

Collections

(d'après Rémi Forax)

Structures de données

En Java, il existe 3 sortes de structures de données

Les tableaux

- Structure de taille fixe, accès direct aux éléments

Les Collections

- Structure modifiable, différents algorithmes de stockage

Les Map

- Structure modifiable, stocke des couples clé -> valeur

Contrats !

Tous les algorithmes sur les structures de données pré-supposent que les classes des objets stockés respectent certains contrats

Il vous faudra peut-être pour cela implanter correctement `equals`, `hashCode`, `toString` et/ou `compareTo` sur les objets de la collection

Si ce n'est pas le cas, cela fait n'importe quoi (avec pas mal de chance une exception est levée)

Example

```
public class StupidInteger {  
    private final int value;  
    public StupidInteger(int value) {  
        this.value = value;  
    }  
}
```

```
StupidInteger[] array = new StupidInteger[1];  
array[0] = new StupidInteger(1);  
Arrays.sort(array); // ClassCastException
```

```
HashSet<StupidInteger> set = new HashSet<>();  
set.add(new StupidInteger(1));  
set.add(new StupidInteger(1)); // no problemo  
set.contains(new StupidInteger(1)); // renvoie false !
```

Null comme valeur d'une collection

Stocker null dans une collection n'est pas une bonne idée car cela plantera quand on sortira l'élément de la collection

De plus, en fonction des versions du JDK, null est accepté ou non

1.0, 1.1, null est pas accepté

HashTable, Vector, Stack

1.2, 1.3, 1.4, null est accepté

HashMap, ArrayList, etc

1.5 ..., null est pas accepté

PriorityQueue, ArrayDeque, etc...

Null comme Collection

On utilise **jamais** null lorsque l'on s'attend à avoir un tableau ou une Collection

```
public Collection<String> getFoo() {  
    return null;    // ahhhh, utiliser  
                   // Collections.emptyCollection()  
}  
  
public String[] getBar() {  
    return null;    // ahhhh, utiliser NULL_ARRAY  
}  
private static final String[] NULL_ARRAY = new String[0];
```

Il existe des collections vides qui sont des constantes,
Collections.**emptySet()**, Collections.**emptyList()**, etc.

Les tableaux

Ils ont une taille fixe définie à l'initialisation et sont toujours mutables (faites des copies défensives !)

Les tableaux sont spécialisés pour les types primitifs

- `byte[]`, `int[]`, `long[]`, `double[]`...
- Ils n'héritent pas de `Object[]` mais de `Object`

Algos classiques (java.util.Arrays)

fill() remplit un tableau (memset en C)

copie

- System.arraycopy (memcpy en C)
- Arrays.copyOf() (créé et recopie un tableau + grand ou + petit)

toString, deepToString, equals, hashCode, etc

Quand utiliser des tableaux

On utilise rarement des tableaux dans le code utilisateur

- demande à connaître la taille à l'avance
- marche pas bien avec les types paramétrés

`new ArrayList<String>[3]` compile pas !

Les tableaux sont utiles principalement lorsque l'on veut s'écrire sa propre structure de données

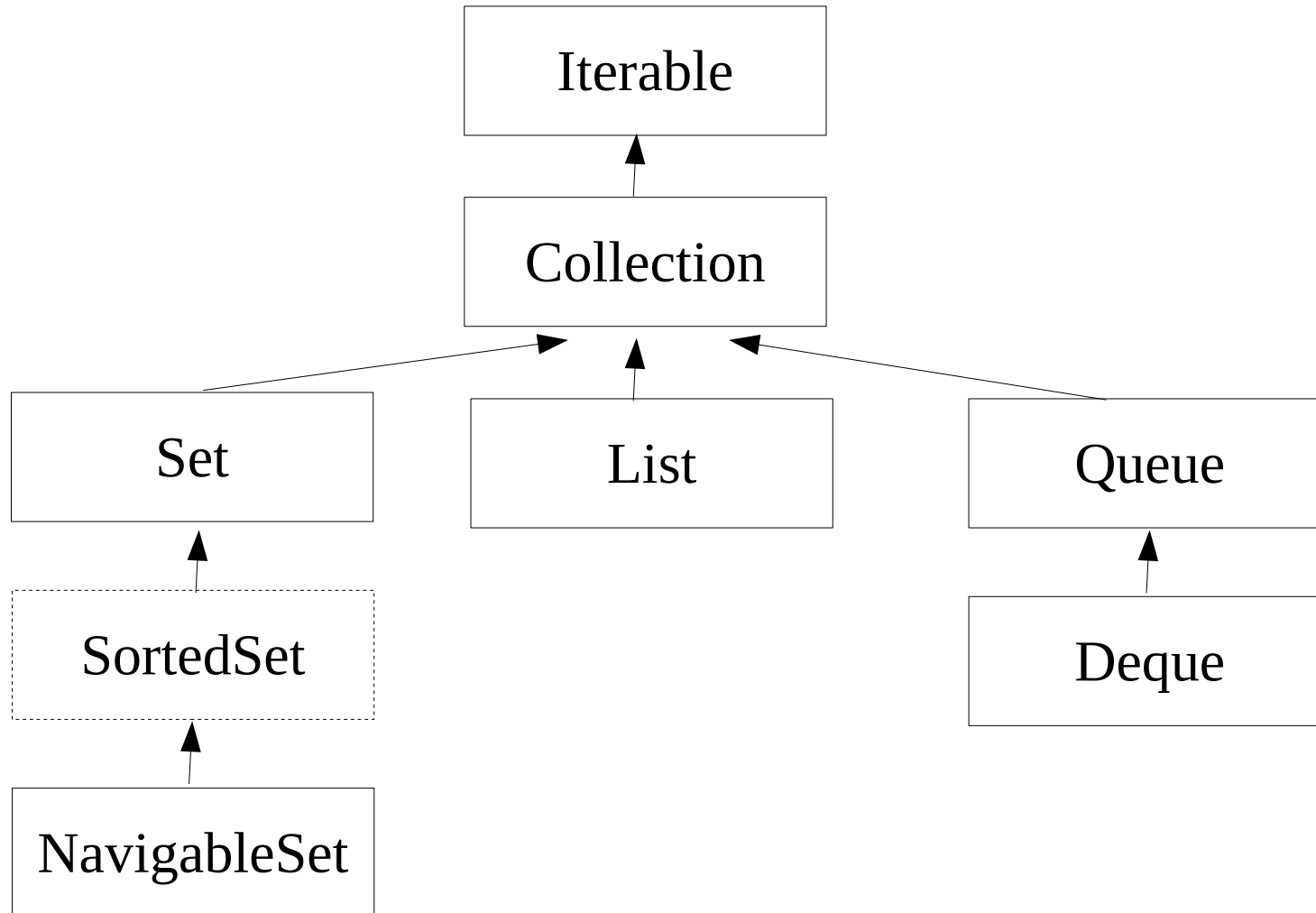
Collection

L'interface `java.util.Collection` est l'interface de base de toutes les structures de données qui stockent des éléments

Il existe 4 sous-interfaces

- Set, ensemble sans doublon
- List, les listes ordonnées et indexées
- Queue, les files (FIFO)
- Deque, les files “*double ended queue*”

La hiérarchie des collections



Collection<E> paramétrée

Les collections sont homogènes, elles contiennent des éléments qui ont le même type (pas forcément la même classe)

Donc elles sont paramétrées par une variable de type, souvent nommée E pour type d'un élément

Collection et type primitif

Les collections ne sont pas spécialisées pour les types primitifs, ce sont des collections d'objets (références)

On utilise le boxing/unboxing si il n'y a pas d'exigence de performance

```
ArrayList<Integer> list = new ArrayList<>();  
list.add(1); // boxing en Integer.valueOf(1)  
int value = list.get(0); // unboxing avec .intValue()
```

attention si on stocke null dans la list,
il y aura un NullPointerException

Collection et taille

A l'exception des collections concurrentes, toutes les collections maintiennent une taille accessible en temps constant ($O(1)$)

int **size()**

permet d'obtenir la taille (sur 31 bits)

boolean **isEmpty()**

permet de savoir si la collection est vide

Collection & mutabilité

Les Collections sont mutables par défaut

boolean **add**(E)

- Ajoute un élément, renvoie vrai si la collection est modifiée

boolean **remove**(Object)

- Supprime un élément, renvoie vrai si la collection est modifiée

boolean **removeIf**(Predicate<? super E> predicate)

- Supprime tous les éléments pour lesquels le predicate est vrai, renvoie vrai si la collection est modifiée

void **clear**()

- Supprime tous les éléments d'une collection (peu utilisée)

Opérations optionnelles

Les opérations de mutation (add, remove, clear, addAll, etc) peuvent lever l'exception `UnsupportedOperationException` pour dire que l'opération n'est pas supportée

Cela permet de représenter des collections non-mutable ou des collections mutables mais de taille fixe

Collection non mutable

`Collections.unmodifiableCollection()`
permet de créer un proxy (relais) implantant seulement les opérations non-optionnelles devant une collection modifiable

```
ArrayList<String> list = new ArrayList<>();  
  
List<String> list2 = Collections.unmodifiableList(list);  
  
list2.add("hello"); // UnsupportedOperationException  
  
list.add("hello"); // ok  
  
list2.size(); // renvoie 1
```

Cela permet d'éviter les copies défensives !

Object ou E ?

Pourquoi remove(**Object**) et add(**E**) ?

A cause des interfaces

```
interface I {}  
interface J {}  
class A implements I, J {}  
  
public static void main(String[] args) {  
    Collection<I> collection = ...  
    A a = new A();  
    collection.add(a); // ok, A est un sous-type de I  
    J j = a;  
    collection.remove(j); // doit pouvoir compiler !  
}
```

Recherche

java.util.Collection possède une méthode de recherche

boolean **contains**(Object)

Renvoie vrai si l'objet est stocké dans la collection

Note sur la complexité:

contains (ou add, remove, etc.) a une complexité différente suivant la structure de donnée

contains pour une HashSet est $O(1)$, pour un TreeSet est $O(\log n)$ et pour une ArrayList $O(n)$

Les méthodes de Object

`equals`, `hashCode` et `toString` sont implantées et délèguent à l'implantation des éléments de la collection

Cela permet d'ajouter une collection à une collection mais dans ce cas la collection ajoutée ne doit pas être modifiée a posteriori !

Ajouter des objets mutables à une collection est potentiellement dangereux !

Exemple

```
ArrayList<String> list = new ArrayList<>();  
HashSet<List<String>> set = new HashSet<>();  
set.add(list);  
list.add("hello");  
set.contains(list); // false :(
```

note : si on a pas de chance, set.contains(list) peut renvoyer vrai car même si la valeur de hashCode a changé, la liste peut être rangée dans la même case de la table de hachage :((

Bulk (opérations groupées)

Les méthodes groupées

boolean **addAll**(Collection<? extends E>)

ajoute tous les éléments de la collection dans this

boolean **removeAll**(Collection<?>)

supprime les éléments de this qui sont aussi dans la collection

boolean **retainAll**(Collection<?>)

retient dans this les éléments qui appartiennent à this et à la collection (intersection)

boolean **containsAll**(Collection<?>)

renvoie vrai si this contient tous les éléments de la collection

Bulk

addAll(), par exemple, est équivalent à

```
public boolean addAll(Collection<? extends E> c) {  
    boolean result = false;  
    for(E element: c) {  
        result |= this.add(element);  
    }  
    return result;  
}
```

mais peut être écrite de façon plus efficace en fonction de l'implantation

java.util.Set

Ensemble d'éléments **sans doublons**

Par ex. les options de la ligne de commande

Exactement la même interface que Collection,
la sémantique des méthodes est pas la même

par ex, add() renvoie false si doublons

Implantations de Set

HashSet

table de hachage

Ensemble sans ordre, add/remove en $O(1)$

LinkedHashSet

Table de hachage + liste chaînée

Ordre d'insertion (ou d'accès), add/remove en $O(1)$

TreeSet

Arbre rouge/noir (ordre de comparaison)

Ordre par un comparateur, add/remove en $O(\log n)$

EnumSet

Bit set

Ordre des valeurs de l'enum, add/remove en $O(1)$

Exemple

Détecter des doublons dans les arguments de la ligne de commande

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

java.util.List

Liste d'éléments **indexée** conservant l'**ordre d'insertion**

Par ex, la liste des vainqueurs du tour de France

Méthodes supplémentaires

E **get**(int index), E **set**(int index, E element)

- accès en utilisant un index

int **indexOf**(E element), **lastIndexOf**(E element)

- comme contains mais qui renvoie un index ou -1

void **sort**(Comparator<? super E>)

- tri en fonction d'un ordre de comparaison

Implantations de List

ArrayList

Tableau dynamique

Ajout à la fin en $O(1)$, ajout au début en $O(n)$,
accès indexé en $O(1)$

LinkedList

List doublement chaînée

Ajout à la fin en $O(1)$, ajout au début en $O(1)$,
accès indexé en **$O(n)$**

Problème de l'interface List

`java.util.List` est une interface dangereuse en terme de complexité

- Accès indexé à une `LinkedList` est en **$O(n)$**
- ajouter un élément en tête d'une `ArrayList` est en **$O(n)$**

On accède **jamais** de façon **indexée** à une `java.util.List`

On peut accéder de façon indexée à une `ArrayList`

java.util.RandomAccess

Les listes à accès indexé en temps constant doivent implanter `java.util.RandomAccess` (marker interface – pas de méthode)

Il est possible de tester à l'exécution si l'accès indexé est en temps constant

```
list instanceof RandomAccess
```

pas très beau mais on a pas mieux :(

Note: ce n'est pas exactement ce que dit la doc de l'interface `RandomAccess` mais c'est vrai en pratique !

Exemple

```
private static int sum(List<Integer> list) {  
    int sum = 0;  
    for(int i = 0; i < list.size(); i++) {  
        sum += list.get(i);  
    }  
    return sum;  
}  
  
public static void main(String[] args) {  
    List<Integer> list;  
    if (args[0].equals("linked")) {  
        list = new LinkedList(1_000_000);  
    } else {  
        list = new ArrayList(1_000_000);  
    }  
    for(int i = 0; i < 1_000_000; i++) {  
        list.add(i);    // boxing... en plus !  
    }  
    System.out.println(sum(list));    // si list est une  
                                       // LinkedList, c'est long !  
}
```

Ne jamais écrire ça !

java.util.Queue

Une file d'éléments FIFO (first-in, first-out)

convention : ajout à la fin, retrait au début pour
add et remove

Méthodes supplémentaires

boolean offer(E)

ajoute à la fin et renvoie vrai si ajouté

E poll()

retire en tête, retourne null si vide

E peek()

regarde l'élément en tête, sans le retirer, null si vide

Summary of Queue methods

	<i>Throws exception</i>	<i>Returns special value</i>
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Map (n'est pas une Collection)

Appelée aussi **table associative** ou dictionnaire, une map fonctionne avec deux éléments

- un élément clé (de type K) et
- un élément valeur (de type V)

Les clés insérées n'ont pas de doublons

Il est possible d'avoir des doublons au niveau des valeurs

java.util.Map<K,V>

méthodes

V put(K key, V value)

insère un couple clé/valeur et supprime si il existe le couple clé/valeur précédent ayant la même clé, renvoie l'ancienne valeur ou null

V get(Object key)

renvoie la valeur correspondant à une clé ou null si il n'y a pas de couple clé/valeur ayant la clé key

isEmpty() et **size()**

renvoie si la map est vide/le nombre de couples clé/valeur

Obtenir un couple

V **getOrDefault**(Object key, V defaultValue)

permet de renvoyer une valeur par défaut au lieu de null comme pour get

V **computeIfAbsent**(K key, Function<K,V> mapFun)

renvoie la valeur associée à la clé, si il n'y a pas de valeur, appelle la fonction pour obtenir valeur et enregistre le couple clé/valeur (pratique pour un cache)

V **computeIfPresent**(K key, BiFunction<K,V,V> remapFun)

remplace et renvoie une nouvelle valeur associée à la clé, à partir de la valeur déjà associée, et la remplace (mise à jour). Retire le couple si le remapping renvoie null.

boolean **containsKey**(Object key)

permet de savoir si la clé existe, **très peu utilisé** car si c'est vrai on va faire un get() ensuite....

Autant autant faire un get() directement et tester s'il est non null !

Insérer/supprimer un couple

en plus de put(), il existe les méthodes

V putIfAbsent(K key, V value)

comme put mais n'ajoute pas value si une valeur est déjà associée à key

V remove(Object key)

boolean remove(Object key, Object value)

supprime un couple (à partir de la clé, ou du couple)

V replace(K key, V value)

boolean replace(K key, V oldValue, V newValue)

remplace un couple (à partir de la clé, ou du couple)

Bulk operations

Opérations groupées

void **putAll**(Map<? extends K, ? extends V> map)

insère tous les éléments de map dans this

void **replaceAll**(BiFunction<? super K, ? super V,
? extends V> fun)

remplace toutes les valeurs des couples existant dans la Map par de nouvelles fournies par la fonction

Il n'y a pas de removeAll(map) car cela peut être fait sur l'ensemble des clés (cf suite du cours)

Map.Entry<K,V>

Interface interne de l'interface Map :
représente un couple <clé,valeur> mutable

Opérations :

K getKey()

- Renvoie la clé

V getValue()

- Renvoie la valeur

V setValue(V value)

- Change la valeur (opération mutable optionnelle)

Exemple

Parcours des couples clés-valeurs

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

en fait il existe une façon plus simple d'écrire le même code (cf chapitre suivant)

HashMap

Implantations

Table de hachage sur les clés

Les couples ne sont pas ordonnés,
insertion/accès/suppression en $O(1)$

LinkedHashMap

Table de hachage sur les clés + liste doublement
chaînée

Couples ordonnés suivant ordre d'insertion (ou
d'accès),
insertion/accès/suppression en $O(1)$

TreeMap

Arbre rouge noir (ordre de comparaison)

Couples ordonnés suivant l'ordre de comparaison,
insertion/accès/suppression en $O(\log n)$

Vues et ponts entre les structures

Ponts et vues entre les collections

Il existe deux types de méthodes permettant de « passer » d'une structure à une autre

- Copier les données d'une structure vers une nouvelle
les données sont dupliquées dans la nouvelle structure
- « Voir » une structure de données comme une autre
les données restent dans la structure initiale et sont aussi vues dans la nouvelle structure, on parle de « **vue** »

Inter-opération par copie

Collection vers les tableaux

`Object[] Collection.toArray()`

Créer un tableau d'Object

`<T> T[] Collection.toArray(T[] array)`

Utilise le tableau pris en paramètre, ou crée un nouveau tableau si le tableau est trop petit

- si le tableau est trop grand, on met null après le dernier élément ??

Exemple

```
ArrayList<Object> list = ...
```

```
// on supprime tout ce qui n'est pas des Strings
```

```
list.removeIf(e -> !(e instanceof String));
```

```
// puis on met le résultat dans un tableau de String
```

```
String[] array = list.toArray(new String[0]);
```

Le code est bizarre mais idiomatique

Inter-opération par **copie**

Collection vers Collection

Toutes les collections ont un constructeur qui prend une `Collection<? extends E>`

Sur une collection `addAll(Collection<? extends E>)` permet d'ajouter

Tableau vers Collection

```
<T> Collections.addAll(  
    Collection<? super T> coll, T... array)
```

Inter-opération par **vue**

Tableau vers List

`<T> List<T> Arrays.asList(T... array)`

List vers List

`list.subList(int start, int end)` permet d'obtenir une sous-liste d'une liste

Map vers Set

`Set<E> Collections.newSetFromMap(
Map<E, Boolean> map)`
pour bénéficier de l'implémentation de la Map...

Queue vers Queue

`Queue<T> Collections.asLifoQueue(Deque<T> deque)`
pour bénéficier d'une pile avec une Queue

Exemple

Il n'existe pas de méthode `contains()` ou `shuffle()` qui prend en paramètre un tableau dans `java.util.Arrays` mais en utilisant une vue ...

```
public static void main(String[] args) {  
    List<String> list = Arrays.asList(args);  
    list.contains("bingo");  
        // search if "bingo" is  
contained in args  
    Collections.shuffle(list);  
        // the array args is now shuffled  
}
```

Inter-opération par vue

Map vers l'ensemble des clés

```
Set<K> map.keySet()
```

Map vers la collection des valeurs

```
Collection<V> map.values()
```

Map vers l'ensemble des couples clé/valeur

```
Set<Map.Entry<K,V>> map.entrySet()
```

Itération

Itération Interne vs Externe

Il existe deux types d'itérations sur une collection

- Itération interne

On envoie du code (sous forme de lambda) à exécuter par la structure de données, la structure parcourt sa structure et pour chaque élément appelle la lambda avec l'élément en paramètre

- Itération externe

On demande à la collection un curseur (un Iterator) qui va servir à parcourir la collection en retournant un élément à chaque appel

`java.util.Iterable` sert d'interface pour les 2 façons d'itérer

Itération interne

`Iterable.forEach(Consumer<? super E> consumer)`

avec

```
@FunctionalInterface
    public interface Consumer<T> {
        public void accept(T t);
    }
```

exemple :

```
List<String> list = ...
list.forEach(e -> System.out.println(e));
```

Itération interne sur une Map

`Map.forEach(BiConsumer<? super K, ? super V> bc)`

avec

```
@FunctionalInterface
public interface BiConsumer<T, U> {
    public void accept(T t, U u);
}
```

exemple :

```
HashMap<String, Integer> map = ...
map.forEach((key, value) -> {
    System.out.println("key: " + key +
                       " value: " + value);
});
```

Itération externe

```
Iterator<E> Iterable.iterator()
```

existe t'il un élément suivant ?
la méthode n'a pas d'effet de bord

avec

```
public interface Iterator<E> {  
    public boolean hasNext();  
    public E next();  
}
```

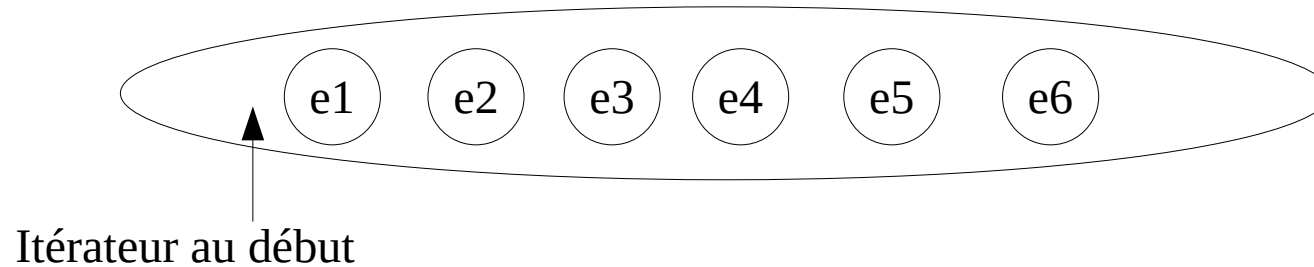
renvoie l'élément courant et passe au
suivant ou NoSuchElementException

exemple :

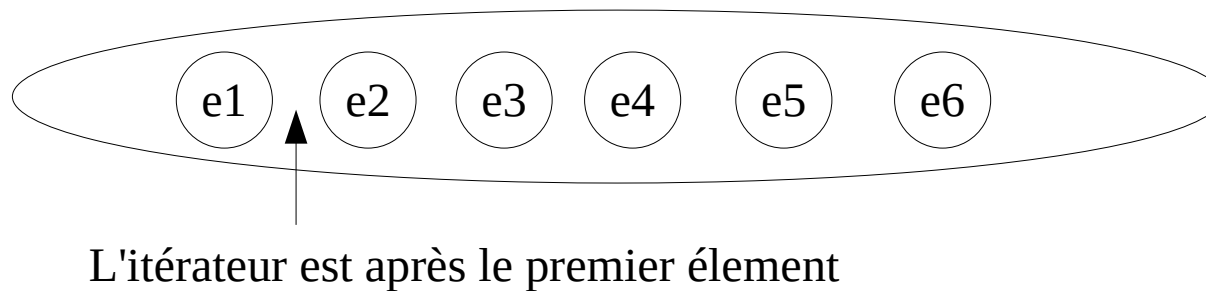
```
List<String> list = ...  
Iterator<String> it = list.iterator();  
while(it.hasNext()) {  
    System.out.println(it.next());  
}
```

Position de l'itérateur

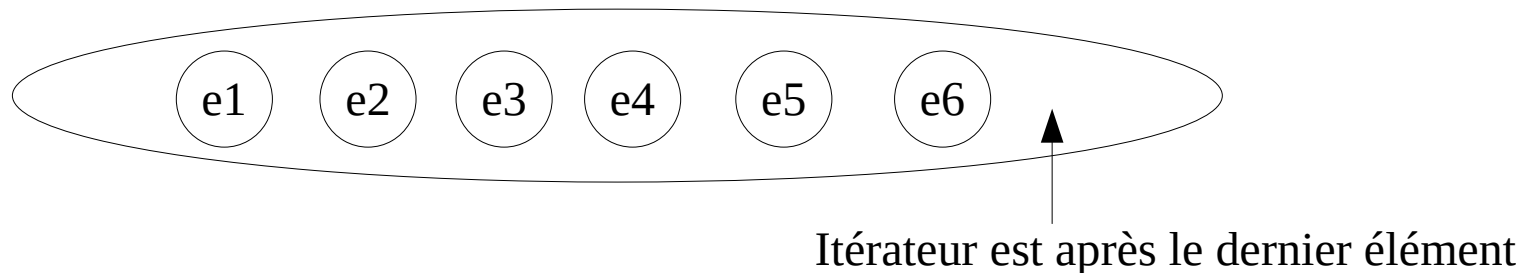
A la création :



Après un appel à `iterator.next()` :



Si `hasNext()` renvoie `false` :



Syntaxe pour l'itération externe

La boucle `for (Type var: iterable)` sur un **Iterable** est transformée en

```
Iterator<type> it = iterable.iterator();  
while (it.hasNext()) {  
    type var = it.next();  
    ...  
}
```

Attention à ne pas confondre avec
`for (type var: array)` qui parcourt le **tableau**
avec des index

Mutation de la collection et itération

Il est **interdit** d'effectuer une **mutation** sur la structure de données **durant l'itération** !

Itération interne :

```
Set<String> set = ...  
set.forEach(e -> set.add(e));  
    // throws ConcurrentModificationException
```

Itération externe :

```
Set<String> set = ...  
for(String e: set) {  
    set.add(e);  
} // throws ConcurrentModificationException
```

ConcurrentModificationException

Exception levée lorsque, durant le parcours, on modifie la structure de données sur laquelle on itère

L'exception est **mal nommée**

Concurrent veut dire normalement accédé par différents fils d'exécution (thread) mais ce n'est pas le cas ici :(

La technique qui consiste lors du parcours à regarder si une mutation a été effectuée ou pas pour remonter le problème ASAP est appelée **fail-fast**.

Mutations lors du parcours

Il est possible d'éviter la levée de `ConcurrentModificationException` dans le cas d'une itération externe, car il est possible de faire les mutations directement sur l'itérateur

- **Iterator** possède une méthode **remove()**
- **ListIterator** qui est une version spécialisée de l'itérateur (et donc hérite de `Iterator`) pour les listes possède en plus les méthodes **add()** et **set()**

Iterator.remove

Supprime l'élément qui a été renvoyé précédemment par next()

- donc next() doit être appelé d'abord
- Il n'est donc pas possible de faire 2 remove() de suite sans appeler next() entre les deux
- Lève IllegalStateException si next() pas appelé avant

Exemples

Exemple qui ne marche pas (CME) :

```
LinkedList<String> list = ...  
for(String s: list) {  
    if (s.length() % 2 == 0) {  
        list.remove(s); // en + complexité affreuse  
    }  
}
```

Exemple qui marche :

```
Iterator<String> it = list.iterator();  
while(it.hasNext()) {  
    String s = it.next();  
    if (s.length() % 2 == 0) {  
        it.remove();  
    }  
}
```

Exemples (suite)

il y a une façon plus simple, au lieu d'utiliser l'itérateur:

```
Iterator<String> it = list.iterator();  
while(it.hasNext()) {  
    String s = it.next();  
    if (s.length() % 2 == 0) {  
        it.remove();  
    }  
}
```

On peut aussi utiliser `removeIf` !

```
list.removeIf(s -> s.length() % 2 == 0);
```

Itération à l'envers

Sur une List, il est possible de parcourir celle-ci du dernier élément au premier en utilisant le ListIterator

Exemple:

```
List<String> list = ...
ListIterator<String> it =
list.listIterator(list.size());
           // on place l'itérateur à la fin
while(it.hasPrevious()) {
    String s = it.previous();
    ...
}
```

Parcours indexé

Attention, le parcours indexé peut être **mortel**

Exemple :

```
List<String> list = ...  
for(int i = 0, i < list.size(); i++) {  
    String s = list.get(i);  
    ...  
}
```

Si `list.get()` est pas en temps constant,
alors la complexité est $O(n^2)$

L'itération externe sur une List doit se faire avec un Iterator sauf si on est sûr que la liste implante RandomAccess.

Itération interne vs externe

L'itération interne est souvent plus efficace mais comme on envoie une lambda à `iterable.forEach()`, on est limité par le fait qu'une lambda n'est pas une closure

il n'est pas possible d'effectuer des mutations sur les variables locales à l'intérieur d'une lambda

De plus, l'itération externe permet aussi la composition d'itérateurs

Legacy Collections

Legacy collection

java.util existe depuis 1.0 mais l'API des collections n'existe que depuis la version 1.2

Vector, Stack, Hashtable et Enumeration sont des anciennes classes

- qui ne doivent plus être utilisées à part pour discuter avec du code legacy
- Les 3 premières ont des problèmes de performance car leurs méthodes sont synchronized

Classe de remplacement

Les classes de remplacement ont la même API (ou une API très similaire) que les classes legacy

Vector -> ArrayList

Stack -> ArrayDeque

Hashtable -> HashMap

Enumeration -> Iterator

plus la méthode `Collections.list(enumeration)` -> List