

# Introduction Programmation Java

## Cours 4

Stéphane Airiau

Université Paris-Dauphine

# Liste chaînée

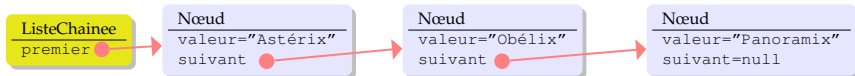
```
1 public class Noeud {
2     String valeur;
3     Noeud suivant;
4
5     public Noeud(String val) {
6         valeur = val;
7     }
8
9     public void setSuivant (Noeud next) {
10        suivant = next;
11    }
12 }
```

```
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"}`,



Il est possible de définir une classe à l'intérieur d'une classe !

⇒ de telles classes peuvent simplifier le développement

- classe interne d'instance

- cette classe va donc accéder à tous les champs de sa classe (même les privés)
- on ne peut créer une instance d'une classe interne que depuis une méthode d'instance de sa classe « englobante »

- classe interne statique

- pas de référence à des membres d'instance de sa classe « englobante »
- on peut instancier cette classe (elle est **static** vis à vis de sa classe englobante).

## exemple

---

On pourrait avoir une classe `Noeud` comme classe interne de la classe `ListeChaine`

On voudrait maintenant faire une liste de Personnages

- changer la classe `Noeud` et la classe `ListeChaine` pour faire deux classes spécialisées `NoeudPersonnage` et `LCPersonnage`.
- 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 IrreductibleGaulois 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);
```



## 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);
```

## Types paramétrés

---

On peut avoir deux paramètres.

```
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  }
```

### Héritage au niveau de la classe

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

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

### Héritage au niveau des paramètres

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

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

## Exemple

---

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

Sur la 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 récupère un élément via la liste `lg`, on ne récupère pas 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>`.

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 `?`,
- on peut appeler une méthode comme `get`  
↳ transtypage avec `Object`.

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

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

✗ On ne peut pas utiliser une liste `ListeChaine<Gaulois>`

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

## Méthodes génériques

---

```
1 | public interface List<E> extends Collection<E> {  
2 |     public <T> T[] toArray(T[] a) { ... }
```

le jocker est bornée par une borne inférieur, 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) { ... }
```

# Collections

## Collections

---

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

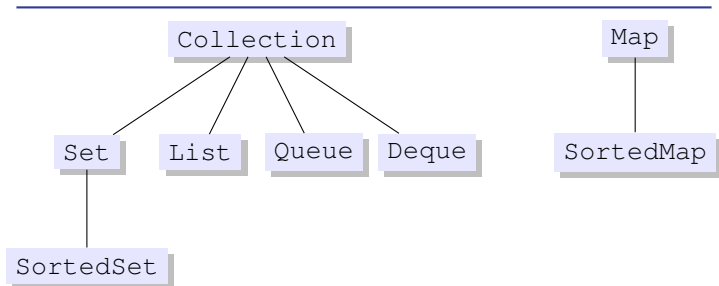
- en commun :
  - mêmes questions : est-ce qu'elles contiennent des éléments ? combien ?
  - mêmes opérations : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure.
- implémentations différentes

Q : Comment peut-on manipuler toutes ces structures ?

R : ➡ utiliser une hiérarchie d'interfaces

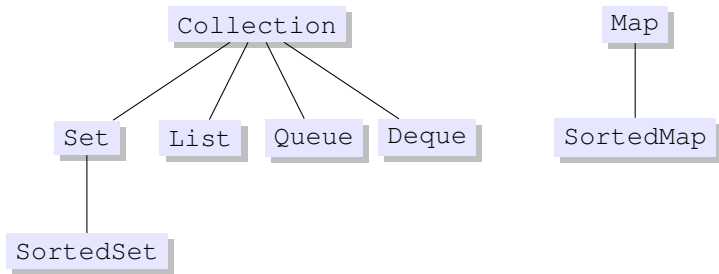


## Hierarchie d'interfaces



- **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

## 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 : 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 : 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 `Collection`)

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

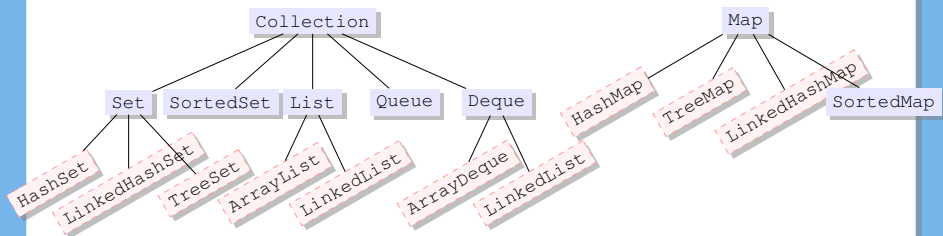
- `hasNext()` retourne un `boolean` qui indique s'il reste des éléments à visiter,
- `next()` donne accès à l'élément suivant,
- `remove` permet d'enlever l'élément de la collection.

**utilisation :**

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

# Implémentations

Pour chacune des interfaces, il existe plusieurs implémentations



Interface `Comparable` contient une seule méthode :

```
public int compareTo(T o)
```

Cette méthode retourne

- un entier négatif si l'objet est plus petit que l'objet passé en paramètre
- zéro s'ils sont égaux
- un entier positif si l'objet est plus grand que l'objet passé en paramètre.

les classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres implémentent toutes l'interface `Comparable`.

## Exemple

---

```
1 public class Gaulois extends Personnage
2     implements Comparable<Gaulois>{
3     String nom;
4     int quantiteSanglier;
5     ...
6
7     public int compareTo(Gaulois ixis) {
8         return this.quantiteSanglier - ixis.quantiteSanglier;
9     }
10 }
```

## interface Comparator

ex : trier les éléments d'une collection : utilisation interface Collections

```
1 // interface Collections
2 public static <T extends Comparable<? super T>
3     void sort (List<T> list)
4 public static <T> void sort
5     (List<T> list, Comparator<? super T> c)
```

```
1 public interface Comparator<T> {
2     int compare(T o1, T o2);
3     boolean equals (Object obj);
4 }
```

Pour comparer des Gaulois, et même tous les Personnage selon leur taille, on peut écrire la classe suivante :

```
1 public class TriHauteur implements Comparator<Personnage> {
2     public int compare(Personnage gauche, Personnage droit) {
3         return gauche.hauteur < droite.hauteur ? -1 :
4             (gauche.hauteur == droite.hauteur ? 0 : 1);
5     }
6 }
```



## Exemple

---

Ensuite, on peut utiliser cette nouvelle classe pour trier des Personnages selon leur taille.

```
1 public static void main(String[] args) {
2     Personnage obelix = new IrreductibleGaulois("Obelix", 1.81);
3     Gaulois asterix = new IrreductibleGaulois("Astérix", 1.60);
4     Personnage cesar = new Personnage("César", 1.75);
5
6     ArrayList<Personnage> personnages =
7         new ArrayList<Personnage>();
8     personnages.add(asterix);
9     personnages.add(obelix);
10    personnages.add(cesar);
11
12    for (Personnage p: personnages)
13        System.out.println(p.presentation());
14
15    Comparator<Personnage> hauteur = new TriHauteur();
16    Collections.sort(personnages, hauteur);
17
18    for (Personnage p: personnages)
19        System.out.println(p.presentation());
```