Java et OOP
Introduction à l'héritage

Réutilisation du code

Comportements similaires à différents endroits du code

- C'est une des idées directrices de la programmation.
- Contourne la "redondance du code" (copier-coller), *fléau du developpement et de la maintenance* !
- Mécanismes fondamentaux de réutilisation du code :

1. **Passage des paramètres** : définition des actions paramétrés.
2. **Composition** : attributs d'un type classe dans des nouvelles classes

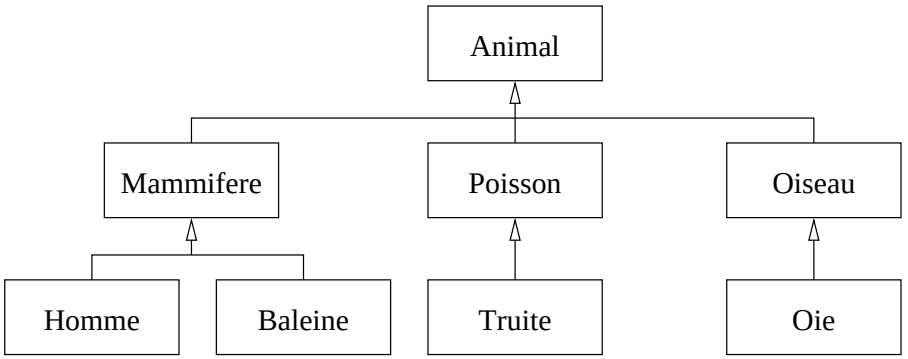
```
class Poly_Complexe{
    String nom;
    Complex[] coef;
    ...}
```

   Composition = relation **POSSEDE_UN** entre classes : un `Poly_Complexe` possède un attribut String et une table de Complex.
3. **Héritage** : *extension* des *fonctionnalités* d'une classe, *spécialisation* d'une classe.

Héritage : le principe

- Concepts de base / concepts dérivés :
  - Animal
  - Mammifère, Poisson, Oiseau.
    Chaque mammifère *est_un* oiseau !
  - Homme, Baleine, Truite, Oie, ... Chaque homme *est_un* mammifère, chaque truite *est_un* poisson !
- Certaines caractéristiques sont communes entre les classes :
  - Tout animal a une tête (attribut) et mange (méthode).
  - Tout poisson a une queue (attribut) et nage (méthode).
- Certaines caractéristiques sont spécifiques :
  - Tout homme parle, mais pas tout mammifère !
  - Tout oiseau possède des plumes, mais pas tout animal !

Héritage – hiérarchie de classes

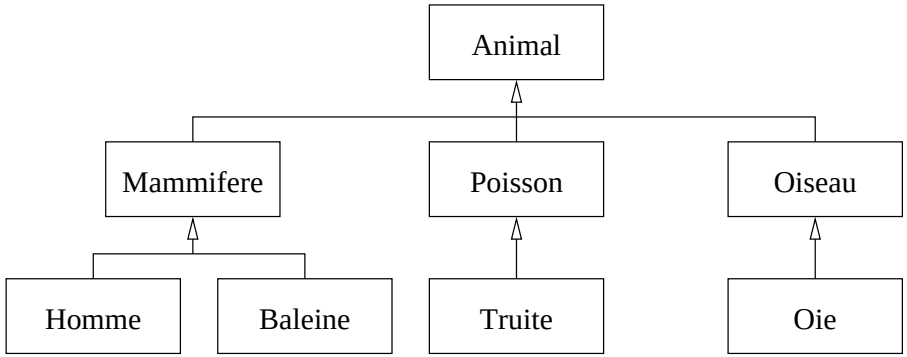


- L'héritage représente un lien de type **est_un** ou **est_comme_un**.
- Permet la spécialisation d'une nouvelle classe
  - **classe dérivéé** ou **sous-classe** ou **classe enfant**

  à partir d'une classe existente
  - **classe de base**, ou **super-classe**, ou **classe parent**.
- Réutilisation du code :
  - Sans toucher le code existant ;
  - En indiquant les nouvelles caractéristiques ;
  - En *redéfinissant* eventuellement certaines caractéristiques.
- Développement incrémental !

Héritage – syntaxe et définitions

```
class Animal{...}
class Mammifere extends Animal{...}
class Homme extends Mammifere{...}
class Possion extends Animal{...}
```

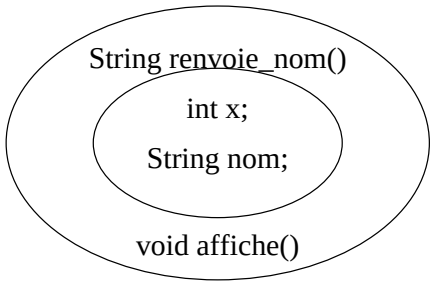


- `Homme` *est un descendant de* `Animal` – il existe un chemin ascendent de `Homme` a `Animal`.
- `Animal` *est un ancêtre de* `Homme`.
- On obtient une *hiérarchie* de classes (ou d'héritage).

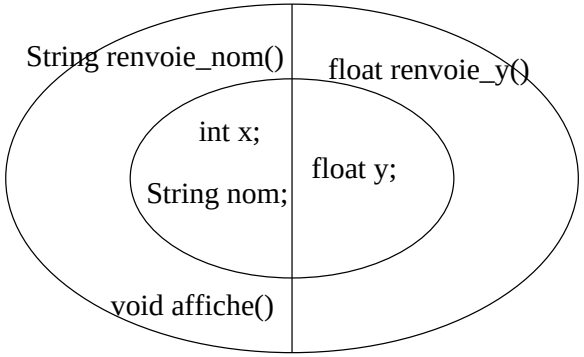
Relation entre classe de base et classe dérivée

```
class Base{
    int x;
    String nom;
    String renvoie_nom();
    void affiche();
}
```

```
classe Derivee extends Base{
    float y;
    float renvoie_y();
}
```



Classe "Base"



Classe "Derivee"

- Les attributs de la classe de base existent dans la classe dérivée : un objet de la classe dérivée "possède" aussi les attributs et les méthodes de la classe de base.
- Mais peut-on les utiliser/appeler ?

Règles de visibilité

- `public` : visible par tout, y compris pour les héritiers.
- `private` : invisible pour les héritiers aussi !
- implicite (dans le même paquetage) : visible.
- `protected` : visible pour les héritiers, protégé contre les autres !

```
class Base{
 int x;
 private String nom;
 public void affiche();
 protected void mod_nom(String s);
}
class Derivee extends Base{
 public fait_qq_chose(){
   x=2                    // OK
   nom = new String("?"); // ERR
   affiche();             // OK
   mon_nom("?");          // OK
 }
}
```

```
class UtiliseDerivee{
 public static void main(String[] a){
   Derivee nvobj = new Derivee();
   nvobj.affiche();      // OK
   nvobj.fait_qq_chose() // OK
   nvobj.x = 2           // OK !
   nvobj.mon_nom("??")   // ERREUR !
```

Possibilités d'héritage

- Etendre la définition de la classe de base en ajoutant des *attributs*, des *méthodes* et/ou des *constructeurs*.
- **Surcharger** les méthodes de la classe de base : même nom mais paramètres de types différents.
- **Redéfinir** les méthodes de la classe de base : même signature (nom, type de paramètres, type de retour) mais *comportement différent*.
- Si on veut *accéder* à la méthode qui a été redéfinie on fait appel à `super`.

```
class Personne{
 private String nom, prenom;
 public void test(){...}
 public void affiche(){
   System.out.println(nom + prenom);
 }
}
class Etudiant extends Personne{
 private String filiere;
 public void test(int x){...}
 public void affiche(){
   super.affiche();
   System.out.println(filiere);
 }
}
```

```
class UtiliseEtudiant{
 public static void main(){
   Personne p = new Personne();
   Etudiant e = new Etudiant();
   e.affiche();   // affiche la filiere
   p.affiche();   // pas de filiere...
   p.test();      // test() de Personne
   e.test();      // test() de Personne toujours !
   p.test(2);     // Erreur ! test n'est pas defini dans Personne
   e.test(2);     // test(int) de Etudiant
 }
}
```

Règles et restrictions

- La méthode redéfinie ne doit jamais être moins accessible que la méthode de la super-classe.
  Donc pas de redéfinition en `private` de méthode prevue `protected` dans la classe de base !
- Il n'existe pas de `super.super...` ! Donc pas d'accès aux ancêtres plus lointains que le seul parent !
- Une classe dérive d'une seule autre classe !
  Pas de pères/mères multiples, pas de `extends Cls1, Cls2`.

La classe `Object`

- Classe "primordiale", de laquelle **toutes** les classes dérivent (y compris les classes que vous avez vu déjà : `String, Integer,` etc.) :
  Même les classes pour lesquelles aucun ancêtre n'est précisé !
  (e.g. les classes `UtiliseDerivee, UtiliseEtudiant` ou les autres qui ne contiennent *que* le `main`)
- Pas d'attribut !
- Des méthodes bien utiles *qu'on peut redéfinir* :
  - `String toString()` – implicitement une description de l'objet (classe d'appartenence, code de hachage).
  - `boolean equals(Object)` – c'est la méthode à employer pour comparer les objets, mais il faut la **redéfinir** !
    Implicitement, elle fonctionne comme ==
  - `Object clone()` – prétend renvoyer une copie de l'objet *mais* ça n'arrive pas toujours !
    Exemple : elle ne copie pas les tableaux !
    Solution : il faut la redéfinir !

Transtypage

Comment peut-on utiliser `clone`, si elle prend un `Object` en paramètre ?

- La solution est le **transtypage implicite** que Java met à disposition ! C'est le même mécanisme employé dans `double x = 2`.
- **Upcasting** : *déclarer* une variable d'un type *classe de base*, mais l'*instancier* avec un objet d'une *classe dérivée*.
- Tout appel à une méthode redéfinie dans la classe dérivée sera exécuté sur cette méthode redéfinie !
- C'est une des **bases du polymorphisme !**
- Il est aussi possible de faire du **downcasting** :

On a déjà vu cela :

```
long x = 2;
int y = (int) x;
```

La même chose est possible pour les classes :

```
Etudiant e = new Etudiant();
Personne p = (Personne) e;
```

Constructeurs et héritage

- Un objet d'une classe dérivée "contient" un un "sous-objet" de sa super-classe (qui contient un sous-objet de sa classe de base qui...)
- Qu'est-ce qui se passe lors d'une création d'un objet de la classe dérivée avec son sous-objet de la classe de base ?

```
class Personne{
 Personne() {System.out.println("Constructeur de Personne");
}
class Etudiant extends Personne{
 Etudiant() {System.out.println("Constructeur de Etudiant");
}
class Et_en_DEUG extends Etudiant{
 Et_en_DEUG() {System.out.println("Constructeur de Et_en_DEUG");
}
```

- Les trois constructeurs seront appelés !
- La règle de base est que le "sous-objet" doit être construit avant de procéder à la construction des membres (attributs) spécifiques à la classe dérivée.

Appels de constructeurs

- Appel d'un constructeur pour la classe de base : mot-clé `super` :

```
class Personne{
 private String nom, prenom;
 Personne(String n, String p){
   nom = new String(n);
   prenom = new String(p);
 }
}
```

```
class Etudiant extends Personne{
 String filiere;
 Etudiant(String f){
   super(n,p);
   filiere = new String(f);
 }
}
```

Mais que se passe-t-il si on oublie d'appeler explicitement un constructeur ?

- S'il n'y a pas d'appel explicite `this` ou `super`, alors *le constructeur sans arguments de la classe de base sera appelé.*
  - Rappel : on peut appeler aussi un autre constructeur *de la même classe* avec la syntaxe `this(parametres)`.)
- Ces règles sont appliquées recursivement !

Bloquer la possibilité d'hériter

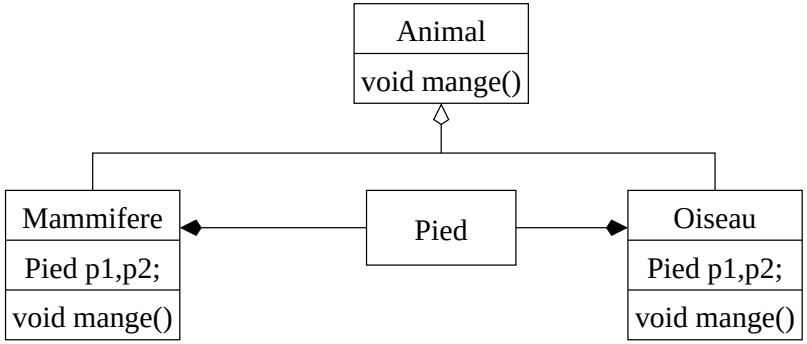
Mot-clé `final` :

- Une méthode déclarée `final` ne peut pas être redéfinie.
- Un attribut déclaré `final` ne peut pas être redéfini – attribut constant (on l'a déjà vu).
- Un objet déclaré `final` ne peut pas être realloué.
  - *mais ses attributs peuvent être modifiés*
  - c'est la référence qu'il engendre qui est constante (finale) !
- Une classe finale ne peut pas être dérivée.
  - Toutes ses méthodes sont alors finales.
  - `String` et `Vector` sont des classes finales.
  - Si vous voulez dériver d'une classe de chaînes de caractères, utilisez `StringBuffer`.

Héritage ou composition ?

- Ce n'est pas parce qu'on aime hériter qu'on peut utiliser l'héritage n'importe quand et n'importe ou !
- Se rappeler les rélations :
  - Est-ce que les objets du type à définir sont/ressemblent à ceux d'un type précédemment défini ?
    Reponse **est_un** – alors on utilise l'héritage.
  - Est-ce que les objets du type à définir ont des particularités d'un type précédemment défini ?
    Reponse **possède_un** – alors on utilise la composition.

Héritage ou composition ?



- Un mammifère **possède_un** pied (au moins un...).
- Un oiseau **possède_un** pied (pareil...).
- Mais ni un mammifère, ni un oiseau **n'est pas un** pied !
- Donc la classe `Pied` ne doit pas être classe de base pour les deux classes `Mammifère` ou `Oiseau`.
- Par contre, chacun **est_un** `Animal` !