Class Activity (Exercise 95,96)

-Lakshmi Madhuri Yarava,2018101116

Tutorial 5: Ch. 8 Q. 96

Part1 :- 1. Implement this class using locks and condition variables.

```
public class BathroomConditional {
        Lock lck;
    Condition conditional;
    int amountOfMale,amountOfFemale;

    public BathroomConditional(){
        lck = new ReentrantLock();
        conditional = lck.newCondition();
        amountOfMale = 0;
        amountOfFemale = 0;
    }

    public void enterMale(){
        try{
            lck.lock();
        try {
            while (amountOfFemale>0) {conditional.await();}
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
            amountOfMale++;
            System.out.println("HOMENS: "+amountOfMale );
        }finally{
            lck.unlock();
        }
    }
    public void enterFemale(){
        try{
            lck.lock();
        try {
            while (amountOfMale>0) {

                conditional.await();
                }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
            amountOfFemale++;
            System.out.println("MULHERES: "+amountOfFemale);
        }finally{
            lck.unlock();
        }
    }
    public void leaveMale(){
        try{
            lck.lock();
            amountOfMale--;
            conditional.signalAll();
        }finally{
            lck.unlock();
        }
    }
    public void leaveFemale(){
        try{
            lck.lock();
            amountOfFemale--;
            conditional.signalAll();
        }finally{
            lck.unlock();
        }
    }
}
```

Part2:-Implement this class using synchronized, wait(), notify(), and notifyAll().

```
public class BathroomSync {
        volatile boolean areYouThere;
    volatile int amountOfMale,amountOfFemale;
    Object lock;
    public BathroomSync(){
        areYouThere = false;
        lock = new Object();
        amountOfMale = 0;
        amountOfFemale = 0;
    }

    public void enterMale(){
        synchronized (lock){
            try {
                while (areYouThere && (amountOfFemale>0)) {lock.wait();}
            } catch (Exception e) {
                e.printStackTrace();
            }finally{areYouThere = true; amountOfMale++;}
        }
    }
    public void enterFemale(){
        synchronized (lock){
            try {
                while (areYouThere && (amountOfMale>0)) {lock.wait();}
            } catch (Exception e) {
                e.printStackTrace();
            }finally{ areYouThere = true; amountOfFemale++;}
        }
    }
    public void leaveMale(){
        synchronized (lock){
            try{
                areYouThere = false;
                amountOfMale--;
                lock.notifyAll();
            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }
    public void leaveFemale(){
        synchronized (lock){
            try{
                areYouThere = false;
                amountOfFemale--;
                lock.notifyAll();
            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

Tutorial 6: Ch. 8 Q. 95

Part1:-1. Implement this savings account using locks and conditions.

```
class SavingsAccountTest {

    @org.junit.jupiter.api.BeforeEach
    void setUp() {

    }

    @org.junit.jupiter.api.AfterEach
    void tearDown() {
    }

    @Test
    void withdrawTestNoFunds() throws InterruptedException {
        SavingsAccount ac01 = new SavingsAccount(200.82);

        Thread mainRunner = new Thread(() -> {
            try {
                ac01.withdraw(false, 200.83);
            } catch (InterruptedException ignored) { }
        });

        mainRunner.start();

        Thread.sleep(5000);

        assertEquals(Thread.State.WAITING, mainRunner.getState());

        mainRunner.interrupt();
    }

    @Test
    void withdrawTestNormal() throws InterruptedException {
        SavingsAccount ac01 = new SavingsAccount(200.82);
        ac01.withdraw(false, 100.82);
        assertEquals(100.00, ac01.getBalance(), 0.001);
    }

    @Test
    void withdrawTestPreferred() throws InterruptedException {
        SavingsAccount ac01 = new SavingsAccount(200.82);
        ac01.withdraw(true, 100.82);
        assertEquals(100.00, ac01.getBalance(), 0.001);
    }

    @Test
    void depositTest() {
        SavingsAccount ac01 = new SavingsAccount(200.82);
        ac01.deposit(99.18);
        assertEquals(300.00, ac01.getBalance(), 0.001);

    }

    @Test
    void transferNormal() throws InterruptedException {
        SavingsAccount ac01 = new SavingsAccount(200.82);
        SavingsAccount ac02 = new SavingsAccount(199.18);
        ac01.transfer(ac02, 99.18, false);
        assertEquals(300.00, ac01.getBalance(), 0.001);
        assertEquals(100.00, ac02.getBalance(), 0.001);
    }

    @Test
    void transferPriority() throws InterruptedException {
        SavingsAccount ac01 = new SavingsAccount(200.82);
        SavingsAccount ac02 = new SavingsAccount(199.18);
        ac01.transfer(ac02, 99.18, true);
        assertEquals(300.00, ac01.getBalance(), 0.001);
        assertEquals(100.00, ac02.getBalance(), 0.001);
    }

}
```

Part2:-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.

So, there are 2 conditions, one for ordinary threads and one for preferred ones. If the account contains less than the amount asked, both ordinary and preferred threads await for the condition of satisfaction. When a new deposit is made all threads in preferred condition are being notified and if there is none, then the threads in the ordinary condition are being notified.

```
      void withdraw(boolean preferred, double amount) throws InterruptedException {
        transactionLock.lock();
        try {
            if (preferred) {
                preferredWaiting++;
                while (balance < amount) {
                    sufficientFundsPriorityCondition.await();
                }
                preferredWaiting--;
                balance -= amount;
                notifyNextThread();
            } else {
                while (balance < amount) {
                    sufficientFundsCondition.await();
                }
                balance -= amount;
                notifyNextThread();
            }
        } finally {
            transactionLock.unlock();
        }
    }

    private void notifyNextThread() {
        if (preferredWaiting == 0) {
            sufficientFundsCondition.signal();
        } else {
            sufficientFundsPriorityCondition.signal();
        }
    }
```