharedData.getData());
    }
}

class CounterTask implements Runnable {
    private Counter counter;
```



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

        // Printing the final count
        System.out.println("Final count: " + counter.getCount());
        System.out.println("Shared data: " + sharedData.getData());
    }
}

class CounterTask implements Runnable {
    private Counter counter;

    // Constructor to initialize the CounterTask with a Counter
    public CounterTask(Counter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        // Incrementing and decrementing the counter
        for (int i = 0; i < 10; i++) {
            counter.increment();
            counter.decrement();
            counter.increment();
        }
    }
}
```



```
Final count: 30
Shared data: 42
```