

# Collections en Java (cours 3)

Virginia Aponte

CNAM-Paris

25 mars 2019

# 1. Structure « Tableau associatif »

# Tableaux associatifs

## Tableau associatif

Table permettant :

- stocker des **valeurs**  $v$  (de type  $V$ ),
- organisées/récupérables via leur **clé de recherche**  $k$  (de type  $K$ ).

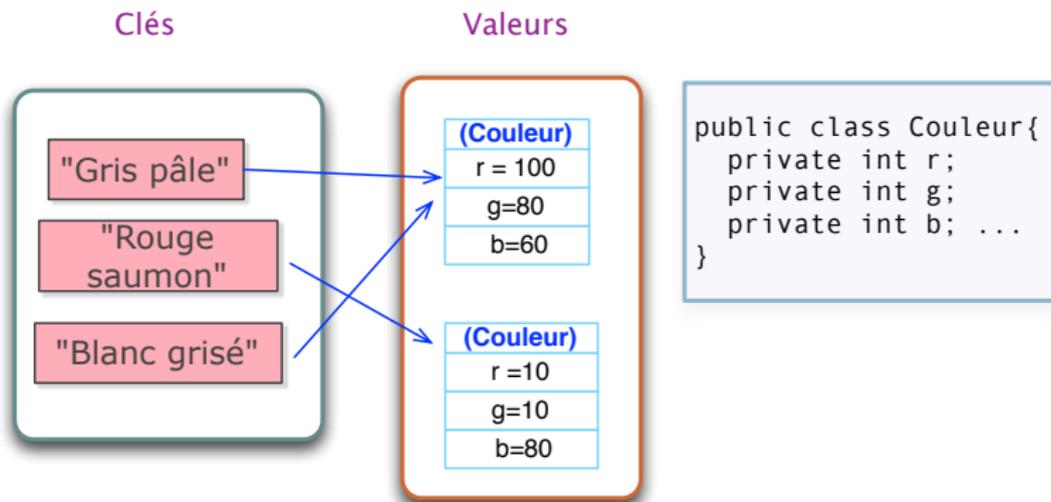
Fonctionne comme :

- un **ensemble** d'associations  $[k] \mapsto [v]$
- entre clés  $k$  et valeurs  $v$ ,
- $v$  est la « valeur » associée à la clé  $k$  dans la table.

Chaque **association**  $(k, v)$  entre une clé  $k$  et une valeur  $v$  peut être vue comme une *rangée* de la table.

# Tableau « couleurs »

Dans ce tableau : l'encodage d'une couleur en niveau de rouge, vert et bleu (type `Couleur`) correspond à son nom courant (type `String`).



Cette table contient 3 associations

## Tableaux associatifs (2)

- à chaque **clé**  $k$  correspond **une unique valeur**  $v$ .
- clés peuvent être de n'importe quel type  $K$ ,
- les clés **doivent** former un ensemble (pas de clé en double).
- pour récupérer la valeur associée à la clé  $k$  dans  $t$  :

`t.get(k)`  $\Rightarrow$  *retourne la valeur  $v$  associé à  $k$  dans  $t$*

- `t.put(k, v)`  $\Rightarrow$  *modifie avec  $v$  la valeur associée à  $k$  dans  $t$*

### Tableau associatif « clés $\mapsto$ valeurs »

Sorte de « *tableau généralisé* » où l'on trouve des clés de type quelconque en lieu et place d'indices entiers. A chaque clé  $k$  correspond une unique valeur  $v$  dans le tableau.

## 2. « Tableaux associatifs » en Java :

**Interface** `Map<K, V>`

# « Map » = table d'associations

## Map

*Map* (en anglais)  $\approx$  *association ou fonction*.

En Java  $\Rightarrow$  interface `Map<K, V>` où :

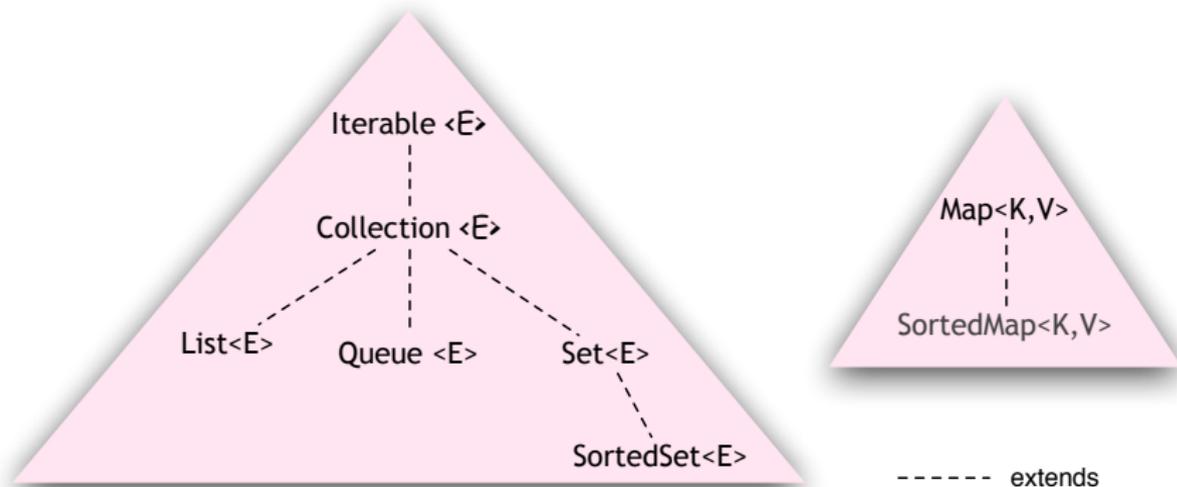
- K : type des clés
- V : type des valeurs

Exemples :

- **annuaire** (« contact  $\mapsto$  téléphone »)  $\Rightarrow$  `Map<String, Integer>`
- **banque** (« numéro  $\mapsto$  compte »)  $\Rightarrow$  `Map<Integer, Compte>`
- **dictionnaire** (« terme  $\mapsto$  définition »)  $\Rightarrow$  `Map<String, String>`

# Rappel : hiérarchie d'interfaces collections

(Extrait) :



<E> : type générique E pour les composantes

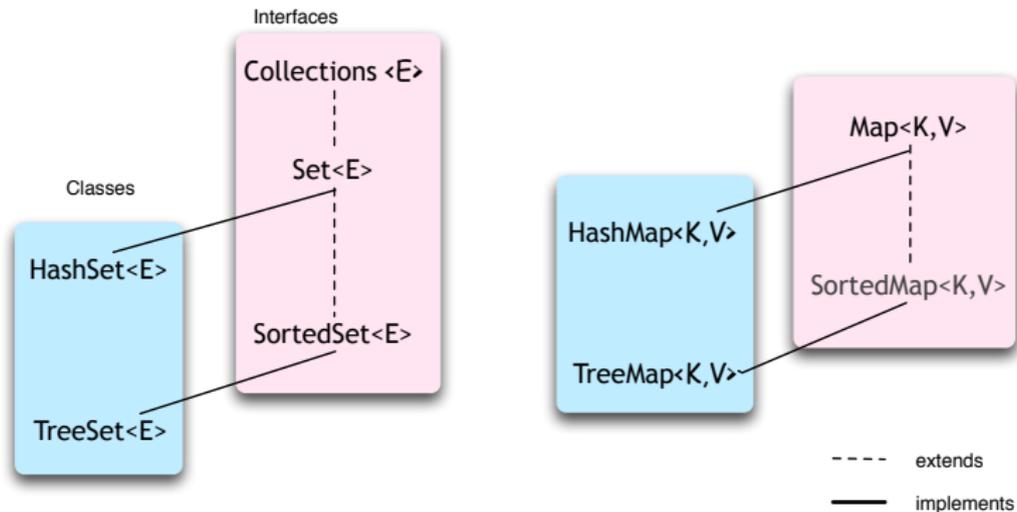
# Mappes triées et non triées

Deux sortes de mappes :

- non triées : `Map<K, V>`
- triées : `SortedMap<K, V>`
  - ordre établi sur l'ensemble de clés K.
  - permet d'organiser/parcourir données selon l'ordre des clés ;
  - Exemples :
    - annuaire trié sur l'ensemble des noms de contacts ;
    - dictionnaire trié par ordre alphabétique des termes ;

# Interfaces + classes $\Rightarrow$ ensembles + mappes

(Extrait)



<E> : type générique E pour les composantes

# Extrait de l'interface Map

```
public interface Map<K,V> {  
    boolean isEmpty();  
    V get(K key);  
    void put(K key, V value);  
    boolean containsKey(K key);  
    int size();    ...  
}
```

- `isEmpty()` : tester si la table est vide ;
- `get(k)` : obtenir valeur associée à `k` ;
- `containsKey(k)` : tester si la clé `k` est déjà dans la table ;
- `put(k, v)` : ajoute une association `(k, v)`. Si `k` existe dans la table, change sa valeur associée par `v`.

**Attention** : Map n'est pas une Collection  $\Rightarrow$  pas de *for each*, pas d'itérateur !

# Map<K, V> vs. List<V> en 1 transparent

Si on souhaite ajouter/récupérer une valeur  $v$  dans une ...

- List<V> tab :

- se fait via la position  $i$  de  $v$  : `tab.get(i)`
- dans notre application la position de nos données est aussi une donnée ?
- Ex : no. d'arrivée d'un patient (file attente)  
⇒ `ArrayList<Patient>`.

- Map<K, V>tab :

- se fait via un identifiant ou clé  $k$  pour  $v$  : `tab.get(k)`
- Ex : nom de contact dans annuaire ⇒ `Map<String, Contact>`.

List et Map : deux sortes de « tables ». Pour ranger/recupérer, l'une utilise la position, l'autre une clé. Map est plus générale.

# Implantations de « Map »

Deux implantations principales :

- `HashMap<K, V>` :
  - basée sur des *tables de hash* ;
  - *fonction de hash* appliquée sur les clés  $k$  ;
  - ordre « non déterminé » pour les clés ;
  - la plus rapide.
- `TreeMap<K, V>` :
  - basée sur des *arbres de recherche* ;
  - clés  $K$  ordonnées (on doit pouvoir établir un ordre entre elles)

### 3. La classe HashMap<K,V>

# Exemple : application « Banque »

- 3 classes : `Compte`, `Banque` et `Titulaire`
- 1 titulaire → plusieurs comptes, certains *partagés* avec d'autres titulaires (comptes joints);
- comptes *pas directement* accessibles :
  - gardés en interne par la banque;
  - seule la banque peut les créer ...
  - ... puis, elle les ajoute « dans » ses titulaires;
- opérations sur les comptes :
  - effectués uniquement par la banque;
  - aucune opération *publique* **ne renvoie/reçoit un compte**;
  - utilisent `no. compte` pour identifier le compte à modifier!

---

```
Banque bnp = new Banque("BNP"); ...  
bnp.depot(10003012579067, 150); // sur no. compte
```

---

# Application Banque (2) : les données internes

## Contrainte sur les opérations

on donne no. de compte  $\Rightarrow$  effectuer opération sur le compte associé.

- Implantation simplifié : numéros de compte entiers (pas réaliste) ;
- Les données `Comptes de Banque` :
  - **doivent** être organisées/recupérables par leur no. de compte
  - $\Rightarrow$  structure `Map` : (clé=`No.Compte`  $\mapsto$  valeur= objet `Compte`).
    - `Map<Integer, Compte>`
    - `Integer` redéfinit `hashCode()`, `equals()`  $\Rightarrow$  pas à re faire !
- Les données `Comptes de Titulaire` : mêmes contraintes  $\Rightarrow$  même structure `Map`.

`HashMap` pour représenter les comptes d'une banque + d'un titulaire.

# Attributs de la classe Banque

Comptes de la banque (variable tous)  $\Rightarrow$  Map<Integer, Compte>

---

```
public class Banque {
    private String nom;           // nom de la banque
    private int numComptes = 1;  // Prochain numero compte

    // Tous les comptes de la banque
    private HashMap<Integer,Compte> tous =
        new HashMap<Integer,Compte>();

    /* Constructeur */
    public Banque(String n){nom =n;}
}
```

---

# Les méthodes de la classe Banque

```
/* Creation compte, ajout pour banque + titulaires */
public void creerCompte(Titulaire [] tit, double init);
/* Ce numero de compte existe dans la banque? */
public boolean isNumCompte(int n);
/* Effectuer un depot pour un numero de compte */
public boolean depot(int num, double m);
/* Virement de banque a banque */
public boolean virerVers(double m, int numS, int numD, Banque b);
/* Virement interne */
public boolean virerVers(double m, int numS, int numD);
/* Methode interne pour recuper un compte */
private Compte getCompteDeNum(int n)
/* Bilan (solde total) des comptes de la banque */
public double getSoldeTous();
```

# Méthode (publique) de création d'un compte

- paramètres : liste de titulaires `tit`; montant initial `init`
- créer un compte de numéro `numComptes`;
- ajout dans mappe `tous`
- ajout dans comptes de chaque titulaire de `tit`

---

```
public void creerCompte(Titulaire [] tit, double init){  
    int numc = numComptes;  
    Compte c = new Compte(numc,init);  
    tous.put(numc, c);    // ajout dans comptes banque  
    for (Titulaire t : tit) {  
        t.ajouterDans(c); // ajout dans chaque titulaire  
    }  
    numComptes++; // increment numero des comptes  
}
```

---

# Tester si un numéro de compte existe dans la banque

---

```
/** Teste si
 * @param n est le numero d'un compte dans la banque
 * @return true dans ce cas.
 */
public boolean isNumCompte(int n) {
    return tous.containsKey(n);
}
```

---

# Obtenir le compte de numéro n (méthode interne)

Cette méthode renvoie un Compte  $\Rightarrow$  elle est privée.

---

```
/** Obtenir le compte de numero
 * @param n
 * @return ce compte s'il existe, null sinon
 */
private Compte getCompteDeNum(int n) {
    return tous.get(new Integer(n));
}
```

---

Méthode triviale : nous la gardons pour dépendre moins de *l'implantation interne* du registre des comptes.

# Dépôt sur compte de numéro n

- Tester si le numéro de compte existe,
- si oui, le récupérer,
- lui appliquer sa méthode depot (de la classe Compte);

---

```
/* Depot du montant
 * @param m dans compte de numero @param num
 * @return true si depot effectue
 *         false si compte inexistant
 */
public boolean depot(int num, double m) {
    if (!isNumCompte(num))
        return false;
    Compte c = getCompteDeNum(num);
    c.depot(m); // appel methode depot de Compte
    return true;
}
```

# Virement entre banques

- Tester si le numéro de compte existe + le récupérer,
- lui appliquer sa méthode depot;

---

```
/** Virement du montant m depuis compte numero
 * @param numS, vers le compte numero
 * @param numD de la banque @param b
 * @return true si virement effectue
 */
public boolean virerVers(double m, int numS,
                        int numD, Banque b) {
    if (!isNumCompte(numS)) return false;
    Compte c = getCompteDeNum(numS);
    c.retrait(m);
    return b.depot(numD, m);
}
```

---

# Virement interne à la banque

Utilise méthode précédente avec banque externe  $\mapsto$  `this`

```
/** Virement de montant m a partir du compte numero
 * @param numS, vers le compte numero
 * @param numD de la banque courante
 * @return true si virement effectue
 */
public boolean virerVers(double m, int numS, int numD) {
    if (!isNumCompte(numD) || !isNumCompte(numS))
        return false;
    else
        return virerVers(m, numS, numD, this); // b -> this
}
```

## 4. Parcours d'un tableau associatif

# Les associations d'un tableau associatif

Un tableau associatif est formé d'*associations* entre clés et valeurs :

## Tableau Associatif = Ensemble d'associations

- 1 association  $(k, v)$  = 1 paire (clé,valeur) de la table
- **Analogie** : 1 association  $(k, v) \approx$  1 rangée de la table.

Tableau d'associations est formé par l'ensemble  $(k_1, v_1), (k_2, v_2) \dots$  de toutes les associations (rangées) de la table.

**Attention** : la notion d'association est importante en Java.

# Pour parcourir une Map

Si `tab` est une table de type `Map<K, V>` :

- les opérations `tab.get(k)`, `tab.put(k, v)` permettent d'agir sur une seule association (rangée) de la table.

Comment effectuer des opérations sur *toute* la table : recherche d'une clé, recherche d'une valeur, etc ?

- Une mappe n'est pas une collection  $\Rightarrow$  pas de for-each, pas d'itérateur.
- Mais toute mappe contient des méthodes permettant d'obtenir « ses collections » internes, à savoir :
  - le `Set` qui contient toutes ses clés
  - la `Collection` qui contient toutes ses valeurs,
  - le `Set` formé de toutes ses associations.

# Méthodes pour obtenir les collections d'une table

Si `tab` est une table de type `Map<K, V>` :

- `tab.keySet()`  $\Rightarrow$  ensemble de clés de la table (`Set<K>`);
- `tab.values()`  $\Rightarrow$  collection de valeurs de la table (`Collection<V>`);
- `tab.entrySet()`  $\Rightarrow$   
ensemble d'associations (de type `Set<Map.Entry<K, V>`)
  - chaque association  $\Rightarrow$  appelée *entry* (rangée en anglais),
  - son type : `Map.Entry<K, V>` (celui d'une paire (k,v) )
  - chaque *entry* est un **objet** avec 2 méthodes :
    - `getKey()` (clé dans cette entrée)  $\Rightarrow$  K
    - `getValue()` (valeur dans cette entrée)  $\Rightarrow$  V

Pour un parcours : obtenir la collection appropriée et la parcourir avec les méthodes habituelles.

# Ensemble d'associations d'une mappe

Calculons l'ensemble d'associations de la mappe `tous` :

---

```
Set< Map.Entry<Integer, Compte>> s = tous.entrySet();
```

---

- `tous` : mappe de type `<Integer, Compte>`;
- `Map.Entry<Integer, Compte>` : type d'une association (couple de type `<Integer, Compte>`);
- `Set<Map.Entry<Integer, Compte>>` : ensemble d'associations (`Integer, Compte`) → ensemble de couples !
- `tous.entrySet()` ⇒ ensemble d'associations de la mappe `tous`

# Bilan des soldes dans la banque

Parcourir l'ensemble des associations en additionnant soldes des comptes :

---

```
public double bilan() {  
    double res= 0;  
    Set<Map.Entry<Integer, Compte>> s = tous.entrySet();  
    for (Map.Entry<Integer, Compte> a : s) {  
        res = res + a.getValue().getSolde();  
    }  
    return res;  
}
```

---

$s$  est une collection  $\Rightarrow$  boucle *for-each*

- pour chaque association  $a$  dans ensemble  $s$  :
  - $a.getValue().getSolde() \Rightarrow$  solde du compte dans  $a$ .

## Bilan des soldes dans la banque (2)

Modifier ce code en utilisant les autres « vues » collections sur la variable tous :

- `keyset()`
- `values()`

---

```
public double bilan() {  
    double res= 0;  
    ?? s = tous.??;  
    for ( ?? a : s) {  
        res = res + ??;  
    }  
    return res;  
}
```

---

# Trier les comptes par ordre des soldes (avec un Comparator et values ())

Dans le cours précédent nous avons écrit la classe :

```
ordreParSolde
```

qui implante `Comparator<Compte>` selon l'ordre des soldes courants.

---

```
public void afficheParSolde() {  
    // copie de values() dans une liste  
    ArrayList<Compte> t=new ArrayList<Compte>(tous.values());  
    // on les trie  
    Collections.sort(t,new ordreParSolde());  
    // et on les affiche  
    for (Compte c: t) { c.afficher(); }  
}
```

---

`tous.values()` ⇒ collection de valeurs de la Map `tous`.

## 5. Modifier une mappe pendant son parcours

# Éliminer tous les comptes avec solde nul

```
Set<Map.Entry<Integer,Compte>> s = tous.entrySet();
Iterator<Map.Entry<Integer,Compte>> its=s.iterator();
while (its.hasNext()) {
    if (its.next().getValue().getSolde()==0)
        its.remove();
}
```

Éliminer les comptes en parcourant la mappe  $\Rightarrow$  devons passer par une collection + son itérateur :

- `s=tous.entrySet()`  $\Rightarrow$  fabrique collection (set) d'associations ;
- `its = s.iterator()`  $\Rightarrow$  itérateur sur cette collection ;
- `its.next().getValue().getSolde()`  $\Rightarrow$  solde du compte dans composante « valeur » de prochaine association itérateur `its`.