

CONCEPTION LOGICIELLE L'HÉRITAGE



Présentée par: Mme HASSAM-OUARI Kahina

Email: kahina.hassam@junia.com

Bureau: T336

Département: SSE (Smart Systems & Energies)

JUNIA HEI

L'opérateur d'affectation =

- Affectation de références

- Soient refObj1 et refObj2 deux références d'objets, l'instruction `refObj2 = refObj1;` affecte le contenu de refObj1 dans refObj2. Donc les deux références référencent le même objet.

Exemple:

```
Etudiant refObj1= new Etudiant("Dupond", "Alex", "Gambetta");  
Etudiant refObj2;  
refObj2=refObj1;  
System.out.println(refObj2.getAdresse()); //ça affichera quoi ?
```



refObj1
refObj2

L'opérateur de comparaison ==

- **Egalité de références**

- L'expression `refObj1 == refObj2` vaut `true` si `refObj1` et `refObj2` **réfèrent le même objet** ou si elles valent toutes les deux `null`.

- **Egalité d'objets**

- Toute classe a par défaut une méthode nommée `equals`. Elle permet de tester l'égalité des contenus de deux objets d'une même classe.
- L'expression `refObj1.equals(refObj2)` vaut `true` ou `false` suivant que les deux objets référencés ont des contenus égaux ou non.

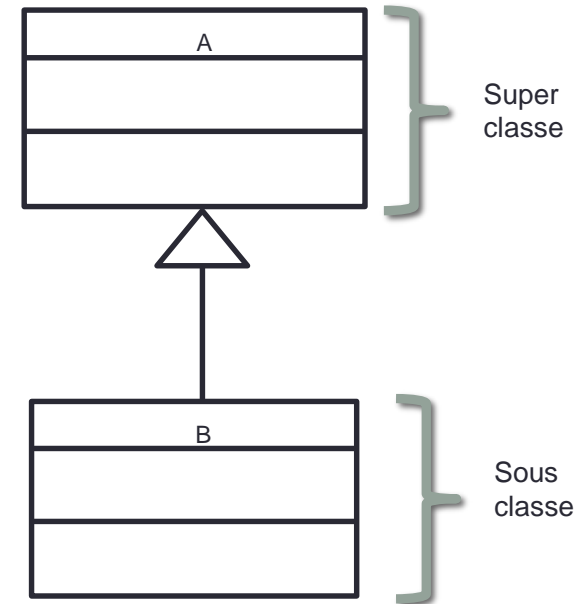
Héritage

Définition

- La relation d'héritage se définit entre 2 classes:
 - La super classe est nommée la classe mère
 - La ou les sous classes est/sont nommée(s)
 - On a une relation qui respecte le principe de « est une sorte de »

- **Modélisation UML:**

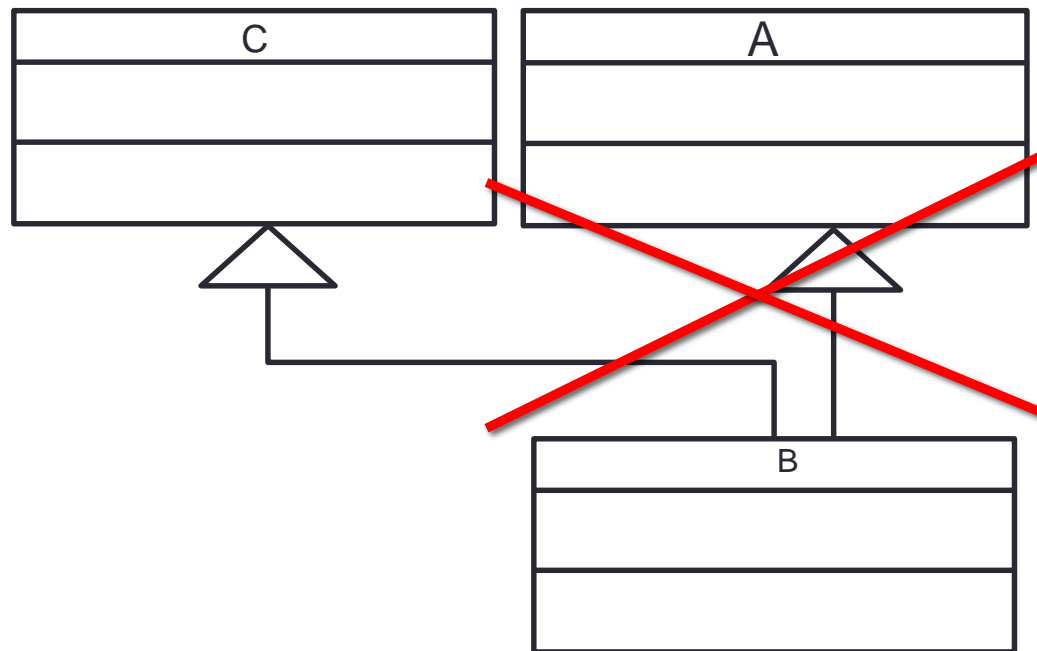
- La relation d'héritage est exprimée à l'aide d'une flèche vide
- B « est une sorte de » A
- A est la classe mère
- B est la classe fille



Héritage

En java: oui mais héritage simple

- Java permet l'héritage simple → Une classe ne peut hériter que d'une seule classe.



Héritage

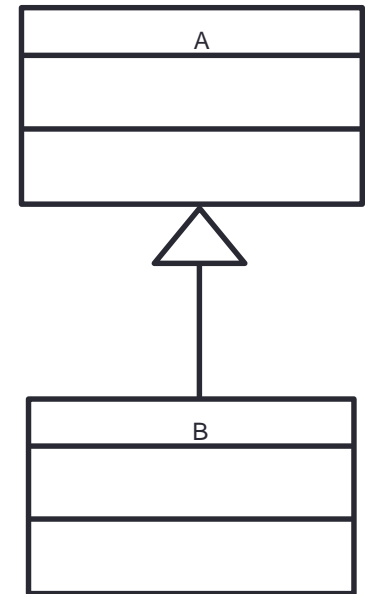
En java

```
public class A {  
  
    //Propriétés de A  
    //Méthodes de A  
  
}
```

A.java

```
public class B extends A  
{  
    //Propriété de B  
    //Méthodes de B  
}
```

B.java



Héritage de propriétés

Dans les faits..

- Quand une classe B hérite d'une classe A, les instances (objets) de la classe A jouissent à la fois des propriétés de la classe B et les propriétés de la A
- Rien n' empêche à la classe B d'avoir d'autres propriétés, **d'ailleurs c'est le but.**
- **Exemple avec une modélisation UML:**

Diagramme de classes

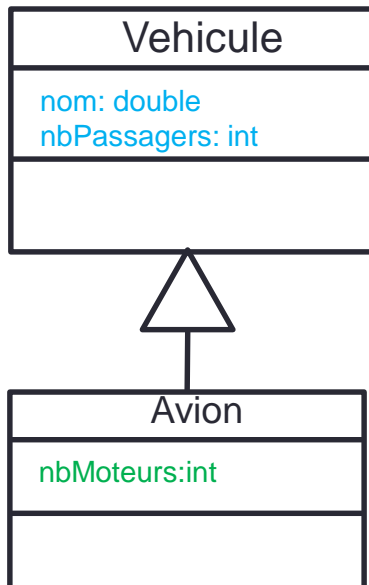
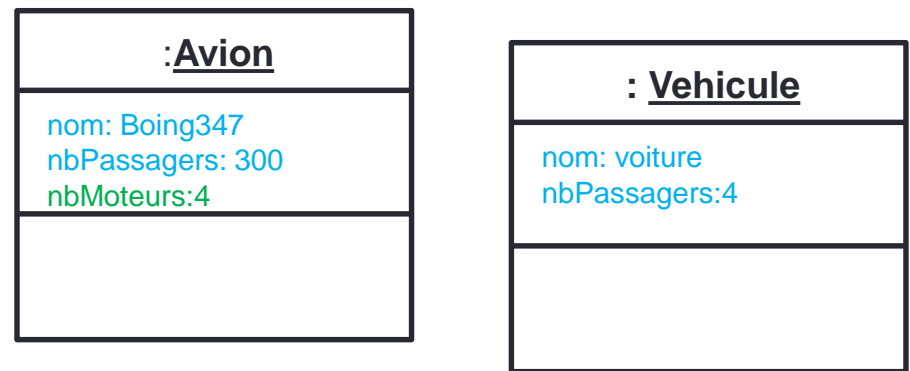


Diagramme d'objets



Tous les avions ont un nom et un nombre de passagers

Héritage de méthodes

- Quand B hérite de A, les instances de B savent réaliser toutes les méthodes de A (l'inverse est faux)
- Les objets de la classe B peuvent avoir leurs propres méthodes
- **Exemple avec une modélisation UML:**

Diagramme de classes

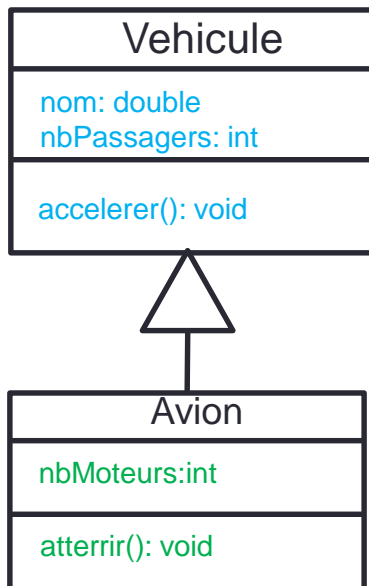
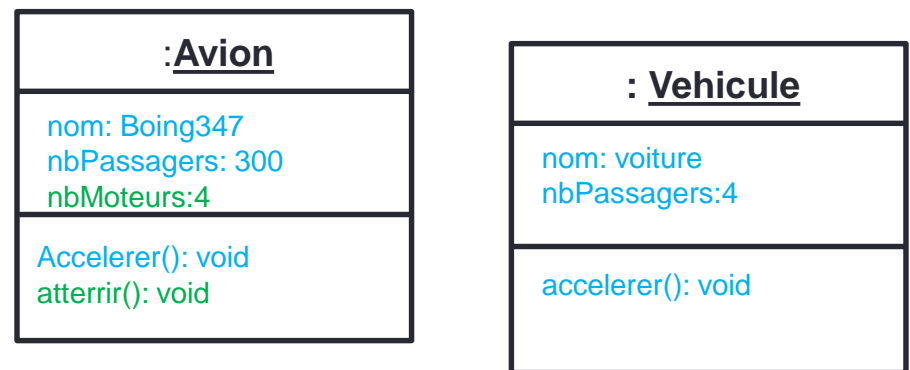


Diagramme d'objets



Tous les avions peuvent exécuter la méthode accélérer()

Héritage de méthodes

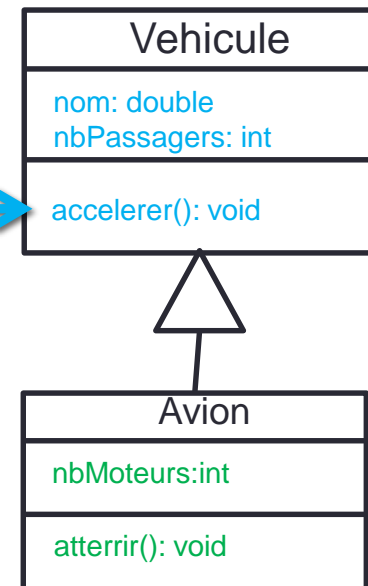
principe de redéfinition de méthodes

- Lorsqu'une classe fille appelle une méthode héritée, c'est la méthode de sa mère qui s'exécute:

```
Avion v= new Avion();  
v.accelerer();
```

L'objet **v** exécute la méthode `accelerer()` de `Vehicule`

Diagramme de classes



Héritage de méthodes

Polymorphisme

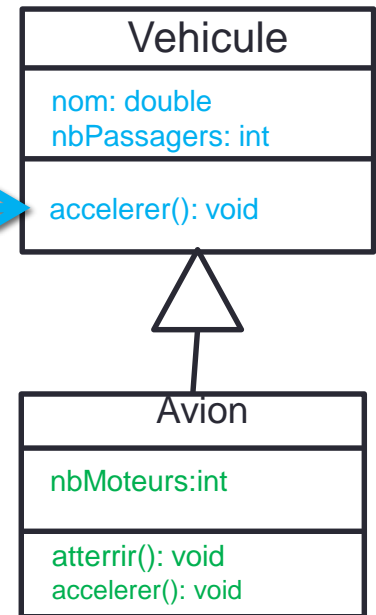
- ~~Lorsque une classe fille appelle une méthode héritée, c'est la méthode de sa mère qui s'exécute:~~

Diagramme de classes

```
Avion v= new Avion();  
v.accelerer();
```

L'objet v exécute la méthode
accelerer() de Vehicule

Sauf si la classe fille redéfinit la
méthode



Polymorphisme

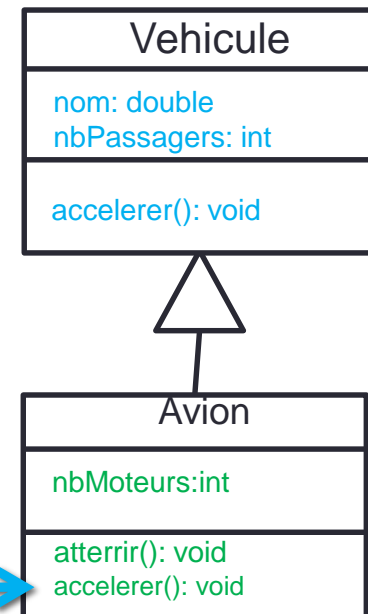
Héritage de méthodes

Polymorphisme

```
Avion v= new Avion();  
v.accelerer();
```

L'objet **v** exécute la méthode
définie par sa classe

Diagramme de classes



Héritage en Java:

Appel du constructeur de la super-classe

- La sous classe doit prendre en charge la construction de la super classe.
 - Pour construire un **Avion**, il faut d'abord construire un **Véhicule**;
- **Le constructeur de la classe de base (Vehicule) est donc appelé avant le constructeur de la classe dérivée (Avion).**
- Si un constructeur de la sous classe appelle **explicitement** un constructeur de la classe de base:
 - **cet appel doit être obligatoirement la première instruction de constructeur. Il doit utiliser pour cela, le mot clé**
super (paramètres de la super classe)

Héritage en Java: constructeur

```
package ExempleCoursHeritage;
```

```
public class Vehicule {
```

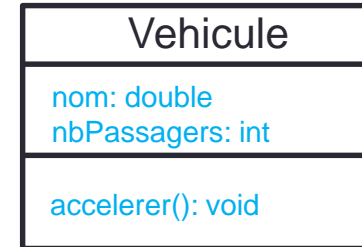
```
    private String nom;  
    private int nbPassagers;
```

```
    //Constructeur de la classe Vehicule
```

```
    public Vehicule(String nom, int nbPassagers){  
        this.nom=nom;  
        this.nbPassagers=nbPassagers;  
    }
```

```
    // Méthode de la classe Vehicule
```

```
    public void accelerer(){  
        //code la méthode  
    }  
}
```



Constructeur de la
classe «Vehicule»

Mot clé pour
exprimer l'instance courante
Utilisé indépendamment de
la relation d'héritage

Héritage en Java: constructeur

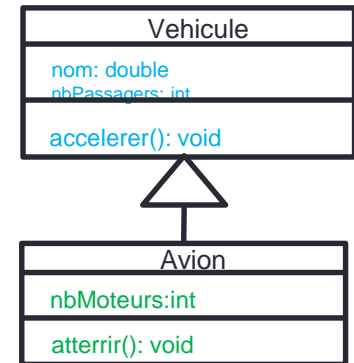
```
package ExempleCoursHeritage;

public class Vehicule {

    private String nom;
    private int nbPassagers;

    //Constructeur de la classe Vehicule
    public Vehicule(String nom, int nbPassagers){
        this.nom=nom;
        this.nbPassagers=nbPassagers;
    }

    // Méthode de la classe Vehicule
    public void accelerer(){
        //code la méthode
    }
}
```



```
package ExempleCoursHeritage;
```

```
public class Avion extends Vehicule{
    private int nbMoteurs;
    // constructeur de la classe Voiture
    public Voiture(String nom, int nbPassagers,int nbMoteurs )
    {
        super(nom, nbPassagers);
        this.nbMoteurs=nbMoteurs;
    }
}
```

Constructeur de la classe «Avion»

Appel du constructeur de la classe Vehicule.

```
//Methode de la classe Vehicule
public void atterrir(){
    //code de la methode
}
}
```

Héritage en Java: plusieurs constructeurs

```
package ExempleCoursHeritage;
```

```
public class Vehicule {
```

```
    private String nom;
```

```
    private int nbPassagers;
```

```
    //Constructeur de la classe Vehicule
```

```
    public Vehicule(String nom, int nbPassagers){  
        this.nom=nom;  
        this.nbPassagers=nbPassagers;  
    }
```

Constructeur 1

```
    public Vehicule(String nom){  
        this.nom=nom;  
    }
```

Constructeur 2

```
    // Méthode de la classe Vehicule
```

```
    public void accelerer(){  
        //code la methode  
    }  
}
```

On parle aussi de surcharges de méthodes

Héritage en Java:

Appel de constructeur

```
package ExempleCoursHeritage;
```

```
public class Avion extends Vehicule{  
    private int nbMoteurs;
```

```
// constructeur de la classe Voiture
```

```
public Avion(String nom, int nbPassagers, int nbMoteurs )  
{  
    super(nom, nbPassagers);  
    this.nbMoteurs=nbMoteurs;  
}
```

```
public Avion(String nom, int nbMoteurs ) {  
    super(nom);  
    this.nbMoteurs=nbMoteurs;  
}
```

```
//Methode de la classe Vehicule
```

```
public void atterrir(){  
    //code de la methode  
}  
}
```

On est pas obligé de tous les appeler mais il faut en appeler au moins UN

Appel du constructeur de la classe <<Vehicule>>.

Héritage en Java: Exemple

Vehicule.java

```
package ExempleCoursHeritage;

public class Vehicule {

    private String nom;
    private int nbPassagers;

    //Constructeur de la classe Vehicule
    public Vehicule(String nom, int nbPassagers){
        this.nom=nom;
        this.nbPassagers=nbPassagers;
    }

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

    // Méthode de la classe Vehicule
    public void accelerer(){
        System.out.println("méthode accelerer de la classe
        Vehicule");
    }
}
```

Avion.java

```
package ExempleCoursHeritage;

public class Avion extends Vehicule{
    private int nbMoteurs;

    // constructeur de la classe Voiture
    public Avion(String nom, int nbPassagers,int nbMoteurs ) {
        super(nom, nbPassagers);
        this.nbMoteurs=nbMoteurs;
    }

    public Avion(String nom, int nbMoteurs ) {
        super(nom);
        this.nbMoteurs=nbMoteurs;
    }

    //Methode de la classe Vehicule
    public void atterrir(){
        System.out.println("méthode atterrir de la classe Avion");
    }

    public void accelerer(){
        System.out.println("méthode accelerer de la classe Avion");
    }
}
```

MonMain.java

```
package ExempleCoursHeritage;

public class MonMain {

    public static void main(String args[]){

        Vehicule v1= new Vehicule("Vehicule1");
        Vehicule v2= new Vehicule("Le vehicule 2", 2);
        Avion a1 =new Avion("Boing 335", 500, 4);
        Avion a2 =new Avion("Boing 512", 2);

        v1.accelerer();
        a1.accelerer();
        a2.atterrir();

    }}
}
```

Héritage

Redéfinition de méthodes en résumé

- La redéfinition intervient lorsqu'une classe fille fournit une nouvelle définition d'une méthode d'une classe Mère.
- Cette nouvelle méthode doit posséder:
 1. Le même nom que la méthode de la classe Mère,
 2. La même signature (même nombre d'arguments et même types)
 3. le même type de valeur de retour.



Ne pas confondre avec la SURCHARGE
de méthode

La surcharge de méthode

- La notion de surcharge de méthode n'est pas forcément liée à l'héritage
- On parle de surcharge, lorsque 2 méthodes d'une même classe ou héritées d'une autre classe ont:
 1. Le même nom,
 2. Pas les mêmes paramètres en types et/ou en nombres Et/ou le type de retour

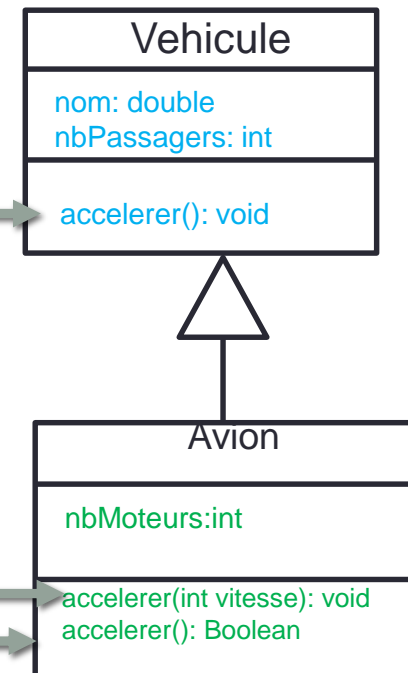
- Exemple de surcharge de méthode:

```
Avion v= new Avion();
```

```
v.accelerer();
```

```
v.accelerer(50);
```

```
Boolean acce=v.accelerer();
```



Quand utiliser l'héritage

- **Factorisation du code:** vos classes ont des méthodes et des propriétés communes
- **Réutilisation:** une classe existante
- **Imposer un cadre:** vos classes proposent un noyau qui doit être complété

Factorisation du code

Comment?

- Deux classes qui ont en commun des méthodes et/ou des propriétés
- Construire une nouvelle classe qui sera LA super classe.
- Y déplacer les méthodes et/ou propriétés communes
- Ajouter le lien de l'héritage.

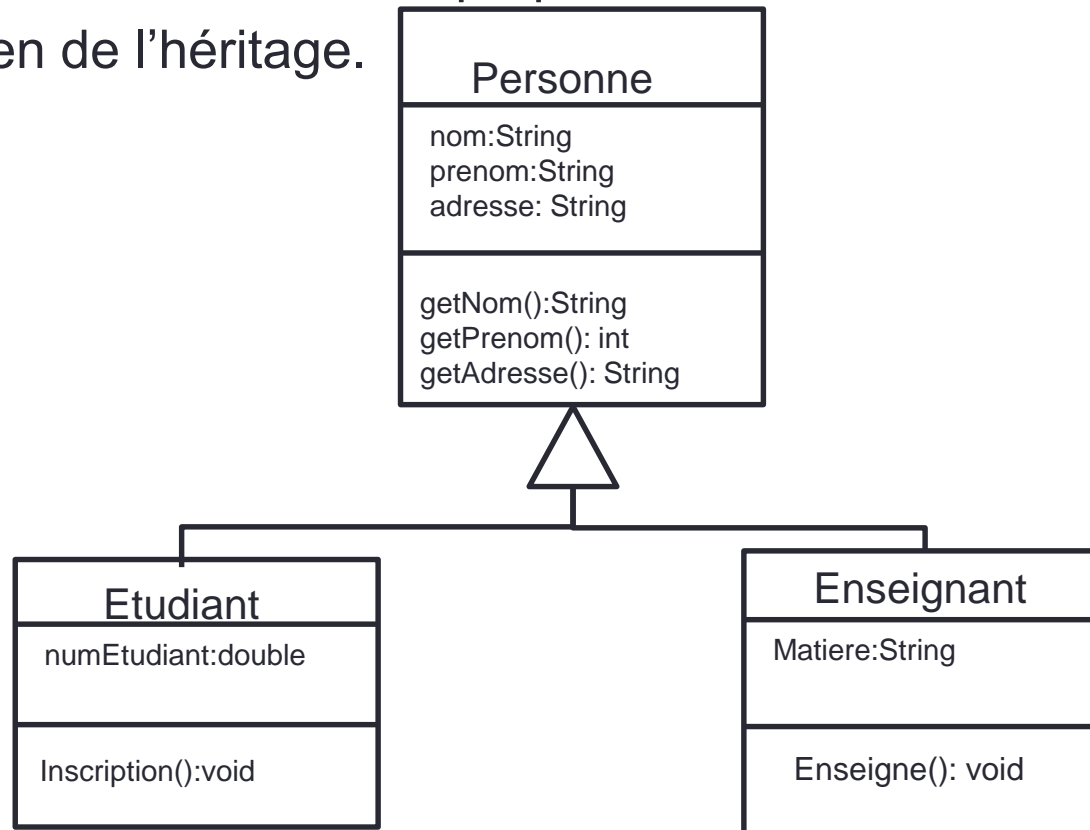
Etudiant
nom:String prenom:String adresse: String numEtudiant:double
getNom():String getPrenom(): int getAdresse(): String Inscription():void

Enseignant
nom:String prenom:String adresse: String Matiere:String
getNom():String getPrenom(): int getAdresse(): String Enseigne(): void

Factorisation du code

Comment?

- Deux classes qui ont en commun des méthodes et/ou des propriétés
- Construire une nouvelle classe qui sera LA super classe.
- Y déplacer les méthodes et/ou propriétés communes
- Ajouter le lien de l'héritage.



Réutilisation

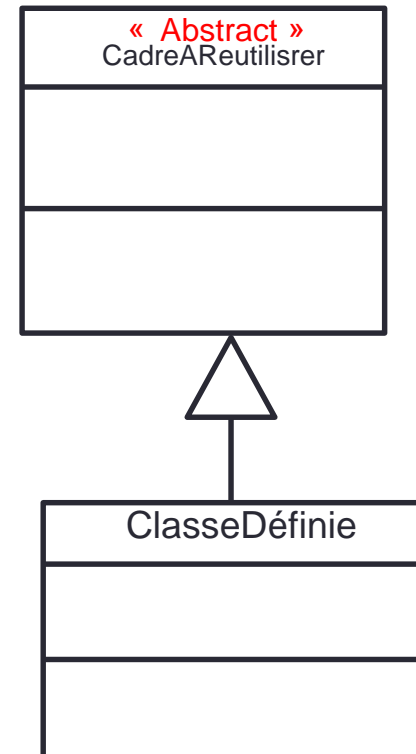
- Avant de coder une nouvelle classe, on peut chercher une classe existante
- Hériter de cette classe
- Y ajouter des méthodes et des propriétés
- Exploiter les mécanismes de polymorphisme pour adapter le comportement

Imposer un cadre

- Mettre à disposition du code pour qu'il soit réutilisé et complété
- Création d'une classe qui contient le code qui va être réutilisé en utilisant l'héritage
- La vocation de la classe initiale est d'être seulement héritée



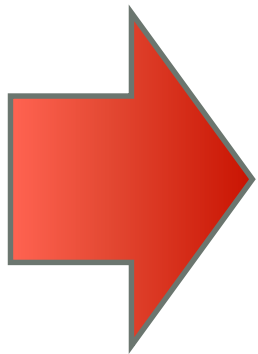
Déclarer la classe « Abstract »



Classe abstraite

Définition

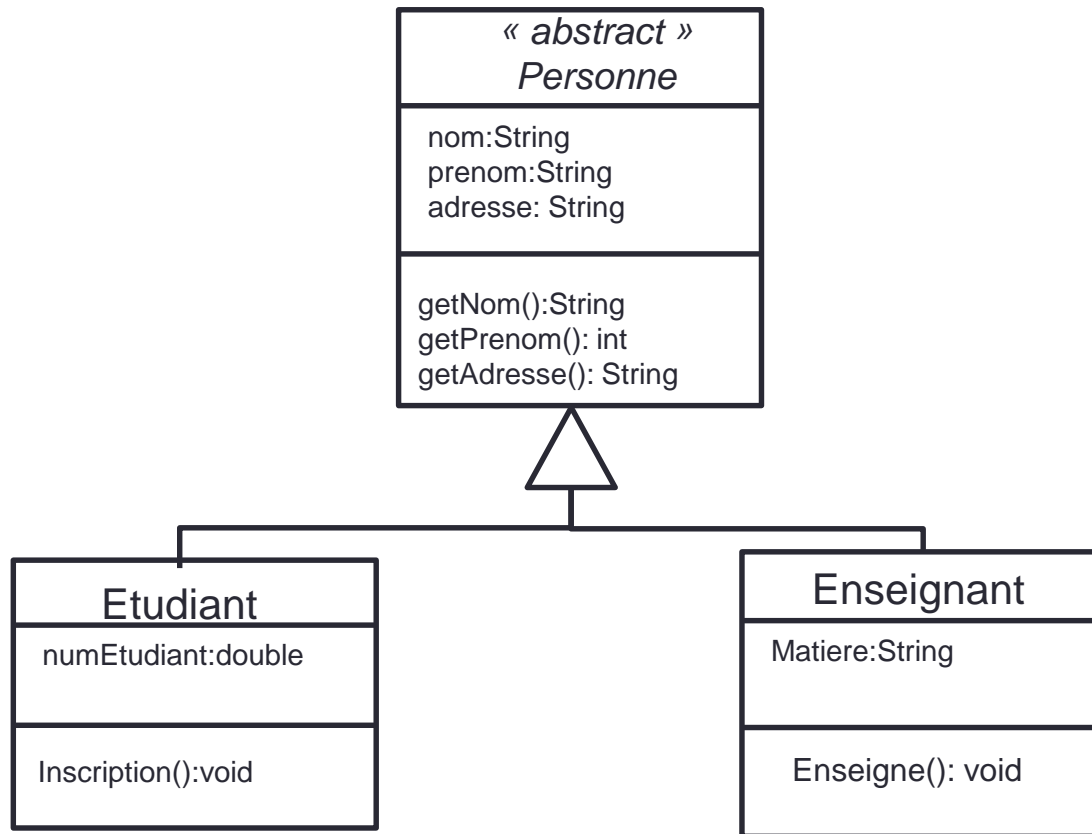
- Une classe abstraite est une classe créée pour que les autres classes puissent hériter de ses propriétés et méthodes
- MAIS cette classe ne peut pas être instanciée



Pas d'instances de cette classe dans l'application objets.

Classe abstraite

Exemple d'une école



Classe abstraite

Exemple d'une école en java

```
package Cours1Exemple;

public abstract class Personne {
```

```
    private String nom;
    private String prenom;
    private String adresse;
```

```
    Personne(String nomEtud, String prenomEtu, String adrEtud){
        this.nom=nomEtud;
        this.prenom=prenomEtu;
        this.adresse=adrEtud;
    }
```

```
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public String getAdresse() {
        return adresse;
    }
}
```

```
package Cours1Exemple;

public class MainPrg {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Personne p1=new Personne(" Van Damme",
        "Jean-Claude", "Hollywood");

        Etudiant e1=new Etudiant("Dupont",
        "ALex", "rue de toul", 256);

        Enseignant prof1=new Enseignant("Toto",
        "titi", "HEI", "Algo");

    }

}
```

```
package Cours1Exemple;

public class Etudiant extends Personne {

    private double numEtudiant;

    public Etudiant(String nomEtud, String prenomEtu,
    String adrEtud, double num) {
        super(nomEtud, prenomEtu, adrEtud);
        this.numEtudiant=num;
    }

    public void inscription(){
        //code de la methode inscription
    }
}
```

```
package Cours1Exemple;

public class Enseignant extends Personne {

    private String matiere;

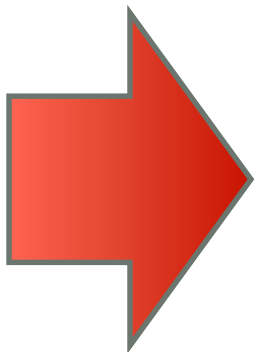
    public Enseignant(String nom, String prenom,
    String adr,String mat){
        super(nomE, prenom, adr);
        this.matiere=mat;
    }

    public void Enseigne(){
        // code de la methode enseigner
    }
}
```

Classe abstraite

Méthode abstraite

- Une classe abstraite peut contenir:
 - Des méthodes concrètes
 - Des méthodes abstraites
- Deux points importants :
 - Une méthode abstraite n'a pas de corps !
 - `public abstract typeRetour nomMethode(typesNomsParams);`
 - Une méthode abstraite **est toujours contenue** dans une classe abstraite.



**une méthode abstraite doit
obligatoirement être redéfinie dans les
sous-classes plus spécifiques.**

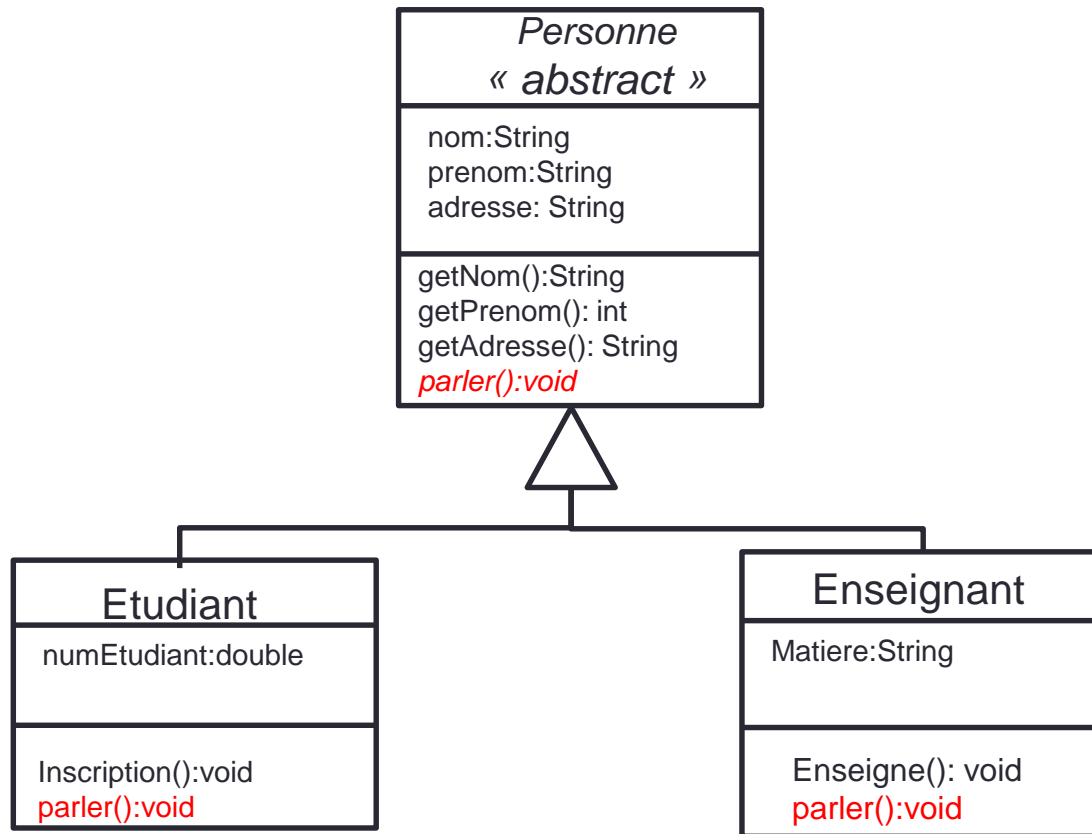
Classe abstraite

Méthode abstraite

- Implémenter une méthode abstraite revient à redéfinir cette méthode.
- Les méthodes abstraites n'ont pas de corps. Elles ne servent qu'à mettre en œuvre le polymorphisme.
- Une méthode abstraite peut être implémentée dans une sous-classe abstraite.
- La première classe concrète dans votre hiérarchie d'héritage doit implémenter toutes les méthodes abstraites qui ne l'ont pas encore été.

Classe abstraite

Exemple d'une école



Classe abstraite

Exemple d'une école en java

```
package Cours1Exemple;

public abstract class Personne {

    private String nom;
    private String preom;
    private String adresse;

    Personne(String nomEtu, String prenomEtu, String adrEtu){
        nom=nomEtu;
        prenom=prenomEtu;
        adresse=adrEtu;
    }

    public abstract void parler();

    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
    public String getAdresse() {

        return adresse;
    }
}
```

Déclaration de la
méthode abstraite

Redéfinition
de la méthode
parler()

```
package Cours1Exemple;

public class Etudiant extends Personne {

    private double numEtudiant;
    public Etudiant(String nomEtu, String prenomEtu,
String adrEtu, double num) {
        super(nomEtu, prenomEtu, adrEtu);
        this.numEtudiant=num;
    }

    public void inscription(){
        //code de la methode inscription
    }

    public void parler() {
        System.out.println("je parle avec mon
camarade pendant les cours :)");
    }
}
```

```
package Cours1Exemple;

public class Enseignant extends Personne {

    private String matiere;

    public Enseignant(String nomEtu, String
prenomEtu, String adrEtu,String mat){
        super(nomEtu, prenomEtu, adrEtu);
        this.matiere=mat;
    }

    public void Enseigne(){
        // code de la methode enseigner
    }

    public void parler() {
        System.out.println("Je présente le cours");
    }
}
```



Classe abstraite

Exemple d'une école en java

```
package Cours1Exemple;

public class MainPrg {

    public static void main(String[] args) {
        Etudiant e1= new Etudiant("Dupont", "ALex", "rue de toul", 256);
        Enseignant prof1=new Enseignant("Toto", "titi", "HEI", "Algo");
        e1.parler();
        prof1.parler();
    }
}
```

 Console 

```
<terminated> MainPrg [Java Application] C:\Program Files\Java\jre7\
je parle avec mon camarade pendant les cours :))
Je présente le cours
```


Modificateurs d'accès

- En Java, la déclaration d'une classe, d'une méthode ou d'un membre peut être précédée par **un modificateur d'accès**.
- Un modificateur indique si les autres classes de l'application pourront accéder ou non à la classe/méthode/propriété.
- Chaque classe ou membre (attribut ou méthode) est précédé par un modificateur d'accès **private** ou **public, protected, package**
 1. **private** veut dire que le membre est encapsulé, inaccessible de l'extérieur de la classe
 2. **public** veut dire que le membre fait partie de l'interface, accessible de l'extérieur
 3. **protected** veut dire que les membres sont **visible par les classes du même package** et **par ses sous classes** (même celles se trouvant dans des packages différents)
 4. **package** veut dire que les membres d'une classe sont visible par les classes du même package

Package1

ClasseA

-privateProp
~packageProp
#protectedProp
+publicProp



Classe B

ClasseC

Package2

ClasseD

ClasseE

Modificateurs d'accès
La visibilité en résumé

	ClasseA	ClasseB	ClasseC	ClasseD	ClasseE
privateProp	V				
packageProp	V	V	V		
protectedProp	V	V	V	V	
publicProp	V	V	V	V	V

Le rôle des modificateurs d'accès

- **Préservation de la sécurité des données**

- Les données privées sont simplement **inaccessibles** de l'extérieur
- Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques

- **Préservation de l'intégrité des données**

- La modification directe de la valeur d'une variable privée étant impossible,
- seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place le mécanisme d'encapsulation.

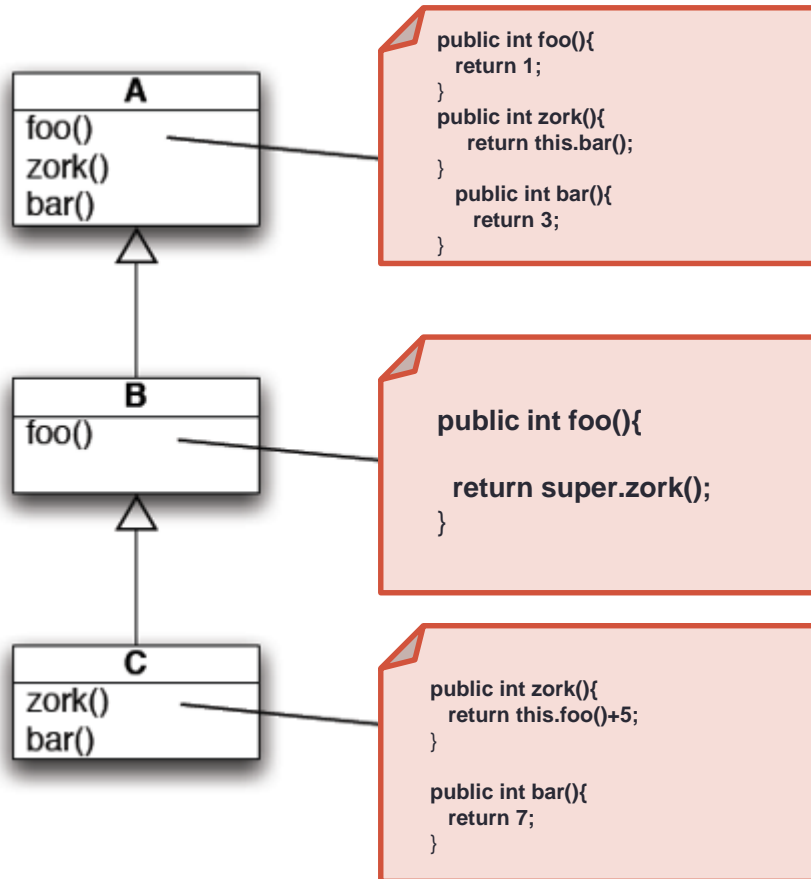
- **Cohérence des systèmes développés en équipes**

- Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de connaître la signature. Leur code est donc indépendant de l'implémentation des méthodes

Exercice de cours:

Héritage

Etant donné le diagramme suivant:



que renvoient les 6 instructions ci-dessous ?

- **A a = new A();**
int val1= a.zork();
- **B b = new B();**
int val2=b.zork();
- **C c = new C();**
int val3 =c.zork();
- **B bc = new C();**
int val4=bc.zork();
- **A ac = new C();**
int val5= ac.zork();
- **A ab = new B();**
int val6= ab.zork();

Méthodes et propriétés static

Définition

- **Propriété static:**

Une propriété static appartient à la classe et non aux objets. Sa valeurs peut être changée par n'importe quel objets qui la modifie

- **Méthode static**

- ✓ Une méthode static peut s'exécuter sans instancier d'objets.
- ✓ Dans une méthode static, **on ne fait appel qu'aux propriétés static de la classe**

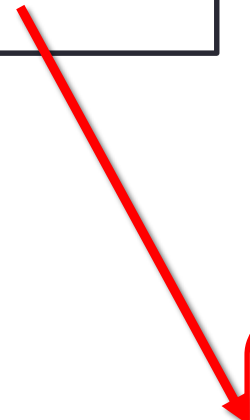
Exemple:

Propriété et méthode déclarées Static

Il existe UNE
cagnotte



Impot
-Static cagnotte: double
+getCagnotte: double +static afficheCagnotte():void



```
public class Impot {
    private static double cagnotte;
```

```
    public Impot(){
        this.cagnotte=0
    }
```

```
    public void encaisser(double somme) {
        Impot.cagnotte+=somme;
    }
```

```
    public double getCagnotte() {
        return cagnotte;
```

```
    }
    public static void afficheCagnotte(){
        System.out.println("Le montant des Impots
récoltés est de "+ Impot.cagnotte);
    }
```

Exemple: Pg Principal

```
public class MainProgram {  
    public static void main(String[]  
        args) {
```

```
        Impot taxeHabitation= new Impot();  
        Impot impotRevenus= new Impot();
```

```
        Impot.afficheCagnotte();
```

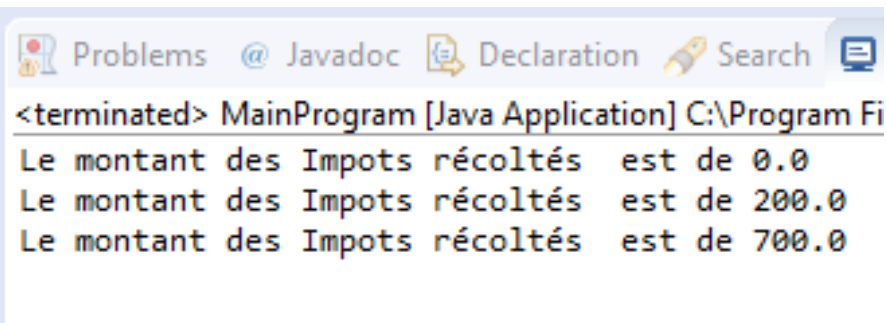
```
        taxeHabitation.encaisser(200);
```

```
        Impot.afficheCagnotte();
```

```
        impotRevenus.encaisser(500);
```

```
        Impot.afficheCagnotte();
```

```
    }}
```



Méthodes et propriétés final

Définition

- **Une propriété finale:** Représente une constante dont la valeur ne varie jamais. L'affectation doit être effectuée, **au plus tard**, dans le constructeur de la class.
 - Exemple: `public final double PI=3.14;`
- **Une Classe finale:** est une classe dont on ne peut pas hériter (correspond à la dernière feuille d'une arborescence)
 - Exemple: `public final class Feuille{}`
- **Une méthode finale:**

Une méthode déclarée finale ne peut pas être redéfinie dans les sous classes (Imposer un autre comportement dans les sous classes)

Exemple: `public final TypeDeRetour nomMethode(parametres){}`