```java
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 Assignment5 {
    private static final ForkJoinPool pool = new ForkJoinPool();
    private static final List<String> sequentialTimeline = List.of("s1", "s2", "s3",
"s4", "s5");
    private static final List<String> parallelTimelines = List.of("p1", "p2", "p3",
"p4", "p5");
    private static final SubmissionPublisher<String> publisher = new
SubmissionPublisher<>();
    private static final Object sharedResource = new Object();
    private static boolean resourceAvailable = false;

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

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

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

        System.out.println("\n----- Thread Synchronization -----");
        threadSynchronizationExample();
    }

    // Sequential exploration method
    static void sequentialExplore() {
        sequentialTimeline.forEach(event -> System.out.println("Sequential Task: " +
event));
    }

    // Reactive communication method
    static void reactiveCommunication() {
        TimelineSubscriber subscriber = new TimelineSubscriber();
        publisher.subscribe(subscriber);

        System.out.println("Publishing events...");
        sequentialTimeline.forEach(publisher::submit);
        parallelTimelines.forEach(publisher::submit);

        publisher.close();
    }

    // Thread synchronization method
    static void threadSynchronizationExample() {
        Thread producer = new Thread(() -> {
```

```
            synchronized (sharedResource) {
                System.out.println("Producer: Producing a shared resource...");
                resourceAvailable = true;
                sharedResource.notify(); // Notify a waiting thread
            }
        });

        Thread consumer = new Thread(() -> {
            synchronized (sharedResource) {
                while (!resourceAvailable) {
                    try {
                        System.out.println("Consumer: Waiting for the resource...");
                        sharedResource.wait(); // Wait until notified
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        System.err.println("Consumer interrupted while waiting!");
                    }
                }
                System.out.println("Consumer: Resource consumed!");
                resourceAvailable = false;
            }
        });

        producer.start();
        consumer.start();

        // Ensure the threads complete
        try {
            producer.join();
            consumer.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Main thread interrupted!");
        }
    }
}

// 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.");
        }
}
```

s