

# POO

---

Manipuler un objet

# Concept de classe - Généralités

On peut être amené à écrire 3 sortes de classes :

- Les « vraies classes » :

Servent à définir de nouveaux types

- Les « classes librairies » :







Servent à regrouper des fonctions (méthodes statiques)

- Les « classes applications » :

Servent à écrire le programme principal (main)

# Classes librairies - Exemple

- Classe Math : que des méthodes statiques !
- Pas besoin d'objets pour les appeler

	<code>Ceiling(Double)</code>	Retourne la plus petite valeur intégrale supérieure ou égale au nombre à virgule flottante double précision spécifié.
	<code>Cos(Double)</code>	Retourne le cosinus de l'angle spécifié.
	<code>Cosh(Double)</code>	Retourne le cosinus hyperbolique de l'angle spécifié.
	<code>DivRem(Int32, Int32, Int32)</code>	Calcule le quotient de deux entiers signés 32 bits et retourne également le reste dans un paramètre de sortie.
	<code>DivRem(Int64, Int64, Int64)</code>	Calcule le quotient de deux entiers signés 64 bits et retourne également le reste dans un paramètre de sortie.
	<code>Exp(Double)</code>	Retourne $e$ élevé à la puissance spécifiée.

```
double res = 1.57;  
res= Math.Ceiling(res);
```

# Ecrire une classe librairie

- Pas très objet ! Mais parfois utile !

```
class Read
{
    public static float ReadFloat()
    {
        String saisie = Console.ReadLine();
        float d;
        while (Single.TryParse(saisie, out d) == false)
        {
            Console.WriteLine("Vous devez saisir un nombre réel.");
            saisie = Console.ReadLine();
        }
        return d;
    }
    public static int ReadInt()
    {
        String saisie = Console.ReadLine();
        int entier;
        while (Int32.TryParse(saisie, out entier) == false)
        {
```

float f = Read.ReadFloat();  
int i = Read.ReadInt();

... •

# Vocabulaire objet

Avant en Prog. Procédural	Maintenant en POO
On déclarait une variable	On déclare un <b>objet</b>
On initialisait une variable	On <b>instancie</b> un <b>objet</b> On <b>instancie</b> une <b>instance</b> de <b>Classe</b>
Une variable était définie par son type	Un <b>objet</b> est défini par sa <b>Classe</b> (ou structure)
On utilisait des fonctions	On utilise des <b>méthodes</b>

# Conventions de nommage

## Valables pour beaucoup de langages OO: Java, c# ,....

Une classe commence par une majuscule.	Ex : Point
--	------------

Un objet commence par une minuscule.	Ex : p1
--------------------------------------	---------

## Valables uniquement pour C# :

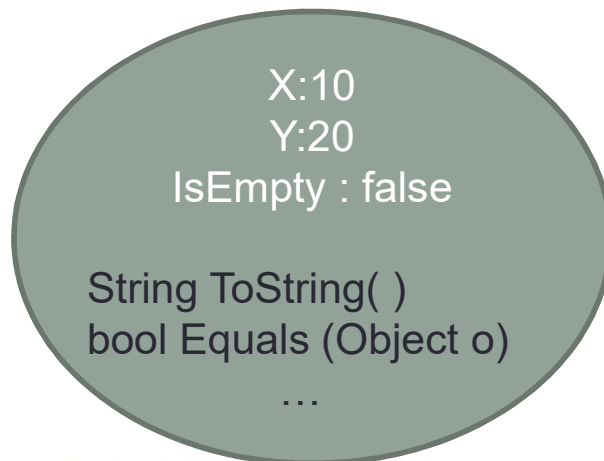
Toute méthode commence par une majuscule.	Ex : ToString
---	---------------

Toute propriété commence par une majuscule.	Ex : IsEmpty
---	--------------

# Objet = variable complexe

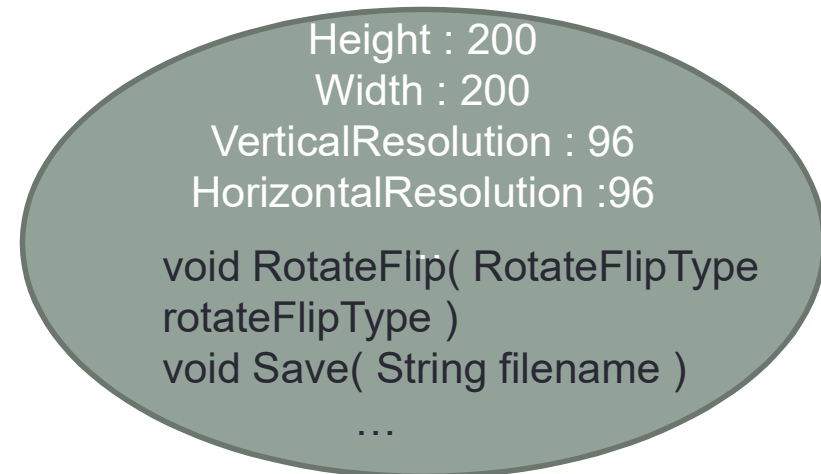
- Constituée de plusieurs données
- Capable d'exécuter un traitement
- Définie par une classe (ou une structure : C# uniquement)

Exemple : objet Point



Point, structure

Exemple : objet Bitmap



Bitmap, classe

# Déclarer un objet

C'est déclarer une variable d'un type rangé dans un namespace (« répertoire », package) auquel il faut faire référence.

```
using System.Drawing;  
  
namespace DemoPOO_02  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Point p1 ; // p1 est un objet  
        }  
    }  
}
```

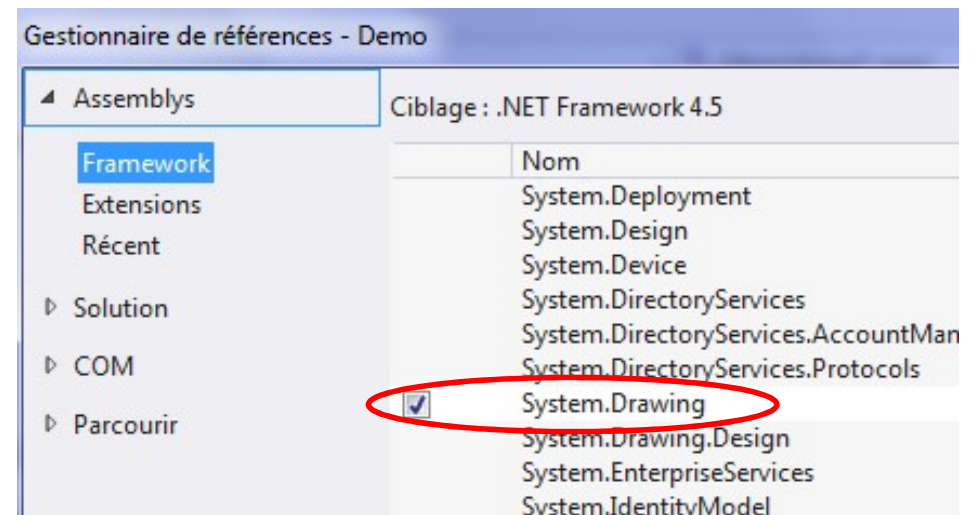
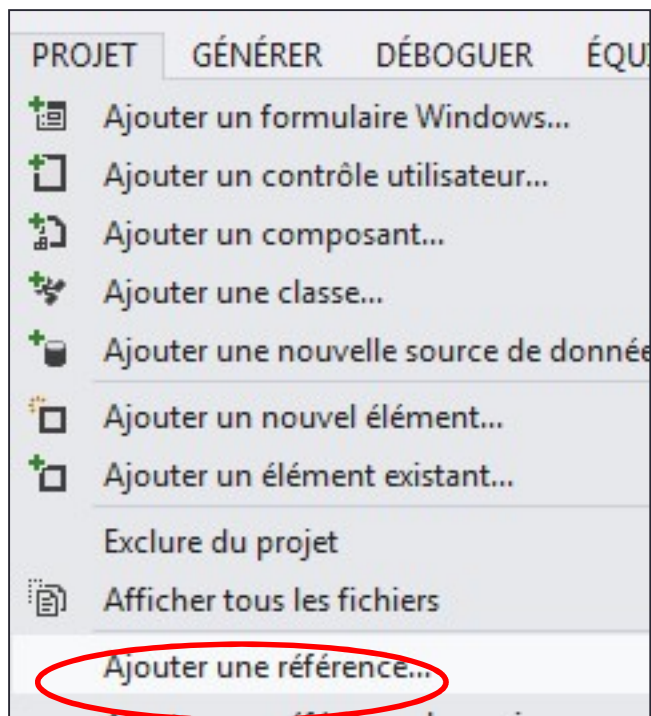
Point est rangé au sein du namespace System.Drawing



# Déclarer un objet

Si le namespace ne fait pas parti des namespaces les plus utilisés, il faut alors l'ajouter au projet.

**using System.Drawing;**



# Instancier un objet

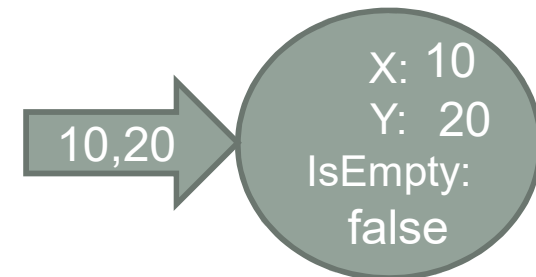
C'est :

- Allouer la mémoire suffisante. Mot clef : **new**
- Initialiser les propriétés de l'objet. Appel à un **constructeur** : méthode qui porte le même nom que la classe

```
// on déclare p objet de classe Point  
Point p ;  
// on instancie p  
p = new Point(10,20);
```

constructeur

```
// on peut déclarer et instancier  
Point p = new Point(10,20);
```



# Constructeur

- Méthode qui initialise les données internes à l'objet
- Méthode bien souvent surchargée

<code>Point(Size)</code>	Initialise une nouvelle instance de la classe Point à partir d'un <code>Size</code> .
<code>Point(Int32, Int32)</code>	Initialise une nouvelle instance de la classe Point avec les coordonnées spécifiées.

```
Size s = new Size (10,20);  
Point p = new Point( s );
```

```
Point p = new Point(10,20);
```

- Constructeur par défaut (sans paramètre) souvent existant.




```
Point p = new Point( );
```

# Manipuler les propriétés d'un objet

- Pour cela, il faut connaître sa classe ! Consultez la doc MSDN.

## Point

### ▲ Propriétés

	Nom	Description
	IsEmpty	Obtient une valeur indiquant si ce Point est vide.
	X	Obtient ou définit la coordonnée x de ce Point.
	Y	Obtient ou définit la coordonnée y de ce Point.


Uniquement consultable

Consultable et modifiable

- Certaines propriétés sont modifiables. D'autres sont parfois uniquement consultables.

# Manipuler les propriétés d'un objet

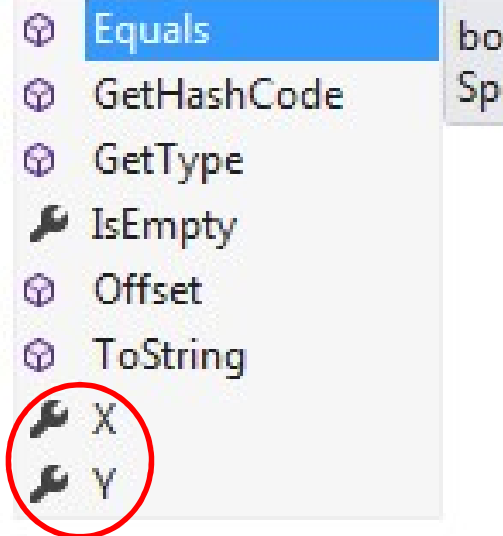
Les propriétés sont :

- à préfixer par l'objet. **Syntaxe:** objet.propriété
- représentées par  au sein de visual studio

```
Point p = new Point( );  
p.X = 10 ;
```

```
Point p = new Point();
```

```
p.
```



# Déclencher des traitements à partir d'un objet

- Voir quels traitements sont possibles : consultez les méthodes


## Point

### ▲ Méthodes

	Nom	Description
	Add	Ajoute le <code>Size</code> spécifié au Point indiqué.
	Ceiling	Convertit le <code>PointF</code> spécifié en Point en arrondissant les valeurs de <code>PointF</code> aux valeurs entières supérieures.
	Equals	Spécifie si ce Point contient les mêmes coordonnées que le <code>Object</code> spécifié. (Substitue <code>ValueType.Equals(Object)</code> .)
	<u>ToString</u>	Convertit ce Point en chaîne explicite. (Substitue <code>ValueType.ToString()</code> .)

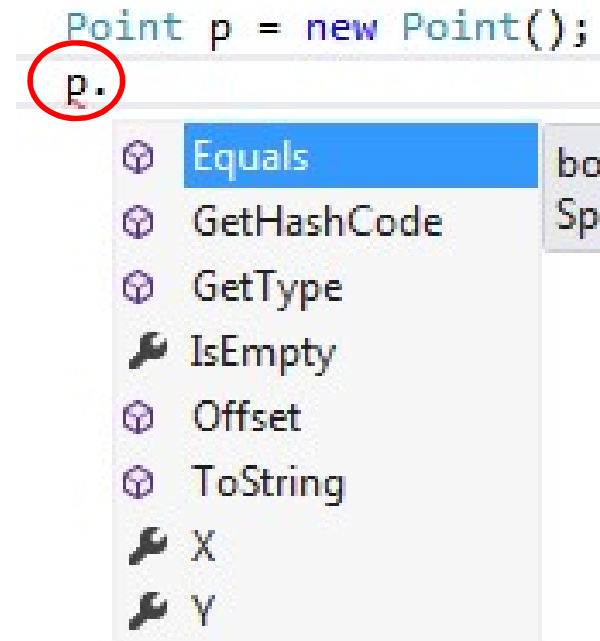
# Déclencher une méthode (d'instance)

Les méthodes sont :

- à préfixer par l'objet. **Syntaxe** objet.Méthode( );
- représentées par  au sein de visual studio

```
Point p = new Point( );  
String textSurP = p.ToString();
```

L'objet



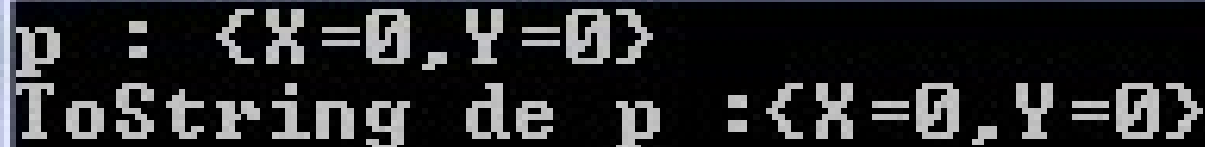
## Des méthodes communes à tous les objets

Certaines méthodes sont redéfinies dans toutes les classes :

- **String ToString( )** : Convertit l'objet en chaîne explicite.
- **bool Equals( Object obj )** : Spécifie si l'objet contient les mêmes données que l'objet passé en paramètre.
- ...

Elles sont appelées parfois à « votre insu »

```
Console.WriteLine("p : " + p);  
Console.WriteLine("ToString de p : " + p.ToString());
```






```
p : {X=0,Y=0}  
ToString de p : {X=0,Y=0}
```



# Méthodes de classe

- Facile à différencier dans la doc :
- **S** => statique = Méthode de classe


	Nom	Description
	Add	Ajoute le <a href="#">Size</a> spécifié au Point indiqué.
	Ceiling	Convertit le <a href="#">PointF</a> spécifié en Point en arrondissant les valeurs de <a href="#">PointF</a> aux valeurs entières supérieures.
	Equals	Spécifie si ce Point contient les mêmes coordonnées que le <a href="#">Object</a> spécifié. (Substitue <a href="#">ValueType.Equals(Object)</a> .)
	GetHashCode	Retourne un code de hachage pour ce Point. (Substitue <a href="#">ValueType.GetHashCode()</a> .)
	GetType	Obtient le <a href="#">Type</a> de l'instance actuelle. (Hérité de <a href="#">Object</a> .)
	Offset(Point)	Convertit ce Point selon le Point spécifié.
	Offset(Int32, Int32)	Convertit ce Point selon la valeur spécifiée.

# Méthodes d'instance <> Méthodes de classe

Méthodes de classe ou méthode statiques :

- Résidu de fonctions « procédurales ».
- Ne s'appliquent pas sur un objet : objet passé en paramètre.


Méthode d'instance

	Nom	Description
	ToString	Convertit ce Point en chaîne explicite.

```
Point p = new Point(5,5);
Size s = new Size (10,10);
Console.WriteLine(p.ToString() );
```

L'objet

Méthode de classe

	Nom	Description
	Add	Ajoute le Size spécifié au Point indiqué.

```
Point p = new Point(5,5);
Size s = new Size (10,10);
Point np = Point.Add (p , s);
```

La classe

L'objet

## Structure (type valeur) / Classe (type référence)

Structure (type valeur) Point	Classe (type référence) Bitmap
<pre>Point p1 = new Point(100, 100);  // copie de valeur Point p2 = p1;  // modification de p1 p1.X = 150; Console.WriteLine("p1 " + p1); Console.WriteLine("p2 " + p2);</pre>	<pre>Bitmap image1 = new Bitmap(200, 200);  // copie de référence Bitmap image2 = image1;  // modification de image1 for (int y = 1; y &lt; image1.Height; y++)     for (int x = 1; x &lt; image1.Width; x++)         image1.SetPixel(x, y, Color.Red); image1.Save("image1.bmp"); image2.Save("image2.bmp");</pre>

```
p1 {X=150,Y=100}  
p2 {X=100,Y=100}
```

p2 copie de p1. p1 est modifié, pas p2

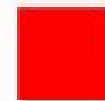


image1.bmp



image2.bmp

image2 copie de image1. image1 est modifiée, image2 aussi

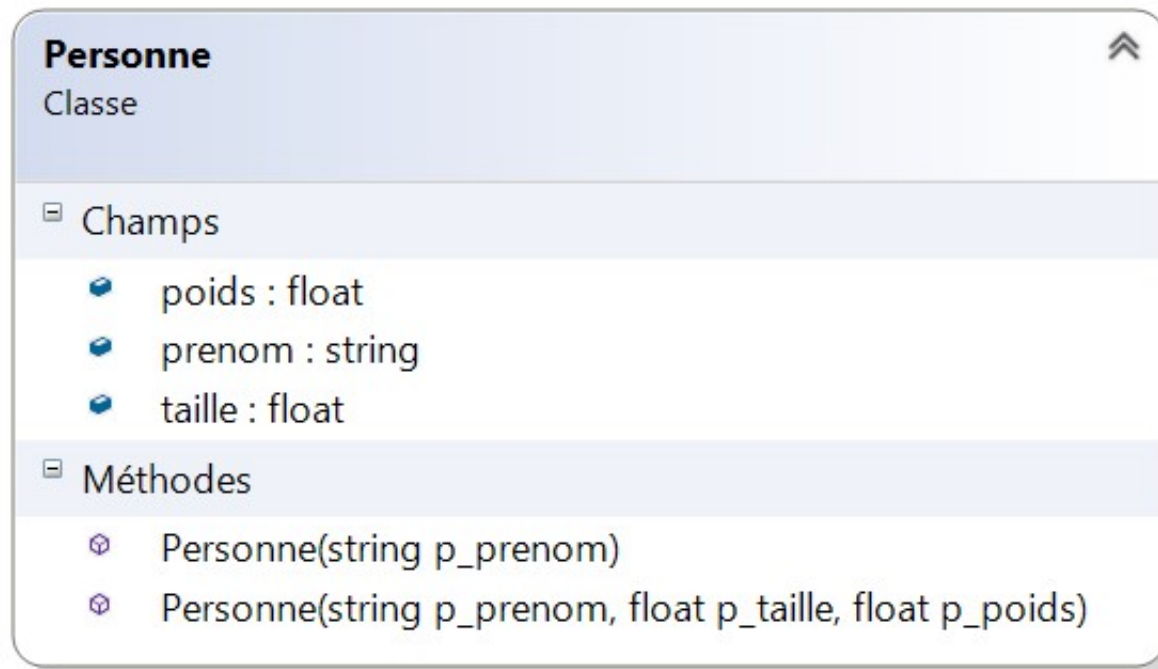
# POO

---

Définir ses propres classes

# Exemple simple

- Une personne, c'est :
  - Un prenom
  - Une taille
  - Un poids



# Exemple simple

On a alors 2 classes : 2 fichiers

- La classe Personne : définition du type Personne
- La classe Program : main utilisant Personne



```
Personne p = new Personne("Anae", 1.40F, 35);
```

# Définir les champs : public/ private

Un champ peut être « public » : accès autorisé en dehors de la classe de définition.

```
class Personne
{
    public String prenom;
    public float taille;
    public float poids;
}
```

Possible mais déconseillé !  
Une classe doit veiller à la  
cohérence des données internes.

```
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        p.prenom = "aNaé";
        p.taille = 4.50;
        p.poids = -45;
    }
}
```

Données incohérentes ! Mais  
l'affectation se fait.

# Définir les champs : private

Un champ :

- Doit être « private »
- Peut être associé à une propriété « public » pour y donner accès

```
class Personne
{
    private String prenom;
    public string Prenom
    {
        get
        {
            return prenom;
        }
        set
        {
            prenom = value;
        }
    }
}
```

Prenom : propriété pour champ prenom

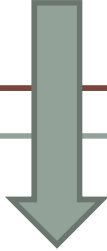
Code (squelette) généré  
automatiquement avec le raccourci  
CTRL R puis CTRL E  
À modifier pour assurer la cohérence  
des données internes



# Définir les champs : encapsulation

On passe donc par un intermédiaire interne : `Prenom` à la classe pour modifier la donnée interne : `prenom`.

```
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        p.Prenom = "aNaé";
    }
}
```



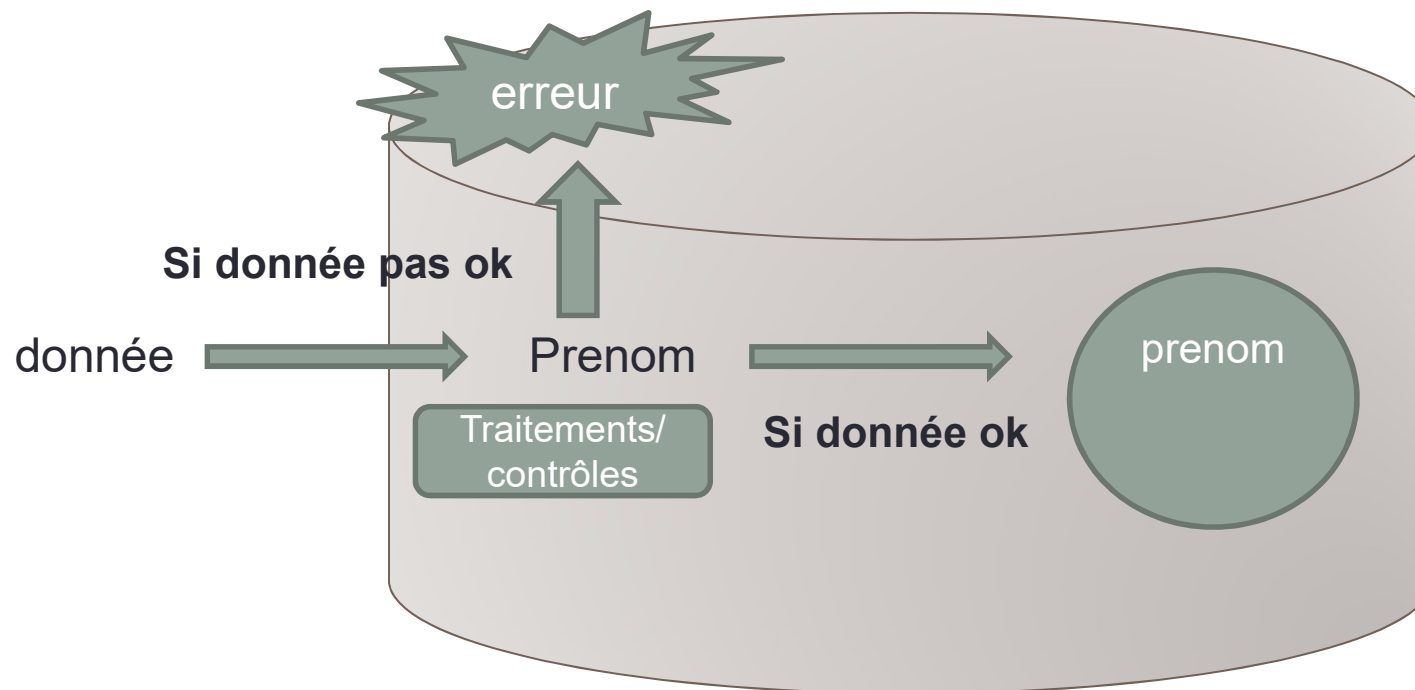
```
class Personne
{
    public string Prenom
    {
        set
        {
            prenom = value;
        }
    }
}
```

value = "aNaé"

# Définir les champs : encapsulation

La propriété fait des contrôles et/ou des traitements avant d'accéder au contenu d'un objet. (comme une capsule)

```
Personne p = new Personne();  
p.Prenom = "aNaé";
```



# Définir les champs : encapsulation

Avec un contrôle, ça a un réel intérêt !

```
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        p.Prenom = "aNaé";
    }
}
```

```
class Personne
{
    public string Prenom
    {
        set
        {
            if (value == null)
                throw new ArgumentNullException("Le prénom ne peut pas être nul");

            if (value.Length == 0)
                throw new ArgumentException("Le prénom ne peut pas être vide");

            prenom = value.Substring(0,1).ToUpper() + value.Substring(1).ToLower();
        }
    }
}
```

La classe vérifie la cohérence de la valeur puis la « met en forme » avant de la mettre dans le champs prenom

# Propriétés qui assurent !

```
class Personne
{
    ...
    public String Prenom
    {
        set
        {
            if(value == null)
                { throw new Argu
            if (value.Length == 0)
                { throw new ArgumentException("Le prenom ne peut pas être une
chaine
```

//classe d'utilisation

class Program

```
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        p.Nom = "";
        // provoquera une exception
    }
}
```

p est protégé !  
Sa classe veille à  
l'intégrité de ses  
données

```
Exception non gérée : System.ArgumentException: Le prénom ne peut pas être une chaine vide
à DemoPersonne.Personne.set_Prenom(String value) dans D:\mesCours\2016-2017\M2103-C#\DemoCours\
DemoPersonne\DemoPersonne\Personne.cs:ligne 27
à DemoPersonne.Program.Main() dans D:\mesCours\2016-2017\M2103-C#\DemoCours\DemoPersonne\Progr
```



DemoPersonne



DemoPersonne a cessé de fonctionner

Un problème est à l'origine du dysfonctionnement du

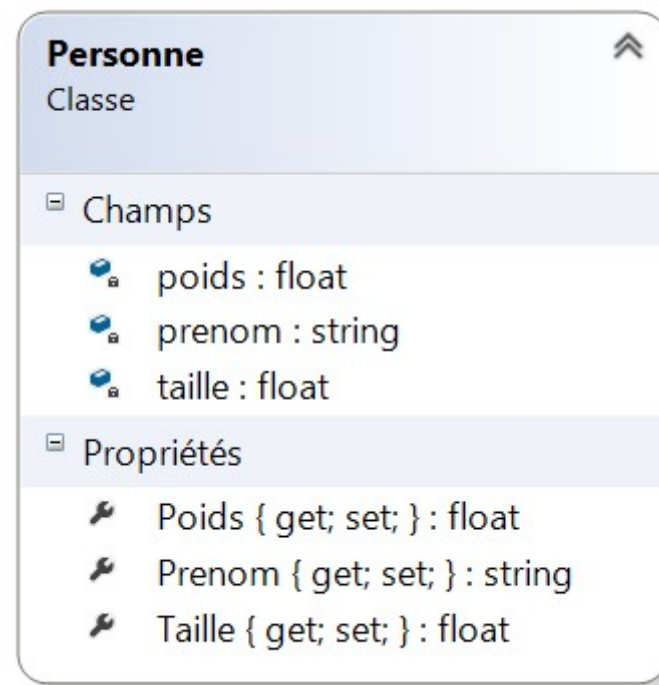
# Une exception ?

- Pour indiquer une erreur d'exécution
- Les plus répandues :

Exception	Cause	Exemple
<a href="#"><u>NullReferenceException</u></a>	Levée par le runtime uniquement si un objet Null est référencé.	String nom = null; nom.ToUpper();
<a href="#"><u>IndexOutOfRangeException</u></a>	Levée par le runtime uniquement en cas de mauvaise indexation du tableau.	tab[arr.Length+1]
<a href="#"><u>ArgumentException</u></a>	Classe de base pour toutes les exceptions d'argument.	Argument incorrect

# Définir les champs : encapsulation

- A chaque champ (variable interne à un objet), peut donc correspondre une propriété :



Poids protège poids  
Prenom protège prenom  
Taille protège taille

# Accès aux champs

Un champ peut être rendu (dans une classe autre que la classe de définition) :

- Inaccessible si on ne définit pas de propriété publique
- Non modifiable si on met le mot clef `private` devant `set`

```
public string Prenom
{
    get
    {
        return prenom;
    }

    private set
    {
    }
```



```
//classe d'utilisation
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        // p.Prenom = « Anaé »; est impossible !
    }
}
```

# Champ (Variable membre) en lecture seule

- Un exemple connu :

String, classe

Propriétés

	Nom	Description
	<code>Chars[Int32]</code>	Obtient l'objet <code>Char</code> à une position de caractère spécifiée dans l'objet String actuel.
	<code>Length</code>	Obtient le nombre de caractères de l'objet String actuel.

Length est une propriété qui donne accès à la longueur de la chaîne, mais ne permet pas de modifier sa valeur



# Propriétés simplifiées

- Il existe une définition simplifiée :

```
public String Libelle { get; set; }
```

A utiliser si le rôle d'une propriété donnée consiste uniquement à lire et à écrire le contenu d'un champ sans effectuer de contrôles particuliers.

# 1<sup>er</sup> objet

- Instance de classe initialisée à l'aide :
  - Du mot clef **new**
  - D'un constructeur : méthode pour initialiser un objet

```
//classe de définition
namespace DemoPersonne
{
    class Personne
    {
        public String prenom;
    }
}
```

```
//classe d'utilisation
namespace DemoPersonne
{
    class Program
    {
        static void Main(string[] args)
        {
            Personne p ;
            p = new Personne();
        }
    }
}
```

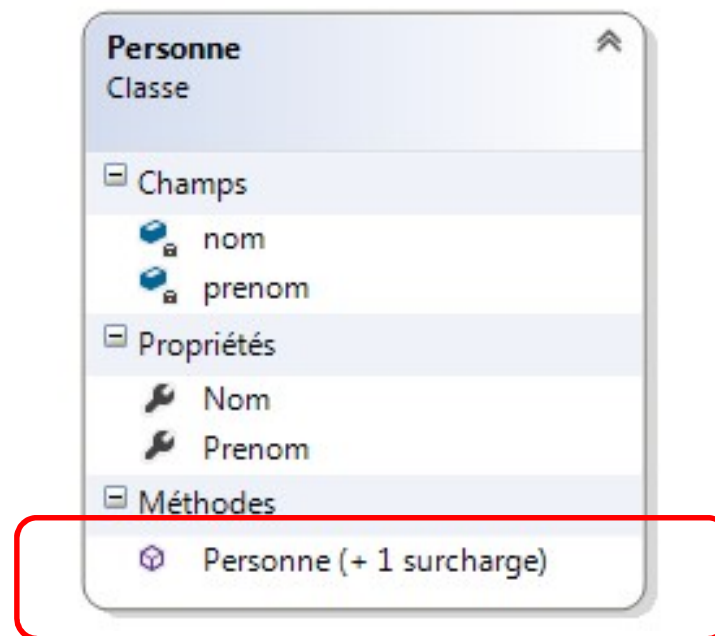
p

nom : null  
prenom: null

Ici, constructeur par défaut car aucun constructeur n'a été encore défini => initialisation avec des valeurs par défaut

# Constructeur

- Un constructeur est une méthode qui :
  - Ne retourne pas de résultat
  - Porte le même nom que la classe
  - Sert à initialiser un objet : initialiser ses différentes variables membres
  - Est bien souvent surchargée



# Constructeur

- Il initialise les variables membres à l'aide des paramètres : Les **paramètres** sont des variables locales au constructeur qui servent à alimenter « durablement » les **variables d'instances**
- Il se sert des propriétés pour s'assurer de l'intégrité des valeurs

```
//Classe de définition
class Personne
{
    private String nom;
    public String Nom {
        set
        { nom = value.ToUpper(); }
        ...
    }
}
```

```
public Personne(String unNom, String unPrenom)
```

```
{
    Nom = unNom;
    Prenom = unPrenom;
}
```

```
}
```

```
//classe d'utilisation
class Program
```

```
{
    static void Main(string[] args)
```

```
{
    Personne p = new Personne("Gruson","Nathalie");
}
```

```
}
```

# Constructeur

- Les paramètres et les variables membres peuvent porter le même nom.
- On différencie ce qui est interne à l'objet avec le mot clef : `this` (= objet en cours)

```
class Personne
{
    private String nom;
    public String Nom {
        set
        { this.nom = value.ToUpper(); }
        ...
    }

    public Personne(String nom, String prenom)
    {
        this.Nom = nom;
        this.Prenom = prenom;
    }
}
```

`Personne p = new Personne("Gruson", "Nathalie");`

Ici, `this` = `p`

# Constructeur par défaut

- Remarque : si l'on ne définit pas de constructeur, un constructeur par défaut (sans paramètre) est mis en place. Ce n'est plus le cas dès lors qu'on en définit un.  
=> Il faut alors définir un constructeur par défaut si on en a besoin

```
//Classe de définition  
class Personne  
{
```

```
    ...  
    public Personne( )  
    {}
```

```
    public Personne(String unNom, String unPrenom)
```

```
    {  
        Nom = unNom;  
        Prenom = unPrenom;  
    }
```

```
}
```

```
//classe d'utilisation  
class Program
```

```
{  
    static void Main(string[] args)
```

```
    {  
        Personne p = new Personne( );  
    }
```

```
}
```

# Constructeur par défaut

- Ce constructeur sans code est équivalent à une affectation des variables internes (sans passer par les propriétés) à l'aide des valeurs par défaut pour chaque type donné !

```
//Classe de définition  
class Personne
```

```
{
```

```
...
```

```
public Personne( )  
{ }
```

```
public Personne(String unNom, String unPrenom)
```

```
{
```

```
    Nom = unNom;
```

```
    Prenom = unPrenom;
```

```
}
```

```
}
```

null pour les objets  
0 pour les numériques  
false pour les booléens  
...

**//revient à écrire**

**// attention : ici c'est les variables privées qui**

**// sont directement affectées, on ne passe pas**

**// par les propriétés**

**public Personne ( )**

**{**

**nom = null ;**

**prenom = null;**

**}**

# Surcharger un constructeur

- Sert à faciliter l'instanciation d'un objet.
- Exemple : on aura
  - Un constructeur complet pour initialiser toutes les variables membres :
    - `Personne ( String unNom , String unPrenom, double uneTaille, double unPoids)`
  - Des constructeurs avec moins de paramètres : pour une initialisation plus rapide : possible si les champs ne sont pas obligatoires !
    - `Personne ( String unNom, string unPrenom )`
    - `Personne ( String unNom )`



# Surcharger un constructeur

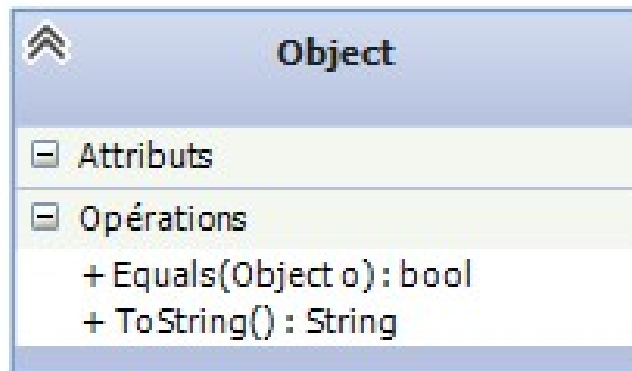
- Définir plusieurs constructeurs, ne veut pas dire recopier du code  
=> **Veillez à factoriser le code !**

```
//Classe de définition
class Personne
{
    ...
    public Personne(String unNom, String unPrenom, double uneTaille, double unPoids)
    {
        Nom = unNom;
        Prenom = unPrenom;
        Taille = uneTaille;
        Poids = unPoids ;
    }
    public Personne(String unNom, String unPrenom ) : this ( unNom, unPrenom, 0, 0 )
    {}
    public Personne(String unNom) : this ( unNom, null, 0, 0 )
    {}
}
```

Fait appel à l'autre constructeur. Evite de réécrire du code !

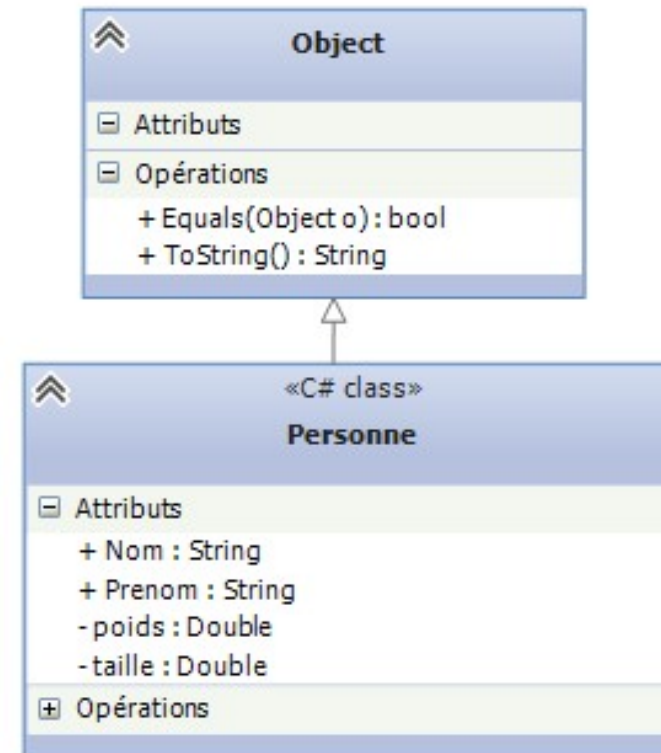
# Les méthodes usuelles, héritées

- Certaines méthodes se retrouvent dans toutes les classes. C'est un modèle qu'il faut reproduire. Il faut donc les définir :
  - ToString() convertit un objet en chaîne de caractère
  - Equals() permet de savoir si 2 objets sont identiques
- Ses méthodes sont présentes dans la classe Object. Pour information, Object est la classe mère de toutes les autres classes par défaut.



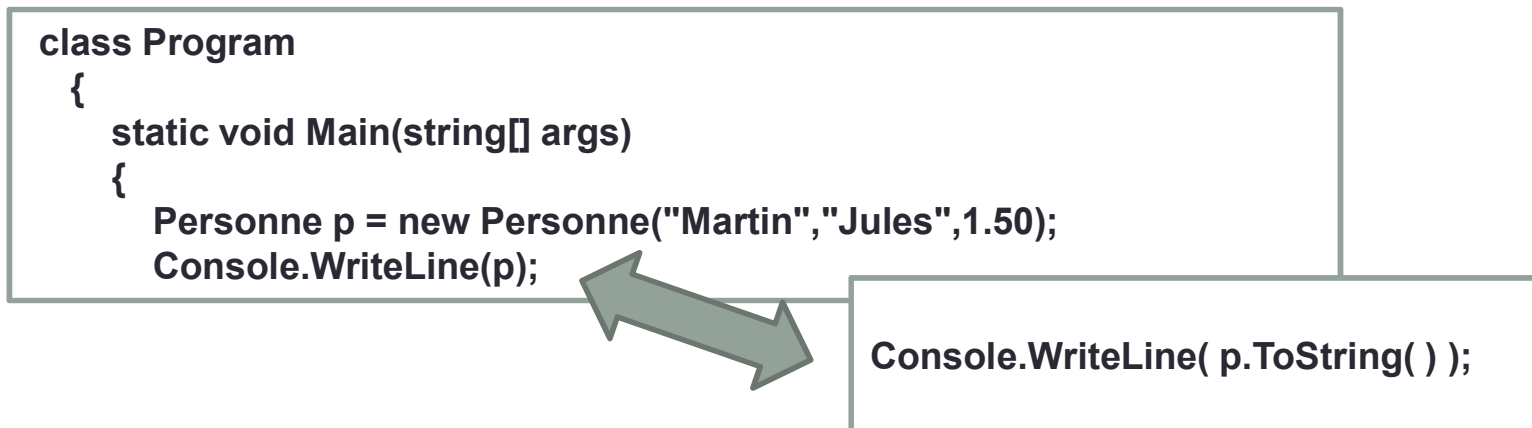
# Classe mère ? Héritage ?

- Toutes les classes héritent au moins de la classe Object
- Ex: si la classe Personne n'a pas de définition de ToString et de Equals, l'objet Personne déclenche la définition de ToString de la classe Object



# La méthode ToString

Elle est nécessaire même si vous ne l'appellez pas explicitement !  
Quand on veut afficher un objet, elle est appelée implicitement.



Si elle n'est pas définie, par défaut, cela affiche la classe (ainsi que le namespace de l'objet) : chaîne retournée par la méthode `ToString` héritée de la classe `Object`.

DemoPersonne.Personne

# La méthode ToString

La méthode ToString retourne les valeurs contenues dans l'objet (de préférence par l'intermédiaire des propriétés) sous forme textuelle :

**override** : indique qu'on redéfinit la méthode ToString héritée

```
//Classe de définition
class Personne
{
    ...
    public override String ToString()
    { return "Nom :" + Nom + "\nPrenom :" + Prenom; }
}
```

```
Personne p = new Personne("Martin","Jules");
Console.WriteLine( p );
```

```
Nom :MARTIN
Prenom :Jules
```

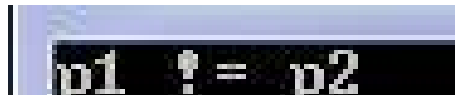
# La méthode Equals

Pourquoi est elle nécessaire ?

Car les objets sont de type référent.

Le `==` teste si les 2 références pointent vers le même objet !

```
Personne p1 = new Personne("Martin","Jules");  
Personne p2 = new Personne("Martin", "Jules");  
if (p1 == p2)  
    Console.WriteLine("p1 == p2 ");  
else  
    Console.WriteLine("p1 != p2 ");
```

A terminal window with a black background and white text. The text displayed is 'p1 != p2', indicating that the equality check failed because two separate objects were created.

```
Personne p1 = new Personne("Martin","Jules");  
Personne p2 = p1;  
if (p1 == p2)  
    Console.WriteLine("p1 == p2 ");  
else  
    Console.WriteLine("p1 != p2 ");
```

A terminal window with a black background and white text. The text displayed is 'p1 == p2', indicating that the equality check succeeded because p2 was assigned the same reference as p1.

# La méthode Equals

Pour comparer 2 objets, il faut donc utiliser la méthode Equals !

```
Personne p1 = new Personne("Martin","Jules");  
Personne p2 = new Personne("Martin", "Jules");  
  
if (p1.Equals( p2))  
    Console.WriteLine("p1 == p2 ");  
else  
    Console.WriteLine("p1 != p2 ");
```

Maintenant, si elle n'est pas redéfinie. C'est la méthode Equals de la classe Object qui est déclenché et cette dernière ne fait que comparer les références !

# La méthode Equals

La méthode Equals compare les valeurs contenues au sein de l'objet puis retourne un booléen indiquant si les objets sont identiques.

Attention, Equals attend un Objet (et non forcément une Personne) : il faut alors tenter de convertir l'objet en Personne !

```
//Classe de définition
class Personne
{
    ...

    public override bool Equals ( Object o )
    {
        if (o == null)
            return false;

        if (o.GetType() != this.GetType())
            return false;

        Personne p = (Personne)o;

        return Prenom == p.Prenom && Taille == p.Taille && Poids == p.Poids
    }
}
```



# Les objets : variables de type référent

Un objet est associé à une référence mémoire.

Pour un objet, l'affectation ou le passage de paramètre est une copie de cette référence mémoire, mais ce n'est pas une copie de l'objet !


```
Personne p1 = new Personne("Martin","Jules");  
Personne p2 = p1;  
p2.Nom = "Marteen";  
Console.WriteLine(p1);
```



```
Nom : MARTEEN  
Prenom : Jules
```

# La méthode Equals ( et GetHashCode )

La compilation risque maintenant d'indiquer un warning.

 1 'DemoPersonne.Personne' se substitue à Object.Equals(object o) mais pas à Object.GetHashCode()

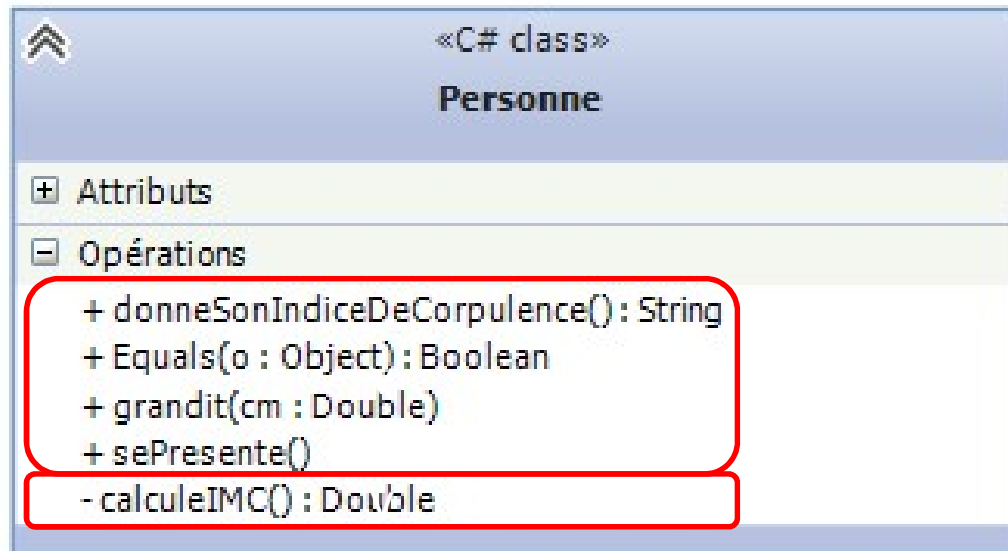
Effectivement, cette méthode est héritée aussi de la classe Object. Elle sert à identifier un objet (clé de hashage) de manière rapide (puisque numérique) pour des classes comme Dictionnary, HashTable, ... qui servent à manipuler des collections de données.

```
//Classe de définition
class Personne
{
    ...
    public override int GetHashCode()
    { int nb = 0 ;
      if ( Nom != null)
        nb += 1* Nom.GetHashCode() ;
      if ( Prenom != null)
        nb += 2* Prenom.GetHashCode();
      return nb;
    }
}
```

# Définir les autres méthodes

Elles peuvent être :

- Publiques : utilisables en dehors de la classe
- Privées : utilisables uniquement au sein de la classe : traitements internes



```
Personne p = new Personne("Martin","Jules");
p.sePresente();


p.calculeIMC();


```

# Définir les autres méthodes

- Elles s'appuient :
  - Sur les variables internes à la classe ou mieux les propriétés utilisables partout au sein de la classe
  - Sur des variables locales (si besoin de stocker des résultats intermédiaires)
  - Sur les paramètres

# Définir les autres méthodes

- Exemple de méthode sans paramètre : elle s'appuie uniquement sur les données contenues au sein de l'objet en passant par ses propriétés !

```
//Classe de définition
class Personne
{
    private String prenom;
    public String Prenom
    {
        get { return prenom; }
        set { prenom = value; }
    }
    ....
    public void sePresente()
    { Console.WriteLine("Bonjour, je m'appelle"
        + Prenom + " " + Nom + "."); }
}
```



```
Personne p = new Personne("Martin","Jules");  
p.sePresente();
```

# Définir les autres méthodes

- Exemple de méthode avec un paramètre :

```
//Classe de définition
class Personne
{
    ....
    public double Taille { get; private set; }
    ...
    public void grandit(double cm)
    {
        this.Taille += cm / 100;
    }
}
```

```
Personne p = new Personne("Martin","Jules", 1.55,40);  
p.grandit(2);
```

# Définir les autres méthodes

- Exemple de méthode privée :

```
//Classe de définition
class Personne
{
    ....
    public double Taille { get; private set; }
    ...
    private double calculeIMC()
    { return Poids / (Taille * Taille); }}
```

## Définir les autres méthodes

- Autre exemple de méthode : ici, elle :
  - utilise une variable locale : pas besoin de déclarer une variable membre pour stocker le résultat d'un calcul.
  - S'appuie sur une autre méthode de la classe

```
public String donneSonIndiceDeCorpulence()  
{  
    double IMC = this.calculerIMC();  
  
    if (IMC < 25)  
        return "normal";  
    else if ( IMC < 30 )  
        return "surpoids";  
    else  
        return "obese";  
}
```

```
Personne p = new Personne("Martin","Jules", 1.55,40);  
Console.WriteLine( p.donneSonIndiceDeCorpulence() );
```



## Une méthode encore bien utile : CompareTo!

- On risque de vouloir manipuler plusieurs Personnes et faire des tris. Or ce type étant nouveau, comment peut se faire le tri ?

```
Personne[ ] amis = new Personne[10];  
amis [0]= new Personne("Martin","Jules", 1.85 );  
amis [1]= new Personne("Martin", "Ana" , 1.60);  
amis[2] = new Personne("Bad", "Marc" , 1,70);  
Array.Sort(amis); // ???
```

- Il faut alors définir une méthode pour rendre les Personnes comparables

```
class Personne : IComparable  
{  
    public int CompareTo(Object o)  
    {  
  
        // retourne valeur >0 si l'objet doit être placé après celui passé en paramètre  
        // retourne valeur < 0 si l'objet doit être placé avant celui passé en paramètre  
        // retourne 0 si l'objet doit être placé à la même position que celui passé en  
paramètre  
    }  
}
```

IComparable est une interface !

# Une méthode encore bien utile : CompareTo!

- Exemple : tri par taille par ordre croissant

```
class Personne : IComparable
{
    public int CompareTo(Object o)
    {
        Personne p = o as Personne ;
        return Taille.CompareTo(p.Taille);
    }
}
```

Si la Taille de l'objet > taille  
du param, le retour est + .  
L'objet sera placé après  
l'objet passé en paramètre !

```
Personne[ ] amis = new Personne[10];
amis [0]= new Personne("Martin","Jules",1.85);
amis [1]= new Personne("Martin", "Ana",1.60);
amis[2] = new Personne("Bad", "Marc",1.70);
Array.Sort(amis);
```

Appel implicite à  
CompareTo

```
Nom :MARTIN
Prenom :Ana
Taille :1,6
```

```
Nom :BAD
Prenom :Marc
Taille :1,7
```

```
Nom :MARTIN
Prenom :Jules
Taille :1,85
```

# Surcharger les opérateurs

- On peut redéfinir les opérateurs de base : == != < > <= >=
- Obligation de les définir par paire

```
public static bool operator <(Personne p1, Personne p2)
{
    return (p1.Taille < p2.Taille);
}
public static bool operator >(Personne p1, Personne p2)
{
    return (p1.Taille > p2.Taille);
}
```

# Commenter son code

- Commenter ses classes est indispensable afin de rendre son code pérenne.
- Un mécanisme de « tag » permet de générer une documentation au format XML.
- Certains logiciels (Doxygen, SandCastle par exemple) permettent de générer une documentation au format HTML.
- Liste des « tag » les plus utiles :
  - **<summary>** correspond à la description brève d'une classe, champ, méthode, namespace...
  - **<remarks>** correspond à la description détaillée d'une classe, champ, méthode, namespace...
  - **<param name="monParam">** sert à documenter le paramètre d'entrée d'une méthode
  - **<returns>** permet de documenter la valeur de retour
- Liste complète des « tag » :  
<https://www.stack.nl/~dimitri/doxygen/manual/xmlcmds.html>

# Commenter son code

- VisualStudio vous permet de générer automatiquement un squelette de « tag » :
- Tapez **///** devant chaque élément à commenter :

```
/// <summary>  
/// obtient la taille de la personne exprimée en mètre  
/// </summary>
```

```
public double Taille { get; private set; }
```

# Pour générer une doc HTML

- La documentation au format HTML est générée :

The screenshot shows a web interface for 'DemoPersonne' documentation. At the top, there's a navigation bar with tabs: 'Page principale', 'Espaces de nommage', and 'Classes' (which is selected). Below this is a sub-navigation bar with 'Liste des classes' (selected), 'Index des classes', 'Hiérarchie des classes', and 'Membres de classe'. A search bar labeled 'Recherche' is also present. The main content area is titled 'Liste des classes' and contains the text 'Liste des classes, structures, unions et interfaces avec une brève description :'. Below this, there's a table listing classes: 'DemoPersonne' (expanded), 'Personne', and 'Program'. At the bottom, it says 'Généré le Jeudi 29 Janvier 2015 21:22:13 pour DemoPersonne par doxygen 1.8.9.1'.

## Propriétés

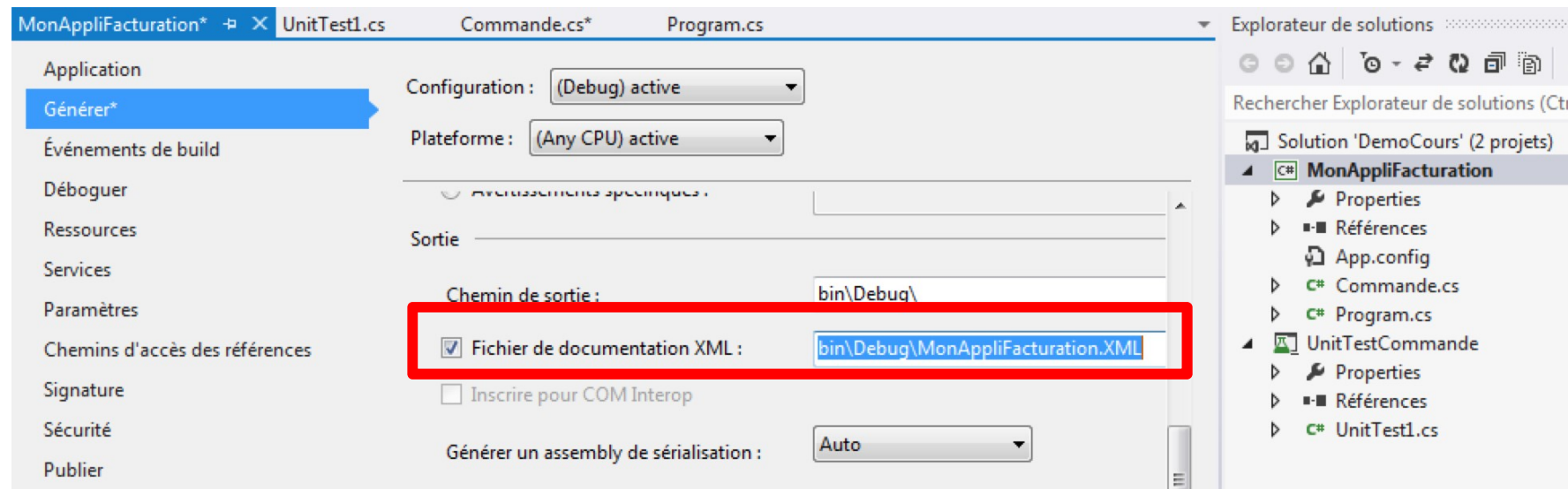
double **Taille** [get]

obtient la taille de la personne exprimée en mètre [Plus de détails...](#)

double **Poids** [get]

# Exporter en XML

- Pour exporter la documentation en HTML, il faut d'abord configurer l'exportation en XML dans les propriétés du projet :

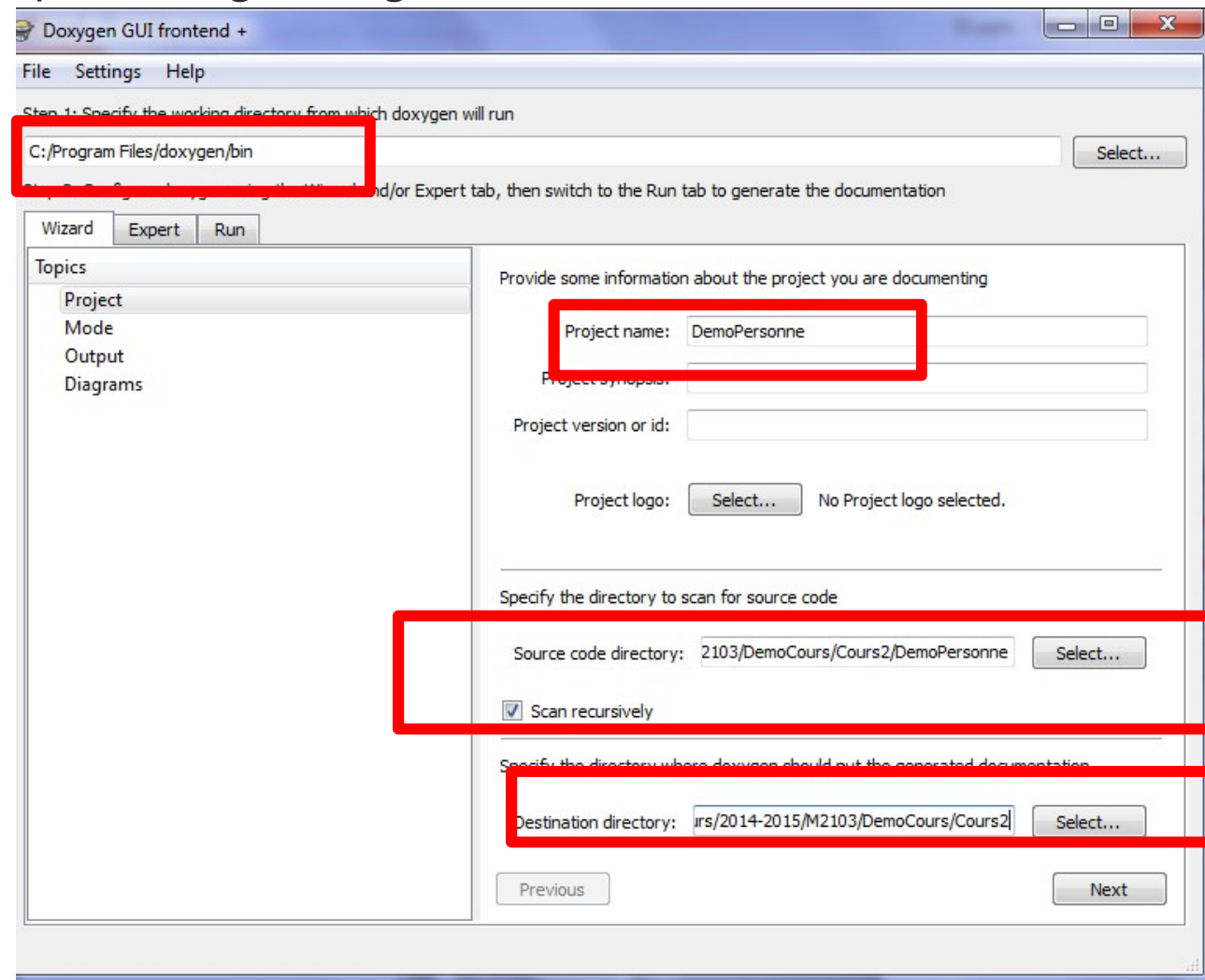


- Le fichier doit être généré :



# Exporter en HTML

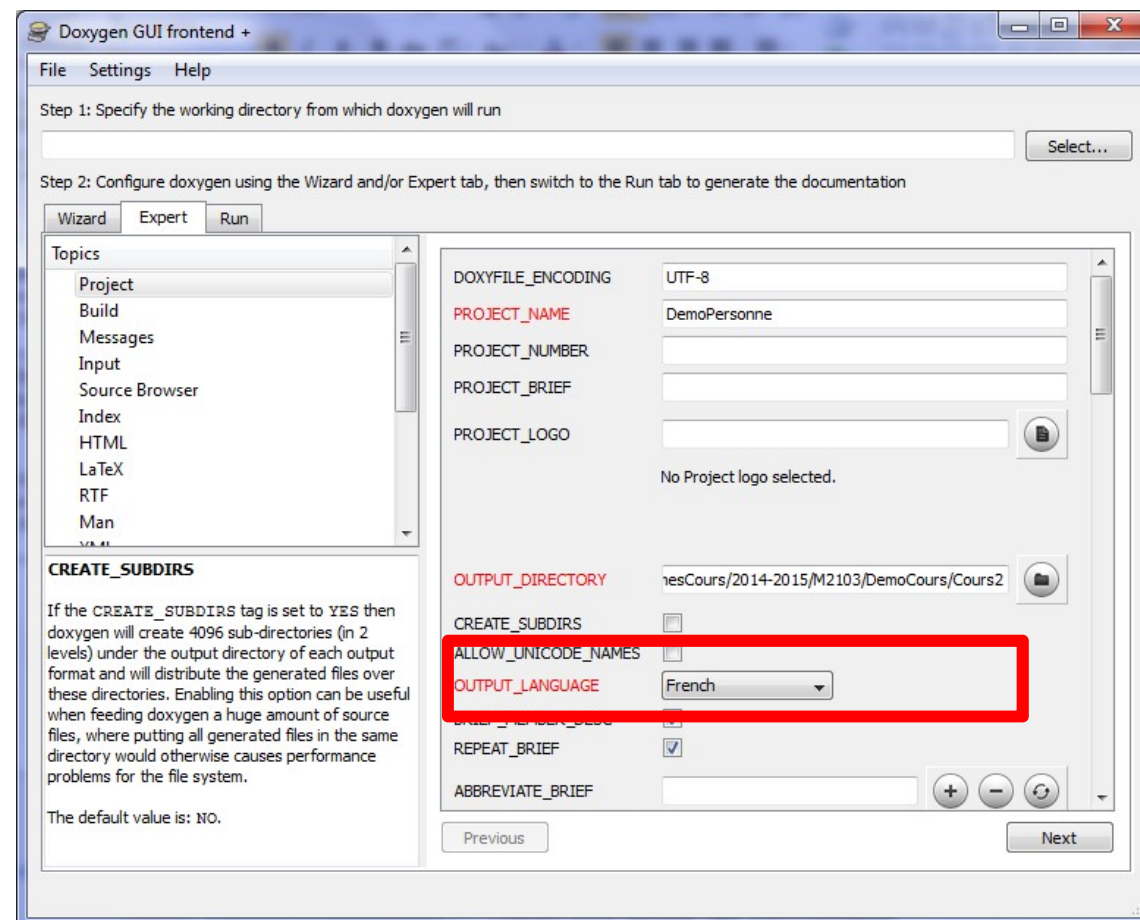
- Il faut alors utiliser un outil externe comme **Doxygen** s'appuyant sur le fichier XML généré.
- Utiliser **DoxyWizard** pour configurer la génération de documentation :





# Exporter en HTML

- Il faut alors utiliser un outil externe comme **Doxygen** s'appuyant sur le fichier XML généré.
- Utiliser **DoxyWizard** pour configurer la génération de documentation :



# Exporter en HTML

- La documentation au format HTML est générée :

