

Méthodes

R1.01 – Initiation au développement

N. Gruson

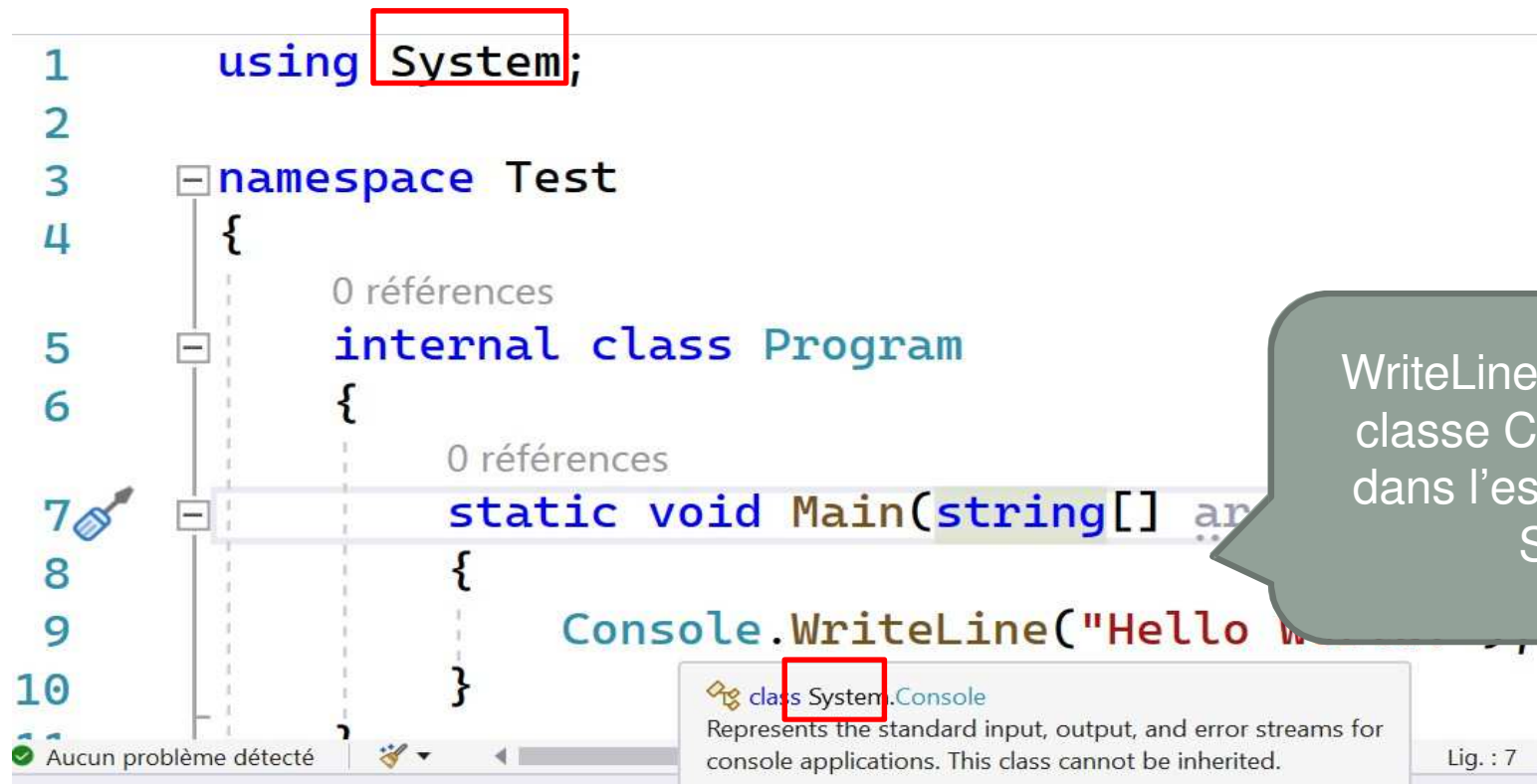
UTILISER DES METHODES

Using namespace

Toutes les méthodes sont rangées dans des classes (fichiers) rangées elles-mêmes dans des espaces de noms (namespace = projet = répertoire).

Pour les utiliser, il faut indiquer au début du fichier « program.cs » un lien vers l'espace de nom à l'aide du mot clef **using**

```
1  using System;
2
3  namespace Test
4  {
5      0 références
6      internal class Program
7      {
8          0 références
9          static void Main(string[] args)
10         {
11             Console.WriteLine("Hello World!");
12         }
13     }
14 }
```



Aucun problème détecté

class System.Console
Represents the standard input, output, and error streams for console applications. This class cannot be inherited.

Lig. : 7

WriteLine appartient à la classe Console rangée dans l'espace de nom : System

Signature (entête)

Toutes les méthodes ont une entête : une signature.

Une signature indique :

- Sa nature (static ou non)
- Le type du retour
- Les type et le nombre des éventuels paramètres

Ex : dans la classe Math : méthode Pow

```
public static double Pow (double x, double y);
```

Ex : dans la classe String : méthode Substring

```
public string Substring (int startIndex);
```

Une méthode commence toujours par une majuscule

Nature : méthode statique ou d'instance

En POO : il existe

- des **méthodes statiques** purement utilitaires : toutes les données sont passées en paramètre.

Ex :

```
double nb = 4;  
double carreNb = Math.Pow( nb, 2 );
```

Pow s'appuie sur la classe Math

- des **méthodes d'instances** (non statiques) : ce sont des méthodes à appliquer sur des objets. Seules les données externes à l'objet sont passées en paramètre.

Ex:

```
String nomGroupe = "TD1";  
String numero = nomGroupe.Substring(2);
```

Substring s'appuie sur l'objet nomGroupe

0	1	2
T	D	1

Nature : méthode statique ou d'instance

On peut simplifier ainsi : si je travaille

Avec des variables primitives de type
int, double, char, ...



Méthodes statiques
Méthode de classe

Avec des objets (=instances) ou
variables structurées de classe
String, Rectangle,....



Méthodes d'instances

Nature : méthode statique ou d'instance

Mais parfois, le même traitement peut être sous forme de méthode statique et de méthode d'instance.

Ex : au sein de la classe String :

- public **static** int **Compare** (string? strA, string? strB);
- public int **CompareTo** (string? strB);

Ex:

String **mot1** = "Abricot";

String **mot2** = "Champs";

<u>Méthode statique</u>	<u>Méthode d'instance</u>
int cmp = String .Compare(mot1 , mot2);	int cmp = mot1 .CompareTo(mot2);

renvoie un entier :

- < 0 si la 1ere chaine avant 2eme chaine
- == 0 si la 1ere chaine = 2eme chaine
- > 0 si la 1ere chaine après 2eme chaine

Signature – type de retour

Une signature : indique si une méthode retourne un résultat ou non

<u>Sans</u> résultat utilisation sans affectation	<u>Avec</u> résultat utilisation <u>avec</u> affectation
public static void WriteLine (string? value); Console.WriteLine(" Nom : ") ;	public static string? ReadLine (); <u>String nom =</u> Console.ReadLine() ;

Notion de retour

Retour : donnée transmise par le sous-programme au programme appelant : c'est le résultat du traitement.

```
// PROGRAMME APPELANT
public class Program
{
    public static void main ( String [ ] args )
    {
        String nom = Console.ReadLine() ;
        ...
    }
}

// SOUS-PROGRAMME
public class Console
{
    public static String ReadLine ()
    { ...
      return .... ;
    }
}
```

Signature – type de retour

Une signature : indique le **type** du résultat retourné.

Type de retour : chaîne de caractères

public static **string?** ReadLine ();

String? nom = Console.ReadLine() ;

Utilisation avec affectation d'une variable du type indiqué

//ou
String nom = Console.ReadLine() ;

Signature – nombre et type des paramètres

Une signature : indique le nombre et le type des paramètres.

1 paramètre de type chaîne de caractères est attendu

```
public static void WriteLine (string? value);
```

```
public static string? ReadLine ();
```

Aucun paramètre n'est attendu

Notion de paramètre

// PROGRAMME APPELANT

```
public class Program
{
    public static void main ( String [ ] args )
    {
        Console.WriteLine(" Nom : " ) ;
        ...
    }
}
```

Paramètre : donnée transmise par le programme appelant au sous-programme.

// SOUS-PROGRAMME

```
public class Console
{
    public static void WriteLine ( String value )
    { ...
    }
}
```

Ordre des paramètres

Pensez à bien lire la documentation pour en savoir plus sur les paramètres formels.

```
public static double Pow (double x, double y);
```

Paramètres

x Double

Nombre à virgule flottante double précision à élever à une puissance.

y Double

Nombre à virgule flottante double précision. qui spécifie une puissance.

Signature – ordre des paramètres

// PROGRAMME APPELANT

```
public class Program
{
    public static void main ( String [ ] args )
    {
        double nb = 4 , puissance = 2;
        double res = Math.Pow ( nb, puissance) ;
        ...
    }
}
```

4

2

Les paramètres doivent être passés dans le même ordre que celui défini par la signature et décrit dans la doc !

// SOUS-PROGRAMME

```
public class Math
{
    public static double Pow ( double x, double y )
    { ...
    }
}
```

Paramètres

x Double

Nombre à virgule flottante double précision à élever à une puissance.

y Double

Nombre à virgule flottante double précision. qui spécifie une puissance.

Paramètres effectifs / formels

Les variables utilisées comme paramètres effectifs n'ont pas à porter le même nom que les paramètres formels

// PROGRAMME APPELANT

```
public static void main ( String [ ] args )  
{  
    double nb = 2 , puissance = 3;  
    double res = Math.Pow ( nb, puissance) ;
```

Param effectifs :

- nb
- puissance

// SOUS-PROGRAMME

```
public class Math  
{  
    public static double Pow ( double x, double y)  
    { ...
```

Param formels :

- x
- y

Signature – surcharge

Les méthodes sont souvent **surchargées** :

Elles ont plusieurs signatures, et acceptent un nombre et, ou un type de paramètres différents.

Ex: WriteLine est surchargée 18 fois !

```
public static void WriteLine (string? value);  
public static void WriteLine (int value);  
public static void WriteLine (decimal value);  
...
```

```
Console.WriteLine()
```

▲ 1 sur 18 ▼ void Console.WriteLine()

Writes the current line terminator to the standard output stream.

Passage par valeur

Par défaut, les méthodes ne peuvent pas modifier directement les paramètres effectifs. Généralement, elles retournent une copie modifiée.

```
public static double Round (double value, int digits);
```

```
double nb = 3.5555 ;  
nb = Math.Round ( nb , 2 );
```

Ici, Round ne modifie pas nb.
Elle retourne une copie arrondie qu'on peut réattribuer à nb!

Passage par référence

Dans certaines signatures, vous trouverez **les mots clefs : ref, out**.
Cela donne le droit aux méthodes de modifier les paramètres effectifs
Souvent utilisé pour retourner plusieurs résultats.

```
public static bool TryParse (string? s, out double result);
```

```
double nb ;  
bool ok = double.TryParse( Console.ReadLine( ) , out nb ) ;
```

ici nb est un « faux paramètre ».
car il sert de zone de stockage du résultat de la conversion

Remarque

Il est possible :

- de ne pas stocker le résultat d'une méthode
- d'utiliser le résultat d'une méthode au sein d'une autre méthode ou expression

```
double nb = 2 , puissance = 3;  
Console.WriteLine( Math.Pow ( nb, puissance) ) ;
```

```
double nb = 2 , puissance = 3;  
double res = Math.Pow ( nb, puissance);  
Console.WriteLine( res ) ;
```

```
double taille = Double.Parse( Console.ReadLine( ) ) ;
```

Utilisation méthode d'instance

On préfixe la méthode par un objet (instance) ou variable structurée de la classe.

Pour créer un objet ou une variable structurée : A l'exception des objets de classe String, il faut :

- Utiliser le mot clef **new**
- Utiliser un **constructeur** : méthode de construction de l'objet qui porte le même nom que la classe

Ex avec la stucture DateTime:

```
DateTime anni = new DateTime (2000, 11, 28 );
```

Exemples - Méthodes dédiées aux DateTime

La grande majorité des méthodes définies dans DateTime sont des méthodes non statiques

- public TimeSpan Subtract (DateTime value);
- public string ToLongDateString ();
- public DateTime AddYears (int value);
- public DateTime AddMonths (int value);
- ...

Ex avec un objet de classe DateTime:

```
DateTime date = new DateTime(2022,9,11 );  
TimeSpan temps = date.Subtract( DateTime.Today );  
String formatLongDate=date.ToLongDateString();  
DateTime dateDans10Ans=date.AddYears(10);
```

Dimanche 11 Septembre 2022

Day :11
Month :9
Year : 2032

Exemples – Méthodes dédiées aux String

Une chaîne de caractères est un objet.
String est une classe « intégrée », simplifiée : pas besoin de new ...

```
public int IndexOf (string value);  
public string ToUpper ();  
public string ToLower ();  
public string Substring (int startIndex, int length);
```

```
String txt = "tout est ok";  
int positionOk = txt.IndexOf("ok");
```

0	1	2	3	4	5	6	7	8	9	10
t	o	u	t		e	s	t		o	k

9

```
txt = txt.ToUpper();  
Console.WriteLine(txt);
```

TOUT EST OK

```
txt = txt.Substring(5, 3);  
Console.WriteLine(txt);
```

EST

Ne pas confondre méthodes et propriétés

Tous les objets disposent :

- De méthodes : traitements : toujours mettre () avec ou sans paramètres
- De propriétés : infos contenues dans l'objet

The screenshot shows a code editor with the following C# code:

```
DateTime dateNaissance = new DateTime(2001, 12, 23);
```

Below the code, the text `dateNaissance.` is entered, and the IntelliSense menu is open, displaying a list of methods and properties. The methods listed are:

- AddSeconds
- AddTicks
- AddYears
- CompareTo
- Date
- Day
- DayOfWeek

Two callout boxes are present:

- A box labeled "méthodes" points to the list of methods.
- A box labeled "propriétés" points to the list of properties (which are not visible in the current view).

DÉFINIR DES MÉTHODES STATIQUES

Exemple simple avec un type void

```
class Politesse
{
    public static void Bonjour()
    {Console.WriteLine ( "Bonjour"); }
```

Définition

```
class Program
{
    static void Main(string[] args)
    {
        Politesse.Bonjour () ;
    }
```

Utilisation

Où définir des méthodes ?

On peut les définir en dessous du main dans le fichier Program.cs
Préférez tout de même un rangement au sein d'une autre classe.

```
class Program
{
    static void Main(string[] args)
    {
        Program.Bonjour();
    }

    static void Bonjour()
    {
        Console.WriteLine ( "Bonjour");
    }
}
```

```
class Program
{
    public static void Main(string[] args)
    {
        Politesse.Bonjour();
    }
}
```

```
class Politesse
{
    public static void Bonjour()
    {
        Console.WriteLine ( "Bonjour");
    }
}
```

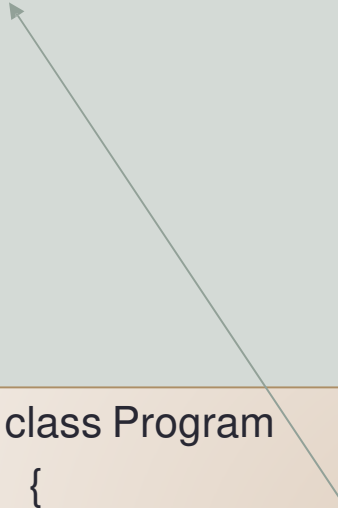
Exemple simple de surcharge

On redéfinit la méthode, mais cette fois avec un paramètre.

```
class Politesse
{
    public static void Bonjour( String prenom )
    {
        Console.WriteLine ( "Bonjour " + prenom);
    }

    public static void Bonjour()
    {
        Console.WriteLine ( "Bonjour");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Politesse.Bonjour("Noémie") ;
    }
}
```



Exemple simple avec un retour

```
class MesMaths
```

```
{
```

```
    public static double CalculPerimetre(double largeur)
```

```
    {
```

```
        double res ;
```

```
        res = 4 * largeur;
```

```
        return res;
```

```
    }
```

```
class Program
```

```
{
```

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

```
    {
```

```
        double largeurCarre = 5;
```

```
        double perimetre = MesMaths.CalculPerimetre(largeurCarre);
```

```
        Console.WriteLine("Perimetre : " + perimetre);
```

```
    }
```

```
}
```

Notion de variable locale

Chaque programme a son espace mémoire à lui !

```
static void Main(string[ ] args)
{
    double largeurCarre = 5;
    double perimetre = MesMaths.CalculPerimetre(largeurCarre);
    Console.WriteLine("Perimetre : " + perimetre);
}
```

largeurCarre
perimetre

```
public static double CalculPerimetre(double largeur)
{
    double res;
    res = 4 * largeur;
    return res;
}
```

largeur
res

Notion de variable locale

Même si les variables ont le même nom, ce n'est pas les mêmes emplacements mémoires !

```
static void Main(string[ ] args)
{
    double largeur = 5;
    double perimetre = MesFctMath.CalculPerimetre(largeur);
    Console.WriteLine("Perimetre : " + perimetre);
}
```

largeur
perimetre

```
public static double CalculPerimetre(double largeur)
{
    double perimetre;
    perimetre = 4 * largeur;
    return perimetre;
}
```

largeur
perimetre

Notion de variable locale

Chaque programme a son espace mémoire à lui !

```
static void Main(string[ ] args)
{
    double largeur = 5;
    double perimetre = MesMaths.CalculPerimetre(largeur);
}
```

5

20

```
public static double PerimetreCarre(double largeur)
{
    double perimetre;
    perimetre = 4 * largeur;
    return perimetre;
}
```

largeur = 5
perimetre = **20**

largeur = **5**
perimetre = 20

Existence
temporaire

Notion de retour

Une méthode avec type de retour != void

- renvoie une seule valeur.
- a un seul objectif.
- retourne toujours une valeur.

Conseil : placez le return toujours sur la dernière ligne !

```
public static int Methode ( )  
{  
    int res ;  
    ....  
    return res ;  
}
```

Reflexe : je déclare une variable locale res conforme au type de retour et j'écris le return tout de suite à la fin !

Notion de paramètre

Un paramètre est une variable locale un peu spécifique :

Elle est :

- initialisée avec la valeur contenue dans le paramètre effectif (donnée provenant du programme appelant lors de l'appel du sous-programme.)
- déclarée au sein de la signature

Notion de paramètre

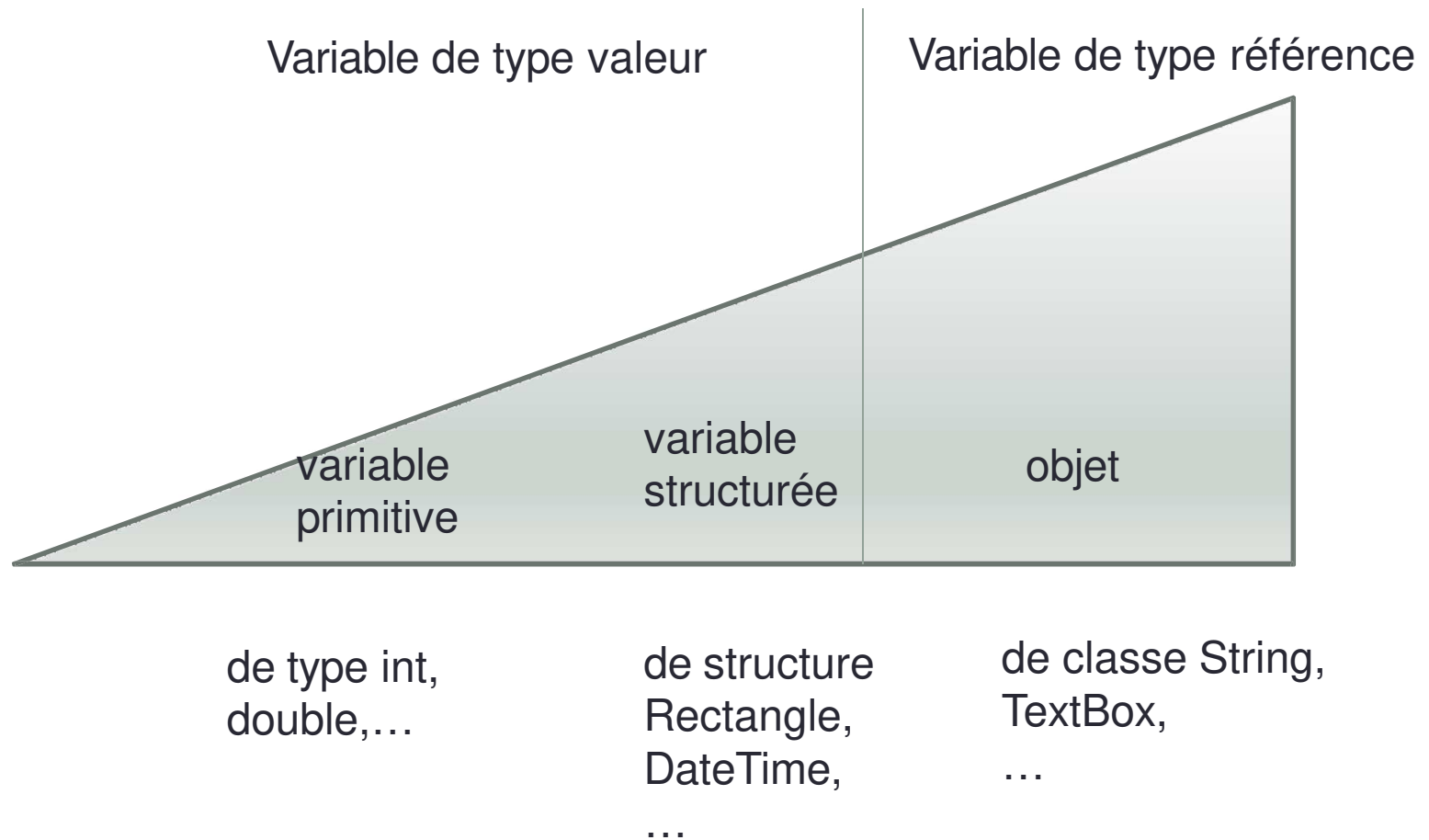
Les paramètres sont passés **par valeur** (par défaut) : c'est une recopie ! mais peuvent être explicitement passés **par référence** (ref, out)

Mais attention au type du paramètre !

- Les **types valeurs** : int, double, char, ... mais aussi les structures: **les variables des types valeur contiennent directement leurs données.**
- Les **types références** : les classes. Ex: RectangleShape, CircleShape, Sprite : **les variables des types référence contiennent la référence (l'adresse de la zone mémoire) de leurs données.**

Par défaut, une méthode ne peut pas modifier un paramètre de type primitif ou structuré, mais peut modifier un objet

Evolution des variables

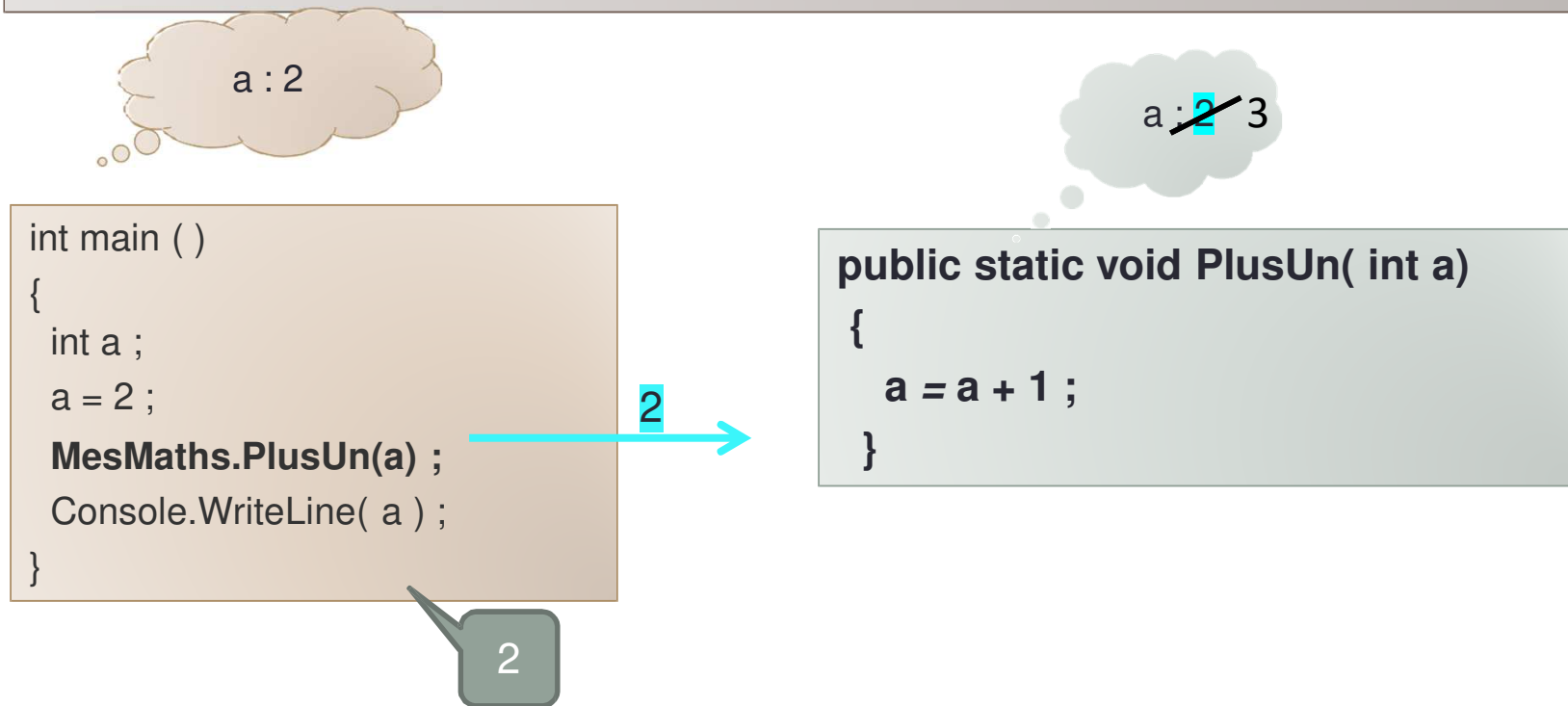


Par défaut : passage par valeur d'un type valeur

La valeur du paramètre est recopiée.

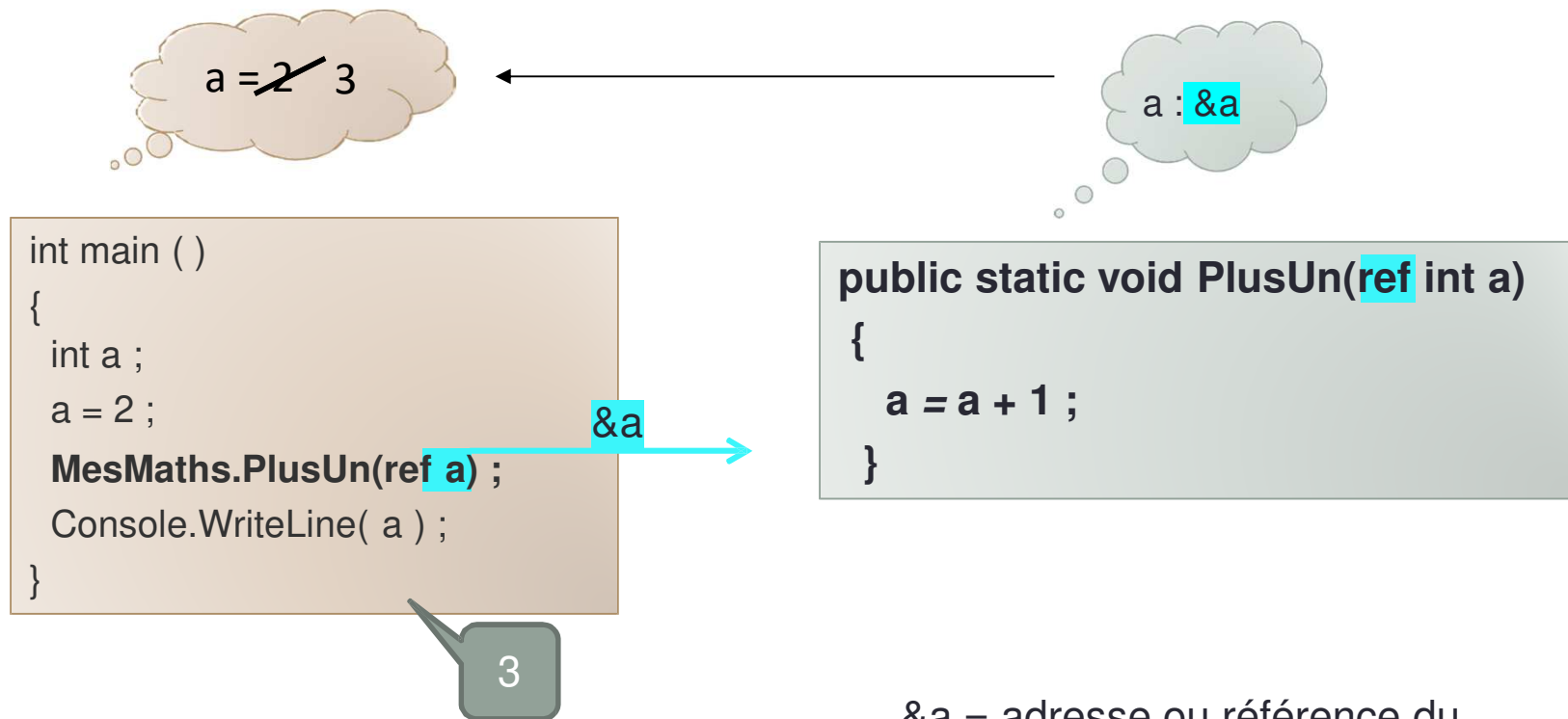
Toute modification du paramètre ne modifie que la copie.

Par défaut, les paramètres de type valeurs sont non modifiables



Passage par référence d'un type valeur

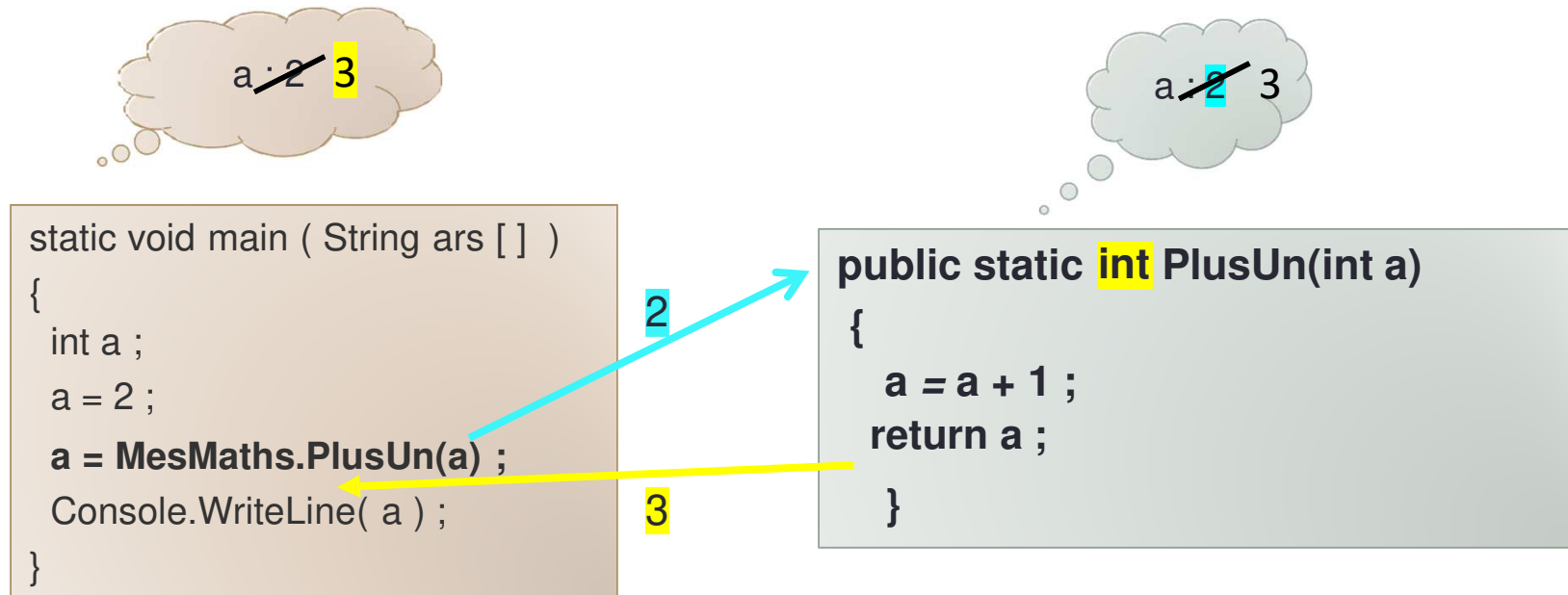
Une méthode peut modifier les variables issues du programme appelant si elles sont passées par référence.



&a = adresse ou référence du stockage pour a

Se passer du passage par référence

On peut souvent se passer du passage par référence, il suffit de faire un retour !



Passage par référence d'un type valeur

- **ref** : passage par référence, le paramètre peut être modifié
- **out** : passage par référence, le paramètre va être modifié : il n'a pas à être initialisé avant l'appel de la méthode
- **in** : passage par référence, le paramètre ne doit pas être modifié : Utile pour un gain de non recopie : optimisation

Attention à ne pas confondre objet et variable structurée

Il existe la notion d'objet et de variable structurée : elles sont très proches. Une variable structurée s'initialise et se manipule comme un objet. Par contre, c'est un type valeur !

```
Rectangle r = new Rectangle(0,0,200, 100);  
MesFct.Double(r);
```

X:0
Y:0
Width: 200
Height: 100

```
class MesFct  
{  
    public static void Double(Rectangle r)  
    {  
        r.Width = r.Width * 2;  
        r.Height = r.Height * 2;  
    }  
}
```

X:0
Y:0
Width: ~~200~~ 400
Height: ~~100~~ 200

Le rectangle initial ne sera pas doublé !

Tester la validité des paramètres

Au sein d'une méthode, avant de faire tout traitement, il faut s'assurer de la validité des paramètres

```
static void Main(string[] args)
{
    double dividende, diviseur ;
    dividende = double.Parse(Console.ReadLine());
    diviseur = double.Parse(Console.ReadLine());
    Console.WriteLine( Program.Division(dividende, diviseur));
}

public static double Division ( double dividende , double diviseur )
{
    if (diviseur == 0 )
        throw new ArgumentException("Attention, le diviseur ne peut pas être 0");
    double res = dividende /diviseur ;
    return res;
}
```

Exception

Pour indiquer un problème avec un paramètre, on lance une exception de type [ArgumentException](#).

On peut être plus précis : avec les 2 sous catégories suivantes :

[ArgumentNullException](#)

[ArgumentOutOfRangeException](#)