

POO

Définir ses propres classes

R1.01 – Algorithmie et Programmation en C#

Pourquoi ?

- Pour regrouper des informations de différentes natures concernant le même sujet
 - Ex: stocker toutes les informations concernant une personne
- Pour limiter le nombre de variables : on pourra donc stocker plusieurs données dans une seule variable.
 - *Ex : stocker nom, prenom, poids, taille dans une seule variable*
- Pour généraliser les traitements sur ces variables de même nature.

Pourquoi faire une classe ?

Au lieu d'avoir ce code, au sein du main :

```
static void Main(string[ ] args)
{
    String nom ="Gruson";
    String prenom = "Nathalie";
    double taille =1.65;
    double poids = 55;
    Console.WriteLine("\nNom : " + nom + "\nPrenom : " + prenom
        + "\nTaille : " + taille + " m"
        + "\nPoids : " + poids + " k");
}
```

données
indépendantes

On pourrait avoir, à condition de définir une classe Personne :

```
static void Main(string[ ] args)
{
    Personne p = new Personne("Gruson","Nathalie",1.65,55);
    Console.WriteLine(p);
}
```

données regroupées dans une variable
d'un nouveau type : Personne

Vocabulaire

- Une classe, c'est un type
- Un objet, c'est une variable, une instance de classe.
- Instancier, c'est initialiser un objet

Conventions de nommage

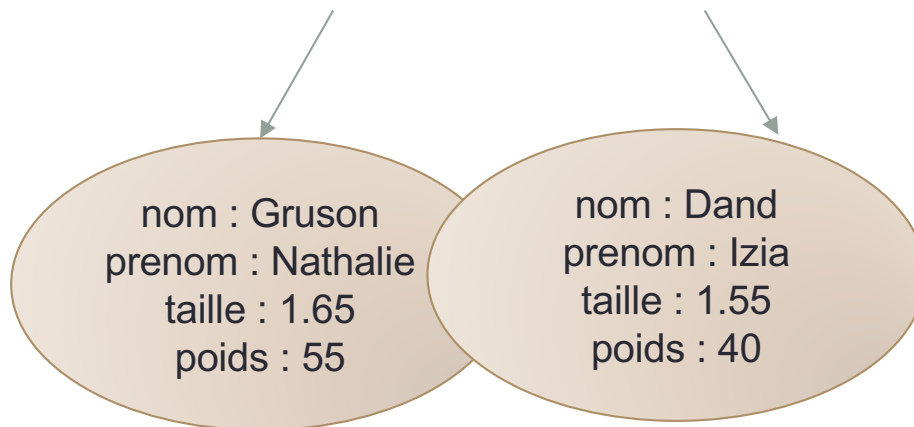
- Une classe (ou structure) commence par une majuscule.
Ex : Personne
- Une objet commence par une minuscule.
Ex : unePersonne

Classe et objets (instances de classe)

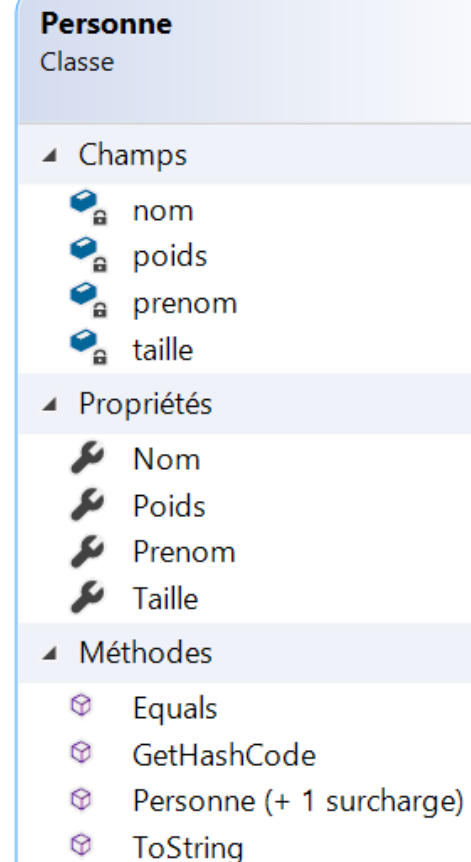
Une classe donnera naissance à plusieurs objets.

p1 et p2 sont des objets (= variables) de classe Personne

```
Personne p1 = new Personne("Gruson", "Nathalie", 1.65, 55);  
Personne p2 = new Personne("Dand", "Izia", 1.55, 40);
```



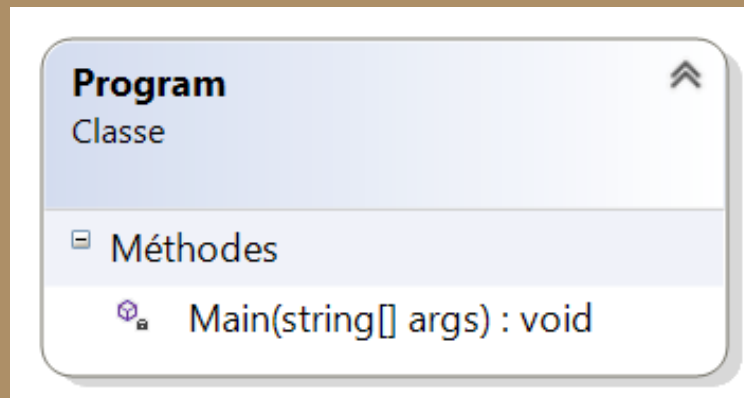
Personne est une classe (= type)



Définition/utilisation

On a alors 2 classes : 2 fichiers

La classe Program : main utilisant la classe Personne à travers des objets.



La classe Personne : contenant la définition du type Personne



```
Personne p = new Personne("Gruson","Nathalie",1.65,55);
```

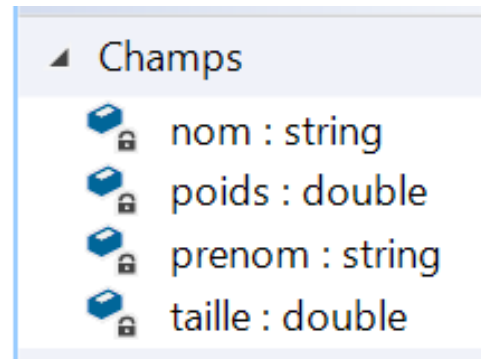
Pourquoi faire une classe ?

- Pour structurer, organiser les données d'un programme :
⇒ **de nouveaux types !**
- Pour vérifier les données :
⇒ **des types sans faille !**
- Pour définir des traitements spécifiques et réutilisables :
⇒ **des types riches en opérateurs et traitements !**

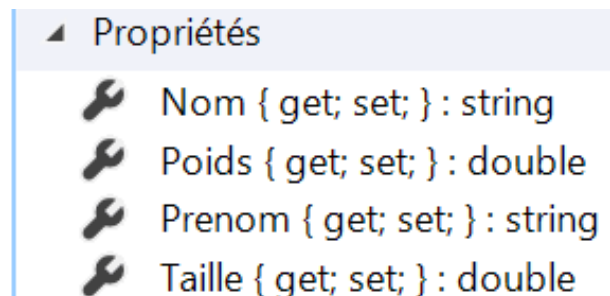
Définir une classe

C'est créer un nouveau fichier pour définir :

- **Des champs généralement privés (inaccessibles en dehors de la classe).**



- **Des propriétés publiques : pour accéder aux champs privés. Elles portent le même nom que le champ à la majuscule près.**



get/set ⇔ peut récupérer et modifier la valeur du champ

Définir une classe

- **Des constructeurs : méthodes pour initialiser les champs.**

▲ Méthodes

- ❏ `Personne(string nom, string prenom)`
- ❏ `Personne(string nom, string prenom, double taille, double poids)`

- **Des méthodes usuelles**

▲ Méthodes

- ❏ `Equals(object obj) : bool`
- ❏ `GetHashCode() : int`
- ❏ `ToString() : string`

Définir une classe

- **Des méthodes spécifiques**

- ▲ Méthodes

- 📦 CalculeImc() : double
 - 📦 EstPlusGrande(Personne autre) : bool

- **Des surcharges d'opérateurs.**

- **Ex: +, <, >, ...**

- ▲ Méthodes

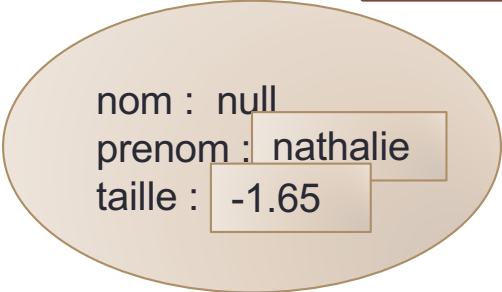
- 📦 operator <
 - 📦 operator <=
 - 📦 operator >
 - 📦 operator >=

Classe minimale

- Constructeur par défaut mis en place
- Champs publics = DANGER. La classe ne fait pas de vérification.

```
public class Personne
{
    public String prenom;
    public String nom;
    public double taille;
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        p.prenom = "nathalie";
        p.taille = -1.65 ;
    }
}
```



nom : null
prenom : nathalie
taille : -1.65

Principe d'encapsulation

- Rendre les **champs privés** (non accessibles en dehors de la classe)
- Créer des intermédiaires publics : les **propriétés**



```
public class Personne
{
    private String prenom;

    public String Prenom
    {
        get
        {
            return this.prenom;
        }
        set
        {
            this.prenom = value;
        }
    }
}
```

this = objet en cours d'utilisation (ici p)

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

nom :
prenom : nathalie
taille :
poids :

Principe d'encapsulation avec vérification

Pour protéger le champ, au sein du bloc set, on vérifie la valeur (value), si elle valide (compatible), on l'affecte au champ, sinon on lance un exception

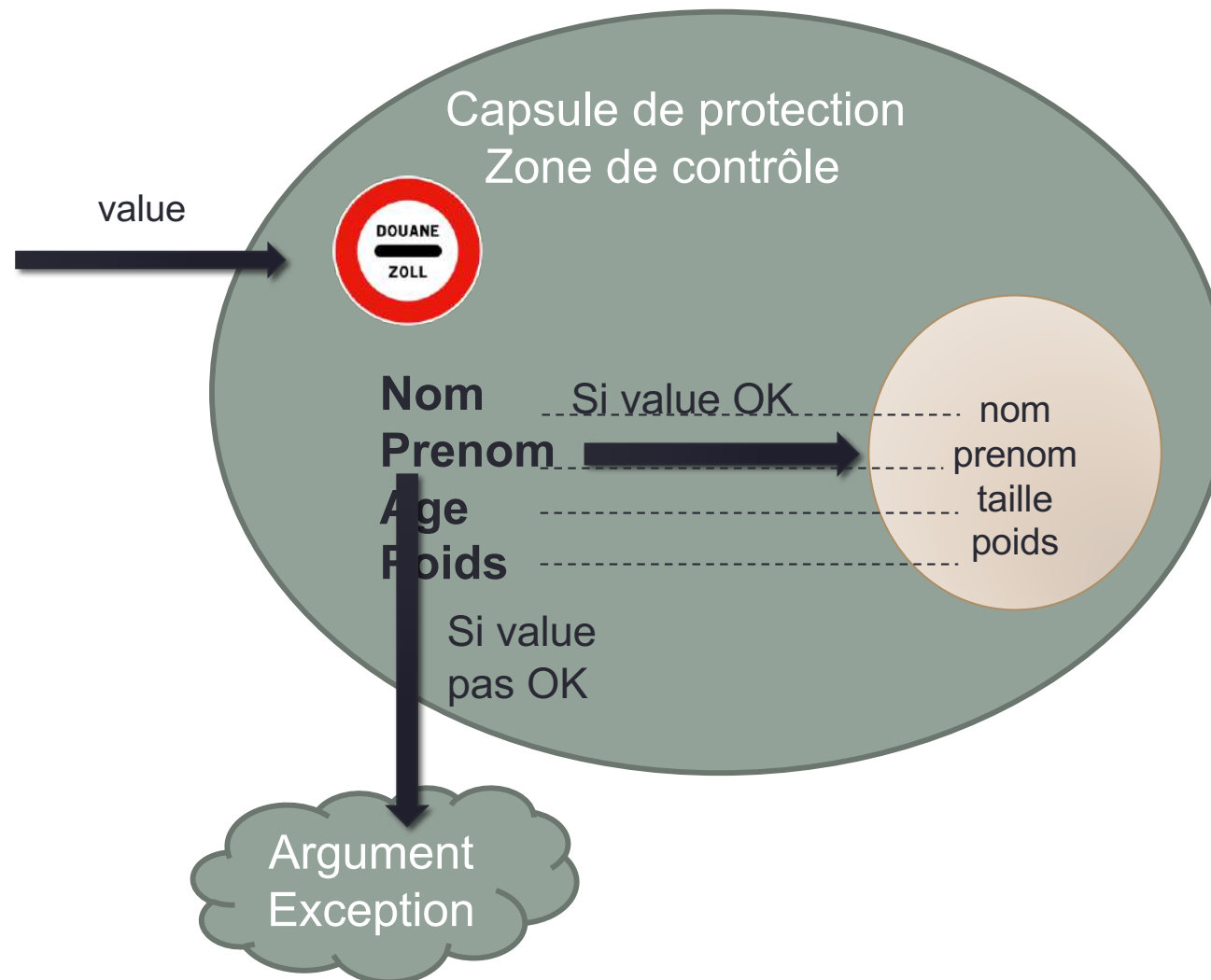
```
public class Personne
{
    private String prenom;
    public string Prenom
    {
        set
        {
            if(String.IsNullOrEmpty(value)
                { throw new ArgumentException("Le prenom doit être renseigné"); }
            this.prenom = value.Substring(0,1).ToUpper() + value.Substring(1).ToLower() ;
        }
    }
}
```

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

Ici, la propriété Prenom s'assure que la valeur n'est pas null ou vide
Puis la stocke dans le champ prenom après avoir fait une mise en forme.

Principe d'encapsulation avec vérification

encapsulation = surcouche (capsule de protection) avant d'arriver au cœur de l'objet.



Une exception ?

- Pour indiquer une erreur d'exécution : une mauvaise utilisation
- A destination des programmeurs (et non des utilisateurs finaux)

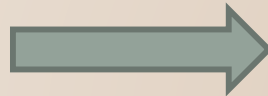
Exception	A lancer (à lever)
ArgumentNullException	Si la méthode reçoit un paramètre null alors qu'il ne devrait pas l'être.
ArgumentOutOfRangeException	Si la méthode reçoit un paramètre qui n'a pas une valeur compatible avec l'éventail des valeurs attendues.
<u>ArgumentException</u>	Si la méthode a un problème avec un paramètre qui ne correspond à aucun cas des problèmes précédents.

Propriété - résumé

- Nécessaire pour atteindre les champs d'un objet
- Get/Set déclenchée de manière automatique

Ex : `Personne p = new Personne() ;`

`p.Nom = "Gruson" ;`



set

`Console.WriteLine (p.Nom)`



get

```
public String Nom
{
    set { // filtre la valeur devant être mise dans le champs }
    get { // retourne la valeur contenue dans le champs
        return this.nom ;}
}
```



Propriété – set (squelette par défaut)

Accesseur set : code déclenché lors d'une affectation d'une propriété
On fait une mise à jour du champ.

```
public class Personne
{
    private String prenom;

    public string Prenom
    {
        // ...

        set
        {
            this.prenom = value;
        }
    }
}
```



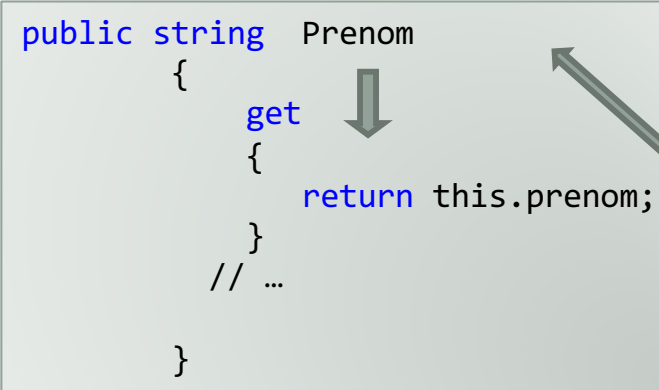
```
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne();
        p.Prenom = "nathalie";
    }
}
```

Propriété – get (squelette par défaut)

Accesseur get : code déclenché lors de l'utilisation d'une propriété
On accède en lecture au champ.

```
public class Personne
{
    private String prenom;

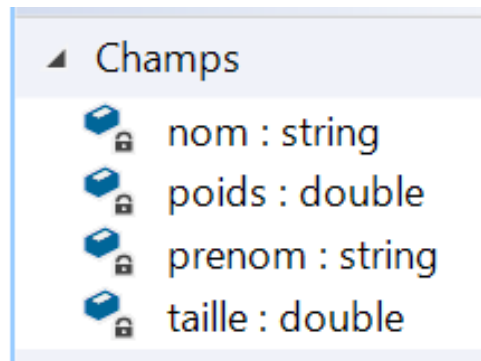
    public string Prenom
    {
        get
        {
            return this.prenom;
        }
        // ...
    }
}
```



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

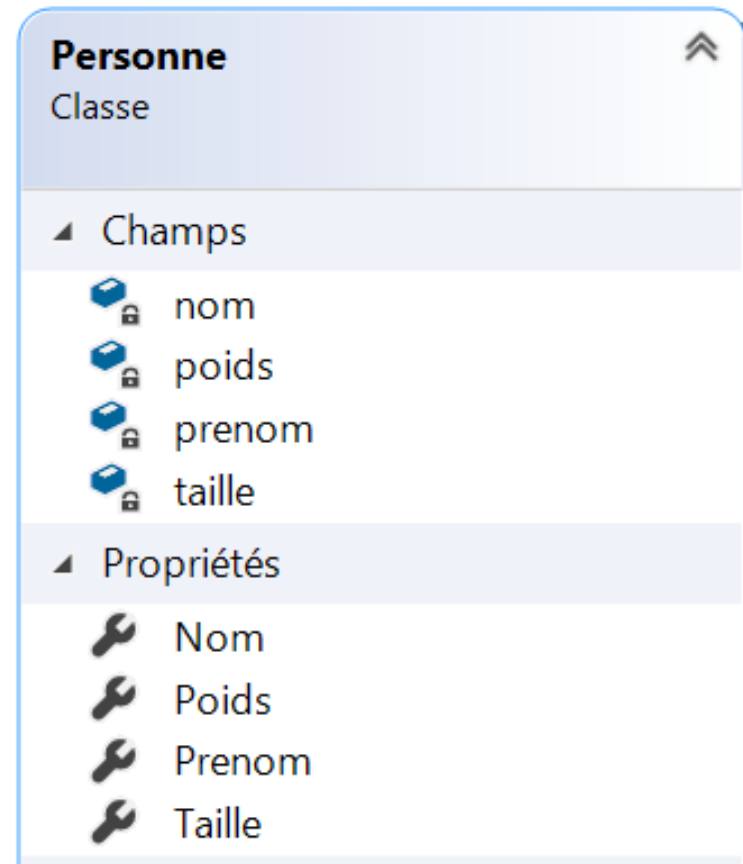
Un champ

- Un champ est :
 - une variable interne à l'objet
 - Une variable globale à toute la classe : Utilisable dans toutes les méthodes : cependant, on préférera utiliser la propriété !



Définir les champs : encapsulation

- A chaque champ, peut/doit donc correspondre une propriété :



Nom protège nom
Prenom protège prenom
Poids protège poids
Taille protège taille

Automatiser l'encapsulation




Génération automatique du squelette de la propriété :

- Placez vous sur le champ ou les champs à encapsuler
- Ctrl + R puis ctrl +E !

```
namespace Projet1
{
    0 références
    class Personne
    {
        private String nom;
    }
}
```

Aperçu des modifications - Encapsuler le champ

Encapsuler le champ :

- ☒  Projet1.Personne.nom
- ☒  Personne.cs
- ☒  public string Nom ... }

Vous avez aussi l'ampoule
ou l'instruction propfull

Aperçu des modifications du code :

```
7 namespace Projet1
8 {
9     class Personne
10    {
11        private String nom;
12
13        public string Nom
14        {
15            get
16            {
17                return this.nom
```

Aucun problème détecté

Automatiser l'encapsulation

- Cherchez « style de code » dans le champs de recherche:
 - Sélectionnez «C# » puis «Préferer this » pour tous les sous items « Preferences this »

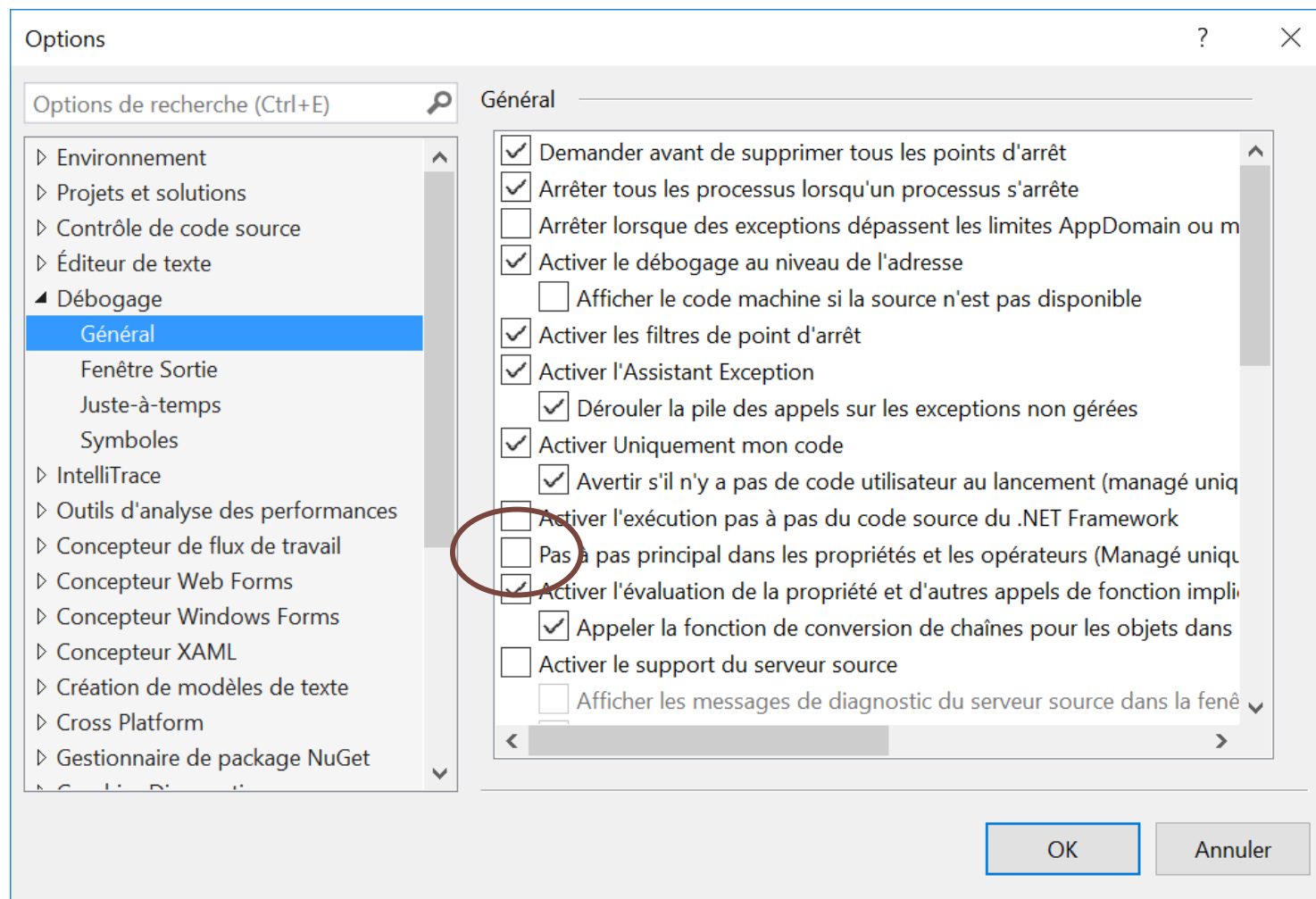
Description	Préférence	Gravité
'Préférences 'this.' :		
Qualifier l'accès au champ avec 'this'	Préferer 'this.'	<input type="radio"/> Refactoriser
Qualifier l'accès à la propriété avec 'this'	Préferer 'this.'	<input type="radio"/> Refactoriser
Qualifier l'accès à la méthode avec 'this'	Préferer 'this.'	<input type="radio"/> Refactoriser

- Indiquez « Jamais » pour tous les sous items «utiliser un corps d'expression »

Description	Préférence	Gravité
Utiliser un corps d'expression pour les méthodes	Jamais	<input type="radio"/> Refactoriser
Utiliser un corps d'expression pour les constructeurs	Jamais	<input type="radio"/> Refactoriser
Utiliser un corps d'expression pour les opérateurs	Jamais	<input type="radio"/> Refactoriser

Paramétrez le debug pas à pas des Propriétés

- **Menu : Déboguer, Options et paramètres /Général**



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 champs
 - Est bien souvent surchargée

▲ Méthodes

- ❏ `Personne(string nom, string prenom)`
- ❏ `Personne(string nom, string prenom, double taille, double poids)`

Si vous ne définissez pas de constructeurs, un constructeur par défaut est mis en place, celui-ci attribue des valeurs par défaut à chacun de vos champs

Constructeur

- initialise les champs à l'aide des paramètres.
- se sert des propriétés pour s'assurer de l'intégrité des valeurs

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

```
public Personne(String unNom, String unPrenom)
{
    this.Nom = unNom;
    this.Prenom = unPrenom;
}
```

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

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

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

Nom
Prenom
Age

nom : Gruson
prenom : Nathalie
....

Constructeur

- Les paramètres et les champs portent souvent le même nom.
- Attention : un paramètre n'existe que le temps de l'exécution de la méthode
- On différencie les variables qui sont internes à l'objet de celles qui sont volatiles et externes (locales à des méthodes) avec le mot clef : this

```
public class Personne
```

```
{
```

```
    private String nom;
```

```
    private String prenom;
```

```
    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");

nom et prenom durent aussi longtemps que l'objet

nom et prenom sont locales et temporaires, elles sont détruites à la fin de construction

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  
public class Personne  
{
```

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

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

Instruction ctor
Ou l'ampoule

Nom et prenom seront
null, taille et poids à 0.

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

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

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

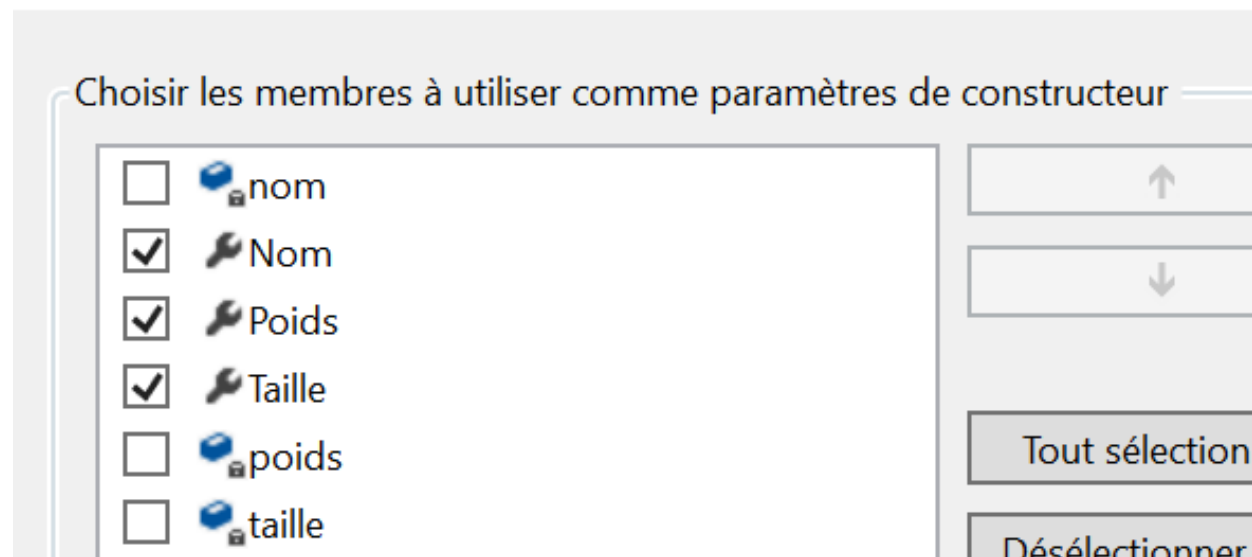
```
}
```

Générer un constructeur automatiquement

- Placez vous sur la ligne de déclaration de votre classe puis avec l'ampoule, choisissez l'option générer un constructeur.
- Choisissez les propriétés (qui vont initialiser les champs)

Choisir les membres

?



Notion de surcharge

- Un constructeur peut être surchargé = redéfini avec des signatures différentes
- Utile pour avoir des paramètres optionnels ou de types différents


Ex : ici, on rend la taille et le poids optionnel avec la 2eme définition

- `public Personne (String nom, String prenom, float poids, float taille)`
- `public Personne (String nom, String prenom)`

Surcharger un constructeur

- Définir plusieurs constructeurs, ne veut pas dire forcément recopier du code => **On peut factoriser le code !**
- **Définissez en 1er celui qui a la signature avec les paramètres les plus proches (en nombre et en type) des champs puis déclinez.**

```
//Classe de définition
public class Personne
{
    ...
    public Personne(String nom, String prenom, , double taille, double poids) {
        this.Nom = nom;
        this.Prenom = prenom;
        this.Taille = taille ;
        this.Poids = poids ;
    }
    public Personne(String nom , String prenom) : this (nom, prenom, 0, 0 )
    {}
}
```



Champ constant ou static

- Un champ static : c'est un champ qui :

- appartient à la classe
- est partagé par tous les objets
- est initialisé dès sa déclaration

```
private static String unite_taille = "m";  
private static String unite_poids = "k";
```

```
public static String Unite_taille  
{ get  
  { return Personne.unite_taille; }  
  set  
    { Personne.unite_taille = value; }  
}
```

- Une constante est :

- avant tout « static »
- Souvent « public » puisque non modifiable

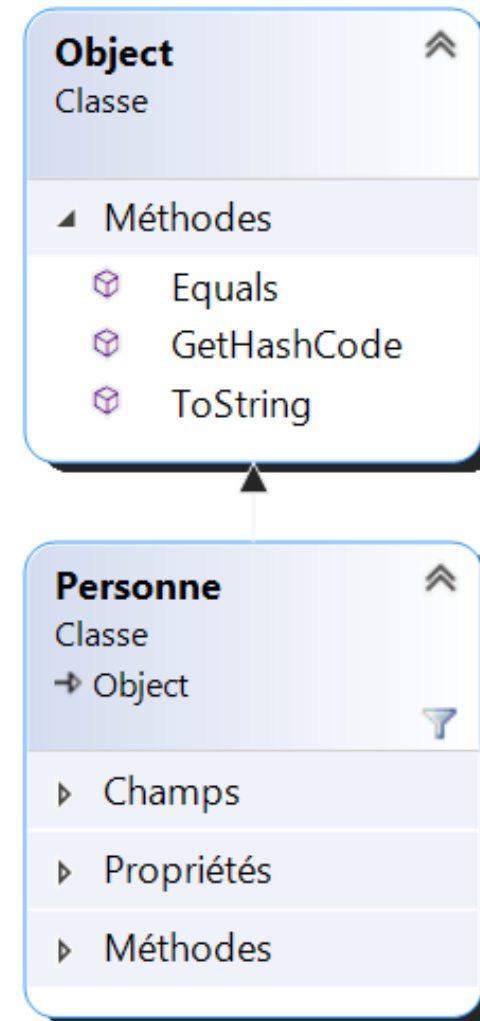
```
public const String UNITE_TAILLE = "m";  
public const String UNITE_POIDS = "k";
```

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
 - GetHashCode() - retourne un numérique servant à identifier rapidement l'objet.
- Ces 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 de classe Personne déclenche la définition de ToString et Equals de la classe Object



Substituer la méthode ToString

Elle est nécessaire même si vous ne l'appellez pas explicitement.

Elle est **appelé implicitement lors d'une concaténation ou d'un affichage**.

```
class Program
{
    static void Main(string[] args)
    {
        Personne p = new Personne("Martin","Jules",1.50,45);
        Console.WriteLine(p);
    }
}
```



```
Console.WriteLine( p.ToString( ) );
```

Si elle n'est pas définie, c'est celle d'Object qui est déclenchée et retourne un résumé
« Namespace.Classe »

Demo1.Personne

Substituer La méthode ToString

La méthode ToString doit retourner les valeurs contenues dans l'objet (de préférence par l'intermédiaire des propriétés) sous forme d'un String :

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

override : indique qu'on redéfinit la méthode ToString héritée

```
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.

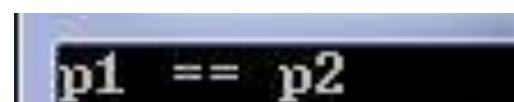
Le `==` teste si les 2 références pointent vers le même objet !

```
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 screenshot of a console window with a black background and white text. The text displayed is "p1 != p2".

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



A screenshot of a console window with a black background and white text. The text displayed is "p1 == p2".

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
```

Substituer 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 !

Substituer la méthode Equals

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

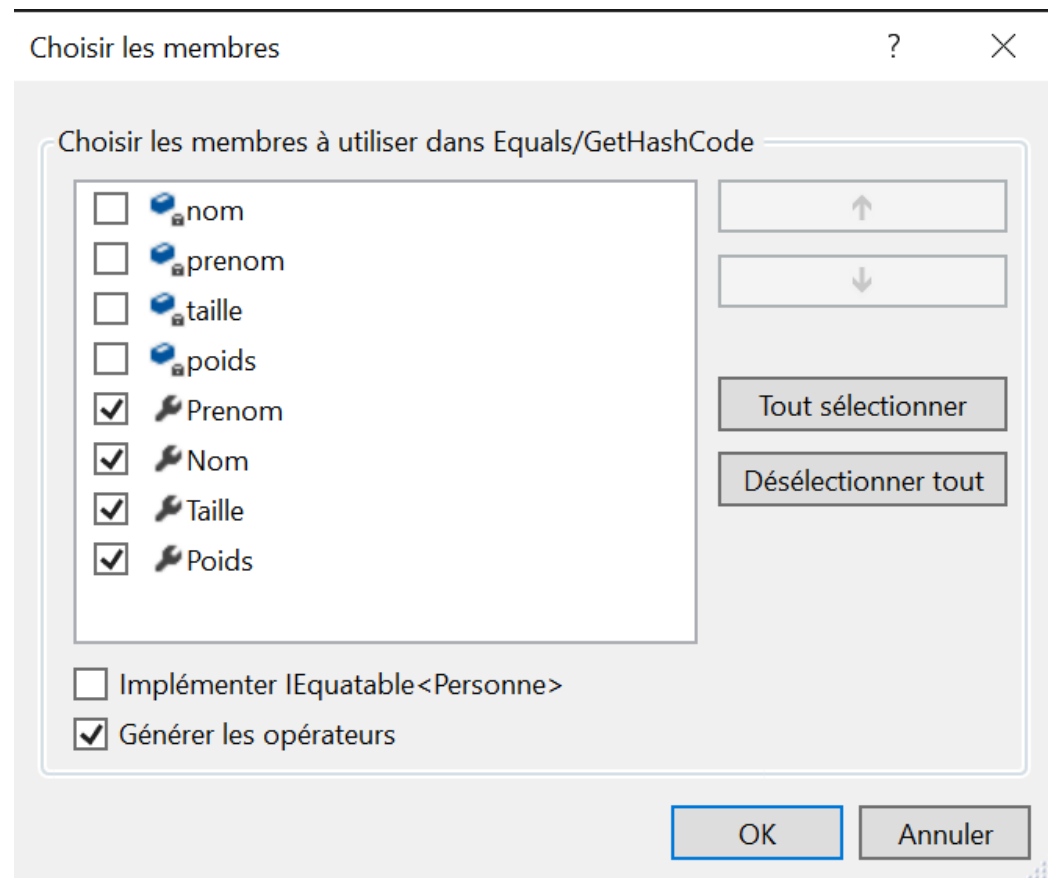
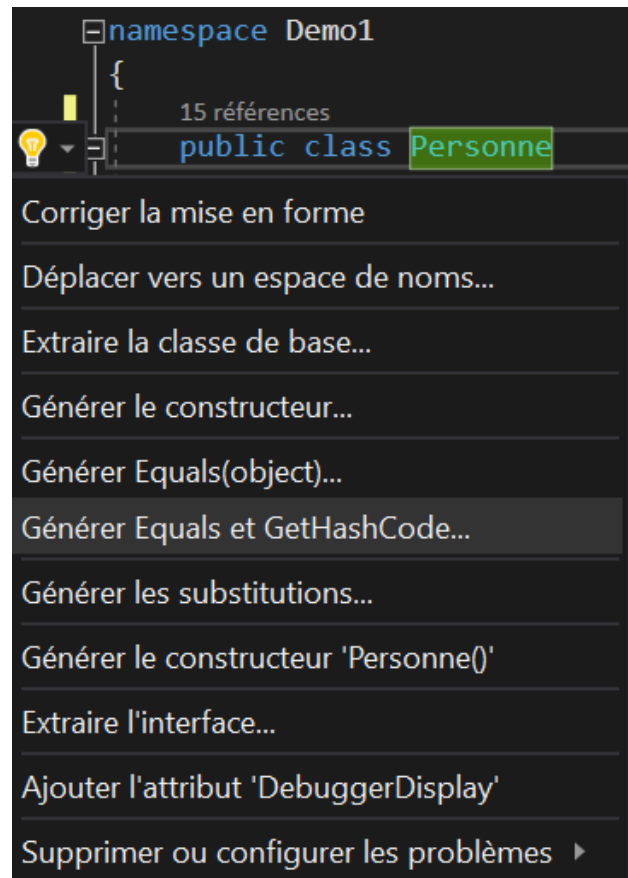
```
public override bool Equals(object obj)
{
    return obj is Personne personne &&
           this.Prenom == personne.Prenom &&
           this.Nom == personne.Nom;
}
```

```
Personne p1 = new Personne("Martin","Jules");
Personne p2 = new Personne("Martin", "Jules");

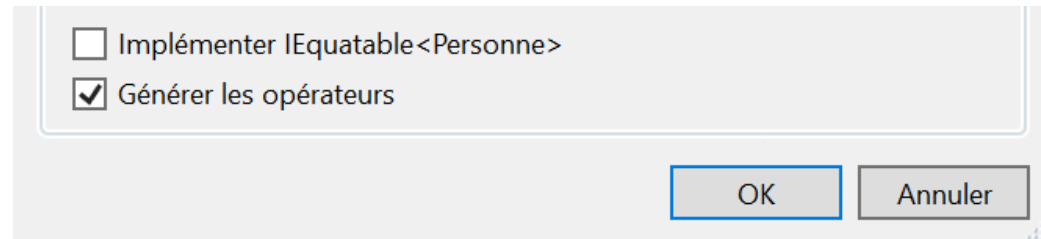
if (p1.Equals(p2))
....
```

Ce code peut être généré grâce à l'ampoule

Générer les substitutions



Générer les opérateurs



☐ Implémenter IEquatable<Personne>
☒ Générer les opérateurs

OK Annuler

```
public static bool operator ==(Personne left, Personne right)
{
    return EqualityComparer<Personne>.Default.Equals(left, right);
}

public static bool operator !=(Personne left, Personne right)
{
    return !(left == right);
}
```

S'appuient sur le Equals précédemment défini

```
Personne p1 = new Personne("Martin","Jules");
Personne p2 = new Personne("Martin", "Jules");
```

```
if (p1 == p2)
```

```
....
```

Surcharger les opérateurs

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

```
public static bool operator <(Personne left, Personne right)
{   return (left.Taille < right.Taille); }

public static bool operator >(Personne left, Personne right)
{   return (left.Taille > right.Taille); }
```

```
Personne p1 = new Personne("Julie", 45, 1.56,55);
Personne p2 = new Personne("Jean", 45, 1.86,85);
If ( p1 > p2 )
....
```

left = p1
right = p2

Définir les autres méthodes

Elles peuvent être :

- « public » : utilisables en dehors de la classe
- « private » : utilisables uniquement au sein de la classe : traitements internes

▲ Méthodes

- 📦 CalculeImc() : double
- 📦 EstPlusGrande(Personne autre) : bool

Définir les autres méthodes

Elles s'appuient :

- Sur les propriétés (c'est-à-dire sur les champs) qui sont 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 :

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

```
{
```

```
....
```

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

```
...
```

```
public double CalculeImc()  
{ return this.Poids / (this.Taille * this.Taille); }
```

```
}
```

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

Pas besoin de paramètre ici. Toutes les données nécessaires sont contenues dans l'objet

Définir les autres méthodes

- Autre exemple de méthode :

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

Imc est une variable locale qui stocke le résultat ramené par CalculeImc

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

Approfondissement propriétés

Les propriétés sont :

- souvent en lecture-écriture : elles ont un accesseur get et set
- parfois en lecture seule: elles ont un accesseur get, mais pas d'accessor set : ça peut être des données calculées sans champ privé associé. Ex:

```
public double Imc
{
    get { return Math.Round(this.Poids / (this.Taille * this.Taille),1); }
}
```

- très rarement en écriture seule (elles ont un accesseur set, mais pas d'accessor get).

Les propriétés simples qui ne nécessitent pas de code d'accessor personnalisé peuvent être implémentées en tant que propriétés implémentées automatiquement, sans champ privé

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