Les collections

Rémi Forax

Plan

- Tableaux, Collection, Map
- Vues et ponts entre les structures
- Itération
- Legacy

Structures de données

En Java, il existe 3 sortes de structures de données

Les tableaux

- Structure de taille fixe, accès direct aux élements

Les Collections

- Structure modifiable, différents algorithmes de stockage

Les Map

- Structure modifiable, stocke des couples clé -> valeur

Contrats !

Tous les algorithmes sur les structures données pré-suppose que les classes des objets stockés respect un certain contrat

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

Si ce n'est pas le cas, au mieux une exception est levée, au pire cela fait n'importe quoi

Exemples

```
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.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 qund on sortira l'élement 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;  // ahhh, 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 vide qui sont des constantes, Collections.**emptySet**(), Collections.**emptyList**(), etc.

Les tableaux

Ils ont une taille fixe définie à l'initialisation et sont toujours mutable (copie défensive!)

Les tableaux sont spécialisés pour les types primitifs

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

Pour les tableaux d'objets, les cases sont initialisés à null (=> ahhh), pour les primitifs, les cases sont initialisé à false, 0, 0.0

Les tableaux

Les tableaux hérite de Object (ou Object[] qui hérite de Object) et n'ont pas les méthodes toString, equals et hashCode redéfinie

- en fait seul clone() est redéfinie

Quasiment toutes les méthodes (statique) de manipulation des tableaux sont regroupés dans java.util.Arrays

Algos classiques (java.util.Arrays)

fill() remplie un tableau (memset en C)

copies

- System.arrayCopy (memcopy comme en C)
- Arrays.copyOf() (créé et recopie un tableau + grand ou + petit)
- Object.clone() duplique mais on utilise plutôt Arrays.copyOf(array, array.length)

toString, deepToString, equals, hashCode, etc

Collection

L'interface java.util.Collection est l'interface de base de toutes les structures de donnée qui stocke des éléments

Il existe 4 sous-interfaces

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

Collection<E> paramétrée

Les collections sont homogénes, elle contiennent des élements qui ont le même type (pas forcément la même classe)

donc elles sont paramétrés par une variable de type, souvent nommée E pour type d'un élement

Collection et type primitif

Les collections ne sont pas spécialisées pour les types primitfs, ce sont des collections d'objet

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 élement, renvoie vrai si la collection est modifiée

boolean **remove**(Object)

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

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

- Supprime un élement si le predicate est vrai, renvoie vrai si la collection est modifiée

void **clear**()

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

Operations optionnelles

Les opérations de mutations (add, remove, clear, addAll, etc) peuvent levées l'exception UnsupportedOperationException pour dire que 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.unmodifiable*Collection*() permet de créer un proxy implantant seulement les opération non-optionnel 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 copie défensive !

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 être valide
}
```

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(ln n) et pour une ArrayList O(n)

Les méthodes de object

equals, hashCode et toString sont implantés et délègue à l'implantation des élements de la collection

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

- Ajouter des objets **mutables** à une collection est poetentiellement **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 groupés

- boolean **addAll**(Collection<? extends E>)
  - ajoute tous les élements de la collection dans this
- boolean **removeAll**(Collection<?>)
  - supprime les élements de this qui sont aussi dans la collection
- boolean **retainAll**(Collection<?>)
  - retient dans this les élements qui appartiennent à this et à la collection (intersection)
- boolean **containAll**(Collection<?>)
  - renvoie vrai si this contient tous les élements 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

Les concepts de collections

Il existe 4 sous-interfaces

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

java.util.Set

Ensemble d'élements **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 + list chainée

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(ln 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'élement **indexé** 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>)

- Trie en fonction d'un ordre de comparaison

Implantations de List

ArrayList

Tableau dynamique

Ajout à la fin en 0(1), ajout au début en O(n), accès indexé en O(1)

LinkedList

List doublement chaînée

Ajout à la fin en 0(1), ajout au début en O(1), accès indexé en O(n)

Problème de l'interface List

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

- Accès indexé à une LinkedList est en O(n)
- ajouter un élement en tête d'une ArrayList est en O(n)

On accède pas de façon indexée à une java.util.List, pas de problème si c'est une ArrayList

java.util.RandomAccess

Les listes à accès indexées en tant constant doivent implanter java.util.Random (marker interface)

Il est possible de tester à l'exécution l'accès indexée 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

Liste ou tableau ?

Si on connait le nombre d'éléments, on utilise plutôt un tableau car c'est plus efficace

Mais un tableau de type paramétré est unsafe

- Il est habituelle dans du code utilisateur d'utiliser List<Set<String>> au lieu de Set<String>[]

Il est aussi possible de voir un tableau comme une liste (cf plus tard)

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);
  }
  System.out.println(sum(list));    // si list est une LinkedList, c'est long !
}
```

java.util.Queue

Représente une file d'élements FIFO
(la convention est ajout à la fin et retire au début)

- Par ex, une file d'attente

Méthodes supplémentaires

boolean **offer**(E)

ajoute à la fin, renvoie vrai si ajouté

E **poll**()

retire en tête, null si vide

E **peek**()

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

Implantations de Queue

- ArrayDeque
  - Tableau dynamique circulaire
  - Ajout/suppression en O(1)
- PriorityQueue
  - Tas (ordre de comparaison)
  - Ajout/suppression en O(ln n)
- LinkedList
  - Liste double chainée
  - Ajout/suppression en O(1)

Autre sémantique

En plus de offer/poll/peek, il existe 3 autres méthodes qui ont la même sémantique mais plante au lieu de renvoyer null/false,
elles sont peu utilisées en pratique

boolean **add**(E element)

comme offer() mais lève IllegalArgumentException si plein

E **remove**()

comme poll() mais lève NoSuchElementException si vide

E **element**()

comme peek() mais lève NoSuchElementException si vide

Exemple

```
public class Tree {
  private final int value;
  private final Tree left, right;
  public Tree(int value, Tree left, Tree right) {
    this.value = value;
    this.left = left;
    this.right = right;
  }
  public void breadthSearch(Consumer<? super Tree> consumer) {
    ArrayDeque<Tree> queue = new ArrayDeque<>();
    queue.offer(this);
    while (!queue.isEmpty()) {
      Tree tree = queue.poll();
      consumer.accept(tree);
      Tree left = tree.left, right = tree.right;
      if (left != null) { queue.offer(left); }
      if (right != null) { queue.offer(right); }
    }
  }
  public static void main(String[] args) {
    Tree tree = ...
     tree.breadthSearch(System.out::println);
  }
}
```

java.util.Deque

Sous-type de Queue qui permet d'ajouter ou retirer en début ou en fin de la file

Méthodes supplémentaires

boolean **offerFirst**(E), boolean **offerLast**(E)

ajouté au début ou à la fin

E **pollFirst**(), E **pollLast**()

retirer au début ou à la fin

E **peekFirst**(E), E **peekLast**()

voir sans retirer au début ou à la fin

et aussi addFirst(E), addLast(E), E removeFirst(), E removeLast(), E getFirst(), E getLast()

Map

Appelée aussi table associative ou dictionnaire une map fonctionne avec deux élements

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

Les clés insérés 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)

  insére 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

Implantations

HashMap

Table de hachage sur les clés

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

LinkedHashMap

Table de hachage sur les clés + liste doublement chaînée

Couples ordonnées par ordre d'insertion (ou d'accès), insertion/accès/suppression en O(1)

TreeMap

Arbre rouge noir (ordre de comparaison)

Couple ordonée suivent l'ordre de comparaison, insertion/accès/suppression en O(ln n)

IdentityHashMap

Table de hachage fermé, ==/System.identityHashCode pour le test égalité/hashCode

Couple non ordonée, insertion/accès/supression en O(1)

WeakHashMap

Table de hachage avec clés stocké dans des références faibles

Couple non ordonée, insertion/accès/supression en O(1)

Obtenir un couple

en plus de get(), il existe les méthodes

V **getOrDefault**(V defaultValue)

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

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

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

boolean **containsKey**()

permet de savoir si la clé existe, très peut utilisé car

- soit cela veut dire que l'on veut un Set
- soit que l'on va faire un get() derrière dans ce cas autant faire un get() uniquement

Insérer/supprimer un couple

en plus de put(), il existe les méthodes

V **putIfAbsent**(K key,V value)

comme put mais n'ajoute pas un couple existe déjà

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

Opérations groupées

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

insére tous les élements 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 pas 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 des couples clé/valeur mutable

Opérations

K **getKey**()

renvoie la clé du couple

V **getValue**()

renvoie la valeur du couple

V **setValue**(V value)

change la valeur du couple (opération mutable optionelle)

Implantations de Map.Entry

Map.Entry possède deux implantations, déclarés bizarrement en tant que classe interne de AbstractMap

- implantent aussi equals/hashCode/toString

AbstractMap.SimpleImmutableEntry<K,V>

- version non mutable, setValue jète une UOE

AbstractMap.SimpleEntry<K,V>

- version mutable, equals/hashCode/toString passe par les getters et donc ils peuvent être redéfinie

Exemple

Calculer un histogramme d'occurences

```
public static <T> void histo(
        List<? extends T> list, Map<? super T, Integer> map) {
  for(T element: list) {
    map.put(element, 1 + map.getOrDefault(element, 0));
  }
}
public static void main(String[] args) {
  HashMap<String, Integer> map = new HashMap<>();
  histo(Arrays.asList(args), map);
  System.out.println(map);
}
```

Exemple

```
public class Trie {
  private boolean terminal;
  private final HashMap<Character,Trie> map = new HashMap<>();
  public void add(String s) {
    Trie trie = this;
    for(int i = 0; i < s.length(); i++) {
      char letter = s.charAt(i);
      trie = trie.map.computeIfAbsent(letter, l -> new Trie());
    }
    trie.terminal = true;
  }
  public boolean contains(String s) {
    Trie trie = this;
    for(int i = 0; i < s.length(); i++) {
      char letter = s.charAt(i);
      trie = trie.map.get(letter);
      if (trie == null) { return false; }
    }
    return trie.terminal;
  }
}
```

Vues et ponts entre les structures

Pont et vue entre les collections

Il existe deux types méthodes de “conversions”

- Copier les données d'une structure vers une nouvelle

  les données sont dupliquer dans la nouvelle structure

- Voir une structure de donnée comme une autre

  les données reste dans la structure initiale et sont aussi vue dans la nouvelle structure, on parle de vue

Interopération 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 élement ??

Interopération 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)

Interopération 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)

Queue vers Queue

Queue<T> Collections.asLifoQueue(
Deque<T> deque)

Interopération 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()

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 suffled
}
```

Itération

Itération Interne vs Externe

Il existe deux types d'itération sur une collection

- Itération interne

  On envoie du code (sous forme de lambda) à exécuter par la structure de donnée, la structure parcours sa structure et pour chaque élement appel la lambda avec l'élement en paramètre

- Itération externe

  On demande à la collection un curseur (un Iterator) qui va servir à parcourir la collection en retournant un élement 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 :

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

Iteration externe

Iterator<E> Iterable.iterator()

avec

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

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

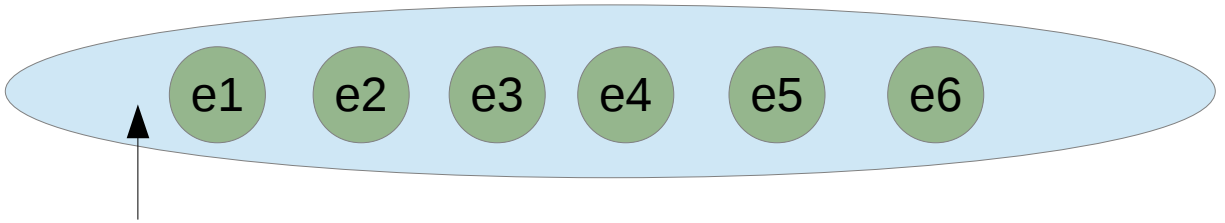
renvoie l'élement courant et passe au suivant ou NoSuchElementException

exemple :

```
List<String> list = ...
Iterator<String> it = list.iterator(); // on récupère le 'parcoureur'
while(it.hasNext()) {
  System.out.println(it.next());
}
```

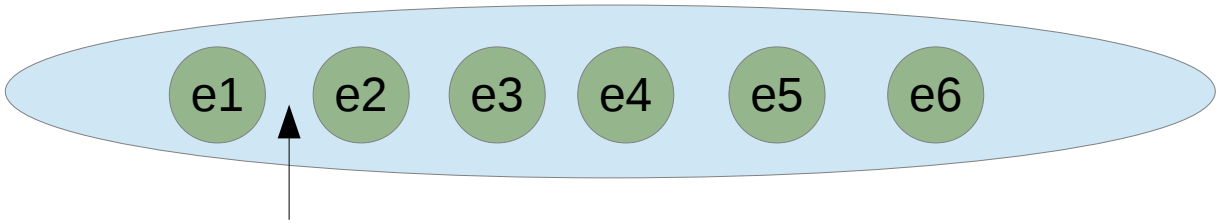
Position de l'itérateur

A la création :



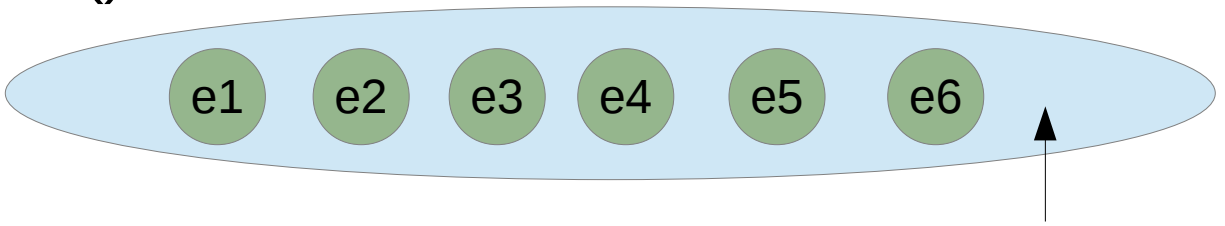
Itérateur au début

Après un appel à iterator.next() :



L'itérateur est après le premier élement

Si hasNext() renvoie false :



Itérateur est après le dernier élement

Syntaxe pour l'itération externe

La boucle for(type var: iterable) sur un **Iterable** est transformé en

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

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

Ecrire son itérateur

2 phases

- On écrit le code itératif sans itérateur
- On répartit le code dans l'itérateur

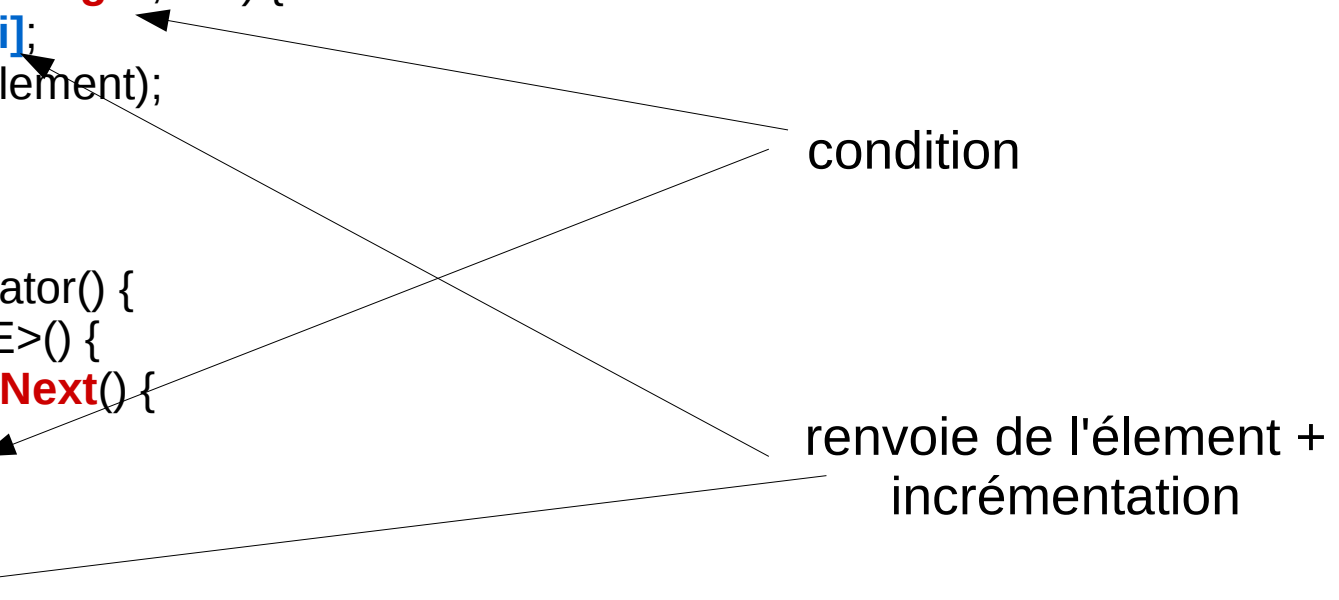
Par exemple, avec un code d'itération interne

```
public class MyArray<E> {
  private final E[] array;
  ...
  public void forEach(Consumer<? super E> consumer) {
    for(int i = 0; i < array.length; i++) {
      E element = array[i];
      consumer.accept(element);
    }
  }
}
```

Ecrire son itérateur (2)

```
public class MyArray<E> {
  private final E[] array;
  ...
  public void forEach(Consumer<? super E> consumer) {
    for(int i = 0; i < array.length; i++) {
      E element = array[i];
      consumer.accept(element);
    }
  }

  public Iterator<E> iterator() {
    return new Iterator<E>() {
      public boolean hasNext() {
        ...
      }
      public E next() {
        ...
      }
    }
  }
}
```



Ecrire son itérateur (3)

```
public class MyArray<E> {
  private final E[] array;
  ...
  public void forEach(Consumer<? super E> consumer) {
    for(int i = 0; i < array.length; i++) {
      E element = array[i];
      consumer.accept(element);
    }
  }

  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private int i = 0;
      public boolean hasNext() {
        return i < array.length;
      }
      public E next() {
        E element = array[i];
        i++;
        return e;
      }
    }
  }
}
```

La variable locale i
se transforme en champ

Attention cet Iterateur est **FAUX** !

Interface Iterator !

L'interface Iterator n'oblige pas à ce que hasNext() et next() soit appelé dans cette ordre

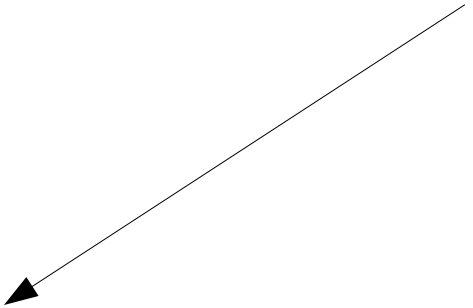
Il faut vérifier qu'un code utilisateur puisse appeler

- hasNext() plusieurs fois sans appel à next()
- next() sans appeler une seul fois hasNext()

Ecrire son itérateur (3)

```
public class MyArray<E> {
  private final E[] array;
  ...
  public void forEach(Consumer<? super E> consumer) {
    for(int i = 0; i < array.length; i++) {
      E element = array[i];
      consumer.accept(element);
    }
  }
  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private int i = 0;
      public boolean hasNext() {
        return i < array.length;
      }
      public E next() {
        if (!hasNext()) { throw new NoSuchElementException(); }
        E element = array[i];
        i++;
        return e;
      }
    }
  }
}
```

next() peut être appelée sans appel à hasNext()



Mutation de la collection et itération

Il est **interdit** d'effectuer une **mutation** sur la structure de donnée **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 donnée sur laquelle on itére

- Le nom de l'exception est vraiment pas terrible
- Concurrent veut dire normalement accéder 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 est appelée fail-fast.

Mutations lors du parcours

Il est possible d'éviter la lévée 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 spécialiser de l'itérateur (et donc hérite de Iterator) pour les listes possèdes en plus les méthodes **add**() et **set**()

Iterator.remove

Méthode par défaut dans Iterator

Supprime l'élement qui a été renvoyer précédemment par next()

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

L'implantation par défaut (Java8) dans Iterator lève UnsupportedOperationException

Exemples

Exemple qui ne marche pas (CME) :

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

Exemple qui marche :

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

Iteration à l'envers

Sur une List, il est possible de parcourir celle-ci du dernier élement 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 external ce se fait avec l'Iterator à part si on est sûre que la liste implante RandomAccess.

ltération interne vs externe

L'itération interne est souvent plus efficace car

- Il n'y a pas deux appels (hasNext()/next())
- Les checks de mutation (failfast) peuvent être fait une seul fois à la fin

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 lamnbda

L'itération externe permet aussi la composition d'itérateurs

Ordre et trie

Ordre et trie

Pour pouvoir trier ou ordoner des objects, il faut une fonction de comparaison

Il y a deux sortes de façon de specifier un ordre en Java

- Ordre des élements, l'ordre est définie sur les élements de la collection grâce à l'interface Comparable
- Ordre externe, l'ordre est spécifier par un objet externe un Comparator

Ordre des élements

Ordre des élements ou ordre naturel est spécifié en implantant l'interface Comparable

```
public interface Comparable<E> {
  int compareTo(E element);
}
```

a.compareTo(b) < 0 <=> a < b
a.compareTo(b) > 0 <=> a > b
a.compareTo(b) == 0 <=> a == b

CompareTo doit marcher avec equals,
a.compareTo(b) <=> a.equals(b) == true

Ordre extérieur

Un ordre extérieur est une fonction de comparaison externe. Cela permet de spécifier un autre ordre que l'ordre naturel.

```
public interface Comparator<T> {
  int compare(T t1, T t2);
}
```

compare doit aussi être compatible avec equals,
compare(a, b) == 0 <=> a.equals(b) == true

Exemple

En utilisant l'ordre naturel, java.lang.String implante l'interface Comparable

```
String[] array = new String[] { "foo", "bar" };
Arrays.sort(array);
```

En utilisant un ordre externe

```
List<String> list = Arrays.asList("foo", "bar");
list.sort((s1, s2) -> s1.trim().compare(s2.trim());
```

Implanter un Comparable ou un Comparator

Trouver les 2 bugs ?

```
public class Point implements Comparable<Point> {
  private final int x;
  private final int y;

  public Point(int x, int y) { … }

  public int compareTo(Point p) {
    int dx = x – p.x;
    if (dx != 0) { return dx; }
    return y – p.y;
  }
}
```

Implanter un Comparable ou un Comparator

Il manque le equals() et il y a un problème d'overflow

```
public class Point implements Comparable<Point> {
  private final int x;
  private final int y;

  public int compareTo(Point p) {
    int dx = Integer.compare(x, p.x);
    if (dx != 0) { return dx; }
    return Integer.compare(y, p.y);
  }

  public boolean equals(Object o) {
    if (!(o instanceof Point)) { return false; }
    Point p = (Point)o;
    return x == p.x && y == p.y;
  }
}
```

Comparaison par rapport à un champs

Comparator possède des méthodes statique comparing(), comparing**Int**(), etc. de comparaison par rapport à un champ

```
public class Author {
  private final String name;

  public String getName() { return name; }
  ...
}

List<Author> list = ...
list.sort(Comparator.comparing(Author::getName));
```

Trier

Il y a deux solutions

- Utiliser une collection qui garde un ordre
  - new **TreeSet**<E>(Comparator<? super E> c)
  - new **TreeMap**<K,V>(Comparator<? super K> c)
    - On a un ordre uniquement sur les clés
- Trier une collection
  - list.**sort**(Comparator<? super E> c)
  - <T> Arrays.**sort**(T[], Comparator<? super T> c)
  - <T> Arrays.**parallelSort**(T[], Comparator<? super T> c)
    - Ils existent aussi des versions pour les types primitifs

BinarySearch

Pour faire une recherche dichotomique (couper en deux)

sur un tableau

Arrays.**binarySearch**(T[] array, T element,
Comparator<? super T> c)

sur une liste

<T> Collections.**binarySearch**(List<? extends T> list,
T element, Comparator<? super T> c)

Il faut pas oublier que la structure doit être triée !

NavigableSet

interfaces sous-type de Set qui permet d'obtenir pour une valeur l'élement précédent ou suivant de la collection

Opérations

E floor(E) <=, lower(E) <, ceiling(E) >=, higher(E) >

- Recherche d'un élement

NavigableSet<E> headSet(E element, boolean inclusive)

NavigableSet (vue) entre first() et element

NavigableSet<E> subSet(E from, boolean, E to, boolean)

NavigableSet (vue) entre from et to

NavigableSet<E> tailSet(E element, boolean inclusive)

NavigableSet (vue) entre element et last()

NavigableMap

interfaces sous-type de Map,
permet d'obtenir pour une valeur (de type clé K)
le couple clé/valeur précédent/suivant

Opérations

- Entry<K,V> floorEntry(K) <=, lowerEntry(K) <, ceilingEntry(K) >=, higherEntry(K) >
- NavigableMap<K,V> headMap, subMap, tailMap
- NavigableKeySet<K> navigableKeySet()
  - Ensemble des clés sous forme d'un navigable set
- NavigableMap<K,V> descendingMap()
  - this mais en ordre décroissant

Legacy Collections

Legacy collection

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

Vector, Stack, Hashtable et Enumeration sont des anciennes classes

- qui ne doivent plus utilisé à part pour discuter avec du code legacy
- Les 4 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

Exemple

Récupérer l'ensemble des interfaces réseaux

```
Enumeration<NetworkInterface> enumeration =
    NetworkInterface.getNetworkInterfaces();

List<NetworkInterface> interfaces =
    Collections.list(enumeration); // hop on triche

for(NetworkInterface networkInterfaces: interfaces) {
  System.out.println(networkInterfaces);
}
```

Implanter sa propre collection

Abstract helper

Si l'on veut créer sa propre collection pour par exemple implanter une vue, il existe déjà des classes abstraites que l'on peut redéfinir

- AbstractCollection,
- AbstractSet
- AbstractQueue
- AbstractMap
- AbstractList / AbstractSequentialList

  Pour les listes AbstractSequentialList hérite de AbstractList ce qui est une erreur de design !

Abstract helper

Les classes abstraites

- Demande d'implanter une ou deux méthodes abstraites
- Fournissent une implantation des autres méthodes non optionelles en utilisant l'implantation des méthodes abstraites

Comme les méthodes optionnels ne sont pas implanter, la collection est par défaut non mutable

pour avoir une version mutable il faut aussi redéfinir les méthodes mutables

Methodes à implanter

Pour

- AbstractCollection/AbstractSet
  - size() et iterator()
- AbstractMap
  - entrySet()
    - Attention get sera en O(n) !
- AbstractList
  - size() et get(int)
- AbstractSequentialList
  - size() et listIterator(int)

Exemple

Renvoie une liste dont chaque élement est calculé par la fonction de projection (mapper) à partir de l'élement dans la liste initiale

```
public static <T,U> List<U> map(List<T> list,
                                Function<? super T, ? extends U> mapper) {
  class MappedList extends List<U>
                   implements RandomAccess {
    @Override
    public int size() {
      return list.size();
    }
    @Override
    public U get(int index) {
      return mapper.apply(list.get(index));
    }
  }
  return new MappedList();
}
```

Implantations des collections

ArrayList

Tableau dynamique (qui s'agrandit tout seul)

facteur à 1.5 par défaut

Complexité

- Insertion à la fin en 0(1) amortie
- Insertion en début ou au millieu en O(n)
- Accès au index-ième élement en O(1)

Parcours:

- Avec un itérateur O(n)
- Avec un index, O(n) mais plus rapide

LinkedList

Liste doublement chaînée

Complexité

- Insertion au début ou fin en O(1)
- Insertion au millieu
  - avec add O(n)
  - avec listIterator.add 0(1)
- Accès au index-ième élement en O(n)

Parcours:

- Avec un itérateur O(n)
- Avec un index, $O(n^2)$!!!

ArrayDeque

Buffer circulaire dynamique à deux pointeurs

- Taille en puissance de 2

Complexité

- Insertion/suppression au début/à la fin en 0(1) amortie

Parcours:

- Avec un itérateur O(n)

HashMap

Table de hachage dynamique avec liste chainée ou arbre pour gérer les collisions

- Agrandissement si plein a plus de 75%
- taille en puissance de 2
- Protection DDOS

Complexité

- Insertion/suppression/accès 0(1) amortie

Parcours:

- Avec un itérateur O(n)

    sur le résultat de keySet(), entrySet() ou values()

WeakHashMap

Table de hachage dynamique

- Les clés sont stockées avec des références faibles (pas compté par le GC)
  - Bug: marche pas si la clé est référencé par la valeur (pas d'ephemeron :( )

peut être utilisé comme un cache qui se vide automatiquement si il n'y a plus assez de mémoire

- Attention, à ne pas mettre trop d'élément !
- Attention, le cache doit se recréer incrémentalement !

IdentityHashMap

Table de hachage dynamique fermée

- Utilise == et System.identityHashCode à la place de equals et hashCode
- Gestion des collisions l'intérieur de la table
- Utilise un seul tableau avec les clés et les valeurs à des index consécutif

Même complexité que HashMap

- Plus lent si beaucoup d'élement
- Consomme moins de mémoire si peu d'élement

TreeMap

Arbre rouge/noir, maintient l'ensemble des clés triés

- Auto-équilibrage donc hauteur en ln(n)

Complexité

- Insertion/suppression/accès 0(ln(n)) amortie

Parcours:

- Avec un itérateur O(n)
    - sur le résultat de keySet(), entrySet() ou values()

EnumSet

Set spécialisée si les élements viennent tous du même enum

- ordinal() est une fonction de hash parfait
- Deux implantations

  Si enum.values().length <=64, utilise 1 long (regular)
  sinon utilise un tableau de long (jumbo)

Même complexité que HashSet mais représentation en mémoire très compacte

EnumMap

Map spécialisée si les clés viennent tous du même enum, utilise un tableau de taille fixe.

- ordinal() est une fonction de hash parfait

Même complexité que HashMap mais représentation en mémoire très compacte