PARALLELISM AND CONCURRENCY
PRACTICAL ASSIGNMENT #1 WINTER TERM 2024
JAVA THREADS AND SYNCHRONIZATION

Exercise 1: The "JACK-QUEEN-KING" problem

The JACK-QUEEN-KING (JQK) problem is inspired by POKER and SLOT machines. There are four different types of participants (threads):

- JACKs: they always generate a JACK card and put it on the table.
- QUEENs: they always generate a QUEEN card and put it on the table.
- KINGs: they always generate a KING card and put it on the table.
- CHECKER (only one): verifies the cards on the table and decides the outcome.

The problem unfolds, hand by hand, on a virtual table where the generators (try to) put a card on it. A hand ends when four cards have been put. Then it is time for the checker to verify if the four cards constitute a trio, a poker or nothing. Once the checker has done its job, a new hand can start (the table is "reset")

In this problem, the TABLE is a shared resource that must be used in a synchronized fashion. Synchronization is required for

- Avoiding that two (or more generators) put a card on the table simultaneously
- Avoiding that the checker tries to verify what is on the table before four generators have put their cards on it
- Avoiding that generators put cards on the table when it is full (four cards already on it)

The table can be regarded as a four-slot buffer where generators orderly fill the slots until none is empty.

The table is described by an abstract class that cannot be modified:

```java
public abstract class Table {

    private StringBuilder [] contents;
    protected final int NUM_SLOTS = 4;
    protected volatile int ffs; // first free slot

    public Table () {
        this.contents = new StringBuilder[NUM_SLOTS];
        this.ffs=0;
    }

    public void putCard (String card) {
        // places the card in the first free slot
        // ...
    }

    public StringBuilder [] getContents () {return this.contents;}

    protected abstract void gainExclusiveAccess ();
    protected abstract void releaseExclusiveAccess ();

    public abstract void putJack (int id);
    public abstract void putQueen (int id);
    public abstract void putKing (int id);
    public abstract void cardPut ();

    public abstract void startCheck (int id);
    public abstract void endCheck(int id);
}
```

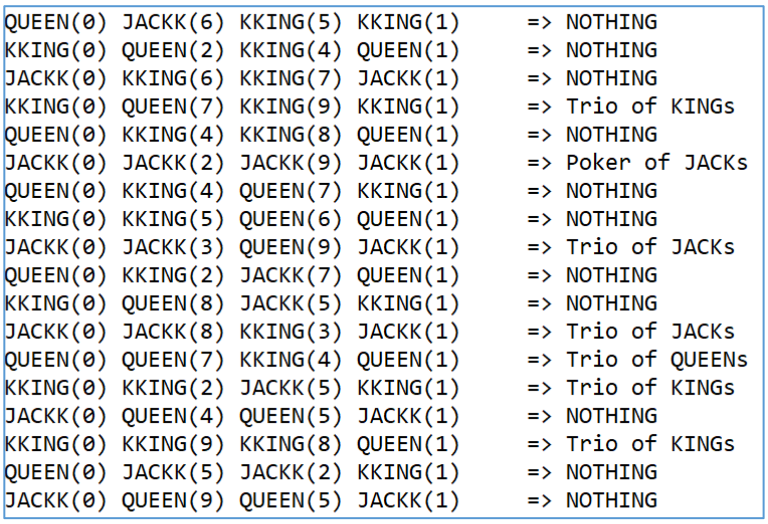
Method **gainExclusiveAccess** implements the "main" synchronization. When **gainExclusiveAccess** returns, the invoking thread is guaranteed to be the only one having the right to "manipulate" the table. After the invocation of **releaseExclusiveAccess**, other threads will have the opportunity to gain exclusive access to the table.

In order to make the problem more challenging some extra rules must be obeyed. The extra rules depend on the particular version of the problem. Methods **putJack**, **putQueen** and **putKing** return only if the conditions stated by the rules hold **and** the invoking thread has exclusive access to the table. **putJack**, **putQueen** and **putKing** will use **gainExclusiveAccess** and **releaseExclusiveAccess** to manage mutual exclusion. The meaning of **cardPut** should be evident. Methods **startCheck** and **endCheck** are used by the Checker thread.

PART 1: The mutex-based table [20%]

In this part, all synchronization is based on a single semaphore with a mutex behaviour.

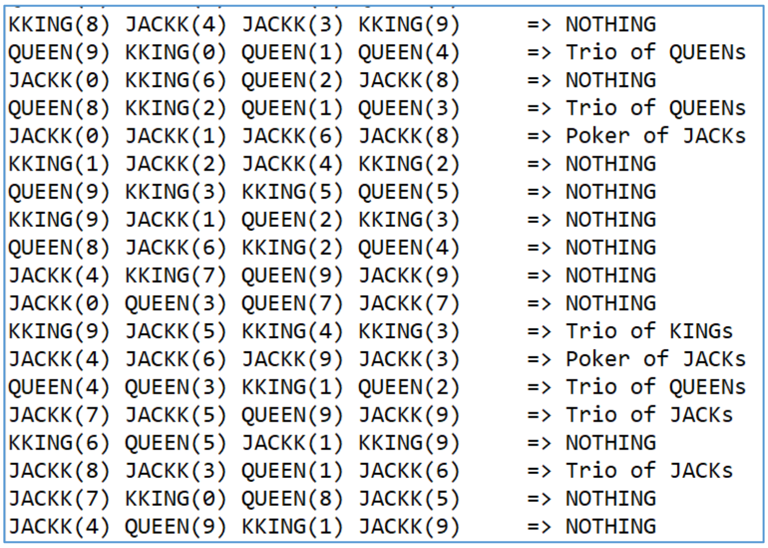
EXTRA-RULE(S): all hands must start with a card put by a generator with id = 0 and end with a card put by a generator with id = 1


```
QUEEN(0) JACKK(6) KKING(5) KKING(1)     => NOTHING
KKING(0) QUEEN(2) KKING(4) QUEEN(1)     => NOTHING
JACKK(0) KKING(6) KKING(7) JACKK(1)     => NOTHING
KKING(0) QUEEN(7) KKING(9) KKING(1)     => Trio of KINGs
QUEEN(0) KKING(4) KKING(8) QUEEN(1)     => NOTHING
JACKK(0) JACKK(2) JACKK(9) JACKK(1)     => Poker of JACKs
QUEEN(0) KKING(4) QUEEN(7) KKING(1)     => NOTHING
KKING(0) KKING(5) QUEEN(6) QUEEN(1)     => NOTHING
JACKK(0) JACKK(3) QUEEN(9) JACKK(1)     => Trio of JACKs
QUEEN(0) KKING(2) JACKK(7) QUEEN(1)     => NOTHING
KKING(0) QUEEN(8) JACKK(5) KKING(1)     => NOTHING
JACKK(0) JACKK(8) KKING(3) JACKK(1)     => Trio of JACKs
QUEEN(0) QUEEN(7) KKING(4) QUEEN(1)     => Trio of QUEENs
KKING(0) KKING(2) JACKK(5) KKING(1)     => Trio of KINGs
JACKK(0) QUEEN(4) QUEEN(5) JACKK(1)     => NOTHING
KKING(0) KKING(9) KKING(8) QUEEN(1)     => Trio of KINGs
QUEEN(0) JACKK(5) JACKK(2) KKING(1)     => NOTHING
JACKK(0) QUEEN(9) QUEEN(5) JACKK(1)     => NOTHING
```

PART 2: The implicit-lock-based table [25%]

In this part, all synchronization is based on the implicit-lock in the table object

EXTRA-RULE(S): all hands must start and end with the same card (generated by the same or by a different generator)



```
KKING(8) JACKK(4) JACKK(3) KKING(9)     => NOTHING
QUEEN(9) KKING(0) QUEEN(1) QUEEN(4)     => Trio of QUEENs
JACKK(0) KKING(6) QUEEN(2) JACKK(8)     => NOTHING
QUEEN(8) KKING(2) QUEEN(1) QUEEN(3)     => Trio of QUEENs
JACKK(0) JACKK(1) JACKK(6) JACKK(8)     => Poker of JACKs
KKING(1) JACKK(2) JACKK(4) KKING(2)     => NOTHING
QUEEN(9) KKING(3) KKING(5) QUEEN(5)     => NOTHING
KKING(9) JACKK(1) QUEEN(2) KKING(3)     => NOTHING
QUEEN(8) JACKK(6) KKING(2) QUEEN(4)     => NOTHING
JACKK(4) KKING(7) QUEEN(9) JACKK(9)     => NOTHING
JACKK(0) QUEEN(3) QUEEN(7) JACKK(7)     => NOTHING
KKING(9) JACKK(5) KKING(4) KKING(3)     => Trio of KINGs
JACKK(4) JACKK(6) JACKK(9) JACKK(3)     => Poker of JACKs
QUEEN(4) QUEEN(3) KKING(1) QUEEN(2)     => Trio of QUEENs
JACKK(7) JACKK(5) QUEEN(9) JACKK(9)     => Trio of JACKs
KKING(6) QUEEN(5) JACKK(1) KKING(9)     => NOTHING
JACKK(8) JACKK(3) QUEEN(1) JACKK(6)     => Trio of JACKs
JACKK(7) KKING(0) QUEEN(8) JACKK(5)     => NOTHING
JACKK(4) QUEEN(9) KKING(1) JACKK(9)     => NOTHING
```

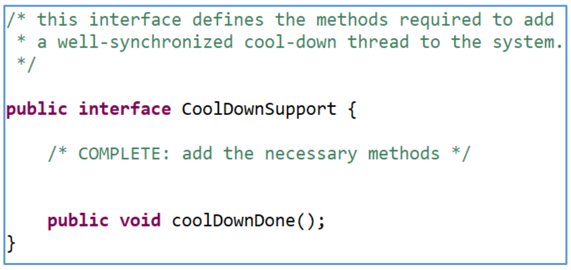
PART 3: The lock-based table with a COOL-DOWN [25%]

In this part, all synchronization is based on an explicit re-entrant lock

EXTRA-RULE(S): all hands must end with a QUEEN generated by a generator with an even id. This rule does not apply to hands prematurely ended by the cool-down thread.

There is an extra participant: the **cool-down thread**. This thread becomes active once every second aprox. When active, it cools down the whole system. Cooling down is just a countdown from 3 to 0 that starts as soon as the second element on the table is a KING. While the cool down thread is counting no cards are put on the table. After cooling down a new hand starts.

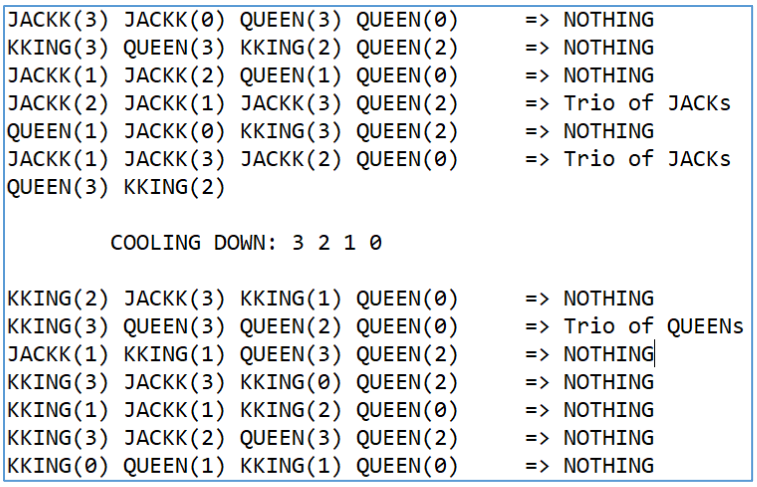
For synchronization, the cool-down thread relies on several methods declared in the CoolDownSupport interface (You must complete it). This interface must be implemented by the same class that provides synchronization support to the rest of active elements.



```java
/* this interface defines the methods required to add
 * a well-synchronized cool-down thread to the system.
 */

public interface CoolDownSupport {

    /* COMPLETE: add the necessary methods */


    public void coolDownDone();
}
```

The incomplete CoolDownSupport interface

The following image shows a hand "interrupted" by the cool-down thread. Notice that the cooling down started after a KING and that the interrupted hand is not continued: after the cool down a new hand starts.



```
JACKK(3) JACKK(0) QUEEN(3) QUEEN(0)     => NOTHING
KKING(3) QUEEN(3) KKING(2) QUEEN(2)     => NOTHING
JACKK(1) JACKK(2) QUEEN(1) QUEEN(0)     => NOTHING
JACKK(2) JACKK(1) JACKK(3) QUEEN(2)     => Trio of JACKs
QUEEN(1) JACKK(0) KKING(3) QUEEN(2)     => NOTHING
JACKK(1) JACKK(3) JACKK(2) QUEEN(0)     => Trio of JACKs
QUEEN(3) KKING(2)

        COOLING DOWN: 3 2 1 0

KKING(2) JACKK(3) KKING(1) QUEEN(0)     => NOTHING
KKING(3) QUEEN(3) QUEEN(2) QUEEN(0)     => Trio of QUEENs
JACKK(1) KKING(1) QUEEN(3) QUEEN(2)     => NOTHING
KKING(3) JACKK(3) KKING(0) QUEEN(2)     => NOTHING
KKING(1) JACKK(1) KKING(2) QUEEN(0)     => NOTHING
KKING(3) JACKK(2) QUEEN(3) QUEEN(2)     => NOTHING
KKING(0) QUEEN(1) KKING(1) QUEEN(0)     => NOTHING
```

Exercise 2: The “DING^n-DANG^n-DONG” problem

There are three different types of threads, those writing ***DING-***, those writing ***daang-*** and those writing ***DOOONG*** and a *newline*.

These three types of threads must be synchronized in such a way that they always write lines that have the following pattern:

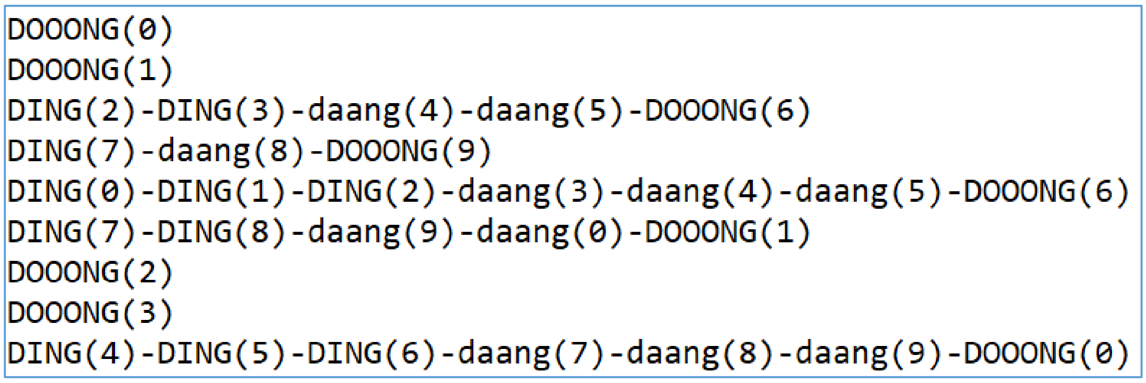
1. All lines end with a single DONG
2. A line can start with a DING or with a DONG. If it starts with a DONG, that element is the first and also the last.
3. At the beginning of a line there can be up to three DINGs.
4. After the DINGs there must follow an equal number of DANGs.

These are ALL the possible lines:

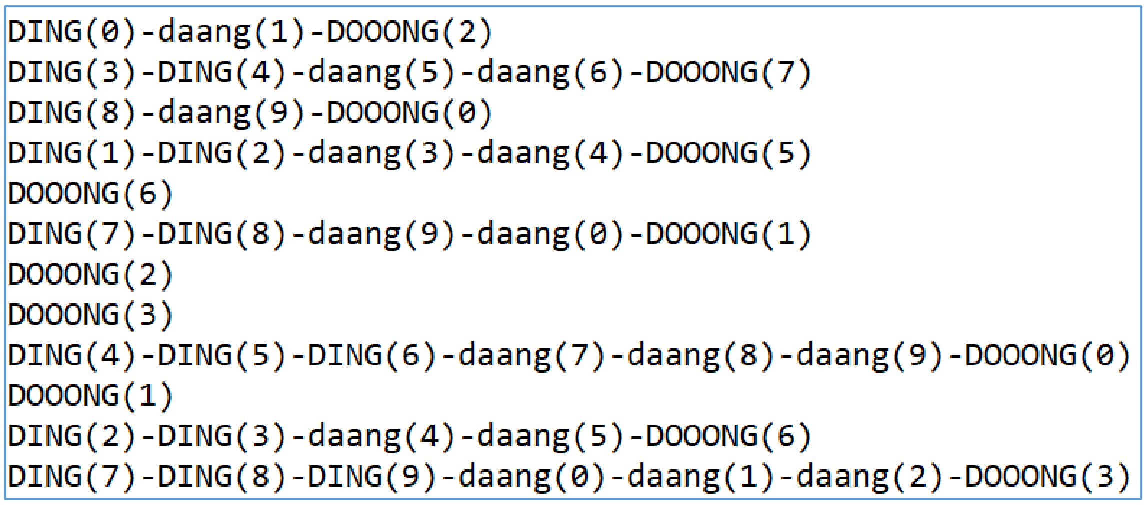
DOOONG
DING daang **DOOONG**
DING DING daang daang **DOOONG**
DING DING DING daang daang daang **DOOONG**

There is also an extra rule regarding the IDs of the threads producing the elements: they must be consecutive.

The following are screen captures of the actual outputs produced during the execution of the program. As usual, the numbers in brackets are the ids of the threads writing the element.



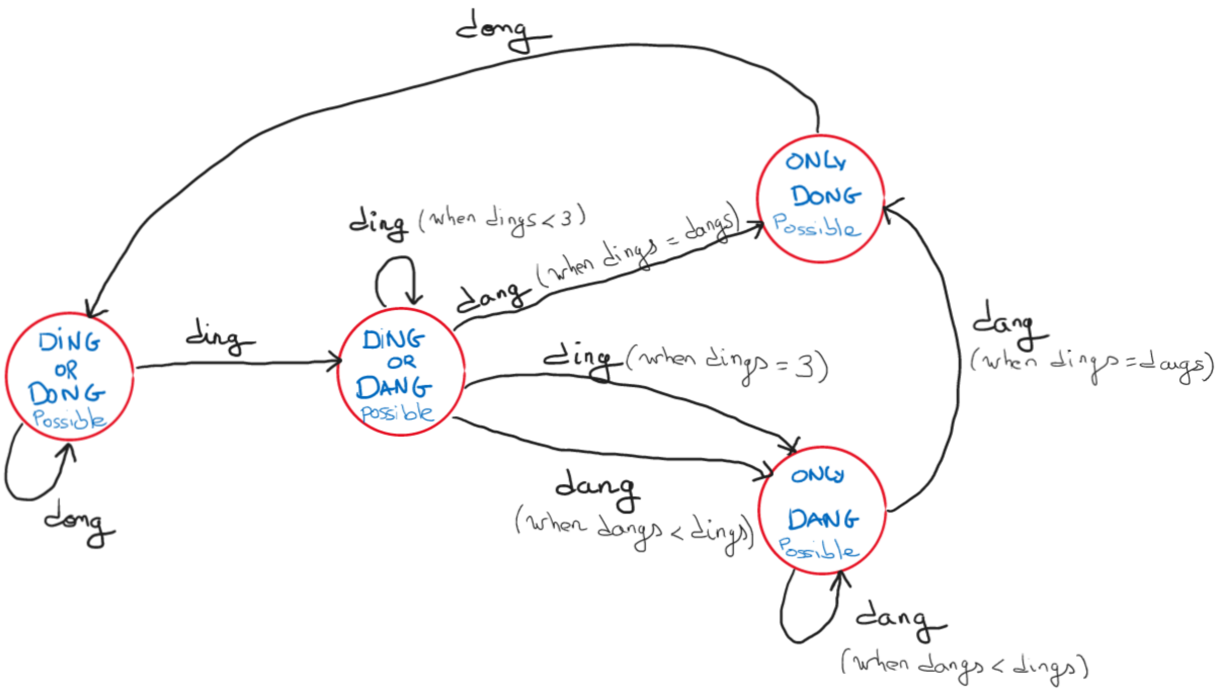
```
DOOONG(0)
DOOONG(1)
DING(2)-DING(3)-daang(4)-daang(5)-DOOONG(6)
DING(7)-daang(8)-DOOONG(9)
DING(0)-DING(1)-DING(2)-daang(3)-daang(4)-daang(5)-DOOONG(6)
DING(7)-DING(8)-daang(9)-daang(0)-DOOONG(1)
DOOONG(2)
DOOONG(3)
DING(4)-DING(5)-DING(6)-daang(7)-daang(8)-daang(9)-DOOONG(0)
```



```
DING(0)-daang(1)-DOOONG(2)
DING(3)-DING(4)-daang(5)-daang(6)-DOOONG(7)
DING(8)-daang(9)-DOOONG(0)
DING(1)-DING(2)-daang(3)-daang(4)-DOOONG(5)
DOOONG(6)
DING(7)-DING(8)-daang(9)-daang(0)-DOOONG(1)
DOOONG(2)
DOOONG(3)
DING(4)-DING(5)-DING(6)-daang(7)-daang(8)-daang(9)-DOOONG(0)
DOOONG(1)
DING(2)-DING(3)-daang(4)-daang(5)-DOOONG(6)
DING(7)-DING(8)-DING(9)-daang(0)-daang(1)-daang(2)-DOOONG(3)
```

To solve the problem, it is important to notice that it goes through several well-defined states:

1. A state (the initial one) where both a DING and a DONG are possible.
2. A state where both DING and DANG are possible (this state is entered after the first DING)
3. A state where only DANG is possible (this state is entered after the first DANG following a DING)
4. A state where only DONG is possible (this state is entered after an equal number of DINGs and DANGs have been produced)



Classes for DING, DANG and DONG are already written and cannot be modified. Synchronization will be achieved by the use of protocol-like **letMe*** and ***Done** methods: **letMeDing**, **letMeDang** and **letMeDong** (of blocking nature); **dingDone, dangDone** and **dongDone**.

```
public class Dang extends Thread {

    private int id;
    private InterfaceSync synchro;

    public Dang (int id, InterfaceSync synchro) {
        this.synchro = synchro;
        this.id = id;
    }

    public void run () {
        while (true) {
            synchro.letMeDang(id);
            synchro.writeString("daang("+id+")-");
            synchro.dangDone();
        }
    }
}
```

Method `writeString` is equivalent to `System.out.println` and is already written.

Part A: [30%] Complete the `CSSync` class using as many simple-typed variables as you deem necessary and a single instance of **`AtomicBoolean`**. `compareAndSet` is the ONLY synchronization mechanism allowed in this problem. Use backOff (a substitute for `Thread.yield`) where it could help improve overall efficiency.