DCU School of Computing

Assignment Submission

Student Name(s):	**Shaun Kinsella, Shane Creedon**
Student Number(s):	**14740175, 15337356**
Programme:	**BSc in Computer Applications**
Project Title:	**Concurrency Assignment 1**
Module code:	**CA4006**
Lecturer:	**Dr Martin Crane, Dr Rob Brennan**
Project Due Date:	**22/03/2019**

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.
I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Shaun Kinsella, Shane Creedon

How to run

To run our code:
1. Ensure you are in the directory: `src/`
2. Run our bash script `*build.sh*` via `*bash build.sh*` or `*./build.sh*`

If this does not work, try compile/run manually with:
3. Compile all java files: `javac main/java/*.java`
4. Run with: `java main/java/Launcher`

View the output via the terminal logs, GUI and once closed, via the 'output.dat' file generated in the project source directory.

Optional:
If the GUI is too verbose, one can disable it using the `disable-gui` command line argument.
E.G. `java main/java/Launcher --disable-gui`

Architecture

- Launcher
- Airport
- Person
- Luggage
- Elevator

In our solution we utilise two **ExecutorService Thread Pools**, one a **fixedThreadPool** Executor for the elevator(s) and then a **ScheduledExecutorService** for all person objects. Both the Elevator and Person classes implement runnable, and all instances are created upon the program starting. Each person is created with an instance of the luggage class with its own random weight, the person's own weight, the floor they arrive on and the floor they will wish to travel to. Each property is randomised to some degree, providing a bit of dynamism among our threads.

They are also created with a randomised arrival time, which is passed to the **scheduledExecutorService** to simulate this. The idea of this all being done to start is to avoid the overhead of creating threads on the fly while maintaining a dynamic (chaotic) system.

The foundation of our approach is based around the usage of Concurrent Hashmaps. Specifically, we use a concurrent hashmap to map the floor number (Integer) to a LinkedBlockingQueue<Person> to provide each floor a unique list of requests, which retains the order of access. We believe it was wise to use this data structure as it made storing and retrieving requests trivial and also gave us the added benefit of having multiple concurrent threads which could read from and write to the hashmap without the chance of receiving out-of-date or corrupted data.

Locks & Conditions. Atomic Variables
In addition, Our solution makes use of **Reentrant Locks** and **Conditions** as proposed in the *java.util.concurrent* Package. Specifically, we make use of 3 Reentrant locks and conditions. The primary lock we use describes how person objects interact with the elevator system and provides a realistic interpretation of how people would act in the real world. We use this '*PersonLock*' to communicate with the elevator as to when the person should:

1. 'Sleep' while waiting' for the elevator.

2. Get on the elevator once it has arrived at their arrival floor.
3. ‘Sleep’ while waiting for the elevator to travel to the destination.
4. Get off the elevator once it has arrived at their destination floor.

These processes are dictated by the specific Reentrant Lock and Condition mentioned above. We utilise **Signalling** and **Awaiting** as offered via Conditions to do this. The elevator arriving at a requested floors will signal the ‘sleeping’ person(s) objects to awake and get on the elevator given they do not exceed the elevator weight capacity. The same happens once the ‘sleeping’ passengers arrive at their destination floors.

We also utilise an ‘ElevatorLock’ and ‘ElevatorCondition’ to put the elevator into a ‘sleep’ state once there are no sufficient requests on the elevator. This way, we save resources on the elevator thread and can utilise them elsewhere.

Finally, we use another lock purely for the purposes of ending the program gracefully. Once all of the generated passengers have respectively arrived at their destination floors, the Elevator class will signal back to the Airport class to ‘shutdown’ the elevator service. Once the Airport receives this signal, we shut down each executor service via .shutdown(). We do this for both the person executor service (Scheduled Pool) and the elevator executor service (Fixed Pool). This will end the program gracefully and give the desired result.

Multiple people on lift at same time.
To handle multiple people getting on and getting off the elevators, we synchronised between person objects and elevator objects via having an **Atomic Integer Variable** labeled **currentFloor** which kept the elevators current floor constant among each thread, while the comparison between each persons destination floor and the current floor took place. Each floor would be checked for any destinations or potential arrivals with respect to the current weight of the elevator. In the case of multiple people getting on/off, our system is very capable of handling these situations via the *Person Lock* synchronisation discussed previously.

When dealing with many locking situations, we would wrap our **.await()** methods within a while loop to refute any ‘Spurious Wakeups’ and also to ensure **mutual exclusion** among elevator usage. Among person threads, we conditioned the while loops using **Atomic Booleans** to ensure correct synchronisation among which threads can ‘get on’ or ‘get off’.

In relation to how passengers get off the elevator, we would iterate through them, giving them the lock individually, and checking atomically their destination floor against the elevators current floor. We would then call for them to remove themselves if this is the correct floor. Then regardless of whether they wished get off or not we would unlock the thread.

Fig 1.0 - Removal of passengers method - each passenger claims the lock before signalling back.

```
private void removePassengers() {
    for (Person person : currentPassengers) {
        personLock.lock();
        try {
            if (person.getDestFloor() == currentFloor.get()) {
                this.currentPassengers.remove(person);
                this.currentElevatorWeight -= person.getPassengerPlusLuggageWeight();
                this.amountOfPeople.decrementAndGet();

                person.getOffElevator();
                personCondition.signalAll();
            }
        } finally {
            personLock.unlock();
        }
    }
}
```

Logic/Fairness

In terms of how the elevator deals with requests, if an elevator begins on floor 1 and is called up to floor 6, it will still pick people up along the way, allowing people and luggage on as long as the weight of 400kg is not exceeded. This is true even if the other peoples destination floors are in the opposite direction of the lifts current travel. The lift will allow these people off on floors as it passes by their floors, but it will not deviate from the first request, ie it will not pass by 6 without stopping, or start to descend again. This prevents the original person thread from **starvation**, being blocked outside of the elevator thread, (or inside it unable to get off).

The elevator even while full will continue to stop at floors if there is a request for it to pick people up, as it has no concept of the person and their luggage weight until they step on. If someone wishes to get out on that floor they will attempt to before anyone gets on.

However it is not guaranteed that this will take place before someone attempts to get on, instead, the person thread will attempt to get back on after. If there are multiple people attempting to get on at the same floor it is not guaranteed that the person thread who first “pushed” the button will be the one to get on. (Simulation of real life rudeness that we have all dealt with).

If again that person fails to get on that floor and the lift leaves, the lift will again attempt to process their request once it has dealt with the other person threads on the lift OR when it passes by the floor in the process of dealing with their requests.

Problems we had:

- Elevator visiting correct floors but not letting people on
    - Caused by stale values in a boolean comparison between the necessary floor and the floor the elevator was currently on.
    - Solved by: **Atomic Booleans -** Does the two things as a single atomic step. Used for basic synchronisation.

What we would do next time:

Our Elevator class would retain its movement related methods, but we would introduce a new elevator controller class that would handle job/request dispatching, allowing scaling by introducing multiple elevators.