© Philippe PRADOS

Utiliser Synchronized

Philippe PRADOS

pp@philippe.prados.name

TABLE DES MATIERES

1. **Consultation d'attributs** 3
2. **Retour d'un attribut** 3
3. **Paramètres** 4
4. **Objet modal** 5
5. **Synthèse** 5
6. **ORB** 5
   - 6.1 **Objet immuable** 5
   - 6.2 **Objet classique** 5
   - 6.3 **Objet fabricant** 6

Avant-propos

Ce document explique comment utiliser correctement l'adjectif `synchronized` *de java.*

L'adjectif `synchronized` de Java permet d'obtenir un accès exclusif à un objet. Ceci équivaut à un sémaphore (ou `mutex`) dans l'objet ou à déclarer une section critique. Toutes les méthodes `synchronized` se bloqueront mutuellement lorsqu'elles sont appliquées au même objet.

Lorsqu'une méthode statique est déclarée `synchronized`, elle bloque l'instance `Class` associée.

```
class MaClass
{
  static synchronized void f()
  { ...
  }
}
```

est équivalent à

```
class MaClass
{
  static void f()
  {
    synchronized(MaClass.class)
    { ...
    }
  }
}
```

Très bien. Mais comment utiliser cet adjectif correctement ?

1. Consultation d'attributs

Si on utilise une méthode qui ne fait que consulter l'objet et manipule deux attributs, elle doit être `synchronized`.

Exemple :

```
public class Toto
{
  int x;
  int y;
....
  public synchronized int somme()
  {
    return x+y;
  }
}
```

Cela permet d'interdire que `x` ou `y` soit modifié pendant l'addition.

Une méthode manipulant au moins deux attributs, doit être `synchronized`

2. Retour d'un attribut

Maintenant, les choses se compliquent. Si une méthode retourne un attribut, cela retourne une *référence* sur celui-ci. L'appelant peut donc modifier l'attribut directement, sans passer par l'objet propriétaire.

Exemple :

```
public class Toto
{
  private ObjetX obj_;
  public synchronized ObjetX getObj()
  {
    return obj_;
  }
}

....
Toto t=new Toto();
...
ObjetX x=t.getObj();
x.methode(); // Pas de synhronized par rapport à t !
```

Cette modification n'est pas gérée par `synchronized`. Si une méthode modifie l'attribut ou remplace celui-ci dans l'objet `Toto`, la référence `x` n'est plus valide.

```
public class Toto
{ ...
  public synchronized void setObj(ObjetX x)
  {
    obj_=x ;
  }
}
```

Si l'appel de `setObj()` intervient dans une autre tâche, juste avant `x.methode()`, l'objet `x` ne représente plus l'attribut de `Toto`.

On doit régler le problème en synchronisant l'instance `x` et l'utilisation de l'attribut.

```
Toto t=new Toto();
...
synchronized(t)
{
  ObjetX x=t.getObj();
  x.methode();
}
```

L'utilisateur de l'objet `Toto` peut facilement oublier d'utiliser `synchronized`. Cela entraîne des erreurs extrêmement difficiles à localiser car elles ont tendance à être aléatoires. On peut éviter ce risque en renvoyant une copie de l'attribut pour obtenir une sémantique par valeur.

Exemple :

```
public class Toto
{
  private ObjetX obj_;
  public synchronized ObjetX getObj()
  {
    return obj_.clone();
  }
}
```

On informe alors clairement l'appelant que l'on retourne la *valeur* de l'attribut. Cela représente une copie de celui-ci au moment précis de l'appel à `getObj()`. L'appelant sait qu'il ne possède qu'une copie ; l'objet est garanti qu'aucune modification sera faite à son insu.

Si la méthode `clone` de l'attribut n'est pas `synchronized`, on doit déclarer la méthode `getObj()` avec cet adjectif. L'objet `Toto` assume la responsabilité des modifications de son attribut.

Une méthode manipulant un seul attribut doit être `synchronized`

Ne jamais retourne un attribut dans une méthode `synchronized`**. Retourner une copie.**

3. PARAMETRES

Lorsqu'une méthode reçoit un objet en paramètre, ce dernier doit-il être `synchronized` avant l'appel ? L'appelant doit-il le faire ? Ou doit-il de lui-même régler cela dans toutes ces méthodes ?

Synchroniser le paramètre dans la méthode ou avant la méthode n'est pas suffisant.

```
public class Toto
{
  void methode(ObjetX para)
  {
    synchronized(para)
    { ...
    }
  }
}
```

ou

```
ObjetX x=new ObjetX();
...
Toto toto=new Toto();
synchronized(x) ;
{
  toto.methode(x);
}
```

Une méthode du paramètre peut ne pas être `synchronized`. Dans ce cas, une tâche peut modifier le paramètre à l'insu de la méthode.

Pour régler le problème, il faut utiliser une copie du paramètre.

```
ObjetX x=new ObjetX();
...

Toto toto=new Toto();
toto.methode((Toto)x.clone());
```

Cette difficulté intervient lorsque le paramètre est utilisé simultanément par plusieurs tâches.

Dans quel cas est-il possible de ne pas utiliser `synchronized` qui pénalise les performances ?

Le seul cas possible est d'être sûr que l'instance n'est utilisée que par une seule tâche.

S'il existe une méthode `synchronized` dans un objet, toutes les méthodes de celui-ci doivent l'être. (Sauf éventuellement les méthodes `private`)

4. OBJET MODAL

Un objet est modal si un traitement est dépendant de traitement précédent. Une méthode entraîne un effet de bord pour les autres méthodes. Un objet modal est difficile à utiliser dans une approche multi-tâches. En effet, imaginons une méthode `set()` qui modifie l'état d'un objet en mémorisant deux entiers. La méthode `addition()` dépend de cet état pour retourner l'addition des deux entiers. Une utilisation classique de cet objet est d'appeler `set()` avec les deux entiers de l'opération, puis d'appeler la méthode `addition()`.

Regardons une utilisation avec deux tâches.

- Tâche 1 : `calc.set(2,3)`
- Tâche 2 : `calc.set(5,6)`
- Tâche 2 : `calc.addition()` retourne 11
- Tâche 1 : `calc.addition()` retourne 11 et non 5

Cet exemple démontre la difficulté d'utiliser un objet modal en multi-tâches. Même si les méthodes `set` et `addition` sont `synchronized`, l'utilisateur ne peut pas les manipuler sans synchroniser de l'extérieur l'objet. Java offre pour cela une syntaxe particulière :

```
synchronized(calc)
{
  calc.set(2,3);
  calc.addition();
}
```

L'ensemble des traitements doit être non modal. Après l'addition, l'utilisateur de l'objet ne doit pas sous-entendre un état particulier de l'objet.

Comment avec java, synchroniser plusieurs objets ? Par exemple, un programme désire lire tous les attributs d'un objet pour les sauver dans un autre. Il faut bloquer la source et la destination simultanément.

```
synchronized(obj1)
{
  synchronized(obj2)
  {
    obj1.set(obj2);
  }
}
```

5. SYNTHESE

Le modèle objet désire cacher l'implémentation. L'utilisateur d'un objet ne doit pas connaître le corps d'un objet. Il ne doit donc pas savoir s'il existe des effets de bords ou des risques d'utiliser une méthode, vis-à-vis du multi-tâches. Normalement, lorsque l'on rédige une classe Java, celle-ci doit être utilisable en multi-tâches. Dans ce cas, il faut avoir des méthodes `synchronized`. Cela entraîne que toutes les méthodes de l'objet le sont. Donc, **toutes** les méthodes de **tous** les objets doivent être `synchronized` si l'on désire une utilisation correcte de l'objet. De plus, il faut transformer la sémantique par référence de Java par une sémantique par valeur en dupliquant à la main les attributs et les paramètres.

Une approche plus pragmatique oblige à éviter au maximum l'utilisation simultanée d'une instance. Il ne faut pas hésiter à faire des copies d'objets. Les rares candidats utilisables en multi-tâches doivent être encapsulés dans un objet de gestion. C'est le cas des instances `Image` de la librairie Java. Celles-ci lancent une tâche de fond s'occupant de récupérer l'image sur le réseau. Une instance `Image` est utilisable avant la fin du chargement.

6. ORB

Dans une architecture d'objets distribués du type CORBA/RMI ou COM, les objets serveurs peuvent être appelés par plusieurs clients. Il propose alors une utilisation en multi-tâches. Plusieurs stratégies sont possibles pour remédier aux problèmes évoqués ci-dessus.

6.1 Objet immuable

Un objet serveur immuable n'offre pas de difficulté lors de l'utilisation en multi-tâches. Chaque traitement est autonome et ne dépend pas des précédents. C'est une approche possible pour faciliter l'utilisation en multi-tâches d'un objet.

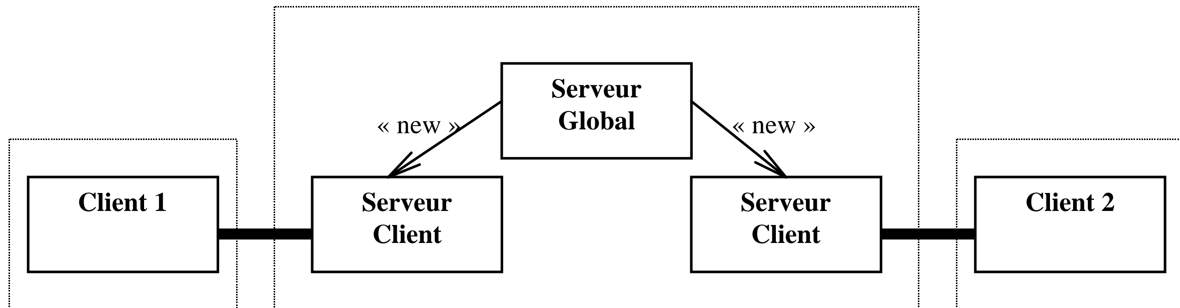
6.2 Objet classique

Un objet classique est difficile à utiliser dans une approche multi-tâches. Offrir le même service que le mot clef `synchronized` avec un serveur que n'est pas chose aisé. Il faut envoyer un message de blocage au serveur ; faire les différents traitements ; puis envoyer un message

de déblocage. Tous les services doivent se synchroniser sur le sémaphore de blocage. Celui-ci doit identifier le client lors du blocage. Que se passe-il si le client est perdu avant d'avoir débloqué une instance ?

6.3 Objet fabricant

L'objet serveur peut offrir un service permettant d'instancier un autre objet spécifiquement pour un seul client. Un client demande à l'objet serveur global de lui fournir un serveur particulier. Celui-ci n'a alors plus besoin de gérer les problèmes de multi-tâches. Il est sûr qu'un seul client à la fois l'utilise.



Attention, le client peut envoyer simultanément plusieurs requêtes aux serveurs. Dans ce cas, il peut être judicieux de sérialiser les traitements. L'inconvénient de cette approche est que le client doit explicitement détruire son serveur personnel. Si le client « plante », le serveur du client ne sera pas détruit. On peut ajouter un mécanisme de battement de cœur entre le client et son serveur, pour que celui-ci détecte automatiquement l'absence du client.