

**R2-01-03**

**DÉVELOPPEMENT ORIENTÉ OBJETS**

**QUALITÉ DE DÉVELOPPEMENT**

---

## **Semaine 3**

- Polymorphisme
- Collections et comparaison

**Francis Brunet-Manquat**

Université Grenoble Alpes

IUT 2 – Département Informatique

# Points abordés

---

## Exercice 1 : héritage et association

 Redéfinition et polymorphisme

## Exercice 2 : collections et comparaison

 Liste et ensemble

 Interfaces Comparable et Comparator

## Exercice « fil rouge » : la bataille de Faërun


# Polymorphisme d'héritage (1/2)

---

## Point vocabulaire

 Manipuler des objets de types différents mais qui ont une base commune

 Exemple :


 Manipuler un tableau de personnes comprenant des étudiants et du personnel indistinctement et leur demander à tous leur email.

# Polymorphisme d'héritage (2/2)


---

## Comment le polymorphisme fonctionne ?


 **Surclassement** (transtypage fille  $\rightarrow$  mère)

 Un objet peut être manipulé comme s'il appartenait à une autre classe dont il hérite

 **Redéfinition** de méthode (**@Override**)

 Une méthode peut se comporter différemment sur différentes classes de la hiérarchie

 **Lien dynamique**

 Le type d'un objet peut être retrouvé à l'exécution et ainsi la méthode appropriée peut être effectuée

## Mécanisme utilisé dans le TP2 !

# Exemple de polymorphisme (1/2)

```
// Tableau d'objets de type Personne
ArrayList<Personne> personnes = new ArrayList<>();

// Les étudiants
Etudiant et1 = new Etudiant("blanchonp", "blanchon", "philippe", groupeA);
Etudiant et2 = new Etudiant("martinf", "martin", "francis", groupeA);
personnes.add(et1);          // ajouts de Etudiant dans un ArrayList de Personnes
personnes.add(et2);

// Le personnel
Personnel per1 = new Personnel("gouliah", "gouliah", "herve");
Personnel per2 = new Personnel("brunetj", "brunet", "jerome");
personnes.add(per1);        // ajouts de Personnel dans un ArrayList de Personnes
personnes.add(per2);

// afficher les email des Personnes
for(Personne personne : personnes) {
    System.out.println(personne.getMail());
}
```


**Surclassement**  
**Redéfinition + Lien Dynamique**

# Exemple de polymorphisme (2/2)

---

 Résultat produit :

```
philippe.blanchon@etu.univ-grenoble-alpes.fr  
francis.martin@etu.univ-grenoble-alpes.fr  
herve.gouliau@univ-grenoble-alpes.fr  
jerome.brunet@univ-grenoble-alpes.fr
```

 Lors de l'exécution, les objets de Type Etudiant, même s'ils ont subi un surclassement dans un tableau d'objet de type Personne, ont utilisé leur méthode redéfinie

# COLLECTIONS ET COMPARAISON

---

# Les Collections

---

 Une collection est un objet qui regroupe (encapsule) des données de même nature

 des étudiants, des billets de train, ...

 Une collection propose

 une organisation des données

 ordonnées ou non, avec ou sans doublons, ...

 des traitements (algorithmes)

 insertion, accès, recherche, tris, ...

 Les Collections sont réparties en 2 « groupes »

 **Collection** : liste et ensemble

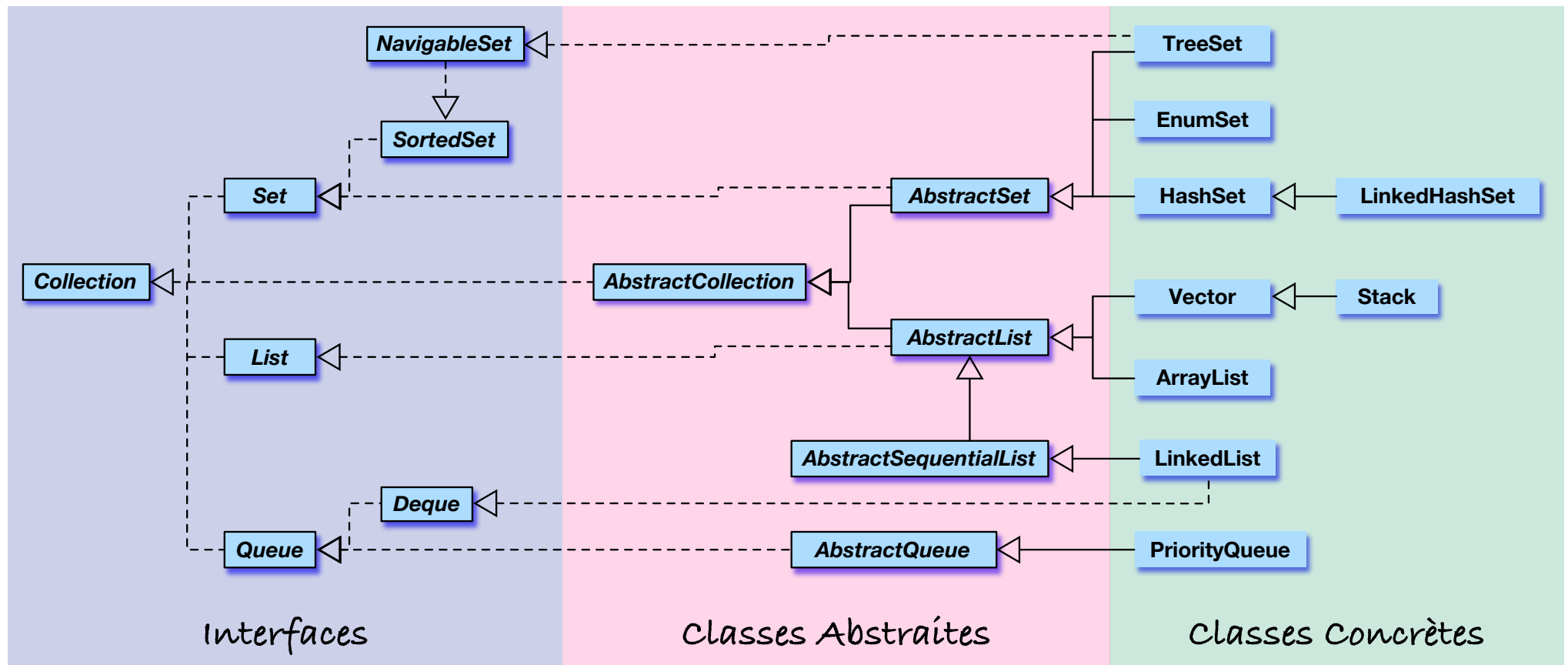
 **Map** : dictionnaire (que nous verrons plus tard)



# Interface Collection

Classes disponibles dans **Collection<E>**





Collection d'objets de type E



# Classe abstraite

---



## Classe abstraite A

-  une classe doit être déclarée abstraite lorsqu'elle possède au moins une méthode abstraite
-  méthode abstraite = méthode sans corps (implantation)
-  une sous-classe de A implantera les méthodes abstraites de A
-  une classe abstraite ne peut pas être instanciée sous forme d'objets

# Classe concrète

---




## Classe concrète C

-  une classe destinée à être instanciée sous forme d'objets
-  si elle hérite d'une classe abstraite A : elle doit implanter les méthodes abstraites de la classe A

# Interface (ce n'est pas une classe !)

---





## Interface I

-  **définition abstraite de services (spécification)**
-  une interface est définie par un ensemble de méthodes abstraites (services) et éventuellement des constantes de classe
  -  les méthodes sont implicitement publiques (public) et abstraites (abstract)

## Définition



```
public interface Mon_Interface {  
    // déclaration de constantes  
    type nomDeConstante = valeur ;  
    // signature de méthodes publiques abstraites  
    type maMethode(type et nom des paramètres formels) ;  
}
```

## Implantation

-  n'importe quelle classe peut implanter une interface ...
-  **public class** Ma\_Classe **implements** Une\_Interface {...}
-  ... en implantant toutes les méthodes que l'interface propose
-  une classe peut implanter plusieurs interfaces

# Classes concrètes d'implantation des interfaces

## Les classes concrètes





-  implantent les objets de la collection (structure de données physique)
-  implantent les méthodes d'utilisation de la collection (accès et mutation)

		Classe d'implantation			
		Table de hachage	Tableau redimensionnable	Arbre équilibré	Liste chaînée
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Deque		ArrayDeque		LinkedList
	Map	HashMap		TreeMap	




# Pour l'exercice 2

---

## `java.util.ArrayList<E>`

-  Un `java.util.ArrayList` utilise un tableau en interne pour ranger les données.
-  Un `ArrayList` fournit un accès aux éléments par leur indice très performant et est optimisé pour des opérations d'ajout/suppression d'éléments en fin de liste.
-  Complexité : Les opérations `size`, `isEmpty`, `get`, `set`, `iterator` sont exécutées en temps constant.
-  Les opérations d'ajout/suppression sont exécutées en temps constant amorti (les ajouts/suppressions en fin de liste sont plus rapides).



## `java.util.TreeSet<E>`

-  La classe `java.util.TreeSet` utilise un conteneur `TreeMap` (arbre binaire équilibré de couples (clé, valeur) pour ranger les données.
-  Les éléments sont triés dans l'**ordre naturel** ou en utilisant un comparateur (`Comparator`) utilisé lors de la création du `TreeSet`.
-  Complexité : Les opérations `add`, `remove`, `contains` sont en  $O(\log n)$ . `HashSet` est donc plus performant pour ces opérations.

# Pour l'exercice fil rouge

---

## `java.util.LinkedList<E>`

-  Un `java.util.LinkedList` utilise une liste chaînée pour ranger les données. L'ajout et la suppression d'éléments est aussi rapide quelle que soit la position, mais l'accès aux valeurs par leur indice est très lent.
-  Complexité : Les opérations `size`, `isEmpty`, `add`, `remove`, `set`, `get` sont exécutées en temps constant. Toutes les méthodes qui font référence à un indice sont exécutées en temps  $O(n)$ .

 Possible aussi : `java.util.PriorityQueue<E>`

# LES INTERFACES DE COMPARAISON

---


Mettre en pratique une interface



# Ordre naturel en Java : Comparable

---

## L'interface Comparable ...

-  Une classe MaClasse pour laquelle on a besoin de l'**ordre naturel** utilisé par le tri d'une collection doit implémenter l'interface Comparable

```
public class MaClasse implements Comparable<MaClasse>
{...}
```

## ... ne déclare qu'une seule méthode :

**int** compareTo(**Object**)

-  MaClasse **doit** redéfinir la méthode compareTo

-  **objetDeMaClasse**.compareTo(**autreObjetDeMaClasse**)

# Ordre naturel en Java : Comparable




---

 La méthode

**int** compareTo(**Object**)

 `objetDeMaClasse.compareTo(autreObjetDeMaClasse)`

 ... doit retourner

-  une valeur entière négative (-1) si cet objet (`objetDeMaClasse`) est inférieur à l'objet (`autreObjetDeMaClasse`) fournit en paramètre
-  une valeur entière positive (+1) si cet objet (`objetDeMaClasse`) est supérieur à l'objet (`autreObjetDeMaClasse`) fourni en paramètre
-  une valeur nulle (0) si cet objet (`objetDeMaClasse`) est égal à l'objet (`autreObjetDeMaClasse`) fourni en paramètre

# Ordre naturel en pratique pour MaClasse

---

## Implanter Comparable en redéfinissant compareTo()

```
public class MaClasse implements Comparable<MaClasse> {  
    // mes attributs  
  
    // mes méthodes  
  
    @override  
    public int compareTo(MaClasse unObjet) {  
        ...  
        return -1; // je suis inférieur à unObjet  
        ...  
        return 0;  // je suis égal à unObjet  
        ...  
        return 1;  // je suis supérieur à unObjet  
    }  
  
    public String toString() {  
        ...  
    }  
}
```

# Ordre naturel : comment l'utiliser ?

---

 Les objets implémentant l'interface Comparable seront triés à l'insertion dans une collection « triée » comme TreeSet

 Avec la méthode Collections.sort(**collection**)

 **collection** doit pouvoir être triée à tout moment

 **OK** pour ArrayList, Vector, LinkedList

 **KO** pour Stack, PriorityQueue, TreeSet

**CompareTo()** sera appelée de manière transparente !

# Quand l'ordre naturel ne suffit pas ?

---

 Il se peut que les données doivent être comparées selon plusieurs critères

 pour des Etudiants

 ordre naturel sur le nom et le prenom

 ordre sur la moyenne() (pour un classement)


 L'interface Comparator permet de créer des objets « comparateur » monComparateur

```
interface Comparator {  
    public int compare(Object o1, Object o2);  
    public boolean equals(Object o);  
}
```

 ...et les passer en argument à certaines méthodes

 Collection.sort(maCollection, monComparateur)

 ou à certains constructeurs

 new TreeSet<>(monComparateur)



### Une classe comparateur autonome

```
public class MaClasse implements Comparable<MaClasse> { ... }

public class MaClasseCompTriParAttrXY implements Comparator<MaClasse> {
    @Override
    public int compare(MaClasse o1, MaClasse o2) {
        ...
        return -1; // o1 inférieur à o2 pour les attributs X, Y
        ...
        return 0;  // o1 égal à o2 pour les attributs X, Y
        ...
        return 1;  // o1 supérieur à o2 pour les attributs X, Y
    }
}
```



### À instancier (**new**) avant chaque usage

```
// arrayListeDeMaClasse trié avec la classe MaClasseCompTriParAttrXY
Collections.sort(arrayListeDeMaClasse, new
MaClasseCompTriParAttrXY());
```

### Un **comparateur static final** définit dans MaClasse

```
public class MaClasse implements Comparable<MaClasse> {  
    public static final Comparator<MaClasse> CompMaClasseTriParAttrXY  
        = new Comparator<MaClasse>() {  
        @override  
        public int compare(MaClasse o1, MaClasse o2) {  
            ...  
            return -1; // o1 inférieur à o2 pour les attributs X, Y  
            ...  
            return 0; // o1 égal à o2 pour les attributs X, Y  
            ...  
            return 1; // o1 supérieur à o2 pour les attributs X, Y  
        }  
    };  
}
```

### À utiliser tel quel

```
// arrayListeDeMaClasse trié avec comparateur MaClasseTriParAttrXY  
Collections.sort(arrayListeDeMaClasse, CompMaClasseTriParAttrXY);
```