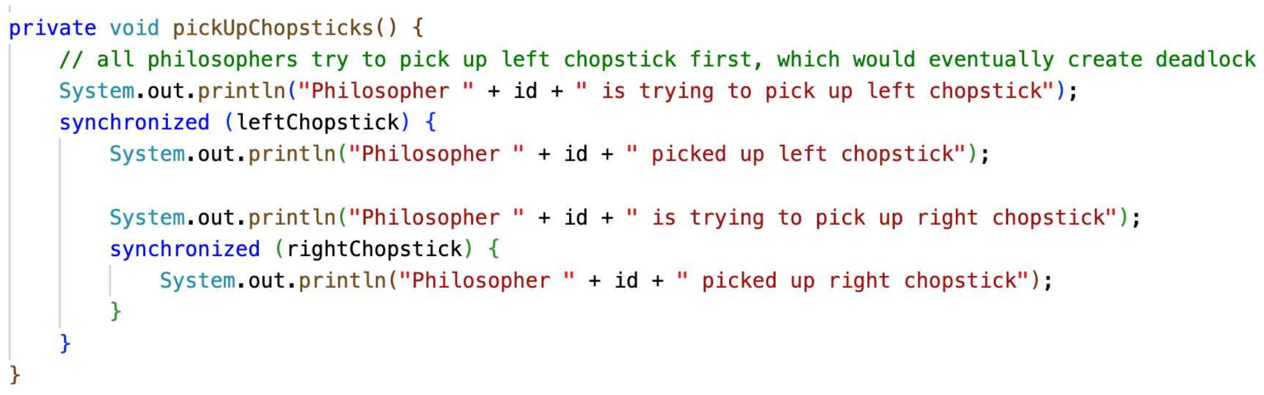
3.1

Preventing Simultaneous Chopstick Access

The program prevents two philosophers from holding the same chopstick by using Java's synchronized blocks with each chopstick object as a monitor lock. The JVM guarantees that only one thread can hold the lock on any given object at a time. When a philosopher enters a synchronized (chopstick) block, other philosophers trying to access that same chopstick must wait until the lock is released when the first philosopher exits the synchronized block. The following code demonstrates the implementation for using synchronized keyword:

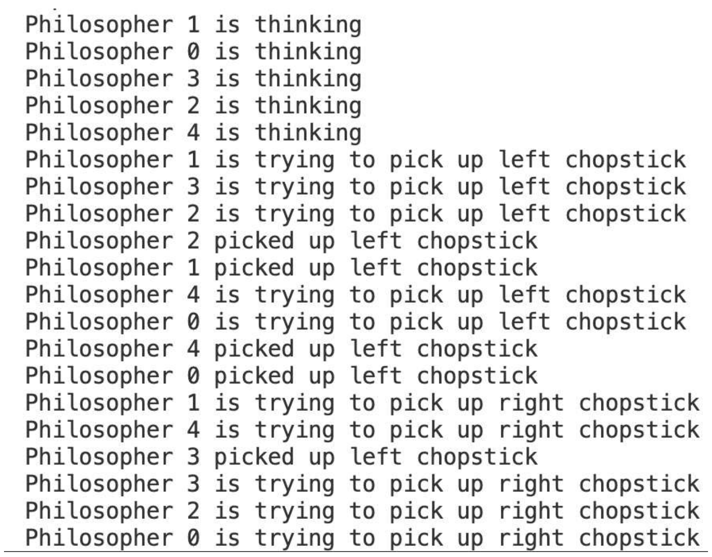


```java
private void pickUpChopsticks() {
    // all philosophers try to pick up left chopstick first, which would eventually create deadlock
    System.out.println("Philosopher " + id + " is trying to pick up left chopstick");
    synchronized (leftChopstick) {
        System.out.println("Philosopher " + id + " picked up left chopstick");

        System.out.println("Philosopher " + id + " is trying to pick up right chopstick");
        synchronized (rightChopstick) {
            System.out.println("Philosopher " + id + " picked up right chopstick");
        }
    }
}
```

Why Deadlock Will Eventually Occur

Deadlock occurs because all philosophers follow the same strategy: pick up the left chopstick first, then the right chopstick. When all philosophers simultaneously grab their left chopstick, each one holds exactly one chopstick and needs their right chopstick to proceed. However, each philosopher's right chopstick is actually another philosopher's left chopstick, creating a circular wait where everyone waits for a resource held by someone else. Since no philosopher will release their chopstick until they finish eating, and no one can eat without both chopsticks, they all become permanently blocked. The following screen shot shows the output of the program that created a deadlock:



```
Philosopher 1 is thinking
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 1 is trying to pick up left chopstick
Philosopher 3 is trying to pick up left chopstick
Philosopher 2 is trying to pick up left chopstick
Philosopher 2 picked up left chopstick
Philosopher 1 picked up left chopstick
Philosopher 4 is trying to pick up left chopstick
Philosopher 0 is trying to pick up left chopstick
Philosopher 4 picked up left chopstick
Philosopher 0 picked up left chopstick
Philosopher 1 is trying to pick up right chopstick
Philosopher 4 is trying to pick up right chopstick
Philosopher 3 picked up left chopstick
Philosopher 3 is trying to pick up right chopstick
Philosopher 2 is trying to pick up right chopstick
Philosopher 0 is trying to pick up right chopstick
```

As we can see, all philosophers are trying to pick up the right chopstick and everyone got stuck. This creates a deadlock.

3.2

Deadlock Prevention Solution

We decided to use ReentrantLock to prevent deadlock. More specifically, we added a timeout-based lock acquisition with ReentrantLock and implemented an all-or-nothing strategy where philosophers must acquire both chopsticks within the timeout period or release any they're holding. Each philosopher uses tryLock(100, TimeUnit.MILLISECONDS) instead of blocking indefinitely on lock acquisition. If a philosopher successfully acquires their left chopstick but cannot get their right chopstick within 100ms, they immediately release the left chopstick in the finally block and retry after a brief pause. This approach breaks the "hold and wait" deadlock condition because no philosopher holds a chopstick indefinitely while waiting for another. Even if all philosophers simultaneously acquire their left chopstick, the timeout mechanism forces them to release it and retry, preventing the circular wait that causes deadlock. The provided code snippet illustrates a significant logic alteration designed to prevent deadlock when picking up chopsticks.:

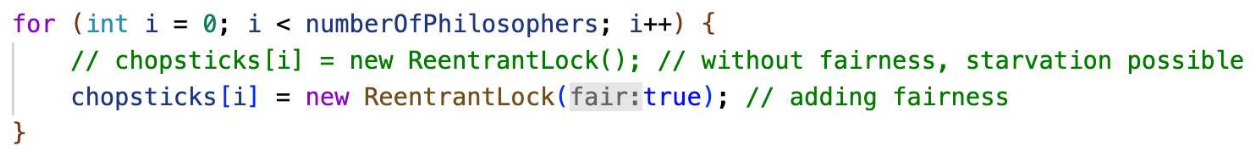
```java
private boolean pickUpChopsticks() throws InterruptedException {
    boolean leftAcquired = false;
    boolean rightAcquired = false;
    try {
        leftAcquired = leftChopstick.tryLock(timeoutMs, java.util.concurrent.TimeUnit.MILLISECONDS);
        if (leftAcquired) {
            System.out.println("Philosopher " + id + " picked up left chopstick.");
            rightAcquired = rightChopstick.tryLock(timeoutMs, java.util.concurrent.TimeUnit.MILLISECONDS);
            if (rightAcquired) {
                System.out.println("Philosopher " + id + " picked up right chopstick.");
                return true;
            } else {
                System.out.println("Philosopher " + id + " failed to pick up right chopstick.");
                return false;
            }
        } else {
            // Failed to acquire left chopstick
            return false;
        }
    } finally {
        if (!rightAcquired && leftAcquired) {
            leftChopstick.unlock();
            System.out.println("Philosopher " + id + " put down left chopstick.");
        }
    }
}
```

Is Starvation Possible?

Yes, starvation is still possible with the basic ReentrantLock solution. The basic ReentrantLock does not provide fairness guarantees, so some philosophers might consistently lose the competition for chopsticks to their neighbors. A philosopher could theoretically keep failing to acquire both chopsticks due to unlucky timing while other philosophers successfully eat repeatedly. Since there's no mechanism to prioritize philosophers who have been waiting longer or failed multiple attempts, individual philosophers could be starved indefinitely even though the system as a whole continues making progress.

To Prevent Starvation:

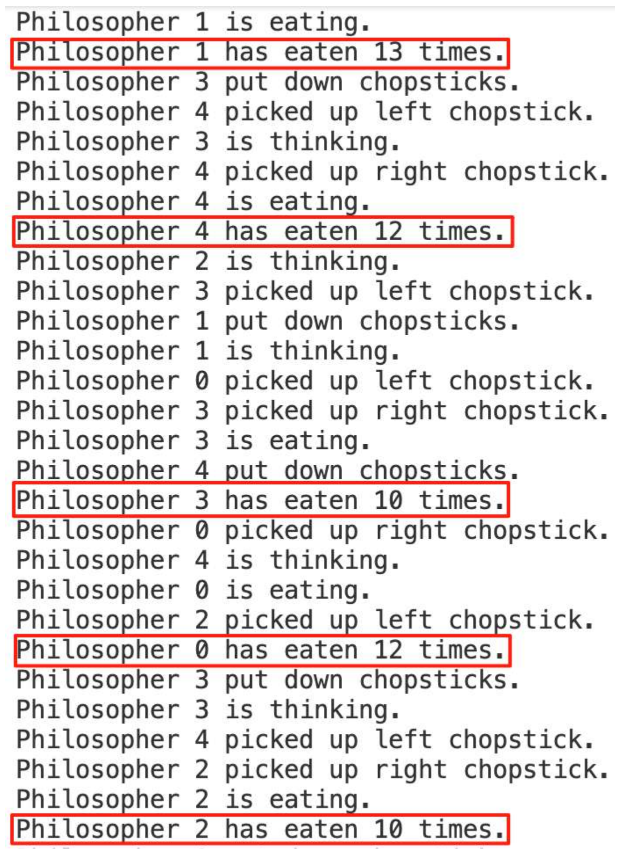
We prevent starvation by adding the fairness property to ReentrantLock using new ReentrantLock(true). Fair ReentrantLocks maintains a FIFO queue of waiting threads and guarantees that the thread that has been waiting the longest will acquire the lock next. This ensures that no philosopher can be indefinitely bypassed by others when competing for chopsticks. When multiple philosophers are waiting for the same chopstick, the fair lock prioritizes the one who started waiting first, preventing any philosopher from being consistently unlucky in the competition for resources.



```java
for (int i = 0; i < numberOfPhilosophers; i++) {
    // chopsticks[i] = new ReentrantLock(); // without fairness, starvation possible
    chopsticks[i] = new ReentrantLock(fair:true); // adding fairness
}
```

How to Test for Starvation-Free Behavior

We can test it by tracking how many times each philosopher eats and reports statistics every 10 meals. In a starvation-free system, all philosophers should have roughly similar eating counts over time, with no philosopher having significantly fewer meals than others. We introduced another variable called eatCount to keep track the number of times each philosopher eats and when we stop the program manually, we can check if every philosopher has eaten similar amount of times, here is the output:



```
Philosopher 1 is eating.
Philosopher 1 has eaten 13 times.
Philosopher 3 put down chopsticks.
Philosopher 4 picked up left chopstick.
Philosopher 3 is thinking.
Philosopher 4 picked up right chopstick.
Philosopher 4 is eating.
Philosopher 4 has eaten 12 times.
Philosopher 2 is thinking.
Philosopher 3 picked up left chopstick.
Philosopher 1 put down chopsticks.
Philosopher 1 is thinking.
Philosopher 0 picked up left chopstick.
Philosopher 3 picked up right chopstick.
Philosopher 3 is eating.
Philosopher 4 put down chopsticks.
Philosopher 3 has eaten 10 times.
Philosopher 0 picked up right chopstick.
Philosopher 4 is thinking.
Philosopher 0 is eating.
Philosopher 2 picked up left chopstick.
Philosopher 0 has eaten 12 times.
Philosopher 3 put down chopsticks.
Philosopher 3 is thinking.
Philosopher 4 picked up left chopstick.
Philosopher 2 picked up right chopstick.
Philosopher 2 is eating.
Philosopher 2 has eaten 10 times.
```