SCC 211: Programming Coursework on Concurrency

Aim

The aim of this exercise to make students familiar with basic concepts in concurrency, such as threads, critical sections, and race conditions. As a practical matter, the exercise will be done in Java and will make students familiar with the use of Java threads and the synchronization mechanism designated by the *synchronized* keyword in Java. The use of any other synchronization mechanism is forbidden in this exercise.

Submission

1. Each submission must be a zip file that contains all the source code (.java) files. There must be no subfolders in the zip file.
2. Each submission must contain a Readme.txt with the student's full name and student ID number.
3. The Java main method by which the application is launched must be in a file named *InventoryMain.java*.
4. Test that the following works:
    a. Extracting the contents of the zip file into some folder and running *javac InventoryMain* from the command line in that folder compiles the application. This should generate the .class files in the same folder.
    b. Running *java InventoryMain* with the appropriate arguments from the command line runs the application as desired. Your program must take three arguments as described below in the program specification, even if you don't use all of them (that will only be because you haven't implemented the program completely).

You must submit the coursework on Moodle by the submission deadline – it is not optional! If, for some reason, the in-lab evaluation cannot take place, the marks will be determined based solely on the submission.

Program Specification

Imagine a warehouse and some number of items in its inventory. We care only about the size of the inventory, not the nature of the items. In particular, we care about maintaining the size of the inventory correctly despite concurrent operations on it. The operations specifically are (1) **adding a single item** to the inventory, and (2) **removing a single item** from the inventory.

Your task is to implement a multithreaded Java program that supports concurrent addition and removal of items. Assume that the initial inventory size is 0.

Your program must take three command-line arguments.

1. The first argument is the number of **add** operations. For example, giving 50 as the first argument means 50 add operations must be performed, each in a separate thread. That is, your program must launch 50 threads for doing the 50 add operations.

2. The second argument is the number of **remove** operations. For example, giving 20 as the second argument means 20 remove operations must be performed, each in a separate thread. That is, your program must launch 20 threads for doing the 20 remove operations
3. The third argument is a *bug* flag: either 0 or 1.
    a. **0 means that the program must end with the correct inventory size** no matter how many concurrent add and remove operations are specified. It should be obvious that after *n* add operations and *m* remove operations, the inventory size must be n-m (Yes, inventory size can be negative!).
    b. 1 means that your program must end with an incorrect inventory size after the operations are performed. (There is always a chance that the program will still produce an incorrect inventory size because of nondeterminism, but write your code so that it is highly unlikely that that will happen.)

Hint: You will get the correct result (inventory size) if you use *synchronized* properly. To get an incorrect result, it may not be enough to fail to use synchronized appropriately. You may have to do more to produce problematic interleavings of the threads.

Desired Output:

Let's say you ran *java InventoryMain 5 10 0*. This means your program will do 5 add and 10 remove operations. The bug flag is 0, which means your program must produce the correct result (-5) after the operations.

Your program output must be printed to the console. It consists of a sequence of statements one for each operation. Each statement prints the operation performed and the inventory size resulting from it.

UPDATE: In addition, you must print the final inventory size after all the add and remove threads have finished.

```
Figure 1.
Added. Inventory size = 1
Removed. Inventory size = 0
Removed. Inventory size = -1
Removed. Inventory size = -2
Removed. Inventory size = -3
Removed. Inventory size = -4
Removed. Inventory size = -5
Removed. Inventory size = -6
Removed. Inventory size = -7
Removed. Inventory size = -8
Removed. Inventory size = -9
Added. Inventory size = -8
Added. Inventory size = -7
Added. Inventory size = -6
Added. Inventory size = -5
Final inventory size = -5
```

Figure 1 shows a possible correct program output for the givens inputs.

Hint: Every time you perform an operation, atomically print the corresponding statement to the console.

UPDATE The thread that prints the final inventory size must wait for the other threads to finish. Look up join() on the Thread class.

```
Figure 2.
Added. Inventory size = 1
Removed. Inventory size = 0
Removed. Inventory size = -1
Removed. Inventory size = -2
Removed. Inventory size = -3
Added. Inventory size = 1
Removed. Inventory size = 0
Removed. Inventory size = -1
Added. Inventory size = 1
Added. Inventory size = 1
Removed. Inventory size = 1
Removed. Inventory size = 0
Removed. Inventory size = -1
Removed. Inventory size = -2
Added. Inventory size = 1
Final Inventory size = 1
```

Let's say you ran *java InventoryMain 5 10 1*. Now the bug flag is set to 1, so your program should produce an incorrect result. Figure 2 shows such a program output.

UPDATE: Again, you must print the final inventory size. You can clearly tell the result is incorrect because the final inventory size printed is 1 instead of -5.

Hint: Make sure your program handles the edge cases. For example, running *java InventoryMain 0 2 1* (with no add operations and two remove operations) should produce an incorrect result. As long as there are at least two operations (whether all adds, all removes, or some combination), your program should produce an incorrect inventory size when bug flag is set to 1.

Evaluation

In the lab sessions in Weeks 4 and 5. Be ready to run your code with the arguments you are given. Be prepared to show and explain your code. You get full marks if we are fully satisfied with your outputs, code, and explanations.

The coursework is worth 20 points.

Up to 10 marks: For being able to launch the threads corresponding to the number of add and remove operations as specified on the argument line.

Up to 5 marks: For getting the desired output when bug flag is set to 0.

Up to 5 Marks: For getting the desired output when bug flag is set to 1

Suggested Steps for Completing the Program

Week 1: Implement a multithreaded program. Take a command line argument for the number of threads to launch. Let the threads increment a shared counter. Here are a couple of links to get you started. You can also refer to the first lecture's slides on Moodle.

https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

https://www.tutorialspoint.com/java/java_multithreading.htm

Extend the idea so that instead of a shared counter you are maintaining a shared inventory size (initially 0) and treat the command line argument as the number of add operations. I suggest using a Warehouse object with a variable inventorySize and methods to add.

Week 2: Add another command line argument for the number of remove operations. Now you should be able to launch the specified number of add and remove operation threads that modify the inventory size appropriately.

Then use synchronized in the appropriate places so that there are no bugs from race conditions. Make sure when you add or remove from inventory, you atomically print to console (using System.out.println) the operation name and the resulting inventory size (as described earlier).

Week 3: Take a third argument, which is the bug flag. If it is 0, then your code should execute with synchronization (as you have implemented so far); if it is 1, then your code should execute without synchronization and manifest a race condition. Hint: Use synchronized blocks, which gives you the flexibility of doing things depending on the flag.