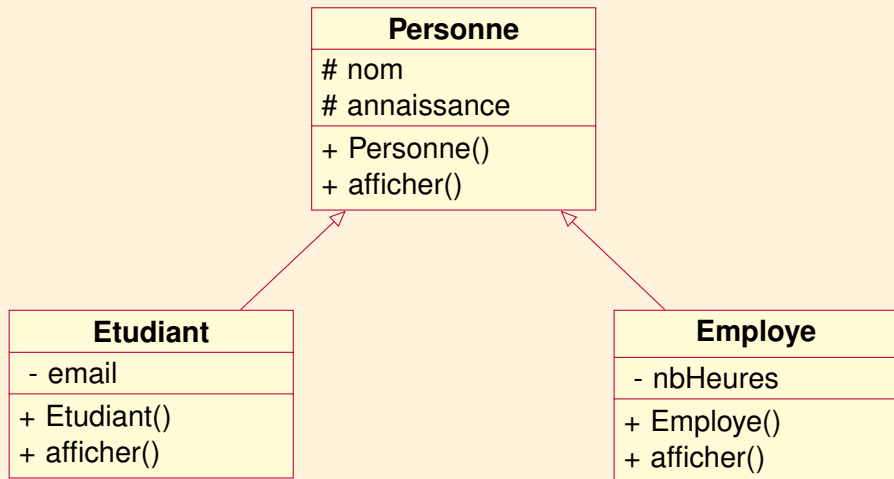


- Héritage
- Classe abstraite
- Interface

# Héritage I



L'héritage peut être réalisé :

- Soit par **ajout d'informations** indépendamment 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".

# Un premier exemple d'héritage I

Les attributs sont déclarés **protected** pour l'héritage

Personne
#nom
#naissance
+Personne()
+afficher()

```
1  class Personne{
   protected String nom;
3   protected int naissance;

5   // constructeur
   public Personne(String name, int a){
7       nom= name ;
       naissance=a;
9   }

   // fonction afficher
11  public void afficher(){
       System.out.println(nom+naissance);
13  }
}
```

# Etudiant hérite de Personne I

- La classe **Etudiant** hérite de la classe **Personne**.

```
class Etudiant extends Personne{
2 private String email;
  // constructeur
4 public Etudiant(string ch,int a,string e)
  {
6     super(ch,a);
    email=e;
8 }
  // methode afficher
10 public void afficher(){
    super.afficher();
12 System.out.println("email_: "+email);
}
```

- Pour construire un étudiant, j'appelle le constructeur de **Personne** avec **super**
- Pour afficher un étudiant, j'appelle d'abord la fonction afficher de **Personne** avec **super**

# Employe hérite de Personne I

- La classe **Employe** hérite de la classe **Personne**.

```
1  class Employe extends Personne{  
    private double nbheure;  
3  // constructeur  
    public Employe(string ch,int a,double nb)  
5  {  
        super(ch,a);  
7    nbheure=nb;  
    }  
9  // Methode afficher  
    public void afficher(){  
11     super.afficher();  
        System.out.println("nbheure_: "+nbheure);  
13     }  
}
```

- Pour construire un employé, j'appelle le constructeur de **Personne** avec **super**
- Pour afficher un employé, j'appelle d'abord la fonction afficher de **Personne** avec **super**

# Exemple d'héritage avec JFrame I

- Ma classe **Mafenetre** hérite de **JFrame**

```
1  import javax.swing.JFrame;
   class Mafenetre extends JFrame {
3  // Constructeur
   public Mafenetre(){
5      this.setSize(200,300);
      this.setTitle("ma_premiere_fenetre")
          ;
7  }}
```

- Pour tester ma fenêtre **Mafenetre**

```
1  class TestMafenetre{
   public static void main(String arg[]){
3      Mafenetre f=new Mafenetre();
      f.setVisible(true);
5  }}
```

# Un deuxième exemple d'héritage avec "super" I

- Voici un deuxième exemple d'héritage avec **JFrame**

```
1 public class FenetreDessin extends JFrame{  
    // Constructeur  
2 public FenetreDessin(){  
    this.setSize(200, 300);  
3 this.setTitle("Fenetre_de_Dessin");  
4 }  
5 }
```

- Je redéfinie la méthode **paint** de la classe JFrame.

```
    // methode paint  
2 public void paint(Graphics g){  
    super.paint(g);  
3 g.setColor(Color.PINK);  
4 g.drawRect(40, 50, 80, 40);  
5 g.setColor(Color.BLACK);  
6 g.drawString("Bonjour", 50, 60);  
7 }  
8 }
```

# Utiliser final pour une méthode I

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

```
class A{  
2  final void afficher(){  
    System.out.println("Voici_une_  
        methode_final");  
4  }  
}
```

- On ne peut pas réécrire la méthode afficher() dans la classe B

```
1  class B extends A{  
    void afficher(){  
3  System.out.println("Impossible"  
        );  
    }  
5  }
```

# 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  .....  
}
```

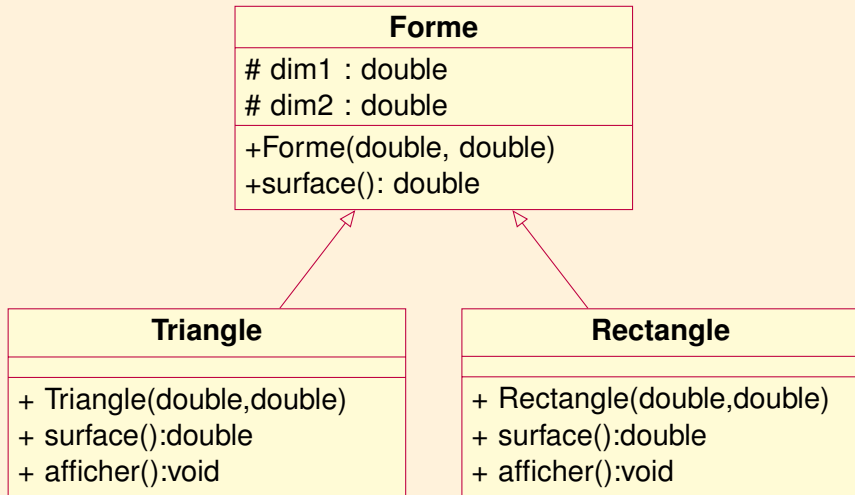
- La classe suivante n'est pas valide

```
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

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



```
1  abstract class Forme{  
    protected double dim1;  
3   protected double dim2 ;  
    // Constructeur  
5   Forme(double a, double b){  
        dim1=a,  
7        dim2=b;  
    }  
9    //methode abstraite  
    abstract double surface();  
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  Rectangle(double longueur, double  
        largeur){  
        super(longueur, largeur);  
5  }  
    //Methode Surface  
7  double surface(){  
        return longueur*largeur;  
9  }  
}
```

- La classe **Rectangle** doit implémenter la méthode `surface()`

# Classe abstraite I

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

```
class Triangle extends Forme{  
2  // Constructeur  
  Triangle(double base, double  
    hauteur)  
4  {  
    super(base, hauteur);  
6  }  
  // Methode surface  
8  double surface(){  
    return base*hauteur/2;  
10 }  
}
```

- La classe **Triangle** doit implémenter la méthode `surface()`

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

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

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

# 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'hérarchie peuvent implémenter une même interface

# 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.

<<interface>>  
**Livraison**

+*coefPoids* = 0.2

+*coefVolume* = 0.5

+*fraisLivraison()*: double

```
1 public interface Livraison {  
  public static final double coefPoids = 0.2;  
3  public static final double coefVolume =0.5;  
  public double fraisLivraison();  
5 }
```

# Classe ElectroMenager implémente Livraison I

<<interface>>

**Livraison**

+coefPoids = 0.2

+coefVolume = 0.5

+fraisLivraison(): double



**ElectroMenager**

- nom:String

- double:volume

+ ElectroMenager(String,double)

+ fraisLivraison():double

+ afficher():void

# Classe ElectroMenager I

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

```
1  public class ElectroMenager implements Livraison{
2      private String nom;
3      private double volume;
4      // constructeur
5      public ElectroMenager(String nom, double volume) {
6          super();
7          this.nom = nom;
8          this.volume = volume;
9      }
10     // fraisLivraison
11     public double fraisLivraison() {
12         return coefVolume*volume;
13     }
14     // afficher()
15     public void afficher(){
16         System.out.println(nom + "_" + volume + "frais_" +
17             fraisLivraison());
18     }
```

# Classe PieceMecanique implémente Livraison I

<<interface>>

**Livraison**

+coefPoids = 0.2

+coefVolume = 0.5

+fraisLivraison(): double



**PieceMecanique**

- nom:String

- double:poids

+ PieceMecanique(String,double)

+ fraisLivraison():double

+ afficher():void

# Classe PieceMecanique I

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

```
1 public class PieceMecanique implements Livraison{
   private String nom;
3   private double poids;
   // constructeur
5   public PieceMecanique(String nom, double poids) {
       super();
7       this.nom = nom;
       this.poids = poids;
9   }
   // fraisLivraison
11  public double fraisLivraison() {
       return coefPoids*poids;
13  }
   // afficher
15  public void afficher()
   {
17      System.out.println(nom + "␣" + poids + "frais␣=" +
          fraisLivraison());
   }
19 }
```

# 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

```
1 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

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

## ■ transtypage explicite

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

## ■ ok à la compilation, mais erreur à l'exécution

```
Employee emp= new Employee("Sadio", 2002);  
2 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");
```