Génie logiciel II
Série de TP N°02

TP N°02 : Concepts UML vers Déclaration JAVA

I. Attributs et opérations statique

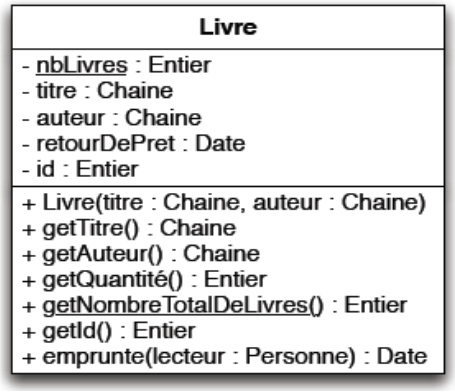
I.1. Attribut statique

Par défaut, chaque instance d'une classe possède sa propre copie des attributs de la classe. Les valeurs des attributs peuvent donc différer d'un objet à un autre. Cependant, il est parfois nécessaire de définir un attribut de classe (**static en Java**) qui garde une valeur unique et partagée par toutes les instances de la classe. Les instances ont accès à cet attribut, mais n'en possèdent pas une copie. Graphiquement, **un attribut de classe est souligné.**

I.2. Opération statique

Comme pour les attributs de classe, il est possible de déclarer des méthodes de classe. Une méthode de classe ne peut manipuler que des attributs de classe et ses propres paramètres. Cette méthode n'a pas accès aux attributs de la classe (i.e. des instances de la classe). L'accès à une méthode de classe ne nécessite pas l'existence d'une instance de cette classe. Graphiquement, une méthode de classe est soulignée.

I.3. Exemple



```java
public class Livre {
    private static int nbLivres = 0;
    private String titre;
    private String auteur;
    private int id;
    public Livre (String titre, String auteur) {
        this.titre = titre;
        this.auteur = auteur;
        id = ++nbLivres;
    }
    public static int getNombreLivres() {
        return nbLivres;
    }
    public int getId() {
        return id;
    }

    public static void main(String[] args) {
        System.out.println("Total"+getNombreLivres());
        Livre L1 = new Livre("Processus simplifié", "Pascal Rock") ;
        System.out.println("Total"+getNombreLivres());
        Livre L2 = new Livre("L'alchimiste", "Paulo Koelho") ;
        System.out.println(L1);
        System.out.println("Total"+getNombreLivres());

    }
```

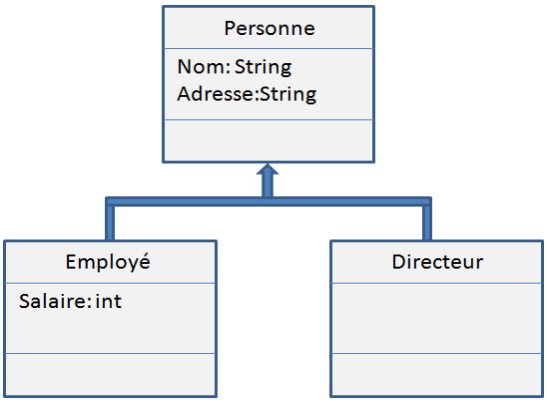
II. Héritage

Il est possible de décrire une nouvelle classe en la faisant hériter (**extend**) d'une autre classe. Cela implique que les attributs et méthodes sont hérités de la classe mère (**superclasse**) et des superclasses de celle-ci jusqu'à arriver à la classe Object, dont héritent implicitement toutes les classes.

Important : Il est impossible de faire dériver une classe de plus d'une autre classe.

II.1. Exemple :

Dans cet exemple, on a déclaré une classe personne avec les attributs nom et adresse. Les deux sous classes héritent de Personne: la première classe **Directeur** et la deuxième classe **Employé** qui a comme attribut en plus salaire.



```
class Personne
{
   protected String nom;
   protected String adresse;
}
class Employe extends Personne
{
   Private int salaire;
   public Employe(String nom, String adresse, int
salaire)
   {
       this.nom=nom;
       this.adresse=adresse;
       //ou super(nom, adresse)
       this.salaire=salaire;
   }
}
class Directeur extends Personne
{
   public Directeur()
   {
       this.nom= "nom";
       this.adresse= "adresse";
   }
}
```

III. Méthodes et classes abstraite

III.1. Méthode abstraite

Une méthode est abstraite (modificateur **abstract**) lorsqu'on la déclare, sans donner son implémentation (pas d'accolades mais un simple **« ; »** à la suite de la signature de la méthode) : **public abstract int m(String s);** La méthode sera implémentée par les classes filles.

III.2. Classe abstraite

Une classe doit être déclarée abstraite **(abstract class)** si elle contient une **méthode abstraite.** Il est interdit de créer une instance d'une classe abstraite.

Si on veut empêcher la création d'instances d'une classe on peut la déclarer abstraite même si aucune de ses méthodes n'est abstraite.

III.3. Exemple

Pour bien en comprendre l'utilité, il vous faut un exemple. Imaginez que vous êtes en train de réaliser un programme qui gère différents types des formes géométriques. Nous pouvons donc créer une classe mère : appelons **`FormeGeometrique1`**, et deux classes filles **Rectangle7,** et **Cercle7**. Le code source suivant représente nos classes.

```
abstract public class FormeGeometrique1 {
double posX, posY;
void deplacer(double x,double y) {
posX=x;
posY=y;
}
void afficherPosition() {
System.out.println("position:
("+posX+","+posY+")");
}
abstract double surface() ;
abstract double perimetre() ;
}
```

```
public class Rectangle7 extends FormeGeometrique1 {
double largeur, hauteur;
public Rectangle7() {
posX=0; posY=0; largeur=0; hauteur=0;
  }
public Rectangle7(double x,double y,double la,
double lo) {
posX=x; posY=y; largeur=la; hauteur=lo;
  }
double surface() {
return largeur * hauteur;
  }
double perimetre() {
return 2*(largeur + hauteur);
  }}
```

```
public class Cercle7 extends
FormeGeometrique1 {
double rayon;
Cercle7(double x, double y, double r) {
posX=x; posY=y; rayon=r;
  }
double surface() {
return Math.PI*Math.pow(rayon, 2.);
  }
double perimetre() {
return 2*rayon*Math.PI;
  }
}
```

```
public static void main (String args[]) {
Rectangle7 rect;
Cercle7 cerc;
rect = new Rectangle7(9,3,10,20);
cerc = new Cercle7(43,32,100);
System.out.print("cercle ");
cerc.afficherPosition();
System.out.print("rectangle ");
rect.afficherPosition();
cerc.deplacer(66,88);
System.out.print("cercle maintenant ");
cerc.afficherPosition()
}
```

IV. Redéfinition

La redéfinition est la pratique consistant à réécrire dans une classe une méthode héritée d'une autre classe. Pour qu'une méthode d'une classe B redéfinisse la méthode d'une classe A, il faut que :

— B hérite de A ;

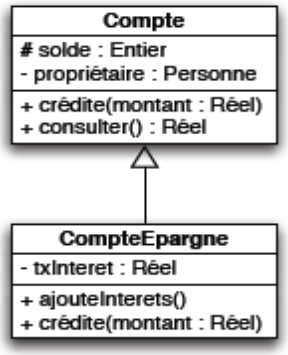
— les deux méthodes aient le même nom ;

— les deux méthodes aient les mêmes types d'arguments en paramètre d'entrée ;

— les deux méthodes aient le même type de retour ;

— les deux méthodes émettent le même type d'exception.

Dans l'exemple des comptes bancaires, si l'on souhaite que des intérêts temporaires soient calculés et ajoutés au solde du compte avant d'y créditer une somme, il faut redéfinir la méthode **credite**.



Voilà ci-dessous une implémentation possible de ce cas de redéfinition en Java. Le code de la classe **Compte** est :

```
1
2  public class Compte {
3  protected double solde;
4  private Personne Proprietaire;
5
6  public void credite (double montant) {
7      solde=solde+montant;
8  }
9  public double consulter() {
.0     return solde;
.1 }
```

Et celui de la classe **CompteEpargne** qui redéfinit la méthode **credite** :

```java
public class CompteEpargne extends Compte{
    private double txInteret;

public void credite (double montant) {
    ajouteInteret();

    solde=solde+montant;
}
public void ajouteInteret() {
    int nbjours = valeur;//valeur qlq
    double interets = (solde * txInteret)* nbjours/365;
    solde=solde+interets;
}
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

V. Surcharge

Lorsqu'on définit une classe, on peut y définir plusieurs méthodes qui **portent le même nom**. La seule contrainte est **que la liste des types des paramètres formels doit être différente**. Ce mécanisme est appelé surcharge de méthode. Il n'y a aucune contrainte sur les différents modificateurs ou sur la valeur de retour et on peut également appliquer la surcharge aux constructeurs.

VI. Polymorphisme

Le polymorphisme est la faculté de désigner un objet comme étant une instance de plusieurs classes différentes et donc d'effectuer un traitement différent pour l'appel d'un même symbole. Comme la classe **CompteEpargne** hérite de la classe Compte, toutes les instances de **CompteEpargne** sont aussi des instances de **Compte**. L'invocation de la méthode **credite** sur une instance de Compte provoquera des traitements différents selon que cette instance est une instance de **CompteEpargne** ou non. L'exemple de code Java ci-dessous montre un exemple d'utilisation du polymorphisme sur de tels objets :

```java
Vector<Compte> listeDeComptes = new Vector<Compte>();
listeDeComptes.add(new Compte());
listeDeComptes.add(new Compte());
listeDeComptes.add(new CompteEpargne());
listeDeComptes.add(new Compte());
listeDeComptes.add(new CompteEpargne());
...
for (int i = 0; i < listeDeComptes.size(); i++) {
  listeDeComptes.get(i).credite(100);
}
```

Exercice 1:

Soit les classes suivantes:

La classe **Personne** qui comporte trois champs privés, nom, prénom et date de naissance. Cette classe comporte un constructeur pour permettre d'initialiser les données. Elle comporte également une méthode polymorphe Afficher pour afficher les données de chaque personne.

Une classe **Employé** qui dérive de la classe Personne, avec en plus un champ *Salaire* accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

Une classe **Chef** qui dérive de la classe Employé, avec en plus un champ *Service* accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

Une classe **Directeur** qui dérive de la classe Chef, avec en plus un champ *Société* accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

Travail à faire:

- Créer les classes Personne, Employé, Chef, et Directeur.
- Créer une classe Test qui permet de tester les différentes classes.

Exercice 2:

On prend les classes suivantes : Etudiant, Personne et Travailleur.

1. dessinez une arborescence cohérente pour ces classes en la justifiant, où se situeront les champs suivants : salaire, anneDEtude, nom, age.

2. Considérez les classes Personne, Etudiant, Travailleur mentionnées ci-dessus. Pour chaque classe écrivez une méthode « superieur » qui compare un objet à un autre. Une personne est supérieure à une autre, si elle est plus âgée que l'autre. Un étudiant est supérieur à un autre, s'il étudie depuis plus longtemps. Un travailleur est supérieur à un autre, si son salaire est plus grand. Qu'est-ce qui se passe quand on compare un étudiant avec un travailleur?

Exercice 3:

Vous allez programmer le calcul des salaires hebdomadaires des employés d'une entreprise. Cette entreprise comporte plusieurs types d'employés :

- Des employés qui sont payés suivant le nombre d'heures qu'ils ont travaillées dans la semaine. Ils sont payés à un certain tarif horaire et leurs heures supplémentaires (au-delà de 39 heures) sont payées 30 % de plus que les heures normales.
- D'autres employés, payés de la même façon, mais leurs heures supplémentaires sont payées 50 % de plus que les heures normales.
- Les commerciaux sont payés avec une somme fixe à laquelle on ajoute 1 % du chiffre d'affaires qu'ils ont fait dans la
- semaine.

Modélisez cette situation d’un diagramme de classe :

- Vous donnerez un nom à chacun des employés. On ne pourra modifier le nom d'un employé.
- Vous commencerez par écrire une classe Employe dont hériteront les autres classes.
- Le calcul des salaires se fera dans la méthode getSalaire()

Une classe Paie comportera une unique méthode main() qui entrera les informations sur des employés des différents types. Les employés seront enregistrés dans un tableau employés.
La méthode main() affichera le salaire hebdomadaire de chacun des employés dans une boucle "for" qui parcourra le tableau des employés. Vous utiliserez le polymorphisme avec un accesseur pour le salaire. L'affichage aura exactement la forme : "nom prénom 125000 DA". Vérifier les calculs des salaires !