CS 313 Project 1

Linked lists are usually viewed as storing a sequence of items in a linear order, from first to last. In many applications, data is more naturally viewed as having a cyclic order.
For example, an operating system must manage many processes that might be running simultaneously on a computer (typically a computer has only one to four processors).

To do this, most operating systems allow processes to "take turns" using a CPU. A process is given a time slice to execute and then interrupted for the next process to run, even if it is not yet finished. This is called **round-robin scheduling.**

Read more about it in Section 3.3.1 of the text.

Note how you can use both *SinglyLinkedList* and *CircularLinkedList* to implement round-robin scheduling. CircularLinkedList is just considerably more efficient.

The textbook in section 3.3.2 provides an implementation of a ***CircularLinkedList***. You should use this implementation for this project. The code is provided to you, so you do not have to rewrite it, only thoroughly understand it. You are also provided the code for ***SinglyLinkedList***.

For this project, we will try to simulate an operating system running many different processes via round-robin scheduling by using a ***CircularLinkedList.***

Actual Java Threads are a somewhat complicated concept that is beyond the scope of this class. You will encounter and learn much more about them in a class such as Distributed Systems.

For the purposes of this project, you are provided a *Process* class to emulate the behavior of a thread. It has two important methods: `run(int)` to make the Process "run" for a specified number of iterations, and `isFinished()`, which is a flag to let the calling method know that its work is complete.

A class *PrimeFinder*, also provided, extends *Process*. This class finds all prime numbers in a range and prints them to console within its run method. It also announces to the console when it has finished its task (for clarity). Put these files (*SinglyLinkedList.java, CircularLinkedList.java*, *Process.java*, and *PrimeFinder.java*) in the same directory as your solution.

Your tasks:

1. All your code should be placed in a file called Solution.java. Your work can be done in the main method or in other methods within Solution.java and called by the main method (which you must provide). Your code must compile to receive a grade.
2. Create ten *PrimeFinder* objects, named "Process 1" to "Process 10".
3. First *PrimeFinder* instance should be constructed to find primes between 0 and 1000.
4. Second *PrimeFinder* should be constructed to find primes between 1001 and 2000, etc.
5. Tenth *PrimeFinder* should find primes between 9001 and 10000;
6. Add all of these to an instance of Circular*LinkedList* such that the first element has name "Process 1" and the last element has name "Process 10" (hint: there is two good ways to do this)

7. The above can all be done in a single for-loop if you are clever, but this is not required. Try it!
8. Next, use the CircularLinkedList instance to "run" the *PrimeFinder*s, as per the text 3.3.1. Each *Process* should be run for an interval 10-100 (Hint: pass this as a parameter to its run method), before the CircularLinkedList is rotated and the next process is run for the same duration. This process is continued until all the threads are finished (Hint: use the `isFinished()` method to find out if a *PrimeFinder* has finished its task). ***(8 pts.)***
9. Create a text or MS Word file named Solution.txt (or Solution.docx, respectively)
10. In this text or Word file, briefly explain how you would alter the code in your solution to use SinglyLinkedList instead of CircularLinkedList ***(1 pt.)***
11. In another paragraph, briefly explain why using CircularLinkedList is better than SinglyLinkedList for this scenario ***(1 pt.)***
12. Send Solution.java and your short answer file (Solution.txt or Solution.docx) to my e-mail address by 11:59 pm February 25th, 2019. As a reminder, the e-mail address to use is: stanislav.ostrovskii@qc.cuny.edu. ***Please put "CS 313 Project 1 [FirstName] [LastName]" as the subject line to ensure your project is not lost.*** You do not need to send the files pre-written for you i.e. CircularLinkedList.java, Process.java, etc.

Extra Credit for the more creative, +1 pt to final grade:

Instead of using the *PrimeFinder* class, create your own subclass of *Process* to perform some work for a specified number of iterations, and do the instructions using your custom class instead of *PrimeFinder*.

Some ideas could be: finding perfect squares in a given range, finding Pythagorean triplets in a given range, etc. Make sure your class extends *Process* and can be set with a constructor to run a specified number of total iterations. Use *PrimeFinder* as a template; if done correctly you only have to change a few lines of code. If you are making your own *Process* class, you can use make the values in the range be something other than 1000 (so long as it's significantly greater than the run duration interval, of course). For example, Process 1 can have a range of 1-5000, Process 2 can have range 5001-10000 etc. You should still have ten such "threads", that are run in the specified manner.

If you are doing the extra credit make sure to send your custom *Process* subclass Java file (along with the other two files).

If you do not want to risk losing points, do the base assignment as per instructions and then provide this solution in a file called Solution2.java. This would mean turning in four files in total.

Sample run for the base assignment (this uses a run duration of 25):

```
Process 1 found a prime: 2
Process 1 found a prime: 3
Process 1 found a prime: 5
Process 1 found a prime: 7
```

```
Process 1 found a prime: 11
Process 1 found a prime: 13
Process 1 found a prime: 17
Process 1 found a prime: 19
Process 1 found a prime: 23
Process 2 found a prime: 1009
Process 2 found a prime: 1013
Process 2 found a prime: 1019
Process 2 found a prime: 1021
Process 3 found a prime: 2003
Process 3 found a prime: 2011
Process 3 found a prime: 2017
Process 4 found a prime: 3001
Process 4 found a prime: 3011
Process 4 found a prime: 3019
Process 4 found a prime: 3023
Process 5 found a prime: 4001
Process 5 found a prime: 4003
Process 5 found a prime: 4007
Process 5 found a prime: 4013
Process 5 found a prime: 4019
Process 5 found a prime: 4021
Process 6 found a prime: 5003
Process 6 found a prime: 5009
Process 6 found a prime: 5011
Process 6 found a prime: 5021
Process 6 found a prime: 5023
Process 7 found a prime: 6007
Process 7 found a prime: 6011
Process 8 found a prime: 7001
Process 8 found a prime: 7013
Process 8 found a prime: 7019
Process 9 found a prime: 8009
Process 9 found a prime: 8011
```

```
Process 9 found a prime: 8017
Process 10 found a prime: 9001
Process 10 found a prime: 9007
Process 10 found a prime: 9011
Process 10 found a prime: 9013
Process 1 found a prime: 29
Process 1 found a prime: 31
Process 1 found a prime: 37
Process 1 found a prime: 41
Process 1 found a prime: 43
Process 1 found a prime: 47
Process 2 found a prime: 1031
Process 2 found a prime: 1033
Process 2 found a prime: 1039
Process 2 found a prime: 1049
Process 3 found a prime: 2027
Process 3 found a prime: 2029
Process 3 found a prime: 2039
Process 4 found a prime: 3037
Process 4 found a prime: 3041
Process 4 found a prime: 3049
Process 5 found a prime: 4027
Process 5 found a prime: 4049
Process 6 found a prime: 5039
Process 7 found a prime: 6029
Process 7 found a prime: 6037
Process 7 found a prime: 6043
Process 7 found a prime: 6047
Process 8 found a prime: 7027
Process 8 found a prime: 7039
Process 8 found a prime: 7043
Process 9 found a prime: 8039
Process 10 found a prime: 9029
Process 10 found a prime: 9041
```

```
Process 10 found a prime: 9043
Process 10 found a prime: 9049
Process 1 found a prime: 53
... (rest of the primes up to 10000 are printed in similar fashion)…
```