CentraleSupélec

IS1220

Tutorial 10

Francesca Bugiotti, Paolo Ballarini

General instructions:

- Ask if you have any problem.
- If you have not done so already, create a project for your tutorial `fr.ecp.IS1220.Tutorial10`.
- Within the project, create a new package for every exercice N of this TD called `fr.ecp.IS1220.Tutorial10/`.
- Carefully document your code (JavaDoc), i.e., explain the general idea of your algorithm, explain the relevant steps in the code implementing the algorithm, document assumptions (if any), corner cases, and error conditions.

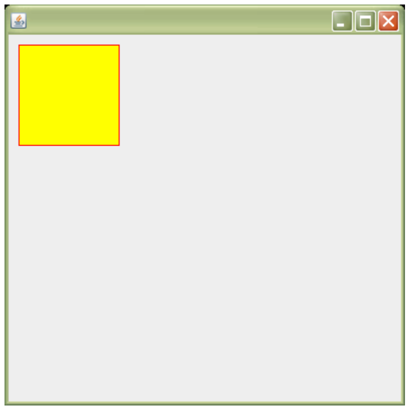
Main Concepts:

- Concurrency
- Graphical interfaces

Exercise 1. Summary

E1 Q1) Describe the Lock Mechanism

E1 Q2) Implement a simple graphical interface. This interface must contain a colored square like in the figure below. You can choose the color of the square.



Exercise 2. Dining Philosophers

Let us consider a group of N (for example $N = 5$) philosophers sitting around a circular table with N chairs. The behaviour of each philosopher consists of perpetually switching between two activities: thinking for a random duration (uniformly distributed over a given interval) and eating for a random duration. Eating is performed using chopsticks. In order to eat, a philosopher sitting at the table must acquire two chopsticks, situated on its left and right side. Only one chopstick is placed between two philosophers, therefore philosopher i shares his right chopstick with philosopher $i + 1$ and his left chopstick with philosopher $i - 1$.

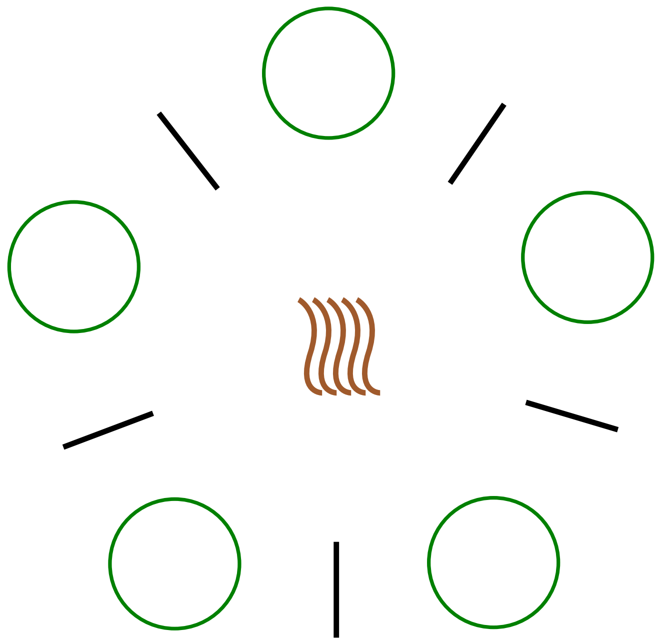


Figure 1: The disposition of the philosopher's table for $N = 5$.

The behaviour of a philosopher is described by the following actions:

1. think (for a duration $\delta_{think} \in [minThink, maxThink]$);
2. pick up chopsticks (one at a time, i.e. cannot pick both chopsticks in one operation);

3. eat when acquired both chopsticks (for a duration $\delta_{eat} \in [minEat, maxEat]$);
4. put down chopstick and start over (thinking).

Notice that since each philosopher shares chopsticks with its two neighbors, he cannot eat at the same time as either immediate neighbor. You are asked to implement a Java solution of the problem, making sure that the philosophers cannot starve i.e. can forever continue to alternate between eating and thinking assuming that any philosopher cannot know when others may want to eat or think. A possible structure of the code can be the following:

- a *Philosopher* class that will represent the thread;
- a *DiningPhilosophers* main class that will define 5 *Philosopher* instances and 5 chopsticks.

A philosopher must be able to think, eat, pick up the left chopstick, pick up the right chopstick and put down both chopsticks.

E2 Q1) You are first asked to implement a circular chain, each Philosopher tries to get the left Chopstick first and then the right. We suppose that a philosopher thinking process takes a duration $\delta_{think} \in [10, 15]$ time units while the eating takes him a duration $\delta_{eat} \in [5, 10]$ time units.

E2 Q2) What do you notice when you change the duration of each action to zero? In other words, no thread is put to sleep at any action that it performs (eating, thinking).

E2 Q3) What is the design solution to avoid the problematic situation? Update the implementation.

E2 Q4) **Limited food supply:** Let us consider now that the amount of available food is no longer unlimited. The philosophers will be able to eat and think only for a limited amount of time. The food is defined by a quantity that diminishes by a random quantity based on the given appetite of each philosopher defined by $\delta_{appetite} \in [minApp, maxApp]$.

E2 Q5) Extend the previous solution in order to simulate the consumption of a shared food resource. Only a philosopher at a time can help himself from the food.

E2 Q6) What could go wrong in the case of the limited amount of food? Find a solution and test it (in the sens of JUnit test case) for 10 philosophers that compete for a food resource of 30 units with a hunger defined by $\delta_{appetite} \in [1, 5]$. The test should launch the dining simulation for 100 times and make sure that every simulation was valid with regards to the observed finit supply issue.

E2 Q7) Extend the previous solution in order to ensure the fairness of the thread synchronisation, in other words that the food is shared in an equitable way by the hungry philosophers.

Exercise 3. Dining Philosophers: Locks vs Synchronised)

Let us now consider an architecture for the Dining Philosophers problem that is able to take into account different implementation of the concurrency to shared resources.
With minimum modifications to the existing DiningPhilosophers class, implement two version of the shared resource management, one using `synchronised` methods and another one using an implementation of the `Locks` interface in the Java API. You may have implemented one of them already in the previous exercice. The user must now be able to choose between the synchronisation mechanism before launching the problem.

E3 Q1) What can you say about the *fairness* property? Do all philosophers eat in a fair way?

E3 Q2) Implement a test case that can analyse and display the fairness of both methods.