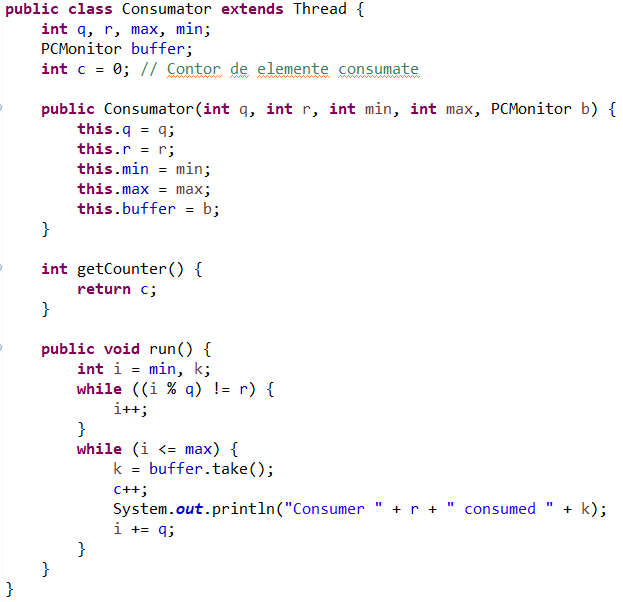
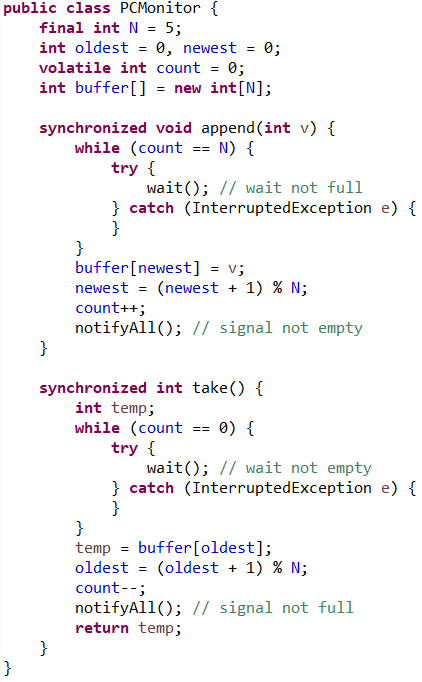
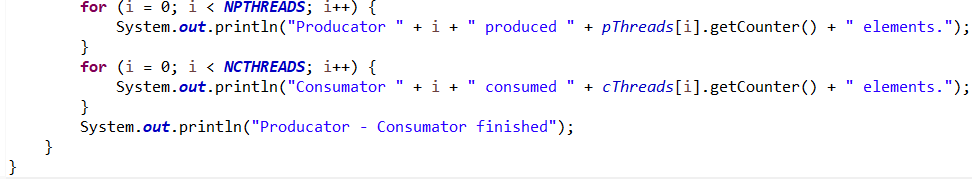
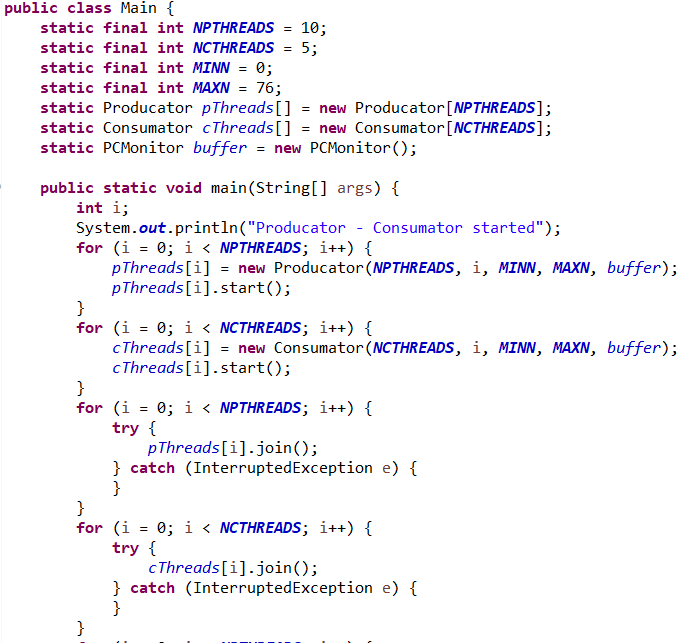
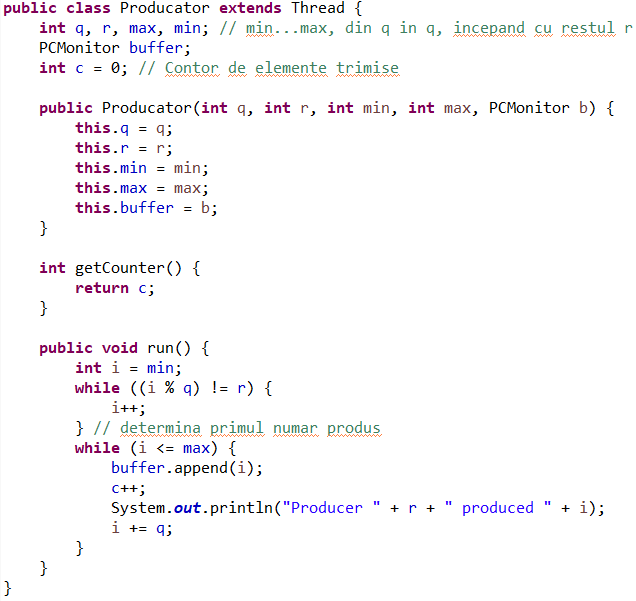
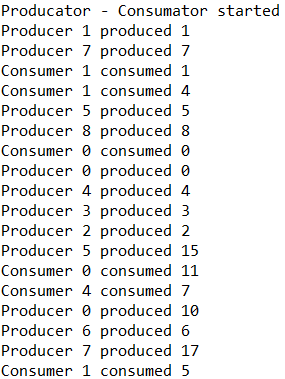
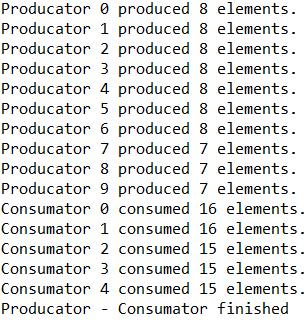
**Laboratory 7**

1. I implemented the Producer-Consumer example as follows:





The result of running this code is:

 ………. 

The previous code has 10 producers and 5 consumers, each of them with its own task.

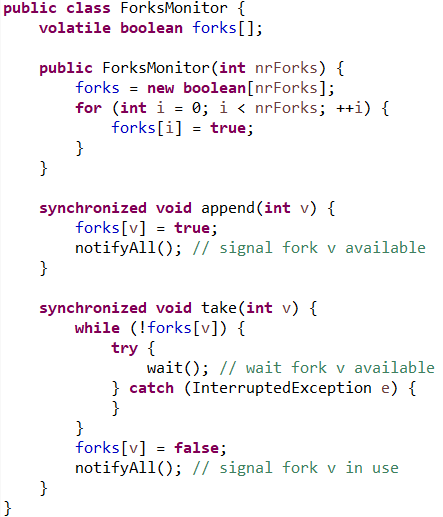
Each producer is creating an integer that has the rest of division by the number of producers equal to their ID number, and puts the generated integer in a buffer of size 5, common for all producers and consumers. If this buffer is fully occupied, it will wait for the last entity to be consumed before putting the newly created one in its place.

Each consumer will count from a minimum given number up to a maximum given number, and every time the counter is a number that divided by the total number of consumers creates the rest equal to the consumer id, the consumer will try to consume the oldest integer generate by the producers in the buffer. If the buffer is empty, the consumer will wait till a new entity will be placed in the buffer.

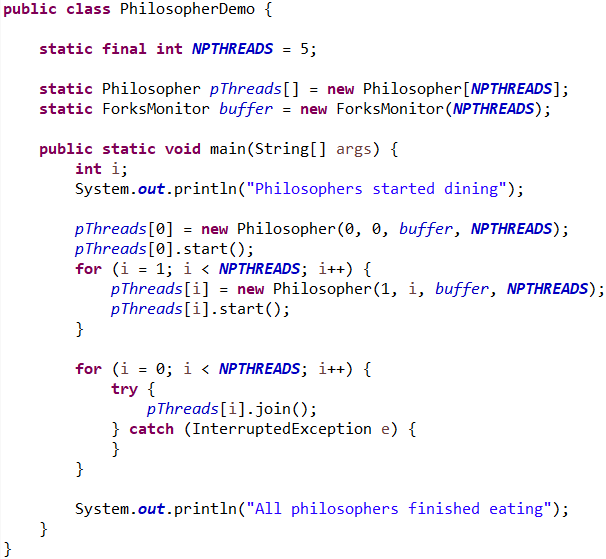
In the end all the producers will have similar number of integers produced (the range of the interval divided by number of producers), each of them creating an integer by the algorithm explained above, resulting in all the elements in the given range being produced only once.

The consumers will also have similar number of integers consumed (the range of the interval divided by number of consumers), but those will simply consume the oldest entity produced in the buffer.

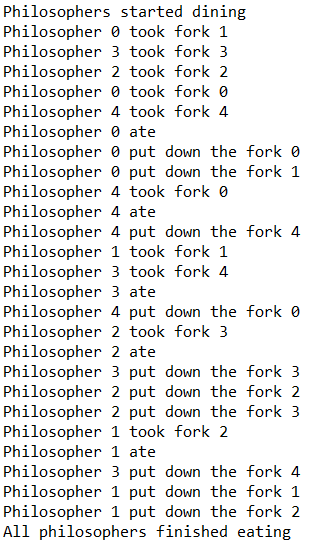
1. I implemented the “Dining philosophers problem” as follows:







The output for the previous code is:



As it can be observed, I used for the implementation the variant 3 possible solutions for this problem. I used a variable **type** with only 2 values (0 and 1) to specify if the philosopher first takes the left fork, then the right one, or the other way around.

I set only the first Philosopher to pick first the right fork, then the left one, by setting its **type** to 0. All the subsequent philosophers will have 1 to the variable **type.** In this way, I will assure that always at least one philosopher can get both forks to eat, solving the situation when there could be all the philosophers stuck with only the left or right fork.

There is no need in verifying in the **append** method if the fork **v** is in use. There is only one fork with the value **v**, and a philosopher was using it is trying to put it down on the table, therefore the fork is not on the table when calling the method **append**.

The **take** method will wait until the fork **v** will be available to be taken by the philosopher who called this method, after that the fork **v** will be set as unavailable.

The advantage of this implementation is that it can solve the problem of the dining philosophers for any number of philosophers, simply by changing the value for the static variable **NPTHREADS**.