

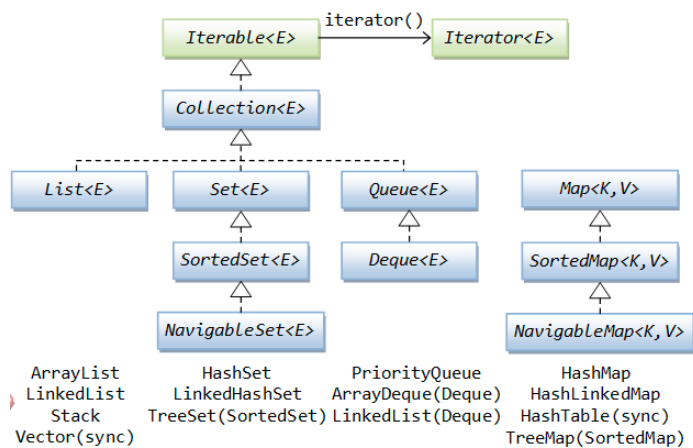
2. Feuille de TD

semaine 16 :

Lecture de documentation

Voici des extraits de documentation qui vous seront utiles pour répondre aux questions de cet exercice

Hierarchie de classes de Collection :



Extrait de la doc de Iterable :

Interface Iterable<T>

Type Parameters:

T - the type of elements returned by the iterator

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

Method Summary

Methods

Modifier and Type	Method and Description
Iterator<T>	iterator() Returns an iterator over a set of elements of type T.

Extrait de la doc de Iterator :

Interface Iterator<E>

```
public interface Iterator<E>
```

An iterator over a collection.

Method Detail

hasNext

```
boolean hasNext()
```

Returns true if the iteration has more elements. (In other words, returns true if `next()` would return an element rather than throwing an exception.)

Returns:

true if the iteration has more elements

next

```
E next()
```

Returns the next element in the iteration.

Returns:

the next element in the iteration

Throws:

`NoSuchElementException` - if the iteration has no more elements

remove

```
void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:

`UnsupportedOperationException` - if the remove operation is not supported by this iterator

`IllegalStateException` - if the next method has not yet been called, or the remove method has already been called after the last call to the next method

1. Citez 3 parents de **ArrayList**.

Dans la doc de **Iterable** on lit en particulier : "*Implementing this interface allows an object to be the target of the 'foreach' statement.*"

Autrement dit, quelque chose qui est **Iterable<E>** peut être *itéré*, c'est-à-dire qu'on peut faire une boucle *foreach* (for par élément) dessus. Par exemple :

```
Iterable<T> truc = new TrucQuiImplementeIterableDeT();
for(T elem : truc) {
    System.out.println(elem);
}
```

2. Parmi les classes suivantes, lesquelles sont *itérables* ?

- **ArrayList**
- **HashMap**
- **HashSet**
- **String**

3. Dans la hiérarchie de classes présentée précédemment, on peut remarquer que l'interface **Collection<E>** est une sous-interface de **Iterable<E>**. Les **Collection** sont donc *itérables*. Cela signifie-t-il qu'il y a un ordre sur les éléments d'une **Collection** ?

4. Rendre une classe *itérable* consiste à implémenter la seule méthode de l'interface **Iterable**. Quel est le nom de cette méthode ? Combien prend-elle de paramètres ? Quel est le type de retour ?

Sachant qu'une liste est *itérable*, on peut donc faire :

```
List<Integer> l = new ArrayList<>();
l.add(5);
l.add(4);
Iterator<Integer> itereur = l.iterator();
```

5. Que renverrait `itereur.hasNext()` ?
6. Que renverrait `itereur.next()` ? Et si on le fait une seconde fois ? Et une troisième fois ?
7. Que ferait le code suivant sur l :

```
int x = itereur.next();
itereur.remove();
x = itereur.next();
itereur.remove();
```

8. Donnez le code (utilisant un itérateur bien sûr) d'une fonction qui prend en paramètre une liste et affiche dans le terminal tous les éléments de cette liste.
9. Voici le code d'une méthode **mystere**. Que fait cette méthode ?

```
public static <T extends Comparable<T>> T mystere(Collection<T> coll) {
    Iterator<T> itereur = coll.iterator();
    T candidate = itereur.next();
    while (itereur.hasNext()) {
        T next = itereur.next();
        if (next.compareTo(candidate) < 0)
            candidate = next;
    }
    return candidate;
}
```

Entrainement

1. On vous donne la classe suivante

```
public class Iterateur {

    private Iterateur() {}

    public static int mystere0(Iterable<Integer> coll){
        Iterator<Integer> itereur = coll.iterator();
        int res = 0;
        while (itereur.hasNext()) {
            res += 1;
            itereur.next();
        }
        return res;
    }

    public static <T> boolean mystere1(Iterable<T> coll, T elem){
        Iterator<T> itereur = coll.iterator();
        while (itereur.hasNext()) {
            if(itereur.next().equals(elem)){
                return true;
            }
        }
    }
}
```

(suite sur la page suivante)

```

    return false;
}
}

```

- Quel est le résultat de l'appel de `mystere0` avec la liste `[3, 6, 3, 2, 1, -3, 2]`. Que fait la méthode `mystere0` ?
 - Quel est le résultat de l'appel de `mystere1` avec l'ensemble extrait de la liste précédente et `-4`. Que fait la méthode `mystere1` ?
 - Écrivez un exécutable permettant de tester ces deux méthodes
2. On vous propose la classe suivante

```

public class IterateurMystere implements Iterator<Character> {
    String s;
    int position;

    IterateurMystere(String s){
        this.s = s;
        this.position = 0;
    }

    @Override
    public boolean hasNext(){
        return ((position+2) <= s.length());
    }

    @Override
    public Character next() throws NoSuchElementException {
        if(!this.hasNext()){
            throw new NoSuchElementException();
        }
        int p = this.position;
        this.position += 2;
        return this.s.charAt(p);
    }
}

```

```

public class Mystere implements Iterable<Character>{
    private String s;
    Mystere(String s){
        this.s = s;
    }

    public Iterator<Character> iterator(){
        return new IterateurMystere(s);
    }
}

```

- Écrivez un exécutable permettant de tester cet itérateur, quel est le résultat sur la chaîne "Bonjour" ? Que fait cet itérateur ?

Utilisation de l'interface Iterable

1. Méthode AfficheTous (2 versions)

Écrivez une méthode `AfficheTous` prenant en paramètre un `Iterable<T>` et affichant tous les éléments de l'itérable. Commencez par écrire le code en utilisant une boucle `for` (la version que vous auriez fait sans connaître les itérateurs) puis en utilisant la méthode `iterator()`.

2. Méthode getMin

Écrivez une méthode `getMin()` prenant en paramètre un `Iterable<Integer>` et renvoyant le minimum de celui-ci s'il existe et levant une Exception **ListeVideException** sinon. Vous bien sûr ici utiliser la notion d'itérateur et les méthodes associées.

3. Méthode somme

De la même façon, utilisez un itérateur pour calculer la somme des éléments d'un itérable d'entiers.

4. Méthode plusLongPlateau

Définissez la méthode `plusLongPlateau` prenant en paramètre une liste d'entiers et renvoyant la longueur du plus long plateau de la liste (i.e. la longueur de la plus longue sous-suite de valeurs identiques), en utilisant un itérateur.

Entier

Écrivez une classe **Entier** itérable telle que lorsqu'on itère dessus on récupère un à un les chiffres composant le nombre. Par exemple

```
Entier n = new Entier(1230);
for (Integer i: n){
    System.out.println(i);
}

// Affiche :
// 0
// 3
// 2
// 1
```

Exemple d'implémentation de l'interface Itérable : la classe Range

Dans cet exercice on va créer notre propre itérable : une classe **Range** qui simule le `range` de Python.

On voudrait pouvoir écrire quelque chose ressemblant à

```
for indice in range(10) :
    println(indice)
```

Pour nous la syntaxe ressemblera à:

```
public class Executable {
    public static void main(String [] args){
        for(Integer i : new Range(10)){
            System.out.println(i);
        }
    }
}
```

Notre classe **Range** doit donc être `Iterable<Integer>` et pouvoir fournir un `Iterator<Integer>`.

1. Pour commencer, on va définir une classe **IterateurSimple**. Complétez la.

```
import java.util.Iterator;

public class IterateurSimple implements Iterator<Integer> {
    private int pos;
    private int fin;

    public IterateurSimple(int fin) { //TODO
    @Override
    public Integer next() throws NoSuchElementException { //TODO
    @Override
    public boolean hasNext() { //TODO
}
}
```

2. Complétez maintenant la classe **RangeSimple** :

```
public class RangeSimple implements Iterable<Integer> {
    private int limite;

    public RangeSimple(int limite) { this.limite = limite; }

    public Iterator<Integer> iterator() { //TODO
}
}
```

3. Vérifiez que la classe **Executable** donnée ci-dessus s'exécute correctement.
4. Complétez les classes **Executable** et écrivez une classe **Range** pour simuler le code python suivant, permettant d'afficher les valeurs 1, 3, 5, 7, 9 :

```
for indice in range(1, 10, 2) :
    println(indice)
```