Bloc2 Bases de la POO	Chapitre 6 L'héritage – Notions de base

Sommaire

I. L'expression du besoin 2

II. L'héritage : classe de base et classe dérivée 3

1) La classe Secretaire 3

2) La classe Commercial 3

3) Définition : classe de base et classe dérivée 3

4) Instances de la classe de base et des classes dérivées 4

a) Les attributs des instances de la classe de base 4

b) Les attributs des instances des classes dérivées 4

c) Les méthodes appelables par une instance de la classe de base 5

d) Les méthodes appelables par une instance de la classe dérivée 5

III. Héritage et constructeur 6

IV. Héritage et UML 8

V. Redéfinition de méthode 8

1) Besoin 8

2) La redéfinition de méthode 8

2) Signature d'une méthode redéfinie (virtual et override) 9

5) Appel d'une méthode de la classe de base dans une méthode redéfinie 10

I. L'EXPRESSION DU BESOIN

Vous êtes chargé de développer une application qui permettra de gérer les informations des **secrétaires** et des **commerciaux,** employés dans la société VONEDO. Une classe Employe a été développée pour une autre application.Voici ce qu'elle contient :

```
class Employe
{
   private string matricule;
   private string nom;
   private float salaireMensuel;

   public string GetMatricule()
   {
      return matricule;
   }
   public string GetNom()
   {
      return nom;
   }
   public void SetNom(string nom)
   {
      this.nom = nom;
   }
   private void Augmenter(float augment)
   {
      salaireMensuel = salaireMensuel + augment;
   }
}
```

Pour chaque commercial, on a besoin de gérer son **taux de commission** ainsi que le **montant du chiffre d'affaires** qu'il a réalisé.
Pour chaque secrétaire, on a besoin de gérer son **niveau d'anglais.**

Il est nécessaire de créer 2 nouvelles classes : la classe Commercial et la classe Secretaire. Etant donné qu'un **commercial est un employe** et qu'une **secrétaire est un employe**, on va mettre en œuvre l'héritage de classe.

- la classe Employe contiendra **les attributs et les méthodes** relatifs à n'importe quel employé
- la classe Commercial contiendra **les attributs spécifiques et les méthodes spécifiques** aux commerciaux
- la classe Secrétaire contiendra **les attributs spécifiques et les méthodes spécifiques** aux secrétaires
- les classes Secrétaire et Commercial doivent hériteront de la classe Employe

Avantage de l'héritage :

- **coûts de maintenance réduits** : si on souhaite ajouter (ou modifier) un attribut commun ou une méthode commune à tous les employés, une seule classe devra être modifiée.

II. L'HERITAGE : CLASSE DE BASE ET CLASSE DERIVEE

1) La classe Secretaire

```
class Secretaire : Employe
{
    private int niveauAnglais;

    public int GetNivAnglais()
    {
       return this.niveauAnglais;
    }
    public void SetNivAnglais(int niveau)
    {
       this.niveauAnglais = niveau;
    }
 }
```

: Employe signifie que Secretaire hérite de Employe

2) La classe Commercial

```
class Commercial : Employe
{
    private float txCommission;
    private float mtCa;
    public float GetTxCom()
    {
       return txCommission;
    }
    public float GetMtCA()
    {
       return mtCa;
    }
    public float GetMtCom()
    {
       return this.GetMtCA() * this.GetTxCom() / 100;
    }
    public void SetTxCom(float txCom)
    {
       this.txCommission = txCom;
    }
    public void AugmentVente(float montant)
    {
            mtCa = mtCa + montant;
    }
}
```

: Employe signifie que Commercial hérite de Employe

3) Définition : classe de base et classe dérivée

- La classe Employe est appelée **classe de base** (on parle aussi de **classe mère ou super classe**).
- Commercial et Secrétaire sont des **classes dérivées** de Employe (on parle aussi de **sous-classes** ou encore de **classes filles**).
- On dit que la classe Commercial et la classe Secretaire **héritent (ou dérivent)** de Employe.

4) Instances de la classe de base et des classes dérivées

A partir de ces 3 classes, nous allons pouvoir créer des :

- des instances de la classe **Secretaire** pour gérer des secrétaires
- des instances de la classe **Commercial** pour gérer des commerciaux
- des instances de la classe **Employe** pour gérer des employés qui ne sont ni commercial ni des secrétaire.

> Remarque : on pourra **empêcher la création d'instances** de la classe Employe en ajoutant le mot clé ***abstract*** devant class comme ceci
>
> ```
> abstract class Employe
> {
>     // contenu de la classe
> }
> ```
>
> Mais attention dans ce cas, on considère que la société VONEDO n'embauche que des commerciaux et des secrétaires !!

Réalisons un programme principal suivant qui permet la création d'une instance de la classe Employe, d'une instance de la classe Commercial et d'une instance de la classe Secretaire :

```
static void Main(string[] args)
{
    // déclaration d'une référence à une instance de Employe
    Employe unEmploye;

    // déclaration d'une référence à une instance de Secrétaire
    Secretaire uneSecretaire;

    // déclaration d'une référence à une instance de Commercial
    Commercial unCommercial;

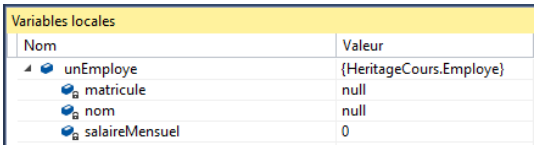
    // création d'une instance de Employe
    unEmploye= new Employe();

    // création d'une instance de Secrétaire
    uneSecretaire = new Secretaire();

    // création d'une instance de Commercial
    unCommercial = new Commercial();
```

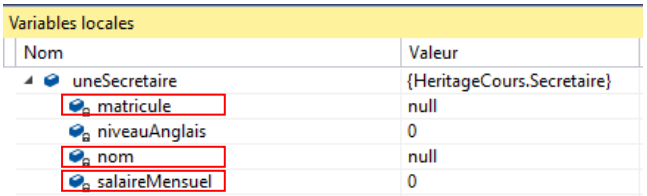
Nous allons utiliser le débogueur pour voir les attributs des 3 instances créées

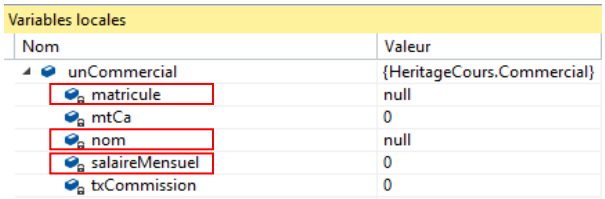
a) Les attributs des instances de la classe de base



Sans surprise,on voit que l'objet dispose d'une valeur pour chaque attribut déclaré dans la classe.

b) Les attributs des instances des classes dérivées





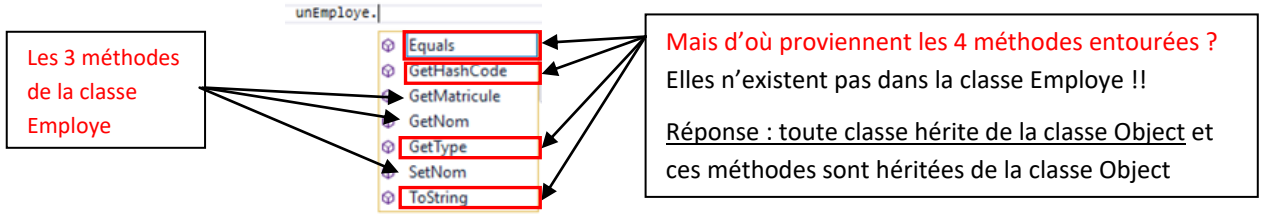
Remarque : les attributs sont affichés par le débogueur par ordre alphabétique.

On voit que l'objet dispose :

- d'une valeur **pour chaque attribut déclaré dans la classe de base** (encadré en rouge)
- d'une valeur **pour chaque attribut déclaré dans la classe dérivée**

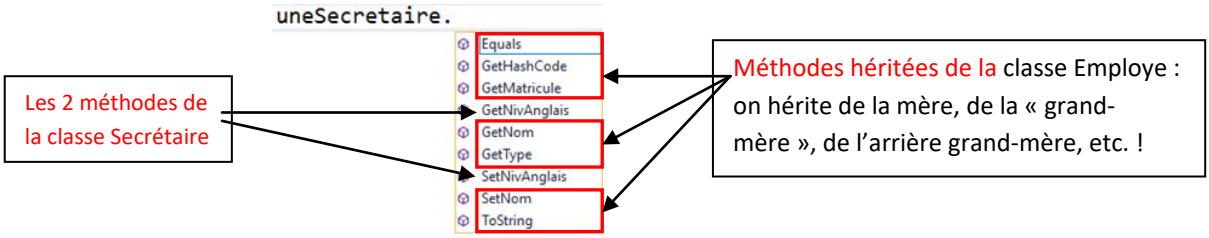
c) Les méthodes appelables par une instance de la classe de base

Voici la liste des méthodes pouvant être appelées par l'instance *unEmploye* :

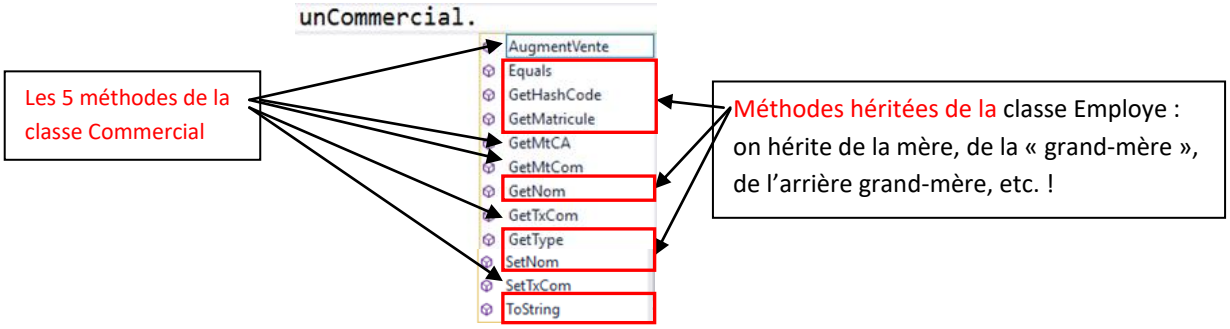


d) Les méthodes appelables par une instance de la classe dérivée

Voici la liste des méthodes pouvant être appelées par l'instance *uneSecrétaire*:



Voici la liste des méthodes pouvant être appelées par l'instance *unCommercial*:



Dans un programme principal, **une instance de la classe dérivée peut appeler :**
- **toutes ses méthodes publiques**
- **ainsi que toutes les méthodes publiques** de sa classe de base.

```
// affichage des caractéristiques de la secrétaire
//-----------------------------------------------------
Console.WriteLine("Caractéristiques de la secrétaire : ");

// affichage du matricule de la secrétaire
Console.WriteLine("matricule de la secrétaire : " + uneSecretaire.GetMatricule());

// affichage du niveau d'anglais de la secrétaire
Console.WriteLine("niveau d'anglais de la secrétaire : " + uneSecretaire.GetNivAnglais());

// affichage des caractéristiques du commercial
//-----------------------------------------------------
Console.WriteLine("Caractéristiques du commercial : ");

// affichage du matricule du commercial
Console.WriteLine("matricule du commercial : " + unCommercial.GetMatricule());

// affichage du taux de commission du commercial
Console.WriteLine("taux de commission du commercial : " + unCommercial.GetTxCom());
```

III. HERITAGE ET CONSTRUCTEURS

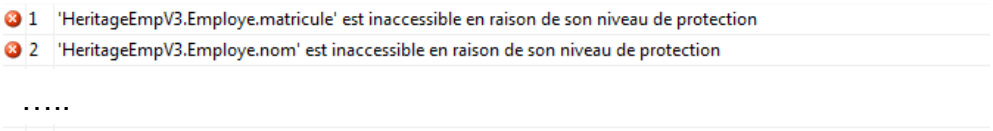
- Créons un constructeur pour la classe de base

```
public Employe(string matricule, string nom, float salaireMensuel)
{
        this.matricule = matricule;
        this.nom = nom;
        this.salaireMensuel = salaireMensuel;
}
```

- Créez un constructeur pour la classe dérivée Secretaire

```
public Secretaire(string matricule, string nom, float salaireMensuel,int nivAnglais)
{
        this.matricule = matricule;
        this.nom = nom;
        this.salaireMensuel = salaireMensuel;
        this.niveauAnglais = nivAnglais;
}
```

Vous obtenez les erreurs suivantes sur le constructeur de la classe Secretaire :



…..

Pour résoudre ce problème vous avez **trois solutions** :

- **1ère solution (à rejeter)**: ~~mettre tous les attributs de la classe de base en public~~ : mauvaise solution car elle ne respecte pas le principe d'encapsulation

- **2ème solution** : utiliser un nouveau mode de protection pour **tous les attributs de la classe de base** : le mode **protected** (protégé) au **lieu de privé** : cela signifie que tous les attributs de la classe de base ne seront accessibles que par les méthodes de la classe de base ET par les méthodes des classes dérivées.

  - Seule la classe Employe (uniquement !) doit être modifiée : tous les attributs doivent être en mode protection **protected**

```
class Employe
{
    protected string matricule;       // matricule
    protected string nom;             // nom
    protected float salaireMensuel;          // salaire mensuel
      etc …
 }
```

  - Voici le constructeur de la classe Employe :

```
public Employe(string matricule, string nom, float salaireMensuel)
{
        this.matricule = matricule;
        this.nom = nom;
        this.salaireMensuel = salaireMensuel;
}
```

- Voici le constructeur de la classe Commercial :

```
public Commercial(string matricule, string nom, float salaireMensuel, float txCom, float CA)
{
        this.matricule = matricule;
        this.nom = nom;
        this.salaireMensuel = salaireMensuel;
        this.txCommission = txCom;
        this.mtCa = CA;
}
```

- Voici le constructeur de la classe Secretaire :

```
public Secretaire(string matricule, string nom, float salaireMensuel, int nivAnglais)
{
        this.matricule = matricule;
        this.nom = nom;
        this.salaireMensuel = salaireMensuel;
        this.niveauAnglais = nivAnglais;
}
```

Les attributs de la classe de base sont maintenant accessibles car ils sont en mode protégé

- **3ème solution** : laissez tous les attributs en **mode de protection privé** (tous ceux de la classe de base et tous ceux des classes dérivées) et **appelez le constructeur de la classe de base** pour valoriser les attributs de la classe de base. Voici comment faire :

  - Voici le constructeur de la classe Commercial :

```
public Commercial(string matricule, string nom, float salaireMensuel,
                  float txCom, float CA) : base(matricule, nom, salaireMensuel)
{
        this.txCommission = txCom;
        this.mtCa = CA;
}
```

:base(…) permet d'appeler le contructeur de la classe de base (la classe Employe) il reçoit les valeurs transmises au constructeur de la classe dérivée

  - Voici le constructeur de la classe Secretaire :

```
public Secretaire(string matricule, string nom, float salaireMensuel,
                  int nivAnglais) : base(matricule, nom, salaireMensuel)
{
        this.niveauAnglais = nivAnglais;
}
```

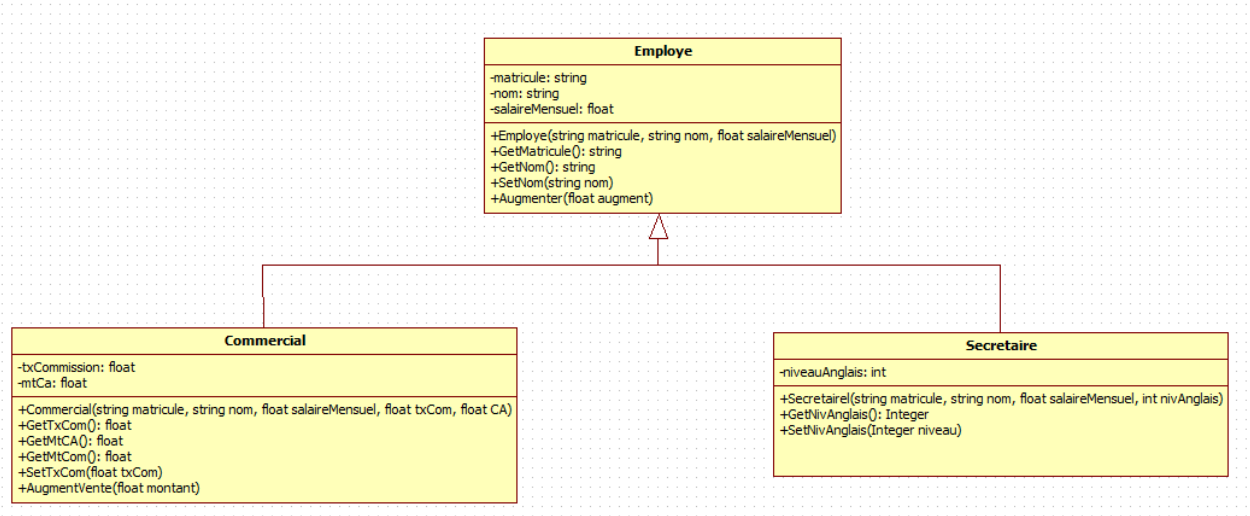
Remarque importante : en Java, on n'utilise pas **:base** mais le mot clé **super** de cette manière :

```
public Commercial(string matricule, string nom, float salaireMensuel,
                                                float txCom, float CA)
{
        super(matricule, nom, salaireMensuel) ;
        this.txCommission = txCom;
        this.mtCa = CA;
}

public Secretaire(string matricule, string nom, float salaireMensuel,  int nivAnglais)
{
        super(matricule, nom, salaireMensuel) ;
        this.niveauAnglais = nivAnglais;
}
```

super(…) permet d'appeler le contructeur de la classe de base (la classe Employe) il reçoit les valeurs transmises au constructeur de la classe dérivée

IV. HERITAGE ET UML



V. REDEFINITION DE METHODE

1) Besoin

On souhaite :

- ajouter une méthode **public void AfficherInfos() dans la classe Employe** qui affiche le matricule, le nom, le prénom et le salaire mensuel de l'employé
- une méthode **public void AfficherInfos()** dans la classe **Commercial** qui affiche le taux de commission et le montant du chiffre d'affaires
- une méthode **public void AfficherInfos()** dans la classe **Secretaire** qui affiche le niveau d'anglais

2) La redéfinition de méthode

- On parle de **redéfinition de méthode** lorsque une classe dérivée définit une méthode **identique** à la classe dont elle hérite.
  ATTENTION : "Identique" signifie même nom, même mode de retour et même(s) paramètre(s). Le corps de la méthode (contenu) est bien sûr différent.

- La classe Secrétaire redéfinit la méthode AfficherInfos.

- La classe Commercial redéfinit la méthode AfficherInfos.

- On dit également que **la méthode AfficherInfos est polymorphe** car **selon l'objet qui l'appelle, la méthode fera des choses différentes** :
  - Si la méthode AfficherInfos est appelée par un **objet de la classe Employe**, alors c'est la méthode située dans la classe **Employe** qui sera appelée.
  - Si la méthode AfficherInfos est appelée par un **objet de la classe Secretaire**, c'est la méthode située dans la classe **Secretaire** qui sera appelée.
  - Si la méthode AfficherInfos est appelée par un objet de la classe **Commercial**, alors c'est la méthode située dans la classe **Commercial** qui sera appelée.

Exemple :

```
Employe unEmploye;
Secrétaire uneSecretaire;
Commercial unCommercial ;

unEmploye = new Employe (……) ;
unCommercial  = new Commercial(……);
uneSecretaire =  new Secrétaire(……) ;

unEmploye.AfficherInfos() ;       // affiche le matricule, le nom et le salaire
unCommercial.AfficherInfos();  // affiche le taux de commission et le CA
uneSecretaire.AfficherInfos() ;  // affiche le niveau d'anglais
```

2) Signature d'une méthode redéfinie (virtual et override)

- Voici la méthode **AfficherInfos() de la classe Employe (classe de base)**

```
public virtual void AfficherInfos()
{
   Console.WriteLine(matricule);
   Console.WriteLine(nom);
   Console.WriteLine(salaireMensuel);
}
```

- Créons une méthode AfficherInfos dans la classe Commercial qui affiche, pour l'instant le taux de commission et le montant du chiffre d'affaires:

```
public override  void AfficherInfos()
{
   Console.WriteLine(txCommission);
   Console.WriteLine(mtCa);
}
```

- Créons une méthode AfficherInfos dans la classe Secrétaire qui affiche le niveau d'anglais :

```
public override void  AfficherInfos()
{
   Console.WriteLine(niveauAnglais);
}
```

5) Appel d'une méthode de la classe de base dans une méthode redéfinie

On souhaite que la méthode AfficherInfos de la classe Secrétaire et la méthode AfficherInfos de la classe Commercial affiche les caractéristiques de la classe de base (matricule, nom et salaire de l'employé). Comment peut-on faire ?

1ère solution : ajouter 3 instructions dans AfficherInfos de la classe Commercial et Secretaire, la première instruction permettant d'afficher le matricule, la deuxième le nom, la troisième le salaire

Voici la méthode AfficherInfos de la classe Commercial :

```
public void AfficherInfos()
{
   Console.WriteLine(matricule);
   Console.WriteLine(nom);
   Console.WriteLine(salaireMensuel);
   Console.WriteLine(txCommission);
   Console.WriteLine(mtCa);
}
```

Voici la méthode AfficherInfos de la classe Secretaire :

```
public void AfficherInfos()
{
   Console.WriteLine(matricule);
   Console.WriteLine(nom);
   Console.WriteLine(salaireMensuel ;
   Console.WriteLine(niveauAnglais);
}
```

Problèmes lié à cette solution :

- Elle ne fonctionne que si on passe les attributs de Employe en mode protected !
- Les instructions permettant l'affichage du matricule, du nom et du salaire est codé à trois endroits… la maintenance sera plus coûteuse

2ème solution : appeler la méthode AfficherInfos() de la classe de base. Mais comment faire ?

- Voici la méthode AfficherInfos dans la classe Commercial

```
public override void AfficherInfos()
{
   base.AfficherInfos() ;   // on appelle AfficherInfos de la classe de base
   Console.WriteLine(txCommission);
   Console.WriteLine(mtCa);
}
```

- Voici la méthode AfficherInfos dans la classe Secretaire

```
public override void AfficherInfos()
{
   base.AfficherInfos() ;   //AfficherInfos de la classe de base est appelée
   Console.WriteLine(niveauAnglais);
}
```

ATTENTION :

- En java, au lieu de **base.**AfficherInfos() on écrira **super.**AfficherInfos()
- Ne confondez pas **surcharge de méthode** et **redéfinition de méthode**