1 Introduction

In this project, you will get experience using threads, thread synchronization, and thread communication by writing a simulation of people traveling throughout a building using elevators. Each actor in the simulation is implemented with a thread, and the threads interact via synchronization and signalling. In the three sections below, we describe the Person class, which you will be given, and the basics of the Elevator and Building classes, which you must implement.

Please note: how you use synchronization, immutability, isolation, signalling (e.g. wait, notify, etc.) to implement these classes is entirely up to you. None of the methods on the signatures given will have synchronized specified on them. It is up to you to determine which ones should be synchronized, if any. The interaction between the Elevator and Building classes is also up to you, within the limits defined below. That is, you can add data and methods as you wish to implement the simulation. You should not modify the Person class.

2 Person

    public class Person extends Thread {
        public Person(String name, int[] itinerary,
                       int busyTimeSecs, int startingFloor,
                       Building building)
        public void run()
        public String toString()
    }

A person is an autonomous actor (it subclasses Thread). A person is created with a name, an itinerary, how much time (in seconds) to spend on each floor, a starting floor, and the Building in which it operates. The itinerary is an array
of floors that the person wishes to visit. Floors are numbered starting at 0, and should be less than Building.NUM_FLOORS.

The run() method iterates through the itinerary. For each floor f in the itinerary, the person will call for an elevator using building.callElevator(c), where c is the floor that the person is currently on. This function will return an Elevator e when it arrives on the person’s floor. The person then calls e.takeElevator(c). In the typical case, this method will block until the desired floor is reached, and then return that floor. The person then waits busyTimeSecs, and goes to the next item in the itinerary. When the itinerary is completed, the thread exits.

In a corner case, e.takeElevator(c) will return the person’s current floor c, rather than dest. Assuming that dest is not the current floor, this signals that the person could not get on the elevator because it has already left the floor, or because it was full. In particular, in between the time that b.callElevator(c) returned an available elevator e, and the time that this person tried to get on by calling e.takeElevator(dest, c), either the elevator left the floor, or one or more persons got on e and filled it up. In this case, the person needs to call another elevator.

The Person class also prints out events as they happen, like waiting for an elevator, boarding an elevator, etc. These generate a trace of the simulation.

The code for the Person class has been provided for you.

3 Building

```java
public class Building {
    public final int NUM_FLOORS;
    public final int NUM_ELEVATORS;

    public Building(int numFloors, int numElevators,
                    int elevatorCapacity, int serviceFloorMS,
                    boolean useSimple)
    {
        public void startElevators()
    
        public Elevator callElevator(int personFloor)
    }
}
```

The Building class describes the simulation environment that Person threads interact with. Every building has some number of floors, and some number of elevators; these values are passed to the constructor, and then used to initialize the public fields NUM_FLOORS and NUM_ELEVATORS, respectively. In addition, the constructor specifies the capacity of all of the elevators, the amount of time the elevator should spend servicing each floor (in milliseconds) and whether or not they should use the simple scheduling algorithm. These last three parameters are passed to the Elevator constructor; see below. When creating elevators, the building should name them “0,” “1,” “2,” etc. In addition, it should set their starting floor as n % numFloors, where n is the elevator number, 0 ≤ n < numElevators.
The `startElevators()` method will start `Elevator` threads, one for each elevator in the building. Elevator threads should be daemon threads, meaning that once the `Person` threads all terminate, the entire simulation should terminate.

The `callElevator` method is called by a `Person` thread to request an elevator, as specified above. It is given the floor that the person is currently on, and should not return until a non-full elevator reaches that floor. Once such an elevator arrives, it will be returned to the caller, who can then take the elevator.

You will probably need to design and implement a few more `Building` methods to work properly with your elevators.

## 4 Elevator

An elevator is an autonomous actor (implemented as a thread), that moves throughout the building, picking up and dropping off passengers.

```java
public class Elevator extends Thread {
    public Elevator(String name, int startingFloor,
                     int capacity, Building b,
                     boolean useSimple, long serviceTimeMS)

    public int takeElevator(int destFloor, int currFloor)
    public int takeElevator(int destFloor, int currFloor,
                             long waitTimeMS)
    public void run()
    public String toString()
}
```

An elevator is created with a name, its starting floor, its maximum capacity, the `Building` it is operating in, the algorithm it should use, and the time it should wait on each floor to let passengers on and off (the `service time`), specified in milliseconds. If the parameter `useSimple` is true, then it uses the simple algorithm, otherwise it uses a slightly more advanced one (these algorithms are described below). The `toString` method should simply return the name of the elevator.

The `void takeElevator(int destFloor, int currFloor)` method is called by a `Person` thread on floor `currFloor` who wants to get on the elevator and go to floor `destFloor`. The method checks that `currFloor` the same floor the elevator is on, and that the elevator is not full. If either condition is false, then the method returns with the `person`’s current floor (not the elevator’s). Otherwise, the method blocks until the elevator actually reaches the desired floor, at which point it returns that floor’s number (i.e. `destFloor`).

There is another version of `takeElevator` that takes an additional timeout parameter, specified in milliseconds. This specifies an upper bound on the amount of time a person is on the elevator. The idea is to allow a person thread to leave the elevator before it reaches his desired floor if the timeout has expired. In this case, the elevator will return whatever floor it is currently on (signifying
that the user got off at that floor). This will allow the person to wait for a
different elevator on that floor, that hopefully will reach his destination more
quickly! Of course, your mother always told you never to switch lines in the
grocery store ... To test this method, you can modify the provided Person class
to be impatient.

4.1 Scheduling

The run method implements the scheduling algorithms for the elevator. A
scheduling algorithm is an infinite loop that services the floor it is on, decides
which floor the elevator should go to next, and then goes to that floor. For this
project, you will implement two different scheduling algorithms, a simple one
and an “advanced” one, so run is simply an if-statement that uses the simple
algorithm when the user specifies useSimple as true in the constructor, or else
uses the advanced algorithm.

Whatever the algorithm, the elevator services a floor in the same way. First,
it must declare that it has arrived at the floor, by printing:

Elevator n arrived at floor c

Here, n is the name of the elevator, and c is the number of its current floor.
Next, it must inform any passengers currently in the elevator that it has arrived
at their floor. In addition, it must inform any persons that are waiting for an
elevator (that is, they called Building.callElevator on the current floor and
are blocked). Next, it sleeps for serviceTimeMS milliseconds, as specified in the
Elevator’s constructor. This gives up the processor, and allows the relevant
Person threads to wake up and either get on or get off the elevator. Finally, it
leaves the floor, printing:

Elevator n departed floor c

An Elevator should service every floor it reaches. Also, each floor should be
reached in order. That is, after servicing floor 1, it should next service floor 2,
or floor 0, depending on whether it goes up or down. Note that the idea of it
arriving at and leaving a floor is somewhat figurative. They really just define
when the elevator is allowing persons to enter (by calling takeElevator) or
leave (by returning from takeElevator) on that floor.

4.1.1 Simple Algorithm

The second part of scheduling is to determine which floor to go to next. For the
simple algorithm, an elevator’s starting direction is up, unless it is on the top
floor, in which it case the starting direction is down. The elevator will simply go
in its current direction until it reaches either the top or the bottom, and then
will switch directions, forever trolling up and down the building. It will service
each floor it reaches.
4.1.2 Advanced Algorithm

The advanced algorithm determines the next floor based on the persons it needs to drop off, and the floors at which a person has requested an elevator. The elevator can either be moving, or idle. When idle, the elevator has no passengers to drop off, and is not aware of any passengers requesting elevators on the floors of the building. Therefore, it does not move anywhere, but waits until either (a) a person gets on the elevator by calling `takeElevator`, or (b) a person requests an elevator on some floor of the building. If (a) occurs, or (a) and (b) occur, the elevator starts moving in the direction of its passenger’s desired floor. If just (b) occurs, it starts moving in the direction of the closest request (in case two came in at once). Note that a request could occur on the current floor, so the elevator will simply “arrive” there, rather than actually going anywhere.

When the elevator is moving in direction $d$ (either up or down), it does the following in a loop:

1. Service the current floor, as described above.
2. Determine whether the elevator should still be moving, and in what direction $d'$, where $d'$ is either up, down, here, or none. If the elevator is moving, go one floor in the desired direction $d'$ (or go nowhere if here). If the elevator is idle, then behave as described above.

The direction $d'$ is determined as follows:

1. Determine the direction $n$ of the next floor where we could drop off a passenger (either up, down, or none)\(^1\), preferring the current direction to the opposite direction. For example, if the elevator’s direction $d$ is up, and one passenger wants to be dropped off on a lower floor, and one passenger wants to be dropped of at a higher floor, $n$ will be up. However, if $d$ is up, and passengers have only requested lower floors, then the $n$ will be down. When there are no passengers, the direction is none.
2. Determine the direction $r$ of any floor with an outstanding request for an elevator to pick up a passenger (again, either up, down, or none), preferring here to either up or down, or else the current direction to the opposite one, as described above.
3. If either $n$ or $r$ is the current direction $d$, then $d' = d$. Otherwise, if $n$ is not none, then $d' = n$. Otherwise $d' = r$. If at the end of this, $d'$ is set to none, then elevator should be idle. In this case, you should print

\begin{verbatim}
Elevator n idle
\end{verbatim}

We suggest that you implement all that you need to for the simple algorithm first. It requires a lot less from the `Building` and `Elevator` classes than does the advanced one. Once you get this to work, you can implement the advanced algorithm.

\(^{1}\)It could not be here in this case, because we just serviced this floor.
5 Running the simulation

You can create a simulation to test your classes. In a separate class, call it Simulation, you need to create a Building. Then you create some Person threads. Then, you call Building's startElevators() method, and you would call start() on each of your Person threads. At this point all of the threads will start interacting, and printing an execution trace. When all of the Person threads have terminated, the simulation will end.

A sample Simulation class has been provided for you, along with a sample trace using the simple algorithm, and one for the advanced one.

6 Updates

November 11, 2003 (1pm): Fixed bullet in last update—removed the destFloor parameter from building.callElevator, not personFloor.

November 11, 2003:

- Removed method requestOutstanding from signature of the Building class; you don’t need to implement this—it pertained to our reference implementation.

- Corrected constructor for the Elevator class to take serviceTimeMS as a parameter. Fixed later description of elevator servicing that referred to timeToWaitSeconds to refer to this parameter instead.

- Clarified that for the simple algorithm, elevators always start by moving up. If they are on the top floor, then they will immediately switch directions and start moving down.

- Clarified description of Building constructor with regard to the creation of elevators, including how to name elevators, and what floor they should start on.

- Removed the destFloor parameter from the Building.callElevator method, since it is never used.