TP Thread en JAVA

Exercice 1 : Utiliser les threads

A. Compiler et exécuter le programme suivant :

Essayer de prévoir ce qui devrait s'afficher et comparer à l'exécution.

```
public class TwoThread extends Thread {
 public void run() {
       for ( int i = 0; i < 10; i++ ) {
             System.out.println("New thread");
       }
 }

 public static void main(String[] args) {
       TwoThread tt = new TwoThread();
       tt.start();

       for ( int i = 0; i < 10; i++ ) {
             System.out.println("Main thread");
       }
 }
}
```

A noter : le démarrage d'un thread se fait via la méthode start()
La méthode main() se trouve elle-même dans un thread (le thread principal)

B. Modifier le programme précédent pour que le Thread créé utilise la propriété name de sa classe parente Thread pour faire l'affichage de son nom dans la boucle (faire appel les méthodes setName() et getName() de la classe Thread).

A noter : pour accéder au thread de main il faut utiliser la méthode Thread.currentThread().

C. Modifier le code précédent afin que les 2 threads se passent la main (utiliser la méthode Thread.yield()) après chaque affichage : les 2 affichages doivent être alternés.

D. Ajouter dans la classe la méthode suivante :

```
static void Wait(long milli) {
    System.out.println("pause de "+milli+" ms") ;
    try {
       Thread.sleep(milli);
    } catch (InterruptedException x) {
       // ignorer
    }
 }
```

Remplacer dans le code précédent les appels à yield() par un appel à Wait(200). Quelle est la différence ?

E. Créer à partir de l'exercice précédent une classe permettant de :

- Créer un nombre n de threads, n passé en paramètre de main (utiliser String[] args de main). (max de n étant 10).
- Chaque thread créé possède un nom de la forme « thread x » avec x de 1 à n.
- Faire en sorte que le programme main () se bloque jusqu'à ce que les n threads aient terminé leur exécution (utiliser la méthode join() de Thread).

A noter : par exemple, si main () veut attendre la fin du thread « tt », il appellera : tt.join();

Exercice 2 : compteurs (concurrence)

Modifier le programme de l'exercice précédent (partie E) pour que chaque thread fasse une pause (appel à Wait() ci-dessus) de durée aléatoire entre 0 et 1000 milliseconde avant de rentrer dans la boucle. Chaque thread « compteur » aura le comportement suivant :

- Il compte de 1 à n et affiche chaque nombre (Toto affichera par exemple, "Toto : 3") et il affiche un message du type "*** Toto a fini de compter jusqu'à 10" quand il a fini. Il marque une pause aléatoire entre chaque nombre (de 0 à 5000 millisecondes par exemple).
- Le n est fixé à 10.

Ecrivez la classe compteur et testez-la en lançant plusieurs compteurs qui comptent jusqu'à 10.

Exercice 3 : runnable

Dans les exercices précédents, nous avons créé des threads en héritant de la classe Thread.
Une autre possibilité est d'implémenter l'interface Runnable. Cette interface contient une méthode run() qui sera appelé lorsque le thread démarre (cad après l'appel de sa méthode start()).

Consulter la documentation en ligne de l'interface Runnable pour plus d'informations.

A. Soit la classe suivante :

```
public class Alphabet {
   public void affiche() {
      for (char a = 'A'; a <= 'Z'; a++) {
         System.out.print(a);
         try { Thread.sleep(10); // ms
         } catch (InterruptedException e) {}
      }
      System.out.print("\n");
   }

   public static void main(String args[]) {
      Alphabet A = new Alphabet();
      A.affiche();
   }
}
```

Modifier cette classe en utilisant l'interface Runnable pour que l'affichage se fasse dans un thread séparé. On pourra donc écrire le programme de test suivant :

```
AlphabetThread A1 = new AlphabetThread();
Thread T1 = new Thread(A1);
AlphabetThread A2 = new AlphabetThread();
Thread T2 = new Thread(A2);
T1.start();
T2.start();
```

B. Modifier le code de l'exercice 1 pour que la classe hérite non plus de la classe Thread mais plutôt de l'interface runnable.

Exercice 4 : problème d'accès concurrent

Voici 2 classes **Compte** (correspond à un compte bancaire) et **Operation** (thread qui effectue des opérations sur un compte bancaire).

```
public class Compte {
 private int solde = 0;

 public void ajouter(int somme) {
  solde += somme;
  System.out.print(" ajoute " + somme);
 }

 public void retirer(int somme) {
  solde -= somme;
  System.out.print(" retire " + somme);
 }

 public void operationNulle(int somme) {
  solde += somme;
  System.out.print(" ajoute " + somme);
  solde -= somme;
  System.out.print(" retire " + somme);
 }

 public int getSolde() {
  return solde;
 }
}
```

```
public class Operation extends Thread {
 private Compte compte;

 public Operation(String nom, Compte compte) {
  super(nom);
  this.compte = compte;
 }

  public void run() {
```

```
  while (true) {
    int i = (int) (Math.random() * 10000);
    String nom = getName();
    System.out.print(nom);
//      compte.ajouter(i);
//      compte.retirer(i);
    compte.operationNulle(i);
    int solde = compte.getSolde();
    System.out.print(nom);
    if (solde != 0) {
 System.out.println(nom + ":**solde=" + solde);
 System.exit(1);
    }
  }
 }

 public static void main(String[] args) {
  Compte compte = new Compte();
  for (int i = 0; i < 20; i++) {
    Operation operation = new Operation("" + (char)('A' + i), compte);
    operation.start();
  }
 }
}
```

A. Examinez le code et faites exécuter la classe Opération. Constatez le problème : opération effectue des opérations qui devraient laisser le solde du compte inchangé, et pourtant, après un moment, le solde ne reste pas à 0. Expliquez.

B. Modifiez le code pour empêcher ce problème. (utilisation de synchronized)

C. Dans le code de Operation, remplacez l'opération nulle par 2 opérations ajouter et retirer qui devraient elles aussi laisser le solde du compte à 0 (elles sont en commentaire dans le code). Lancez l'exécution et constatez le problème. Modifiez le code pour que ça marche.

Exercice 5 : Communication inter-thread ou coopération

Considérons la classe Customer suivante :

```
class Customer {
        int amount = 10000;

        synchronized void withdraw(int amount) {
                System.out.println("going to withdraw...");

                if (this.amount < amount) {
                        System.out.println("Less balance; waiting for deposit...");
                        try {
                                wait();
                        } catch (Exception e) {
                        }
                }
```

```
                this.amount -= amount;
                System.out.println("withdraw completed...");
        }

        synchronized void deposit(int amount) {
                System.out.println("going to deposit...");
                this.amount += amount;
                System.out.println("deposit completed... ");
                notify();
        }
}
```

Et la classe Test suivante :

```
class Test {
        public static void main(String args[]) {
                final Customer c = new Customer();

                new Thread() {
                        public void run() {
                                c.withdraw(15000);
                        }
                }.start();

                new Thread() {
                        public void run() {
                                c.deposit(10000);
                        }
                }.start();

        }
}
```

Analyser le code et donner le résultat d'exécution de la classe de Test.