TD #2 – Threads Java

Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, N. de Palma, Université Grenoble Alpes, nov 2020

1. Concept de thread

Rappeler les principes des threads et ce qui les différencie des processus (1 à 2 lignes max).

Dans la suite du TD, on se place dans le contexte des Threads Java.

2. Rappels

Répondez en 1 ou 2 lignes maximum aux questions ci après.

1) Comment lance t'on un programme Java ?
2) Que se passe t'il en termes de création de processus ?
3) Quel point d'entrée est exécuté ?
4) Comment la machine Java trouve t-elle la classe associée au point d'entrée ?
5) Que se passe t'il en termes de threads ?

3. Création et lancement de threads Java

Dans la machine virtuelle Java (JVM), les flots d'exécution (threads) sont modélisés par des objets sous-classe de la super classe *Thread*. Les méthodes publiques définies dans la classe Thread permettent de contrôler les threads (démarrage, priorité, etc.).

Tout objet thread implémente l'interface *java.lang.Runnable*. Cette interface spécifie une méthode *run()* définissant le code exécuté par le thread.

```
...
class PrimeThread extends Thread {
 long minPrime;
 PrimeThread(long minPrime) {
  this.minPrime = minPrime;
 }
 public void run() {
  // compute primes larger than minPrime
  . . .
 }
}
...
```

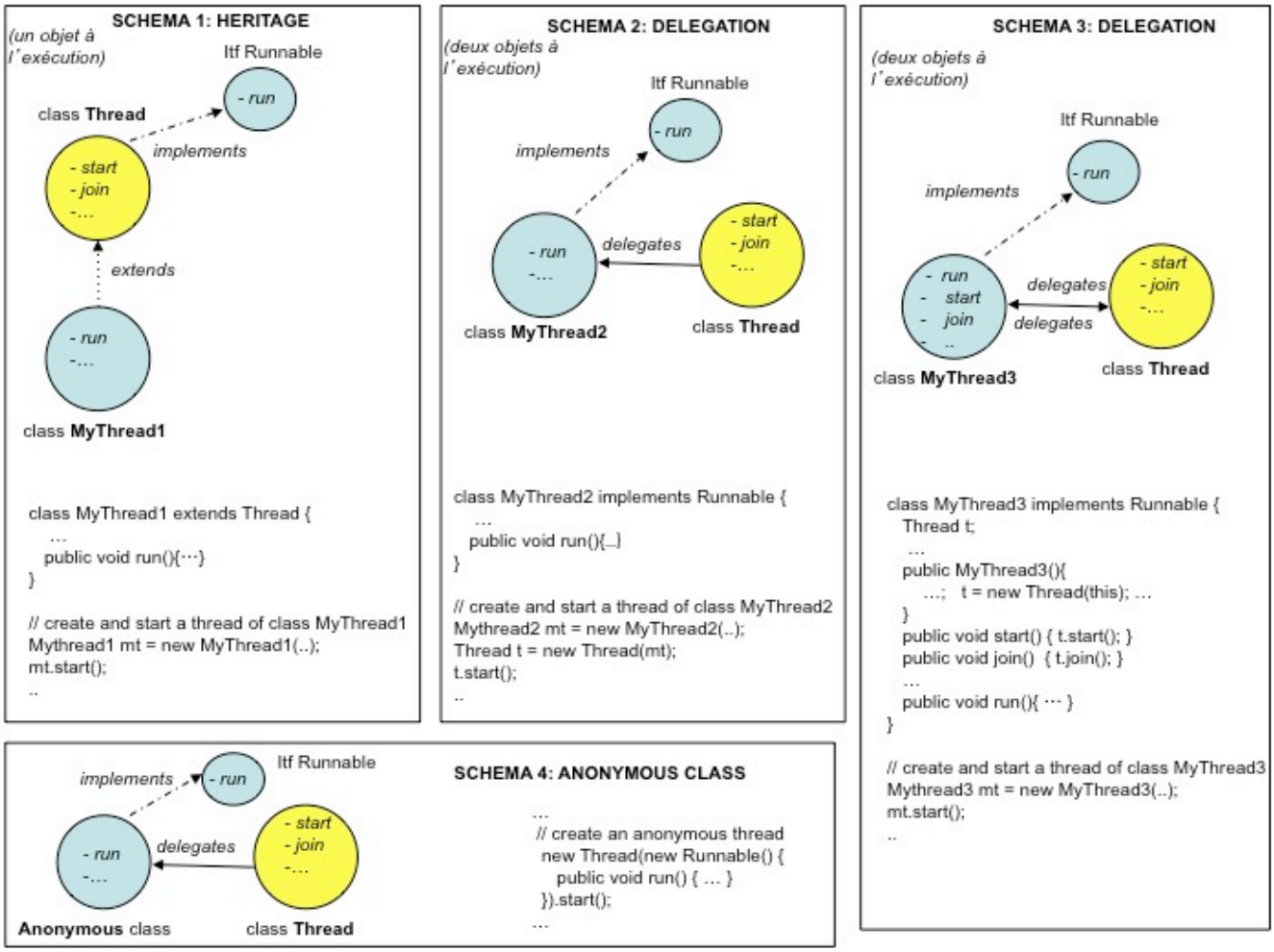
ref : https://docs.oracle.com/

Comme vu en cours, tout thread Java possède un cycle de vie dans lequel il passe par les états suivants :

- NEW: le thread vient d'être créé
- RUNNABLE: le thread est prêt à s'exécuter
- BLOCKED, WAITING: le thread est bloqué (typiquement sur une condition de synchronization)

Il y a principalement trois moyens permettant de définir des threads : par héritage, par délégation, par classe anonyme (voir schéma ci-après). Le schéma par délégation permet de définir une classe de thread qui hérite d'une classe mère quelconque, contrairement au schéma par héritage. Le schéma par classe anonyme est généralement utilisé quant on a besoin de créer et démarrer un thread dont le code est très court. Dans tous les cas, lorsque l'on veut créer et démarrer un thread, il faut instancier la classe définissant son comportement, puis invoquer la méthode *start()* pour le démarrer. La méthode *start()* va placer le thread dans l'état RUNNABLE et l'insérer dans la *ReadyQueue*.

Le schéma ci-après récapitule les principaux patrons de création et démarrage de threads, patrons que l'on trouve dans les programmes existants. Beaucoup de détails ne sont pas présents sur ce schéma par souci de simplicité (par exemple, la classe *Thread* possède plusieurs constructeurs). Nous vous laissons aller voir la Javadoc pour avoir plus de détails.



4. Primitives de contrôle des threads

Nous listons ci-après quelques méthodes de contrôle des threads, certaines statiques, d'autres non statiques. Vous devez vous rappeler clairement de la différence entre les méthodes statiques et non statiques. Si ce n'est pas clair, demandez des explications.

setPriority(..) / getPriority(). Tout thread est associé à une priorité par défaut. Cette priorité peut être redéfinie. La classe Thread définit des constantes au travers de variables de type *static* (NORM_PRIORITY, LOW_PRIORITY, MAX_PRIORITY).

setName(..) / getName(). Tout thread est associé à un nom, qui peut être attribué par la JVM.

getState(). Renvoie l'état courant du thread **(**NEW, RUNNABLE, BLOCKED, WAITING, etc.)

join(). Force le thread courant à attendre la terminaison du thread sur lequel la méthode join() est invoquée (x.join() pour attendre terminaison de x).

static yield(). Si d'autres threads de même priorité que le thread courant sont exécutables, *yield* donne le processeur au thread suivant dans la file des threads exécutables. *Yield()* ne permet pas de lancer un thread de priorité inférieure (à la difference de *sleep()*).

static sleep(..). Endort le thread courant pour un temps donné en millisecondes.

static Thread currentThread(). Retourne la référence du thread courant.

void ***interrupt()***. Si le thread est bloqué, il sort de son été bloqué avec l'exception *InterruptedException*, sinon le marqueur *interrupt-status* est positionné.

static boolean ***interrupted()***.Teste si le marqueur *interrupt-status* est positionné.

void ***setDaemon()***. Associe le rôle de démon au thread, ce qui signifie qu'il ne sera pas pris en compte dans la condition d'arrêt de la JVM.

5. Compréhension d'un petit programme multi-threadé

Commençons par comprendre et commenter le programme Bavard donné ci-après.

```
public  class BavardA extends Thread {
   int no, lifetime;

   public BavardA(int no, int lifetime) {
     this.number = no;
     this.lifetime = lifetime;
     this.start();
   }

   public void run() {
    for (int i = 0; i < lifetime; i++) {
      System.out.println("A " + no + "says  hello " +  i);
        Thread.yield();
     }
     System.out.println("A " + no + " completed\n");
   }
 }
```

```
public  class BavardB implements Runnable {
  int no, lifetime;
 Thread t;

 public BavardB(int no, int lifetime) {
   this.number = no;
   this.lifetime = lifetime;
   t = new Thread(this);
   t.start();
 }

 public void run() {
   for (int i = 0; i < lifetime; i++) {
     System.out.println("B " + no + "says  hello " +  i);
     Thread.yield();
   }
   System.out.println("B " + no + " completed\n");
 }

public void join()throws InterruptedException {
   t.join();
 }
```

```
public class Starter {
   public static void main(String args[]) throws
InterruptedException {
     System.out.println("Number of threads of each kind : ");
     int nthreads = Util.readInt();
     System.out.println("Number of loops : ");
     int lifetime = Util.readInt();

     BavardA  tab_bavardA = new BavardA[nthreads];
     BavardB  tab_bavardB = new BavardB[nthreads] ;
     for (int i = 0; i < nthreads; i++){
       tab_bavardA[i] = new BavardA(i, lifetime);
       tab_bavardB[i] = new BavardB(i+nthreads, lifetime);
     }

     for (int i = 0; i < nthreads; i++){
       tab_bavardA[i].join();
       tab_bavardB[i].join();
     }
     System.out.println("End of program");
   }
}
```

A votre avis, à l'exécution, comment les traces vont-elles s'afficher ?

- est-ce que la trace « End of program» peut s'afficher avant les traces «.. completed» ?
- est-ce que les traces « .. says ..» peuvent s'entrelacer entre les différents threads ?
- est-ce qu'une trace « .. says..» peut être tronquée (ne pas s'afficher complètement) ?

6. Ecriture d'un petit programme multi-threadé

Ecrire un programme Java qui crée un thread par héritage et un autre par délégation. Chaque thread crée affiche son nom, puis s'endort pendant 3 secondes et affiche un message de fin avant de terminer. Le thread principal attend la fin des deux fils et affiche « That's all folk».

Si le temps le permet, exécuter ce programme sous Eclipse.

7. Threads et Interruptions

Tout thread peut envoyer une interruption à un autre thread au travers de la méthode *interrupt()* définie sur la classe *Thread*. Cependant, le thread qui reçoit l'interruption aura un comportement différent selon qu'il est bloqué sur un appel bloquant (ex : *Thread.sleep()*) ou pas :

- si le thread qui reçoit l'interruption est dans un appel bloquant, il en sort avec l'exception *InterruptedException*,
- sinon un booléen est positionné, testable via la méthode *Thread.interrupted()*.

Reprendre le programme précédent, et faire en sorte que le thread principal interrompe l'un des deux threads créés au bout d'environ 1 seconde d'exécution. Au niveau de chaque thread, afficher un message particulier si la sortie du *Sleep()* a été provoquée par une interruption.

Si le temps le permet, exécuter ce programme sous Eclipse.

8. Paralléliser un traitement sur des objets

Etant donné le programme séquentiel ci-après, écrire une (ou plusieurs) versions parallèles, dans lesquelles le traitement effectué sur chaque objet est réalisé par un thread différent.

Si le temps le permet, exécuter votre programme dans Eclipse en mode *debug*. Placez un point d'arrêt au début de votre méthode main, et suivez en pas à pas. Voyez comment Eclipse montre les threads en cours (on peut voir leur état courant et leur pile d'exécution).

```
public ProcessCollection {
     static Collection<Object> c = new
HashSet<>(Arrays.asList("uk","us","france","india","germany");

public void process(Object o) {
          System.out.println(" processing object: " + o);
          Thread.sleep(5000); // simulation d'un traitement sur l'objet o
          System.out.println(" processing object: " + o + "  done");
          }

     public void main(String args[]){
          for (Object o : c)
                    process(o);
          System.out.println("All work done");
          }
}
```

De manière générale, quelles sont les conditions dans lesquelles ce schéma de traitement parallèle s'applique ?

9. Threads démons

Nous avons vu que les threads de type démon sont des threads qui ne sont pas pris en compte dans la condition d'arrêt de la JVM. Autrement dit, une JVM s'arrête lorsque tous les threads qui s'exécutent et qui ne sont pas des démons ont terminé leur exécution.

Reprendre le programme de la question précédente, et considérer l'ajout d'un thread qui affiche des points (…..), au rythme d'un point par demi-seconde, tant que le programme n'est pas terminé. Vous verrez que vous aurez le choix de définir ce thread comme un démon (et

donc de ne pas gérer sa terminaison), ou bien de le définir comme un thread classique, et dans ce cas d'être obligé de l'interrompre quand tous les autres threads ont terminé.

Si le temps le permet, exécuter votre programme dans Eclipse.

10. Extra : Tri parallèle

Nous allons ici considérer la parallélisation d'un programme de tri par fusion (https://fr.wikipedia.org/wiki/Tri_fusion). Les méthodes principales de ce programme sont listées ci-après (la classe intégrant ces méthodes, nommée *MergeSort.java*, vous est donnée). Nous vous demandons de réfléchir à la parallélisation de ce programme et à mettre en œuvre cette parallélisation.
Vous verrez que paralléliser ne signifie pas automatiquement gagner en performances. C'est un ration entre le surplus de temps nécessaire à la manipulation des threads (créations et commutations notamment), et le gain de temps du à l'exécution parallèle sur une machine multi-cœurs.
Concernant le tri de tableau, on commence à obtenir des gains lorsque le tableau à trier est très grand. Pour des petits tableaux, il vaut mieux rester sur du tri mono-thread. Une autre option est d'utiliser les facilités de type *fork-join* fournies par Java, qui a pour objectif d'introduire du parallélisme tout en contrôlant le nombre de threads créés par la JVM, mais nous verrons cela plus tard dans le semestre.

Au final, retenez que le multi-threading est avant tout très utile à la mise en œuvre d'applications concurrentes qui sont composées de différents flôts devant s'exécuter en parallèle pour gérer des fonctionnalités différentes (ex : un thread s'occupant du GUI, un thread de calcul, un thread écoutant les connexions réseaux, etc). Lorsque l'on souhaite paralléliser des traitements pour obtenir des gains de performances, il vaut mieux préalablement évaluer le niveau de multi-threading adéquat (par exemple, en effectuant des simulations).

```
private static void mergeSort(int tab[],int start,int end) {
        if (start!=end){
                int middle=(end+start)/2;
                mergeSort(tab,start,middle);
                mergeSort(tab,middle+1,end);
                merge(tab,start,middle,end);
        }
}

/*
 * Merge the given array
 * Assume it is split in two ordered sub-arrays (tab1: from start to middle, tab2: from middle+1 to end)
 */
private static void merge(int tab[],int start,int middle,int end) {
        //copy the first tab in a temporary array
        int tab1[]=new int[middle-start+1];
        for(int i=start;i<=middle;i++)
                tab1[i-start]=tab[i];

        int start2=middle+1;
        int iter1=start;
        int iter2=start2;
```

```
        for(int i=start;i<=end;i++){
                if (iter1==start2)
                        //we are done with tab1, thus all elts are ordered
                        break;
                else if (iter2==(end+1)) {
                       //we are done with tab2, we just need to add the remaining elts from tab1
                        tab[i]=tab1[iter1-start];
                        iter1++;
                } else if (tab1[iter1-start]<tab[iter2]){
                        //insert the element of tab1
                        tab[i]=tab1[iter1-start];
                        iter1++;
                }else {
                        //insert the element of tab2
                        tab[i]=tab[iter2];
                        iter2++;
                }
        }
}
```