COSI 131 Programming Assignment 1, Part Two
Due October 7th, 2016

As most medical advice goes, there is some good news and some bad news. The bad news is that your work with the Unix-ish system is not yet complete. The good news is that you are very close to completing the second part, and you will be given a completed version of Part 1, regardless of whether you finished it. From this completed version of Part 1, you will not have to modify too much code to complete this part of the project, but you will not be successful without a clear mental picture of how threading works, and how threads can be started and waited upon.

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

Part 1: Concurrent Filters

You will need to change the general structure of your program very little; however, your Sequential classes will all have to be transformed into Concurrent counterparts (to be placed in the concurrent package instead of the sequential package). 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. **The description assumes that you understand how threads operate**. **If you have any ambiguity about these concepts, please read about it online, in the textbook, or from the lecture slides/notes before attempting to complete this assignment.** These reminders/ideas/questions should get you set up to do this portion of the assignment.

- ConcurrentFilters should implement the Runnable Interface. When will you call run()?
- You will need to change the mailboxes for the filters from LinkedLists 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?)
- A reminder: creating threads != concurrency. How can we have sequential code using threads?
- Your isDone() methods 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 part 1, and the end user should have no idea that there is a difference. 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 clean, 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 you don't have to provide functionality that allows to suspend a command, to bring a command from the background to foreground and to kill a command.

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 "bad 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 &
>
```

Command "repl_jobs" has to be part of your REPL and not a new concurrent filter. It can only run as a single command and it cannot be part of any command that involves pipes. It does not require any arguments, piped input and piped output.

For this part, you are provided a Starter Kit that includes a solution of part one and 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 submit your code by creating an archive (using zip, tar, or gzip, but not rar) and upload it on LATTE.