

JAVA SE – Singletons

Des Singletons

Note : le contenu de cette fiche suppose la connaissance de la problématique des ClassLoaders ([voir la fiche correspondante](#)).

Les objets utilisés en java sont généralement créés en de multiples exemplaires : c'est le principe même des objets dont l'état encapsule un contexte particulier. On aura donc, par exemple, plusieurs *instances* de la classe Produit ; chacune "contenant" un ensemble d'informations qui lui est propre. Les codes vont ensuite demander des services aux instances : "quel est le prix TTC de **ce** Produit", etc.

Il existe des cas où les services demandés ne dépendent d'aucun état d'instance particulier. C'est la cas de la classe Math : le calcul de Math.sin(3.14) se suffit à lui même. La méthode sin est statique et la classe Math ne rend que des services statiques (il est d'ailleurs impossible d'en créer une instance!).

On peut donc nous même créer des ensembles de services qui ne s'appuient pas sur la création d'une instance.

Soit par exemple cette classe CalculsIntensifs :

```
public class CalculsIntensifs {
    public static Future<BigInteger>
        fonctionAckerman(int x, int y) {
        oooo
    }
    // autres méthodes
}
```

Ici la méthode est un calcul très long qui s'exécute en tâche de fond et qui rend une "promesse" (un résultat qui sera disponible plus tard quand le calcul sera terminé). Le code de la méthode peut bien sûr être dans le code de la classe mais on pourrait aussi se réserver la possibilité de faire évoluer le code. L'exécution dépendrait alors de considérations de déploiement : par exemple délégation de l'exécution à une autre machine. On aurait ici un point d'entrée bien connu mais un découplage vers une réalisation spécifique s'appuyant sur un objet unique chargé de faire exécuter les calculs. Comme souvent lorsque l'on s'adresse à un service extérieur il n'y a pas lieu d'avoir plusieurs objets. Un seul objet sera chargé, par exemple, d'écouter sur un port TCP (si un deuxième objet tentait d'accaparer ce port il échouerait).

On entre ici dans les problématiques de réalisation de *Singletons* (objets qui ne disposent que d'une seule instance).

Comme pour la plupart des *patterns* il s'agit d'un sujet complexe pour lequel il faut confronter les raisons d'être et les pistes de réalisation. Cette fiche n'a pas la prétention d'en explorer la totalité des aspects : nous nous contenterons d'ouvrir quelques perspectives.

D'abord une première question : dans quelle mesure voulons nous un *singleton* VRAIMENT unique?

Soit un service EcouteSurPortBienConnu : on a ici une exigence totale d'unicité sur l'ensemble de l'application (et même sur l'ensemble des processus de la machine tant qu'on y est!).

Inversement un service comme ChargementImages pourra se limiter au contexte d'un ClassLoader : on aura un objet unique dans les frontières du ClassLoader mais sur l'ensemble de l'application il pourrait y avoir plusieurs singletons chargeant des images selon des modalités différentes !

Pour la réalisation on a deux problèmes principaux :

- comment garantir une instance unique?
- comment fournir cette instance aux codes "client"?

Quelques illustrations :

Un code de l'interface `ChargementImages` s'appuyant sur le déploiement standard de services (à partir de 1.6) :

```
public interface ChargementImages {  
    Image getImage(String nom) ;  
    ChargementImages SERVICE = ServiceLoader.load(ChargementImages.class).  
        iterator().next() ;  
}
```

Ici le code client **sait** qu'il n'a affaire qu'à une seule instance. La création de cette instance ne se fera qu'à la première consultation de `SERVICE` (les interfaces ne sont chargées qu'à ce moment). Sous réserve d'une cohérence au niveau des `ClassLoaders` (et d'une gestion appropriée des incidents de déploiement) on pourra avoir une partie de l'application qui centralise à son niveau les demandes d'image.

La vision de l'instance unique peut être encapsulée par une méthode "fabrique" :

```
public class EcouteReseau {  
    private static final EcouteReseau SERVICE =  
        codeQuiGénèreInstance() ;  
    public static EcouteReseau fabriqueEcoute() {  
        return SERVICE ;  
    }  
    // pas de constructeur public de EcouteReseau  
}
```

Ici, classiquement, le `codeQuiGénèreInstance` peut produire un objet dont le type effectif est une sous-classe de `EcouteReseau`.

Il est possible que, dans le cas précédent, les opérations d'initialisations ne puissent pas s'effectuer au *loadtime* de la classe elle même et que l'on doive les reporter à la première utilisation :

```
public class EcouteReseau {  
    private class ServiceAChaud {  
        private static final EcouteReseau SERVICE =  
            codeQuiGénèreInstance() ;  
    }  
    public static EcouteReseau fabriqueEcoute() {  
        return ServiceAChaud.SERVICE ;  
    }  
}
```

(Il est possible également que ce chargement paresseux ne soit absolument pas souhaitable -par exemple il peut compliquer inutilement les tests de déploiement)

La nature *singleton* de l'objet peut enfin être complètement transparente pour le code client :

```
public class ServeurImage implements ChargementImages {  
    public ServeurImage() {}  
    private static class ServeurAChaud { // pas forcément nécessaire  
        private static final ChargementImages SERVICE =  
            new ChargementImages(){  
                public Image getImage(String nom) {  
                    oooo  
                }  
            };  
    }  
    public Image getImage(String nomImage) {  
        return ServeurAChaud.SERVICE.getImage(nomImage) ;  
    }  
}
```

Ici les méthodes de chaque instance sont déléguées à une instance unique.

Entre la vision des codes applicatifs et la vision des codes de déploiement on peut imaginer que différents *singletons* se combinent. On aurait alors un *singleton* pour accéder au service et un *singleton* pour rendre le service.

Nous l'avons dit au départ de nombreuses autres questions peuvent se poser : sous-classes du singleton, pertinence et technique de la sérialisation, etc. Comme pour tous les *patterns* il est donc important de s'inspirer des solutions proposées pour construire une solution personnalisée adaptée aux spécificités de chaque situation.

Formation JAVA sur ce sujet

- Nos formations longues (standards ou spécifiques).
- Nos cours "java avancé" adaptés à la demande.

Ici plutôt que de faire un inventaire exhaustif de *patterns* nous préférons porter l'accent sur l'analyse même du concept de *pattern* et les techniques d'adaptation aux réalités du terrain.

Pour des cours en profondeur dans le domaine Java EE :

- [USL425 - Architecture et Conception d'Applications Java EE](#)
- [USL500 - Patterns Java EE](#)

[Toutes les formations Développement](#)

Bernard AMADE

© Demos