Zadanie I - Bezpieczna synchronizacja

Do zaimplementowania klasa o nazwie **Locker** w pakiecie domyślnym i pliku "Locker.java". Klasa Locker ma definiować wewnętrzne klasy: statyczną *DeadlockException* i niestatyczną *Lock*

Wątki operujące na obiekcie klasy Locker uzyskują od niego, przy pomocy metody `getLock()`, obiekt typu `Locker.Lock`. Kazdy tak uzyskany obiekt udostępnia dwie metody:

- `lock()` - lockuje obiekt na rzecz którego została wykonana i wychodzi. Jeśli wątek który woła metodę *lock()* jest już właścicielem tego locku to zwiększa on o jeden ilość locków jakie trzyma na tym obiekcie. Jeśli inny wątek (przy użyciu tej metody) zalockował wcześniej ten obiekt, metoda zawiesi wykonanie wątku wywołującego ją, do czasu kiedy będzie on mógł uzyskać tenże lock. Jeśli w trakcie czekania na uzyskanie praw do tego locku wątek zostanie przerwany (porzez interrupt) to z metody *lock* powinien zostać wyrzucony odpowiedni wyjątek.
- `unlock()` - oddaje lock na obiekcie, tzn. zmiejsza ilość locków które aktualny wątek trzyma na tym obiekcie o jeden. Jeśli liczba ta spadła do zera wątek oddaje prawo do obiektu i inny wątek (czekający na lock, albo wywołujący *lock()* później) może zalockować obiekt. Jeśli *unlock()* wykonany jest nie przez wątek trzymający lock, metoda rzuca wyjątek `IllegalMonitorStateException`.

Powyższe funkcjonalności są prawie identyczne do zapewnianych przez normalnie zaimplementowane w Javie `ReentrantLock`. Dodatkowa funkcjonalność jest następująca: klasa *Locker* trzyma informacje o wszystkich wykonaniach operacji *lock* i *unlock* **na obiektach które udostępniła przez getLock()**. Jeśli zawieszenie wątku w metodzie *lock()* miało by spowodować zakleszczenie tego wątku, to zamiast go zawieszać, wątek powinien opuścić metodę z wyjątkiem `Locker.DeadlockException`.

Uwagi:

- niedozwolone jest definiowanie klas o nazwach zaczynających się słowem Test
- niedozwolone jest używanie pakietu *java.lang.management*
- kolejnośc w jakiej wątki czekające na lockach uzyskują dostęp do właśnie zwolnionego obiektu nie ma znaczenia
- Locker trzyma informacje o synchronizacjach, które są robione przy użyciu *lock()* i *unlock()* **tylko** na obiektach które on sam wytworzył.

Przykładowe testy

```
import java.util.concurrent.TimeUnit;
public class Test0 {

    public static class SillyThread extends Thread {
        Locker.Lock one, two;
        String me;
        int delay;
```

```
        public SillyThread(String me, int delay, Locker.Lock one, Locker.Lock two) {
            this.me = me;
            this.delay = delay;
            this.one = one;
            this.two = two;
        }
        protected void log(String msg) {
            System.out.println("<" + me + ">: " + msg);
        }
        @Override
        public void run() {
            try {
                one.lock();
                log("got first lock");
                try {
                    TimeUnit.SECONDS.sleep(delay);
                    log("about to wait on second lock");
                    two.lock();
                    TimeUnit.MILLISECONDS.sleep(10);
                    log("got second lock");
                    two.unlock();
                } finally {
                    log("releasing first lock");
                    one.unlock();
                }
            } catch (Locker.DeadlockException e) {
                log("DeadlockException was thrown.");
            } catch (Exception s){
                log("Other exception");
            }
            log("finishing smoothly");
        }
    }

    public static void main(String[] args) throws Locker.DeadlockException, InterruptedException {
        Locker locker = new Locker();
        Locker.Lock lock1 = locker.getLock(), lock2 = locker.getLock();
        new SillyThread("ONE", 1, lock1, lock2).start();
        TimeUnit.MILLISECONDS.sleep(10);
        new SillyThread("TWO", 2, lock2, lock1).start();
    }
}
```

```
<ONE>: got first lock
<TWO>: got first lock
<ONE>: about to wait on second lock
<TWO>: about to wait on second lock
<TWO>: releasing first lock
<TWO>: DeadlockException was thrown.
<TWO>: finishing smoothly
<ONE>: got second lock
```

```
<ONE>: releasing first lock
<ONE>: finishing smoothly
```

```
import java.util.concurrent.TimeUnit;

public class Test1 {

    public static class SillyThread extends Thread  {
        Locker.Lock one, two;
        String me;
        int delay;

        public SillyThread(String me, int delay, Locker.Lock one, Locker.Lock two) {
            this.me = me;
            this.delay = delay;
            this.one = one;
            this.two = two;
        }

        protected void log(String msg) {
            System.out.println("<" + me + ">: " + msg);
        }
        @Override
        public void run() {
            try {
                one.lock();
                TimeUnit.MILLISECONDS.sleep(10);
                log("got first lock");
                try {
                    TimeUnit.SECONDS.sleep(delay);
                    log("about to wait on second lock");
                    two.lock();
                    log("got second lock");
                    two.unlock();
                } finally {
                    log("releasing first lock");
                    one.unlock();
                }
            } catch (Locker.DeadlockException e) {
                log("DeadlockException was thrown.");
            } catch (Exception s){
                log("Other exception");
            }
            log("finishing smoothly");
        }
    }

    public static void main(String[] args) throws Locker.DeadlockException, InterruptedException {
        Locker locker = new Locker();
        Locker.Lock lock1 = locker.getLock(), lock2 = locker.getLock();
        new SillyThread("ONE", 1, lock1, lock2).start();
```

```
        TimeUnit.MILLISECONDS.sleep(10);
        new SillyThread("TWO", 2, lock1, lock2).start();
    }
}
```

```
<ONE>: got first lock
<ONE>: about to wait on second lock
<ONE>: got second lock
<ONE>: releasing first lock
<ONE>: finishing smoothly
<TWO>: got first lock
<TWO>: about to wait on second lock
<TWO>: got second lock
<TWO>: releasing first lock
<TWO>: finishing smoothly
```

```
import java.util.concurrent.TimeUnit;
public class Test2 {

    public static class SillyThread extends Thread {

        Locker.Lock one;
        String me;
        int delay;

        public SillyThread(String me, int delay, Locker.Lock one) {
            this.me = me;
            this.delay = delay;
            this.one = one;
        }

        protected void log(String msg) {
            System.out.println("<" + me + ">: " + msg);
        }
        @Override
        public void run() {
            try {
                one.lock();
                log("got first lock");
                try {
                    TimeUnit.SECONDS.sleep(delay);
                    log("about to wait on second lock");
                    one.lock();
                    log("got second lock");
                    one.unlock();
                    log("unlocked second, time to sleep");
                    TimeUnit.SECONDS.sleep(delay);
                } finally {
                    log("releasing first lock");
                    one.unlock();
```

```
                }
            } catch (Locker.DeadlockException e) {
                log("DeadlockException was thrown.");
            } catch (Exception s){
                log("Other exception");
            }
            log("finishing smoothly");
        }
    }

    public static void main(String[] args) throws Locker.DeadlockException, InterruptedException {
        Locker locker = new Locker();
        Locker.Lock lock = locker.getLock();
        new SillyThread("ONE", 1, lock).start();
        TimeUnit.MILLISECONDS.sleep(10);
        new SillyThread("TWO", 1, lock).start();
    }
}
```

```
<ONE>: got first lock
<ONE>: about to wait on second lock
<ONE>: got second lock
<ONE>: unlocked second, time to sleep
<ONE>: releasing first lock
<ONE>: finishing smoothly
<TWO>: got first lock
<TWO>: about to wait on second lock
<TWO>: got second lock
<TWO>: unlocked second, time to sleep
<TWO>: releasing first lock
<TWO>: finishing smoothly
```