Programmation orientée objet. Cours 6

Marie-Pierre Béal et Carine Pivoteau
BUT 1

Tableaux. Collections. Dictionnaires.

Tableaux

Les tableaux sont des objets spéciaux. L'accès se fait par référence et la création par `new`.

Un tableau se déclare, se construit, et s'utilise comme un objet mais ne possède pas de méthodes.

```
int[] tab; // déclaration
int n = 10;
tab = new int[n] ; // construction
//utilisation
for (int i = 0; i < tab.length; i++){
  tab[i] = i;
}
for (int x : tab){
  System.out.println(x);
}
```

Exemple de tableaux de dimension 2

```
int n = 3;
int p = 5;
var matrix = new double[n][p]; // n lignes, p colonnes
matrix[1][3] = 7; // ligne 1, colonne 3
for (int i = 0; i < matrix.length; i++) {
  for (int j = 0; j < matrix[0].length; j++) {
    System.out.print(matrix[i][j] + " ");
  }
  System.out.print("\n");
}
```

```
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 7.0 0.0
0.0 0.0 0.0 0.0 0.0
```

Tableaux - suite

On peut fusionner déclaration et construction et faire une initialisation énumérative :

```
var jours = new String[] { "Lundi", "Mardi", "Mercredi",
                "Jeudi", "Vendredi", "Samedi", "Dimanche" }
```

La taille d'un tableau est accessible grâce au champs public `length` (c'est totalement contraire au principe d'encapsulation...).

```
System.out.println(jours.length); // 7
```

Attention, comme en Python, les indices des cases vont de 0 à la taille du tableau exclue. Chacune des instructions suivantes provoque une exception (de la classe `ArrayIndexOutOfBoundsException`) :

```
tab[tab.length] = 100; // case inexistante
tab[-1] = 100; // correction : tab[tab.length-1]
```

Autres structures de données : vue d'ensemble

- Deux packages :
  - `java.util` : collections, maps, itérateurs, ... mais aussi `Random`, `Objects`, `Scanner`, ...
  - `java.util.concurrent` : pas cette année.
- Deux interfaces principales :
  - `Collection<E>` pour les collections d'objets de type `E`, avec ou sans répétitions, ordonnés ou non (listes et ensembles).
  - `Map<K,V>` pour les dictionnaires (ou tables d'association), c'est-à-dire des ensembles de couples (clé, valeur), où la clé et la valeur sont de types paramétrés respectifs K et V.
- De nombreuses classes qui les implémentent, comme `ArrayList`, par exemple.

`https://docs.oracle.com/javase/tutorial/collections/index.html`

Utilitaires

En plus des classes implémentant les interfaces `Collection` et `Map`, on a trois classes d'utilitaires (contenant des méthodes statiques qui prennent des collections en paramètre) :

- `Collections` : avec des algorithmes généraux tels que :
  - copie (mutable ou non) d'une collection,
  - recherche dichotomique dans une collection triée,
  - tri (pour les collections non triées d'objets comparables),
  - calcul de min, max, fréquence ...
- `Arrays` : algorithmes spécialisés pour les tableaux.
- `Objects` : algorithmes spécialisés pour les objets.

▷ La plupart de ces algorithmes sont polymorphes : ils s'appliquent aux interfaces et pas directement aux classes qui les implémentent.

▷ Les méthodes fournies implémentent en général les meilleurs algorithmes existants (les plus efficaces).

Principales méthodes des collections

`Collection<E>` est une interface générique, c'est à dire qu'une collection peut contenir des objets de n'importe quelle classe.

Le type des éléments est un type paramétré `E`. On doit le remplacer par une classe lorsque l'on utilise une collection.

Les opérations principales sur une collection sont :

- `boolean add(E e)` : pour ajouter un élément.
- `boolean remove(Object o)` : pour enlever un élément,
- `boolean contains(Object o)` : test d'appartenance,
- `int size()` : pour obtenir le nombre d'éléments,
- `boolean isEmpty()` : pour tester si l'ensemble est vide.

Attention, si la collection n'est pas mutable, `add` et `remove` peuvent ne pas être implémentées (`UnsupportedOperationException`).

Contrat Objet

Tous les algorithmes sur les structures données présupposent que les objets stockés dans une collection respectent un certain contrat ; il sont sensés implémenter correctement les méthodes suivantes :

- `equals`
- `hashCode`
- `toString`
- (éventuellement) `compareTo`

Contrat Objet

Autrement, le fonctionnement de ces algorithmes n'est pas garanti :

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

```
var array = new StupidInt[] { new StupidInt(1), new StupidInt(2) };
Arrays.sort(array);
// ClassCastException: cannot be cast to java.lang.Comparable
var set = new HashSet<StupidInt>();
set.add(new StupidInt(1));
set.contains(new StupidInt(1)); // renvoie false
```

Interface `List`

- `List<E>` spécifie une séquence, c'est à dire une collection ordonnée (mais pas triée) avec possibilité de répétitions.

  En plus de celles permettant d'implémenter `Collection<E>`, elle déclare notamment les méthodes suivantes :

  - `int indexOf(Object o)` : retourne la position de `o` .
  - `E get(int index)` : retourne l'objet à la position `index`.
  - `E set(int index, E element)` : remplace l'élément en position `index`, et retourne l'élément qui y était précédemment.
  - `E getFirst()` : retourne l'objet à la position 0, lève `NoSuchElementException` si la liste est vide.
  - `E getLast()` : retourne l'objet à la dernière position, lève `NoSuchElementException` si la liste est vide.
  - `List<E> subList(int fromIndex, int toIndex)` : renvoie une vue des éléments de la liste compris entre les deux indices.

Implémentation d'une `List`

Deux implémentations principales :

- `ArrayList<E>` (recommandée), qui utilise un tableau dont on augmente la capacité maximale si nécessaire.
- `LinkedList<E>` qui est une liste doublement chaînée (les données ne sont pas stockées de façon consécutive en mémoire).
- les autres sont soit obsolètes, soit des cas particuliers.

Le choix de l'implémentation dépend de l'efficacité recherchée : par exemple, pour une liste de taille n :

	`ArrayList`	`LinkedList`
accès indicé	$O(1)$	$O(n)$
insertion à une position donnée	$O(n)$	$O(1)$ ou $O(n)$
insertion en tête / fin	$O(n)$ / $O(1)$	$O(1)$ / $O(1)$
suppression d'un élément donné	$O(n)$	$O(n)$
suppression à une position donnée	$O(n)$	$O(1)$ ou $O(n)$
...	...	...

Les files et les piles : interfaces Queue et Deque

Comme les listes, ce sont des collections ordonnées avec possibilité de répétitions, mais elles ne fournissent pas d'accès indicé.

- L'interface `Queue<E>` permet notamment de spécifier une file d'attente (FIFO : first-in-first-out) :
  - `boolean add(E e)` : insertion en fin.
  - `E remove()` : suppression en tête.
  - `E element()` : renvoie la tête de la file sans la supprimer.

  ▷ Parmi les implémentations : `ArrayDeque`, `LinkedList`, `PriorityQueue` (attention, la notion d'ordre est différente).
- L'interface `Deque<E>` permet notamment de spécifier une pile (LIFO : last-in-first-out) :
  - `addLast`, `removeFirst` et `getFirst` : comportement d'une file.
  - `addFirst`, `removeFirst` et `getFirst` : comportement d'une pile.

  ▷ Parmi les implémentations : `ArrayDeque`, `LinkedList`.

Interface Set

Pour les ensembles, on a deux interfaces différentes :

- `Set<E>` spécifie les ensembles non ordonnés sans répétions. Lorsque que l'on essaie d'ajouter un élément déjà présent avec `add`, l'ensemble n'est pas modifié et la méthode renvoie `false`.

  ▷ Implémentation recommandée : `HashSet` pour avoir les opérations de base en $\mathcal{O}(1)$ en moyenne.

- `SortedSet<E>` sous-interface de `Set` pour les ensembles triés. La classe `E` doit implémenter `Comparable<E>`. Un `SortedSet` possède, en plus des méthodes de `Set`, des méthodes spécifiques :
  - `E first()` retourne le premier objet.
  - `E last()` retourne le dernier objet.

  ▷ Implémentation recommandée : `TreeSet` pour avoir les opérations de base en $\mathcal{O}(\log n)$ en moyenne.

Exemple

Programme qui détecte les répétitions dans les chaînes de caractères entrées sur la ligne de commande.

```
import java.util.HashSet;
public class SetTest {
  public static void main(String[] args) {
    var s = new HashSet<String>();
    for (var w : args){
      if (!s.add(w)){
        System.out.println("Deja vu : " + w);
      }
    }
    System.out.println(s.size() + " distincts : " + s);
  }
}
```

```
$ java SetTest a b c a b d
Deja vu : a
Deja vu : b
4 distincts : [d, a, c, b]
```

Opérations ensemblistes sur les collections

Ces méthodes sont des méthodes de l'interface `Collection<E>`.

- Tester l'inclusion : `boolean containsAll(Collection<?> c)`
- Union : `boolean addAll(Collection<? extends E> c)`
- Différence : `boolean removeAll(Collection<?> c)`
- Intersection : `boolean retainAll(Collection<?> c)`

Les trois dernières méthodes renvoient `true` si elles ont bel et bien modifié la collection.

Ces méthodes peuvent être écrites en utilisant uniquement des méthodes fournies par l'interface `Collection`, mais les méthodes spécifiques fournies par les classes qui l'implémentent sont souvent plus efficaces.

Les méthodes de la classe `Collections` : exemples

```
public static void main(String[] args) {
  var birds = new ArrayList<Bird>();
  Collections.addAll(birds,
      new Bird("moineau", 10), new Bird("moineau", 10),
      new Bird("colibri", 5),  new Bird("albatros", 90),
      new Bird("galinette", 25)); // ajouts multiples
  var pie = new Bird("pie", 20);
  birds.addAll(Collections.nCopies(3, pie)); // répétitions
  System.out.println(birds);
  Collections.shuffle(birds); // mélange
  System.out.println(birds);
  System.out.println(Collections.max(birds)); // maximum (selon quel ordre ?)
  System.out.println(Collections.frequency(birds, pie)); // fréquence
  Collections.sort(birds); // tri
  System.out.println(birds);
  Collections.reverse(birds); // retournement
  System.out.println(birds);
}
```

Les méthodes de la classe `Collections` : exemples

```
[moineau (10), moineau (10), colibri (5), albatros (90),
galinette (25), pie (20), pie (20), pie (20)]
[galinette (25), colibri (5), moineau (10), albatros (90),
moineau (10), pie (20), pie (20), pie (20)]
albatros (90)
3
[colibri (5), moineau (10), moineau (10), pie (20), pie (20),
pie (20), galinette (25), albatros (90)]
1
[albatros (90), galinette (25), pie (20), pie (20), pie (20),
moineau (10), moineau (10), colibri (5)]
```

Interface Comparable<E>

Un ensemble trié SortedSet<E> utilise l'ordre naturel des ses éléments de la classe E qui est spécifié en implantant l'interface Comparable :

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

Les classes comme String, Integer, Double, etc, implémentent toutes l'interface Comparable.

Pour vos propres classes, il suffit d'implémenter la méthode compareTo en respectant les règle suivantes :

- a.compareTo(b) < 0 $\iff$ a < b
- a.compareTo(b) > 0 $\iff$ a > b
- a.compareTo(b) == 0 $\iff$ a = b
- a.compareTo(b) == 0 $\iff$ a.equals(b) == true

Exemple : Bird implements Comparable<Bird>

Comparaison sur l'envergure :

```
public record Bird(String name, int wingspan) implements Comparable<Bird> {
  @Override
  public int compareTo(Bird bird) {
    return Integer.compare(wingspan, bird.wingspan);
  }
}
```

Plus sophistiqué : on compare d'abord l'envergure et en cas d'égalité, on utilise l'ordre lexicographique sur les noms.

```
public record Bird(String name, int wingspan) implements Comparable<Bird> {
  @Override
  public int compareTo(Bird bird) {
    int comparison = Integer.compare(wingspan, bird.wingspan);
    if (comparison != 0) {
      return comparison;
    }
    return name.compareTo(bird.name);
  }
}
```

Et pour les tableaux ?

La classe `Arrays` (package `java.util`) fournit des méthodes utilitaires (helpers) pour effectuer des opérations "haut niveau" sur les tableaux.

Attention, toutes ces méthodes sont statiques.

```
var numbers = new int[] { 4, 11, 24, 7, 18, 91, 6, 2 ,55 };
System.out.println(Arrays.toString(numbers));
// [4, 11, 24, 7, 18, 91, 6, 2, 55]

Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers));
// [2, 4, 6, 7, 11, 18, 24, 55, 91]

int[] smallNumbers = Arrays.copyOf(numbers, 4);
System.out.println(Arrays.toString(smallNumbers));
// [2, 4, 6, 7]

System.out.println(Arrays.binarySearch(smallNumbers,5)); // -3
```

Les dictionnaires ou tables : l'interface `Map`

L'interface `Map<K,V>` représente une table d'association : un ensemble de couples (clé de type `K`, valeur de type `V`). Les clés ne peuvent pas être dupliquées et au plus une valeur est associée à une clé.

- `V put(K key, V value)`
  Insère l'association `(key, value)` dans la table et retourne la valeur précédemment associée à la clé ou bien `null` .
- `boolean containsKey(Object key)`
  Retourne vrai s'il y a une valeur associée à cette clé.
- `V get(Object key)`
  Retourne la valeur associée à la clé dans la table, ou `null` si c'est la valeur associée ou si `key` n'est pas une clé de la table.
- `V remove(Object key)`
  Supprime l'association de clé `key`. Retourne la valeur associée précédemment. Retourne `null` si `key` n'est pas une clé de la table (ou si `null` est la valeur associée).

L'interface Map - suite

L'intereface Map fournit aussi des méthodes de manipulation plus sophistiquées comme :

- `V putIfAbsent(K key, V value)`
- `V getOrDefault(Object key, V defaultValue)`
- `boolean replace(K key, V oldValue, V newValue)`

et des méthodes d'accès aux données :

- `Set<K> keySet()`
- `Collection<V> values()`
- `Set<Map.Entry<K,V>> entrySet()`

Vous pouvez consulter la javadoc pour plus de détails.

La sous-interface `SortedMap <K,V>` spécifie les tables d'association dont l'ensemble des clés est ordonné.

Implémentation des Map

Pour les tables avec clés non ordonnées :

- `HashMap<K,V>` (recommandée) est l'implémentation la plus efficace pour les ajouts/suppressions.
- `LinkedHashMap<K,V>` conserve les clés dans l'ordre d'insertion.

Pour les tables avec clés ordonnées :

- `TreeMap<K,V>` implémente `SortedMap<K,V>`.

`TreeMap` utilise des arbres rouge-noir qui stockent les clés (ordonnées) de telle sorte que ces opérations ont une complexité $O(\log n)$:

- `boolean containsKey(Object key)`
- `V get(Object key)`
- `V put(K key, V value)`
- `V remove(Object key)`

Les classes `HashSet` et `HashMap` reposent sur l'utilisation de `hashCode`

- C'est pour assurer le bon fonctionnement des tables de hachage que l'on doit implémenter `hashCode`.
- Leurs performances – insertion, suppression, accès en $O(1)$ (complexité amortie) – reposent sur l'implémentation de `hashCode`.
- La méthode `hashCode` doit être consistante (plusieurs appels donnent le même résultat) ;
- La méthode `hashCode` doit donner la même valeur pour deux objets égaux par `equals`.

Comment garantir tout cela : pour redéfinir `hashCode`, on utilise la méthode statique `hash(Object... values)` de la classe `Objects` sur tous les champs dont l'égalité est testée par `equals`. S'il n'y a qu'un seul champ, on utilise `hashCode(Object o)` de `Objects` à la place.

Exemple : un magasin de vêtements

Le magasin de vêtements contient une map qui associe à un vêtement (représenté ici sous forme de `String`) son nombre d'exemplaires dans le magasin.

```
public class Shop {
  private final HashMap<String,Integer> map;

  public Shop () {
    map = new HashMap<>();
  }
  public void add(String clothing){
    Objects.requireNonNull(clothing);
    map.put(clothing, map.getOrDefault(clothing, 0) + 1);
  }
  @Override
  public String toString(){
    return map.toString();
  }
```

Exemple : un magasin de vêtements

```
  public static void main(String[] args){
    var v1 = "shirt";
    var v2 = "polo";
    var shop  =  new Shop();
    shop.add(v1);
    shop.add(v1);
    shop.add(v2);
    System.out.println(shop); // {shirt=2, polo=1}
  }
}
```

Bonus : le type enum

Le type enum est un type de données spécial qui permet d'assigner à une variable une valeur choisie dans un ensemble de constantes prédéfinies. On mettra la classe dans un fichier de même nom.

```
public enum Meal { // un enum se défit comme une classe
  BREAKFAST, LUNCH, TEATIME, DINNER // constantes, donc en majuscules
}
```

```
public enum WeekDay {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Compléments :

https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Exemple avec un `switch`

```
public class Day {
  private final WeekDay weekDay;

  public Day(WeekDay weekDay) {
    this.weekDay = weekDay;
  }

  @Override
  public String toString() {
    // particulièrement adapté pour les switch
    return switch (weekDay) {
      case MONDAY -> "lundi";
      case TUESDAY -> "mardi";
      ...
      case SUNDAY -> "dimanche";
    };
  }
}
```

Exemple d'enum – dans le main

```
public class Test {
  public static void main(String[] args) {
    System.out.println(Meal.LUNCH);  // LUNCH
    var day = new Day(WeekDay.FRIDAY);
    System.out.println(day); // * vendredi *
    for (var weekDay : WeekDay.values() ) {
      System.out.println(weekDay.ordinal() + " : " + weekDay);
    }
    // 0 : MONDAY
    // 1 : TUESDAY
    // ...
    // 6 : SUNDAY
  }
}
```