HERITAGE

IUT ANNECY UNIVERSITÉ SAVOIE MONT BLANC

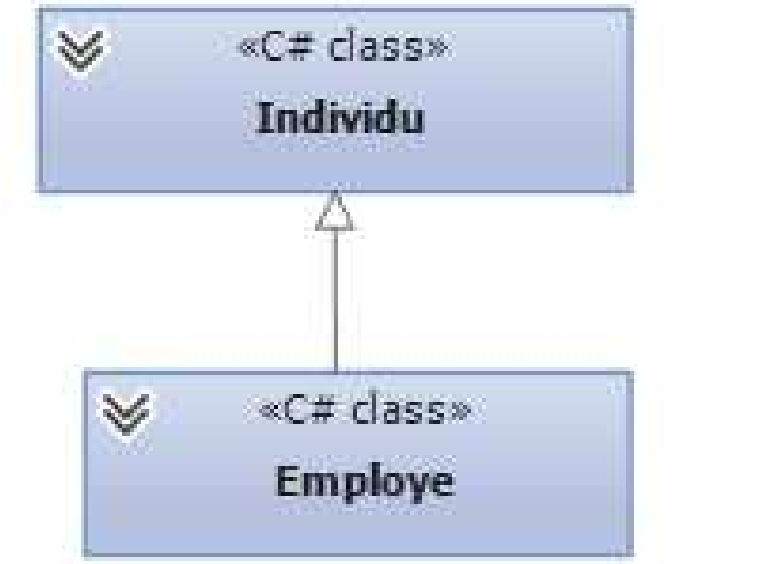
N. Gruson

INFO | INFORMATIQUE

Vocabulaire

Un Employe est avant tout un Individu.

- Classe mère.
  - Ex : Individu
- Classe fille = classe dérivée.
  - Ex : Employe

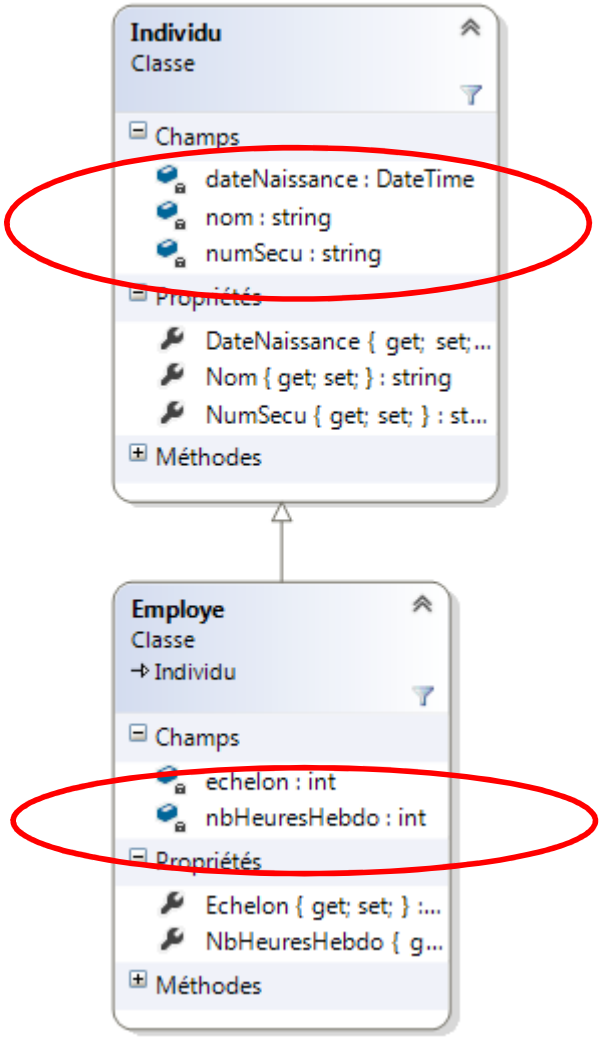



```
public class Employe : Individu
```

La fille : mieux que la mère !

Une classe fille hérite de toutes les caractéristiques de sa mère, et elle a des caractéristiques supplémentaires. Ici :

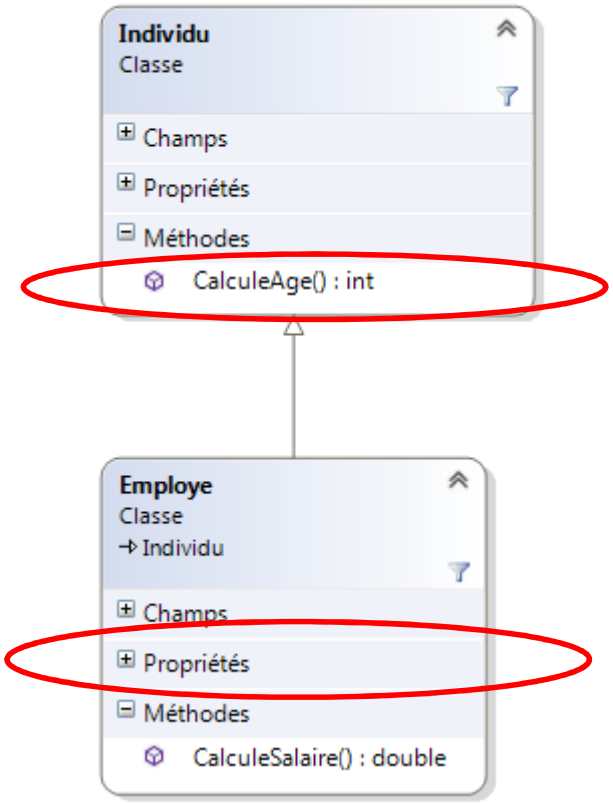
- Un Employe aura comme un Individu :
  - une date de naissance
  - un nom
  - un numéro de sécurité sociale
- Et il aura en plus :
  - Un échelon
  - Un nombre d'heures de travail hebdomadaire



La fille : mieux que la mère !

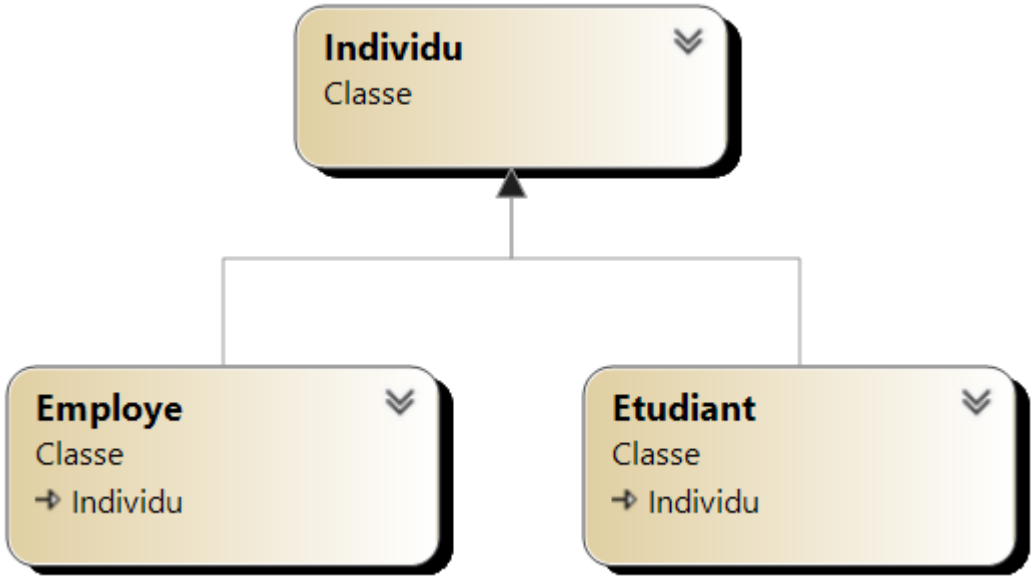
Une classe fille hérite de toutes les méthodes de sa mère, et elle a des méthodes en plus. Ici :

- Un Employe tout comme un Individu sait :
  - Calculer son âge
- Et il sait en plus :
  - Calculer son salaire



Une Mère, plusieurs filles

Ici, Employe et Etudiant sont filles de Individu.

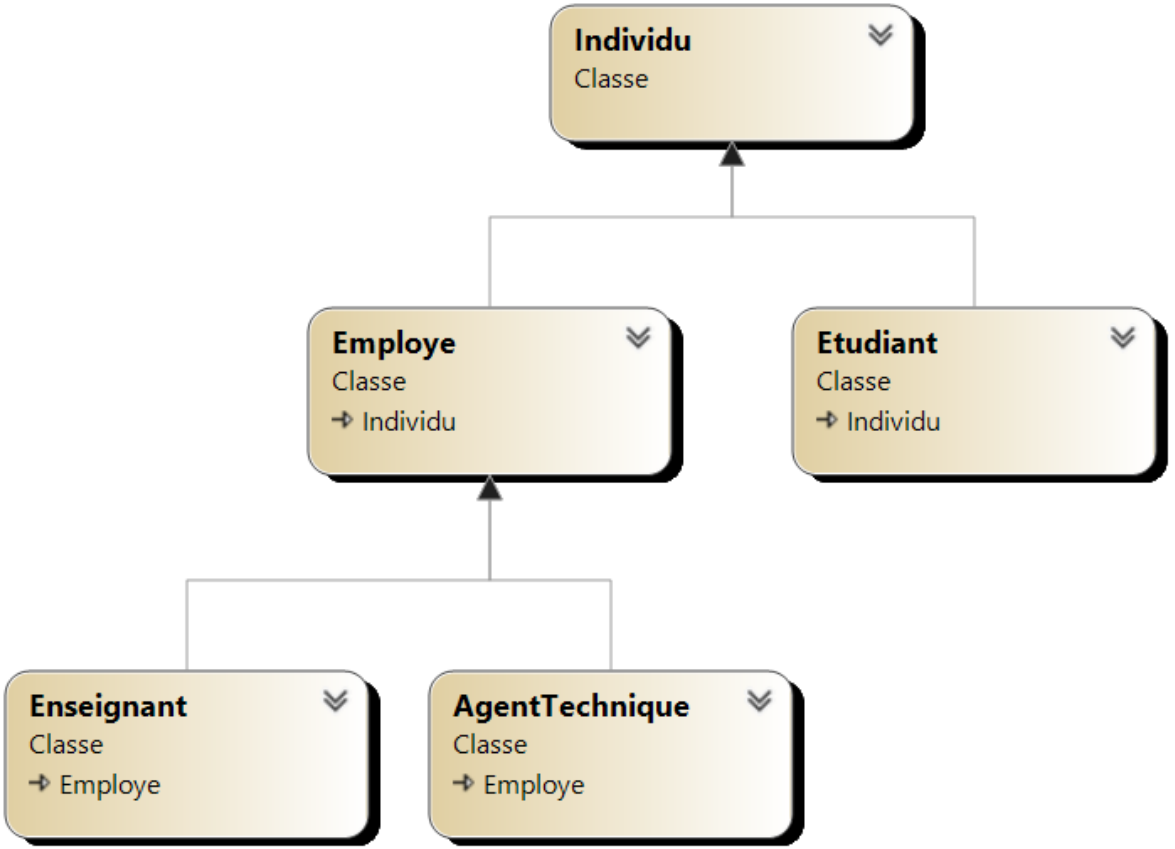



```
public class Employe : Individu
```

```
public class Etudiant : Individu
```

Une Mère, plusieurs filles et « petites filles »

Ici, Enseignant est aussi fille de Individu, mais avant tout fille de Employe.

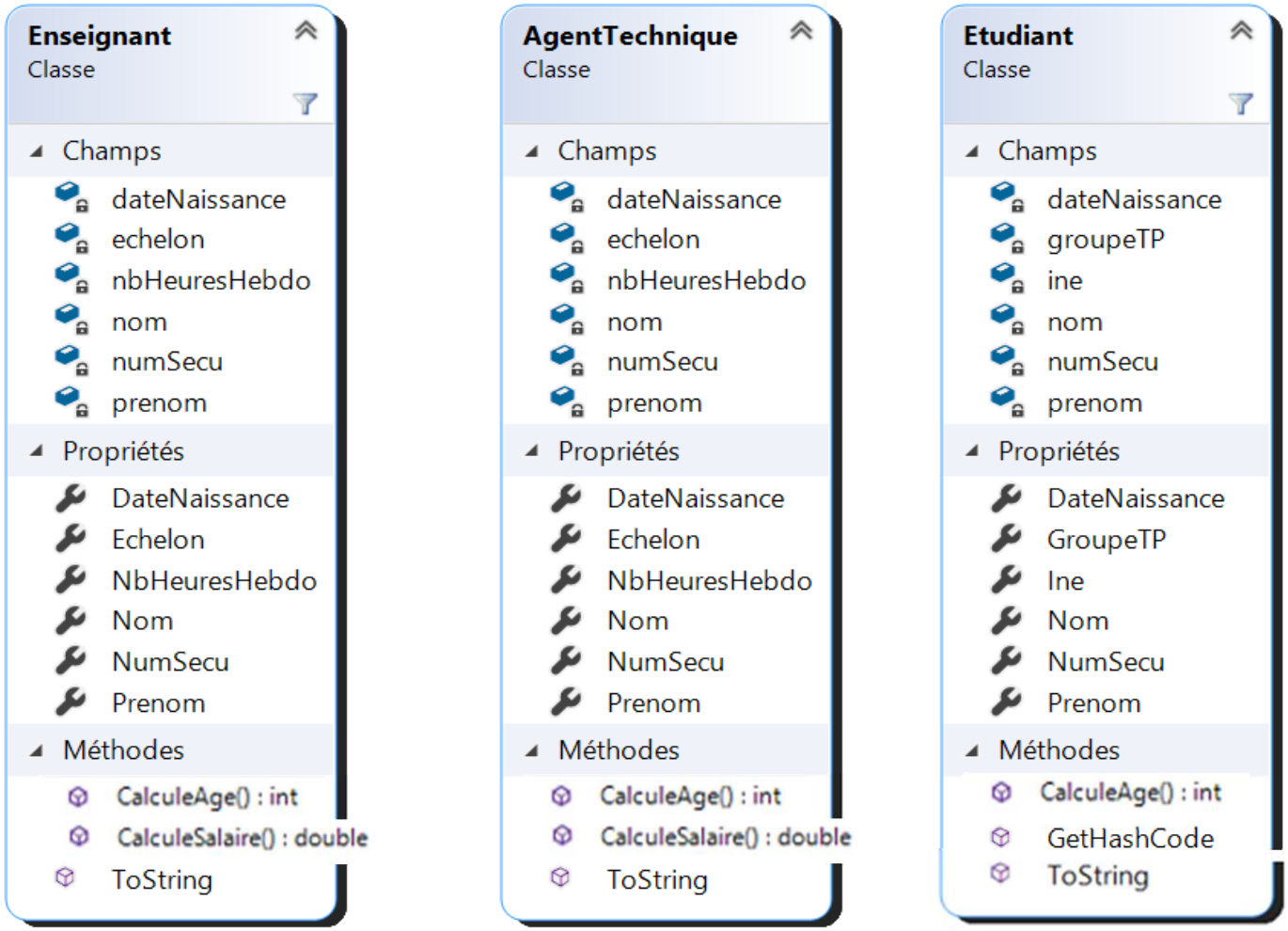



```
public class Enseignant : Employe
```

Héritage : factorisation de code

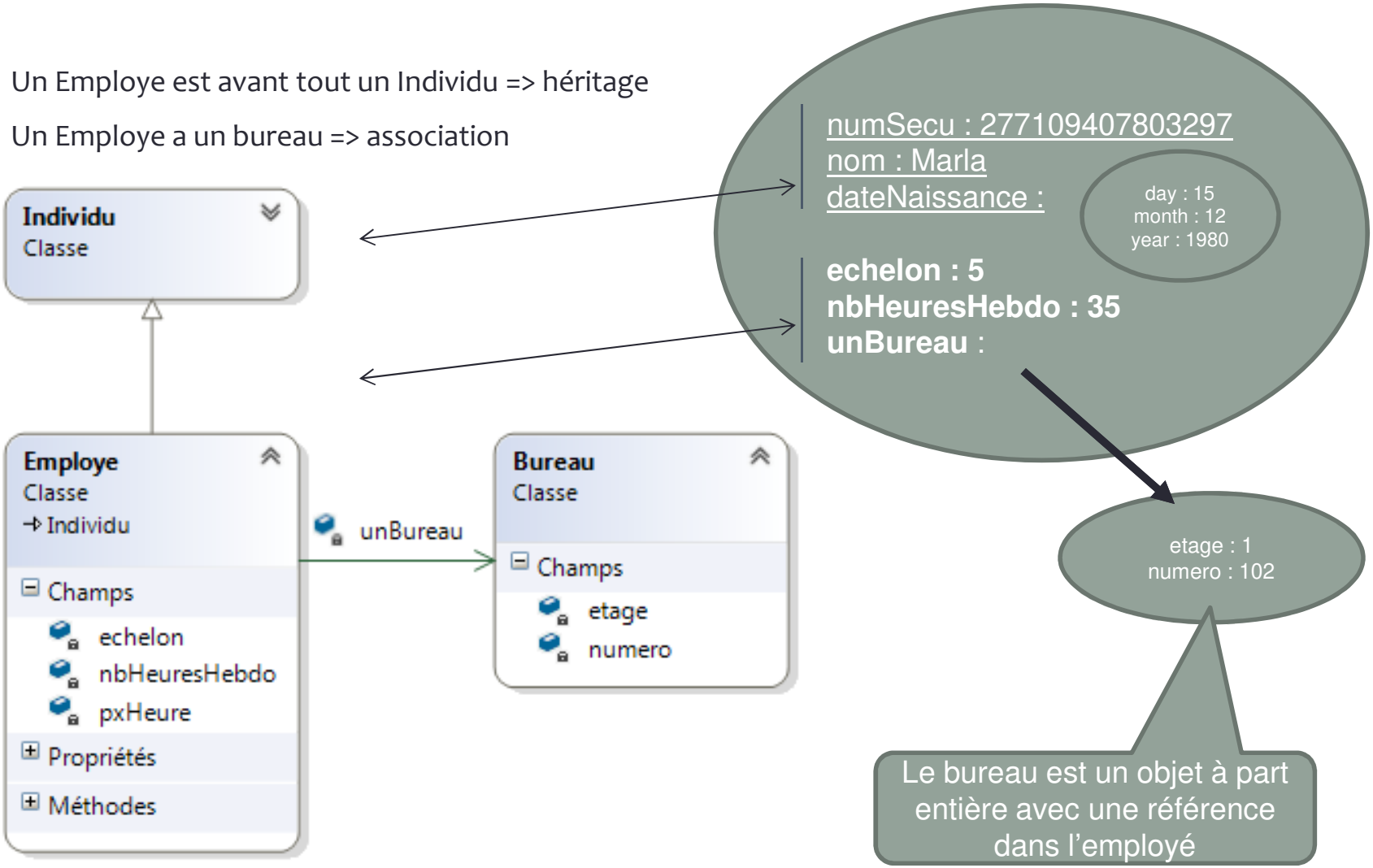
Sans héritage, il y 3 classes mais il faut dupliquer les champs et les méthodes !

Avec héritage, il y a 5 classes mais aucune redondance de code !



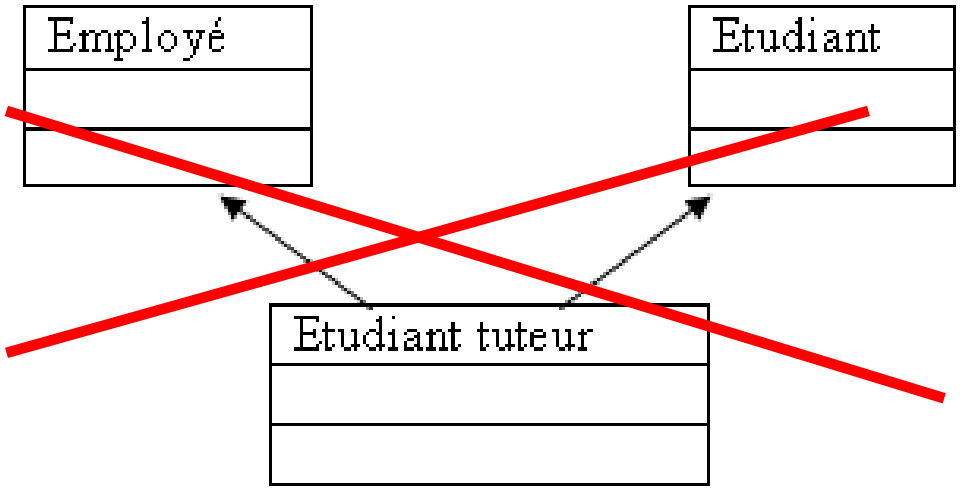
Ne pas mélanger association et héritage

- Un Employe est avant tout un Individu => héritage
- Un Employe a un bureau => association



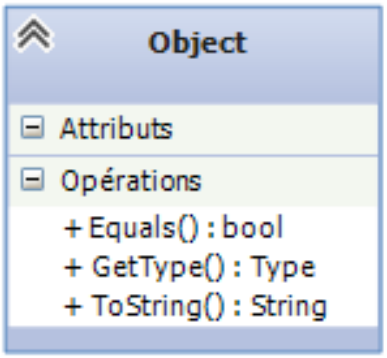
Une seule mère directe

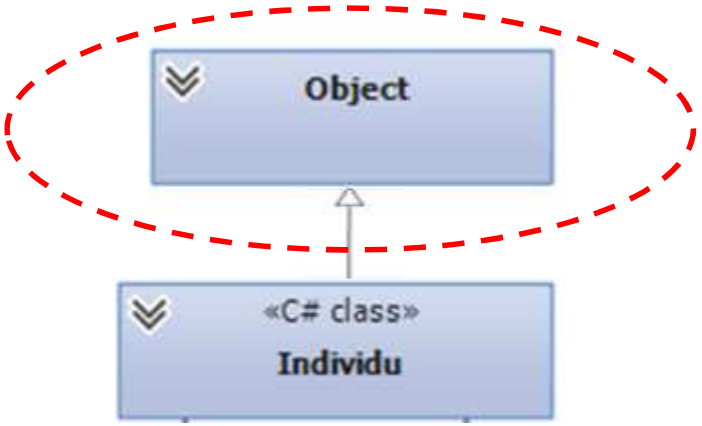
L'héritage multiple n'existe pas en C#



La classe mère suprême

Toute classe hérite (implicitement) de la classe Object



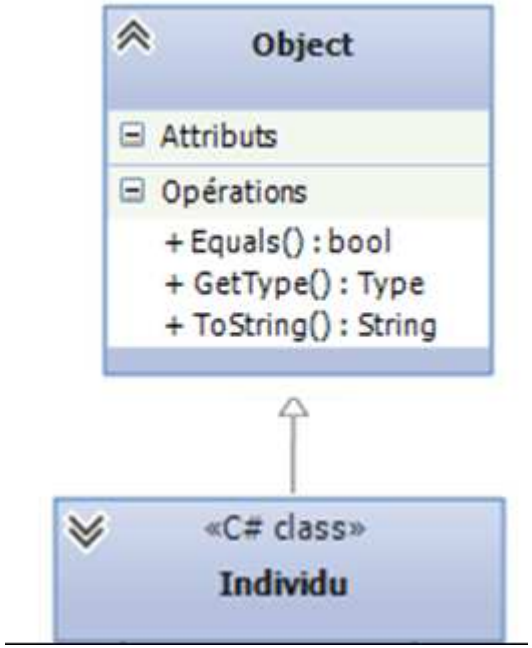


public class **Individu** ⟺ public class **Individu : Object**

La classe mère suprême

Toute classe hérite donc du savoir faire de la classe Object :

- ToString()
- Equals()
- …




```
Individu i = new Individu("277109407803297",
"Marla", new DateTime(1980, 12, 15));
Console.WriteLine(i.ToString());
```

```
ConsoleApplication1.Individu
```

Namespace.nomClasse

SI ToString() n'est pas définie ici => appel ToString la classe Object

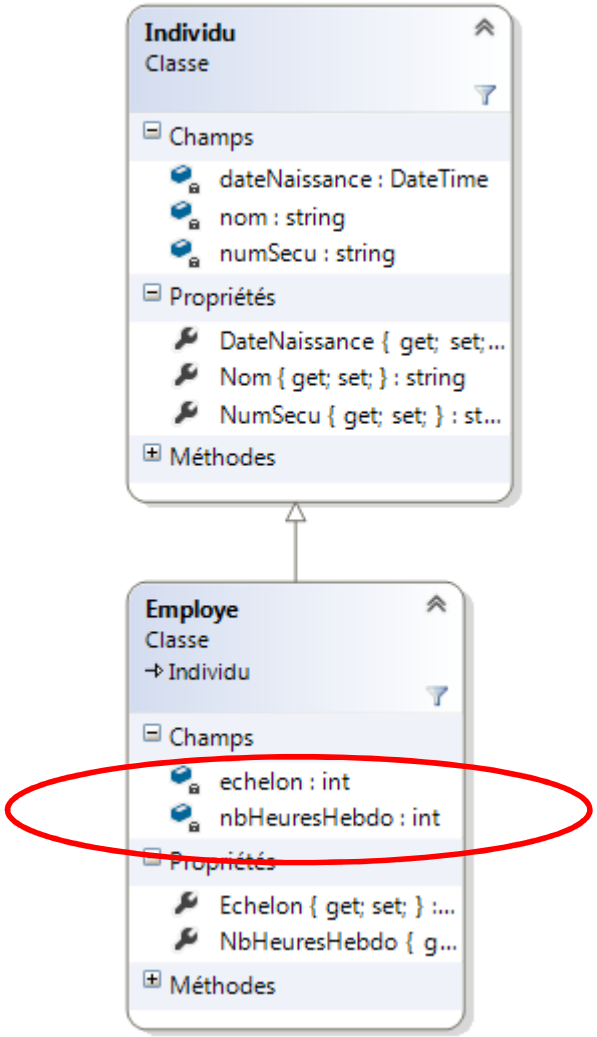
Définir une classe fille

Dans la classe fille : on définit uniquement les nouveaux champs. On ne recopie pas les champs de la classe mère !

```
public class Individu
   {
      private String numSecu;
      private String nom;
      private DateTime dateNaissance;

       public String NumSecu
      {
         get { return this.nu
         set {this. numSecu
      }
     public String Nom
      {
         get { return this.no
         set { this.nom = va
      }
public DateTime DateNais
      {
         get { return this.da
         set { this.dateNaiss
      }
```

```
public class Employe : Individu
   {
      private int echelon;
      private int nbHeuresHebdo;
     public int Echelon
      {
         get { return this.echelon; }
         set { this.echelon = value; }
      }
      public int NbHeuresHebdo
      {
         get { return this.nbHeuresHebdo; }
         set { this.nbHeuresHebdo = value;
}
      }
```



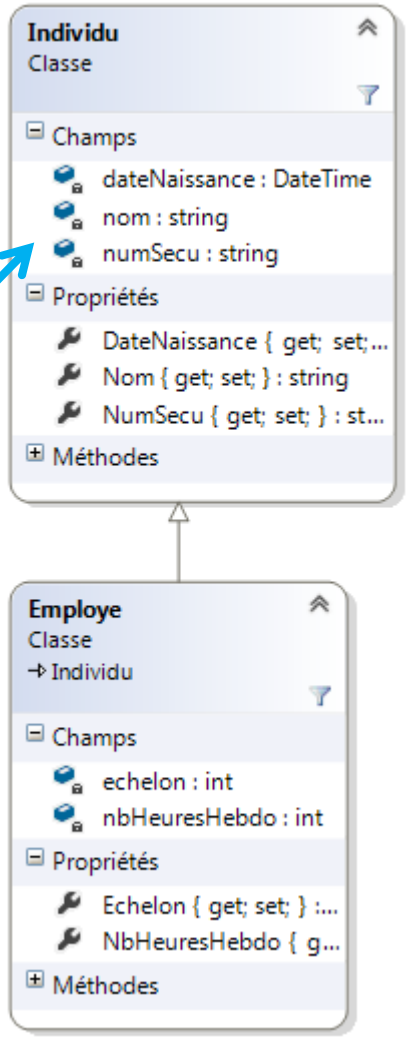
Constructeur d'une classe fille

- Il attend les paramètres nécessaires à l'initialisation des **champs spécifiques** mais aussi des **champs hérités.**

```
public Employe(string numSecu, String nom,
        DateTime dateNaissance, int echelon, int nbHeures)
```

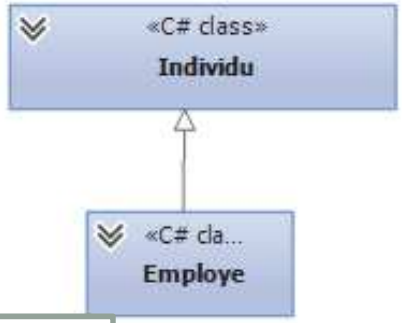
```
Employe e = new Employe("277109407803297", "Marla",
                        new DateTime(1980, 12, 15),
                        5,35);
```

numSecu : 277109407803297
nom : Marla
dateNaissance : day : 15, month : 12, year : 1980
echelon : 5
nbHeuresHebdo : 35



Constructeur d'une classe fille

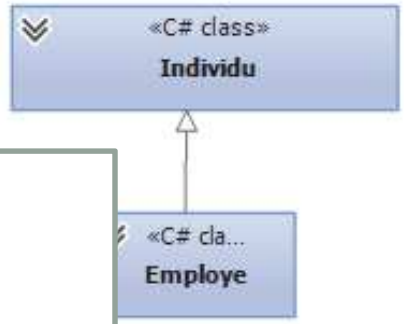
- Il s'appuie sur le constructeur de la classe mère pour initialiser les champs hérités : on réutilise !
- Il initialise les champs spécifiques




```
public Individu( string numSecu, String nom, DateTime dateNaissance)
        {  this.NumSecu = numSecu;
           this.Nom = nom;
           this.DateNaissance = dateNaissance;
        }
```

```
public Employe(string numSecu, String nom, DateTime dateNaissance,  int echelon, int nbHeures)
   : base(numSecu, nom, dateNaissance)
   { this.Echelon = echelon;
     this.NbHeuresHebdo = nbHeures;
   }
```

Constructeur d'une classe fille



- Attention : this / base !

```
public Individu( string numSecu, String nom, DateTime dateNaissance)
        {  this.NumSecu = numSecu;
           this.Nom = nom;
           this.DateNaissance = dateNaissance;
        }
```

base : cherche dans la classe mère

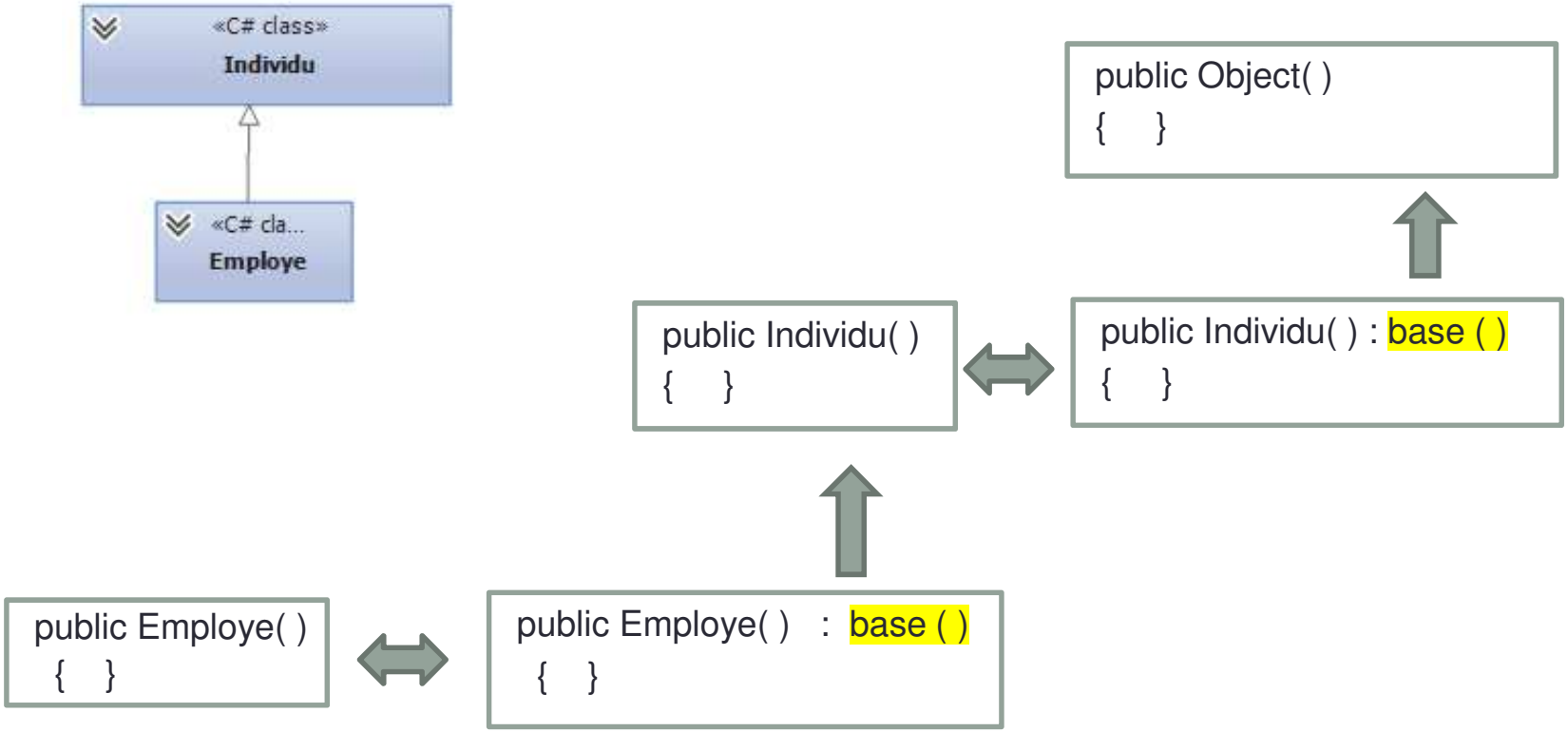
```
public Employe(string numSecu, String nom, DateTime dateNaissance,
int echelon, int nbHeures)  : base(numSecu, nom, dateNaissance)
      { this.Echelon = echelon;
        this.NbHeuresHebdo = nbHeures;
      }
```

this : cherche dans la même classe

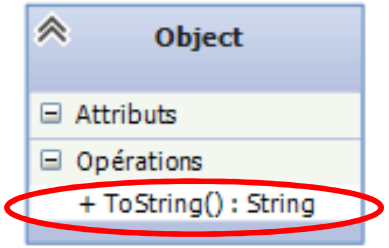
```
public Employe(string numSecu, String nom, DateTime dateNaissance, int echelon)  :
this(numSecu, nom, dateNaissance, echelon, 35)
      {    }
```

Constructeur par défaut d'une classe fille

- L'appel au constructeur de la classe mère est implicite

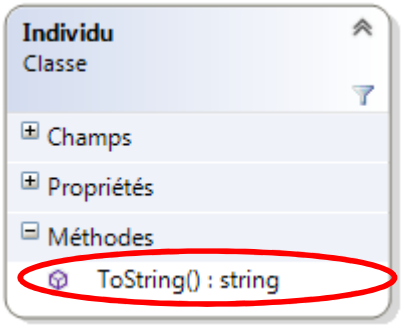


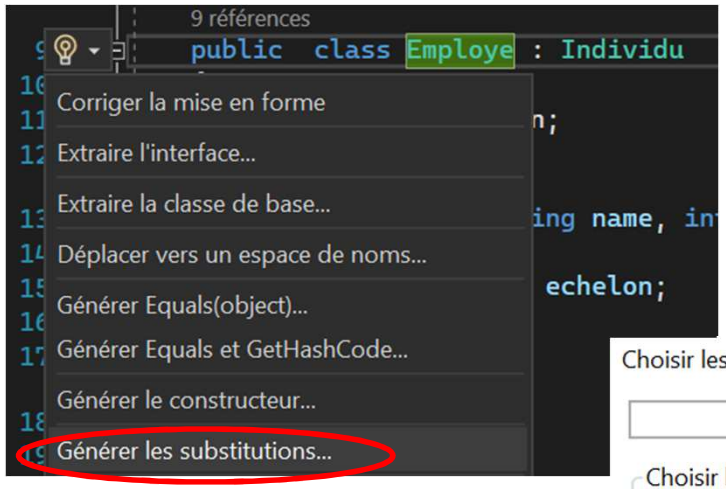
Substituer des méthodes héritées



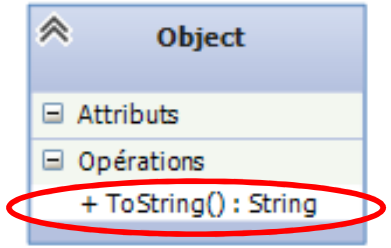
- Concept de **substitution (ou redéfinition)** : au sein d'une classe fille, on peut redéfinir le comportement des méthodes héritées : OVERRIDE

- Substitutions habituelles : ToString, Equals, GetHashCode



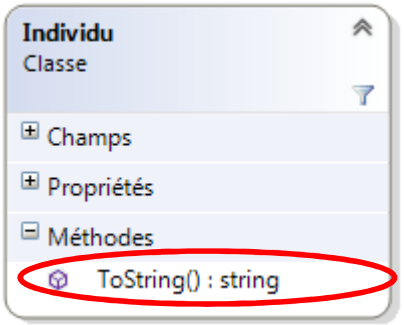


Substituer des méthodes héritées



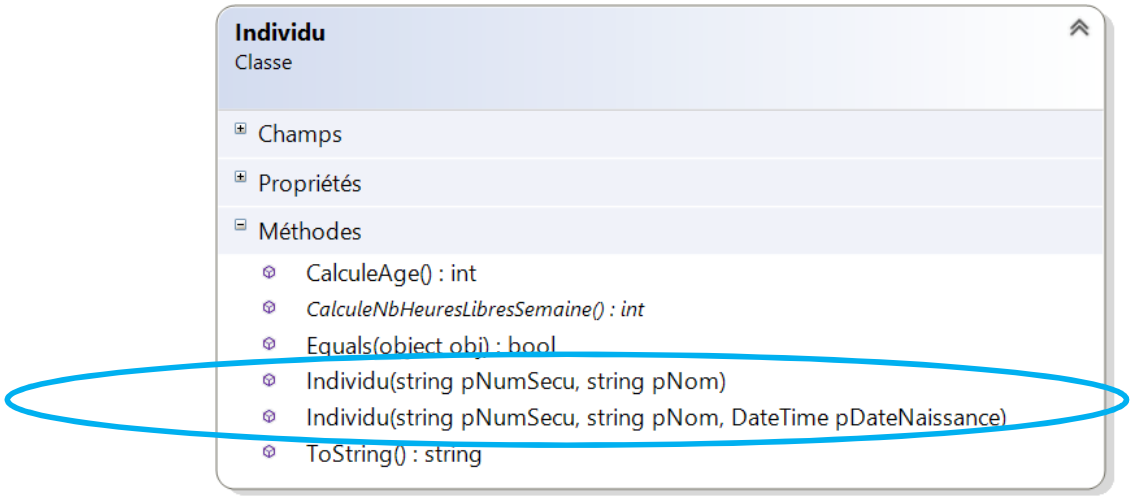
ATTENTION : ne pas confondre

- Concept de **substitution** (redéfinition) :
  - au sein des classes filles
  - même signature (forme)



AVEC

- Concept de **surcharge** :
  - Au sein d'une même classe ou classe fille
  - Différentes signatures



Substituer des méthodes héritées

On peut alors faire :

- Une substitution **partielle** : on reprend ce que sait faire la classe mère et on ajoute un nouveau comportement

- Une substitution **totale** : on définit un comportement autre que celui défini dans la classe mère

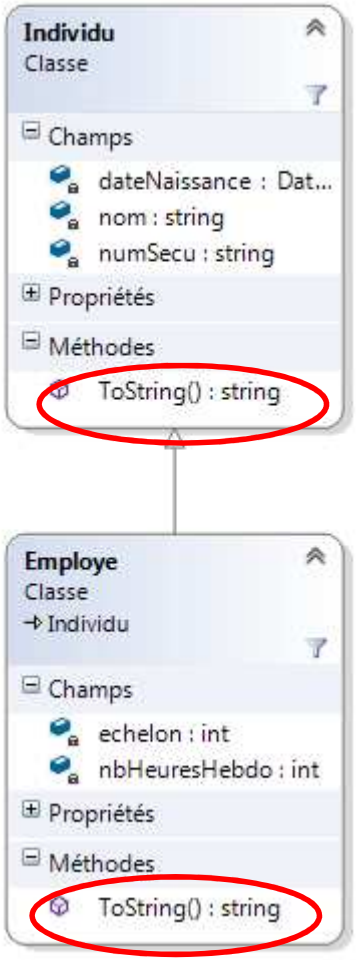
Substitution « partielle »

- Appel à la méthode héritée : base

```
public class Individu
{
  public override string ToString()
        {
            return   "\nNumero de sécurité sociale : " + this.NumSecu
              + "\nNom : " + this.Nom
              + "\nDate de naissance : " + this.DateNaissance.ToString("d") ;
        }
```

```
public class Employe : Individu
{
 public override string ToString()
     {
        return   base.ToString()      + "\nNb d'heures : " + this.NbHeuresHebdo
          + "\nEchelon : " + this.Echelon;
     }
```

```
Numero de sécurité sociale : 1351094078032 97
Nom : Billat
Date de naissance : 05/10/1970
Nb d'heures : 35
Echelon : 2
```

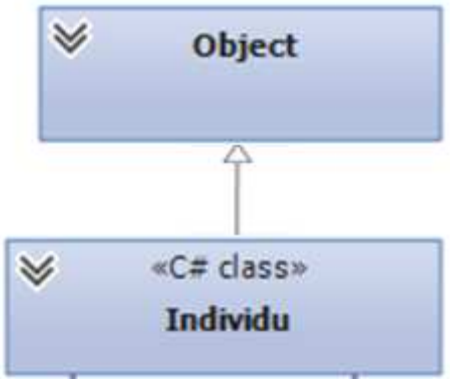


Substitution totale

On redéfinit le comportement de la méthode sans prendre en compte ce que fait la classe mère.

Ici, la méthode ToString de la classe mère Object renvoie uniquement le namespace et la classe : ce qui n'est pas très utile.

```
public class Individu
{
public    override    string   ToString()
  {
    return "\nNumero de sécurité sociale : " + this.NumSecu
      + "\nNom : " + this.Nom
      + "\nDate de naissance : " + this.DateNaissance.ToString("d") ;
   }
```




```
Numero de sécurité sociale : 2771094078032 97
Nom : Marla
Date de naissance : 15/12/1980
```

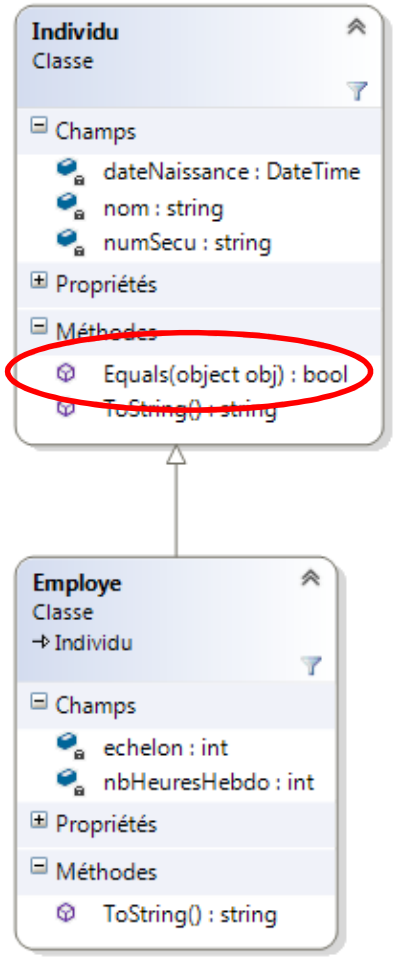
Substitution : choix ou obligation ?

- **Pas une obligation !** La fille a alors le même comportement que la mère.
- Dépend du contexte.

```
public class Individu
{

 public override bool Equals(object obj)
        {
          return obj is Individu i &&
              this.NumSecu.Equals(i.NumSecu) ;
        }
}
```

```
Employe e1 = new Employe("277109407803297",
"Marla", new DateTime(1980, 12, 15),5,35);
Employe e2 = new Employe("177109407803297",
"Marla", new DateTime(1982, 12, 15),5,35);
if ( e1.Equals ( e2 ))
```



Si seul le numéro de sécurité sociale sert à identifier un individu comme un employe, pas de substitution de Equals dans Employe !

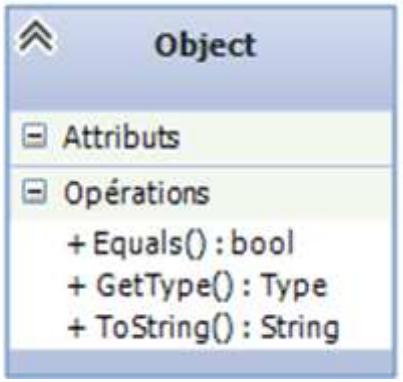
Substitution : la mère décide !

- **Une mère contrôle le droit à la substitution ! Seules les méthodes qualifiées par le mot clef virtual sont substituables.**

```
public virtual bool Equals( Object obj )

public virtual string ToString()

public virtual int GetHashCode()
```



- **La fille ne peut substituer le comportement des méthodes héritées que si sa mère lui permet !**

Substitution : la mère décide !

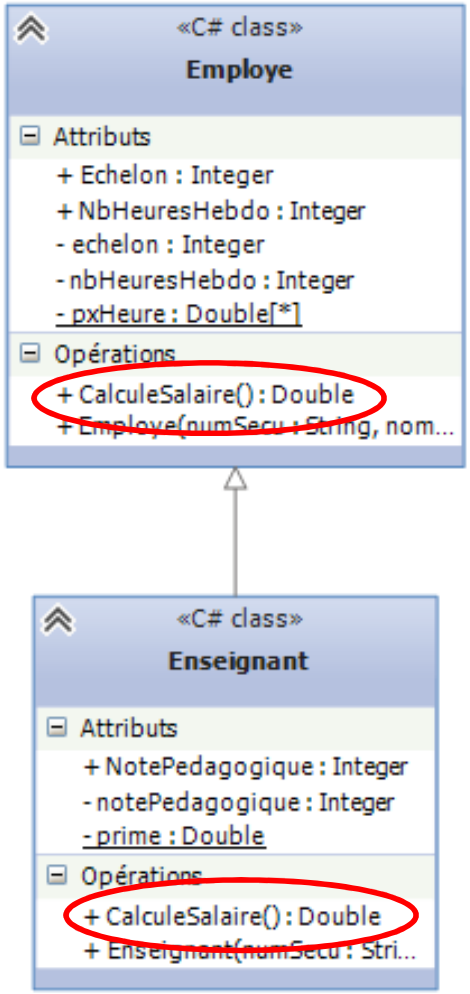
```
public  class Employe : Individu
{
   public  virtual  double CalculeSalaire()

        { return Employe.pxHeure[ this.Echelon]
                 * this.NbHeuresHebdo*4; }
}
```

```
public  class Enseignant : Employe
{
    public  override  double CalculeSalaire()

        { return base.CalculeSalaire()

                + prime * this.NotePedagogique; }
}
```



Polymorphisme

Au moment de l'exécution, les objets peuvent avoir une autre forme que celle déclarée : elles peuvent être issues d'une classe fille à la classe de déclaration.

- Soit dans une collection :

```
List<Individu> lesIndividus = new List<Individu>();
lesIndividus.Add (new Individu("277109497", "Marla", new DateTime(1980, 12, 15)));
lesIndividus.Add (new Employe("13510947", "Billat", new DateTime(1970, 10, 5), 2,35));
lesIndividus.Add( new Enseignant("25610", "Bois", new DateTime(1972, 2, 24), 2, 25,42));
```

Ici, dans la liste d'individus, se cachent des Employés des enseignants …

- Soit dans les paramètres :

```
public override bool Equals(object obj)
```

Ici, dans obj, il y a en réalité : un Individu, un employe, ….

Avantage du polymorphisme

Faire du code générique, mais qui déclenchera le traitement adéquat.

Ex : avec le même code, on obtient un résultat différent !

```
List<Individu> lesIndividus = new List<Individu>();
lesIndividus.Add (new Individu("277109407803297", "Marla", new DateTime
lesIndividus.Add (new Employe("135109407803297", "Billat", new DateTime
lesIndividus.Add( new Enseignant("25610940780999", "Bois", new DateTime

 foreach (Individu unIndividu in lesIndividus)
        Console.WriteLine("----------------\n"    + unIndividu);
```

Polymorphisme:
Un individu peut prendre
plusieurs formes. La métho
ToString sera déclenchée e
fonction de la nature de l'obj

ToString d'Individu

ToString d'Employe

ToString d'Enseignant

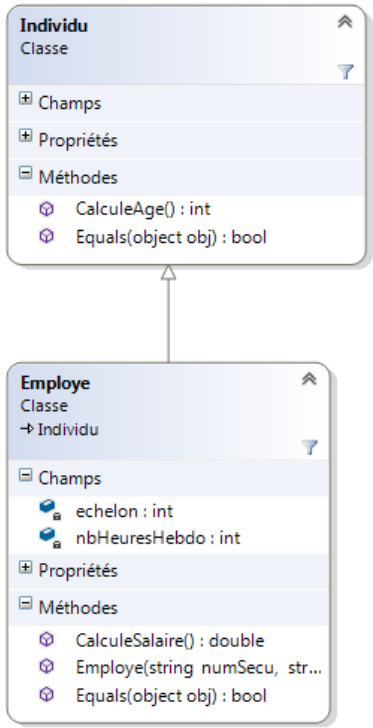
```
----------
ConsoleAppl
Numero de securit      iale : 2771094078
Nom : Marla
Date de naissance : 15/12/1980
----------------
ConsoleApplication1.Employe
Numero de sécurité sociale : 1351094078
Nom : Billat
Date de naissance : 05/10/1970
Nb d'heures : 35
Echelon : 2
----------------
ConsoleApplication1.Enseignant
Numero de sécurité sociale : 2561094078
Nom : Bois
Date de naissance : 24/02/1972
Nb d'heures : 25
Echelon : 2
Note peda : 42
```

Préciser le type : faire un Cast

Trop de généricité peut poser pb : pour déclencher des traitements supplémentaires, il faut parfois « retyper » les objets.

```
foreach (Individu unIndividu in lesIndividus)
        {
           // Affichage de l'age
            Console.WriteLine( unIndividu + "\n.Age  : " + unIndividu.CalculeAge ());

            // Affichage du salaire (seulement pour les employés)
            if ( unIndividu  is  Employe )
             {
              Employe e = (Employe) unIndividu ;
              Console.WriteLine("Salaire :\n" + e.CalculeSalaire());
              }
         }
```



Ou

```
Console.WriteLine("Salaire :\n"  + ((Employe)unIndividu).CalculeSalaire());
```

Faire un Cast – Revenons sur Equals

Equals : pour être générique et commun à toutes les classes : le paramètre est de type object

```
class Individu
{
public override bool Equals(object obj)
        {
          return obj is Individu i &&
              this.NumSecu.Equals(i.NumSecu) ;
        }
}
```

```
Individu i1 = new Individu(….);
Individu i2 = new Individu(….);
if (i1.Equals(i2))
 ….
```

```
class Individu
{
public override bool Equals(object obj)
        {
          if (! ( obj is Individu ))
              return false ;
          Individu i = (Individu) obj ;
          return  this.NumSecu.Equals(i.NumSecu) ;
        }
}
```

Accès aux champs hérités

- Accès protected = private sauf pour les classes filles

```
public  class Individu
{
   protected  String numSecu ;
   public String NumSecu
      {
         get { return this.numSecu; }
         set { this.numSecu = value; }
      }
```

```
public class Employe : Individu
{
   public  double Calcule.... ( )
    {   if ( this.numSecu == ... )
```

Accès direct possible. Mais NumSecu serait mieux ! Autant passer par les propriétés publiques et ne pas transgresser le principe d'encapsulation

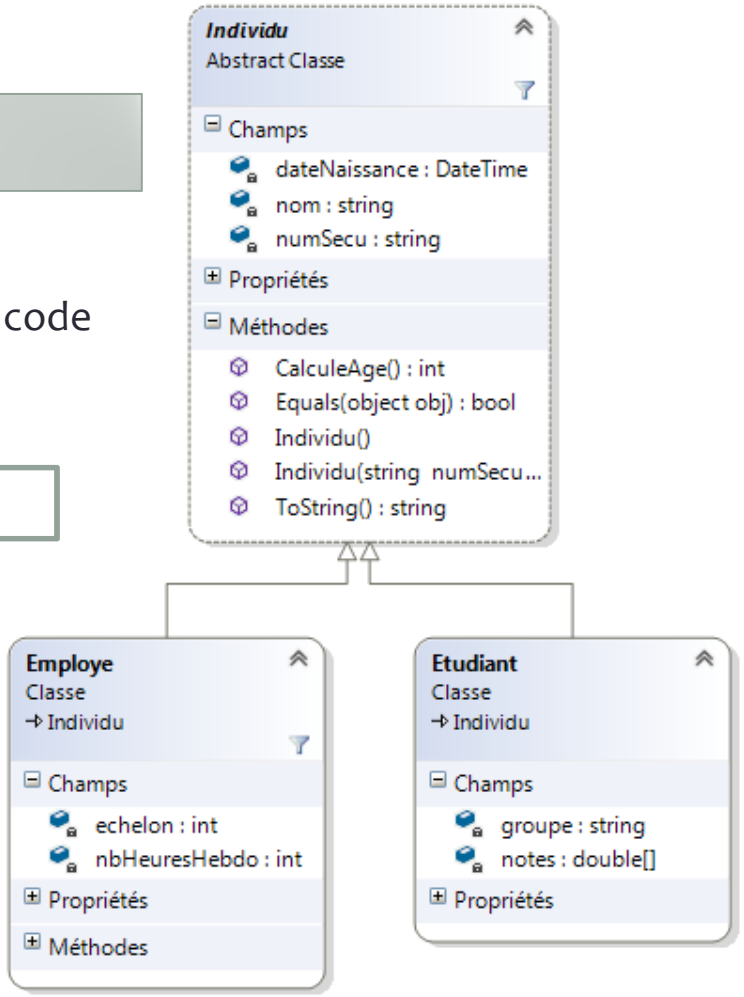
Notion de classe abstraite

- Classe abstraite = classe non instanciable

```
Individu i = new Individu("277109407803297","Marla",
        new DateTime(1980, 12, 15));
```

- Pour définir un concept et factoriser du code commun à des classes instanciables

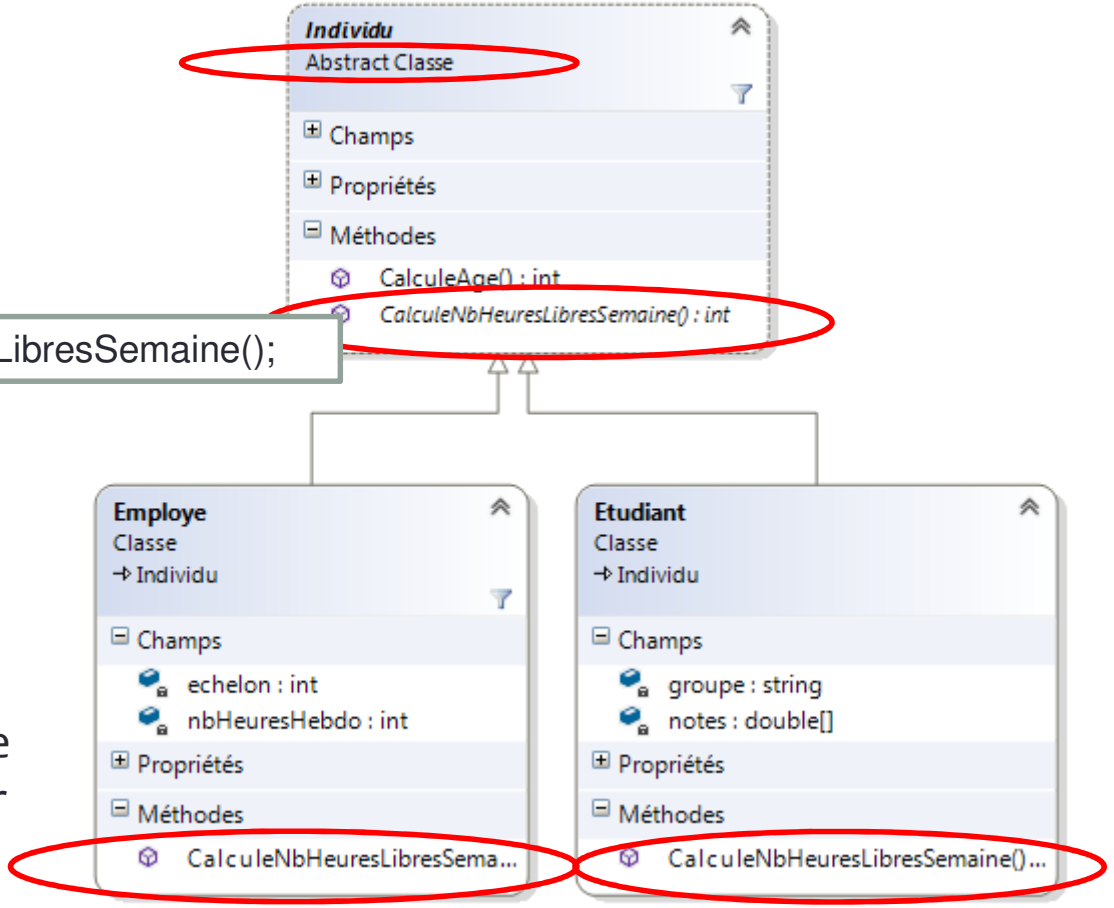
```
public abstract class Individu {
```



Méthode abstraite : la mère ordonne !

- Une classe mère peut exiger de ses filles de faire un traitement sans elle-même savoir comment…

```
public abstract int CalculeNbHeuresLibresSemaine();
```



- Toute classe ayant une méthode abstraite est abstraite.

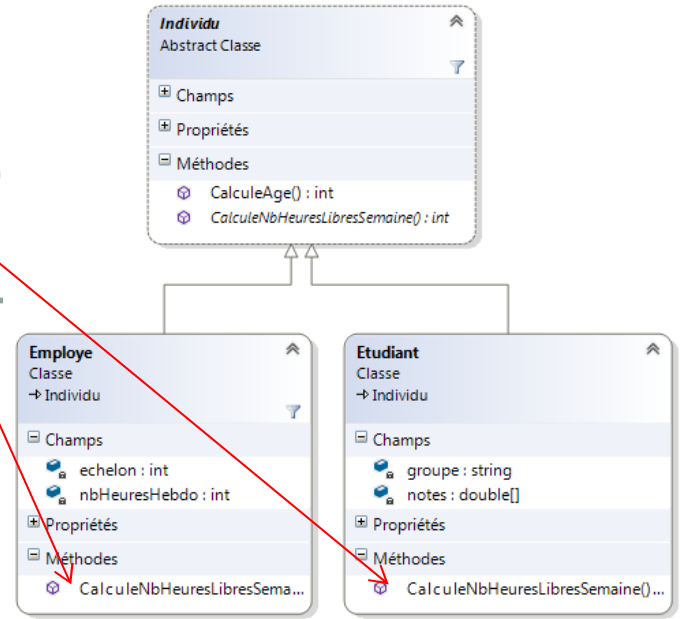
- Les classes filles doivent faire les traitements annoncés par la mère !

Liste à partir d'une classe abstraite

- On peut tout de même créer une liste d'individus : dans laquelle on mettra des Employes, des Enseignants (mais pas des Individus en tant que tels !).

```
List<Individu> lesIndividus = new List<Individu>();
lesIndividus.Add (new Individu("277109407803297", "Marla", new DateTime(1980, 12, 15)));
lesIndividus.Add (new Employe("135109407803297", "Billat", new DateTime(1970, 10, 5), 2,35));
lesIndividus.Add( new Etudiant("25610940780999", "Bois", new DateTime(1992, 2, 24),"1A"));

 foreach (Individu unIndividu in lesIndividus)
 {
    Console.WriteLine( unIndividu);
    Console.Write( unIndividu. CalculeNbHeuresLibresSemaine()
}
```



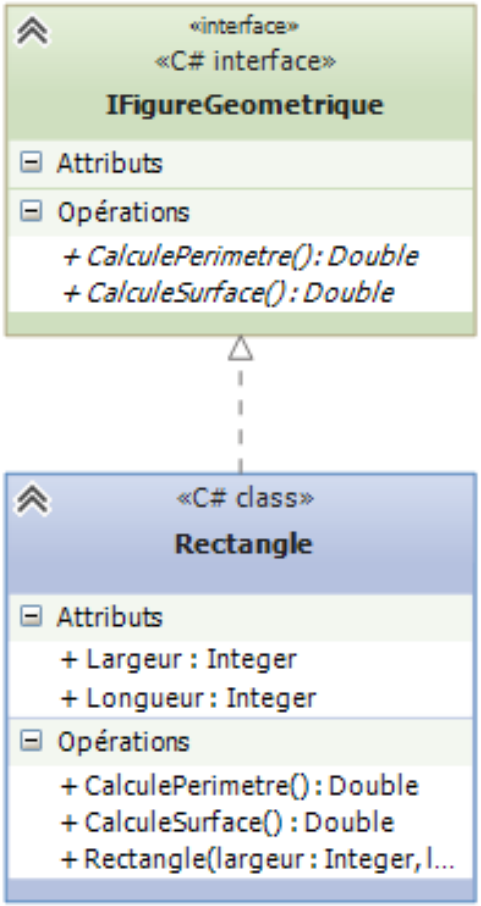
Interface : un héritage pas facile !

- Interface : classe totalement abstraite
- Hériter d'une interface = hériter d'obligations

```
interface IFigureGeometrique
  {
     double CalculePerimetre();
     double CalculeSurface();
  }
```

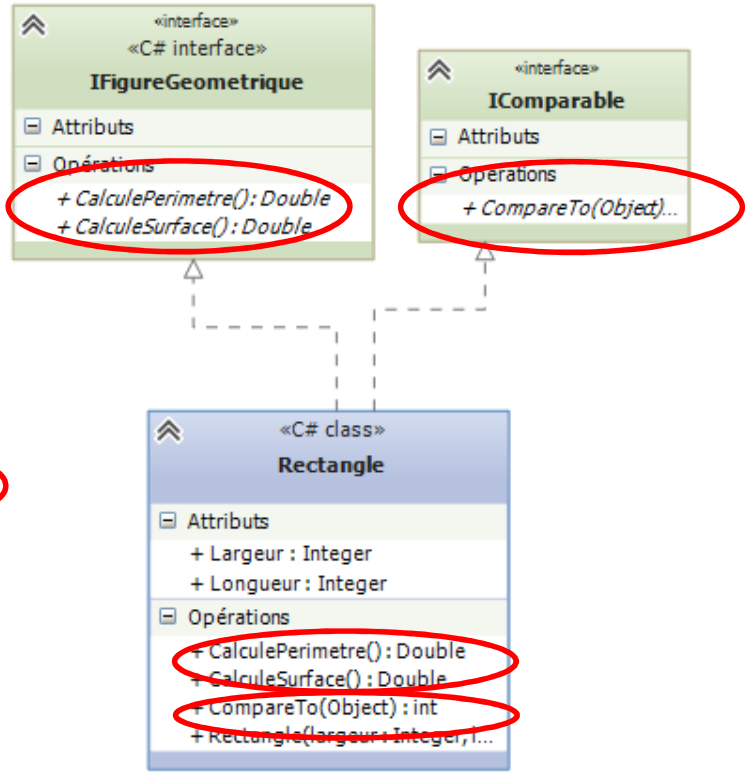
Pas besoin de mettre abstract

```
public class Rectangle : IFigureGeometrique
   {
      public double CalculePerimetre()
      { return 2 * (this.Longueur + this.Largeur); }
      public double CalculeSurface()
      { return this.Largeur * this.Longueur; }
    …
}
```



Interface et héritage

- On parle d'implémentation plus que d'héritage.
- Une classe peut implémenter plusieurs interfaces : Rectangle implémente IFigureGeometrique et IComparable




```
public class Rectangle : IFigureGeometrique , IComparable
   {
      public double CalculePerimetre()
      { return 2 * (this.Longueur + this.Largeur); }

      public double CalculeSurface()
      { return this.Largeur * this.Longueur; }

      public int CompareTo(Object o)
      {
        if (! ( o is IFigureGeometrique))
              throw new ArgumentException("Le paramètre passé n est pas une figure");
          IFigureGeometrique fig = (IFigureGeometrique)o;
          return  this.CalculeSurface().CompareTo(fig.CalculeSurface());
      }
    …
}
```

Liste à partir d'une interface

- On peut créer une liste de figures géométriques…

```
List<IFigureGeometrique> l = new List<IFigureGeometrique>();
l.Add ( new Cercle (6) );
l.Add ( new Rectangle(6,2) );
l.Sort() ;
foreach (IFigureGeometrique fig in l)
      {
        Console.WriteLine(fig);
        Console.WriteLine("Surface:" + fig.CalculeSurafce());
       }
```