Problem 1. A savings account object holds a non-negative balance, and provides deposit(k) and withdraw(k) methods, where deposit(k) adds k to the balance, and withdraw(k) subtracts k, if the balance is at least k, and otherwise blocks until the balance becomes k or greater.

1. Implement this savings account using locks and conditions in SavingsAccount.java.

2. Now suppose there are two kinds of withdrawals: ordinary and preferred. Devise an implementation that ensures that no ordinary withdrawal occurs if there is a preferred withdrawal waiting to occur. Submit this as SavingsAccount2.java.

3. Now let us add a transfer method that transfers a sum from one account to another:

```java
void transfer (int k, Account reserve) {
    lock .lock ();
    try {
        reserve .withdraw(k);
        deposit(k);
    } finally {
        lock .unlock ();
    }
}
```

We are given a set of 10 accounts, whose balances are unknown. At 1:00, each of n threads tries to transfer $100 from another account into its own account. At 2:00, a Boss thread deposits $1000 to each account. Is every transfer method called at 1:00 certain to return?

Problem 2. In the CIT, there are limited spaces available for TA staffs to gather to grade. Furthermore, if TAs are grading for a course in a computer lab, no students in that course should also be in the lab. This sometimes causes difficulties for the CS32 (Software Engineering) and CS145 (Probability) TAs, as both have a lot of grading to do, but are also TAs from other classes that are trying to use the labs. To help alleviate this, a single lab has been designated for use by only those two courses – that is, no other course may use that lab, and they may not use any other lab. However, many TAs of each course are taking the other, so the problem is not yet entirely solved.

In this statement of the TA grading problem, there are two classes of threads, SoftwareTAs and ProbabilityTAs. There is a single Lab resource that must be used in the following way:

1. Mutual exclusion: TAs of different courses may not occupy the lab simultaneously,
2. Starvation-freedom: everyone who needs to grade in the lab eventually enters.

The protocol is implemented via the following four procedures: enterSoftwareTa() delays the caller until it is ok for a CS32 TA to enter the lab, while leaveSoftwareTa() is called when a CS32 TA leaves the lab. enterProbTa() and leaveProbTa() do the same for CS145 TAs. For example,

```java
enterSoftwareTa();
grade("bacon")
leaveSoftwareTa();
```
class Driver {
    void main() {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(n);
        for (int i = 0; i < n; ++i) // start threads
            new Thread(new Worker(startSignal, doneSignal)).start();
        doSomethingElse(); // get ready for threads
        startSignal.countDown(); // unleash threads
        doSomethingElse(); // bidding my time ...
        doneSignal.await(); // wait for threads to finish
    }
}

class Worker implements Runnable {
    private final CountDownLatch startSignal, doneSignal;
    Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
        startSignal = myStartSignal;
        doneSignal = myDoneSignal;
    }
    public void run() {
        startSignal.await(); // wait for driver's OK to start
        doWork();
        doneSignal.countDown(); // notify driver we're done
    }
}

Figure 1: The CountDownLatch class: an example usage.

1. Implement this class in ConditionLab.java using locks and condition variables.

2. Implement this class in SynchronizedLab.java using synchronized, wait, notify, and notifyAll.

For each implementation, explain why it satisfies mutual exclusion and starvation-freedom.

Problem 3. Consider an application with distinct sets of active and passive threads, where we want to block the passive threads until all active threads give permission for the passive threads to proceed.

A CountDownLatch encapsulates a counter, initialized to be n, the number of active threads. When an active method is ready for the passive threads to run, it calls countDown(), which decrements the counter. Each passive thread calls await(), which blocks the thread until the counter reaches zero. (See Figure 1.)

Implement a reusable CountDownLatch in CountDownLatch.java.