

Introduction programmation Java

Cours 7: Généricité et Collections

Stéphane Airiau

Université Paris-Dauphine

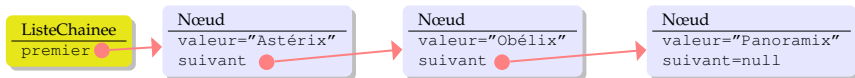
Liste chaînée

```
1 public class Noeud {
2     String valeur;
3     Noeud suivant;
4
5     public Noeud(String val) {
6         valeur = val;
7     }
8
9     public void setSuivant (Noeud next) {
10        suivant = next;
11    }
12 }
```

```
1 public class ListChaine {
2     Noeud premier;
3
4     public ListChaine() {
5         premier = null;
6     }
7
8     public void add(String val) {
9         Noeud nouveau = new Noeud(val);
10        if (premier == null)
11            premier = nouveau;
12        else {
13            Noeud dernier = premier;
14            while (dernier.suivant != null)
15                dernier = dernier.suivant;
16            dernier.suivant = nouveau;
17        }
18    }
19 }
```

Exemple

{"Astérix", "Obélix", "Panoramix"},



On voudrait maintenant faire une liste de Personnages.

- **Idée 1** : changer la classe `Noeud` et la classe `ListeChaine` pour faire deux classes spécialisées `NoeudPersonnage` et `LCPersonnage`.

On voudrait maintenant faire une liste de Personnages.

- ~~Idée 1 : changer la classe Noeud et la classe ListeChaine pour faire deux classes spécialisées NoeudPersonnage et LCPersonnage.~~ **duplication de code !**

On voudrait maintenant faire une liste de Personnages.

- ~~Idée 1 : changer la classe Noeud et la classe ListeChainee pour faire deux classes spécialisées NoeudPersonnage et LCPersonnage.~~ **duplication de code!**
- **idée 2** : mettre Object à la place de String et faire une liste chaînée d'Object.

On voudrait maintenant faire une liste de Personnages.

- ~~Idée 1 : changer la classe Noeud et la classe ListeChainee pour faire deux classes spécialisées NoeudPersonnage et LCPersonnage.~~ **duplication de code!**
 - ~~idée 2 : mettre Object à la place de String et faire une liste chaînée d'Object.~~
- ➡ possible mais nécessitera des transtypages explicites

On voudrait maintenant faire une liste de Personnages.

- ~~Idée 1 : changer la classe Noeud et la classe ListeChainee pour faire deux classes spécialisées NoeudPersonnage et LCPersonnage.~~ **duplication de code!**
- ~~idée 2 : mettre Object à la place de String et faire une liste chaînée d'Object.~~
- ➡ possible mais nécessitera des transtypages explicites
- et si on pouvait ajouter un paramètre à la classe Noeud... c'est ce que Java propose

On voudrait maintenant faire une liste de Personnages.

- ~~Idée 1 : changer la classe Noeud et la classe ListeChainee pour faire deux classes spécialisées NoeudPersonnage et LCPersonnage.~~ **duplication de code!**
- ~~idée 2 : mettre Object à la place de String et faire une liste chaînée d'Object.~~
- ➡ possible mais nécessitera des transtypages explicites
- et si on pouvait ajouter un paramètre à la classe Noeud... c'est ce que Java propose

```
1 public class Noeud<E> {
2     E valeur;
3     Noeud<E> suivant;
4
5     public Noeud(E val) {
6         valeur = val;
7     }
8
9     public void setSuivant (Noeud<E> next) {
10        suivant = next;
11    }
12 }
```

```

1 public class ListChaine<E> {
2     Noeud<E> premier;
3
4     public ListChaine() {
5         premier = null;
6     }
7
8     public void add(E val) {
9         Noeud<E> nouveau = new Noeud<E>(val);
10        if (premier == null)
11            premier = nouveau;
12        else {
13            Noeud<E> dernier = premier;
14            while (dernier.suivant != null)
15                dernier = dernier.suivant;
16            dernier.suivant = nouveau;
17        }
18    }
19
20    public E get(int index) {
21        int i=0;
22        Noeud<E> courant=premier;
23        while (courant.suivant != null && i<index) {
24            i++;
25            courant = courant.suivant;
26        }
27        if (index == i) // on a trouvé l'élément numéro i
28            return courant;
29        else
30            return null;
31    }
32 }

```

Utilisation

```
1 Gaulois asterix =
2     new IrreductibleGaulois("Astérix");
3 IrreductibleGaulois obelix =
4     new IrreductibleGaulois("Obélix");
5 Gaulois Informatix = new Gaulois("Informatix");
6 ListeChaine<Gaulois> liste = new ListeChaine<Gaulois>();
7 liste.add(asterix);
8 liste.add(obelix);
9 liste.add(informatix);
```

- Attention, le paramètre de type ne peut pas être un type primitif (ex **int**, **char**, **double**, etc...)
ex : Noeud<**int**> n'est pas permis.
- Lors de l'appel au constructeur, on n'est pas obligé de répéter les paramètres (mais il ne faut pas oublier les chevrons)
ex : ListeChaine<Gaulois> liste = **new** ListeChaine<>();
Java va inférer les paramètres

Autoboxing

Maintenant que Java peut connaître le type des objets contenus dans une structure, Java offre des possibilités pour simplifier le code : par exemple la transformation automatique dans des types primitifs

```
1 ListeChaine<Integer> maListe = new ListeChaine<Integer> ();
2 //old style
3 maListe.add(new Integer(7));
4 Integer sept = maListe.get(1);
5 System.out.println(sept.intValue());
6 //new style
7 maListe.add(6);
8 int six = maListe.get(2);
```

Quelques aspects plus techniques

On peut avoir deux paramètres (ou plus)

```
1 class <nom classe> < paramètre 1 [, paramètre 2] >{  
2     //on peut déclarer des attributs des classes paramètres  
3     <paramètre 1> attribut;  
4     //on peut déclarer des méthodes qui retournent l'attribut  
5     <paramètre 1> <nom méthode> (<liste arguments>) { ... }  
6     ...  
7 }
```

Par exemple, il existe une classe `Map<K, V>` qui associe à une clé de type `K` une valeur de type `V`.

Héritage au niveau de la classe

```
1 | class <nom classe> < paramètre >  
2 |     extends <classe mère> < paramètre >  
3 | { ... }
```

```
1 | class Tuple<T,U> { ... }  
2 | class ApprentisMentor<T,U> extends Tuple<T,U> { ... }
```

Héritage au niveau des paramètres (borne de types)

```
1 | class <nom classe> < paramètre extends <classe mère>>  
3 | { ... }
```

```
1 | class Distribution<E extends Personnage>{ ... }
```

On indique ici que le type E doit hériter du type Personnage.

On peut avoir plusieurs bornes, mais cela devient plus avancé.

But : écrire une méthode `swap` qui permute deux éléments d'un tableau.

- Peu importe le type du tableau, la méthode pour permuter deux éléments est la même.
- écrire une méthode `static` qui prend un tableau en paramètre

➡ écrire une méthode `static` générique

Quand on déclare une méthode générique, le paramètre de type est déclaré avant le type de retour et après la portée (`public`, `private`) et l'indication d'une méthode de classe (`static`).

```
1 | public class ArrayUtil {  
2 |     public static <T> void swap(T[] array, int i, int j) { ... }
```

Notez qu'il n'est pas utile de préciser un paramètre pour la classe `ArrayUtil`. Lorsqu'on va utiliser `swap`, on ne va pas instancier un objet, on va juste appeler la méthode `static`, il est donc important que ce soit cette méthode qui utilise un paramètre.

Appel d'une méthode `static` générique

Quand on appelle une méthode générique, on n'a pas besoin spécifier le paramètre de type, il est inféré par Java.

```
ex: ArrayUtil.swap(villageois, 2, 6);
```

Si on y tient, on peut quand même donner le type (cela permettra un meilleur message d'erreur si quelque chose se passe mal).

```
ex: ArrayUtil.<Gaulois>swap(villageois, 2, 6);
```

Une subtilité qu'il est important de comprendre

```
1 | ListeChaine<Gaulois> lg = new ListeChaine<Gaulois> ();  
2 | ListeChaine<Personnage> lp = lg;
```

Ligne 2, on a envie de dire qu'une liste de Gaulois est une liste de Personnages.
Cependant...

```
3 | lp.add(new Personnage("Jules César"));  
4 | Gaulois g = lg.get(1);
```

Quand on obtient un élément via la liste `lg`, on ne récupère pas forcément un Gaulois!

Le compilateur Java **ne** permettra **pas** la ligne 2.

⇨ si `F` est une classe de la descendance de la classe `M`
si `G` est une classe générique,
`G<F>` n'est pas dans la descendance de `G<M>`
Autrement dit, il n'y a pas de relation entre `G<F>` et `G<M>`

Une conséquence

But : On veut utiliser la méthode suivante :

```
static void presente(List<Personnage> l)  
affiche les informations sur chaque Personnage de la liste l.
```

Supposons qu'on ait la liste suivante : `List<Gaulois> villageois`, on voudrait donc utiliser notre méthode `presente` et donc faire l'appel

```
Personnage.presente(villageois);
```

Hélas !

```
Personnage.java:72: error: incompatible  
types: List<Gaulois> cannot be  
converted to List<Personnage>  
Personnage.presente(villageois);
```

On le **répète** : il n'y a pas de relation entre `List<Personnage>` et `List<Gaulois>`!!!

Java propose une solution, mais il nous faut présenter quelques outils.

Jockers

Java offre la possibilité d'utiliser des « jockers » qui va servir à exprimer un type inconnu.

```
1 | ListeChaine<?> list = new ListeChaine<Gaulois>();
```

- On ne pourra pas utiliser la méthode `add` car on devrait passer un élément de type ?
? sera bien un type lors de l'exécution, si on ajoute quelque chose dans la liste, il faut que ce soit du bon type.
➡ puisqu'on ne connaît pas le type, on ne permet pas d'ajout !
- on peut appeler une méthode comme `get`
➡ on pourra effectuer un transtypage avec `Object` (ou mieux si on sait ce que l'on fait)

Pour être utile, on va utiliser le jocker avec une borne (inférieure ou supérieure)

Jockers & borne supérieure

ListChaine<? **extends** Gaulois> pour dire que le type inconnu doit être dans la descendance de la classe Gaulois.

On retrouve notre problème

```
1 | public void presente(ListeChaine<Personnage> liste){
```

✗ on ne peut pas utiliser une liste ListeChaine<Gaulois>

➡ Pour le résoudre, on utilise donc un jocker avec une borne supérieure : la méthode accepte toute liste qui contient des listes d'objets qui sont dans la descendance de Personnage.

Comme cela, on sera sûr de pouvoir appeler des méthodes de la classe Personnage.

```
1 | public void presentezVous(ListeChaine<? extends Personnage> liste){
```

le jocker est borné par une borne inférieure, le type inconnu doit être un parent du paramètre T.

```
1 public class Collections {  
2     public static <T> void copy  
3         (List<? super T> dest, List<? extends T> src) { ... }
```

Dans l'exemple, la borne est un type paramétré (c'est méchant!)

L'exemple est une méthode présente dans Java dans la classe `Collections`.

On veut copier une liste dans une autre.

- on ne veut pas imposer que les deux listes soient du même type
ex : copier un liste de `Gaulois` dans une autre liste de `Gaulois`
- ➡ on veut avoir le droit de copier un liste de `Gaulois` dans une liste de `Personnage`.

Evidemment, on n'aurait pas le droit de copier un liste de `Gaulois` dans une liste d'`IrreductibleGaulois`!

Quelques restrictions

- on ne peut pas utiliser de types primitifs comme paramètres
- on ne peut pas construire un tableau avec des types paramétrés
ex : `Noeud<Gaulois>[] tableauDeNoeuds = new Noeud<Gaulois>[10];` n'est **pas** permis.
- le paramètre d'une classe ne peut pas être utilisé dans un contexte `static`.

```
1 public class Paire<C,V>{  
2     private static V valeurDefaut;  
3     erreur!  
4     public static void setDefaut (V valeur) {valeurDefaut=valeur;}  
6     erreur!
```

- et d'autres subtilités dont on ne parlera pas ici

Collections

Collections

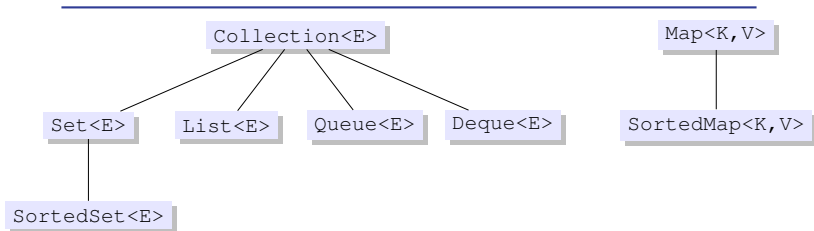
Les listes, les ensembles, les piles, les files d'attente sont des objets qui regroupent plusieurs éléments en une seule entité.

- en commun :
 - mêmes questions : est-ce qu'elles contiennent des éléments ? combien ?
 - mêmes opérations : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure.
- implémentations différentes

Q: Comment peut-on manipuler toutes ces structures ?

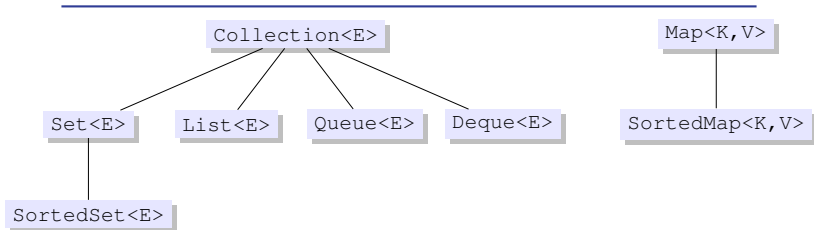
R: ➡ utiliser une hiérarchie d'interfaces

Hierarchie d'interfaces



- **Collection** : méthodes de base pour parcourir, ajouter, enlever des éléments.
- **Set** : cette interface représente un ensemble, et donc, ce type de collection n'admet aucun doublon.
- **List** : cette interface représente une séquence d'éléments : l'ordre d'ajout ou de retrait des éléments est important (doublons possibles)
- **Queue** : file d'attente : il y a l'élément en tête et il y a les éléments qui suivent. L'ordre d'ajout ou de retrait des éléments est important (doublons possibles)
- **Deque** : cette interface ressemble aux files d'attente, mais les éléments importants sont les éléments en tête et en queue.

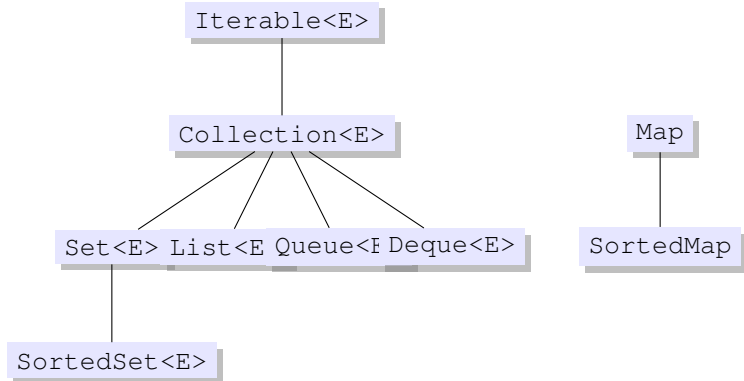
Hierarchie d'interfaces



- Map : cette interface représente une relation binaire (surjective) : chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
- SortedSet est la version ordonnée d'un ensemble
- SortedMap est la version ordonnée d'une relation binaire ou les clés sont ordonnées.

Ces interfaces sont génériques, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection de Gaulois, de Integer, de String, etc...

Parcourir une collection



On peut utiliser une boucle **“for each”** sur tout objet implémentant l'interface `Iterable`.

Parcourir une collection : première solution

En utilisant la généricité ➡ le compilateur va connaître le type des éléments qui sont contenus dans la collection.

- **Situation** : on a une collection `maCollection` qui contient des objets de type `E`.
- On va accéder à chaque élément de la collection `maCollection` en utilisant le mot-clé `for`,
- chaque élément sera stocké dans une variable `<nom>` de type `E` (évidemment).

```
1 | Collection<E> maCollection;  
2 | ...  
3 | for (E <nom> : maCollection)  
4 |     // block d'instructions
```

Parcourir une collection : première solution

```
1 List<Gaulois> villageois = new ArrayList<Gaulois>();
2 villageois.add(new Gaulois("Asterix"));
3 villageois.add(new Gaulois("Cétaumatix"));
4 villageois.add(new Gaulois("Agecanonix"));
5 villageois.add(new Gaulois("Ordralfabétix"));
6
7 for (Gaulois g: villageois)
8     System.out.println(g);
```

Parcourir une collection : deuxième solution

Utilisation d'un objet dédié au parcourt d'éléments dans une collection : un objet qui implémente l'interface `Iterator`.

Obtention : appel à la méthode `iterator()` (interface `Iterable`)

```
1 public interface Iterator<E> {
2     boolean hasNext ();
3     E next ();
4     void remove (); //optional
5 }
```

- `hasNext()` retourne un `boolean` qui indique s'il reste des éléments à visiter,
- `next()` donne accès à l'élément suivant,
- `remove()` permet d'enlever l'élément de la collection.

utilisation :

- enlever un élément lors de l'itération
- parcourt de plusieurs collections en parallèle.

Parcourir une collection : deuxième solution

```
1 List<Gaulois> villageois = new ArrayList<Gaulois>();
2 villageois.add(new Gaulois("Asterix"));
3 villageois.add(new Gaulois("Cétaumatix"));
4 villageois.add(new Gaulois("Agecanonix"));
5 villageois.add(new Gaulois("Ordralfabétix"));
6 villageois.add(new Gaulois("Bonemine"));
7
8 Iterator<Gaulois> it = villageois.iterator();
9 while (it.hasNext()) {
10     Gaulois g = it.next();
11     if (g.getName().equals("Asterix"))
12         it.remove();
13     else
14         System.out.println(g);
15 }
16
17
```


Implémentations

Pour chacune des interfaces, il existe plusieurs implémentations

