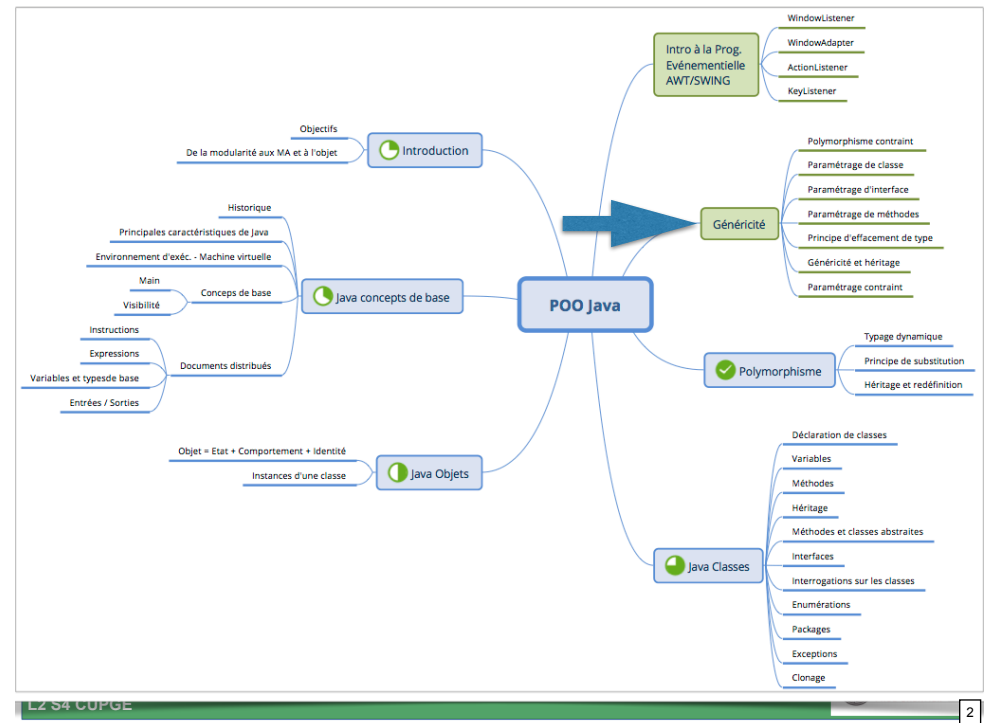




## Objectifs

### □ Généricité

- ❖ Autoboxing
- ❖ Polymorphisme non contraint (avant java 1.5)
- ❖ Paramétrage de classe, d'interface, de méthodes
- ❖ Principe d'effacement de type
- ❖ Généricité et héritage
- ❖ Paramétrage contraint
- ❖ Wildcard



## Polymorphisme non-contraint

### □ Exemple :

```
public class Couple {
    protected Object _fst;
    protected Object _snd;
    public Couple(Object fst, Object snd) {_fst = fst; _snd = snd;}
    public Object getFst() {return _fst;}
    public void setFst(Object fst) {this._fst = fst;}
    public Object getSnd() {return _snd;}
    public void setSnd(Object snd) {this._snd = snd;}
}
```

### ❖ Typage dynamique : paramètre non contraint

```
Couple c1 = new Couple ("Noel", 24); // autoboxing pour 24 -> Integer
```

### ❖ Transtypage obligatoire pour récupérer la valeur

```
String s = (String)c1.getFst();
```

```
Couple c1 = new Couple (24, "Noël");
// récupération => erreur
String s = (String)c1.getFst();
```

### ❖ Mais aucun contrôle

## Polymorphisme non-contraint

### ❑ Possibilité :

#### ❖ Une classe par configuration

```
public class CoupleIntString {
    protected int _fst;
    protected String _snd;
    public CoupleIntString(int fst, String snd) {this._fst = fst;_snd = snd;}
    public int getFst() {return _fst;}
    public void setFst(int fst) {this._fst = fst;}
    public String getSnd() {return _snd;}
    public void setSnd(String snd) {this._snd = snd;}
}
```

#### ❖ Le type est fixe

```
CoupleIntString c2 = new CoupleIntString (24, "Noel");
```

#### ❖ Plus de castage

```
String s = c2.getSnd();
```

Plus de polymorphisme .....

## Paramétrage de classe

### ❑ Paramètres formels entre < >

```
public class GenCouple<T,U> {
    protected T _fst;
    protected U _snd;
    public GenCouple(T fst, U snd) {_fst = fst;_snd = snd;}
    public T getFst() {return _fst;}
    public void setFst(T fst) {this._fst = fst;}
    public U getSnd() {return _snd;}
    public void setSnd(U snd) {this._snd = snd;}
}
```

### ❑ Valables pour

#### ❖ Variables et méthodes d'instances

### ❑ MAIS PAS pour

#### ❖ Variables et méthodes de classe (static)

### ❑ Q : exemple d'utilisation ?

## Paramétrage d'interfaces

### ❑ Exemple : interface Comparable

### ❑ Attention : ici valeur est un entier

#### ❖ Si valeur est une instance, il faut qu'il soit Comparable

#### ❖ Il faut alors utiliser la méthode compareTo pour comparer les valeurs

```
public interface Comparable<T> {
    public int compareTo (T a);
}

public class MaClasse implements Comparable<MaClasse> {
    protected int valeur;
    public int compareTo (MaClasse c) {
        return this.valeur - c.valeur;
    }
}
```

## Paramétrage d'interfaces

```
public interface IPile<T> { // signature des méthodes
    public boolean isEmpty(); public boolean isFull();
    public boolean add(T val); public boolean remove();
    public T get(); public int size();
}
```

```
public class Pile<T> implements IPile<T> {
    ArrayList<T> elements = new ArrayList<>(); // attributs
    public int hauteur=0;
    // constructeur par défaut
    public boolean isEmpty() {...}
    public boolean isFull(){return false}
    public boolean add(T val){...}
    public boolean remove(){...}
    public T get(){...}
    public int size(){...}
    public String toString() {return elements.toString();}
    public boolean equals(Object o) {...}
}
```

```
....dans le main :
Pile<Integer> p = new Pile<>();
p.add(10);
System.out.println(p);
```

## Paramétrage de méthodes

### ❑ Méthodes d'instances ou de classes

- ❖ Paramètres différents de ceux de la classe générique

```
public class GenCouple<T,U> {  
    protected T _fst;  
    protected U _snd;  
    public GenCouple(T fst, U snd) {_fst = fst;_snd = snd;}  
  
    public T getFst() {return _fst;}  
    public <V> boolean sameFst(GenCouple<T,V> p) {  
        return p.getFst().equals(_fst);  
    }  
    ....  
}
```

## Généricité et héritage

### ❑ Une classe générique peut hériter d'une classe non générique

```
class Graphe {...}  
class GraphePondere<P> extends Graphe {...}
```

### ❑ Une classe générique peut hériter d'une classe générique

```
class Reseau<P> extends GraphePondere<P> {...}
```

### ❑ Une classe non générique peut être l'instanciation d'une classe générique

```
class ReseauRoutier extends Reseau<Distance> {...}
```

## Principe d'effacement de type

### ❑ Représentation interne d'une classe générique

- ❖ Totalemment polymorphe +
- ❖ Transtypes

### ❑ Vérification :

- ❖ Compteur d'instances
- ❖ Q : résultat ?

```
public class GenCouple<T,U> {  
    static Integer nbInstances = 0;  
    protected T _fst;  
    protected U _snd;  
    public GenCouple(T fst, U snd) {  
        _fst = fst;  
        _snd = snd;  
        nbInstances++;  
    }  
    ...  
}
```

```
public static void main(String[] args) {  
    GenCouple<Integer,String> c3 = new GenCouple<Integer,String> (24, "Noel");  
    GenCouple<Integer,Integer> c4 = new GenCouple<Integer,Integer> (24, 2007);  
    System.out.print("c3.getClass()==c4.getClass()" + (c3.getClass()==c4.getClass()));  
    System.out.println("nbInstances = " + GenCouple.nbInstances);  
    System.out.println("c3 instanceof GenCouple : " + (c3 instanceof GenCouple));  
    ...  
}
```

## Paramétrage contraint

### ❑ On peut contraindre le paramètre de généricité :

```
class Dictionnaire<A extends Comparable<A>>
```

### ❑ Ainsi, le paramètre de généricité A devra être instancié

- ❖ avec une classe qui est Comparable
- ❖ ou dont un parent est Comparable
- ❖ une instance de la classe utilisée comme argument pour A DOIT posséder la méthode **public int compareTo(A)**

## Wildcard

### ❑ Exemple : classification de figures géométriques

```
public abstract class Figure {...}
public class Rectangle extends Figure {...}
public class Cercle extends Figure {...}
public class Triangle extends Figure {...}
...
public static void lister(Collection<Figure> pw) {
    for(Figure f : pw)
        System.out.println(f);
}
...
ArrayList<Figure> figures = new ArrayList<Figure>();
figures.add(new Cercle()); // ici on peut remplir la liste
figures.add(new Rectangle());
figures.add(new Triangle());
lister(figures); // pas de pb
...
```

## Exercice : Files génériques

### ❑ Classe PersonneAvecPrio

- ❖ attribut priorité générique et comparable
- ❖ les personnes sont comparables (comparaison des priorités)

## Wildcard

### ❑ Mais avec une collection de Cercle (par exemple) :

```
ArrayList<Cercle> cercles = new ArrayList<Cercle>();
cercles.add(new Cercle());
lister(cercles); // !!!!!!!!!!!!!!!
```

- ❖ Ça ne compile pas ....!!

### ❑ Il faut utiliser un "joker" :

- ❖ Instanciation avec une classe qui hérite de Figure
- ❖ sans être obligé d'instancier avec Figure

```
public static void lister(Collection<? extends Figure> pw) {
    for(Figure f : pw)
        System.out.println(f);
}
```

## Exercice : Files génériques

### ❑ Classe PersonneAvecPrio

- ❖ priorité générique et comparable
- ❖ personne comparable

```
public class PersonneAvecPrio <P extends Comparable<P>>
    implements Comparable<PersonneAvecPrio<P>> {

    protected String nom;
    protected String prenom;
    protected P prio;
    public PersonneAvecPrio(String n, String p, P pr) {...}
    // accepteurs getters et setters
    public String toString() {...}
    public int compareTo(PersonneAvecPrio<P> p) {...}
    public boolean equals(Object o) {...}
}
```

## Exercice : Files génériques

### Interface IFileAvecPrio

- ❖ Méthodes de l'interface file, avec gestion de priorités
- ❖ Plus *toutes* les méthodes de l'interface Collection

```
public interface IFileAvecPrio<T extends Comparable<T>> extends
Collection<T>{
    /** supprime un des éléments le plus prioritaire */
    public boolean remove();
    /** @return élément de plus forte priorité */
    public T maxPrioritaire();
}
```

### Ecrire le code de FileAvecPrio

### Ecrire le code de la classe Main de test, avec une file de PersonneAvecPrio<Integer>

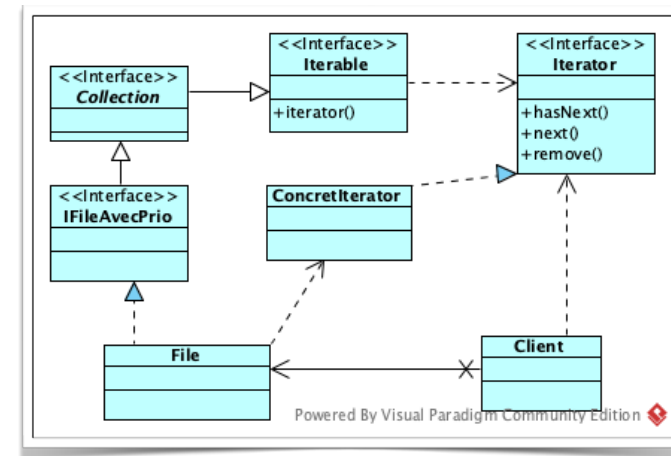
## Iterateurs

- ❑ Implémentation de la méthode iterator() qui retourne un itérateur
- ❑ L'itérateur parcourt les données dans l'ordre naturel
  - ❖ ici, ordre d'ajout dans la collection
  - ❖ car on utilise l'itérateur de l'ArrayList

```
public Iterator<T> iterator() {
    return al.iterator();
}
```

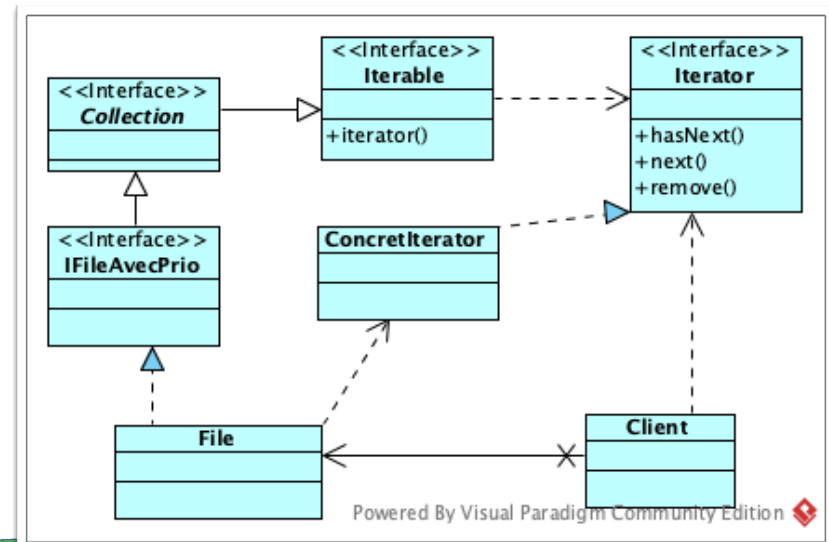
## Iterateurs

### Ordre « naturel »



## Comparators

### Ordre imposé



## Comparators

- ❑ Pour parcourir les données selon un ordre imposé
  - ❖ Classes qui implémentent l'interface Comparator en fonction de l'ordre souhaité par le client

```
public class TriPrioriteDecroissante<P> extends Comparable<P> implements
Comparator<PersonneAvecPrio<P>> {
    public int compare(PersonneAvecPrio<P> p1, PersonneAvecPrio<P> p2) {
        // TODO
    }
}
```

```
public class TriNomCroissant<P> extends Comparable<P> implements
Comparator<PersonneAvecPrio<P>> {
    public int compare(PersonneAvecPrio<P> p1, PersonneAvecPrio<P> p2) {
        // TODO
    }
}
```

## Comparators

- ❑ Permettre à FileAvecPrio d'être parcourue dans un ordre imposé

```
// retourne un itérateur qui parcourt dans l'ordre donné par le comparateur
public Iterator<T> iteratorComparator(Comparator<T> comp) {
    return (new IteratorComparator(comp));
}

// inner class pour créer un itérateur
protected class IteratorComparator implements Iterator<T>{
    protected ArrayList<T> instantData; //safe iterator
    protected int index;

    protected IteratorComparator(Comparator<T> comp) {
        instantData = new ArrayList<>(); // triés par priorité décroissante
        index = 0;
        instantData.addAll((Collection<? extends T>) al); // wildcard
        // tri des données dans l'ordre imposé par le comparateur
        instantData.sort(comp);
    }

    public boolean hasNext() {...}
    public T next() {...}
    public void remove() {...}
}
```

## Comparators

- ❑ Tester
  - ❖ Parcours dans l'ordre naturel
  - ❖ Parcours dans l'ordre imposé
    - tri par ordre alphabétique
    - tri par priorités décroissantes