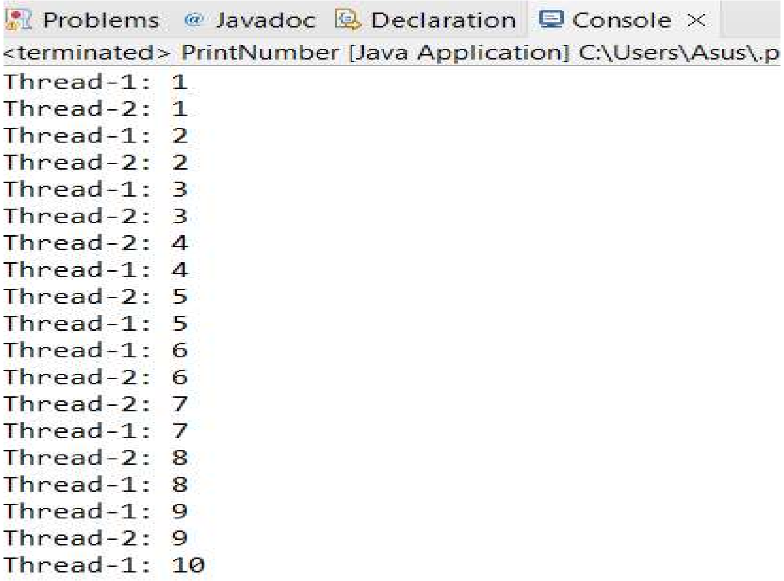
DAY 18

Task 1: Creating and Managing Threads:

Write a program that Starts two threads, where each thread points numbers from 1 to 10 with a 1 second delay between each Number.

```
package practice;
        public class PrintNumber implements Runnable {
            @Override
            public void run() {
                try {
                    for (int i = 1; i <= 10; i++) {
                        System.out.println(Thread.currentThread().getName() + ": " + i);
                        Thread.sleep(1000); // 1 second delay
                    }
                } catch (InterruptedException e) {
                    System.out.println(Thread.currentThread().getName() + " interrupted.");
                }
            }
            public static void main(String[] args) {
                Runnable task = new PrintNumber();


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

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

                try {
                    thread1.join();
                    thread2.join();
                } catch (InterruptedException e) {
                    System.out.println("Main thread interrupted.");
                }
                System.out.println("Both threads have finished.");
            }
        }
```

Output:



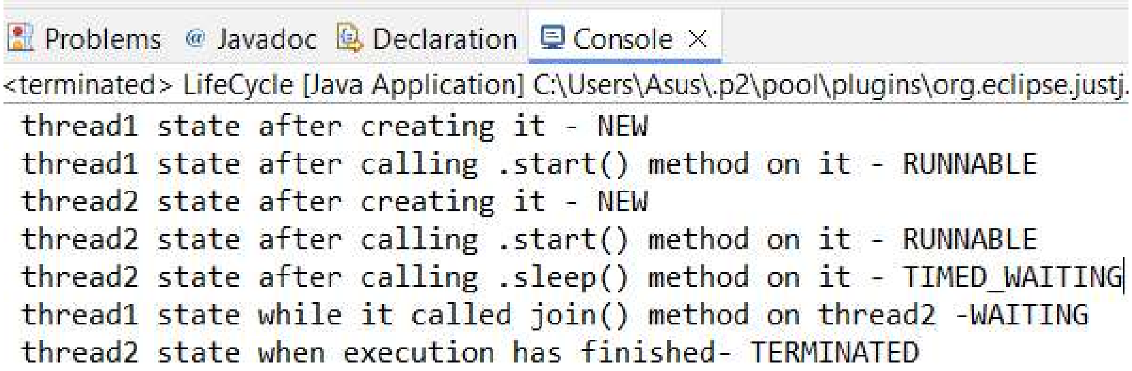
```
<terminated> PrintNumber [Java Application] C:\Users\Asus\.p
Thread-1:  1
Thread-2:  1
Thread-1:  2
Thread-2:  2
Thread-1:  3
Thread-2:  3
Thread-2:  4
Thread-1:  4
Thread-2:  5
Thread-1:  5
Thread-1:  6
Thread-2:  6
Thread-2:  7
Thread-1:  7
Thread-2:  8
Thread-1:  8
Thread-1:  9
Thread-2:  9
Thread-1:  10
```

Task 2:States and Transitions

Create a java class that simulates a thread going through different life cycles: NEW ,RUNNABLE, WAITING, TIMED_WAITING, BLOCKED and TERMINATED. Use methods like sleep(), wait(), and notify() and join() to demonstrate these states.

```
 package practice;
class thread implements Runnable {
  public void run()
  {

      try {
         Thread.sleep(1500);
      }
      catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println(" thread1 state while it called join() method on thread2 -" +
LifeCycle.thread1.getState());
      try {
         Thread.sleep(200);
      }
      catch (InterruptedException e) {
         e.printStackTrace();
      }
  }
}
public class LifeCycle implements Runnable {
  public static Thread thread1;
  public static LifeCycle obj;
  public static void main(String[] args)
  {
        obj = new LifeCycle();
        thread1 = new Thread(obj);

        System.out.println(" thread1 state after creating it - " + thread1.getState());
        thread1.start();

        System.out.println(" thread1 state after calling .start() method on it - " + thread1.getState());
    }
    public void run()
    {
        thread myThread = new thread();
        Thread thread2 = new Thread(myThread);

        System.out.println(" thread2 state after creating it - "  + thread2.getState());
        thread2.start();

        System.out.println(" thread2 state after calling .start() method on it - "
            + thread2.getState());

        try {

            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(" thread2 state after calling .sleep() method on it - " + thread2.getState());
        try {

            thread2.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(" thread2 state when execution has finished- " + thread2.getState());
    }
}
```

Output:



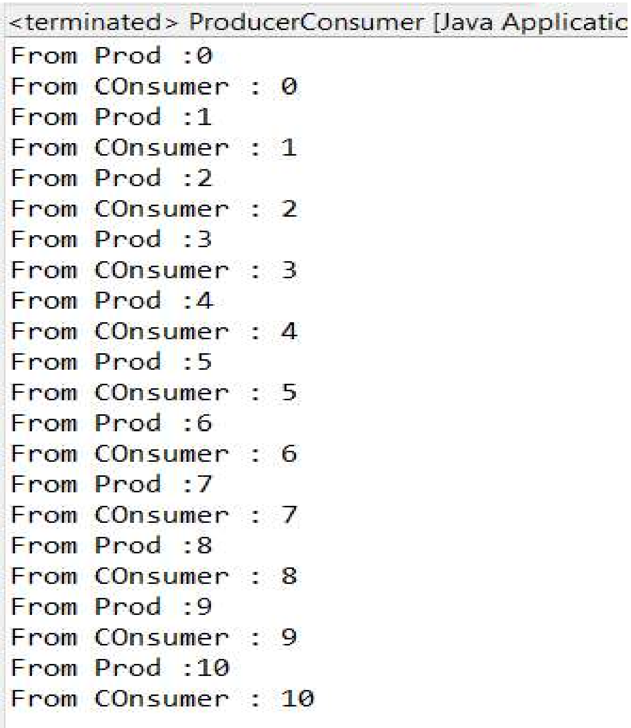
```
<terminated> LifeCycle [Java Application] C:\Users\Asus\.p2\pool\plugins\org.eclipse.justj.
 thread1 state after creating it - NEW
 thread1 state after calling .start() method on it - RUNNABLE
 thread2 state after creating it - NEW
 thread2 state after calling .start() method on it - RUNNABLE
 thread2 state after calling .sleep() method on it - TIMED_WAITING
 thread1 state while it called join() method on thread2 -WAITING
 thread2 state when execution has finished- 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 practice;
class Common {
    int num;
    boolean available = false;
    public synchronized int put(int num) {
        if (available)
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        this.num = num;
        System.out.println("From Prod :" + this.num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        available = true;
        notify();
        return num;
    }
    public synchronized int get() {
        if (!available)
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        System.out.println("From COnsumer : " + this.num);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        available = false;
        notify();
        return num;
    }
}
class Producer extends Thread {
    Common c;
    public Producer(Common c) {
        this.c = c;
        new Thread(this, "Producer :").start();
    }
    public void run() {
        int x = 0, i = 0;
        while (x <= 10) {
            c.put(i++);
            x++;
        }
    }
}
class Consumer extends Thread {
    Common c;
    public Consumer(Common c) {
        this.c = c;
        new Thread(this, "Consumer :").start();
    }
    public void run() {
        int x = 0;
        while (x <= 10) {
            c.get();
            x++;
        }
    }
}
public class ProducerConsumer {
    public static void main(String[] args) {
        Common c = new Common();
        new Producer(c);
        new Consumer(c);
    }
}
```

Output:



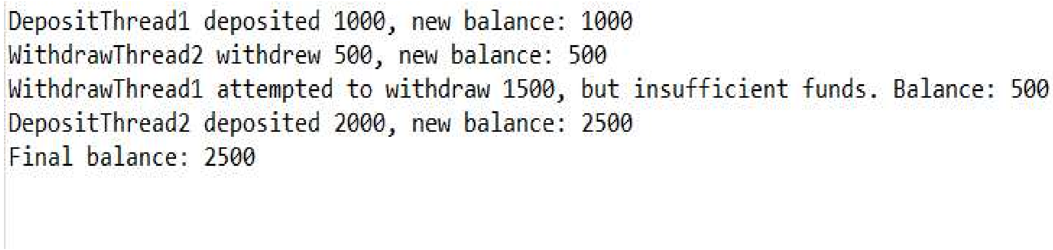
```
<terminated> ProducerConsumer [Java Applicatic
From Prod :0
From COnsumer : 0
From Prod :1
From COnsumer : 1
From Prod :2
From COnsumer : 2
From Prod :3
From COnsumer : 3
From Prod :4
From COnsumer : 4
From Prod :5
From COnsumer : 5
From Prod :6
From COnsumer : 6
From Prod :7
From COnsumer : 7
From Prod :8
From COnsumer : 8
From Prod :9
From COnsumer : 9
From Prod :10
From COnsumer : 10
```

Task 4: Synchronized blocks and Methods

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

```
package practice;
public class BankAccountDemo {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();

    Thread depositThread1 = new Thread(new DepositTask(account, 1000), "DepositThread1");
    Thread depositThread2 = new Thread(new DepositTask(account, 2000), "DepositThread2");

    Thread withdrawThread1 = new Thread(new WithdrawTask(account, 1500), "WithdrawThread1");
    Thread withdrawThread2 = new Thread(new WithdrawTask(account, 500), "WithdrawThread2");

    depositThread1.start();
    depositThread2.start();
    withdrawThread1.start();
    withdrawThread2.start();
    try {
      depositThread1.join();
      depositThread2.join();
      withdrawThread1.join();
      withdrawThread2.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Final balance: " + account.getBalance());
  }
}
class BankAccount {
  private int balance = 0;
  public synchronized void deposit(int amount) {
    balance += amount;
    System.out.println(Thread.currentThread().getName() + " deposited " + amount + ", new balance:
" + balance);
  }
  public synchronized void withdraw(int amount) {
    if (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. Balance: " + balance);
    }
  }
  public int getBalance() {
    return balance;
  }
}
class DepositTask implements Runnable {
  private final BankAccount account;
  private final int amount;
  public DepositTask(BankAccount account, int amount) {
    this.account = account;
    this.amount = amount;
  }
  @Override
  public void run() {
    account.deposit(amount);
  }
}
class WithdrawTask implements Runnable {
  private final BankAccount account;
  private final int amount;
  public WithdrawTask(BankAccount account, int amount) {
    this.account = account;
    this.amount = amount;
  }
  @Override
  public void run() {
    account.withdraw(amount);
  }
}
```

Output:

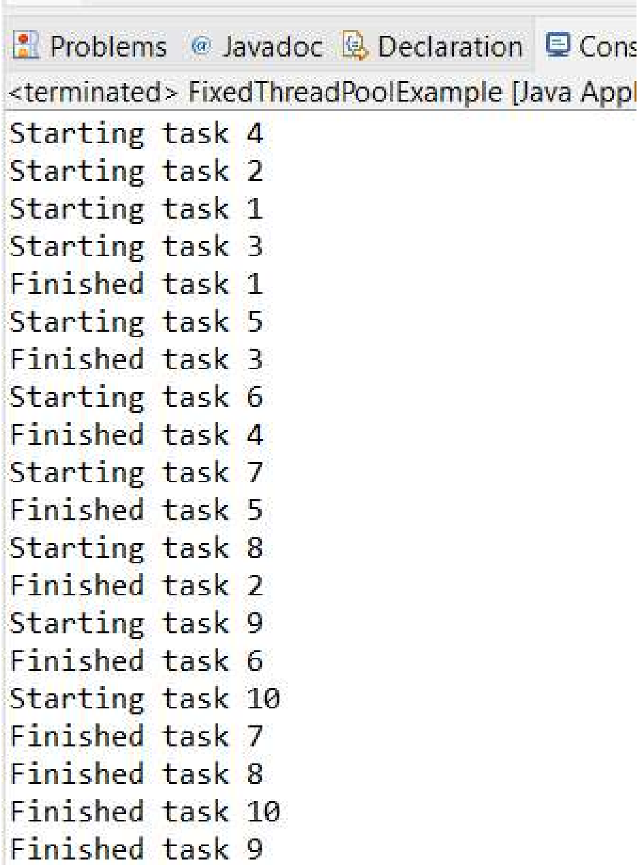


```
DepositThread1 deposited 1000, new balance: 1000
WithdrawThread2 withdrew 500, new balance: 500
WithdrawThread1 attempted to withdraw 1500, but insufficient funds. Balance: 500
DepositThread2 deposited 2000, new balance: 2500
Final balance: 2500
```

Task 5: Thread pool 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 practice;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class FixedThreadPoolExample {
  public static void main(String[] args) throws Exception {

    int numThreads = 4;
    int numTasks = 10;
    ExecutorService executor = Executors.newFixedThreadPool(numThreads);
    class ComplexTask implements Runnable {
      private final int number;
      public ComplexTask(int number) {
        this.number = number;
      }
      @Override
      public void run() {
        try {

          System.out.println("Starting task " + number);
          Thread.sleep((long) (Math.random() * 3000));
System.out.println("Finished task " + number);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    }

    Future<?>[] futures = new Future<?>[numTasks];
    for (int i = 0; i < numTasks; i++) {
      futures[i] = executor.submit(new ComplexTask(i + 1));
    }
    for (Future<?> future : futures) {
      try {
        future.get();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    executor.shutdown();
    executor.awaitTermination(10, TimeUnit.SECONDS);
  }
}
```



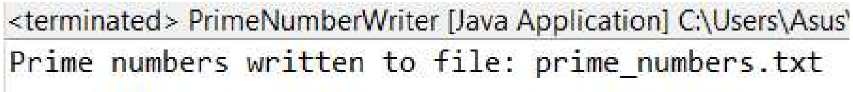
```
<terminated> FixedThreadPoolExample [Java Appl
Starting task 4
Starting task 2
Starting task 1
Starting task 3
Finished task 1
Starting task 5
Finished task 3
Starting task 6
Finished task 4
Starting task 7
Finished task 5
Starting task 8
Finished task 2
Starting task 9
Finished task 6
Starting task 10
Finished task 7
Finished task 8
Finished task 10
Finished task 9
```

Task 6: Executors, Concurrent Collections, CompletableFuture

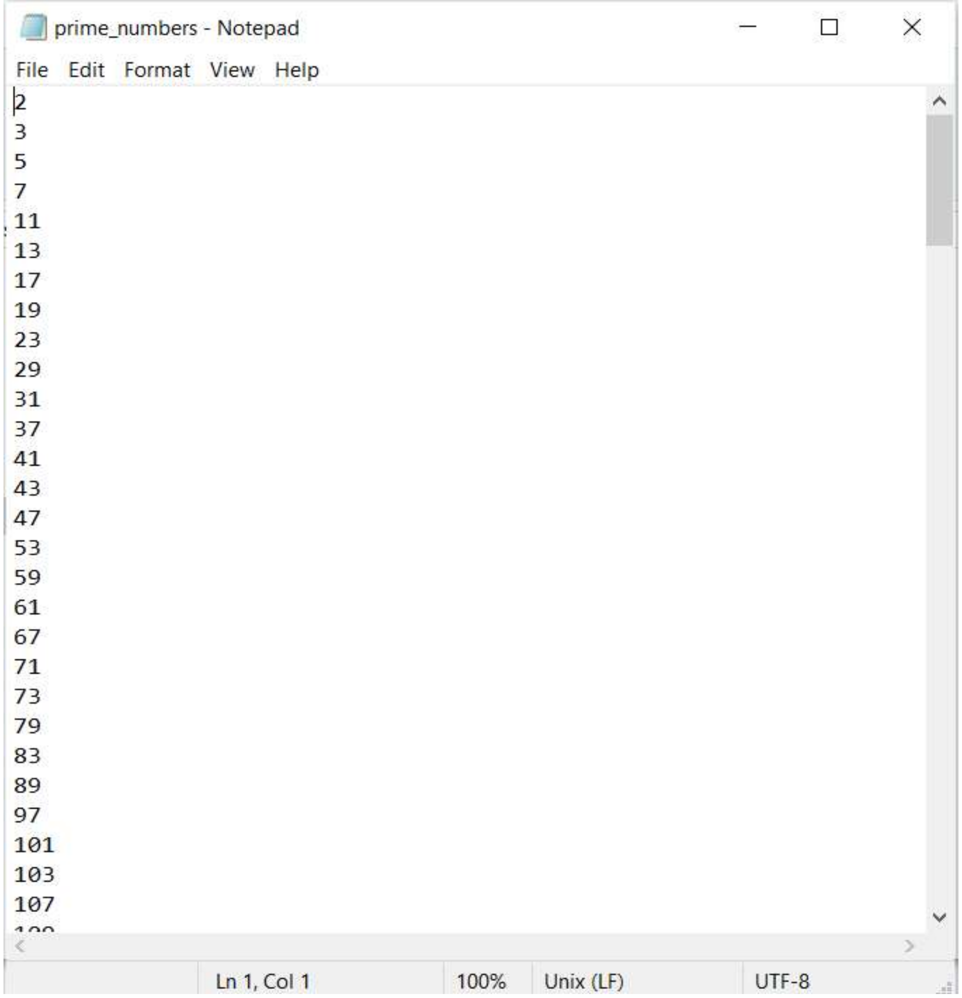
Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

```
package practice;
import java.io.BufferedWriter;
import java.io.FileWriter;
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.Future;
import java.util.concurrent.TimeUnit;
import java.util.stream.IntStream;
public class PrimeNumberWriter {
    private static final int NUM_THREADS = 4;
    private static final String FILE_NAME = "prime_numbers.txt";
    public static void main(String[] args) throws Exception {
        int upperLimit = 1000;
        List<Future<List<Integer>>> primeNumberFutures = calculatePrimes(upperLimit);

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

        writePrimesToFileAsync(allPrimes);
        System.out.println("Prime numbers written to file: " + FILE_NAME);
    }
    private static List<Future<List<Integer>>> calculatePrimes(int upperLimit) throws Exception
{
        ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
        List<Future<List<Integer>>> futures = new ArrayList<>();
        int chunkSize = upperLimit / NUM_THREADS;
        for (int i = 0; i < upperLimit; i += chunkSize) {
            int start = i;
            int end = Math.min(start + chunkSize, upperLimit);
            futures.add(executor.submit(() -> findPrimesInRange(start, end)));
        }
        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.SECONDS);
        return futures;
    }
    private static List<Integer> findPrimesInRange(int start, int end) {
        List<Integer> primes = new ArrayList<>();
        for (int num = start; num <= end; num++) {
            if (isPrime(num)) {
                primes.add(num);
            }
        }
        return primes;
    }
    private static boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }
        for (int i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                return false;
            }
        }
        return true;
```

```java
  }
  private static void writePrimesToFileAsync(List<Integer> primes) throws Exception {
    CompletableFuture<Void> writeFuture = CompletableFuture.supplyAsync(() -> {
      try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME))) {
        for (int prime : primes) {
          writer.write(String.valueOf(prime) + "\n");
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
      return null;
    });
    writeFuture.get();
  }
}
```



```
<terminated> PrimeNumberWriter [Java Application] C:\Users\Asus\
Prime numbers written to file: prime_numbers.txt
```



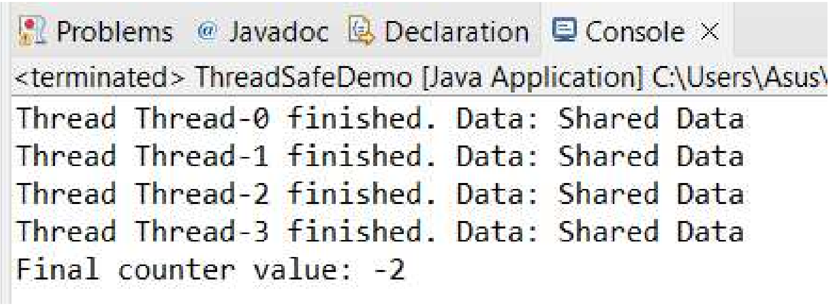
```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
101
103
107
```

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.

```java
package practice;
import java.util.concurrent.atomic.AtomicInteger;
class ThreadSafeCounter {
  private final AtomicInteger count;
  public ThreadSafeCounter() {
    this.count = new AtomicInteger(0);
  }
  public void increment() {
    count.incrementAndGet();
  }
  public void decrement() {
    count.decrementAndGet();
  }
  public int get() {
    return count.get();
  }
}
class ImmutableData {
  private final String data;
  public ImmutableData(String data) {
    this.data = data;
  }
  public String getData() {
    return data;
  }
}
public class ThreadSafeDemo {
  public static void main(String[] args) {
    ThreadSafeCounter counter = new ThreadSafeCounter();
    ImmutableData data = new ImmutableData("Shared Data");
    int numThreads = 4;
    for (int i = 0; i < numThreads; i++) {
      Thread thread = new Thread(() -> {
        for (int j = 0; j < 1000; j++) {
          if (Math.random() > 0.5) {
            counter.increment();
          } else {
            counter.decrement();
          }
        }
        System.out.println("Thread " + Thread.currentThread().getName() + " finished. Data: " +
data.getData());
      });
      thread.start();
    }
    for (int i = 0; i < numThreads; i++) {
      try {
```

```
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Final counter value: " + counter.get());
    }
}
```



```
<terminated> ThreadSafeDemo [Java Application] C:\Users\Asus\
Thread Thread-0 finished. Data: Shared Data
Thread Thread-1 finished. Data: Shared Data
Thread Thread-2 finished. Data: Shared Data
Thread Thread-3 finished. Data: Shared Data
Final counter value: -2
```