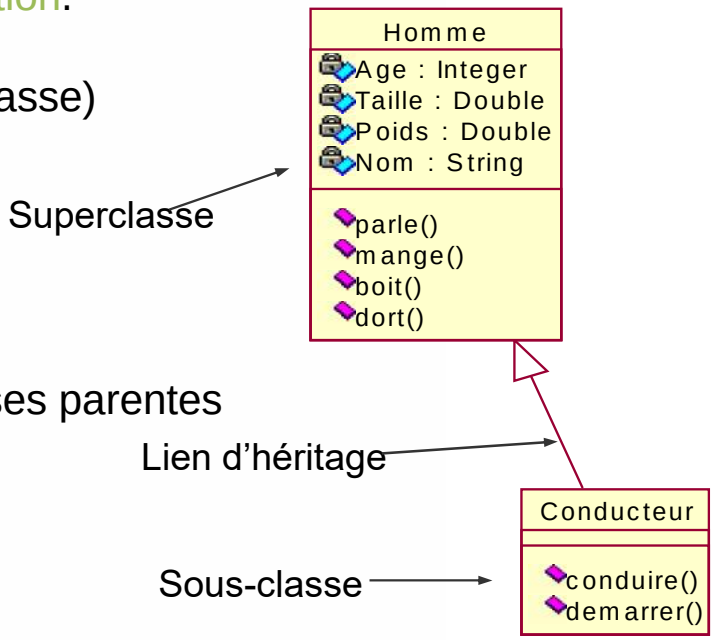
L’héritage avec JAVA

Héritage: mécanisme permettant le partage et la réutilisation de propriétés entre les objets.

- La relation d’héritage est une relation de généralisation / spécialisation.
- La « super super » classe, est la classe Object (parente de toute classe)
- La classe parente est la superclasse.
- La classe qui hérite est la sous-classe.
- Une sous-classe hérite des variables et des méthodes de ses classes parentes
- La clause extends apparaît dans la déclaration de la classe
- Java n’offre pas la possibilité d’héritage multiple.



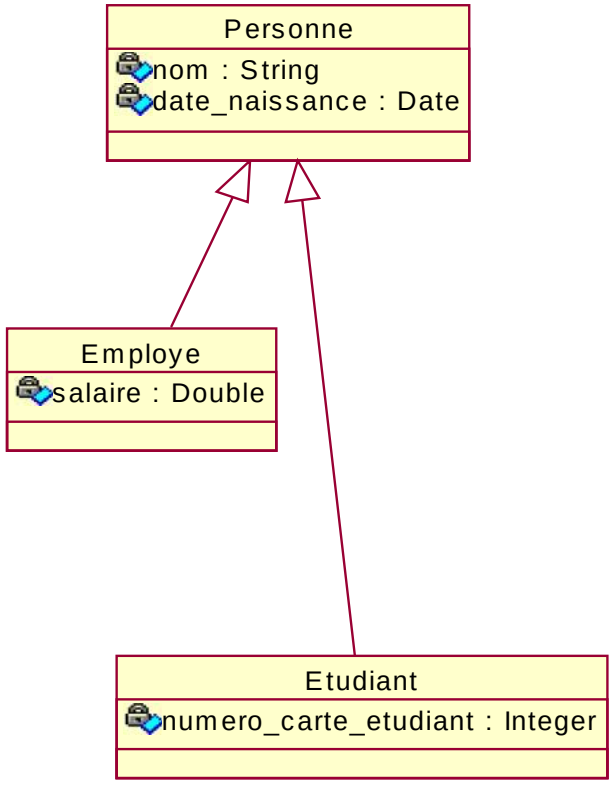
L’héritage avec JAVA

Exemple

```
class Personne
  {
    private String nom;
    private Date date_naissance;
    // ...
  }

class Employe extends Personne
  {
    private float salaire;
    // ...
  }

class Etudiant extends Personne
  {
    private int numero_carte_etudiant;
    // ...
  }
```



L’héritage

Polymorphisme :

- C’est la capacité pour une entité de prendre plusieurs formes.
  - En Java, toute variable désignant un objet est potentiellement polymorphe, à cause de l’héritage.
  - Polymorphisme dit « d’héritage »

- On distingue deux types de polymorphismes :

- **Polymorphisme de classe :** un objet possèdent plusieurs types : sa classe et ses ascendantes.

- **Polymorphisme de méthodes** : une même écriture peut correspondre à différents appels de méthodes ; la méthode qui sera exécuté est déterminée à l’exécution (late binding) suivant la classe de l’objet courant.

L'héritage avec JAVA

Tous les champs sont hérités

- Ils peuvent être manipulés si leur accessibilité le permet
  - Les classes filles de A ont un accès aux membres protected de A, les autres classes non filles de A n'y ont pas accès.
  - Protected autorise l'utilisation par les classes du même paquetage que la classe où est défini le membre ou le constructeur

  Si x n'est pas private dans Pixel, on peut utiliser this.nom dans Employe
- Ils peuvent être masqués par la définition de champs qui ont le même nom dans la classe dérivé
- Si String date naissance est déclaré dans Employe, c'est celui qui sera considéré dans cette classe quand on parle de this. date_naissance

  Il est possible de manipuler celui qui est masqué (s'il est accessible) par la notation super. date_naissance
  - La résolution des champs est effectuée par le compilateur, en fonction du type déclaré de la variable qui contient la référence

L'héritage avec JAVA

Constructeurs : « this » et « super »

- Il existe toujours un constructeur.
- S'il n'est pas explicitement défini, il sera un constructeur par défaut, sans arguments
- « this » est toujours une référence sur l'objet en cours (de création) lui-même
- « super » permet d'appeler le constructeur de la classe parent, ce qui est obligatoire si celui-ci attend des arguments

```
public class Employe extends Personne
{private float salaire;
 public Employe () {}
 public Employe (String nom,
                 String prenom,
                  int anNaissance, float s)
 {   super(nom, prenom, anNaissance);
Salaire = s; }
}
```

```
public class Personne
{
   public String nom, prenom;
   public int anNaissance;
   public Personne()
   {
     nom=""; prenom="";
   }
   public Personne(String nom,
                   String prenom,
                   int anNaissance)
   {
     this.nom=nom;
     this.prenom=prenom;
     this.anNaissance=anNaissance;
   }
}
```

L'héritage avec JAVA

Redéfinition de méthodes

- En plus des champs, en tant que « membres », la **classe dérivée hérite des méthodes de la classe de base**
- La redéfinition n'est pas obligatoire !! Mais elle permet d'adapter un comportement et de le spécifier pour la sous-classe.
  - Le terme anglophone est "**overriding**". On parle aussi de masquage.
  - La méthode redéfinie doit avoir **la même signature**.
- Obligation de redéfinir les méthodes déclarées comme abstraites (*abstract*)
- Interdiction de redéfinir les méthode déclarées comme finales (*final*)
- On ne redéfinit pas une méthode **static.**

```
Public class Employe extends Personne
  {
     private float salaire;
      public toString() {
   return super.toString()+« salaire=
"+salaire;
  }
 public static void main(String[] args) {
Personne  p = new
Personne("aa","bb,2000);
    System.out.println(p);
Employe e = new
Employe("aa","bb,2000,1600.5);
    System.out.println(p);
    System.out.println(e); }
  }
```

L'héritage avec JAVA

Redéfinition (méthodes) versus masquage (champs)

- **Les champs** définis dans les classes dérivées sont tous présents dans l'objet instance de la classe dérivée
  - Même s'ils ont même nom et même type
  - On peut accéder à celui immédiatement supérieur par super.x
  - La résolution dépend du type déclaré du paramètre
  - Ça permet d'accéder à chacun d'entre eux par transtypage
- Pour **la méthode,** une seule est conservée
  - On peut accéder à celle immédiatement supérieure par super.m()
  - La résolution est faite en deux temps
    - *Compiletime*: on vérifie que c'est possible sur le type déclaré
    - *Runtime*: on cherche la plus précise étant donnée le type « réel »
  - Les autres ne sont plus accessibles

L’héritage avec JAVA

Redéfinition versus surcharge

- Si la signature de la méthode qu’on définit dans la classe dérivée n’est pas la même que celle de la classe de base, il s’agit de **surcharge:**
- Les deux méthodes cohabitent dans la classe dérivée

```
class A {
  void m1() { ... }
  void m2() { ... }
  Personne m3() { ... }
  void m4(Personne p) { ... }
}
class B extends A {
  @Override void m1() { ... }            // redefinition
          void m2(int a) { ... }         // surcharge
  @Override Employe m3() { ... }         // redefinition
  @Override void m4(Personne p) { ... }  // redefinition
          void m4(Employe p) { ... } // surcharge
}
```

L’héritage avec JAVA

Les classes et méthodes « final »

Le mot clé final existe pour les méthodes:

- Il signifie que la méthode ne pourra pas être redéfinie dans une sousclasse
- Peut être utile pour garantir qu’aucune autre définition ne pourra être donnée pour cette méthode (sécurité)

Le mot clé final existe pour les classes:

- Il devient alors impossible d’hériter de cette classe
- Les méthodes se comportent comme si elles étaient final

L'héritage avec JAVA

L'opérateur instanceof

- Il est possible d'assurer un transtypage sans exception en utilisant l'opérateur **x instanceof T**
- x doit être une (variable contenant une) référence ou null
- T doit représenter un type
- Le résultat vaut true si x n'est pas null et s'il peut être affecté dans T sans exception ; sinon c'est false.

```
class A { }
class B extends A { }
class C extends B { }
A ab = null;
System.out.println(ab instanceof A);  // false
ab = new B();
System.out.println(ab instanceof A);  // true
System.out.println(ab instanceof B);  // true
System.out.println(ab instanceof C);  // false
```

```
if ( ... )
  Personne jean = new Etudiant();
else
 Personne jean = new Employe();
//...
if (jean instanceof Employe)
 // discuter affaires
else
 // proposer un stage
```

L'héritage avec JAVA

Transtypage/Cast :

- Pour les objets, les seuls casts autorisés sont les casts entre classe mère et classe fille.

Notation : pour caster un objet o en classe C : (C)o.

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

❑ Upcast (implicite) :

- Un objet est considéré comme une instance d'une des classes ancêtres de sa classe réelle.
- Il **est toujours possible de faire un upcast : à cause de la relation est-un de** l'héritage, tout objet peut être considéré comme une instance d'une classe ancêtre

❑ Downcast (explicte):

- Un objet est considéré comme étant une classe fille de sa classe de déclaration.
- Toujours accepté par le compilateur.
- Mais peut provoquer une erreur à l'exécution si l'objet n'est pas du type de la classe fille.
- Un downcast doit toujours être explicite.
- **Utilisation :** Utilisé pour appeler une méthode de la classe fille qui n'existe pas dans une classe ancêtre.

```
Personne jean = new Employe ();
float i = jean.salaire;  // Erreur de compilation
float j = ( (Employe) jean ).salaire;    // OK
```

L’héritage avec JAVA

Exercice d’application : Qu’affiche le programme suivant:

```
class Fruit {
public void quiEstTu( ) {
System.out.println(« Je suis un
Fruit ») ; }
}
class Pomme extends Fruit {
public void quiEstTu( ) {
 System.out.println(« Je suis une
Pomme ») ; }
}
class Poire extends Fruit {
public void quiEstTu( ) {
 System.out.println(« Je suis une
Poire ») ; }
}
```

```
class Test {
public void main(String[] args){
Pomme pm = new Pomme( );
Poire pr = new Poire( );
Fruit f ;
f = (Fruit)pm ;
f.quiEstTu( ) ;
f = (Fruit)pr ;
f.quiEstTu( ) ;
```