**The Solution to the Dining Philosophers Problem:**

The solution to the dining philosophers' problem is to use [**Semaphore**](https://www.javatpoint.com/java-semaphore). It is a tool that is used for concurrent processes. There is a drawback of using Semaphore as a solution. It may lead to **a deadlock**. Suppose a scenario **when all philosophers pick up the left fork and wait for the right fork**. The situation leads to a deadlock.

To avoid the deadlock solution, there are some measures that should have been taken are:

* There should be at most four philosophers around the table.
* A philosopher should allow picking forks when both forks are available at the same time.
* The philosophers can alternatively eat and think. For example, if the first philosopher is eating then the adjacent neighbor philosophers should wait and think, and so on.

**Implementation of Dining Philosophers Problem:**

In the following program, first, we have initialized the number of **philosophers** (5). The two arrays **philosophers[]** and **chopsticks[]** initialized with the number of philosophers (5).

In order to implement the logic for chopsticks, we have created a class named **Chopstick**. Inside the class, we have created a constructor of the Semaphore class and defined three methods namely **grab(), release(),** and **isFree().**

The grab() method invokes the **acquire()** method that acquires a permit from this semaphore. It **reduces** the number of permits by 1. If there is no permit is available the current thread becomes disabled.

The user-defined release() method invokes the **release()** method of the Semaphore class. It releases the given number of permits and **increments** the permit by 1.

The **isFree()** method checks for the availability of permits in the semaphore. It invokes the **availablePermits()** method of the Semaphore class that returns the number of permits available in the Semaphore.

After that, we created another class named **Philosopher** that extends the **Thread** class. Inside the class, we have defined three variables **number, leftchopstick**, and **rightchopstick.** Also, we have created a **constructor** of the **Philosopher** class and the **eat()** method. The [**run()**](https://www.javatpoint.com/java-thread-run-method) method of the Thread class is called if the thread was constructed using a separate Runnable object.

The **run()** method executes when a philosopher has both chopsticks (right and left). A philosopher starts eating by invoking the **eat()** method and holds the fork for a specified time (sleep time). In order to determine the sleep time, we have used the [**ThreadLocalRandom**](https://www.javatpoint.com/java-threadlocalrandom-nextint-method). [**current()**](https://www.javatpoint.com/java-threadlocalrandom-current-method). [**nextInt()**](https://www.javatpoint.com/java-threadlocalrandom-nextint-method) method. The method returns a pseudorandom, uniformly distributed integer value between **0** and **1000**. The value is determined by the **nextInt()** method passed in the **sleep()** method that sleeps the thread for a specified time. In our case, if sleep time is more than 1000ms, the program will exit. So, we can set the range of pseudorandom numbers (sleep time).

Inside the **main()** method, we have defined the two for loops one for chopsticks, and the other for philosophers. After that, we checked for deadlock conditions. If a deadlock occurs, it means each philosopher is eating by acquiring the chopsticks. The execution of the program breaks. The deadlock condition will not occur until resources (a single chopstick) are available.

**References:**

<https://www.javatpoint.com/dining-philosophers-problem-and-solution-in-java>