Héritage I

Héritage I

- Soient les 3 classes Chat, Merle et Vache

Chat
- nbpattes
+ Chat() + marcher() + crier()

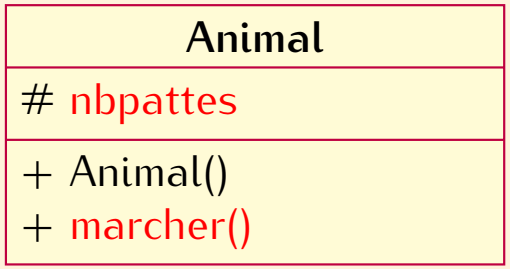
Merle
- nbpattes
+ Merle() + marcher() + voler()

Vache
- nbpattes
+ Vache() + marcher() + crier()

- Elles partagent
  ⇒ un attribut commun : nbpattes
  ⇒ une opération commune : marcher()

Héritage I

⇒ La classe Animal contient ce qui est commun à Chat, Merle et Vache.
⇒ La classe Animal est une classe de base.

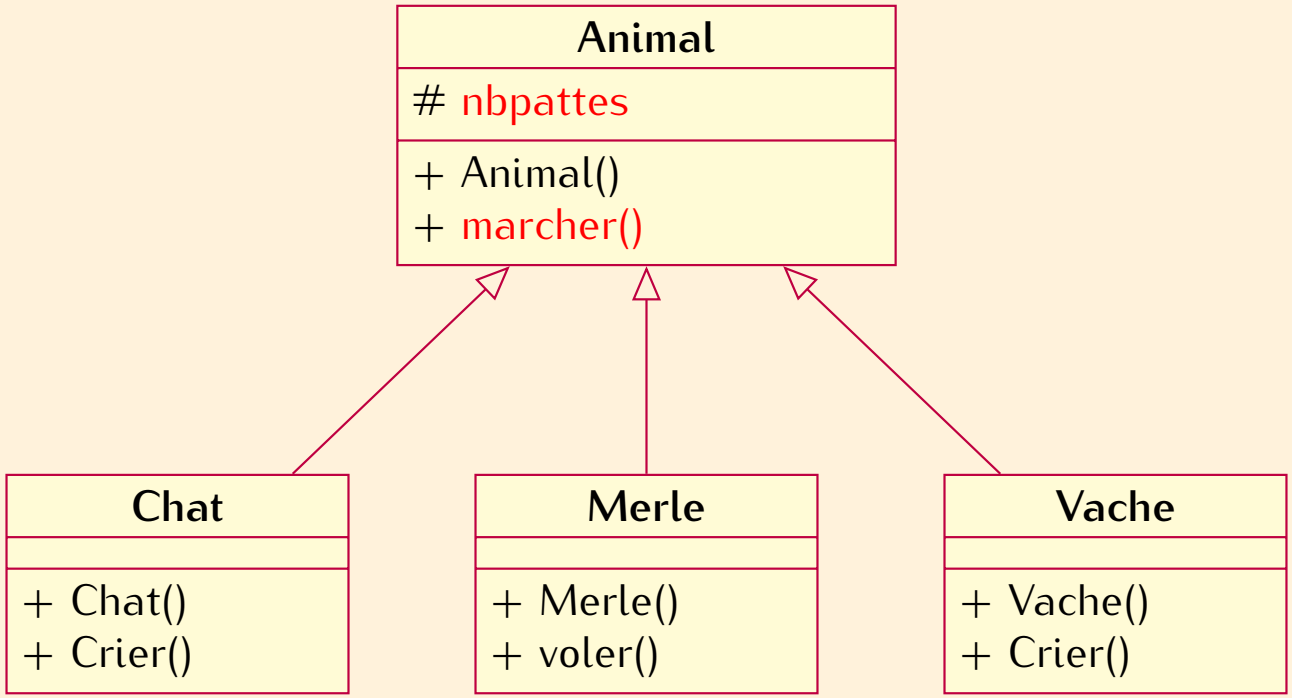



```
1  class Animal{
   protected int nbpattes;
3  public Animal(int nb){
   nbpattes=nb;
5  }
   public void marcher(){
7  System.out.println("jemarche");}
   };
```

⇒ Les attributs sont déclarés **protected** dans la classe de base

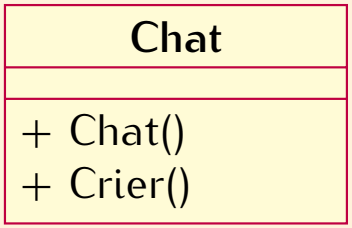
Héritage I

⇒ Les classes dérivées Chat, Merle et Vache héritent de la classe de base Animal.



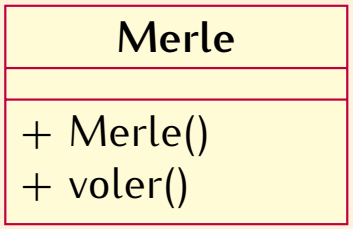
Héritage I

1. La classe Chat hérite de la classe Animal



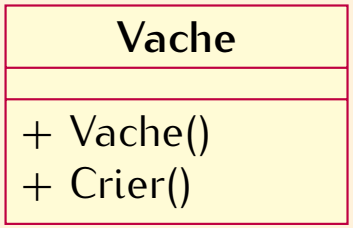

```
class Chat extends Animal{
public Chat(int nb){super(nb);}
public void Crier(){
System.out.println("jemiaule");}}
```

2. La classe Chien hérite de la classe Animal




```
class Merle extends Animal{
public Merle(int nb){super(nb);}
public void voler(){
System.out.println("jesaisvoler");}}
```

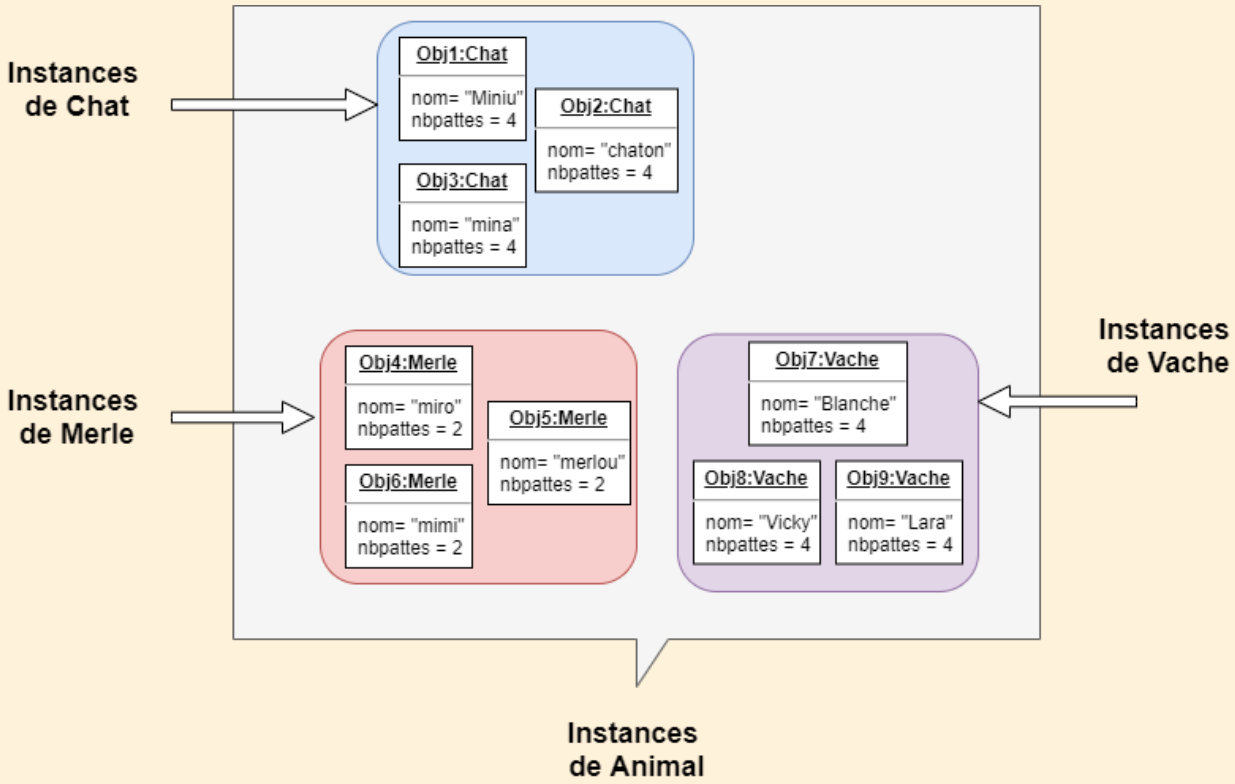
3. La classe Vache hérite de la classe Animal




```
class Vache extends Animal{
public Vache(int nb){super(nb);}
public void Crier(){
System.out.println("jemeugle");}}
```

Héritage I

- Un chat est un animal particulier ⇒ Ensemble Chat ⊂ Ensemble Animal,
- Un Merle est un animal particulier ⇒ Ensemble Merle ⊂ Ensemble Animal,
- Une vache est un animal particulier ⇒ Ensemble Vache ⊂ Ensemble Animal,



Héritage I

- Soient les deux classes Etudiant et Employe

Etudiant
- nom:string - age:int - niveau:int
+ Etudiant(string,int,int) + afficherNiveau() + afficher()

Employe
- nom :string - age:int - nbHeures:double
+ Employe(string,int,double) + afficherHeures() + afficher()

- Elles partagent :
  ⇒ deux attributs communs : nom et age
  ⇒ une méthode commune : afficher()

Héritage : je connais pas I

Héritage

⇒ Voici son implémentation java de la classe Etudiant

Etudiant
- nom:string - age:int - niveau:int
+ Etudiant(string,int,int) + afficherNiveau() + afficher()

```
    class Etudiant{
2       private string nom;
        private int age;
4       private int niveau;
    public Etudiant(string n,int a,int n){
6       this.nom=n;
        this.age=a;
8       this.niveau=n;
    }
10  public void afficher(){
        System.out.println(nom+age+niveau)
            ;
12  }
    void afficheeNiveau(){
14      System.out.printl(niveau);
    }}
```

⇒ Voici son implémentation java de la classe Employe

Employe
- nom :string - age:int - nbHeures:double
+ Employe(string,int,double) + afficherHeures() + afficher()

```
1   class Employe{
        private string nom;
3       private int age;
        private double nbheures;
5   public Employe(string n,int a,double h){
        this.nom= n;
7       this.age=a;
        this.nbheures=h;
9   }
    public void afficher(){
11      System.out.println(nom+age+
            nbheures);
    }
13  public void affichHeures(){
        System.out.println(nbheures);
15  }}
```

Héritage

Mr David Tonsac

L'héritage
Je ne connais que ça
extends c'est super

Héritage

⇒ On peut définir une classe de base : Personne
⇒ la classe Personne contient tout ce qui est commun à Etudiant et Employe

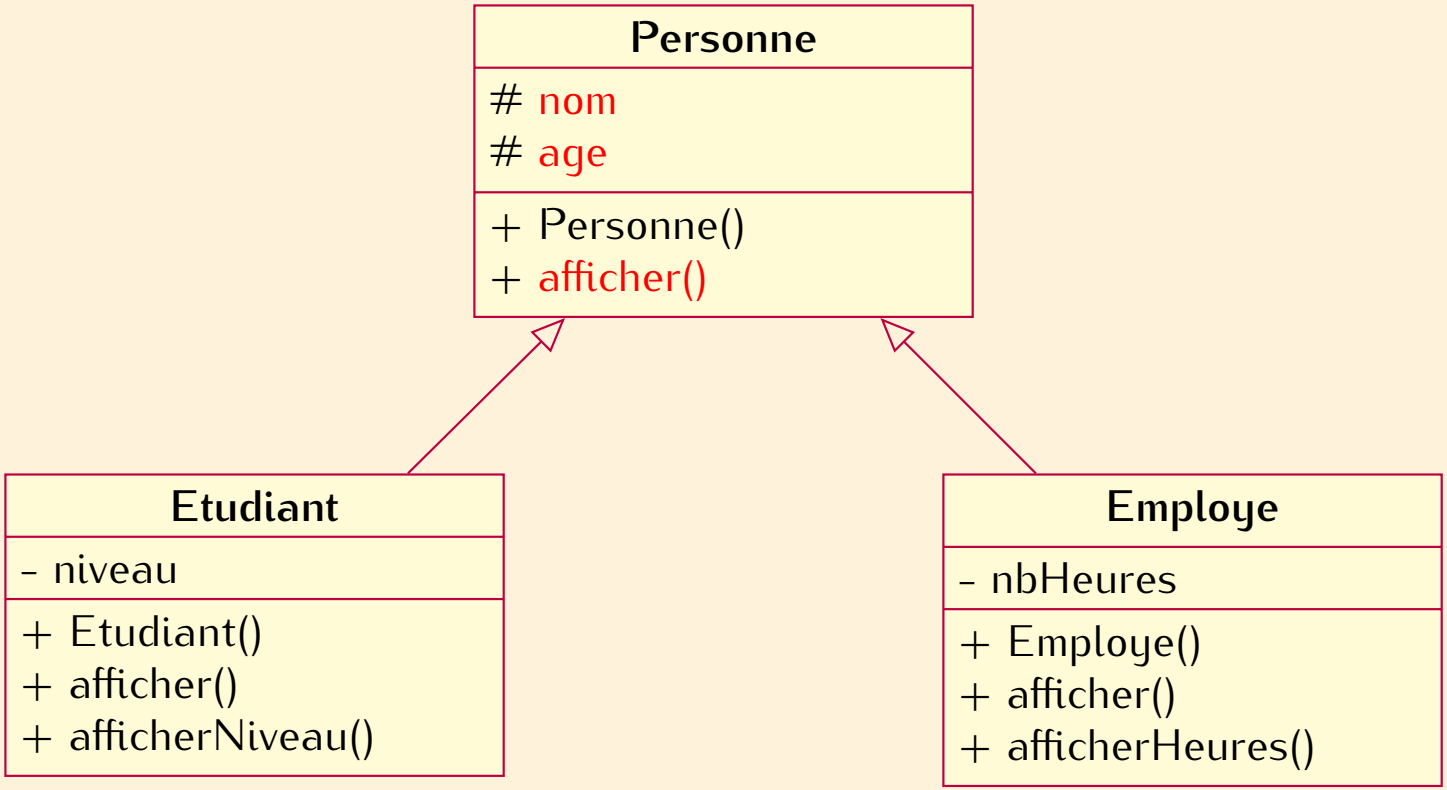
Personne
nom # age
+ Personne(string,int) + afficher()

```
1   class Personne{
    protected string nom; // attribut protected
3   protected int age;    // attribut protected
    public Personne(string nom,int age){
5   this.nom=nom;
    this.age=age;
7   }
    public void afficher(){
9   System.out.println(nom,age);
    }
11  };
```

⇒ Les classes à Etudiant et Employe sont des classes dérivées

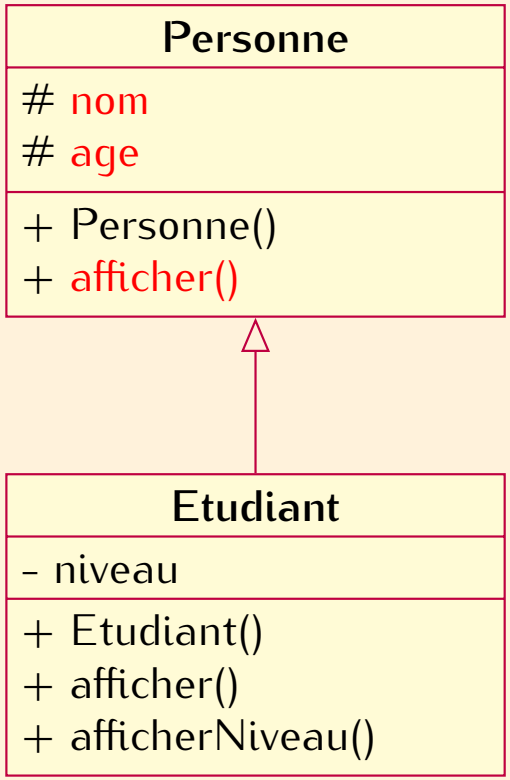
Héritage

⇒ Les classes Etudiant et Employe héritent de la classe Personne.



Héritage

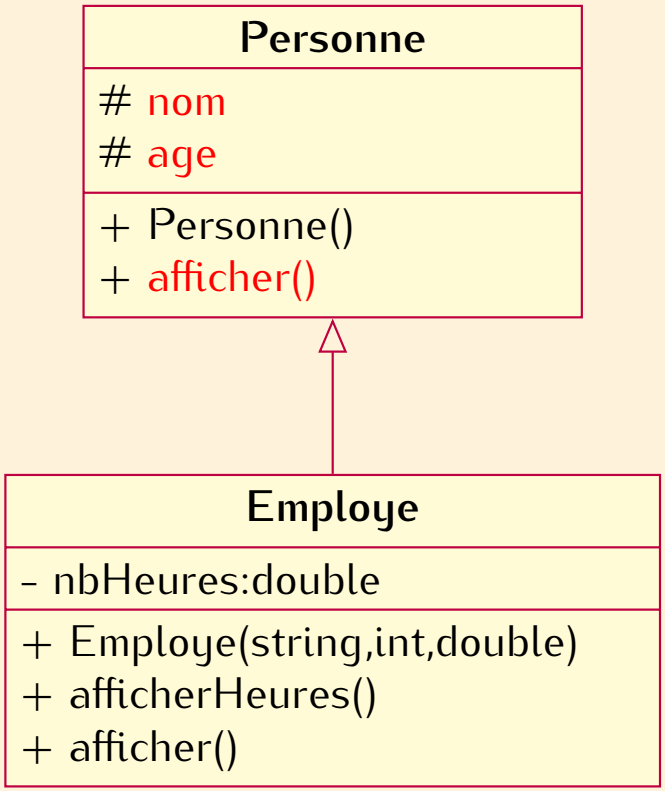
Etudiant hérite de Personne




```
1  class Etudiant extends Personne{
     private int niveau;
3  public Etudiant(string nom,int age,int niv){
     super(nom,age);//constructeur Personne
5    this.niveau=niv;
   }
7  public void afficher(){
     super.afficher(); //affiche Personne
9    System.out.println(niveau);
   }
11 public void afficherNiveau(){
     System.out.println(niveau);
13 }}
```

Héritage

Employe hérite de Personne




```
1   class Employe extends Personne{
     private double nbheures;
3   public Employe(string nom,int age,double nbh
     super(nom,age);//constructeur Personne
5    this.nbheures=nbh;}
    public void afficher(){
7    super.afficher();//afficher de Personne
     System.out.println(niveau);}
9   public void afficherHeures(){
     System.out.println(nbheures);}
11  }
```

Héritage

- J'instancie des objets Personne, Etudiant et Employe

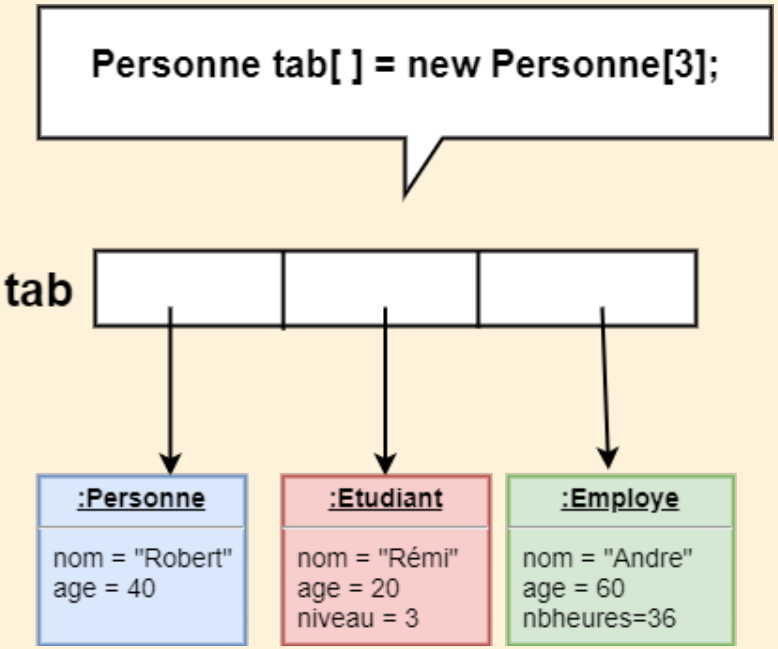
```
1  Personne p1=new Personne("Robert",40);
   Personne p2=new Etudiant("Remi",20,3);
3  Personne p3=new Employe("Andre",60,36);
   p1.afficher(); // affiche une personne
5  p2.afficher(); // affiche un etudiant
   p3.afficher(); // affiche un employe
```

- A l'aide d'un tableau de Personne $\Rightarrow$.

```
   Personne tab[]=new Personne[3];
2  tab[0]=new Personne("Robert",40);
   tab[1]=new Etudiant("Remi",20,3);
4  tab[2]=new Employe("Andre",60,36);
   for (int i=0; i<3;i++)
6      tab[i].afficher();
```

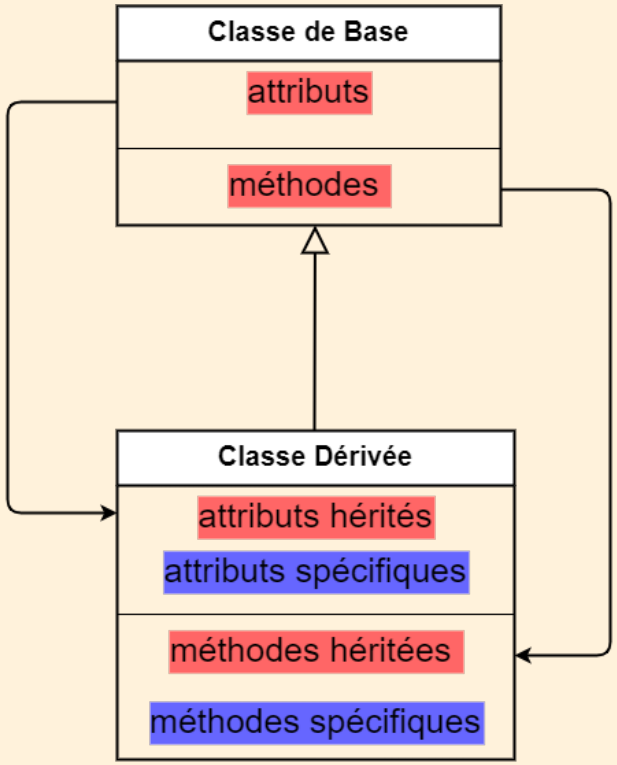
Gestion d'une collection avec un tableau I

- Dans un tableau de Personne, on peut stocker des objets : Personne, Etudiant et Employe



Héritage I

1. L'héritage a pour but d'éviter la réecriture inutile de code lorsqu'une classe n'est qu'un cas spécial d'une autre classe :



2. La classe Derivee hérite les attributs et les méthodes de la classe de base.

Héritage I

L'héritage peut être réalisé :

- Soit par ajout d'informations indépendemment de la classe mère.
- Soit par ajout de propriétés sans ajouter d'informations.
- Il ne faut pas confondre composition et héritage.
- Il est absurde de dire que une **Voiture** hérite d'une **Roue**.
- Une voiture est composée de quatre roues. La classe Voiture contient une composante de classe "Roue".

Héritage I

1. Soit la classe de base Base avec une fonction afficher()

```
class Base {
    public Base(){
    }
    public void afficher(){
    System.out.println("Jesuisdansla classe de base");
    }
}
```

2. Soit la classe Derivee qui hérite de la classe Base avec une méthode afficher.

```
class Derivee extends public Base{
    public Derivee(){
    }
    public void afficher(){
    System.out.println("Jesuisdansla classe Derivee");
}
}
```

Héritage I

- Toutes les méthodes de la classe de base sont virtuelles.

```
1  Base objet1=new Base();
   Base objet2=new Derive();
3  objet1.afficher(); // afficher de Base
   objet2.afficher(); // afficher de Derive
```

- La fonction afficher de la classe de base est masquée si l'objet est une instance de la classe Derivee.
- Les fonctions de la classe de Base sont virtuelles

Exemple d'héritage avec une classe Java : JFrame I

- Ma classe Mafenetre hérite de JFrame

```
import javax.swing.JFrame;
class Mafenetre extends JFrame {
// Constructeur
public Mafenetre(String titre){
    this.setTitle(titre);
    this.setSize(200,300);
    this.setVisible(true);
}}
```

- Pour tester ma fenétre Mafenetre

```
class TestMafenetre{
public static void main(String arg[]){
  Mafenetre f=new Mafenetre("Mapremierefenetre");
}}
```

Un deuxième exemple d'héritage avec "super" I

- Voici un deuxième exemple d'héritage avec JFrame

```
public class FenetreDessin extends JFrame{
// Constructeur
public FenetreDessin(){
this.setSize(200, 300);
this.setTitle("FenetredeDessin");
 }
```

- Je redéfinie la méthode paint de la classe JFrame.

```
// methode paint
public void paint(Graphics g){
super.paint(g);
g.setColor(Color.PINK);
g.drawRect(40, 50, 80, 40);
g.setColor(Color.BLACK);
g.drawString("Bonjour", 50, 60);
}
```

Utiliser final pour une méthode I

- Les méthodes déclarées avec le mot clé **final** ne peuvent être redéfinies

```
class A{
final void afficher(){
System.out.println("Voiciunemethodefinal");
}
}
```

- On ne peut pas reéfinir la méthode afficher() dans la calsse B

```
class B extends A{
void afficher(){
System.out.println("Impossible");
}
}
```

final et héritage I

- Pour empêcher l'héritage d'une classe, on la déclare **final** .
- Toutes les méthodes sont alors implicitement *final*.

```
1  final class A{
       ..............
3  }
```

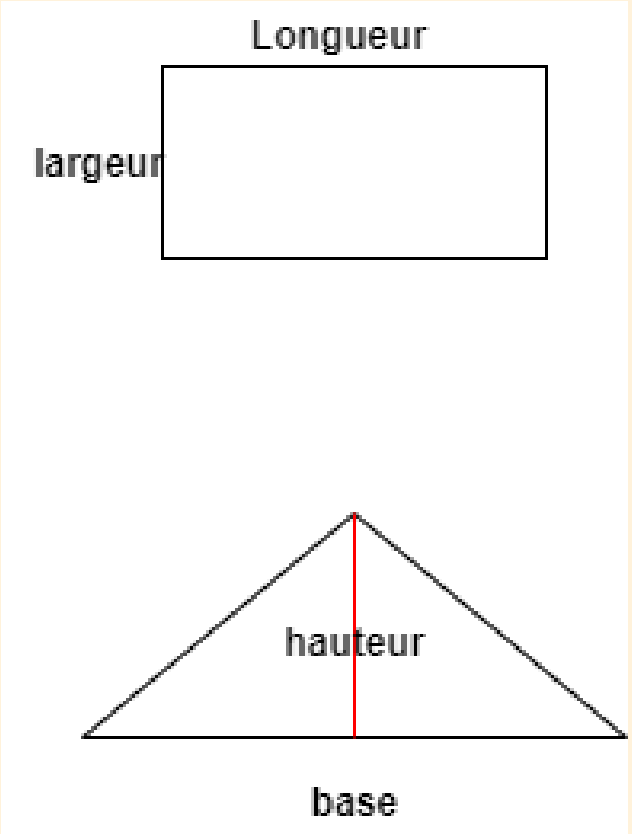
- La classe suivante n'est pas valide

```
1  class B extends A{
   //Erreur:impossible de creer une sous-classe de A
3  }
```

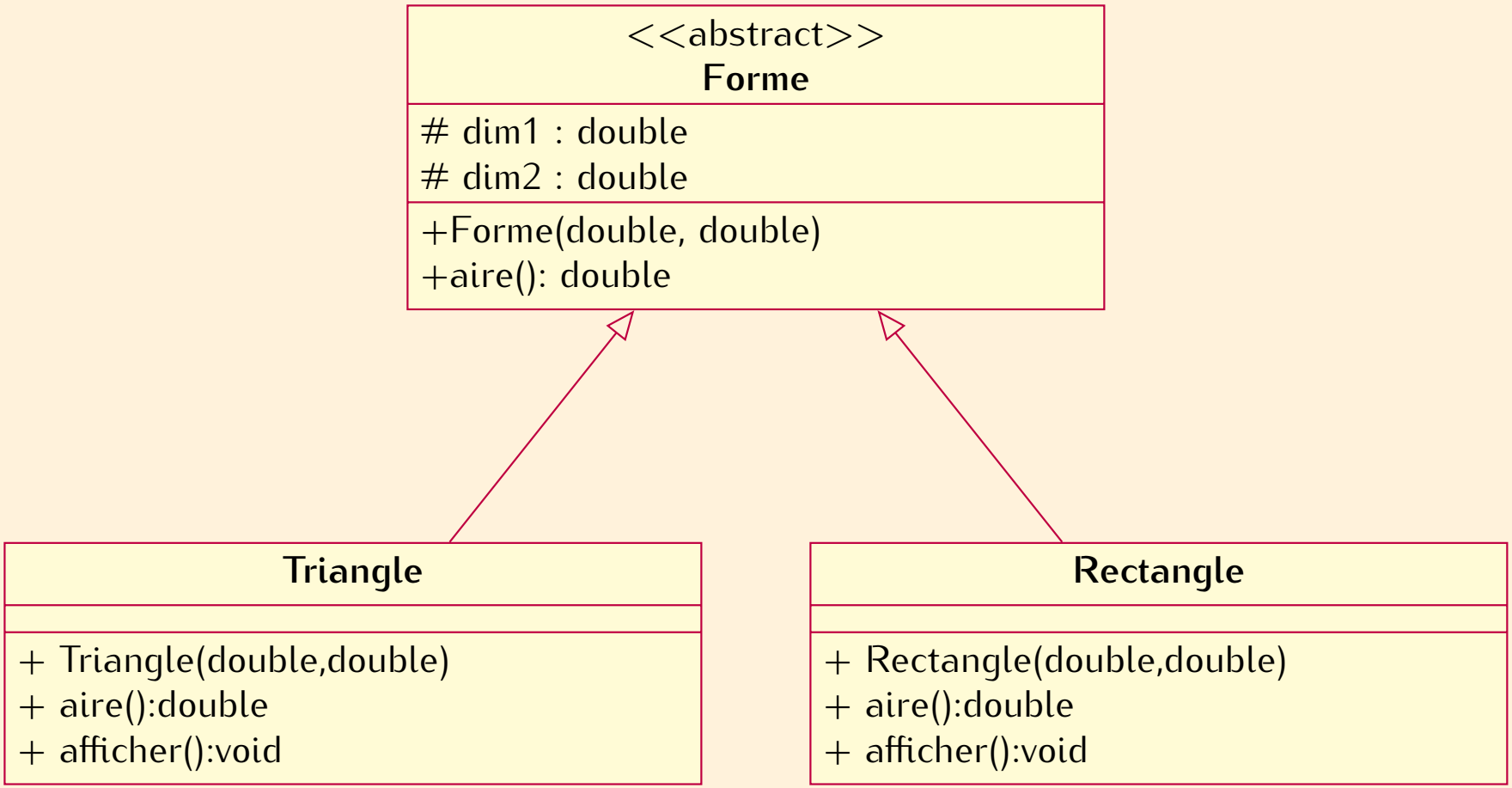
Classe abstraite I

- Pour éviter de définir dans chaque classe des méthodes communes, on peut les **factoriser**, en les isolant dans une **classe abstraite**.
- Une classe abstraite est une sorte de prototype qui sert à définir d'autres classes qui en hériteront et qui implémenteront les méthodes abstraites.
- Une méthode abstraite est une méthode définie uniquement par son prototype, sans ajouter la définition de son code.

Classe abstraite I



Classe abstraite I



Classe abstraite I

- la classe **Forme** comprend une méthode abstraite **aire**.

```
1  abstract class Forme{
     protected double dim1;
3    protected double dim2 ;
     // Constructeur
5    public Forme(double a, double b){
       dim1=a,
7      dim2=b;
     }
9    //methode abstraite
     public abstract double aire();
11 }
```

- une classe abstraite ne peut être instanciée, elle peut servir de référence vers un objet d'une sous-classe.

Classe abstraite I

- La classe **Rectangle** étend la classe Forme et définit la méthode abstraite.

```
1  class Rectangle extends Forme{
   //Constructeur
3  public Rectangle(double longueur, double largeur){
     super(longuer,largeur);
5  }
   //Methode aire
7  public double aire(){
     return longueur*largeur;
9  }
   }
```

- La classe **Rectangle** doît implémenter la méthode aire()

Classe abstraite I

- La classe Triangle étend la classe Forme et définit la méthode abstraite.

```
class Triangle extends Forme{
// Constructeur
public Triangle(double base, double hauteur)
{
  super(base,hauteur);
}
// Methode aire
public double aire(){
  return base*hauteur/2;
}}
```

- La classe Triangle doît implémenter la méthode aire()

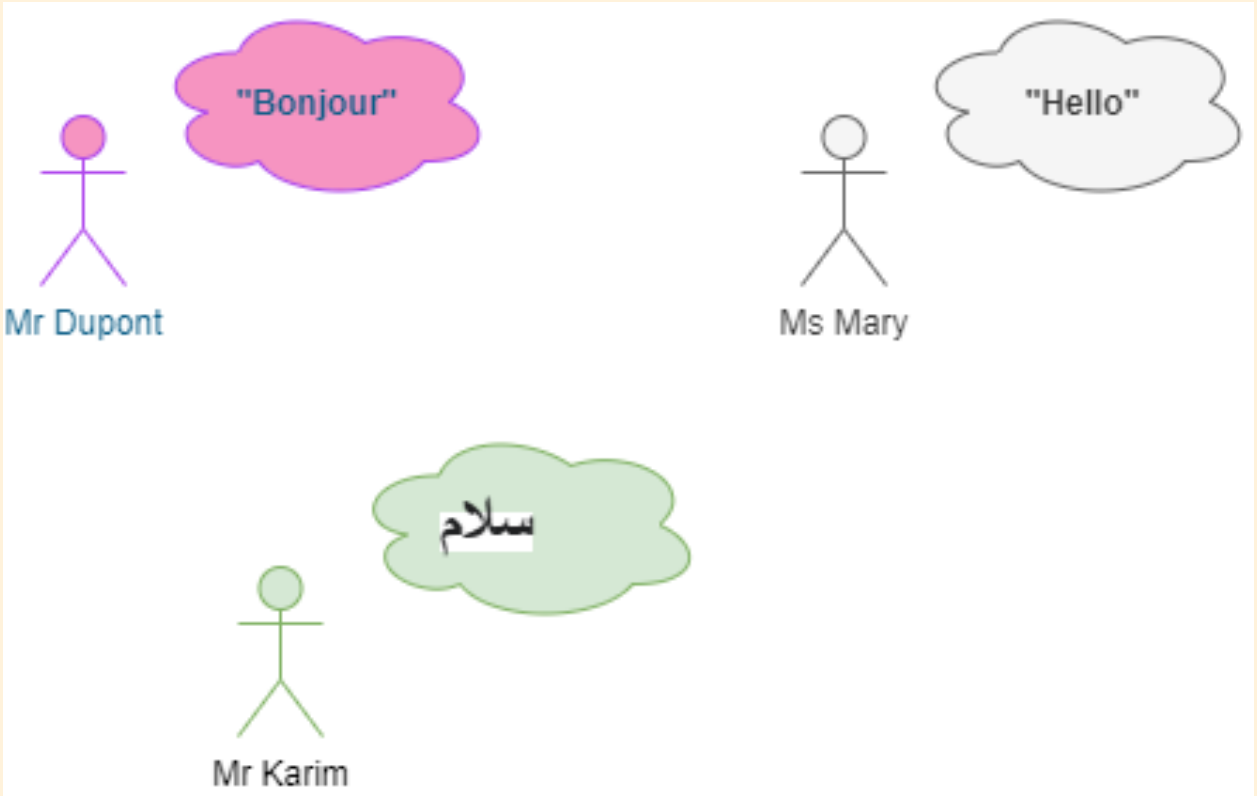
Classe abstraite I

On peut stocker des triangle et des rectangle dans un tableau de forme

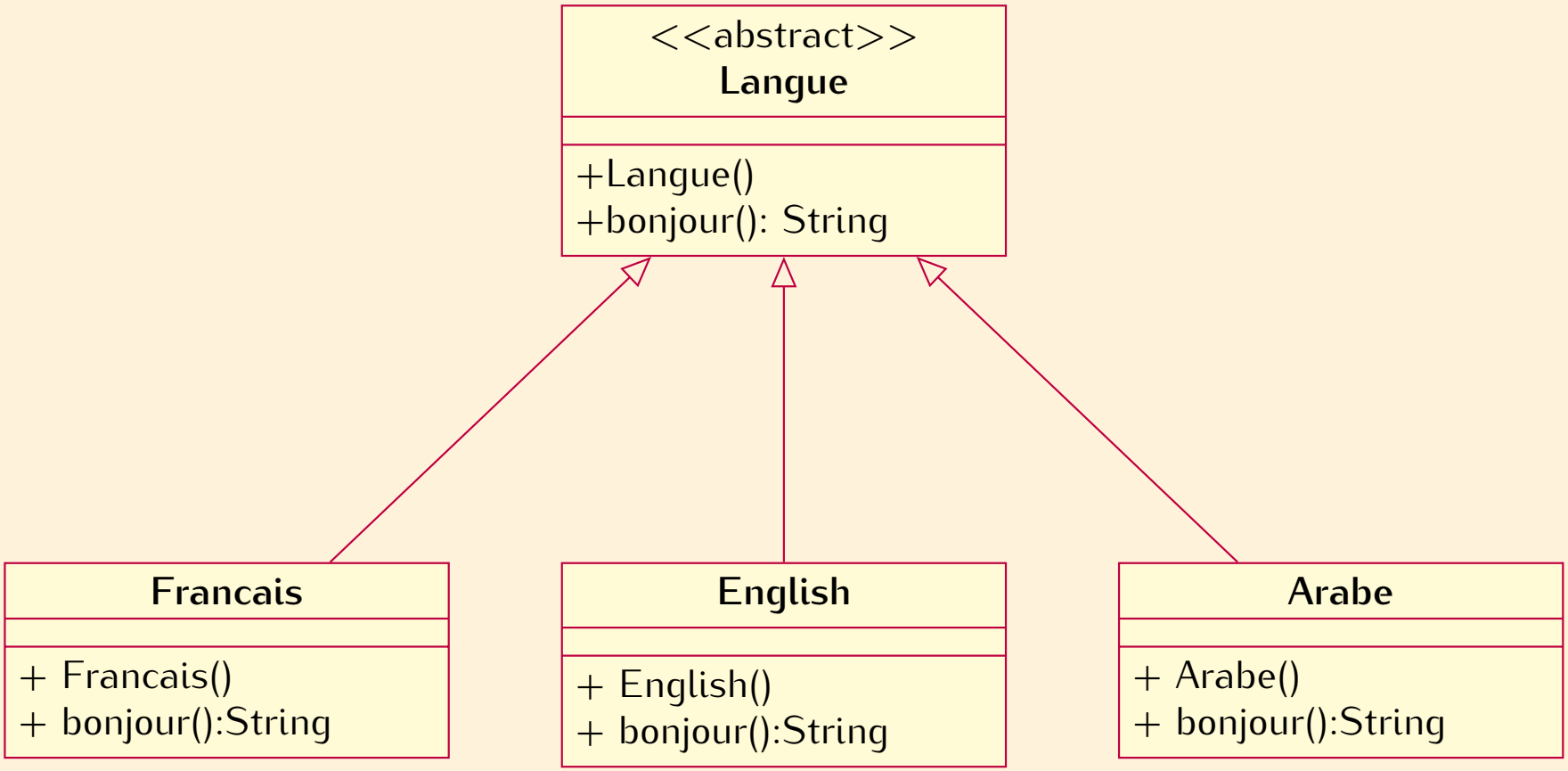
- La classe **EssaiFormes** ci-dessous permet de tester

```
class EssaiFormes{
  public static void main(String[] argv){
  Forme forme[]=new Forme[3];
  forme[0]=new Rectangle(2,1);
  forme[1]=new Triangle(4,1);
  forme[2]=new Rectangle(3,1);
  for(int i=0;i<3;i++)
  System.out.println("l'aireest" +forme[i].aire();
}
```

Classe abstraite I



Classe abstraite I



Classe abstraite I

- la classe **Langue** comprend une méthode abstraite **bonjour()**.

```
1  abstract class Langue{
     // Constructeur
3    public Langue(){
     }
5    //methode abstraite
     public abstract String bonjour();
7  }
```

- Je ne sais pas comment dire bonjour dans une langue qui n'est pas connue.

Classe abstraite I

- La classe Francais étend la classe Langue et définit la méthode abstraite.

```
1  class Francais extends Langue{
   // Constructeur
3  public Francais(){
   }
5  // Methode bonjour
   public String bonjour(){
7    return "Bonjour";
   }}
```

- La classe Francais doît implémenter la méthode bonjour()

Classe abstraite I

- La classe English étend la classe Langue et définit la méthode abstraite.

```
class English extends Langue{
// Constructeur
public English(){
}
// Methode bonjour
public String bonjour(){
  return "GoodMorning";
}}
```

- La classe English doît implémenter la méthode bonjour()

Classe abstraite I

- La classe Arabe étend la classe Langue et définit la méthode abstraite.

```
class Arabe extends Langue{
// Constructeur
public Arabe(){
}
// Methode bonjour
public String bonjour(){
  return "Salam";
}}
```

- La classe Arabe doît implémenter la méthode bonjour()

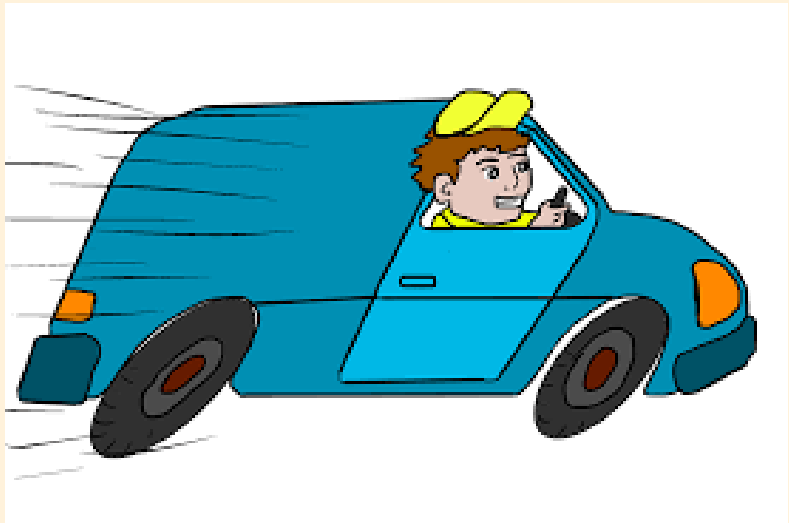
Q'est ce qu'une interface I

- En Java, il n y a que l'héritage simple. Chaque classe ne peut hériter que d'une classe.
- Il est possible à une classe dérivée d'**implémenter** une ou plusieurs classes abstraites particulières appelée **Interface**.
  - Toutes les méthodes d'une interface sont **abstraites** et **public**.
  - Une interface ne définit aucune variable d'instance.
  - Les seules variables sont des variables **static** et constantes avec le modificateur **final**

Qu'est ce qu'une interface ? I

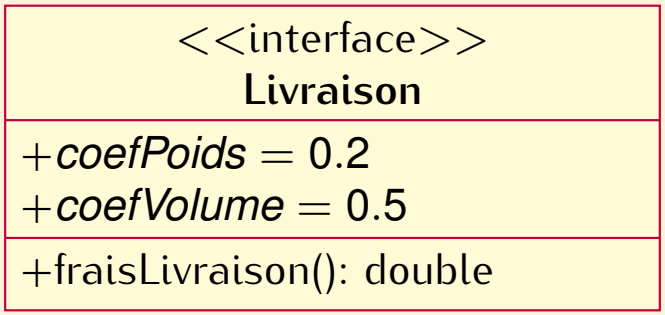
- Si une classe A implémente une interface I,
  - les sous-classes de A implémentent aussi I.
  - toutes les méthodes de I doivent être définies et déclarées publiques par la classe A.
- Une classe **hérite** au plus d'une super-classe mais elle peut **implémenter plusieurs interfaces**
- Des classes sans rapport entre elles en terme d'hiérarchie peuvent implémenter une même interface

Classe abstraite I



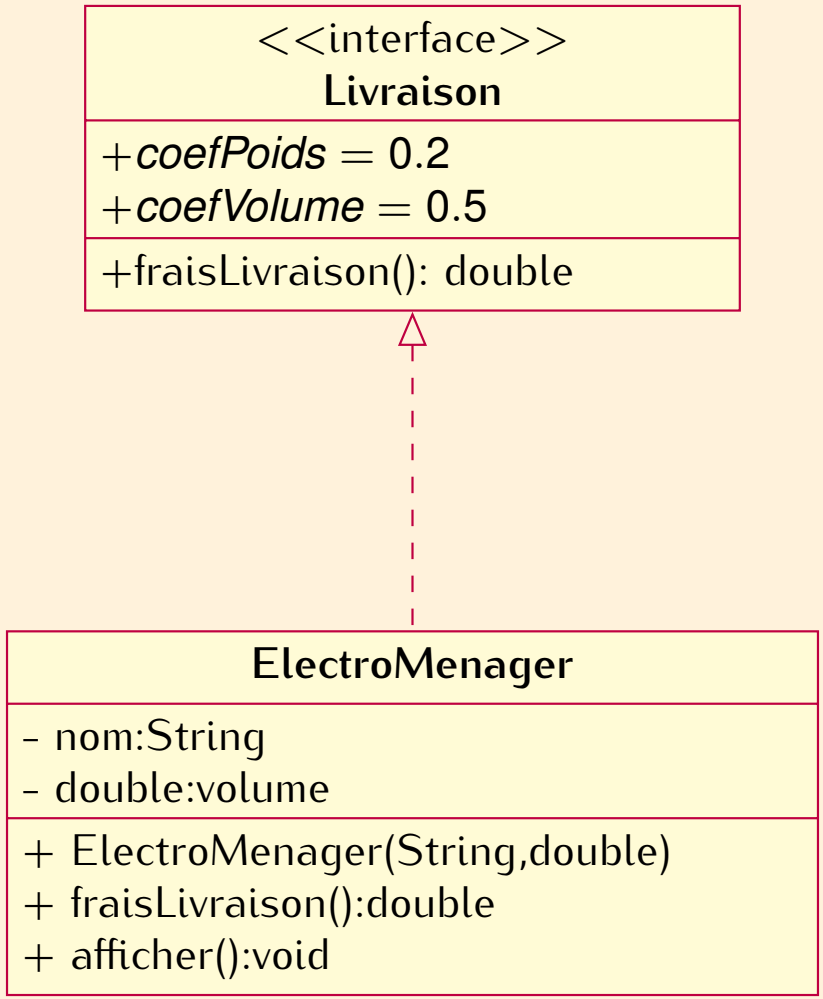
Interface Livraison I

Problème : On veut calculer les frais de livraison de deux types de produits : lélectroménager en fonction du volume et les pièces mécaniques en fonction du poids.




```
public interface Livraison {
public static final double coefPoids = 0.2;
public static final double coefVolume =0.5;
public double fraisLivraison();
}
```

Classe ElectroMenager implémente Livraison I

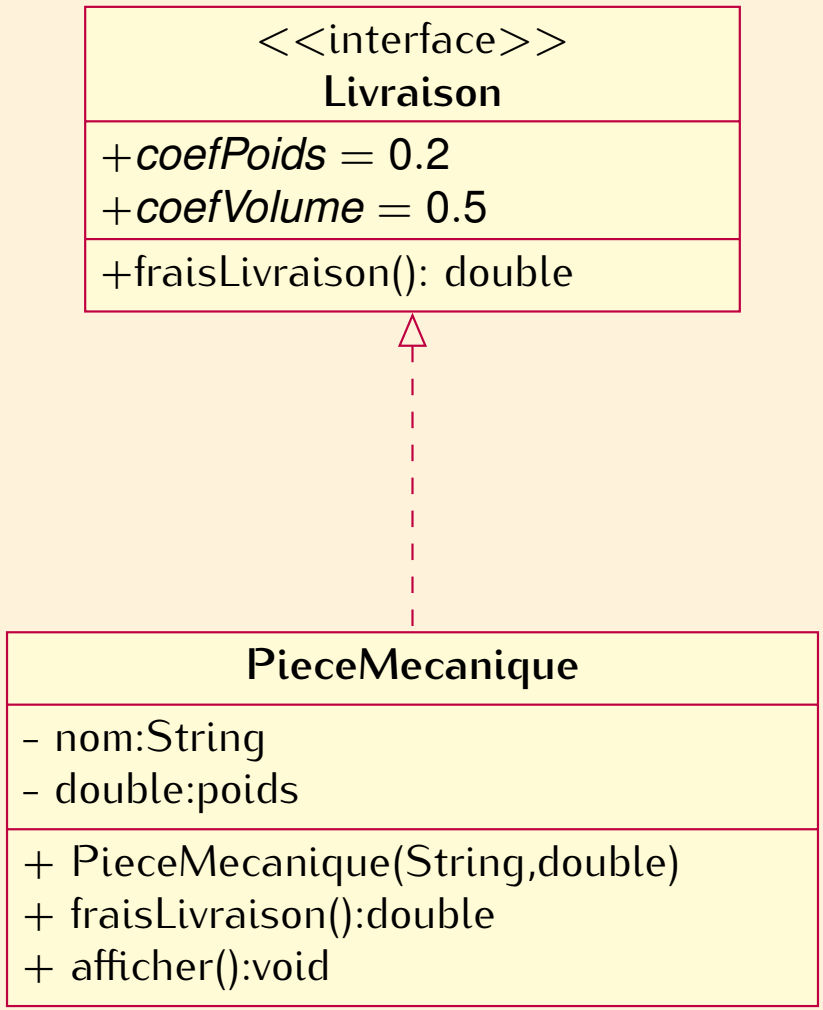


Classe ElectroMenager I

La classe ElectroMenager doit donner corps à la méthode fraisLivraison()

```
1  public class ElectroMenager implements Livraison{
   private String nom;
3  private double volume;
   // constructeur
5  public ElectroMenager(String nom, double volume) {
     this.nom = nom;
7    this.volume = volume;
     }
9      // fraisLivraison
   public double fraisLivraison() {
11   return coefVolume*volume;
     }
13   // afficher ()
   public void afficher(){
15  System.out.println(nom + ""+ volume + "frais="+
        fraisLivraison());
     }
```

Classe PieceMecanique implémente Livraison I



Classe PieceMecanique I

La classe PieceMecanique doit donner corps à la méthode fraisLivraison()

```
public class PieceMecanique implements Livraison{
  private String nom;
  private double poids;
    // constructeur
  public PieceMecanique(String nom, double poids) {
    this.nom = nom;
    this.poids = poids;
  }
    // fraisLivraison
  public double fraisLivraison() {
    return coefPoids*poids;
  }
  // afficher
  public void afficher()
  {
    System.out.println(nom + ""+ poids + "frais="+
        fraisLivraison());
  }
}
```

Conversion entre types I

Java distingue 3 types de conversions

- Si les variables sont de type primitif, les conversions se font comme en langage C (conversion implicite et explicite avec le cast)
- Si les variables sont de type primitif et de type classe

```
Integer val =new Integer(48);
int nb=val.intValue();
```

- Si les variables sont de type classe, les conversions possibles sont celles qui concernent des classes d'un même arbre d'héritage, uniquement de manière ascendante : transtypage

Exemple de transtypage I

- transtypage implicite

```
Employe empl= new Employe("toto",1998);
Etudiant etud= new Etudiant("Remi",
      2000,"alain_Remi@hotmal.com");
empl=etud // c'est possible
etud=empl // interdit
```

- transtypage explicite

```
Object o;
Employe empl;
o=new Etudiant("Andre", 2004, "andre@yahoo.fr");
empl=o // refuse la compilation
empl=(Etudiant) o; // ok
empl.afficher();
```

- ok à la compilation, mais erreur à l'exécution

```
Employe emp= new Employe("Sadio", 2002);
Etudiant etud= (Etudiant) empl;
etud.afficher();
```

Exemple de transtypage II

- une double vérification à la compilation et à l'exécution

```
1  Employe c= new Etudiant("toto",1998,"
       toto@hotmail.com");
   if (c instanceof Etudiant)
3  { Etudiant etud= (Etudiant) empl;
     etud.afficher()
5  }
   else System.out.println("probleme");
```