Dans ce document, la description des classes de l'API ne prétend aucunement être exhaustive. Reportez-vous à l'API en question pour connaître tous les détails de cette classe.

Chapitre II – Collections - Généricité

Généricité

Depuis la version 5.0 Java autorise la définition de classes et d'interfaces contenant un (des) paramètre(s) représentant un *type(s)*. Cela permet de décrire une structure qui pourra être personnalisée au moment de l'*instanciation* à tout type d'objet.

Exemple

On veut définir une notion de paire d'objets avec deux attributs de même type.

```
public class PaireEntier {
private int premier;
private int second;
public PaireEntier(int x, int y){
premier =x ; second = y;}
public int getPremier(){return this.premier;}
public void setPremier(int x){this.premier=x;}
public int getSecond(){return this.second;}
public void setSecond(int x){this.second=x;}
public void interchanger(){
int temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

remarque : on a créé une classe spécialement pour des paires d'entiers ; si on veut des paires de booléens il faudrait réécrire une autre classe (avec un autre nom) qui contiendrait les mêmes méthodes.

```
public class PaireObjet {
private Object premier;
private Object second;
public PaireObjet(Object x, Object y){
premier =x ; second = y;}
public Paire(){}
public Object getPremier(){return this.premier;}
public void setPremier(Object x){this.premier=x;}
public Object getSecond(){return this.second;}
public void setSecond(Object y){this.second=y;}
public void interchanger(){
Object temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

Inconvénients :

- les deux attributs peuvent des instances de classes différentes,
- on peut être amené à faire du transtypage (vers le bas),
- on risque des erreurs de transtypage qui ne se *détecteront qu'à l'exécution*.

```
public class PaireObjet {
private Object premier;
private Object second;
public PaireObjet(Object x, Object y){
premier =x ; second = y;}
public Paire(){}
public Object getPremier(){return this.premier;}
public void setPremier(Object x){this.premier=x;}
public Object getSecond(){return this.second;}
public void setSecond(Object y){this.second=y;}
public void interchanger(){
Object temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

Inconvénients :

- les deux attributs peuvent des instances de classes différentes,
- on peut être amené à faire du transtypage (vers le bas),
- on risque des erreurs de transtypage qui ne se *détecteront qu'à l'exécution*.

A et B étant deux classes, on peut avoir ce genre d'utilisation

```
A a = new A();
B b = new B();
PaireObjet p = new PaireObjet(a,b);
A a2 = (A) p.getPremier(); // downcasting ok
p.interchanger();
A a2 = (A) p.getPremier(); // erreur
```

La solution est l'utilisation de la *généricité*, c'est-à-dire l'usage de *type paramètre*.
La généricité est une notion de *polymorphisme paramétrique*.

```
public class Paire<T> {
private T premier;
private T second;
public Paire(T x, T y){ // en-tête du constructeur sans <T>
premier =x ; second = y;}
public Paire(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public T getSecond(){return this.second;}
public void setSecond(T y){this.second=y;}
public void interchanger(){
T temp = this.premier;
this.premier = this.second;
this.second = temp;}
}
```

Cette définition permet de définir ici des `Paire` contenant des objets de type (uniforme) mais arbitraire.

utilisation

- une classe générique doit être instanciée pour être utilisée
- on ne peut pas utiliser un type primitif pour l'instanciation, il faut utiliser les classes enveloppantes
- on ne peut pas instancier avec un type générique
- une classe instanciée ne peut pas servir de type de base pour un tableau

```
Paire<String> p = new Paire<String>("bonjour"," Monsieur"); // oui
// le constructeur doit contenir <...> pour l'instanciation
Paire<>p2 = new Paire<>();  // non
Paire<int> p3 = new Paire<int>(1, 2); // non
Paire<Integer> p4 = new Paire<Integer>(1,2);// oui
Paire<Paire> p5 = new Paire<Paire>(); // non
Paire<Paire<String>> p6 = new Paire<Paire<String>>(p,p); // oui
Paire<Integer>[] tab = new Paire<Integer>[10]; // non
```

plusieurs types paramètres

On peut utiliser plusieurs types paramètres

```
public class PaireD<T,U> {
private T premier;
private U second;
public PaireD(T x, U y){ // en-tête du constructeur sans <T,U>
premier =x ; second = y;}
public PaireD(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public U getSecond(){return this.second;}
public void setSecond(U y){this.second=y;}
}
...
PaireD<Integer , String> p = PaireD<Integer , String>(1, "bonjour");
```

Utilisation du type paramètre

- le type paramètre peut être utilisé pour *déclarer* des variables (attributs) sauf dans une méthode de classe
- le type paramètre ne peut pas servir à *construire* un objet.

```
public class Paire<T> {
...
T var ; // oui
T var = new T(); //non
T[] tab ; // oui
T[] tab = new T[10];// non
```

méthodes et généricité

Une méthode de classe (`static`) ne peut pas utiliser une variable du type paramètre dans une classe générique.

```
public class UneClasseGenerique<T>{
...
public static void methodeDeClasse(){
T var ; // erreur à la compilation
...
}}
```

méthodes et généricité

Une méthode (de classe ou d'instance) peut être générique dans une classe non générique. Elle utilise alors son propre type paramètre.

```
public class ClasseA{
...
public <T> T premierElement(T[] tab){
return tab[0];} // méthode d'instance
//<T> est placé après les modificateurs et avant le type renvoyé
public static <T> T dernierElement(T[] tab){
return tab[tab.length-1];} // méthode de classe
//<T> est placé après les modificateurs et avant le type renvoyé
...
}
```

Pour utiliser une telle méthode on doit préfixer le nom de la méthode par le type d'instanciation entre < et >.

```
ClasseA a = new ClasseA();
String[] t = {"game", "of", "thrones"};
System.out.println(a.<String> premierElement(t));
System.out.println(ClasseA.<String> dernierElement(t));
```

méthodes et généricité

Une méthode (de classe ou d'instance) peut être générique dans une classe générique. Elle peut utiliser le type paramètre de la classe et son propre type paramètre.

```
public class Paire<T> {
private T premier;
private T second;
public Paire(T x, T y){ // en-tête du constructeur sans <T>
premier =x ; second = y;}
public Paire(){}
public T getPremier(){return this.premier;}
public void setPremier(T x){this.premier=x;}
public T getSecond(){return this.second;}
public void setSecond(T y){this.second=y;}
public void interchanger(){
T temp = this.premier;
this.premier = this.second;
this.second = temp;}
public <U> void  voir(U var){
System.out.println("qui est là ?" + var);
System.out.println("le premier est " + this.premier);}
}
...
Paire<Integer> p = new Paire<Integer>(1,2);
p .<String> voir("un ami");
```

exercices

exercice 1 : Réécrire les méthodes `equals` et `toString` pour les deux classes `Paire` et `PaireD`.

Limitation du type paramètre

Instancier une classe générique à un type quelconque peut empêcher d'écrire certaines méthodes.
Par exemple pour la classe `Paire`, on voudrait connaître le plus grands des 2 attributs : cela n'a de sens que si l'instanciation se fait avec un type dont les objets sont comparables donc qui implémente l'interface `Comparable` avec sa méthode `compareTo`.
Java permet de préciser que le type paramètre doit être ainsi :

```
public class Paire<T extends Comparable> { ...}
```

On peut limiter le type paramètre T par plusieurs interfaces et une classe au plus.

```
public class Paire<T extends Comparable & Cloneable & UneAutreClasse> { ...}
```

`Comparable` et `Cloneable` sont des interfaces et `UneAutreClasse` est une classe.
A l'instanciation le type choisi pour T devra implémenter les 2 interfaces et être une sous-classe de `UneAutreClasse`.

Généricité et héritage

- une classe générique peut étendre une classe (générique ou pas)

```
public class Triplet<T> extends Paire<T>{
T troisieme;
...}
```

- Triplet< T > est une sous classe de Paire< T >
- Triplet< *String* > est une sous classe de Paire< *String* >
- Triplet< *String* > n'est pas une sous classe de Paire< T >
- Triplet< *String* > n'est pas une sous classe de Paire< *Object* > bien que `String` soit une sous classe de `Object`
- Triplet< *String* > n'est pas une sous classe de Triplet< *Object* > bien que `String` soit une sous classe de `Object`

Ce dernier point interdit donc une affectation du genre

```
Triplet<Integer> t = new Triplet<Short>();
```

Collections

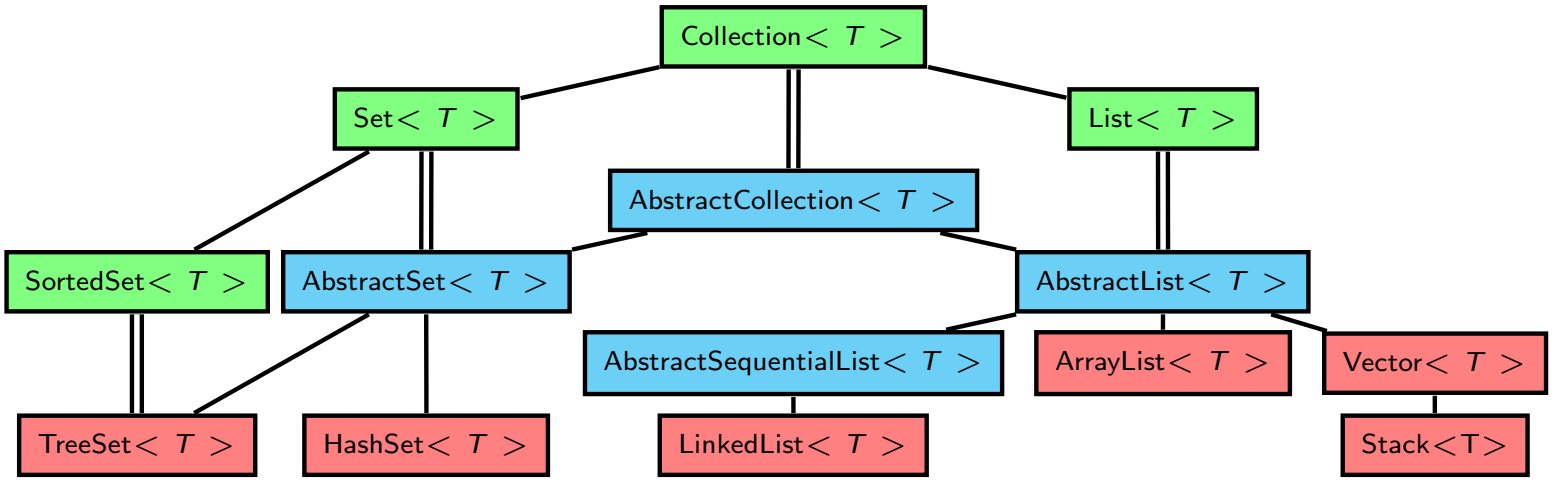
Java propose plusieurs moyens de manipuler des ensembles d'objets : on a vu les tableaux dont l'inconvénient est de ne pas être dynamique vis à vis de leur taille.
Java fournit des interfaces qui permettent de gérer des ensembles d'objets dans des structures qui peuvent être parcourues.
Ce chapitre donne un aperçu de ces collections. Elles sont toutes génériques.
Toutes les collections d'objets

- sont dans le paquetage *java.util*
- implémentent l'interface générique *Collection*

L'interface Set$< T >$ sert à implémenter les collections de type ensemble : les éléments n'y figurent qu'une fois et ne sont pas ordonnés.
L'interface List$< T >$ sert à implémenter les collections dont les éléments sont ordonnées et qui autorisent la répétition.

Interface Collection

les interfaces sont en vert, les classes abstraites en bleu et les classes en rouge, et T est le type paramètre des éléments des collections ; les lignes simples indiquent l'héritage et les lignes doubles l'implémentation. (Le schéma est partiel il existe d'autres classes).



Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(T e)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(T e)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

Les méthodes - suite

- `boolean remove(T e)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection
- `int hashCode()`

Remarque : en Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.

La classe Collections

La classe `java.util.Collections` (notez le pluriel) contient des méthodes *statiques* qui opérent sur des objets `List` ou autre (`Set`, `Map` ...) ou bien renvoie des objets.

- `void sort(List list)` trie le paramètre list
- `void sort(List list, reverseOrder() )` trie le paramètre list en ordre décroissant
- `Object max(Collection coll)` renvoie le plus grand objet
- `Object min(Collection coll)` renvoie le plus petit objet
- ...

On peut utiliser ces méthodes statiques sur des objets de toutes les classes du schéma précédent.

Itérateurs

Un iterateur est un objet utilisé avec une collection pour fournir un accès séquentiel aux éléments de cette collection.
L'interface `Iterator` permet de fixer le comportement d'un itérateur.

- `boolean hasNext()` indique s'il reste au moins un élément à parcourir dans la collection
- `T next()` renvoie le prochain élément dans la collection
- `void remove()` supprime le dernier élément parcouru (celui renvoyé par le dernier appel à la méthode next())

La méthode `next()` lève une exception de type `NoSuchElementException` si elle est appelée alors que la fin du parcours des éléments est atteinte.
La méthode `remove()` lève une exception de type `IllegalStateException` si l'appel ne correspond à aucun appel à `next()`. Cette méthode est optionnelle (exception `UnsupportedOperationException`).
On ne peut pas faire deux appels consécutifs à `remove()`.

Remarques

- A sa construction un itérateur doit être lié à une collection.
- A sa construction un itérateur se place tout au début de la collection.
- On ne peut pas « réinitialiser » un itérateur ; pour parcourir de nouveau la collection il faut créer un nouvel itérateur.
- Java utilise un itérateur pour implémenter la boucle `for each` de syntaxe suivante

```
Collection<T> c = new ... ;
for (T element : c) {...} // pour chaque objet element de type T de ma collaction c faire
```

méthode `toString()`

Pour tout objet de type `Collection`, la méthode `print` (ou `println`) appelle itérativement la méthode `toString()` de chacun de ses éléments.

Interface ListIterator

L'interface `ListIterator<T>` étend l'interface `Iterator<T>` et permet de parcourir la collection dans les deux sens.

- `T previous()` renvoie l'élément précédent dans la collection
- `boolean hasPrevious()` teste l'existence d'un élément précédent
- `T next()` renvoie l'élément suivant de la liste
- `T previous()` renvoie l'élément précédent de la liste
- `int nextIndex()` renvoie l'indice de l'élément qui sera renvoyé au prochain appel de `next()`
- `int previousIndex()` renvoie l'indice de l'élément qui sera renvoyé au prochain appel de `previous()`
- `void add(T e)` ajoute l'élément e à la liste à l'endroit du curseur (*i.e.* juste avant l'élément retourné par l'appel suivant à `next()`)
- `void remove()` supprime le dernier élément retourné par `next()` ou `previous()`
- `void set(T e)` remplace le dernier élément retourné par `next()` ou `previous()` par e

Interface ListIterator

remarques :

- `next()` et `previous()` lèvent une exception de type `NoSuchElementException`
- si l'itérateur est en fin de liste alors `nextIndex()` renvoie la taille de la liste
- si l'itérateur est au début de la liste alors `nextIndex()` renvoie -1
- `add()`, `remove()` et `set(T e)` lèvent une exception de type `IllegalStateException` si l'appel ne correspond à aucun appel à `next()` ou `previous()`. Elles sont toutes les trois optionnelles.
- `set(T e)` lève une exception de type `ClassCastException` si le type de e ne convient pas.
- dans toutes les classes prédéfinies implémentant `Iterator` ou `ListIterator`, les méthodes `next()` et `previous()` renvoient les *références* des objets de la collection.

Classe ArrayList<T>

Un `ArrayList` est un tableau d'objets dont la taille est dynamique. La classe `ArrayList<T>` implémente en particulier les interfaces `Iterator`, `ListIterator` et `List`.

Constructeurs

- `public ArrayList(int initialCapacite)` crée un `arrayList` vide avec la capacité `initialCapacite` (positif)
- `public ArrayList()` crée un `arrayList` vide avec la capacité 10
- `public ArrayList(Collection<? extends T> c)` crée un `arrayList` contenant tous les éléments de la collection c dans le même ordre avec une dimension correspondant à la taille réelle de c et non sa capacité ; le `arrayList` créé contient les références aux éléments de c (copie de surface).

Méthodes

- `add` et `addAll` ajoute à la fin du tableau
- `void add(int index, T element)` ajoute au tableau le paramètre element à l'indice index en décalant d'un rang vers la droite les éléments du tableau d'indice supérieur
- `void ensureCapacity(int k)` permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
- `T get(int index)` renvoie l'élément du tableau dont la position est précisée
- `T set(int index, T element)` renvoie l'élément à la position index et remplace sa valeur par celle du paramètre element

Méthodes

- `int indexOf(Object o)` renvoie la position de la première occurrence de l'élément fourni en paramètre
- `int lastIndexOf(Object o)` renvoie la position de la dernière occurrence de l'élément fourni en paramètre
- `T remove(int index)` renvoie l'élément du tableau à l'indice index et le supprime décalant d'un rang vers la gauche les éléments d'indice supérieur
- `void removeRange(int j,int k)` supprime tous les éléments du tableau de la position j incluse jusqu'à la position k exclue
- `void trimToSize()` ajuste la capacité du tableau sur sa taille actuelle

Exemple

On veut gérer un ensemble de personnes connaissant leur age, poids et taille par ordre de risque décroissant de problème cardiaque compte tenu de ces données.
On définit

- la classe `Personne` (nom, prenom)
- la classe `PersonneMedicalise` (étend `Personne` avec age, taille, poids, implémente l'interface `Comparable`)
- la classe `EnsPersonneMedicale` qui utilise `ArrayList`.

```
public class PersonneMedicalise extends Personne implements Comparable {
....
public int compareTo(Object p){
if (this.getAge()> ((PersonneMedicalise)p).getAge()) return -1; else
if (this.getAge()< ((PersonneMedicalise)p).getAge()) return 1; else
if (this.getPoids()> ((PersonneMedicalise)p).getPoids()) return -1; else
if (this.getPoids()< ((PersonneMedicalise)p).getPoids()) return 1; else
return 0;
}}

import java.util.*;
public class EnsPersonneMedicale {
ArrayList <PersonneMedicalise> e;
public EnsPersonneMedicale() {}
...
public  PersonneMedicalise quiEstEnDanger(){
Collections.sort(e);
return(e.get(0));}
public int ageMoyen(){
Iterator <PersonneMedicalise> it = e.iterator();
int a=0;
while (it.hasNext()) a= a+ it.next().getAge();
if (e.size()>0) return (a/e.size());else return 0;}}
```

Exercices

exercice 2 : appliquer le crible d'Eratosthène aux cent premiers entiers puis afficher tous les nombres premiers inférieurs à 100 en utilisant la classe ArrayList et un itérateur.

exercice 3 : Écrire un programme qui accepte, sur la ligne de commande, une suite de nombres et qui stocke dans un ArrayList ceux qui sont positifs.

exercice 4 : Écrire un programme qui accepte, sur la ligne de commande, une suite de chaînes de caractères et qui stocke dans un ArrayList celles qui contiennent au moins une fois le caractère 'a'. Faire afficher à l'écran toutes les chaînes ainsi stockées dans la structure ArrayList.
Ecrire une méthode qui classe le ArrayList par ordre de **longueur de chaînes croissantes** puis de nouveau faire afficher les chaînes dans cet ordre.

Exercices

exercice 5 : la princesse Eve a de nombreux prétendants ; elle décide alors de choisir celui qu'elle épousera de la façon suivante :

- les prétendants sont numérotés de 1 à n
- en partant du numéro 1 elle compte par numéro croissant 3 prétendants et élimine le troisième
- elle réitère le procédé en partant du prétendant suivant le dernier éliminé
- lorsque la fin de la liste est atteinte elle compte en recommençant au premier de la liste
- lorsque le début de la liste est atteint elle compte par numéro croissant

Ecrire un programme qui affichera le prétendant restant pour une valeur n quelconque saisie au clavier

HashSet<T>

La classe `HashSet<T>` implémente l'interface `Set<T>` et l'interface `Iterator<T>`.
Elle permet de représenter un ensemble ; les éléments ne sont pas ordonnés par leur ordre d'insertion et chaque élément sera en un seul exemplaire : par conséquent la méthode `boolean add(T e)` n'ajoutera pas l'élément e s'il est déjà contenu dans le `HashSet`. Pour cela l'implémentation d'un `HashSet` s'appuie sur une *table de hachage* et sur les méthodes `equals` et `hashCode` de `T`.

Définition

Une *table de hachage* est un tableau indexé par des entiers (en général) contenant les couples *clef-valeur*. L'indexation est réalisée par une *fonction de hachage* qui associe un indice du tableau à chaque clef.

Cette fonction doit être idéalement *injective* pour éviter une *collision* (deux clefs différentes ont le même indice). Elle doit de plus assurer une bonne dispersion des valeurs dans le tableau.

Exemple

Par exemple en Java il existe une fonction de hachage standard des chaînes de caractères :

$$h(s) = \sum_{i=0}^{i=n-1} c(s[i]) \times 31^{n-i-1}$$

où s est de type String, et n est la longueur de s et $c(s[i])$ est le code ASCII du $i+1$ ème caractère de s.
On a par exemple, $h(\text{"}toto\text{"}) = 3566134$.

hashCode()

En Java, chaque instance d'une classe a un *hashCode* fourni par la méthode `hashCode()` de la classe `Object`. Cette méthode associe à l'adresse mémoire de l'instance une valeur entière de type `int`.
Dans certaines architectures, l'espace d'adressage est beaucoup plus grand que l'intervalle de valeur du type `int` ; il est donc possible que deux objets distincts aient le même `hashCode`.
Donc on constate en pratique que cette fonction de hachage n'est pas injective.
Dans le cas où l'on réécrit la méthode `hashCode()`, on peut toujours revenir à la valeur initiale du `hashCode` de la classe `Object` en utilisant la méthode statique `System.identityHashCode(Object o)`.

Difficultés

Donc deux objets distincts peuvent avoir le même `hashCode` ; et de surcroît dans une classe ayant redéfini la méthode `equals` mais pas la méthode `hashCode()`, deux instances peuvent être égales (selon la redéfinition de `equals`) alors que leur `hashCode` sont différents. Cela peut conduire un hashSet a accepté d'ajouter un élément qu'il possède déjà.
En effet puisqu'un `HashSet` doit vérifier si un objet `e` lui appartient pour exécuter la méthode `add(e)` l'implémentation du `HashSet` va

- calculer `hashCode(e)` modulo la taille de la table et
- à l'indice correspondant à ce calcul vérifier qu'il ne possède pas d'élément `e2` tel que `e.equals(e2)` est vrai.

C'est pourquoi il est indispensable que deux éléments égaux aient le même `hashCode`.

Exemple

```
public class TestElement{
String nom;
public TestElement(String s){nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
}
public class TestHashSet{
HashSet<TestElement> hs = new HashSet<TestElement>();
void ajouter(String s){hs.add(new TestElement(s));}
void ajouter(TestElement t){ hs.add(t);}
}
public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);}
}
```

Exécution :

```
oui
[TestElement@e76cbf7, TestElement@22998b08]
```

Donc les 2 éléments qui sont égaux au sens de la méthode `equals` de `TestElement` ont été ajoutés ! Cela contredit la définition d'un `HashSet`. Ici le `hashCode` de t1 (et t2) est calculé à partir de son adresse.

Exemple

```
public class TestElement{
String nom;
public TestElement(String s){nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
}
public class TestHashSet{
HashSet<TestElement> hs = new HashSet<TestElement>();
void ajouter(String s){hs.add(new TestElement(s));}
void ajouter(TestElement t){ hs.add(t);}
}
public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);}
}
```

Exécution :

oui

[TestElement@e76cbf7, TestElement@22998b08]

Donc les 2 éléments qui sont égaux au sens de la méthode `equals` de `TestElement` ont été ajoutés ! Cela contredit la définition d'un `HashSet`. Ici le `hashCode` de t1 (et t2) est calculé à partir de son adresse.

Réécrire la méthode `hashCode()`

Afin d'utiliser les instances d'une classe comme éléments d'une collection basée sur des tables de hachage (`HashSet`, `HashMap`, ...) il est indispensable de réécrire la méthode `hashCode()` de façon :

- elle soit compatible avec `equals()`
- elle soit rapide
- elle produise le même résultat pour un objet quelque soit le moment de l'appel

ATTENTION : si on change la valeur d'un objet et si son `hashCode` *réécrit* à cause de `equals` est alors modifié *on ne pourra pas le retrouver dans la table grâce à son code*.

Exemple

```
public class TestElement{
private String nom;
public TestElement(String s){nom=s;}
public void setNom(String s){this.nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
public int hashCode(){return this.nom.hashCode();}
}

public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);
t1.setNom("loulou");
if(ths.hs.contains(t1)) System.out.println("oui"); else System.out.println("non");
}
}
```

Exécution :

```
oui
[TestElement@366a36]
non
```

Exemple

```
public class TestElement{
private String nom;
public TestElement(String s){nom=s;}
public void setNom(String s){this.nom=s;}
public boolean equals(Object o){
if(o == null) return false;
if(o.getClass()!= this.getClass()) return false;
TestElement oT = (TestElement) o;
if(this.nom.equals(oT.nom)) return true; else return false;}
public int hashCode(){return this.nom.hashCode();}
}

public class Test{
public static void main(String[] arg){
TestHashSet ths = new TestHashSet();
TestElement t1 = new TestElement("toto");
TestElement t2 = new TestElement("toto");
if(t1.equals(t2)) System.out.println("oui"); else System.out.println("non");
ths.ajouter(t1); ths.ajouter(t2);
System.out.println(ths.hs);
t1.setNom("loulou");
if(ths.hs.contains(t1)) System.out.println("oui"); else System.out.println("non");
}
}
```

Exécution :

```
oui
[TestElement@366a36]
non
```

Réécrire la méthode `hashCode()`

Voici un procédé donnée par Joshua Block dans *Effective Java* :
On initialise un entier : `int resultat = 17 ;` (un nombre premier)
Pour *chaque attribut* on calcule une valeur entière c selon le tableau suivant

type de l'attribut	calcul de c
boolean boo	c vaut 0 pour boo vrai c vaut 1 pour boo faux
char, byte, short, int n	(int) n
long l	c vaut (int)(l^ (l >>> 32))
float x	c vaut Float.floatToIntBits(x)
double y	on calcule l=Double.doubleToLongBits(y) et c vaut (int)(l^ (l >>> 32))
null	c vaut 0
Objet o (intervenant dans le code de `equals`)	c vaut o.hashCode()
tableau t	chaque élément de t est traité comme un attribut à part entière

Pour *chaque attribut* on calcule `resultat = resultat * 37 + c`
`resultat` est le `hashCode()`.

Méthodes

Un `HashSet` peut contenir `null`. Les méthodes `add`, `remove`, `contains`, `size` sont exécutées en temps O(1).

- `boolean add(T e)` si e n'appartient pas à l'instance de `HashSet` alors e est ajouté et `true` est renvoyé ; sinon l'instance n'est pas modifiée et `false` est renvoyé.
- `boolean addAll(Collection<? extends T> c)` fait l'union de la collection c avec l'instance de `HashSet` ; si cette instance est modifiée `true` est renvoyé sinon `false` est renvoyé.
- `boolean remove(Object e)` si e appartient à l'instance de `HashSet` alors e est supprimé et `true` est renvoyé ; sinon `false` est renvoyé.
- `boolean removeAll(Collection<? extends T> c)` se comporte de même
- `boolean retainAll(Collection<?> c)` fait l'intersection de la collection c avec l'instance de `HashSet` ; si cette instance est modifiée `true` est renvoyé sinon `false` est renvoyé.
- `void clear()` supprime tous les éléments de la collection

remarque : voir API pour les exceptions possibles

Exemple

```
public class PersonneMedicalise extends Personne implements Comparable {
....
public int compareTo(Object p){
if (this.getAge()> ((PersonneMedicalise)p).getAge()) return -1; else
if (this.getAge()< ((PersonneMedicalise)p).getAge()) return 1; else
if (this.getPoids()> ((PersonneMedicalise)p).getPoids()) return -1; else
if (this.getPoids()< ((PersonneMedicalise)p).getPoids()) return 1; else
return 0;
}}

import java.util.*;
public class EnsPersonneMedicale {
HashSet<PersonneMedicalise> e;
public EnsPersonneMedicale() {}
...
public  PersonneMedicalise quiEstEnDanger(){
return(Collections.min(e));
}
public int ageMoyen(){
Iterator<PersonneMedicalise> it = e.iterator();
int a=0;
while (it.hasNext()) a= a+ it.next().getAge();
if (e.size()>0) return (a/e.size());else return 0;}}
```

TreeSet<T>

La classe `TreeSet<T>` implémente l'interface `Set<T>` et l'interface `SortedSet<T>`.
Un `TreeSet` s'appuie sur un arbre (rouge-noir) pour représenter un *ensemble d'objets triés* par ordre croissant (ordre naturel ou précisé par la méthode `compareTo`).
Attention : la classe des éléments d'un `TreeSet` doit redéfinir la méthode `compareTo` et la `equals` de façon cohérente (`equals` vrai doit être équivalent à `compareTo` vaut 0).
Les méthodes `add`, `remove`, `contains`, `size` sont exécutées en temps $O(\log n)$.

Méthodes

Outre les méthodes similaires à celles de `HashSet` on peut citer :

- `public SortedSet<T> subSet(T fromElement, T toElement)` renvoie le sous-ensemble des éléments compris entre `fromElement` inclus jusqu'à `toElement` exclu
- `public SortedSet<E> headSet(T toElement)` renvoie le sous-ensemble des éléments strictement inférieurs à `toElement`
- `public SortedSet<E> tailSet(E fromElement)` renvoie le sous-ensemble des éléments supérieurs ou égal à `fromElement`
- `public T first()` renvoie le plus petit élément
- `public T last()` renvoie le plus grand élément
- `public T floor(T e)` renvoie le plus grand élément inférieur ou égal à `e`
- `public T ceiling(T e)` renvoie le plus petit élément supérieur ou égal à `e`

remarque : voir API pour les exceptions possibles

Utilisation d'un joker

Dans le code suivant il y a une affectation interdite.

```
class A {}
class B extends A{}
...
List<A> lA = new ArrayList<A>();
List<B> lB = new ArrayList<B>();
lA=lB;// interdit
```

Pour contourner l'interdiction précédente, Java permet d'utiliser un joker (ou *wildcard*) noté ?

```
List<B> lb = new ArrayList<B>();
List<? extends A> l =lB;// permis
```

ATTENTION : la liste l ainsi définie ne pourra être utilisée qu'en *lecture*.

Usage du joker

Le *wildcard* est très utile pour étendre le type de paramètre d'une méthode.
Ici la méthode `afficher` n'autorise que les paramètres de type `List<AWild>`

```
public class AWild {...}
public class BWild extends A{... }
public class TestAwildBWild{
static void afficher(List<AWild> l){
for(AWild a:l){System.out.println(a);}
}
public static void main(String[] arg){
List<AWild> lA = new ArrayList<AWild>();
List<BWild> lB = new ArrayList<BWild>();
lA.add(new AWild(1));
lA.add(new BWild(2));
lB.add(new BWild(3));
afficher(lA);
afficher(lB);//erreur à la compilation
// car lB n'est pas d'un sous-type du type de lA
}}
```

Usage du joker

Avec le *wildcard* la méthode `afficher` fonctionnera sur toutes listes contenant des sous-types de `AWild` :

```
public class AWild {...}
public class BWild extends A{... }
public class TestAwildBWild{
static void afficher(List<? extends AWild> l){
for(AWild a:l){System.out.println(a);}
}
public static void main(String[] arg){
List<AWild> lA = new ArrayList<AWild>();
List<BWild> lB = new ArrayList<BWild>();
lA.add(new AWild(1));
lA.add(new BWild(2));
lB.add(new BWild(3));
afficher(lA);
afficher(lB);// autorisé
}}
```

Remaque : on peut aussi utiliser le *wildcard* pour toute sur-classe d'une classe donnée

```
maMethode(List<? super uneClasse> l) {...}
```

L'interface `Map`

Un objet de type `Map` stocke des couples *clef-valeur*. On parle aussi de *dictionnaire*.
La clef doit être unique mais une valeur peut avoir plusieurs clefs.
L'interface `Map<K,V>` fixe les méthodes pour manipuler de tels couples. Les clefs sont de type `K` et sont associées aux valeurs de type `V`.
Toutes les collections de couples clef-valeur

- sont dans le paquetage *java.util*
- implémentent l'interface générique *Map<K,V>*

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Des méthodes de l'interface Map<K,V>

- `V put(K clef, V valeur)` associe *valeur* à *clef*. Si clef est déjà associée à un objet `V` alors il est remplacé par le paramètre *valeur* puis il est renvoyé (`null` est renvoyé si *clef* était associée à rien)
- `V get(Object clef)` renvoie la valeur associée à *clef* si elle existe ou `null` sinon.
- `boolean containsKey(Object clef)` indique si un élément est associée au paramètre *clef*
- `boolean containsValue(Object valeur)` indique si le paramètre *valeur* est associée à au moins une clef
- `boolean isEmpty()` indique si la collection est vide
- `Set<K> keySet()` renvoie un ensemble constitué des clefs de l'objet
- `Collection<V> values()` renvoie la collection constituée des valeurs de l'objet
- `Set<Map.Entry<K,V> > entrySet()` renvoie un ensemble constitué des couples (clefs,valeurs)

Se référer à l'API Java pour voir toutes les méthodes.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Recommandation sur la classe `K`

Pour toutes les classes ci-dessous il est fortement recommandé d'utiliser des objets *immuables* pour les clefs.
Un objet *immuable* est un objet que l'on ne peut pas modifier une fois qu'il est créé.
Pour rendre immuables les instances d'une classe il faut :

- la marquer `final` pour qu'elle n'ait pas de classe fille
- marquer `private` les attributs
- marquer `final` les attributs (conseillé)
- ne pas écrire d'accesseurs en écriture (`set..`)
- écrire des accesseurs en lecture (`get..`) qui renvoient une nouvelle instance de l'attribut lu ou bien un objet immuable
- ne pas implémenter `Cloneable`

Les classes enveloppantes (`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`) et la classe `String` sont immuables.

Les implémentations

La classe `HashMap<K,V>` implémente l'interface `Map<K,V>` avec une table de hachage. `null` peut être une clef.
La classe `Hashtable<K,V>` implémente l'interface `Map<K,V>` avec une table de hachage. `null` ne peut pas être une clef.
La classe `TreeMap<K,V>` implémente l'interface `Map<K,V>` avec un arbre rouge noir pour un ordre naturel sur les clefs ou bien avec un `Comparator` sur les clefs.
Cette dernière classe implémente l'interface `SortedMap` et a des méthodes spécifiques s'appuyant sur l'ordre des clefs.

Itérer sur un objet `Map`

On peut utliser plusieurs méthodes :

- utiliser une boucle `foreach` sur l'ensemble des clefs obtenu par la méthode `keySet()`
- utiliser un itérateur sur l'ensemble des clefs obtenu par la méthode `keySet()`
- utiliser une boucle `foreach` sur l'ensemble des couples (clefs,valeurs) obtenu par la méthode `entrySet()`
- utiliser un itérateur sur l'ensemble des couples (clefs,valeurs) obtenu par la méthode `entrySet()`

Exemple

```
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String>  s:hm.entrySet())
System.out.println(s);
}}
```

Exécution :

nom ? : Pierre

+33612121212

Paul=+33614141414

Jacques=+33615151515

Pierre=+33612121212

Exemple

```
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String>  s:hm.entrySet())
System.out.println(s);
}}
```

Exécution :

nom ? : Pierre

+33612121212

Paul=+33614141414

Jacques=+33615151515

Pierre=+33612121212

Exemple

```
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String>  s:hm.entrySet())
System.out.println(s);
}}
```

Exécution :

nom ? : Pierre

+33612121212

Paul=+33614141414

Jacques=+33615151515

Pierre=+33612121212

Exemple

```
import java.util.*;
public class TestHashMap{
public static void main(String[] arg){
HashMap<String, String> hm = new HashMap<String, String> ();
Scanner sc = new Scanner(System.in);
hm.put("Pierre","+33612121212");
hm.put("Paul","+33614141414");
hm.put("Jacques","+33615151515");
System.out.print("nom ? :");
String nom =sc.next();
System.out.println(hm.get(nom));
for(Map.Entry<String,String>  s:hm.entrySet())
System.out.println(s);
}}
```

Exécution :

nom ? : Pierre

+33612121212

Paul=+33614141414

Jacques=+33615151515

Pierre=+33612121212

Exercice

exercice 6 : Dans le but d'établir des statistiques sur les mots employés dans un document texte de 100000 mots on crée un dictionnaire de couples (int, liste de (mots,flottant))) de la façon suivante : la longueur d'un mot constitue une clef d'une liste ne contenant que des mots de cette longueur avec leur fréquence.
Ecrire une classe interne `Paire` caractérisée par deux attributs `String` et `Double`
Ecrire une méthode `void setMot(String s)` qui place la chaine de lettres `s` dans le dictionnaire en mettant à jour sa fréquence.
Ecrire une méthode `double frequenceMot(String s)` qui renvoie la fréquence de la chaine `s`