

Cours 5 : Les collections

Généralités

Tableaux

Collections

Maps

Iterator

Vues et ponts entre structures

Cours 5 : Les collections

Généralités

Tableaux

Collections

Maps

Iterator

Vues et ponts entre structures

Structures de données

- ▶ En Java, 3 sortes de structures de données :

Structures de données

- ▶ En Java, 3 sortes de structures de données :
 - ▶ Les **tableaux**
 - ▶ Taille fixe, accès direct aux éléments.

Structures de données

- ▶ En Java, 3 sortes de structures de données :
 - ▶ Les **tableaux**
 - ▶ Taille fixe, accès direct aux éléments.
 - ▶ Les **collections**
 - ▶ Structure modifiable, différents algorithmes de stockage.

Structures de données

- ▶ En Java, 3 sortes de structures de données :
 - ▶ Les **tableaux**
 - ▶ Taille fixe, accès direct aux éléments.
 - ▶ Les **collections**
 - ▶ Structure modifiable, différents algorithmes de stockage.
 - ▶ Les **Maps**
 - ▶ Structure modifiable, stockage de couples CLEF → VALEUR.

Contrats

```
1 public class Stupid {
2     private final int i ;
3     public Stupid(int i) {
4         this.i = i ;
5     }
6
7     public static void main(String[] args) {
8         Stupid[] a = new Stupid[2] ;
9         a[0] = new Stupid(2) ;
10        a[1] = new Stupid(1) ;
11        Arrays.sort(a) ;
12
13        HashSet<Stupid> hs = new HashSet<>() ;
14        hs.add(new Stupid(1)) ;
15        System.out.println(hs.contains(new Stupid(1))) ;
16    }
17 }
```

► Problème(s) ?

Contrats

```
1 public class Stupid {
2     private final int i ;
3     public Stupid(int i) {
4         this.i = i ;
5     }
6
7     public static void main(String[] args) {
8         Stupid[] a = new Stupid[2] ;
9         a[0] = new Stupid(2) ;
10        a[1] = new Stupid(1) ;
11        Arrays.sort(a) ;
12
13        HashSet<Stupid> hs = new HashSet<>() ;
14        hs.add(new Stupid(1)) ;
15        System.out.println(hs.contains(new Stupid(1))) ;
16    }
17 }
```

► Problème(s) ?

1. sort lève une `ClassCastException` (cast de `Stupid` en `Comparable`).
2. `contains` renvoie faux.

Contrats

- ▶ Les algorithmes existant sur les collections supposent que les objets stockés respectent un certain **contrat**.
- ▶ Il faut implémenter correctement `equals`, `hashCode`, `compareTo`....
- ▶ Sinon, n'importe quoi et/ou exceptions !

Cours 5 : Les collections

Généralités

Tableaux

Collections

Maps

Iterator

Vues et ponts entre structures

Tableaux

- ▶ Taille fixe définie à l'initialisation.
- ▶ Toujours mutable.

Arrays

- ▶ Classe Arrays (avec un s).
- ▶ Méthodes statiques utilitaires sur les tableaux.
 - ▶ Remplissage (`fill`).
 - ▶ Tri, égalité, recherche binaire.
 - ▶ Conversion en liste, `toString...`

Utilisation

- ▶ Rarement utilisé dans du code utilisateur.
 - ▶ Doit connaître la taille à l'avance.
 - ▶ Problématique avec les types paramétrés.
- ▶ Tableaux essentiellement utiles pour écrire sa propre structure de données.
- ▶ Ou pour les types primitifs si soucis de performance (pour éviter le wrapping).

Cours 5 : Les collections

Généralités

Tableaux

Collections

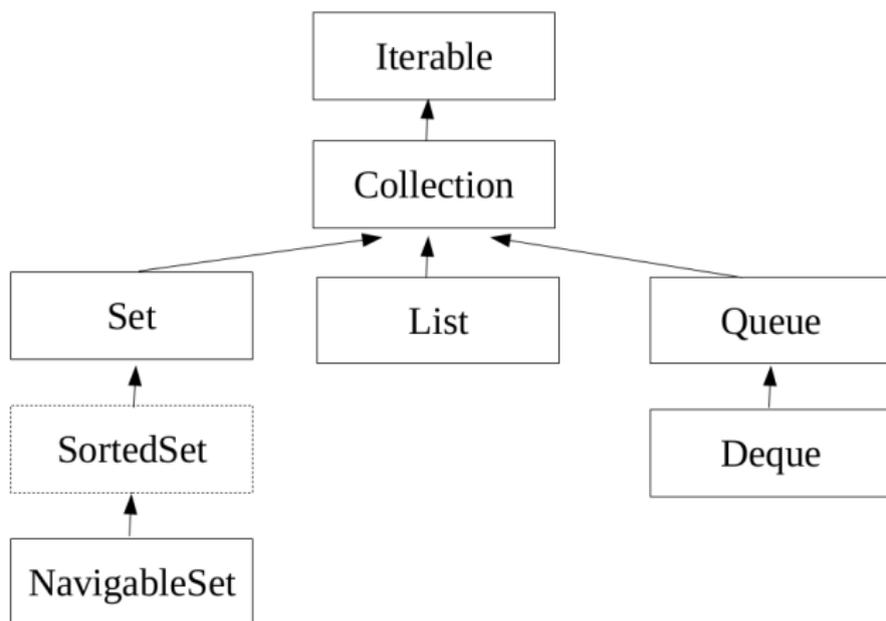
Maps

Iterator

Vues et ponts entre structures

Collections

- ▶ Hiérarchie d'interfaces.
- ▶ Permet de prendre en paramètre ou en type de retour une collection d'objets **sans préciser comment ils sont stockés en mémoire**.



Collections - Définitions abstraites

- ▶ **Collection** : ensemble de données.
- ▶ **Set** : ensemble de données **sans doublon**.
- ▶ **SortedSet** : ensemble de données sans doublon et **trié**.
- ▶ **NavigableSet** : ensemble de données sans doublon, trié et **avec précédent/suivant**.
- ▶ **List** : liste indexée ou séquentielle.
- ▶ **Queue** : file (FIFO).
- ▶ **Deque** : queue avec IO sur chaque côté.

Collections paramétrées

- ▶ Les collections sont **homogènes** : contiennent des éléments qui ont le **même type** (mais pas forcément la même classe).
- ▶ Les collections sont donc **paramétrées** par une variable de type, souvent nommée E (pour Element).
- ▶ Si on veut stocker des éléments de type différent, on utilise le super-type commun (au pire, Object donc).

Interface Collection

- ▶ Interface parente des collections (pas des Maps !).
 - ▶ Méthodes d'ajout d'un ou plusieurs objets.
 - ▶ Test d'appartenance.
 - ▶ Suppression.
 - ▶ Conversion en tableau.
 - ▶ Taille.
 - ▶ Demande d'itérateur.
- ▶ **Complexité variable** selon le type de collection !
- ▶ Certaines opérations sont optionnelles
(`UnsupportedOperationException` levée).

Mutabilité

- ▶ Les collections sont mutables par défaut.
 - ▶ `add(E)`, `remove(Object)`, `removeIf`, `clear`.
- ▶ Les opérations de mutation peuvent lever des `UnsupportedOperationException` pour représenter des **collections non-modifiables**.
- ▶ **Vue** via `Collections.unmodifiableCollection()`
 - ▶ Par ex., plutôt que renvoyer une copie défensive, faire une vue non modifiable

```
1 List<String> list = new ArrayList<>();
2 List<String> list2 = Collections.unmodifiableList(list);
3 list2.add("hello"); // UnsupportedOperationException
4 list.add("hello"); // ok
5 list.add("hello2");
6 list2.size(); // 2 : c'est une vue !
```

Recherche

- ▶ On recherche un élément dans une collection via `boolean contains(Object)`.
- ▶ Complexité ?

Recherche

- ▶ On recherche un élément dans une collection via `boolean contains(Object)`.
- ▶ Complexité ?
- ▶ Dépend de la structure de données !
 - ▶ HashSet en $O(1)$.
 - ▶ TreeSet en $O(\ln n)$.
 - ▶ ArrayList en $O(n)$.

Object

- ▶ Pourquoi dans la JDK pour Set ou List on a par exemple :
 - ▶ `boolean contains(Object o)`
 - ▶ `boolean remove(Object o)`
- et pas des E ?

Object

- ▶ Pourquoi dans la JDK pour Set ou List on a par exemple :
 - ▶ `boolean contains(Object o)`
 - ▶ `boolean remove(Object o)`et pas des E ?
- ▶ `contains` et `remove` utilisent `equals`.
- ▶ `equals` peut renvoyer `true` pour des types différents
 - ▶ Par ex., une `ArrayList` et une `LinkedList` peuvent avoir **le même contenu** (et `equals` renvoie `true`) et pourtant de type différent !

Object

```
1    LinkedList<Integer> ll = new LinkedList<>();
2    ll.add(1);
3
4    ArrayList<Integer> al = new ArrayList<>();
5    al.add(1);
6
7    System.out.println(ll.equals(al)); //true
8
9    Set<LinkedList<Integer>> s = new HashSet<>();
10   s.add(ll);
11   //s.add(al); //compile pas
12
13   System.out.println(s.contains(al)); //true
```

List

- ▶ Liste d'éléments indexé conservant l'ordre d'insertion.
 - ▶ Type les noms des majors de promo chaque année.
- ▶ Des méthodes supplémentaires par rapport à Collection :
 - ▶ `E get(int index), E set(int index, E e)`
 - ▶ `int indexOf(E e), int lastIndexOf(E e)`
 - ▶ `void sort(Comparator<? super E>)` (Java 8).

List

- ▶ Implémentations :
 - ▶ ArrayList : **tableau dynamique**. Ajout fin en $O(1)$, début en $O(n)$, accès en $O(1)$.
 - ▶ Tableau avec taille initiale (10) puis agrandi au besoin.
 - ▶ LinkedList : **liste doublement chaînée**. Ajout fin/début en $O(1)$, accès en $O(n)$.
 - ▶ Utilise plus de mémoire. Temps de parcours plus long.
- ▶ Interface List dangereuse niveau complexité!!

List init

Depuis Java 9, possibilité de créer des listes **non mutables** plus facilement avec `List.of()`

```
1 List<String> l = List.of("salut", "c'est", "cool");
```

List init

Depuis Java 9, possibilité de créer des listes **non mutables** plus facilement avec `List.of()`

```
1 List<String> l = List.of("salut", "c'est", "cool");
```

Identique à

```
1 List<String> l = Arrays.asList("salut", "c'est", "cool");  
2 l = Collections.unmodifiableList(l);
```

List

```
1 private static int sum(List<Integer> l) {
2     int sum = 0;
3     for(int i=0 ; i < l.size() ; ++i) {
4         sum += l.get(i);
5     }
6     return sum;
7 }
8 private static int sum2(List<Integer> l) {
9     int sum = 0;
10    for(int i :l) {
11        sum+=i;
12    }
13    return sum;
14 }
15 public static void main(String[] args) {
16     LinkedList<Integer> l = new LinkedList<>();
17     for(int i=0;i<99999 ; ++i) {l.add(i);}
18     long t1 = System.currentTimeMillis();
19     sum(l);
20     long t2 = System.currentTimeMillis();
21     System.out.println(t2-t1 + " ms");
22     sum2(l);
23     long t3 = System.currentTimeMillis();
24     System.out.println(t3-t2 + " ms");
25 }
```

▶ 14006 ms

▶ 22 ms

Accès

- ▶ L'interface `RandomAccess` est un “marqueur” (n'impose aucune méthode !) pour indiquer que l'implémentation de la liste supporte l'accès à un élément en temps constant.
- ▶ Pas mieux que de faire un `instanceof` moche pour le tester...

List

```
1 private static int sum(List<Integer> l) {  
2     int sum = 0 ;  
3     for(int i=0 ; i < l.size() ; ++i) {  
4         sum += l.get(i) ;  
5     }  
6     return sum ;  
7 }
```

► Problème ?

List

```
1 private static int sum(List<Integer> l) {  
2     int sum = 0 ;  
3     for(int i=0 ; i < l.size() ; ++i) {  
4         sum += l.get(i) ;  
5     }  
6     return sum ;  
7 }
```

- ▶ Problème ?
- ▶ Oui si c'est une LinkedList ! Parcours en $O(n^2)$!
- ▶ Comment faire ?

List

```
1 private static int sum(List<Integer> l) {  
2     int sum = 0 ;  
3     for(int i=0 ; i < l.size() ; ++i) {  
4         sum += l.get(i) ;  
5     }  
6     return sum ;  
7 }
```

- ▶ Problème ?
- ▶ Oui si c'est une LinkedList ! Parcours en $O(n^2)$!
- ▶ Comment faire ?
- ▶ Foreach, qui utilise un iterator (plus de détails plus tard).

AbstractList

- ▶ Manque seulement les méthodes **get et size**.
- ▶ Les autres méthodes (ex. iterator) utilisent get, size... (attention à la complexité !)
- ▶ Évite de tout re-implémenter...

```
1 public class MyArrayList<E> extends AbstractList<E> {
2     private final E[] array ;
3     public MyArrayList(E[] array) {
4         this.array = array ;
5     }
6     @Override
7     public E get(int index) {
8         return array[index] ;
9     }
10    @Override
11    public int size() {
12        return array.length ;
13    }
14    public static void main(String[] args) {
15        String[] a = {"alpha", "beta"} ;
16        List<String> l = new MyArrayList<String>(a) ; //est une List !
17        for(String s :l) { //iterator fourni par AbstractList
18            System.out.println(s) ;
19        }
20    }
21 }
```

Set

- ▶ Représente un ensemble d'éléments **sans doublon**.
- ▶ Différentes implémentations :
 - ▶ HashSet : **Table de hachage**, ensemble sans ordre, add/remove en $O(1)$.
 - ▶ Attention au hashCode>equals !
 - ▶ Grossit en fonction du remplissage (défaut 75%).
 - ▶ TreeSet : **Arbre rouge/noir**, ordre selon un comparateur, add/remove en $O(\ln n)$.
 - ▶ LinkedHashMap : **Table de hachage+liste chaînée** sur les entrées, préserve l'ordre d'insertion, add/remove en $O(1)$. Évite le chaos sur l'ordre d'HashSet sans le coût supplémentaire de TreeSet.
 - ▶ Plus de mémoire utilisé que HashSet.

Set

- ▶ Représente un ensemble d'éléments **sans doublon**.
- ▶ Différentes implémentations :
 - ▶ HashSet : **Table de hachage**, ensemble sans ordre, add/remove en $O(1)$.
 - ▶ Attention au hashCode>equals !
 - ▶ Grossit en fonction du remplissage (défaut 75%).
 - ▶ TreeSet : **Arbre rouge/noir**, ordre selon un comparateur, add/remove en $O(\ln n)$.
 - ▶ LinkedHashSet : **Table de hachage+liste chaînée** sur les entrées, préserve l'ordre d'insertion, add/remove en $O(1)$. Évite le chaos sur l'ordre d'HashSet sans le coût supplémentaire de TreeSet.
 - ▶ Plus de mémoire utilisé que HashSet.
- ▶ Objet algorithmique complexe.
- ▶ Choisir avec soin l'implémentation (et la comprendre...).

Set

- ▶ Exemple : éviter les doublons sur la ligne de commande.
- ▶ `add` renvoie faux si l'élément est déjà présent.

```
1 public static void main(String[] args) {
2     HashSet<String> set = new HashSet<>();
3     for(String arg : args) {
4         if (!set.add(arg)) {
5             System.err.println("argument " + arg + " specified twice");
6             return ;
7         }
8     }
9 }
```

Set - complexités

- Parcours classé par ordre de vitesse pour beaucoup d'éléments.

	size	clear	add	remove	contains	parcours
HashSet	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	3
LinkedHashSet	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	1
TreeSet	$O(1)$	$O(1)$	$O(\ln n)$	$O(\ln n)$	$O(\ln n)$	2

Files / Queues

- ▶ Interface Queue (Java 5).
- ▶ FIFO : insertion en fin, suppression en début.

Deque

- ▶ Depuis Java 6.
- ▶ Hérite de Queue.
- ▶ Interface pour piles/files.
- ▶ Insertion en début ou fin.
- ▶ Suppression en début ou fin.

Queue et Deque - méthodes

- ▶ Des méthodes différentes selon 2 sémantiques :
 - ▶ Lever une exception si vide ou plein.
 - ▶ Valeur de retour null/faux si vide ou plein.

Queue et Deque - méthodes

- ▶ Des méthodes différentes selon 2 sémantiques :
 - ▶ Lever une exception si vide ou plein.
 - ▶ Valeur de retour null/faux si vide ou plein.
- ▶ Demander un élément sans le retirer :
 - ▶ `peek()` ou `element()` et `getFirst()` ou `peekFirst()`.
- ▶ Ajouter en queue :
 - ▶ `add(e)` ou `offer(e)` et `addLast(e)` ou `offerLast()`.
- ▶ Retirer en tête :
 - ▶ `remove()` ou `poll()` et `removeFirst()` ou `pollFirst()`

Queue/Deque - Implémentations

- ▶ `PriorityQueue` (Queue).
 - ▶ ABR codé dans un tableau.
 - ▶ Ordre entre éléments ($O(\ln n)$ pour poll et offer).
- ▶ `ConcurrentLinkedQueue` (Queue).
 - ▶ Autorise les accès concurrents.
- ▶ `LinkedList` (Queue et Deque).
- ▶ `ArrayDeque` (Deque).

Classe Collections

- ▶ Classe Collections (avec un s).
- ▶ **Méthodes statiques utiles** sur les collections.
 - ▶ Recherche binaire dans une collection triée.
 - ▶ Tri, mélange, min, max...
 - ▶ Listes à un seul élément non mutable.
 - ▶ Vues non mutables d'une liste.

Collections - emptyList...

- ▶ Ne pas utiliser `null` quand on s'attend à avoir un tableau ou une collection : NPE ! (si oubli etc).

```
1 public class Shop {
2     public List<Shop> getCheeses() {
3         if(cheesesInStock.size() == 0) return null ; //NON
4         //utiliser Collections.emptyList() !
5     }
6
7     public static void main(String[] args) {
8         List<Cheese> cheeses = shop.getCheeses() ;
9         if(cheeses != null && cheeses.contains(Cheese.CALENDOS)) {
10             ...
11         }
12         //au lieu de if(shop.getCheeses().contains(Cheese.CALENDOS)) {
13     }
14 }
```

- ▶ Stocké en mémoire donc ne crée pas d'objet.

Cours 5 : Les collections

Généralités

Tableaux

Collections

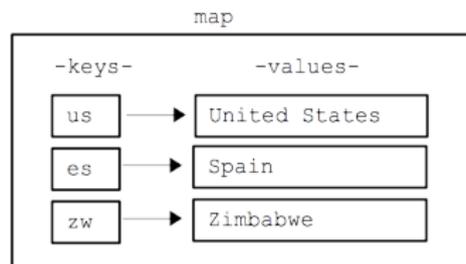
Maps

Iterator

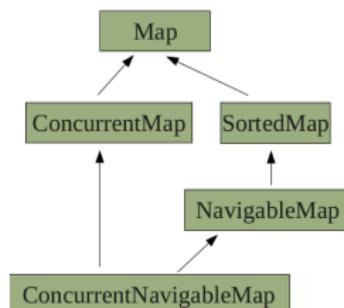
Vues et ponts entre structures

Map

- ▶ Aussi appelé “table associative”, “dictionnaire” ...
- ▶ Deux éléments :
 - ▶ Un élément clef.
 - ▶ Un élément valeur.
- ▶ **Pas de doublon** sur les clefs.
- ▶ **Doublon possible** sur les valeurs.



Maps - Définitions abstraites



- ▶ Map : association sans relation d'ordre.
- ▶ SortedMap : association avec **clefs triées**.
- ▶ NavigableMap : association avec clefs triées et **suivant/précédent**.
- ▶ ConcurrentMap : association avec accès concurrent.

Map - méthodes

- ▶ `V put(K, V)` : insère un couple clef/valeur, supprime le couple précédent avec la même clef, renvoie l'ancienne valeur ou null.
 - ▶ `V putIfAbsent(K, V)` : n'ajoute pas si le couple existe déjà (Java 8).
- ▶ `V get(Object key)` : renvoie la valeur correspondant à la clef ou null si pas de couple correspondant à la clef.
 - ▶ Prend un `Object` car uniquement selon le `equals`
 - ▶ `V getOrDefault(Object key, V defaultValue)` : renvoie une valeur par défaut plutôt que null (Java 8).
- ▶ Faisable en groupé (`putAll`, `replaceAll`).

Implémentations

- ▶ `HashMap` : **table de hachage** sur les clefs
 - ▶ Pas d'ordre sur les couples
 - ▶ Accès/ajout/suppression en $O(1)$
- ▶ `LinkedHashMap` : **Table de hachage sur les clefs + liste doublement chaînée**
 - ▶ Couples ordonnés par ordre d'insertion
 - ▶ Accès/ajout/suppression en $O(1)$
- ▶ `TreeMap` : **arbre rouge/noir**
 - ▶ Couples triés suivant un ordre de comparaison donné
 - ▶ Accès/insertion/suppression en $O(\ln n)$

Exemple

```
1 private static Map<String,String> map = new HashMap<>();
2 static {
3     map.put("France","Paris");
4     map.put("Allemagne","Berlin");
5 }
6 static String getCapital(String country) {
7     String resp = map.get(country);
8     if (resp==null)
9         throw new UnknownCountryException(country);
10    return resp;
11 }
12 static void listKnownCapital() {
13     Set<Map.Entry<String,String>> entries = map.entrySet();
14     for(Map.Entry<String,String> entry :entries) {
15         System.out.println(entry.getKey()+ " has for capital "+ entry.getValue());
16     }
17 }
```

Cours 5 : Les collections

Généralités

Tableaux

Collections

Maps

Iterator

Vues et ponts entre structures

Iterator

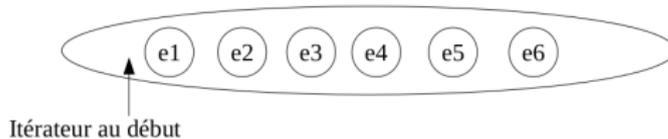
- ▶ Interface Collection a une méthode :
 - ▶ `Iterator<E> iterator()`
- ▶ Renvoie un objet qui implémente l'interface Iterator.
- ▶ Sorte de curseur pour **parcourir une collection élément par élément**.

Iterator

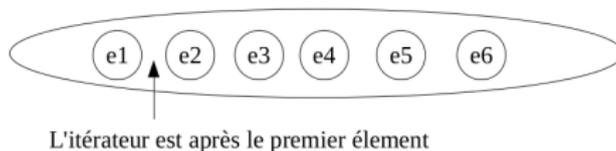
- ▶ Interface `Collection` a une méthode :
 - ▶ `Iterator<E> iterator()`
- ▶ Renvoie un objet qui implémente l'interface `Iterator`.
- ▶ Sorte de curseur pour **parcourir une collection élément par élément**.
- ▶ Permet de parcourir les éléments de la collection :
 - ▶ `boolean hasNext()` : reste des éléments à parcourir ?
 - ▶ `Object next()` : retourne l'élément suivant de la liste (et avance). (Exception si pas d'elt).
 - ▶ `void remove()` : supprime de la collection le dernier élément envoyé par `next` (donc pas possible de faire 2 `remove` de suite sans `next` entre).
- ▶ Parcourir une collection avec un itérateur et la modifier (ajout/supp) (sans utiliser `remove`) est interdit : lève une `ConcurrentModificationException`.

Iterator - utilisation

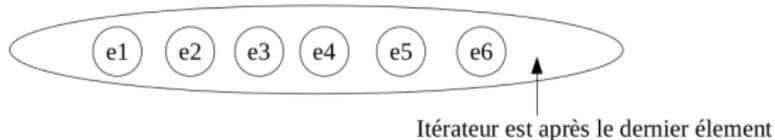
A la création :



Après un appel à `iterator.next()` :



Si `hasNext()` renvoie `false` :



Iterator - intérêt

- ▶ Pas toujours possible d'effectuer un parcours d'une collection (Set, Queue...)
- ▶ Complexité pour certaines.
- ▶ L'itérator garantie un parcours en $O(n)$.

Iterator - utilisation

- Plus efficace pour parcourir une collection sans accès direct !

```
1 private static int sum3(List<Integer> l) {  
2     int sum=0 ;  
3     Iterator<Integer> it = l.iterator() ;  
4     while(it.hasNext()) {  
5         sum += it.next() ;  
6     }  
7     return sum ;  
8 }
```

Iterator - suppression

```
1   LinkedList<String> l = ...
2   for(String s : l) {
3       if(s.length() % 2 == 0){
4           l.remove(s);
5       }
6   }
```

OK?

Iterator - suppression

```
1   LinkedList<String> l = ...
2   for(String s : l) {
3       if(s.length() % 2 == 0){
4           l.remove(s);
5       }
6   }
```

OK?

- ▶ Marche pas ! Et mauvaise complexité !

```
1   LinkedList<String> l = ...
2   Iterator<String> it = l.iterator();
3   while(it.hasNext()) {
4       String s = it.next();
5       if(s.length()%2 == 0) {
6           it.remove();
7       }
8   }
```

Mieux

Iterator - suppression

```
1  LinkedList<String> l = ...
2  for(String s : l) {
3      if(s.length() % 2 == 0){
4          l.remove(s);
5      }
6  }
```

OK?

- ▶ Marche pas ! Et mauvaise complexité !

```
1  LinkedList<String> l = ...
2  Iterator<String> it = l.iterator();
3  while(it.hasNext()) {
4      String s = it.next();
5      if(s.length()%2 == 0) {
6          it.remove();
7      }
8  }
```

Mieux

- ▶ Autre version avec java 8, removeIf et les lambdas...

Iterable

- ▶ Si une classe implémente l'interface `Iterable` : elle est capable de fournir un `Iterator`.
- ▶ `Collection` implémente `Iterable`.
- ▶ Les éléments `Iterable` (et les tableaux) peuvent être utilisés dans un `foreach`.

Iterable

```
1 public class Firm {
2     private final LinkedList<Employee> employees = new LinkedList<>();
3
4     public static void main(String[] args) {
5         Firm f = new Firm();
6         for(Employee e : f) { //impossible
7             System.out.println(e);
8         }
9     }
10 }
```

Iterable

```
1 public class Firm {
2     private final LinkedList<Employee> employees = new LinkedList<>();
3
4     public static void main(String[] args) {
5         Firm f = new Firm();
6         for(Employee e : f) { //impossible
7             System.out.println(e);
8         }
9     }
10 }
```

```
1 public class Firm implements Iterable<Employee>{
2     private final LinkedList<Employee> employees = new LinkedList<>();
3
4     @Override
5     public Iterator<Employee> iterator() {
6         return employees.iterator();
7     }
8     public static void main(String[] args) {
9         Firm f = new Firm();
10        for(Employee e : f) { //ok !
11            System.out.println(e);
12        }
13    }
14 }
```

ListIterator

- ▶ Version spécialisée d'Iterator (en hérite).
- ▶ Ajoute les méthodes add (avant le curseur) et set (du dernier next),
- ▶ et le parcours à l'envers hasPrevious() previous().

```
1 ListIterator<String> li = l.listIterator(l.size()); //demarre à la fin
2 while(li.hasPrevious()) {
3     String s = li.previous();
4 }
```

Cours 5 : Les collections

Généralités

Tableaux

Collections

Maps

Iterator

Vues et ponts entre structures

Ponts

- ▶ Deux types de méthodes de conversions :
 - ▶ **Copier** les données d'une structure vers une autre.
 - ▶ **Voir** une structure comme une autre.
 - ▶ Les données restent dans la structure initiale et sont **vues** dans la nouvelle structure.

Ponts via copie

- ▶ De **collections** vers **tableaux** :
 - ▶ `Object[] toArray()` dans l'interface `Collection`.
- ▶ **Collections** vers **collections** :
 - ▶ Toutes les collections ont un constructeur qui prend une `Collection< ? extends E>`.
 - ▶ `addAll(Collection< ? extends E>)`.
 - ▶ **ATTENTION** copie des **références**!
- ▶ **Tableaux** vers **collections** :
 - ▶ `<T> Collections.addAll(Collection< ? super T>, T... array)`.
- ▶ **Tableaux** vers **tableaux** :
 - ▶ `Arrays.copyOf` (totalité ou début)
 - ▶ `Arrays.copyOfRange` (intervalle)
 - ▶ `System.arraycopy` (plus précis)

Ponts via vue

- ▶ Vue d'une **liste** :
 - ▶ `AbstractList`
 - ▶ `AbstractSequentialList`
- ▶ **Tableau** vers **liste** :
 - ▶ `<T> List<T> Arrays.asList(T...array).`
- ▶ **Liste** vers **liste** :
 - ▶ `l.subList(int start, int end)`

Ponts via vue - exemple

- ▶ Pas de méthode `contains` ou `shuffle` pour les tableaux.
- ▶ On utilise une vue !

```
1 List<String> list = Arrays.asList(args) ;  
2 list.contains("miage"); //miage dans args ?  
3 Collections.shuffle(list); //args shuffled !
```

Ponts via vue - maps

- ▶ Map vers l'ensemble des clefs :
 - ▶ `Set<K> map.keySet()`.
- ▶ Map vers collection de valeurs :
 - ▶ `Collection<V> map.values()`.
- ▶ Map vers couples clef/valeur :
 - ▶ `Set<Map.Entry<K,V>> map.entrySet()`.

Map.Entry

- ▶ Interface interne de Map.
- ▶ Représente des couples clef/valeur mutables.
- ▶ Opérations :
 - ▶ K getKey().
 - ▶ V getValue().
 - ▶ V setValue(V value).

```
1 HashMap<String, Integer> map = ...
2 for(Map.Entry<String,Integer> entry : map.entrySet()) {
3     System.out.println("key " + entry.getKey());
4     System.out.println("value " + entry.getValue());
5 }
```

Conclusion

- ▶ Le choix de la structure de données est important.
- ▶ Selon l'algorithme à implémenter :
 - ▶ Choisir l'interface (List, Set, Queue, Map...).
 - ▶ Choisir l'implémentation.
- ▶ Si hésitation entre deux implémentations, faire des tests avec de larges données.

Quiz

Quiz