SE assignment **Univ. Rennes 1**

Concurrent Airline Agents

The director of an Airline company wishes to improve the performances of his agents. The instructions that the agents execute are supposed to be specified in Java language, and every agent is supposed interacting with two main Java classes: the `Customer` and the `Aircraft`. The director wants his agents to get used to work with a new computer system, where every agent will not have to wait for one another in order to have access to the reservation system.

In order to help the director to achieve this task, let's start to give a look at the implementation of the two main Java classes. The director himself has coded these two classes, and he asks that *nobody* changes his code without discussing the matter personally with him.

Customer

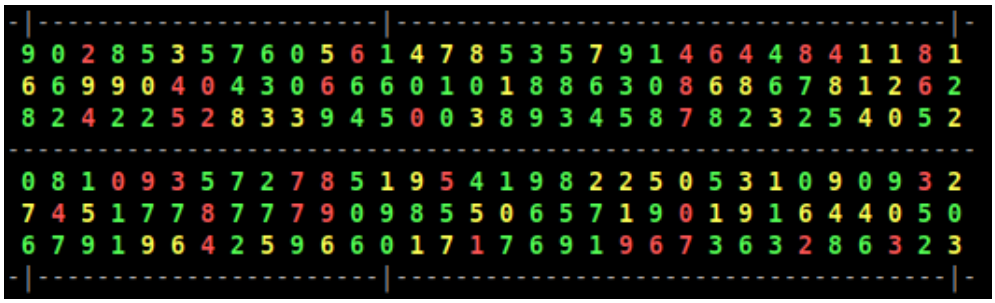
The `Customer` class provides information about Airline customers. Please open the Java file and explore its attributes and its methods. The Java file also contains a main method that performs some tests on its methods. When printing a `Customer` object, this is an example of information given on the screen:

```
(Customer 68; aged 32; flyer level = 2; ticket number = 21270745; flight cost = 1889)
```

Aircraft

The `Aircraft` class provides information about the flight, including its seat map, where objects of the `Customer` class can be affiliated. The `Aircraft` class has a constructor that requires no input arguments and that generates a predefined model for the Aircraft seat map. Another constructor can be used for genering other Aircraft's models.

As previously done with the `Customer` class, please take some time to explore the attributes and the methods of the `Aircraft` class. The director took his time to implement a very nice graphical representation of the seat map (see method `toString`). An example of representation is given below:



where the red color is used to represent customers that need assistance, the yellow color indicates customer that are over 60, while the green color marks all other customers. The displayed numerical value is the flyer frequency number of the customer (the higher, the most frequent). The symbol `x` shows that the seat is still empty. Notice that, in case your color system is not compatible with the one used by the director, you can use the method `switchoffColors` to have an alternative black-and-white representation.

Simulating the behavior of a set of agents

The two main agents

The director had asked each of his agents to write a Java method indicating the actions they perform for the reservation of a customer seat in a given flight. The two main agents, *agent1* and *agent2*, wrote their Java method immediately upon the director's request. To make things easy (for themselves), they negleted some details, but the two codes seem nevertheless able to work properly (notice that we don't need to correct or improve these codes). Please give a look at these codes (see the `AirSimulation` class) to find out how these two methods are supposed to perform the reservations.

The third agent

The third agent is a little ashamed to reveal the way he proceeds, and this is the reason why he has not written yet his Java code. In practice, the director has recruited him with the aim of taking care of the mistakes that the other two agents may make in their reservations. The main task of this third agent is in fact to verify if customers with a high frequent flyer number are actually positioned "in front of" customers with a smaller flyer number. As you can see from the codes of *agent1* and *agent2*, they both do not consider at all this kind of information when selecting a seat for the customers.

The shame of *agent3* comes from the fact that his method is not really professional, even if it has some effectiveness. In practice, he picks randomly two seats in the seat map, he verifies the frequent flyer numbers of the two customers placed on them, and if the rule "the most frequent flyer flies on upper airplane rows" is not satisfied, then he simply switches the seats of the two customers.

Since we urge to continue working on this code, the director kindly asks you to code the Java method (inside `AirSimulation`) implementing the behavior of *agent3*.

A new computer system

In order to improve the overall performances of the Airline, the director proposes therefore to install a new multi-tasking computer machine with shared memory. Every agent can this way be associated to one single and independent computing resource. In Java, this can be implemented by using the `Thread` class. New threads can be invoked from a running thread with the method `start()` of `Thread`, but it is important to remark that the default execution of `Thread` is empty. It is necessary therefore to *extend* the `Thread` class in order to specify what code every new generated thread needs to run. To do so, the method `run` of `Thread` needs to be overridden (when `start` is invoked, the code in `run` should in fact be executed).

Modify your sequential main method in `AirSimulation` so that only *agent1* is executed in the main thread, while all other agents are executed by newly generated threads.

Comparing the performances

The Airline director is very excited about the new computer system, and decides to make a performance verification, aiming at comparing the new system to the old one. In order to evaluate the performances of the two systems, the director includes, in the main method of `AirSimulation`, the method `System.currentTimeMillis()`. What do you think the director is going to discover: is the multi-thread version actually faster?

Troubles in the Airline

An ancient employer, apparently kicked out from the Airline company for no serious reasons, is able to introduce a virus in the new Airline computer system. The code implementing a new agent, let's call it *agent4*, is included by the virus inside the `AirSimulation` class. This new agent has the following code:

```
for (int i = 0; i < this.a.getNumberOfRows(); i++)
{
   for (int j = 0; j < this.a.getSeatsPerRow(); j++)
   {
      Customer c = this.a.getCustomer(i,j);
      this.a.freeSeat(i,j);
      if (c != null)  this.a.add(c,i,j);
   }
}
this.nAgent4++;
```

Unfortunately, this code makes the new computer system unstable! Some small execution errors that the director had actually already noticed when the new system was finally functional, are now more and more frequent ... :-(

An expert suggestion

Very concerned for his new computer system, the director asks for an expert advise. The expert explains to the director that, in this kind of computer systems, the memory is shared by the agents and it is necessary to pay particular attention to the way every agent has access to it. In Java, he continues, `Semaphores` can be used for controlling the access to the memory resources.

Even if the director had never heard about semaphores before, after a quick online search, he finds some simple examples of `Semaphore` use that allow him to fix the system! By using only one `Semaphore`, he makes sure that only one agent per time can have access to the seat map. Thanks to this modification, the system is stable again, while the virus is still active (*agent4* continues to run the code given above; actually the director had not recognized the virus code and he has introduced the `Semaphore` also in there...).

Doubts about the overall performances

After having successfully fixed the system (even if the virus is still running), all director happiness fades away when he finds out that the use of the `Semaphores` makes his system slower. And actually, the computer system is now even slower than his old system. A solution is necessary in order to make the system competitive again with the old one.

Can you imagine a less restrictive use of `Semaphores` that allows the new computer system to avoid crashing? Finally, the director's happiness is totally in your hands. ;-)