Java : les Collections

1. Les généralités

1.1. Les caractéristiques

Les **collections** sont des classes prédéfinies de Java qui permettent de **stocker des objets, sans qu'il soit besoin d'en fixer a priori le nombre.**

La définition d'une collection est donc une définition très générale. On verra ici, par exemple, que certaines collections acceptent les doublons alors que d'autres non et que certaines collections sont ordonnées par défaut alors que d'autres non.

Pour toutes les collections il existe néanmoins de base des méthodes permettant d'ajouter des éléments, de supprimer des éléments, de vérifier la présence d'objets dans la collection, de parcourir la collection et d'autres méthodes agissant sur la totalité de la collection (cf. A p. 9).
Les propriétés communes à toutes les collections sont décrites dans l'interface[1] Collection.

On ne s'intéressa ici qu'aux deux familles de collections suivantes : les listes et les ensembles.

Les **listes** sont des collections « **ordonnées »** d'objets, **indicés à partir de 0, dans lesquelles les doublons sont autorisés.**
Du fait que les éléments d'une liste soient ordonnés, il existe des méthodes spécifiques aux listes : des méthodes permettant d'ajouter, de supprimer ou de substituer des éléments à un indice donné, d'accéder directement aux éléments à partir d'un indice, de parcourir la liste à partir d'un indice donné et d'obtenir des sous-listes comportant les éléments compris entre deux indices de la liste (cf. B p. 10).
Les propriétés communes à toutes les listes sont décrites dans l'interface List.

On s'intéressa ici à deux implémentations de listes : **la classe ArrayList et la classe LinkedList**.

Les **ensembles** sont des collections **dans lesquelles les doublons ne sont pas autorisés**. Par défaut, **aucun ordre** n'est défini sur un ensemble.
Il existe cependant **des ensembles dans lesquels les éléments sont triés**, ce qui permet de définir des méthodes permettant d'accéder au plus grand ou au plus petit élément d'un ensemble et des méthodes permettant d'obtenir des sous-ensembles comportant les éléments compris entre deux éléments de l'ensemble (cf. C p. 11).
Les propriétés communes à tous les ensembles sont décrites dans l'interface Set, celles communes aux ensembles contenant des éléments triés sont décrites dans l'interface SortedSet.

On s'intéressa ici à deux implémentations d'ensembles : **la classe HashSet et la classe TreeSet**.

1.2. La méthode equals

De manière générale, c'est la méthode equals qui est utilisée pour comparer l'identité de deux objets.
Par exemple, elle est utilisée lors de la recherche de l'indice d'un objet dans une liste (méthode indexOf).

Par défaut, toutes les classes héritent cette méthode equals de la classe Object, elle ne retourne true que si les deux objets testés ont la même adresse mémoire. Ainsi deux objets différents, même s'ils possèdent les mêmes attributs avec les mêmes valeurs, ne sont pas « égaux » selon cette définition de la méthode.

[1] Une interface est un type abstrait qui permet de décrire des propriétés communes à plusieurs classes. Lorsqu'une classe implémente une interface, elle doit redéfinir (sauf exception) toutes les méthodes déclarées dans l'interface. La notion d'interface sera approfondie dans un autre chapitre du cours.

Il est donc recommandé de redéfinir cette méthode, lorsque l'on veut utiliser une collection pour stocker des instances d'une classe personnelle.

Pour écrire une méthode equals pertinente, le contrat à respecter est le suivant :

- réflexivité : quel que soit l'objet x non null, x.equals(x) doit toujours retourner true,
- symétrie : quels que soient les objets x et y non null, x.equals(y) doit retourner true si et seulement si y.equals(x) retourne true,
- transitivité : quels que soient les objets x, y et z non null, si x.equals(y) retourne true et y.equals(z) retourne true alors x.equals(z) doit retourner true,
- consistance : quels que soient les objets x et y non null, plusieurs appels de x.equals(y) doivent toujours retourner true ou toujours retourner false (sauf bien sûr si les attributs utilisés dans la méthode equals ont changé de valeur),
- pour tout objet x non null, x.equals(null) doit toujours retourner false.

1.3. Les itérateurs

Toutes les collections peuvent être parcourues. Un **itérateur** est un outil permettant de **parcourir une collection de manière séquentielle**, sans se soucier ni de la classe des éléments de cette collection, ni du nombre d'éléments de cette collection.
Toutes les collections implémentent une méthode retournant un itérateur (méthode iterator, cf. A.4 p.9).
Lors de sa création l'itérateur est placé au début de la collection. Il permet ensuite d'accéder aux éléments de la collection, les uns après les autres. Il ne sert qu'une seule fois, pour re-parcourir la collection il faut créer un autre itérateur.
Il existe des méthodes permettant d'obtenir le prochain élément de la collection, de vérifier s'il existe encore des éléments à parcourir, de supprimer le dernier élément parcouru (cf. D p. 12).

Pour le traitement des listes, il est également possible d'utiliser un itérateur spécifique : un itérateur qui permet de **parcourir une liste de manière bidirectionnelle** (i.e. dans les deux sens). Toutes les listes implémentent deux méthodes retournant un tel itérateur (méthodes listIterator, cf. B.4 p. 10) : l'une permet d'obtenir un itérateur bidirectionnel placé au début de la liste, l'autre permet d'obtenir un itérateur bidirectionnel placé à un indice donné.
Il existe des méthodes supplémentaires spécifiques à ces itérateurs. Elles permettent de parcourir la liste en sens inverse, d'obtenir l'élément précédent de la liste ou de vérifier s'il en existe un, d'obtenir l'indice du prochain élément ou de l'élément précédent, d'ajouter ou de substituer un élément à la position courante de l'itérateur (cf. D.2 p. 12).

Les propriétés des itérateurs sont décrites dans les interfaces Iterator et ListIterator.

1.4. Les vues

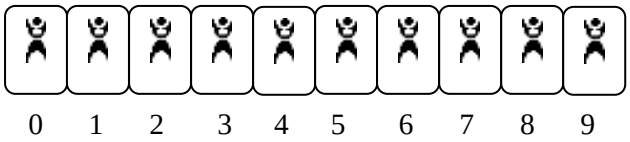
Une vue est une sous-collection d'une collection. Il existe des méthodes permettant d'obtenir des sous-listes de listes (cf. B.5 p. 10), il existe des méthodes permettant d'obtenir des sous-ensembles d'ensembles triés (cf. C.2 p. 11), il existe également des méthodes spécifiques aux instances de la classe TreeSet (cf. H.5 p. 15).

Une vue reste toujours liée à la collection dont elle est extraite, **une vue n'est pas une copie de la collection d'origine**. Ainsi, toute modification effectuée sur la vue (en particulier un ajout ou une suppression) est aussi effectuée dans la collection d'origine, et vice versa.

2. Les classes

2.1. La classe ArrayList

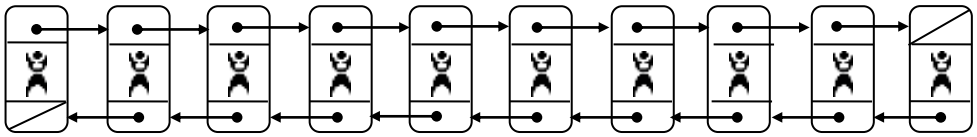
La classe ArrayList permet de représenter **des listes** dans lesquelles les éléments sont stockés dans un **tableau redimensionnable** (i.e. un tableau dont la taille est dynamique).
Bien que l'on puisse en fixer **la taille initiale** (« initial capacity »), le tableau se redimensionne automatiquement, si besoin, lors d'ajouts de nouveaux éléments.



Outre les méthodes décrites pour les collections et les listes, il existe des méthodes particulières à la classe ArrayList : des constructeurs permettant de créer une instance vide en donnant ou non une taille initiale, un constructeur permettant de créer une instance à partir d'une autre collection et des méthodes permettant de réduire ou d'augmenter la taille du tableau (cf. E p. 13).

2.2. La classe LinkedList

La classe LinkedList permet de représenter **des listes** dans lesquelles les éléments sont stockés dans une **liste doublement chaînée**.



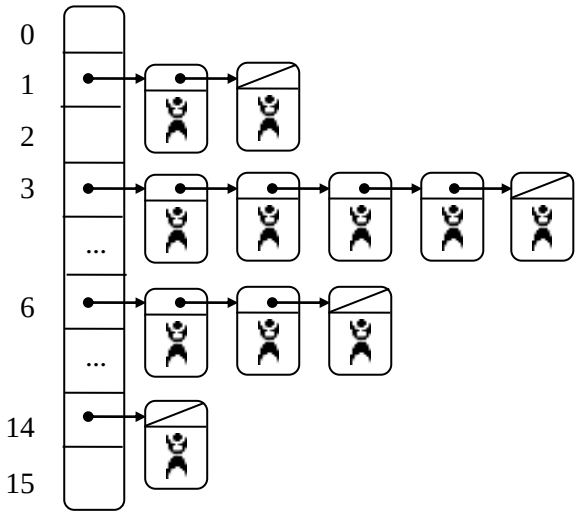
Outre les méthodes décrites pour les collections et les listes, il existe des méthodes particulières à la classe LinkedList : un constructeur permettant de créer une instance vide, un constructeur permettant de créer une instance à partir d'une autre collection, des méthodes permettant d'ajouter ou de supprimer un élément au début ou à la fin de la liste, des méthodes permettant d'accéder au premier ou au dernier élément de la liste (cf. F p. 13).
Ces méthodes permettent d'utiliser aussi cette classe pour gérer une liste d'éléments sous forme de pile ou de file.

2.3. La classe HashSet

La classe HashSet permet de représenter **des ensembles** dans lesquels les éléments sont stockés dans une **table de hachage**.

Deux informations caractérisent une table de hachage : **la taille initiale** de la table (« initial capacity ») et **son facteur de charge** (« load factor », rapport entre le nombre d'éléments stockés dans la table et la taille de la table). Lors d'ajouts de nouveaux éléments, si le seuil défini par le facteur de charge est atteint, il y a automatiquement augmentation de la taille de la table et redistribution des éléments. On parle alors de rehachage.

Dans une instance de la classe HashSet les éléments ne sont pas ordonnés, **l'ordre d'itération** de ces éléments n'est donc **pas garanti**.



Outre les méthodes décrites pour les collections, il existe plusieurs constructeurs dans la classe HashSet : des constructeurs permettant de créer une instance vide en précisant ou non une taille initiale et un facteur de charge, et un constructeur permettant de créer une instance à partir d'une autre collection (cf. G p. 14).

Lorsqu'un élément est ajouté dans une instance de la classe HashSet, on calcule sa valeur de hachage et on vérifie que l'élément n'est pas déjà dans la table (méthodes hashCode et equals).

Par défaut, toutes les classes héritent de la méthode hashCode de la classe Object, elle retourne une valeur qui est fonction de l'adresse mémoire de l'objet.
Il est donc recommandé de redéfinir cette méthode, lorsque l'on veut utiliser une instance de la classe HashSet pour stocker des instances d'une classe personnelle.

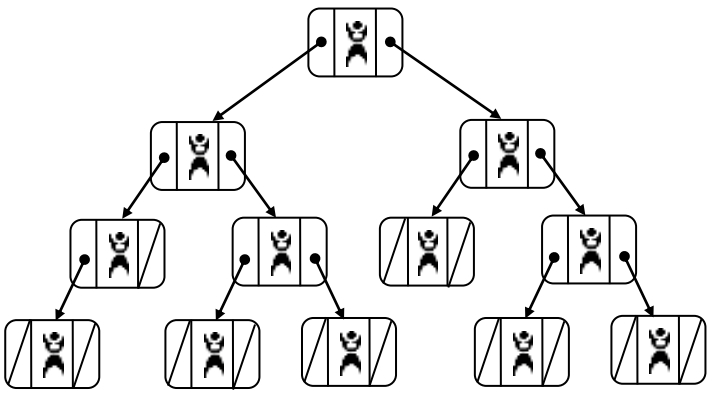
Le contrat à respecter est le suivant :

- à un objet donné ne doit correspondre qu'une seule valeur de hachage,
- si deux objets sont égaux (selon la méthode equals) ils doivent avoir la même valeur de hachage (i.e. la méthode equals doit être consistante avec la méthode hashCode).

On remarque que cela n'implique pas que si deux objets sont différents (selon la méthode equals) leur valeur de hachage est différente, mais cette contrainte supplémentaire permet d'augmenter les performances de la table de hachage.

2.4. La classe TreeSet

La classe TreeSet permet de représenter des **ensembles triés** dans lesquels les éléments sont stockés dans un **arbre binaire de recherche équilibré**.



Pour construire un tel arbre, il doit exister une relation d'ordre entre les éléments stockés dans l'arbre, i.e. les éléments doivent être **comparables**. Il existe deux moyens de définir cette relation d'ordre : définir un ordre naturel entre les éléments ou écrire un comparateur (cf. ci-dessous).

Dans une instance de la classe TreeSet les éléments étant triés, **l'ordre d'itération** de ces éléments respecte **la relation d'ordre utilisée**.

Outre les méthodes décrites pour les collections et les ensembles triés, il existe des méthodes particulières à la classe TreeSet : des constructeurs permettant de créer une instance vide utilisant l'ordre naturel des éléments ou utilisant un comparateur, des constructeurs permettant de créer une instance à partir d'une autre collection ou à partir d'un autre ensemble trié, des méthodes permettant de supprimer le plus petit ou le plus grand élément de l'arbre, des méthodes permettant d'accéder au plus grand ou au plus petit élément inférieur ou supérieur à un objet donné, des méthodes permettant de parcourir l'arbre dans l'ordre décroissant, des méthodes permettant d'obtenir des sous-ensembles comportant les éléments inférieurs et/ou supérieurs à des objets donnés (cf. H p. 14).

2.5. Les relations d'ordre

Il existe deux manières de définir une relation d'ordre entre des éléments : la première consiste à écrire une nouvelle classe, la seconde consiste à écrire une nouvelle méthode dans la classe correspondant aux éléments stockés dans la collection.

(a). Les comparateurs

Pour définir un comparateur, on écrit une classe qui implémente l'interface Comparator et dans laquelle on implémente la méthode compare.

```
public class MonComparateur implements Comparator <MesElements>{

        public int compare(MesElements elt1, MesElements elt2){
            // retourne      un entier strict. positif     si elt1 est strict. supérieur à elt2
            //            0                          si elt1 est égal à elt2
            //            un entier strict. négatif     si elt1 est strict. inférieur à elt2
                ...
        }
}
```

(b). Les ordres naturels

Pour définir un ordre naturel sur des éléments, on doit définir la classe correspondant à ces éléments comme étant une implémentation de l'interface Comparable et on doit implémenter la méthode compareTo.

```
        public class MesElements implements Comparable <MesElements>{
            …
            public int compareTo (MesElements elt) {
                // retourne      un entier strict. positif     si l'élément appelant est strict. supérieur à elt
                //            0                          si l'élément appelant est égal à elt
                //            un entier strict. négatif     si l'élément appelant est strict. inférieur à elt
                    ...
            }
            ...
        }
```

3. Les performances

Soit n le nombre d'éléments stockés dans la collection.

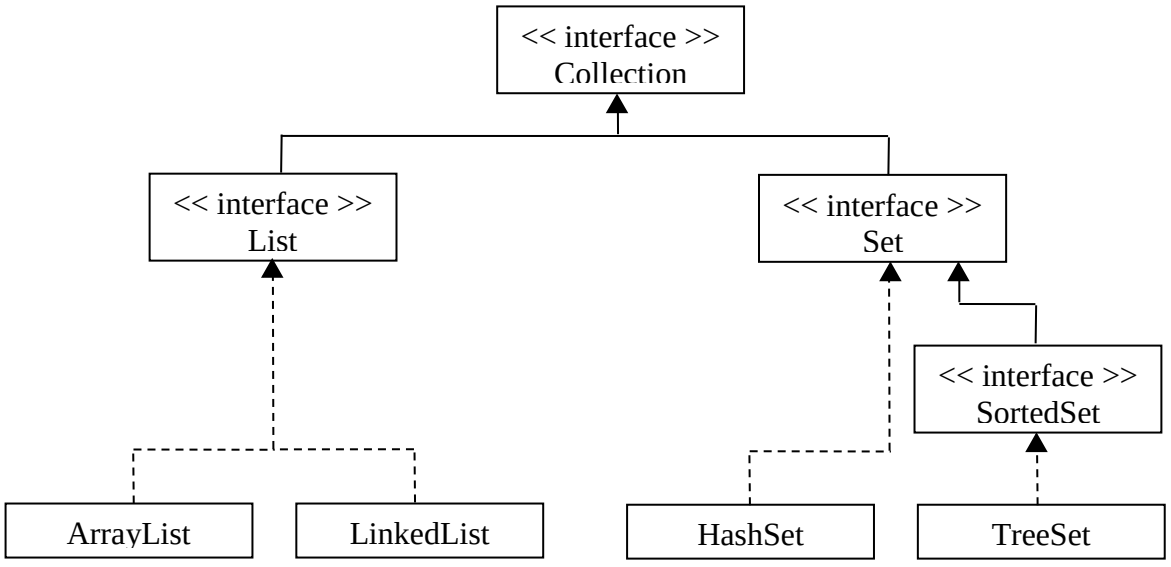
	Accès	Recherche	Ajout	Suppression
ArrayList	O(1)	O(n)	O(1)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)
HashSet	O(1)	O(1)	O(1)	O(1)
TreeSet	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$	$O(\log_2(n))$

L'accès à un élément est une opération en temps constant dans une instance de la classe ArrayList. En revanche, c'est une opération coûteuse dans une instance de la classe LinkedList.
L'ajout et la suppression d'un élément sont des opérations en temps constant dans une instance de la classe LinkedList (à condition qu'elles se fassent au début ou à la fin de la liste). L'ajout d'un élément est également une opération en temps constant dans une instance de la classe ArrayList, mais la suppression est une opération coûteuse.
La recherche d'un élément est une opération coûteuse dans les deux types de listes.

Les listes les plus fréquemment utilisées sont les instances de la classe ArrayList. L'avantage des instances de cette classe est de fournir un accès direct aux éléments, tout en gardant la possibilité de redimensionner la liste. Cependant, ce redimensionnement a un coût. Si l'on doit fréquemment ajouter ou supprimer des éléments dans la liste, il vaut mieux utiliser une instance de la classe LinkedList plutôt qu'une instance de la classe ArrayList.

Toutes les opérations, accès, recherche, ajout et suppression se font en temps constant dans une instance de la classe HashSet (à condition d'avoir une bonne fonction de hachage), alors qu'elles se font en temps logarithmique dans une instance de la classe TreeSet.

Les ensembles les plus fréquemment utilisés sont les instances de la classe HashSet.
Cependant, les performances d'une instance de la classe HashSet ne sont garanties que si le facteur de charge reste optimal. Lorsqu'elle est nécessaire, l'opération de rehachage a un coût.
De plus, dans les instances de la classe HashSet l'ordre de parcours n'est pas garanti et les opérations définies dans l'interface SortedSet (trouver le plus petit élément, trouver tous les éléments supérieurs à un autre, etc.) sont plus coûteuses que dans une instance de la classe TreeSet.



4. La classe Collections

La classe Collections est une classe dans laquelle sont écrites différentes méthodes statiques[2] s'appliquant à des collections.
Nous donnons en annexe quelques-unes de ces méthodes. Elles permettent d'obtenir le plus grand ou le plus petit élément d'une collection, d'obtenir le nombre d'occurrences d'un objet dans une collection, d'obtenir l'intersection de deux collections, de mélanger les éléments d'une liste, de trier les éléments d'une liste, d'inverser les éléments d'une liste ou d'obtenir des comparateurs (cf. I p. 16).

5. Les collections : ordonnées ou triées ?

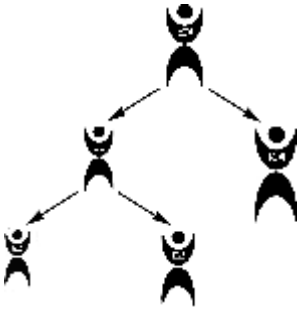
Nous avons vu dans ce chapitre qu'une collection peut être ordonnée ou non et qu'elle peut être aussi triée ou non.

Une liste est ordonnée dans le sens où il existe une relation précédent/suivant entre ses éléments (i.e. dans une liste, il y a un premier élément et un élément est toujours devant et/ou derrière un autre).

Une liste peut être en plus triée ou non : il peut en plus exister une relation d'ordre entre ses éléments. Par exemple, une instance de la classe ArrayList ou de la classe LinkedList peut contenir des personnes triées par ordre croissant de taille.

Un ensemble n'est pas intrinsèquement ordonné (i.e. un élément de l'ensemble n'est pas devant ou derrière un autre).

Dans une instance de la classe HashSet, les éléments ne sont ni ordonnés, ni triés.
Dans une instance de la classe TreeSet les éléments sont obligatoirement triés, c'est la relation qui est à la base de ce tri qui permet de les organiser sous forme arborescente. Dans l'exemple ci-dessous, les personnes sont stockées dans l'arbre en les organisant en fonction de leur taille.



[2] Une méthode statique est une méthode qui s'applique non pas à une instance de la classe, mais à la classe elle-même. Elle peut donc être appelée sans qu'aucune instance de la classe n'ait été créée. La notion de méthode statique sera approfondie dans un autre chapitre du cours.

Annexes

A. Méthodes communes à toutes les collections (interface Collection)

A.1. Ajouts

- boolean add (Object obj) : permet d'ajouter un objet obj à la collection appelante. La méthode retourne true si la collection appelante a été modifiée (i.e. si l'objet obj a pu être ajouté).
- boolean addAll (Collection coll) : permet d'ajouter tous les éléments d'une collection coll à la collection appelante. La méthode retourne true si la collection appelante a été modifiée.

A.2. Suppressions

- boolean remove (Object obj) : permet de supprimer la première occurrence de l'objet obj de la collection appelante. La méthode retourne true si la collection appelante a été modifiée (i.e. si l'objet obj a pu être supprimé).
- boolean removeAll (Collection coll) : permet de supprimer de la collection appelante tout élément égal à l'un des éléments de la collection coll. La méthode retourne true si la collection appelante a été modifiée.
- boolean retainAll (Collection coll) : permet de supprimer de la collection appelante tout élément non présent dans la collection coll. La méthode retourne true si la collection appelante a été modifiée.
- void clear() : permet de supprimer tous les éléments de la collection appelante.

A.3. Tests d'appartenance

- boolean contains (Object obj) : retourne true si la collection appelante contient au moins un élément égal à l'objet obj.
- boolean containsAll (Collection coll) : retourne true si la collection appelante contient tous les éléments de la collection coll.

A.4. Parcours

- Iterator iterator() : retourne un itérateur permettant de parcourir les éléments de la collection appelante.

A.5. Autres méthodes

- boolean isEmpty() : retourne true si la collection appelante est vide.
- int size() : retourne le nombre d'éléments de la collection appelante.
- boolean equals (Object obj) : retourne true si la collection appelante est égale à l'objet obj.
- int hashCode () : retourne la valeur de hachage de la collection appelante.
- Object[] toArray() : retourne un tableau contenant tous les éléments de la collection appelante.
- String toString() : retourne une chaîne de caractères décrivant les éléments de la collection appelante (appel de la méthode toString pour chaque élément).

B. Méthodes communes à toutes les listes (interface List)

Sauf mention contraire, les indices utilisés comme paramètres dans les méthodes ci-dessous doivent être supérieurs ou égaux à 0 et inférieurs strictement à la taille de la liste.

B.1. Ajouts

- boolean add (int ind, Object obj) : permet d'ajouter un objet obj à l'indice ind dans la liste appelante. Si l'indice ind est égal à la taille de la liste, l'objet est ajouté en fin de liste.
- boolean addAll (int ind, Collection coll) : permet d'ajouter tous les éléments d'une collection coll à l'indice ind dans la liste appelante. Si l'indice ind est égal à la taille de la liste, l'objet est ajouté en fin de liste.

B.2. Suppressions

- Object remove (int ind) : permet de supprimer l'objet qui se trouve à l'indice ind. La méthode retourne l'objet supprimé.
- Object set (int ind, Object obj) : permet de substituer l'objet d'indice ind par l'objet obj. La méthode retourne l'objet qui a été substitué.

B.3. Accès

- Object get (int ind) : retourne l'élément d'indice ind.
- int indexOf (Object obj) : retourne l'indice de la première occurrence d'un objet obj dans la liste appelante, -1 si l'objet n'est pas présent dans la liste.
- int lastIndexOf (Object obj) : retourne l'indice de la dernière occurrence d'un objet obj dans la liste appelante, -1 si l'objet n'est pas présent dans la liste.

B.4. Parcours

- ListIterator listIterator () : retourne un itérateur bidirectionnel permettant de parcourir les éléments de la liste appelante.
- ListIterator listIterator (int ind) : retourne un itérateur bidirectionnel placé à la position ind. Si l'indice ind est égal à la taille de la liste, l'itérateur est placé en fin de liste.

B.5. Vues

- List subList (int indDep, int indFin) : retourne une vue de la liste appelante contenant les objets compris entre l'indice indDep (inclus) et l'indice indFin (exclu). L'indice indDeb doit être supérieur ou égal à 0 et inférieur ou égal à l'indice indFin, l'indice indFin doit être inférieur ou égal à la taille de la liste.

C. Méthodes communes à tous les ensembles triés (interface SortedSet)

C.1. Accès

- Object first() : retourne le premier (i.e. le plus petit) élément de l'ensemble appelant.
- Object last() : retourne le dernier (i.e. le plus grand) élément de l'ensemble appelant.

C.2. Vues

- SortedSet subSet (Object objDep, Object objFin) : retourne une vue de l'ensemble appelant contenant tous les éléments supérieurs à l'objet objDep (inclus) et inférieurs à l'objet objFin (exclu). Si les deux objets sont égaux, la méthode retourne un ensemble vide**.**
- SortedSet headSet (Object obj) : retourne une vue de l'ensemble appelant contenant tous les éléments inférieurs à l'objet obj (exclu).
- SortedSet tailSet (Object obj) : retourne une vue de l'ensemble appelant contenant tous les éléments supérieurs à l'objet obj (inclus).

C.3. Comparateurs

- Comparator comparator() : retourne le comparateur utilisé pour définir la relation d'ordre sur l'ensemble appelant. La méthode retourne null si l'ensemble est trié en utilisant l'ordre naturel des éléments.

D. Méthodes concernant les itérateurs

D.1. Méthodes communes à tous les itérateurs (interface Iterator)

- boolean hasNext () : retourne true s'il existe encore au moins un élément à parcourir dans la collection.
- Object next () : retourne le prochain élément de la collection et déplace l'itérateur à l'élément suivant. La fin de la collection ne doit donc pas être atteinte.
- void remove() : permet de supprimer le dernier élément retourné par l'itérateur. Cette méthode ne peut être appelée qu'une seule fois par appel de la méthode next (ou de la méthode previous pour les itérateurs bidirectionnels).

D.2. Méthodes spécifiques aux itérateurs bidirectionnels (interface ListIterator)

- boolean hasPrevious() : retourne true s'il existe encore au moins un élément à parcourir quand on parcourt la liste en sens inverse.
- Object previous() : retourne l'élément précédent de la liste et déplace l'itérateur à l'élément précédent. Le début de la collection ne doit donc pas être atteint.
- int nextIndex () : retourne l'indice du prochain élément de la liste, la taille de la liste si l'itérateur a atteint la fin de la liste.
- int previousIndex () : retourne l'indice de l'élément précédent de la liste s'il y en a un, -1 si l'itérateur a atteint le début de la liste.
- void add (Object obj) : permet d'insérer l'objet obj dans la liste à la position à laquelle se trouve l'itérateur.
- void set (Object obj) : permet de substituer le dernier élément retourné par l'itérateur par l'objet obj. Cette méthode ne peut être utilisée que si ni la méthode add ni la méthode remove n'ont été appelées après le dernier appel des méthodes next ou previous.

E. Méthodes spécifiques de la classe ArrayList

E.1. Constructeurs

- ArrayList () : permet de créer un tableau vide de taille initiale 10.
- ArrayList (int tailleInit) : permet de créer un tableau vide de taille initiale tailleInit.
- ArrayList (Collection coll) : permet de créer un tableau contenant tous les éléments de la collection coll.

E.2. Autres méthodes

- void trimToSize () : permet de réduire la taille du tableau au nombre réel d'éléments qu'il contient.
- void ensureCapacity (int tailleMin) : permet d'augmenter la taille du tableau pour qu'il puisse contenir au moins tailleMin éléments.

F. Méthodes spécifiques de la classe LinkedList

F.1. Constructeurs

- LinkedList () : permet de créer une liste chaînée vide.
- LinkedList (Collection coll) : permet de créer une liste chaînée contenant tous les éléments de la collection coll.

F.2. Ajouts

- void addFirst (Object obj) : permet d'ajouter un objet obj au début de la liste chaînée.
- void addLast (Object obj) : permet d'ajouter un objet obj à la fin de la liste chaînée.

F.3. Suppressions

- Object removeFirst () : supprime et retourne le premier élément de la liste chaînée. La liste ne doit pas être vide.
- Object removeLast () : supprime et retourne le dernier élément de la liste chaînée. La liste ne doit pas être vide.

F.4. Accès

- Object getFirst () : retourne le premier élément de la liste chaînée. La liste ne doit pas être vide.
- Object getLast () : retourne le dernier élément de la liste chaînée. La liste ne doit pas être vide.

G. Méthodes spécifiques de la classe HashSet

G.1.Constructeurs

- HashSet () : permet de créer une table de hachage vide de taille initiale 16 et de facteur de charge 0.75.
- HashSet (int tailleInit) : permet de créer une table de hachage vide de taille initiale tailleInit et de facteur de charge 0.75.
- HashSet (int tailleInit, float fact) : permet de créer une table de hachage vide de taille initiale tailleInit et de facteur de charge fact.
- HashSet (Collection coll) : permet de créer une table de hachage contenant tous les éléments de la collection coll.

H. Méthodes spécifiques de la classe TreeSet

H.1.Constructeurs

- TreeSet () : permet de créer un arbre vide, utilisant comme relation d'ordre l'ordre naturel des éléments.
- TreeSet (Comparator comp) : permet de créer un arbre vide, utilisant comme relation d'ordre celle définie par le comparateur comp.
- TreeSet (Collection coll) : permet de créer un arbre contenant tous les éléments de la collection coll, utilisant comme relation d'ordre l'ordre naturel des éléments. Tous les éléments de la collection coll doivent être comparables.
- TreeSet (SortedSet ens) : permet de créer un arbre contenant tous les éléments de l'ensemble trié ens, utilisant comme relation d'ordre la même que celle utilisée dans l'ensemble ens.

H.2.Suppressions

- Object pollFirst () : retourne et supprime le premier (i.e. le plus petit) élément de l'arbre, retourne null si l'arbre est vide.
- Object pollLast () : retourne et supprime le dernier (i.e. le plus grand) élément de l'arbre, retourne null si l'arbre est vide.

H.3.Accès

- Object floor (Object obj) : retourne le plus grand élément inférieur ou égal à l'objet obj, retourne null s'il n'y en a pas.
- Object lower (Object obj) : retourne le plus grand élément strictement inférieur à l'objet obj, retourne null s'il n'y en a pas.
- Object ceiling (Object obj) : retourne le plus petit élément supérieur ou égal à l'objet obj, retourne null s'il n'y en a pas.
- Object higher (Object obj) : retourne le plus petit élément strictement supérieur à l'objet obj, retourne null s'il n'y en a pas.

H.4.Parcours

- Iterator descendingIterator () : retourne un itérateur permettant de parcourir l'arbre dans l'ordre inverse de celui utilisé dans l'arbre.

H.5.Vues

- NavigableSet subSet (Object objDep, boolean inclusDep, Object objFin, boolean inclusFin) : retourne une vue contenant les éléments supérieurs à l'objet objDep et inférieurs à l'objet objFin. Les booléens inclusDep et inclusFin indiquent si les objets doivent être inclus ou non. Si les deux objets sont égaux, la méthode retourne un ensemble vide (sauf si les deux booléens valent true).
- NavigableSet headSet (Object obj, boolean inclus) : retourne une vue contenant les éléments inférieurs à l'objet obj. Le booléen inclus indique si l'objet obj doit être inclus ou non.
- NavigableSet tailSet (Object obj, boolean inclus) : retourne une vue contenant les éléments supérieurs à l'objet obj. Le booléen inclus indique si l'objet obj doit être inclus ou non.
- NavigableSet descendingSet () : retourne une vue contenant les éléments rangés dans l'ordre inverse de celui utilisé dans l'arbre.

I. Méthodes de la classe Collections

I.1. Méthodes agissant sur les collections

- static Object max (Collection coll) : retourne le plus grand objet de la collection coll, en utilisant comme relation d'ordre l'ordre naturel. Tous les éléments de la collection coll doivent être comparables.
- static Object max (Collection coll, Comparator comp) : retourne le plus grand objet de la collection coll, en utilisant comme relation d'ordre l'ordre défini par le comparateur comp. Tous les éléments de la collection coll doivent être comparables avec le comparateur.
- static Object min (Collection coll) : retourne le plus petit objet de la collection coll, en utilisant comme relation d'ordre l'ordre naturel. Tous les éléments de la collection coll doivent être comparables.
- static Object min (Collection coll, Comparator comp) : retourne le plus petit objet de la collection coll, en utilisant comme relation d'ordre l'ordre défini par le comparateur comp. Tous les éléments de la collection coll doivent être comparables avec le comparateur.
- static int frequency (Collection coll, Object obj) : retourne le nombre d'éléments de la collection coll égaux à l'objet obj.
- static boolean disjoint (Collection coll1, Collection coll2) : retourne true si les deux collections coll1 et coll2 n'ont pas d'élément en commun.

I.2. Méthodes agissant sur les listes

- static void shuffle (List lis) : permet de mélanger les éléments de la liste lis.
- static void sort (List lis) : permet de trier les éléments de la liste lis en ordre croissant, en utilisant comme relation d'ordre l'ordre naturel. Tous les éléments de la liste lis doivent être comparables.
- static void sort (List lis, Comparator comp) : permet de trier les éléments de la liste lis en ordre croissant, en utilisant comme relation d'ordre celui défini par le comparateur comp. Tous les éléments de la liste lis doivent être comparables avec le comparateur.
- static void reverse (List lis) : permet d'inverser l'ordre des éléments de la liste lis.

I.3. Méthodes permettant d'obtenir des comparateurs

- static Comparator reverseOrder () : retourne un comparateur imposant l'ordre inverse de l'ordre naturel.
- static Comparator reverseOrder (Comparator comp) : retourne un comparateur imposant l'ordre inverse de celui défini par le comparateur comp.