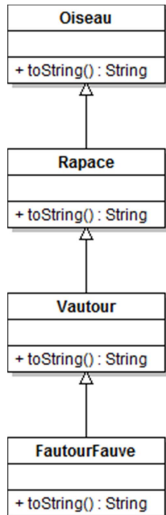


## 14) L'héritage

L'héritage est un concept primordial dans la programmation orientée objets, il sert à créer une classe à partir d'une autre classe ce qui permet **de réutiliser du code existant**.

**Exercice 13 : Saisir et tester le code suivant dans une nouvelle application :**



```

package oiseauxheritage;

public class Oiseau {
    public Oiseau() {}

    public String toString() {
        return "--> Je suis un oiseau\n";
    }
}
  
```

```

public class Rapace extends Oiseau{
    public Rapace() {
        super();
    }

    @Override
    public String toString(){
        return super.toString() + "\t--> Je suis un rapace\n\t";
    }
}
  
```

```

public class Vautour extends Rapace{
    public Vautour() {
        super();
    }

    @Override
    public String toString(){
        return super.toString() + "\t--> J'appartiens à la famille des vautours\n\t\t";
    }
}
  
```

```

public class VautourFauve extends Vautour{
    public VautourFauve() {
        super();
    }

    @Override
    public String toString(){
        return super.toString() + "\t--> Je suis un vautour fauve";
    }
}
  
```

```

public class OiseauxHeritage {

    public static void main(String[] args) {
        VautourFauve vf = new VautourFauve();
        System.out.println(vf.toString());
    }
}
  
```

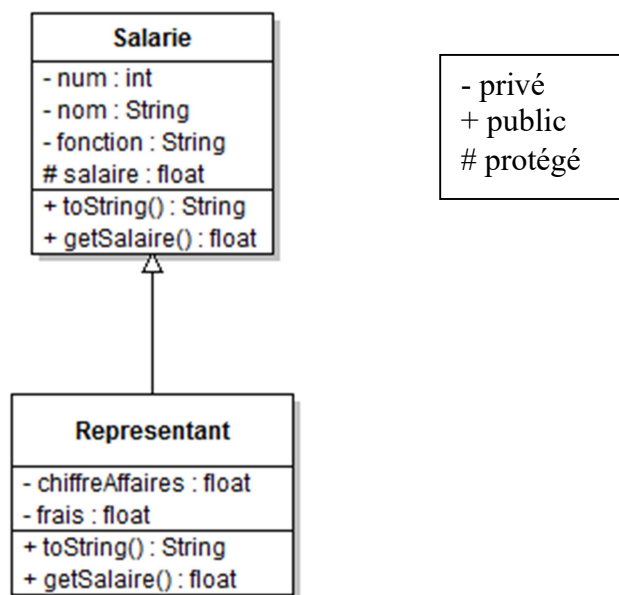
run:

```

--> Je suis un oiseau
    --> Je suis un rapace
        --> J'appartiens à la famille des vautours
            --> Je suis un vautour fauve
  
```

Le formalisme de l'héritage est représenté de la façon suivante au format UML :

La classe Représentant **hérite** de la classe Salarie ce qui signifie qu'un représentant "**est un**" salarié avec des attributs spécifiques.



Dans ce contexte un représentant est un salarié avec des attributs spécifiques (un chiffre d'affaires et des frais). Par polymorphisme il peut contenir des méthodes qui portent le même nom que des méthodes de sa classe mère (par exemple **getSalaire()**).

### Attention de ne pas confondre la surcharge et le polymorphisme !

**surcharge** : même nom de méthode au sein d'une même classe (mais nombre et/ou types des paramètres différents)

**polymorphisme** : même nom de méthode dans la classe fille et la classe mère dans le cadre d'un héritage. (poly = plusieurs morphe = forme en grec)

Dans le contexte de l'héritage, il existe une 3<sup>ème</sup> famille d'état pour déclarer les propriétés et les méthodes (en plus de privé et public) : l'état "**protégé**" (# en UML).

La classe mère peut accorder à ses classes dérivées l'accès de certains de ses attributs et méthodes non publics. Pour cela elle doit les déclarer en "**protégé**".

Une méthode de la classe dérivée accède donc :

- aux membres privés (propriétés et méthodes) de sa propre classe,
- aux membres publics de sa propre classe,
- aux membres publics de la classe mère,
- aux membres protégés de la classe mère.

**Mais pas aux membres privés de la classe mère !**

Codage de l'héritage en java

La classe "**mère**" Salarie

```
package salarieheritage;

public class Salarie {
    //Propriétés privées
    private int num;
    private String nom;
    private String fonction;

    //Propriétés protégées
    protected float salaire;

    //Constructeur
    public Salarie(int n, String nom, String f, float s){
        this.num = n;
        this.nom = nom;
        this.fonction = f;
        this.salaire = s;
    }

    //Méthodes publiques
    public float getSalaire(){
        //A compléter
    }

    @Override
    public String toString(){
        //A compléter
    }
}
```

La classe "fille" Representant :

```
package salarieheritage;

public class Representant extends Salarie{
    //Propriétés privées
    private float chiffreAffaires;
    private float frais;

    //Constructeur
    public Representant(int n, String nom, String f, float s, float ca, float fr){
        //Création d'un objet de la classe mère
        super(n, nom, f, s);

        this.chiffreAffaires = ca;
        this.frais = fr;
    }

    //Méthodes publiques
    @Override
    public float getSalaire(){
        //A compléter
    }

    @Override
    public String toString(){
        //A compléter
    }
}
```

Le constructeur reçoit en paramètres les propriétés d'un salarié (puisque c'en est un) plus ses propriétés spécifiques

Le mot clé @Override indique que l'on redéfinit par polymorphisme une méthode héritée

Le mot clé **extends** indique qu'il s'agit d'un héritage


Le mot clé **super** permet d'accéder à l'objet de la classe mère (donc aux membres publics et protégés )

Par exemple :

- super**(n, nom, f, s); // Appel du constructeur de Salarie (public)
- super**.getNom(); // Appel à un accesseur de Salarie (public)
- super**.salaire; // Appel à une propriété protégée de Salarie
- super**.nom; // Interdit !

## Exercice 14 : Coder les classes Salarie et Representant

- Ecrire les méthodes getSalaire() des 2 classes ;

 Règles de gestion pour calculer le salaire d'un représentant :

**Salaire représentant = salaire de base + frais + commission**

Les frais sont plafonnés à 400 €.

Si le chiffre d'affaires est  $\geq 15000$  € la commission se monte à 7 % du salaire de base sinon à 4 %.

- Ecrire la méthode toString() de la classe Salarie renvoie une chaîne conforme à l'exemple suivant :

```
Numéro : 1
Nom : momo
Fonction : Web designer
Salaire de base : 1830.0 €
```

- Ecrire la méthode toString() de la classe Representant renvoie une chaîne conforme à l'exemple suivant :

```
Numéro : 2
Nom : manu
Fonction : Représentant secteur Ouest
Salaire de base : 1450.0 €
Chiffre d'affaires : 13000.0 €
Frais : 350.0 €
Salaire total : 1858.0 €
```

- Créer dans la classe de test la situation suivante :

Créer un salarié :	num : 1 nom : momo fonction : Web designer salaire : 1830 €
Créer un représentant :	num : 2 nom : manu fonction : représentant secteur Ouest salaire : 1450 € Chiffre d'affaires : 13000 € frais : 350 €
Créer un représentant :	num : 3 nom : mumu fonction : représentante secteur Est salaire : 1450 € Chiffre d'affaires : 18000 € frais : 550 €

- Afficher le salaire des salariés n°1 et 3 ;
- **Question** : quel est le mécanisme de l'héritage utilisé ici ?
- Créer une collection **ArrayList<Salarie> lesSalaries** et y insérer les 3 salariés ;
- Afficher les renseignements des 3 salariés à l'aide d'un for ;
- **Question** : pourquoi la collection de salarié accepte-t-elle des représentants ?

L'opérateur **instanceof** permet de tester si un objet est une instance d'une classe. Par exemple :

```
Article a = new Article("VTT1", "VTT Femme", 5, 250, 310);
if(a instanceof Article){ //renvoie true ou false
    System.out.println("a est un objet de la classe Article);
}
else{
    System.out.println("a n'est pas un objet de la classe Article);
}
```

- **Tester :**

Pour Momo, Manu et Mumu afficher s'il s'agit d'un salarié et/ou un représentant.