```
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;
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<>();

    // Shared Resources for Synchronization
    private static final Object sharedResource = new Object();
    private static final Object dataBufferLock = new Object();
    private static final Object timelineCoordinatorLock = new Object();

    // Shared data structures
    private static final ConcurrentHashMap<String, String> sharedDiscoveries = new
ConcurrentHashMap<>();
    private static final List<String> sharedEventBuffer = new java.util.ArrayList<>();
    private static final List<String> criticalTimelineData = new java.util.ArrayList<>();

    // Synchronization control variables
    private static boolean dataReady = false;
    private static boolean explorationComplete = false;
    private static final AtomicInteger activeExplorers = new AtomicInteger(0);
    private static int bufferSize = 0;
    private static final int MAX_BUFFER_SIZE = 5;

    public static void main(String[] args) {
```

```
    System.out.println("=== Parallel Universe Explorer with Thread Synchronization ===");

    // Register subscribers for reactive communication
    registerSubscribers();

    // Start synchronization demonstrations
    demonstrateWaitNotify();
    demonstrateProducerConsumer();
    demonstrateSpinWait();
    demonstrateTimelineCoordination();

    // Run exploration with synchronization
    exploreWithSynchronization();

    // Close publishers
    closePublishers();
}

/**
 * Demonstration of basic wait() and notify() pattern
 */
public static void demonstrateWaitNotify() {
    System.out.println("\n--- Wait/Notify Demonstration ---");

    // Data Producer Thread
    Thread producer = new Thread(() -> {
        synchronized (sharedResource) {
            try {
                System.out.println("Producer: Preparing timeline data...");
                Thread.sleep(1000);
                dataReady = true;
                sharedResource.notify(); // Notify waiting threads
                System.out.println("Producer: Data ready! Notified waiting threads.");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    });

    // Data Consumer Thread
    Thread consumer = new Thread(() -> {
        synchronized (sharedResource) {
            try {
                System.out.println("Consumer: Waiting for timeline data...");
                while (!dataReady) {
                    sharedResource.wait(); // Wait for notification
                }
                System.out.println("Consumer: Data received! Starting analysis...");
                // Simulate data processing
                Thread.sleep(500);
                System.out.println("Consumer: Analysis complete.");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    });

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

    try {
        consumer.join();
```

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

    /**
     * Producer-Consumer pattern with bounded buffer using wait/notify
     */
    public static void demonstrateProducerConsumer() {
        System.out.println("\n--- Producer-Consumer Pattern (Bounded Buffer) ---");

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 10; i++) {
                String event = "Timeline Event " + i;
                synchronized (dataBufferLock) {
                    while (bufferSize >= MAX_BUFFER_SIZE) {
                        try {
                            System.out.println("Producer: Buffer full, waiting...");
                            dataBufferLock.wait();
                        } catch (InterruptedException e) {
                            Thread.currentThread().interrupt();
                            return;
                        }
                    }

                    sharedEventBuffer.add(event);
                    bufferSize++;
                    System.out.println("Producer: Added " + event + " (Buffer size: " +
bufferSize + ")");
                    dataBufferLock.notifyAll(); // Notify consumers
                }

                try {
                    Thread.sleep(200); // Simulate event generation time
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }

            synchronized (dataBufferLock) {
                explorationComplete = true;
                dataBufferLock.notifyAll();
                System.out.println("Producer: All events produced. Notifying consumers.");
            }
        });

        Thread consumer = new Thread(() -> {
            while (true) {
                String event;
                synchronized (dataBufferLock) {
                    while (bufferSize == 0 && !explorationComplete) {
                        try {
                            System.out.println("Consumer: Buffer empty, waiting...");
                            dataBufferLock.wait();
                        } catch (InterruptedException e) {
                            Thread.currentThread().interrupt();
                            return;
                        }
                    }

                    if (bufferSize == 0 && explorationComplete) {
```

```
                        System.out.println("Consumer: No more events. Exiting.");
                        return;
                    }

                    event = sharedEventBuffer.remove(0);
                    bufferSize--;
                    System.out.println("Consumer: Processed " + event + " (Buffer size: " +
bufferSize + ")");
                    dataBufferLock.notifyAll(); // Notify producers
                }

                // Simulate event processing time
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });

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

        try {
            producer.join();
            consumer.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    /**
     * Demonstration of Thread.onSpinWait() for brief waiting periods
     */
    public static void demonstrateSpinWait() {
        System.out.println("\n--- SpinWait Demonstration ---");

        AtomicInteger spinCounter = new AtomicInteger(0);
        final int MAX_SPINS = 1000;

        Thread coordinator = new Thread(() -> {
            try {
                System.out.println("Coordinator: Preparing timeline coordination...");
                Thread.sleep(500);

                // Set up a condition that will be true after some spins
                new Thread(() -> {
                    try {
                        Thread.sleep(100);
                        spinCounter.set(MAX_SPINS);
                        System.out.println("SpinWait Helper: Condition satisfied!");
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }).start();

                // Use spin wait for very brief waiting period
                System.out.println("Coordinator: Waiting for condition with spin wait...");
                while (spinCounter.get() < MAX_SPINS) {
                    Thread.onSpinWait(); // Hint to CPU that we're spinning
                }
```

```
                System.out.println("Coordinator: Condition met! Proceeding with coordination.");

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

        coordinator.start();

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

    /**
     * Complex timeline coordination using multiple synchronization techniques
     */
    public static void demonstrateTimelineCoordination() {
        System.out.println("\n--- Timeline Coordination ---");

        final int NUM_EXPLORERS = 3;
        activeExplorers.set(NUM_EXPLORERS);

        // Start multiple explorer threads
        for (int i = 0; i < NUM_EXPLORERS; i++) {
            final int explorerId = i;
            new Thread(() -> {
                exploreTimeline(explorerId);
            }).start();
        }

        // Coordinator thread that waits for all explorers to complete
        Thread coordinator = new Thread(() -> {
            synchronized (timelineCoordinatorLock) {
                while (activeExplorers.get() > 0) {
                    try {
                        System.out.println("Coordinator: Waiting for " + activeExplorers.get() +
" explorers...");
                        timelineCoordinatorLock.wait(1000); // Wait with timeout
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
                System.out.println("Coordinator: All explorers completed! Consolidating
discoveries...");
                consolidateDiscoveries();
            }
        });

        coordinator.start();

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

    /**
     * Individual timeline exploration with synchronization
```

```
     */
    private static void exploreTimeline(int explorerId) {
        System.out.println("Explorer " + explorerId + ": Starting timeline exploration...");

        try {
            // Simulate exploration work
            for (int i = 0; i < 3; i++) {
                Thread.sleep(500 + explorerId * 100);
                String discovery = "Discovery from Explorer " + explorerId + "-" + i;

                synchronized (sharedResource) {
                    sharedDiscoveries.put(discovery, "Explorer" + explorerId);
                    System.out.println("Explorer " + explorerId + ": Added discovery - " +
discovery);

                    // Notify coordinator of important discovery
                    if (i == 1) {
                        sharedResource.notify();
                        System.out.println("Explorer " + explorerId + ": Notified coordinator of
important discovery");
                    }
                }
            }

            // Signal completion
            synchronized (timelineCoordinatorLock) {
                activeExplorers.decrementAndGet();
                System.out.println("Explorer " + explorerId + ": Exploration complete. " +
                                   activeExplorers.get() + " explorers remaining.");
                timelineCoordinatorLock.notifyAll();
            }

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

    /**
     * Consolidate discoveries from all explorers
     */
    private static void consolidateDiscoveries() {
        System.out.println("\n--- Discovery Consolidation ---");
        System.out.println("Total discoveries made: " + sharedDiscoveries.size());
        sharedDiscoveries.forEach((discovery, explorer) -> {
            System.out.println("Consolidated: " + discovery + " by " + explorer);
        });
    }

    /**
     * Main exploration with all synchronization techniques
     */
    public static void exploreWithSynchronization() {
        System.out.println("\n--- Comprehensive Exploration with Synchronization ---");

        // Reset synchronization state
        dataReady = false;
        explorationComplete = false;
        activeExplorers.set(0);
        sharedEventBuffer.clear();
        sharedDiscoveries.clear();
        criticalTimelineData.clear();
```

```
        // Start coordinated exploration
        demonstrateTimelineCoordination();
    }

    /**
     * Register reactive subscribers
     */
    private static void registerSubscribers() {
        timelinePublisher.subscribe(new TimelineSynchronizationSubscriber());
        discoveryPublisher.subscribe(new DiscoveryAnalysisSubscriber());
        emergencyPublisher.subscribe(new EmergencyAlertSubscriber());
    }

    /**
     * Close all publishers
     */
    private static void closePublishers() {
        try {
            Thread.sleep(2000);
            timelinePublisher.close();
            discoveryPublisher.close();
            emergencyPublisher.close();
            System.out.println("\nAll reactive publishers closed");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

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

    static class TimelineSynchronizationSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

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

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

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

        @Override
        public void onComplete() {
            System.out.println("TimelineSynchronizationSubscriber: Complete");
        }
    }

    static class DiscoveryAnalysisSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

        @Override
        public void onSubscribe(Flow.Subscription subscription) {
```

```
            this.subscription = subscription;
            subscription.request(1);
        }

        @Override
        public void onNext(String item) {
            System.out.println("ðŸ”¬ DISCOVERY_SUBSCRIBER: " + item);
            subscription.request(1);
        }

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

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

    static class EmergencyAlertSubscriber implements Flow.Subscriber<String> {
        private Flow.Subscription subscription;

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

        @Override
        public void onNext(String item) {
            System.out.println("ðŸš¨ EMERGENCY_SUBSCRIBER: " + item);
            subscription.request(1);
        }

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

        @Override
        public void onComplete() {
            System.out.println("EmergencyAlertSubscriber: Complete");
        }
    }
}
```