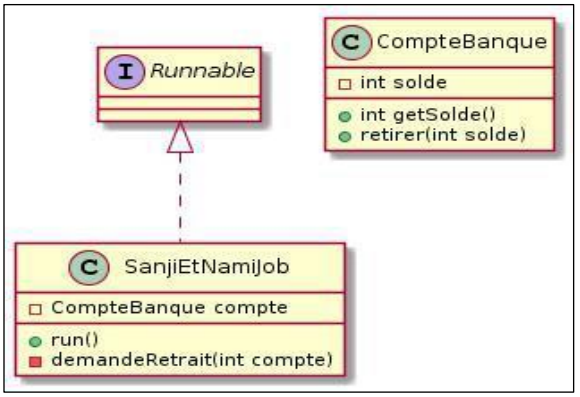
# TP2 : Multithreading

## Threads et accès concurrent

### A. Gestion d'un compte bancaire

Implémenter le code java relatif au digramme de classes ci-dessous en suivant les étapes suivantes:

1. Déclarer une instance de la classe SanjiEtNamiJob (puisque Nami et Sanji font la même chose). Cette classe représente le travail à faire.
2. Créer deux threads avec le même Runnable (l'instance de SanjiEtNamiJob).
3. Attribuer des noms aux threads et lancer la méthode start().
4. Dans la méthode run(), le thread rentre à l'intérieur d'une boucle et tente de faire appel à la méthode demandeRetrait à chaque itération. Après la demande, il vérifie la balance et affiche un message quand le compte est à découvert.




5. La méthode vérifie le solde du compte. S'il n'y a pas assez d'argent le message suivant est affiché :

```
System.out.println("Pas assez d'argent pour "+Thread.currentThread().getName());
```

6. Par contre, s'il y a assez, nous affichons le message :

```
System.out.println(Thread.currentThread().getName() + "est sur le point de retirer.");
```

7. Par la suite, nous allons endormir le thread pour 500ms, puis à son réveil, il exécutera les instructions suivantes :

```
System.out.println(Thread.currentThread().getName() + " reveillé.");
compte.retirer(somme);
System.out.println(Thread.currentThread().getName() + " à compléter le retrait);
```

➔ Expliquer le résultat obtenu et proposer une solution au problème.

## B. Gestion des transferts bancaires :

L'objectif de cet exercice étant d'étudier les **accès concurrents** à une structure de données.

```
public static void main(String[] args) {
    //Simulation
    Banque banque = new Banque(100, 1000);
    for (int depuis = 0; depuis < 100; depuis ++) {
    Runnable r = new Transfert(banque,depuis,1000);
    new Thread(r).start();
    }
}
```

| Banque |
| --- |
| final double[] comptes |
| Banque (int, double)<br>double soldeTotal()<br>void transferer(in,int,double)<br>int size() |

Nous allons simuler plusieurs opérations de transfert d'argent dans une *banque*. Pour les besoins de la simulation, notre banque est un simple tableau de *comptes* chacun réduit à son *solde* donc un simple **double**. Nous considérons exactement 100 comptes créés avec un solde initial de 1000 Dhs. Chaque compte lance *en boucle infinie et d'une façon simultanée des opérations de transfert. Le compte destination et le montant à transférer sont aléatoires* (Math.*random*()).

1. Compléter les classes Banque et Transfert (la tâche –**thread**- effectuant des transferts **depuis** un compte). Exécuter le programme et remarquez que **les données sont corrompus**: le solde total change alors *qu'il doit rester inchangé à 100000 Dhs*.
2. Résoudre le problème précédent en **synchronisant** les transferts de deux façons différentes :
   a. En utilisant le mécanisme des verrous (**ReentrantLock**) et des conditions (**Condition)** du package **java.util.concurrent.locks**.

```
private Lock bankLock = new ReentrantLock(); //implements Lock interface
private Condition soldeSuffisant = bankLock.newCondition();

public void transferer (int de, int vers, double m)
                        throws InterruptedException
 {
   bankLock.lock();
   try{ if(comptes[de] < montant)
            soldeSuffisant.await();
//attendre la condition
    comptes[de] -= m;
    comptes[vers] += m;
     soldeSuffisant.signalAll();
     //débloquer les threads attendant la condition
   } finally{
   bankLock.unlock();
   }
 }
 public double
   soldeTotal() {
```

```
      bankLock.lock();
      try{
            double somme=0;
           for (double c : comptes) {
               somme += c;
         }
         return somme;
      } finally{
      bankLock.unlock();
      }
   } public synchronized double soldeTotal()
   { … }
```

b. En utilisant le mot clé synchronized

```
public synchronized void transferer(int de,int vers, double m)
throws InterruptedException     {
      while(comptes[de] < montant) {
      // boucle !!
      System.out.printf(" Solde insuffisant\n");
      wait(); //attendre
      } comptes[de] -=
      montant;
      comptes[vers] +=
      montant;
      notifyAll(); //notifier les threads en attente
      }
```

***Rappel***

**JDK 1.0 (mot-clé synchronized)** : Chaque objet dispose d'un **verrou implicite**, chaque verrou a une **condition implicite**. Le verrou gère les threads qui tentent de rentrer dans la méthode synchronisée. La condition gère les threads ayant appelés wait().

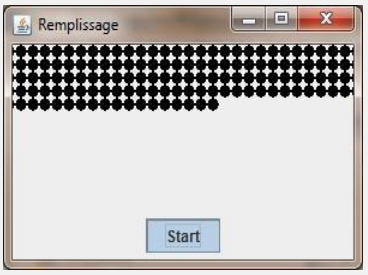
**JDK 5.0 (java.util.concurrent.locks)**: On peut créer explicitement un **verrou** (ReentrantLock) pour protéger une section critique. On utilise aussi un objet (Condition) pour gérer les threads ayant acquis le verrou mais attendent qu'une condition soit satisfaite (par exemple pour un compte, attendre que le solde soit suffisant)

## C. Remplissage et déplacement à l'aide des petites balles :

Ecrire un programme Java qui remplit une partie d'une fenêtre graphique avec des petites balles comme sur la figure ci-dessous.

1. La première version consiste à tout dessiner au niveau de la tâche principale. Utiliser un bouton « start » pour déclencher le remplissage.

```
private void remplir() {
   Graphics g = getGraphics();
   for (int y = 0; y < 100; y = y+10) {
      for (int x = 0; x < 300; x=x+10) {
        g.fillOval(x, y, 10, 10);
         Thread.sleep(100);
      }
   }
}
```




2. Vérifier qu'au cours du remplissage il est **impossible de fermer la fenêtre** ? Proposer une solution avec les threads.

3. Au lieu de remplir l'écran par des petites balles. On lance une balle comme au **billard**. Les parois de la fenêtre réfléchissent la balle. On doit pouvoir lancer plusieurs balles en même temps (**Multithreading**)