

### 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

Ans:

```
package Assignments;
```

```
public 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);
```

```
            } catch (InterruptedException e) {
```

```
                System.err.println(threadName + " interrupted.");
```

```
            }
```

```
        }
```

```
    }
```

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

```
        Thread thread1 = new Thread(new NumberPrinter("Thread 1"));
```

```
        Thread thread2 = new Thread(new NumberPrinter("Thread 2"));
```

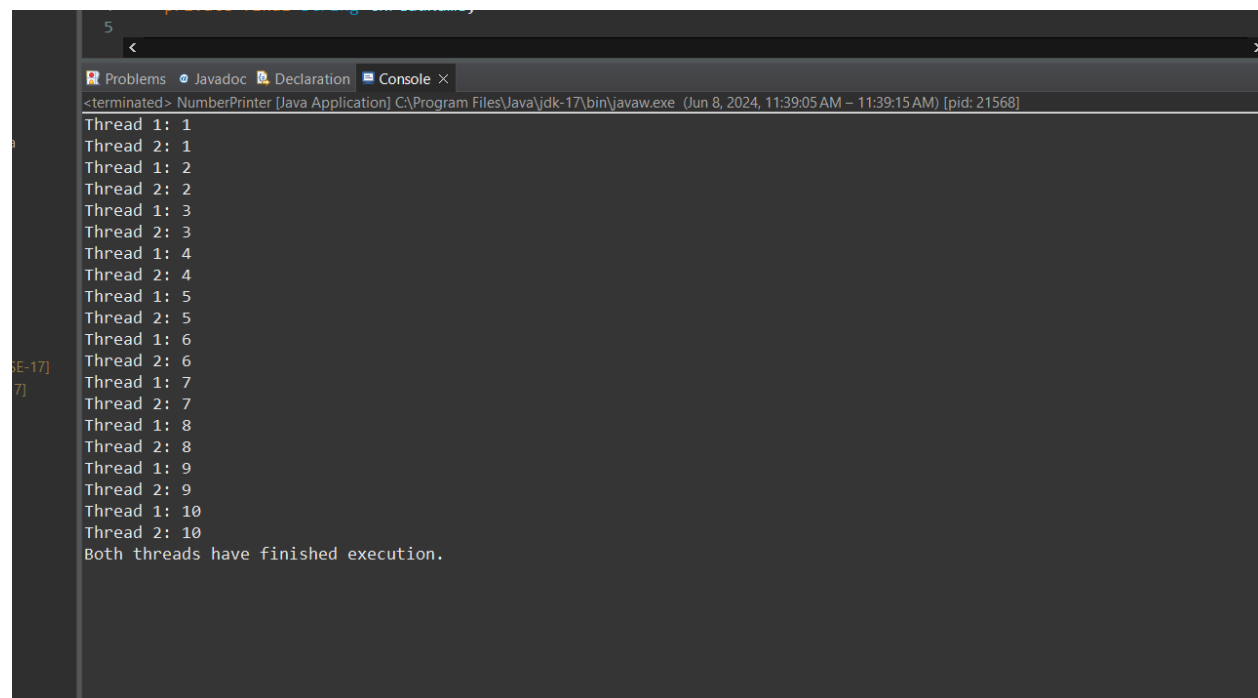
```
thread1.start();

thread2.start();

try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    System.err.println("Main thread interrupted.");
}

System.out.println("Both threads have finished execution.");
}
```

Output:



```
<terminated> NumberPrinter [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Jun 8, 2024, 11:39:05 AM – 11:39:15 AM) [pid: 21568]
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5
Thread 1: 6
Thread 2: 6
Thread 1: 7
Thread 2: 7
Thread 1: 8
Thread 2: 8
Thread 1: 9
Thread 2: 9
Thread 1: 10
Thread 2: 10
Both threads have finished execution.
```

## 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..

Ans:

```
package Assignments;
```

```
public class ThreadLifecycle implements Runnable {
```

```
    private final Object lock = new Object();
```

```
    @Override
```

```
    public void run() {
```

```
        try {
```

```
            System.out.println("Thread is in RUNNABLE state.");
```

```
            System.out.println("Thread is going to sleep for 2 seconds (TIMED_WAITING state).");
```

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

```
            synchronized (lock) {
```

```
                System.out.println("Thread is waiting on lock (WAITING state).");
```

```
                lock.wait();
```

```
            }
```

```
            System.out.println("Thread is back to RUNNABLE state after being notified.");
```

```
synchronized (lock) {  
    System.out.println("Thread is going to wait on lock for 2 seconds (TIMED_WAITING state).");  
    lock.wait(2000);  
}
```

```
System.out.println("Thread is trying to acquire lock (BLOCKED state simulation).");  
synchronized (lock) {  
    System.out.println("Thread acquired the lock (RUNNABLE state).");  
}
```

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

```
System.out.println("Thread is in TERMINATED state.");  
}
```

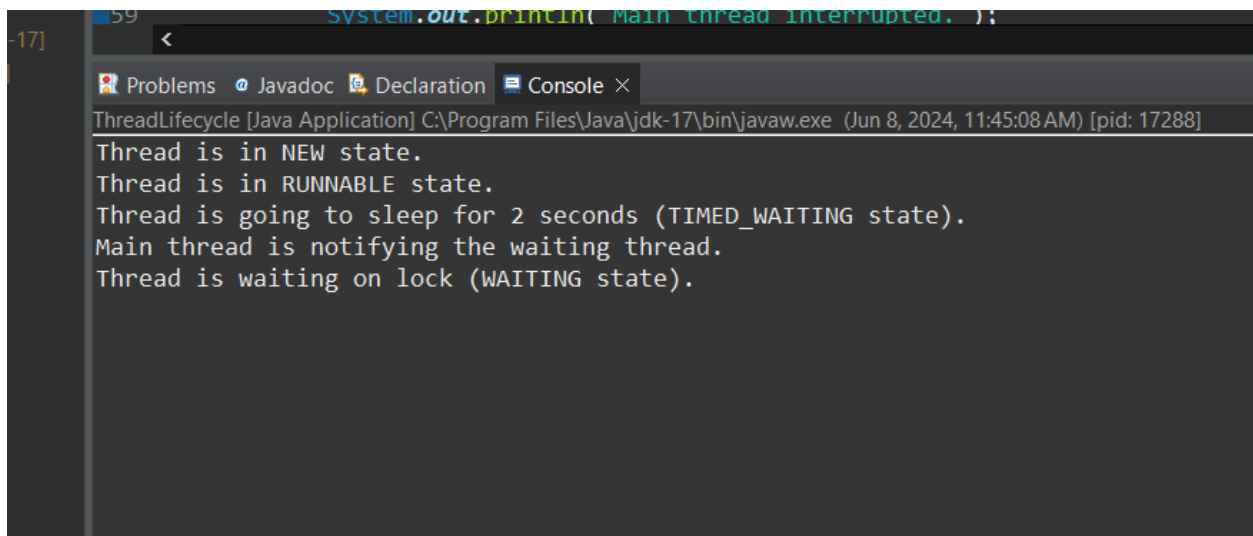
```
public static void main(String[] args) {  
    ThreadLifecycle runnableInstance = new ThreadLifecycle();  
    Thread thread = new Thread(runnableInstance);  
    System.out.println("Thread is in NEW state.");  
    thread.start();  
  
    try {  
        Thread.sleep(1000);  
        synchronized (runnableInstance.lock) {
```

```

        System.out.println("Main thread is notifying the waiting thread.");
        runnableInstance.lock.notify();
    }
    thread.join();
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
}
}

```

Output:



The screenshot shows a Java IDE window with a console tab. The console output is as follows:

```

ThreadLifecycle [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Jun 8, 2024, 11:45:08 AM) [pid: 17288]
Thread is in NEW state.
Thread is in RUNNABLE state.
Thread is going to sleep for 2 seconds (TIMED_WAITING state).
Main thread is notifying the waiting thread.
Thread is waiting on lock (WAITING state).

```

### 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.

Ans:

```
package com.wipro;
```

```
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 Producer: "+this.num);

    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    available=true; //imp var
    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 Producer_Consumer {

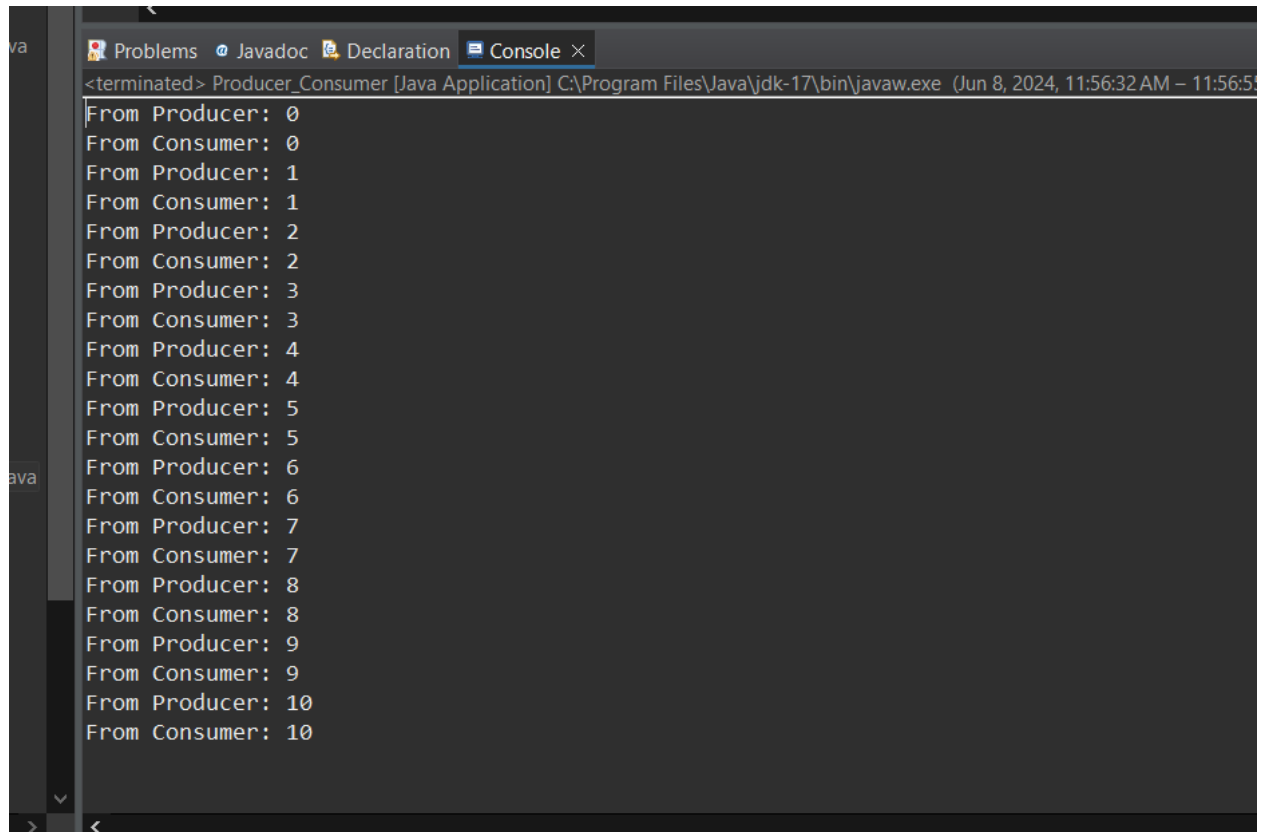
    public static void main(String[] args) {
        Common c=new Common();
        new Producer(c);
    }
}

```



```
        new Consumer(c);
    }
}
```

OUTPUT:



```
<terminated> Producer_Consumer [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Jun 8, 2024, 11:56:32 AM - 11:56:55 AM)
From Producer: 0
From Consumer: 0
From Producer: 1
From Consumer: 1
From Producer: 2
From Consumer: 2
From Producer: 3
From Consumer: 3
From Producer: 4
From Consumer: 4
From Producer: 5
From Consumer: 5
From Producer: 6
From Consumer: 6
From Producer: 7
From Consumer: 7
From Producer: 8
From Consumer: 8
From Producer: 9
From Consumer: 9
From Producer: 10
From Consumer: 10
```

#### 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.

```
package Assignments;
```

```
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 + ", New
Balance: " + balance);
    }
}

public synchronized void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;

        System.out.println(Thread.currentThread().getName() + " withdrew: " + amount + ", New Balance:
" + balance);
    } else {
        System.out.println(Thread.currentThread().getName() + " attempted to withdraw: " + amount + ",
Insufficient Funds");
    }
}

public synchronized double getBalance() {
    return balance;
}

class BankCustomer implements Runnable {

```

```

private BankAccount account;

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

@Override
public void run() {
    for (int i = 0; i < 5; i++) {
        double amount = Math.random() * 100;
        if (Math.random() > 0.5) {
            account.deposit(amount);
        } else {
            account.withdraw(amount);
        }
        try {
            Thread.sleep((int)(Math.random() * 1000)); // Simulate time taken for transactions
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class BankSimulation {

    public static void main(String[] args) {

        BankAccount sharedAccount = new BankAccount(1000);

        Thread customer1 = new Thread(new BankCustomer(sharedAccount), "Customer 1");
    }
}

```

```
Thread customer2 = new Thread(new BankCustomer(sharedAccount), "Customer 2");
```

```
Thread customer3 = new Thread(new BankCustomer(sharedAccount), "Customer 3");
```

```
customer1.start();
```

```
customer2.start();
```

```
customer3.start();
```

```
try {
```

```
    customer1.join();
```

```
    customer2.join();
```

```
    customer3.join();
```

```
} catch (InterruptedException e) {
```

```
    e.printStackTrace();
```

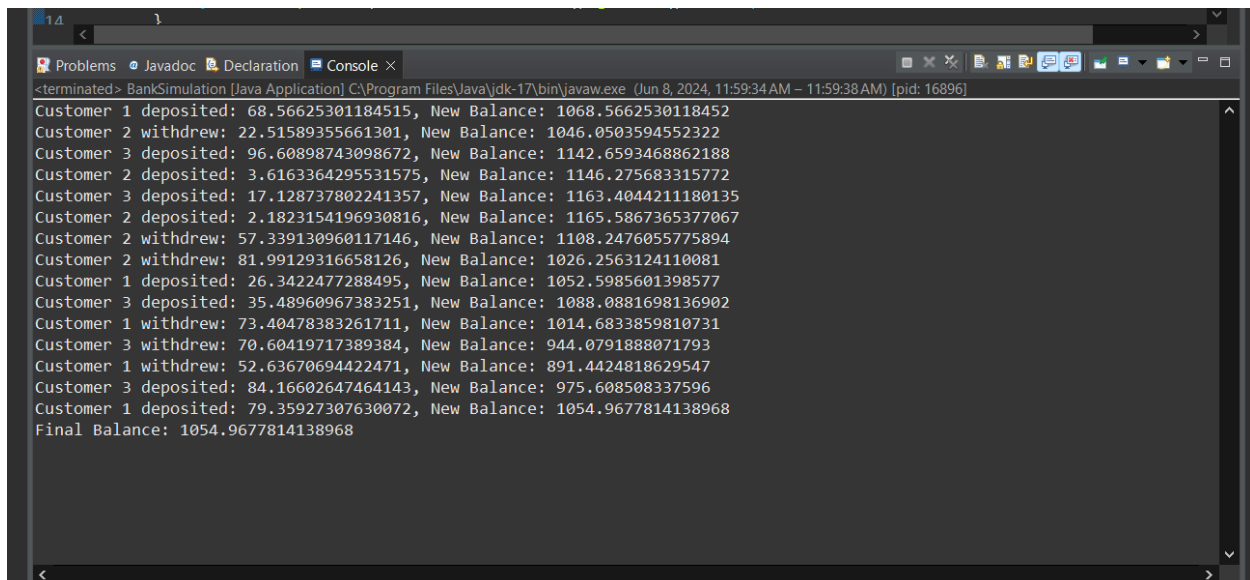
```
}
```

```
System.out.println("Final Balance: " + sharedAccount.getBalance());
```

```
}
```

```
}
```

Output:



```
<terminated> BankSimulation [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Jun 8, 2024, 11:59:34 AM - 11:59:38 AM) [pid: 16896]
Customer 1 deposited: 68.56625301184515, New Balance: 1068.5662530118452
Customer 2 withdrew: 22.51589355661301, New Balance: 1046.0503594552322
Customer 3 deposited: 96.60898743098672, New Balance: 1142.6593468862188
Customer 2 deposited: 3.6163364295531575, New Balance: 1146.275683315772
Customer 3 deposited: 17.128737802241357, New Balance: 1163.4044211180135
Customer 2 deposited: 2.1823154196930816, New Balance: 1165.5867365377067
Customer 2 withdrew: 57.339130960117146, New Balance: 1108.2476055775894
Customer 2 withdrew: 81.99129316658126, New Balance: 1026.2563124110081
Customer 1 deposited: 26.3422477288495, New Balance: 1052.5985601398577
Customer 3 deposited: 35.48960967383251, New Balance: 1088.0881698136902
Customer 1 withdrew: 73.40478383261711, New Balance: 1014.6833859810731
Customer 3 withdrew: 70.60419717389384, New Balance: 944.0791888071793
Customer 1 withdrew: 52.63670694422471, New Balance: 891.4424818629547
Customer 3 deposited: 84.16602647464143, New Balance: 975.608508337596
Customer 1 deposited: 79.35927307630072, New Balance: 1054.9677814138968
Final Balance: 1054.9677814138968
```

### 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.

Ans:

```
package Assignments;
```

```
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
```

```
import java.util.concurrent.TimeUnit;
```

```
public class ThreadPoolExample {
```

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

```
        ExecutorService executorService = Executors.newFixedThreadPool(4);
```

```
        for (int i = 0; i < 10; i++) {
```

```
            int taskId = i;
```

```
            executorService.submit(() -> {
```

```
                performTask(taskId);
```

```
            });
```

```
        }
```

```
        executorService.shutdown();
```

```
        try {
```

```
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
```

```
                executorService.shutdownNow();
```

```
    }  
    } catch (InterruptedException e) {  
        executorService.shutdownNow();  
    }  
}
```

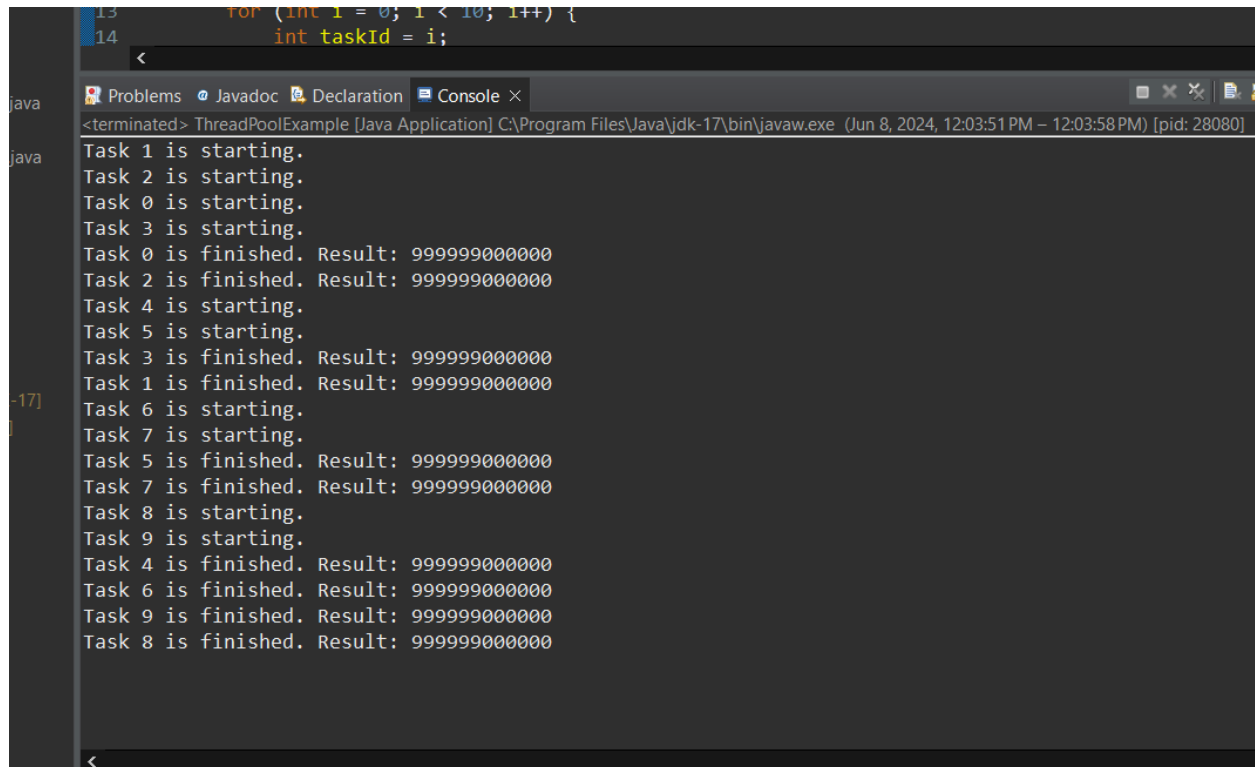
```
private static void performTask(int taskId) {  
    System.out.println("Task " + taskId + " is starting.");
```

```
    long result = 0;  
    for (int i = 0; i < 1000000; i++) {  
        result += i * 2;  
    }
```

```
    try {  
        TimeUnit.SECONDS.sleep(2);  
    } catch (InterruptedException e) {  
        Thread.currentThread().interrupt();  
    }
```

```
    System.out.println("Task " + taskId + " is finished. Result: " + result);  
}  
}
```

Output:



```
13     for (int i = 0; i < 10; i++) {
14         int taskId = i;
        <
    java Problems Javadoc Declaration Console x
    <terminated> ThreadPoolExample [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Jun 8, 2024, 12:03:51 PM – 12:03:58 PM) [pid: 28080]
    java Task 1 is starting.
    Task 2 is starting.
    Task 0 is starting.
    Task 3 is starting.
    Task 0 is finished. Result: 999999000000
    Task 2 is finished. Result: 999999000000
    Task 4 is starting.
    Task 5 is starting.
    Task 3 is finished. Result: 999999000000
    Task 1 is finished. Result: 999999000000
    -17] Task 6 is starting.
    Task 7 is starting.
    Task 5 is finished. Result: 999999000000
    Task 7 is finished. Result: 999999000000
    Task 8 is starting.
    Task 9 is starting.
    Task 4 is finished. Result: 999999000000
    Task 6 is finished. Result: 999999000000
    Task 9 is finished. Result: 999999000000
    Task 8 is finished. Result: 999999000000
    <
```

## 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.

Ans:

```
package Assignments;
```

```
import java.io.IOException;
```

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

```
import java.nio.file.Paths;
```

```
import java.util.List;
```

```
import java.util.concurrent.CompletableFuture;
```

```
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;
```

```

import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class PrimeNumberCalculator {

    public static void main(String[] args) {

        int maxNumber = 100;

        String filePath = "file.txt";

        ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        try {

            CompletableFuture<List<Integer>> future = CompletableFuture.supplyAsync(() ->
calculatePrimes(maxNumber), executor);

            future.thenAccept(primes -> writeToFile(primes, filePath))
                .exceptionally(ex -> {
                    System.err.println("Error: " + ex.getMessage());
                    return null;
                }).join();
        } finally {

            executor.shutdown();
        }
    }
}

```



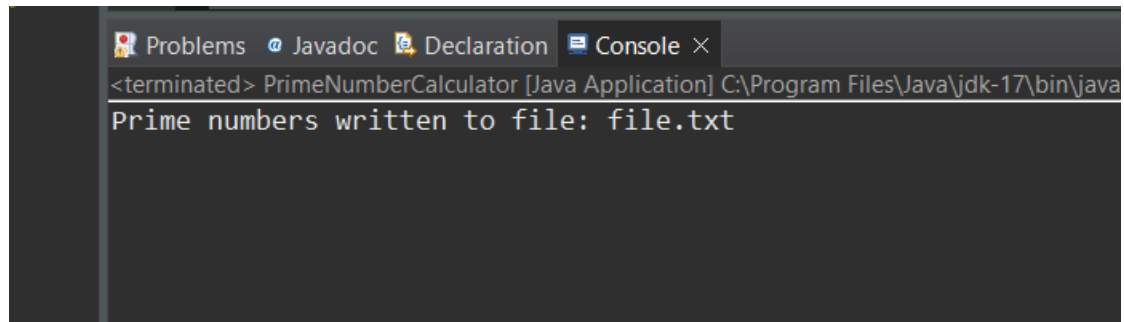
```
private static List<Integer> calculatePrimes(int maxNumber) {  
    return IntStream.rangeClosed(2, maxNumber)  
        .parallel()  
        .filter(PrimeNumberCalculator::isPrime)  
        .boxed()  
        .collect(Collectors.toList());  
}
```

```
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 writeToFile(List<Integer> primes, String filePath) {  
    try {  
        Files.write(Paths.get(filePath), primes.stream()  
            .map(String::valueOf)  
            .collect(Collectors.toList()));  
        System.out.println("Prime numbers written to file: " + filePath);  
    } catch (IOException e) {  
        System.err.println("Error writing to file: " + e.getMessage());  
    }  
}
```

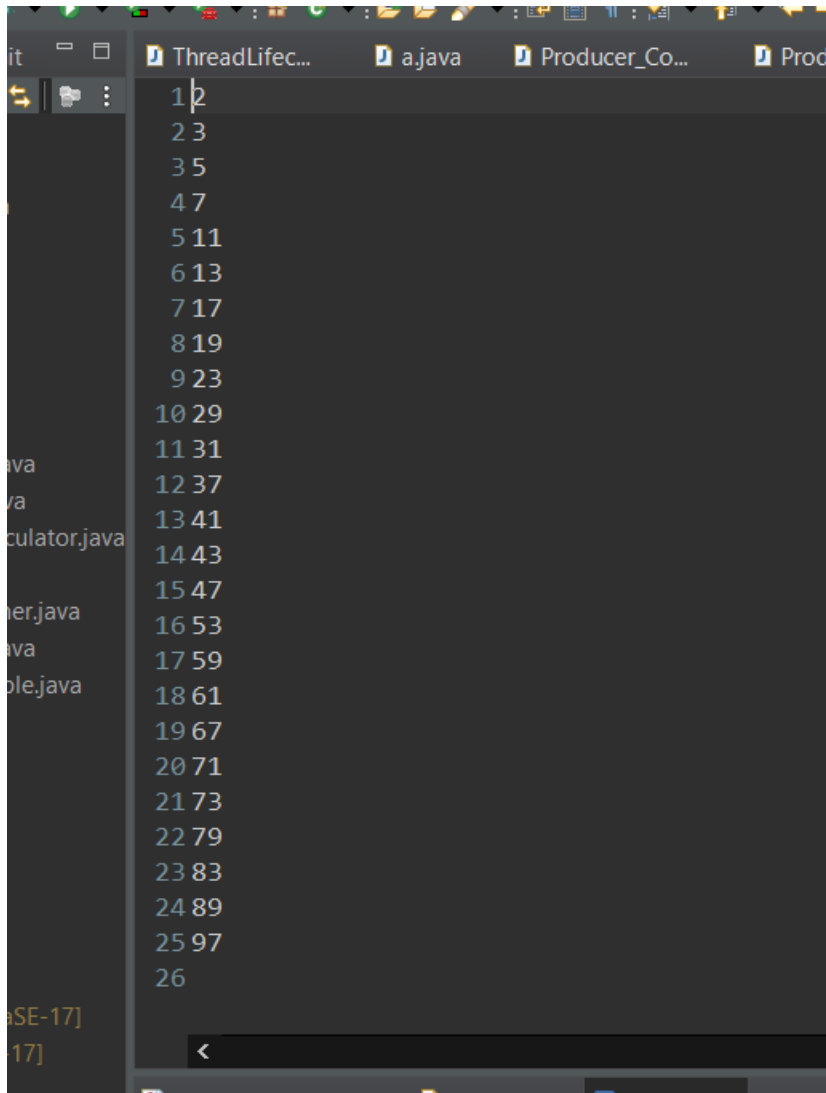
```
    }  
}  
}
```

Output:



The screenshot shows an IDE's console window with a dark background. The title bar at the top contains four tabs: 'Problems' with a bug icon, 'Javadoc' with a '@' icon, 'Declaration' with a magnifying glass icon, and 'Console' with a speech bubble icon and a close button 'X'. The console text area shows the following output: a green status line '<terminated> PrimeNumberCalculator [Java Application] C:\Program Files\Java\jdk-17\bin\java' followed by the message 'Prime numbers written to file: file.txt' in a light blue color.

```
<terminated> PrimeNumberCalculator [Java Application] C:\Program Files\Java\jdk-17\bin\java  
Prime numbers written to file: file.txt
```



### 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.

Ans:

```
package Assignments;
```

```
public class Counter {
```

```
    private int count;
```

```
public Counter(int initialCount) {  
    this.count = initialCount;  
}
```

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

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

```
public synchronized int getCount() {  
    return count;  
}  
}
```

```
package Assignments;
```

```
public final class ImmutableData {  
    private final int value;  
  
    public ImmutableData(int value) {  
        this.value = value;  
    }  
}
```

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

```
package Assignments;
```

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

```
public class Main {  
    public static void main(String[] args) {  
        Counter counter = new Counter(0);  
        ImmutableData immutableData = new ImmutableData(100);  
  
        ExecutorService executor = Executors.newFixedThreadPool(10);  
  
        for (int i = 0; i < 5; i++) {  
            executor.submit(() -> {  
                for (int j = 0; j < 1000; j++) {  
                    counter.increment();  
                }  
            });  
        }  
    }  
}
```

```

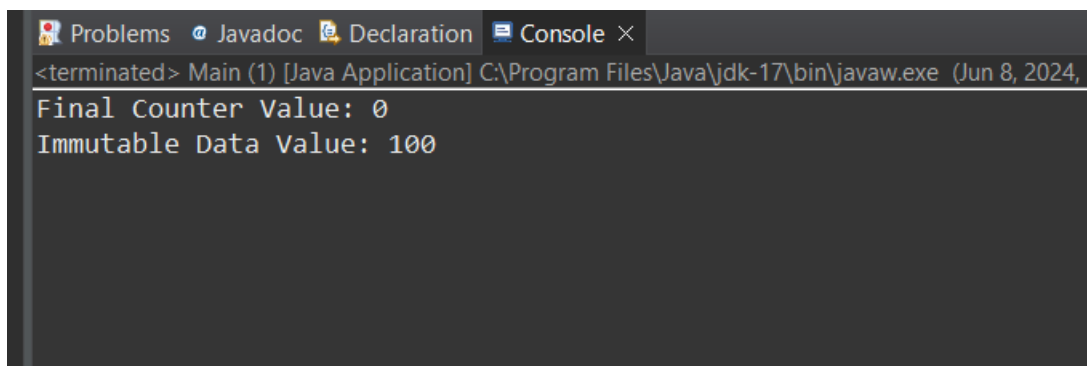
        executor.submit(() -> {
            for (int j = 0; j < 1000; j++) {
                counter.decrement();
            }
        });
    }

    executor.shutdown();
    try {
        if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
            executor.shutdownNow();
        }
    } catch (InterruptedException e) {
        executor.shutdownNow();
    }

    System.out.println("Final Counter Value: " + counter.getCount());
    System.out.println("Immutable Data Value: " + immutableData.getValue());
}
}

```

Output:



```

<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (Jun 8, 2024,
Final Counter Value: 0
Immutable Data Value: 100

```