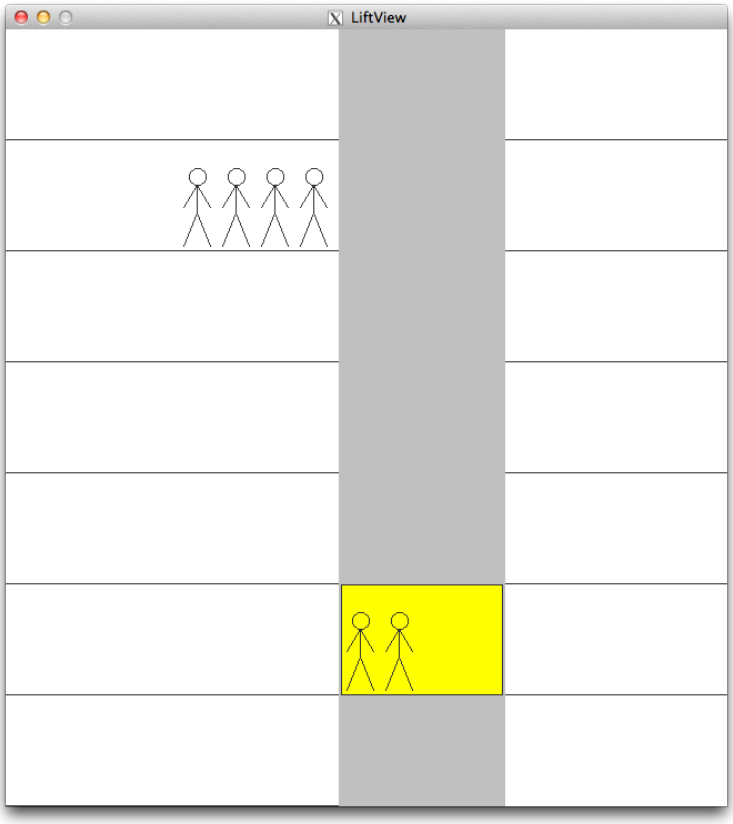
1 Lab 2 Overview



Read the entire document before starting!

In this lab you are going to implement a lift simulation using threads, monitors and wait/notify.

1.1 Objective

After this lab you will have implemented a simulation using the basic Java concurrency primitives. You should know the following terms:

- Monitor
- Wait/notify

and be able to

- Share data and application logic between threads using a monitor
- Use wait/notify to avoid race conditions
- Design and implement a small concurrent system from a written specification

1.2 Preparation (before the lab)

It is generally a good idea to start programming the exercise before the lab occasion. *Before* you start programming you need to have a design of the system. This is what you prepare at the exercise session before the lab. The design should answer the following questions:

- Which monitor attributes are accessed/modified by which threads?
- What condition must be true for a person should be allowed to enter the lift?
- What condition must be true for a person to leave the lift?
- What condition must be true for the lift to start moving to another floor?
- Which monitor operations are necessary in order to guarantee proper synchronization of the various threads? What, and how much, must be performed as a part of each monitor operation?

1.3 Getting started

The instruction assumes you are sitting at a linux school computer with a shell (terminal/console). It is preferable if you have created a home folder for the course, i.e. `mkdir eda040; cd eda040`. You need to copy the handout code to your home folder:

```
cp -r /usr/local/cs/rtp/lab/lift .
```

At home you can use sftp to get the files:

```
scp -r username@login.student.lth.se:/usr/local/cs/rtp/lab/lift .
```

Do not forget the end point (.). The handout code contains only the visualization class (*LiftView*), the rest you must implement yourself. The LiftView class contains a main method with a test case.

```
1   public static void main(String[] args) {
2       LiftView lv = new LiftView();
3       lv.drawLift(0,3);
4       lv.drawLevel(5,4);
5       try {
6           Thread.currentThread().sleep(1000);
7       } catch(InterruptedException e) { }
8       lv.moveLift(0,1);
9       lv.drawLift(1,2);
10  }
```

To check that everything works compile the LiftView class and run it.

```
javac LiftView.java
java LiftView
```

You are not supposed to modify this main method. You should write your own main in your own class.

1.4 Programming

1. Write and run a program according to your design.
2. When your program is working, make the following modification to it: Change it such that the lift is halted when no persons are either waiting for the lift to arrive or are located within the lift itself. The lift should thus not move unnecessary. The lift should be started again when a person arrives at a floor.
3. Show your program to your supervisor.
4. Possibly implement an optional extension to the simulation. Suggestions for further modifications to the lift simulation include:
    - The lift could be intelligent and change direction before reaching the top or bottom floors if possible.

- A person should avoid getting onboard the lift if it is headed in the wrong direction.
- Discuss the problems these modification causes. What strategies should the person and lift threads utilise? What new shared variables are required in the monitor?

1.5 Getting approved

To pass you need to show your solution to the lab supervisor. You should be able to explain what you have done and motivate why you have done it. A working Java solution free of realtime problems is compulsory, common problems are race conditions (the visualization seem to be "jumping"), data corruption and excessive use of CPU.

1.6 Miscellaneous

- *At home* You should be able to complete this exercise on your own computer. You need to download the code using a sftp client such as filezilla.
- In this exercise you *must* use monitors with wait/notify in your solution.
- Also, it is not allowed for you to share thread and monitor classes, i.e. you are not allowed to have thread classes that double as monitors. This is often a good design praxis when building embedded systems.