

# Développement Orienté Objets

## Héritage et Polymorphisme

Petru Valicov  
petru.valicov@umontpellier.fr

<https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets>

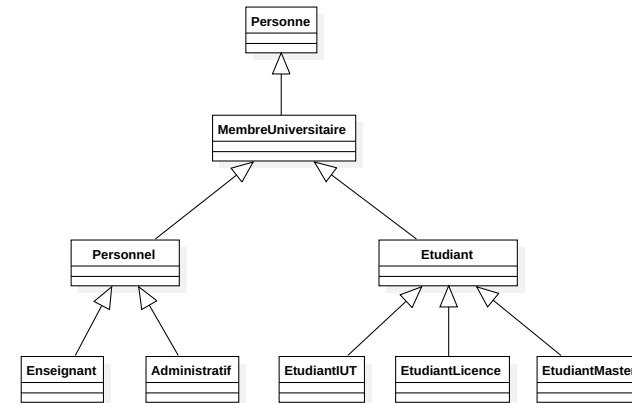
2021-2022



1

## Héritage/Généralisation

- Souvent on est amené à hiérarchiser les classes en factorisant les caractéristiques communes. Exemple :



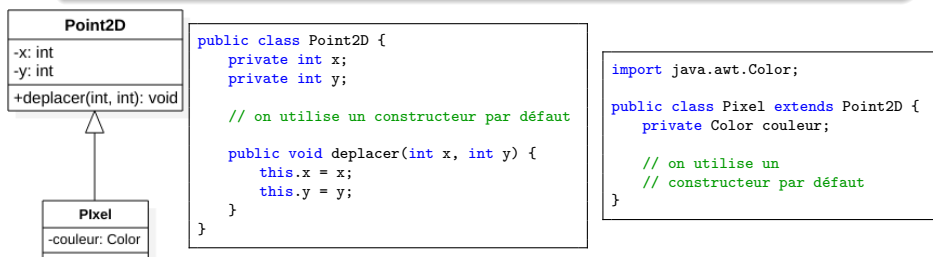
- Ici, un Etudiant est un MembreUniversitaire avec des particularités qui le distingue de Enseignant par exemple.

2

## Héritage/Généralisation

### Définition

L'héritage est le mécanisme par lequel une classe hérite de la définition (attributs et méthodes) d'une autre classe.



- ici les objets de type Pixel sont également de type Point2D :  
possèdent **toutes les propriétés** des Point2D  
possèdent **toutes les compétences** des Point2D
- ici le pixel est donc déplaçable... **peut-on toujours déplacer un pixel dans la vie réelle ?**

3

## Pourquoi hériter ?

- pour la modularité :
  - aide à éviter la duplication
  - meilleure structuration de la définition des classes
- pour spécialiser (*Personne* → *Personnel* → *Enseignant*)
- pour étendre les fonctionnalités (un étudiant a plus de propriétés et fonctionnalités qu'un membre universitaire)

... bref, **pour mieux réutiliser**

**En contrepartie, il y a des conséquences à assumer**

### Vocabulaire

Si Enseignant hérite de Personnel alors on dit que :

- Personnel est une *super-classe* de Enseignant
- Enseignant est une *sous-classe* de Personnel

4

## Héritage : respect du contrat

- l'héritage est une relation *transitive* de type "**EST\_UN**" : si la classe **A** hérite de la classe **B** et **B** hérite de la classe **C**, alors **A** hérite aussi de **C**

- La sous-classe est liée par le "*contrat*" de sa classe parente :
  - elle ne peut pas restreindre la visibilité d'une méthode de la classe parente :

code OK

```
public class B {  
    public void maMethode(){  
        // corps  
    }  
}
```

code interdit à la compilation

```
public class A extends B {  
    private void maMethode(){  
        // corps  
    }  
}
```

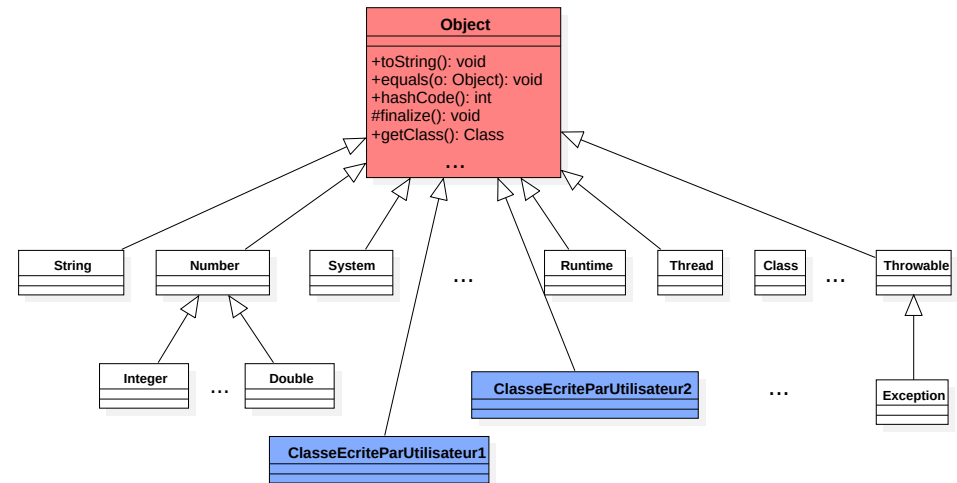
- elle ne peut pas ne pas hériter une méthode/attribut définie dans sa classe parente

**MALGRÉ TOUT ÇA** la classe-fille n'a pas accès aux méthodes/attributs privés de la classe mère.

5

## Héritage : exemple avec l'API Java

En Java toutes les classes héritent d'une classe spéciale : **Object**



Par respect du contrat d'**Object**, toutes les classes possèdent les méthodes d'**Object** et héritent du comportement d'**Object**.

6

## Héritage : exemple simple

```
public class Personnel {  
    private String nom;  
  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

```
public class Enseignant extends Personnel {  
    private int numero;  
  
    public void attribuerNumero(int n) {  
        numero = n;  
    }  
}
```

```
public class App {  
  
    public static void main(String[] args) {  
  
        // instantiation d'un objet de type Personnel en tant qu'Enseignant  
        Personnel membre = new Enseignant();  
        membre.setNom("Toto");  
  
        // instantiation d'un objet de type Enseignant en tant qu'Enseignant  
        Enseignant prof = new Enseignant();  
        prof.setNom("Lolo"); // utilisation de la méthode publique héritée de Personnel  
        prof.attribuerNumero(10059392);  
  
    }  
}
```

7

## Héritage : comment ça marche ?

```
public class App {  
  
    public static void main(String[] args) {  
  
        // instantiation d'un objet de type Personnel en tant qu'Enseignant  
        Personnel membre = new Enseignant();  
        membre.setNom("Toto");  
  
        // instantiation d'un objet de type Enseignant en tant qu'Enseignant  
        Enseignant prof = new Enseignant();  
        prof.setNom("Lolo"); // utilisation de la méthode publique héritée de Personnel  
        prof.attribuerNumero(10059392);  
  
    }  
}
```

- En Java le typage se fait *statiquement* : **Personnel membre**
- upcasting* - instantiation d'un objet d'un type de base en tant que sous-type : **membre = new Enseignant()**
- Une méthode spéciale dans la classe **Object** permet de retrouver le type effectif de l'objet : **getClass()**  
**(vous devriez l'éviter dans 99% de cas)**

8

## Héritage et constructeurs

La construction d'une instance de la sous-classe **commence toujours** par la construction de sa partie héritée.

- Si le constructeur de la classe mère est sans paramètres ou est par défaut, alors Java l'appelle "artificiellement"
- Si un constructeur avec des paramètres est défini dans la classe mère, alors la sous-classe est **forcée** à avoir un constructeur appelant **un des constructeurs** de la classe mère :

```
public class A {  
    private int attribut;  
  
    public A(int attribut){  
        this.attribut = attribut;  
    }  
}
```

```
public class B extends A{  
  
    public B(int valeur){  
        super(valeur); // appel au constructeur de la classe mère  
    }  
}
```

9

## Héritage et constructeurs : respect du contrat

```
public class Personnel {  
    private String nom;  
  
    public Personnel(String nom) {  
        this.nom = nom;  
    }  
}
```

```
public class Enseignant extends Personnel {  
    private int numero;  
  
    // cette définition ne passe pas à la compilation  
    public Enseignant(int numero) {  
        this.numero = numero;  
    }  
}
```

```
public class App {  
  
    public static void main(String[] args) {  
  
        Personnel perso = new Enseignant(10059392); // ne peut pas fonctionner  
  
        Enseignant ens = new Enseignant(10059392); // ne peut pas fonctionner  
  
    }  
}
```

10

## Héritage et constructeurs : respect du contrat

```
public class Personnel {  
    private String nom;  
  
    public Personnel(String nom) {  
        this.nom = nom;  
    }  
}
```

```
public class Enseignant extends Personnel {  
    private int numero;  
  
    public Enseignant(String nom, int numero) {  
        super(nom);  
        this.numero = numero;  
    }  
}
```

```
public class App {  
  
    public static void main(String[] args) {  
  
        Personnel perso = new Personnel("Fifi");  
  
        Personnel perso = new Enseignant("Toto", 10059392);  
  
        Enseignant ens = new Enseignant("Lolo", 10059392);  
  
    }  
}
```

11

## Mot-clé super vs this en Java

- **Rappel** : **this** est la référence vers l'objet courant. Cas d'utilisation :
  - depuis un constructeur, appel d'un autre constructeur de la même classe
  - accéder depuis une fonction à un attribut, ayant le même nom qu'une variable locale
  - passer la référence **this** en paramètre d'une fonction
- Le mot-clé **super** permet d'accéder aux éléments de la classe mère la plus proche dans la hiérarchie. Cas d'utilisation :
  - pour appeler le constructeur de la classe mère
  - pour accéder à une fonction de la classe mère

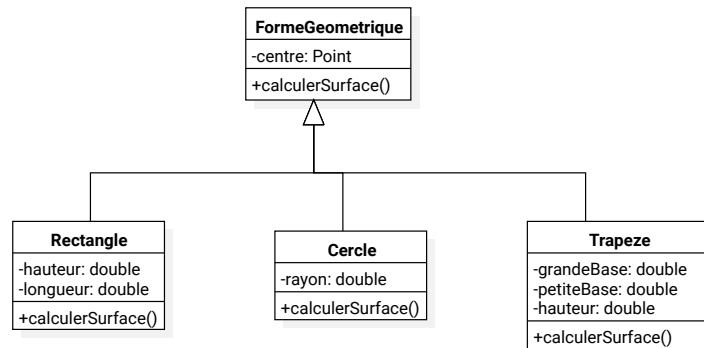
```
public class Personnel {  
    private String nom;  
  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
  
    public void travailler() {  
        System.out.println("Personnel qui travaille");  
    }  
}
```

```
public class Enseignant extends Personnel {  
    private int numero;  
  
    public void attribuerNumero(int n) {  
        numero = n;  
    }  
  
    public void travailler() {  
        // appel de la méthode de la super-classe  
        super.travailler();  
        System.out.println("Enseignant qui travaille");  
    }  
}
```

12

## Polymorphisme

- parfois certaines méthodes de la classe mère nécessitent une certaine adaptation, il est donc possible des les *redéfinir*.



- Ici, lorsque l'on invoque la méthode `calculerSurface()` sur un objet de type `Cercle`, c'est la méthode de `Cercle` qui sera invoquée (et pas celle de la classe mère).

13

## Polymorphisme : illustration

```
public class FormeGeometrique {
    private double centre;
    // on suppose avoir défini les coordonnées de chaque point pour décrire la forme géométrique

    public void calculerSurface() {
        // calcul intégral en utilisant les coordonnées
    }
}
```

```
public class Rectangle extends FormeGeometrique {
    private double hauteur, largeur;

    public void calculerSurface() {
        double resultat = hauteur * largeur;
    }
}
```

```
public class Cercle extends FormeGeometrique {
    private double rayon;

    public void calculerSurface() {
        double resultat = Math.PI * rayon * rayon;
    }
}
```

```
public class ClasseClient {
    public static void main(String[] args) {
        FormeGeometrique forme = new FormeGeometrique();
        forme.calculerSurface(); // la méthode de calcul générale

        forme = new Cercle();
        forme.calculerSurface(); // la méthode de calcul de Cercle

        forme = new Rectangle();
        forme.calculerSurface(); // la méthode de calcul de Rectangle
    }
}
```

14

## Polymorphisme

**Poly** = plusieurs, **morphisme** = forme :

### Définition - Polymorphisme

- Propriété d'un élément de pouvoir se présenter sous plusieurs formes
- Capacité donnée à une même opération de s'effectuer différemment suivant le contexte de la classe où elle se trouve

Dans le cas des figures géométriques, la fonction `calculerSurface()` est une fonction polymorphe.

Le polymorphisme est réalisable grâce à la **Liaison Dynamique** :

### Liaison dynamique

Mécanisme permettant que la définition d'une méthode soit décidée au moment de l'exécution de celle-ci (et non à la compilation).

⇒ **Introspection de type**

15

## Polymorphisme : surcharge

```
public class Etudiant {
    private ArrayList<Double> notes = new ArrayList<>();

    public void ajouterNote(double note) { notes.add(note); }

    public double getMoyenne(){
        double moyenne = 0;
        for (Double d : notes) {
            moyenne += d;
        }
        return moyenne / notes.size()
    }

    // méthode surchargée avec des paramètres différents
    public double getMoyenne(double malusAbsence){
        return getMoyenne() - malusAbsence;
    }
}
```

- la surcharge d'une méthode a lieu lorsqu'on définit une nouvelle méthode ayant le même nom, mais avec :
  - un nombre différents de paramètres et sinon,
  - les types de paramètres différents dans l'ordre (de gauche à droite)
- cas que vous avez déjà vu : la surcharge du constructeur

16

## Polymorphisme : exemples en Java

L'annotation `@Override` indique au compilateur qu'on est en train de redéfinir une méthode de la classe parente

⇒ s'il y a incohérence (erreur dans le nom, ou paramètres), le compilateur va le signaler par une erreur

La classe `Object` possède plusieurs méthodes pratiques mais trois d'entre elles souvent sont à redéfinir :

- `int hashCode()` affecte un code "pseudo-unique" à `this` afin de permettre de l'identifier numériquement
- `String toString()` : une chaîne de caractères censée décrire l'objet  
par défaut : nom de la classe + @ + hash code obtenu à partir des valeurs de chaque attribut de l'objet → peu lisible
- `boolean equals(Object o)` compare `this` à `o` de façon à garantir une relation d'équivalence

17

## `equals(Object o)` et `hashCode()`

Par défaut dans `Object` la méthode `equals(Object o)` fait la comparaison la plus "naïve" :

```
public class Object {
    // du code de Object ici ...

    public boolean equals(Object obj) {
        return this == obj;
    }
}
```

La situation est similaire pour `hashCode()` : typiquement l'adresse de `this` est convertie en un entier (mais ça dépend de la JVM...)

Pensez toujours à redéfinir ces deux fonctions **en même temps** dans vos classes afin de respecter le contrat de `Object` :

- `equals(Object o)` définit une *relation d'équivalence*
- si l'état de l'objet n'est pas modifié alors plusieurs appels à `hashCode()` retournent la même valeur
- si les hash codes sont différents alors `equals(Object o)` retourne `false` (l'inverse n'est pas nécessairement vrai)

18

## `equals(Object o)` et `hashCode()` : exemple

Pensez à vous faire aider par l'IDE en générant ces deux méthodes et en adaptant le code obtenu.

```
import java.util.Objects;

public class Employe {
    private String nrINSEE, nom;
    private int echelon;
    private double base, nbHeures;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Employe employe = (Employe) o;
        return echelon == employe.echelon &&
            employe.base == base &&
            employe.nbHeures == nbHeures &&
            nrINSEE.equals(employe.nrINSEE) &&
            nom.equals(employe.nom);
        // la classe String de Java a déjà redéfini son equals(Object o) - utilisez-le !
    }

    @Override
    public int hashCode() {
        return Objects.hash(nrINSEE, nom, echelon, base, nbHeures);
        // on peut choisir sa propre fonction de hachage
    }
}
```

19

## Héritage : classe abstraite

Des objets de types différents peuvent avoir *quelque chose* en commun, mais ce *quelque chose* ne peut pas exister de manière indépendante.

Exemple : un objet de type *Humain* est trop abstrait pour être instancié dans un système de gestion de personnel...

```
public abstract class Humain {
    private String nom;

    public Humain(String nom) {
        this.nom = nom;
    }

    public String toString(){
        return "mon nom c'est : " + nom;
    }
}
```

```
public class Femme extends Humain {
    // obligation de définir un constructeur invoquant
    // le constructeur explicite de la classe mère
    public Femme(String nom) {
        super(nom);
    }
}
```

```
Humain h1 = null; // On déclare l'objet de type Humain
Humain h2 = new Femme("Toto"); // On l'instancie avec un type concret
System.out.println(h2); // affiche "mon nom c'est : Toto"
Humain h3 = new Humain(); // ERREUR de compilation
```

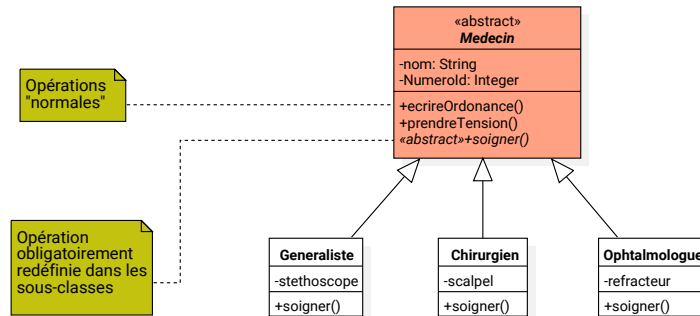
20

## Classe abstraite

### Définition

Une **classe abstraite** est une classe qui ne peut pas être instanciée.

- sert uniquement de super classe à d'autres classes
- peut contenir des **méthodes abstraites**
- ses sous-classes doivent définir toutes les méthodes abstraites (ou être abstraites elles-mêmes)



21

## Classe abstraite : illustration en Java

```
public abstract class Medecin {
    private String nom;
    private int numeroID;

    public void ecrireOrdonnance() { /* du code ici */ }

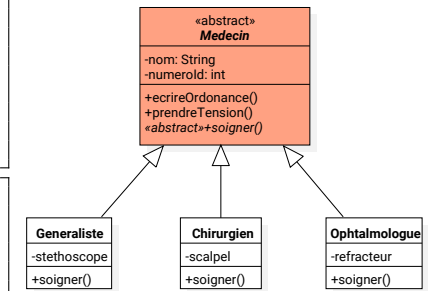
    public void prendreTension() { /* du code ici */ }

    public abstract void soigner(); // méthode abstraite
}
```

```
public class Chirurgien extends Medecin {
    private String scalpel;
    public void soigner() {
        System.out.println("J'utilise un " +
            scalpel + " et j'opère");
    }
}
```

```
public class Generaliste extends Medecin {
    private String stethoscope;
    public void soigner() { System.out.println("J'utilise un " + stethoscope + " et je décide le soin"); }
}
```

```
public class Ophtalmologue extends Medecin {
    private String refracteur;
    public void soigner() { System.out.println("J'utilise un " + refracteur + " pour mesurer la vue"); }
}
```



22

## Classe/méthode abstraite : à quoi bon ?

**Important** : Dès qu'une méthode est déclarée abstraite, la classe qui la contient doit être déclarée abstraite. **Pourquoi ?**

Respect du *contrat* de la classe mère : toutes les classes filles "concrètes" savent effectuer les opérations

- méthode abstraite = méthode "*promise*" ; dans notre exemple :
  - tous les médecins doivent pouvoir "soigner" ...
  - mais chacun à sa façon
- méthode concrète (ordinaire) :
  - toutes les instances vont hériter son implémentation...
  - et ça sera aux sous-classes de *s'y conformer* ou de la *redéfinir*

**Quel est alors l'intérêt des classes abstraites sans méthodes abstraites ?**

23

## Classe/méthode abstraite

Malgré le fait qu'une méthode soit abstraite, on peut l'utiliser dans la classe abstraite... et souvent c'est extrêmement utile :

```
public abstract class Produit {
    private double fraisLivraison;

    // on ne sait pas vraiment calculer le prix
    public abstract double getPrix();

    // mais on sait comment calculer la facture
    public double getPrixFacturé(){
        return getPrix() + fraisLivraison;
    }
}
```

```
public class Livre extends Produit {
    private String nom;
    private double royalties;

    public Livre(String nom, double royalties) {
        this.royalties = royalties;
        this.nom = nom;
    }

    public double getPrix() {
        return royalties + 10;
    }
}
```

```
public class Smartphone extends Produit {
    private String marque;
    private double marge;
    private double taxeRecyclage;

    public Smartphone(String marque, double marge,
        double taxeRecyclage) {
        this.marque = marque;
        this.marge = marge;
        this.taxeRecyclage = taxeRecyclage;
    }

    public double getPrix() {
        return taxeRecyclage + marge;
    }
}
```

```
public class AppProduit {
    public static void main(String[] args) {
        Produit p1 = new Smartphone("Poire", 989.99, 10);
        Produit p2 = new Livre("Tom Sawyer", 35);
        System.out.println("p1 : " + p1.getPrixFacturé() +
            "..., cher pour ce que c'est...");
        System.out.println("p2 : " + p2.getPrixFacturé());
    }
}
```

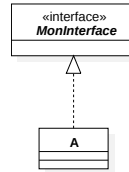
24

## Interfaces en Java

### Rappel

Classe  $\approx$  nom + attributs + opérations

- Une **interface** Java est équivalente à une classe qui n'a pas d'attributs et où toutes les méthodes sont abstraites.
- interface**  $\approx$  classe "totalement" abstraite



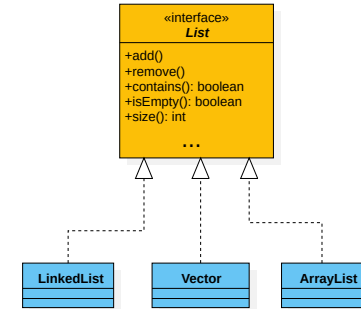
```
public interface MonInterface {
    public int methodeSansAttributs();
    public void methodeAvecAttributs(int x);
}
```

```
public class A implements MonInterface {
    public int methodeSansAttributs() {
        // implémentation concrète ici
    }
    public int methodeAvecAttributs(int x) {
        // implémentation concrète ici
    }
}
```

Dans cet exemple on dit que **A** *implémente* l'interface **MonInterface** (ou encore, **A** est une *réalisation* de l'interface **MonInterface**)

25

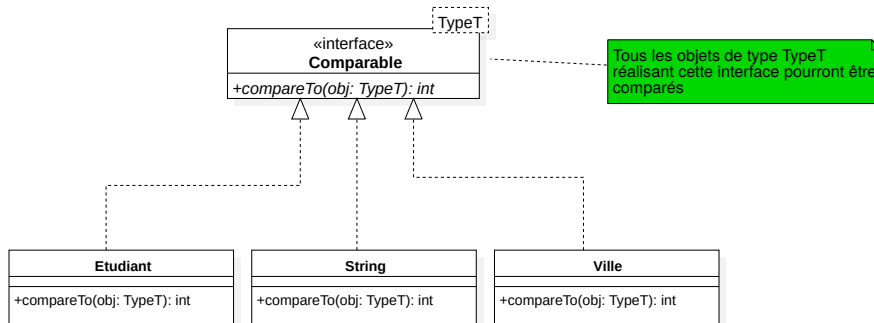
## Exemple d'interface : java.util.List



- Une **List** en tant que telle est purement abstraite (rien ne peut être défini à l'intérieur)
- Mais les différentes implémentations de **List** doivent respecter le contrat de **List** i.e. fournir une implémentation de toutes les méthodes imposées par **List**

26

## Exemple d'interface : java.lang.Comparable



- La méthode **compareTo()** est abstraite et retourne une valeur entière, censée être :
  - positive si l'objet comparable est supérieur à l'objet **obj**
  - 0 (zéro) si l'objet comparable est égal à l'objet **obj**
  - négative si l'objet comparable est inférieur à l'objet **obj**
- Cette méthode abstraite **doit être implémentée** dans chaque réalisation de **Comparable**.

27

## Interface Comparable : illustration

Définition dans l'API Java :

```
// l'interface à implémenter par toute
// classe T destinée à être "comparable"
public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
public class Ville implements Comparable<Ville> {
    private String nom;
    private Integer codePostal;
    private Integer population;

    public Ville(String n, int cp, int p) {
        nom = n;
        codePostal = cp;
        population = p;
    }

    public int compareTo(Ville o) {
        return population.compareTo(o.population);
    }
}
```

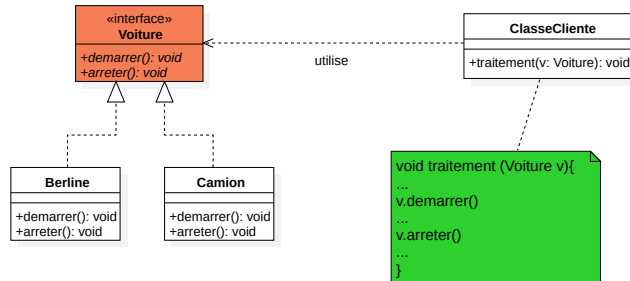
```
public class TestComparable {
    public static void main(String[] args) {
        Ville montpellier = new Ville("Montpellier", 34000, 282143);
        Ville sete = new Ville("Sète", 34301, 43620);

        if (montpellier.compareTo(sete) > 0)
            System.out.println("Montpellier plus grand");
        else if (montpellier.compareTo(sete) < 0)
            System.out.println("Sète plus grand");
        else System.out.println("ils ont la meme population !");
    }
}
```

28

## Classe cliente d'une interface

- Quand une classe dépend d'une interface pour réaliser ses opérations, elle est dite **classe cliente de l'interface**
- On utilise une relation de dépendance entre la classe cliente et l'interface requise



Code Java

- Toute classe implémentant l'interface pourra être utilisée - principe de substitution

29

## Héritage : interface

En règle générale, une interface est équivalente à une classe abstraite qui n'a **que** des méthodes abstraites et éventuellement des champs **static final**.

Depuis Java 8 les interfaces peuvent avoir des méthodes par défaut :

```
public interface MonInterface {
    public void faireQuelqueChose();

    default public void faireAutreChose(){
        System.out.println("Implémentation par défaut");
    }
}
```

```
public class MaClasse implements MonInterface {
    public void faireQuelqueChose(){
        System.out.println("Mon implémentation");
    }
}
```

```
public class ClasseDeTest {
    public static void main(String[] args) {
        MonInterface obj = new MaClasse();
        obj.faireQuelqueChose(); // classique
        obj.faireAutreChose(); // default
    }
}
```

Héritage de **comportement**

### Important

Les méthodes par défaut ne peuvent pas manipuler des attributs dynamiques !

30

## Transtypage

### Définition - Transtypage

Le **transtypage** (ou **cast** en anglais) est un mécanisme de conversion d'une expression d'un certain type vers un autre type. En Java il peut être *implicite* ou *explicite*.

Cast *implicite* :

```
int i = 10;
System.out.println(i); // affiche 10
double d = i; // transtypage implicite de la valeur de i en une valeur de type double
System.out.println(d); // affiche 10.0
```

Cast *explicite* :

```
double reel = 13.42;
System.out.println(reel); // affiche 13.42
int entier = (int)reel; // transtypage explicite
System.out.println(entier); // affiche 13
```

En l'utilisant avec attention, parfois le transtypage peut être utile...

31

## Transtypage et objets

Une classe définit un **type**, donc le cast est également possible :

```
public class Animal{
    private String nom;
    public Animal(String nom){
        this.nom = nom;
    }

    public void dormir(){
        // du code ici
    }
}
```

```
public class Chien extends Animal{
    public Chien(String nom){
        super(nom)
    }

    public void aboyer(){
        // du code ici
    }
}
```

```
public class Chat extends Animal{
    public Chat(String nom){
        super(nom)
    }

    public void miauler(){
        // du code ici
    }
}
```

Cast *implicite* :

```
Animal animal = new Chien("Doggy"); // upcasting : a est un Animal de sous-type Chien
Chien chien = new Chien("Bernny");

Animal[] zoo = new Animal[5]; // un tableau d'objets de type Animal
zoo[0] = animal;
zoo[1] = chien; // cast implicite
```

Cast *explicite* :

```
Animal a = new Chien ("Bernny");
Chien c = (Chien) a; // cast explicite
```

**Voyez-vous un danger avec le cast explicite ?**

32



## Héritage et casting : bêtisier

```
public class Animal{
    private String nom;
    public Animal(String nom){
        this.nom = nom;
    }

    public void dormir(){
        // du code ici
    }
}
```

```
public class Chien extends Animal{
    public Chien(String nom){
        super(nom)
    }

    public void aboyer(){
        // du code ici
    }
}
```

```
public class Chat extends Animal{
    public Chat(String nom){
        super(nom)
    }

    public void miauler(){
        // du code ici
    }
}
```

```
Animal a = new Chien ("Bernny"); // OK ou pas ?
Chien c = (Chien) a; // OK ou pas ?
c.aboyer(); //OK ou pas?
((Chien)a).aboyer(); //OK ou pas?
Animal b = new Chat ("Tom"); //OK ou pas?
Chien c1 = (Chien) b; //OK ou pas ?
c1.aboyer(); //OK ou pas ?
```

L'opérateur `instanceof` permet de tester si un objet est d'un type (ou sous-type) donné.

De manière générale, les architectures bien réfléchies doivent pouvoir éviter son utilisation... **préférez le polymorphisme !**

33

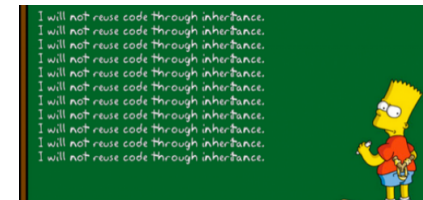
## Héritage : bilan

`extends`  $\approx$  héritage de type, de comportement et d'état  
`implements`  $\approx$  héritage de type (+ comportement en Java 8)

- l'héritage est très pratique et souvent naturel...
- ...mais attention au respect du contrat des classes-parentes

### Conseils :

- si possible préférez les interfaces plutôt que les classes-mères concrètes  $\rightarrow$  alternative plus souple
- si votre **seul but** c'est réutiliser du code, dites-vous bien



- ... et cherchez une alternative pour le réutiliser

34