PARALLELISM AND CONCURRENCY
JAVA THREADS. EXERCISES

EXERCISE Multi TIC-TACtac-TOE SEMAPHORE-BASED SOLUTIONS

In the TIC-TACtac-TOE problem, several threads compete to endlessly write TIC-, TAC or tac and -TOE. Now, there are several threads of each kind (TIC, TAC and TOE), making synchronization more difficult that in the previous versions.

The purpose of the program is to endlessly print a TIC-TAC-TOE line followed by a TIC-tac-TOE line (the first TAC is uppercase while the second is lowercase).

PART A solve the problem using three semaphores encapsulated in an object shared by all the threads. These four elements being public can be directly accessed by all the threads.

```
// encapsulation of the elements the threads need to share
class SharedBundle {
    public volatile boolean uppercase = true;
    public Semaphore canTic = new Semaphore(●);
    public Semaphore canTac = new Semaphore(●);
    public Semaphore canToe = new Semaphore(●);
}
```

PART B now add the following restrictions

- Two consecutive TICs cannot have the same id and
- In the same line the TIC and the TOE must have the same id

```
TIC(1)-tac[9]-TOE(1)
TIC(3)-TAC(1)-TOE(3)
TIC(4)-tac[6]-TOE(4)
TIC(7)-TAC(5)-TOE(7)
TIC(5)-tac[4]-TOE(5)
TIC(2)-TAC(8)-TOE(2)
TIC(0)-tac[0]-TOE(0)
TIC(8)-TAC(3)-TOE(8)
TIC(6)-tac[7]-TOE(6)
TIC(9)-TAC(2)-TOE(9)
TIC(1)-tac[9]-TOE(1)
TIC(3)-TAC(1)-TOE(3)
TIC(4)-tac[6]-TOE(4)
TIC(7)-TAC(5)-TOE(7)
TIC(5)-tac[4]-TOE(5)
```

PART C 1 encapsulate the synchronization code in the shared object (we will call it now a synchronizer). Use public **letMe*** and ***Done** operations.

```
class Synchronizer {
    private volatile boolean uppercase = true;
    private volatile int lastTicId = -1;
    private Semaphore canTic = new Semaphore(1);
    private Semaphore canTac = new Semaphore(0);
    private Semaphore canToe = new Semaphore(0);

    public void letMeTic (int id) {/*...*/}
    public void ticDone () {/*...*/}
    public void letMeTac () {/*...*/}
    public void tacDone ()  {/*...*/}
    public void letMeToe (int id) {/*...*/}
    public void toeDone () {/*...*/}

    public boolean nowUppercase () {return this.uppercase;}
}
```

PART C 2 In this part there's another "character": a Frog. A Frog is a thread that every second (approx.) leaps and then croaks (writes CROAK). The frog can only croak after a TOE.

```
TIC(0)-TAC[9]-TOE(0)
TIC(1)-tac[2]-TOE(1)
TIC(2)-TAC[1]-TOE(2)

    CROAK!

TIC(3)-tac[3]-TOE(3)
TIC(4)-TAC[4]-TOE(4)
TIC(5)-tac[5]-TOE(5)
TIC(6)-TAC[0]-TOE(6)
TIC(7)-tac[6]-TOE(7)
TIC(8)-TAC[7]-TOE(8)
TIC(9)-tac[8]-TOE(9)
TIC(0)-TAC[9]-TOE(0)
TIC(1)-tac[2]-TOE(1)
TIC(2)-TAC[1]-TOE(2)
TIC(3)-tac[3]-TOE(3)
TIC(4)-TAC[4]-TOE(4)
TIC(5)-tac[5]-TOE(5)
TIC(6)-TAC[0]-TOE(6)
TIC(7)-tac[6]-TOE(7)
TIC(8)-TAC[7]-TOE(8)

    CROAK!

TIC(9)-tac[8]-TOE(9)
TIC(0)-TAC[9]-TOE(0)
```

```
class Frog extends Thread {

    private Synchronizer synchronizer;

    public Frog (Synchronizer synchronizer) {
        this.synchronizer = synchronizer;
    }

    public void run () {
        while (true) {
            try {sleep(1000);} catch (InterruptedException ie) {}
            synchronizer.letMeLeap();
            System.out.println();
            System.out.println("    CROAK!");
            System.out.println();
            synchronizer.leapDone();
        }
    }
}
```

```
class Synchronizer {
    private volatile boolean uppercase = true;
    private volatile int lastTicId = -1;
    private volatile boolean frogReady = false;

    private Semaphore canTic = new Semaphore(1);
    private Semaphore canTac = new Semaphore(0);
    private Semaphore canToe = new Semaphore(0);
    private Semaphore canLeap = new Semaphore(0);

    /* ... */
```

In this part there is a new semaphore (canLeap) and a boolean variable that registers whether the frog is ready to croak or not. There are also **letMeLeap** and **leapDone** methods.

PART D Redo part C 1 using a single semaphore.