

Jakarta Commons Collections

par [Sébastien Le Ray](#)

Date de publication : 28/06/2005

Dernière mise à jour : 28/06/2005

Nous continuons la série d'articles sur les API Commons du projet Apache Jakarta en nous intéressant aujourd'hui aux commons collections.

- I - Introduction
- II - Concepts
 - II.A - Design-Pattern Decorator
 - II.B - Décorateurs commons collections
 - II.C - Catégories de sous-interfaces communes
- III - Entités support
 - III.A - Comparator
 - III.B - Comparable
- IV - Functors
 - IV.A - Transformer
 - IV.A.1 - Implémentation utilisables directement
 - IV.A.2 - Décorateurs
 - IV.B - Predicate
 - IV.B.1 - Implémentations utilisables directement
 - IV.B.2 - Décorateurs
 - IV.C - Factory
- V - Bag
- VI - BidiMap
- VII - Buffer
- VIII - List
- IX - Map
- X - Set
- XI - Autres
- XII - Remerciements
- XIII - Téléchargements

I - Introduction

Les Commons Collections sont des extensions développées par Jakarta pour compléter le framework Collections du JDK de Sun. Elles définissent un certain nombre d'interfaces ainsi qu'une multitude d'implémentations relatives au comportement et à la manipulation des collections. Nous n'entrerons pas dans le détail des *closures* qui offrent une autre approche de la manipulation des collections.

Après une présentation des concepts clés et des entités supportant le fonctionnement de l'API, nous décrirons, package par package, les fonctionnalités offertes pour chaque type de collection.

Notez que l'organisation des packages peut sembler assez désordonnée. En effet, l'API a été modifiée et le découpage en packages a été entièrement revu. On trouve donc des doublons, des classes qui n'ont pas encore été déplacées mais qui le seront peut-être bientôt. Regardez la JavaDoc, vous verrez qu'un certain nombre de classes sont taggées *deprecated*, ce qui signifie qu'elle vont être déplacées prochainement. Ce tutoriel est basé sur la version 3.1 des commons collections, j'essaierai de mettre à jour ce tutoriel si une nouvelle version sort un jour. Le passage à la version 4 devrait faire le ménage dans les paquetages et n'assurera donc plus la compatibilité avec les versions antérieures à la 3.

Pour certaines entités, vous avez le choix entre un constructeur et une méthode **getInstance**. Il est préférable d'utiliser cette dernière car elle est généralement optimisée et valide les paramètres qui lui sont passés.

II - Concepts

II.A - Design-Pattern Decorator

Afin d'apporter des fonctionnalités supplémentaires, les commons collections utilisent massivement le design-pattern *Decorator* (les entités suivant ce design-pattern seront appelées décorateurs). Les types de décorateurs fournis par les commons collections sont décrits plus loin dans ce document mais une rapide présentation de ce principe peut être utile avant de continuer.

Un décorateur est utilisé pour ajouter des responsabilités à un objet, l'objet décoré, tout en respectant la même interface que celui-ci. Pour une description plus en détails de ce Design-Pattern référez vous à l'[article](#) à ce sujet sur [developpez.com](#).

II.B - Décorateurs commons collections

Les commons collections viennent avec un certain nombre de décorateurs standards. Certains peuvent s'appliquer à toutes les *Collections*, d'autres uniquement à un certain type de *Collection*. Le nom du décorateur est de la forme *TypeDeDécorateurTypeDuDecoré*. Par exemple, un décorateur de type *Typed* pour les *Bags* s'appellera **TypedBag**.

Les décorateurs ne définissent qu'une seule méthode en plus de celles de l'objet décoré : **decorate**, dont les arguments dépendent du type de décorateur. Il existe deux façons d'obtenir un décorateur, soit en appelant cette méthode :

```
Bag bagDecore = SynchronizedBag.decorate(new HashBag()); // Crée un nouveau Bag synchronisé
```

soit en utilisant les classes utilitaires de la forme *TypeEntiteUtils* :

```
Bag bagDecore = BagUtils.synchronizedBag(new HashBag()); // Crée un nouveau Bag synchronisé
```

Voici la description des différents types de décorateurs. Ne vous inquiétez pas si vous ne comprenez pas tout tout de suite, cela deviendra plus clair dans la suite de cet article.

- **Synchronized** : La collection décorée voit toutes ses méthodes synchronisées (utile en environnement multi-threads) ;
- **Unmodifiable** : La collection décorée soulève une **UnsupportedOperationException** lors de l'appel à une quelconque méthode pouvant la modifier (utile pour les getters). Toutes les entités de ce type sont indirectement des instances de l'interface **Unmodifiable** ;
- **Predicated** : L'ajout d'un élément à la collection est soumis à sa conformance par rapport à un *Predicat* ;
- **Typed** : La collection décorée n'accepte que les éléments du type spécifié lors de sa décoration. Obsolète dans les versions de Java post 1.5 (apparition des templates). Si l'élément n'est pas du bon type, une **IllegalArgumentException** est levée ;
- **Transformed** : La collection décorée transforme les données avant l'ajout *via* le *Transformer* spécifié lors de sa décoration ;
- **FixedSize** : Empêche les ajouts/suppressions mais permet les modifications (*ie set* dans le cas d'une **List**, **put** si l'objet existe déjà pour une **Map**) ;
- **Lazy** : On précise une *Factory* qui crée un objet au moment de la demande (**get**) s'il n'existe pas.

Voici un tableau résumant les décorateurs disponibles pour chaque interface majeure, ces interfaces seront détaillées plus loin dans ce document) :

	Bag	Buffer	Communs(1)	List	Map	BidiMap	Set
Synchronized	✓	✓	✓ (2)	✓ (2)	✗ (2)	✗	✓ (2)
Unmodifiable	✓	✓	✓ (2)	✓ (2)	✓ (2)	✓	✓ (2)
Predicated	✓	✓	✓	✓	✓	✗	✓
Typed	✓	✓	✓	✓	✓	✗	✓
Transformed	✓	✓	✓	✓	✓	✗	✓
FixedSize	✗	✗	✗	✓	✓	✗	✗
Lazy	✗	✗	✗	✓	✓	✗	✗

- (1) : Utilise l'interface **java.util.Collection**, on perd les méthodes spécifiques.
- (2) : Disponible directement *via* JDK à partir de la version 1.4.

Il existe des décorateurs qui sont propres à un type d'interface en particulier, nous les présenterons en même temps que l'interface concernée.

II.C - Catégories de sous-interfaces communes

Certaines interfaces sont dérivées en sous-interfaces destinées à assurer un comportement particulier qui ne pourrait être mis en place *via* un décorateur. Bien qu'elles soient propres à leur interface mère, on peut classer ces sous interfaces en deux grands types dont le comportement est comparable. Ces sous-interfaces sont reconnaissables à leur nom : *Sorted* et *Ordered*.

Les collections de types *Sorted* sont similaires à celles proposées par le Framework Collections du JDK (**java.util.SortedMap** et **java.util.SortedSet**). Elles assurent que les éléments sont maintenus triés (soit par le *Comparator* passé au constructeur, soit selon *l'ordre naturel* auquel cas ils doivent implémenter l'interface **java.util.Comparable**). L'API commons collections introduit les interfaces **SortedBag** et **SortedBidiMap**.

Les collections de type *Ordered* sont introduites par les commons collections. Elles assurent que le parcours de la collection *via* un **Iterator** se fait dans un certain ordre. Il est possible de parcourir ce type d'entités dans les deux sens en utilisant un **OrderedIterator**, elles permettent également l'accès à la première et à la dernière clé *via* les méthodes **firstKey()** et **lastKey()**. L'API commons collections définit les interfaces **OrderedMap** et **OrderedBidiMap**.

III - Entités support

Les commons collections utilisent un certain nombre d'entités annexes utilisées au sein des méthodes. Nous les présentons ici.

III.A - Comparator

Cette interface n'est pas définie directement par les commons collections, mais dans le package **java.util**. Néanmoins, elle est largement utilisée par l'API de Jakarta. Comme son nom l'indique, elle sert à comparer deux objets. L'interface **Comparator** définit deux méthodes :

- **compare(Object o1, Object o2)** : renvoie un entier inférieur à zéro si $o1 < o2$, zéro si $o1 = o2$ (au sens logique) et un entier supérieur à zéro si $o1 > o2$;
- **equals(Object o)** : renvoie true si l'autre objet est un comparateur qui implique le même ordre que celui pour lequel la méthode est appelée.

III.B - Comparable

Alors que **Comparator** est une interface destinée à donner des entités distinctes, **Comparable** est une interface destinée à être implémentée par un objet classique. C'est-à-dire que si l'on a un objet **Chaise**, **Comparator** donnera un objet de la forme **ChaiseComparator** qui compare deux chaises, alors que **Comparable** sera implémentée directement par **Chaise** ce qui permettra de comparer directement une *Chaise* à une autre.

Comparable ne définit qu'une seule méthode, **compareTo(Object o)** qui retourne un entier négatif si l'objet considéré est inférieur à l'objet passé en paramètre, zéro s'il sont égaux et un entier positif si l'objet considéré est plus grand que l'objet passé en paramètre.

IV - Functors

Les *functors* sont un ensemble d'interfaces destinées à apporter des fonctionnalités supplémentaires aux collections. Les interfaces sont définies dans le package **org.apache.commons.collections**, les implémentations dans **org.apache.commons.collections.functors**.

IV.A - Transformer

L'interface **Transformer** permet de transformer un objet en un autre selon des critères que l'utilisateur définit dans les implémentations.

La seule méthode définie par cette interface est **transform(Object o)** qui doit renvoyer l'objet correspondant à la transformation de celui passé en paramètre.

L'API commons collections propose plusieurs implémentations de cette interface. Certaines sont utilisables directement, d'autres sont des décorateurs.

IV.A.1 - Implémentations utilisables directement

Les *Transformer* dont le constructeur ne prend pas de paramètre sont des Singletons, ils disposent donc d'une méthode **getInstance()** et ne peuvent pas être instanciés. Les autres *Transformers* peuvent être obtenus soit *via* une instanciation classique, soit en appelant la méthode **getInstance** avec les mêmes paramètres que ceux que l'on aurait passés au constructeur.

IV.A.2 - Décorateurs

- **ChainedTransformer** : Utilise le Design-Pattern "Chaîne de Responsabilités", c'est-à-dire qu'un premier *Transformer* est appelé, son résultat est passé à un second et ainsi de suite. Le résultat de la transformation est celui du dernier *Transformer* de la chaîne. Les *Transformers* à appliquer sont passés en paramètres au constructeur ;
- **SwitchTransformer** : Ce *Transformer* permet d'appliquer une transformation selon des *Predicats*, l'objet à transformer est évalué par une série de *Predicats*, lorsqu'un *Predicat* renvoie true, la transformation correspondante est appliquée. Si aucun *Predicat* ne renvoie true, une transformation par défaut est appliquée. Les *Predicats* et *Transformers* sont passés au constructeurs soit *via* une **Map** dont les clés sont les *Predicats* et les valeurs les *Transformers* (le *Transformer* par défaut a une clé null), soit *via* deux tableaux (le *Transformer* par défaut est un troisième paramètre).

Comme pour les *Transformers* classiques, les décorateurs peuvent être instanciés directement, ou il est possible de passer par une méthode **getInstance**. Notez que cette dernière offre plus de fonctionnalités.

IV.B - Predicate

L'interface **Predicate** est définie dans le package **org.apache.commons.collections**. Elle est utilisée pour évaluer un objet, c'est-à-dire vérifier qu'il remplit bien certaines conditions (à définir par l'utilisateur).

La seule méthode définie par l'interface est **evaluate(Object o)** qui renvoie true si l'objet remplit les conditions du *Predicat*, false sinon.

L'API commons collections définit un certain nombre de *Predicats*. Certains sont utilisables directement, d'autres

sont des décorateurs.

IV.B.1 - Implémentations utilisables directement

- **EqualPredicate, IdentityPredicate** : Les constructeurs de ces deux classes prennent un objet en paramètre. L'évaluation renvoie true si et seulement si l'objet passé à **evaluate** est égal au précédent (comparaison **equals**) ou identique (**==**) ;
- **FalsePredicate, TruePredicate** : L'évaluation renvoie toujours false ou toujours true, respectivement ;
- **NullPredicate, NotNullPredicate** : L'évaluation renvoie true selon que l'objet évalué est null ou non, respectivement ;
- **ExceptionPredicate** : L'évaluation soulève toujours une **FunctionException** ;
- **InstanceOf** : Le constructeur prend un objet de type **Class**, l'évaluation renvoie true si l'objet évalué est une instance de la classe passée au constructeur ;
- **UniquePredicate** : L'évaluation renvoie true si l'élément n'a pas déjà été évalué auparavant
- **TransformerPredicate** est particulier, son constructeur prend en paramètre un *Transformer* qui renvoie une instance de **Boolean** et c'est le résultat de ce transformer qui est renvoyé.

La plupart des *Predicats* utilisables directement sont des Singletons, ils possèdent donc tous une méthode **getInstance** et ne peuvent pas être instanciés. **EqualPredicate, IdentityPredicate, InstanceOfPredicate, TransformerPredicate UniquePredicate** disposent d'une telle méthode mais ne sont pas des singletons, les paramètres indiqués pour le constructeur peuvent également être passés à **getInstance**.

Vous pouvez bien sûr définir vos propres *Predicates* en implémentant l'interface.

IV.B.2 - Décorateurs

Les *Predicats* décorateurs permettent de combiner plusieurs *Predicats* en effectuant des opérations logiques.

- **AnyPredicate** décore un tableau de *Predicates*, l'évaluation renvoie true dès que l'un des *Predicates* décorés renvoie true. **OrPredicate** est l'équivalent mais le constructeur ne prend que deux *Predicates* ;
- **AllPredicate** décore un tableau de *Predicates*, l'évaluation renvoie true uniquement si tous les *Predicates* sont vérifiés. **AndPredicate** est l'équivalent mais le constructeur ne prend que deux *Predicates* ;
- **NullsTruePredicate, NullsFalsePredicate** et **NullsExceptionPredicate** permettent de décorer un *Predicat* afin de paramétrer son comportement lorsqu'un élément null est évalué, comme leur nom l'indique, ils renverront true, false ou soulèveront une **FunctionException**, respectivement ;
- **NonePredicate** décore un tableau de *Predicates*, l'évaluation renvoie true uniquement si aucun des *Predicates* ne renvoie true ;
- **NotPredicate** décore un *Predicat*, l'évaluation renvoie l'inverse de celle du *Predicat* décoré ;
- **OnePredicate** décore un tableau de *Predicates*, l'évaluation renvoie true uniquement si un seul des *Predicates* est vérifié (équivalent d'un ou exclusif) ;
- **TransformedPredicate** décore un *Predicat* et prend un *Transformer* en paramètre, celui-ci est appliqué à l'objet avant qu'il soit évalué par le *Predicat* décoré.

IV.C - Factory

L'interface **org.apache.commons.collections.Factory** met en application le design-pattern du même nom. Elle ne définit d'une seule méthode : **create** qui doit renvoyer un objet. C'est au développeur de définir les caractéristiques de cet objet.

L'API commons collections définit un certain nombre d'implémentations de référence situées dans le package **org.apache.commons.collections.functors** :

- **ConstantFactory** : renvoie toujours l'objet qui a été passé au constructeur ;
- **ExceptionFactory** : l'appel à **create** soulève toujours une **FunctorException** ;
- **InstantiateFactory** : lors de l'appel à **create**, une nouvelle instance de la classe passée au constructeur est créée en appelant un constructeur qui prend en paramètres les différentes entités passées au constructeur de la *Factory* ;
- **PrototypeFactory** est utilisée pour obtenir une *Factory* qui renvoie des instances créées par duplication. Pour cela, la méthode **getInstance** prend un objet en paramètre (le prototype), c'est cet objet qui sera dupliqué lors de l'appel à **create**. Pour cela, le prototype doit disposer d'une méthode clone publique, d'un constructeur par copie public ou étendre l'interface **java.io.Serializable**. Selon ce qui est disponible, la *Factory* renvoyée par **getInstance** sera une **PrototypeCloneFactory**, une **InstantiateFactory** ou une **PrototypeSerializationFactory**.

V - Bag

L'interface **org.apache.commons.collections.Bag** définit une collection qui peut contenir plusieurs fois le même objet et permet de compter le nombre de fois où un objet apparaît. Certaines méthodes sont en violation par rapport au contrat spécifié par l'interface **Collection** du JDK ce qui peut dérouter les habitués. Voici une rapide présentation des méthodes :

- **add(Object o, int nb)** : Ajoute *nb* copies de l'objet dans le *Bag*, si le nombre à ajouter n'est pas précisé, n'ajoute qu'une seule copie. Si l'objet n'existait pas déjà dans le *Bag* renvoie true, false sinon. **Ce n'est pas conforme aux spécifications de Collection** ;
- **containsAll(java.util.Collection c)** : Renvoie true si le *Bag* contient tous les objets contenus dans *c*. Si un objet est présent *n* fois dans *c*, il doit l'être **au moins** *n* fois dans le *Bag* pour que la fonction renvoie true. **Ce n'est pas conforme aux spécifications de Collection** ;
- **iterator()** : Renvoie un itérateur qui parcourt toutes les copies contenues dans le *Bag*
- **remove(Object o, int nb)** : Retire *nb* copies de l'objet indiqué ou toutes les occurrences si *nb* n'est pas précisé (**Ce qui n'est pas conforme aux spécifications de Collection**) ;
- **removeAll(java.util.Collection c)/retainAll(java.util.Collection c)** : Respectivement, supprime ou ne conserve que les éléments contenus dans *c* (si un élément est présent *n* fois dans *c*, il sera supprimé ou conservé *n* fois exactement). **Ce n'est pas conforme aux spécifications de Collection** ;
- **size()** : Renvoie le nombre d'éléments contenus dans le *Bag*, tient compte des objets identiques (*ie.* si un objet est présent plusieurs fois, il compte plusieurs fois).

Les autres méthodes héritées de collections n'ont pas de particularité. En plus de ces méthodes classiques, **Bag** définit d'autres méthodes qui lui sont propres :

- **getCount(Object o)** : Renvoie le nombre d'occurrences de l'objet *o* dans le *Bag* ;
- **uniqueSet()** : Renvoie un **java.util.Set** qui ne contient qu'un exemplaire de chaque élément du *Bag*.

Les entités liées à l'interface **Bag** sont définies dans le package **org.apache.commons.collections.bag**.

L'implémentation de référence de l'interface **Bag** est la classe **HashBag**. Celle-ci utilise en interne une *Map* pour effectuer la correspondance objet <-> nombre d'occurrences.

Pour ce qui est de **Sortedbag**, l'implémentation de référence est **TreeBag**, son comportement est strictement identique au **HashBag** (ils étendent tous deux **AbstractMapBag**) si ce n'est qu'il utilise une **TreeMap** en interne.

VI - BiDiMap

L'interface **org.apache.commons.collections.BiDiMap** définit une *Map* qui permet l'accès aussi bien dans le sens clé -> valeur que dans le sens valeur -> clé. Cela impose une restriction : une valeur ne peut se trouver plusieurs fois dans la *Map*. L'interface définit des méthodes supplémentaires spécifiques :

- **getKey(Object valeur)** : renvoie la clé associée à la valeur passée en paramètre ou null si la clé n'existe pas ;
- **inverseBiDiMap()** : les valeurs deviennent les clés et les clés les valeurs ;
- **removeValue(Object valeur)** : supprime la paire clé-valeur associée à la valeur.

Par ailleurs, le comportement de **put(Object cle, Object value)** est modifié afin de garantir l'unicité des clés et des valeurs. Si la clé existe déjà, la valeur correspondante est remplacée, si la valeur existe déjà, la clé correspondante est remplacée.

Les différentes entités liées à l'interface **BiDiMap** se trouvent dans le package **org.apache.commons.bidimap**.

L'implémentation de référence pour l'interface **BiDiMap** est la classe **DualHashBiDiMap** qui utilise deux **HashMap** en interne afin de faire la correspondance clé <-> valeur.

La classe **BiDiMap** a son équivalent *Sorted* avec la classe **SortedBiDiMap**. Contrairement à ce que l'on pourrait croire, elle n'utilise pas deux **TreeMap** en interne mais bénéficie d'un algorithme de recherche entièrement recodé, basé sur la recherche au sein d'une **TreeMap**. Cette approche a été choisie afin d'éviter le supplément de mémoire nécessaire au stockage de deux **TreeMap** lorsque le nombre de données est important.

VII - Buffer

L'interface **org.apache.commons.collections.Buffer** définit une *Collection* pour laquelle l'ordre de suppression des éléments est bien défini. Cela permet de créer des files FIFO ou des files à priorités. Deux méthodes spécifiques sont définies en dehors de celles de **Collection**

- **get()** : renvoie le prochain objet, sans le supprimer ;
- **remove()** : renvoie et supprime le prochain objet.

Le nombre d'implémentations de l'interface **Buffer** est assez important, elles sont regroupées avec les autres entités liées à l'interface *Buffer* dans le package **org.apache.commons.collections.buffer**. Nous les décrivons ici en insistant sur les particularités de chacune. Excepté lorsque le *Buffer* est décoré par un **BlockingBuffer**, toute tentative de **get()** ou de **remove()** sur un *Buffer* vide soulève une **BufferUnderFlowException**.

FifoBuffer définit une file FIFO (First In, First Out, file d'attente), c'est-à-dire que les éléments sont supprimés dans l'ordre d'ajout. On trouve trois variations de l'implémentation :

- **UnboundedFifoBuffer** : file redimensionnée à la demande, au fur et à mesure des ajouts. L'accès et la suppression se font en temps constant, l'ajout également sauf dans le cas du redimensionnement ;
- **BoundedFifoBuffer** : la taille de la file est définie une fois pour toute lors de la déclaration (32 par défaut). Lorsque la file est pleine (la méthode **isFull** permet de savoir si c'est le cas), toute tentative d'ajout soulève une **BufferOverflowException**. La contrepartie est que les ajouts se font également en temps constant. Il est interdit d'ajouter des valeurs null au *Buffer* ;
- **CircularFifoBuffer** : type particulier de **BoundedFifoBuffer**, lorsque l'on ajoute un élément à un *Buffer* déjà plein, la première valeur est écrasée.

L'itération se fait dans le même ordre que la suppression.

L'implémentation de la file à priorité est fournie par la classe **org.apache.commons.collections.PriorityBuffer** qui maintient les éléments triés grâce au *Comparator* passé au constructeur ou s'il n'y en a pas, selon l'ordre naturel (les éléments doivent alors implémenter **java.util.Comparable**). Notez que contrairement aux *FifoBuffers*, l'itération ne se fait pas dans l'ordre de suppression.

Aucune des implémentations précédentes n'est synchronisée, il est possible de passer par le décorateur **SynchronizedBuffer** pour pallier à cela.

Les *Buffers* disposent d'un décorateur qui leur est propre, il s'agit du **BlockingBuffer**. Celui-ci permet d'implémenter un schéma producteur/consommateur similaire aux pipes sous Unix. Lors d'un **get()** ou d'un **remove()** sur un *Buffer* vide, le *Thread* se met en attente jusqu'à ce qu'un élément soit disponible ou qu'il soit interrompu. Cette implémentation est synchronisée.

Enfin, l'interface **Buffer** est implémentée par un nouveau type de pile, la classe **org.apache.commons.collections.ArrayStack**. Celle-ci est identique à la classe **java.util.Stack** mais utilise une **ArrayList** en interne, ses méthodes ne sont donc pas synchronisées et elle est donc plus rapide que l'implémentation originale.

VIII - List

L'API commons collections définit diverses implémentations de l'interface **java.util.List**. Celles-ci se trouvent dans le package **org.apache.commons.collections.list**.

CursorableLinkedList définit un type de liste pour lequel les modifications apportées à la liste sont reflétées au niveau des itérateurs et réciproquement.

NodeCachingLinkedList est une liste qui implémente un système de cache. Lors de la suppression d'un noeud de la *LinkedList*, celui-ci est mis en cache s'il reste de la place. Lors de l'ajout d'un noeud, le cache est interrogé, si un noeud est disponible il est supprimé du cache et renvoyé, sinon, un nouveau noeud est créé.

TreeList est une implémentation de **List** optimisée pour l'ajout et la suppression à n'importe quel indice. Néanmoins, l'**ArrayList** est plus efficace pour les autres opérations et elle occupe moins de place en mémoire. La **TreeList** est donc une implémentation à privilégier dans le cas de mises à jour fréquentes de la liste (insertions et non pas ajouts en fin).

FixedSizeList est un décorateur qui interdit toute opération susceptible de modifier la taille de la liste (famille des **add** et des **remove**) en soulevant une **UnsupportedOperationException**.

SetUnique est un décorateur qui empêche les doublons (à la manière d'un **Set**) tout en permettant de conserver les propriétés des listes (notamment au niveau de l'itération).

IX - Map

L'API commons collections introduit plusieurs améliorations en ce qui concerne les *Maps* (définies dans le **org.apache.commons.collections.map**). Parmi celles-ci, on trouve un nouveau concept : le **MapIterator**. Comme son nom l'indique, il s'agit d'un itérateur qui permet de parcourir une *Map*. En plus des méthodes traditionnelles **hasNext**, **next** et **remove**, l'interface définit les méthodes **getKey**, **getValue** qui renvoient respectivement la clé et la valeur correspondantes à la position en cours, **setValue(Object o)** qui positionne la valeur de la position en cours. A noter que **next** avance d'une position et renvoie la valeur de la clé correspondant à la position suivante. L'interface **OrderedMapIterator** permet de parcourir une *Map* ordonnée et définit les méthodes **hasPrevious** et **previous** dont les noms sont explicites.

Les deux implémentations de référence permettant d'utiliser ces itérateurs sont respectivement **org.apache.commons.collections.HashMap** et **org.apache.commons.collections.LinkedMap**. La classe **LinkedMap** maintient les éléments dans l'ordre d'insertion.

Différentes implémentations répondant à des besoins particuliers sont également fournies :

- **CaseInsensitiveMap** : Effectue une recherche par clé sans tenir compte de la casse (la méthode **toString** de la clé est utilisée) ;
- **IdentityMap** : *Map* dans laquelle la comparaison des clés se fait avec l'opérateur **==** au lieu de la méthode **equals** ;
- **Flat3Map** : *Map* optimisée pour contenir trois éléments ou moins. Les performances sont entre 0 et 10% meilleures qu'une **HashMap** pour les **gets** et quatre fois plus pour les **puts** avec moins de

trois éléments. Au-delà de trois éléments, les performances sont 5% moins bonnes ;

- **LRUMap** : *Map* de taille fixe (par défaut 100 éléments). Lorsqu'un élément est ajouté alors que la *Map* est pleine, l'élément qui n'a pas été utilisé depuis le plus longtemps (Least Recently Used) est supprimé. Seules les opérations **put** et **get** mettent à jour l'utilisation d'un élément.
- **MultiKeyMap** : *Map* dans laquelle les objets sont identifiés par un ensemble de clés. Il est possible de passer les objets constituant la clé directement à **put** ou à **get** (jusqu'à cinq objets) ou d'utiliser la classe **MultiKey** qui permet de créer une clé composée d'un nombre arbitraire d'objets (les méthodes **put** et **get** avec respectivement deux et un paramètre n'accepte qu'une instance de cette classe en tant que clé) ;
- **SingletonMap** : *Map* conçue pour ne contenir qu'une seule paire clé/valeur, toute tentative de suppression se solde par une **UnsupportedOperationException**. La méthode **put** n'est autorisée que si elle concerne la clé déjà présente ;
- **StaticBucketMap** : *Map* destinée aux environnements multi-threads, les méthodes sont synchronisées tout en minimisant l'impact sur les performances. La contrepartie est que les méthodes **putAll** et **retainAll** ne sont pas atomiques et se comportent de façon aléatoire lorsqu'elles sont utilisées en concurrence ;
- **BeanMap** : Cette *Map* prend un objet en constructeur (censé être un *Bean*). Les appels à **get** et **put** sont en fait relayés vers les méthodes **getNomClé** et **setNomClé** du *Bean*.

En plus des décorateurs communs présentés au début de cet article, les commons collections proposent des décorateurs spécifiques aux *Maps* :

- **CompositeMap** : permet d'offrir une vue unifiée de plusieurs *Maps* passées au constructeur ou à la méthode **addComposited**. Le constructeur peut prendre en plus une implémentation de **CompositeMap.MapMutator** qui définit le comportement à adopter lors de l'ajout d'une valeur ou lors de la présence d'une collision dans les clés des différentes *Maps* décorées ;
- **ListOrderedMap** : L'ordre de parcours de la *Map* décorée via un *Iterator* se fait dans l'ordre d'ajout.

L'interface **MultiMap** permet de faire correspondre plusieurs valeurs à une même clé. Les clés renvoient en fait à des collections auxquelles sont ajoutées les valeurs lors de l'appel de **put**. L'implémentation de référence de la **MultiMap** est la classe **MultiHashMap**.

X - Set

L'API commons collections ne définit pas de nouvelle interface pour les *Set* mais offre divers décorateurs, définis dans la package **org.apache.commons.collections.set** :

- **CompositeSet** : comme pour **CompositeMap**, offre une vue unifiée de plusieurs *Set*. Afin de gérer les collisions, une implémentation de **CompositeSet.SetMutator** doit être passée en paramètre, si ce n'est pas le cas, les opérations d'ajout soulèveront une **UnsupportedOperationException**. La méthode **remove** supprime le premier élément qu'elle trouve qui correspond à l'objet qu'on lui passe en paramètre ;
- **ListOrderedSet** : décorateur assurant que l'ordre d'ajout des éléments est utilisé pour l'itération ;
- **MapBackedSet** : permet de gérer une *Map* comme un **Set**. Les objets du *Set* sont les clés de la *Map*. Permet par exemple de créer un **WeakSet** à partir d'une **WeakHashMap**.

XI - Autres

Il existe encore d'autres "familles" d'entités qui ajoutent des fonctionnalités à différentes catégories de collections. Nous en présentons ici quelques unes.

FastArrayList, **FastHashMap**, **FastTreeMap** sont destinées à être utilisées dans des environnements multi-threads où les opérations de lecture sont beaucoup plus nombreuses que celles de modifications. Lors de l'instanciation, la collection est en mode slow, c'est-à-dire que toutes les méthodes sont synchronisées. Lorsque la collection a été remplie, il est préférable de basculer en mode fast. Dans ce mode, les accès qui ne modifient pas la collection ne sont pas synchronisés. Lors d'un accès qui modifie la collection, celle-ci est clonée, la modification est effectuée sur le clone, puis c'est ce clone qui est utilisé.

XII - Remerciements

Merci à vedaer pour sa relecture.

XIII - Téléchargements

- Les commons collections sont disponibles sur le site du projet [Jakarta](#) ;
- L'article au format [HTML zippé](#) ([mirroir](#)) ;
- L'article au format [PDF](#) ([mirroir](#)).