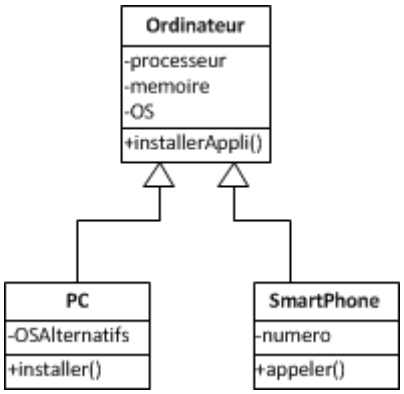
BLOC 2	Ch. 13 – Java – L'héritage

1 Généralisation, spécialisation

1.1 Héritage simple

Définition de l'héritage : mécanisme de la POO qui permet à une classe d'accéder aux caractéristiques (attributs et méthodes) d'une autre classe, dont elle hérite.

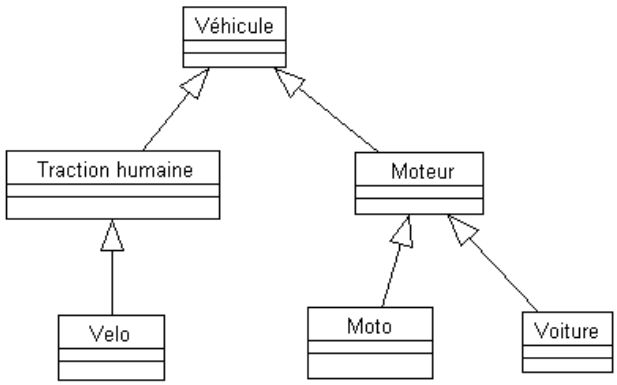
Activité 1 : Les classes **PC** et **SmartPhone** héritent de la classe **Ordinateur**, comme l'indiquent les flèches orientées.



Travail à faire : répondez par vrai ou faux.

Affirmation :	**Vrai**	**Faux**
un PC est une sorte d'Ordinateur		
un SmartPhone est une sorte d'Ordinateur		
un Ordinateur est une sorte de PC		
un Ordinateur possède un processeur, une mémoire, un OS		
un PC possède un processeur, une mémoire, un OS		
on peut installer une application sur un Ordinateur		
on peut installer une application sur un SmartPhone		
un SmartPhone possède un numéro (de tél.)		
un SmartPhone permet d'appeler		
un Ordinateur possède un numéro		
un PC permet d'appeler		

Activité 2 : répondez par vrai ou faux.

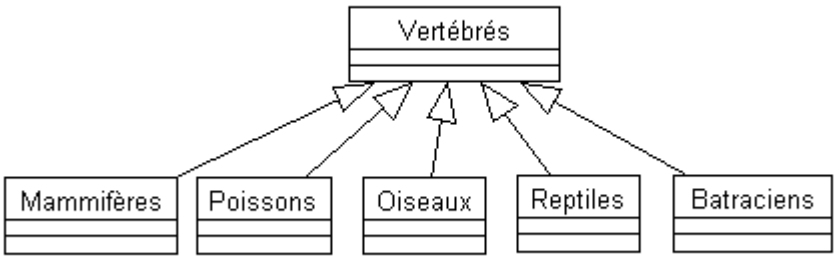


Affirmation :	**Vrai**	**Faux**
Un véhicule à moteur est une sorte de véhicule		
Une voiture est une sorte de véhicule		
Une moto peut aussi être une voiture		
La classe **Velo** hérite de **Moteur**		

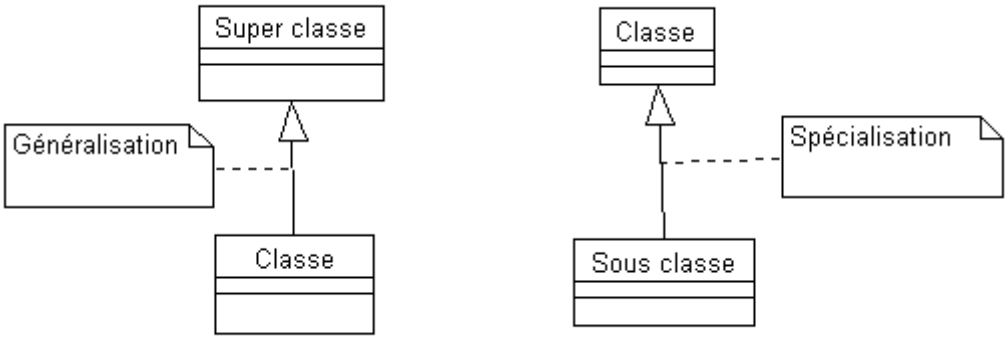
1.2 Généralisation, spécialisation

Ces notions sont très importantes dans le monde objet. Leur implémentation sous forme d'héritage est à l'origine du concept de **réutilisation**.

Le concept de généralisation est largement utilisé dans le domaine scientifique afin de modéliser un classement hiérarchique. Classement des espèces chez les biologistes, classement des végétaux chez les naturalistes, classement des objets célestes pour les physiciens astronomes.



Pour nous la spécialisation s'applique lorsqu'une classe affine les propriétés - attributs et comportements - d'une autre classe. Les termes de généralisation et spécialisation s'appliquent à un même type de relation entre classes ; on peut employer l'un ou l'autre selon le sens de lecture.



A retenir :

- La classe qui hérite est une **classe fille**, **sous-classe** ou **classe dérivée**.
- La classe héritée est une **classe mère** ou **super classe**.

Activité 3 : **Spécialisation.** Dans une université :

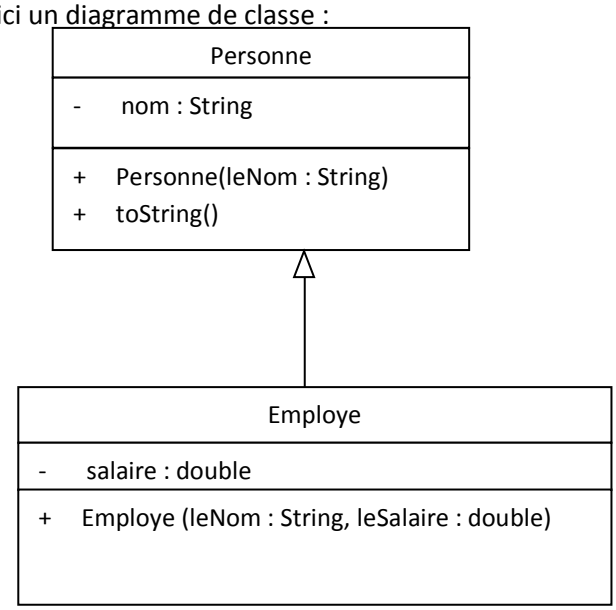
- toute personne a un numéro de sécurité sociale, un nom, un prénom, une adresse.
- Les étudiants ont une filière, un niveau d'étude ; ils ont comme comportement : suivre un cours.
- Les employés ont un service, un salaire et un comportement : calculerSalaire.
- Les enseignants ont en plus un attribut : nbHeures, et un comportement : enseigner.

Travail à faire : dessiner le diagramme de classes.

2 L'héritage en Java

2.1 Classe mère et classe fille

Voici un diagramme de classe :



Nous allons écrire en Java le code des classes **Personne** et **Employe**.

```
public class Personne
{
    private String nom;

    public Personne(String leNom)
    {
        System.out.println("Constructeur de Personne");
        this.nom = leNom;
        System.out.println("Fin Constructeur de Personne");
    }

    public String toString()
    {
        return "je m'appelle " + nom;
    }
}
```

Travail à faire : compléter les commentaires

```
// Employe             hérite de .  .  .
                            ↓
public class Employe extends Personne
{
    private double salaire;

    public Employe(String leNom, double leSalaire)
    {
        super(leNom);                 // .  .  .

        System.out.println("Constructeur de Employe");
        this.salaire = leSalaire;
        System.out.println("Fin Constructeur de Employe");
    }
}
```

Travail à faire : dire ce que le code suivant va afficher :

```
public class Program
{
    public static void main(String[] args)
    {
       Personne p1 = new Personne("Joe");

       System.out.println( p1.toString() );

    }
}
```

Affichage :

. . .

. . .

. . .

Travail à faire : dire ce que le code suivant va afficher :

```
Employe e1 = new Employe("Sarah", 2000);
System.out.println( e1.toString() );
```

. . .

. . .

. . .

. . .

. . .

Pour mémoire :

- l'héritage est réalisé par le mot clé **extends** dans la déclaration de la classe
  `classe Fille extends Mere`
- le mot clé **super** fait référence à la classe mère.

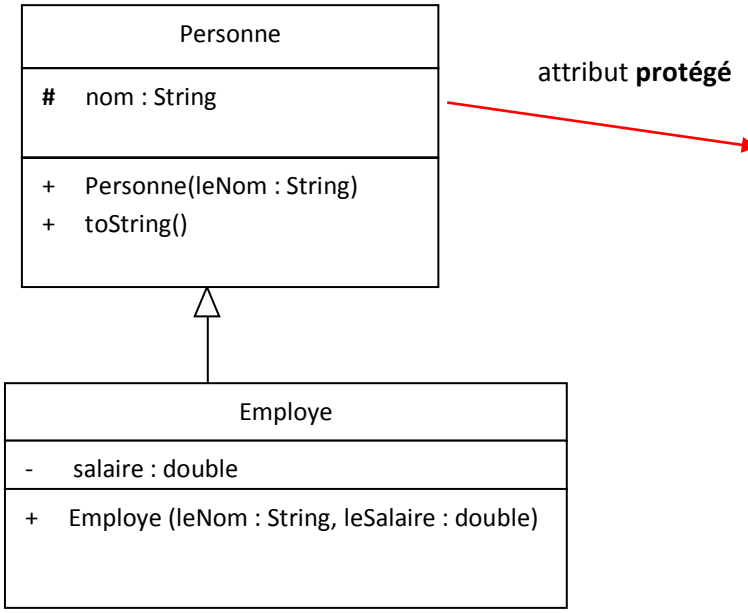
2.2 Accessibilité des membres dans les classes dérivées.

Le code suivant vous semble-t-il correct ? Trouvez les éventuelles erreurs, corrigez et justifiez.

```
public class Employe extends Personne
{
    private double salaire;

    public Employe(String leNom, double leSalaire)
    {
        nom = leNom;
        salaire = leSalaire;
    }
}
```

Le mécanisme d'héritage ne rompt pas le principe d'encapsulation. Il est interdit à toute classe dérivée d'accéder aux données privées de sa super classe. Ainsi une classe dérivée est soumise aux mêmes règles qu'une classe quelconque.
Il peut être utile parfois de donner aux classes dérivées l'accès à des données d'une super classe, ceci est possible en déclarant l'attribut **protégé** (**protected**). On écrit un **#** devant l'attribut dans le diagramme de classes.



attribut **protégé**

```
public class Personne
{
    protected String nom;
    . . .
    public String toString()
    {
        return "je m'appelle " + nom;
    }
}

public class Employe extends Personne
{
    public void sePresenter()
    {
      System.out.println(nom);
    }
}
```

Remarque : une méthode peut aussi être déclarée **protected**.

2.3 Redéfinition de méthode

On écrit le code :

```
public class Program
{
    public static void main(String[] args)
    {
        Employe e1 = new Employe("Sarah", 2000);
       System.out.println( e1.toString() );
    }
}
```

Questions :

- qu'est-ce qui s'affiche à l'écran ? . . . . . .
- quelle méthode de quelle classe est appelée ? . . . . . .

On aimerait avoir un affichage spécifique pour les employés, comme cela :

```
je m'appelle Sarah mon salaire est 2000.0
```

Pour cela on va **redéfinir** la méthode **toString()** dans la classe **Employe** :

```
public class Employe extends Personne
{
    private double salaire;
    …
    public String toString()
    {
        String debut = super.toString();              // .  .  .
        return debut + " mon salaire est " + salaire;
    }
}
```

2.4 Polymorphisme

On ajoute la méthode **afficherPersonne** dans la classe **Program** :

```
public class Program
{
    public static void main(String[] args)
    {
        Personne p1 = new Personne("Joe");
        Employe e1 = new Employe("Sarah", 2000);
        Employe e2 = new Employe("Sunny", 1800);

        afficherPersonne(p1);
        afficherPersonne(e1);
        afficherPersonne(e2);
    }

    public static void afficherPersonne(Personne unePersonne)
    {
        System.out.println(unePersonne.toString());
    }

}
```

Quand on lance le programme, qu'est-ce qui s'affiche ?

. . .

. . .

. . .

Pour chaque objet, c'est la méthode **toString()** de la classe correspondante (**Personne** ou **Employe)** qui a été invoquée.

Dans la méthode **afficherPersonne**, la variable **unePersonne** est déclarée comme étant de type **Personne**, mais l'objet qui est référencé a pour type réel **Personne** (cas de p1) ou **Employe** (cas de e1 et e2). C'est donc la méthode **toString()** de la classe de l'objet qui a été exécutée.

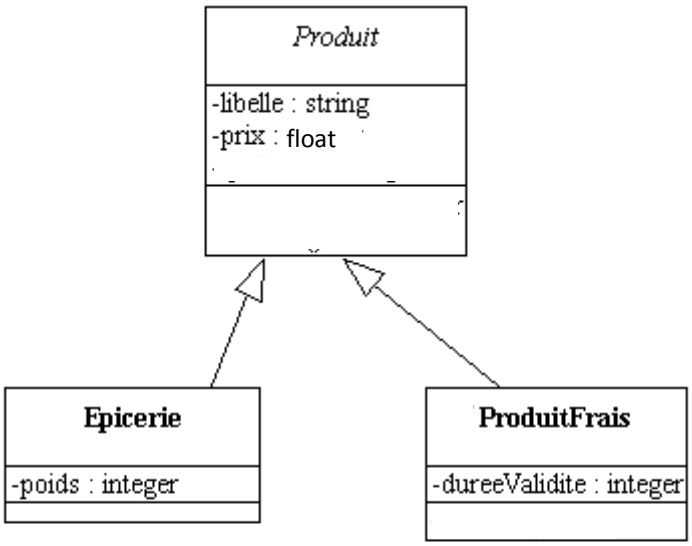
Il s'agit ici du mécanisme de **polymorphisme**.

2.5 Classe Object

En Java, la classe **Object** du package **java.lang** est la classe mère de toutes les classes. Toute classe hérite d'elle, soit directement, soit indirectement, à travers une classe mère. (voir TP).

3 Classe abstraite

Considérons une application qui gère des produits commercialisés. Imaginons que nos produits se déclinent en deux versions, des produits d'épicerie et des produits frais. Le diagramme de classes sera :



La classe **Produit** désigne un produit en général.
Aucun **Produit** ne sera instancié, en effet dans la réalité on aura soit des produits d'épicerie soit des produits frais. La classe **Produit** est dite **abstraite**. Notez sur le modèle le style "italique" qui indique une classe abstraite.

3.1 Définition

Définition : une **classe abstraite** est une classe qui ne peut être instanciée.

On définira la classe Produit ainsi : `public abstract class Produit`

On ne peut pas écrire : ~~`Produit prod = new Produit("biscuits", 1.99);`~~

3.2 Utilisation des classes abstraites.

Une classe abstraite ne peut pas se trouver à la fin d'une hiérarchie de classe. Une classe abstraite doit avoir au moins une sous-classe.

Souvent la classe abstraite ne contient pas suffisamment d'informations pour instancier un objet, autrement dit cette classe n'est présente que pour factoriser des propriétés. Il s'agit souvent d'une classe qui modélise un concept.

(exercice page suivante.)

Exercice 1. On considère un programme permettant de travailler sur la géométrie :

- La classe ***Figure*** est abstraite et elle possède deux méthodes abstraites :

```
public abstract class Figure
{
    // méthodes abstraites : elle n'ont pas de corps
    public abstract double getPerimetre() ;

    public abstract double getAire() ;

}
```

- Elle a deux classe filles (concrètes) : **Rectangle** et **Cercle** (avec les attributs adéquats).

Travail à faire :

a) Faire le diagramme de classe.
b) Ecrire le code des classes **Rectangle** et **Cercle** sans oublier l'implémentation des méthodes.