Traîtement distribué des données

Master 1 ISD

Problèmes de concurrence

Rappels Java, *Threads*, *mmap*

Kim.Nguyen@lri.fr

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Plan

`final`

Que signifie le mot clé **`final`** sur ?

une classe :

On ne peut pas dériver la classe (ex : **`String`**)

une méthode :

On ne peut pas redéfinir (*override*) la méthode. Que signifie redéfinir ?

- définir une méthode avec exactement le même prototype dans une classe dérivée.

un attribut :

L'attribut peut être initialisé au plus tard dans le constructeur :

- si son type est primitif (**`boolean`**, **`char`**, **`int`**, ...) alors l'attribut est une constante
- si l'attribut est un objet, alors c'est une référence constante vers cet objet! (l'état interne de cet objet peut toujours être modifié, cf. les tableaux)

une variable locale ou un paramètre :

on ne peut pas modifier la valeur de la variable après initialisation

Utilité

◆ Contrôle du code utilisateur. Ex: beaucoup de composants de Java (y compris de sécurité) utilisent la classe **`String`**. Si on pouvait dériver la classe et changer ses méthodes (par exemple **`equals`**) le code utilisant **`String`** n'aurait plus aucune garantie (pour les classes et les méthodes **`final`**)

◆ Sûreté et documentation : un attribut ou une variable déclarés **`final`** ne peuvent être modifiés. Le programmeur déclare que ce sont des constantes

◆ Efficacité : un attribut **`final`** étant constant, l'optimiseur de la JVM connait sa valeur et faire des optimisations. Une classe ou une méthode **`final`** ne pouvant pas être dérivée/redéfinie, l'optimiseur peut déterminer son code de manière certaine

finally

(malgré la similitude de nom, ça n'a rien à voir avec **`final`**.)

Opération courante : on souhaite écrire dans un fichier, puis le refermer. Cependant, il faut aussi traiter le cas des exceptions correctement :

```
PrintStream out = …;
try {

  out.println(…);
  out.close();

} catch (IOException e) { out.close(); }
```

Le code de **`out.close()`** est dupliqué. Si le code était plus important, ce ne serait pas maintenable.

finally

On peut utiliser la clause **finally** dans un **try/catch** pour écrire du code qui sera exécuté obligatoirement :

- à la fin du bloc, si aucune exception n'est levée
- après le code du **catch** si une exception est rattrapée
- avant de quitter le bloc (ou la méthode) si l'exception n'est pas rattrapée

```
PrintStream out = …;
try {

    out.println(…);

} catch (IOException e) {
} finally { out.close(); }
```

On utilise cette construction lorsque l'on souhaite libérer une ressource avant de poursuivre le programme, quelle que soit la manière dont il se poursuit (normalement, exception rattrapée, exception propagée).

static

Que signifie le mot clé `static` sur ?

une méthode :

la méthode n'est pas appliquée à un objet :

- Elle n'a pas accès aux attributs non statiques de l'objet

un attribut :

L'attribut est une propriété de la classe et non pas d'un objet particulier. Il ne peut être initialisé que :

- au moment de sa déclaration
- dans un bloc statique

```
class A {
static HashMap<Integer, String> map;
static {
   map = new HashMap<>();
   map.put(new Integer(1), "A");
   map.put(new Integer(2), "B");
}
…
```

Utilité

- Partager un même objet (par exemple un compteur) entre tous les objets d'une même classe
- Similaire à une variable globale (si l'attribut est public static)
- Similaire à une constante globale (si l'attribut est public static final

Attention les champs statiques sont initialisés au chargement de la classe par la JVM, donc bien avant que le moindre code utilisateur ait été exécuté.

Quizz sur les Interfaces

Soit une interface :

```
interface I extends J {
   public R1 meth(T1 t1, …, Tn tn) throws E1, …, Ek
}
```

on considère une classe **`C`** qui implémente **`I`**

- **`C`** doit implémenter **`meth`** ? oui
- **`C`** doit implémenter toutes les méthodes de **`J`** ? oui
- **`C`** peut changer la visibilité de **`meth`** (**`public`** en **`private`**) ? non
- **`C`** peut définir **`meth`** avec des types (arguments ou retour) différents ? non (i.e. il doit au moins définir **`meth`** avec le même prototype. Elle peut par contre la surcharger avec d'autres versions)
- **`C`** peut définir **`meth`** avec plus d'exceptions ? non
- **`C`** peut définir **`meth`** avec moins d'exceptions ? oui
- **`C`** peut définir **`meth`** avec l'attribut **`synchronized`** ? oui
- **`I`** peut déclarer **`meth`** avec l'attribut **`synchronized`** ? non

Plan

> Concurrency is about *dealing* with lots of things at once. Parallelism is about *doing* lots of things at once.
>
> Rob Pike

- *Concurrence* : niveau logique. Plusieurs tâches (qui n'ont peut être rien à voir entre elles) *ont conscience les unes des autres* (et interfèrent entre elles).

- *Parallélisme* : plusieurs opérations ont lieu *simultanément*

Exemples

- Les différents processus d'un OS multi-tâche s'exécutent *de manière concurrente* :
  - Les programmes sont a priori indépendants
  - Il doivent se *synchroniser* pour l'accès aux ressources partagées
  - Existe sur des processeurs scalaires (simple) sans parallélisme
- Soit le programme C suivant :

```
void f(int x, int y, int* p) {
  int z = *p;
  x = x + z;
  y = y + z;
  *p = x + y;
}
```

lors de l'évaluation de `f` plusieurs (sous-)opérations peuvent être effectuées en *parallèle* sur un processeur super-scalaire moderne (lesquelles) ?

- Les différents processus d'un OS multi-tâche s'exécutent en *parallèle* sur un processeur multi-cœur

Plan

Définitions

Un *thread* (tâche ou fil d'exécution en français) est la plus petite séquence d'instructions manipulable par *l'ordonnanceur* d'un système d'exploitation. Chaque *thread* possède sa propre pile d'exécution. Le tas est peut être partiellement partagé.

Un *processus* est l'exécution d'un programme. Il est usuellement composé de plusieurs *threads*. Deux processus ne partagent pas d'espace mémoire (et donc pas de variables).

Threads en Java

En Java, on peut programmer un *thread* (sous-processus) en créant une classe qui hérite de la classe **`Thread`**. Une telle classe doit redéfinir la méthode **`public void run()`** pour y placer le code à éxécuter en parallèle :

```
class MonThread extends Thread {
  MonThread() {
    // constructeur
  }

  @Override
  public void run()  {
    //Code à exécuter en parallèle
  }

}
```

Attention, la méthode **`run()`** n'a pas vocation à être appelée directement par le programmeur. Elle est appelée automatiquement par la JVM lorsque le *thread* démarre.

Création de *threads*

```
public class MonProgramme {
    …

    public static void main() {
      MonThread t = new MonThread();
      t.start();
      …
    }
}
```

La méthode **`start()`** rend la main immédiatement. La méthode **`run()`** du *thread* démarré est éxécutée. Si on appelle **`start()`** plus d'une fois, l'exception **`IllegalThreadStateException`** est levée.

Gestion des *threads*

Les méthodes suivantes permettent d'influencer le comportement des *threads* :

`.join()/.join(int millis)` :
: appelée sur un objet **`Thread`**, attend **`millis`** millisecondes qu'il se termine (sans argument, équivalent à **`t.join(0)`** attendre jusqu'à ce que le thread se termine). Peut lever **`InterruptedException`** si un autre thread à interrompu le thread courant pendant un appel à **`.join()`**.

`Thread.sleep(int millis)` :
: (méthode statique) Le *thread* courant interrompt son exécution pendant au moins **`millis`**ms. Après ce délais, il est de nouveau exécutable (mais la JVM peut choisir de terminer d'autres opérations avant de l'exécuter de nouveau). Peut lever l'exception **`InterruptedException`** si un autre *thread* à interrompu le *thread* courant pendant un appel à **`.join()`**.

`Thread.yield()` :
: (méthode statique) permet au *thread* courant de signaler à la JVM qu'il peut être interrompu.

Accès concurrents

Pour pouvoir coopérer les *threads* doivent partager des variables et les modifier. Pour éviter les problèmes de *race conditions* on peut utiliser des méthodes synchronisées :

```
private int x;
public synchronized void incr() { x++; }
public synchronized void decr() { x--; }
```

Si deux threads appellent en même temps la méthode **`incr()`** ou **`decr()`**, les exécutions sont protégées par un verrou.

Accès concurrents (suite)

Sans utilisation de **`synchronized`** la situation suivante peu se produire. On suppose **`x=0`** :

Thread 1

```
→ appel à incr()
  →lecture de x,  x vaut 0
  →calcul x+1, vaut 1
  →écriture x = 1
```

Thread 2

```
→ appel à decr()

  →lecture de x,  x vaut 0

  →calcul x-1, vaut - 1

  →écriture x = -1
```

Après exécution, **`x`** vaut **`-1`**. Avec **`synchronized`** les appels de méthode sont mis en séquence.

synchronized, expliqué (1)

Chaque objet Java (i.e. n'importe quoi qui n'est pas un type primitif comme **`int`**, **`boolean`**, ...) possède un *verrou interne* (*intrinsic lock* ou *monitor lock*), un entier 32 bits stocké dans l'objet.
Lorsque le *thread* courant veut acquerrir le verrou:

- si le verrou vaut 0, incrémenter le verrou
- si le verrou est non nul, mais que le *thread courant* possède déjà le verrou, incrémenter le verrou
- sinon, le verrou ne peut pas être acquis (bloquer ou lever une exception selon les cas)

Pour relacher le verrou il suffit de le décrémenter

synchronized, expliqué (2)

- Lors de l'appel à une méthode *synchronized* sur un objet **o**, le *thread* courant essaye de prendre le verrou de **o** et le relache en fin d'appel (même en cas d'exception)
- Le mot clé synchronized peut aussi être utilisé sur un bloc et un objet Java *arbitraire* :

```
void f(HashMap map) {
  … //traitements longs qui ne dépendent pas de map

  synchronized (map) {
    // seul le thread courant peut
    // exécuter ce bloc
    map.put(…);
    map.get(…);
    map.remove(…);
  }
  … //autre traitements longs
  }
```

Appels périodiques

Une utilisation courante des threads est de vouloir effectuer une tâche à *intervalles réguliers* (ex: recharger le contenu d'un fichier toutes les X secondes).
Java fournit les classes ***Timer*** et ***TimerTask*** pour ce cas d'utilisation très courant.

```
class PeriodicEcho extends TimerTask {
   private int counter = 0;

   @Override
   public void run () {
      System.out.println ("Hello numero " + counter);
      counter++;
   }
}
…
Timer timer = new Timer();
timer.schedule(new PeriodicEcho(), 1000, 4000);//passe en tâche de fond
…
timer.cancel();
```

TimerTask et Timer

La classe contenant l'action doit étendre **`TimerTask`** et doit redéfinir la méthode ***`public void run()`***.
La classe ***`Timer`*** permet de lancer et interrompre des tâches :

`.schedule(task, delay, period)` :
exécute la méthode **`run()`** de **`task`** après **`delay`** milliseconde et la répète toutes les **`periode`** milliseconde avec un délais fixe relativement à la dernière tâche.

`.scheduleAtFixedRate(task, delay, period)` :
exécute la méthode **`run()`** de **`task`** après **`delay`** milliseconde et la répète toutes les **`periode`** milliseconde relativement au début de l'exécution.

`.cancel()` :
Annule toutes les tâches suivantes du **`timer`**. Si une tâche est en court d'exécution, on a la garantie que c'est la dernière exécutée.

Classes anonymes (digression)

Il est souvent inélégant de créer une classe séparée pour définir une seule méthode. On peut utiliser une classe anonyme :

```
Thread th = new Thread () {  //thread est une classe
        @Override
        public run () { … }
};

TimerTask tt = new TimerTask () { //TimerTask est une interface
      @Override
      public run () { … }
};
```

Plan

Fichier MMAPé

Un fichier associé à zone mémoire (*memory-mapped file*) est un fichier auquel le système d'exploitation associe une zone mémoire (un tableau d'octets). On n'accède plus alors dans le fichier par des primitives **`write()`** ou **`read()`** mais directement en écrivant/lisant dans le tableau. Le système d'exploitation s'occupe alors de détecter les mises à jour en mémoire et de les répercuter sur le fichier.

Le principal avantage de cette technique est la très grande efficacité du procédé, à peine un peu plus coûteux que d'écrire dans un tableau en C, le système d'exploitation s'occupe seul d'écrire les données sur le disque au moment opportun.

C'est la base de toutes les implémentations des couches physiques des bases données *modernes*. (i.e. une table de PostgreSQL ou Oracle DB est un fichier « mmapé ».

Exemple en Java (naïf)

On veut lire un tableau de 100 entiers stocké dans un fichier **`tab.data`**, incrémenter tous les entiers, et sauver le tableau modifié dans le même fichier

```
DataInputStream din = new DataInputStream(
                          new FileInputStream("tab.data"));
int tab[] = new int[100];
for (int i = 0; i < 100; i ++)
   tab[i] = din.readInt() + 1;
din.close();
DataOutputStream dout = new DataOutputStream(
                          new FileOutputStream("tab.data"));
for (int i = 0; i < 100; i ++)
   dout.writeInt(tab[i]);
dout.close();
```

Combien de mémoire utilise-t-on ?

Exemple en Java (naïf)

1. En premier lieu, pour des raisons de performances, le système d'exploitation ne lit jamais un fichier octet par octet. Il charge un morceau du fichier dans un *buffer* interne d'un seul coup (1ère copie)

2. Ensuite notre programme lit le fichier et stocke le résultat de la transformation un tableau Java (2ème copie)

3. Enfin, le système d'exploitation n'écrit jamais octet par octet dans un fichier, il stocke les écritures successives dans un *buffer* interne et les écrit en block sur le disque (3ème copie).

On a donc copié 3 fois les données en mémoire. La primitve **`mmap`** permet d'exposer au programmeur les *buffer* systèmes et donc de se passer de une copie (si le fichier de sortie est différent du fichier d'entrée) ou de deux copies (si le fichier d'entrée est le même que le fichier de sortie)

Exemple en Java (efficace)

```
FileChannel in = FileChannel.open(Paths.get("tab.data"),
  StandardOpenOption.READ,
  StandardOpenOption.WRITE);

MappedByteBuffer buff = in.map(MapMode.READ_WRITE,0, 400);

in.close();

IntBuffer ibuff = buff.asIntBuffer();

for (int i = 0; i < 100; i ++)
    ibuff.put(i, ibuff.get(i) + 1);
```

Exemple en Java (efficace)

`FileChannel.open` :
: permet d'ouvrir un fichier. Le fichier doit être représenté par un objet **`Path`**, la classe utilitaire **`Paths`** permet de créer un tel objet à partir du nom de fichier. On doit donner des options lors de l'ouverture du fichier : **`StandardOption.READ`** (fichier ouvert en lecture), **`StandardOption.WRITE`** (fichier ouvert en ecriture), et pleins d'autres (voir la Javadoc).

`FileChannel.map(mode, position, size)` :
: Permet d'obtenir un tableau d'octets représentant le fichier. Le mode peut être **`MapMode.READ`** ou **`MapMode.READ_WRITE`**. On peut restreindre le tableau demandé à un certain fragment du fichier

`.asIntBuffer()` :
: permet de renvoyer une vue « tableau d'entier » du tableau d'octet (les octets sont lus 4 par 4 et combinés pour en faire des entiers).

`.get/.put` :
: permet de lire et d'écrire dans le tableau

Exemple en Java (efficace)

Remarques :

◆ Si on ne demande pas les bons modes, alors des exceptions sont levées lors des opérations invalides (par exemple tenter d'écrire dans un fichier ouvert en lecture)
◆ Un tableau d'octet mappé existe même une fois que le fichier est fermé. Il ne disparaît que lors que le *garbage collector* de Java le supprime car il n'est plus utilisé.
◆ Il existe plein d'options et méthodes, en particulier pour forcer le système à écrire les modifications du tableau sur le disque

Synchronization et verrous

Il est parfois souhaitable que les verrous soient associés à une ressource plutot qu'à du code, et *synchronized* n'est pas forcément adapté. :

◆ Manipulation de plusieurs fichier : la même méthode *synchronized* ne peut pas s'exécuter de manière concurrente, même si elle est appelée sur deux fichiers différents
◆ Manipulation de plusieurs zones d'un fichier : la même méthode *synchronized* ne peut pas s'exécuter de manière concurrente, même si elle est appelée sur deux parties disjointes du fichier

Verrous de fichier

Il est possible de poser un verrou (ou plusieurs) sur un fichier. Un verrou peut être :

- Exclusif : un seul processus peut accéder au fichier
- Partagé : plusieurs processus peuvent accéder au fichier

De plus, on a une granularité fine : on peut vérouiller une portion d'un fichier uniquement (par exemple les octets 20 à 42). On possède trois primitives pour manipuler les verrous :

- Acquérir un verrou (en précisant la portion et le mode) et attendre jusqu'à ce que ce soit possible
- Acquérir un verrou (en précisant la portion et le mode) si possible et renvoyer une erreur si impossible
- Relacher un verrou acquis

En Java

La classe **`FileLock`** représente les verrous, la méthode **`.release()`** permet de les relacher. Pour acquérir un verrou sur un fichier, on doit d'abord créer un objet de la classe **`FileChannel`** (par exemple à partir du nom de fichier). Une fois un tel objet créé, on peut utiliser les deux méthodes suivantes :

`.lock(long position, long size, boolean shared)` :
: essaye d'acquérir un verrou sur le fichier entre les positions **`position`** (incluse) et **`position + size`** (exclue). Si **`shared`** vaut vrai, alors le verrou est partagé, sinon il est exclusif. Le programme est bloqué jusqu'à ce qu'on puisse obtenir le verrou.

`.tryLock(long position, long size, boolean shared)` :
: essaye d'obtenir un verrou (avec les mêmes paramètres que **`.lock`** mais renvoie immédiatement soit le verrou (s'il a pu être acquis) soit **`null`** si ce n'est pas possible.

Il existe une variante **`.lock()`** (resp. **`.tryLock()`**) sans paramètre qui correspond à **`.lock(0, Long.MAX_VALUE,f alse)`**.