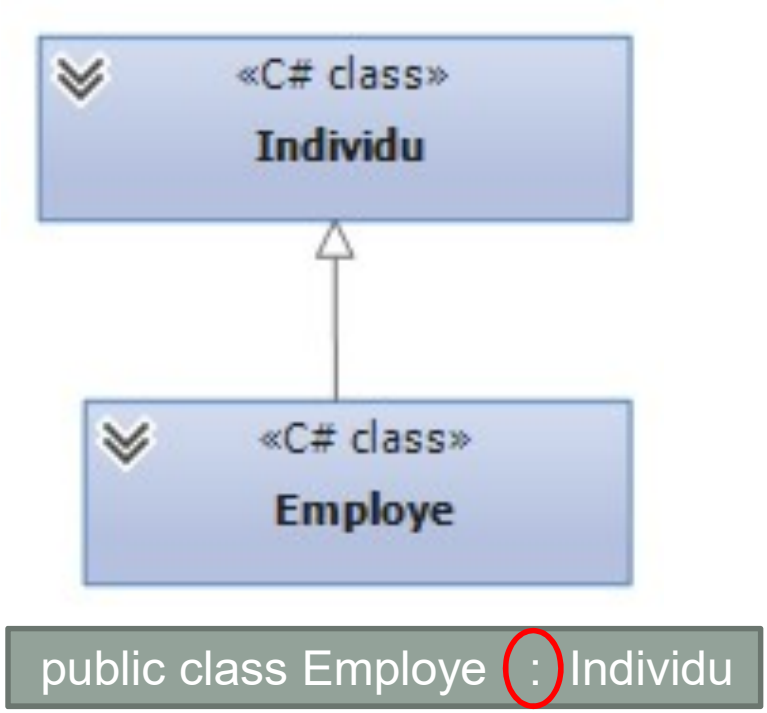
POO

Notion d'héritage

Vocabulaire

Un Employe est avant tout un Individu.

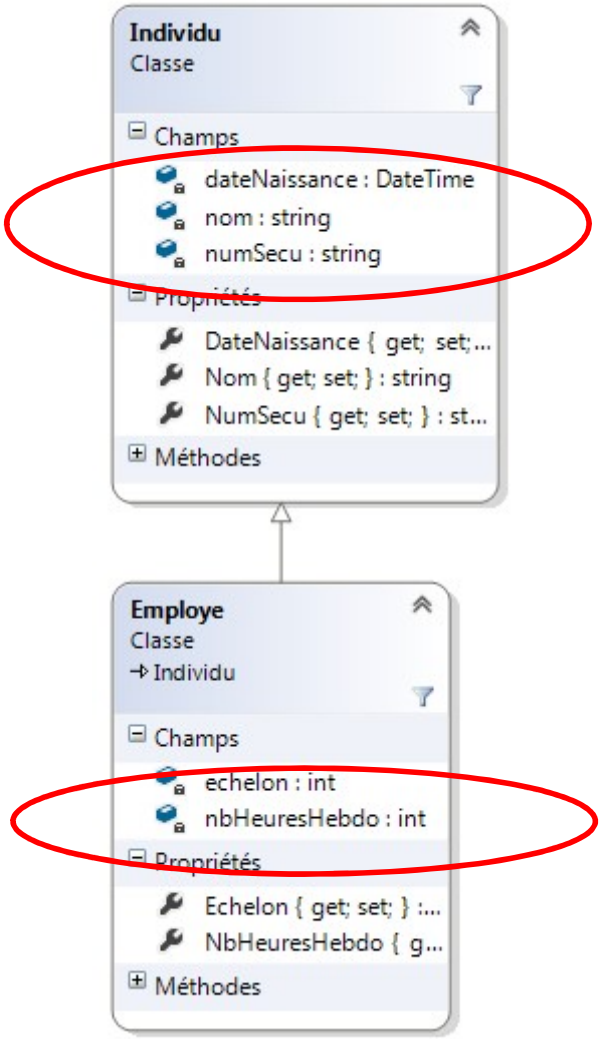
- Classe mère.
  - Ex : Individu
- Classe fille = classe dérivée.
  - Ex : Employe



La fille : mieux que la mère !

Une classe fille hérite de toutes les caractéristiques de sa mère, et elle en aura en plus. Ici :

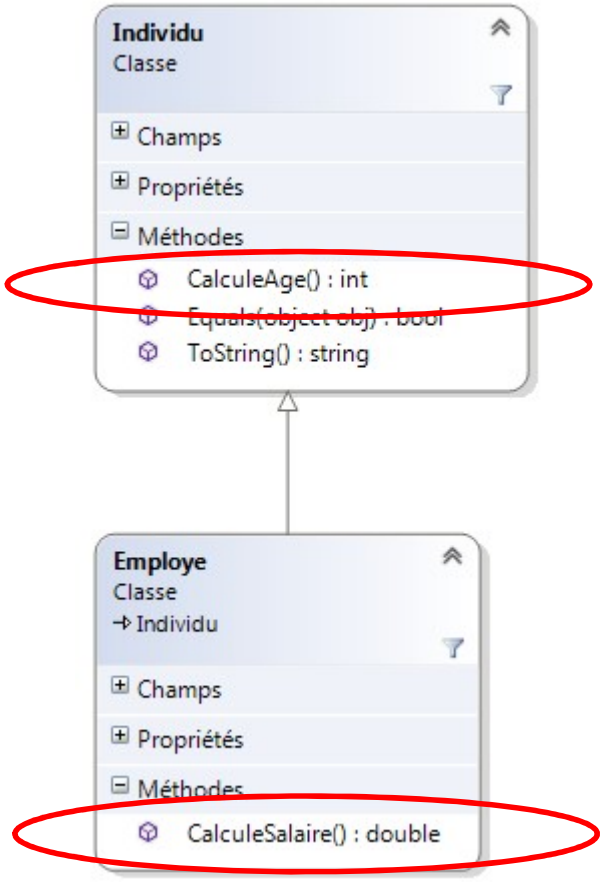
- Un Employe aura comme un Individu :
  - une date de naissance
  - un nom
  - un numéro de sécurité sociale
- Et il aura en plus :
  - Un échelon
  - Un nombre d'heures de travail hebdomadaire



La fille : mieux que la mère !

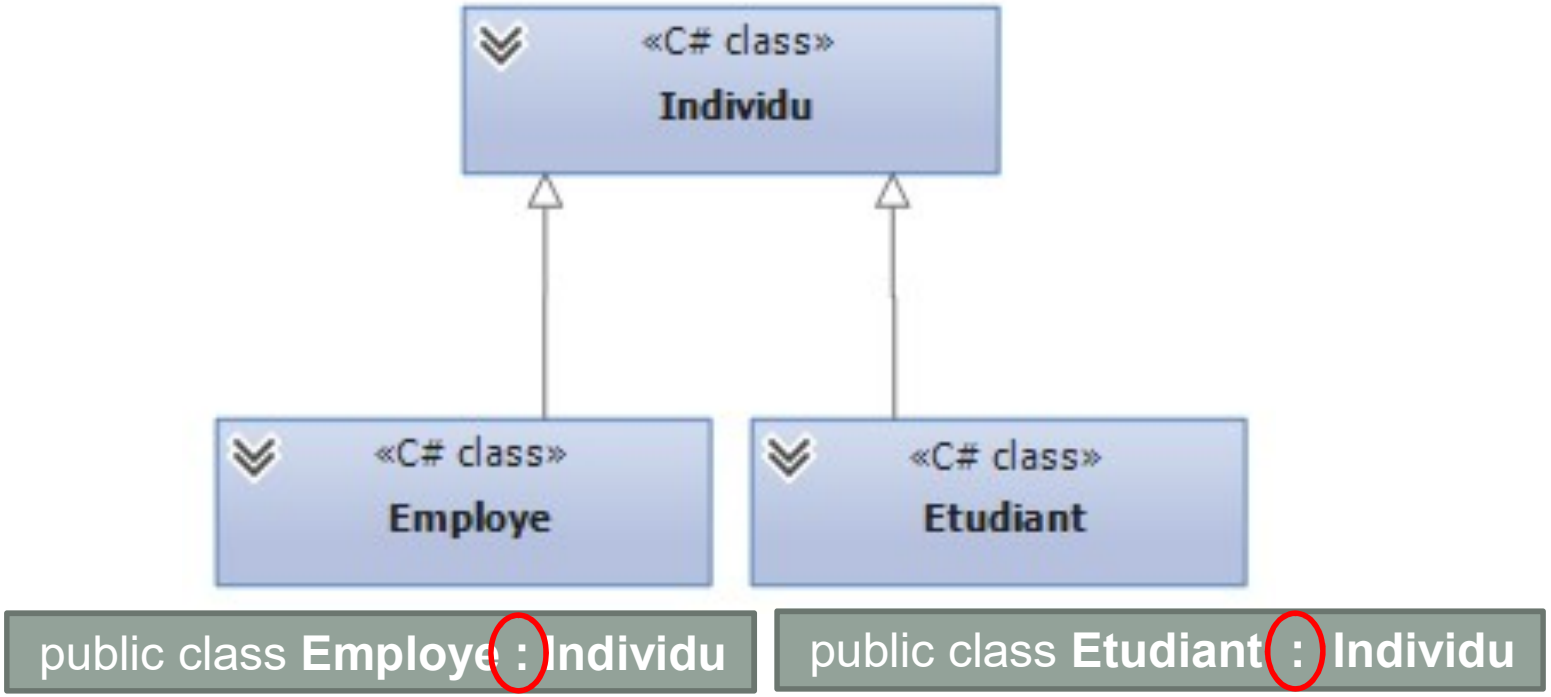
Une classe fille saura déjà faire tout ce que fait sa mère, et fera des choses en plus. Ici :

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



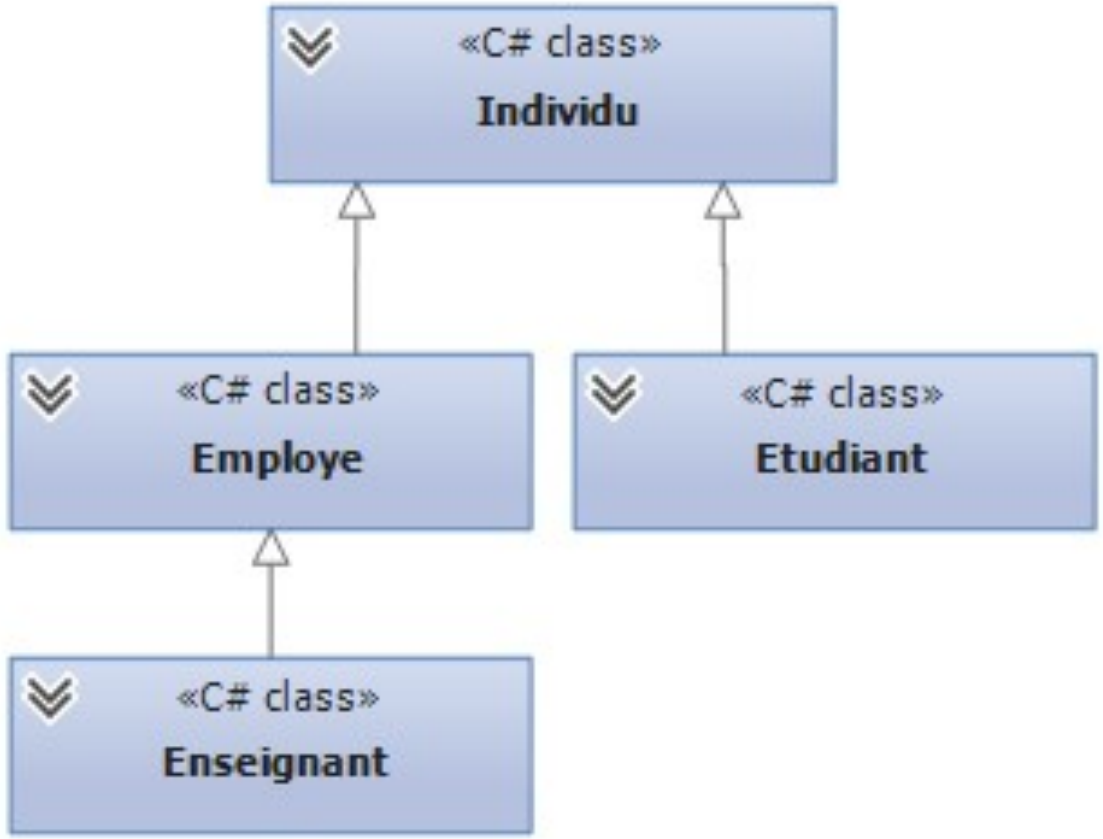
Une Mère, plusieurs filles

Ici, Employe et Etudiant sont filles de Individu.



Une Mère, plusieurs filles et « petites filles »

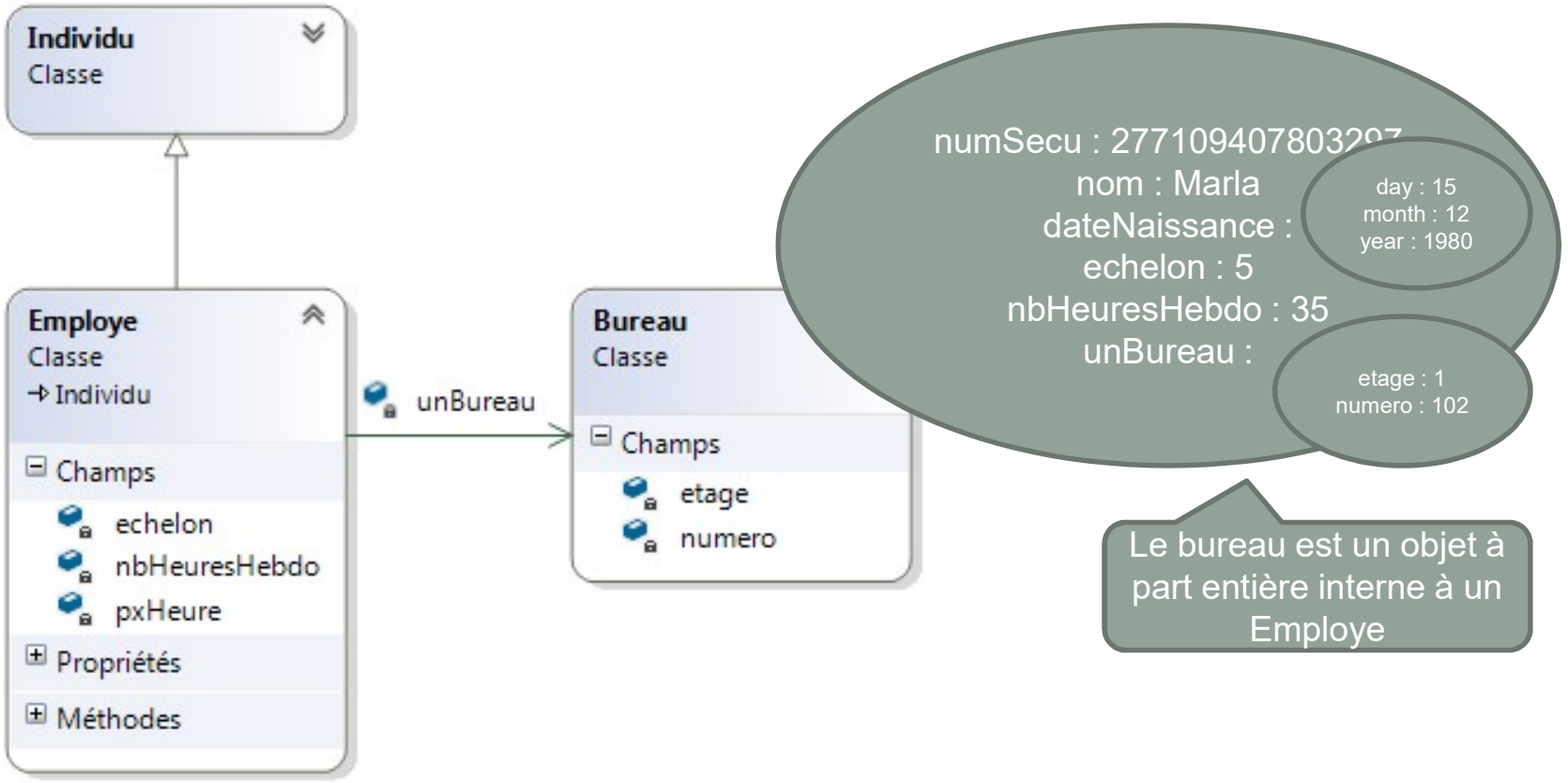
Ici, Enseignant est aussi fille de Individu, mais avant tout fille de Employe.




```
public class Enseignant : Employe
```

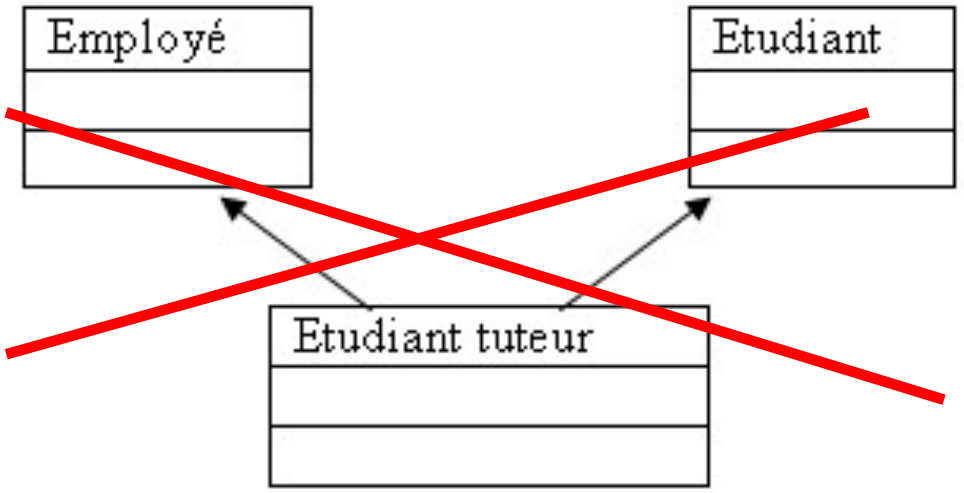
Ne pas mélanger association et héritage

- Un Employe est avant tout un Individu .
- Un Employe a un bureau.



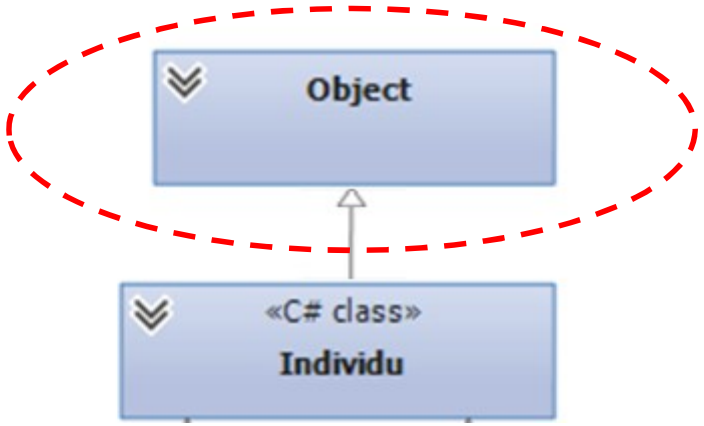
Une seule mère directe

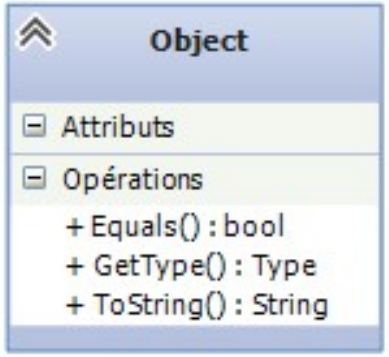
L'héritage multiple n'existe pas.



La classe mère suprême

Toute classe hérite (implicitement) de la classe Object





public class **Individu : Object**

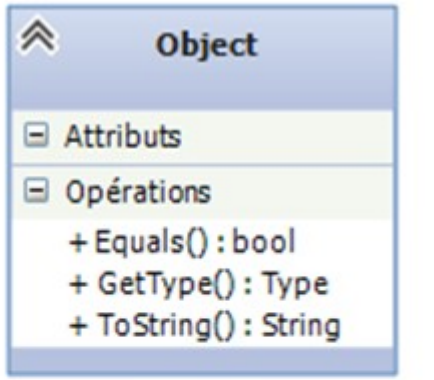
public class **Individu**

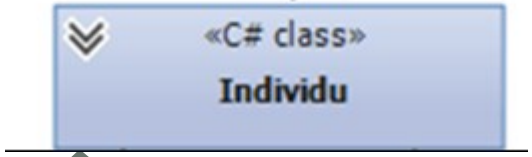
:Object est implicite !

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());
```

ToString() pas défini ici => appel à ToString la classe Object

```
ConsoleApplication1.Individu
```

Namespace.nomClasse

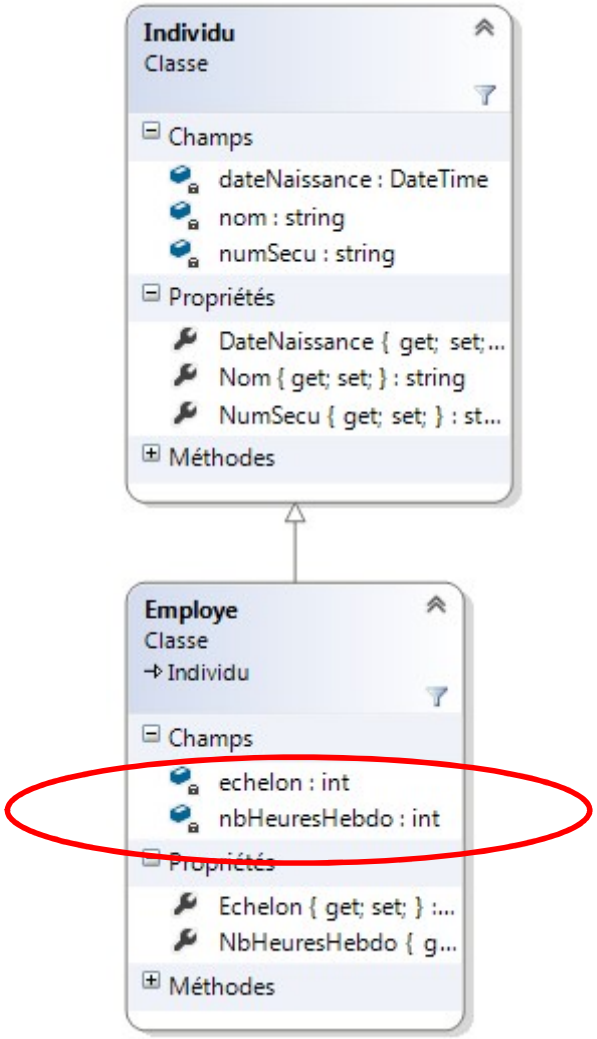
Définir une classe fille

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

```
public class Individu
   {
      private String numSecu;
      public String NumSecu
      {
         get { return numSecu; }
         set { numSecu = value; }
      }
      private String nom;
      public String Nom
      {
         get { return nom; }
         set { nom = value;
      }
      private DateTime dat
      public DateTime Date
      {
         get { return dateNa
         set { dateNaissanc
      }
```

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

      private int nbHeuresHebdo;
      public int NbHeuresHebdo
      {
         get { return nbHeuresHebdo; }
         set { nbHeuresHebdo = value; }
      }
```



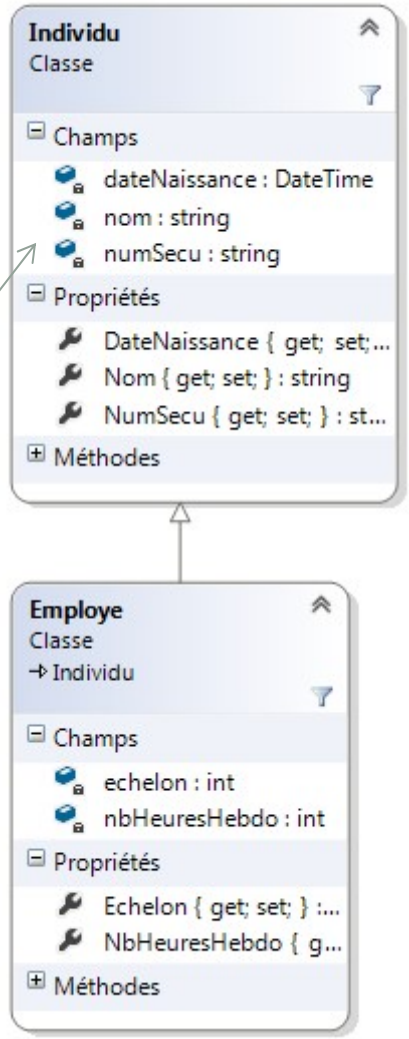
Constructeur d'une classe fille

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

```
public Employe(string pNumSecu, String pNom,
        DateTime pDateNaissance, int pEchelon, int pNbHeures)
```

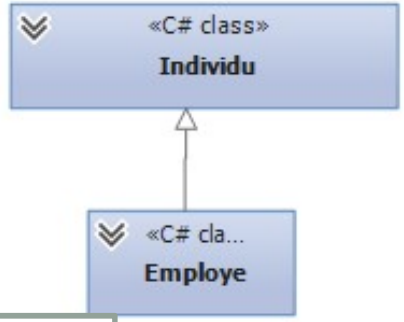
```
Employe e = new Employe("277109407803297", "Marla",
                        new DateTime(1980, 12, 15),
                        5,35);
```

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



Constructeur d'une classe fille

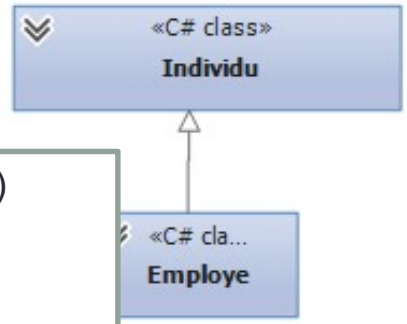
- Il s'appuie sur le constructeur de la classe mère pour initialiser les champs hérités : on réutilise !
- Il initialise les champs spécifiques




```
public Individu( string pNumSecu, String pNom, DateTime pDateNaissance)
        {  this.NumSecu = pNumSecu;
           this.Nom = pNom;
           this.DateNaissance = pDateNaissance;
        }
```

```
public Employe(string pNumSecu, String pNom, DateTime pDateNaissance, int pEchelon, int pNbHeures)
    : base(pNumSecu, pNom, pDateNaissance)
    { this.Echelon = pEchelon;
      this.NbHeuresHebdo = pNbHeures;
    }
```

Constructeur d'une classe fille



- Attention : this / base !

```
public Individu( string pNumSecu, String pNom, DateTime pDateNaissance)
        {  this.NumSecu = pNumSecu;
           this.Nom = pNom;
           this.DateNaissance = pDateNaissance;
        }
```

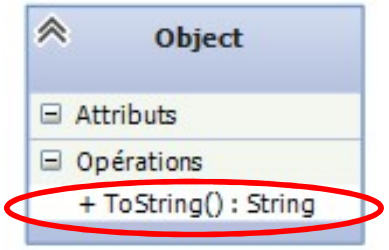
base : cherche dans la classe mère

```
public Employe(string pNumSecu, String pNom, DateTime pDateNaissance,
int pEchelon, int pNbHeures)  : base(pNumSecu, pNom, pDateNaissance)
      { this.Echelon = pEchelon;
        this.NbHeuresHebdo = pNbHeures;
      }
```

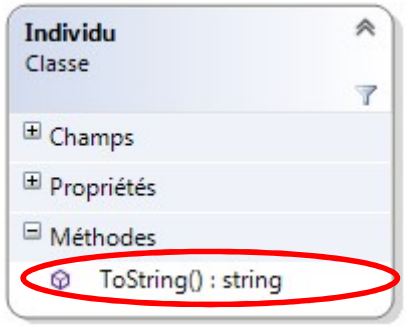
this : cherche dans la même classe

```
public Employe(string pNumSecu, String pNom, DateTime pDateNaissance, int pEchelon)  :
this(pNumSecu, pNom, pDateNaissance, pEchelon, 35)
      {    }
```

Substituer des méthodes héritées

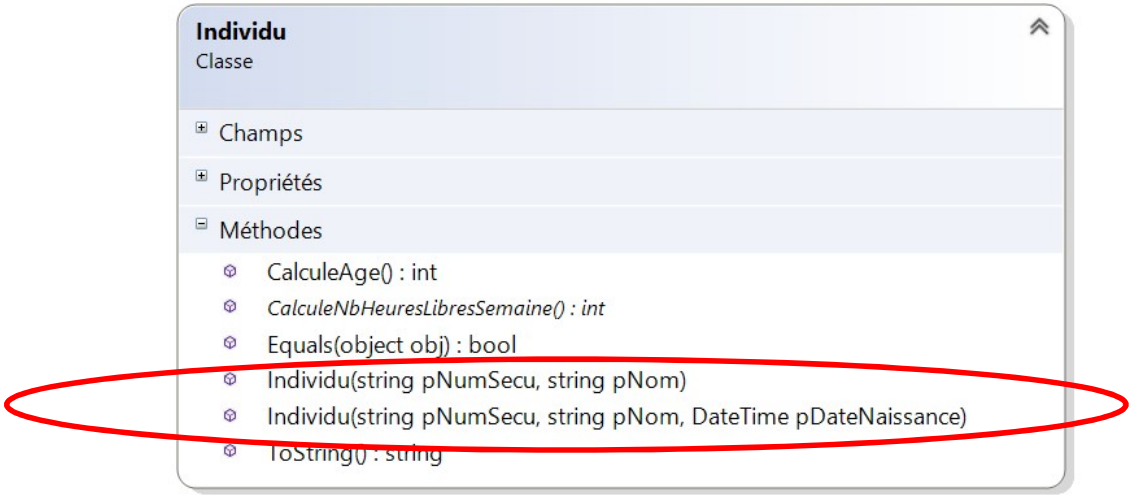


- Concept de **substitution (ou redéfinition)** : au sein d'une classe fille, on peut redéfinir le comportement des méthodes héritées :
  - même signature (forme)



A NE PAS CONFONDRE AVEC

- Concept de **surcharge** : au sein d'une même classe
  - Différentes signatures



Substitution totale

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

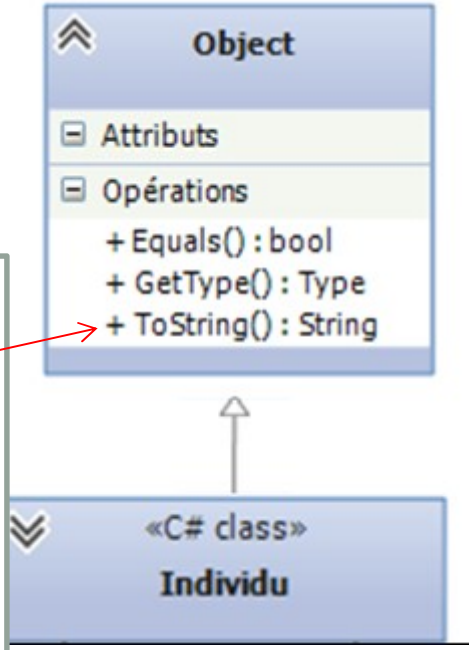
```
public override string ToString()
 {
  return "\nNumero de sécurité sociale : " + this.NumSecu
    + "\nNom : " + this.Nom
    + "\nDate de naissance : " + this.DateNaissance.ToString("d") ;
 }
```

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

Substitution « partielle »

- Appel à la méthode héritée : base

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




```
ConsoleApplication1.Individu
Numero de sécurité sociale : 2771094078032 97
Nom : Marla
Date de naissance : 15/12/1980
```

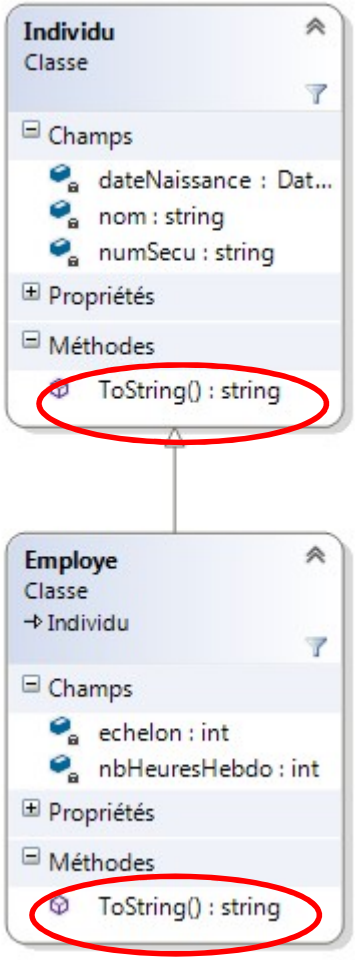
Substitution « partielle »

- Appel à la méthode héritée : base

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

```
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
```



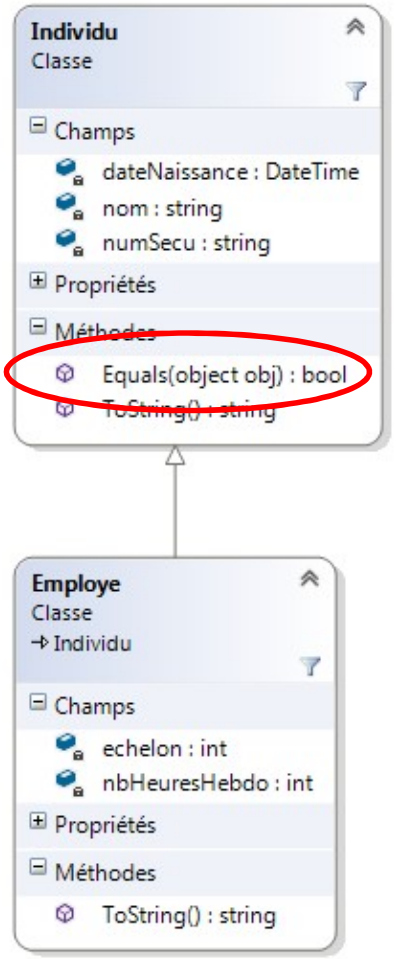
Substitution : choix ou obligation ?

- **Pas une obligation !** La fille a alors le même comportement que la mère.
- Dépend du contexte. Exemple : si seul le numéro de sécurité sociale sert à identifier un individu, pas de substitution de Equals dans Employe !

```
class Individu
{

public override bool Equals(object obj)
        {
          if (obj == null)
              return false;
          if (obj.GetType() != this.GetType())
              return false;
          Individu i = obj as Individu;
          return i.NumSecu.Equals(this.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 ))
```

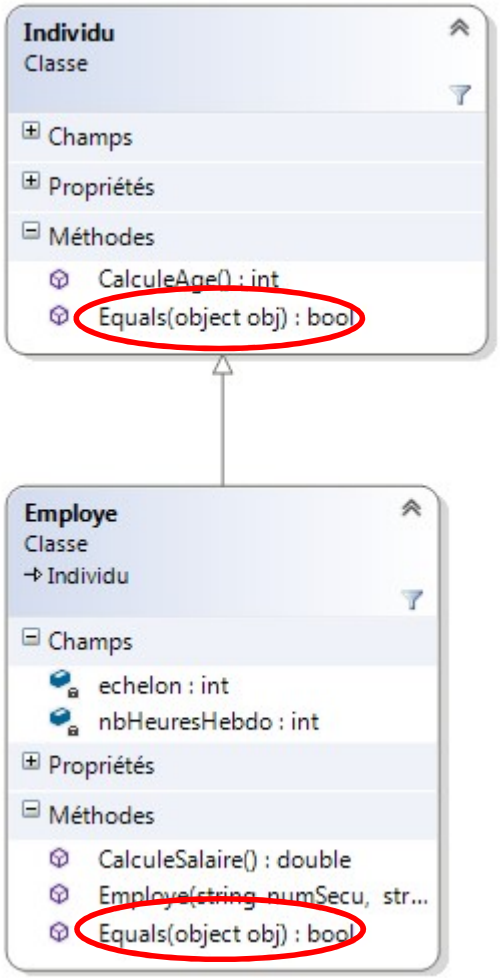


Substitution : choix ou obligation ?

- **Mais souvent nécessaire !** Pour affiner le comportement hérité de la mère.
- Dépend du contexte. Exemple : si par contre, il faut comparer tous les champs, la substitution de Equals est alors nécessaire !

```
class Individu
{
    public override bool Equals(object obj)
        {
          if (obj == null)
              return false;
          if (obj.GetType() != this.GetType())
              return false;
          Individu i = (Individu) obj ;
          return i.NumSecu == this.NumSecu
                && i.Nom == this.Nom && i.DateNaissance == this.DateNaissance;
        }
}
```

```
class Employe : Individu
{
  public override bool Equals(object obj)
      {
          bool res = base.Equals(obj);
          if (res == false)
              return false;
          Employe e = (Employe) obj;
          return e.Echelon == this.Echelon &&
               e.NbHeuresHebdo == this.NbHeuresHebdo;
      }
```



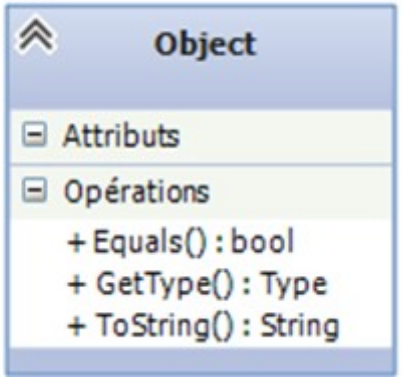
Substitution : la mère décide !

- **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 !

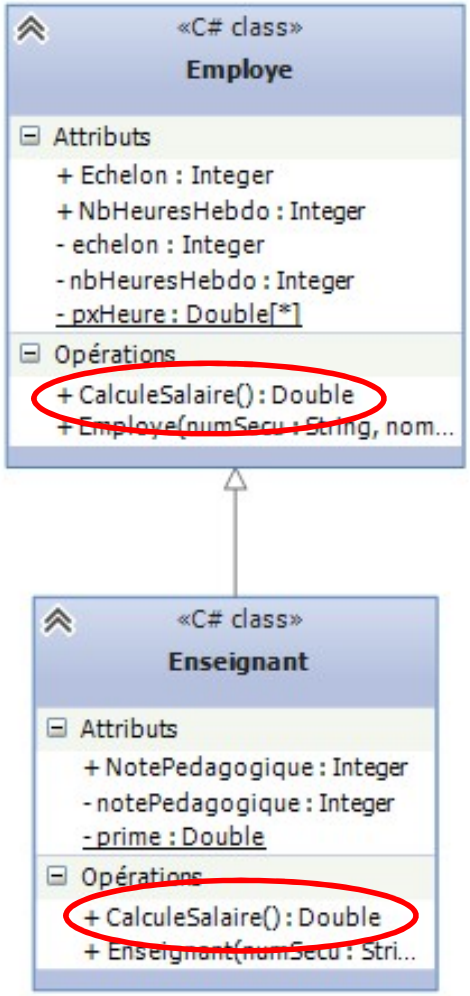
```
public  class Employe
{
   public  virtual  double CalculeSalaire()

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

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

        { return base.CalculeSalaire()

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

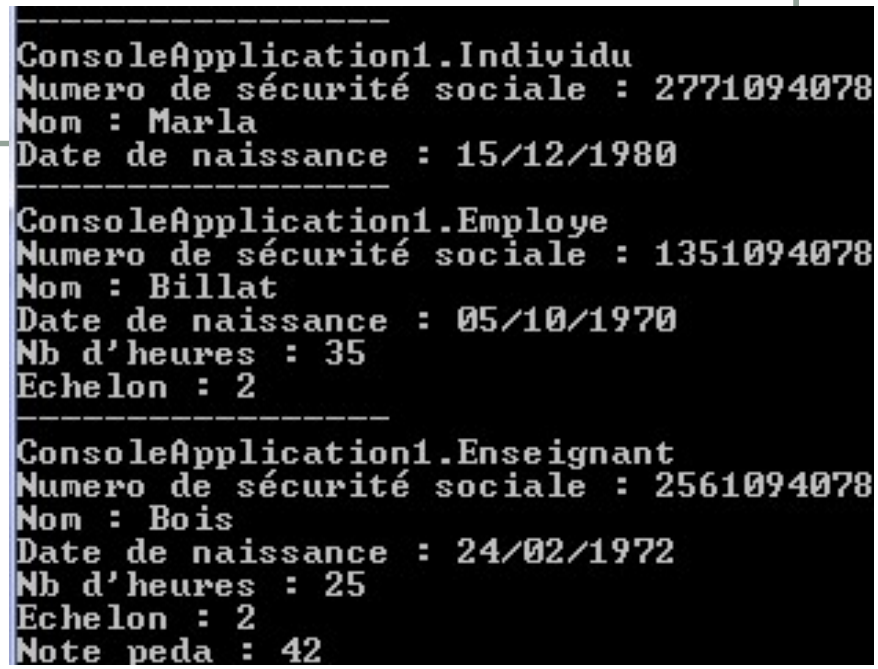


Intérêt de l'héritage

- Factoriser du code commun
- Manipuler des listes d'objets hétérogènes mais ayant des points communs tout de même !

```
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 Enseignant("25610940780999", "Bois", new DateTime(1972, 2, 24), 2, 25,42));

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



Accès aux champs hérités

- Accès protected = private sauf pour les classes filles

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

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

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

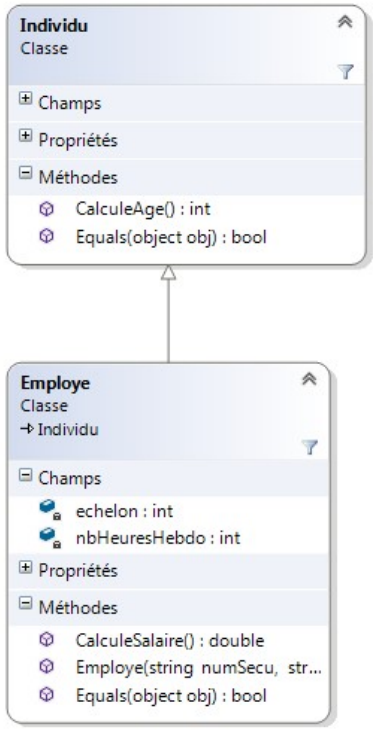
Faire un cast

- Il faut parfois énoncer explicitement le type d'un objet

```
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());

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



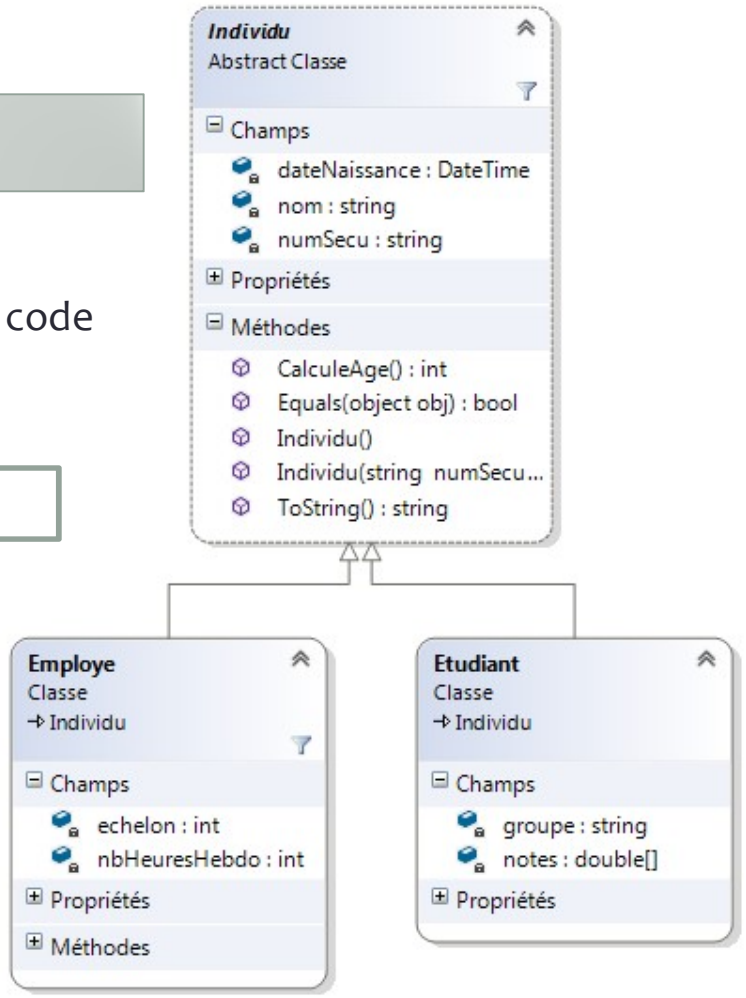
Notion de classe abstraite

- 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

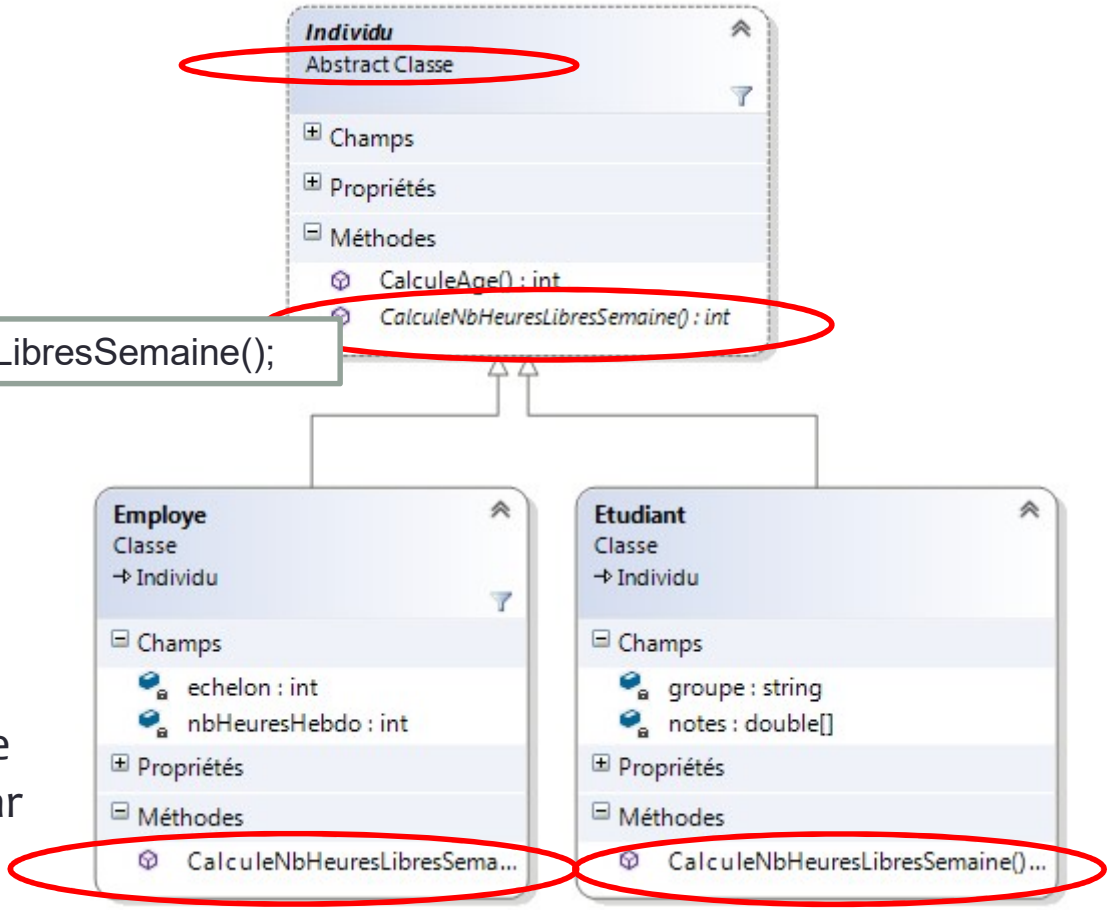
```
public abstract class Individu {
```



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.

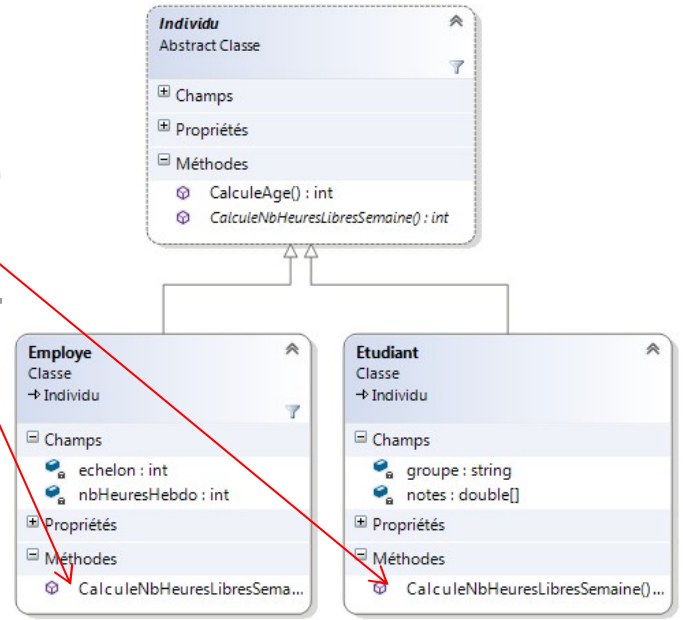
- Les classes filles doivent faire les traitements demandés par la mère !

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()
}
```



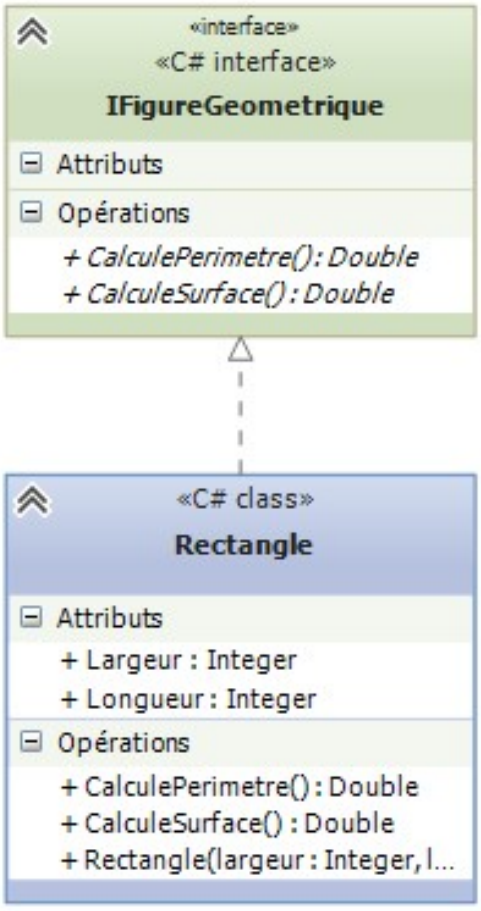
Interface : un héritage pas facile !

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

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

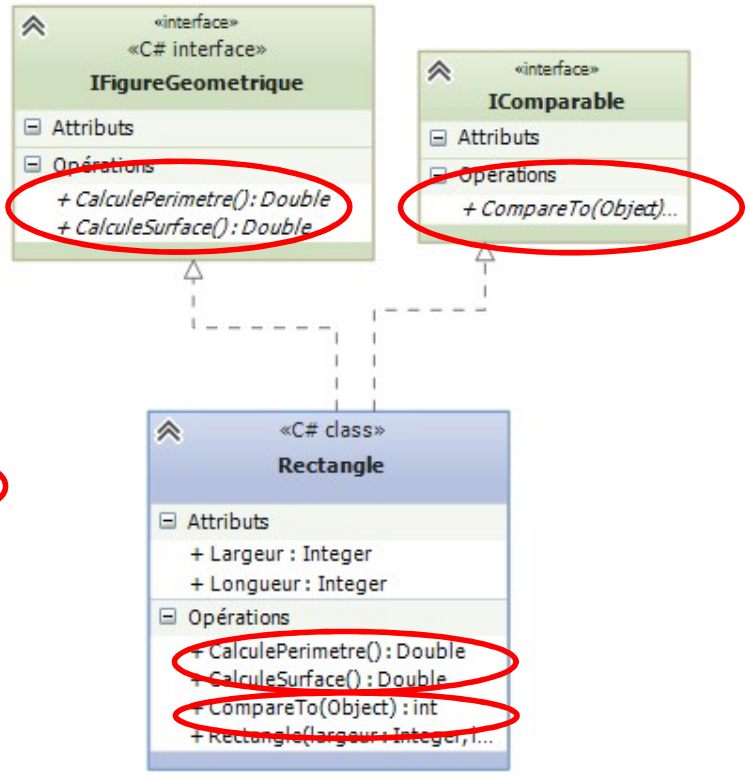
Pas besoin de mettre abstract

```
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)
      {
         Rectangle r = (Rectangle)o;
         if (r == null)
            throw new ArgumentException("Le paramètre passé n'est pas un rectangle");
         else
           return (int) Math.Round( CalculeSurface() - r.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) );
foreach (IFigureGeometrique fig in l)
      {
        Console.WriteLine(fig);
        Console.WriteLine("Perimetre:" + fig.CalculePerimetre());
       }
```