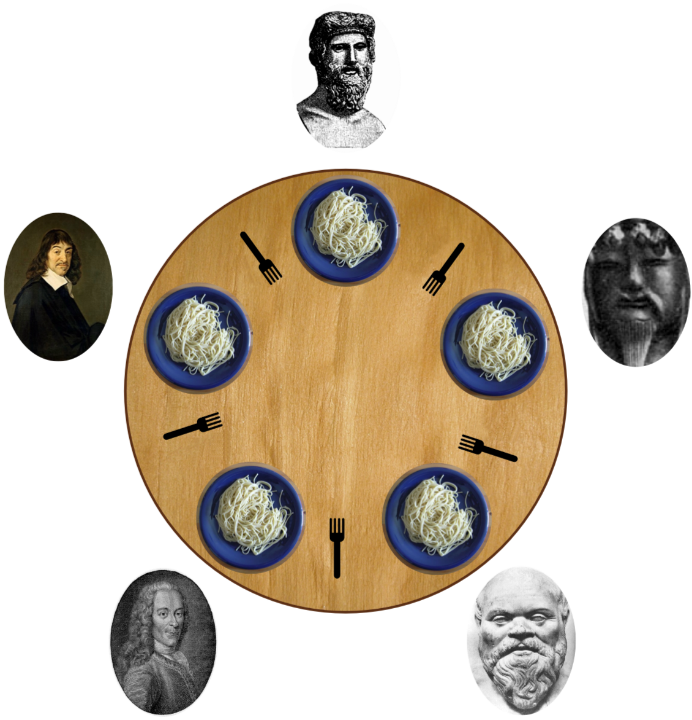
**Threads and Locks – Dining Philosopher Problem**

**Part 1 – The Problem:** This is classical computer science problem used to illustrate the potential synchronization issues with multi-threading. Suppose you have 5 philosophers and 5 forks (it could be any number of philosophers and forks as long as the two variables are equal). Each philosopher is sitting at the table such that there are two forks nearby (one to the left and to the right) and a plate of pasta in front of the philosopher. In order for a philosopher to eat, the philosopher must have both forks. Write a program such that all philosophers can eat while avoiding starvation and deadlock. Starvation is when one philosopher (or in general one thread) never gets access to the resource that is being monitored (in this case the forks). Deadlock is when all each philosopher (or in general each thread) is waiting for another to finish.



**Part 2 – Locking/Monitoring the Resource:** We must create a fork object class such that each fork can only be picked up by one philosopher at a time.

|  |
| --- |
| **public** **class** Fork {  Lock lock = **new** ReentrantLock();  **private** **final** **int** id;  **public** Fork(**int** id) {  **this**.id = id;  }  **public** **boolean** tryToPickUp(Philosopher philospherName, String direction){  **if** (lock.tryLock(10, TimeUnit.*MILLISECONDS*)) {  System.*out*.println(philospherName + " picked up " + direction + " " + **this**);  **return** **true**;  }  **return** **false**;  }  **public** **void** putDown(Philosopher philospherName, String direction) {  lock.unlock();  System.*out*.println(philospherName + " put down " + direction + " " + **this**);  }  @Override  **public** String toString() {  **return** "Fork -" + id;  }  } |

So you have a lock as an attribute. When a philosopher tries to “pick up” this fork, it will either print a statement and return true or print false. This depends on whether the lock is being held by another philosopher thread. The direction is either “right” or “left” and is used for printing.

**Part 2 – The Philosopher Class:**

|  |
| --- |
| **public** **class** Philosopher **implements** Runnable {  **public** **static** **final** **int** *NUM\_MEALS\_UNTIL\_FULL* = 4;  **private** **final** **int** id;  **private** **final** Fork leftFork;  **private** **final** Fork rightFork;  **public** **int** numberOfMealsEaten = 0;    **public** Philosopher(**int** id, Fork leftFork, Fork rightFork) {  **this**.id = id;  **this**.leftFork = leftFork;  **this**.rightFork = rightFork;  }  @Override  **public** **void** run() {  **try** {  **while** (!isFull()) {  think();  tryToEat();  }  } **catch** (Exception e) {  e.printStackTrace();  }  }    **public** **boolean** isFull(){  **return** numberOfMealsEaten >= *NUM\_MEALS\_UNTIL\_FULL*;  }  **private** **void** think() **throws** Exception {  System.*out*.println(**this** + " is thinking");  Thread.*sleep*(**new** Random().nextInt(1000));  }    **public** **void** tryToEat() **throws** Exception{  **if** (leftFork.tryToPickUp(**this**, "left")) {  **if** (rightFork.tryToPickUp(**this**, "right")) {  eat();  rightFork.putDown(**this**, "right");  }  leftFork.putDown(**this**, "left");  }  }  **private** **void** eat() **throws** Exception {  System.*out*.println(**this** + " is eating");  numberOfMealsEaten++;  Thread.*sleep*(**new** Random().nextInt(1000));  }  @Override  **public** String toString() {  **return** "Philosopher-" + id;  }  } |

**Note:** The “try to eat” method is the most thread sensitive method. It is the part of the code that access a resource that is also being accessed by other threads. You must ensure that the program either does it what its try to do with this shared resource (in this case it is to eat) or leaves the resource. The worst thing that can happen is if the thread doesn’t finish yet doesn’t allow others to use the shared resource. Ensure that both forks get dropped (by calling “put down”) when you can eat and also when you cannot.

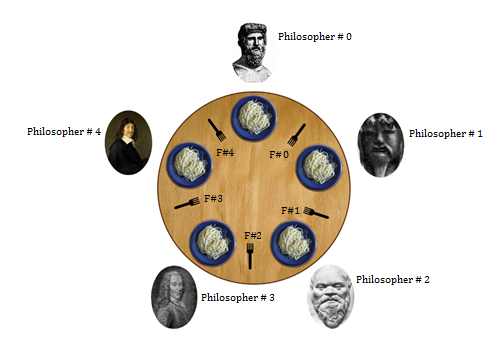
**Part 3 – The Main Class:**

This class simply initializes all the forks, initialize and starts all the threads, then waits for 15 seconds. Once there was enough time for each philosopher to eat (if there was no starvation or deadlock) it prints out the number of times each philosopher ate. Each philosopher should have eaten four times.

|  |
| --- |
| **public** **class** Question {  **private** **static** **final** **int** *NO\_OF\_PHILOSOPHER* = 5;  **private** **static** **final** **int** *NO\_OF\_CHOPSTICKS* = *NO\_OF\_PHILOSOPHER*;  **public** **static** **void** main(String args[]) **throws** InterruptedException {  Philosopher[] philosophers = **new** Philosopher[*NO\_OF\_PHILOSOPHER*];  Fork[] forks = **new** Fork[*NO\_OF\_CHOPSTICKS*];  *initializeAllForks*(forks);  *initializeAndStartAllPhilosophers*(philosophers, forks);  Thread.*sleep*(15000);//pause the program until they are done eating  *printNumberOfMealsEaten*(philosophers);  }    **public** **static** **void** printNumberOfMealsEaten(Philosopher[] philosophers){  System.*out*.println("\n");  System.*out*.println("----------------------------------------------");  **for** (Philosopher philosopher : philosophers) {  System.*out*.println(philosopher + " --> No of meals = "  + philosopher.numberOfMealsEaten);  }  }    **public** **static** **void** initializeAllForks(Fork[] forks){  **for** (**int** i = 0; i < *NO\_OF\_CHOPSTICKS*; i++) {  forks[i] = **new** Fork(i);  }  }    **public** **static** **void** initializeAndStartAllPhilosophers(Philosopher[] philosophers, Fork[] forks){  **for** (**int** i = 0; i < *NO\_OF\_PHILOSOPHER*; i++) {//for each philosopher  //initialize the philosopher  **int** philosopherId = i;  Fork leftFork = forks[i];  Fork rightFork = forks[(i + 1) % *NO\_OF\_PHILOSOPHER*];  philosophers[i] = **new** Philosopher(philosopherId, leftFork, rightFork);    //start the philosopher  Thread thread = **new** Thread(philosophers[i]);  thread.start();  }  }  } |

**Assigning Forks to Philosophers:**

If you have 5 forks (0 – 4) and 5 philosophers (0 – 4), how do you distribute your philosophers such that each philosopher gets one of the forks as his/her left fork and another of the forks as his/her right fork?



* Each philosophers left fork is their number. So for example, philosopher 2’s left fork is fork number 2.
* Each philosopher’s rught fork is their number subtracted by one. For example, philosopher 2’s right fork is fork number 1. The catch is that philosopher 0’s left fork is fork number 4.