DAY 23:

ASSIGNMENT 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.

```
import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

import java.util.stream.Collectors;

public class PrimeNumberCalculator {

     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;
}
     public static List<Integer> calculatePrimes(int limit) {
    List<Integer> primes = new ArrayList<>();
    for (int i = 2; i <= limit; i++) {
        if (isPrime(i)) {
            primes.add(i);
        }
        }

        return primes;

    }

        public static CompletableFuture<Void>
writePrimesToFile(List<Integer> primes, String filename) {

        return CompletableFuture.runAsync(() -> {

            try {

                Files.write(Paths.get(filename),
primes.stream().map(String::valueOf).collect(Collectors.toList
()));

                System.out.println("Primes written to file: " +
filename);

            } catch (IOException e) {
```

```
            e.printStackTrace();

         }

      });

   }



      public static void main(String[] args) {

            int limit = 100000; // Change the limit as needed

      int numberOfThreads = 4;



      // Create a fixed-size thread pool

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

```
        // List to hold Future objects

        List<Future<List<Integer>>> futures = new ArrayList<>();

        int chunkSize = limit / numberOfThreads;

        for (int i = 0; i < numberOfThreads; i++) {

            int start = i * chunkSize + 1;

            int end = (i == numberOfThreads - 1) ? limit : (i + 1) *
chunkSize;

            Callable<List<Integer>> task = () ->
calculatePrimes(end);

            futures.add(executorService.submit(task));

        }
        List<Integer> allPrimes = new ArrayList<>();

        for (Future<List<Integer>> future : futures) {

            try {

                allPrimes.addAll(future.get());

            } catch (InterruptedException | ExecutionException e) {

                e.printStackTrace();

            }

        }

        executorService.shutdown();

        CompletableFuture<Void> fileWriteFuture =
writePrimesToFile(allPrimes, "primes.txt");
```

```
        fileWriteFuture.join();



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



            // TODO Auto-generated method stub



        }



}
```

Explanation

1. *Calculating Prime Numbers Using ExecutorService*:

   - We use an ExecutorService to parallelize the task of calculating prime numbers up to a given number (maxNumber).

   - For each number from 2 to maxNumber, we create a CompletableFuture that calculates the prime numbers up to that number asynchronously using the calculatePrimesUpTo method.

2. *Writing Results to File Asynchronously*:

   - We use another CompletableFuture (allFutures) to wait for all prime number calculations to complete.

   - Once all calculations are done, we asynchronously write the results to a file (outputFile) using a FileWriter.

3. *Shutting Down the Executor Service*:

   - Finally, we shut down the ExecutorService after waiting for all tasks to complete.

Observing Execution

When you run the program, it will asynchronously calculate prime numbers up to the specified maximum number and write the results to a file. You can observe the asynchronous behavior by checking the output file after the program has finished execution.