JAVA – POO

Concepts clés de la POO en Java :

- **Encapsulation**
- **Héritage**
- **Polymorphisme**
- **Abstraction**

Encapsulation

Le principe d'encapsulation dit qu'un objet ne doit pas exposer sa **représentation interne** au **monde extérieur**.

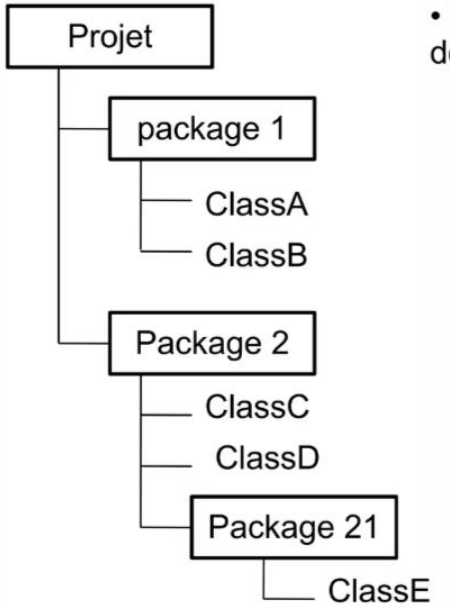
représentation interne d'un objet:

- Les attributs
- Les méthodes

monde extérieur

- Les autres classes dans la même package
- Les classes des autres packages

Encapsulation



- **Package = répertoire**.
- Les classes Java peuvent être regroupées dans des packages.

Déclaration d'un package

```
package pack1;
```

Utilisation d'un package

1- Nom du package suivi de nom de la classe

```
java.util.Date now = new java.util.Date() ;
System.out.println(now) ;
```

2- Import de la classe du package

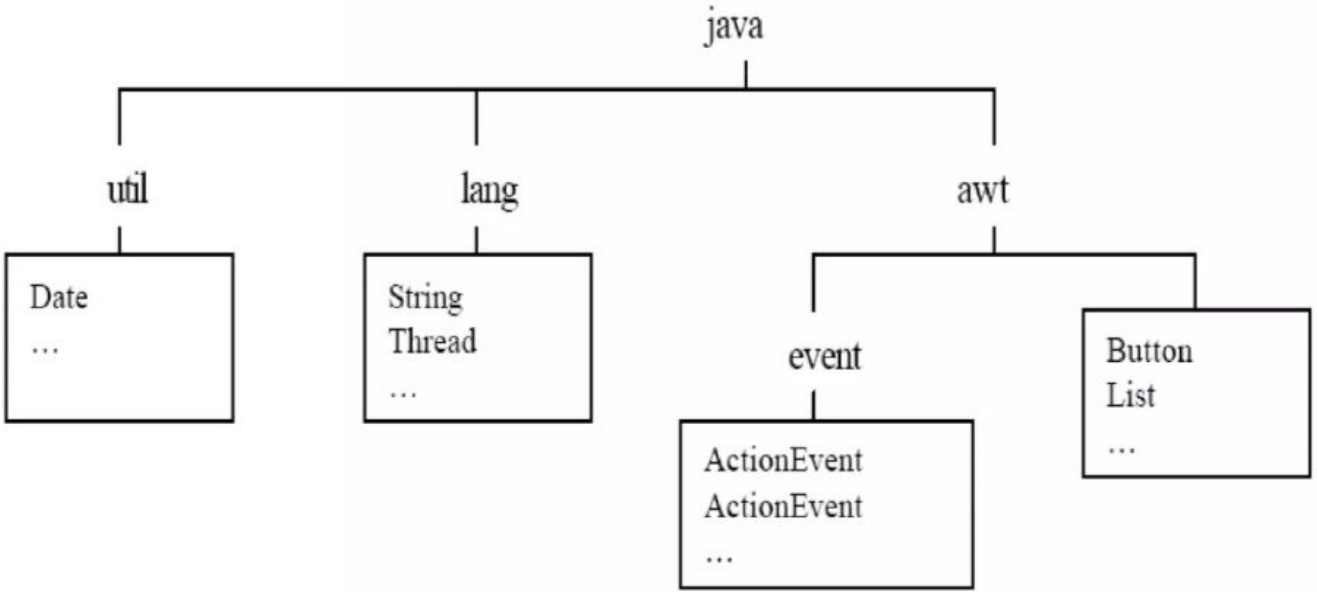
```
import java.util.Date ;
...
Date now = new Date() ;
System.out.println(now) ;
```

2- Import de tout le package

```
import java.util.*;
...
Date now = new Date() ;
System.out.println(now) ;
```

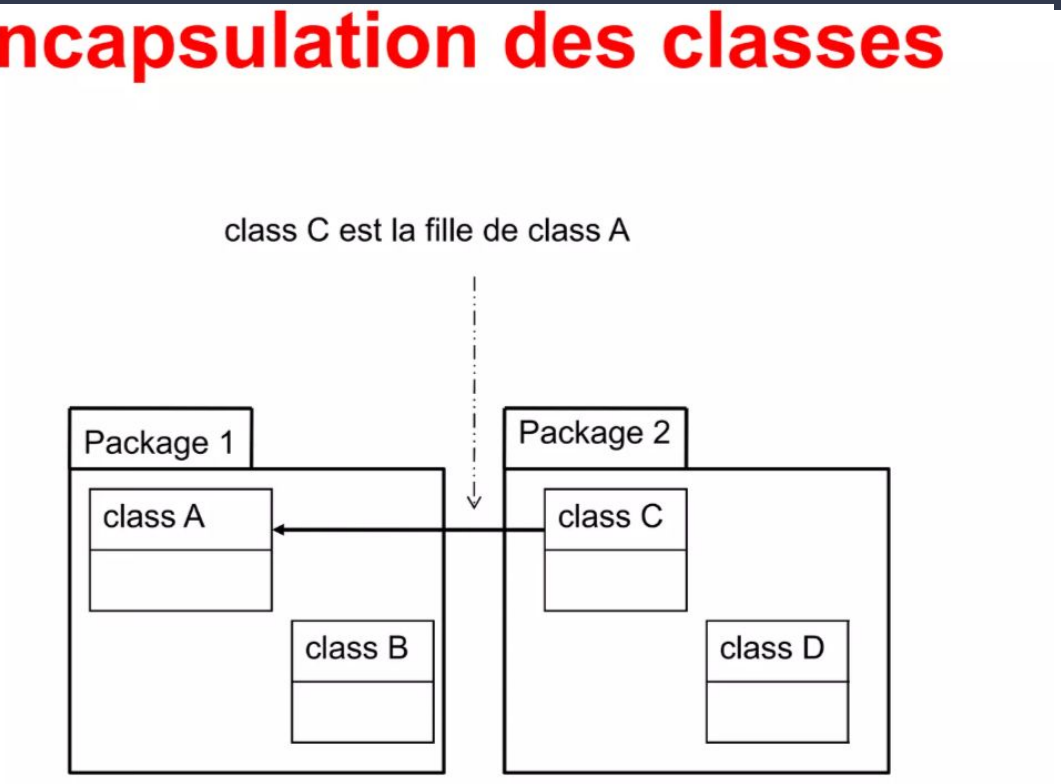
Encapsulation

- Les packages sont eux-mêmes organisés hiérarchiquement. : on peut définir des sous-packages, des sous-sous-packages, …



Encapsulation

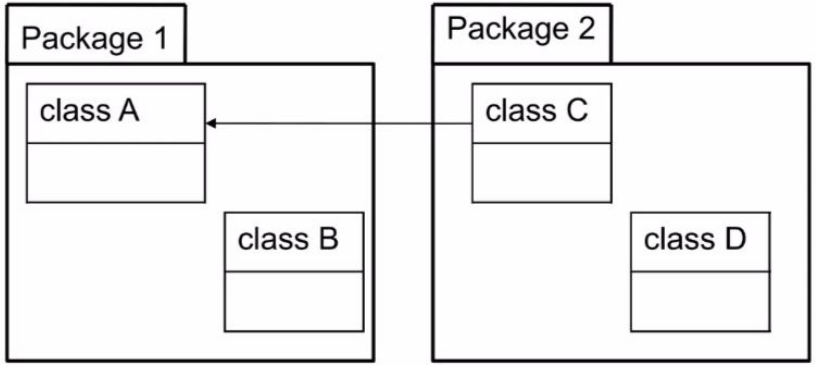
Encapsulation des classes



Encapsulation

Encapsulation des classes

La classe public



public class A ➡ La classe *public* est visible depuis n'importe quelle classe du projet

```
public class A {
…
}
```

```
class B {
    A a;
…
}
```

✓

```
class C  extends A{
    A a;
…
}
```

✓

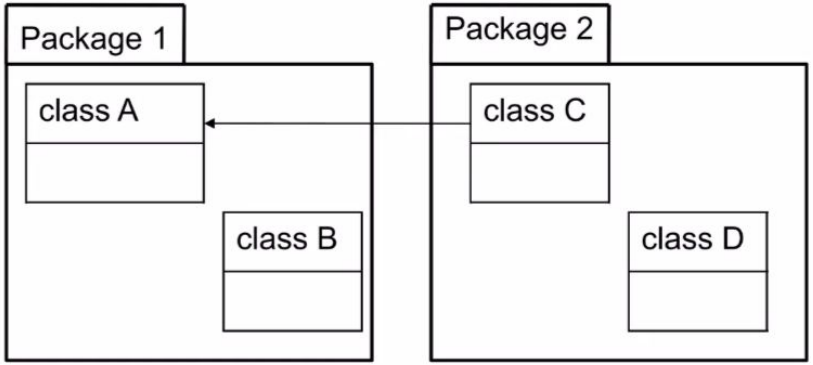
```
class D {
    A a;
…
}
```

✓

Encapsulation

Encapsulation des classes

La classe default



class A ➡ La classe *default* est visible seulement par les classes de son package.

```
class A {
…
}
```

```
class B {
    A a;
…
}
```

✓

```
class C extends A {
    A a;
…
}
```

✗

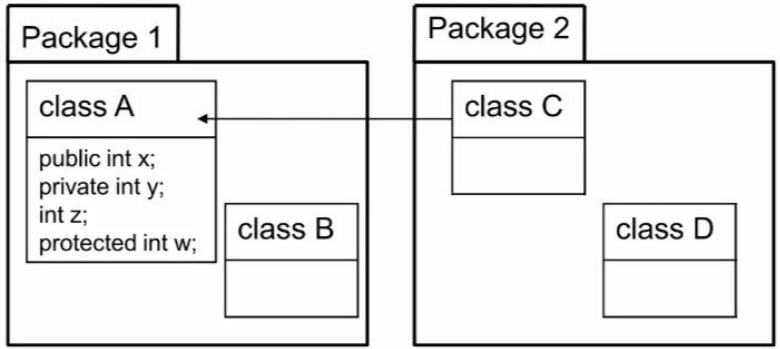
```
class D {
    A a;
…
}
```

✗

Encapsulation

Encapsulation des attributs

L'attribut public



public int x ➡ L'attribut *public* est visible par toutes les classes

```
public class A {
   public int x;
…
}
```

```
class B {
   A a=new A();
   int t=a. x ;
…
}
```

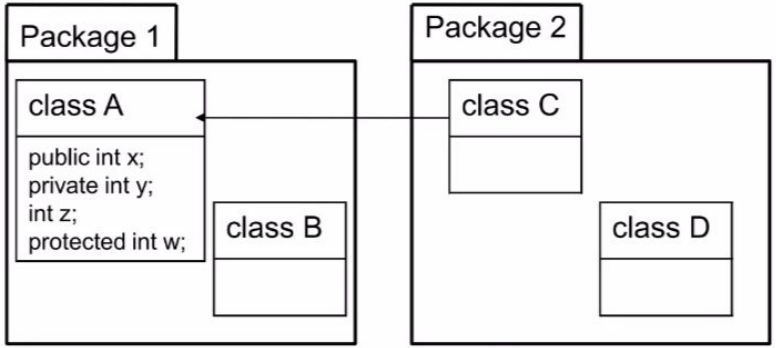
```
class C extends A {
   A a=new A();
   int t=a. x ;
…
}
```

```
class D {
   A a=new A();
   int t=a. x ;
…
}
```

Encapsulation

Encapsulation des attributs

L'attribut private



private int y ➡ L'attribut *private* n'est accessible que depuis l'intérieur même de la classe.

```
public class A {
   private int y;
…
}
```

```
class B {
   A a=new A();
   int t=a. y ;
…
}
```

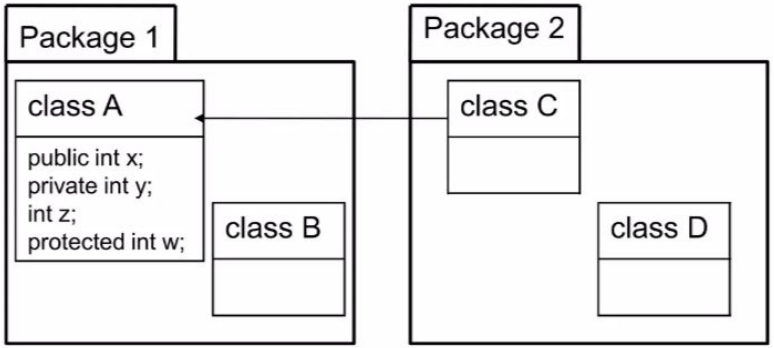
```
class C extends A {
   A a=new A();
   int t=a. y ;
…
}
```

```
class D {
   A a=new A();
   int t=a. y ;
…
}
```

Encapsulation

Encapsulation des attributs

L'attribut default



int z ➡ L'attribut *default* n'est accessible que depuis les classes faisant partie du même *package*.

```
public class A {
int z;
…
}
```

```
class B {
  A a=new A();
  int t=a. z ;
…
}
```

✔

```
class C extends A {
  A a=new A();
  int t=a. z ;
…
}
```

✖

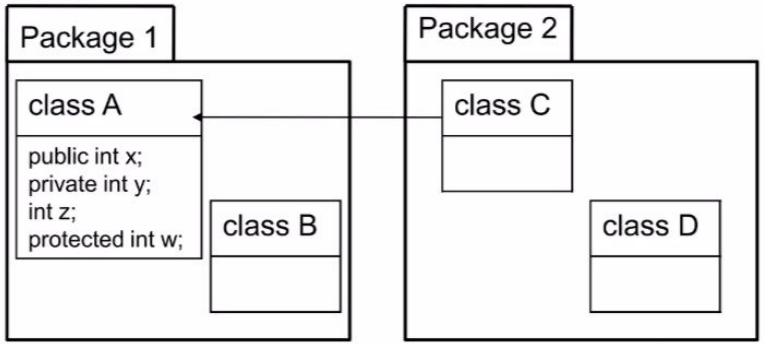
```
class D {
  A a=new A();
  int t=a. z ;
…
}
```

✖

Encapsulation

Encapsulation des attributs

L'attribut protected



protected int w ➡ L'attribut *protected* est accessible uniquement aux classes d'un même package et à ses classes filles (même si elles sont définies dans un package différent.)

```
public class A {
   protected int w;
…
}
```

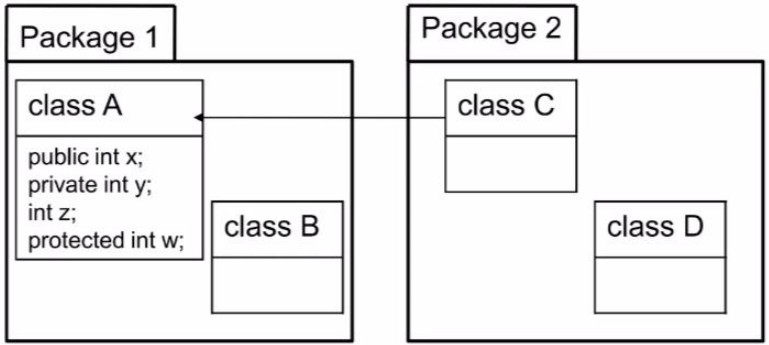
```
class B {
   A a=new A();
   int t=a. w ;
…
}
```

```
class C extends A {
   A a=new A();
   int t=a.w;
…
}
```

```
class D {
   A a=new A();
   int t=a.w;
…
}
```

Encapsulation

Encapsulation des attributs

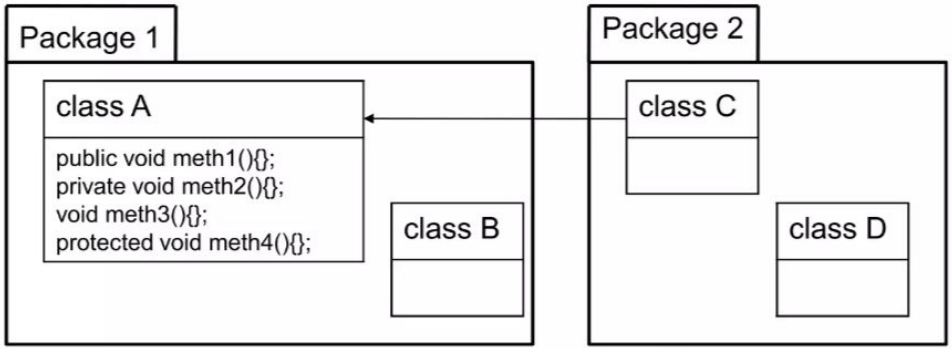


Visibilité à partir de:	Classe A	Package B	Classe fille C	Tout le reste D
public (x)				
private (y)				
default (z)				
protected (w)				

Encapsulation

Encapsulation des méthodes

La méthode public



public void meth1() ➡ La méthode *public* est accessible par toutes les classes du projet.

```
public class A {
   public void meth1(){}
…
}
```

```
class B {
   A a=new A();
   a.meth1() ;
…
}
```

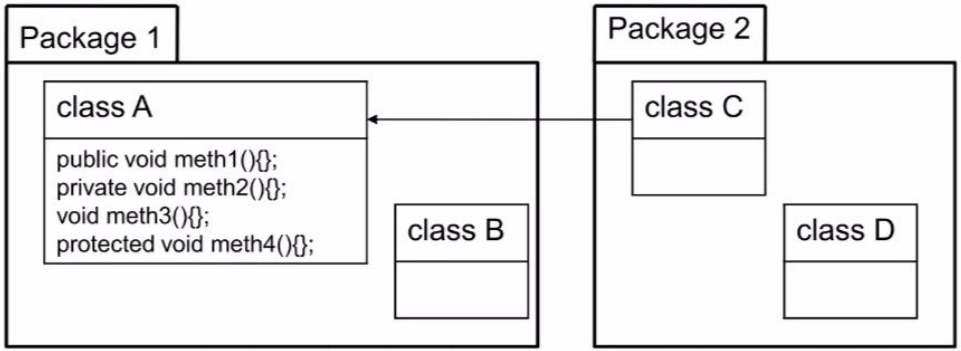
```
class C extends A {
   A a=new A();
   a.meth1() ;
…
}
```

```
class D {
   A a=new A();
   a.meth1();
…
}
```

Encapsulation

Encapsulation des méthodes

La méthode private



private void meth2() ➡ La méthode *private* n'est accessible que depuis l'intérieur même de la classe.

```
public class A {
   private void meth2(){}
…
}
```

```
class B {
   A a=new A();
   a.meth2() ;
…
}
```

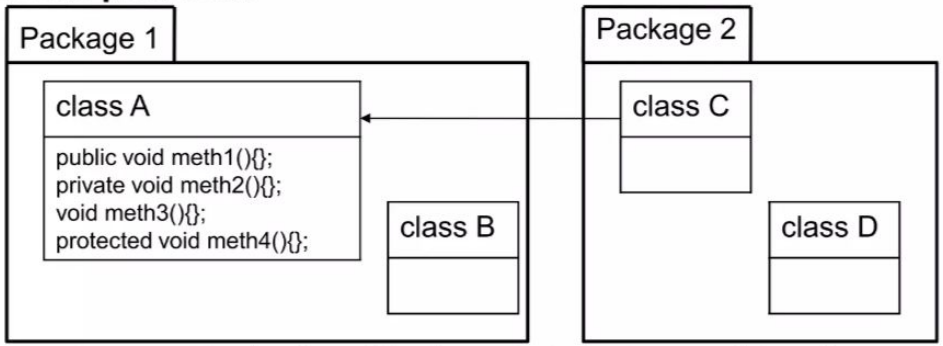
```
class C extends A {
   A a=new A();
   a.meth2() ;
…
}
```

```
class D {
   A a=new A();
   a.meth2();
…
}
```

Encapsulation

Encapsulation des méthodes

La méthode protected



protected void meth4() ➡ La méthode *protected* est accessible uniquement aux classes du même package et à ses sous-classes (même si elles sont définies dans un package différent.)

```
public class A {
   protected void meth4(){}
…
}
```

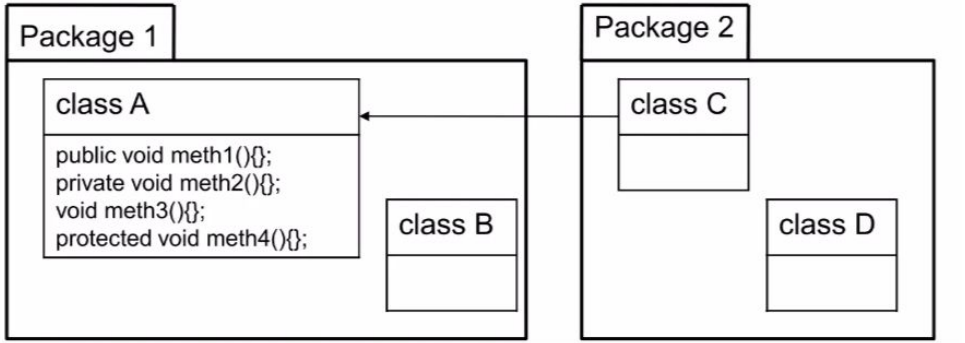
```
class B {
   A a=new A();
   a.meth4() ;
…
}
```

```
class C extends A {
   A a=new A();
   a.meth4() ;
…
}
```

```
class D {
   A a=new A();
   a.meth4();
…
}
```

Encapsulation

Encapsulation des méthodes



Visibilité à partir de:	Classe A	Package B	Classe fille C	Tout le reste D
public (meth1)				
private (meth2)				
default (meth3)				
protected (meth4)				

Encapsulation

Encapsulation des attributs/méthodes

Dans le but de renforcer le contrôle de l'accès aux attributs d'une classe, il est recommandé de les déclarer ***private***.

Pour la manipulation des attributs ***private***, *on utilise:*

- Un **mutateur** (`setter`): une méthode qui permet de définir la valeur d'un attribut particulier.
- Un **accesseur** (`getter`): une méthode qui permet d'obtenir la valeur d'un attribut particulier.

Le setter et le getter doivent être déclarés ***public***

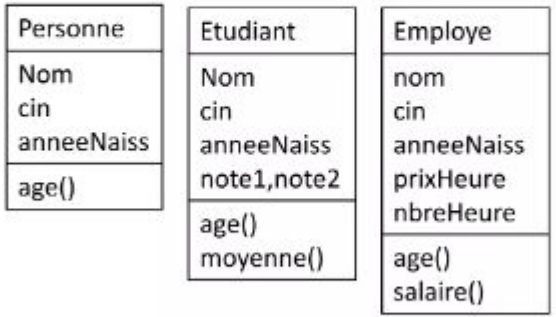
```
private float prix;

public void setPrix(float prix){
      this.prix=prix;
}
public float getPrix(){
      return prix;
}
```

```
private boolean exist;

public void setExist(boolean exist){
      this.exist=exist;
}
public boolean isExist(){
      return exist;
}
```

Héritage

Notion d'héritage

-Définition de 3 classes



Limites

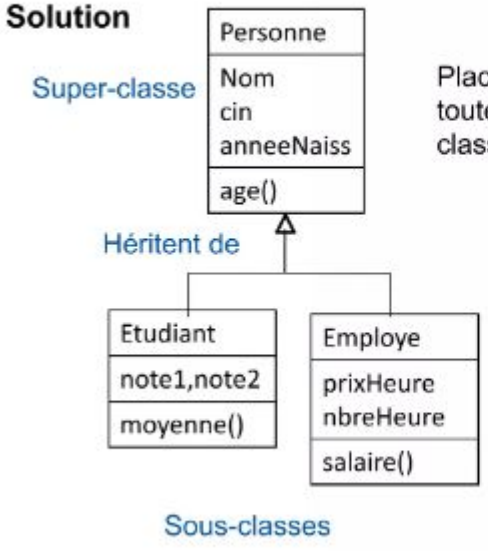
-Duplication de code.

-Une modification faite sur un attribut (ou méthode) commun doit être refaite dans les autres classes.

Héritage

Notion d'héritage

Solution



Placer dans la classe mère tous les attributs et toutes les méthodes communs à toutes les classes.

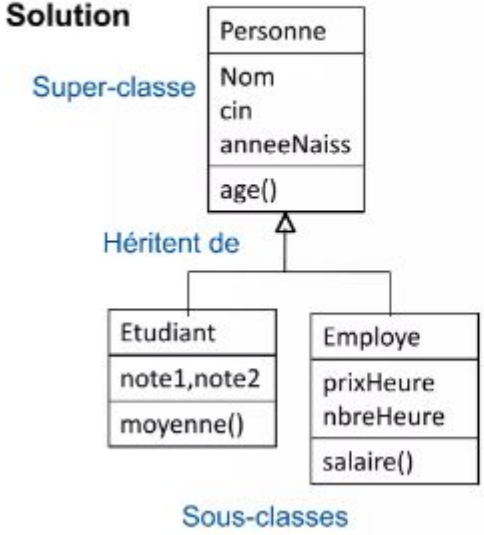
Les classes filles ne comportent que les attributs ou méthodes plus spécifiques.

Les classes filles héritent automatiquement les attributs (et les méthodes) qui n'ont pas besoin d'être ré-écrits.

Héritage

Notion d'héritage

Solution



+Réutilisation du code

+Une seule modification des attributs (ou méthodes) en commun

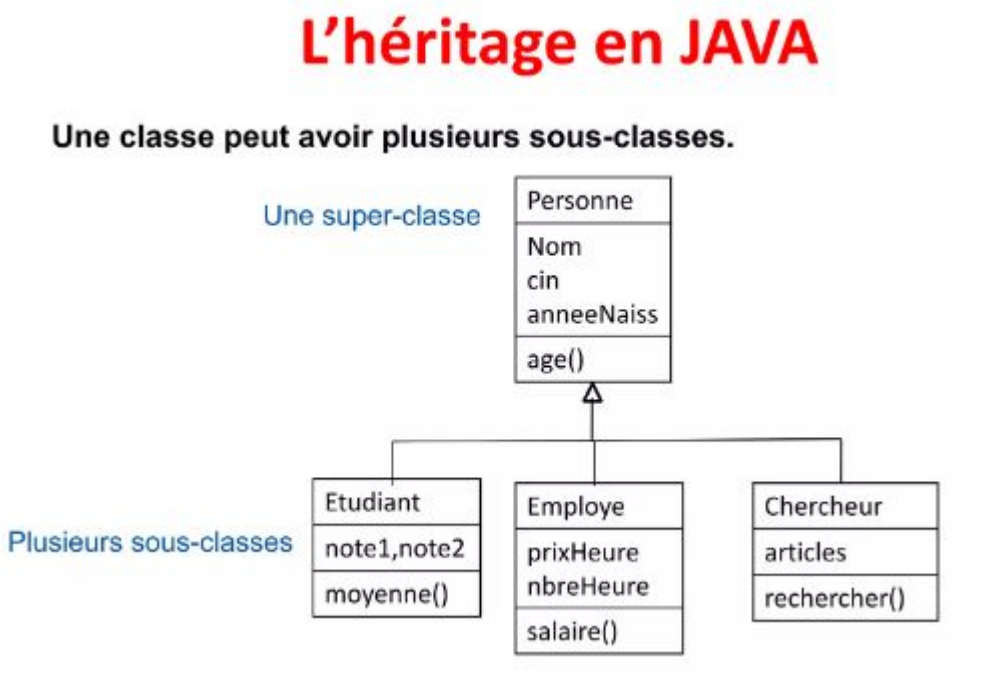
Héritage

L'héritage en JAVA

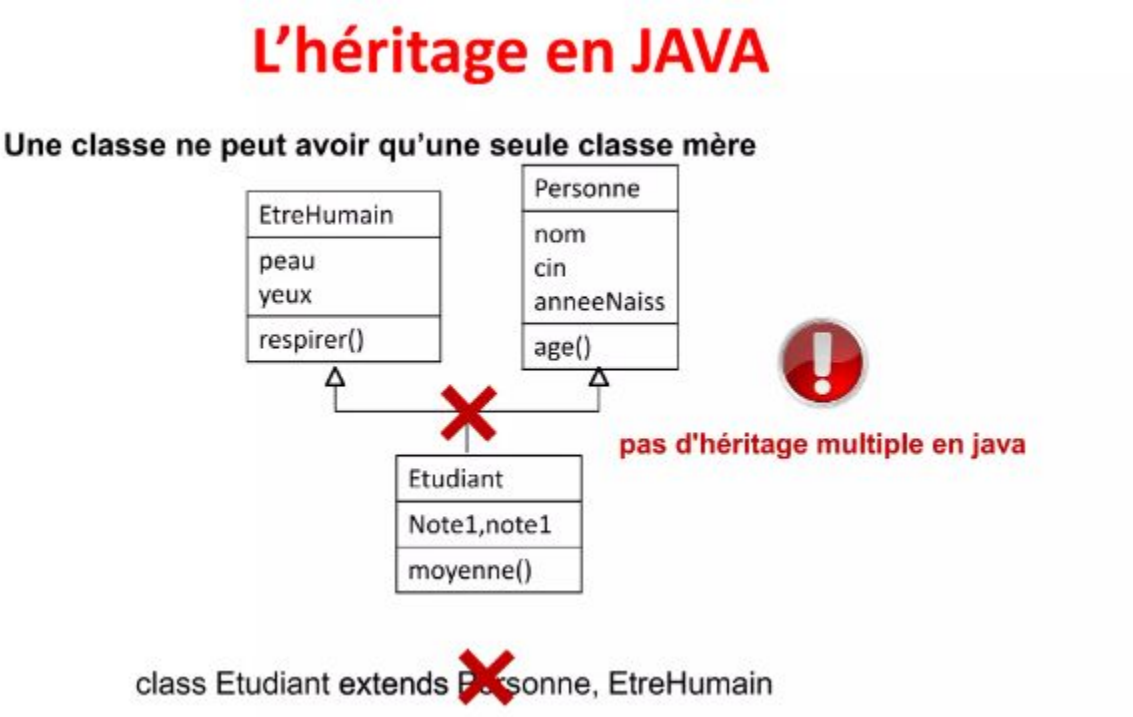
```
class Personne {

String nom;
int cin, anneeNaiss;

int age(){
            return 2014- anneeNaiss;
              }
```

```
class Etudiant extends Personne {

float note1,note2;


public float moyenne (){
            return ((note1+note2)/2);}
}
```

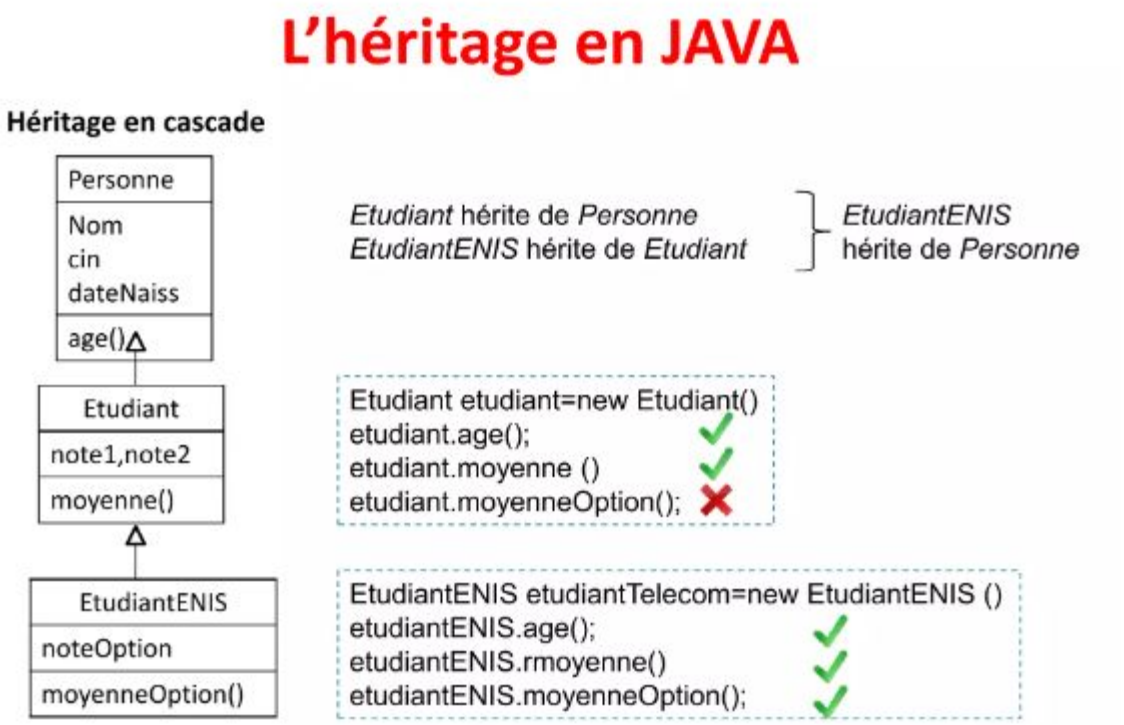
Héritage


L'héritage en JAVA
Une classe peut avoir plusieurs sous-classes.
Une super-classe
Personne
Nom
cin
anneeNaiss
age()
Plusieurs sous-classes
Etudiant
note1,note2
moyenne()
Employe
prixHeure
nbreHeure
salaire()
Chercheur
articles
rechercher()

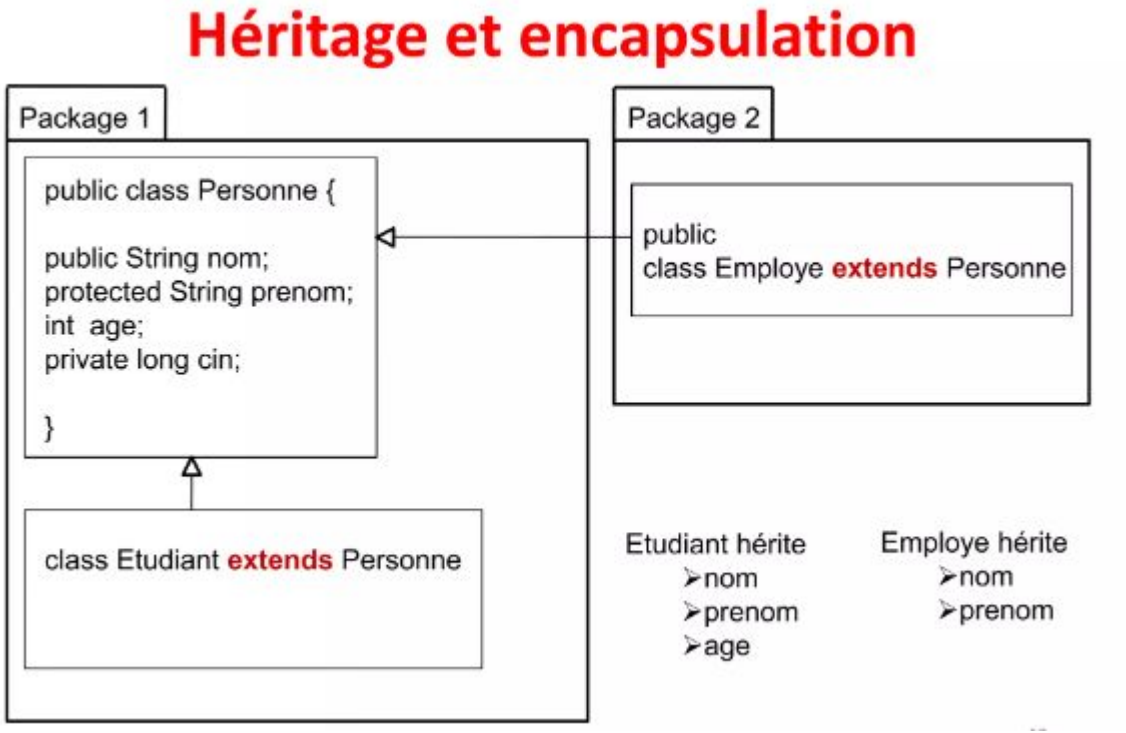
Héritage


L'héritage en JAVA
Une classe ne peut avoir qu'une seule classe mère
EtreHumain
peau
yeux
respirer()
Personne
nom
cin
anneeNaiss
age()
pas d'héritage multiple en java
Etudiant
Note1,note1
moyenne()
class Etudiant extends Personne, EtreHumain

Héritage


L'héritage en JAVA
Héritage en cascade
Personne
Nom
cin
dateNaiss
age()
Etudiant
note1,note2
moyenne()
EtudiantENIS
noteOption
moyenneOption()
Etudiant hérite de Personne
EtudiantENIS hérite de Etudiant
EtudiantENIS hérite de Personne
Etudiant etudiant=new Etudiant()
etudiant.age();
etudiant.moyenne ()
etudiant.moyenneOption();
EtudiantENIS etudiantTelecom=new EtudiantENIS ()
etudiantENIS.age();
etudiantENIS.rmoyenne()
etudiantENIS.moyenneOption();

Héritage


Héritage et encapsulation
Package 1
public class Personne {
public String nom;
protected String prenom;
int age;
private long cin;
}
class Etudiant extends Personne
Package 2
public
class Employe extends Personne
Etudiant hérite
nom
prenom
age
Employe hérite
nom
prenom

Héritage

Le mot-clé super

Le mot-clé *super* permet de désigner la superclasse,
➡ *super* permet d'accéder aux attributs et méthodes de la super-classe

- Pour manipuler un attribut de la super-classe :

  super.cin

- Pour manipuler une méthode de la super-classe :

  super.age()

- Pour faire appel au constructeur de la super-classe:

  super(nom, cin)

Héritage

Constructeurs et héritage

Constructeur par défaut

```
class Personne {

private String nom;
private long cin;

public Presonne(){
            nom="Mohamed";
            cin=00000000;
}

}
```

```
class Etudiant extends Personne {

private float note1,note2;

public Etudiant(){
            super();
            note1=0.0f;
            note2=0.0f;

}

}
```

super fait appel au constructeur par défaut de la classe mère

Héritage

Constructeurs et héritage

Constructeur surchargé

```
class Personne {

private String nom;
private long cin;

public Presonne(String nom, long cin){
            this.nom=nom;
            this.cin=cin;
}

}
```

```
class Etudiant extends Personne {

private float note1,note2;

public Etudiant(String nom, long cin, int
note1, int note 2){
            super(nom, cin);
            this.note1=note1;
            this.note2=note2;
}

}
```

super fait appel au constructeur surchargé de la classe mère

Héritage

Constructeurs et héritage

```
public class Animal {

  public int nbPattes;

  public Animal(int nbPattes) {
        this.nbPattes = nbPattes; }
  }
```

```
public class Chat extends Animal{

  public Chat(){        Constructeur par défaut
        super();        Appel implicite à super
  }
}
```

- Un constructeur surchargé est crée pour la classe *Animal*
  → le constructeur *par défaut* n'existe pas.
- Erreur à la compilation lors de l'exécution de *super()*

Héritage

Constructeurs et héritage

Solution 1

Déclarer explicitement le constructeur par défaut de la classe mère

```
public class Animal {

   public int nbPattes;

   public Animal(){}

   public Animal(int nbPattes) {
            this.nbPattes = nbPattes; }
   }
```

```
public class Chat extends Animal{

}
```

Héritage

Constructeurs et héritage

Solution 2

Faire un appel explicite au constructeur surchargé de la classe mère

```
public class Animal {

  public int nbPattes;

  public Animal(int nbPattes) {
        this.nbPattes = nbPattes;
   }
}
```

```
public class Chat extends Animal{

  public Chat(){
        super(4);
  }
}
```

Héritage

Constructeurs et héritage

Enchaînement des consctructeurs

```
public class Humain{
  public Humain() {
          System.out.println("Je suis un être humain");
  }
}
```

```
public class Client extends Humain {
  public Client (){
          System.out.println("Je suis un client");
  }
}
```

```
public class ClientFidele extends Client {
  public ClientFidele(){
          System.out.println("Je suis un client fidèle");
  }
}
```

```
ClientFidele cFidele=new ClientFidele();
```

Héritage

Redéfinition des méthodes

```
class Personne {

    public void lire() {
            System.out.println("Je sais lire");
    }

}
```

```
class Etudiant extends Personne {

 public void lire() {
            System.out.println("Je suis un étudiant");
            System.out.println("Je sais lire");
    }
}
```

- La classe Etudiant hérite la méthode lire() mais on veut changer son traitement
- On garde **la même signature** de la méthode lire et on change le corps
  - On parle alors de **redéfinition (override)**

Héritage

Redéfinition des méthodes

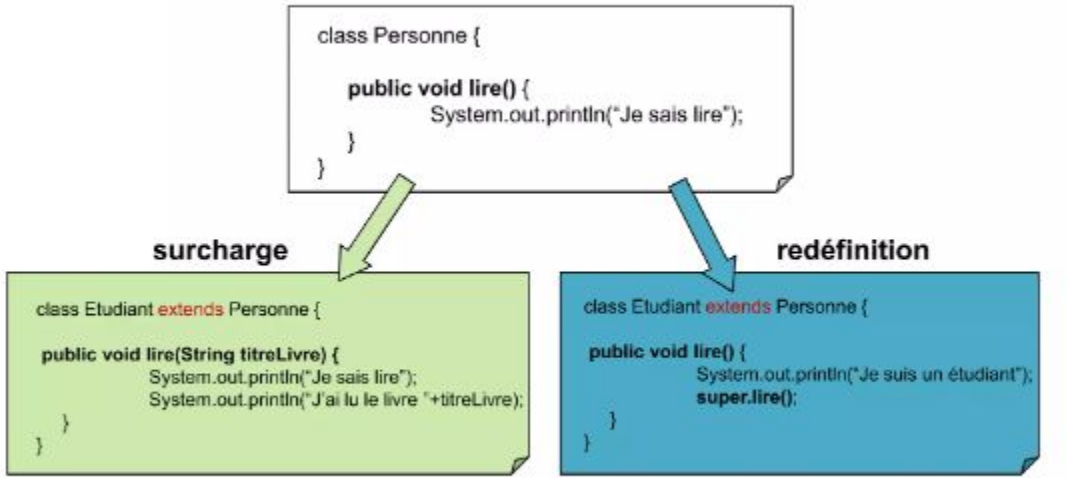
```
class Personne {

    public void lire() {
            System.out.println("Je sais lire");
    }

}
```

```
class Etudiant extends Personne {

 public void lire() {
            System.out.println("Je suis un etudiant");
            super.lire();
    }
}
```

➡ Possibilité de réutiliser le code de la méthode héritée (super)

Héritage

Surcharge vs redéfinition



Etudiant possède deux méthodes

- `lire()`
- `lire(String)`

Etudiant possède une seule méthode

- `lire()`

Héritage

Le mot-clé final

Empêcher la redéfinition

- Une classe peut protéger une méthode afin d'éviter qu'elle ne soit redéfinie dans ses sous-classes.

- En Java, on va simplement ajouter le modificateur **final** dans la signature de la méthode.

```
public final void lire()
```

Empêcher l'hériatage

- Une classe déclarée final ne peut pas être dérivée

```
final class Personne
```

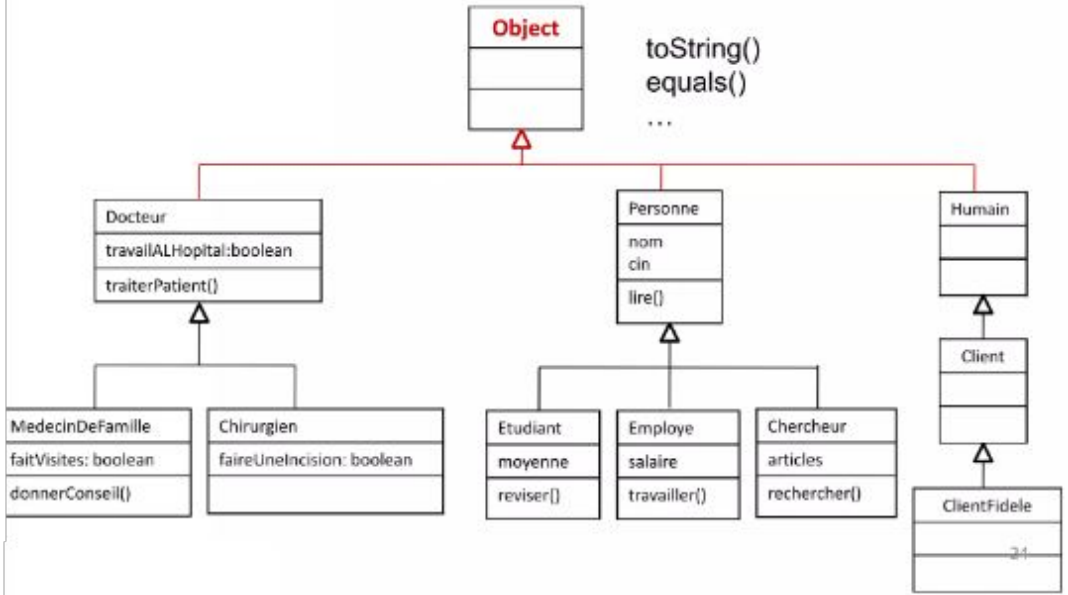
➢Permet d'interdire tout héritage pour cette classe qui ne pourra pas être une classe mère

➢Toutes les méthodes à l'intérieur de la classe final seront implicitement finales

Héritage

La classe Object

- Toutes les classes java héritent implicitement de la classe ***Object.***
- La classe *Object est l'ancêtre* de toutes les classes



Héritage

La méthode equals

- La méthode equals détermine si deux objets sont égaux ou non.
- Son implémentation dans la classe Object vérifie que **les références d'objets sont identiques.**
- On aura souvent besoin de redéfinir la méthode equals **: deux objets seront égaux quand ils auront le même état.**

Redéfinition de la méthode equals pour la classe Personne

➡ Deux personnes sont égaux s'ils ont le même cin et le même nom

```
public boolean equals(Object obj) {
      if(obj instanceof Personne){
            Personne personne = (Personne) obj;
            if((nom.equals(personne.nom)) && (cin==personne.cin))
                  { return true; }

   return false;
}
```

Polymorphisme

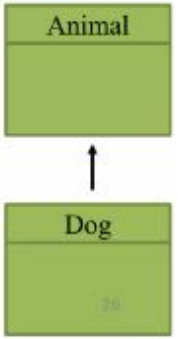
Polymorphisme

- N'importe quel objet en Java qui peut passer d'une relation « est un » peut être considéré polymorphe.
- Le polymorphisme est le fait de référencer une classe ***fille*** avec un référence déclaré de type une classe ***mère***.

```
Animal myDog = new Dog();
```

Référence déclaré en tant qu'Animal().

Objet crée en tant que Dog().



Polymorphisme

Polymorphisme

- Pourquoi?

-Ecrire un code flexible.

-Ecrire un code nettoyé :

–plus efficace.

–plus simple.

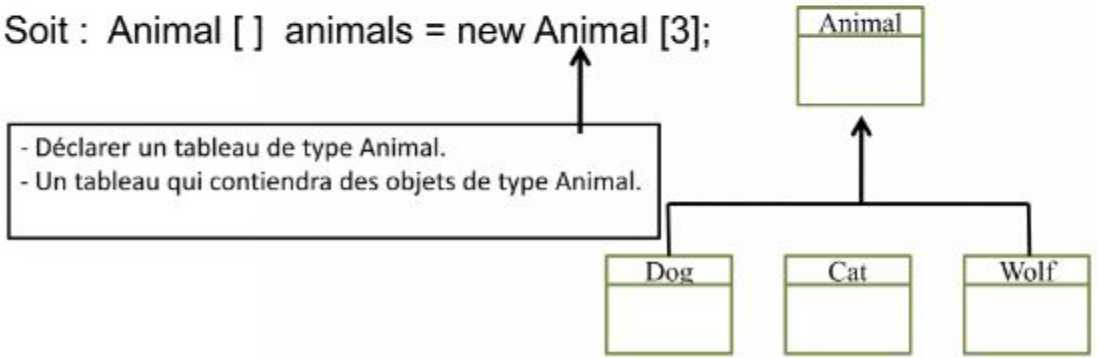
-Facile à développer.

-Facile à étendre.

Polymorphisme

Tableau polymorphique

❖ Avec le polymorphisme : le type de la référence peut être la classe mère de l'objet instancié.

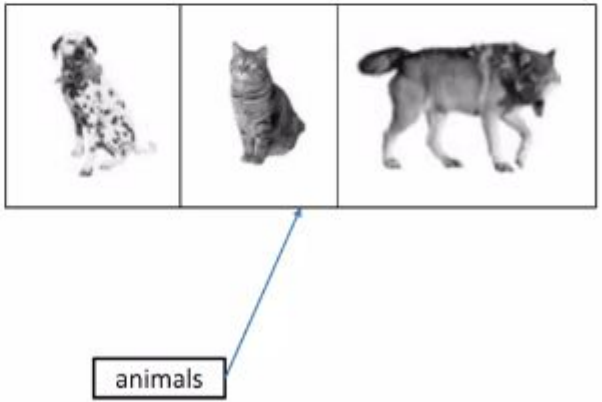


Polymorphisme

Tableau polymorphique

- On peut mettre n'importe quelle classe fille de la classe Animal() dans le tableau.

```
animals [0] = new Dog();
animals [1] = new Cat();
animals [2] = new Wolf();
```



Polymorphisme

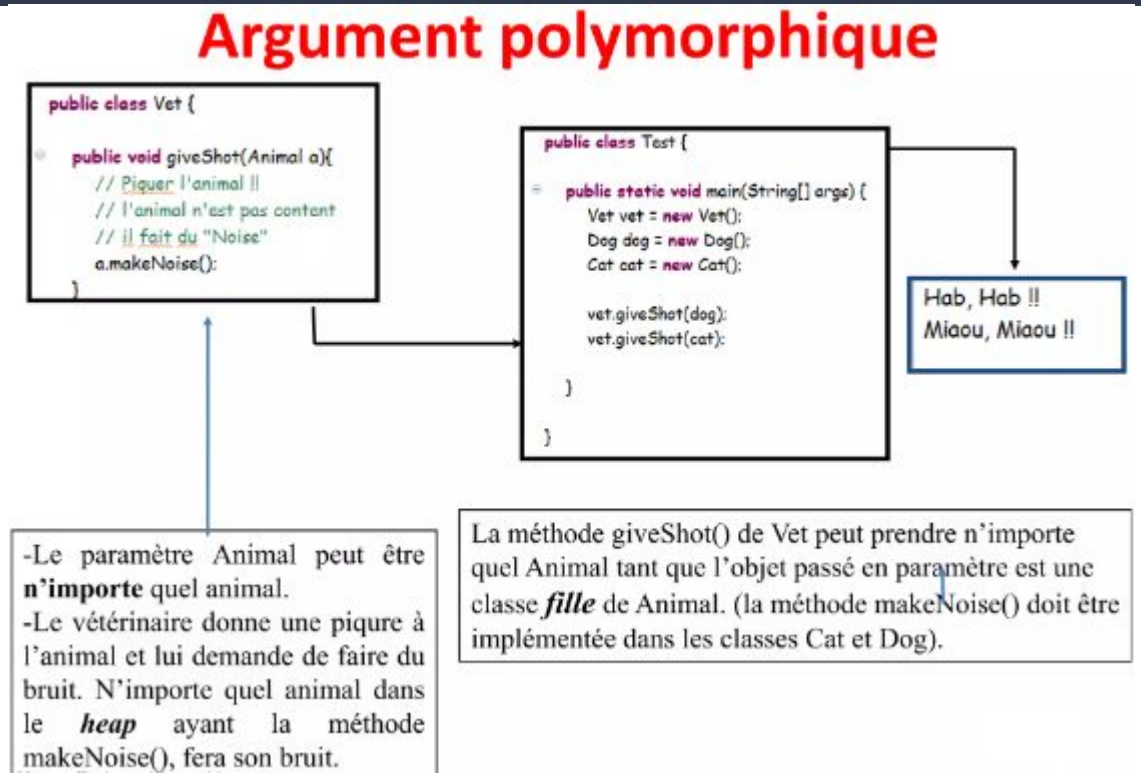
```java
public class Animal {

    public void eat(){
        System.out.println("Un animal peut manger !!");
    }
    public void roam(){
        System.out.println("Un animal peut voyager !!");
    }

}
```

```java
public class Cat extends Animal {

    @Override
    public void eat() {
        System.out.println("Je suis un CHAT, Je mange");
    }

    @Override
    public void roam() {
        System.out.println("Je suis un CHAT, Je voyage");
    }
}
```

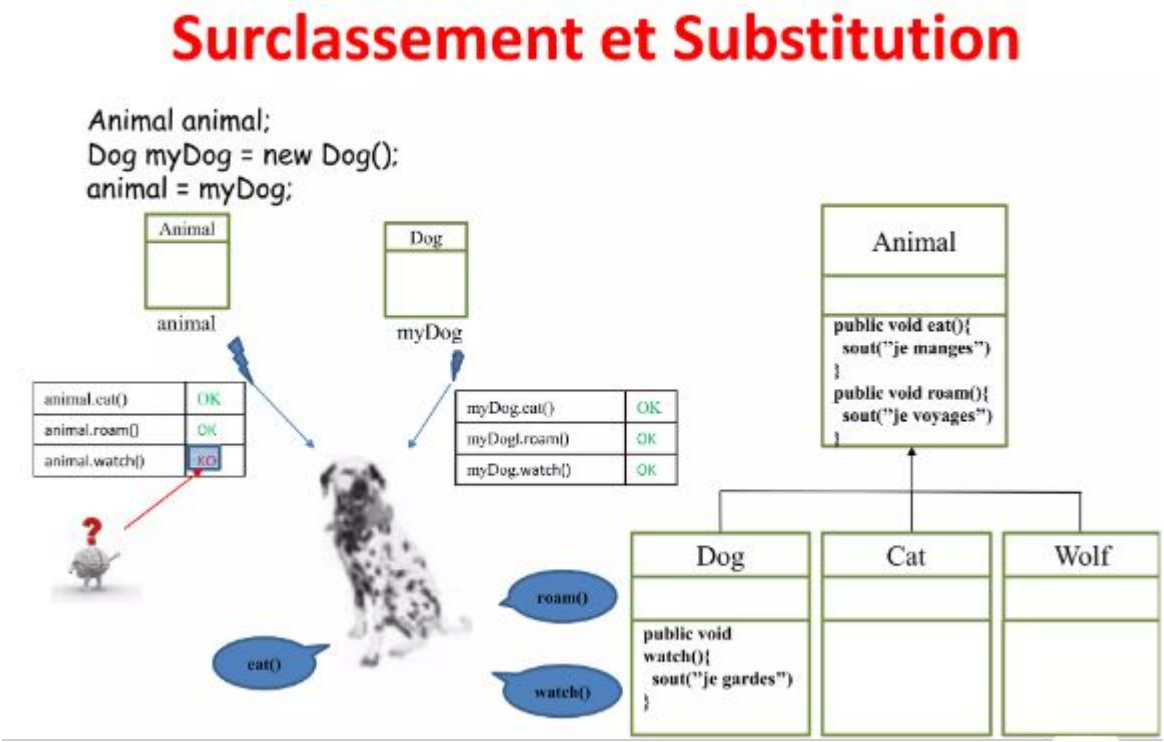
```java
public class Dog extends Animal {

    @Override
    public void eat() {
        System.out.println("Je suis un CHIEN, Je mange");
    }

    @Override
    public void roam() {
        System.out.println("Je suis un CHIEN, Je voyage");
    }
}
```

```java
public class Wolf extends Animal {

    @Override
    public void eat() {
        System.out.println("Je suis un LOUP, Je mange");
    }

    @Override
    public void roam() {
        System.out.println("Je suis un LOUP, Je voyage");
    }
}
```

Polymorphisme



Polymorphisme



Polymorphisme

Surclassement et Substitution

On dit que :

- l'objet Dog crée est « ***surclassé*** ». Il est vu de type Animal de la référence déclarée (animal).
- Les fonctionnalités de Dog sont restreintes à celles de Animal. Le chien **ne** pourra **pas** être un chien de garde (**watch()**).

Il faut donc ***substituer*** animal par :

animal.eat()	OK
animal.roam()	OK
((Dog) animal).watch()	OK

((Dog) animal).watch();

Polymorphisme

Lien dynamique

Quelle est la méthode appelée ?

La méthode appelée est celle existante dans la classe du **type le plus *inférieur* dans l'hiérarchie d'héritage gagne !**

La machine virtuelle *JVM* commence tout d'abord à voir dans la classe *Wolf*.
Si elle ne trouve pas une correspondance de la version de la méthode, elle commence à grimper l'hiérarchie de l'héritage jusqu'à trouver la bonne méthode.

Wolf w = new **Wolf();**

w.makeNoise();

w.roam();

w.eat();

w.sleep();