Les collections (1)

DUT M3105 : Conception et programmation objet avancées

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

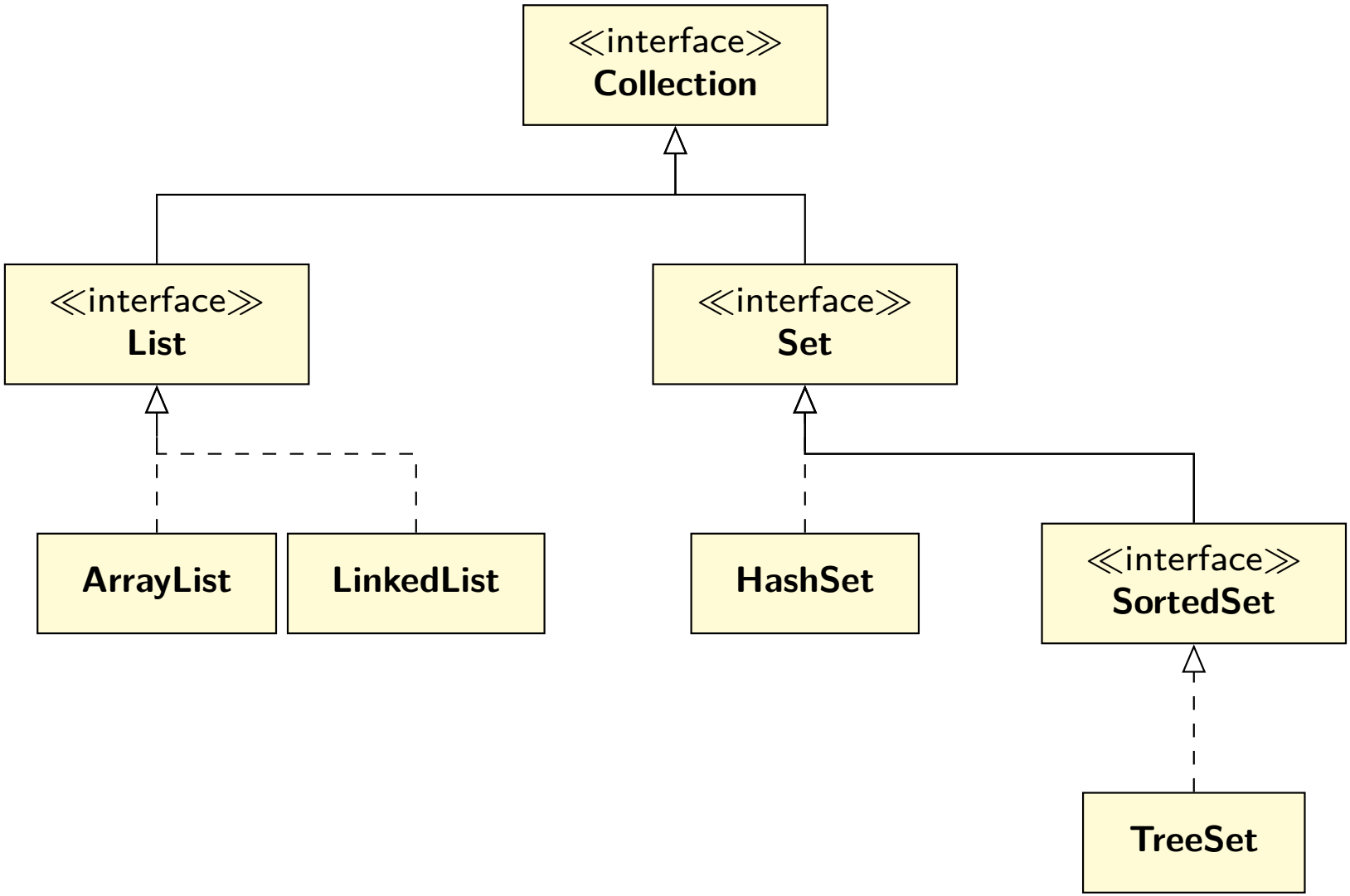
2018-2019

Pourquoi les collections

- une application manipule souvent un grand nombre d'objets d'un même type ;
- application commerciale :
  - les produits contenus dans un catalogue ;
  - les factures ;
  - les lignes de chaque facture...
- représentation d'une carte du métro :
  - les stations ;
  - les lignes qui passent par une station ;
  - ...

On désire avoir mieux que les tableaux...

Architecture des collections



Type paramétrique

- Au début, les collections contenaient des `Objects` sans spécification plus précise ;
- par exemple, `ArrayList` avait les deux méthodes :

```
1  public boolean add(Object o)
2  public Object get(int i)
```

- et pour manipuler une liste de produit on devait écrire :

```
1  ArrayList produits= new ArrayList();
2  produits.add(new Produit("ordinateur")); // ok
3  ...
4  Produit p= (Produit) produits.get(0); // cast obligatoi
```

- problème : le cast est ennuyeux à écrire ;
- plus grave : quand une méthode renvoie une liste, le type n'indique pas ce qu'elle contient ;
- risque de fausses manipulations.

Le polymorphisme paramétrique

- Idée : remplacer le `Objet` du cas précédent par un paramètre de classe T, qu'on pourra remplacer par un type particulier.
- Ce ou ces paramètres sont indiqués après le nom de la classe, entre <...>

On écrit :

```
ArrayList<Produit> produits= new ArrayList<>();
produits.add(new Produit("ordinateur")); // ok
...
Produit p= produits.get(0); // pas de cast !
```

Contenu des collections

- Les collections ne peuvent contenir que des *objets* ;
- donc **pas de types primitifs** (int, boolean, char...) ;
- à la place, on utilise les classes associées (Integer, Boolean, Character...) ;
- ou on va récupérer une bibliothèque comme GNU Trove, qui propose des classes comme `TIntHashSet`.

Les collections en 3 méthodes

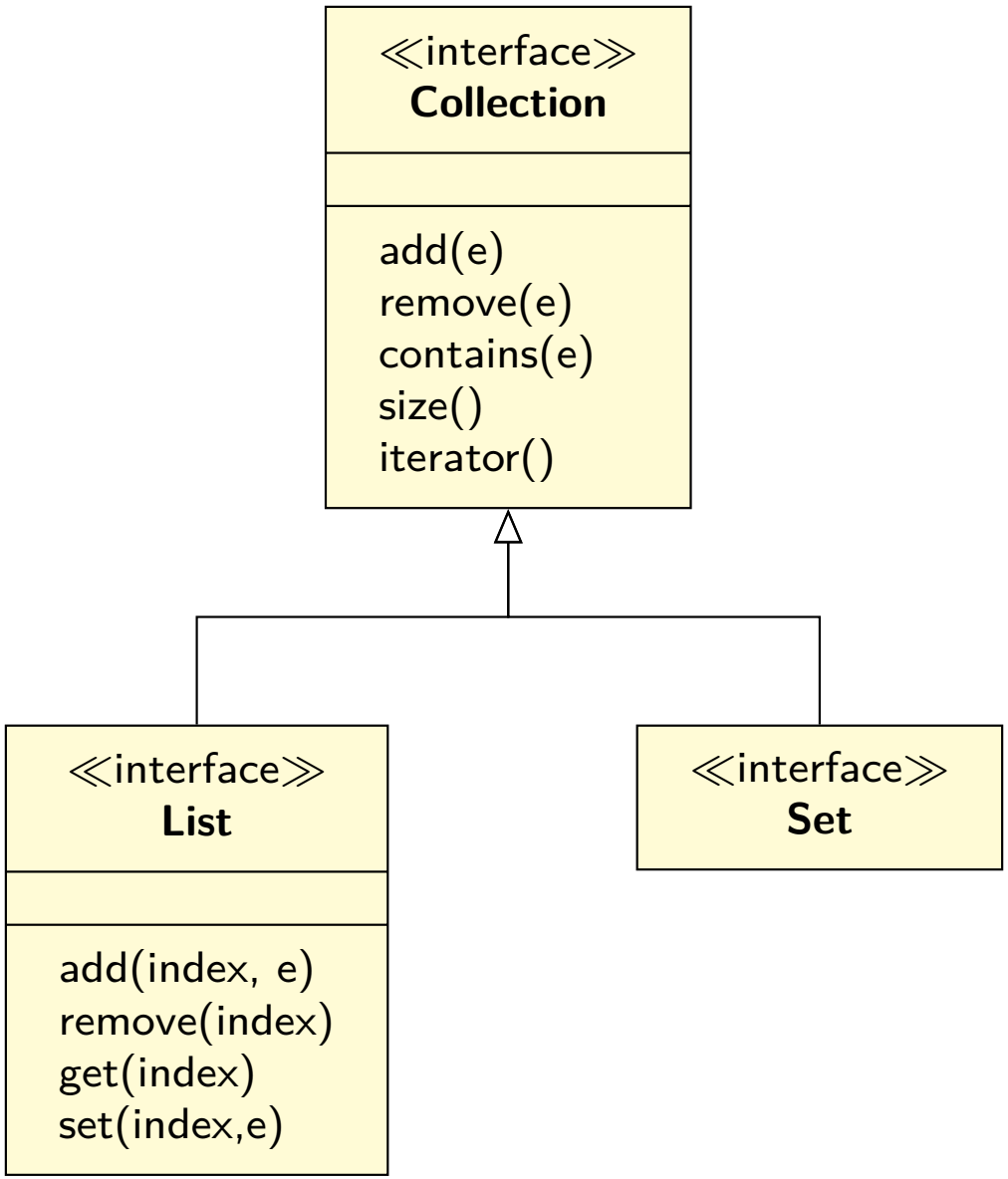
Soit une `ArrayList<String>` nommée `maListe`. Je peux :

- lui ajouter une valeur : `maListe.add("une valeur");`
- savoir si elle contient une valeur donnée :

```
1  if (maListe.contains("une␣valeur")) {...}
```

- savoir *combien* elle contient de valeurs : `maListe.size()`
- on aimerait bien la parcourir ;

Retour sur l'architecture



- *List* et *Set* ont toutes les méthodes de *Collection* ;
- *List* ajoute des méthodes spécifiques ;
- *Set* n'a que les méthodes de *Collection* — mais précise leur sens.

Les listes

Séquence d'éléments, repérés par une position (un index).

- un élément a une position ;
- un élément peut se retrouver plusieurs fois dans une liste ;
- la liste [a,b,c] est différente de la liste [a,c,b]

En pratique :

- On retrouve toutes les méthodes de `Collection` ;
- **plus** d'autres méthodes comme `get(int i)` ;
- le *sens* de certaines méthodes est spécifié ;

Implémentations des listes

- **ArrayList** : implémentée à l'aide d'un tableau sur-dimensionné ; quand il n'y a plus de place, l'objet crée un nouveau tableau, plus grand, et y recopie l'ancien ;
- **LinkedList** : implémentée à l'aide de listes chaînées.
- En théorie :
  - `ArrayList` est plus efficace pour accéder au i-ième élément, et plus compacte en mémoire ;
  - `LinkedList` est plus efficace pour insérer un élément en début ou en fin de liste, prend plus de mémoire, mais n'a pas besoin d'une zone de mémoire contigue.
- En pratique : on utilise `ArrayList` dans la plupart des cas.

Méthodes définies de Collection précisées par List

Collection comporte des méthodes comme add().
Le sens de ces méthodes est précisé dans **List** :

- add(T v) : ajoute en *fin* de liste ;
- remove(T v) : supprime *la première occurrence* de v
- equals : deux listes sont égales (pour equals) ssi elles ont les mêmes éléments dans le même ordre :

$$[a, b, c] = [a, b, c]$$

mais

$$[a, b, c] \neq [a, c, b]$$

- contains(T v) : renvoie true ssi v se trouve dans la liste ;

Construction d'une liste

Typiquement, on crée une liste vide, puis on lui ajoute des éléments :

```
1 public List<String> creerMots() {
2   // création d'une arraylist vide :
3   List<String> result= new ArrayList<>();
4   result.add("un");
5   result.add("deux");
6   result.add("trois");
7   return result;
8 }
```

Copie d'une collection dans une liste

Les classes de liste ont un constructeur qui prend en argument une autre collection, et remplissent la liste avec les éléments de cette collection :

```
1 public List<String> creerMots(Collection<String> source) {
2   List<String> result= new ArrayList<>(source);
3   return result;
4 }
```

Pratique pour :

- passer d'un type de collection à l'autre (on a un ensemble, mais on veut une liste) ;
- créer une copie d'une collection.

Méthodes spécifiques des listes

- T get(int i) : récupère la valeur en position i ;
- boolean remove(int i) : supprime la valeur en position i ;
- void add(int pos, T v) : insère la valeur v à la position pos, en décalant les autres (l'ancienne valeur en position pos passe en position `pos+1`, etc...) ;
- void set(int i, T v) : remplace la valeur en position i par v. il faut $this.size() > i$;
- subList(int i1, int i2) : retourne la sous-liste entre les positions i1 (inclusive) et i2 (exclusive) ; *modifier le contenu de cette sous liste modifie la liste d'origine.*

Petit exemple avec subList

```
1  List<String> l0 = new ArrayList<>();
2  l0.add("un");
3  l0.add("deux");
4  l0.add("trois");
5  l0.add("quatre");
6  List<String> l1 = l0.subList(1, 3);
7  l1.clear();
8  System.out.println(l0);
```

affiche **[un, quatre]** !!

- $\rightarrow$ il est facile de se tromper
- copies défensives :

```
List<String> l1 = new ArrayList<>(l0.subList(1, 3));
```

Parcours des éléments d'une liste

avec un index :

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

Parcours des éléments d'une liste (for each...)

```
1 public static double somme(List<Double> l) {
2     double s= 0;
3     for (Double v: l) {
4         s+= v;
5     }
6     return s;
7 }
```

Les ensembles (Set)

- Idée centrale : notion d'appartenance ;
- deux ensembles sont égaux ssi ils ont les mêmes éléments ;
- l'ordre dans lequel on ajoute les éléments n'a pas d'importance ;
- $\{a, b\} = \{b, a\}$
- si A contient a, alors $A = A \cup \{a\}$

Informatiquement parlant :

- optimisés pour que le test `contains()` soit très rapide (O(log(n)) vs. O(n) pour les listes) ;
- apparaissent naturellement en modélisation : « les produits en rupture de stock » : liste ou ensemble ?

HashSet

Implémentation de `Set` (à l'aide de tables de hachage, on en reparle plus tard).

On peut faire des `HashSet` de n'importe quel type d'objet :

```
1 HashSet<String> mots= new HashSet<>();
2 HashSet<Integer> nombresPremiers= new HashSet<>();
3 HashSet<Produit> commande= new HashSet<>();
```

Interface et implémentation

Tout `HashSet` **est un** `Set`.
Donc :

- une méthode qui doit retourner un `Set` peut retourner un `HashSet` ;
- une méthode qui accepte comme paramètre un `Set` peut prendre comme paramètre un `HashSet` ;
- question : que préférez-vous comme interface de méthode (en tant qu'utilisateur) :

```
1 public double prixTotal(HashSet<Produit> produits)
2 public double prixTotal(Set<Produit> produits)
3 public double prixTotal(Collection<Produit> produits)
```

Constructeurs de HashSet

- Constructeur par défaut : new HashSet<>() : crée un hashSet vide ;
- Constructeur new HashSet<>(collection) : prend comme argument une *collection* quelconque ; très pratique pour
  - créer un set à partir d'un autre type de collection ;
  - créer une *copie* d'une collection ;
- toutes les collections ont ce type de constructeur.

Méthodes (simples) des Set

- boolean add(T v) : ajoute la valeur v à this ; retourne true si this est modifié ; pas de notion de position ! ! !
- boolean remove(T v) : retire la valeur v à this ; retourne true si this est modifié ;
- boolean contains(T v) : teste si v est dans this ;
- void clear() : vide this ;
- boolean isEmpty() : la collection est-elle vide ?
- int size() : taille de la collection ;
- Object[] toArray() : retourne un tableau avec les éléments de l'ensemble ;
- equals : deux ensembles sont égaux (pour equals) ssi ils ont les mêmes éléments ;
- toString() : renvoie une chaîne montrant les éléments ;

Exemple

Combien de mots différents y-a-t-il dans un tableau de mots ?

```
1 public static int motsDifferents(String[] mots) {
2   Set<String> s= new HashSet<>();
3   for (int i= 0; i < mots.length; i++) {
4     s.add(mots[i]);
5   }
6   return s.size();
7 }
```

Méthodes de collection à collection

- Méthodes qui combinent plusieurs collections ;
- Soient a et b deux collections de même type de base :
- a.addAll(b) : $a = a \cup b$;
- a.removeAll(b) : $a = a - b$;
- a.retainAll(b) : $a = a \cap b$;
- a.containsAll(b) : $b \subset a$
- `addAll`, `removeAll`, et `retainAll` retournent un booléen, qui indique si a a été modifié.

Parcours d'une collection (en général)

- il est naturel de vouloir parcourir les éléments d'une collection ;
- l'implémentation du parcours est très différente selon les collections ;
- `get(i)` ne fonctionne que sur les listes (et est très inefficace sur les `LinkedList`) ;
- en gros : $O(n)$ pour une ArrayList, $O(n^2)$ pour une LinkedList ;
- besoin de quelque chose de plus général et de plus efficace

Itérateur

- Idée : un itérateur est un curseur qui correspond à une certaine position dans une collection ;
- il a des méthodes pour :
  - savoir s'il est encore possible d'avancer dans la collection ;
  - récupérer la valeur courante ;
  - avancer
- en java, récupérer la valeur courante et avancer sont réalisées par la même méthode, next().
- `Iterator` est une interface ;
- l'implémentation de l'interface dépend de la collection parcourue : c'est la collection elle-même qui instancie l'itérateur.

Itérateur

```java
// fonctionne avec tous les types de collections !
public static int somme(Collection<Integer> l) {
  int s= 0;
  // initialisation
  Iterator<Integer> it= l.iterator();
  // Tant qu'il y a des données
  while (it.hasNext()) {
    // on récupère la donnée suivante
    Integer v= it.next();
    s+= v;
  }
  return s;
}
```

Itérateurs et modification des collections

- L'itérateur contient des données sur la structure de la collection ;
- si des éléments de la collection sont ajoutés ou supprimés, ces données peuvent devenir obsolètes.
- interdit d'enlever ou d'ajouter des éléments à une collection pendant qu'on la parcourt avec un itérateur ;
- lève une `ConcurrentModificationException`.

Cas spécifique : on veut parcourir une collection pour en supprimer certains éléments.
C'est possible, mais en le demandant à l'itérateur lui-même :

```
1  public static int supprimerPairs(Collection<Integer> l) {
2    Iterator<Integer> it= l.iterator();
3    while (it.hasNext()) {
4      Integer v= it.next();
5      if (v % 2 == 0)
6        it.remove();
7    }
8  }
```

Boucle for each

« sucre syntaxique » au dessus des itérateurs :

```
1  // fonctionne avec tous les types de collections !
2  public static int somme(Collection<Integer> l) {
3    int s= 0;
4    // pour tout v dans l :
5    for (Integer v: l) {
6      s+= v;
7    }
8    return s;
9  }
```

même limitation qu'avec les itérateurs : pas d'élément ajouté et supprimé en cours de route !