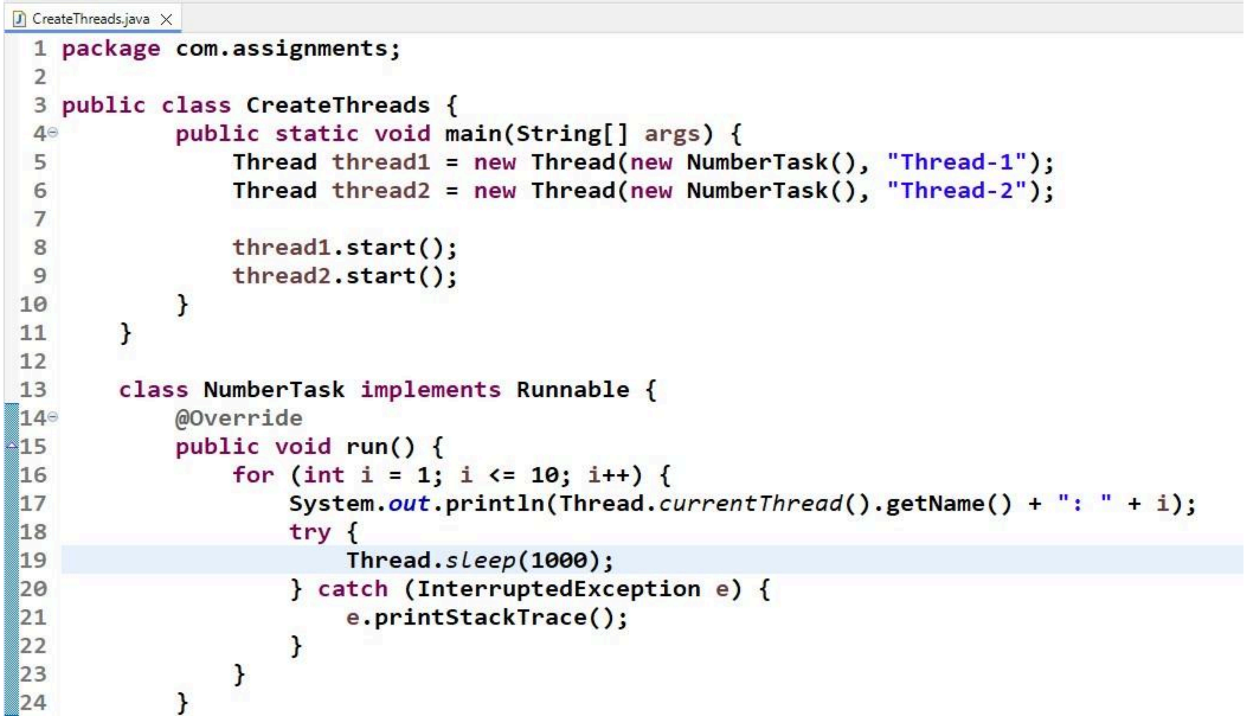
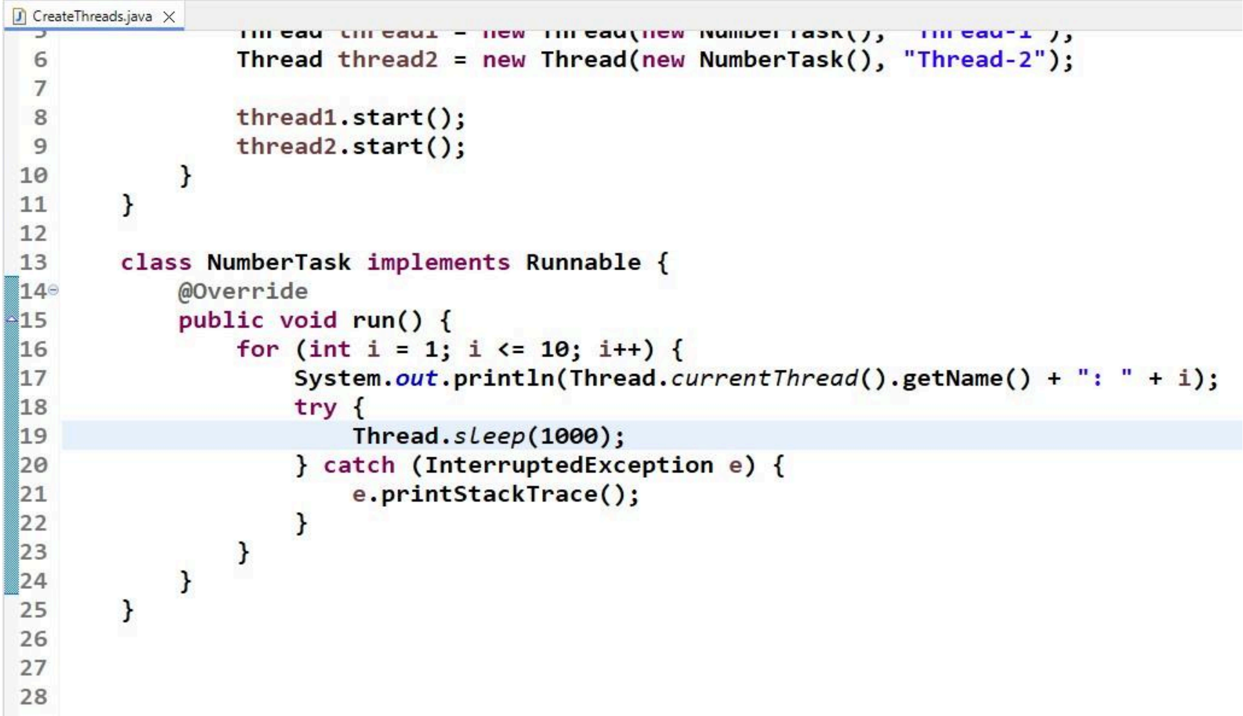
# Day 18

**1]** 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.

Solution:-
Code -

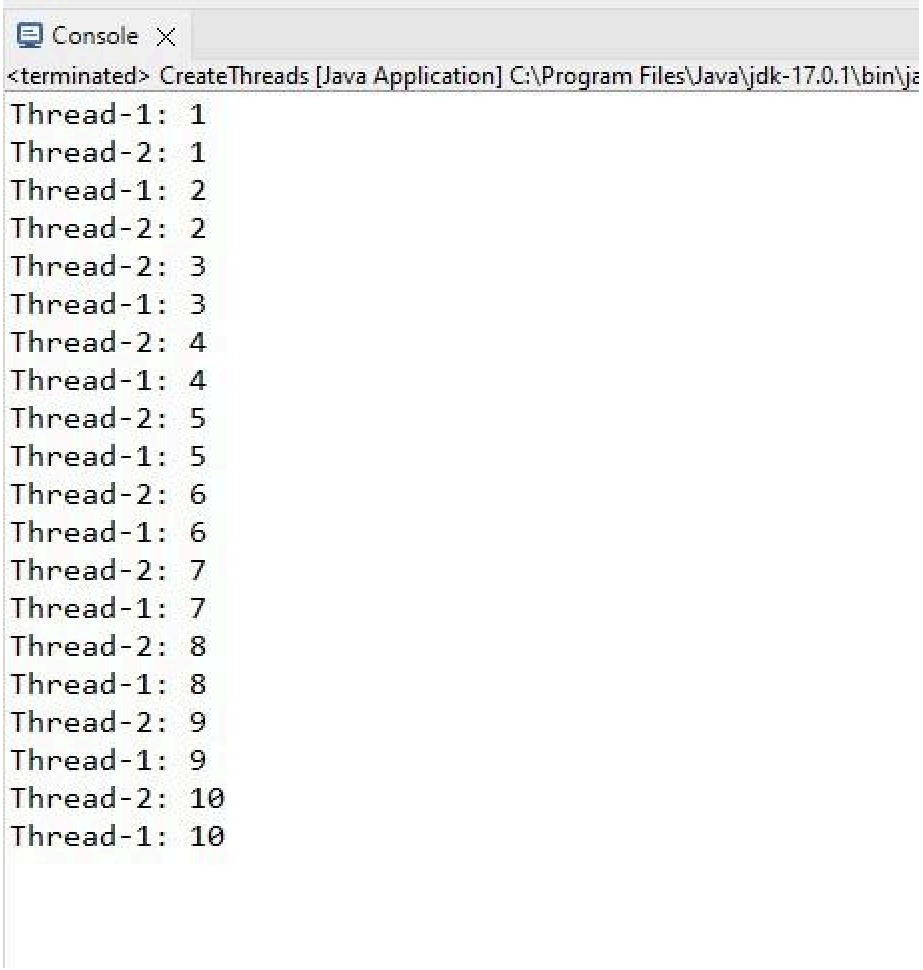


```java
package com.assignments;

public class CreateThreads {
    public static void main(String[] args) {
      Thread thread1 = new Thread(new NumberTask(), "Thread-1");
      Thread thread2 = new Thread(new NumberTask(), "Thread-2");

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

  class NumberTask implements Runnable {
    @Override
    public void run() {
      for (int i = 1; i <= 10; i++) {
        System.out.println(Thread.currentThread().getName() + ": " + i);
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }
```



```
        Thread thread1 = new Thread(new NumberTask(), "Thread-1");
        Thread thread2 = new Thread(new NumberTask(), "Thread-2");

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

class NumberTask implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Output -

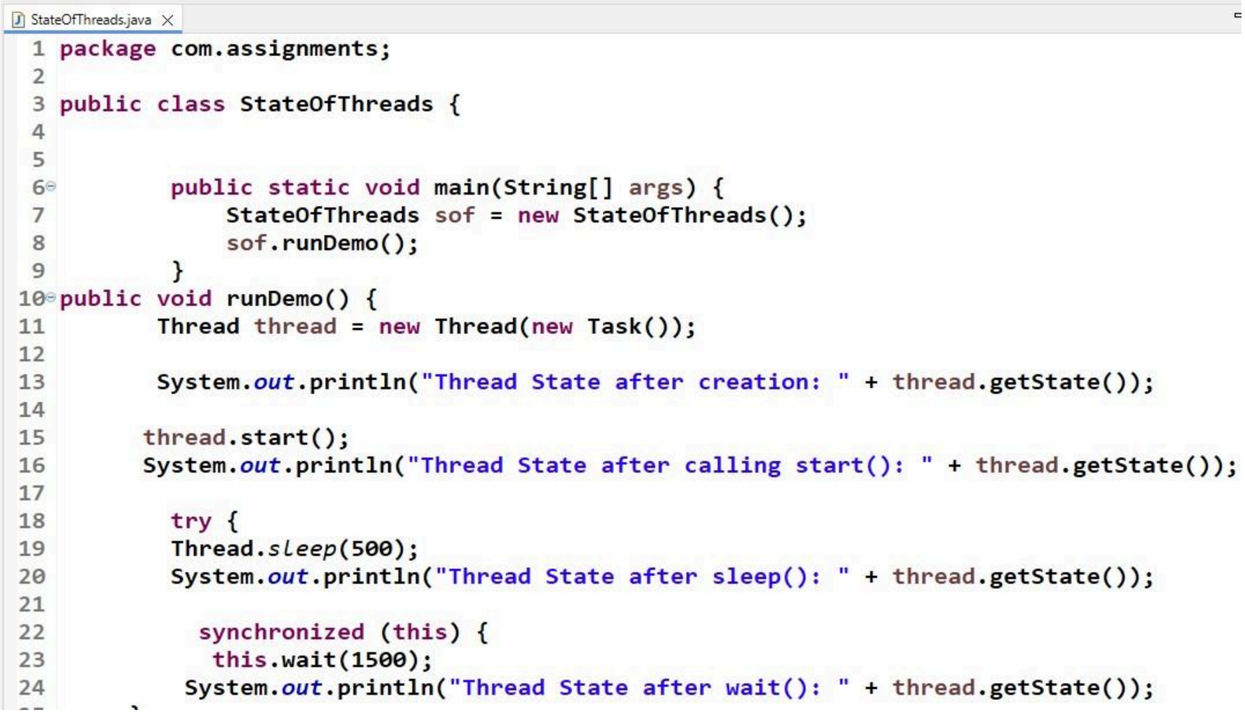


Console
<terminated> CreateThreads [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\ja

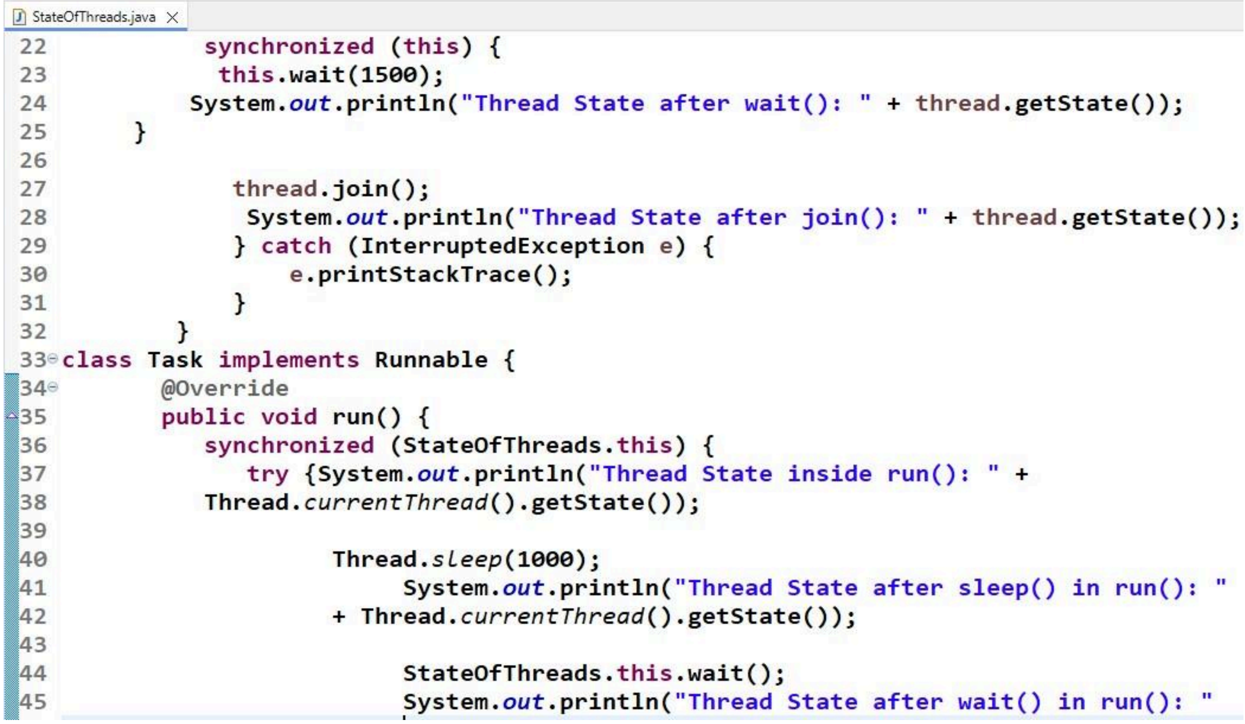
```
Thread-1: 1
Thread-2: 1
Thread-1: 2
Thread-2: 2
Thread-2: 3
Thread-1: 3
Thread-2: 4
Thread-1: 4
Thread-2: 5
Thread-1: 5
Thread-2: 6
Thread-1: 6
Thread-2: 7
Thread-1: 7
Thread-2: 8
Thread-1: 8
Thread-2: 9
Thread-1: 9
Thread-2: 10
Thread-1: 10
```

2] 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.

Solution:-
Code -

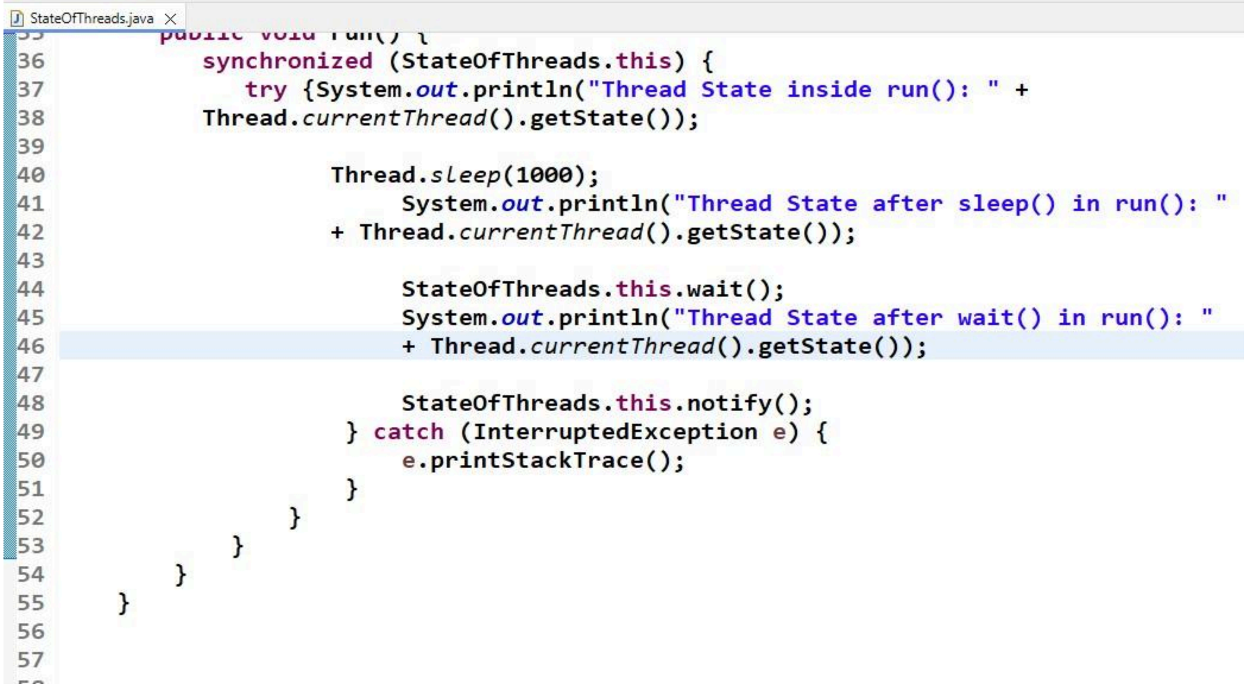


```java
package com.assignments;

public class StateOfThreads {


        public static void main(String[] args) {
            StateOfThreads sof = new StateOfThreads();
            sof.runDemo();
        }
public void runDemo() {
       Thread thread = new Thread(new Task());

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

      thread.start();
      System.out.println("Thread State after calling start(): " + thread.getState());

        try {
        Thread.sleep(500);
        System.out.println("Thread State after sleep(): " + thread.getState());

          synchronized (this) {
           this.wait(1500);
         System.out.println("Thread State after wait(): " + thread.getState());
```



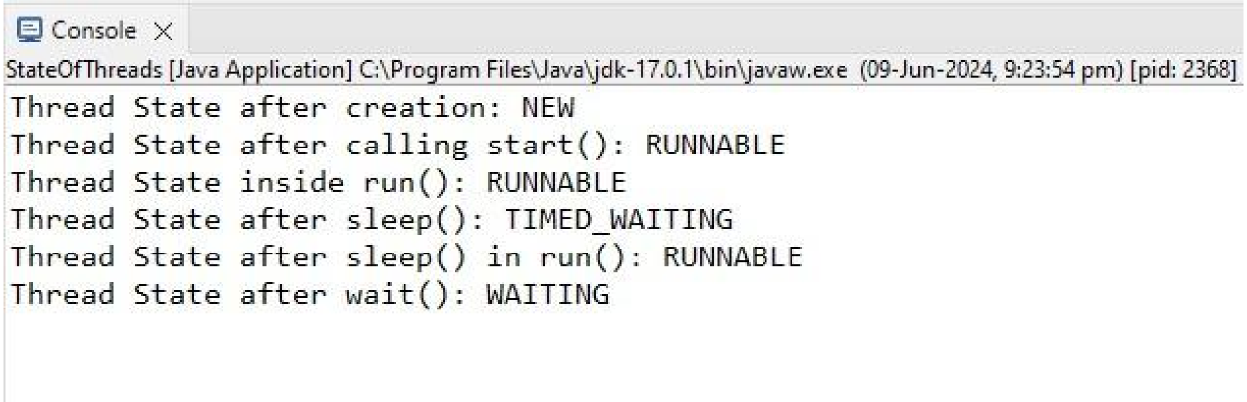
```java
          synchronized (this) {
           this.wait(1500);
         System.out.println("Thread State after wait(): " + thread.getState());
```

```java
     }

            thread.join();
             System.out.println("Thread State after join(): " + thread.getState());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
class Task implements Runnable {
       @Override
       public void run() {
          synchronized (StateOfThreads.this) {
             try {System.out.println("Thread State inside run(): " +
          Thread.currentThread().getState());

                   Thread.sleep(1000);
                        System.out.println("Thread State after sleep() in run(): "
                   + Thread.currentThread().getState());

                        StateOfThreads.this.wait();
                        System.out.println("Thread State after wait() in run(): "
```



```
       public void run() {
          synchronized (StateOfThreads.this) {
             try {System.out.println("Thread State inside run(): " +
          Thread.currentThread().getState());

                   Thread.sleep(1000);
                        System.out.println("Thread State after sleep() in run(): "
                   + Thread.currentThread().getState());

                        StateOfThreads.this.wait();
                        System.out.println("Thread State after wait() in run(): "
                        + Thread.currentThread().getState());

                        StateOfThreads.this.notify();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

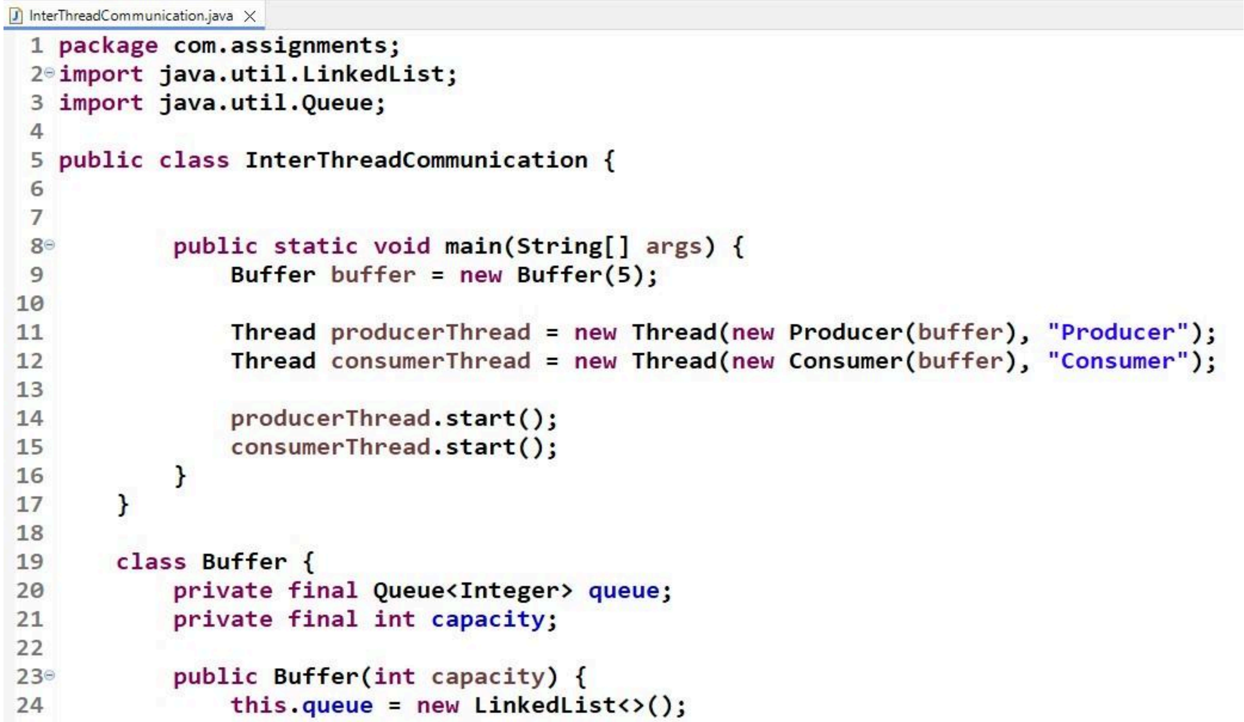
Output -



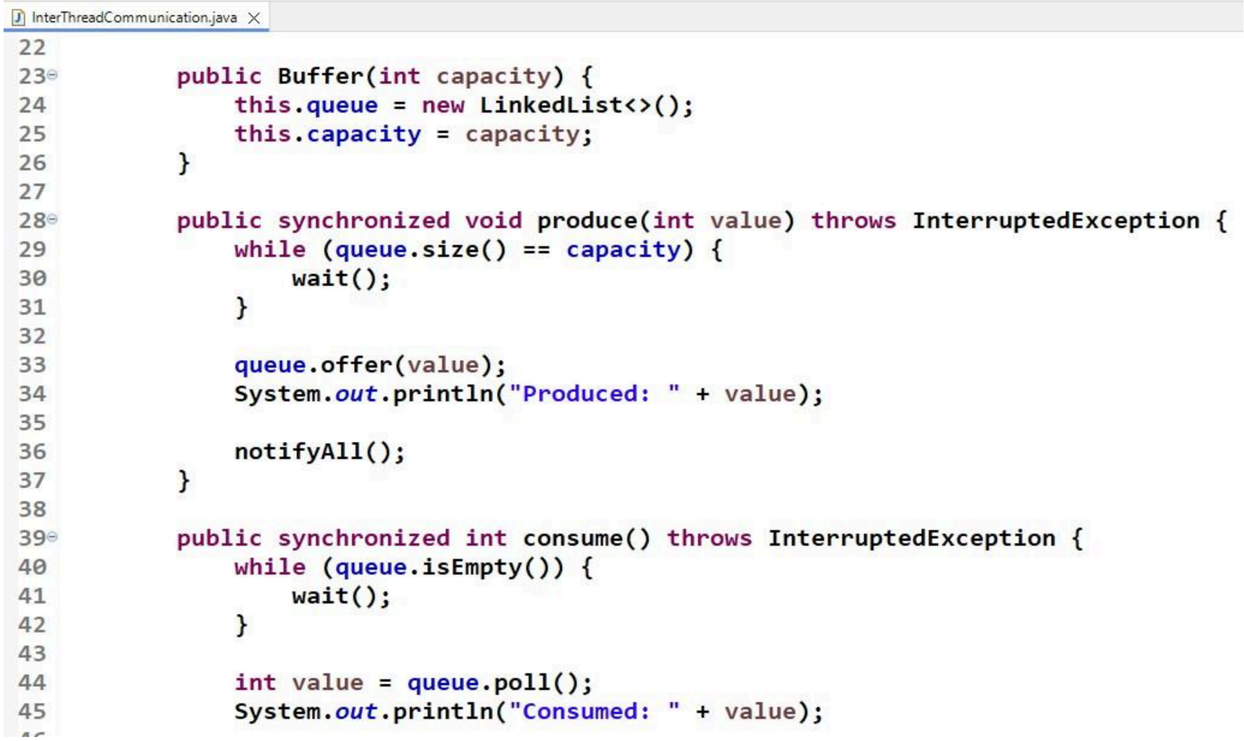
```
StateOfThreads [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe  (09-Jun-2024, 9:23:54 pm) [pid: 2368]
Thread State after creation: NEW
Thread State after calling start(): RUNNABLE
Thread State inside run(): RUNNABLE
Thread State after sleep(): TIMED_WAITING
Thread State after sleep() in run(): RUNNABLE
Thread State after wait(): WAITING
```

3] 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.

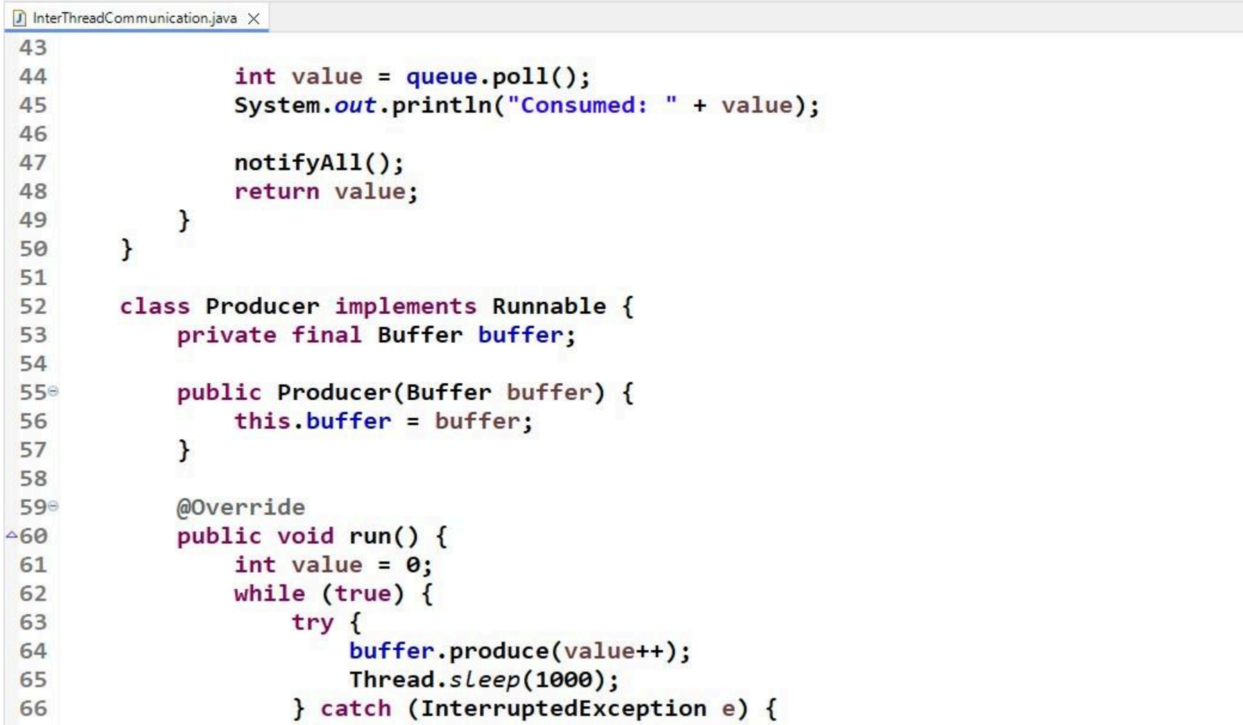
Solution:-
Code -



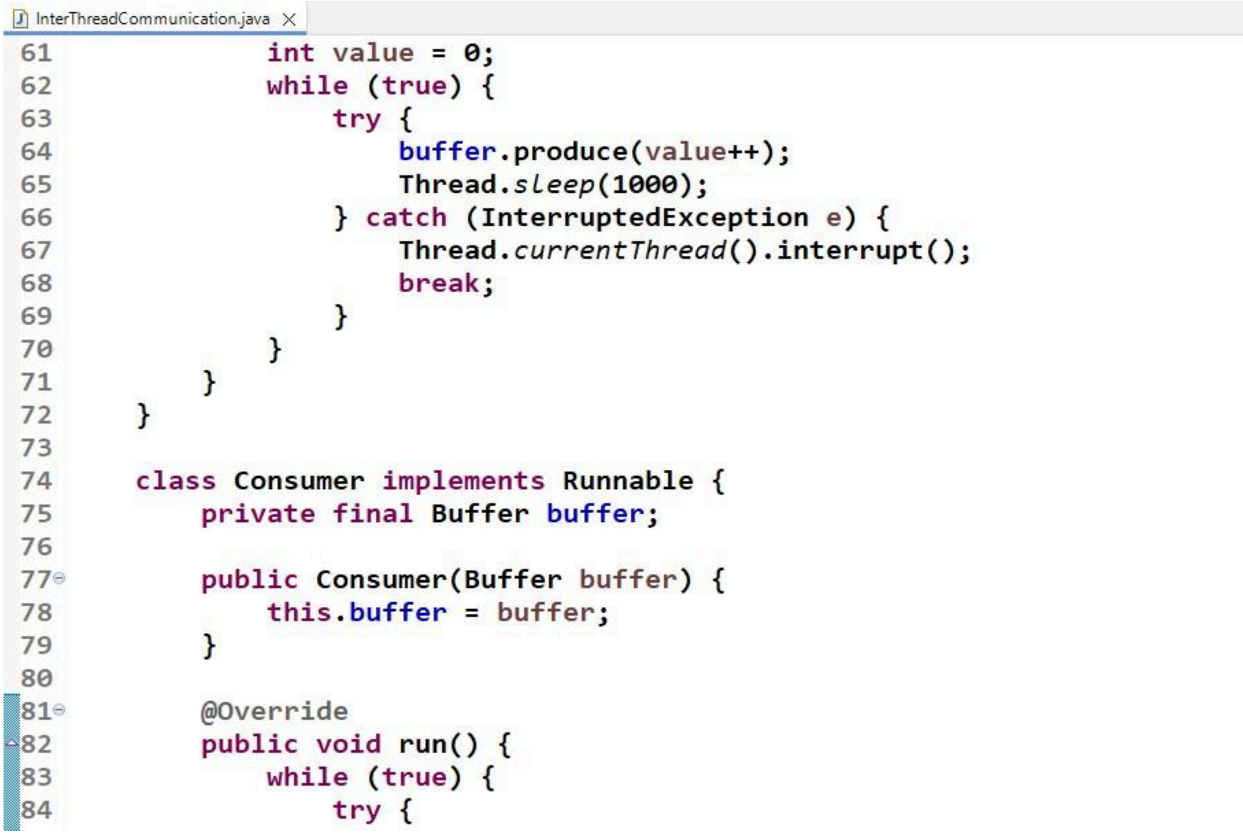
```java
package com.assignments;
import java.util.LinkedList;
import java.util.Queue;

public class InterThreadCommunication {


    public static void main(String[] args) {
        Buffer buffer = new Buffer(5);

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

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

  class Buffer {
    private final Queue<Integer> queue;
    private final int capacity;

    public Buffer(int capacity) {
        this.queue = new LinkedList<>();
```



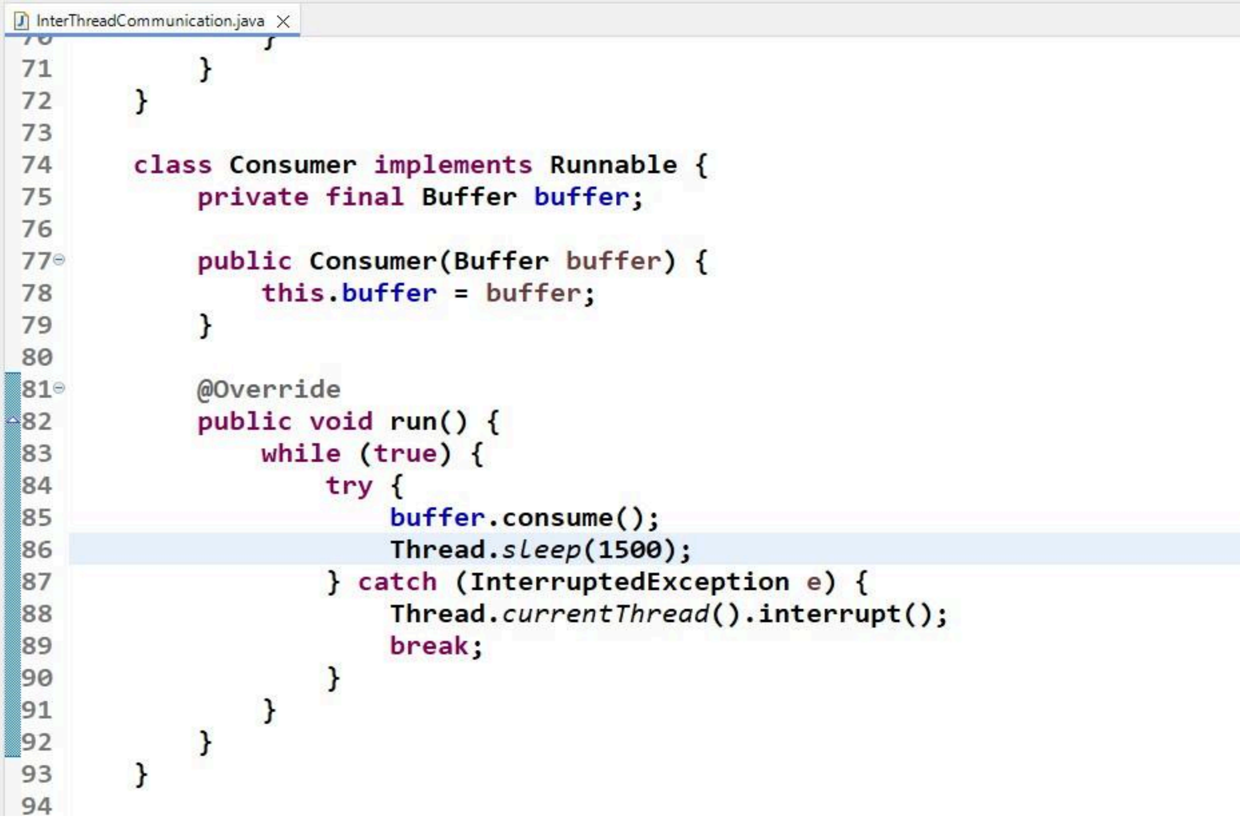
```java

    public Buffer(int capacity) {
        this.queue = new LinkedList<>();
        this.capacity = capacity;
    }

    public synchronized void produce(int value) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }

        queue.offer(value);
        System.out.println("Produced: " + value);

        notifyAll();
    }

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

        int value = queue.poll();
        System.out.println("Consumed: " + value);
```



```java

            int value = queue.poll();
            System.out.println("Consumed: " + value);

            notifyAll();
            return value;
        }
    }

    class Producer implements Runnable {
        private final Buffer buffer;

        public Producer(Buffer buffer) {
            this.buffer = buffer;
        }

        @Override
        public void run() {
            int value = 0;
            while (true) {
                try {
                    buffer.produce(value++);
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
```

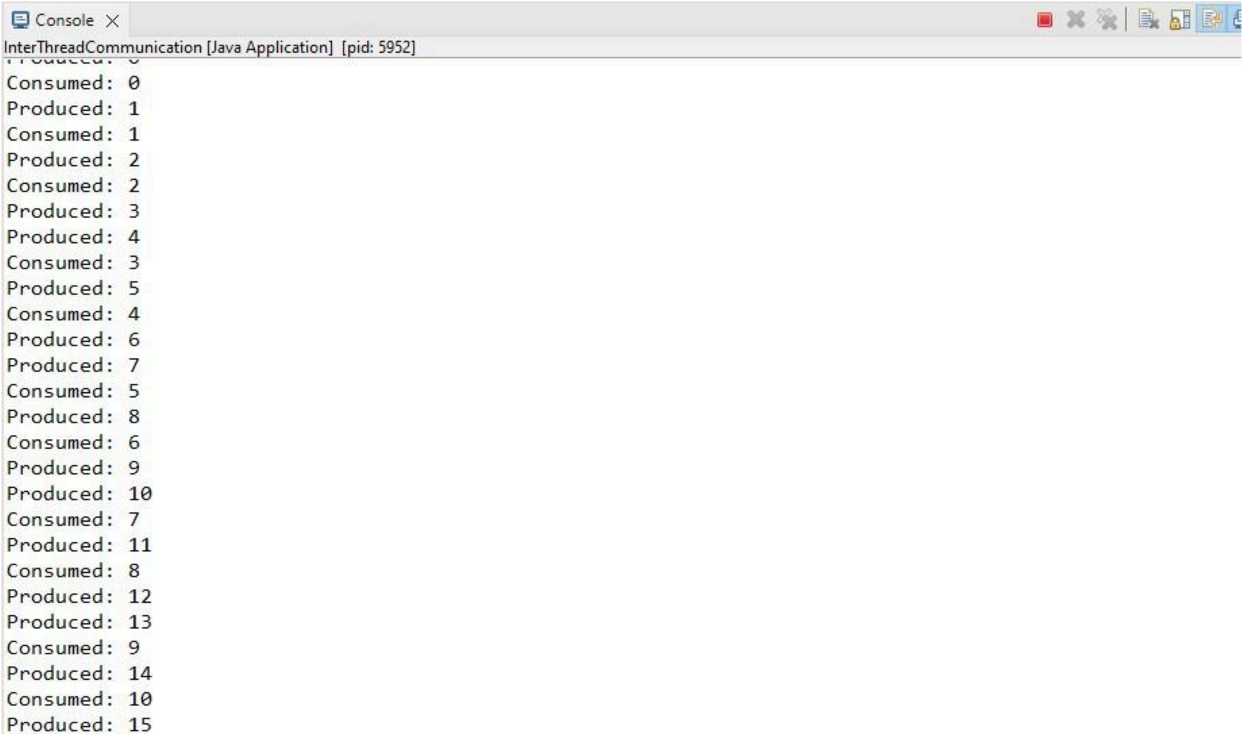


```java
            int value = 0;
            while (true) {
                try {
                    buffer.produce(value++);
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        }
    }

    class Consumer implements Runnable {
        private final Buffer buffer;

        public Consumer(Buffer buffer) {
            this.buffer = buffer;
        }

        @Override
        public void run() {
            while (true) {
                try {
```



```java
        }
    }

    class Consumer implements Runnable {
        private final Buffer buffer;

        public Consumer(Buffer buffer) {
            this.buffer = buffer;
        }

        @Override
        public void run() {
            while (true) {
                try {
                    buffer.consume();
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        }
    }

```

Output -



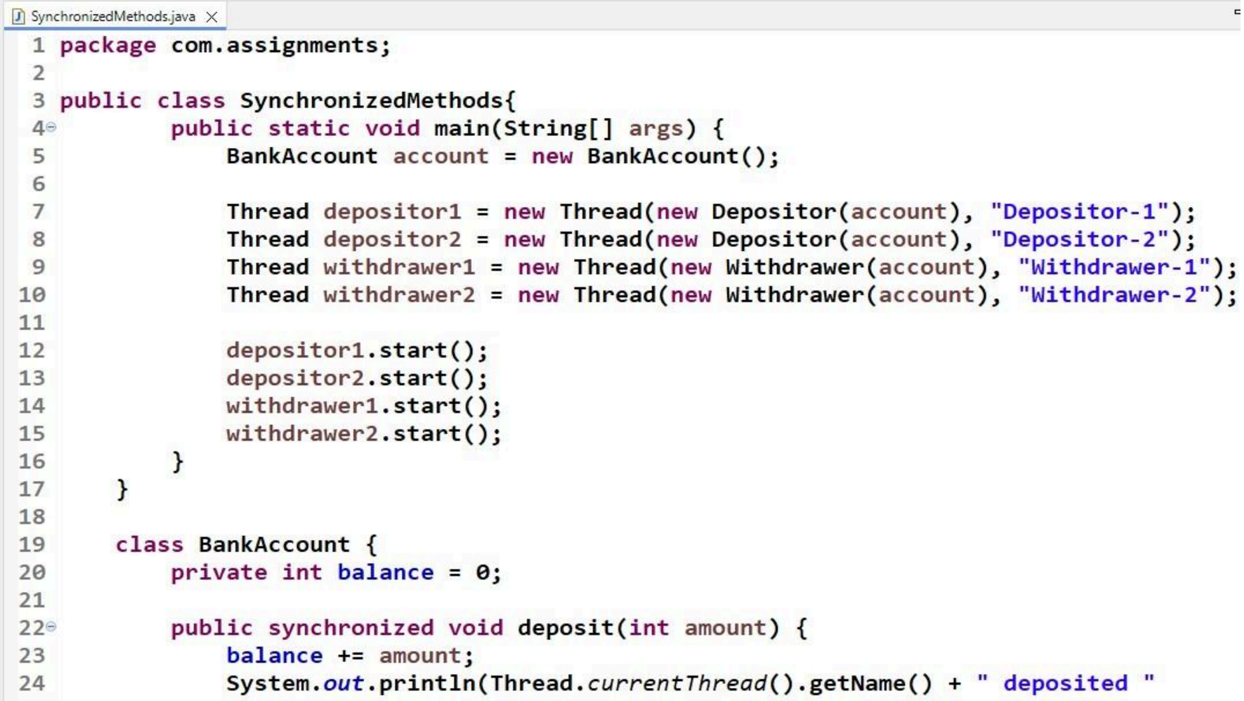
```
InterThreadCommunication [Java Application]  [pid: 5952]
Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Produced: 4
Consumed: 3
Produced: 5
Consumed: 4
Produced: 6
Produced: 7
Consumed: 5
Produced: 8
Consumed: 6
Produced: 9
Produced: 10
Consumed: 7
Produced: 11
Consumed: 8
Produced: 12
Produced: 13
Consumed: 9
Produced: 14
Consumed: 10
Produced: 15
```

**4]** Task 4: Synchronized Blocks and Methods

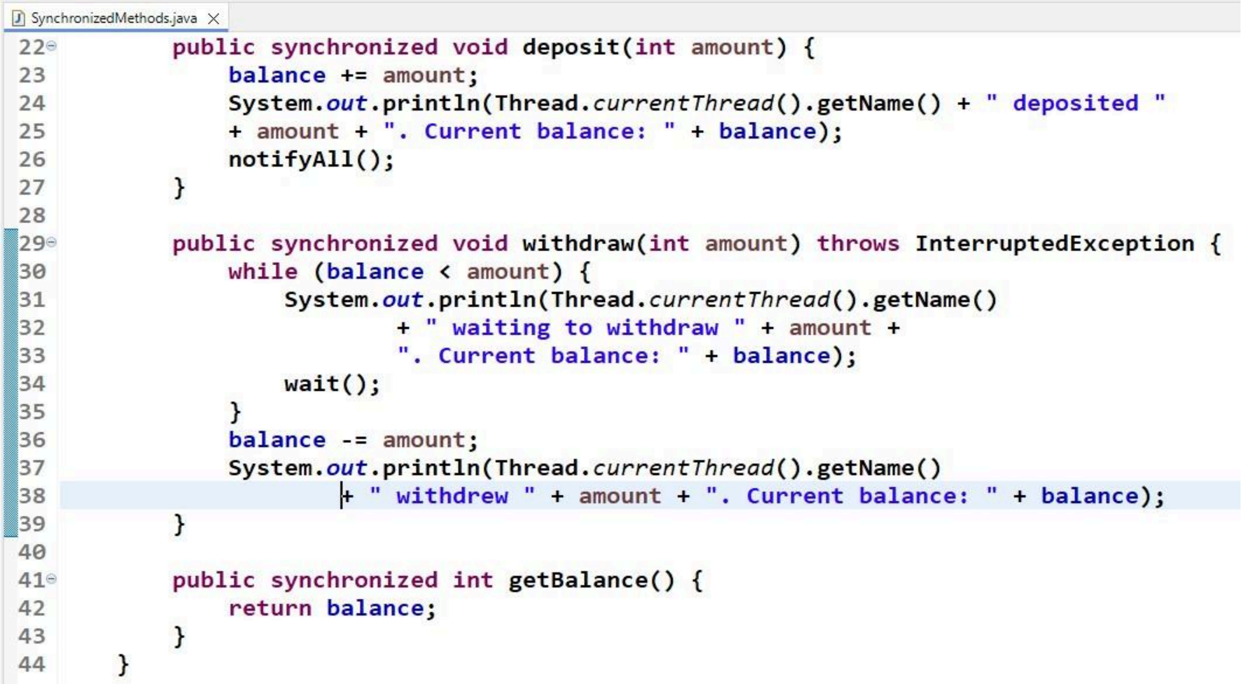
Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

Solution:-

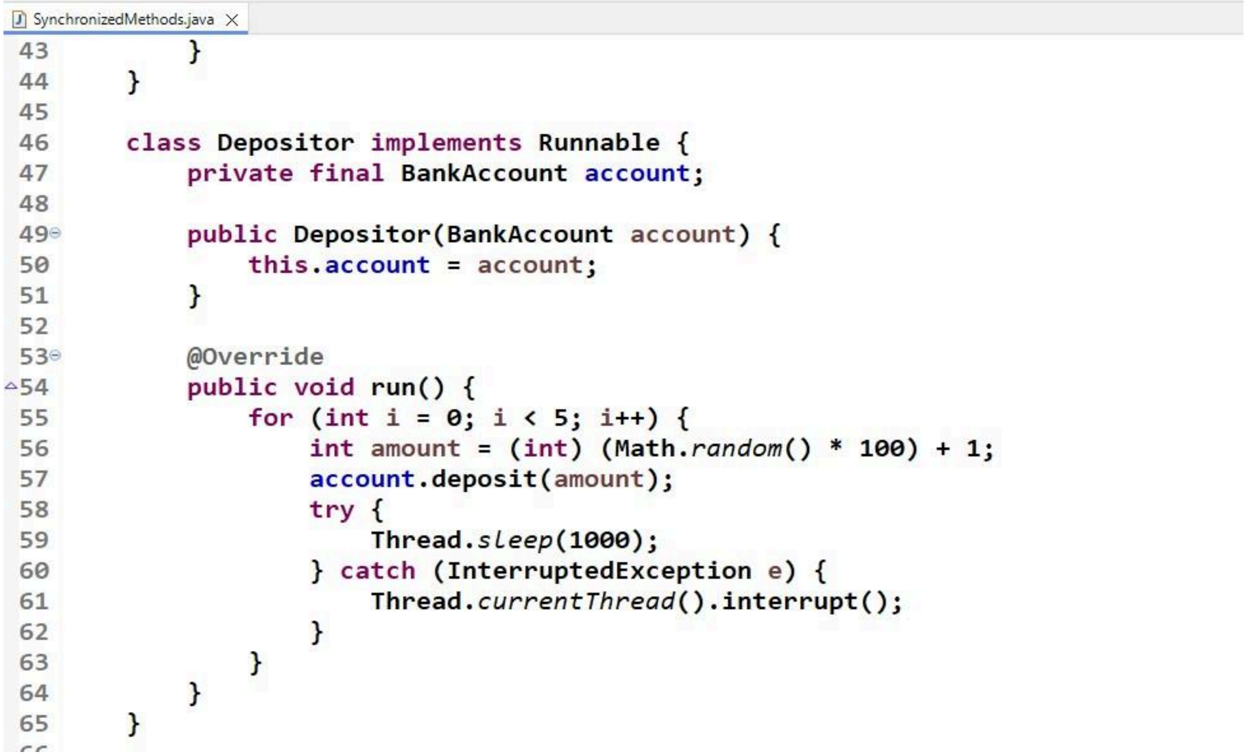
Code -



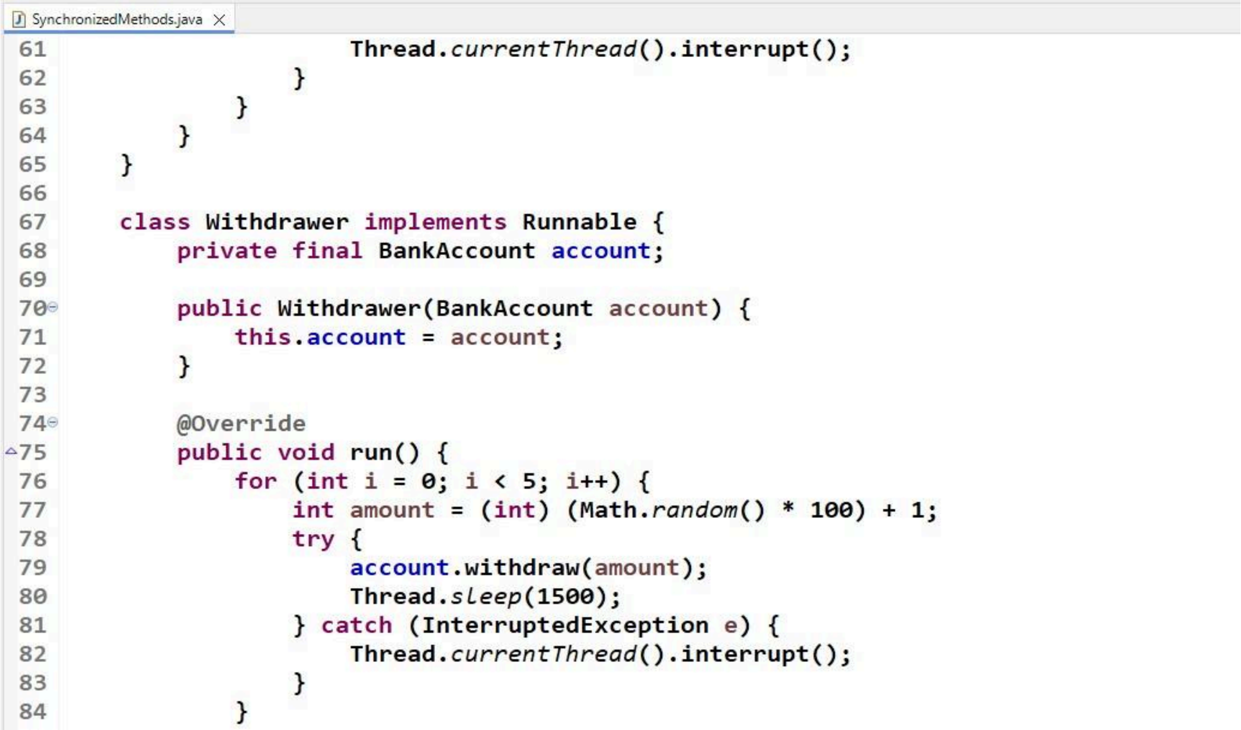
```java
package com.assignments;

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

            Thread depositor1 = new Thread(new Depositor(account), "Depositor-1");
            Thread depositor2 = new Thread(new Depositor(account), "Depositor-2");
            Thread withdrawer1 = new Thread(new Withdrawer(account), "Withdrawer-1");
            Thread withdrawer2 = new Thread(new Withdrawer(account), "Withdrawer-2");

            depositor1.start();
            depositor2.start();
            withdrawer1.start();
            withdrawer2.start();
        }
    }

    class BankAccount {
        private int balance = 0;

        public synchronized void deposit(int amount) {
            balance += amount;
            System.out.println(Thread.currentThread().getName() + " deposited "
```



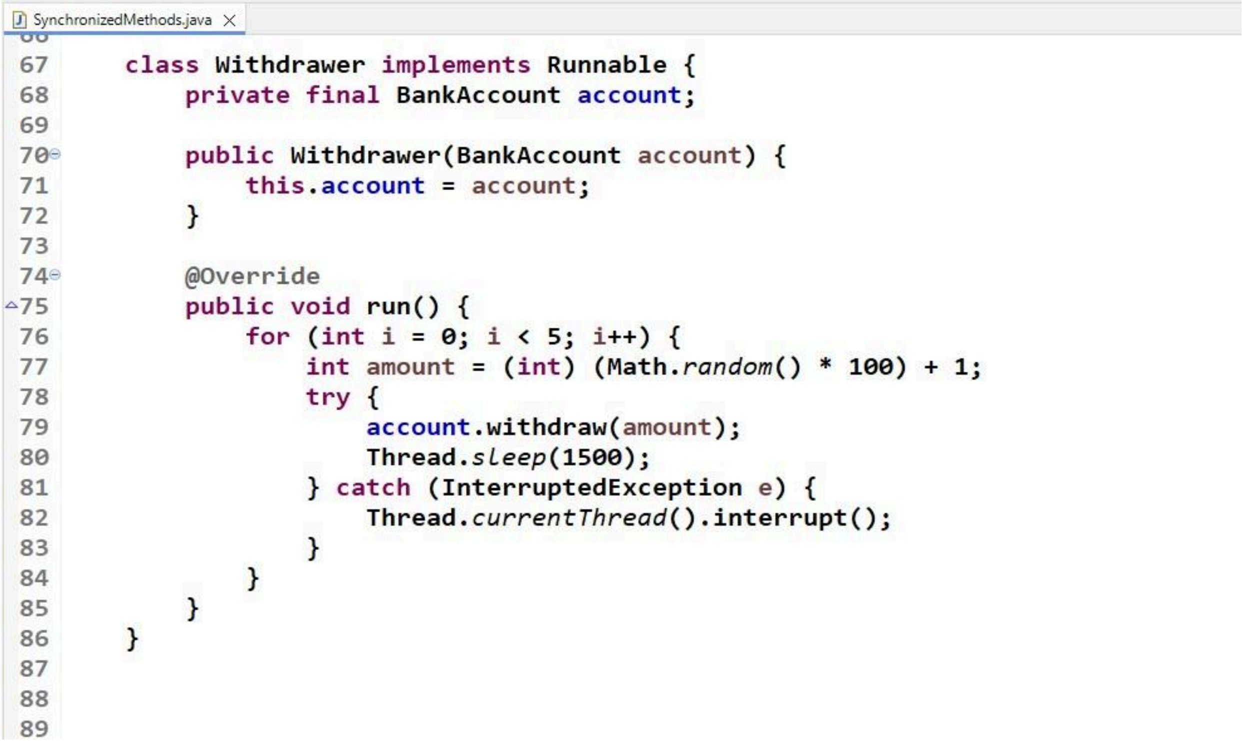
```java
    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println(Thread.currentThread().getName() + " deposited "
        + amount + ". Current balance: " + balance);
        notifyAll();
    }

    public synchronized void withdraw(int amount) throws InterruptedException {
        while (balance < amount) {
            System.out.println(Thread.currentThread().getName()
                    + " waiting to withdraw " + amount +
                    ". Current balance: " + balance);
            wait();
        }
        balance -= amount;
        System.out.println(Thread.currentThread().getName()
                + " withdrew " + amount + ". Current balance: " + balance);
    }

    public synchronized int getBalance() {
        return balance;
    }
}
```



```java
    }
}

class Depositor implements Runnable {
    private final BankAccount account;

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

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            int amount = (int) (Math.random() * 100) + 1;
            account.deposit(amount);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

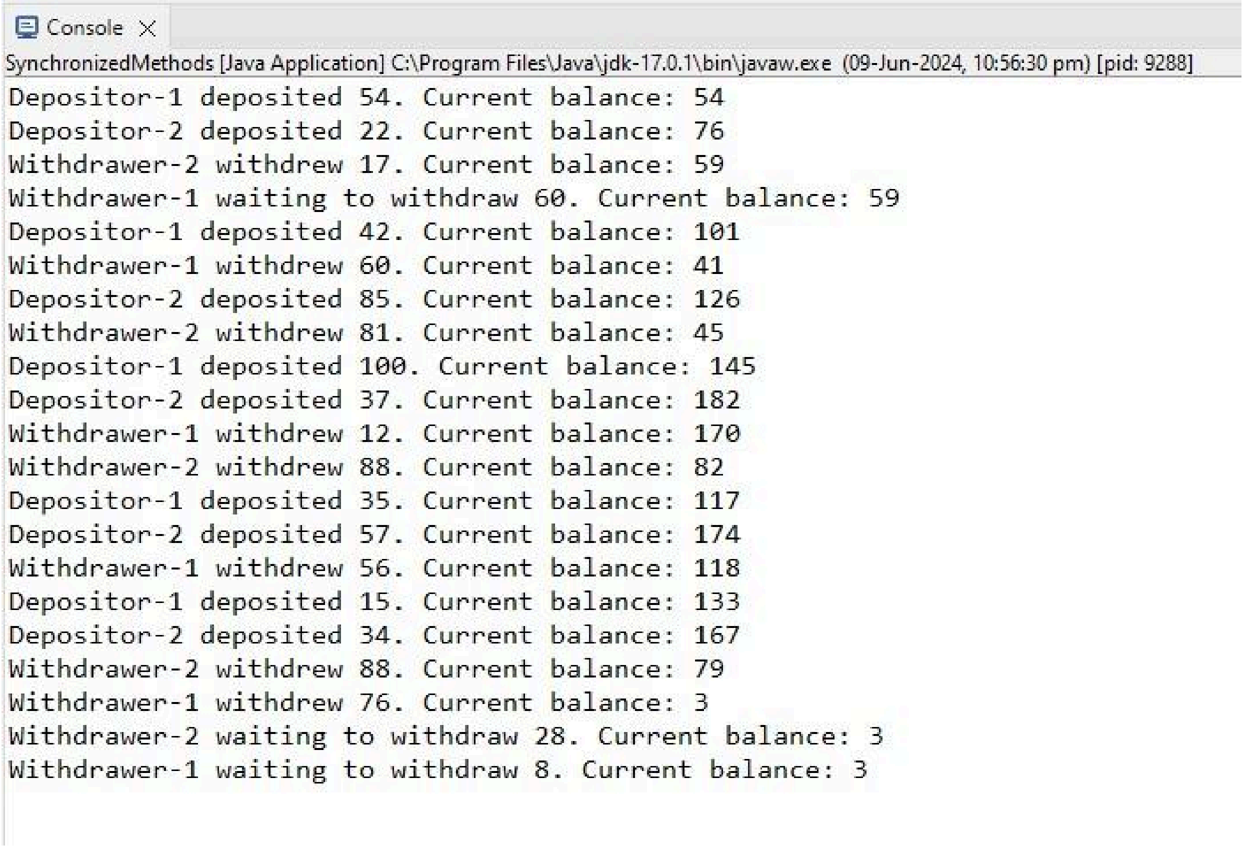


```java
                    Thread.currentThread().interrupt();
                }
            }
        }
    }

    class Withdrawer implements Runnable {
        private final BankAccount account;

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

        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                int amount = (int) (Math.random() * 100) + 1;
                try {
                    account.withdraw(amount);
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
```



```java
    class Withdrawer implements Runnable {
        private final BankAccount account;

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

        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                int amount = (int) (Math.random() * 100) + 1;
                try {
                    account.withdraw(amount);
                    Thread.sleep(1500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }



```

Output -

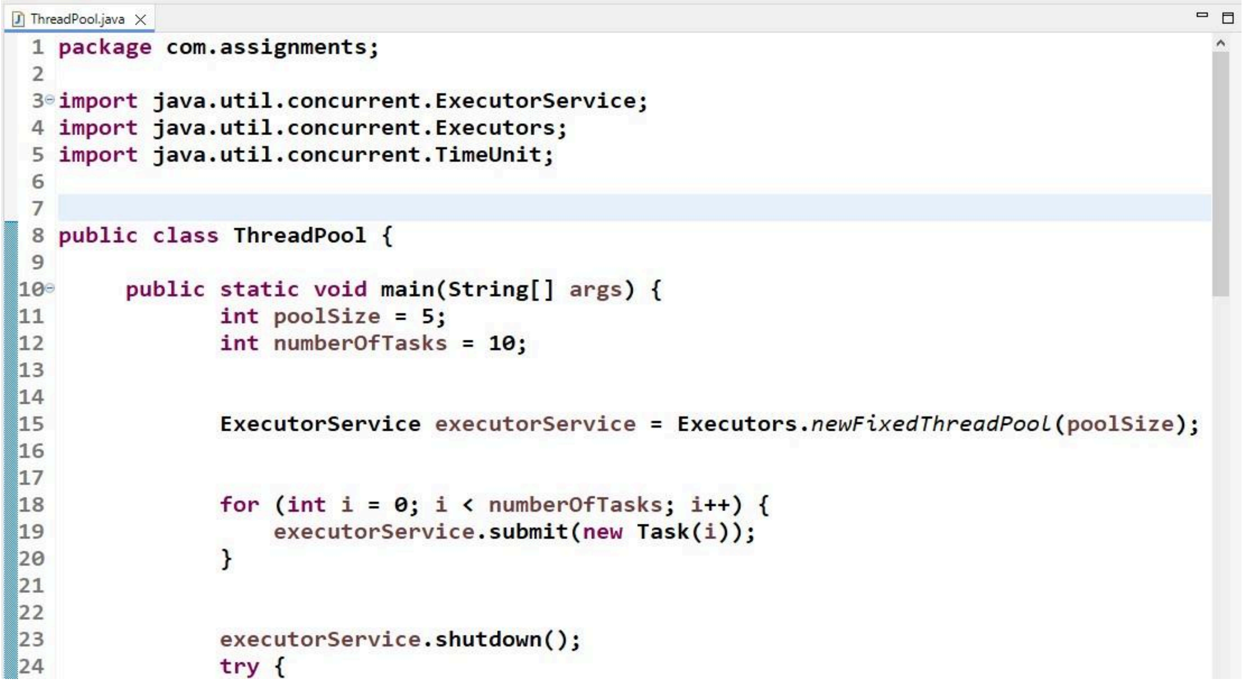


SynchronizedMethods [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe (09-Jun-2024, 10:56:30 pm) [pid: 9288]

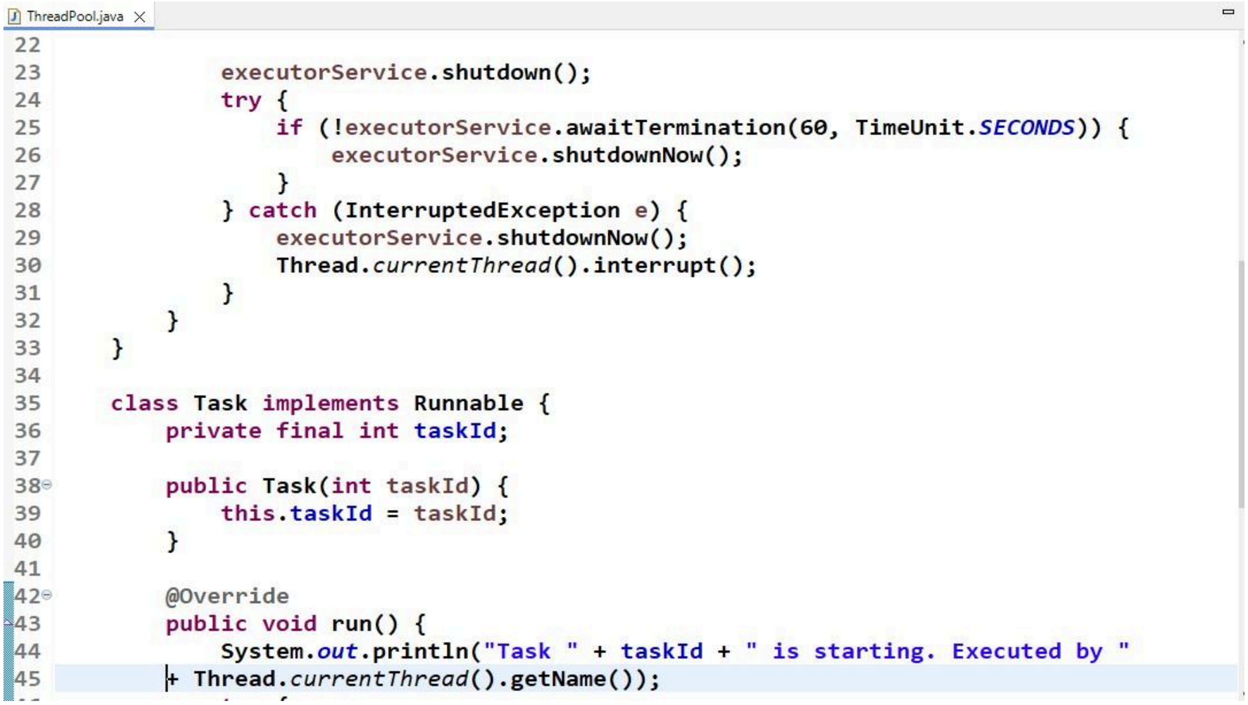
```
Depositor-1 deposited 54. Current balance: 54
Depositor-2 deposited 22. Current balance: 76
Withdrawer-2 withdrew 17. Current balance: 59
Withdrawer-1 waiting to withdraw 60. Current balance: 59
Depositor-1 deposited 42. Current balance: 101
Withdrawer-1 withdrew 60. Current balance: 41
Depositor-2 deposited 85. Current balance: 126
Withdrawer-2 withdrew 81. Current balance: 45
Depositor-1 deposited 100. Current balance: 145
Depositor-2 deposited 37. Current balance: 182
Withdrawer-1 withdrew 12. Current balance: 170
Withdrawer-2 withdrew 88. Current balance: 82
Depositor-1 deposited 35. Current balance: 117
Depositor-2 deposited 57. Current balance: 174
Withdrawer-1 withdrew 56. Current balance: 118
Depositor-1 deposited 15. Current balance: 133
Depositor-2 deposited 34. Current balance: 167
Withdrawer-2 withdrew 88. Current balance: 79
Withdrawer-1 withdrew 76. Current balance: 3
Withdrawer-2 waiting to withdraw 28. Current balance: 3
Withdrawer-1 waiting to withdraw 8. Current balance: 3
```

**5]** 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.

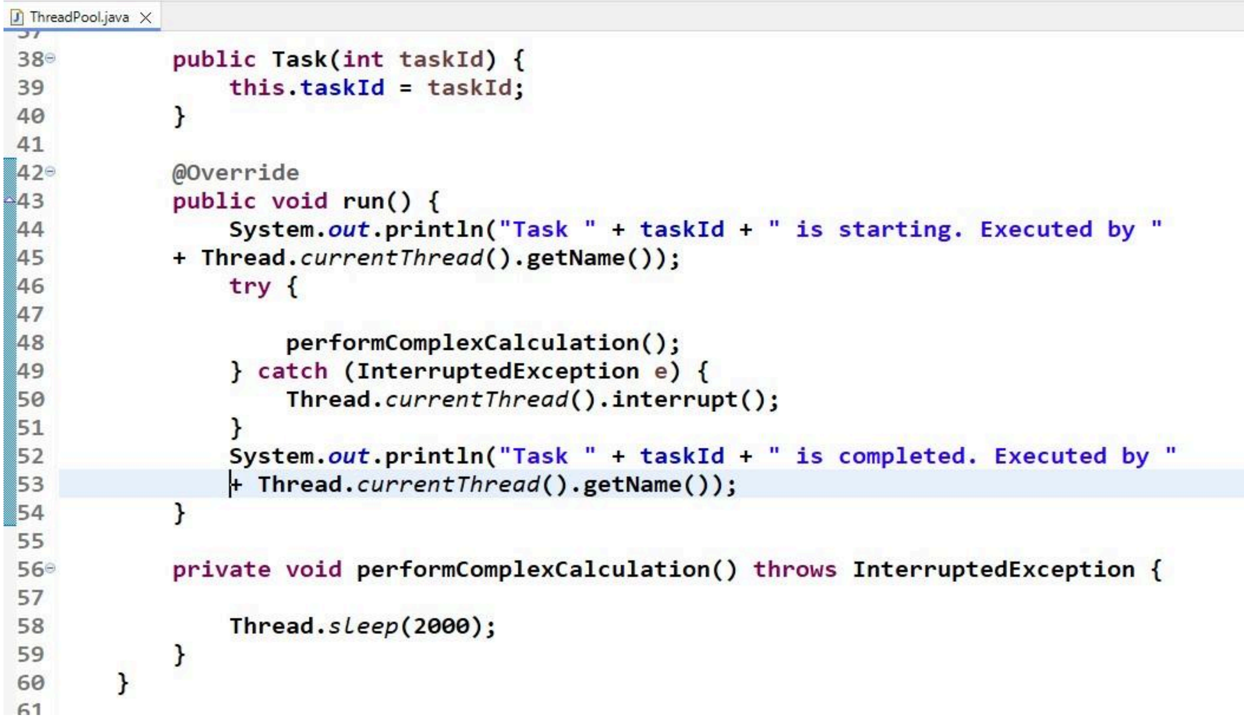
Solution:-
Code -



```java
package com.assignments;

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


public class ThreadPool {

  public static void main(String[] args) {
      int poolSize = 5;
      int numberOfTasks = 10;


      ExecutorService executorService = Executors.newFixedThreadPool(poolSize);


      for (int i = 0; i < numberOfTasks; i++) {
        executorService.submit(new Task(i));
      }


      executorService.shutdown();
      try {
```

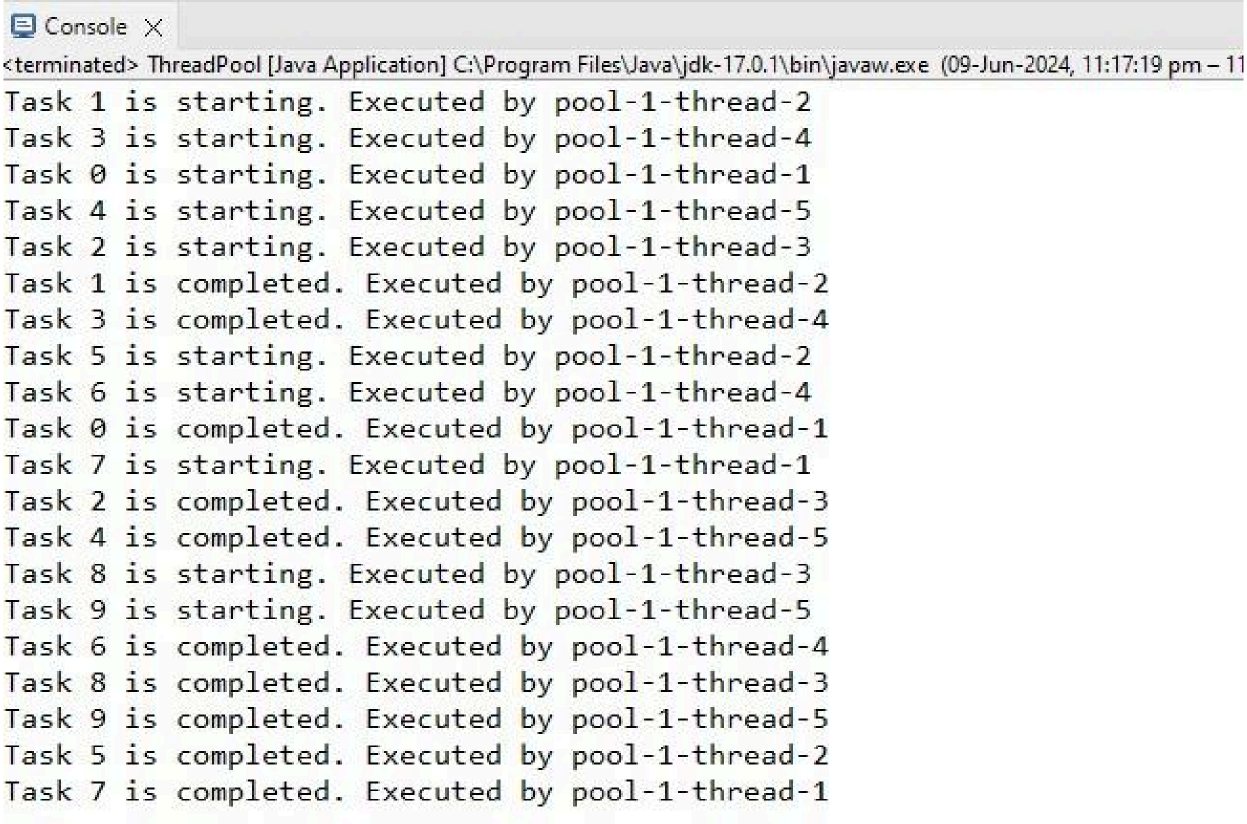


```java

      executorService.shutdown();
      try {
        if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
          executorService.shutdownNow();
        }
      } catch (InterruptedException e) {
        executorService.shutdownNow();
        Thread.currentThread().interrupt();
      }
    }
  }

  class Task implements Runnable {
    private final int taskId;

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

    @Override
    public void run() {
      System.out.println("Task " + taskId + " is starting. Executed by "
    + Thread.currentThread().getName());
```



```
        public Task(int taskId) {
            this.taskId = taskId;
        }

        @Override
        public void run() {
            System.out.println("Task " + taskId + " is starting. Executed by "
        + Thread.currentThread().getName());
            try {

                performComplexCalculation();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            System.out.println("Task " + taskId + " is completed. Executed by "
            + Thread.currentThread().getName());
        }

        private void performComplexCalculation() throws InterruptedException {

            Thread.sleep(2000);
        }
    }
```

Output -

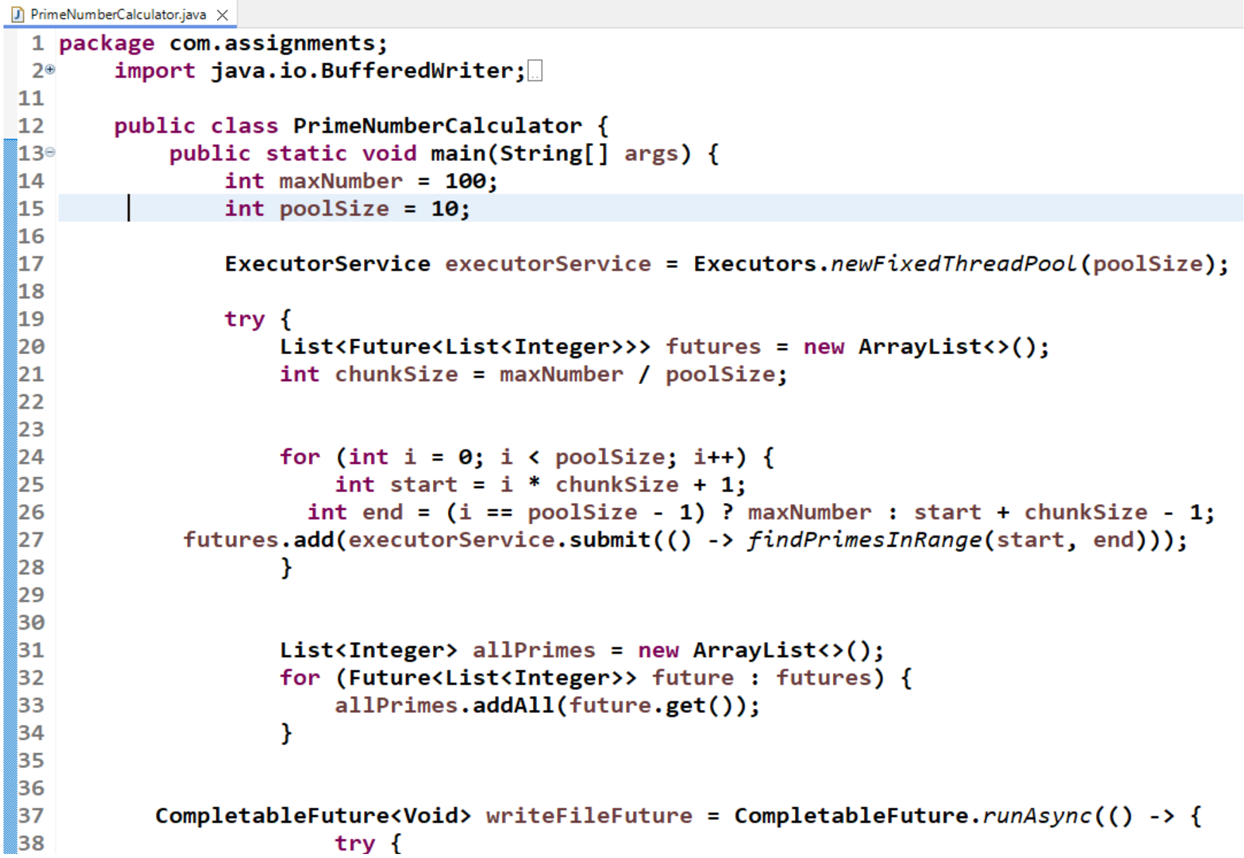


<terminated> ThreadPool [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe (09-Jun-2024, 11:17:19 pm – 11

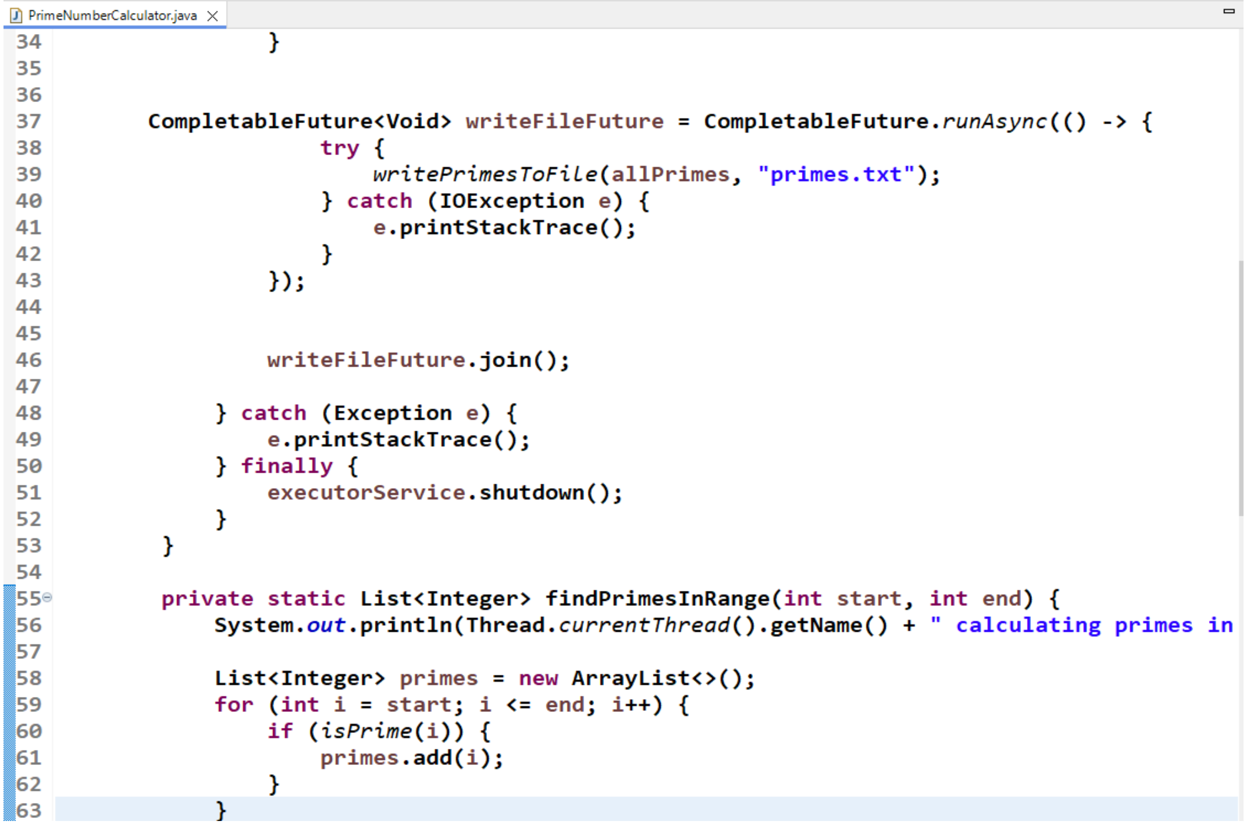
```
Task 1 is starting. Executed by pool-1-thread-2
Task 3 is starting. Executed by pool-1-thread-4
Task 0 is starting. Executed by pool-1-thread-1
Task 4 is starting. Executed by pool-1-thread-5
Task 2 is starting. Executed by pool-1-thread-3
Task 1 is completed. Executed by pool-1-thread-2
Task 3 is completed. Executed by pool-1-thread-4
Task 5 is starting. Executed by pool-1-thread-2
Task 6 is starting. Executed by pool-1-thread-4
Task 0 is completed. Executed by pool-1-thread-1
Task 7 is starting. Executed by pool-1-thread-1
Task 2 is completed. Executed by pool-1-thread-3
Task 4 is completed. Executed by pool-1-thread-5
Task 8 is starting. Executed by pool-1-thread-3
Task 9 is starting. Executed by pool-1-thread-5
Task 6 is completed. Executed by pool-1-thread-4
Task 8 is completed. Executed by pool-1-thread-3
Task 9 is completed. Executed by pool-1-thread-5
Task 5 is completed. Executed by pool-1-thread-2
Task 7 is completed. Executed by pool-1-thread-1
```

**6]** 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.

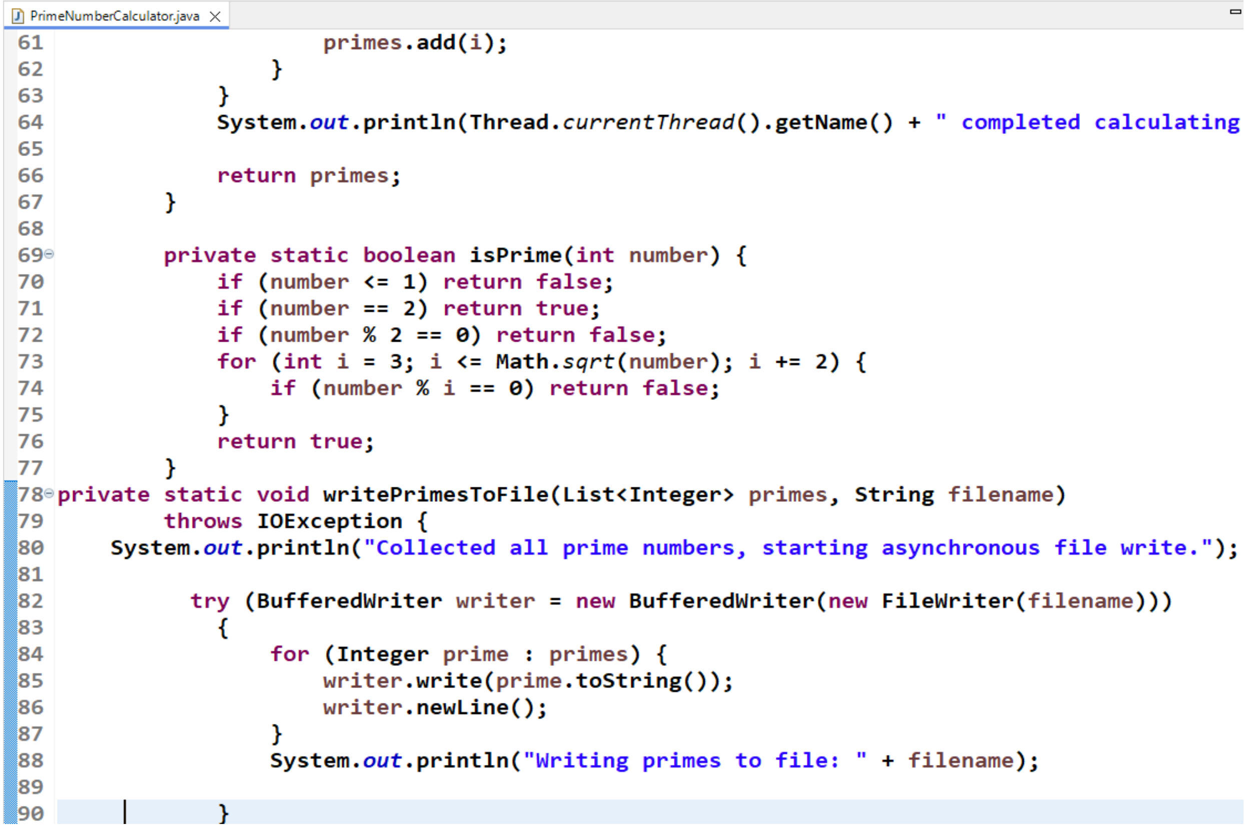
Solution:-
Code-



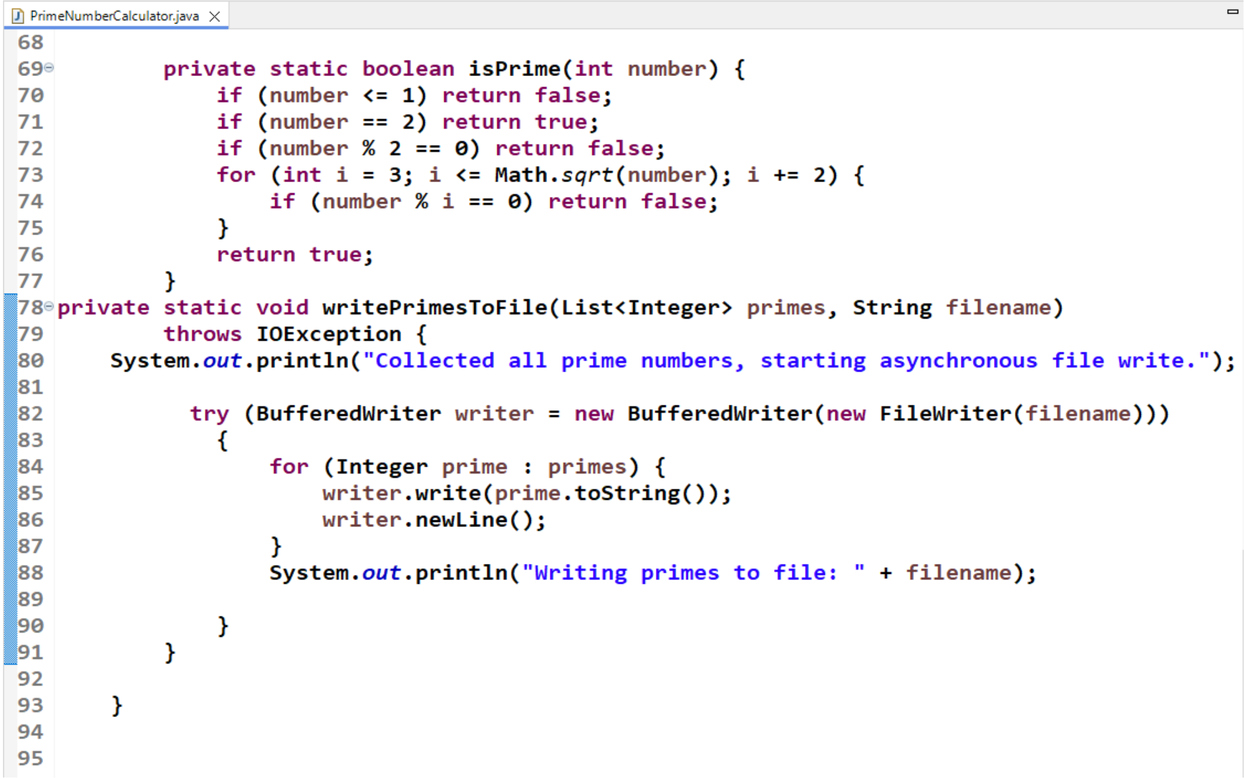
```java
package com.assignments;
  import java.io.BufferedWriter;

  public class PrimeNumberCalculator {
    public static void main(String[] args) {
      int maxNumber = 100;
      int poolSize = 10;

      ExecutorService executorService = Executors.newFixedThreadPool(poolSize);

      try {
        List<Future<List<Integer>>> futures = new ArrayList<>();
        int chunkSize = maxNumber / poolSize;


        for (int i = 0; i < poolSize; i++) {
          int start = i * chunkSize + 1;
         int end = (i == poolSize - 1) ? maxNumber : start + chunkSize - 1;
    futures.add(executorService.submit(() -> findPrimesInRange(start, end)));
        }


        List<Integer> allPrimes = new ArrayList<>();
        for (Future<List<Integer>> future : futures) {
          allPrimes.addAll(future.get());
        }


   CompletableFuture<Void> writeFileFuture = CompletableFuture.runAsync(() -> {
          try {
```



```java
                }


       CompletableFuture<Void> writeFileFuture = CompletableFuture.runAsync(() -> {
                    try {
                        writePrimesToFile(allPrimes, "primes.txt");
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                });


                writeFileFuture.join();

            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                executorService.shutdown();
            }
        }

        private static List<Integer> findPrimesInRange(int start, int end) {
            System.out.println(Thread.currentThread().getName() + " calculating primes in

            List<Integer> primes = new ArrayList<>();
            for (int i = start; i <= end; i++) {
                if (isPrime(i)) {
                    primes.add(i);
                }
            }
```



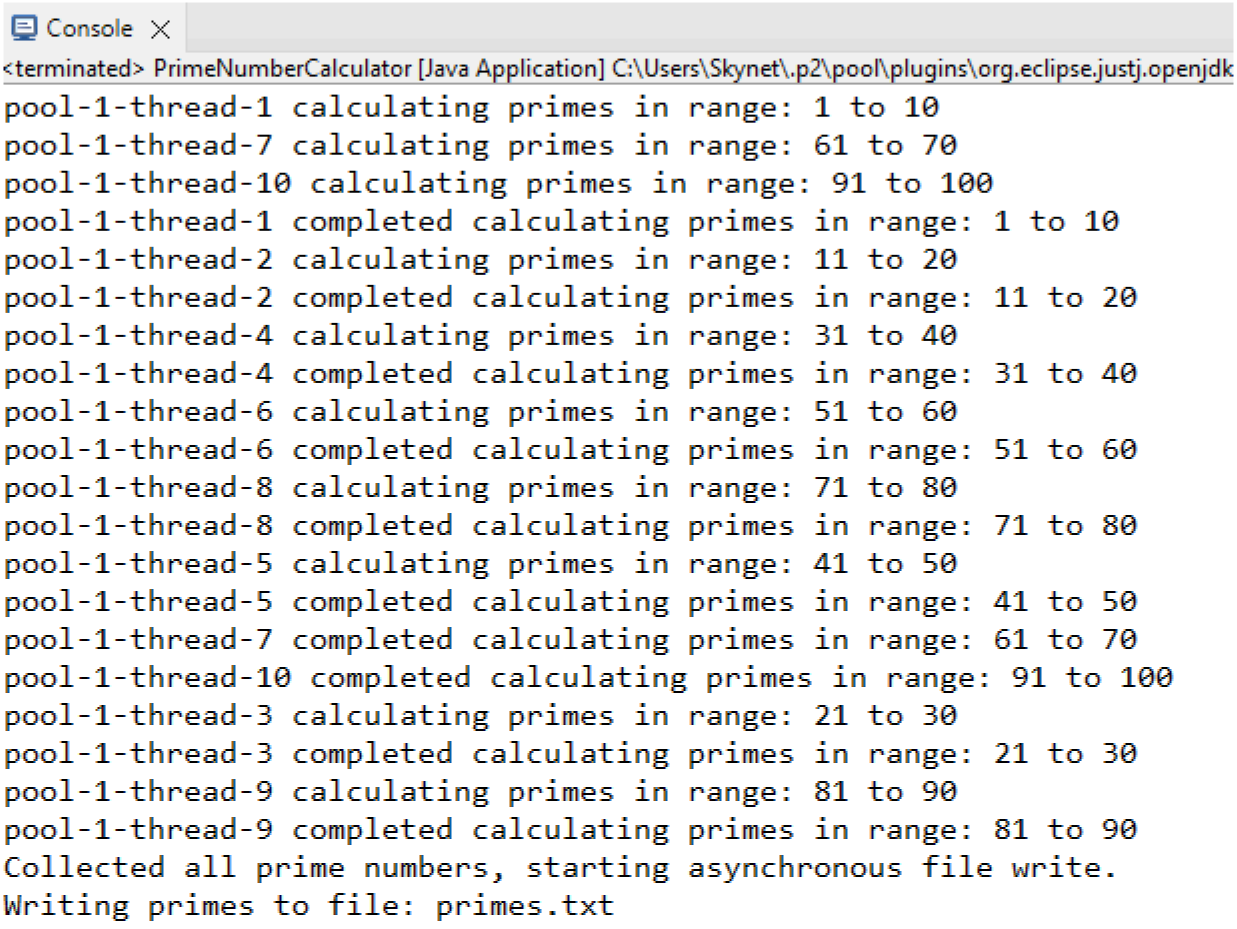
```java
                    primes.add(i);
                }
            }
            System.out.println(Thread.currentThread().getName() + " completed calculating

            return primes;
        }

        private static boolean isPrime(int number) {
            if (number <= 1) return false;
            if (number == 2) return true;
            if (number % 2 == 0) return false;
            for (int i = 3; i <= Math.sqrt(number); i += 2) {
                if (number % i == 0) return false;
            }
            return true;
        }
private static void writePrimesToFile(List<Integer> primes, String filename)
        throws IOException {
    System.out.println("Collected all prime numbers, starting asynchronous file write.");

          try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename)))
            {
                for (Integer prime : primes) {
                    writer.write(prime.toString());
                    writer.newLine();
                }
                System.out.println("Writing primes to file: " + filename);

            }
```



PrimeNumberCalculator.java

```java

        private static boolean isPrime(int number) {
            if (number <= 1) return false;
            if (number == 2) return true;
            if (number % 2 == 0) return false;
            for (int i = 3; i <= Math.sqrt(number); i += 2) {
                if (number % i == 0) return false;
            }
            return true;
        }
private static void writePrimesToFile(List<Integer> primes, String filename)
        throws IOException {
    System.out.println("Collected all prime numbers, starting asynchronous file write.");

          try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename)))
            {
                for (Integer prime : primes) {
                    writer.write(prime.toString());
                    writer.newLine();
                }
                System.out.println("Writing primes to file: " + filename);

            }
        }

    }
```

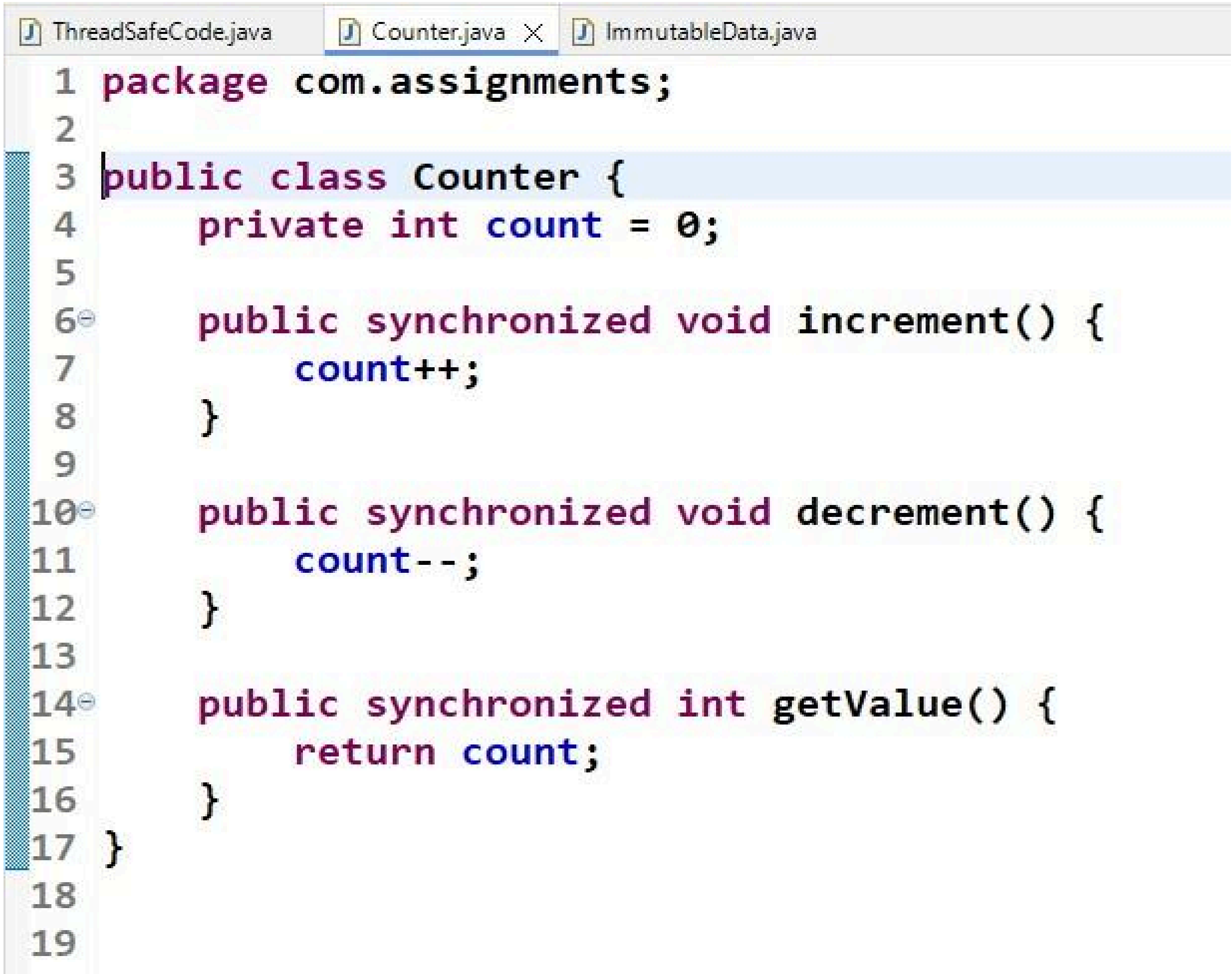
Output :-



```
<terminated> PrimeNumberCalculator [Java Application] C:\Users\Skynet\.p2\pool\plugins\org.eclipse.justj.openjdk
pool-1-thread-1 calculating primes in range: 1 to 10
pool-1-thread-7 calculating primes in range: 61 to 70
pool-1-thread-10 calculating primes in range: 91 to 100
pool-1-thread-1 completed calculating primes in range: 1 to 10
pool-1-thread-2 calculating primes in range: 11 to 20
pool-1-thread-2 completed calculating primes in range: 11 to 20
pool-1-thread-4 calculating primes in range: 31 to 40
pool-1-thread-4 completed calculating primes in range: 31 to 40
pool-1-thread-6 calculating primes in range: 51 to 60
pool-1-thread-6 completed calculating primes in range: 51 to 60
pool-1-thread-8 calculating primes in range: 71 to 80
pool-1-thread-8 completed calculating primes in range: 71 to 80
pool-1-thread-5 calculating primes in range: 41 to 50
pool-1-thread-5 completed calculating primes in range: 41 to 50
pool-1-thread-7 completed calculating primes in range: 61 to 70
pool-1-thread-10 completed calculating primes in range: 91 to 100
pool-1-thread-3 calculating primes in range: 21 to 30
pool-1-thread-3 completed calculating primes in range: 21 to 30
pool-1-thread-9 calculating primes in range: 81 to 90
pool-1-thread-9 completed calculating primes in range: 81 to 90
Collected all prime numbers, starting asynchronous file write.
Writing primes to file: primes.txt
```

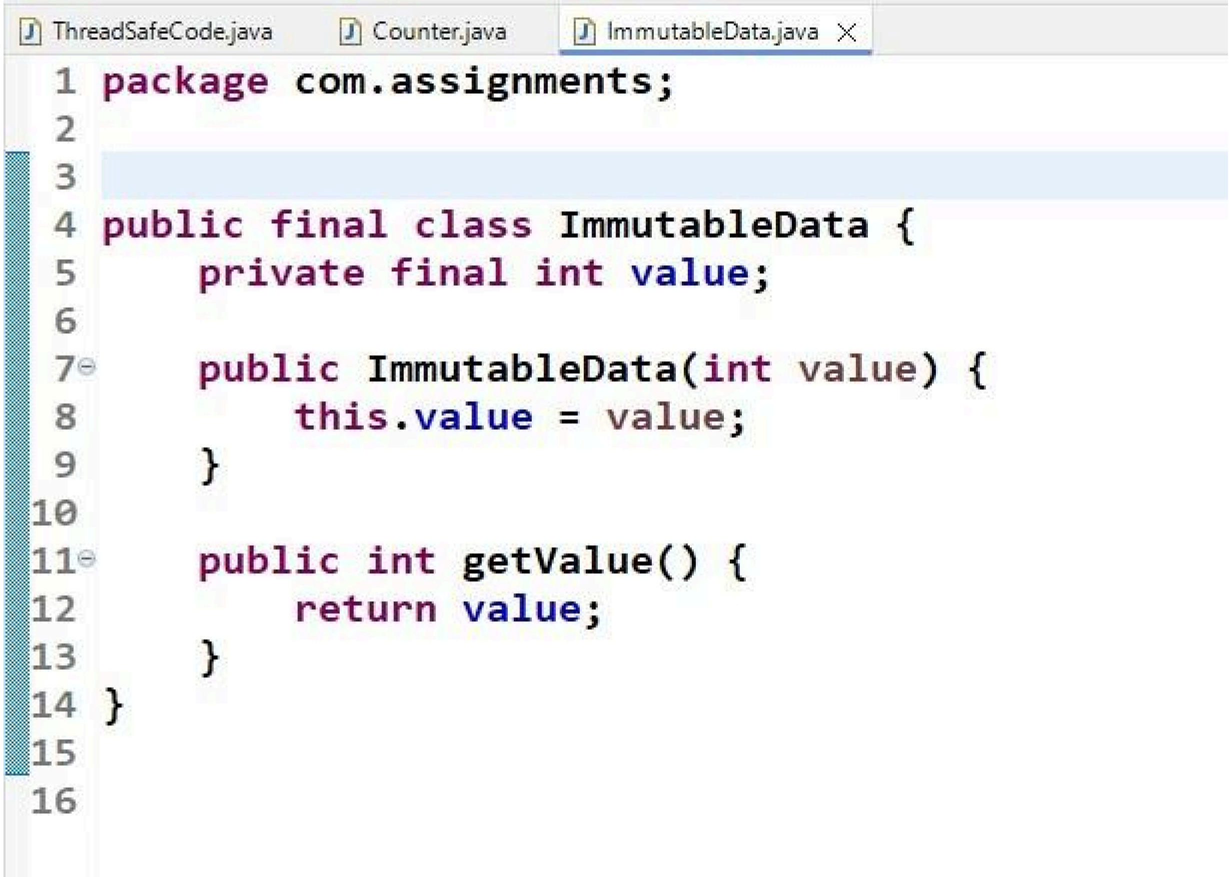
**7]** Task 7: Writing Thread-Safe Code, Immutable Objects
Design a thread-safe Counter class with increment and decrement methods. Then demonstrate its usage from multiple threads. Also, implement and use an immutable class to share data between threads.

Solution:-
Code -
Counter.java



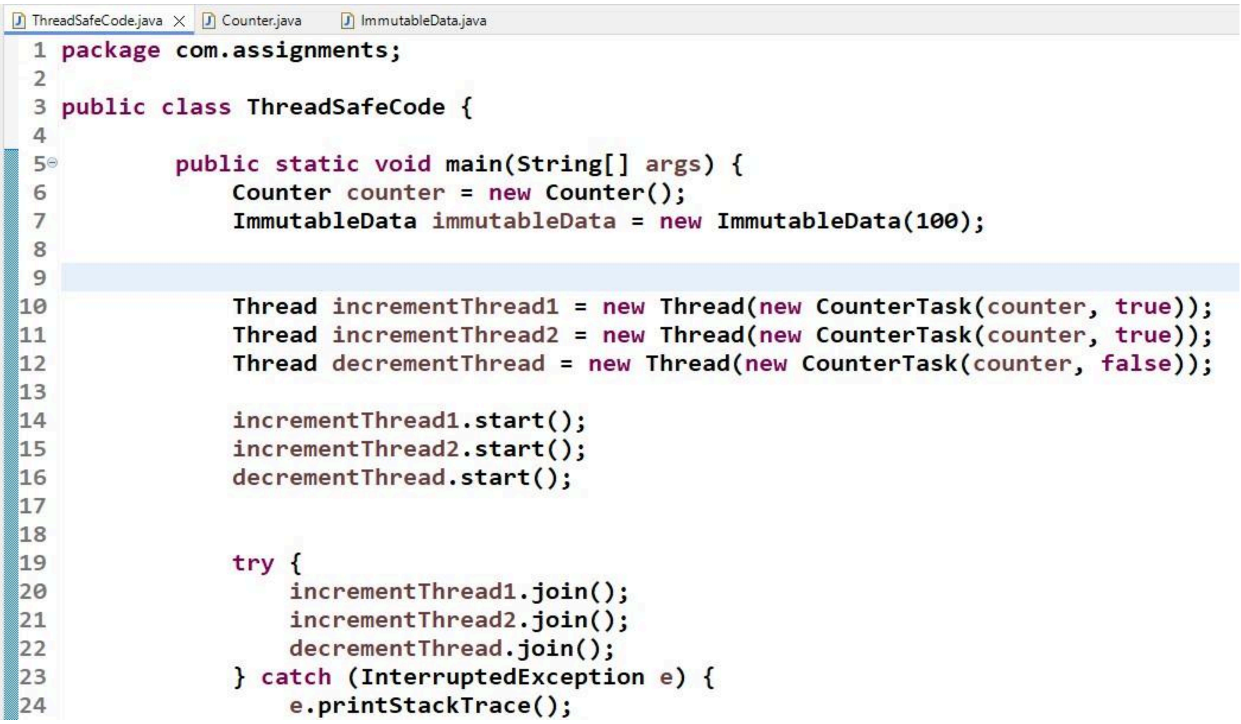
```java
package com.assignments;

public class Counter {
    private int count = 0;

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

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

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


```

Immutable.java

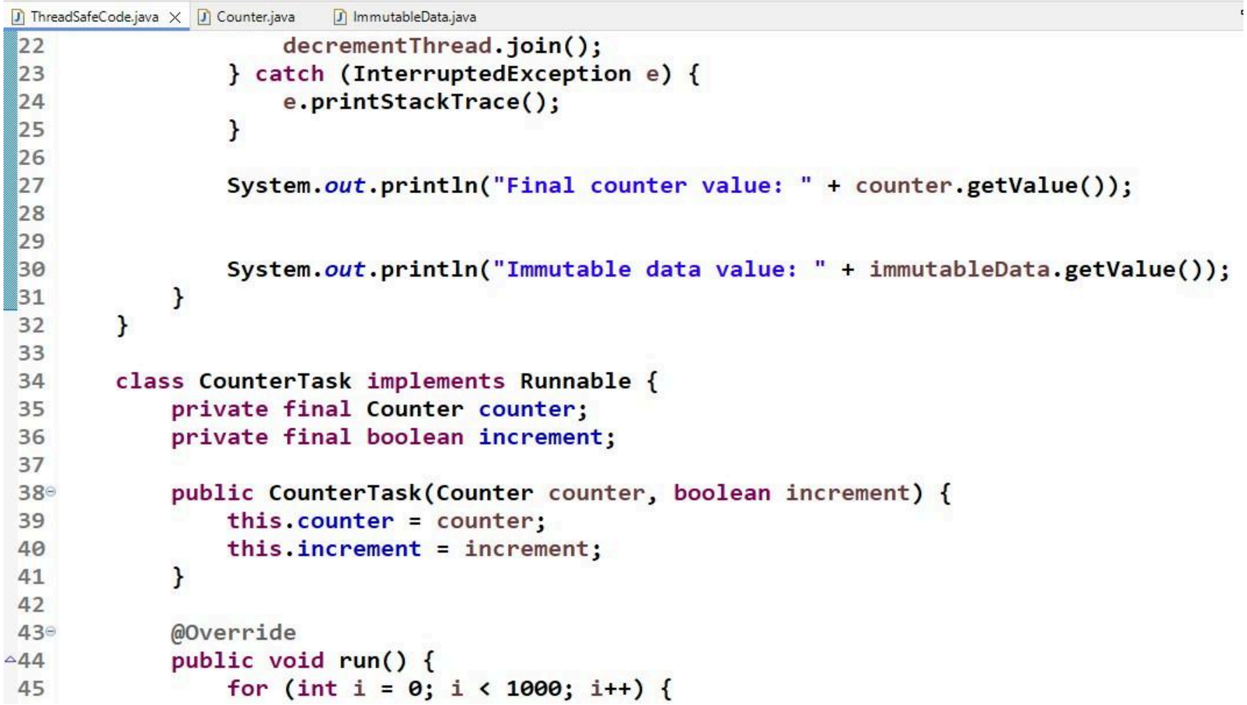


```java
package com.assignments;


public final class ImmutableData {
    private final int value;

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

    public int getValue() {
        return value;
    }
}


```

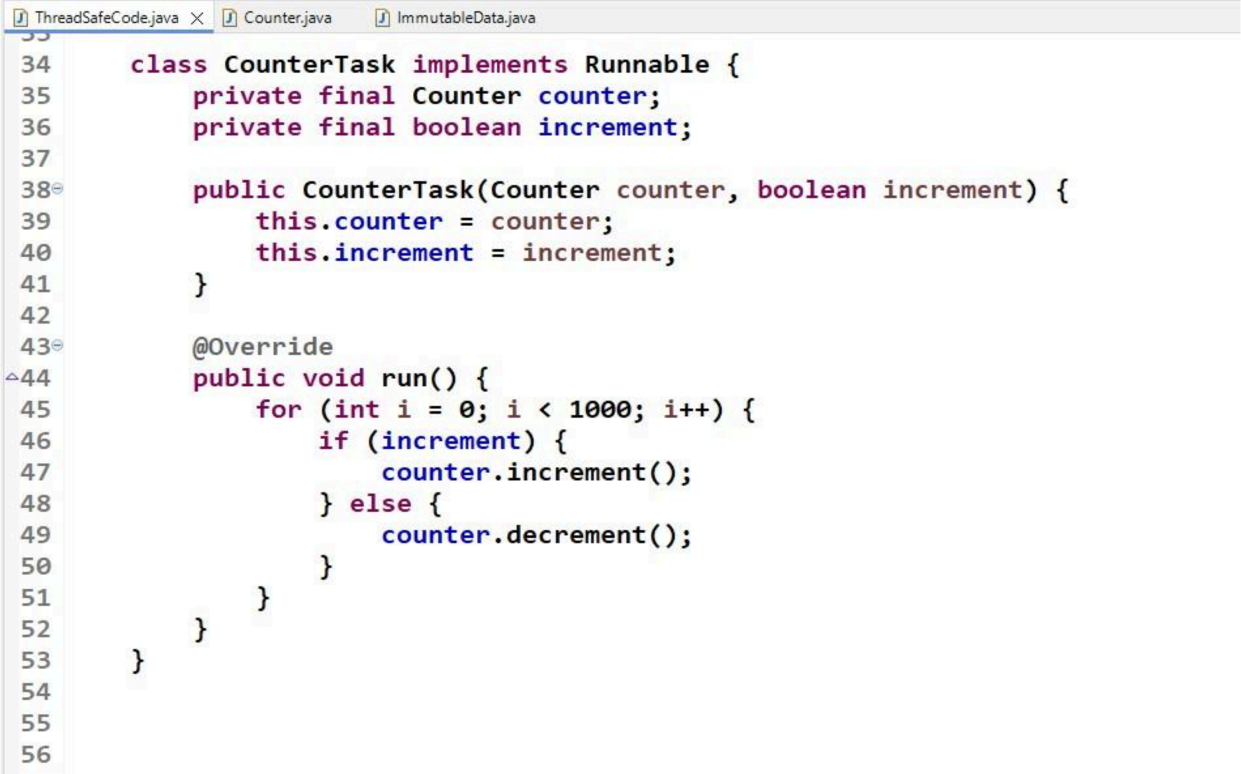
ThreadSafeCode.java



```java
package com.assignments;

public class ThreadSafeCode {

        public static void main(String[] args) {
            Counter counter = new Counter();
            ImmutableData immutableData = new ImmutableData(100);


            Thread incrementThread1 = new Thread(new CounterTask(counter, true));
            Thread incrementThread2 = new Thread(new CounterTask(counter, true));
            Thread decrementThread = new Thread(new CounterTask(counter, false));

            incrementThread1.start();
            incrementThread2.start();
            decrementThread.start();


            try {
                incrementThread1.join();
                incrementThread2.join();
                decrementThread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
```

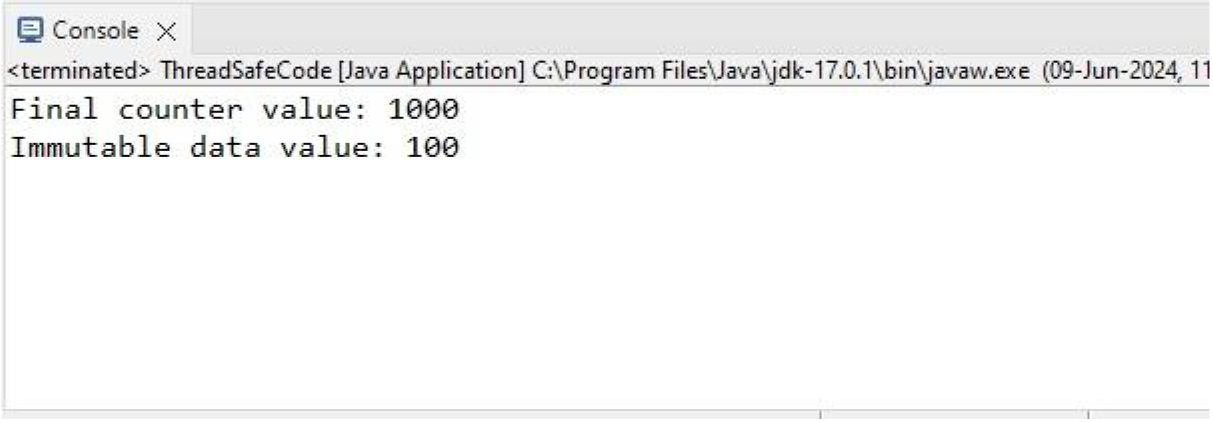


```
                decrementThread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("Final counter value: " + counter.getValue());


            System.out.println("Immutable data value: " + immutableData.getValue());
        }
    }

    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 < 1000; i++) {
```



```
    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 < 1000; i++) {
                if (increment) {
                    counter.increment();
                } else {
                    counter.decrement();
                }
            }
        }
    }
```

Output -



<terminated> ThreadSafeCode [Java Application] C:\Program Files\Java\jdk-17.0.1\bin\javaw.exe  (09-Jun-2024, 11

```
Final counter value: 1000
Immutable data value: 100
```