API Collection- iterateurs , ordre naturel

Marianne Simonot

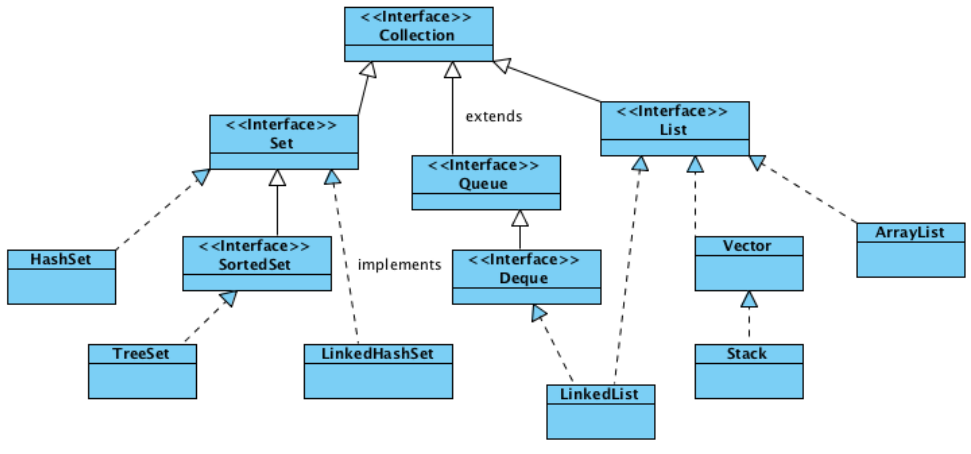
.

1 collections : iterator et iterable

Nous avons appris depuis que nous travaillons sur les Collections à utiliser la boucle foreach pour parcourir ces structures. Nous avons appris dans le TP8 que ceci est permis par le fait que l'interface `Collection` étends l'interface `Iterable` :

Aide mémoire :

L'API collection Vous connaissez maintenant de nombreuses structures servant à stocker des éléments de même type : `ArrayList`, `Set`, `SortedSet`, `HashSet`, `TreeSet`. Toutes ces structures font partie d'une API Java appelée `API Collection` :



Cette API est constituée d'une hiérarchie d'interfaces et de classes regroupant toutes les structures de groupement d'éléments.

L'interface `Collection` est le sommet de la hiérarchie. Une collection est un "sac" dans lequel on peut ajouter, supprimer des éléments **ET** qui étend l'interface `Iterable`

```
public interface Collection<E> extends Iterable<E>{
int size ();
boolean isEmpty();
boolean contains(Object o) ;
boolean add(E o);
boolean remove(Object o);
Iterator iterator () ;
...
}
```

Un itérable est quelqu'un qui fournit un itérateur. On peut utiliser le foreach dès qu'une classe implémente itérable.

La boucle foreach est l'unique moyen de parcourir une collection.

```
public interface Iterable<E>{

Iterator<E> iterator()
}
```

On vous a menti ...

Dans l'aide mémoire, est dit : " La boucle foreach est l'unique moyen de parcourir une collection". Ceci est faux. En fait, la boucle foreach est du sucre syntaxique qui masque l'utilisation d'un itérateur.
On peut tout aussi bien parcourir une structure en utilisant directement l'itérateur fourni par la méthode de `Collection iterator()`.

Reprenons depuis le début

- L'interface `Iterable` contient une unique méthode : `Iterator iterator()`.
  Comme `Collection` étend `Iterable`, toutes ses implémentations et donc toutes les structures de collections que nous utilisons fournissent du code pour cette méthode.
- Cette méthode retourne un objet de type `Iterator` qui est un objet dont l'unique responsabilité est de permettre de parcourir la collection qui le fournit.
  Ainsi dans le code suivant, l'objet `it` va nous permettre de parcourir non pas n'importe quelle collection mais la collection `col1`.

```
public static void main(String[] args) {
    Collection<Gens> col1 = new HashSet<Gens>(
    Gens a, b, c, d;
    a = new Gens("lulu", 18);
    b = new Gens("toto", 17);
    c = new Gens("zaza", 20);
    d = new Gens("bibi", 20);
    col1.add(a);
    col1.add(b);
    col1.add(c);
    col1.add(d);

    Iterator<Gens> it= col1.iterator();

}
```

— Voici les méthodes applicables sur un itérateur :

```
public interface Iterator<E>{

    public boolean hasNext();
    public E next();
    default public void remove();
    default public void forEachRemaining(Consumer<? super E action>;
}
```

Chaque appel de la méthode `next()` fournit un élément **non encore fourni** de la collection. `hasnext()` répond true ssi il existe encore des éléments dans la collection non encore fourni par l'itérateur. On peut donc parcourir `col1` comme suit :

```
Iterator<Gens> it = col1.iterator();
while (it.hasNext()) {
    Gens x = it.next();
    System.out.println(x);
}
```

Ce qui donne :

```
Gens [nom=bibi, age=20]
Gens [nom=zaza, age=20]
Gens [nom=lulu, age=18]
Gens [nom=toto, age=17]
```

foreach et iterator

La boucle foreach est du sucre syntaxique : à la compilation elle est traduite en un parcours utilisant un itérateur. Par exemple,

```
for(Gens x:col1){

    System.out.println(x);

}
```

est traduite en :

```
Iterator<Gens> it1 = col1.iterator();
while (it1.hasNext()) {
    Gens x = it1.next();
    System.out.println(x);
}
```

Quand est on obligé d'utiliser un itérateur ?

La boucle foreach a une syntaxe beaucoup plus plaisante que l'itérateur. Il faut donc l'utiliser dès que possible.
Le seul cas où il est absolument nécessaire d'utiliser un itérateur est celui ou on enlève des éléments de la collection à l'intérieur de la boucle. En effet, dans ce cas, l'utilisation de la méthode `remove` de la collection produit une exception. Il faut donc utiliser la méthode `remove` de l'iterateur.

```
for (Gens x : col1) {
    if (x.getAge() == 20) {
        col1.remove(x);
    }
}
```

produit l'exception :

```
Exception in thread "main"
java.util.ConcurrentModificationException.
```

```
        it = col1.iterator();
        while (it.hasNext()) {
            Gens x = it.next();
            if (x.getAge() == 20) {
                it.remove();
            }
        }
        System.out.println(col1);
```

produit :

```
[Gens [nom=lulu, age=18], Gens [nom=toto, age=17]]
```

Un itérateur = un parcours

Une fois qu'un objet Itérateur a donné tous les éléments de la collection (`hasnext()` `donne false`), il ne peut plus servir. Pour parcourir une seconde fois la structure, il faut réinitialiser l'itérateur par un appel à la méthode iterator() de la collection que lon veut parcourir.

```
        Iterator<Gens> it = col1.iterator();
        System.out.println("parcours 1");
        while (it.hasNext()) {
            Gens x = it.next();
            System.out.println(x);
        }

        System.out.println("parcours 2");
        while (it.hasNext()) {
            Gens x = it.next();
            System.out.println(x);
        }
```

produit l'affichage :

```
parcours 1
Gens [nom=lulu, age=18]
Gens [nom=toto, age=17]
Gens [nom=zaza, age=20]
Gens [nom=bibi, age=20]
parcours 2
```

car lors du deuxième parcours, l'itérateur a épuisé la liste. On ne rentre pas dans la boucle.

```
        Iterator<Gens> it = col1.iterator();
        System.out.println("parcours 1");
        while (it.hasNext()) {
            Gens x = it.next();
            System.out.println(x);
        }

        System.out.println("parcours 2");
        it = col1.iterator();
        while (it.hasNext()) {
            Gens x = it.next();
            System.out.println(x);
        }
```

produit :

```
parcours 1
Gens [nom=lulu, age=18]
Gens [nom=toto, age=17]
Gens [nom=zaza, age=20]
Gens [nom=bibi, age=20]
parcours 2
Gens [nom=lulu, age=18]
Gens [nom=toto, age=17]
Gens [nom=zaza, age=20]
Gens [nom=bibi, age=20]
```

car l'instruction `it = col1.iterator();` réinitialise `it`.

2 retour sur les SortedSet : Comparable et ordre naturel

Nous avons appris à créer des `TreeSet` en utilisant un constructeur qui prend en paramètre un objet de type `Comparator`. Cet objet représente l'ordre avec lequel le `SortedSet` doit ranger les éléments.

Mais sur les éléments du `SortedSet`, il y a bien souvent un ordre utilisé presqu à chaque fois : l'ordre alphabétique pour les `String`, l'ordre croissant sur les entiers .... De la même façon, pour les classes que nous définissons nous même, il y a souvent un ordre privilégié, un ordre **naturel** qui risque d'être utilisé presqu'à chaque fois que nous voulons mettre ces éléments dans un `SortedSet` : ordre alphabétique des noms puis ordre croissant des ages pour les `Gens` etc ....

Il y a un moyen en Java de faire cela. Voici la marche à suivre :

1. Définir dans la classe des éléments qu'on veut "ranger" l'ordre naturel et faire en sorte que cette classe étende l'interface **prédéfinie** `Comparable`

   Cette interface, comme `Comparator`, contient une unique méthode.
   C'est une méthode de comparaison.

```
public interface Comparable<E>{

    public int compareTo(E other);
}
```

   C'est la classe `Gens` qui implémente `Comparable<Gens>`.

   Cela permet de mettre le code de `compareTo` dans la classe `Gens`

   Elle servira de méthode de comparaison par défaut pour les `TreeSet<Gens>`.

```
public class Gens implements Comparable<Gens> {

    private String nom;
    private int age;
....
    @Override
    public int compareTo(Gens o) {
        if (this.getNom().compareTo(o.getNom()) == 0) {
            return this.getAge() - o.getAge();
        } else {
            return this.getNom().compareTo(o.getNom());
        }
    }
}
```

2. A la création du `TreeSet`, utiliser le constructeur sans argument :

   Le constructeur sans argument de `TreeSet<Gens>` va chercher la méthode CompareTo dans `Gens` et l'utilise pour ranger les gens dans la structure.

```
        Collection<Gens> col1 = new TreeSet<Gens>();
        Gens a, b, c, d;
        a = new Gens("lulu", 18);
        b = new Gens("toto", 17);
        c = new Gens("lulu", 20);
        d = new Gens("bibi", 20);

        col1.add(a);
        col1.add(b);
        col1.add(c);
        col1.add(d);
        System.out.println(col1);
```

   Si `Gens` n'implémente pas `Comparable<Gens>`, une exception sera lancée la première fois que vous essaierez d'ajouter un élément dans l'ensemble...

```
[Gens [nom=bibi, age=20], Gens [nom=lulu, age=18],
 Gens [nom=lulu, age=20], Gens [nom=toto, age=17]]
```