CSI3131 – Lab3
Thread Synchronization (Java)

Objectives

To gain experience with Java semaphores in creating a simulation of a ferry that services automobiles and an ambulance. **Read the lab document carefully before starting**.

Part1 - Fibonacci sequence

In mathematics, the Fibonacci numbers, commonly denoted Fn, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.
That is, $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-2} + F_{n-1}$, $n > 1$.
The beginning of the sequence is thus: 0,1,1,2,3,5,8,13,21,34,55,89,144,…

A computer scientist implemented the Fibonacci as a single threaded console application. A mad scientist implemented the sequence using a thread dedicated for each element n>2. In the attached code fibonacci.java include both implementation. The single thread application works fine. The mad scientist application does not work! Why? Your first task is to fix the mad scientist application to work properly using threads.

Part2 - The Ferry

The ferry offers services between two ports, port 0 and port 1. Automobiles travel between these two ports, as well as ambulance(s). Each automobile arrives at a port, boards the ferry (when it arrives at the port), crosses to the other port, disembarks the ferry, travels around for a bit, and then goes back to the port to cross again.

The ambulance is a vehicle (the automobiles and ambulance are both vehicles) that functions like the automobile with the exception: **when an ambulance boards the ferry, the ferry leaves immediately** without waiting to be full.

The ferry travels between the two ports (0 and 1): when it arrives at a port, vehicles on board disembark first, then any waiting vehicles board. When the ferry is full, or an ambulance boards, it leaves for the other port.

The provided java code Lab4.java contains 5 classes as follows:

- Class FerryApp is a console application that create the Ferry and all automobiles in the simulation
- Class Auto simulates a single automobile thread
- Class Ambulance simulates a single ambulance thread
- Class Ferry simulates the ferry thread
- Interface Logger that is used to assert correct operation of the ferry

To compile and run the simulation:

- javac *.java
- java FerryApp, or java **-ea** FerryApp
    - (ea or **Enable Assertion** is used to assert the proper operation of the ferry, with this flag enable the application will generate exceptions when an error is detected. Use this flag before submitting your lab to make sure your solution works fine)

You may pipe the output to a file for further analysis as follows (Linux only?!):

- java FerryApp | tee tmp.txt

Notes

Notice when you run the simulation there are going to be a lot of violations due to thread synchronization errors! Your task is to add the proper thread synchronization constructs to obey all the rules of the ferry simulation and have no errors as a result.

Your simulation must meet the following Rules:

1. The maximum capacity of the ferry is 5 vehicles.
2. A vehicle at port "p" boards the ferry if
    a. The ferry is at the same port p,
    b. All vehicles that arrived with the ferry on board have all disembarked the ferry, and
    c. There still exists room on the ferry for the vehicle.
3. A vehicle at port « p » must wait
    a. If the ferry is not at the same port « p »,
    b. Vehicles are disembarking from the ferry,
    c. The ferry is full and is ready to leave.
4. If an ambulance boards the ferry, the ferry leaves immediately.
5. If the ferry is not full and with no ambulance on board, it waits for other automobiles to arrive.
6. A vehicle can disembark only at the arrival at the next port.

The above rules are checked in the code, you must leave all checks as is. The checks are going to be used to verify your solution.

Your Task

Your task is to modify the Auto, Ambulance, and Ferry classes by adding semaphore constructs to the Ferry class that will make the ferry simulation pass all the ferry rules without any error.

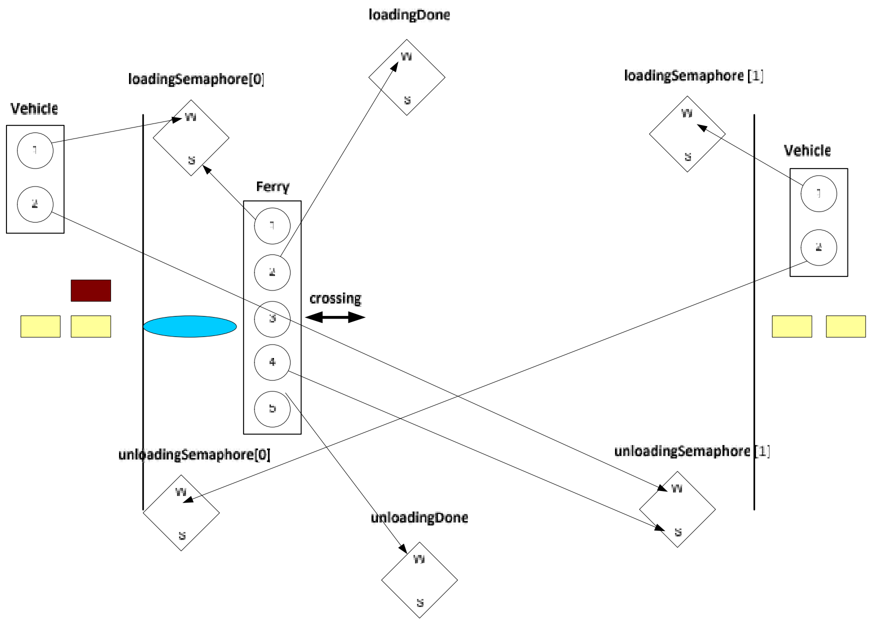
Guidelines

You can use the following guidelines:

- The single Ferry object is visible in all other threads, thus it is an ideal candidate to hold all the required semaphores for simulation
- The **addLoad**, **reduceLoad**, **loadAmbulance**, **unloadAmbulance** methods should be mutually exclusive. You can use the intrinsic locks for that – see the lecture slides about intrinsic locks
- The **Ferry** needs 2 semaphores, namely **loadingDone**, **unloadingDone**, to wait for vehicles to be loaded before starting a crossing, and then waiting for vehicles to unload before allowing vehicles to load
- Vehicles (Auto, Ambulance) needs **loadingSemaphore** and **unloadingSemaphore** on each port (i.e. array of 2 each) to synchronize the loading/unloading of vehicles on each port
- When you provide a handle for InterruptedException, use the following code snip-it to let the application terminate gracefully:

```
          try {
             ////// your code here ///////
          } catch (InterruptedException ie) {
            Thread.currentThread().interrupt();
            System.out.println("Thread terminated by interruption");
            // handle the interrupt
            return;
          }
```

When the Ferry is docking on **port0** the situation looks like the following:

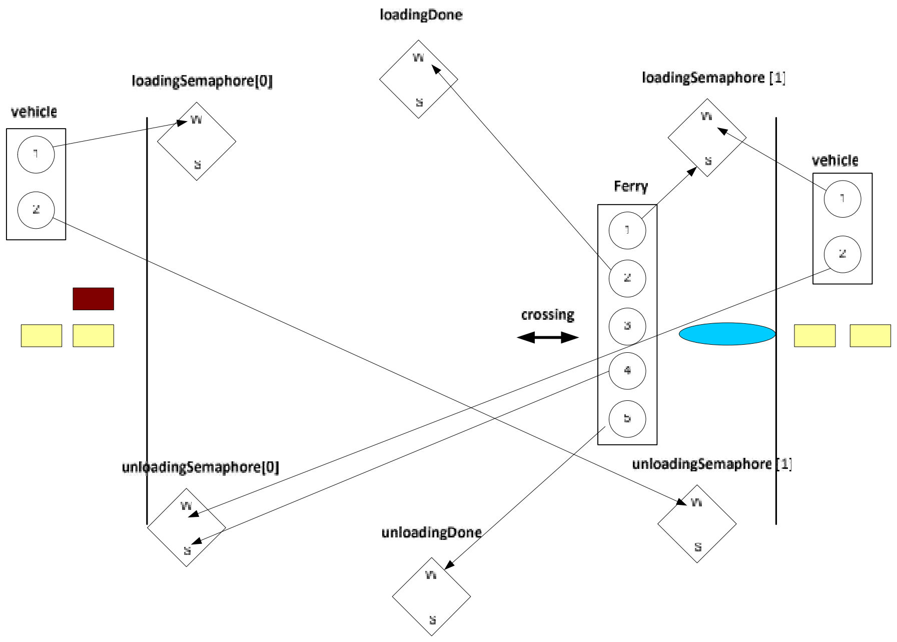


First the ferry signals the loading on port0, then waits for loading done, makes the crossing and then signals the unloading on port1, followed by a wait for unloading done. This finishes one loop of the ferry crossing.

Meanwhile, the **vehicles** on both sides are waiting to load on that port, then waits to unload on the opposite port after the ferry completes the crossing. The difference between the **Ambulance** and **Auto** is that the ambulance makes sure no other Auto is allowed to board after it by draining the permits (see Java Semaphore **drainPermits**) of the loading semaphore after entering it.

What is left is the events to signal the **loadingDone**, and **unloadingDone** semaphores, which can be easily detected in the **addLoad**, **reduceLoad**, **loadAmbulance**, **unloadAmbulance** methods of the ferry.

When the Ferry is docking on **port1** the situation looks like the following:



Submission Note

Your code must compile & run to get the full mark. We're going to use other scenarios (number of crossings, autos, ambulances) to verify your solution. A bug free solution will support any scenario.