MODULE-3 MultiThreads ASSIGNMENT-4

```
//Skill Description:
//Gain hands-on expertise in Java Concurrency and Multithreading by immersing yourself
in the
//captivating narrative of the "Parallel Universe Explorer." Through a series of tasks,
you'll master
//essential concepts, tools, and techniques required for efficient parallel and
reactive programming
//Problem Statement 4:
//Problem Statement Description: Extend the program to implement reactive communication
//between timelines using Reactive Streams. Utilize Publisher and SubmissionPublisher
to
//communicate between timelines.

//Learning Outcomes:
//• Apply Reactive Streams for inter-timeline communication.
//• Gain proficiency in utilizing Publisher and SubmissionPublisher.
//This assignment provides a comprehensive journey into Java Concurrency and
Multithreading,
//allowing students to navigate the challenges of the "Parallel Universe Explorer" and
gain practical
//skills in initializing parallel tasks, exploring timelines sequentially and in
parallel, implementing
//reactive communication, and ensuring thread synchronization for shared resources.

//Codings:
package assignments;

import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.Flow;


public class ParallelSequentialUniverseExplorer {
 private static final ForkJoinPool pool = new ForkJoinPool();
 private static final List<String> sequentialTimeline =
List.of("s1","s2","s3","s4","s5","s6","s6","s7","s8","s9","s10");
 private static final List<String> parallelTimelines =
List.of("p1","p2","p3","p4","p5","p6","p6","p7","p8","p9","p10");
 private static final SubmissionPublisher<String> publisher = new
SubmissionPublisher<>();

 public static void main(String[] args) {
       System.out.println("----- Sequential Exploration -----");
     sequentialExplore(sequentialTimeline);

     System.out.println("\n----- Parallel Exploration -----");
     ParallelTask task = new ParallelTask(parallelTimelines);
     pool.invoke(task);

     System.out.println("\n----- Reactive Communication -----\n");
     reactiveCommunication();
 }

 // Method for sequential exploration
 static void sequentialExplore(List<String> events) {
     events.forEach(event -> System.out.println("Sequential Task: " + event));
 }

 // Method for reactive communication
 static void reactiveCommunication() {
     // Create a subscriber
     TimelineSubscriber subscriber = new TimelineSubscriber();
     publisher.subscribe(subscriber);

     // Publish events from both timelines
     System.out.println("Publishing events to reactive streams...");
     sequentialTimeline.forEach(publisher::submit);
     parallelTimelines.forEach(publisher::submit);

     // Close the publisher
     publisher.close();
 }
}

//RecursiveAction for parallel tasks
class ParallelTask extends RecursiveAction {
      private final List<String> tasks;
      private static final int THRESHOLD = 2;

      ParallelTask(List<String> tasks) {
            this.tasks = tasks;
      }

       @Override
       protected void compute() {
           if (tasks.size() <= THRESHOLD) {
               tasks.forEach(task -> System.out.println("Parallel Task: " + task));
           } else {
               int mid = tasks.size() / 2;
               ParallelTask leftTask = new ParallelTask(tasks.subList(0, mid));
               ParallelTask rightTask = new ParallelTask(tasks.subList(mid,
tasks.size()));
               invokeAll(leftTask, rightTask);
           }
       }
}

//Subscriber for handling reactive communication
class TimelineSubscriber implements Flow.Subscriber<String> {
      private Flow.Subscription subscription;

 @Override
       public void onSubscribe(Flow.Subscription subscription) {
           this.subscription = subscription;
           subscription.request(1); // Request the first item
       }

       @Override
       public void onNext(String item) {
           System.out.println("Received via Reactive Streams: " + item);
           subscription.request(1); // Request the next item
       }

       @Override
       public void onError(Throwable throwable) {
           System.err.println("Error occurred: " + throwable.getMessage());
       }

       @Override
       public void onComplete() {
     System.out.println("All events received. Reactive communication complete.");
 }
}
```