BUI Vu Minh Duc
INF2-A

Compte rendu TP1 & TP2 - Programmation répartie

**TP1 Mobile :**

Le code source de ce TP comporte 3 classes java: UneFenetre, UnMobile et Task.

La classe Task fonctionne comme une Thread tandis que la classe UnMobile représente l'objet carré qui bouge de gauche à droite et finalement, la classe UneFenetre est la fenêtre de l'application donc la classe Main sous forme d'un JFrame.

**Exercice 1 :**

La question 1 consiste à lancer une thread. Il suffit d'instancier, de déclarer la thread et la lancer grâce à la fonction start() comme ci-dessous:

```
public class UneFenetre extends JFrame implements ActionListener
{
    UnMobile sonMobile;
    private final int LARG=400, HAUT=250;
    JButton button = new JButton("test");

    public UneFenetre(String fenetre_mere)
    {
        // TODO
        // ajouter sonMobile a la fenetre
        // creer une thread laThread avec sonMobile
        // afficher la fenetre
        // lancer laThread

        //setTitle("thread");
        //setSize(400, 400);

        sonMobile = new UnMobile(LARG, HAUT);
        Thread laTache = new Thread(sonMobile);
        laTache.start();

        JFrame uneFenetre = new JFrame();
        uneFenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        uneFenetre.setSize(LARG,HAUT);
        uneFenetre.setVisible(true);
        uneFenetre.add(sonMobile);

    }
```

Dans la classe UnMobile, nous avons une boucle for qui permet de faire bouger l'objet carré (de gauche à droite). Donc la question 2 nous demande de faire la même chose mais dans le sens inverse. Il suffit de modifier les conditions dans la boucle for pour changer la direction
de l'objet :

```
for (sonDebDessin=0; sonDebDessin < saLargeur - sonPas; sonDebDessin += sonPas)
{
    repaint();
    try{Thread.sleep(sonTemps);}
    catch (InterruptedException telleExcp)
    {telleExcp.printStackTrace();}
}
for(sonDebDessin=saLargeur; sonDebDessin < saLargeur + sonPas; sonDebDessin -= sonPas){
    repaint();
    try{Thread.sleep(sonTemps);}
    catch (InterruptedException telleExcp)
    {telleExcp.printStackTrace();}
}
```

1ere boucle : de gauche à droite
2e boucle : de droite à gauche

Pour lancer 2 threads en même temps, on a besoin tout d'abord d'ajouter 1 GridLayout, ce qui permet de visualiser cet effet dans une même fenêtre.

Ensuite, on doit créer une nouvelle thread (donc la 2e), ainsi, on doit instancier 1 nouveau mobile et nouveau bouton

**Exercice 2 :**

Pour l'exercice 2, on va créer 1 bouton intitulé ON/OFF et on utilise l'ActionListener pour donner des fonctionnalités à ce bouton.

L'actionPerformed contient 2 conditions :

- state au départ = true donc si la state == true, on suspend la tâche grâce à suspend() et on attribue la state maintenant comme false.
- En revanche, si la state au départ est faux, on reprend la tâche avec resume()

* state est un boolean, donc retourne seulement true or false.

Exercice 3 :

Pour réaliser cet exercice, on doit tout d'abord créer un container (contentPane), qui permet de contenir plusieurs threads à la fois, cela veut dire aussi qu'on doit créer plusieurs threads. Pour chaque thread on crée des mobiles et des boutons différents.

**TP2 synchronized, sémaphore :**

## Exercice 1 :

Dans un premier temps, nous utilisons la méthode synchronized() qui permet de faire l'attente d'un accès à une section critique (ref. diapo 9 cours 2).

Pour cela, nous avons besoin de la présence de l'objet d'exclusion mutuelle. Cet objet doit être static car à chaque instanciation, un nouvel objet de type Exclusion est créé.

La syntaxe est la suivante :

```
public void run() {
    synchronized(exclusionMutuelle) {
        //section critique
    }
}
```

## Exercice 2 :

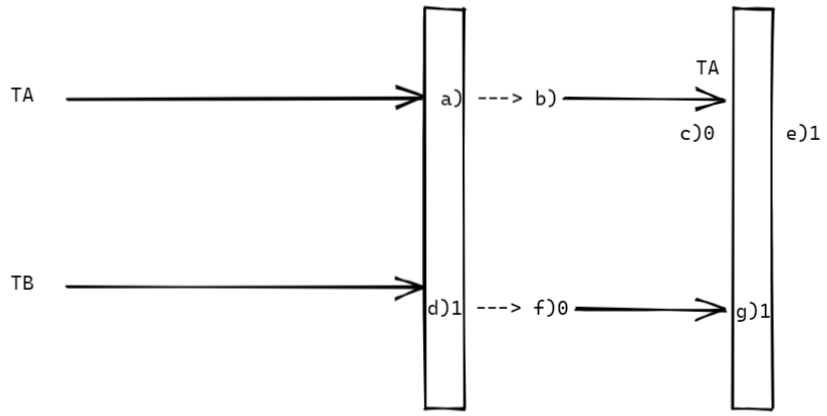
Ensuite, l'exercice 2 consiste à utiliser les méthodes wait() et signal() qui a pour but de faire le même traitement que synchronized.

Pour cela, on a besoin tout d'abord instancier 1 variable static de type semaphoreBinaire, même raison que la variable de type Exclusion. En revanche, on a besoin d'attribuer 1 valeur à cette variable (soit une variable supérieure à 0 soit 0).

Ensuite, nous allons faire l'attente l'accès grâce à la méthode syncWait(). Si la valeur = 0, c'est-à-dire qu'il y a un thread en cours d'exécution et inverse. C'est pour cette raison que si on commence par une valeur de 0, le programme sera bloqué. Une fois le thread est terminé, la valeur incrémentée de 1 pour dire qu'il est libre de lancer une nouvelle thread.

Pour aller plus dans le détail, nous analysons la classe sémaphore que la classe sémaphoreBinaire hérite. Dans cette classe, nous nous intéressons à 2 méthodes syncWait() et syncSignal().

- syncWait() est composée d'un try/catch. En général, si la valeur = 0, on fait l'attente. Si ce n'est pas le cas, on diminue/décrémente la valeur.
- syncSignal() : vu que la valeur = 0 à cause de la méthode précédente (syncWait) donc le travail de syncSignal est d'incrémenter la valeur pour que cette valeur revienne à l'état initial. Ce qui a pour but de notifier qu'il est libre de lancer une nouvelle thread aussi.




sem.wait()
sem.valeur = 1
puis -- → 0

a) 1 b) 0

```java
public class Affichage extends Thread{

    //exercice 1 : synchronized
    //exclusionMutuelle doit etre static car a chaque instanciation, 1 nouveau objet de type Exclusion est cree

    //static Exclusion exclusionMutuelle = new Exclusion ();

    //si la valeur = 0, on est bloque a syncWait donc faut absolument commencer par 1
    static semaphoreBinaire sem = new semaphoreBinaire (1);

    String texte;

    public Affichage (String txt){texte=txt;}

    public void run(){
        //synchronized (exclusionMutuelle) {
            sem.syncWait();
            //section critique
            for (int i=0; i<texte.length(); i++){
                //ressource critique
                System.out.print(texte.charAt(i));
                try {sleep(100);} catch(InterruptedException e){};
            }
            sem.syncSignal();
        }
    }
//}
```

Lien du repo github :
https://github.com/therealminhduc/TP_progpartie.git