# CS21120 Individual Assignment: Ambulance Service

Veronika Karsanova, vek1@aber.ac.uk

9th of May 2019

## Contents

## 1 Introduction

This documentation details the implementation of the Ambulance Simulator, outlined in the assignment bried[3] for the CS21120.

It consist of class descriptions, used data structures listings, big-O notation for some of the operations, analysis of given tasks and additional testing.

Majority of the requirements were fulfilled, including some basic flair attempts.

## 2 Task 1: Implementation of IJob

The implementation of IJob interface (class Job) was pretty straight-forward, but required review the assignment brief, JavaDoc included in the interface and a quick check with the provided JobTests class. Once that was done, I considered myself prepared to code the fairly basic methods. Methods and their short descriptions are provided below. Notice, that in the implementation of IJob interface there was no need to use any data structures.

**Constructor** – constructor for the class Job has arguments: *int id, int priority, int duration and int submitted*, which keeps track of the time the job was actually added to the Simulator. Default value is initialised at -1 and is used in the getTimeSinceSubmit(int now) method, which will be discussed later.

**getID() and getPriority()** – two getters, which return the value of the *id and priority* for the required job.

**tick()** – updates the Job. Essentially, decreases the duration of the Job by one once called. Used throughout testing (see section 5) and in the Simulator class.

**isDone()** – return true if the duration is less or equal 0: pointing out *less* just for safety.

**getTimeSinceSubmit(int now) throws RuntimeException** – is a getter with a twist to it. In order to make sure that no one calls that method before actually assigning the submitted time (aka updating the 'submitted' variable from its default value), we are throwing an error if value of *submitted* = -1. Otherwise, using *int now*, which represents the current tick running in the simulator, and simple substraction we are returning the correct result.

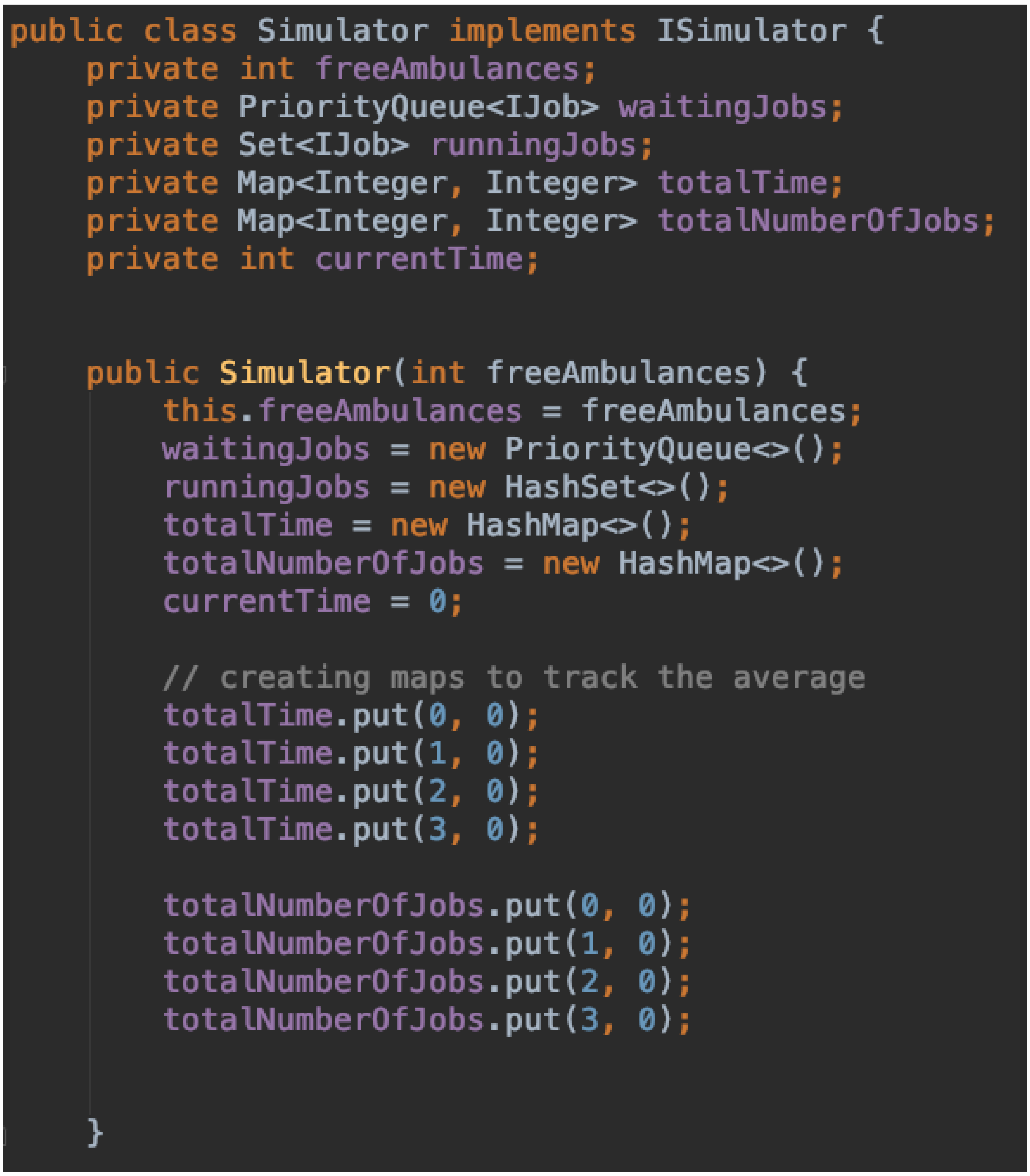
**setSubmitTime(int time)** – methos assignes *time* to the *submitted* variable.

**compareTo()** – method from the *Comparable* interface, which is implemented for the correct work of Priority queue, used in the implementation of section 3. Compares the priorities of Job, which is being passed in the argument and current Job. Returns 1 if current job has higher priority *number*, returns -1, if current job has lower priority number and returns 0 if priorities are equal.

Custom tests are discussed in section 5.

## 3 Task 2: Implementation of ISimulator

### 3.1 Data Structures



```java
public class Simulator implements ISimulator {
    private int freeAmbulances;
    private PriorityQueue<IJob> waitingJobs;
    private Set<IJob> runningJobs;
    private Map<Integer, Integer> totalTime;
    private Map<Integer, Integer> totalNumberOfJobs;
    private int currentTime;


    public Simulator(int freeAmbulances) {
        this.freeAmbulances = freeAmbulances;
        waitingJobs = new PriorityQueue<>();
        runningJobs = new HashSet<>();
        totalTime = new HashMap<>();
        totalNumberOfJobs = new HashMap<>();
        currentTime = 0;

        // creating maps to track the average
        totalTime.put(0, 0);
        totalTime.put(1, 0);
        totalTime.put(2, 0);
        totalTime.put(3, 0);

        totalNumberOfJobs.put(0, 0);
        totalNumberOfJobs.put(1, 0);
        totalNumberOfJobs.put(2, 0);
        totalNumberOfJobs.put(3, 0);


    }
```

Figure 1: The constructor that was being used for the initialisation of Simulator

**Queue, Priority Queue** – defines one of the main things, something that our simulator relies on: *waitingJobs*. I am using a priority queue data structure, which, essentially, works the same way queue does (also, resembles a stack) but initialises a priority per each element of the queue. In our case, priority queue is using the *Comparable¡¿* interface that implements overriden method *compareTo(IJob j)* in the Job interface. By having that method in the Job class, we letting priority queue know, what is supposed to be put first.

**Set, Hashset** – Using specifically Hashset rather than Treeset, because overall it has a better performance (complexity O(1)) and, since we are not required to sort anything, it is considered a preffered choice. We are using Hashset twice throughout our program: first time, for storing *runningJobs*, second time to store *runningJobsDs*. In both cases, we can find beneficial the fact, that sets store unique values. Meaning, that if, for example, something will all of the sudden break, and the jobs will be added to the runningJobs set multiple times – we well still be left with unique jobs running, no duplicates.

**Map, Hashmap** – With this one I still have my doubts. Treemap, on one side, is better, because it is not allowing nulls, but we do not care about entries ever being sorted. Are we prioritizing performance over memory consumption was another question. What dropped me off is a thought about a simulator in which the total job completion will be dealing with huge numbers. In the end though, I decided to go with a Hashmap, since for our current example performance was in the preferance, and it has a complexity of O(1) for all of the basic operations.[1]

**List, Arraylist** – used once throughout the Simulator in tick(), just to delete completed jobs from the Set¡IJob¿ runningJobs. We are storing the jobs that we need to remove in the ArrayList until the iteration completes, and then deleting them all together. Honetsly, I did not base my choice of data structure by performance in this case, it just seemed like something the most natural to use for the task purpose.

## 3.2 Big-O complexity

Below, we are discussing big-O complexity for required operatons. Notice, that Custom tests are discussed in section 5.

**Adding jobs** – *O(log(n))*, since the method itself takes the same amount of time to run no matter the input – O(1), we care only about priority queue – O(log(n)).

**Updating jobs** – *O(n)*, since we are iterating through jobs using while loop.

**Converting them from waiting into running** – *O(n)*, since we just repeating the same actions multiple times.

## 4 Experemental findings

### 4.1 Task 3

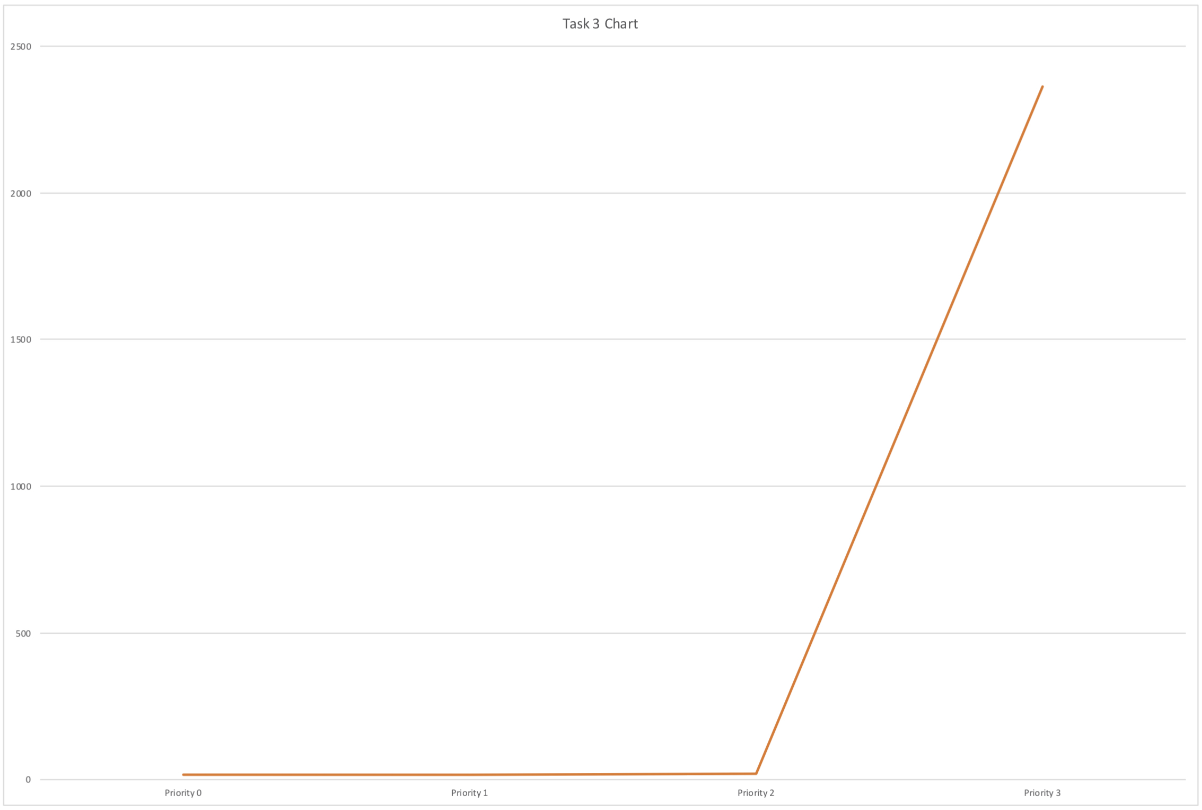


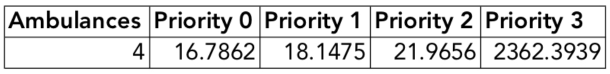
Figure 2: Change of average time completion per priority with 4 ambulances

The diagram records an average completion time per each priority level and represents in as a line graph. Notice how the difference rockets if the priority of the job is nominated to be 3.

I tend to believe that it is hapenning due to the fact that with 4 ambulances, if in every 1:3 chances new job gets created, then a lot of priority 3 jobs get postponed. They are just waiting in the Queue and, in the end, we see rerpesentation of not only the actual duration of the job but also of how long it's been waiting for to be taken care off.

Other priority levels do not share such a dramatic change in average time. Job of Priority 0 take up the least amount of time to be complete, which was expected, since they, essentially, "cut the line" and are being handled first.

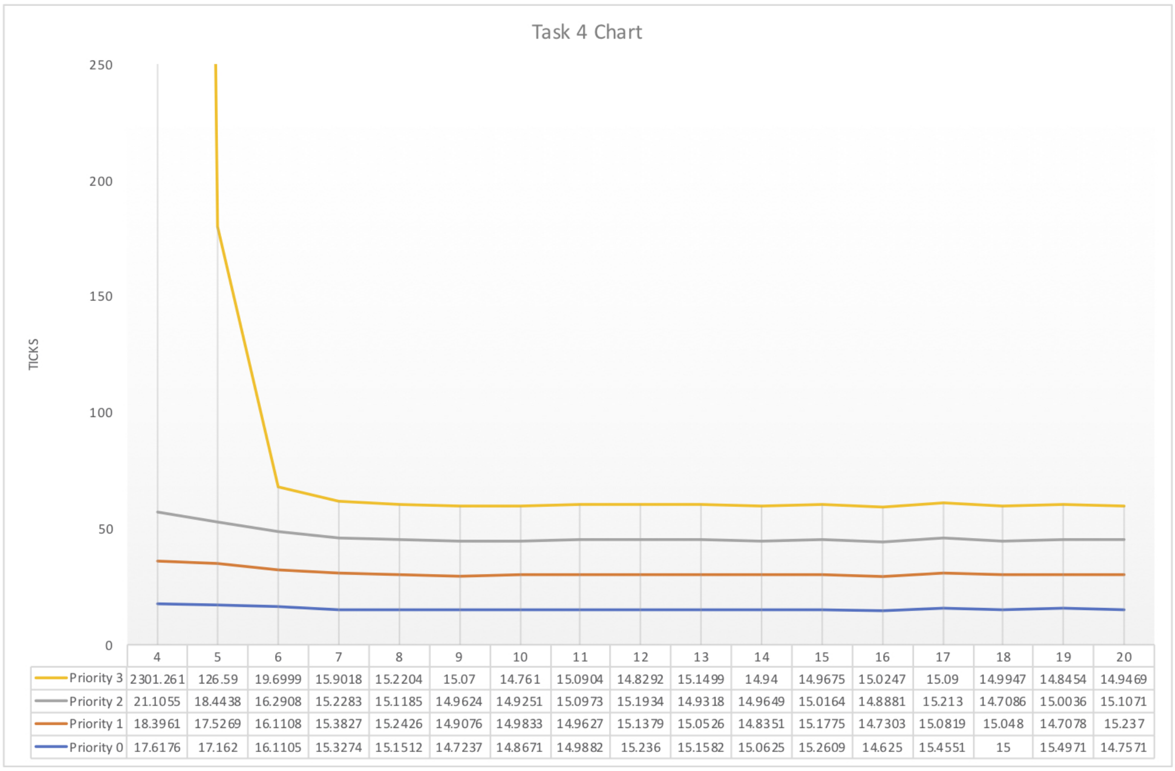
Priority 1 and Priority 2 have similar thing happenening accordingly.



| Ambulances | Priority 0 | Priority 1 | Priority 2 | Priority 3 |
|---|---|---|---|---|
| 4 | 16.7862 | 18.1475 | 21.9656 | 2362.3939 |

Figure 3: Data obtained from the task3() in its original form.

## 4.2 Task 4



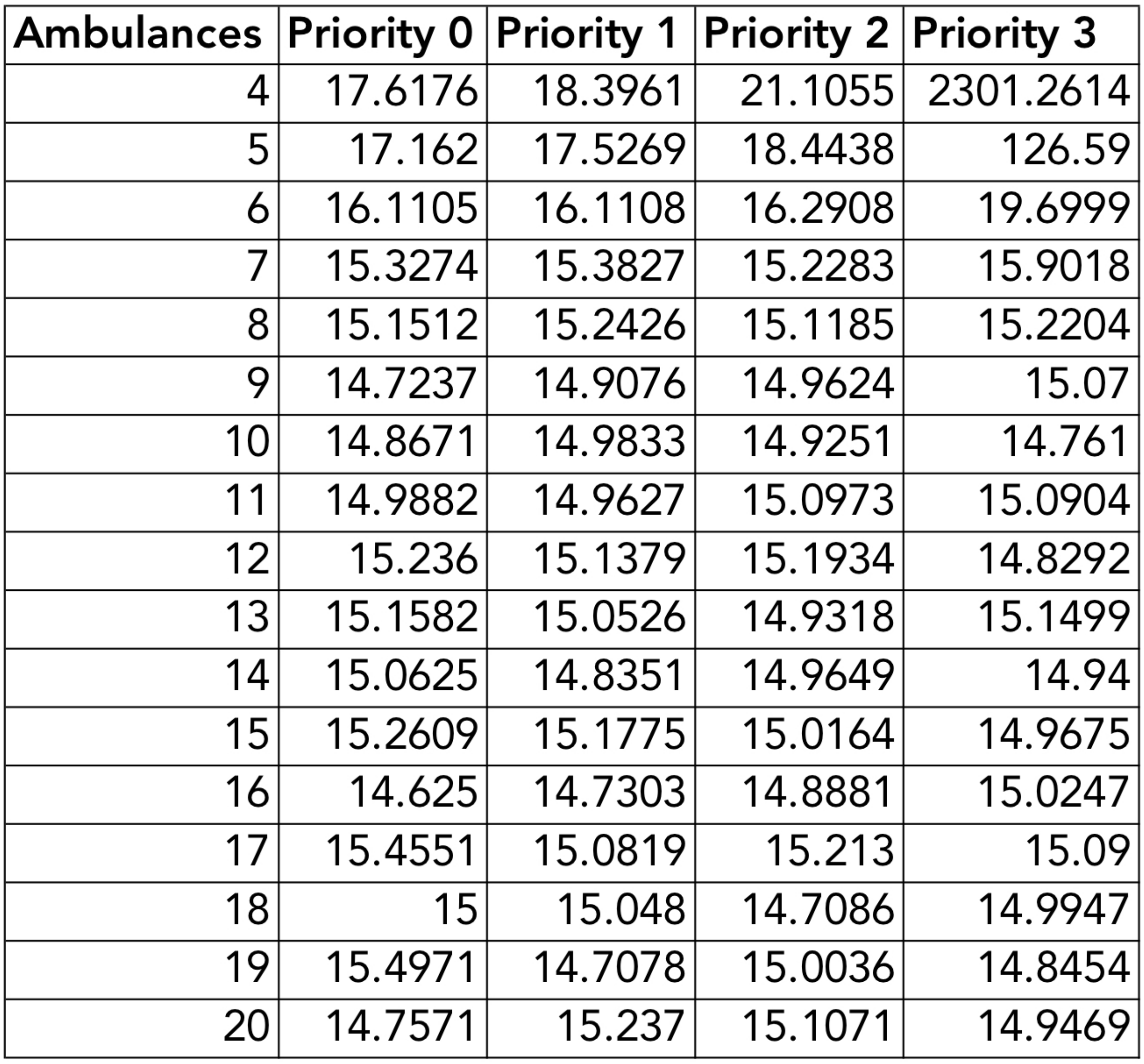
| | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Priority 3 | 2301.261 | 126.59 | 19.6999 | 15.9018 | 15.2204 | 15.07 | 14.761 | 15.0904 | 14.8292 | 15.1499 | 14.94 | 14.9675 | 15.0247 | 15.09 | 14.9947 | 14.8454 | 14.9469 |
| Priority 2 | 21.1055 | 18.4438 | 16.2908 | 15.2283 | 15.1185 | 14.9624 | 14.9251 | 15.0973 | 15.1934 | 14.9318 | 14.9649 | 15.0164 | 14.8881 | 15.213 | 14.7086 | 15.0036 | 15.1071 |
| Priority 1 | 18.3961 | 17.5269 | 16.1108 | 15.3827 | 15.2426 | 14.9076 | 14.9833 | 14.9627 | 15.1379 | 15.0526 | 14.8351 | 15.1775 | 14.7303 | 15.0819 | 15.048 | 14.7078 | 15.237 |
| Priority 0 | 17.6176 | 17.162 | 16.1105 | 15.3274 | 15.1512 | 14.7237 | 14.8671 | 14.9882 | 15.236 | 15.1582 | 15.0625 | 15.2609 | 14.625 | 15.4551 | 15 | 15.4971 | 14.7571 |

Figure 4: Change of average time completion per priority depending on ambulances

Interesting thing to observe. Obviously, at the beggining (number of ambualnces: 4), we get more or less the same results as we got in Task 3 (see subsection 4.2). Number of ticks for completion of priority 3 job is way longer than of any other level. But if we increase the number of ambulances even by one (!), we already getting a huge improvement. Priority 3 jobs still spend a lot of time waiting, increasing there completion time, but decrease from 2301 ticks to 126 ticks does worth mentioning.

Once the number of ambulances reaches 6, everything gets even better now we have the slightest difference in between Priorities 0, 1, 2 averages and Priority 3 average: 3 seconds. But it get even better. Once number of ambulances increases once again, times even out and they will remain +/- like so, despite us increasing the number of ambulances.

That allows us to conclude that the optimal number of ambulances for a Simulator that runs 10k ticks is 7.

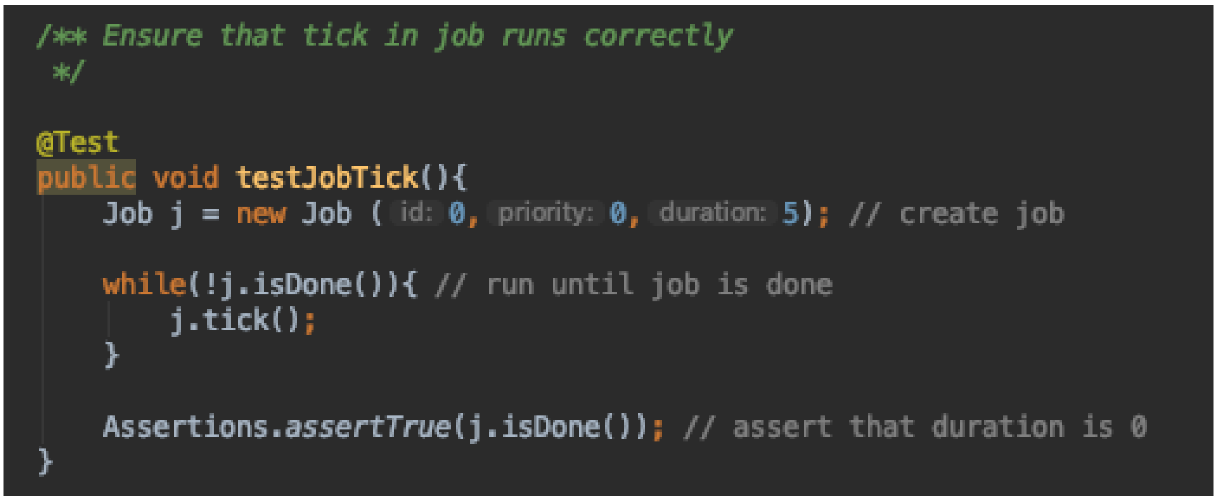


| Ambulances | Priority 0 | Priority 1 | Priority 2 | Priority 3 |
|---:|---:|---:|---:|---:|
| 4 | 17.6176 | 18.3961 | 21.1055 | 2301.2614 |
| 5 | 17.162 | 17.5269 | 18.4438 | 126.59 |
| 6 | 16.1105 | 16.1108 | 16.2908 | 19.6999 |
| 7 | 15.3274 | 15.3827 | 15.2283 | 15.9018 |
| 8 | 15.1512 | 15.2426 | 15.1185 | 15.2204 |
| 9 | 14.7237 | 14.9076 | 14.9624 | 15.07 |
| 10 | 14.8671 | 14.9833 | 14.9251 | 14.761 |
| 11 | 14.9882 | 14.9627 | 15.0973 | 15.0904 |
| 12 | 15.236 | 15.1379 | 15.1934 | 14.8292 |
| 13 | 15.1582 | 15.0526 | 14.9318 | 15.1499 |
| 14 | 15.0625 | 14.8351 | 14.9649 | 14.94 |
| 15 | 15.2609 | 15.1775 | 15.0164 | 14.9675 |
| 16 | 14.625 | 14.7303 | 14.8881 | 15.0247 |
| 17 | 15.4551 | 15.0819 | 15.213 | 15.09 |
| 18 | 15 | 15.048 | 14.7086 | 14.9947 |
| 19 | 15.4971 | 14.7078 | 15.0036 | 14.8454 |
| 20 | 14.7571 | 15.237 | 15.1071 | 14.9469 |

Figure 5: Data obtained from the task4() it its original form.

## 5 Testing strategy

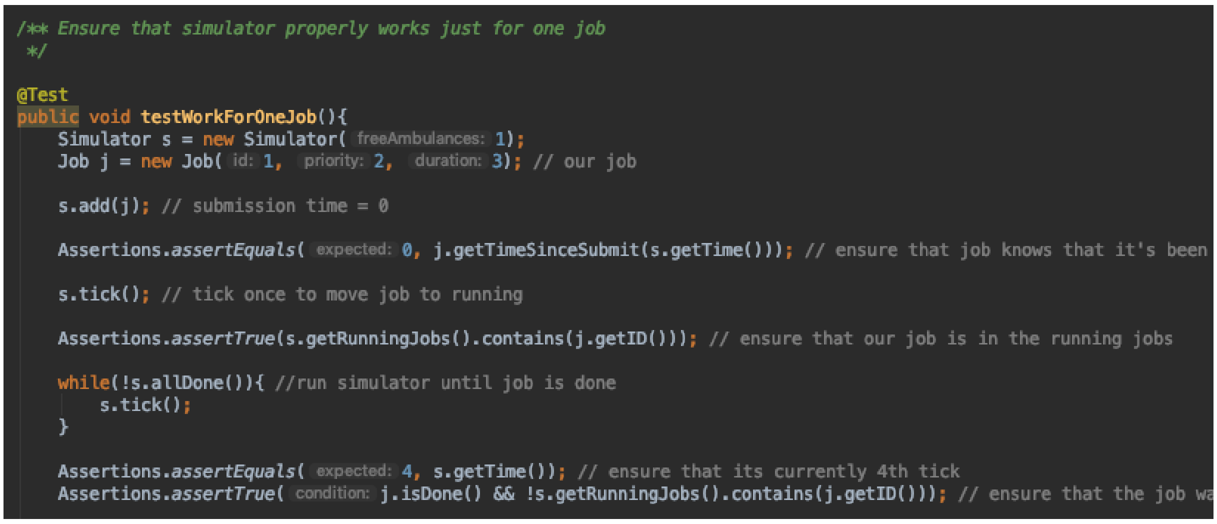
Apart from the tests that have been provided to us in the given 'src' directory, I implemented 4 tests on my own. All of them were successfully passed. You can find them in CustomTests.java and below:



```java
/** Ensure that tick in job runs correctly
 */

@Test
public void testJobTick(){
    Job j = new Job ( id: 0, priority: 0, duration: 5); // create job

    while(!j.isDone()){ // run until job is done
        j.tick();
    }

    Assertions.assertTrue(j.isDone()); // assert that duration is 0
}
```

Figure 6: public void testJobTick()

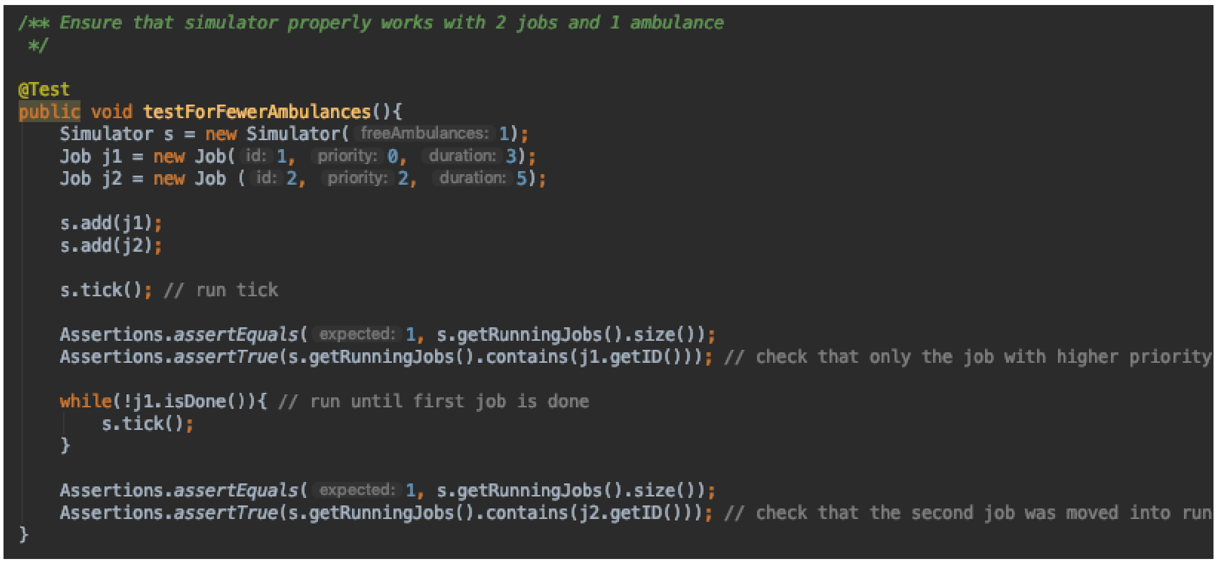
This test (see 6) is dedicated to check class *Job* and make sure that tick is running as intended. We updating the job in the while loop while it is not done and then ensuring that it is actually done (which mean it's duration is 0 or less as stated in section 2).



```java
/** Ensure that simulator properly works just for one job
 */

@Test
public void testWorkForOneJob(){
    Simulator s = new Simulator( freeAmbulances: 1);
    Job j = new Job( id: 1, priority: 2, duration: 3); // our job

    s.add(j); // submission time = 0

    Assertions.assertEquals( expected: 0, j.getTimeSinceSubmit(s.getTime())); // ensure that job knows that it's been

    s.tick(); // tick once to move job to running

    Assertions.assertTrue(s.getRunningJobs().contains(j.getID())); // ensure that our job is in the running jobs

    while(!s.allDone()){ //run simulator until job is done
        s.tick();
    }

    Assertions.assertEquals( expected: 4, s.getTime()); // ensure that its currently 4th tick
    Assertions.assertTrue( condition: j.isDone() && !s.getRunningJobs().contains(j.getID())); // ensure that the job wa
```

Figure 7: public void testWorkForOneJob()

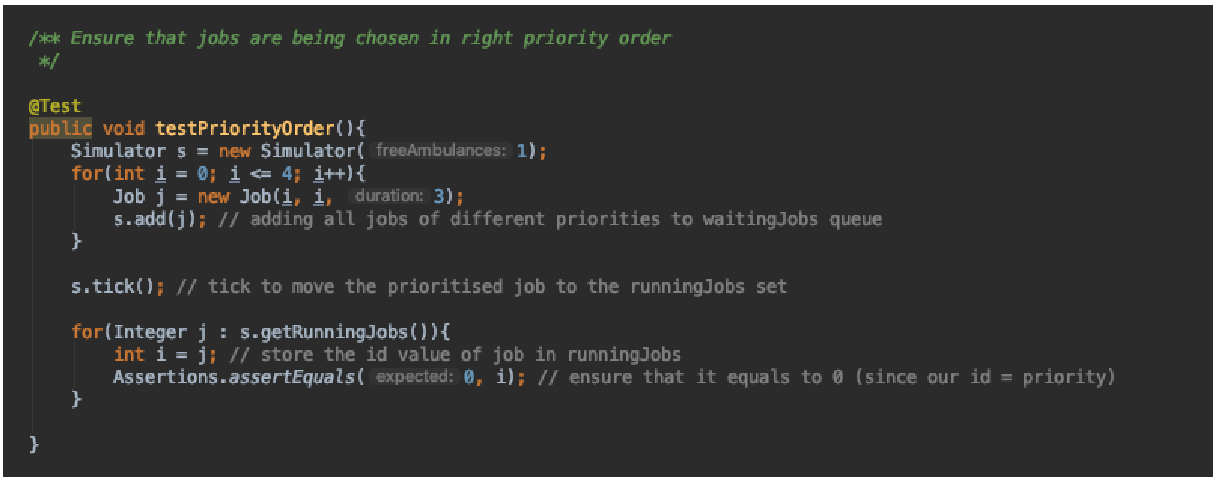
This test (see 7) is made for the Simulator class. What we are doing, is checking that the simulator is working as intended if only one job is being created. First, we checking if, after adding Job to the Simulator, it has been assigned the submitted value by calling getSinceSubmit(). Then we updating the simulator and cheching that job was moved to the *runningJobs*. After that, we updating simulator until the job is done, making sure that *currentTime* is authentic, job was done and is not anymore in runningJobs.



```java
/** Ensure that simulator properly works with 2 jobs and 1 ambulance
 */

@Test
public void testForFewerAmbulances(){
    Simulator s = new Simulator( freeAmbulances: 1);
    Job j1 = new Job( id: 1,  priority: 0,  duration: 3);
    Job j2 = new Job ( id: 2,  priority: 2,  duration: 5);

    s.add(j1);
    s.add(j2);

    s.tick(); // run tick

    Assertions.assertEquals( expected: 1, s.getRunningJobs().size());
    Assertions.assertTrue(s.getRunningJobs().contains(j1.getID())); // check that only the job with higher priority

    while(!j1.isDone()){ // run until first job is done
        s.tick();
    }

    Assertions.assertEquals( expected: 1, s.getRunningJobs().size());
    Assertions.assertTrue(s.getRunningJobs().contains(j2.getID())); // check that the second job was moved into run
}
```

Figure 8: public void testForFewerAmbulances()

This test (see 8) is used to check that the simulator properly works even when number of ambulances is less than number of jobs. After adding the created jobs to the Simulator and updating it, we making sure that only the Job with higher priority was added to runningJobs. Then we completing that job, and making sure that only after that the second job was added.



```java
/** Ensure that jobs are being chosen in right priority order
 */

@Test
public void testPriorityOrder(){
    Simulator s = new Simulator( freeAmbulances: 1);
    for(int i = 0; i <= 4; i++){
        Job j = new Job(i, i,  duration: 3);
        s.add(j); // adding all jobs of different priorities to waitingJobs queue
    }

    s.tick(); // tick to move the prioritised job to the runningJobs set

    for(Integer j : s.getRunningJobs()){
        int i = j; // store the id value of job in runningJobs
        Assertions.assertEquals( expected: 0, i); // ensure that it equals to 0 (since our id = priority)
    }

}
```

Figure 9: public void testPriorityOrder()

This test (see 9) is used to make sure that jobs of different priorities are executed in the right order. Essentially, performs similar actions to test 8, see comments in the code for more in depth explanation.

## 6 Self-evaluation

First of all, my overall impression of this assignment is rather good. I would not consider it the hardest, because I found myself struggling only with couple of bits that I will mention further, but at the same time it is challenging enough that you might actually require to double-check things with slides, debug or read Oracle materials on errors you are getting.

The creation of Job class (see section 2), however, was not the one that caused any difficulties. As discussed earlier in the report, a lot of methods required not more than 1 line of code and were obvious from the first glance.

Simulator class (see section 3), somehow, was a bit of a different story. Despite most of the class being, again, fairly easy and requiring anything between 1 to 4 lines of code, tick() method turned out to be a bit tricky. In this case, it was not necessarily the difficulty of the problem but the specifics of data structures I have been using throughout – for example, the Set taking a pause to remove element while continuing iteration. It was resulting in the collapse of the foreach loop and causing the ConcurrentModificationException [2]. But once I introduced the Iterator class for looping through the running Jobs and an ArrayList in which I have been storing the jobs I need to remove from the Set as a collection (using removeAll()) – the problem fixed itself, and, as seen in the CustomTests.java and Experiments (see section 4), did not bother us again.

Furthermore, once couple of custom tests was written, tasks 3 and 4 were inuitative to understand. My only concern was towards code duplication and whether or not we are allowed to change everything up a bit to avoid it, since I had my doubts. But after double checking the discussing forum [4], I just implemented an addition private method which I called from the task3() and task4(), and it worked perfectly fine.

Taking everything in the consideration, based on the version 1.2 of Assessment Criteria for Development, even though my work most certainly has room for improvement and is not exceptional in any way, I think that I managed to show a good understanding of the problem and was able to present it as such. Due to that,I believe that I have earned myself around 70 percent of the total mark.

## References

[1] Java TreeMap vs HashMap
`shorturl.at/mHMNZ`

[2] Avoiding the ConcurrentModificationException in Java
`shorturl.at/bsBHJ`

[3] CS21120 Assignment, 2019-2020 Brief
`shorturl.at/bloVY`

[4] Thread: task 3 / 4 in Forum: Programming Assignment Q&A
`shorturl.at/hnuQ6`