```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

public class ParallelUniverseExplorer {

    private static final ForkJoinPool pool = new ForkJoinPool();
    private static final int THRESHOLD = 3; // Minimum tasks to process sequentially
    private static final Random random = new Random();

    // Universe data to initialize
    private static final List<String> universeTimelines = Arrays.asList(
        "Quantum Realm", "Multiverse Core", "Alternate Reality Alpha",
        "Parallel Dimension Beta", "Mirror Universe Gamma", "Pocket Dimension Delta",
        "Time Fracture Epsilon", "Cosmic String Zeta", "Quantum Foam Eta",
        "Dark Matter Universe", "Anti-Matter Realm", "Virtual Reality Matrix",
        "Crystal Dimension", "Energy Plane", "Void Realm"
    );

    public static void main(String[] args) {
        System.out.println("=== Parallel Universe Explorer Initialization ===\n");

        // Initialize the explorer with RecursiveAction tasks
        initializeParallelUniverseExplorer();

        // Demonstrate different types of parallel tasks
        demonstrateVariousTasks();

        // Wait for all tasks to complete
        try {
            Thread.sleep(3000);
            System.out.println("\n=== All Parallel Universe Exploration Tasks Completed ===");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    /**
     * Main initialization method using RecursiveAction tasks
     */
    public static void initializeParallelUniverseExplorer() {
        System.out.println("Initializing Parallel Universe Explorer with " +
                           universeTimelines.size() + " timelines...");
        System.out.println("Using ForkJoinPool with parallelism: " + pool.getParallelism() +
"\n");

        // Create main exploration task
        UniverseExplorationTask mainTask = new UniverseExplorationTask(universeTimelines, 0,
universeTimelines.size());

        // Execute the task in the ForkJoinPool
        pool.invoke(mainTask);

        System.out.println("\nMain exploration task submitted to ForkJoinPool");
    }

    /**
     * Demonstrate different types of parallel tasks
```

```
     */
    public static void demonstrateVariousTasks() {
        System.out.println("\n--- Additional Parallel Task Demonstrations ---");

        // Task 1: Timeline Analysis
        TimelineAnalysisTask analysisTask = new TimelineAnalysisTask(universeTimelines);
        pool.execute(analysisTask);

        // Task 2: Resource Discovery
        ResourceDiscoveryTask discoveryTask = new ResourceDiscoveryTask(8); // 8 resources to
discover
        pool.execute(discoveryTask);

        // Task 3: Universe Mapping
        UniverseMappingTask mappingTask = new UniverseMappingTask(1000); // Map 1000 coordinates
        pool.execute(mappingTask);
    }

    /**
     * RecursiveAction for exploring individual universes
     */
    static class UniverseExplorationTask extends RecursiveAction {
        private final List<String> timelines;
        private final int start;
        private final int end;

        public UniverseExplorationTask(List<String> timelines, int start, int end) {
            this.timelines = timelines;
            this.start = start;
            this.end = end;
        }

        @Override
        protected void compute() {
            int length = end - start;

            if (length <= THRESHOLD) {
                // Process sequentially
                exploreTimelinesSequentially();
            } else {
                // Split task and process in parallel
                int mid = start + (length / 2);

                UniverseExplorationTask leftTask = new UniverseExplorationTask(timelines, start,
mid);
                UniverseExplorationTask rightTask = new UniverseExplorationTask(timelines, mid,
end);

                // Fork both tasks for parallel execution
                leftTask.fork();
                rightTask.fork();

                // Wait for completion
                leftTask.join();
                rightTask.join();
            }
        }

        private void exploreTimelinesSequentially() {
            for (int i = start; i < end; i++) {
                String timeline = timelines.get(i);
                System.out.println(Thread.currentThread().getName() +
```

```
                                " - Exploring: " + timeline);

                // Simulate exploration work
                simulateExplorationWork(200 + random.nextInt(300));

                System.out.println(Thread.currentThread().getName() +
                                " - Completed: " + timeline +
                                " [Stability: " + (random.nextInt(100) + "%]");
            }
        }
    }

    /**
     * Different type of RecursiveAction for timeline analysis
     */
    static class TimelineAnalysisTask extends RecursiveAction {
        private final List<String> timelines;
        private final int start;
        private final int end;
        private static final int ANALYSIS_THRESHOLD = 2;

        public TimelineAnalysisTask(List<String> timelines) {
            this(timelines, 0, timelines.size());
        }

        public TimelineAnalysisTask(List<String> timelines, int start, int end) {
            this.timelines = timelines;
            this.start = start;
            this.end = end;
        }

        @Override
        protected void compute() {
            int length = end - start;

            if (length <= ANALYSIS_THRESHOLD) {
                analyzeTimelinesSequentially();
            } else {
                int mid = start + (length / 2);

                TimelineAnalysisTask leftTask = new TimelineAnalysisTask(timelines, start, mid);
                TimelineAnalysisTask rightTask = new TimelineAnalysisTask(timelines, mid, end);

                invokeAll(leftTask, rightTask);
            }
        }

        private void analyzeTimelinesSequentially() {
            for (int i = start; i < end; i++) {
                String timeline = timelines.get(i);
                System.out.println(Thread.currentThread().getName() +
                                " - ðŸ”¬ Analyzing: " + timeline);

                simulateExplorationWork(300 + random.nextInt(400));

                // Generate analysis results
                String[] anomalies = {"Quantum Fluctuation", "Temporal Loop", "Reality Breach",
"Stable"};
                String anomaly = anomalies[random.nextInt(anomalies.length)];
                int energyLevel = random.nextInt(1000);

                System.out.println(Thread.currentThread().getName() +
```

```
                                " - Analysis Complete: " + timeline +
                                " | Anomaly: " + anomaly +
                                " | Energy: " + energyLevel + " units");
            }
        }
    }

    /**
     * RecursiveAction for resource discovery with different splitting strategy
     */
    static class ResourceDiscoveryTask extends RecursiveAction {
        private final int resourcesToDiscover;
        private static final int DISCOVERY_THRESHOLD = 2;

        public ResourceDiscoveryTask(int resourcesToDiscover) {
            this.resourcesToDiscover = resourcesToDiscover;
        }

        @Override
        protected void compute() {
            if (resourcesToDiscover <= DISCOVERY_THRESHOLD) {
                discoverResourcesSequentially();
            } else {
                int half = resourcesToDiscover / 2;
                int remainder = resourcesToDiscover - half;

                ResourceDiscoveryTask leftTask = new ResourceDiscoveryTask(half);
                ResourceDiscoveryTask rightTask = new ResourceDiscoveryTask(remainder);

                leftTask.fork();
                rightTask.compute(); // Compute current thread
                leftTask.join();
            }
        }

        private void discoverResourcesSequentially() {
            for (int i = 0; i < resourcesToDiscover; i++) {
                String[] resourceTypes = {"Quantum Crystal", "Temporal Essence", "Dark Matter",
                                          "Cosmic String", "Reality Shard"};
                String resource = resourceTypes[random.nextInt(resourceTypes.length)];

                System.out.println(Thread.currentThread().getName() +
                                " - ðŸ’Ž Discovering: " + resource + " #" + (i + 1));

                simulateExplorationWork(150 + random.nextInt(200));

                int quantity = random.nextInt(100) + 1;
                System.out.println(Thread.currentThread().getName() +
                                " - Resource Found: " + resource +
                                " x" + quantity + " units");
            }
        }
    }

    /**
     * RecursiveAction for universe mapping with iterative work stealing
     */
    static class UniverseMappingTask extends RecursiveAction {
        private final int coordinatesToMap;
        private static final int MAPPING_THRESHOLD = 100;

        public UniverseMappingTask(int coordinatesToMap) {
```

```
            this.coordinatesToMap = coordinatesToMap;
        }

        @Override
        protected void compute() {
            if (coordinatesToMap <= MAPPING_THRESHOLD) {
                mapCoordinatesSequentially();
            } else {
                // Split into smaller mapping tasks
                List<UniverseMappingTask> subtasks = new ArrayList<>();
                int chunkSize = coordinatesToMap / 4;
                int remaining = coordinatesToMap;

                while (remaining > 0) {
                    int currentChunk = Math.min(chunkSize, remaining);
                    subtasks.add(new UniverseMappingTask(currentChunk));
                    remaining -= currentChunk;
                }

                // Invoke all subtasks
                invokeAll(subtasks);
            }
        }

        private void mapCoordinatesSequentially() {
            System.out.println(Thread.currentThread().getName() +
                               " - ðŸ—ºï¸  Mapping " + coordinatesToMap + " coordinates...");

            for (int i = 0; i < coordinatesToMap; i++) {
                if (i % 20 == 0) { // Print progress every 20 coordinates
                    System.out.println(Thread.currentThread().getName() +
                                       " - Mapping progress: " + i + "/" + coordinatesToMap);
                }

                // Simulate mapping work (shorter delay for mapping)
                simulateExplorationWork(10);
            }

            System.out.println(Thread.currentThread().getName() +
                               " - âœ… Completed mapping " + coordinatesToMap + " coordinates");
        }
    }

    /**
     * Simulates exploration work by sleeping for specified milliseconds
     */
    private static void simulateExplorationWork(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Exploration interrupted!");
        }
    }

    /**
     * Utility method to shutdown the ForkJoinPool (not called in this example to keep it
simple)
     */
    public static void shutdown() {
        pool.shutdown();
    }
```

```
}
```