COSI 131b - Spring 2020
Programming Assignment 2
Due February 14, 2020, 11:59 on LATTE
Recommended Tutorial: Wed February 5th

In this assignment, you will continue working on your Unix-ish and turn it into a multithreaded program. You will be given a completed solution of PA 1, regardless of whether you finished it or not. The starter code you will need is available on LATTE.

To complete this assignment, you need to have a clear understanding of how threads work, and how threads can be started and waited upon. **If you have any ambiguity about these concepts, please attend the tutorial for PA2, read about threads on the lecture slides/notes or on online resources, before attempting to complete this assignment.**

You will leverage your new knowledge about threads in Java to make your REPL loop's components (its filters) from PA1 execute in separate threads, thus allowing the commands to run concurrently. Additionally, you will allow the commands to be executed as background processes.

Part 1: Concurrent Filters

Starter Kit. For this part you will need to change the general structure of the starter code we gave you very little; In our code, all filters are placed in the `concurrent` package (instead of the `sequential` one as in PA1) and they all have been transformed into their concurrent counterparts by extending the `ConcurrentFilter` class (instead of the `SequentialFilter` class in PA1). The `SequentialCommandBuilder` and `SequentialREPL` classes have also been replaced by their concurrent counterparts. **To complete this assignment, <u>you do not need to</u> modify any filter class nor the** `ConcurrentCommandBuilder` **class. You only need to modify the** `ConcurrentFilter` and `ConcurrentREPL` **classes .**

You task. The starter kit code does not allow for concurrent execution of the filters as none of the filters can be executed as a separate thread. Your task is to allow for (1) filters to be started as threads and (2) allow them to run **concurrently.** *Remember that special care (i.e., code) is needed for multiple threads to run concurrently and not sequentially.* **Just because two objects implement the Runnable interface (and hence represent two threads) does not mean that they will execute ALWAYS in parallel. Your code determines whether they will execute sequentially or concurrently.**

<u>Implementation Hints</u>. Some of the changes and considerations you will need to make are given below. This is not a complete list, but should comprise enough to get you started, and thinking about the right pieces of code to create and change.

- `ConcurrentFilter` class should implement the `Runnable` Interface. When and where will you call the `run()` method?
- You will need to change the mailboxes for the filters to a data structure that supports concurrent operations The `LinkedBlockingQueue` class is such a structure. Read about it! Only proceed when you understand what will happen if a thread tries to read from a blocking queue when it is empty.
- You will have to wait for all (or the last) filter(s) to complete before printing out the " >" prompt.
- Your REPL loop will have to create all filters before starting any of them (why?)
- As we said above: creating threads != concurrency. How can we have sequential code using threads? How can we turn into concurrently executing code?
- Your `isDone()` method need a totally different approach now (why?)
- What happens if the threads process at totally different rates (1 : 10000x slower?)

Your resulting code should behave just like the sequential version that you have completed for PA 1, and the end user should have no idea that there is a difference- so your concurrently executing commands should be transparent to the user. Since we already graded you for this functionality, what will be grading you in this portion of the assignment is how your code demonstrates your understanding of multi-threading techniques, and how you go about accomplishing this task. There is a "correct" way to do it, and you will be graded upon how your understanding of threads allows you to approach this "correct" ideal. Thus, it is HIGHLY recommended that your code is clear, well named, well commented, and immediately clear to someone who will determine your grade in the ten minutes they have to read over it.

Part 2: Background Commands

After you have completed the above task, the final portion of the assignment is to support a simple form of background processes. The following page can help you understand what background processes are. After reading through this resource, it is highly recommended that you play around with background processes on a UNIX console.

What you need to know	link
Running background processes, Listing background processes	http://www.ee.surrey.ac.uk/Teaching/Unix/unix5.html

Your task is to allow the end user to run and list the background commands, and to kill any one of the running background commands. You do not have to provide functionality that allows to suspend a command, or to bring a command from background to foreground.

In this version of your program, whenever an end user types the “&” ampersand at the end of a command the REPL loop does not wait for the entire command to complete but it directly prints out the “>” prompt to accept new command while the previous ones might still be running. This allows your Unix-ish system to execute commands in parallel. This is useful in cases where you

have to execute long running commands and you don't want to wait for the previous commands to finish in order to start the new ones. For example, suppose you have a big file of Amazon reviews and you would like to create a file with the reviews that contain the phrase "amazing product" and a second file with the reviews that contain the phrase "defective product". You can accomplish that by executing the following commands:

```
> head -1000000 Amazon-reviews.txt | grep amazing > amazing.txt &
> head -1000000 Amazon-reviews.txt | grep defective > defective.txt &
>
```

You know that these two commands will take a while to run and so you can have them running in the background while you are still able to execute other commands.

Furthermore, the user should be able to monitor what commands are still running. This could help to avoid exiting the Unix-ish system while there are still commands running. To do this you need to create a new command "repl_jobs" which checks which of the background processes are still alive and prints a list of the alive processes. For example:

```
> head -1000000 Amazon-reviews.txt | grep amazing > amazing.txt &
> head -1000000 Amazon-reviews.txt | grep defective > defective.txt &
> repl_jobs
    1. head -1000000 Amazon-reviews.txt | grep amazing > amazing.txt &
    2. head -1000000 Amazon-reviews.txt | grep defective > defective.txt &
>
```


The user should also be able to kill a specified command by providing the number that "repl_jobs" assigned to it. For example, to kill the second command, user would do:


```
> kill 2
```

You are not allowed to use any deprecated methods when implementing the "kill" command. Commands "repl_jobs" and "kill" have to be part of your REPL and not a new concurrent filter. They can only run as single commands, cannot be part of any command that involves pipes and do not require any piped input and piped output. Command "repl_jobs" does not require any arguments.

UnitTests: For this assignment, your Starter Kit includes also a new set of JUnit tests. You are allowed to use either the solution we provide to you or your own solution but make sure you use the new set of JUnit tests. **Please make sure to include Java Docs for your code as it counts towards your grade.**

Please submit your code by creating a zip archive (following the eclipse export instructions on Latte) and upload it on LATTE.