Computation structures

Support for problem-solving lesson #7

Exercise 1

Consider the following Java class:

```
public class MyClass {
    public MyClass ( ) {}
    public synchronized void m1( ) {
        System.out.println("Entering method 1");
        try {Thread.sleep(5000); } catch(Exception e){}
        System.out.println("Exiting method 1");
    }
    public synchronized void m2( ) {
        System.out.println("Entering method 2");
        try {Thread.sleep(5000); } catch (Exception e){}
        System.out.println("Exiting method 2");
    }
```

What could be the outcome of the following programs?

```
MyClass o1, o2 ;
o1 = new MyClass ( ); o2 = new MyClass ( ) ;
new Thread ( ) { public void run ( ) {
    o1.m1( ); } }.start( ) ;
new Thread ( ) { public void run ( ) {
    o1.m2( ); } }.start( ) ;
```

```
MyClass o1, o2 ;
o1 = new MyClass ( ); o2 = new MyClass ( ) ;
new Thread ( ) { public void run ( ) {
    o1.m1( ); } }.start( ) ;
new Thread ( ) { public void run ( ) {
    o2.m1( ); } }.start( ) ;
```

Exercise 1

- Both methods **m1()** and **m2()** are **synchronized**.
- Thus, **for a given object**, several threads cannot access its method simultaneously.
- Consider the first case :
  - One thread wants to execute **o1.m1()**.
  - Another thread wants to execute **o1.m2()**.

Exercise 1

- First possibility:
  - Thread 1 is executed. Entering the function, it aquires the lock on **o1**.
  - From that point, even if Thread 2 wants to execute, it will be blocked outside of **m2()**.
  - Thread 1 executes **o1.m1()** until completion, then releases the lock on **o1**.
  - Thread 2 now can execute **o1.m2()**.

- Second possibility:
  - Same as before, but Thread 2 is executed first, locks Thread 1 outside of **m1()**, finishes **o1.m2()**, releases the lock and Thread 1 can execute **o1.m1()**.

Exercise 1

- Output:

```
Entering method 1
<waits 5 seconds>
Exiting method 1
Entering method 2
<waits 5 seconds>
Exiting method 2
```

or

```
Entering method 2
<waits 5 seconds>
Exiting method 2
Entering method 1
<waits 5 seconds>
Exiting method 1
```

Exercise 1

- Consider the second case:
  - One thread wants to execute **o1.m1()**.
  - Another thread wants to execute **o2.m1()**.

- The lock is aquired **on a given object**, thus Thread 1 obtains the lock on **o1** while Thread 2 obtains the lock on **o2**.

- Thread 1 and Thread 2 are thus **not** in mutual exclusion, even though they both called the same method that was declared **synchronized**.

Exercise 1

- Output:

```
Entering method 1
Entering method 1
<waits ~5 seconds>
Exiting method 1
Exiting method 1
```

Exercise 2

An animal shelter has a room to temporarily store animals that transit from their cages to the vet clinic and reversely.

Rules are :

- The room is only used to hold cats or dogs.
- A cat can never enter the room if it already contains a cat or a dog.
- A dog can never enter the room if it already contains a cat.
- There cannot be more than 4 dogs in the room.

Write a solution to this problem using **synchronized** methods as well as **wait()**, **notify()** and **notifyAll()** calls. Use variables **cats** and **dogs** to represent the number of cats and dogs in the room respectively.

Exercise 2

Recall:

- **wait()** will lock the calling thread and release the mutex aquired by **synchronized** until another thread calls **notify()** or **notifyAll()** in a **synchronized** method of the same object.
- **notify()** will unlock a single thread that called **wait()**. The choice of the Thread to unlock is arbitrary and depends on the implementation. In Java 8, the Thread that called **wait()** first is notified first. If there's none, **notify()** does nothing. The calling thread can continue its execution.
- **notifyAll()** will unlock all threads that called **wait()**,if any. The calling thread can continue its execution.

Exercise 2

Let's first start without any synchronization.

1

```
public class Room {

public Room() { }

public                    void dogEnter()
                                        {



   dogs++;
}

public                    void dogExit(){
   dogs--;

}
```

2

```
public                    void catEnter()
                                        {


   cats++;
}

public                    void catExit(){
   --cats;

}

int dogs=0; //number of dogs in the room
int cats=0; //number of cats in the room

}
```

Exercise 2

I need to protect **cats** and **dogs**, so I use the mutex offered by **synchronized**.

1

```
public class Room {

public Room() { }

public synchronized void dogEnter()
                                        {


   dogs++;
}

public synchronized void dogExit(){
   dogs--;

}
```

2

```
public synchronized void catEnter()
                                        {


   cats++;
}

public synchronized void catExit(){
   --cats;

}

int dogs=0; //number of dogs in the room
int cats=0; //number of cats in the room

}
```

Exercise 2

Dogs can only enter if there's less than 4 dogs, and cats can only be alone.

1

```
public class Room {

public Room() { }

public synchronized void dogEnter()
        throws InterruptedException{
   while(dogs>=4 || cats>0)
      wait();
   dogs++;
}

public synchronized void dogExit(){
   dogs--;

}
```

2

```
public synchronized void catEnter()
     throws InterruptedException{
   while(cats>0 || dogs>0)
      wait();
   cats++;
}

public synchronized void catExit(){
   --cats;

}

int dogs=0; //number of dogs in the room
int cats=0; //number of cats in the room

}
```

Exercise 2

When an animal leaves the room, it might let another animal come in, so we must unlock the waiting threads.

1

```
public class Room {

public Room() { }

public synchronized void dogEnter()
        throws InterruptedException{
   while(dogs>=4 || cats>0)
      wait();
   dogs++;
}

public synchronized void dogExit(){
   dogs--;
   notifyAll();
}
```

2

```
public synchronized void catEnter()
     throws InterruptedException{
   while(cats>0 || dogs>0)
      wait();
   cats++;
}

public synchronized void catExit(){
   --cats;
   notifyAll();
}

int dogs=0; //number of dogs in the room
int cats=0; //number of cats in the room

}
```

Exercise 2

Why did we use **notifyAll()** instead of **notify()**?
Because we could potentially loose a notification.

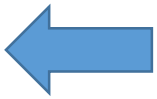
1

```
public class Room {

public Room() { }

public synchronized void dogEnter()
        throws InterruptedException{
   while(dogs>=4 || cats>0)
      wait();
   dogs++;
}

public synchronized void dogExit(){
   dogs--;
   notifyAll();
}
```



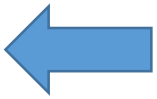
2

```
public synchronized void catEnter()
     throws InterruptedException{
   while(cats>0 || dogs>0)
      wait();
   cats++;
}

public synchronized void catExit(){
   --cats;
   notifyAll();
}

int dogs=0; //number of dogs in the room
int cats=0; //number of cats in the room

}
```



Exercise 2

Loosing a notification:

- Dog 1 wants to enter → OK, **dogs** = 1, **cats** = 0
  Cat 1 wants to enter → KO, calls **wait()**
  Dog 2 wants to enter → OK, **dogs** = 2, **cats** = 0
  Dog 3 wants to enter → OK, **dogs** = 3, **cats** = 0
  Dog 4 wants to enter → OK, **dogs** = 4, **cats** = 0
  Dog 5 wants to enter → KO, calls **wait()**
  Dog 1 leaves the room → calls **notify()**, **dogs** = 3, **cats** = 0
  Cat 1 is awaken by **notify()**, checks **dogs** and **cats** → KO
  → **wait()**

  But Dog 5 could enter and is not notified.

Exercise 2

Why did we use **while** instead of **if**?
Because triggering **notifyAll()** (and even **notify()**) does not guarantee that the condition that lead to the **wait()** is now false.

1

```
public class Room {

public Room() { }

public synchronized void dogEnter()
        throws InterruptedException{
    while(dogs>=4 || cats>0)
       wait();
   dogs++;
}

public synchronized void dogExit(){
   dogs--;
   notifyAll();
}
```

2

```
public synchronized void catEnter()
     throws InterruptedException{
   while(cats>0 || dogs>0)
      wait();
   cats++;
}

public synchronized void catExit(){
   --cats;
   notifyAll();
}

int dogs=0; //number of dogs in the room
int cats=0; //number of cats in the room

}
```

Exercise 2

Using **if** instead of **while** (even replacing **notifyAll()** by **notify()**):

- Dog 1 wants to enter → OK, **dogs** = 1, **cats** = 0
  Cat 1 wants to enter → KO, calls **wait()**
  Dog 2 wants to enter → OK, **dogs** = 2, **cats** = 0
  Dog 1 leaves the room → calls **notify()**, **dogs** = 1, **cats** = 0
  Cat 1 is awaken by **notify()**, and resumes its execution
  → **dogs** = 1, **cats** = 1 → KO

Exercise 3

A bank asks your help to develop a Java program that performs the payments.

Bank accounts are stored in objects of class **Account** that advertise three non-atomic methods:

- **void credit(double amount)** to credit **amount** to the account,
- **void debit(double amount)** to debit **amount** from the account, and
- **String getIBAN()** to get the IBAN of the account.

You must write a method called

transfer(Account from, Account to, double amount)

that will be used in the context of multi-threading, and ensure synchronization is performed in such a way as to keep the accounts in a coherent state while avoiding deadlocks.

Exercise 3

Recall

You can use **synchronized** in three ways:

- For a whole method
  - e.g. **synchronized int theMethod() { /* mutex on the method*/}**
- For a block inside a method (on the current object)
  - e.g. **synchronized(this) { /* mutex for this block only */}**
- For a block inside a method (on a different object)
  - e.g. **synchonized(obj) { /*mutex aquired for object obj*/ }**

Exercise 3

First try

```
public synchronized void transfer(Account from, Account to, double amount)
{
    from.debit(amount);
    to.credit(amount);
}
```

Is it good? If not, why?

The mutex is aquired on the whole object.

I cannot work on account3 and account4 if I'm already working on account1 and account2.

→Too restrictive

Exercise 3

Second try

```
public void transfer(Account from, Account to, double amount)
{
    synchronized(from) {
        synchronized(to) {
            from.debit(amount);
            to.credit(amount);
        }
    }
}
```

Is it good? If not, why?

Imagine the following :

Two transfer orders arrive, one from account1 to account2, and one from account2 to account1

Thread1 gets the lock on account1, then Thread2 gets the lock on account2, hence Thread1 cannot get the lock on account2 and Thread2 cannot get the lock on account1.

→ deadlock!

Exercise 3

Third try

To solve, the deadlock problem, we must find an order for the resources, and lock them accordingly;

```
public void transfer(Account from, Account to, double amount)
{
    if(from.getIBAN().compareTo(to.getIBAN()) < 0) {
        synchronized(from) {
            synchronized(to) {
                from.debit(amount);  to.credit(amount);
            }
        }
    } else {
        synchronized(to) {
            synchronized(from) {
                from.debit(amount);   to.credit(amount);
            }
        }
    }
}
```

Transfer from BE43 0000 0000 0101 to BE32 0000 0000 0202 (Thread1)

Transfer from BE32 0000 0000 0202 to BE43 0000 0000 0101 (Thread2)

BE32… < BE43… → lock on BE32… for both threads

→ No deadlock