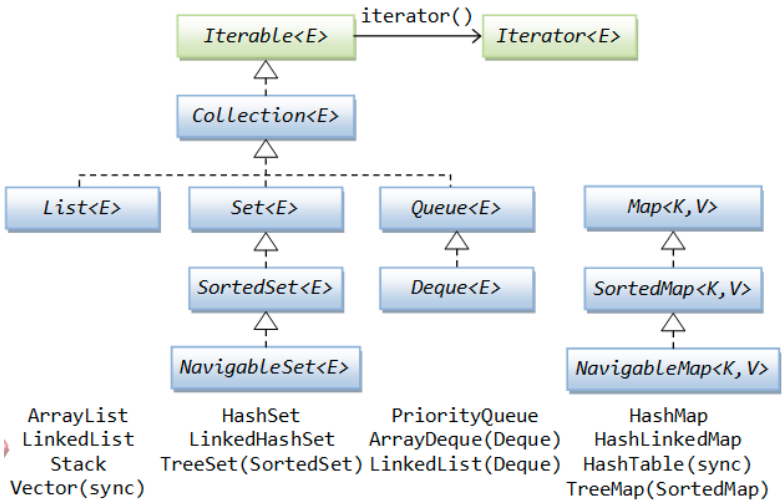
TD/TP : Focus sur les Iterateurs

> **Note**
>
> Le but de cet exercice est de s'intéresser à l'interface **Iterable**.
>
> A quoi sert cette interface ? Comment peut-on l'utiliser autrement qu'avec une boucle for ? Comment écrire nos propres structures itérables ?

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>`

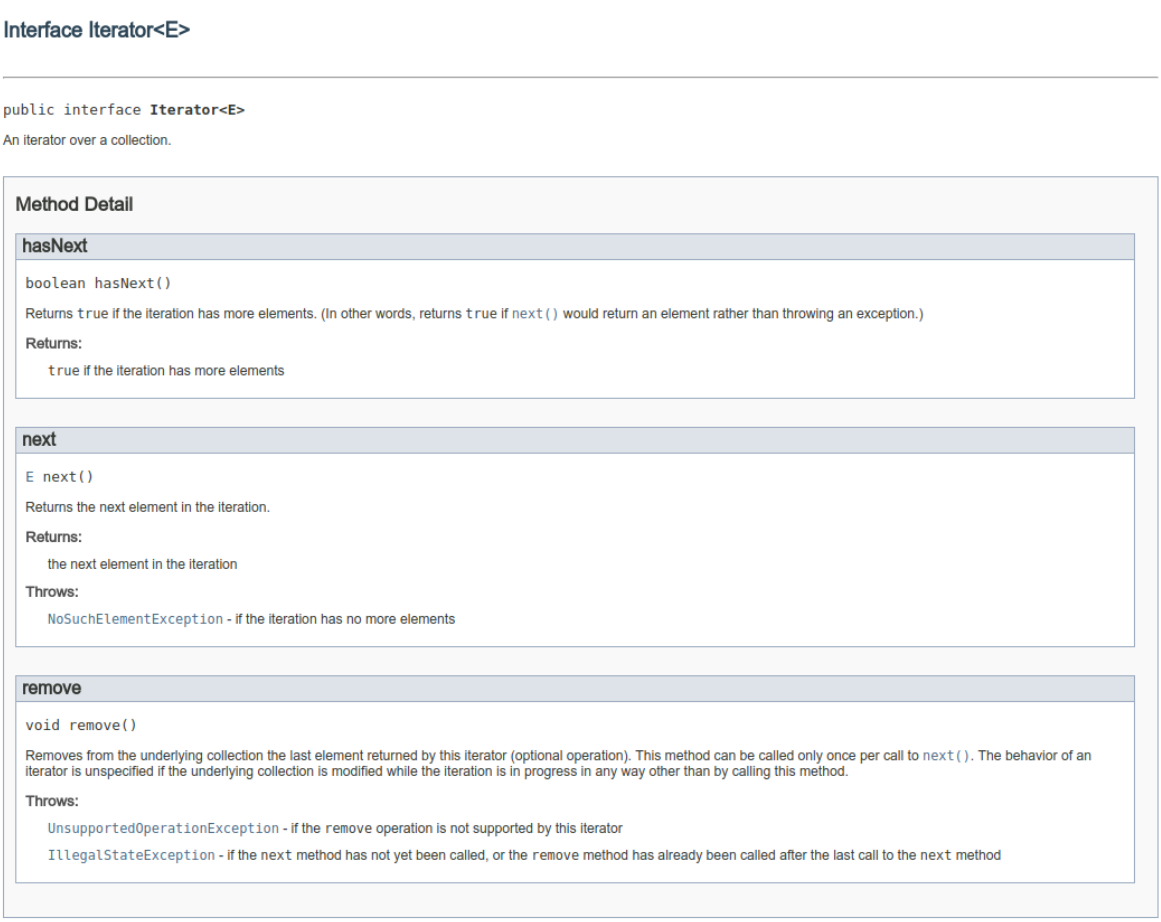
Implementing this interface allows an object to be the target of the "foreach" statement.

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 y 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) sur celle-ci. 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 *iterables*. Cela signifie-t-il qu'il y a un ordre sur les éléments d'une **Collection** (justifiez) ?
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 son type de retour ?

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

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

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

```
int x = iterateur.next();
iterateur.remove();
x = iterateur.next();
iterateur.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> i = coll.iterator();
     T candidate = i.next();
      while (i.hasNext()) {
          T next = i.next();
          if (next.compareTo(candidate) < 0)
              candidate = next;
     }
     return candidate;
 }
```

Exemples d'utilisation de l'interface Itérable

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 faite sans connaitre les itérateurs) puis en utilisant la méthode iterator() et une boucle while (portant sur le fait qu'il reste des éléments).

2. **Méthode getMin**

Écrivez une méthode `getMin()` prenant en paramètre un *iterable<Integer>* et renvoyant le minimum de cette liste s'il existe et levant une Exception **ListeVideException** sinon. Vous utiliserez bien sûr ici 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 iterable 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 longeur de la plus longue sous-suite de valeurs identiques).

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

Dans cet exercice on va créer notre propre iterable : 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}
       public Integer next() {//TODO}
       public boolean hasNext() {//TODO}
}
```

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

```
public class Range implements Iterable<Integer> {
     private int limite;
     public Range(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 **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)
```