Dining Philosophers and Deadlocks

How to Introduce Deadlocks in the Dining Philosophers Solution 2 (Semaphore-Based)

The semaphore-based solution models each chopstick as a binary semaphore. Philosophers try to acquire two chopsticks (their left and right) before eating.

Default Behavior: Deadlock-Free

In the standard implementation, deadlock is avoided by introducing *asymmetry*:

```java
if (i % 2 == 0) {
    chopsticks[left].acquire();
    chopsticks[right].acquire();
} else {
    chopsticks[right].acquire();
    chopsticks[left].acquire();
}
```

This ensures that not all philosophers grab the same chopstick first, breaking the possibility of circular wait.

To Introduce Deadlock

To deliberately create a deadlock, **remove the asymmetry** and have all philosophers acquire chopsticks in the same order:

```java
// DEADLOCK-PRONE VERSION
public void pickup(int i) throws InterruptedException {
    int left = i;
    int right = (i + 1) % NUM_PHILOSOPHERS;

    chopsticks[left].acquire();
    chopsticks[right].acquire();
}
```

With this code:

1. Each philosopher grabs their **left chopstick**.
2. Then, they all try to grab their **right chopstick**.
3. But each right chopstick is already held by the neighbor, creating a circular wait.
4. No philosopher can proceed, and all are blocked — this is a **deadlock**.

Classic Circular Wait

This deadlock stems from the four Coffman conditions being satisfied:

- **Mutual exclusion**: Each chopstick can be held by only one philosopher.
- **Hold and wait**: Each philosopher holds one chopstick and waits for the other.
- **No preemption**: Chopsticks are only released voluntarily.
- **Circular wait**: Each philosopher waits for a chopstick held by their neighbor.

How to Observe It

In practice, when you run this version:

- All philosophers may print "waiting for right chopstick..." and then stop progressing.
- No one eats, and the program appears to hang.

This is a textbook deadlock scenario.