

Name: Vaishali Ramesh Kale

Email ID: kalevaishalir16@gmail.com

Assignments 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:::

Explanation::

1. Main Class (ThreadExample):
 - The main method creates two threads using the Thread class.
 - Each thread is initialized with an instance of the NumberPrinter class which implements the Runnable interface.
 - The start method is called on each thread to begin execution.
2. NumberPrinter Class:
 - Implements the Runnable interface, which requires the run method to be defined.
 - Inside the run method, a loop prints numbers from 1 to 10 with a 1-second delay between each number using Thread.sleep(1000).
 - The Thread.currentThread().getName() method is used to print the name of the current thread, allowing us to see which thread is printing which numbers.

CODE:::

```
public class ThreadExample {  
    public static void main(String[] args) {  
        // Create two threads  
        Thread thread1 = new Thread(new NumberPrinter(), "Thread-1");  
        Thread thread2 = new Thread(new NumberPrinter(), "Thread-2");  
        // Start the threads  
        thread1.start();  
        thread2.start();  
    }  
}
```

```
// Class that implements Runnable to define the task for the threads
class NumberPrinter implements Runnable {

    @Override
    public void run() {

        try {

            // Print numbers from 1 to 10 with a 1-second delay
            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() + " was interrupted.");

        }

    }

}
```

OUTPUT:::

```
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

...Program finished with exit code 0
Press ENTER to exit console.
```

```
*****
*****
```

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::::

Explanation

1. Main Class (ThreadLifecycleExample):
 - Defines an Object named monitor which will be used to demonstrate the WAITING and NOTIFY states.
 - Creates a new thread (DemoThread) which initially is in the NEW state.
2. Thread (DemoThread):
 - Once started, it enters the RUNNABLE state.
 - Calls Thread.sleep(1000) to transition to the TIMED_WAITING state for 1 second.

- Enters a synchronized block on monitor and calls `monitor.wait()`, transitioning to the WAITING state until `notify()` is called.
- After being notified, it continues execution and is in the RUNNABLE state again.

3. Main Thread:

- After starting `DemoThread`, it sleeps for 500 milliseconds to allow `DemoThread` to enter the TIMED_WAITING state.
- Notifies `DemoThread` by calling `monitor.notify()`, moving it from the WAITING state back to RUNNABLE.
- Calls `thread.join()` to wait for `DemoThread` to complete, ensuring it transitions to the TERMINATED state.

CODE::::

```
public class ThreadLifecycleExample {

    public static void main(String[] args) {
        final Object monitor = new Object();

        // Create a new thread (NEW state)
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println(Thread.currentThread().getName() + " is RUNNABLE");

                    // Sleep for 1 second (TIMED_WAITING state)
                    Thread.sleep(1000);

                    synchronized (monitor) {
                        System.out.println(Thread.currentThread().getName() + " is WAITING");
                        // Wait until another thread invokes notify() on the monitor object (WAITING
state)
                        monitor.wait();
                    }
                }
            }
        });
    }
}
```

```

        // Continue execution after being notified
        System.out.println(Thread.currentThread().getName() + " is RUNNABLE again");

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

}, "DemoThread");

// Thread is in NEW state
System.out.println(thread.getName() + " is NEW");

// Start the thread (RUNNABLE state)
thread.start();

// Ensure the main thread waits enough for DemoThread to actually enter WAITING
state
try {
    Thread.sleep(1500); // 1 second for the DemoThread to sleep + 500 ms buffer
} catch (InterruptedException e) {
    e.printStackTrace();
}

// Main thread attempts to notify DemoThread (should be in WAITING state)
synchronized (monitor) {
    System.out.println("Main thread is notifying DemoThread");
    // Notify DemoThread (make it RUNNABLE again)
    monitor.notify();
}

```

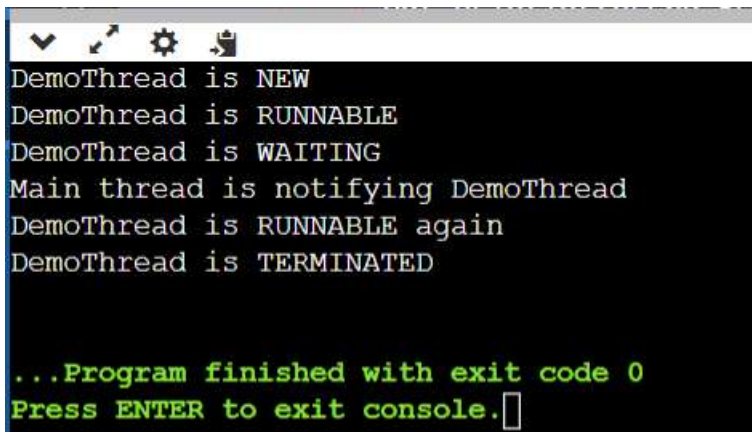
```

// Wait for DemoThread to complete (TERMINATED state)
try {
    thread.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

// Now DemoThread should be TERMINATED
System.out.println(thread.getName() + " is TERMINATED");
}
}

```

OUTPUT::::



```

DemoThread is NEW
DemoThread is RUNNABLE
DemoThread is WAITING
Main thread is notifying DemoThread
DemoThread is RUNNABLE again
DemoThread is TERMINATED

...Program finished with exit code 0
Press ENTER to exit console.

```


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::::

Explanation:

3. Shared Buffer and Synchronization:
 - The shared buffer is implemented using a LinkedList.
 - The maxSize variable represents the maximum capacity of the buffer.

- The produce and consume methods are synchronized to ensure that only one thread can execute them at a time.
4. Producer Method (produce):
 - Checks if the buffer is full using while (buffer.size() == maxSize).
 - If the buffer is full, the producer thread calls wait() to release the lock and go into the waiting state.
 - Once there is space in the buffer, the producer adds an item to the buffer, prints a message, and calls notifyAll() to wake up all waiting threads.
 5. Consumer Method (consume):
 - Checks if the buffer is empty using while (buffer.isEmpty()).
 - If the buffer is empty, the consumer thread calls wait() to release the lock and go into the waiting state.
 - Once there is an item in the buffer, the consumer removes an item from the buffer, prints a message, and calls notifyAll() to wake up all waiting threads.
 6. Main Method:
 - Creates an instance of ProducerConsumer with a buffer size of 5.
 - Creates and starts the producer and consumer threads.
 - Both threads run indefinitely, producing and consuming items with simulated delays using Thread.sleep().
 - This implementation ensures that the producer and consumer operate in a synchronized manner, with the producer waiting when the buffer is full and the consumer waiting when the buffer is empty. The notifyAll() method is used to wake up all waiting threads when the state of the buffer changes.

CODE:::

```
class Common {
    int num;

    boolean available = false;

    public int put(int num) {
        synchronized(this) {
            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();
    }

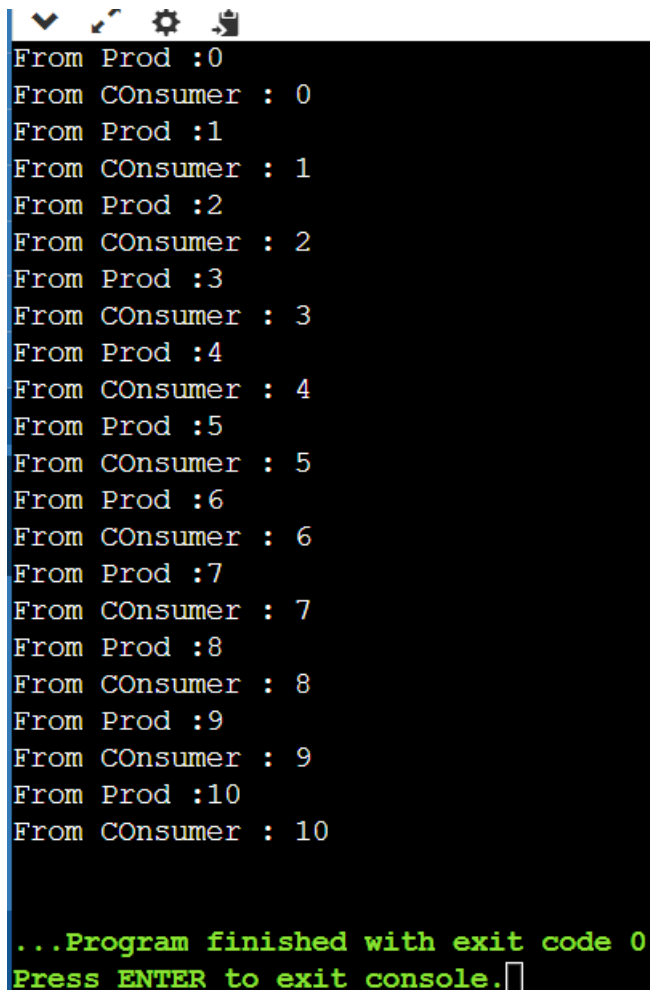
    public void run() {
        int x = 0;
        while (x <= 10) {
            c.get();
            x++;
        }
    }
}

public class PC {

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

```

OUTPUT::::



```

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

...Program finished with exit code 0
Press ENTER to exit console.

```

```

*****
*****

```

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:::

Explanation:

1. BankAccount Class:
 - This class has a private balance variable to keep track of the account balance.
 - The deposit and withdraw methods are synchronized to ensure that only one thread can access them at a time, preventing race conditions.
 - The getBalance method returns the current balance.
2. DepositTask Class:

- This class implements Runnable and is used to create deposit tasks for the threads.
- The run method calls the deposit method on the bank account.

3. WithdrawTask Class:

- This class implements Runnable and is used to create withdrawal tasks for the threads.
- The run method calls the withdraw method on the bank account.

4. BankSimulation Class:

- This class contains the main method to simulate the bank account operations.
- A BankAccount object is created with an initial balance.
- Four threads (t1, t2, t3, t4) are created to perform deposit and withdrawal operations.
- The start method is called on each thread to begin execution.
- The join method is used to ensure the main thread waits for all threads to complete before printing the final balance.

5. Execution Flow:

- t1 deposits 500.
- t2 withdraws 300.
- t3 deposits 200.
- t4 attempts to withdraw 800, which may fail if the balance is insufficient.
- Thread Safety:
 - The use of synchronized methods ensures that the balance updates are atomic and thread-safe.
 - This prevents race conditions where multiple threads might simultaneously read and modify the balance, leading to inconsistent results.

CODE:::

```
class BankAccount {
    private int balance;

    public BankAccount(int initialBalance) {
        this.balance = initialBalance;
    }

    // Synchronized method for depositing money
    public synchronized void deposit(int amount) {
        balance += amount;
    }
}
```

```

        System.out.println(Thread.currentThread().getName() + " deposited " + amount + ".
New balance: " + balance);
    }

    // Synchronized method for withdrawing money
    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() + " tried to withdraw " +
amount + " but insufficient balance. Current 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);  
}  
}
```

```
public class BankSimulation {  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount(1000);  
  
        Thread t1 = new Thread(new DepositTask(account, 500), "Thread-1");  
        Thread t2 = new Thread(new WithdrawTask(account, 300), "Thread-2");  
        Thread t3 = new Thread(new DepositTask(account, 200), "Thread-3");  
        Thread t4 = new Thread(new WithdrawTask(account, 800), "Thread-4");  
    }  
}
```

```

t1.start();

t2.start();

t3.start();

t4.start();


try {
    t1.join();

    t2.join();

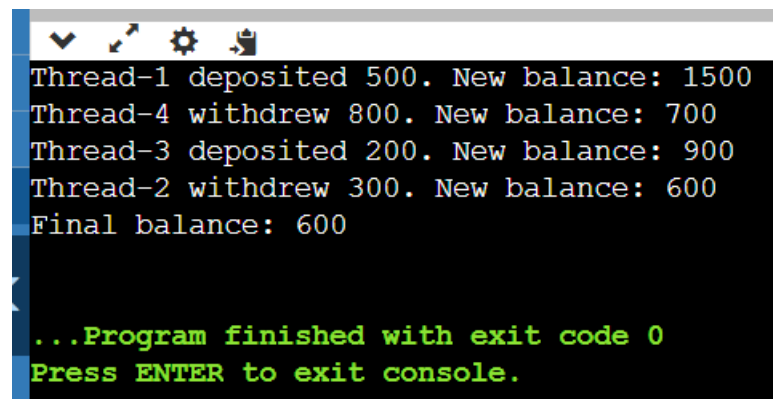
    t3.join();

    t4.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}


System.out.println("Final balance: " + account.getBalance());
}
}

```

OUTPUT:::



```

Thread-1 deposited 500. New balance: 1500
Thread-4 withdrew 800. New balance: 700
Thread-3 deposited 200. New balance: 900
Thread-2 withdrew 300. New balance: 600
Final balance: 600

...Program finished with exit code 0
Press ENTER to exit console.

```

```

*****
*****

```

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::::

Explanation:

1. CalculationTask Class:
 - This class implements the Callable interface, allowing it to return a result.
 - The call method performs a simulated complex calculation or I/O operation by sleeping for a random time between 1 and 3 seconds.
 - Each task prints its start and completion messages and returns a result string.
2. ThreadPoolExample Class:
 - The main method creates a fixed-size thread pool with 4 threads using `Executors.newFixedThreadPool(4)`.
 - It then submits 10 CalculationTask instances to the thread pool and stores the resulting Future objects in a list.
 - The `Future.get()` method is called to retrieve the results of the tasks. This method blocks until the task is completed.
 - After all tasks are completed, the thread pool is shut down using `executorService.shutdown()`.
 - The `awaitTermination` method is used to wait for the executor service to terminate, and if it doesn't terminate within 60 seconds, `shutdownNow` is called to forcefully terminate the executor service.
3. Execution Flow:
 - The thread pool executes up to 4 tasks concurrently.
 - Each task performs its calculation, simulates a delay, and then completes.
 - The results of the tasks are printed after they complete.
 - The program waits for all tasks to finish and then shuts down the thread pool.
 - This example demonstrates how to use a fixed-size thread pool to manage multiple concurrent tasks, ensuring efficient utilization of system resources while avoiding the overhead of creating and destroying threads for each task.

CODE::::

```
import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;
```



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

public class ThreadPoolExample {

    // Task that performs a complex calculation
    static class CalculationTask implements Callable<String> {
        private final int taskId;

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

        @Override
        public String call() throws Exception {
            System.out.println("Task " + taskId + " is starting...");
            // Simulate a complex calculation or I/O operation
            Thread.sleep((long) (Math.random() * 2000) + 1000);
            System.out.println("Task " + taskId + " is completed.");
            return "Result of Task " + taskId;
        }
    }

    public static void main(String[] args) {
        // Create a fixed-size thread pool with 4 threads
        ExecutorService executorService = Executors.newFixedThreadPool(4);
```

```

// Create a list to hold Future objects
List<Future<String>> futures = new ArrayList<>();

// Submit multiple tasks to the thread pool
for (int i = 1; i <= 10; i++) {
    CalculationTask task = new CalculationTask(i);
    Future<String> future = executorService.submit(task);
    futures.add(future);
}

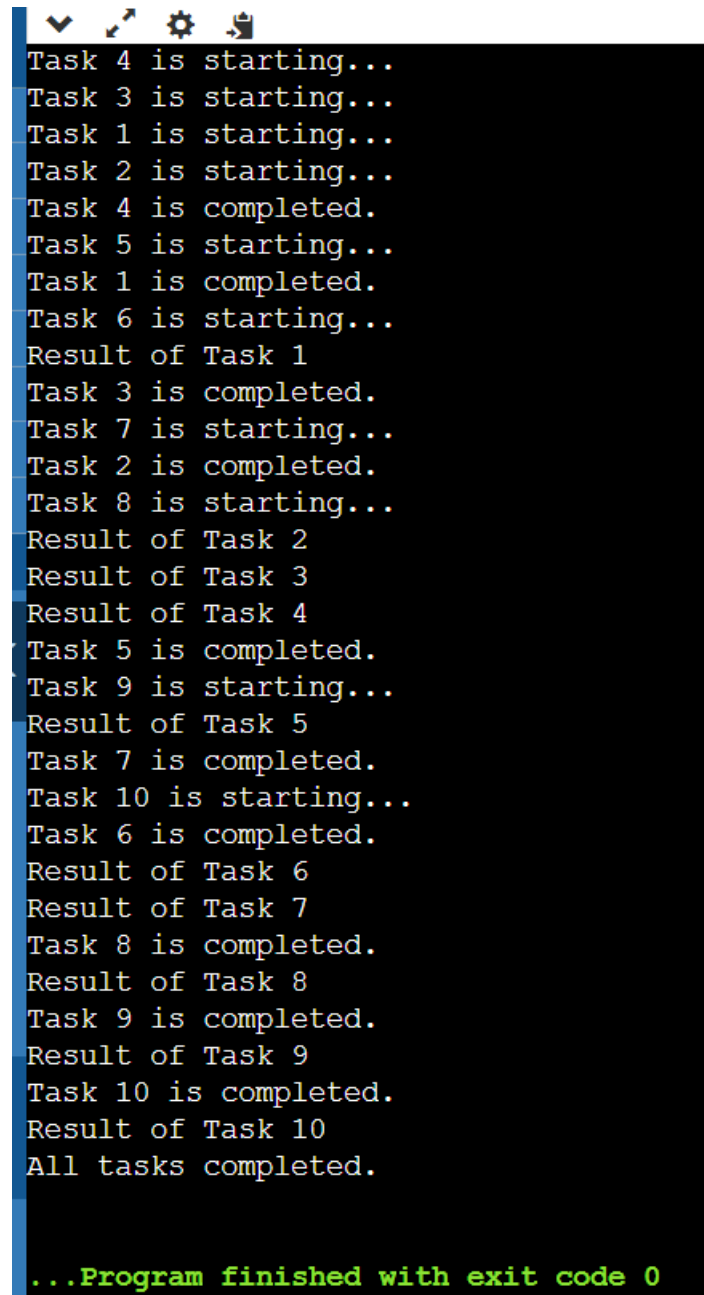
// Retrieve and print the results of the tasks
for (Future<String> future : futures) {
    try {
        String result = future.get(); // Blocking call to wait for task completion
        System.out.println(result);
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}

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

```

```
        System.out.println("All tasks completed.");
    }
}
```

OUTPUT:...



```
Task 4 is starting...
Task 3 is starting...
Task 1 is starting...
Task 2 is starting...
Task 4 is completed.
Task 5 is starting...
Task 1 is completed.
Task 6 is starting...
Result of Task 1
Task 3 is completed.
Task 7 is starting...
Task 2 is completed.
Task 8 is starting...
Result of Task 2
Result of Task 3
Result of Task 4
Task 5 is completed.
Task 9 is starting...
Result of Task 5
Task 7 is completed.
Task 10 is starting...
Task 6 is completed.
Result of Task 6
Result of Task 7
Task 8 is completed.
Result of Task 8
Task 9 is completed.
Result of Task 9
Task 10 is completed.
Result of Task 10
All tasks completed.

...Program finished with exit code 0
```

```
*****
*****
```

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:::

Explanation

1. Prime Number Calculation:
 - The isPrime method checks if a number is prime.
 - The PrimeTask class implements Callable<List<Integer>> to calculate prime numbers in a given range.
2. Parallel Execution Using ExecutorService:
 - An ExecutorService with a fixed thread pool is created.
 - The range of numbers is divided among the threads, and PrimeTask instances are submitted to the executor service.
 - The Future objects returned by the executor service are used to gather the prime numbers once the tasks are completed.
- Asynchronous File Writing Using CompletableFuture:
 - The writeToFileAsync method writes the list of prime numbers to a file asynchronously using CompletableFuture.runAsync.
3. Main Method:
 - The upper limit for prime number calculation and the number of threads are defined.
 - Prime number calculation tasks are submitted to the executor service.
 - The results from the tasks are collected into a single list.
 - The executor service is shut down.
 - The list of prime numbers is written to a file asynchronously.
 - The join method is used to wait for the file writing task to complete before printing the completion message.

CODE:::

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
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.Collectors;
import java.util.stream.IntStream;

public class PrimeNumberCalculator {

    // Method to check if a number is prime
    public static boolean isPrime(int number) {
        if (number <= 1) {
            return false;
        }
        for (int i = 2; i <= Math.sqrt(number); i++) {
            if (number % i == 0) {
                return false;
            }
        }
        return true;
    }

    // Task to calculate primes in a range
    static class PrimeTask implements Callable<List<Integer>> {
        private final int start;
        private final int end;

        public PrimeTask(int start, int end) {
            this.start = start;
            this.end = end;
        }

        @Override
        public List<Integer> call() {
```

```

        return IntStream.rangeClosed(start, end)
            .filter(PrimeNumberCalculator::isPrime)
            .boxed()
            .collect(Collectors.toList());
    }
}

// Method to write results to a file

public static CompletableFuture<Void> writeToFileAsync(List<Integer> primes, String
filename) {
    return CompletableFuture.runAsync(() -> {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            for (Integer prime : primes) {
                writer.write(prime.toString());
                writer.newLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
}

public static void main(String[] args) {
    final int upperLimit = 100;
    final int numThreads = 4;
    final int range = upperLimit / numThreads;
    ExecutorService executorService = Executors.newFixedThreadPool(numThreads);
    List<Future<List<Integer>>> futures = new ArrayList<>();

    // Submit tasks to the executor service
    for (int i = 0; i < numThreads; i++) {
        int start = i * range + 1;
        int end = (i == numThreads - 1) ? upperLimit : start + range - 1;
    }
}

```

```

        futures.add(executorService.submit(new PrimeTask(start, end)));
    }

    // Gather all prime numbers from futures
    List<Integer> allPrimes = new ArrayList<>();
    for (Future<List<Integer>> future : futures) {
        try {
            allPrimes.addAll(future.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Shutdown the executor service
    executorService.shutdown();

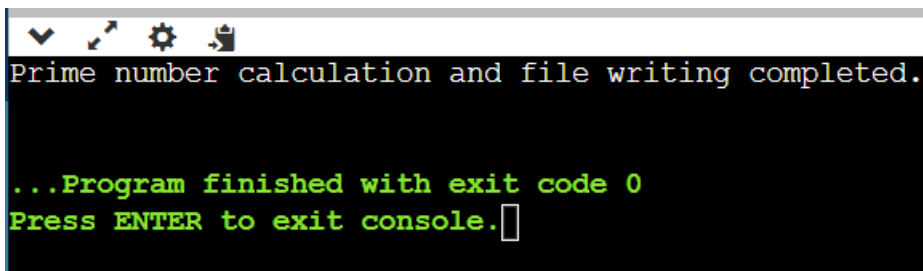
    // Asynchronously write the prime numbers to a file
    CompletableFuture<Void> writeFuture = writeToFileAsync(allPrimes, "primes.txt");

    // Wait for the writing task to complete
    writeFuture.join();

    System.out.println("Prime number calculation and file writing completed.");
}
}

```

OUTPUT:::



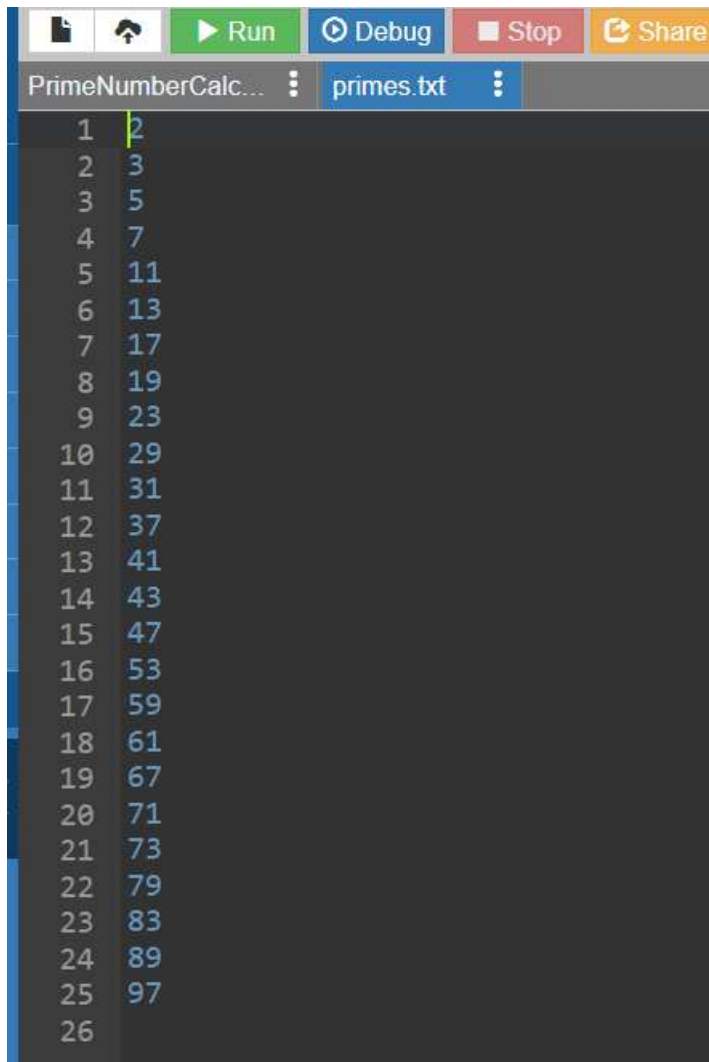
```

Prime number calculation and file writing completed.

...Program finished with exit code 0
Press ENTER to exit console.

```

Created prime.txt file



```
PrimeNumberCalc... primes.txt
1 2
2 3
3 5
4 7
5 11
6 13
7 17
8 19
9 23
10 29
11 31
12 37
13 41
14 43
15 47
16 53
17 59
18 61
19 67
20 71
21 73
22 79
23 83
24 89
25 97
26
```

```
*****
*****
```

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:::

1. Design Thread-Safe Counter Class: Implement a Counter class with synchronized methods for incrementing and decrementing.
2. Demonstrate Usage from Multiple Threads: Create multiple threads that concurrently access the Counter object.
3. Implement Immutable Class: Design an immutable class to share data between threads, ensuring thread safety.

Explanation:

1. Counter Class:
 - The Counter class has synchronized methods increment, decrement, and getCount to ensure thread safety.
 - These methods modify and access the count variable atomically.
2. Demonstrating Usage from Multiple Threads:
 - Two threads (incrementThread and decrementThread) are created to increment and decrement the counter, respectively.
 - Each thread performs its operation in a loop, modifying the counter multiple times.
3. Immutable Class:
 - The ImmutableData class is designed to be immutable, meaning its state cannot be changed after initialization.
 - It has only a single value field, which is initialized through the constructor and has no setter methods.
 - This ensures that the data shared between threads is thread-safe without the need for synchronization.
4. Main Method:
 - In the main method, the threads for incrementing and decrementing the counter are started and joined to wait for their completion.
 - The final count is printed to verify that the counter operations are thread-safe.
 - An immutable object (immutableData) is created and accessed from another thread (readThread) to demonstrate thread safety without synchronization.
 - This program demonstrates the design and usage of a thread-safe Counter class with synchronized methods and an immutable class to share data between threads without the risk of data corruption.

CODE:::

```
// Counter class with synchronized methods

class Counter {

    private int count = 0;


    // Synchronized method to increment the count

    public synchronized void increment() {

        count++;

    }


    // Synchronized method to decrement the count

    public synchronized void decrement() {
```

```

        count--;
    }

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

// Immutable class to share data between threads
final class ImmutableData {
    private final int value;

    // Constructor to initialize the immutable object
    public ImmutableData(int value) {
        this.value = value;
    }

    // Getter method to retrieve the value
    public int getValue() {
        return value;
    }
}

public class ThreadSafeDemo {
    public static void main(String[] args) {
        // Create a thread-safe Counter object
        Counter counter = new Counter();
    }
}

```

```
// Create and start multiple threads to increment and decrement the counter
```

```
Thread incrementThread = new Thread(() -> {
```

```
    for (int i = 0; i < 1000; i++) {
```

```
        counter.increment();
```

```
    }
```

```
});
```

```
Thread decrementThread = new Thread(() -> {
```

```
    for (int i = 0; i < 1000; i++) {
```

```
        counter.decrement();
```

```
    }
```

```
});
```

```
incrementThread.start();
```

```
decrementThread.start();
```

```
try {
```

```
    incrementThread.join();
```

```
    decrementThread.join();
```

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

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

```
}
```

```
// Print the final count
```

```
System.out.println("Final count: " + counter.getCount());
```

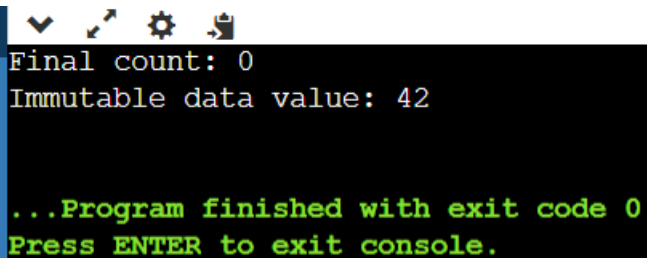
```
// Create an immutable object to share data between threads
```

```
ImmutableData immutableData = new ImmutableData(42);
```

```
// Create and start a thread to read the value from the immutable object
Thread readThread = new Thread(() -> {
    System.out.println("Immutable data value: " + immutableData.getValue());
});

readThread.start();
}
}
```

OUTPUT:...

A screenshot of a terminal window with a black background and white text. At the top, there are four small icons: a downward arrow, a magnifying glass, a gear, and a document. The output text is as follows:

```
Final count: 0
Immutable data value: 42

...Program finished with exit code 0
Press ENTER to exit console.
```