Applications Mobiles

I - Java Multi-thread : les bases

- - - - - - - - - -

- - - - - - - - - -

ENSMA A3-S5 - période B

2023-2024

M. Richard
richardm@ensma.fr

Sommaire

I - Introduction

↪ Processus VS Thread 1/2

Un Processus :

- programme en cours d'exécution.
- mémoire allouée pour chaque processus :
  - segment de code (instructions),
  - segment de données (allocation dynamique),
  - segment de pile (variables locales et adresses de retour des fonctions).
- chaque processus possède en ensemble d'informations :
  - identificateur
  - descripteurs
  - priorités
  - ...
- un processus évolue entre différents états :
  - En exécution (running)
  - Prêt
  - Bloqué

↪ Processus VS Thread 2/2

Un Thread :

- porte aussi le nom de tâche ou de processus léger
- appartient à un processus
- processus peut contenir plusieurs threads
  - au moins 1 : celui existant la fonction `main`
  - les ressources allouées à un processus sont donc partagées entre les threads le composant
    - mémoire,
    - processeur
    - ...
  - partage la même zone mémoire avec les autres threads du processus
    - facilite (ou pas...) la communication entre ceux-ci
  - possède son propre contexte d'exécution
  - possède sa propre pile

↪ Différents types de Thread

Thread :

- Il existe deux types de thread en Java : Classique ou Démon
  - classique : qui correspond à une tâche classique
  - démon : qui correspond à une tâche s'exécutant en fond
    - modification du type de thread avant le lancement : `th.setDeamon(true)`
    - par exemple, le GC de java est un thread de type démon

Groupe de Thread :

- un Thread s'exécute toujours dans un groupe de thread
  - par défaut dans le même groupe que le thread de la méthode `main`
- permet de regrouper les thread ensemble afin de les manipuler plus facilement

II - Mise en œuvre

↪ Implémentation de `Runnable`

Principe :

- Un thread est toujours défini par une méthode particulière d'une classe
  - `void run()`

Deux méthodes :

- Implémenter l'interface `Runnable`
- Hériter de la classe `Thread`

Méthode 1 : Interface `Runnable`

- possède une unique méthode : `public void run()`
- le traitement à effectuer doit donc se trouver dans cette méthode
- si plusieurs threads issus de la même instance sont lancés :
  - partage la même instance, et donc les mêmes variables d'instances
    - possède donc le même état
    - attention au comportement si accès en lecture/écriture aux attributs lors du traitement effectué par le thread
- seule solution si la classe comportant le traitement hérite déjà d'une autre classe

↪ Implémentation de `Runnable` - Exemple 1/2

- classe MonThread :

```
1   package fr.ensma.ia.a3.setr.runnableex1;
2
3   public class MonThread implements Runnable {
4     private int nombre;
5     public MonThread(int n) {
6       nombre = n;
7     }
8     @Override
9     public void run() {
10      for (int i=1; i<=5; i++) {
11        System.out.printf("%s : %d * %d = %d\n",
12        Thread.currentThread().getName(), nombre,
13        i, i*nombre);
14        nombre = i * nombre;
15      }
16    }
17  }
```

↪ Implémentation de `Runnable` - Exemple 2/2

- classe Test :

```
1     package fr.ensma.ia.a3.setr.runnableex1;
2     public class Main {
3       public static void main(String[] args) {
4         MonThread mt = new MonThread(1);
5         Thread th = new Thread(mt);
6         th.start();
7         Thread th2 = new Thread(mt);
8         th2.start();
9         Thread th3 = new Thread(mt);
10        th3.start();
11      }
12    }
```

- résultat :

```
Thread-0 : 1 * 1 = 1
Thread-0 : 1 * 2 = 2
Thread-0 : 2 * 3 = 6
Thread-0 : 6 * 4 = 24
Thread-1 : 1 * 1 = 1
Thread-1 : 24 * 2 = 48
Thread-2 : 1 * 1 = 1
Thread-2 : 48 * 2 = 96
```

```
Thread-2 : 96 * 3 = 288
Thread-2 : 288 * 4 = 1152
Thread-2 : 1152 * 5 = 5760
Thread-1 : 48 * 3 = 144
Thread-1 : 17280 * 4 = 69120
Thread-1 : 69120 * 5 = 345600
Thread-0 : 24 * 5 = 120
```

- Attention, les attributs sont donc bien communs aux deux threads lancés à partir du même objet implémentant `Runnable`
- pour fonctionner avec des attributs propres, il faut instancier de nouveaux objets.

↪ Héritage de `Thread`

- consiste à étendre la classe `Thread`
- en redéfinissant la méthode `run()`

Exemple :

- classe `MonThread` :

```
1   package fr.ensma.ia.a3.setr.extendsex1;
2
3   public class MonThread extends Thread {
4     private int nombre;
5     public MonThread(int n) {
6       nombre = n;
7     }
8     @Override
9     public void run() {
10      for (int i=1; i<=5; i++) {
11        System.out.printf("%s : %d * %d = %d\n",
12        Thread.currentThread().getName(), nombre,
13        i, i*nombre);
14        nombre = i * nombre;
15      }
16    }
17  }
```

↪ Exemple

- classe Test :

```
1     package fr.ensma.ia.a3.setr.extendsex1;
2     public class Main {
3       public static void main(String[] args) {
4         MonThread mt = new MonThread(1); mt.start();
5         MonThread mt2 = new MonThread(1); mt2.start();
6         MonThread mt3 = new MonThread(1);
7         mt3.start();
8       }
9     }
```

- résultat :

```
Thread-0 : 1 * 1 = 1
Thread-0 : 1 * 2 = 2
Thread-0 : 2 * 3 = 6
Thread-0 : 6 * 4 = 24
Thread-0 : 24 * 5 = 120
Thread-2 : 1 * 1 = 1
Thread-2 : 1 * 2 = 2
Thread-2 : 2 * 3 = 6
```

```
Thread-2 : 6 * 4 = 24
Thread-1 : 1 * 1 = 1
Thread-1 : 1 * 2 = 2
Thread-1 : 2 * 3 = 6
Thread-2 : 24 * 5 = 120
Thread-1 : 6 * 4 = 24
Thread-1 : 24 * 5 = 120
```

  - Les deux thread possèdent donc leur propre état, i.e. des valeurs d'attributs différentes
  - Si l'on déclare un attribut de classe et non d'instance, celui-ci sera commun aux différents thread
    - facilite la communication entre Thread, mais attention aux accès concurrents...
    - communication de type ≪ Tableau Noir ≫

↪ Les bases

Un thread possède un certain nombre d'attributs permettant son identification ou le contrôle de son fonctionnement. Ces attributs sont les suivants :

- `ID` :
  - identifiant unique pour chaque thread
- `Name` :
  - permet de stocker le nom d'un thread
- `Priority` :
  - stocke la priorité d'un thread. c'est un entier compris entre 1 et 10. 1 est la plus faible priorité et 10 la priorité la plus élevée. Impacte l'ordonnancement des différents thread.
- `Status` :
  - stocke l'état du thread durant son exécution.

↪ État d'un Thread

`getState()` :

- retourne l'une des valeurs suivantes :
  - `NEW` : pas encore démarré
  - `RUNNABLE` : prêt à être exécuté
  - `TERMINATED` : terminé
  - `TIME_WAITING` : attente connue (resultant d'un `sleep()`)
  - `WAITING` : attente non connue
  - `BLOCKED` : en attente d'un verrou
- attention, à n'utiliser que pour la surveillance de l'état d'un thread, mais pas pour mettre en œuvre une synchronisation
  - En effet, l'état peut avoir changé au moment de l'exécution effective de la synchronisation...

`isAlive()` :

- retourne `true` si le thread a été démarré et n'est pas encore terminé

`isInterrupted()` :

- retourne `true` si le thread a été interrompu

`isDaemon()` :

- retourne `true` si le thread est un démon

III - Exécution & Priorités

↪ Interruption d'un Thread depuis un autre thread

- Interrompre l'exécution d'un thread depuis un autre thread

Principe :

- Deux méthodes peuvent être employées en fonction de la complexité (de l'algorithme et en temps) de la méthode `run`
- Méthode 1 : pour les algorithmes simples
  - Dans la méthode `run`, tester l'état du thread Th_1 par la méthode `isInterrupted()`. Si le booléen est vrai, stopper l'algorithme (avec un `return` par exemple...).
  - Dans le thread Th_2 souhaitant interrompre Th_1, utiliser la méthode `interrupt()`
- Méthode 2 : pour les cas plus complexes
  - Utiliser un mécanisme d'exception, comme le fait l'API Java
  - Plus précisément, l'exception `InterruptedException` est utilisée.
  - La méthode `run` du thread Th_1 écoute et traite cette exception si elle est déclenchée. La levée de l'exception provoque l'arrêt de la méthode `run`.
  - la méthode de traitement complexe teste si une demande d'interruption à été faite sur le thread par l'intermédiaire de la méthode `isInterrupted()`. Dans ce cas, elle déclenche une nouvelle exception de type `InterruptedException`.
  - Dans le thread Th_2 souhaitant interrompre Th_1, utiliser la méthode `interrupt()`

↪ Endormir un thread

Objectif :

- Mettre un thread en pause pendant une certaine durée

Solutions :

- Utilisation de la méthode `yeld()`
  - force le thread à ≪ relacher ≫ le processeur
  - le thread passe donc en attente pendant... une certaine durée
  - cette méthode n'est pas recommandée.
- utilisation de la méthode `sleep(t)`
  - suspend le thread durant une durée t
    - cette méthode peut lever une exception de type `InterruptedException`.
  - Exemple d'utilisation :

```
1       ...
2       try{
3         Thread.sleep(500);
4       }catch(InterruptedException ex){
5         //Traitement en cas d'exception
6       }
7       ...
```

  - Remarque : la méthode `sleep()` est à la fois une méthode de classe et une méthode d'instance.

↪ Attente de terminaison 1/2

Il est possible pour un Thread d'attendre la fin d'un autre Thread :

- méthode `join()`
  - attend que le thread sur lequel est appliquée la méthode se termine
- méthode `join(temps_ms)`
  - attend pendant temps_ms ms que le thread sur lequel est appliquée la méthode se termine
  - si le temps est écoulé avant la fin du thread, l'attente est annulée et le code se poursuit

Exemple :

```
1   public class UnThread extends Thread{
2     @Override
3     public void run(){
4       long reveil = System.currentTimeMillis();
5       while(System.currentTimeMillis()<=(reveil + (1000*5))){
6         System.out.printf("UnThread en exécution ...\n");
7         try{
8           Thread.sleep(500);
9         }catch(InterruptedException ex){
10          System.out.printf("Oupps : Thread interrompu ...");
11        }
12      }
13    }
14  }
```

↪ Attente de terminaison 2/2

Exemple (suite) :

```
1   public class Test {
2     public static void main(String Args []){
3       UnThread th = new UnThread();
4       th.start();
5       try {
6         th.join(2000);
7       } catch (InterruptedException e) {
8         e.printStackTrace();
9       }
10      if(!th.isAlive())
11      System.out.printf("Tous les Threads sont terminés\n");
12      else
13      System.out.printf("Tous les Threads ne sont pas terminés\n");
14    }
15  }
```

Résultat :

```
UnThread en exécution ...
UnThread en exécution ...
UnThread en exécution ...
UnThread en exécution ...
UnThread en exécution ...
UnThread en exécution ...
UnThread en exécution ...
UnThread en exécution ...
Tous les Threads ne sont pas terminés
UnThread en exécution ...
UnThread en exécution ...
```

↪ Ordonnancement 1/2

Principe :

- Java utilise une politique d'ordonnancement à priorité fixe
  - le thread de plus forte priorité obtient le processeur
  - si deux thread ont la même priorité, le critère d'ordonnancement est alors FIFO
- trois niveaux de priorité sont définis :
  - `Thread.MIN_PRIORITY` : vaut 1
  - `Thread.NORMAL_PRIORITY` : vaut 5
  - `Thread.MAX_PRIORITY` : vaut 10
- lors de sa création, un thread hérite de la priorité du thread qui l'a créé.
- Obtenir la priorité d'un thread :
  - `getPriority()`
- Modifier la priorité d'un thread :
  - `setPriority(int prio)`
  - `prio` doit toujours être compris dans l'intervalle `[MIN_PRIORITY,MAX_PRIORITY]`

↪ Ordonnancement 2/2

Relation Java/OS :

- l'ordonnancement dépend du système d'exploitation sur lequel s'exécute les threads
- la majorité des systèmes proposent deux politiques d'ordonnancement :
  - ordonnancement préemptif à priorité
  - round-robin
  - round-robin + priorité
- ainsi, le choix des priorités des threads java n'est donc pas forcément une garantie, surtout si le système d'exploitation utilise une politique basée sur un round-robin

↪ Exemple 1/2

Le thread peu prioritaire :

```
1   public class UnThreadPasPrioritaire extends Thread {
2     @Override
3     public void run(){
4       long reveil = System.currentTimeMillis();
5       while(System.currentTimeMillis()<=(reveil + (1000*5))){
6         System.out.printf("UnThreadPasPrioritaire en exécution ...\n");
7         for(int i=0;i<2000000000;i++);
8         try{ Thread.sleep(500);
9         }catch(InterruptedException ex){
10          System.out.printf("Oupps : Thread interrompu ...");}
11      }
12  }}
```

Le thread plus prioritaire :

```
1   public class UnThreadTresPrioritaire extends Thread{
2     @Override
3     public void run(){
4       long reveil = System.currentTimeMillis();
5       while(System.currentTimeMillis()<=(reveil + (1000*5))){
6         System.out.printf("UnThreadTresPrioritaire en exécution ...\n");
7         try{ Thread.sleep(500);
8         }catch(InterruptedException ex){
9           System.out.printf("Oupps : Thread interrompu ...");}
10      }
11  }}
```

↪ Exemple 2/2

Réglage de priorité & Lancement :

```
1   public class Test {
2     public static void main(String Args []){
3       UnThreadPasPrioritaire thpp = new UnThreadPasPrioritaire();
4       thpp.setPriority(2);
5       UnThreadTresPrioritaire thtp = new UnThreadTresPrioritaire();
6       thtp.setPriority(8);
7       thpp.start(); thtp.start();
8       try { thpp.join();
9       } catch (InterruptedException e) { e.printStackTrace();}
10      System.out.printf("\n Fin de l'appli =\n");
11  }}
```

Résultat :

```
0 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
16 [Thread-0] INFO ...a3.setr.thread  - UnThreadPasPrioritaire en exécution
516 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
1016 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
1517 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
2017 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
2299 [Thread-0] INFO ...a3.setr.thread  - UnThreadPasPrioritaire en exécution
2518 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
3018 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
3519 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
4019 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
4520 [Thread-1] INFO ...a3.setr.thread  - UnThreadTres Prioritaire en exécution
4551 [Thread-0] INFO ...a3.setr.thread  - UnThreadPasPrioritaire en exécution
6754 [main] INFO ...a3.setr.thread  - = Fin de l'appli =
```

IV - Ordonnanceur

↪ Principe 1/2

- Exécuter plusieurs tâches planifiées par un thread spécifique exécuté en arrière plan
- Les tâches peuvent être planifiées pour s'exécuter une seule fois ou plusieurs fois à rythme régulier avec délai ou non
- pas de processus spécifique d'arrêt

Mise en œuvre :

- Utilisation des classes :
  - `Timer` permettant la planification
  - `TimerTask` permettant de construire des tâches *planifiables*

↪ Principe 2/2

La classe `Timer` *propose les méthodes de planification suivantes :*

- `public void schedule(TimerTask tache, long delai)`
  - planifie l'exécution de la tâche tache après le délai.
- `public void schedule(TimerTask tache, Date instant)`
  - planifie l'exécution de la tâche à la date spécifiée.
- `public void schedule(TimeTask tache, long delai,long periode)`
  - planifie l'exécution périodique à partir du délai spécifié.
- `public void schedule(TimerTask, Date PremierInstant, long periode)`
  - planifie l'exécution périodique à partir de l'instant spécifié.
- `public void scheduleAtFixedRate(TimerTask, long delai, long periode)`
  - Idem mais le rythme d'exécution doit être fixe.
- `public void scheduleAtFixedRate(TimerTask,Date premierInstant,long periode)`
  - Idem mais le rythme d'exécution doit être fixe.

La classe `TimerTask` *:*

- modélise une tâche dont une thread Timer aura à planifier l'exécution.
- Le corps d'une tâche doit être spécifié par redéfinition de la méthode run()
- Une tâche peut connaître la date la plus récente à laquelle elle a été programmée en utilisant
  - `public long scheduledExecutionTime()` qui retourne l'instant le plus proche

↪ Exemple 1/2

- Une tâche périodique de période 1s affichant l'instant auquel elle s'exécute
- La classe permettant la planification :

```
1    package fr.ensma.ia.a3.setr.tacheperiodique;
2    import java.util.Timer;
3    public class TimerDemon {
4      Timer monTimer;
5      public TimerDemon() {
6        monTimer = new Timer();
7        monTimer.schedule(new PeriodicTask(), 0, 1*1000);
8      }
9    }
```

- La tâche à exécuter périodiquement :

```
1    package fr.ensma.ia.a3.setr.tacheperiodique;
2    import java.util.GregorianCalendar;
3    import java.util.TimerTask;
4    public class PeriodicTask extends TimerTask{
5      @Override
6      public void run() {
7        System.out.println(Thread.currentThread().getName() +
8        " s'execute a la date " +
9        new GregorianCalendar().getTimeInMillis());
10     }
11   }
```

↪ Exemple 2/2

- La classe de lancement :

```
1     package fr.ensma.ia.a3.setr.tacheperiodique;
2
3     public class Main {
4       public static void main(String[] args) {
5         new TimerDemon();
6       }
7     }
```

- Le résultat de l'exécution :

```
Timer-0 s'execute a la date 1360074468443
Timer-0 s'execute a la date 1360074469433
Timer-0 s'execute a la date 1360074470433
Timer-0 s'execute a la date 1360074471434
Timer-0 s'execute a la date 1360074472436
Timer-0 s'execute a la date 1360074473436
Timer-0 s'execute a la date 1360074474437
Timer-0 s'execute a la date 1360074475438
Timer-0 s'execute a la date 1360074476438
Timer-0 s'execute a la date 1360074477438
Timer-0 s'execute a la date 1360074478439
```

- le thread de planification peut-être stoppé par l'appel de la méthode :
  - `cancel()` qui provoque l'arrêt de toutes les tâches planifiées.