Licence 2 — I4b
Semaines du 07/03 au 19/03/2005
TP 1 & 2 • Création de processus : threads

Java fournit en standard un certain nombre de classes pour gérer, les fenêtres graphiques, les fichiers, le réseau, les processus etc. Cette dernière fonctionnalité est réalisée au moyen de la notion de thread (présentée dans le chapitre III du cours). Un thread est un processus léger qui partage avec son parent l'ensemble de ses données. L'utilisation de plusieurs threads dans un programme permet :

- **d'éviter les engorgements :** avec un seul thread, le programme doit s'arrêter complètement pour attendre des processus lents commne la gestion des accès aux disques durs ou les communications avec d'autres machines. Le CPU est inactif jusqu'à la fin de ces processus. Avec plusieurs threads, l'application peut poursuivre l'exécution des autres threads pendant que le thread qui attend le resultat d'un processus lent est bloqué ;
- **d'organiser le comportement du programme :** avec plusieurs threads on peut decomposer le programme en plusieurs sections indépendants et assigner différentes priorités à chaque thread afin d'attribuer davantage de temps CPU aux tâches critiques ;
- **d'utiliser plusieurs processeurs.**

Dans ce TP on se propose d'étudier la création de threads, l'affectation de priorité et les problèmes d'echange et de partage de données entre plusieurs threads (concurrence).

Interactions de Java et du système d'exploitation

Java est une technologie définie par SUN Microsystems à la fin de l'année 1995. Java est plus qu'un langage de programmation, on parle courament de plateforme Java. La plateforme Java s'articule autour de trois composants essentiels :

1. les spécifications du langage de programmation Java ;
2. un ensemble d'interfaces/bibliothèques de programmation d'application (API) ;
3. une spécification de machine virtuelle.

La langage Java fournit, en plus des aspects traditionnels les langages de programmation orientés objets, un support de haut niveau pour la programmation réseau et la communication entre objets distribués. De plus c'est un langage multi-threads au sens ou un programme Java peut avoir plusieurs flots de contrôle.

L'API Java contient une API de base et un extension standard. L'API de bae fournit un support simple du langage pour le graphique, les entrées sorties, les fonctions courantes et le réseau (`java.lang`, `java.awt`, `java.io`, `java.net`, etc.). Les extensions incluent des services spécifique pour le support d'applications d'entreprise (sécurité, interfaces bases de données etc.). Comme le langage évolue, de nombreux packages faissant partie des extensions sont inclus dans l'API de base.

La machine virtuelle JVM est une spécification de machine abstraite constituée d'un chargeur de classe et d'un interpréteur Java capable d'exécuter le code objet (byte-code) produit par le compilateur `javac`. Le code objet est indépendant de la plateforme matérielle. Le chargeur de classes utilise les fichiers `.class` et les API pour permettre l'exécution des programmes par l'interpréteur. L'interprèteur peut être un logiciel qui interprète instruction après instruction les commandes du code objet ou un compilateur à la volée qui traduit en code machine le code objet (JIT Just-In-Time). Dans certains cas l'interprèteur Java peut être implanté dans une puce spécifique. Afin de pouvoir exécuter un programme Java (`.class`) sur une architecture il faut disposer d'une machine virtuelle spécifique à cette architecture. Le problème de portabilité des programme est donc simplifié puisqu'il suffit de porter la machine virtuelle (quelques Mo). La machine virtuelle réalise donc une

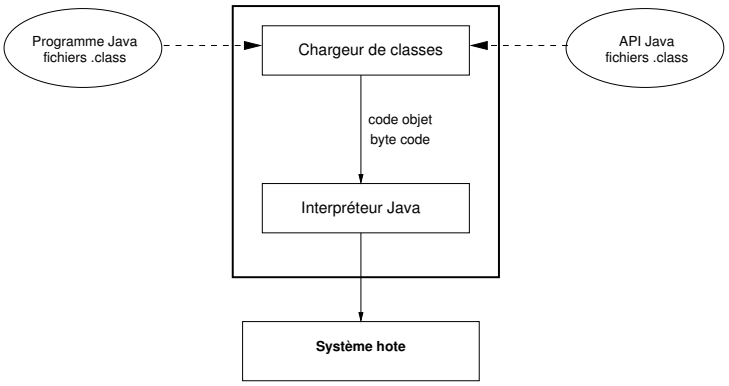

FIG. 1 – Principe de fonctionnement de la JVM

abstraction de système d'exploitation. Ainsi on peut par généralisation considérer la JVM comme un système d'eploitation. Le processeur sur lequel est définie la JVM est un processeur abstrait capable d'exécuter du byte-code.

Un instance de la JVM est créee chaque fois qu'une application Java est lancée. L'instance de la JVM commence son exécution lorsque la méthode `main()` est invoquée. C'est également vrai pour les applets qui *a priori* n'ont pas de méthode main puisque celle ci est gérée par le navigateur. Si on lance n programmes Java sur une machine, il y aura n machines virtuelles en exécution.

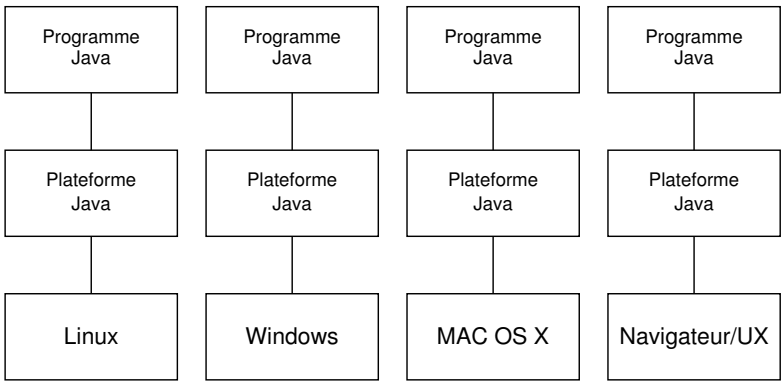

FIG. 2 – Portabilité de la JVM

L'environnement Java que vous allez utiliser en TP est composé du compilateur Java (`javac`) et de la machine virtuelle (`java`). Le principe de développement du programme est décrit à la figure 1.

Exercice 1. Création de threads

Au démarrage un seul thread est crée. Pour en créer un nouveau, il faut créer un nouvel objet de la classe `Thread` puis le démarrer c'est-à-dire lui demander d'exécuter une portion de code (méthode). Il existe deux stratégies. Soit le code à exécuter est spécifié dans une classe qui hérite de la classe `Thread`, soit il est spécifié dans une classe qui implemente l'interface `Runnable`. Dans la seconde stratégie, un objet sera passé en paramètre lors de la création du nouvel objet de la classe `Thread`. La première solution a l'avantage de ne manipuler qu'un seul objet mais elle a l'inconvénient d'interdire tout nouvel héritage.

Pour créer la nouvelle thread il faut utiliser l'un des constructeurs de la classe `Thread`. Les plus courament utilisés sont :

- `public Thread() ;`
- `public Thread(String) ;`
- `public Thread(Runnable, String) ;`
- `public Thread(Runnable) ;`

- `public Thread(ThreadGroup, String);`
- `public Thread(ThreadGroup, Runnable);`
- `public Thread(ThreadGroup, Runnable, String);`

Le paramètre `ThreadGroup` permet de regrouper un ensemble de thread est de leur donner des propriétés communes ou de leur appliquer, à tous, une même méthode. Une fois l'objet thread créé au moyen d'un des constructeurs, il faut l'activer (le démarrer) par un appel à la méthode `start()` qui appellera la méthode `public void run()` de l'objet contenant le code à exécuter. Si la méthode `run()` est appelée directement, le code de cette méthode est exécuté dans le thread courant.

```
1  class monThread extends Thread
2  {
3     public void run() {
4          for (int i=1 ; i<5 ; i++)
5                  System.out.println("je suis le thread "+getName()+" i="+i);
6     }
7  }
8
9  public class exo1
10 {
11    public static void main(String args[]) {
12         Thread th1 = new monThread();
13         Thread th2 = new monThread();
14         th1.start();
15         th2.start();
16         System.out.println("Je suis le thread principal");
17    }
18 }
```

Listing 1 – Création de deux threads

1. En vous inspirant du code donné, écrire deux programmes pour créer des threads au moyen des deux méthodes évoquées.
2. Modifier les programmes pour créer 5 threads dans une boucle.
3. Taper le programme et l'exécuter.
4. Pourquoi ce programme ne met pas en évidence le fonctionnement multi-tâche du système d'exploitation ? Modifier le programme afin de faire apparaître "l'effet multi-tâche".
5. Quel peut être l'inconvénient de la méthode de création et de démarrage des threads ?

Exercice 2. Thread et priorité

Pour chaque thread crée il est possible de lui associer une priorité au moyen de méthodes définies dans la classe java.lang.Thread.

Un objet de la classe Thread permet de contrôler un processus léger qui peut être actif, suspendu ou arrêté.

Au lancement d'un programme une seule thread s'exécute, c'est le thread initial. On peut le contrôler via la méthode de classe : `public static Thread currentThread();` Il est ensuite possible de modifier les attributs du thread en le manipulant via les méthodes d'instance de l'objet retourné.

```
1  class exo2{
2   public static void main(String[] args) throws Exception{
3          Thread threadInitial = Thread.currentThread();
4          // Donner un nom au thread
5          threadInitial.setName("Mon thread");
6          // Afficher le nom du thread
```

```
7           System.out.println(threadInitial);
8           // Faire dormir la thread 1 sec
9           Thread.sleep(1000);
10          System.out.println("fin");
11      }
12  }
```

Il est possible de changer la priorité d'une thread afin qu'il est une priorité particulière pour accéder au processeur. La thread de plus forte priorité accède plus souvent au processeur. Par défaut, une thread a la priorité de sa mère. Pour changer la priorité d'une thread la méthode `public void setPriority (int prio)` est disponible et pour visualiser la priorité on utilise `public int getPriority ()`.

De plus, il existe des constantes prédéfinies pour les valeurs des priorités :

- `public final static int MAX_PRIORITY;`
- `public final static int MIN_PRIORITY;`
- `public final static int NORM_PRIORITY;`

Il n'est pas possible de sortir de ces bornes sous peine de recevoir une exception .

Modifier le programme de l'exercice 1 afin de donner différentes priorités aux deux threads. Notez vos observations pour chaque type de priorité testée.

Exercice 4. Illustration

Réaliser un programme qui simule une course de threads. Chaque thread démarre et compte jusqu'a 1000. Dans la boucle simulant le travail du thread, implanter une attente aléatoire. Le premier thread arrivé a 1000 est déclaré gagant.

Exercice 5. Concurrence

Reprendre l'exemple du cours sur la banque et les distributeurs de billets, afin de mettre en évidence le problème de concurrence.

1. Créer un programme qui lance deux threads qui vont retirer 1000 fois 10 pour l'un et 1000 fois 50 pour l'autre, sur un compte qui comporte 800 000. Quel doit être le solde du compte ? Qu'observez vous ? Ce résultat est il toujours le même ?
2. Modifier le programme afin de mettre en évidence un problème de concurrence.
3. En utilisant le mécanisme de moniteur, modifier votre programme afin qu'à un instant donné, un seul thread puisse accéder au contenu du compte. Vérifier que son éxécution est correcte.

Exercice 5. Un exemple réel

On suppose qu'on dispose d'une machine bi-processeur. On veut créer un programme qui permet de calculer les nombres premier compris entre 1 et 50000. Pour exploiter les deux processeurs on utilise deux threads.

1. Réaliser une première verison du programme en attibutant l'exploration des l'intervalles 1 - 25000 à un thread et 25001 - 50000 à l'autre thread. Quel est l'inconvient de ce programme ?
2. Réaliser un deuxième version comportant un index qui indique la progression du calcul. Chaque thread vient consulter l'index pour connaître le nombre qu'il doit tester. L'algorithme en pseudo-code est le suivant :

```
pour i:=1 jusqu'a 50000
 nb=lire la valeur de l'index
 incrémenter l'index
 tester si nb est premier
fin pour
```

Indications

La JVM exécute les threads jusqu'à ce que :

- la méthode `exit()` soit appelée ;
- toutes les threads qui n'ont pas été marqués Deamon soit terminés.

Un thread se termine lorsque la méthode `run()` se termine ou lors de l'appel à la méthode `stop()` sur une référence au thread. Il est possible de suspendre momentanément l'exécution d'un thread au moyen de la méthode `suspend()` et de reprendre l'exécution au moyen de la méthode `resume()`. La méthode `destroy()` permet de détruire un thread sans aucune possibilité de réaction et en ne libérant aucun moniteur. La méthode `yield()` permet de rendre la main. Toutes les opérations ne sont possibles que si le thread appelant les méthodes a le droit d'accéder au thread dont on veut modifier l'état. Ceci n'est vrai que si les thread appartiennent au même ThreadGroup.

Chaque thread s'exécutant dans le même espace d'adressage, il est nécessaire d'avoir un mécanisme d'exclusion mutuelle entre threads lors des accès à la mémoire. Java implante le mécanisme de moniteur qui assure que quand une thread rentre dans une portion de code gérée par un moniteur aucune autre thread ne peut y pénétrer.

Java garantit atomicité de l'accès et de l'affectation des types primitifs, sauf les long et les double. Ainsi deux threads qui modifient de façon concurrente une variable de type double peuvent entraîner des résultats incohérents. De même pour des objets.

Pour éviter ces problèmes d'incohérences Java propose un mécanisme d'exclusion mutuelle entre deux threads. Pour cela Java propose le mot clef synchronized qui s'applique à une portion de code relativement à un objet particulier. Pendant l'exécution d'un portion de code synchronisée par une thread A , toute autre thread essayant d'exécuter une portion de code synchronisée sur le même objet est suspendue. Une fois que l'exécution de la portion de code synchronisée est terminée par la thread A , une thread en attente et une seule est activée pour exécuter sa portion de code synchronisée. Ce mécanisme est un mécanisme de moniteur. Il peut y avoir un moniteur associé à chaque objet.

Deux constructions sont disponibles :

- synchronized est utilisé comme modifieur d'une méthode. Le moniteur est alors associé à l'objet courant et s'applique au code de la méthode ;
- `la construction synchronized(o){...}`. Le moniteur est associé à l'objet o et s'applique au bloc associé à la construction.