|  |  |  |  |
| --- | --- | --- | --- |
| Le Mot | L’utilisation | Remarque | Utilité |
| Final | Indique que la classe, l’attribut ou bien la méthode ne peut pas être modifié dans la suite du programme | Dans le cas d’une méthode, cette méthode ne sera pas redéfinie dans les classes dérivant de cette méthode. |  |
| Abstract | C’est une classe que le client ne peut pas créer une instance à partir d’elle. | Pour déclarer une méthode abstract i faut déclarer la classe comme abstract. | Interdire l’instanciation |
| Noninstaciable | C’est une classe |  |  |
| Static | Définir qu’une méthode ou un attribut peut être utilisé sans avoir besoin d’instancier la classe. |  |  |
| Subclass | C’est une classe qui hérite d’une autre class |  |  |
| Singleton | Le premier design pattern : 1-créer une unique instance ;2-Assurer l’accés public à cette instance | Utiliser la technique de synchronisation(Local ou Global) ou bien le holder et à base de static factory method (méthode utilitaire) |  |
| Serialisation | Convertir un objet en une serie d’octets |  |  |
| Immutable | C’est un objet qui ne peut pas changer leur état après l’instanciation |  |  |
| Adapter | Design patern consiste à adapter un type à un autre en utilisant une interface. |  |  |
| autoboxing | automatic conversion that the Java compiler makes between the primitive types and their corresponding object |  |  |
| hashcode | Méthode commun pour tous les objets, permet de transférer les valeurs d’un objet en un code en entier. |  |  |
| List | Collection ordonnées, dont l’utilisateur à le control total, il peut récupérer le n ème élément, recherché, |  |  |
| Set | Ne contient pas des éléments dupliqués, et contient au plus un élément null. |  |  |
| Mutex (Exclusion mutuelle) | Primtive de synchronisation qui permet d’assurer que les ressources systèmes ne soient pas utilisées par plusieurs programmes à la fois |  |  |
|  |  |  |  |

# Item 1 : Static Factory Method

Une static factory method permet de déclarer une classe sans avoir besoin du constructeur, , de plus il sont préférer dans l’utilisation. Donc éviter la réflexion directe vers le constructeur sans avoir pensé au statics factory method.

## Avantages

1. Contient un nom :: éviter l’ambiguïté dans l’appel du constructeur :
2. pas besoin de redéfinir la méthode equals et pas besoin de déclarer new à chaque fois
3. Peut retourner plusieurs types exemple EnumSet : RegularEnumSet et JumboEnumSet
4. reduce the berbocity of creating parameterized type instances: exemple : Map<string, list<String >> m = new HashMap<string, list<String>>(); public static <K,V> HashMap<K,V> newInstance() {

return new HashMap<K,V>();

map<string, list<String >> =Hashmap.newinstanciate(); }

## Inconvégnant:

1. on peut pas hériter de cette class si la classe ne contient pas le mot (Public, Protected) et c’est le cas la framework Collections.
2. On ne peut pas les distinguer par rapport au d’autre méthode statique.

Quelque Static Factory method :

Valueof, of, GetInstance, NewInstance,Gettype,newtype

# Item2 Builder Pattern:

## Telescoping constructor pattern:

Déclarer simplement plusieurs constructeur ; exemple 6 champs; 5 constructeurs (2champ, 3champs,4champs,5champs,6champs)

## JavaBeans pattern :

Donner des valeurs par défaut à chaque attributs, puis lors de a déclaration utiliser les setters pour modifier les valeurs.

Builder Pattern : combiner entre les deux méthodes précédentes : CAD : créer une classe static interne, initialiser le constructeur avec les valeurs à 0, puis créer des méthodes qui retourne la classe et qui initialise un champs, puis créer une méthode qui retourne la classe initiale en utilisant le constructeur privé de la classe parent.

Et créer un constructeur privé basé sur la classe interne.

En résumé, le modèle Builder est un bon choix lors de la conception de classes dont les constructeurs auraient plus d'une poignée de paramètres, en particulier si la plupart de ces paramètres sont facultatifs. Le code client est beaucoup plus facile à lire et à écrire avec les Builder qu'avec le modèle de constructeur télescopique traditionnel, et les Builder sont beaucoup plus sûrs que JavaBeans.

# Item 3 : Singleton

## 2 méthodes ancien

1-mot final : mettre le constructeur private et déclarer un static final attribut en interne avec ce constructeur. (Utilisateur privilégie peut accéder au constructeur en utilisant la méthode **AccessibleObject.setAccessible**

2- Static factory method : avantage (faciliter la personnalisation de l’utilisation (unique instance pour chaque thread),

Afin d’éviter qu’un autre objet sera créer si la classe implement serialisable, il faut ajouter la méthode readresolve () qui va retourner l’objet elle-même.

3 - La meilleure approche est d’utiliser une énumération avec un seul élément. Car elle fournit la machine de sérialisation gratuitement, et fournit une garantie à toute épreuve contre l'instanciation multiple,

# Item 4 : uninstanciable classe

Il suffit d’ajouter le mot private au niveau du constructeur de la classe voulu.

# Item 5 : avoid creating unnecessary object

Si la création d’une classe ne nécessite pas l’utilisation de constructeur, on évite d’utiliser celle-ci. Exemple : String.

Utiliser les Item1 au lieu du constructeur pour la création des immutables classes.

Ne pas déclarer les variables qui ne sont pas modifiables au niveau des méthodes mais sera mieux de les déclarer au niveau de la classe en tant que statique.

L’utilisation de l’auto boxing permet parfois de résoudre le problème d’adaptation.

“Don’t create a new object when you should reuse an existing one”

# Item 6 : Eliminate obsolute object

1-Exemple de la partie non null du tabeau pour le garbage, et qui est considérer obsolète pour le programmeur.

2-la 2 source de la perte de mémoire est le cache

3-la 3 sources se sont les listener et les callbacks

# Item 9 : Override Hashcode when you override equals

Il est obligatoire que 2 object qui sont égaux par la méthode Equals, doivent retourner le même entier.

Le contraire n’est pas obligatoire mais il peut améliorer les performances des **Hash-Based** .

Dans les hash-base, si le hashcode est différent alors java ne compare pas avec equal, il prend que les 2 objets sont différents

Dans les classes immutables on peut définir le hashcode au-dedans de la classe, si le cout de calcul du hashcode est important, ceci en déclarant le variable comme volatile, si le variable égale à 0, on calcule le hashcode sinon, on retourne le hashcode précédent.

# Item 10 : Override ToString

Il faut redéfinir la methode toString pour que le code soit le plus lisible possible.

Lorsque vous appeler les fonctions d’affichage, la fonction toString est automatiquement appelé.

Vous pouvez définir le format de la description affichée lors de l’invocation de la méthode to-string en utilisant la méthode ( **String.format()** ).

# Item11 Override Clone judiciously

clones() est diffèrent de X ;

X.equals(X.clone())must return true

1. La méthode clone doit appeler super.clone (), jusqu’à l’arrivée à la classe object, elle doit aussi être public et retourner la classe elle -même.
2. Il faut être attention lors du clonage d’une classe contenant une liste, ou tableau, ou hashmap (exemple du clone de la classe hashtabe, et de la pile)
3. La méthode clone ne doit pas invoquer une méthode non final, car le clonage peut ne pa être terminer
4. Si une classe est déclarer pour être hérité, elle doit suivre le même comportement de la classe (Object: Engendre l’exception clonenotsupportedexception) et elle ne doit pas implémenter l’interface clonable.
5. Si on doit utiliser un Safe Thread class, on doit déclarer un synchronized Clone method.

On peut copier on utilisant CopyConstructor ou bien CopyFactory, cela donne plusieurs avantages, mais elle ne peut pas jouer le rôle d’une interface.

Public Class(Class class) -> CopyConstructor.

Public Static GetClass(Class class)->CopyFactory.

Vous pouvez transformer une HashSet en TreeSet, en utilisant CopyConstructor ou CopyFactory.

Les interfaces ne doivent pas héritées de cloneable et les classes conçues pour l’héritage aussi. Les experts n’utilisent jamais cloneable sauf pour le clonage des tableaux.

# Item 12 : Consider Implementing Comparable

La fonction compareto engendre une exception de type ClassCastException, ou bien retourne -1,0,1.

1-sgn (x.compareTo(y)) == -sgn(y.compareTo(x))

2-(x.compareTo(y) > 0 && y.compareTo(z) > 0) implies x.compareTo(z) > 0.

3-x.compareTo(y) == 0 implies that sgn (x.compareTo(z)) == sgn(y.compareTo(z))

4-(x.compareTo(y) == 0) == (x.equals(y))

Les collections trié utilise le test CompareTo au lieu de Equals (exemple HashSet et TreeSet avec la classe bigdicimal)

Faire attention si la différence est calculé à partir de deux entier négatifs, ou que la différence soit plus grande que le rang integer 2\*31-1

# Item 13 : minimize accessibility to Class and its members

C’est le principe de « hidding information » ou « encapsulation », ce principe est parmi les avantages et les bonnes pratiques du programmeur, car il assure l’isolation des modules, ce qui rend la programmation plus rapide (Plsr Module en parallèle) et plus facile à maintenir.

Mettre chaque classe le plus possible inaccessible

Si une méthode est hérité d’une classe parent, on ne peut pas avoir un niveau d’accès plus bas que celle de la méthode parent.

Une instance d’une sousclasse doit avoir un niveau d’accès plus haut que l’instance de la superclasse

**Remarque :** Ne jamais déclarer un attribut ou un objet mutable comme public.(Exemple de la table de String)

# Item 14 : Use accessors method and mutators

Afin de facilité la modification de la représentation interne de la classe.

Parfois, il est plus simple de déclarer les attributs comme public, dans le cas des private-package et les classe imbriqué ‘Nested-Class’.

**Remarque :** Pour les attributs static final, pas de problèmes pour leur public déclaration.

# Item 15 : minimize mutabilite

Pour mettre une classe non modifiable, suivre les 5 instructions suivantes :

* ne pas utiliser les mutator
* assurer que la classe ne peut pas être hérité (utilisé le mot Final)
* mettre tous les attributs comme Final.
* mettre tous les attributs comme Prive
* assurer l’acces exclusive aux objets modifiables (refesive copie, readobject)

## Fonctionnel Approche : appliquer des fonctions à l’objet sans le modifier, en retournant une nouvelle instance de l’objet.

Avantage Immutable :

* pas besoin d’être synchroniser (meilleur approche pour assurer Safety Thread Principe)(Exemple fonction Arithmétique du nombre Complexe).
* Exemple 0 et I pour les nombre complexe), cela permet de gagner de l’espace et facilite le rôle du Garbage Collector.
* Ils constituent d'excellentes clés pour les Map et Set
* Ils n'ont besoin ni de constructeur par copie, ni d'implémentation de l'interface Cloneable
* Il n'est pas nécessaire d'en faire une copie défensive

Inconvégniant : devoir créer une instance pour chaque objet, ce qui couteux en terme d’espace si l’objet et large.

Exemple de class Immutable : BigInteger et BigDecimal.

## Mutable Campaniom : StringBuilder pour string et BitSet pour BigInteger,

Afin de mettre la classe non héritable, soit on utilise le mot Final ou bien on déclare les constructeur comme privé, et on utilise les static factory method.

Ce n’est pas possible d’extander une classe qui n’a pas un constructeur public or protected.

On peut utiliser des champs modifié afin de ne pas les recalculé en cas de besoin (exemple de la variable qui garde le hashcode)

Toujours penser à mettre la classe immutable et mettre un mutable campaniom en cas de besoin. Et si vous n’arrivez pas à la mettre immutable, essayez de imiter ces états exemple (TimerTask)

# Item 16 : Favor composition over inheritence

La mauvaise documentation de la classe peut engendrer des cas d’erreurs (exepmle de la redéfinition de la classe addAll.

Si vous hériter d’une classe et vous définit une classe avec une signature, si dans les prochaines relaeses la classe parent définit la même classe avec un autre type de retour, le programme ne sera pas compilé (erreur lors de la compilation).

Composition c’est de créer une classe qui implémente l’interface souhaité (Forwarder class), cette classe sera héritée par la classe simple (Wrapper Class).

Il faut pas utilisé l’héritage sans poser a question est ce que la classe B est Vraiment A.

# Item 17 : Design and Document For inheritence

Il faut bien documenter tous les classes conçues pour l’héritage, c.-à-d. documenter chaque méthode qui appelle des fonctions non final, ou public.

Il ne faut jamais utilisé dans le constructeur d’une classe conçue pour l’héritage des appels à des méthodes public ou bien non final.

Afin de tester l’héritage, il faut en minimum écrire trois classes.

Le constructeur ne doit pas appeler une méthode qui peut être redéfinit.(Exemple de la classe super et sub, car la méthode de SUB est invoqué avant l’initialisation de SUB)

N’est pas conseillé de concevoir une classe destiné à l’héritage et qui implémente les interfaces clonable et serializable dans l’inverse (Assurer que clone et ReadObject n’appel pas an overridable method).

Prfois ce principe n’est pas utilisé, exemple item 18.

Ne pas hériter à partir des ordinary classes, qui ne sont pas conçu pour l’héritage.

Dans le cas ou vous permettez l’héritage, assurer de ne pas utiliser les méthodes redéfinis, en déclarant des méthodes Helper pour chaque méthode redéfini, cette méthode contient le corps de la première et jou le même rôle.

# Item 18: Prefer Interfaces To abstract classes

La différence entre les deux (Interface et Abstract Class) est que l’interface permet de définir le principe du MIXIN et les autres non.

Une interface peut hériter de plusieurs autres interfaces

Skeletal implementation : AbstractInterface.

L’avantage c’est de bénéficier des fonctionnalités des interfaces avec les classes abstracts.

Permettre le multihéritage.

Il faut faire attention lors de la conception des interfaces, si une interface est conçue et débuguer, il est pratiquement impossible de la changée.

# Item 19: Use interfaces only to define Types

Définir les interfaces pour décrire ce que l’objet peut faire (constant interface).

Ne pas utiliser les interfaces constantes comme **java.io.ObjectStreamConstants**.

Vous pouvez utiliser les classes non instanciables ou bien les enums au lieu des interfaces constants.

On peut utiliser import static pour importer une variable d’une classe.

# Item 20 : Prefer classes hierarchies to tagged class

Classe hiérarchie : définir une classe abstraite avec une méthode abstraite. Et définir d’autres classes qui héritent de cette classe abstraite.

Avantage : ne pas encombrer la classe, et tous les attributs peuvent être Final.

Lorsqu’une classe utilise un taged attribut, pensez toujours à utiliser la notion des classes hiérarchies.

# Item 21: Use function objects to represent strategies

Les fonctions d’objet sont utilisé afin d’implémenter le Strategy Pattern

On peut déclarer des fonctions object en utilisant une classe singleton qui ne contient pas d’attribut et qui définit la fonction.

On peut aussi utiliser les classes annonymes pour déclarer les fonctions D’objet à l’aide d’une interface comparator.

On peut utiliser une nested-class qui implémente l’interface comparator.

# Item 22: Favor Static member classes over non static

Il y’a quatre types de classes internes:

1. Static Member classes
2. Non Static Member classes
3. Anonymous classes
4. Local Classes.

Static member class, doit porter le mot static, et l’inverse pour l’autre.

Si une instance d’une classe interne peut être utiliser en isolation par rapport à la classe conteneur, elle doit être static member.

Anonyme classe est une classe qui ne porte pas de nom et qui est déclarer lors de l’utilisation de la classe (exemple array sort function object).

Les classes anonymes ne devront pas dépasser 10 lignes.

Les classes anonymes sont utilisé pour créer les prosses objects (Runnable, Thread, or TimerTask).

Anonymes classes and local classe ne peut pas avoir des attributs Static.

# Item 23: Don’t use raw type in new code

Une interface ou classe générique contient une ou plusieurs paramètres Exemple <E>. (Generics Types), et le RawType c’est le nom du générique type Exemple List<E> a comme RawType List.

Avant release 1.5 : ce n’était pas possible de déclarer une collection avec un type prédéfinit ce qui pose des problèmes pour le programmeur.

Il faut assurer que vous compiler dans un release supérieur à 1.5.

Vous n’êtes pas obligé d’utilisé le cast pour récupérer les éléments.

Vous pouvez utiliser juste RawType, mais si vous faites ça, vous allez perdre les avantages des types génériques (Assurer que la collection n’engendre pas des erreurs).

Java permet d’utiliser les rawtype afin d’assurer le principe de ‘Migration Compatibilité’

Unbounded wildcard types représente une liste de type inconnu exemple de déclaration : Set<?> s.

Il y’a deux exception pour utiliser les rawtype :

1. On doit l’utilisé dans les classe littérale car (list<Object>.class n’est pas permis).
2. On doit l’utilisé dans l’expression instanceOf.

# Item 24 : Eliminate unchecked warning

Vous devez éliminer tous les Unchecked Warning afin d’assurer que le code est TypeSafe.

Si vous ne pouvez pas éliminer tous les cas il suffit de montrer que le programme est typesafe afin d’assurer que le programme marche et qu’il ne génère pas des ClassCastException.

Vous ne pouvez pas utiliser l’expression SuppressWarnings (unchecked) au niveau de l’instruction Return. Il faut déclarer une variable local puis retourner cette variable local.

Quand vous utilisez the suppresswarning, ajouter un commentaire pour indiquer pourquoi ce n’est danger de laisser cette warning error. Ceci rend le code plus compréhensible.

Chaque unchecked warning peut générer une exception de type ClassCastException.

# Item 25 : Prefer Lists to Array

Il y’a deux grandes différences entre Array et List :

1. Les tableaux sont covariants c.-à-d. : si une classe1 est soustype d’une autre classe2. Alors classe1[] est soustype de classe2[], mais ce n’est pas correct pour les listes. Exemple des erreurs (Lors de l’exécution pour la table et lors de la compilation pour la liste).
2. Les tableaux sont reified c-à-d que les tableaux ne connaissent les types de ces éléments interne que lors de l’exécution.

Ces déclarations List[], new List[], new E[] ne sont pas permis afin de ne pas perdre la propriété de typesafe (Exemple ListString et ListInteger)

Tous les éléments génériques ne sont pas reifiable sauf unbounded wildcard types List<?> and Map <?,?>. ;;

# Item 66 : Synchronize access to shared mutable data

Le mot synchronisation signifie qu’un seul thread peut exécuter une méthode ou un block dans un intervalle de temps.

La synchronisation garantie qu’aucune méthode ne voit l’objet dans un état incohérents, pour cela l’objet sera créé dans un état consistent et sera verrouillé par la méthode qui y accès.

Sans la synchronisation, les changements ne seront pas visibles entre les threads.

La synchronisation ne permet pas seulement aux méthodes de voir les objets dans des états cohérents mais aussi de voir tous les effets des changements.

Les spécifications du langage ne garantissent pas que les écritures sur une variable sont visibles pour les autres threads.

La synchronisation est nécessaire pour une fiable communication entre les threads et pour aussi assurer l’exclusion mutuelle Mutex.

Il ne faut pas utiliser thread.stop.

On doit synchroniser les lectures et les écritures sur une variable afin que cette variable soit synchronisée entre les threads.

Lorsqu’on utilise la synchronisation juste pour lire les valeurs et sans avoir besoin d’appliquer l’exclusion mutuelle, il sera mieux d’utiliser le mot Volatile qui permet de voir la valeur la plus récente de la variable.

# Thread documentation from JAVA Programming Language Book

La function holdsLock permet d’indiquer si le thread a locker (Verrouiller) l’objet ou non.

En utilisant le mot synchronized pour les méthodes, on garantit qu’il y’aura pas de conflit entre les threads pour l’accès à cette méthode.

Si on hérite et on redéfinit une méthode synchronisé, la nouvelle méthode peut etre synchronisé ou pas. Mais l’invocation de la méthode père est toujours synchronisée.

On peut également ajouté le mot synchronized à une méthode static, dans ce cas, cette méthode de la classe ne sera pas accessible par 2 threads au même temps.(méthode de l’objet).

On peut également utiliser le mot synchronized pour les blocks de code exemple ;

Synchronized(Variable) { ….. }., ou bien pour les classes internes exemple : inner et outer, Synhronized (this.outer).

## Synchronization Designs:

Client side-synchronization: synchronisé de la part de client c’est-à-dire assurer que tous les clients appel l’objet partagé dans un bloc synchronisé (approche utilisé dans le cas où les classes ne sont pas conçues pour un environnement multithread, on peut également définir une sous classe, et on redéfinit les méthodes avec le mot synchronized).

Server side-synchronization: Assurer que tous les méthodes de l’objet partagé porte le mot synchronized ou bien leurs corps est synchronisé (cette approche est meilleur).

La méthodes wait et notifyall() et notify() permet de communiquer entre les threads.

L’avantage de la méthode wait c’est quel permet de liberer l’objet locker par la méthode, elle permet aussi de mettre l’objet sous l’état suspended.

Quand on fait des changements sur des variables qui suspend d’autre thread, on utilise la méthode notify pour éveiller un thread ou bien notifyall pour éveiller tous les threads suspendus.

La méthode wait() engendre l’exception ‘InterreptedException’, on peut aussi passer un temps d’attente en milliseconde en paramètre(assurer que le programme return, ‘temps infini dans le cas contraire’).

La différence entre notify et notifyall est que notifyall permet d’éveiller tous les threads attendant la condition, contrairement à l’autre méthode qui éveille un seule thread.

Avantage de la méthode wait ; pour ne pas surcharger le programme avant la satisfaction de la condition.

Ces méthodes ne peuvent pas être utilisées dans une section de code non synchronisée.

# Item 67 : Avoid Accessors synchronization

Eviter toujours d’appeler (the alien method) des fonctions qui peuvent etre redéfinit par l’utilisateur (Overridden Method), ou bien les fonctions de type Fonction Object, ceci est afin d’empêcher les exceptions, les deadlocks, et la corruption des données. Exemple de : Observable Set, et l’interface SetObserver.

On peut utiliser l’interface Java.util.concurent qui permet de faciliter les actions de synchronisation entre les threads. Exemple : [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html)<E>, CopyOnWriteArrayList<E>, [CopyOnWriteArraySet](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArraySet.html)<E>.

Essayer de faire le minimum de travail au sein d’une region synchronisé.

Le problème de performance lors de l’utilisation accesseur de la synchronisation ce n’est pas le temps perdu pour l’obtention des locks, mais c’est la perte de possibilité de parallélisme, et le délai perdu pour assurer que chaque corps à une vue consistent de la mémoire.

De préférence, laisser le client faire la synchronisation, donc ne pas synchroniser votre classe mais, documentez quelle n’est pas thread safe(Mais lorsque vous utiliser des attribut static, faire la synchronisation de ces attribut en interne de la classe) (cas de la classe StringBuilder qui est une StringBuffer non Synchronisé) « generateSerialNumber page 263 »

## Conclusion :

1. pour éviter les interblocages et la corruption de données, n'appelez jamais une méthode alien depuis une région synchronisée. Plus généralement, essayez de limiter la quantité de travail que vous effectuez à l'intérieur des régions synchronisées.
2. Dans les versions modernes des multicœurs, il est plus important que jamais de ne pas trop synchroniser.
3. Synchronisez votre classe en interne seulement s'il y a une bonne raison de le faire, et documentez clairement votre décision.

## LockSplitting :

Utiliser les locks au niveau interne des méthodes (Utilisé des blocks de synchronisation au lieu de l’utilisation des mots Synchronization), pour but de séparer les locks de variable.

## Lock Striping :

Il est utilisé pour permettre l’accès simultané à une Collection, le concept est de définir une table de locks, est de diviser l’accès à la collection par la fonction modulo, c’est-à-dire diviser la collection en ‘N’ sous collection, ou n représente le nombre de locks possible (Avoir N accès en parallèle).

Striping Map est une implementation d’une hash-based map en utilisant le principe du lock Striping.

# Item 68: Prefer Executors and Tasks to threads

L’interface Future est utilisée afin de vérifier si le traitement a été complété avec succès.

Callable interface comme Runnable, la différence est que Callable peut retourner un résultat et Runnable non. Callable définit une méthode Public Object Call ().

Dans les cas où le programme est petit, il est préféré d’utiliser CachedThreadPool, dans ce cas les submit ne sont pas empilé mais exécuté immédiatement.

L’utilisation des Tasks offre plus de flexibilité pour le programmeur.

# Item 69 : Prefer concurrency utilities to wait and notify

Le package Java.util.concurrent contient trois type de classe, les executors (Abordé au niveau du Item précédent), les Concurents Collection et les synchronisateurs.

C’est impossible d’exclure l’activité de la gestion de concurrence pour les collections dédiées à la concurrence.

L’utilisation de la classe concurrentHashMap au lieu de TableSet et Collections.SynchronizedMap, peut augmenter considérablement les performances d’un programme multithread. (Utilisé les classe du package Concurrent au lieu les collections synchronisé externe).

Quelque classe permet les opérations bloquantes : exemple BlockingQueue ; la plupart des ExecutorService Implement a BlockingQueue.

Les synchronisateurs les plus utilisé sont ; CountDownLatch, Sémaphores et les moins utilisé sont CyclicBarrier and Exchanger.

CountDownLatch : a comme paramètre dans le constructeur un entier qui indique le nombre d’appels de la méthode Count Down avant de libérer les threads en attente.

Toujours utilisé System.nanoTime au lieu de System.currenttimemillis.

Il est plus raisonnable d’utiliser NotifyAll que Notify. On peut utiliser notify dans le cas où tous les threads attendent une condition et seulement un seul peut bénéficier de cette condition.

# Item 70 : Document Thread Safety

Le manque de documentation des classes conçue pour etre utilisé en concurrence, peut engendrer une insuffisance synchronisation, ou bien une excessive synchronisation.

La présence du mot Synchronized dans la déclaration est un détail d’implémentation, et non pas une part de de l’API exporté (il n’indique pas que la méthode est ThreadSafety).

Il y’a plusieurs niveau de ThreadSafety, et pour garantir la bonne utilisation de la concurrence, il faut indiquer le niveau de ThreadSafety :

1. Immutable : n’a pas besoin d’être synchroniser, car la classe ne se change pas, Exemple : String, Long, BigInteger.
2. Unconditionnaly ThreadSafe : la classe contient un niveau de synchronisation suffisant pour quelle travail sans aucun besoin d’une synchronisation externe ; Exemple : Random, ConcurentHashMap.
3. Conditionnaly ThreadSafe : quelque méthode nécessite une synchronisation externe ; exemple de : Collections.synchronized, que leurs iterator nécessite une synchronisation externe.
4. Not ThreadSafe : tous les méthodes de cette classe nécessite une synchronization externe ; Exemple : ArrayList, HashMap.
5. Thread Hostile : ce type n’est pas ThreadSafe même si tous les méthodes sont synchronisée en externe ; il y’a très peu de classe java qui appartient à ce type, ces classes sont engendré par la modification des données static sans synchronisation. Exemple : System.runFinalizersOnExit.

Les 4 premiers types sont couvrés dans le livre Java Concurruncy In Practice  sous les types : Immutable, ThreadSafe (Condi, Uncond), Not ThreadSafe.

Pour les classes Conditionnaly thread Safe, on doit indiquer prudemment quelle séquence nécessite une synchronisation externe, et quelle locks doit etre acquis afin d’exécuter la séquence ; Exemple : Collections.SynchronizedMap ().

Il est préféré d’utiliser une variable privé Lock au lieu du Synchronized afin de synchroniser les méthodes (Locks n’est pas accessible par les clients) le locks est déclarer Final et il est préféré dans la conception des classes dédiés à l’héritage. (cas des classes Unconditonnaly thread Safe)

# Item 71 : Use lazy initialization judiciously

Lazy initialization : est le fait de ne pas initialiser un attribue (Static et non static) que lors du besoin, elle permet de gagner en performance.

Pour augmenter la performance, n’utilisé pas ce principe que dans le cas où vous avez vraiment besoin de le utilisé.

Elle diminue le cout d’initialisation, mail elle augmente le cout de l’accès.

Afin de savoir si elle est nécessaire, on mesure la performance de la classe avec est sans LazyInitialization.

La gestion des attributs laziInitialized est difficile dans un environnement multithread.

Lorsque vous voulez augmentez les performances des static field, utiliser LazyInitializationHolderClass. Cela permet d’initialisé le champ lors de la création de l’instance de la classe interne.

Lorsque vous voulez augmentez les performances des non static Field, utiliser le double check.

Quand on peut initialiser l’attribut plusieurs fois, on peut utiliser Single Check Idiom.

Ces méthodes sont appliquées pour les types primitifs comme pour les types objects.

La classe String utilise la technique Race Single Check pour cacher son HashCode.

Thread.join.