# Assignment #4: Introducing Threads and Synchronization in Java

## 1 Preliminaries

This practical introduces the principles of Java synchronization through the simulation of a kitchen. Starting from a sequential implementation of the kitchen, your task will be to augment it with parallel processing capabilities. The sequential implementation of the kitchen is available on the portal Moodle UniNe of the course.

In order to have a true parallelism, it is recommended to use a multi-core machine in this practical.

As usual, you have to comment your code *appropriately*. This means that you should indicate (i) the problem you were solving, and (ii) the modifications you made to the code in order to solve it; you can be verbose. Feel free to provide separate `.java` modules in separate folders if questions overlap too much for your liking (individual folders should compile independently). The deadline for uploading your commented code on Moodle is **May 6, 2022 at 14:00**.

## 2 Description of the Sequential Implementation

Below, we detail each of the classes in the sequential implementation.

### 2.1 Stocks

The `Stock` class represents a *stock of food*. Two access methods, `put()` and `get()` respectively add and remove one food. The `display()` method prints the name of the stock and the number of remaining food on the standard output. The constructor takes the name of the stock and the initial number of food, as parameters.

### 2.2 Stoves

The Stove class represents a *stove* to prepare food. The `prepare()` method gets a food from an initial stock A, waits for 64 ms (to simulate some real work done on the food, *e.g.*, boiling, cutting), and then adds a crafted food to stock B. The `work()` method performs several such preparations. The constructor takes as parameters two instances of the Stock class and the number of preparations to perform.

### 2.3 Kitchen

The Kitchen class represents a kitchen with two stoves, and two stocks of food. Initially, stock A contains 16 food, while stock B is empty. Stock B shall receive 16 processed food by the end of the execution. The `work()` run two stoves *one after the other*, then displays the status of each stock.

## 3 Let's Work

It is recommended to first have an overview at the code before delving into the questions that follow.

In general, a good approach to get familiar with some source code, is to run/create a *unit test* for each class. A unit test consists in a `main()` method, specific to that class, and several tests of the class' methods, in order to determine if they are correct.

### 3.1 Q1: Working in Parallel

*For questions Q1 and Q4, it is recommended to first ignore the concurrency problems. We will implement synchronization mechanisms that solve them later.*

It takes 1.024 seconds to prepare the 16 food of stock A. If the two stoves work in parallel, 512 ms is enough. Thus,

- Change the code of `Stove` to inherit from `Thread`.
- Using the above modification, parallelize the method `work()` in the `Kitchen` class.
- Compile, run and check the correctness of your results.
- Do you observe any problem? If there is one, correct it.

### 3.2 Q2: When Things Go Wild

Every execution of Q1 appears to be correct now; but is this really the case? We will now consider a much faster pace of accessing stocks and preparing food, and see if such an execution is still correct.

- Change the production capacity of your kitchen to 100 million food, 50 millions per stove.
- To avoid waiting several days that this work be done, comment the call to `sleep()` in the `prepare()` method. (Each preparation is now almost instantaneous.)
- Compile and run.
- Find the concurrency problem and correct it.

### 3.3 Q3: Thread Scheduling Determinism

We would like to see when the processor decides to swap between threads.

- Go back to a production of 16 food with a waiting time of 64 ms for `prepare()`.
- Change the source code to follow the stock evolution as in the example below.[1]

```
Thread 2:  stock "initial" contains 9 food.
```

- Recompile, run several times and observe the results;
- Can you draw any conclusions about the scheduler?

[1] The name of the current thread can be obtained with a call to `Thread.currentThread().getName()`.

# 4 Let's Get More Productive

## 4.1 Q4: Work Re-organization

*Similarly to Q1, do not take into account synchronization problems that appear at first glance.*

The chef decides that stoves should be specialized. To that end, you should modify the balancing of work between the two stoves. Instead of having each stove operating on half of the initial stock, each will do half of the work on all food. This means that (i) the first stove picks the initial food in stock A and feeds them in an intermediate stock C, and (ii) the second stove picks food from C, and stores them in the final stock B.

- Do the necessary modifications to the code in order to implement the changes.
- Recompile, run and observe the traces.

## 4.2 Q5: A Little Nap Is Always Good

You certainly observed that the intermediate stock C oscillates between -1 and 0 food. Indeed, the second stove tries to pick a food in stock C before the first stove feeds it. Actually, if a stock is empty, the stove who comes to pick up food should go for a little nap, until there is something available. It is then the responsibility of the one who feeds the stock to wake up the sleeper.

- Implement this behavior by adding the necessary synchronization mechanisms. For this question, it is recommended to be thrifty: if `notify()` works, do not employ `notifyAll()`.
- Recompile, run several times and observe the traces.

## 4.3 Q6: Things Always Happen in Threes

Once again, we try to push our kitchen simulation code to cases where it is not correct. To this goal, we shall create a third stove. It will share half of the work of the second stove (getting/putting food from the same stocks).

- Give half of the work of stove 2 to the new stove 3;
- Launch stove 1 last;
- Recompile, run, check the traces;
- Fix problems, if any.

## 4.4 Q7: Productivity, Even More Productivity

To further increase return on investments, the chef now decides to turn the kitchen into a fast-food kitchen (*i.e.*, the pace of working will be increased), and that the work will be organized in a just-in-time manner (*flux tendus*). The `prepare()` operations will now be instantaneous and the intermediate stock limited to solely one food. The first stove will be doubled, leading to a total of four stoves now.

- To make the preparation instantaneous, comment the call to `sleep()` and remove intermediate prints to standard output.
- We now use 10 thousand food in stock A;

- Add a new parameter to the `Stock` class constructor that represents the maximum amount of food an instance is allowed to store;
- Add the necessary synchronization to forbid the stock quantity going over this value;
- Double stove 1, assign half of its work to a new stove;
- Recompile, run several times, check the logs;
- What happens? Find the problem and fix it.