

## Séance 3

---

# Polymorphisme & compléments sur les collections

Dans cette séance, la notion de polymorphisme est introduite. Cette notion est particulièrement adaptée aux collections d'objets, car elle permet pour une collection d'objets hétérogènes (i.e. de types différents), mais héritant d'une classe commune, d'appliquer pour chaque objet la version la plus spécialisée d'une méthode. Une description plus approfondie des fonctionnalités des collections en Java est donnée en seconde partie.

### Sommaire

---

<b>3.1 Polymorphisme</b>	<b>40</b>
<b>3.2 Collections</b>	<b>42</b>
3.2.1 L'interface Collection	43
3.2.2 Les listes doublement chaînées	44
3.2.3 les vecteurs ou tableaux dynamiques	45
3.2.4 Les tables de hachage et les collections partiellement ordonnées	45
3.2.5 Les ensembles	46
3.2.6 Les associations	46
3.2.7 Les vues et les opérations de masse	46
3.2.8 Algorithmes	47

---

Le code de base de ce chapitre est le code écrit lors du chapitre précédent :

```
public class Personne {  
  
    private String nom;  
    private static int nbTotalPersonnes=0;  
  
    public Personne(){  
        nbTotalPersonnes++;  
    }  
  
    public Personne(String nom){  
        this();  
        setNom(nom);  
    }  
  
    public String getNom() {  
        return nom;  
    }  
}
```

```

    public void setNom(String nom) {
        this.nom = nom;
    }

    public static int getNbTotal() {
        return nbTotalPersonnes;
    }

    public String toString(){
        return "[Personne] Je suis "+getNom()+" , une grande personne";
    }
}

public class Etudiant extends Personne {

    private static int nbTotalEtudiants=0;

    Etudiant(String nom){
        setNom(nom);
        nbTotalEtudiants++;
    }

    public static int getNbTotal() {
        return nbTotalEtudiants;
    }

    public String toString(){
        return "[Etudiant] Je suis "+getNom()+" , tres studieux";
    }
}

public class Professeur extends Personne{

    private static int nbTotalProfesseurs=0;

    public Professeur( String nom){
        setNom(nom);
        nbTotalProfesseurs++;
    }

    public static int getNbTotal() {
        return nbTotalProfesseurs;
    }
}

```

L'archive TD05.zip contient le code de ces trois classes et peut être importée directement dans NetBeans.

### 3.1 Polymorphisme

Si une classe B hérite du type A, on peut instancier un objet de la classe B avec le type de A :

```
A o = new B();
```

Dans ce cas, seuls les méthodes et les attributs de la classe A sont disponibles dans l'instance o.

**Exercice 44.** (\*) *Ecrire une méthode void jamesBond() dans la classe Etudiant qui affiche "[Etudiant] Je m'appelle (nom), (nom) Bond", puis vérifier que celle-ci n'est pas accessible pour Casimir déclaré avec `Personne casimir = new Etudiant("Casimir")`. Ecrire ensuite une méthode void jamesBond() dans la classe Professeur qui affiche "[Professeur] How do you do, Miss Money Penny?".*

Si B redéfinit une méthode f de la classe A, alors, en Java, l'appel :

```
o.f();
```

provoque l'exécution du corps de la méthode f de la classe B (au lieu de A) : la méthode la plus spécialisée est appelée.

En ce sens, le typage en Java est dynamique : bien que l'instance o ait été déclarée comme de type A, le type effectif à l'exécution tient compte du type utilisé lors de la création de l'objet référencé. Dans notre exemple, on dit que le type statique de l'instance o est A tandis que son type dynamique est B.

**Exercice 45.** (\*) *Ecrire maintenant une méthode void jamesBond() dans la classe Personne qui affiche "[Personne] Miss Money Penny, je vous saurais gré de ne pas dire "le vieux" en parlant de moi.". Vérifier qu'elle n'est pas appelée par le code `casimir.jamesBond()`, où casimir est l'instance d'une Personne de l'exercice précédent.*

Ce mécanisme est tout particulièrement intéressant pour une collection d'objets qui héritent d'un même type. Par exemple, dans le code suivant,

```
public void afficherCollection(Collection<Personne> personnes){
    for (Personne p: personnes)
        p.jamesBond();
}
```

pour chacun des objets de la collection, la méthode jamesBond() exécutée est fonction de son type dynamique.

**Exercice 46.** (\*) *Tester avec une collection hétérogène de personnes.*

Ce mécanisme est un cas particulier de la notion de polymorphisme :

**Définition 3.1** (Polymorphisme). *Le polymorphisme permet d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.*

On peut spécifier explicitement un type (dynamique) à utiliser à un moment précis, sous réserve que ce type vérifie soit "compatible" avec l'instance :

**Définition 3.2** (Transtypage ou conversion de type). *Pour un type de base, une conversion de type est autorisée dans certains cas. La valeur de l'attribut n'est pas forcément préservée :*

```
int i = 2; float j = (float) i; // j=2.0 (conversion de int vers float)
double f = 2.5; int n = (int) f; // n=2 (conversion de double vers int
    en prenant la partie entière)
```

Pour une instance d'une classe, il existe deux possibilités de conversion de type :

- *conversion ascendante* : elle permet de prendre une instance d'une classe fille B pour une instance directe d'une classe parente A. Dans certains langages, cela peut servir à appeler une méthode f de la classe A au lieu de la méthode f de la classe B, lorsqu'on souhaite utiliser l'implémentation de la méthode f de A. En java, avec une conversion ascendante, les seules méthodes de la classe parente seront accessibles, mais les méthodes appelées seront toujours celles les plus spécialisées.
- *conversion descendante* : elle permet de déclarer un type dynamique d'une classe fille du type statique, sous réserve que l'instance est été créée avec un type dynamique "suffisamment bas" dans la hiérarchie d'héritage.

Exemple :

```
// on suppose que
//      B herite de A
//      C herite de B
//      A,B et C possèdent une methode f

B o = new B();
A a = (A) o; // conversion ascendante de o

A z = new B();
B b = (B) z; // conversion descendante de z (OK)
C c = (C) z; // conversion descendante de z : echec car o a ete cree
           avec un type B seulement

o.f(); // appel de la methode f de B
((A) o).f(); // Java: appel de la methode f de B ...

z.f(); // appel de la methode f de B (polymorphisme)
((A) z).f(); // Java: appel de la methode f de B
```

**Exercice 47.** (★★) *Tester les conversions ascendante et descendante en exploitant les classes `Personne` et `Etudiant`.*

En java, l'instruction `instanceof` permet de tester le type d'une instance :

**Définition 3.3** (L'instruction `instanceof`). *Cette instruction renvoie un booléen qui indique si une instance hérite d'une classe.*

```
if o instanceof A { ... }
```

Par conséquent, `instanceof Object` est toujours vrai, quel que soit l'instance à laquelle on l'applique.

L'instruction `instanceof` est utile notamment lorsqu'on souhaite appeler des méthodes spécifiques à certaines classes.

Supposons par exemple que la classe `Etudiant` possède une méthode `int getIdentifiantEtudiant()`. Soit le code suivant :

```
public void afficherCollection(Collection<Personne> personnes){
    for (Personne p: personnes)
        if (p instanceof Etudiant){
            System.out.println(((Etudiant)p).getIdentifiantEtudiant());
        }
}
```

La méthode `afficherCollection` affichera les identifiants pour les personnes de la collection qui sont des étudiants.

**Avertissement** : il est assez tentant d'utiliser `instanceof` dans de multiples situations. En fait, cette instruction est à utiliser avec *parcimonie* car elle ajoute de la rigidité au code, en introduisant des dépendances pour son fonctionnement à des types explicites. Dans la plupart des cas, il existe des solutions plus élégantes que l'emploi de `instanceof`, en général en exploitant le polymorphisme.

## 3.2 Collections

Cette partie est consacrée à l'API des Collections en Java. Comme cette librairie est très vaste, il ne s'agit pas d'une étude exhaustive.

### 3.2.1 L'interface Collection

L'interface `Collection` est une des principales interfaces de Java. Elle sert pour le stockage et le parcours d'un ensemble d'objets, pas nécessairement de même type.

Cette interface possède deux méthodes essentielles :

```
boolean add(Object obj)
Iterator iterator()
```

La méthode `add` ajoute un objet dans la collection. Elle renvoie `true` si l'ajout de l'objet a effectivement modifié la collection, `false` sinon.

La méthode `iterator` renvoie un objet implémentant l'interface `Iterator`, qui permet notamment de parcourir une collection. Cette interface possède trois méthodes principales :

```
Object next()
boolean hasNext()
void remove()
```

En appelant plusieurs fois la méthode `next`, vous pouvez parcourir tous les éléments de la collection un par un. Lorsque la fin de la collection est atteinte, la méthode `next` déclenche une exception `NoSuchElementException`<sup>1</sup>. Enfin, la méthode `remove` supprime l'élément renvoyé par le dernier appel à `next`.

**Exemple de parcours d'une collection `c` en utilisant un itérateur :**

```
Iterator i = c.iterator();
while (i.hasNext()){
    Object o = i.next();
}
```

Pour le parcours d'une collection, les boucles `for` sont bien adaptées.

**Exemple de parcours d'une collection `c` avec une boucle `for` :**

```
for (Object o: c){
    ...
}
```

L'interface `Collection` déclare aussi ces méthodes, dont les prototypes sont suffisamment explicites :

```
int size()
boolean isEmpty()
boolean containsObject( Object )
boolean containsAll( Collection )
boolean equals( Object )
boolean addAll( Collection )
boolean remove( Object )
boolean removeAll( Collection )
void clear()
boolean retainAll( Collection )
Object[] toArray()
```

La classe `AbstractCollection` définit les méthodes fondamentales `size` et `Iterator` comme abstraites et implémente toutes les autres. Une classe de collection concrète peut donc se limiter à l'implémentation des méthodes fondamentale.

---

1. Les exceptions sont l'objet du chapitre 4.

### 3.2.2 Les listes doublement chaînées

La classe `LinkedList` est l'implémentation Java des listes doublement chaînées. Elle implémente les interfaces `Collection` et `List`.

La figure 3.1 donne une illustration de la structure de données d'une liste doublement chaînée.

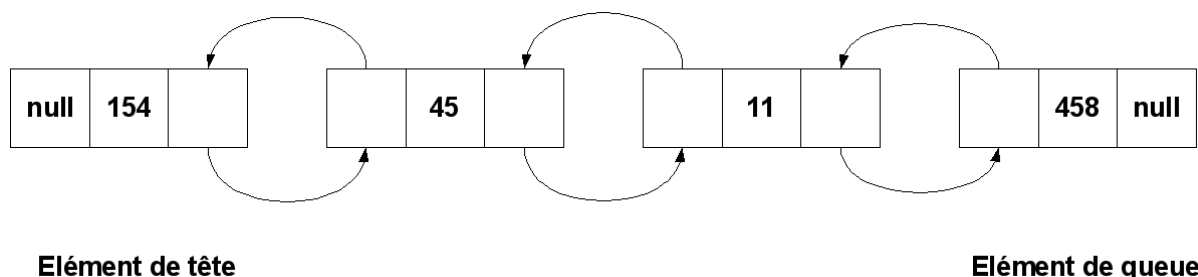


FIGURE 3.1 – Exemple d'une liste doublement chaînée, contenant les éléments 154, 45, 11 et 458.

Les avantages/inconvénients d'une liste doublement chaînée sont les suivants :

- *Avantages* : suppression, ajout d'un élément rapide (à l'endroit de la liste où 'on est positionné) ;
- *Inconvénients* : la récupération d'un élément nécessite le parcours des éléments qui le "précède".

La méthode `add` fournie ajoute un élément en fin de liste.

Pour ajouter un élément au sein de la liste, il faut utiliser un objet `ListIterator` (une interface héritant de l'interface `Iterator`) qui permet l'insertion d'un élément à la position courante, via sa méthode `add`.

La classe `ListIterator` ajoute également 2 méthodes `previous` and `hasPrevious` qui permettent le parcours de la liste à rebours.

**Exemple** (ajout de la chaîne "bob", en deuxième position) \_\_\_\_\_

```
LinkedList ll = new LinkedList();
...
ListIterator li = ll.listIterator();
li.next();
li.add("Bob");
```

**Exercice 48.** (★★) Dans les questions ci-après, afficher à chaque étape le contenu (via une boucle) de votre liste doublement chaînée afin de contrôler le bon fonctionnement de vos opérations :

1. créer une liste doublement chaînée contenant deux étudiants *a* et *b* ;
2. insérer un étudiant *c* entre *a* et *b* ;
3. insérer un étudiant *d* entre *a* et *c* ;
4. supprimer l'étudiant *c*.

**Exercice 49.** (★★) Ecrire une méthode `afficherInverse` qui affiche le contenu d'une liste doublement chaînée en la parcourant en sens inverse.

La classe `ListIterator` possède aussi une méthode `set` qui permet de substituer un objet à l'élément courant. La classe `LinkedList` dispose d'une méthode `get(int i)` retournant l'élément d'indice *i*, mais si vous l'utilisez, c'est que vos données ne devraient pas être stockées sous forme d'une liste chaînée !

### 3.2.3 les vecteurs ou tableaux dynamiques

La classe `ArrayList` encapsule un tableau classique `Object []` dans un tableau dynamique (i.e. sans restriction sur le nombre d'éléments). Elle implémente les interfaces `List` et `Collection`. Les méthodes `get` et `set` sont donc disponibles pour accéder directement aux éléments d'un `ArrayList`.

La classe `ArrayList` est probablement l'implémentation d'une collection la plus utilisée, car elle offre des fonctionnalités très complètes, incluant toutes celles d'un tableau classique et la souplesse d'un stockage d'un nombre quelconque d'éléments.

La figure 3.2 donne une illustration de la structure de données d'un tableau.

<b>Valeur</b>	<b>45</b>	<b>154</b>	<b>58</b>	<b>78</b>	<b>31</b>	<b>5</b>	<b>74</b>
<b>Index</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>

FIGURE 3.2 – Exemple d'un tableau, contenant les éléments 45, 154, 58, 78, 31, 5 et 74.

Les avantages/inconvénients d'un tableau sont les suivants :

- *Avantages* : accès rapide à n'importe quel élément ;
- *Inconvénients* : suppression/insertion d'un élément lent.

Lorsque deux processus sont susceptibles de faire des accès "simultanés" sur un tableau, il est possible d'utiliser la classe `Vector` au lieu de `ArrayList`. La classe `Vector` gère les accès concurrents par des processus, mais offre en contre-partie de moins bonnes performances.

**Exercice 50.** (★) Refaire l'exercice 48, en utilisant un tableau dynamique comme structure de données.

### 3.2.4 Les tables de hachage et les collections partiellement ordonnées

Les listes chaînées et les tableaux vous permettent de spécifier l'ordre (linéaire) dans lequel vous organisez vos éléments. Si l'ordre n'a pas d'importance, il existe des structures de données qui vous permettent de retrouver un élément beaucoup plus rapidement.

Une structure de données classique pour retrouver simplement un élément est la table de hachage. Une table de hachage calcule un nombre entier, appelé code de hachage, pour chacun des éléments. La table de hachage est constituée en général d'un tableau de listes chaînées.

En java, le code de hachage d'un objet est donné par la méthode `int hashCode()`.

**Remarque importante** : lorsque l'on redéfinit la méthode `equals` d'une classe, il faut toujours redéfinir conjointement la méthode `hashCode`, pour garantir le bon fonctionnement des tables de hachage pour les objets de cette classe.

Un élément est stocké dans la liste correspondant à son code de hachage modulo le nombre de listes. Ainsi pour retrouver un élément, il suffit de réduire son code de hachage modulo le nombre de listes, et parcourir la liste à sa recherche.

La classe `TreeSet` repose sur une table de hachage et permet de stocker des ensembles partiellement ordonnés, moyennant une petite pénalité.

La comparaison entre deux éléments repose sur la méthode `compareTo` de l'interface `Comparable`.

Pour plus d'informations sur les tables de hachage, voir votre cours d'algorithmique.

### 3.2.5 Les ensembles

Un ensemble correspond à une collection d'éléments ne figurant qu'une seule fois chacun dans la collection. La méthode `add` n'ajoute un élément que s'il n'est pas déjà présent. Naturellement, c'est la méthode `equals` qui est utilisée pour tester si un objet est égal à un autre.

La classe `HashSet` offre une implémentation de l'interface `Set` (qui correspond aux ensembles).

**Exercice 51.** (\*\*) *Faites en sorte en redéfinissant les bonnes méthodes que la collection `hs` définie ci-dessous ne contienne que 2 étudiants au final (et non pas 3) :*

```
Etudiant a = new Etudiant("Arsene Lupin");
Etudiant b = new Etudiant("Dark Vador");
Etudiant b2 = new Etudiant("Dark Vador");
HashSet hs = new HashSet();
hs.add(b);
hs.add(a);
hs.add(b2);
```

### 3.2.6 Les associations

L'interface `Map` modélise des associations (i.e. des applications au sens mathématique) : à chaque clef est associée au plus une valeur. La classe `HashMap` fournit une implémentation de cette interface.

On peut l'utiliser par exemple pour stocker un dictionnaire dont les clefs sont des noms et les valeurs sont les descriptions associées aux noms.

Extrait de code :

```
HashMap<String,String> dictionnaire = new HashMap<String, String>();
dictionnaire.put("Bateau", "Objet flottant");
dictionnaire.put("Sous-marin", "Bateau ayant connu un souci");
System.out.println(dictionnaire.get("Bateau"));
System.out.println(dictionnaire.get("Sous-marin"));
```

**Exercice 52.** (\*) *On souhaite pouvoir associer des remarques à certains étudiants : comment procéder ? Donner un exemple d'utilisation.*

### 3.2.7 Les vues et les opérations de masse

Les méthodes des collections ne sont pas synchronisées, ce qui signifie que pour améliorer les performances, il n'y a pas de protection gérant les accès concurrents de la part des processus. Il est possible néanmoins d'obtenir des vues synchronisées, en faisant appel à certaines méthodes de la classe `Collections`, comme dans l'exemple ci-dessous :

```
HashMap hm = new HashMap();
Map map = Collections.synchronizedMap(hm);
```

Vous pouvez également obtenir une collection en lecture seule (vue non modifiable), comme dans cet exemple

```
List l = new LinkedList();
List l2 = new Collections.unmodifiableList(l);
```

Il est aussi possible de travailler directement avec un sous-ensemble (vue restreinte) : ainsi le code

```
List groupe2 = etudiants.subList(10,20);
```



définit le groupe 2 comme les éléments d'indice 10 à 19.

Une opération de masse permet de manipuler directement un ensemble d'éléments d'une liste. Par exemple pour calculer l'intersection entre deux ensembles a et b, vous pouvez utiliser :

```
a.retainAll(b);
```

Autre exemple, pour ajouter les 10 premiers éléments d'une liste b dans une liste a :

```
a.addAll( b.subList( 0,10 ) );
```

Toute opération de masse peut être appliquée à un sous-ensemble, et elle se reflète automatiquement à la liste entière.

**Exercice 53.** (\*\*) *Créez deux listes de personnes distinctes et faites ensuite une troisième liste formée de la concaténation de ces deux listes.*

### 3.2.8 Algorithmes

La méthode `Collections.sort` permet de trier une collection ordonnée (interface `List`). Par défaut, elle utilise comme critère de tri l'implémentation de `compare` de la collection à trier. Il est possible de lui préciser un autre critère de tri directement, en lui fournissant une implémentation de l'interface `Comparator`.

Exemple de comparateur pour des salariés, selon leur salaire :

```
public class Comparateur implements Comparator<Salarie>{
    public int compare(Salarie a, Salarie b){
        double differenceSalaires = a.getSalaire() - b.
            getSalaire();
        if (differenceSalaires < 0 ) return -1;
        if (differenceSalaires > 0 ) return 1;
        return 0;
    }
}
```

Ainsi, la ligne :

```
Collections.sort( cadres, new Comparateur());
```

trie une liste `cadres` d'objets de type `Salarie` en fonction de leur salaire.

Tout ceci s'applique aussi aux méthodes génériques de

- calcul d'un plus grand élément : `Collections.max`;
- recherche dichotomique d'un élément : `Collections.binarySearch` (à n'appliquer qu'à des collections triées;-))

**Exercice 54.** (\*\*\*) *Triez une collection de `Personne` en fonction de l'ordre alphabétique de leurs noms.*

## Fiche de synthèse

### Concepts de base de la programmation objet

- polymorphisme
- types statique et dynamique
- conversions de type (transtypage) ascendant et descendant

### Java

- test du type : instanceof
- ensemble d'objets :
  - l'interface Collection, les listes doublement chaînées LinkedList, les tableaux dynamiques Vector/ArrayList, les tables de hachage, les ensembles HashSet, les associations Map
  - les vues, les opérations de masse
  - les algorithmes de la classe Collections

Exemple de code :

```
public class Main {

    Main(){
        // Collection d'etudiants
        Etudiant a = new Etudiant("a");
        Etudiant b = new Etudiant("b");
        HashSet<Etudiant> etudiants = new HashSet<Etudiant>();
        etudiants.add(a); etudiants.add(b);
        System.out.println("Liste d'etudiants:");
        afficher(etudiants);
        // Collection de professeurs
        Professeur e = new Professeur("e");
        Professeur f = new Professeur("f");
        ArrayList<Professeur> profs = new ArrayList<Professeur>();
        profs.add(e); profs.add(f);
        System.out.println("Liste de professeurs:");
        afficher(profs);
        // Reunion des deux collections
        ArrayList<Personne> tous = new ArrayList<Personne>();
        tous.addAll(etudiants);
        tous.addAll(profs);
        System.out.println("Tous:");
        afficher(tous);
    }

    void afficher(Collection<Personne> personnes){
        for (Personne p: personnes)
            System.out.println(p.toString());
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        new Main();
    }
}
```