# **Dining Philosophers and Starvation**

## Why None of the Three Dining Philosophers Solutions Are Starvation-Free

Although all three classical solutions (using synchronized, Semaphore, or ReentrantLock + Condition) prevent deadlock, they do not guarantee starvation freedom. That is, a philosopher might remain hungry indefinitely—even though the system as a whole continues to make progress. Here's why:

## Solution 1: synchronized Monitor

In the monitor-based solution, a hungry philosopher waits in a loop until both neighbors are not eating. When a philosopher finishes eating, they call notifyAll(), waking up all waiting threads.

#### Why starvation can happen:

- When notifyAll() is called, many philosophers may be awakened simultaneously.
- The awakened threads re-check whether they are eligible to eat using canEat().
- However, thread scheduling is **not fair**—Java's monitor does not guarantee which thread will re-acquire the lock first.
- As a result, some philosophers may keep getting preempted by others and never get a chance to eat.

## **Solution 2: Semaphores**

The semaphore-based solution avoids deadlock by acquiring chopsticks in an asymmetric order (left then right, or right then left). However, it does not coordinate between philosophers using any shared state.

#### Why starvation can happen:

 If one philosopher is slightly slower in acquiring a chopstick, their neighbors might continuously acquire and release both chopsticks before the slow philosopher has a chance. • Since semaphores do not have fairness built-in by default, the **same philosopher may keep getting blocked indefinitely**—even though the others are cycling through eating and thinking.

## Solution 3: ReentrantLock + Condition

This solution uses explicit conditions to allow neighbors to signal when they are done eating. A philosopher waits on their own condition variable until both neighbors are not eating.

#### Why starvation can happen:

- A philosopher can only eat when both neighbors are not eating and the lock is available.
- Even though this solution uses a *fair* lock (new ReentrantLock(true)), **Condition** signaling is not guaranteed to be fair.
- A philosopher who becomes eligible may be skipped over in favor of others who happen to signal each other more quickly or more often.

### Conclusion

To guarantee starvation-freedom, additional mechanisms must be introduced, such as:

- Queueing philosophers in order of arrival (FIFO fairness).
- Tracking how long each philosopher has been waiting and prioritizing the longestwaiting.
- Using explicit fairness policies or token-passing mechanisms.

The classic solutions are good for avoiding deadlock, but not sufficient to ensure **fairness** or **bounded waiting**.