Développement Orienté Objets
Héritage et Polymorphisme

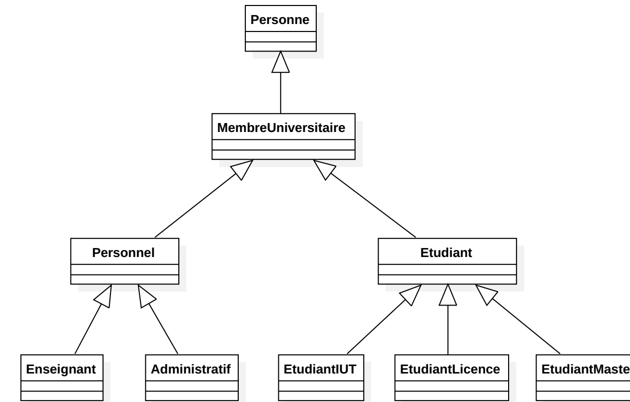
Petru Valicov
petru.valicov@umontpellier.fr

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

2023-2024

Héritage/Généralisation

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

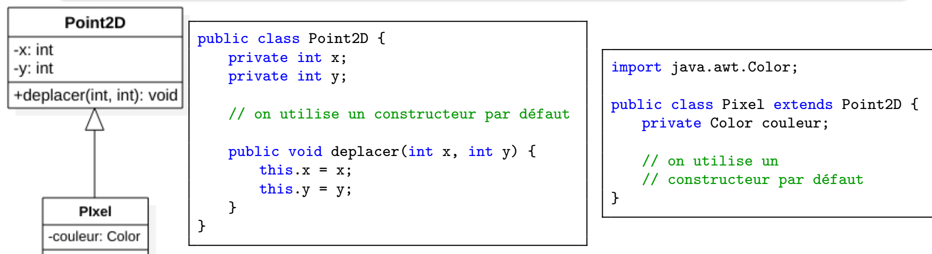


- Ici, un `Etudiant` est un `MembreUniversitaire` avec des particularités qui le distinguent d'un `Enseignant` par exemple.

Héritage/Généralisation

Définition

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




```java
public class Point2D {
    private int x;
    private int y;

    // on utilise un constructeur par défaut

    public void deplacer(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

```java
import java.awt.Color;

public class Pixel extends Point2D {
    private Color couleur;

    // on utilise un
    // constructeur par défaut
}
```

- 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 ?

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 etudiant 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`

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*" à sa classe parente
  - impossible de restreindre la visibilité d'une méthode de la classe parente :

code OK

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

code interdit à la compilation

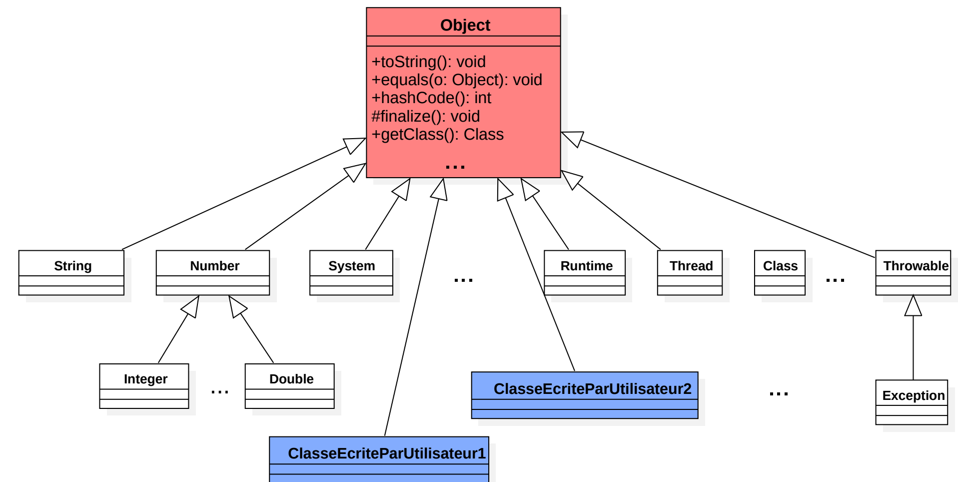
```
public class A extends B {
   private void maMethode(){
     // corps
   }
}
```

  - impossible de 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.

Héritage : exemple avec l'API Java

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



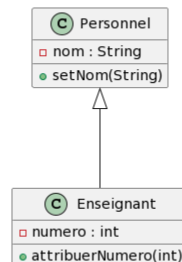
Par respect du contrat d'`Object`, toutes les classes possèdent les méthodes d'`Object` et héritent du comportement d'`Object`.

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 static void main(String[] args) {

    // instanciation d'un objet de type Personnel en tant qu'Enseignant
    Personnel membre = new Enseignant();
    membre.setNom("Toto");

    // instanciation d'un objet de type Enseignant en tant qu'Enseignant
    Enseignant prof = new Enseignant();

    // utilisation de la méthode publique héritée de Personnel
    prof.setNom("Lolo");

    prof.attribuerNumero(10059392);

}
```



Héritage : comment ça marche ?

```
public class App {

   public static void main(String[] args) {

      // instanciation d'un objet de type Personnel en tant qu'Enseignant
      Personnel membre = new Enseignant();
      membre.setNom("Toto");

      // instanciation 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* - instanciation 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)**

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
   }

}
```


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 àla 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

    }
}
```


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);

    }
}
```


Mot-clé super *vs* this en Java

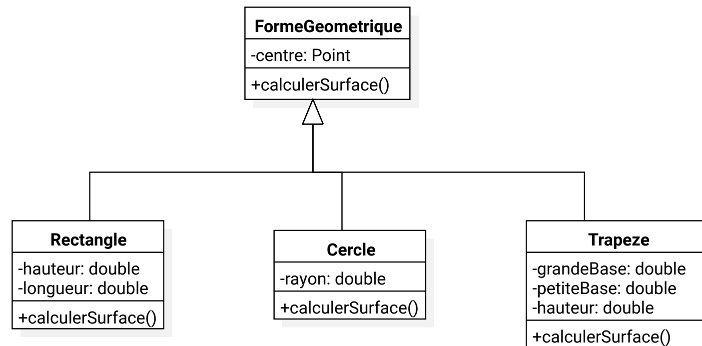
- **Rappel** : `this` est la référence vers l'objet courant. Cas d'utilisation :
  - depuis un constructeur, appel *d'un autre* constructeur de la même classe
  - accéder depuis une fonction à un attribut, ayant le même nom qu'une variable locale
  - passer la référence `this` en paramètre d'une fonction
- 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");
    }
}
```

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).

Polymorphisme : illustration

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

    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 ClasseCliente {
    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
    }
}
```


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).

$\Longrightarrow$ **Introspection de type**

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, <u>mais</u> avec :
  1. un nombre différents de paramètres et sinon,
  2. les types de paramètres différents dans l'ordre (de gauche à droite)
- cas que vous avez déjà vu : la surcharge du constructeur

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. Trois d'entre elles souvent sont à redéfinir :

- `int hashCode()` affecte un code "pseudo-unique" à `this` afin de permettre de l'identifier numériquement
- `boolean equals(Object o)` compare `this` à `o` de façon à garantir une relation d'équivalence
- `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

equals(Object o) et hashCode()

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

```java
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 `o1.hashCode()`≠`o2.hashCode()` alors `o1.equals(o2)==false` (**l'inverse n'est pas toujours vrai**)

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.

```java
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
    }
```


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 *Appareil* est trop abstrait pour être instancié dans un parc de gestion d'appareils connectés...

```java
public abstract class Appareil {
    private String marque;

    public Appareil(String marque) {
        this.marque = marque;
    }

    public String toString(){
        return "mon logo est " + marque;
    }
}
```

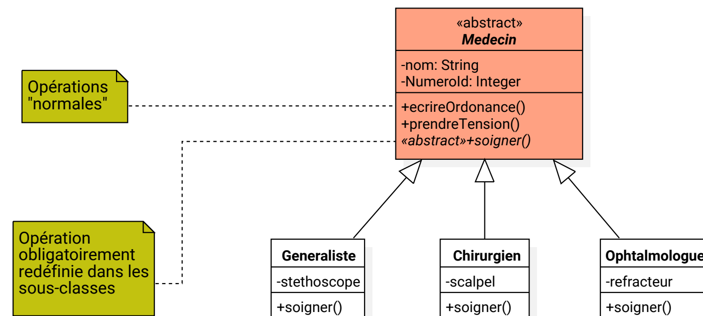
```java
public class Ordinateur extends Appareil {

    // obligation de définir un constructeur invoquant
    // le constructeur explicite de la classe mère
    public Ordinateur(String margue) {
        super(marque);
    }

    public void faireCalcul() {
        //du code propre à un Ordinateur
    }
}
```

```java
Appareil a1 = null; // On déclare l'objet de type Appareil
Appareil a2 = new Ordinateur("ENIAC"); // On l'instancie avec un type concret
System.out.println(a2); // affiche "mon logo est ENIAC"
Appareil a3 = new Appareil(); // ERREUR de compilation
```

Classe abstraite

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

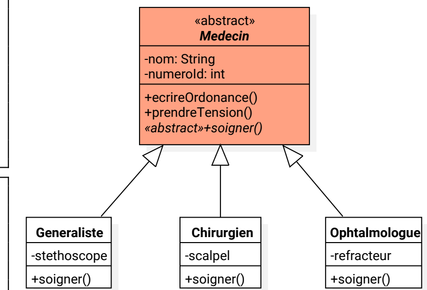
- sert uniquement de super classe à d'autres classes
- peut contenir des **méthodes abstraites**
- ses sous-classes doivent définir toutes les méthodes abstraites (ou être abstraites elles-mêmes)



Classe abstraite : illustration en Java

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

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

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

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

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




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

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


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 "abstraites" de leur mère

- 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 ?

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 :

```java
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;
    }
}
```

```java
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;
    }
}
```

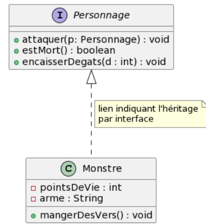
```java
public class Smartphone extends Produit {
    private String marque;
    private double marge;
    private double taxeRecyclage;

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

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

```java
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.getPrixFacture() +
                  "..., cher pour ce que c'est...");
      System.out.println("p2 : " + p2.getPrixFacture());
  }
}
```

Interfaces en Java

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

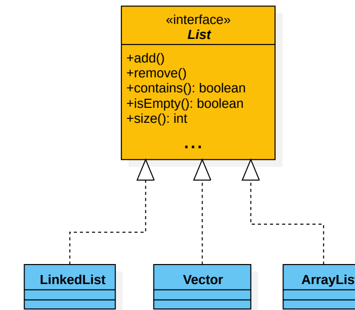
```
public interface Personnage {
   public void attaquer(Personnage p);
   public boolean estMort();
   public void encaisserDegats(int d);
}
```




```
public class Monstre implements Personnage {
   private int pointsDeVie;
   private String arme;

   public void attaquer(Personnage p) {
      if (arme.equals("lance"))
         p.encaisserDegats(2);
      else
         p.encaisserDegats(1);
   }

   public boolean estMort() { return pointsDeVie <= 0; }

   public void encaisserDegats(int d) {
      pointsDeVie -= d;
   }

   public void mangerDesVers() { pointsDeVie++; }
}
```

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

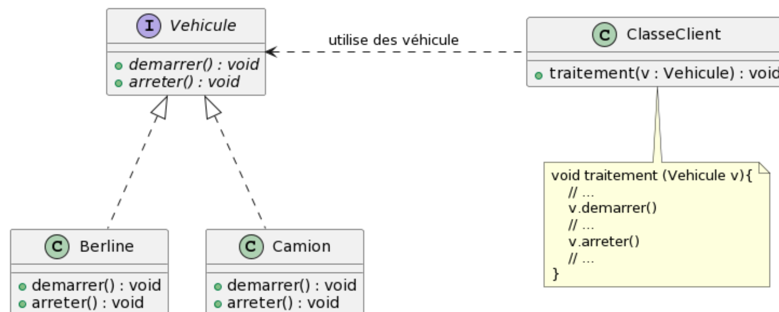
Exemple d'interface : `java.util.List`



- 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`

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



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

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`.

Mais 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 !

Héritage : les classe scellées

Depuis Java 17, il est possible de restreindre l'héritage qu'à certaines sous-classes en utilisant les *classes/interfaces scellées* :

```
// Déclaration d'une classe scellées
public sealed class Forme permits Cercle, Triangle, Rectangle {
    private double centreX;
    private double centreY;

    public Forme(double centreX, double centreY) {
        this.centreX = centreX;
        this.centreY = centreY;
    }
}
```

3 possibilités pour respecter la restriction de la classe-mère :

```
// on interdit tout héritage de Cercle
public final class Cercle extends Forme {
    // du code de Cercle ici
}
```

```
/* on définit Cercle comme une classe "ouverte" : toutes
   les sous-classes de Cercle seront acceptées */
public non-sealed class Cercle extends Forme {
    // du code de Cercle ici
}
```

```
 // on définit Cercle comme une classe scellées à son tour
public sealed class Cercle extends Forme permits JoliCercle, BeauCercle {
    // du code de Cercle ici
}
```

Pour plus de détails : https://openjdk.java.net/jeps/409

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...

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 ?

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 !**

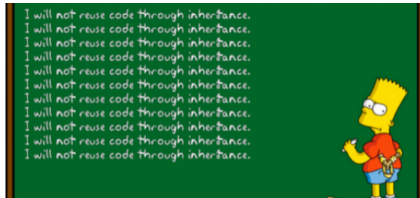
Héritage : bilan

`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 $\longrightarrow$ alternative plus souple
- si votre **seul but** c'est réutiliser du code, dites-vous bien



- ... et cherchez une alternative pour le réutiliser