Dining Philosophers Problem

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class DiningPhilosophers {

    private static final int NUM_PHILOSOPHERS = 5;

    private Philosopher[] philosophers;
    private Lock[] forks;
    private Condition[] conditions;

    public DiningPhilosophers() {
        philosophers = new Philosopher[NUM_PHILOSOPHERS];
        forks = new Lock[NUM_PHILOSOPHERS];
        conditions = new Condition[NUM_PHILOSOPHERS];

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            forks[i] = new ReentrantLock();
            conditions[i] = forks[i].newCondition();
        }

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            philosophers[i] = new Philosopher(i, forks[i], forks[(i + 1) %
NUM_PHILOSOPHERS]);
            new Thread(philosophers[i]).start();
        }
    }

    class Philosopher implements Runnable {

        private int id;
        private Lock leftFork;
        private Lock rightFork;

        public Philosopher(int id, Lock leftFork, Lock rightFork) {
            this.id = id;
            this.leftFork = leftFork;
            this.rightFork = rightFork;
        }

        private void think() throws InterruptedException {
            System.out.println("Philosopher " + id + " is thinking");
            Thread.sleep((long) (Math.random() * 1000));
        }
```

```
        private void eat() throws InterruptedException {
            System.out.println("Philosopher " + id + " is eating");
            Thread.sleep((long) (Math.random() * 1000));
        }

        private void pickUpForks() throws InterruptedException {
            leftFork.lock();
            System.out.println("Philosopher " + id + " picked up left fork");
            rightFork.lock();
            System.out.println("Philosopher " + id + " picked up right fork");
        }

        private void putDownForks() {
            leftFork.unlock();
            rightFork.unlock();
            System.out.println("Philosopher " + id + " put down both forks");
        }

        @Override
        public void run() {
            try {
                while (true) {
                    think();
                    pickUpForks();
                    eat();
                    putDownForks();
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                return;
            }
        }
    }

    public static void main(String[] args) {
        DiningPhilosophers diningPhilosophers = new DiningPhilosophers();
    }
}
```

Readers Writer Problem

```
import java.util.concurrent.Semaphore;

public class DiningReadersWriters {
    private static final int NUM_READERS = 5;
    private static final int NUM_WRITERS = 2;

    private static Semaphore mutex = new Semaphore(1);
    private static Semaphore resource = new Semaphore(1);
    private static int readCount = 0;

    public static void main(String[] args) {
        for (int i = 0; i < NUM_READERS; i++) {
            Thread readerThread = new Thread(new Reader(i));
            readerThread.start();
        }

        for (int i = 0; i < NUM_WRITERS; i++) {
            Thread writerThread = new Thread(new Writer(i));
            writerThread.start();
        }
    }

    static class Reader implements Runnable {
        private int readerId;

        public Reader(int id) {
            readerId = id;
        }

        @Override
        public void run() {
            try {
                while (true) {
                    // Acquire mutex to update the readCount
                    mutex.acquire();
                    readCount++;
                    if (readCount == 1) {
                        // Acquire resource lock to ensure exclusive access for readers
                        resource.acquire();
                    }
                    mutex.release();

                    // Read the resource
                    System.out.println("Reader " + readerId + " is reading");
```

```
                // Release mutex after reading
                mutex.acquire();
                readCount--;
                if (readCount == 0) {
                    // Release resource lock if no readers are reading
                    resource.release();
                }
                mutex.release();

                // Perform some other operations or sleep
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

static class Writer implements Runnable {
    private int writerId;

    public Writer(int id) {
        writerId = id;
    }

    @Override
    public void run() {
        try {
            while (true) {
                // Acquire resource lock for exclusive access
                resource.acquire();

                // Write to the resource
                System.out.println("Writer " + writerId + " is writing");

                // Release resource lock
                resource.release();

                // Perform some other operations or sleep
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}
```