Systems Programming and Concurrency

Lab nº4. Threads in Java

1. Develop a Java program with 3 threads (in addition to the main one). Each thread displays **t** times the character **c**, where both **t** and **c** are parameters of the thread constructor. Are the letters displayed mixed? Comment about the observed behavior.

   b) Once developed the previous program, you have to modify it to display the letters A, B and C following the pattern ABBCCC. Therefore, the only valid sequence to be displayed is ABBCCCABBCCCABBCCC…

   To do this, some synchronization is needed. It's recommended using a shared variable to assign the turns to each thread over time, so a thread displays its letter when it is its turn. With this approach, when a thread is created it receives a constant identifier as parameter in order to compare it against the current value of the shared variable; the thread have to wait while it is not its turn (e.g., `while (turn!=myId){yield();};`).

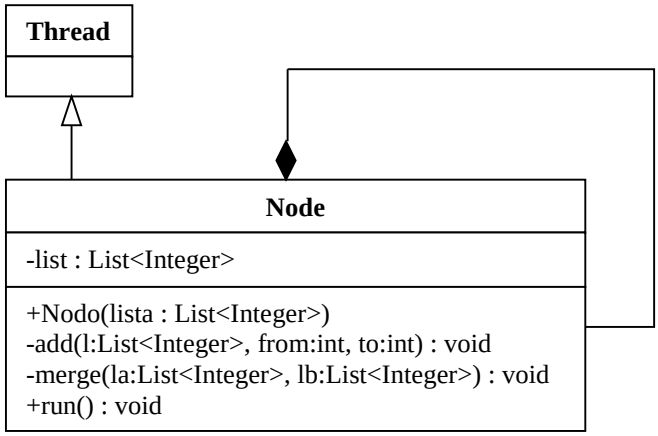
2. Let's say we have a class named `SharedVariable` which encapsulates the `int` value of a variable `v`. This class provides methods to `set` the value of `v`, to `get` its value and to increment (`inc`) it.
   Develop a Java program with two threads that share an instance of the class `SharedVariable`. Each thread has to increment ten times the value of `v`. After both threads had finished, show the value of `v` in the main thread: has `v` the value you expected? Raise progressively the number of times a thread increments `v` until you obtain an unexpected result. Explain the observed behavior.

3. Reuse the previous class `SharedVariable` and create a shared object of it. Develop a Java program that, within a thread, modifies the value of `v` from 0 to 99 by using the method `set`. With a different thread try to display all the changes over `v` by using the method `get` (if you need, this thread may know that `v` will be modified 100 times). Does this second thread display all the values? What happens exactly? Try to solve the problems detected.

4. Develop a Java program to calculate the items of a Fibonacci sequence. To compute the N-th term of the sequence you have to create N threads, so the thread with the i-th identifier is in charge of calculating the i-th term of the sequence starting from the results produced by the threads i-1th and i-2th.

Develop a solution using busy waiting: each thread has to wait for the previous items of the sequence to be calculated. The 0-th and 1-st items of the sequence must be considered as base cases.

5. Develop a Java program to implement a concurrent version of the merge sort algorithm. The program must create a binary tree of threads whose depth depends on the number of elements to sort. This project requires a class `Node` with the next structure:



To start with, the `main` method creates a root `Node` with the list of integers to sort as a parameter. Every `Node` created in the system has the same behavior, i.e. i) if the list contains a single element (or no element) then it does nothing (the list is already sorted); ii) otherwise, the thread creates two new children nodes passing as parameters the left and right halves of its own list. Once each child has sorted its half, the parent node merges both halves to sort its whole list.
As may be seen in the diagram, the `run` method is supported by two auxiliary methods: a) `add(l,f,t)` which adds to `l` a slice of `list` which starts in position `f` (included) and ends in position `t` (excluded); and b) `merge(la, lb)` which merges the ordered lists `la` and `lb` into the internal `list` variable.

Hint: Use the interface `List<Integer>` and the class `ArrayList<Integer>` to manage the lists in this exercise. This allows you using the built-in methods `add`, `addAll` and `subList` provided by these classes.