```
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;
import java.util.stream.Collectors;

public class ParallelUniverseExplorer {

    private static final ForkJoinPool pool = new ForkJoinPool();

    private static final List<String> sequentialTimeline = List.of(
        "Big Bang", "Formation of First Stars", "First Galaxies Form",
        "Solar System Formation", "Early Earth", "First Life Forms",
        "Cambrian Explosion", "Age of Dinosaurs", "Mass Extinction Event",
        "Age of Mammals", "Early Hominids", "Stone Age", "Bronze Age",
        "Iron Age", "Classical Antiquity", "Middle Ages", "Renaissance",
        "Industrial Revolution", "Digital Age", "Present Day"
    );

    private static final List<List<String>> parallelTimelines = List.of(
        List.of("Timeline Alpha: Big Bang", "Timeline Alpha: Alternate Evolution",
                "Timeline Alpha: Advanced Civilization", "Timeline Alpha: Cosmic Ascension"),
        List.of("Timeline Beta: Quantum Fluctuation", "Timeline Beta: Crystal Worlds",
                "Timeline Beta: Mechanical Life", "Timeline Beta: Singularity"),
        List.of("Timeline Gamma: Dark Matter Dominance", "Timeline Gamma: Energy Beings",
                "Timeline Gamma: Time Manipulation", "Timeline Gamma: Multiverse Travel"),
        List.of("Timeline Delta: Organic Technology", "Timeline Delta: Symbiotic Worlds",
                "Timeline Delta: Universal Harmony", "Timeline Delta: Transcendence"),
        List.of("Timeline Epsilon: Artificial Genesis", "Timeline Epsilon: Digital Reality",
                "Timeline Epsilon: Virtual Dimensions", "Timeline Epsilon: Code Universe")
    );

    // Reactive Streams Components
    private static final SubmissionPublisher<String> timelinePublisher = new
SubmissionPublisher<>();
    private static final SubmissionPublisher<String> discoveryPublisher = new
SubmissionPublisher<>();
    private static final SubmissionPublisher<String> emergencyPublisher = new
SubmissionPublisher<>();

    public static void main(String[] args) {
        System.out.println("=== Parallel Universe Explorer with Reactive Communication ===");

        // Register subscribers for different types of events
        registerSubscribers();

        // Simulate sequential exploration
        exploreSequentialTimeline();

        // Simulate parallel exploration
        exploreParallelTimelines();

        // Demonstrate reactive communication
        demonstrateReactiveCommunication();

        // Simulate timeline events and communication
        simulateTimelineEvents();

        // Close publishers after some time
        closePublishers();
```

```
    }

    /**
     * Register various subscribers for reactive communication
     */
    private static void registerSubscribers() {
        // Subscriber for timeline synchronization events
        timelinePublisher.subscribe(new TimelineSynchronizationSubscriber());

        // Subscriber for scientific discoveries
        discoveryPublisher.subscribe(new DiscoveryAnalysisSubscriber());

        // Subscriber for emergency alerts
        emergencyPublisher.subscribe(new EmergencyAlertSubscriber());

        // Chain publishers for complex event processing
        discoveryPublisher.subscribe(new Flow.Processor<String, String>() {
            private Flow.Subscription subscription;
            private final SubmissionPublisher<String> analyticsPublisher = new
SubmissionPublisher<>();

            {
                analyticsPublisher.subscribe(new AnalyticsSubscriber());
            }

            @Override
            public void onSubscribe(Flow.Subscription subscription) {
                this.subscription = subscription;
                subscription.request(1);
            }

            @Override
            public void onNext(String item) {
                // Transform discovery into analytics data
                String analytics = "ANALYTICS: " + item + " [Priority: " +
                    (item.contains("Breakthrough") ? "HIGH" : "MEDIUM") + "]";
                analyticsPublisher.submit(analytics);
                subscription.request(1);
            }

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

            @Override
            public void onComplete() {
                analyticsPublisher.close();
                System.out.println("Analytics processing completed");
            }
        });
    }

    /**
     * Sequential exploration of the main timeline
     */
    public static void exploreSequentialTimeline() {
        System.out.println("\n--- Sequential Timeline Exploration ---");

        sequentialTimeline.stream()
            .forEach(event -> {
                System.out.println("Exploring: " + event);
```

```
                simulateExplorationDelay(100);

                // Publish significant events reactively
                if (event.contains("Bang") || event.contains("Extinction") ||
                    event.contains("Revolution")) {
                    timelinePublisher.submit("TIMELINE_EVENT: " + event);
                }
            });
    }

    /**
     * Parallel exploration of multiple timelines
     */
    public static void exploreParallelTimelines() {
        System.out.println("\n--- Parallel Timelines Exploration ---");

        parallelTimelines.parallelStream()
            .forEach(timeline -> {
                String timelineName = timeline.get(0).split(":")[0];

                timeline.forEach(event -> {
                    System.out.println(Thread.currentThread().getName() + " - Processing: " +
event);
                    simulateExplorationDelay(150);

                    // Publish discoveries reactively
                    if (event.contains("Advanced") || event.contains("Singularity") ||
                        event.contains("Transcendence")) {
                        discoveryPublisher.submit("DISCOVERY: " + event + " from " +
timelineName);
                    }
                });
            });
    }

    /**
     * Demonstrate reactive communication between timelines
     */
    public static void demonstrateReactiveCommunication() {
        System.out.println("\n--- Reactive Communication Demo ---");

        // Simulate cross-timeline communication
        new Thread(() -> {
            String[] messages = {
                "TIMELINE_SYNC: Alpha and Beta timelines converging",
                "EMERGENCY: Timeline Gamma experiencing quantum instability",
                "DISCOVERY: Cross-timeline communication protocol established",
                "TIMELINE_SYNC: All timelines synchronized at temporal coordinate 7G.3F",
                "EMERGENCY: Reality breach in Timeline Epsilon!",
                "DISCOVERY: Breakthrough in multiverse energy transfer"
            };

            for (String message : messages) {
                try {
                    Thread.sleep(500);
                    if (message.startsWith("TIMELINE_SYNC")) {
                        timelinePublisher.submit(message);
                    } else if (message.startsWith("EMERGENCY")) {
                        emergencyPublisher.submit(message);
                    } else if (message.startsWith("DISCOVERY")) {
                        discoveryPublisher.submit(message);
                    }
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }).start();
}

/**
 * Simulate ongoing timeline events with reactive publishing
 */
public static void simulateTimelineEvents() {
    System.out.println("\n--- Ongoing Timeline Event Simulation ---");

    // Simulate random timeline events
    for (int i = 0; i < 10; i++) {
        try {
            Thread.sleep(800);

            // Random event generation
            String eventType = switch (i % 3) {
                case 0 -> "TIMELINE_SYNC: Periodic synchronization pulse #" + i;
                case 1 -> "DISCOVERY: New temporal anomaly detected #" + i;
                case 2 -> "EMERGENCY: Quantum fluctuation level " + i + " detected";
                default -> "UNKNOWN_EVENT";
            };

            // Publish to appropriate publisher
            if (eventType.startsWith("TIMELINE_SYNC")) {
                timelinePublisher.submit(eventType);
            } else if (eventType.startsWith("DISCOVERY")) {
                discoveryPublisher.submit(eventType);
            } else if (eventType.startsWith("EMERGENCY")) {
                emergencyPublisher.submit(eventType);
            }

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

/**
 * Close all publishers gracefully
 */
private static void closePublishers() {
    try {
        Thread.sleep(5000);
        timelinePublisher.close();
        discoveryPublisher.close();
        emergencyPublisher.close();
        System.out.println("\nAll reactive publishers closed");
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

/**
 * Simulates exploration delay
 */
private static void simulateExplorationDelay(int milliseconds) {
    try {
        Thread.sleep(milliseconds);
```

```
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    // ==================== REACTIVE SUBSCRIBERS ====================

    /**
     * Subscriber for timeline synchronization events
     */
    static class TimelineSynchronizationSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

        @Override
        public void onSubscribe(Flow.Subscription subscription) {
            this.subscription = subscription;
            System.out.println("TimelineSynchronizationSubscriber: Subscribed to timeline
events");
            subscription.request(1);
        }

        @Override
        public void onNext(String item) {
            System.out.println("ðŸ”— SYNC_SUBSCRIBER: " + item + " [Processing
synchronization...]");
            simulateProcessingDelay(200);
            subscription.request(1);
        }

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

        @Override
        public void onComplete() {
            System.out.println("TimelineSynchronizationSubscriber: All timeline events
processed");
        }
    }

    /**
     * Subscriber for discovery analysis
     */
    static class DiscoveryAnalysisSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

        @Override
        public void onSubscribe(Flow.Subscription subscription) {
            this.subscription = subscription;
            System.out.println("DiscoveryAnalysisSubscriber: Subscribed to discovery events");
            subscription.request(1);
        }

        @Override
        public void onNext(String item) {
            System.out.println("ðŸ”¬ DISCOVERY_SUBSCRIBER: " + item + " [Analyzing scientific
data...]");
            simulateProcessingDelay(300);
            subscription.request(1);
        }
```

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

        @Override
        public void onComplete() {
            System.out.println("DiscoveryAnalysisSubscriber: All discoveries analyzed");
        }
    }

    /**
     * Subscriber for emergency alerts
     */
    static class EmergencyAlertSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

        @Override
        public void onSubscribe(Flow.Subscription subscription) {
            this.subscription = subscription;
            System.out.println("EmergencyAlertSubscriber: Subscribed to emergency events - MAX
PRIORITY");
            subscription.request(1);
        }

        @Override
        public void onNext(String item) {
            System.out.println("ðŸš¨ EMERGENCY_SUBSCRIBER: " + item + " [IMMEDIATE ACTION
REQUIRED!]");
            simulateProcessingDelay(100); // Faster processing for emergencies
            subscription.request(1);
        }

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

        @Override
        public void onComplete() {
            System.out.println("EmergencyAlertSubscriber: Emergency monitoring ended");
        }
    }

    /**
     * Subscriber for analytics data
     */
    static class AnalyticsSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

        @Override
        public void onSubscribe(Flow.Subscription subscription) {
            this.subscription = subscription;
            System.out.println("AnalyticsSubscriber: Subscribed to analytics events");
            subscription.request(1);
        }

        @Override
        public void onNext(String item) {
            System.out.println("ðŸ“Š ANALYTICS_SUBSCRIBER: " + item + " [Storing in
database...]");
```

```
            simulateProcessingDelay(150);
            subscription.request(1);
        }

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

        @Override
        public void onComplete() {
            System.out.println("AnalyticsSubscriber: Analytics processing complete");
        }
    }

    /**
     * Simulates processing delay for subscribers
     */
    private static void simulateProcessingDelay(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```