

COURS DE C#

LES FONDEMENTS



CogniTIC
Building Together Better IT Solutions

- Premier programme
- Visual Studio
- Les variables
- Types de données prédéfinis
- Les conversions
- Les instructions conditionnelles
- Les opérateurs
- Les boucles
- Les tableaux
- Les structures
- Les méthodes
- Les énumérations

C# - LES FONDEMENTS



RÉFÉRENCES

C# - LES FONDEMENTS

- O'Reilly :
C# 4.0 in a nutshell (ISBN-13 : 978-0-596-80095-6)
C# 5.0 in a nutshell (ISBN-13 : 978-1-4493-2010-2)
- Microsoft :
Spécification du langage C# 5.0 ([http://msdn.microsoft.com/fr-fr/library/vstudio/ms228593\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/vstudio/ms228593(v=vs.110).aspx))



PREMIER PROGRAMME

C# - LES FONDEMENTS

- Le Framework .NET
- Avec Notepad++
- Avec Visual Studio

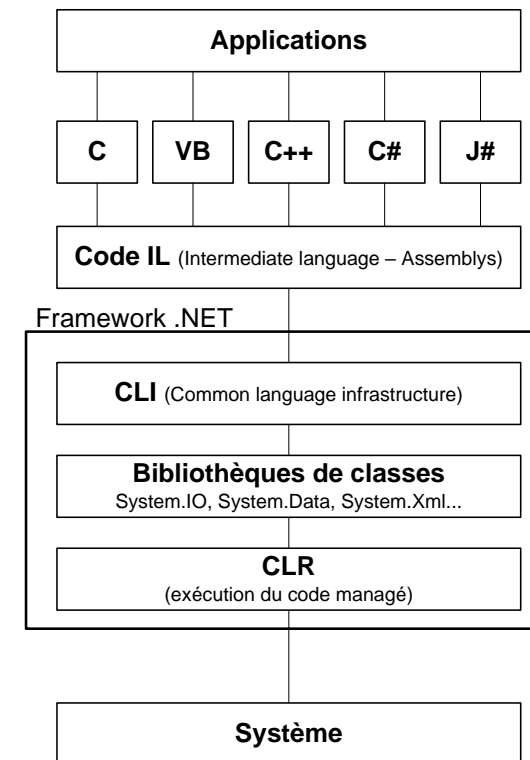
PREMIER PROGRAMME

LE FRAMEWORK .NET

Le **.NET Framework** est un Framework pouvant être utilisé par un système d'exploitation Microsoft Windows, il s'appuie sur la norme Common Language Infrastructure (CLI) qui est indépendante du langage de programmation utilisé. Ainsi tous les langages compatibles respectant la norme CLI ont accès à toutes les bibliothèques installées (installables) dans l'environnement d'exécution.

La **Common Language Infrastructure** (CLI) décrit un environnement qui permet d'utiliser de nombreux langages de haut niveau sur différentes plates-formes sans nécessité de réécrire le code pour des architectures spécifiques.

Le code répondant aux spécifications CLI est dit « managed code » en anglais, littéralement « code géré » qui est souvent traduit abusivement par « code managé ».

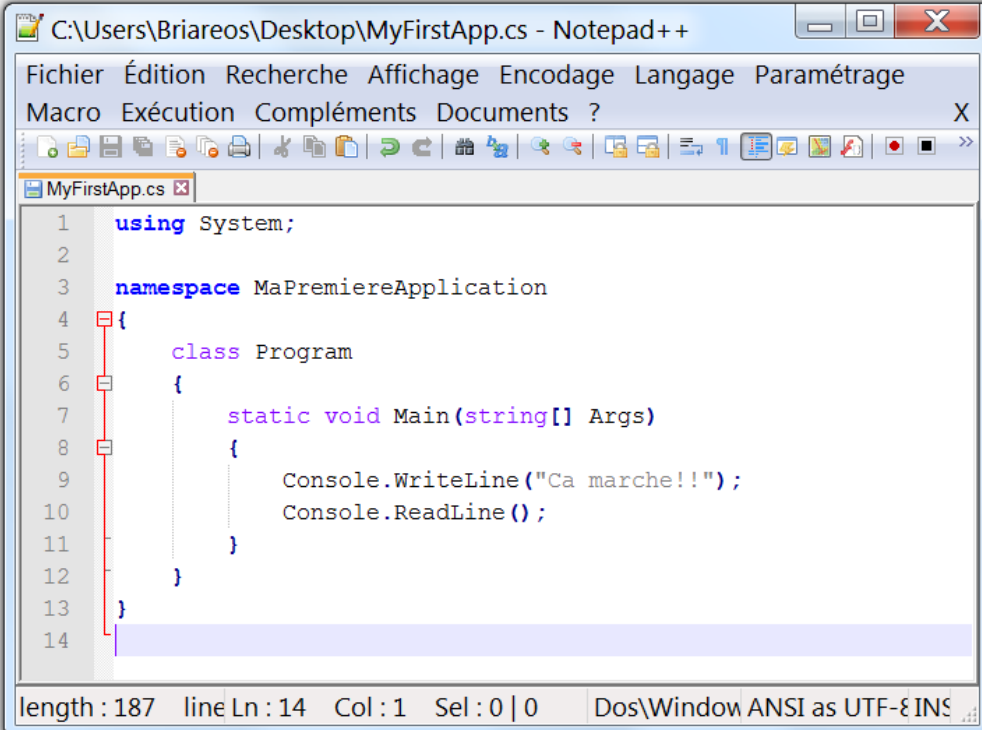


AVEC NOTEPAD++

Commençons par créer notre premier programme, pour cela nous allons utiliser Notepad++

- Ouvrons Notepad++
- Tapons le code ci-joint
- Enfin, sauvegardons le fichier, sur le bureau, sous le nom « MaPremiereApplication.cs »

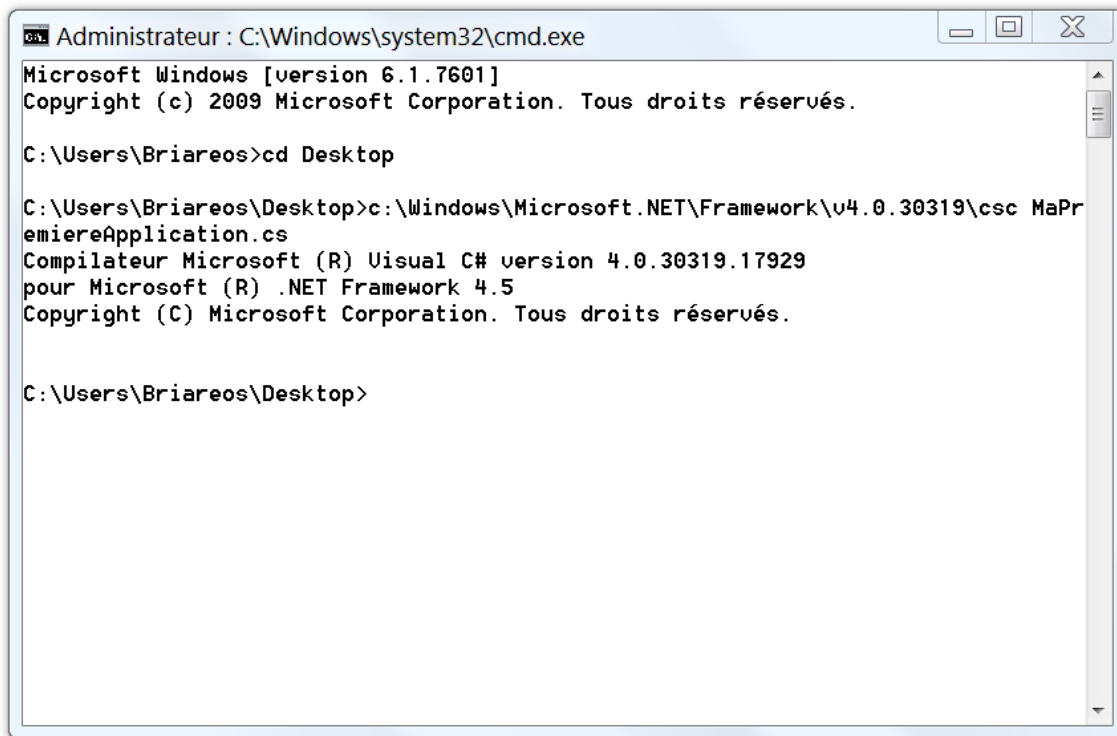
Nous verrons les mots-clés
« using », « namespace » et « class »
dans la partie « Orienté Object »



```
1  using System;
2
3  namespace MaPremiereApplication
4  {
5      class Program
6      {
7          static void Main(string[] Args)
8          {
9              Console.WriteLine("Ca marche!!");
10             Console.ReadLine();
11         }
12     }
13 }
14
```

length : 187 line Ln : 14 Col : 1 Sel : 0 | 0 Dos\Window ANSI as UTF-8 INS

AVEC NOTEPAD++



```
Administrateur : C:\Windows\system32\cmd.exe
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\Briareos>cd Desktop

C:\Users\Briareos\Desktop>c:\Windows\Microsoft.NET\Framework\v4.0.30319\csc MaPr
emiereApplication.cs
Compilateur Microsoft (R) Visual C# version 4.0.30319.17929
pour Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. Tous droits réservés.

C:\Users\Briareos\Desktop>
```

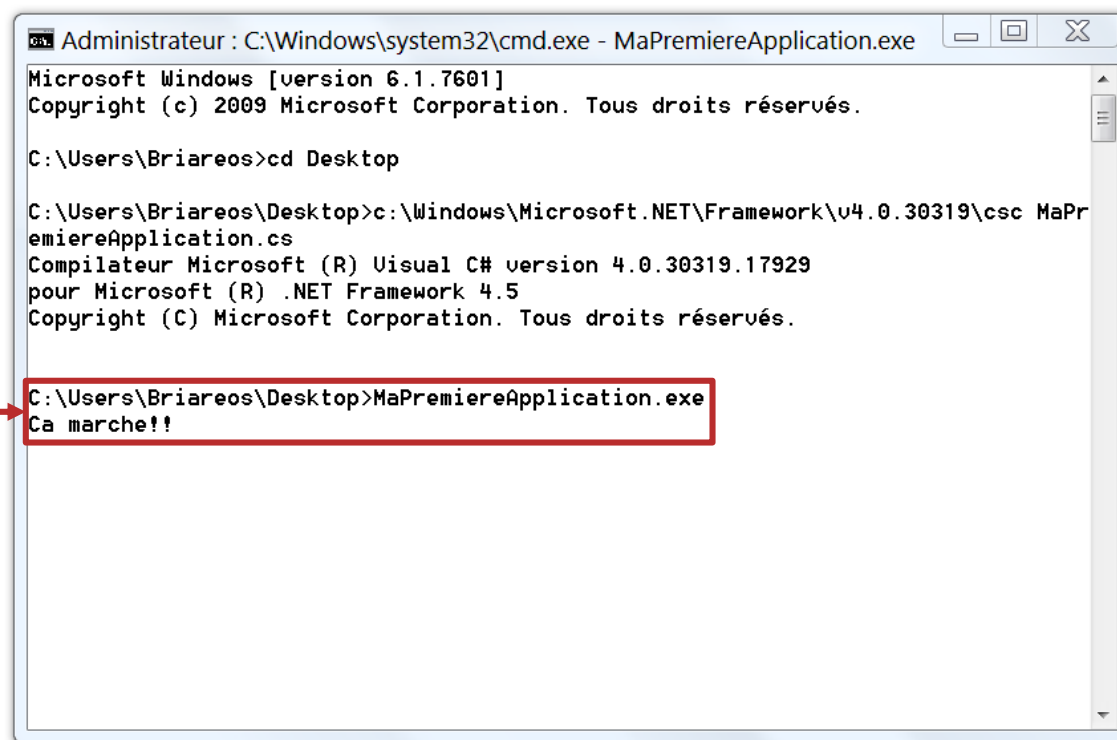
Une fois notre code écrit, passons à la compilation afin de créer notre fichier exécutable.

- Pour cela, ouvrons une fenêtre DOS - Touche « Windows » + « R »
- Allons sur notre bureau - cd Desktop
- Écrivons : « c:\Windows\Microsoft.NET\Framework\v4.0.30319\csc MaPremiereApplication.cs »

Le Framework 4.5 remplace le Framework 4.0 lors de son installation

AVEC NOTEPAD++

Ile ne nous reste qu'à exécuter notre programme nommé par défaut : « MaPremiereApplication.exe ».



```
Administrateur : C:\Windows\system32\cmd.exe - MaPremiereApplication.exe
Microsoft Windows [version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.

C:\Users\Briareos>cd Desktop

C:\Users\Briareos\Desktop>c:\Windows\Microsoft.NET\Framework\v4.0.30319\csc MaPr
emiereApplication.cs
Compilateur Microsoft (R) Visual C# version 4.0.30319.17929
pour Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. Tous droits réservés.

C:\Users\Briareos\Desktop>MaPremiereApplication.exe
Ca marche!!
```

VIA VISUAL STUDIO

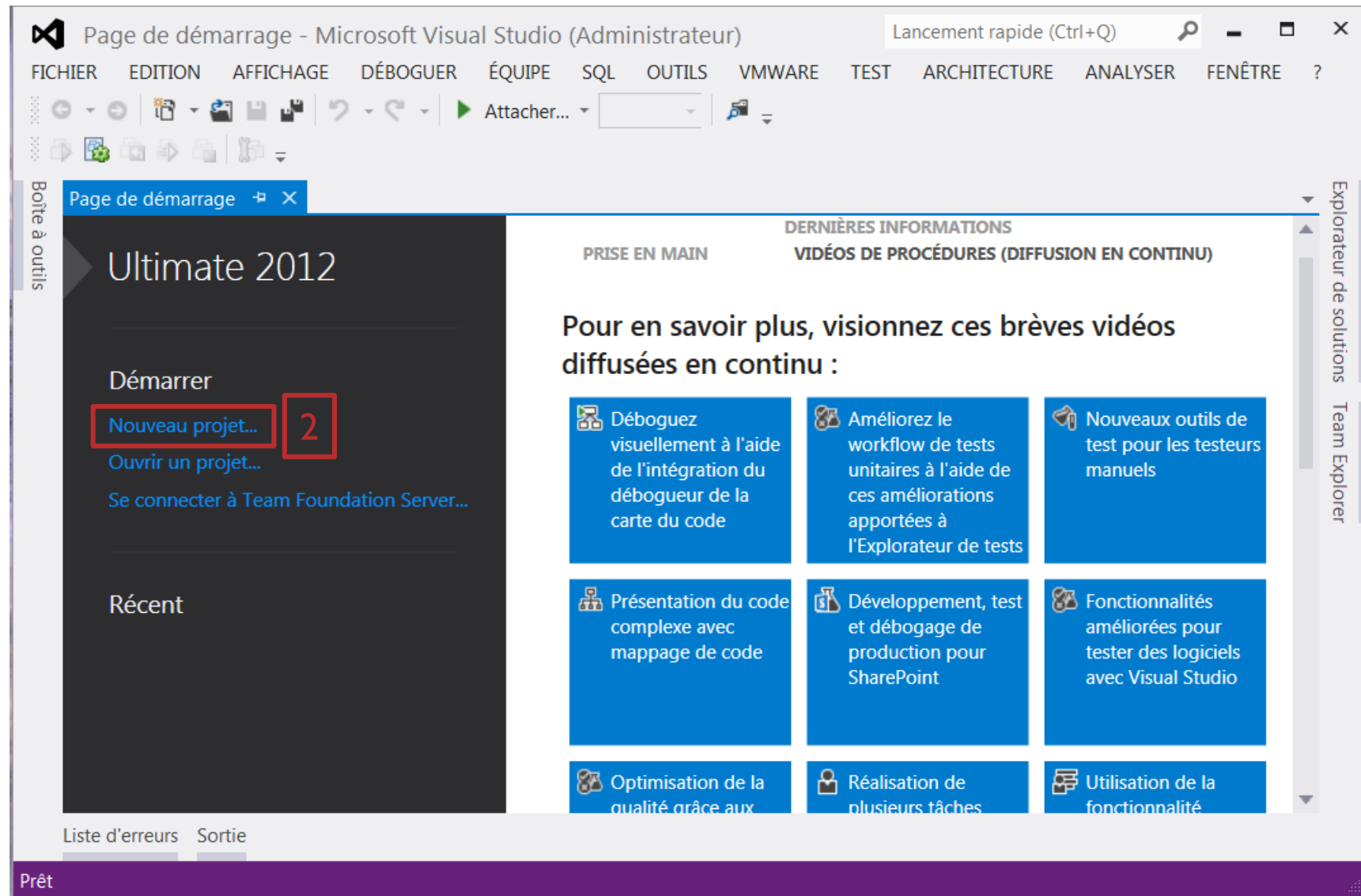
Afin de réaliser plus aisément nos applications, nous utiliserons à l'avenir Visual Studio.

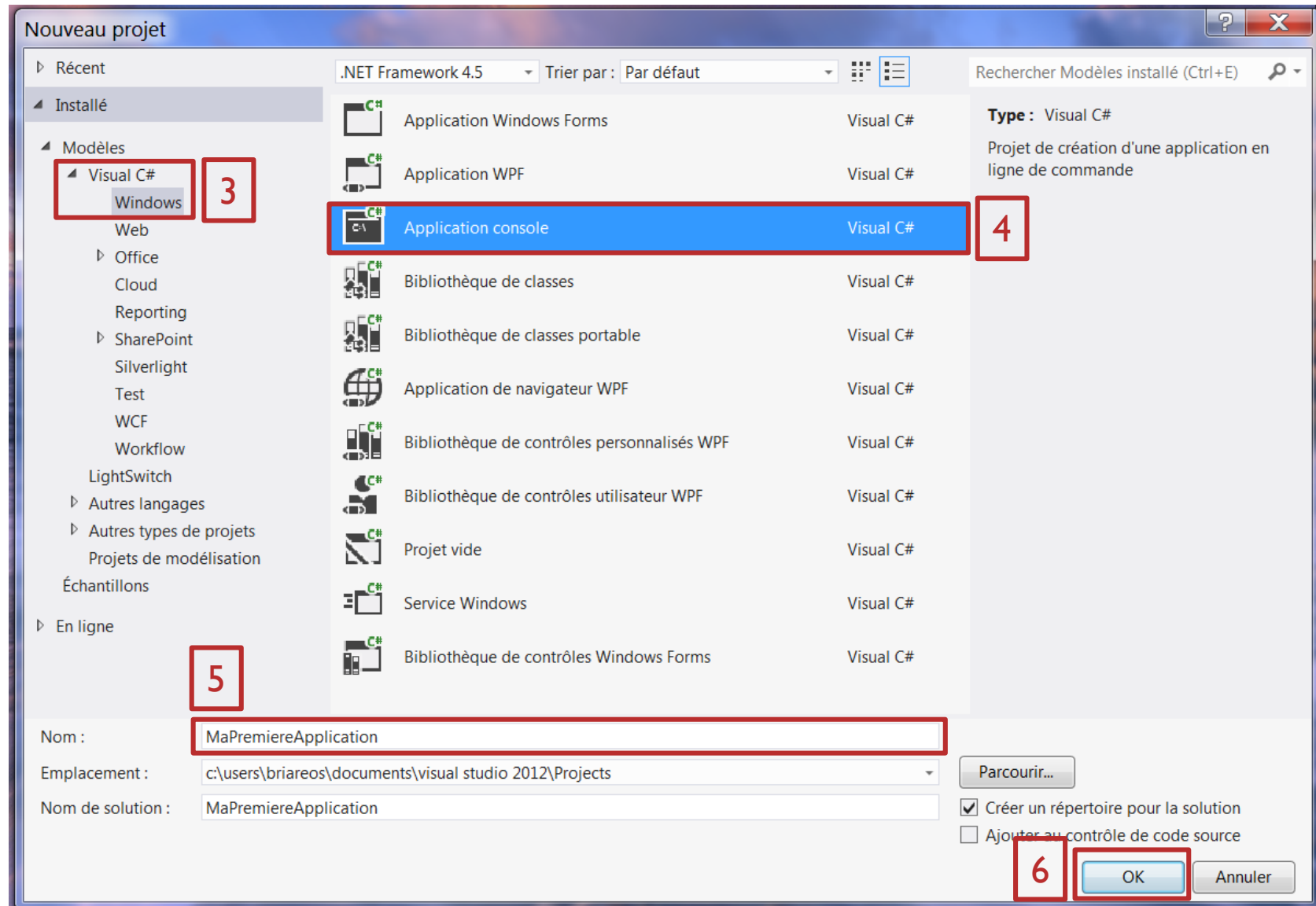
Visual Studio est un ensemble complet d'outils de développement permettant de générer des applications Web ASP.NET, des Services Web, des applications bureautiques et des applications mobiles.

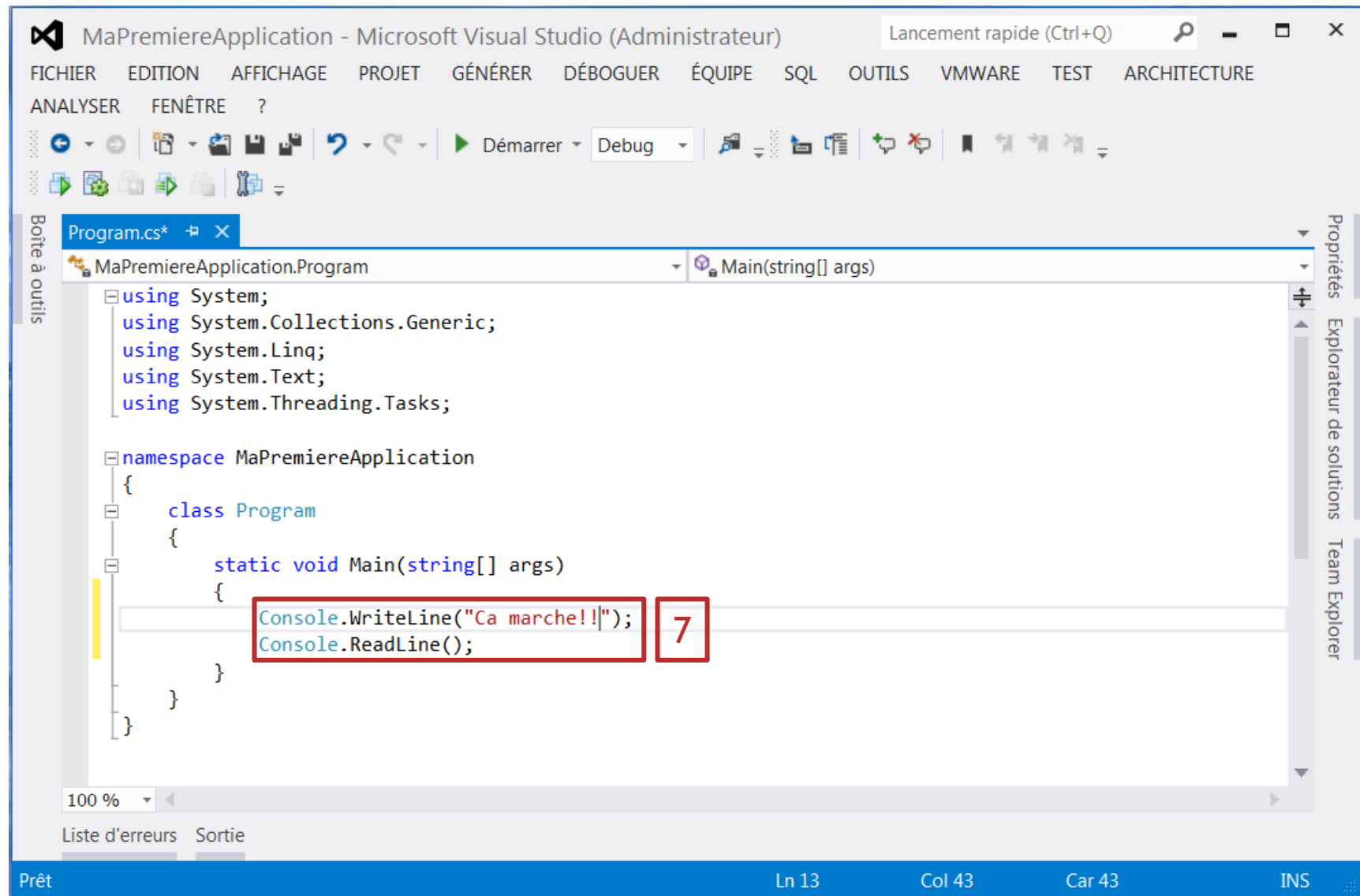
Visual Basic, Visual C++, Visual C# et Visual F# utilisent tous le même environnement de développement intégré qui leur permettent de partager des outils et facilite la création de solutions faisant appel à plusieurs langages.

Créons donc notre application!

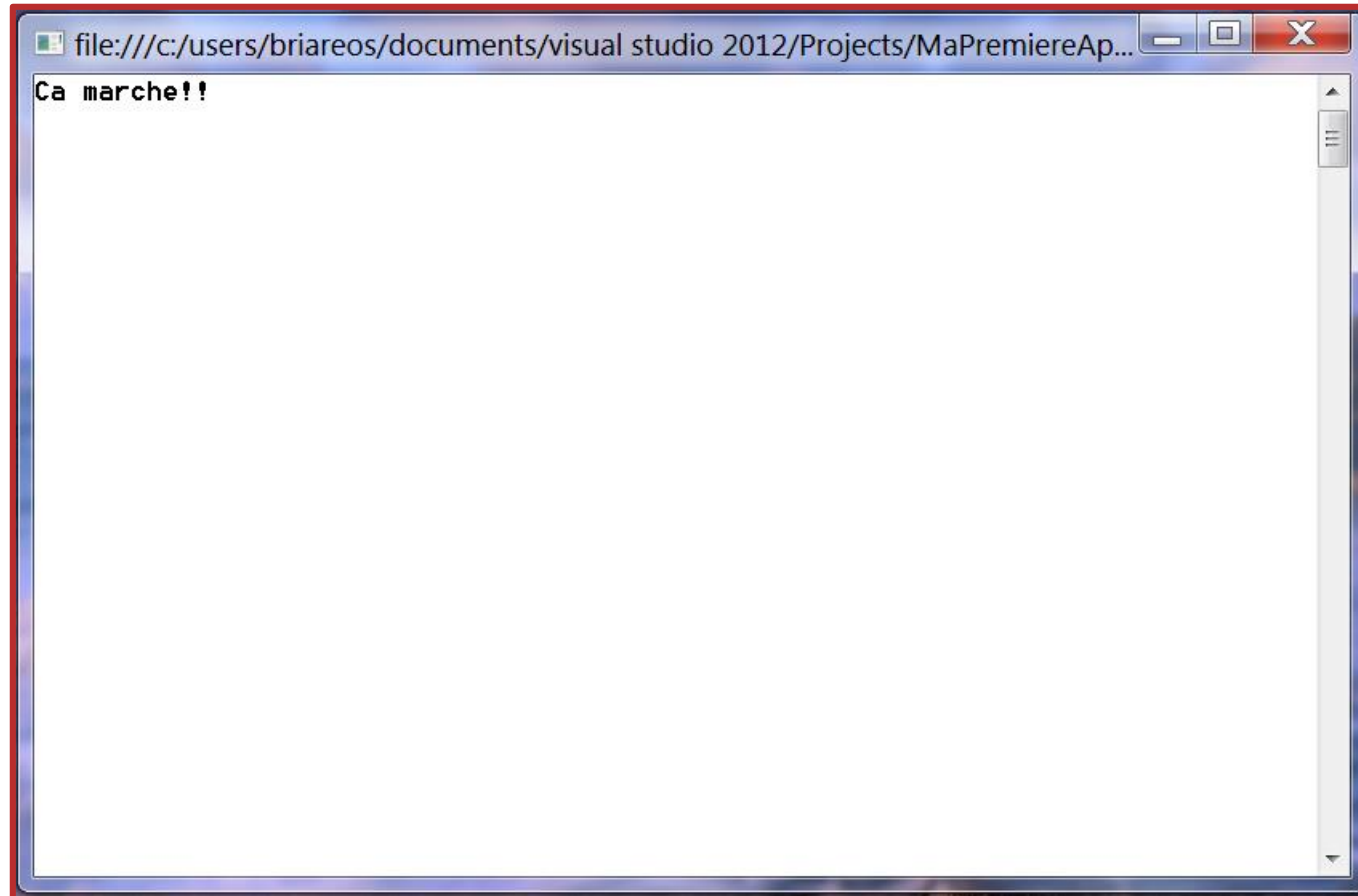
1. Ouvrons Visual Studio
2. Cliquons sur Nouveau Projet
3. Sélectionnons le langage « Visual C# » → « Windows »
4. Sélectionnons « Application Console »
5. Donnons lui un nom : « MaPremiereApplication »
6. Validons
7. Retapons le code C#
8. Pressons la touche « F5 » pour lancer une session de débogage







8 - résultat





VISUAL STUDIO

C# - LES FONDEMENTS

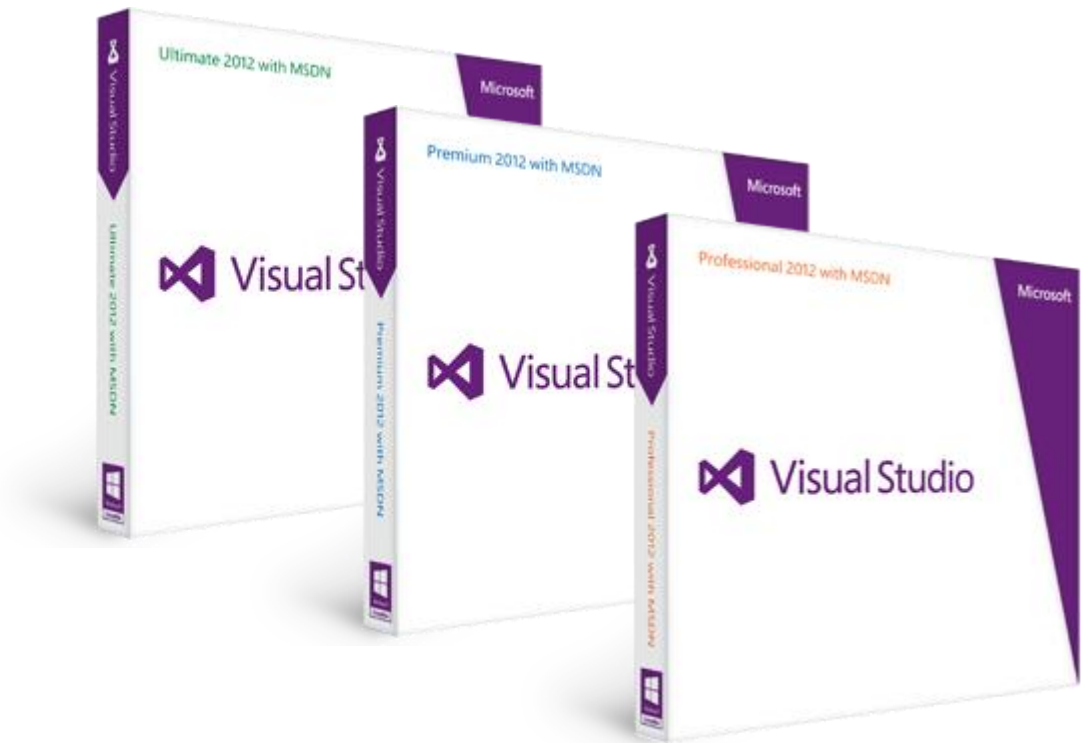
- Présentation de Visual Studio
- Gestion des projets récents
- Création d'un projet
- Présentation de l'interface
- La gestion du code
- Déboguer son application

PRÉSENTATION DE VISUAL STUDIO

Visual Studio 2015 est une IDE (Integrated Development Environment) prévu pour développer des applications .Net, Il se décline en 6 variantes.

- Ultimate (13 299 \$)
- Premium (6 119 \$)
- Test Professional (2 169 \$)
- Professional (1 199 \$)
- Team Foundation Server
- Community (Gratuite)

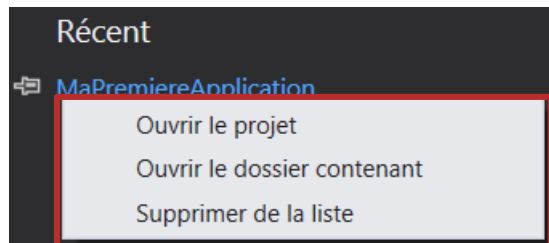
<http://www.microsoft.com/visualstudio/fra/>



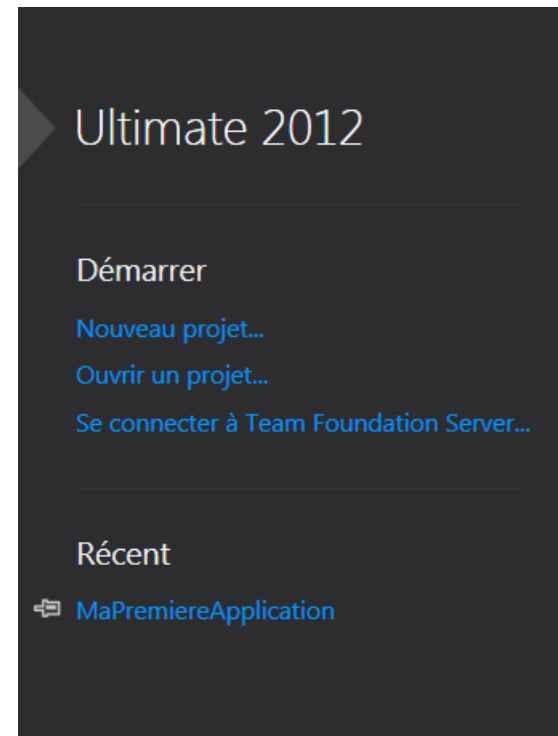
GESTION DES PROJETS RÉCENTS

La liste de nos derniers projets est disponible dès l'ouverture de Visual Studio.

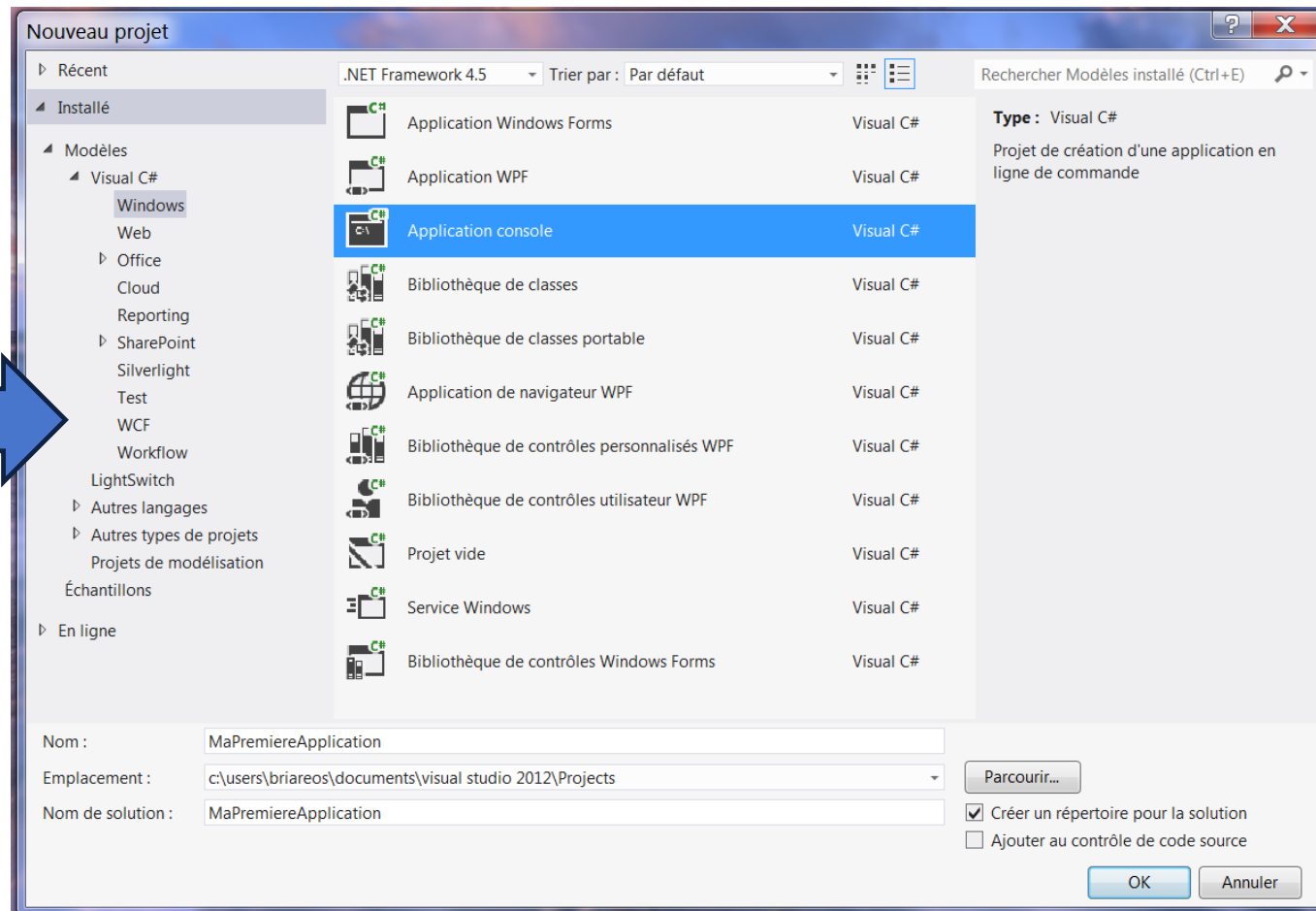
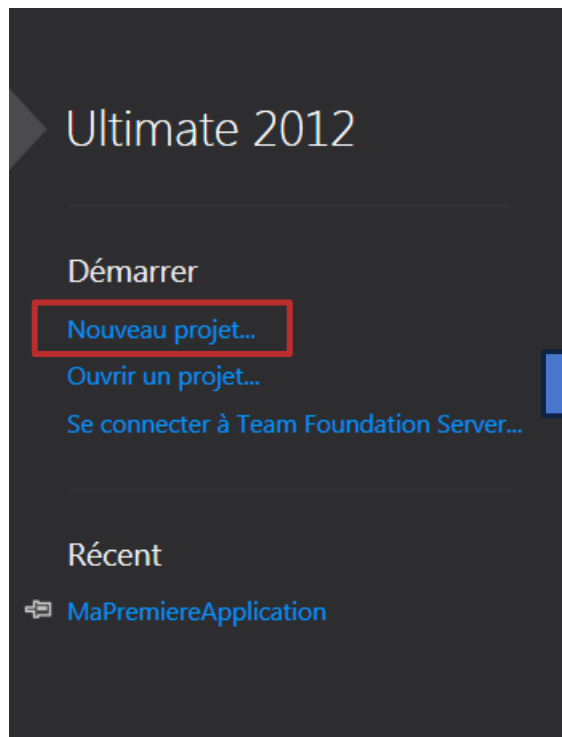
Il nous est possible d'organiser les projets récent via un « clique droit »



Il nous sera également possible de les verrouiller afin qu'il restent dans la liste, en cliquant sur la punaise à gauche du projet concerné.

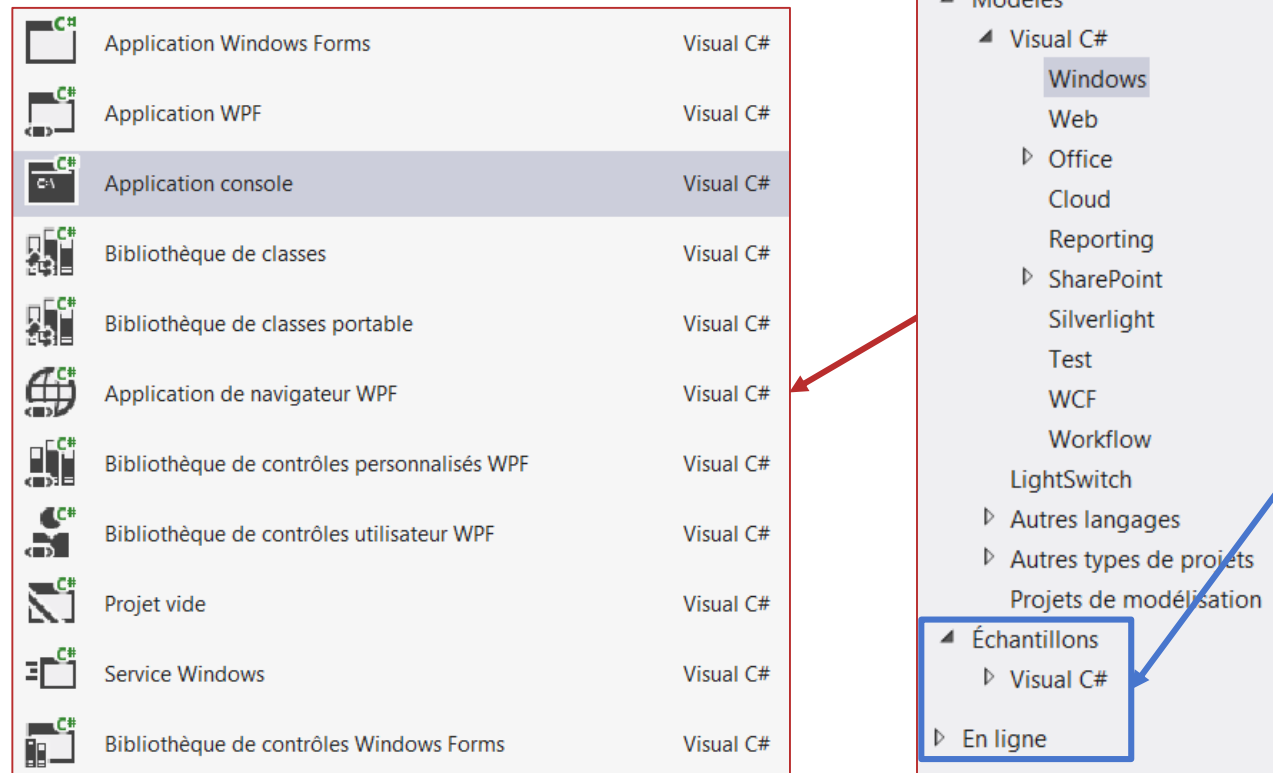


CRÉATION D'UN PROJET



CRÉATION D'UN PROJET

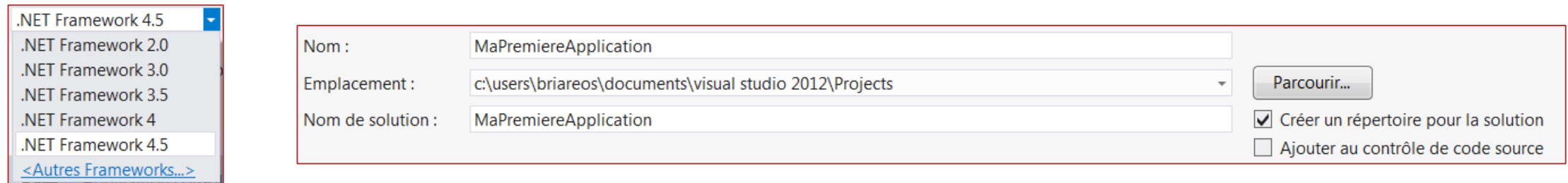
Nous avons la possibilité de choisir le langage et le modèle de projet à utiliser.



De plus, Il nous est également possible d'ajouter de nouveau modèle de projets ou de télécharger des bouts de projets avec des fonctionnalités depuis le web.

CRÉATION D'UN PROJET

Dans cette fenêtre, nous avons également la possibilité de choisir la version du Framework .Net que notre application utilisera.



Framework selection dropdown:

- .NET Framework 4.5
- .NET Framework 2.0
- .NET Framework 3.0
- .NET Framework 3.5
- .NET Framework 4
- .NET Framework 4.5
- <Autres Frameworks...>

Form fields and options:

- Nom : MaPremiereApplication
- Emplacement : c:\users\briareos\documents\visual studio 2012\Projects
- Nom de solution : MaPremiereApplication
- Parcourir...
- ☒ Créer un répertoire pour la solution
- ☐ Ajouter au contrôle de code source

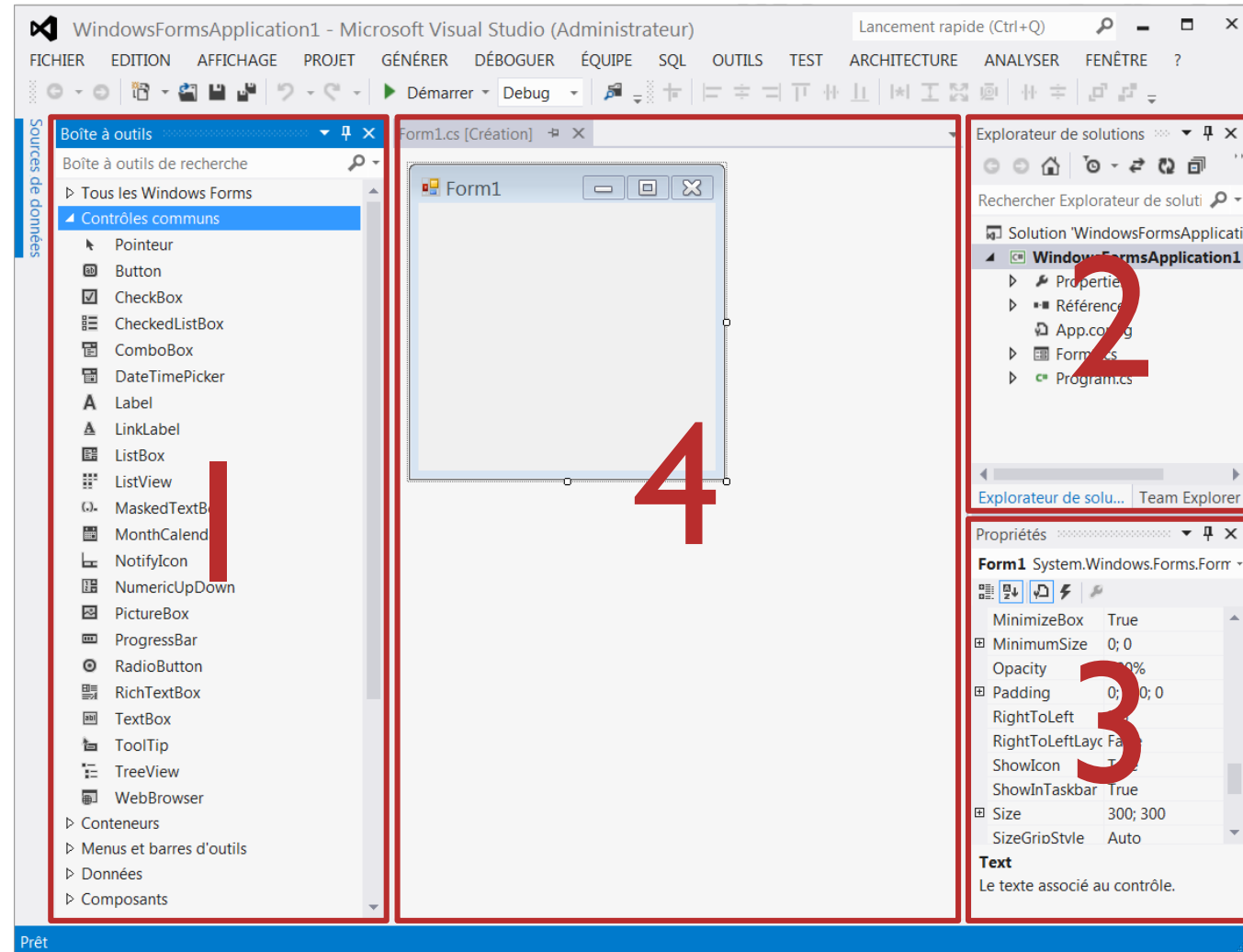
Nous pouvons spécifier le nom de notre projet, son emplacement et le nom de la solution qui sera créée pour notre projet.

Enfin, il nous est également possible de spécifier si nous souhaitons créer un répertoire pour notre solution et d'ajouter un contrôle de source en coopération avec Team Fondation Server.

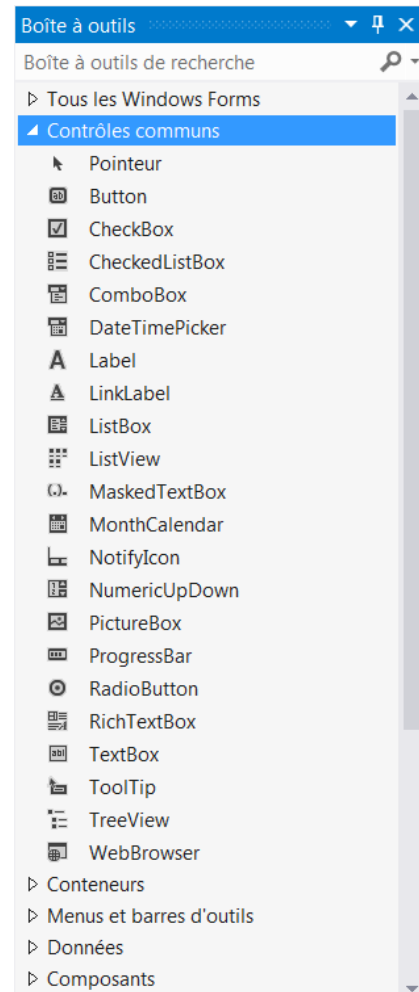
Si lorsque nous créons notre projet celui-ci n'a pas de solution affectée, Visual Studio créera automatiquement cette solution.

PRÉSENTATION DE L'INTERFACE

1. La boîte à outils
2. L'explorateur de solution
3. Les propriétés
4. La zone Design/Code

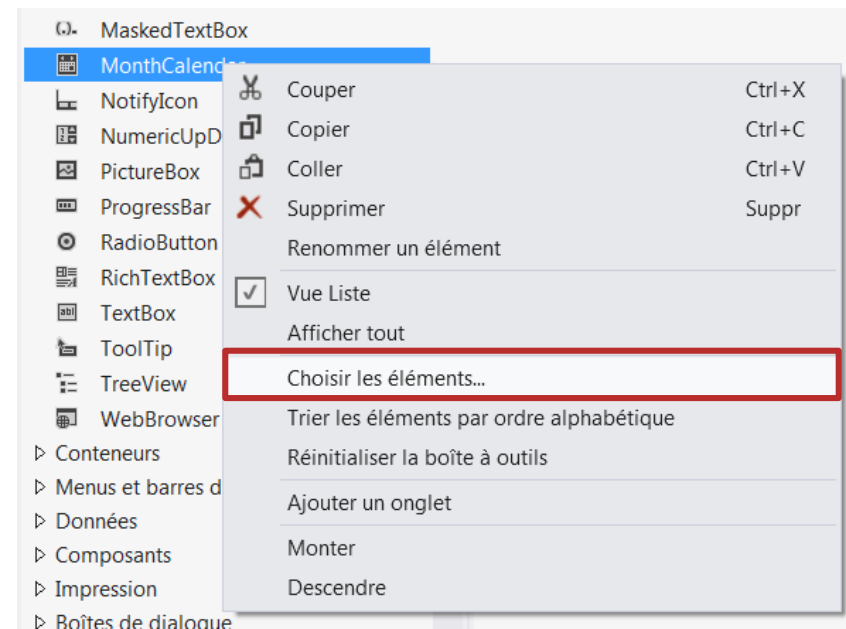


PRÉSENTATION DE L'INTERFACE - LA BOÎTE À OUTILS



Lorsque nous travaillons avec du design, la boîte à outil nous permet d'obtenir tous les composants visuels pouvant être déposés sur notre interface.

Il nous est possible d'ajouter, à cette liste, des contrôles à partir du Framework .Net ou d'autre COM,

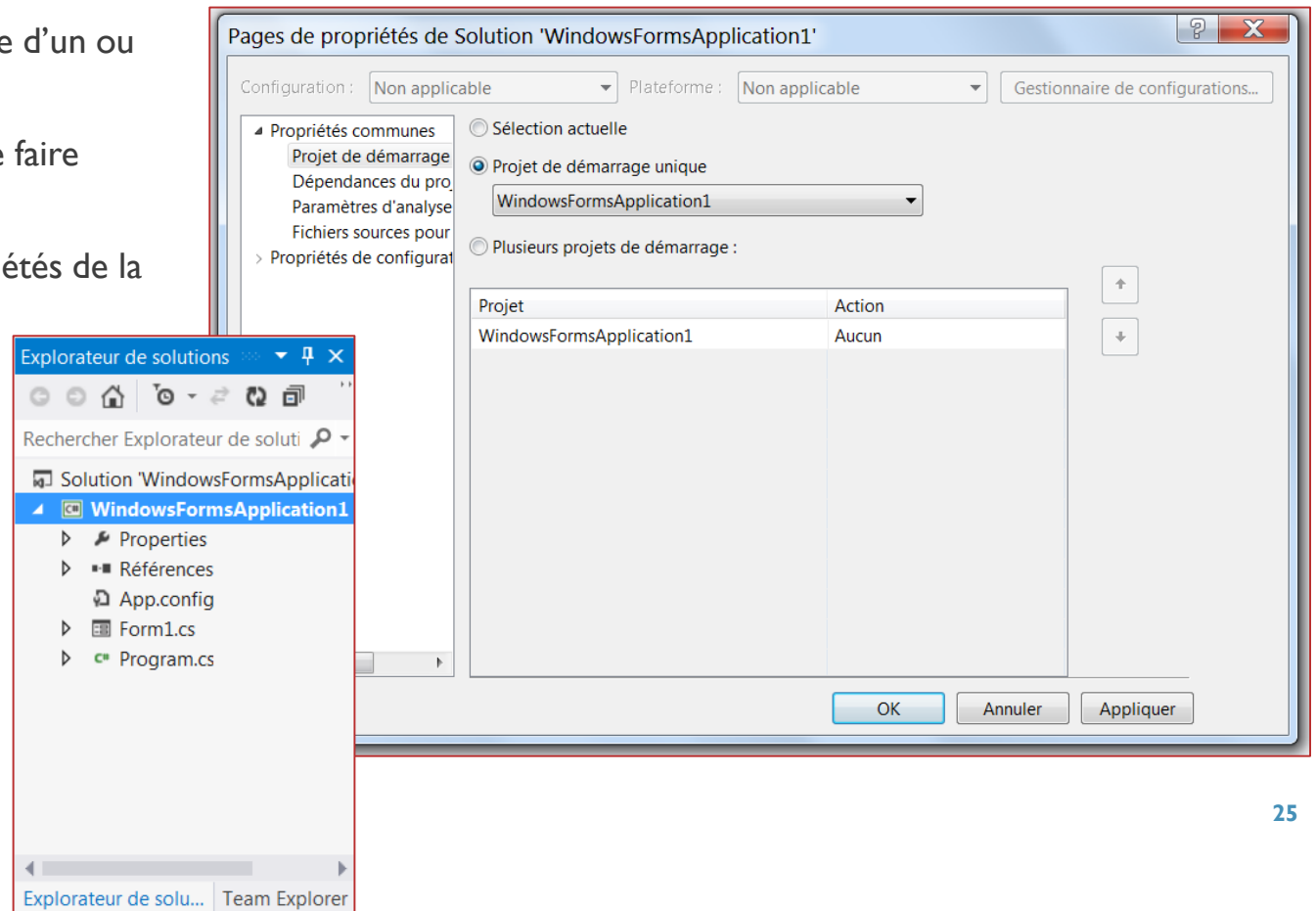


PRÉSENTATION DE L'INTERFACE - L'EXPLORATEUR DE SOLUTION

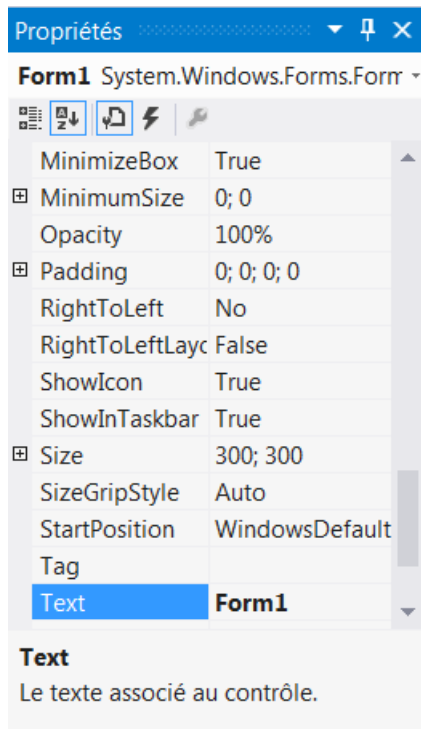
Le concept de solution est propre à notre IDE et est composée d'un ou plusieurs projets.

Le gros avantage de travailler avec une solution est la facilité de faire références aux différents projets la composant.

Il est possible de spécifier le projet de démarrage via les propriétés de la solution.



PRÉSENTATION DE L'INTERFACE - LES PROPRIÉTÉS



Affiche les propriétés de l'objet actuellement sélectionné.

S'il s'agit d'un élément graphique, nous pouvons accéder à ses propriétés et à ses événements.

Nous pouvons les trier par ordre alphabétique ou regroupé par fonctionnalité.

PRÉSENTATION DE L'INTERFACE - LE MULTI-ÉCRANS

Dans les versions antérieures à Visual Studio 2010, il nous était impossible de déplacer la fenêtre d'affichage de nos documents et designers que ce soit à l'intérieur ou à l'extérieur de l'environnement de Visual Studio. Cette absence de fonctionnalité pouvait vite apparaître comme frustrante pour nous, utilisateurs, étant donné qu'il nous était impossible de docker les autres fenêtres où on voulait dans l'environnement voir les disposer même à l'extérieur (comme exemples, les fenêtres Toolbox, Solution Explorer, Debug, Properties, etc.).

Depuis sa version, Visual Studio nous permet de déplacer vos fenêtres où vous le désirez que ce soit à l'intérieur de Visual Studio qu'à l'extérieur et même sur un autre écran.

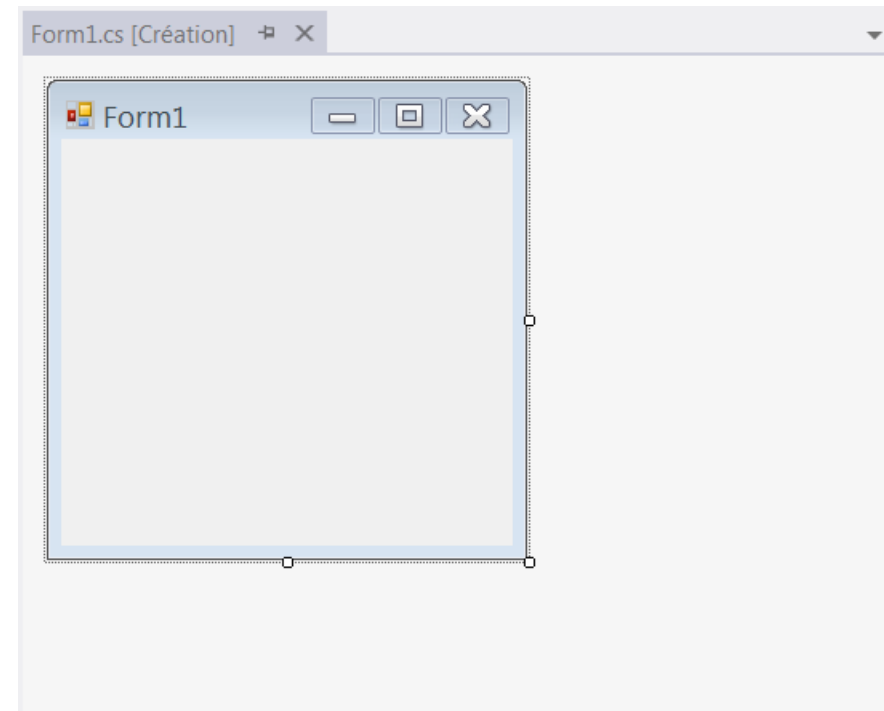
En plus de cela, Visual Studio sauvegarde votre configuration afin d'afficher la même disposition des fenêtres lorsque vous rouvrirez votre projet.

PRÉSENTATION DE L'INTERFACE - LA ZONE DESIGN/CODE

La zone design est là pour représenter notre application graphiquement qu'il s'agisse d'une application Windows Forms, WPF ou ASP .Net.

Nous avons aussi la possibilité de faire du glissé/déposé depuis la boîte à outils, de sélectionner les contrôles afin d'en afficher ses propriétés et ses événements.

À tout moment nous pourrons switcher de l'affichage graphique à l'affichage code en appuyant du « F7 » et « Maj » + « F7 » pour revenir au mode graphique.



Liste des objets

Possibilité de réduire
certaine partie de code

```
Form1.cs [X]
WindowsFormsApplication1.Form1
Form1()

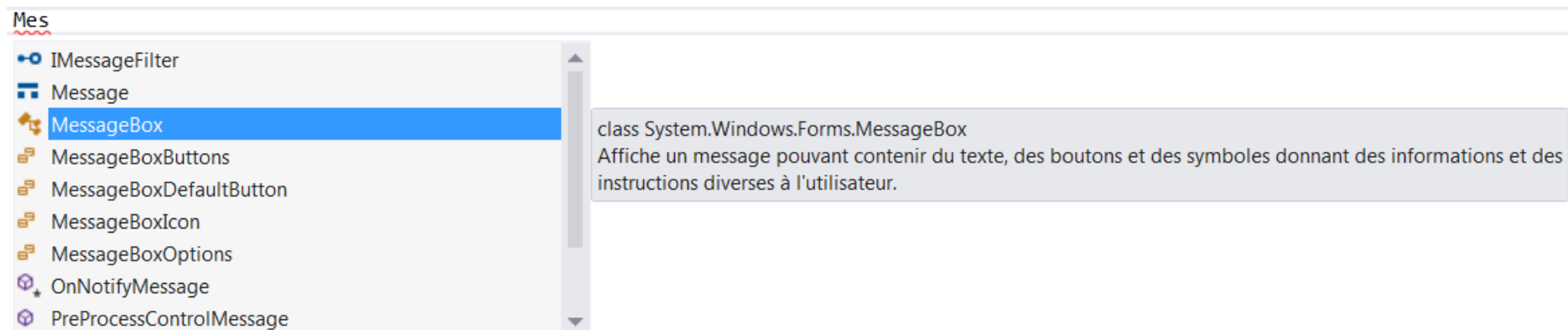
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

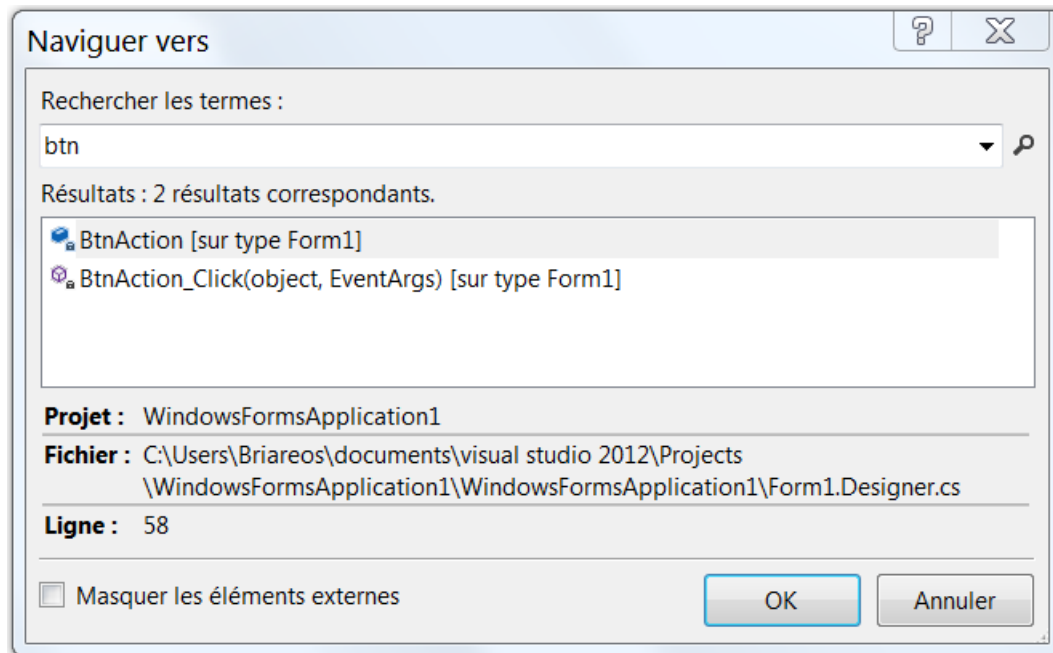
Les méthodes de l'objet

LA GESTION DU CODE - L'INTELLISENSE

L'IntelliSense de Visual Studio a été grandement améliorée afin de répondre au mieux à nos demandes quotidiennes. Lorsque nous tapons du code dans votre éditeur de code et que par exemple vous voulez atteindre un objet, vous n'avez qu'à écrire les premières lettres et l'IntelliSense va vous proposer tous les éléments accessibles qui commencent ou contiennent par ce que nous avons entré.



LA GESTION DU CODE - NAVIGATE TO



Visual Studio nous propose une fenêtre de recherche « CTRL + , » intelligente nous proposant en direct tous les résultats correspondants dans notre solution en spécifiant pour chaque résultat son type (Variable membre, constructeur, méthode, propriété, événement, etc.) et son emplacement.

LA GESTION DU CODE - SURBRILLANCE

Lorsque vous êtes dans le code d'un fichier, il n'est pas toujours évident de repérer toutes les références à une méthode ou une variable.

A présent, lorsque vous sélectionnez un élément de votre code, toutes les références associées sont surlignées.

```
private void BtnAction_MouseEnter(object sender, EventArgs e)
{
    AfficherStatus("La souris survole le bouton");
}

private void BtnAction_MouseLeave(object sender, EventArgs e)
{
    AfficherStatus("La souris ne survole pas le bouton");
}

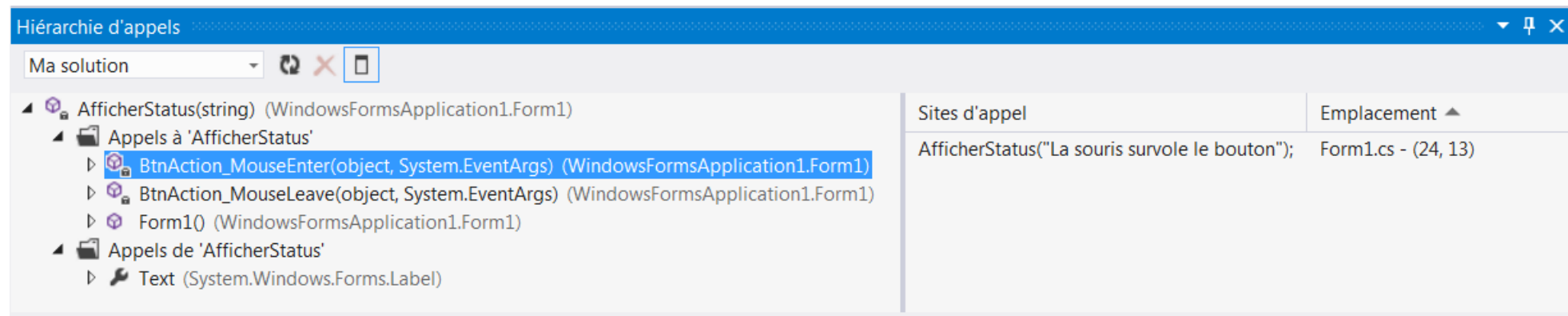
private void AfficherStatus(string Status)
{
    label1.Text = Status;
}
```


LA GESTION DU CODE - HIÉRARCHIE D'APPEL

Un problème récurrent lorsque nous arrivons à des projets de grandes envergures avec des centaines de classes et de nombreux projets dans la solution, c'est de déterminer quels sont les appels effectués vers un élément (méthodes, propriétés, variables membre, etc.) de notre code.

Bien que nous ayons notre « Find All References », il faut avouer qu'il n'offre pas une vue globale et claire dans le résultat.

Maintenant, nous pouvons connaître la hiérarchie des appels vers un élément du code. Pour cela, il suffit de se placer par exemple sur une méthode, de faire clique-droit et choisir l'option « View Call Hierarchy » (« CTRL K T »).



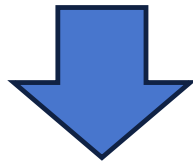
LA GESTION DU CODE - GÉNÉRATION AUTOMATIQUE

Visual Studio vous permet d'utiliser des classes, des méthodes, des propriétés ou des variables membres avant de les avoir définis.

```
int x = Addition(5, 7);
```



Générer un stub de méthode pour 'Addition' dans 'WindowsFormsApplication1.Form1'



```
private int Addition(int p1, int p2)
{
    throw new NotImplementedException();
}
```

```
MaVariable = "Exemple";
```



Générer le stub de propriété pour 'MaVariable' dans 'WindowsFormsApplication1.Form1'

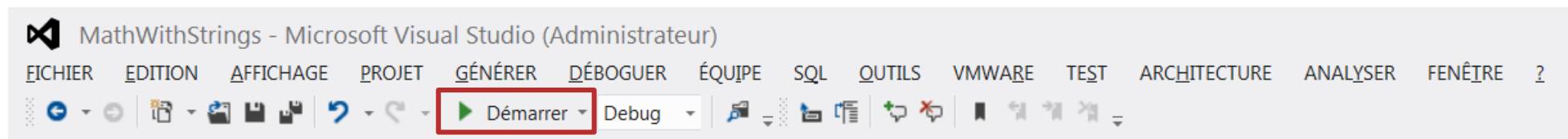
Générer le stub de champ pour 'MaVariable' dans 'WindowsFormsApplication1.Form1'



```
public string MaVariable { get; set; }
```

DÉBOGUER SON APPLICATION

Bien entendu, un de nos rôles majeurs sera de déboguer notre application pour cela nous appuierons sur « F5 » afin de lancer le débogage ou en appuyant sur la flèche « Start » de notre menu.



F5 : Exécution en mode débogage
CTRL + F5 : Exécution sans débogage
F11 : Exécution en mode débogage pas à pas

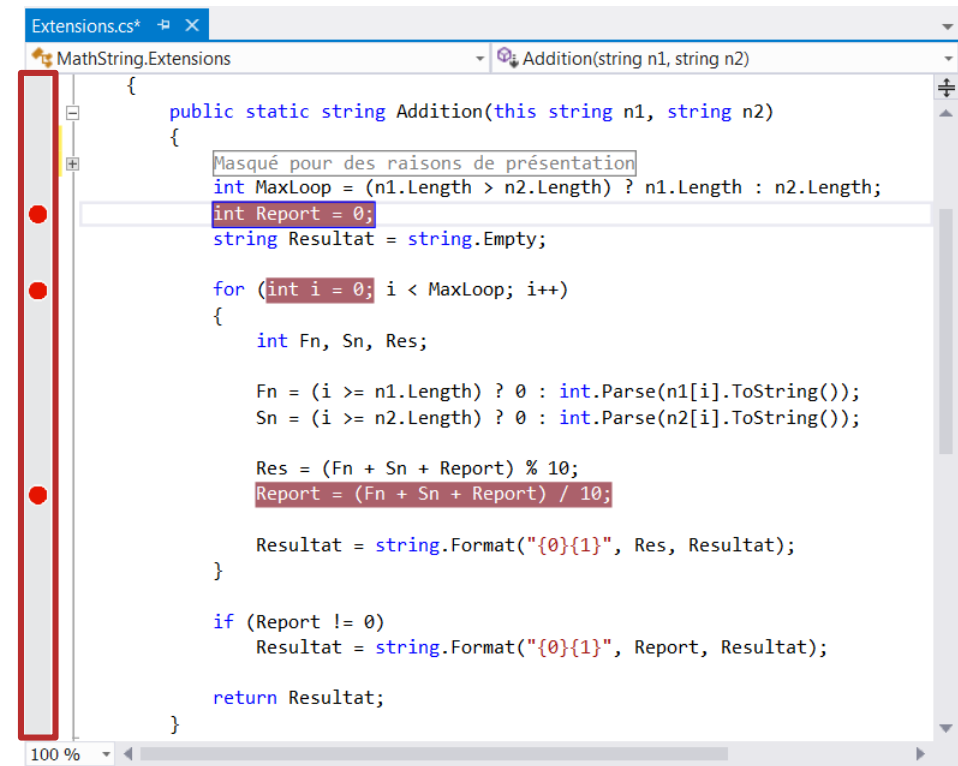
DÉBOGUER SON APPLICATION - LES BREAKPOINTS

Les breakpoints se placent simplement, sur la ligne où on désire arrêter le débogage, en cliquant dans la bande grise vertical.

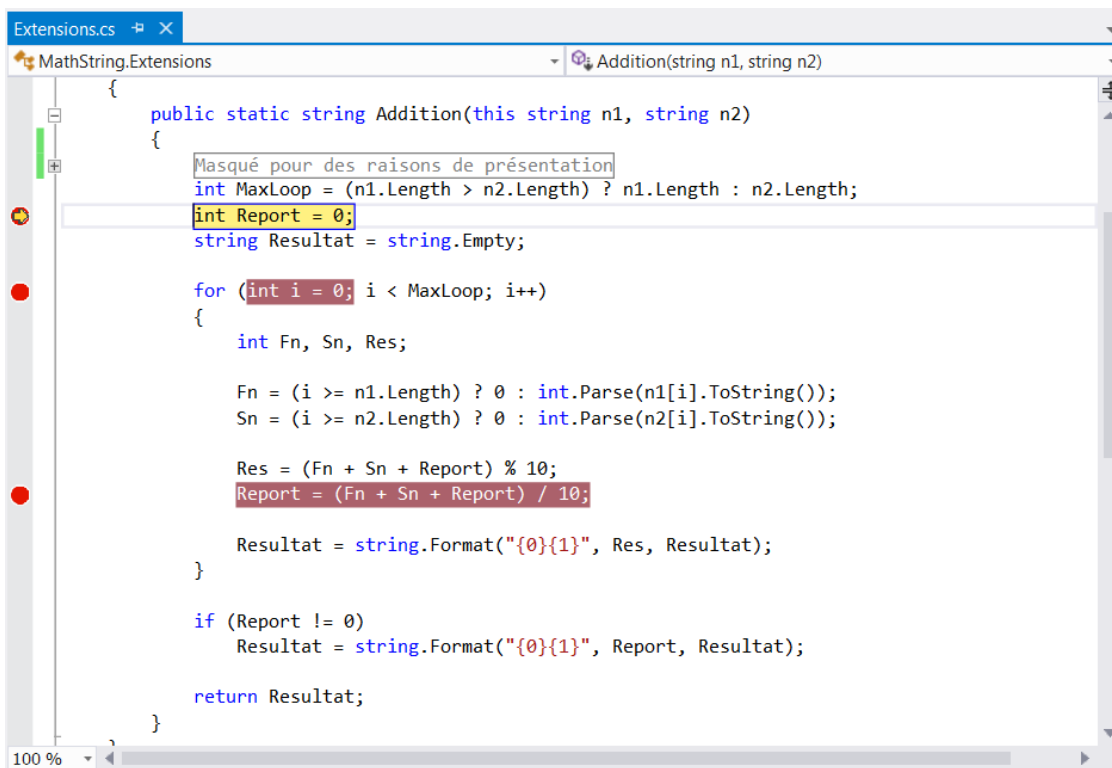
Par défaut, chaque fois que la ligne de code sera atteinte, le compilateur attendra notre décision.

En appuyant sur « F5 », il continuera jusqu'au prochain breakpoint

En appuyant sur « F11 », il continuera pas à pas.



DÉBOGUEUR SON APPLICATION - LES BREAKPOINTS



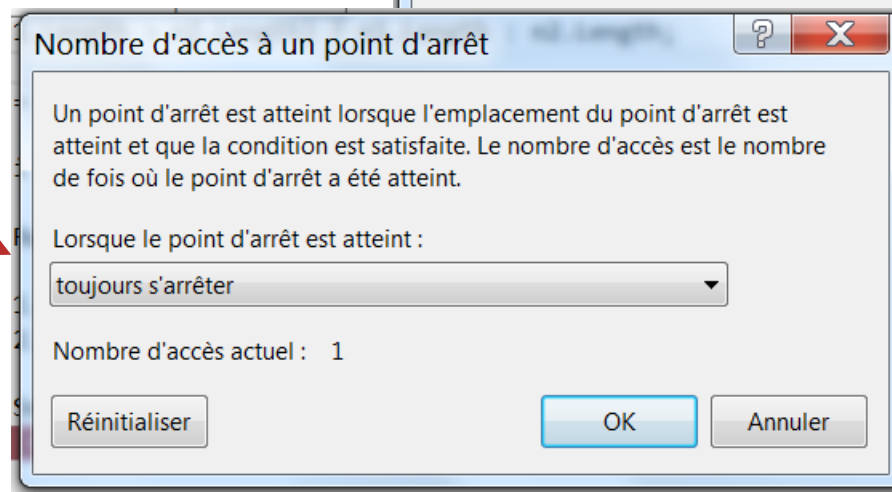
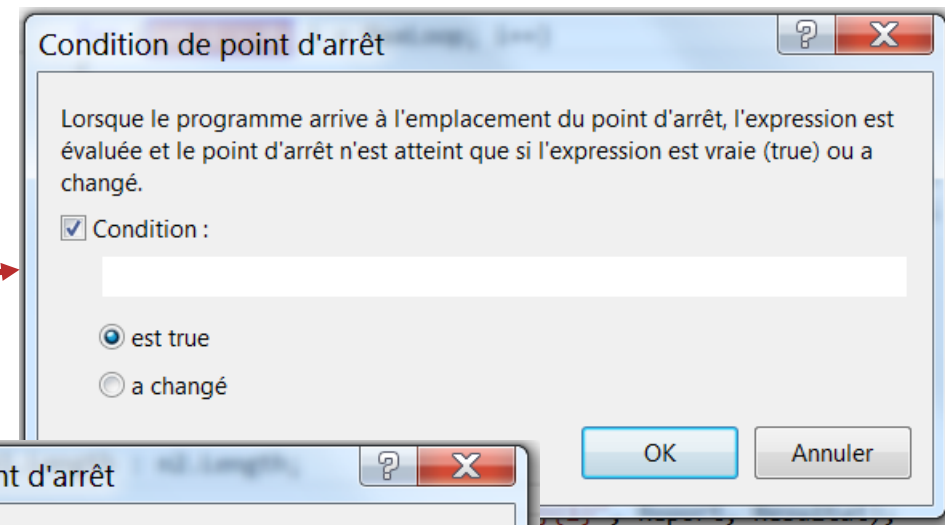
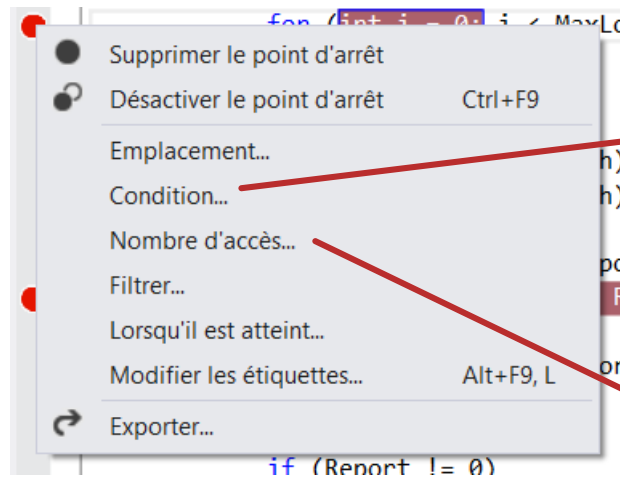
En mode débogage, lorsque le programme s'arrête sur un breakpoint, cela est représenté par une flèche jaune.

Cette flèche jaune signale quelle sera la prochaine action exécutée par notre programme.

En cliquant et en déplaçant cette flèche nous pouvons revenir en arrière ou passer les actions que nous ne voudrions pas exécuter dans notre test.

DÉBOGUEUR SON APPLICATION - LES BREAKPOINTS













Enfin chaque breakpoint est fourni d'un menu contextuel nous permettant de paramétrer nos breakpoints.



DÉBOGUER SON APPLICATION - VARIABLES LOCALES

Lorsque nous déboguons, nous avons toujours accès aux variables locales qui nous donne en temps réel la valeur de ces dernières et leur type.

Lorsqu'une de ces variables change de valeur, la valeur de celle-ci est indiquée en rouge.



Variables locales			▼ ↑ ×
Nom	Valeur	Type	
 n1	"987654321987654321987654321987654321987654321"	string	🔍 ▼
 n2	"321987654321987654321987654321987654321987654"	string	🔍 ▼
 Fn	8	int	
 Sn	2	int	
 Res	2	int	
 i	1	int	
 RegularExpression	"^\\d+\$"	string	🔍 ▼
  sb2	{321987654321987654321987654321987654321987654}	System	
 MaxLoop	45	int	
 Report	1	int	
 Resultat	"2"	string	🔍 ▼

DÉBOGUER SON APPLICATION - LES ESPIONS

Il arrive cependant que les variables locales ne soient pas suffisantes à notre débogage, tester le résultat d'une expression booléenne telle que « i plus petit que 5 par exemple ».

Par conséquent nous pourrons à tout moment ajouter des espions afin de connaître ces informations en temps réel.

Pour cela, sélectionnons l'expression à surveiller, clique droit « Add Watch ».

Espion 1			▼	📌	✕
Nom	Valeur	Type			
 i < MaxLoop	true	bool			
 i >= n1.Length	false	bool			
 i >= n2.Length	false	bool			
 (Fn + Sn + Report) / 10	1	int			
 (Fn + Sn + Report) % 10	1	int			

DÉBOGUER SON APPLICATION - L'INTELLITRACE

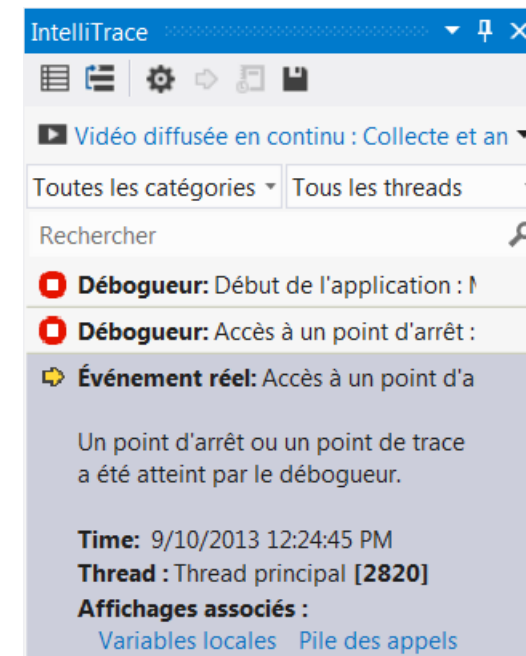
L'IntelliTrace a été introduit par Visual Studio 2010 Ultimate. Grâce à lui, nous pourrions connaître l'état de votre application à un moment donné.

Cependant, lorsque nous nous trouvons à un endroit donné dans notre application :

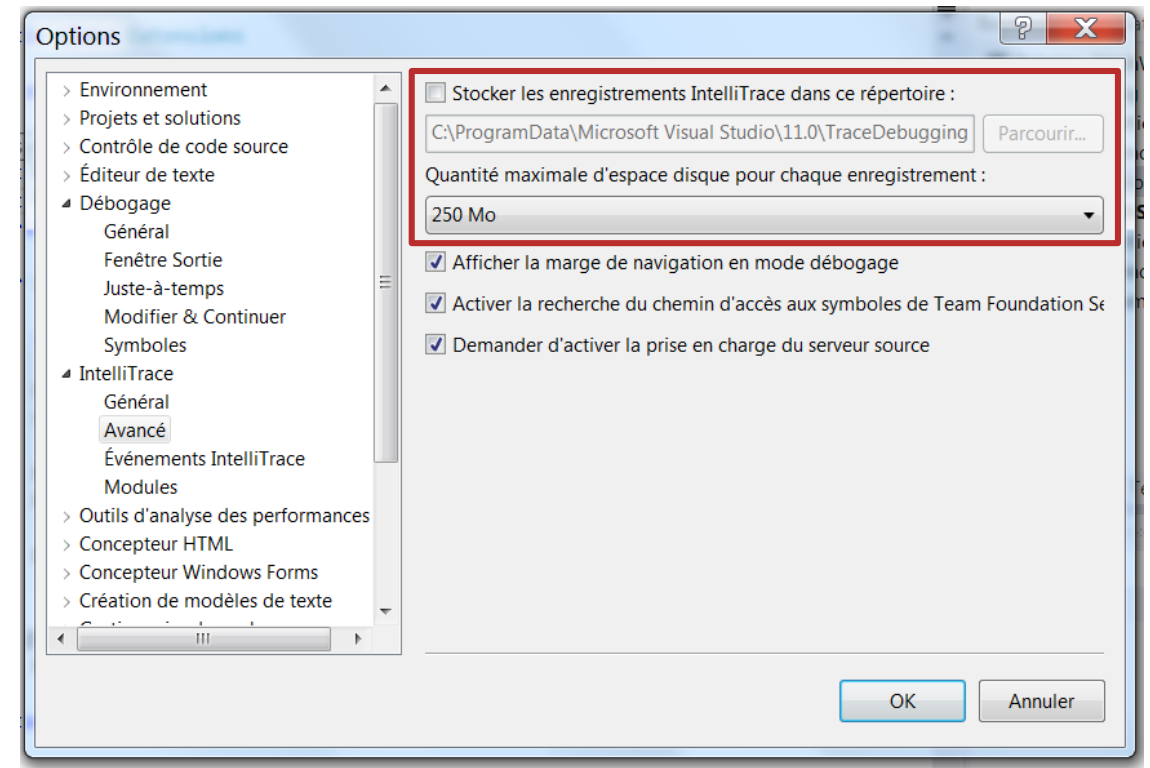
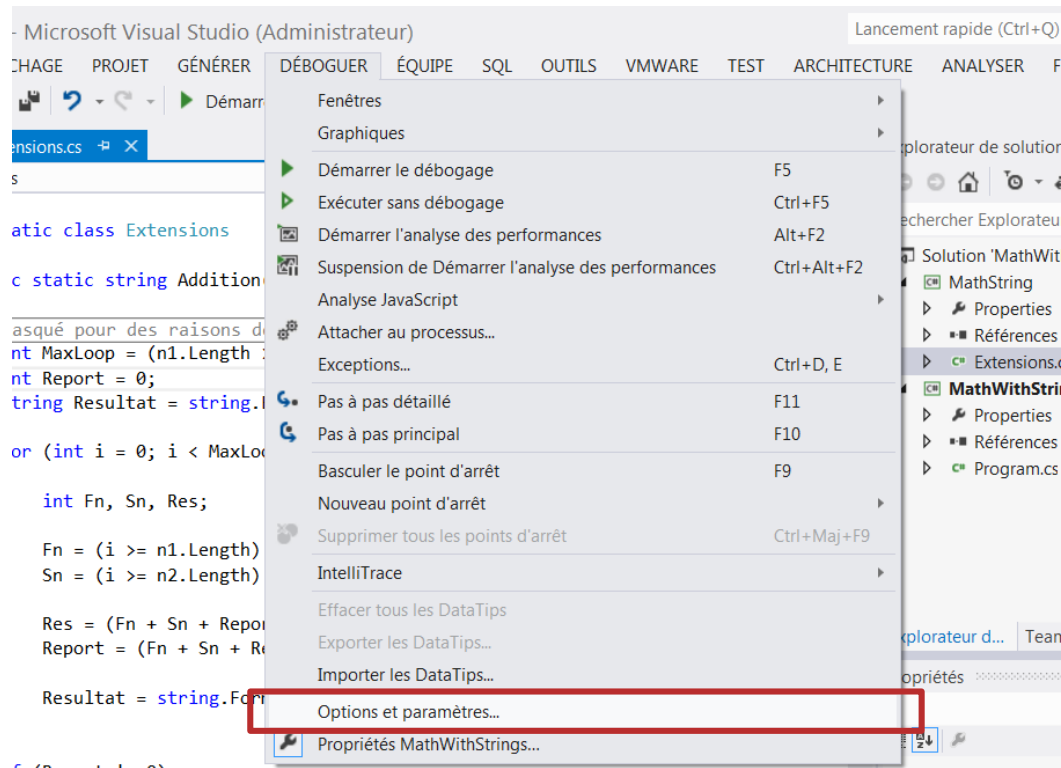
- Comment faisons-nous pour savoir ce qui s'est passé précédemment ? Par quelles méthodes notre code est-il passé ?
- Quel était l'état de nos objets ?

Le mode de débogage IntelliTrace va nous permettre de revoir les événements générés dans le passé mais aussi de connaître le contexte d'exécution au moment où chaque événement s'est produit.

De plus, en activant la fonctionnalité les options nous pouvons demander à Visual Studio d'enregistrer ces informations dans un fichier « .tdlog » dont nous pouvons spécifier la taille maximale.



DÉBOGUER SON APPLICATION - INTELLITRACE





AVANT DE COMMENCER

C# - LES FONDEMENTS

- Écrire dans la console
- Lire depuis la console
- Nettoyer la console

LES VARIABLES

ECRIRE DANS LA CONSOLE

```
using System;

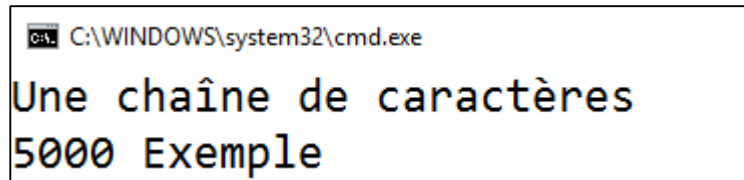
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            //Affiche le texte dans la console et va à la ligne
            Console.WriteLine("Une chaîne de caractères");
            //Affiche sur la même ligne
            Console.Write(5000);
            Console.Write(" Exemple");
            //pour un retour à la ligne sans rien afficher
            Console.WriteLine();
        }
    }
}
```

Avant d'entamer ce cours, il est important que nous puissions dialoguer avec notre environnement.

Puisqu'au début, cet environnement de travail sera principalement la console, nous allons voir les manières d'envoyer un message et de demander quelque chose à l'utilisateur ainsi que comment nettoyer la console.

Pour écrire quelque chose dans la console, nous allons utiliser les instructions « Console.Write(...) » et « Console.WriteLine(...) ».

La différence entre ces deux instructions est que l'instruction « Console.WriteLine([,,]) » retournera à la ligne.



```
C:\WINDOWS\system32\cmd.exe
Une chaîne de caractères
5000 Exemple
```

ECRIRE DANS LA CONSOLE

Parfois, nous aurons plusieurs informations à écrire sur une ligne.

Nous pourrions formater la ligne comme suit.

Nous devons obligatoirement commencer par {0} pour la première valeur.

```
using System;
```

```
namespace CoursCSharpFondements
```

```
{
```

```
    class Program
```

```
    {
```

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

```
        {
```

```
            Console.WriteLine("{0} fois {1} = {2} ou {3}", 5, 10, 50, "cinquante");
```

```
        }
```

```
    }
```

```
}
```

C:\WINDOWS\system32\cmd.exe

5 fois 10 = 50 ou cinquante



LIRE DEPUIS LA CONSOLE


```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            //Lit une ligne dans la console
            //et stocke l'information dans la variable s
            string s = Console.ReadLine();
            //Ecrire la variable s dans la Console
            Console.WriteLine(s);
            //pour un retour à la ligne sans rien afficher
            Console.ReadLine();
        }
    }
}
```

Pour lire une information depuis la console, c'est-à-dire de demander à l'utilisateur de fournir une information.

Nous allons, le plus souvent, utiliser l'instruction pour lire une ligne depuis la console. Cette instruction, « `Console.ReadLine()` », retournera une chaîne de caractère (« string »).

De plus, « `Console.ReadLine()` » sera souvent utilisée pour conserver la console ouverte à la fin de la méthode « Main ».

 c:\users\briar\documents\visual studio 2017\Projects\ConsoleApp1\Con:

```
C'est l'histoire de Toto...
C'est l'histoire de Toto...
```

■



Nous verrons les variables et le type « string »
un peu plus loin dans ce cours

NETTOYER LA CONSOLE

Pour nettoyer la console, rien de plus simple, il suffit d'utiliser l'instruction « `Console.Clear()` »

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Clear();
        }
    }
}
```




LES VARIABLES

C# - LES FONDEMENTS

- Déclaration
- Initialisation
- Champs et variables locales
- Portée des variables
- Constantes
- Les types « Nullable »

LES VARIABLES

DÉCLARATION

Dans tous nos programmes, nous serons amenés à stocker de façon temporaire des valeurs de différents types afin de les manipuler dans notre traitement, qu'il s'agisse de calculs mathématique, de texte à imprimer, de texte à envoyer sous forme d'email, etc...

La résultante est que nous devons déclarer des réceptacles capable de contenir ces informations. Ceux-ci sont appelés « Variable ».

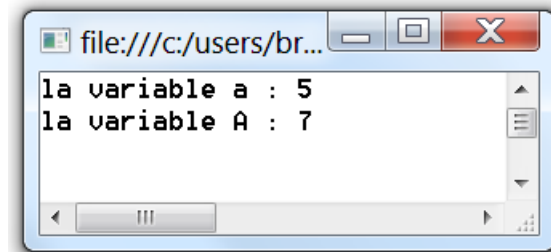
Le C# est un langage fortement typé, ce qui veut dire que nous devons donner un type (entier, booléen, chaîne de caractères) à nos variables lors de leur déclaration.

De plus, il est à noter que, le C# en plus d'être fortement typé est aussi « Case Sensitive », par conséquent il faudra se montrer prudent lors de la déclaration de nos variables.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 5;
            int A = 7;

            Console.WriteLine("la variable a : {0}", a);
            Console.WriteLine("la variable A : {0}", A);
            Console.ReadLine();
        }
    }
}
```



DÉCLARATION

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            //Exemples correctes
            int i;
            int x, y;

            //Exemples incorrectes
            int a; b;
            int c, bool d;

            Masqué pour des raisons de mise en page
        }
    }
}
```

Pour déclarer nos variables nous devons respecter une nomenclature particulière. Cette nomenclature autorise la déclaration de plusieurs variables du même type en les séparant par des virgules.

Type identifiant [=value][,identifiant2 [=value]][,...];

INITIALISATION

L'initialisation d'une variable consiste à lui fournir sa première valeur; ensuite nous parlerons d'affectation de valeurs.

L'initialisation peut se faire à deux endroits, soit à la déclaration de la variable, soit dans une méthode. Pour cela, nous utiliserons l'opérateur égal (`=`).

Le compilateur C# impose que toute variable doit être initialisée avant d'être utilisée sous peine de lever une erreur de compilation



```
int x = 9, y;  
Console.WriteLine(y);  
y = 10;
```

(variable locale) int y

Erreur :

Utilisation d'une variable locale non assignée 'y'

Masqué pour des ra

```
using System;  
  
namespace CoursCSharpFondements  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            //déclaration de la variable i  
            int i;  
            i = 5; //initialise la variable i avec la valeur 5  
  
            //initialisation à la création de la variable x  
            int x = 9, y;  
            y = 10;  
  
            Masqué pour des raisons de mise en page  
  
        }  
    }  
}
```

CHAMPS ET VARIABLES LOCALES

Nous avons également deux genres de variables, nous avons les variables membres et les variables locales.

Les variables membres sont définies au niveau de la classe, tandis que les variables locales sont définies dans un bloc d'instructions.

Rien ne nous choque dans cet exemple ?

Lors de la déclaration d'une variable membre, celle-ci est initialisée automatiquement. Un nombre recevra automatiquement la valeur 0. (default(T))

```
using System;

namespace CoursCSharpFondements
{
    class Exemple
    {
        //Déclaration d'une variable membre
        int X;

        public void MaMethode()
        {
            //Déclaration d'une variable locale
            int Y = 5;
            Console.WriteLine(X);
            Console.WriteLine(Y);
        }
    }
}
```

CHAMPS ET VARIABLES LOCALES

```
using System;

namespace CoursCSharpFondements
{
    class Exemple
    {
        //Déclaration de la variable membre Y
        int Y;

        public void MaMethode()
        {
            //La variable locale Y qui dissimule la variable membre Y
            int Y = 5;
            Console.WriteLine(Y); //affichera 5
        }
    }
}
```

De plus, il arrive que nous ayons une variable membre et une variable locale qui portent le même identifiant.

Le compilateur est tout à fait capable de faire la distinction entre une variable locale et une variable membre.

Dans ce cas, la variable locale « dissimule » la variable membre.

Comment atteindre la variable membre dans ce cas?

En utilisant le mot-clé « this » :

```
this.Y = 7;
Console.WriteLine(this.Y); //affichera 7
```

PORTABILITÉ DES VARIABLES

La portée d'une variable est la zone de code à partir de laquelle la variable est accessible, cette portabilité est définie par 2 règles :

- La variable membre (ou champs) est visible dans toute la classe qui la contient.
- Une variable locale est visible jusqu'à ce qu'une accolade fermante « } » indique la fin du bloc d'instructions ou de la méthode dans laquelle elle a été déclarée.

```
using System;

namespace CoursCSharpFondements
{
    class Exemple
    {
        //Déclaration de la variable membre X
        int X;

        public void MaMethode()
        {
            //Affectation de la valeur 7 à la variable X
            X = 7;
            //Déclaration de la variable locale Y et initialisation à la valeur 5
            int Y = 5;

            for (int i = 0; i < 10; i++)
            {
                //code
            } // fin de la portée de la variable i
        } // fin de la portée de la variable Y
    } // fin de la portée de la variable X
}
```


CONSTANTE

```
using System;

namespace CoursCSharpFondements
{
    class Exemple
    {
        const int X = 20;

        public void MaMethode()
        {
            Console.WriteLine(X);
            X = 10; //opération interdite;
        }
    }
}
```

Le fait d'ajouter le mot « const » comme préfixe à la déclaration d'une variable désigne celle-ci comme constante.

Une constante est une variable dont la valeur ne peut-être modifiée pendant sa durée de vie.

Les constantes ont les caractéristiques suivantes :

- Elles doivent être initialisées à la déclaration, et ne peuvent plus être modifiées.
- La valeur d'une constante doit être calculable à la compilation.
- Les constantes sont implicitement statiques, on ne peut donc les préfixer du mot-clé « static ».

Nous verrons le mot-clé « static » en détail dans la partie Orientée Objet du cours

LES TYPES « NULLABLE »

Dans le cadre des bases de données, les champs de tables peuvent être « null » indépendamment qu'il s'agisse d'un champs de type « nvarchar », « int », « datetime », « nchar », etc..

Or en .Net les types valeur primitifs ne peuvent pas avoir cette valeur « null ».

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = null;

            Console.ReadLine();
        }
    }
}
```

Impossible de convertir null en 'int', car il s'agit d'un type valeur qui n'autorise pas les valeurs null

LES TYPES « NULLABLE »

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Nullable<int> i = null;
            int? j = null; //raccourci pour Nullable

            Masqué pour des raisons de mise en page

            Console.ReadLine();
        }
    }
}
```

Pour résoudre ce problème, nous devons les définir comme étant « Nullable ».

Pour cela, nous avons deux possibilités soit utiliser « Nullable<T> » ou son raccourci d'écriture.



TYPES DE DONNÉES PRÉDÉFINIS

C# - LES FONDEMENTS

- Les types CTS
- Les types Valeur
- Le caractère
- Les types Références
- Les valeurs littérales
- La valeur littérale de type « string »
- La concaténation de « string »

TYPES DE DONNÉES PRÉDÉFINIS

TYPE CTS

Avant de voir les types prédéfinis en C#, nous devons nous attarder un peu sur les Types CTS. Le CTS (Common Type System) définit la façon dont les types sont déclarés, utilisés et gérés au sein du Common Language Runtime (CLR). Il constitue également une partie importante de la prise en charge de l'intégration interlangage (C#, VB.Net, C++ .NET).

Dans le .NET Framework, tous les types sont soit des types valeur, soit des types référence.

Les types CTS du .NET Framework sont regroupés en cinq catégories : « Classe », « Structure », « Énumération », « Interface », « Délégué ».

Le Framework .Net possède 14 types prédéfinis de type valeur et 2 de type référence.

Les classes, interfaces et délégués seront vu en détail dans la partie Orientée Objet

LES TYPES VALEUR

Les types valeur sont des types de données dont les objets sont représentés par la valeur réelle de l'objet. Si une instance d'un type valeur est assignée à une variable, cette variable reçoit une copie actualisée de la valeur.

Les 14 types valeur font partie de la catégorie « Structure » qui hérite de « System.ValueType ».

En mémoire, ceux-ci sont stockés sur la pile.

Exemple : si je déclare une variable de type « int », nous déclarons en fait une instance de type « Int32 ».

Dans ce cas, dois-je utiliser la déclaration C# ou CTS?

Type C#	Type CTS		Type C#	Type CTS
sbyte	SByte		byte	Byte
short	Int16		ushort	UInt16
int	Int32		uint	UInt32
long	Int64		ulong	UInt64
float	Single		char	Char
double	Double		bool	Boolean
decimal	Decimal		DateTime	DateTime

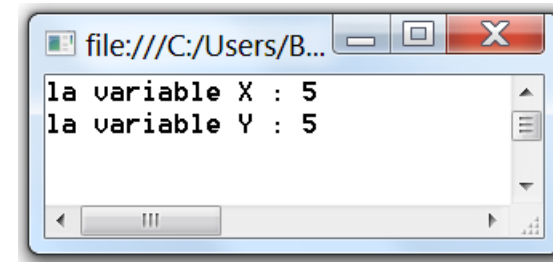
LES TYPES VALEUR

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int X = 5;
            Int32 Y = 5;

            Console.WriteLine("la variable X : {0}", X);
            Console.WriteLine("la variable Y : {0}", Y);
            Console.ReadLine();
        }
    }
}
```

Passons coté pratique, si je souhaite déclarer un entier j'ai donc deux possibilités soit utiliser le type C# ou utiliser le type CTS.



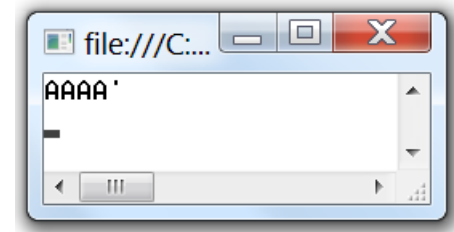
LE CARACTÈRE

Séquence	Caractère
\'	Simple guillemet
\"	Double guillemet
\\	Antislash
\0	Null
\a	Alerte
\b	Retour arrière
\f	Nouvelle page
\n	Retour à la ligne
\r	Retour chariot
\t	Caractère de tabulation
\v	Tabulation verticale

Bien que nous puissions spécifier un caractère sous forme littérale en utilisant les simples guillemets, nous pouvons aussi les spécifier en utilisant l'Unicode, l'hexadécimal, le code ASCII ou la séquence d'échappement.

```
char c1 = 'A';      //Littérale
char c2 = '\u0041'; //Unicode
char c3 = '\x0041'; //Hexadécimale
char c4 = (char)65; //Ascii
char c5 = '\\';     //Séquence d'échappement
```

```
Console.WriteLine("{0}{1}{2}{3}{4}", c1, c2, c3, c4, c5);
Console.ReadLine();
```



LES TYPES RÉFÉRENCES

Les types référence sont des types de données dont les objets sont représentés par une référence (identique à un pointeur) à la valeur réelle de l'objet. Si un type référence est assigné à une variable, celle-ci référence (ou pointe vers) la valeur d'origine. Aucune copie n'est effectuée.

Il n'existe que 2 types prédéfini de type référence, ceux-ci font partie de la catégorie « Classe » et sont stocké sur le tas.

La classe « object » est également appelée en C#, classe de base universelle, mais nous verrons ça en détail dans la partie « Orienté Objet »,

Type C#	Type CTS
string	String
object	Object

LES VALEURS LITTÉRALES

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            var v1 = 5;
            var v2 = 'a';
            var v3 = 3.14;
            var v4 = "Test";

            Console.WriteLine("{0}{4}{1}{4}{2}{4}{3}",
                v1.GetType(), v2.GetType(), v3.GetType(), v4.GetType(), '\n');

            Console.ReadLine();
        }
    }
}
```



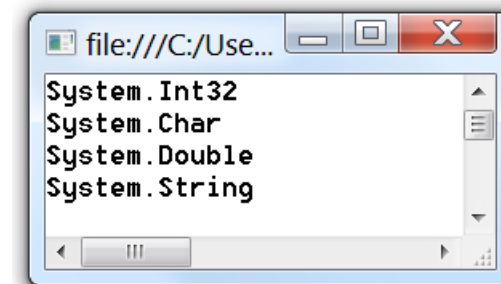
Le mot-clé « var » n'est utilisé qu'à titre d'exemple pédagogique. Il est déconseillé de l'utiliser dans un premier temps.

Les valeurs littérales sont des valeurs primitives constantes « codées en dur » dans notre programme.

Dans l'exemple, il s'agit des valeurs : 5, a, 3.14, Test et \n.

En C#, toute valeur est typée, y compris les valeurs littérales.

Dans ce cas, quelles sont les types de ces valeurs littérales?



LES VALEURS LITTÉRALES

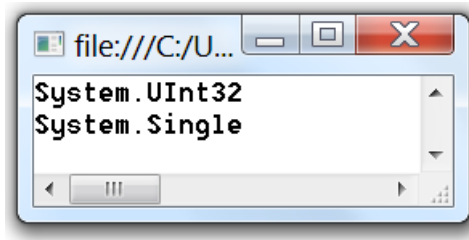
En utilisant les suffixes. En effet, certains types valeurs prédéfinis ont un suffixe spécifique pour la déclaration sous forme littérale.

Maintenant que nous connaissons les types par défaut des valeurs littérales, comment définir que 3.14 est de type « float » et non double ou que 5 est de type « uint » et non de type « int »?

```
static void Main(string[] args)
{
    var v1 = 5U;
    var v2 = 3.14F;

    Console.WriteLine("{0}{2}{1}",
        v1.GetType(), v2.GetType(), '\n');

    Console.ReadLine();
}
```



Type C#	Suffixe
long	L
uint	U
ulong	UL
float	F
double	D
decimal	M

LES VALEURS LITTÉRALES

Exercice : Quelles seront les valeurs et les types affichés

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            var i = 100 / 200 * 5;
            var j = 100F / 200 * 5;

            Console.WriteLine("{0} \t: {1}", i, i.GetType());
            Console.WriteLine("{0} \t: {1}", j, j.GetType());

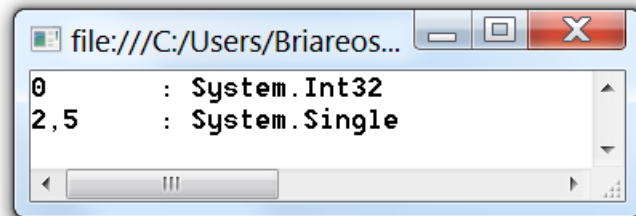
            Console.ReadLine();
        }
    }
}
```

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 100 / 200 * 5;
            float j = 100F / 200 * 5;

            Console.WriteLine("{0} \t: {1}", i, i.GetType());
            Console.WriteLine("{0} \t: {1}", j, j.GetType());

            Console.ReadLine();
        }
    }
}
```



LA VALEUR LITTÉRALE DE TYPE « STRING »

Le type « string » est le seul type référence à posséder une valeur littérale.

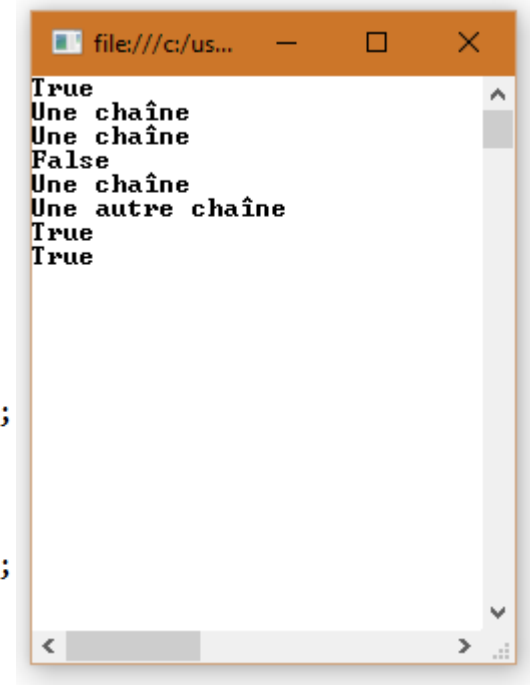
Pour rappel, une valeur littérale est une constante. Ce qui explique pourquoi le type « string » est le seul type référence à pouvoir être utilisé avec le mot clé « const ».

Lorsque nous affectons une valeur littérale de type « string » à une variable c'est donc la référence de cette constante qui est donnée qui écrase l'ancienne dans le tas.

```
static void Main(string[] args)
{
    string s1 = "Une chaîne";
    string s2 = s1;

    Console.WriteLine(ReferenceEquals(s1, s2));
    Console.WriteLine(s1);
    Console.WriteLine(s2);
    s2 = "Une autre chaîne";
    Console.WriteLine(ReferenceEquals(s1, s2));
    Console.WriteLine(s1);
    Console.WriteLine(s2);

    Console.WriteLine(ReferenceEquals(s1, "Une chaîne"));
    Console.WriteLine(ReferenceEquals("Une chaîne", "Une chaîne"));
    Console.ReadLine();
}
```



LA VALEUR LITTÉRALE DE TYPE « STRING »

```
using System;

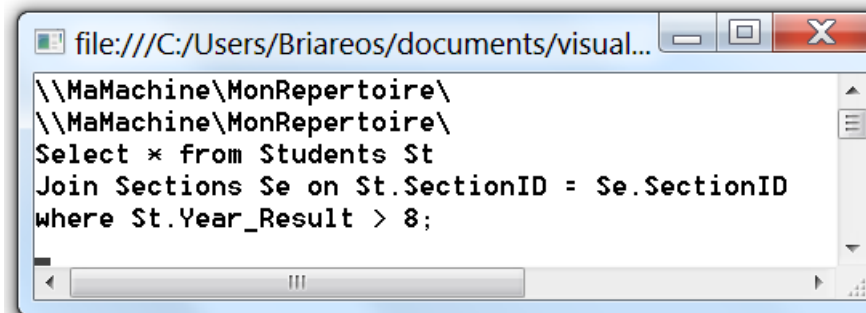
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string NetworkLink1 = "\\MaMachine\MonRepertoire\";
            string NetworkLink2 = @"\\MaMachine\MonRepertoire\";
            string Query =
@"Select * from Students St
Join Sections Se on St.SectionID = Se.SectionID
where St.Year_Result > 8;";

            Console.WriteLine(NetworkLink1);
            Console.WriteLine(NetworkLink2);
            Console.WriteLine(Query);

            Console.ReadLine();
        }
    }
}
```

La classe « string » gère également les caractères d'échappement mais dans certains cas les chaînes de caractères peuvent s'avérer être fastidieuses à déclarer.

Par conséquent nous pouvons la déclarer précédée d'un « @ », ce qui nous permet de ne pas gérer les caractères d'échappement et nous permet également de les déclarer sur plusieurs lignes (bien que peu utilisée, optant plus pour de la concaténation).

A screenshot of a Windows Explorer window. The address bar shows the file path: file:///C:/Users/Briareos/documents/visual... The main pane displays the contents of the file, which are the same C# code snippets shown in the previous blocks, including the string declarations and the SQL query using the @-string literal.

```
\\MaMachine\MonRepertoire\
\\MaMachine\MonRepertoire\
Select * from Students St
Join Sections Se on St.SectionID = Se.SectionID
where St.Year_Result > 8;
```

LA CONCATÉNATION DE « STRING »

Nous avons trois possibilités pour concaténer des chaînes :

- L'opérateur +
- La classe « StringBuilder »
- La méthode « string.Format »

```
using System;
using System.Text; // pour la classe StringBuilder

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            //L'opérateur +
            string Query = "Select * from Students St " +
                           "Join Sections Se on St.SectionID = Se.SectionID " +
                           "where St.Year_Result > 8;";

            //StringBuilder
            StringBuilder sb1 = new StringBuilder();
            sb1.Append("Select * from Students St ");
            sb1.Append("Join Sections Se on St.SectionID = Se.SectionID ");
            sb1.Append("where St.Year_Result > 8;");

            StringBuilder sb2 = new StringBuilder();
            sb2.AppendLine("Select * from Students St");
            sb2.AppendLine("Join Sections Se on St.SectionID = Se.SectionID");
            sb2.AppendLine("where St.Year_Result > 8;");

            Console.WriteLine(Query);
            Console.WriteLine(sb1.ToString());
            Console.WriteLine(sb2.ToString());
            Console.ReadLine();
        }
    }
}
```


LA CONCATÉNATION DE « STRING »

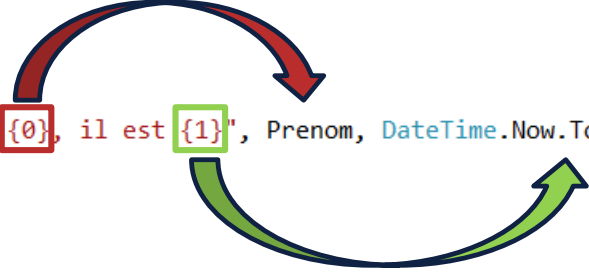
Le principe du « string.Format » est assez simple, nous définissons le format et là où nous souhaiterions ajouter des valeurs nous plaçons {x}, en commençant par {0}, ensuite nous mettons derrière ce format les valeurs à afficher séparées par des virgules.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string Prenom = "Michael";

            string Message = string.Format("Bonjour {0}, il est {1}", Prenom, DateTime.Now.ToShortTimeString());

            Console.WriteLine(Message);
            Console.ReadLine();
        }
    }
}
```





LES CONVERSIONS

C# - LES FONDEMENTS

- De type valeur à « string »
- De « string » à un type valeur
- Conversions implicites
- Conversions explicites
- Boxing et Unboxing

LES CONVERSIONS

DE TYPE VALEUR À « STRING »

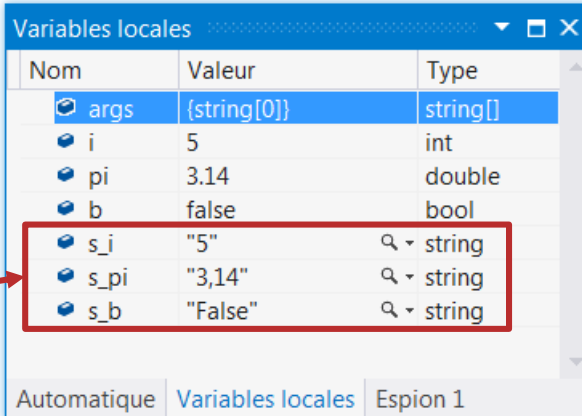
Tout élément en C# possède une méthode « ToString() », dans le cadre des types valeur, nous pourrions utiliser cette méthode pour transformer notre type valeur en type string.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            double pi = 3.14;
            bool b = false;

            string s_i = i.ToString();
            string s_pi = pi.ToString();
            string s_b = b.ToString();

            Console.ReadLine();
        }
    }
}
```



Nom	Valeur	Type
args	{string[0]}	string[]
i	5	int
pi	3.14	double
b	false	bool
s_i	"5"	string
s_pi	"3,14"	string
s_b	"False"	string

Automatique Variables locales Espion 1

DE « STRING » À UN TYPE VALEUR

Pour l'inverse, c'est-à-dire de string vers le type valeur, cela reste tout aussi facile. Cependant plusieurs solutions s'offrent à nous.

- La classe Convert
- La méthode Parse
- La méthode TryParse

Chacune de ces méthodes a ses avantages et inconvénient, mais la plus propres reste la méthode TryParse.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "5";
            string s2 = "3,14";
            string s3 = "AAA";

            //La classe Convert
            int i = Convert.ToInt32(s1);
            //La méthode Parse
            double pi = double.Parse(s2);
            //La méthode tryparse
            float f;
            bool b = float.TryParse(s3, out f);

            Console.ReadLine();
        }
    }
}
```

DE « STRING » À UN TYPE VALEUR - LA CLASSE « CONVERT »

```
using System;

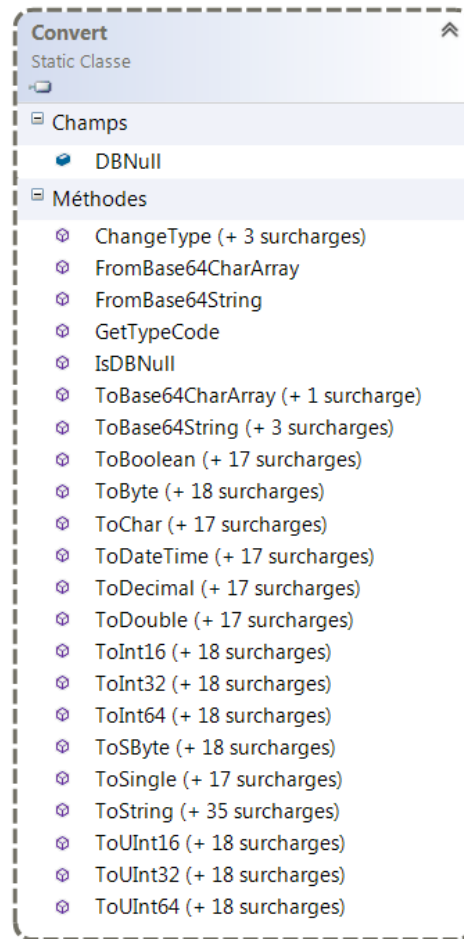
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "5";
            string s2 = "3,14";

            int i = Convert.ToInt32(s1);
            double pi = Convert.ToDouble(s2);

            Console.WriteLine(i);
            Console.WriteLine(pi);

            Console.ReadLine();
        }
    }
}
```

COGNITIC - COURS DE C#



La classe « Convert » propose différentes méthodes permettant de convertir vers un type donné.

Si elle ne parvient pas à faire la conversion, la méthode lèvera une erreur à l'exécution du programme,

De plus, elle permet la conversion de type vers type, en d'autre terme nous pourrions convertir des « double » en « entier », des « bool » en double, etc....

DE « STRING » À UN TYPE VALEUR - LA MÉTHODE « PARSE »

Tout type valeur possède les méthodes « Parse » et « TryParse », ces méthodes ne permettent que de lire une chaîne de caractères et de la convertir en fonction du type valeur choisi.

La méthode « Parse » reçoit en paramètre une « string » et tente de convertir cette chaîne dans le type spécifié.

Si elle n'y parvient pas, une erreur sera déclenchée à l'exécution.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "5";
            string s2 = "3,14";

            int i = int.Parse(s1);
            double pi = double.Parse(s2);

            Console.WriteLine(i);
            Console.WriteLine(pi);

            Console.ReadLine();
        }
    }
}
```

DE « STRING » À UN TYPE VALEUR - LA MÉTHODE « TRYPARSE »

La méthode « TryParse » quant à elle va recevoir en paramètre une « string » et un second paramètre du même type que celui du résultat de la conversion précédé du mot-clé « out ».

Si elle parvient à convertir, la méthode retournera « true » et la valeur sera stockée dans la 2^{ème} variable passée en paramètre.

Si pas, la méthode retournera « false », la valeur de « default(T) » sera assignée à la 2^{ème} variable passée en paramètre et aucune erreur ne sera déclenchée à l'exécution.



Le mot-clé « out » sera vu en détail dans la partie traitant les méthodes

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string s1 = "5";
            string s2 = "3,14";

            int i;
            bool b_i = int.TryParse(s1, out i);

            double pi;
            bool b_pi = double.TryParse(s2, out pi);

            Console.WriteLine(i);
            Console.WriteLine(pi);

            Console.ReadLine();
        }
    }
}
```


CONVERSION IMPLICITE

De	En
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

En dehors des conversions « de » et « vers » des « string », C# prend en charge différentes conversions automatiques, appelée conversion implicite,

Ces conversions implicites sont autorisée par le compilateur car celui-ci pourra garantir que ces conversions n'affecteront d'aucune manière leur valeur.

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            char c = 'a';
            ushort us = c;
            int i = 5;
            long l = i;
            double d = 3.14F;
            decimal dec = i;
            Console.ReadLine();
        }
    }
}
```

CONVERSION EXPLICITE

Dans les autres cas nous devons spécifier la conversion de manière explicite.

Pour ce faire, imaginons la situation suivante, on nous demande de copier des fichiers d'un répertoire à un autre et afficher le pourcentage effectué par rapport au traitement complet.

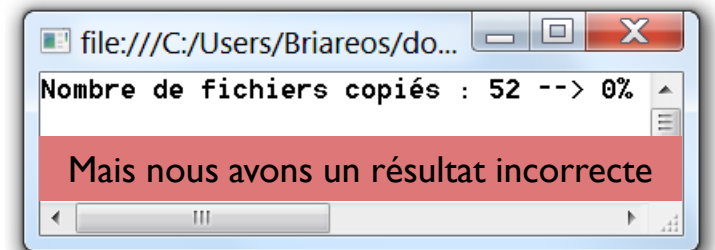
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int NbrFichiersTotal = 208;
            int NbrFichiersCopies = 52;

            int Pourcent = 100 / NbrFichiersTotal * NbrFichiersCopies;

            Console.WriteLine("Nombre de fichiers copiés : {0} --> {1}%", NbrFichiersCopies, Pourcent);

            Console.ReadLine();
        }
    }
}
```

Le calcul consiste donc à faire :
 $100 / \text{Nombre de fichier total} * \text{Nombre de fichiers déjà copiés}$



CONVERSION EXPLICITE

Notre solution consisterait à déclarer la valeur littéral « 100 » en « float ».

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int NbrFichiersTotal = 208;
            int NbrFichiersCopies = 52;

            int Pourcent = 100F / NbrFichiersTotal * NbrFichiersCopies;

            Console.WriteLine(Pourcent);
            Console.ReadLine();
        }
    }
}
```

Maintenant se pose un autre problème, effectivement il n'existe pas de conversion implicite de « float » en « int » en raison du risque de perte de donnée ($0,5 > 0$ = perte de précision)

(variable locale) int NbrFichiersTotal

Erreur :

Impossible de convertir implicitement le type 'float' en 'int'. Une conversion explicite existe (un cast est-il manquant ?)

CONVERSION EXPLICITE

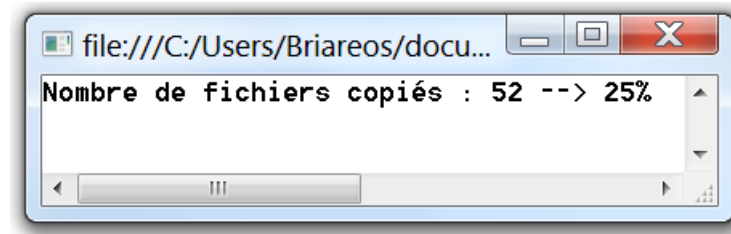
Nous devons spécifier au compilateur que nous voulons transtyper notre « float » en « int » et que par conséquent nous prenons la responsabilité en cas de perte de données (précision).

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int NbrFichiersTotal = 208;
            int NbrFichiersCopies = 52;

            int Pourcent = (int)(100F / NbrFichiersTotal * NbrFichiersCopies);

            Console.WriteLine("Nombre de fichiers copiés : {0} --> {1}%", NbrFichiersCopies, Pourcent);

            Console.ReadLine();
        }
    }
}
```



BOXING & UNBOXING

Le principe de « Boxing » consiste à emballer un type valeur (stocké sur la pile) dans un type référence (stocké sur le tas).

Le principe d'« Unboxing » quant à lui fait l'inverse, nous récupérons la valeur sur le tas pour la remettre sur la pile.



Le « Boxing » est toujours une conversion implicite.
L'« Unboxing » est toujours une conversion explicite.

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            object o = i; //Boxing (Implicite)
            int j = (int)o; //Unboxing (Explicite)

            Console.ReadLine();
        }
    }
}
```



EXERCICES

EXERCICES (MAXIMUM 10')

En utilisant la méthode « `Console.ReadLine()` »

- Demander à l'utilisateur d'encoder 2 nombres (int) et d'en faire l'addition, la conversion devra utiliser la méthode « `int.Parse()` »
- Demander à l'utilisateur d'encoder 2 nombres (int) et d'en faire l'addition, la conversion devra utiliser la méthode « `int.TryParse()` »



LES INSTRUCTIONS CONDITIONNELLES

C# - LES FONDLEMENTS

- Le bloc « if ... else »
- Le bloc « if ... else if ... else »
- Le bloc « switch »

LES INSTRUCTIONS CONDITIONNELLES

LE BLOC « IF ... ELSE »

Les instructions conditionnelles permettent d'insérer des branchements dans le code selon que certaines conditions soient satisfaites ou non, ou bien en fonction de la valeur d'une expression.

Le « C# » a deux constructions pour les branchements dans le code : l'instruction « if », qui permet de tester si une condition est satisfaite ou non, et l'instruction « switch », qui permet de comparer une expression avec des valeurs différentes.

La syntaxe et le fonctionnement de l'instruction « if » est relativement simple à prendre en main. La condition doit renvoyer un Type de valeur « bool » et peut être le résultat d'un test sur des valeurs, d'une variable, le retour d'une méthode ou la combinaison de ces points.



Le bloc « else » est facultatif.

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            bool condition = true;

            if (condition)
            {
                //Code à exécuter si la condition renvoie true
            }
            else
            {
                //Code à exécuter si la condition renvoie false
            }

            Console.ReadLine();
        }
    }
}
```

LE BLOC « IF ... ELSE »

Les opérateurs sur les valeurs

==	Égale	i == 5
!=	Différent	i != 5
<	Plus petit que	i < 5
<=	Plus petit ou égal	i <= 5
>	Plus grand que	i > 5
>=	Plus grand ou égal	i >= 5

Les opérateurs logiques

!	Négation
&&	Et
	Ou
^	Ou Exclusif

LE BLOC « IF ... ELSE »

Exemple :

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 4;
            bool condition = false;

            if (condition || i < 5)
            {
                //Code à exécuter si la condition renvoie true
            }
            else
            {
                //Code à exécuter si la condition renvoie false
            }

            Console.ReadLine();
        }
    }
}
```

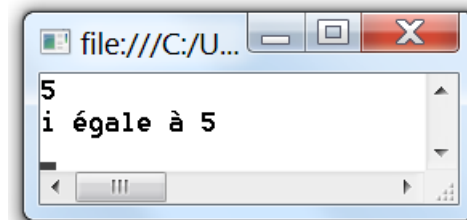
LE BLOC « IF ... ELSE IF ... ELSE »

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;

            if (int.TryParse(Console.ReadLine(), out i))
            {
                if (i > 5)
                {
                    Console.WriteLine("i plus grand que 5");
                }
                else if (i < 5)
                {
                    Console.WriteLine("i plus petit que 5");
                }
                else
                {
                    Console.WriteLine("i égale à 5");
                }
            }

            Console.ReadLine();
        }
    }
}
```

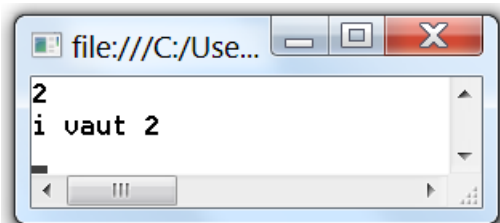
Il arrive, cependant, que nous ayons plusieurs conditions et que nous ayons un traitement spécifique pour chacune d'entre elles. Dans ce cas nous devons utiliser une variante du bloc « if ... else ».



LE BLOC « SWITCH »

L'instruction « switch » est apte à sélectionner un branchement de l'exécution à partir d'un jeu de branchement mutuellement exclusifs.

Cependant, lorsqu'un « case » contient du code, ce dernier doit se terminer par le mot clé « break ».



```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;

            if (int.TryParse(Console.ReadLine(), out i))
            {
                switch(i)
                {
                    case 1:
                        Console.WriteLine("i vaut 1");
                        break;
                    case 2:
                        Console.WriteLine("i vaut 2");
                        break;
                    case 3:
                        Console.WriteLine("i vaut 3");
                        break;
                    default :
                        Console.WriteLine("i ne vaut ni 1, ni 2, ni 3");
                        break;
                }
            }

            Console.ReadLine();
        }
    }
}
```

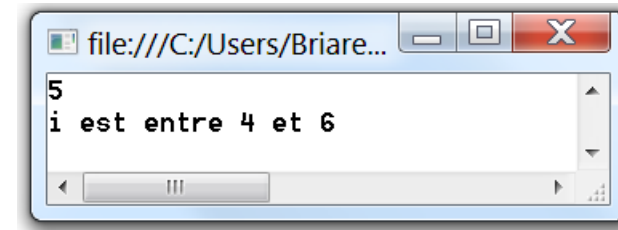
LE BLOC « SWITCH »

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;

            if (int.TryParse(Console.ReadLine(), out i))
            {
                switch(i)
                {
                    case 1:
                    case 2:
                    case 3:
                        Console.WriteLine("i est entre 1 et 3");
                        break;
                    case 4:
                    case 5:
                    case 6:
                        Console.WriteLine("i est entre 4 et 6");
                        break;
                    default :
                        Console.WriteLine("i n'est pas repris dans la plage de valeurs");
                        break;
                }
            }

            Console.ReadLine();
        }
    }
}
```

Nous pouvons à tout moment regrouper les cases ensemble pour n'écrire qu'une fois le code, mais attention, la règle précédente s'applique toujours.





EXERCICES

EXERCICES (MAXIMUM 10')

- Demander à l'utilisateur d'encoder l nombre (int), si la somme des deux moitiés de celui-ci donne le nombre, afficher « le nombre est paire » sinon « le nombre est impaire ».



LES OPÉRATEURS

C# - LES FONDEMENTS

- Les groupes d'opérateurs
- Les opérateurs d'affectations et raccourcis
- Pré et Post incrémentation (décrémentation)
- L'opérateur ternaire
- L'opérateur « is »
- L'opérateur « typeof »
- L'opérateur « checked »
- L'opérateur « unchecked »
- Ordre de priorité des opérateurs

LES OPÉRATEURS

LES GROUPES D'OPÉRATEURS

C# prend en charge différents opérateurs, tous regroupés en plusieurs catégories :

- Arithmétique
- Logique
- Concaténation de chaînes
- Incrément et décrétement
- Déplacement de bits
- Comparaison
- Affectation
- Accès aux membres
- Indexation
- Transtypage
- Conditionnel
- Création d'objets
- Informations de type
- Contrôle d'exception de dépassement de pile
- Indirection et adresses

Quatre d'entre eux ne sont accessibles qu'en mode « unsafe ».

Ces quatre opérateurs sont utilisés dans la gestion des pointeurs et pour connaître la taille des objets en mémoire. Par conséquent, il ne sont utilisés que très rarement dans l'environnement .Net.

Il s'agit des opérateurs :

- sizeof
- *
- ->
- &

Le code « unsafe » ne sera pas vu à ce cours.

LES GROUPES D'OPÉRATEURS

Catégorie d'opérateurs	Opérateur		Catégorie d'opérateurs	Opérateur
Arithmétique	+, -, *, /, %		Indexation	[]
Logique	&, , ^, ~, &&, , !		Transtypage	()
Concaténation de chaînes	+		Conditionnel	? :
Incrément et décrément	++, --		Création d'objets	new
Déplacement de bits	<<, >>		Informations de type	sizeof, is, typeof, as
Comparaison	==, !=, <, >, <=, >=		Contrôle d'exception de dépassement de pile	checked, unchecked
Affectation	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=		Indirection et adresses*	*, ->, &
Accès aux membres	.			

LES OPÉRATEURS D'AFFECTATIONS ET RACCOURCIS

Les opérateurs d'affectations possèdent des raccourcis d'écriture.

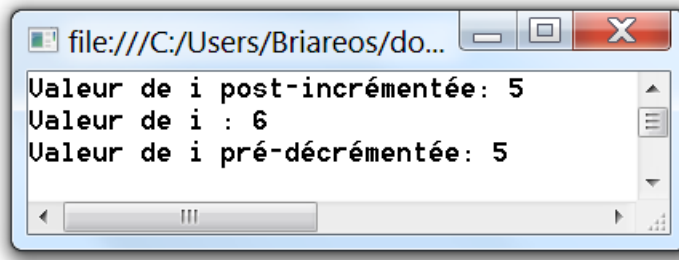
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            int j = i++;
            j += i;

            int k = 7;
            //multiplie k par 2
            k <<= 1;
            //divise k par 2
            k >>= 1;
            Console.ReadLine();
        }
    }
}
```

Raccourci	Equivalence		Raccourci	Equivalence
X++, ++X	X = X + 1		X %= y	X = X % y
X--, --X	X = X - 1		X >>= y	X = X >> y
X += y	X = X + y		X <<= y	X = X << y
X -= y	X = X - y		X &= y	X = X & y
X *= y	X = X * y		X = y	X = X y
X /= y	X = X / y		X ^= y	X = X ^ y

PRÉ ET POST INCRÉMENTATION (DÉCRÉMENTATION)

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            Console.WriteLine("Valeur de i post-incrémentée: {0}", i++);
            Console.WriteLine("Valeur de i : {0}", i);
            Console.WriteLine("Valeur de i pré-décrémentée: {0}", --i);
            Console.ReadLine();
        }
    }
}
```



La différence entre la pré et la post incrémentation ou décrémentation est définie sur le moment où la variable va changer de valeur.

Dans le cadre de la pré-incrémentation ($++x$) ou de la pré-décrémentation ($--x$), x changera de valeur avant d'avoir été utilisée.

Dans le cadre de la post-incrémentation ($x++$) ou de la post-décrémentation ($x--$), x changera de valeur après avoir été utilisée.

L'OPÉRATEUR TERNAIRE

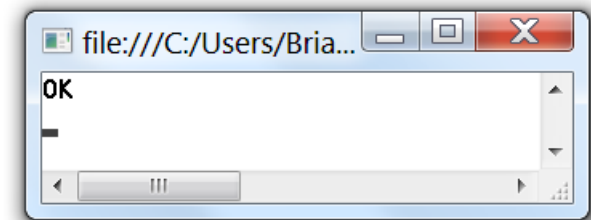
L'opérateur ternaire « ? » est une forme abrégée de la construction « if ... else » optimisée pour l'affectation de valeur. Il tient son nom du fait qu'il implique 3 opérandes :

- La condition
- La valeur si vrai
- La valeur si faux

Opérateur ternaire :
= (condition) ? Valeur si vrai : Valeur si faux;

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;

            string result = (i == 5) ? "OK" : "KO";
            Console.WriteLine(result);
            Console.ReadLine();
        }
    }
}
```



L'OPÉRATEUR COALESCE

L'opérateur ?? est appelé l'opérateur coalesce. Ce dernier retourne l'opérande de partie gauche si ce dernier n'est pas « null » et dans le cas contraire, il retourne l'opérande de partie droite.

Cet opérateur peut-être répété.

```
static void Main(string[] args)
{
    string s = null, t = null, u = null;

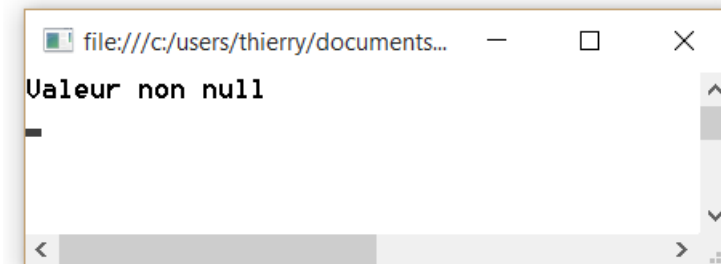
    string result = s ?? t ?? u ?? "Valeur non null";

    Console.WriteLine(result);
    Console.ReadLine();
}
```

```
static void Main(string[] args)
{
    string s = null;

    string result = s ?? "Valeur non null";

    Console.WriteLine(result);
    Console.ReadLine();
}
```



L'OPÉRATEUR « IS »

L'opérateur « is » permet de contrôler si une variable est compatible avec un type donné.

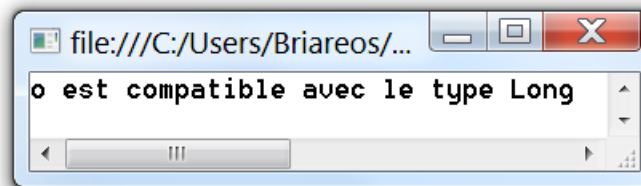
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            object o = 5L; //Boxing

            if (o is long)
            {
                Console.WriteLine("o est compatible avec le type Long");
            }

            Console.ReadLine();
        }
    }
}
```

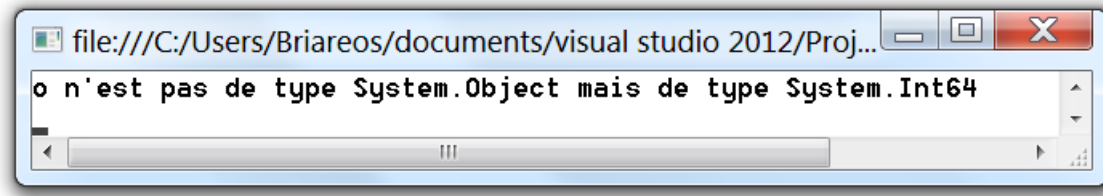
is :

Expression is Type;



L'OPÉRATEUR « TYPEOF »

L'opérateur « typeof » retourne le « Type » d'un type spécifié.



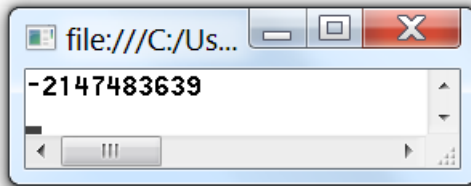
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            object o = 5L; //Boxing
            Type t = typeof(object);

            if (o.GetType() == t)
                Console.WriteLine("o est de type System.Object");
            else
            {
                string Format = "o n'est pas de type System.Object mais de type {0}";
                Console.WriteLine(Format, o.GetType());
            }

            Console.ReadLine();
        }
    }
}
```

L'OPÉRATEUR « CHECKED »

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = int.MaxValue;
            int y = x + 10;
            Console.WriteLine(y);
            Console.ReadLine();
        }
    }
}
```



Par défaut, si une expression ne contient que des valeurs constantes (littérales) vient à dépasser les bornes de valeurs du type de destination, ceci provoque une erreur de compilateur.

Par contre, si cette expression contient au moins une valeur non constantes (identifiant de variable), le compilateur ne détecte pas le dépassement de capacité.

```
int x = int.MaxValue + 10;
```

int int.MaxValue

Représente la plus grande valeur possible de System.Int32. Ce champ est constant.

Erreur :

L'opération engendre un dépassement de capacité au moment de la compilation dans le mode checked

L'OPÉRATEUR « CHECKED »



Le mot clé « checked » sert à activer explicitement le contrôle de dépassement pour les opérations arithmétiques de type entier et les conversions.

Lors de l'exécution, si un dépassement de pile est détecté, l'opérateur checked va demandé de déclencher une exception « `OverflowException` ».

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = int.MaxValue;
            int y = checked(x + 10);

            checked
            {
                //...
                y = x + 10;
                y += 7;
                //...
            }

            Console.WriteLine(y);
            Console.ReadLine();
        }
    }
}
```

 **L'exception `OverflowException` n'a pas été gérée** 

L'opération arithmétique a provoqué un dépassement de capacité.

Conseils de dépannage :

[Assurez-vous que vous ne divisez pas par zéro.](#)

[Obtenir une aide d'ordre général pour cette exception.](#)

[Rechercher de l'aide en ligne complémentaire...](#)

Paramètres d'exception :

☐ Pause lorsque ce type d'exception est levé

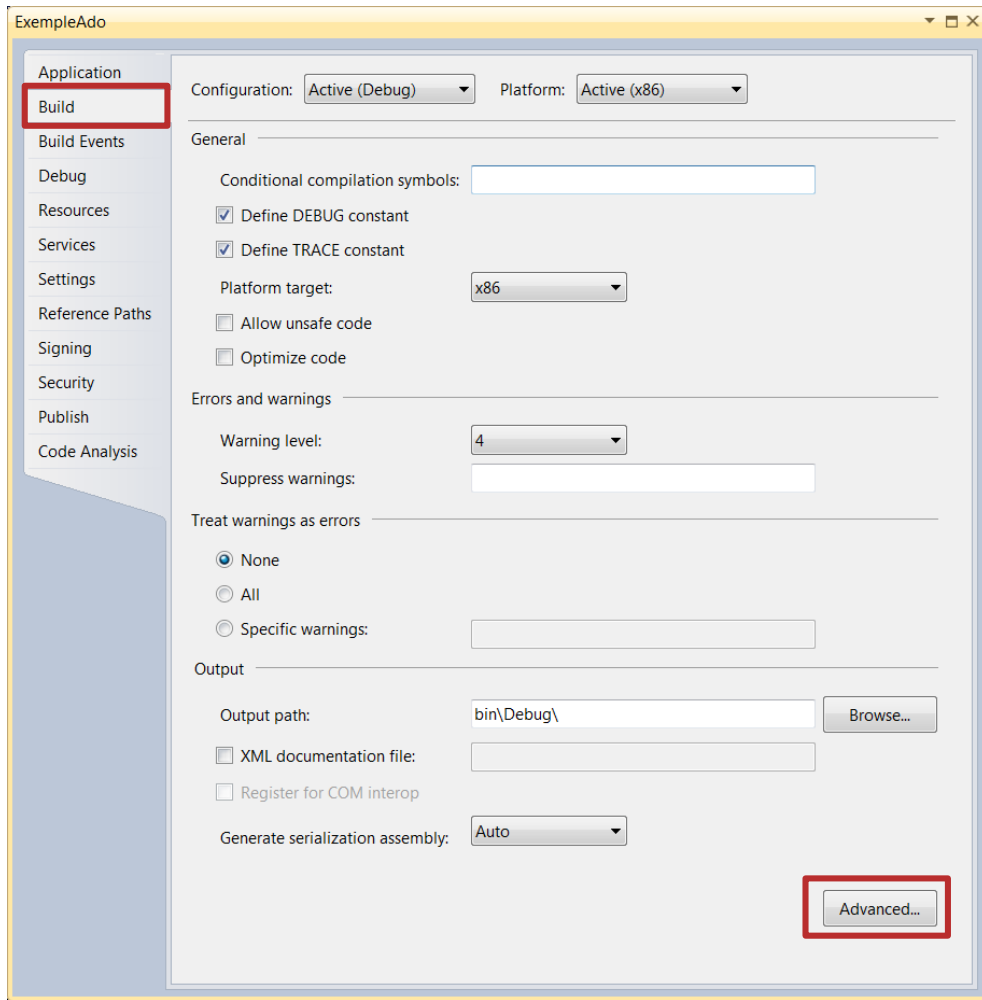
Actions :

[Afficher les détails...](#)

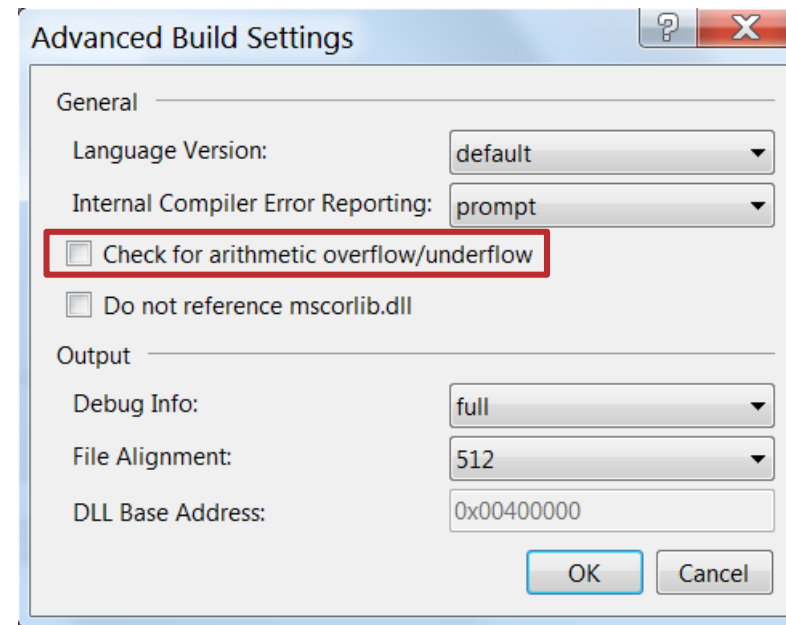
[Copier le détail de l'exception dans le Presse-papiers](#)

[Ouvrir les paramètres d'exception](#)

L'OPÉRATEUR « CHECKED »



Afin d'activer le contrôle de dépassement au niveau du projet nous pouvons le faire dans les options avancées de la génération.



L'OPÉRATEUR « UNCHECKED »

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int y;

            unchecked
            {
                //...
                y = int.MaxValue + 10;
                //...
            }

            Console.WriteLine(y);
            Console.ReadLine();
        }
    }
}
```

Le mot clé « unchecked », quant à lui, sert à supprimer le contrôle de dépassement pour les opérations arithmétiques de type entier et les conversions.

ORDRE DE PRIORITÉ DES OPÉRATEURS

Groupe	Opérateurs		Groupe	Opérateurs
	(), ., [], x++, x--, new, typeof, sizeof, checked, unchecked		ET binaire	&
Unaire	+, -, !, ~, ++x, --x et transtypage		XOR binaire	^
Multiplication / Division	*, /, %		OU binaire	
Addition / Soustraction	+, -		ET logique	&&
Opérateurs de décalage binaires	<<, >>		OU logique	
Relationnel	<, <=, >, >=, is, as		Opérateur ternaire	?:
Comparaison	==, !=		Affectation	=, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=



EXERCICES

EXERCICES (MAXIMUM 40')

- Calcule de la division entière, du modulo et de la division de deux entiers.
Résultat attendu pour 5 et 2 → Division entière : 2, Modulo : 1, Division : 2,5.
- Vérification d'un compte bancaire BBAN, si le compte est bon affichez OK sur la console sinon KO.
Le modulo des 10 premiers chiffres par 97 donne les 2 derniers. Sauf si le modulo = 0 dans ce cas les 2 derniers chiffres sont 97.
(utilisez la méthode « SubString » de la classe « string »).
- Transformer un compte bancaire BBAN Belge (xxx-xxxxxxx-xx) en IBAN (BExx-xxxx-xxxx-xxxx). Trouvez la démarche via un moteur de recherche.



LES BOUCLES

C# - LES FONDLEMENTS

- La boucle « for »
- La boucle « while »
- La boucle « do ... while »
- La boucle « foreach »

LES BOUCLES

LES BOUCLES

Dans le cadre de notre développement, nous serons amenés régulièrement à faire du traitement répétitif.

Pour cela nous utiliserons des blocs Itératifs, C# nous propose 4 constructions itératives afin de répéter nos actions, que le nombre de fois soit connu ou non. Il s'agit des boucles « for », « while », « do ... while » et de « foreach ».

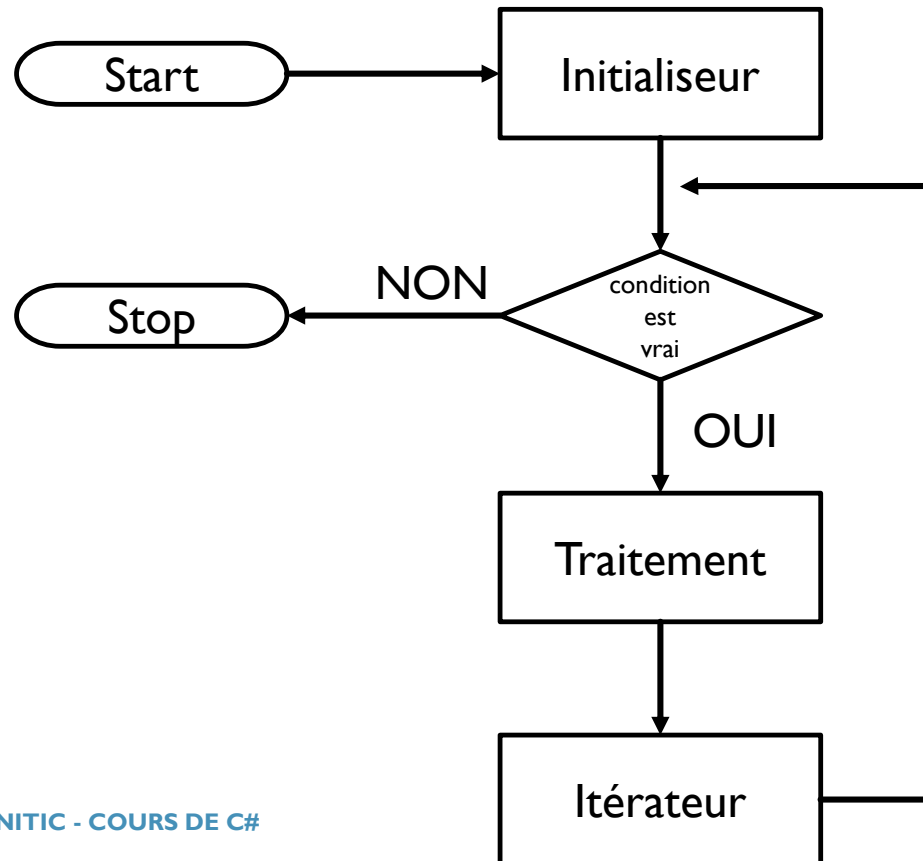
```
//Boucle "for"
for (int i = 0; i < 10; i++)
{
}

//Boucle "while"
while (condition == true)
{
}

//Boucle "do while"
do
{
} while (condition == true);

//Boucle "foreach"
foreach (int i in array)
{
}
```

LA BOUCLE « FOR »



La boucle for est le plus souvent utilisée lorsque nous connaissons le nombre d'itérations à effectuer.

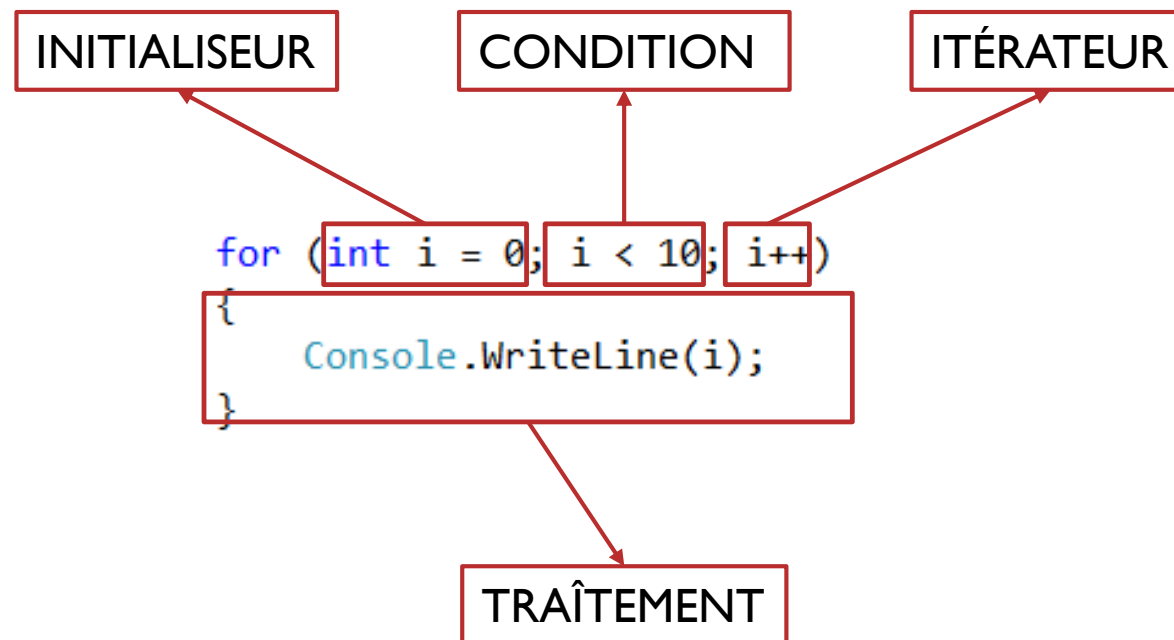
Cette boucle est composée de trois éléments :

L'**initialiseur** est l'expression évaluée avant l'exécution de la première boucle.

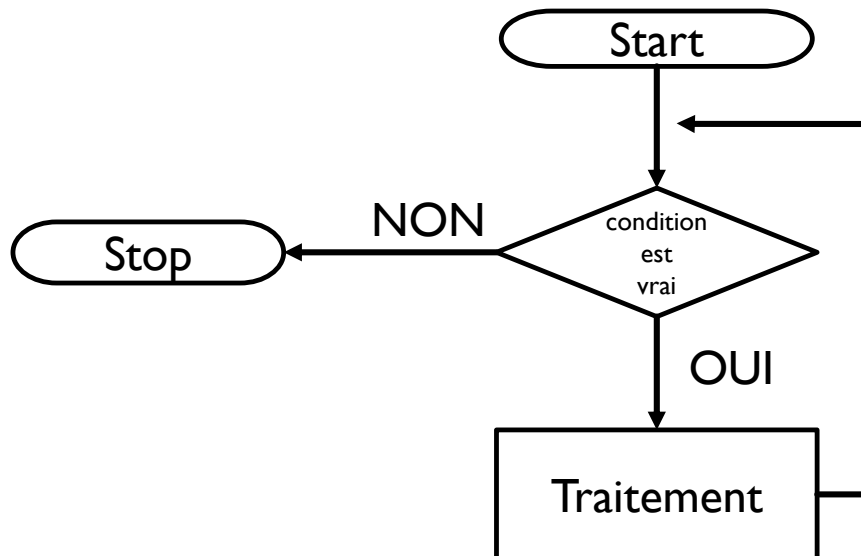
La **condition** est l'expression vérifiée avant chaque nouvelle itération de la boucle. (Elle doit être égale à « true » pour qu'une nouvelle itération puisse être effectuée)

L'**itérateur** est une expression qui est évaluée après chaque itération. Ces itérations cessent quand la condition renvoi « false ».

LA BOUCLE « FOR »



LA BOUCLE « WHILE »



Le principe de la boucle « while » est que :

Tant que la condition est « true », le bloc d'instructions s'exécutera, la condition est vérifiée avant l'itération.

```
int x = 0;

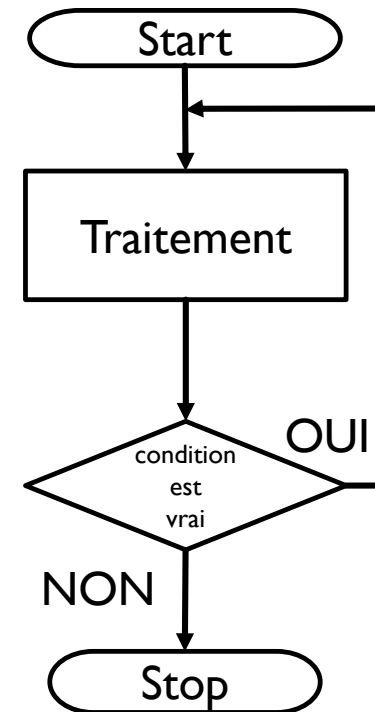
while (x < 10)
{
    Console.WriteLine(x);
    x++;
}
```


LA BOUCLE « DO ... WHILE »

La boucle do « while » fonctionne dans le même principe que la boucle « while » à ceci près que la condition est vérifiée après le traitement.

Ce qui a pour effet, d'imposer à la boucle, d'effectuer au moins une fois le traitement.

```
int x = 0;  
  
do  
{  
    Console.WriteLine(x);  
    x++;  
} while (x < 10);
```



LA BOUCLE « FOREACH »

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

            foreach (int i in array)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

La boucle « foreach » est le plus souvent utilisée lorsque nous souhaitons boucler sur l'entièreté des éléments d'une collection.

Elle va donc parcourir un par un les éléments en stockant l'élément courant dans une variable définie durant de l'itération.



Lors du traitement d'une boucle « foreach » il est interdit de modifier le nombre d'éléments de la collection.



EXERCICES

EXERCICES (MAXIMUM 50')

1. Calculer les 25 premiers nombres de la suite de Fibonacci
2. Calculer le factoriel d'un nombre entré au clavier.
3. Grâce à une boucle « for », calculez les x premiers nombre premier.
4. A l'aide de boucles « for » afficher les 5 premières tables de multiplication en allant jusque « x20 ».
1x1 = 1 ; 2x1 = 2 ;
2x1 = 2 ; 2x2 = 4 ;



LES TABLEAUX

C# - LES FONDEMENTS

- Les tableaux
 - À une dimension
 - À x dimensions ($x > 1$)
- Les collections
 - ArrayList
 - HashTable
 - Queue
 - Stack
- Les collections génériques
 - List<T>
 - Dictionary<T,U>
 - Queue<T> & Stack<T>

LES TABLEAUX

LES TABLEAUX À UNE DIMENSION

Un tableau représente un nombre **fixe** de variables (appelés éléments) d'un **type défini**.

Les éléments du tableau sont stockés de manière contigüe en mémoire, ce qui fournit un accès hautement efficient.

Pour déclarer un tableau nous utiliserons les « [] » à coté du type de la variable que nous déclarerons. Une fois instancié, chaque élément se verra attribué, par défaut, la valeur de « default(T) ».

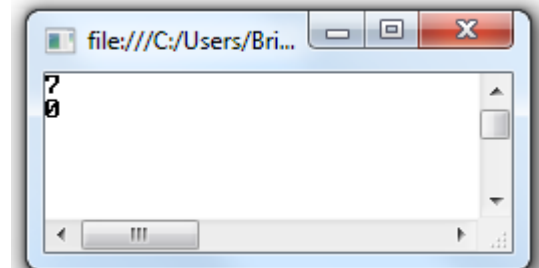
Le premier élément du tableau à pour index 0 (zéro).

Il est à noter également, derrière `int[]` se cache une instantiation de la classe « Array » ce qui fait de notre « `int[]` » un type référence

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] ints1 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            int[] ints2 = new int[10];

            ints2[0] = 7;
            Console.WriteLine(ints2[0]);
            Console.WriteLine(ints2[7]);

            Console.ReadLine();
        }
    }
}
```



LES TABLEAUX À X DIMENSIONS (X > 1) - MATRICIELS

C# gère deux types de tableaux multidimensionnels.

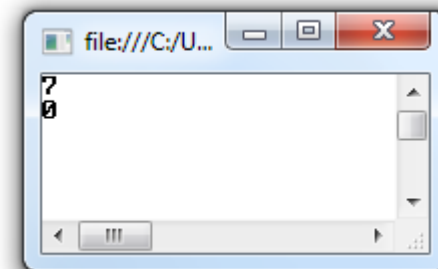
Les tableaux matriciels : où chaque dimension contient le même nombre d'éléments.

Lors de notre déclarations, nous séparerons chaque dimension par une virgule à l'intérieur des crochets.

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int[,] ints1 = { { 0, 1, 2, 3, 4 }, { 5, 6, 7, 8, 9 } };
            int[,] ints2 = new int[2,5];

            ints2[0,0] = 7;
            Console.WriteLine(ints2[0,0]);
            Console.WriteLine(ints2[1,2]);

            Console.ReadLine();
        }
    }
}
```



LES TABLEAUX À X DIMENSIONS (X > 1) - ORTHOGONAUX

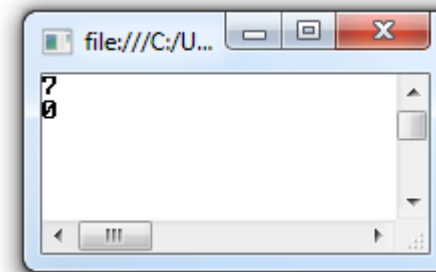
Les tableaux orthogonaux : où chaque dimension peut contenir un nombre différent d'éléments.

Lors de notre déclarations, nous déclarerons chaque dimension par une paire de crochets distincts.

```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int[][] ints1 = { new int[]{0, 1, 2, 3, 4 }, new int[]{ 5, 6, 7, 8, 9 } };
            int[][] ints2 = new int[2][];
            ints2[0] = new int[] { 0, 1, 2, 3, 4 };
            ints2[1] = new int[5];

            ints2[1][0] = 7;
            Console.WriteLine(ints2[1][0]);
            Console.WriteLine(ints2[1][2]);

            Console.ReadLine();
        }
    }
}
```



LES COLLECTIONS

Les tableaux ont un désavantage majeur c'est qu'ils sont de taille fixe. Ce qui veut dire que nous ne pouvons pas mettre plus d'éléments que le nombre défini par la taille du tableau.

Or lorsque nous importons des informations de l'extérieur (Base de données, Web Service, etc.), nous ne connaissons pas le nombre d'éléments que nous allons recevoir.

Il nous faut, par conséquent, des tableaux qui ont la faculté de voir leur taille croître et décroître.

De plus, ces collections peuvent travailler avec n'importe quel type de données puisque qu'elles travaillent avec des objets en paramètres, rendant les conversions implicite et explicite obligatoire.

Ces collections font partie de l'espace de noms « `System.Collections` »

LES COLLECTIONS - ARRAYLIST

L'« ArrayList » propose différentes méthodes afin de gérer les éléments de la collection,

- Add
- AddRange
- Clear
- Contains
- IndexOf
- Remove
- RemoveRange
- ToArray



```
using System;
using System.Collections;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList list = new ArrayList();
            object o = new object();

            list.Add(5);           //Boxing
            list.Add("Hello");     //Conversion implicite
            list.Add(o);

            list.AddRange(new object[] { 0, "Coucou", 5.2, true });

            list.Remove(o);

            Console.WriteLine(list.Count);
            int i = (int)list[0];   //Unboxing
            string s = (string)list[1]; //Conversion explicite

            Console.WriteLine(i);
            Console.WriteLine(s);

            Console.ReadLine();
        }
    }
}
```

LES COLLECTIONS - HASHTABLE

```
using System;
using System.Collections;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable dictionnaire = new Hashtable();

            object o = new object();

            dictionnaire.Add(1, 5);
            dictionnaire.Add("xxxye", new object());
            dictionnaire.Add(o, "Il était une fois");

            dictionnaire[1] = 7;

            int i = (int)dictionnaire[1];
            string s = (string)dictionnaire[o];

            Console.WriteLine(i);
            Console.WriteLine(s);

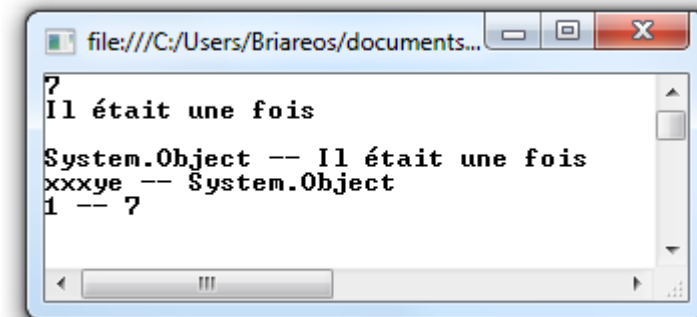
            Console.WriteLine();

            foreach (DictionaryEntry de in dictionnaire)
            {
                Console.WriteLine("{0} -- {1}", de.Key.ToString(), de.Value.ToString());
            }

            Console.ReadLine();
        }
    }
}
```

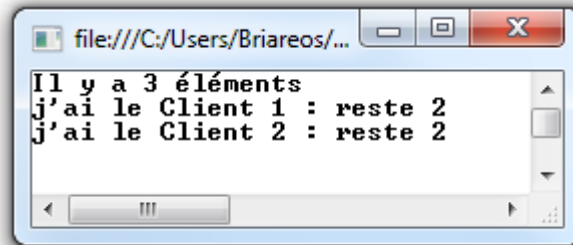
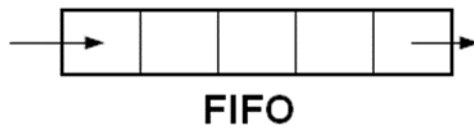
Une « Hashtable » est comparable à un dictionnaire où l'on accède à une valeur en utilisant une clé. La clé et la valeur sont de type « object ». De plus, la clé doit être unique.

Chaque élément de la « Hashtable » est de type « DictionaryEntry » proposant les propriétés Key et Value toutes deux de type « object ».



LES COLLECTIONS - QUEUE

La « Queue » est une liste d'« object » fonctionnant sur le principe FIFO (First In First Out). C'est donc le premier élément inséré qui sera en tête de file.



```
using System;
using System.Collections;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue queue = new Queue();

            //La méthode Enqueue ajoute un élément à la file
            queue.Enqueue("Client 1");
            queue.Enqueue("Client 2");
            queue.Enqueue("Client 3");

            //La propriété Count donne le nombre
            //d'éléments dans la file
            Console.WriteLine("Il y a {0} éléments", queue.Count);
            //La méthode Dequeue extrait un élément de la liste
            string client = (string)queue.Dequeue();
            Console.WriteLine("j'ai le {0} : reste {1}", client, queue.Count);

            //La méthode Peek retourne le premier élément
            //sans le retirer de la file
            client = (string)queue.Peek();
            Console.WriteLine("j'ai le {0} : reste {1}", client, queue.Count);

            Console.ReadLine();
        }
    }
}
```

LES COLLECTIONS - STACK

```
using System;
using System.Collections;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack queue = new Stack();

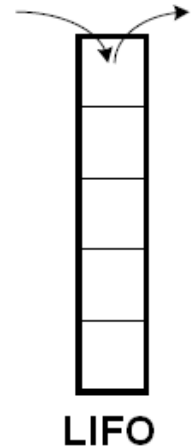
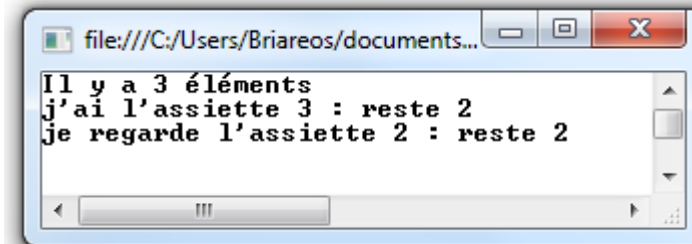
            //La méthode Push ajoute un élément à la pile
            queue.Push("assiette 1");
            queue.Push("assiette 2");
            queue.Push("assiette 3");

            //La propriété Count donne le nombre
            //d'éléments dans la pile
            Console.WriteLine("Il y a {0} éléments", queue.Count);
            //La méthode Pop extrait un élément de la pile
            string assiette = (string)queue.Pop();
            Console.WriteLine("j'ai l'{0} : reste {1}", assiette, queue.Count);

            //La méthode Peek retourne le premier élément
            //sans le retirer de la pile
            assiette = (string)queue.Peek();
            Console.WriteLine("je regarde l'{0} : reste {1}", assiette, queue.Count);

            Console.ReadLine();
        }
    }
}
```

Le « Stack » est une liste d'« Object » fonctionnant sur le principe LIFO (Last In First Out)



LES COLLECTIONS GÉNÉRIQUES

Les collections ont un avantage sur les tableaux, elles sont de taille variable. Mais en les utilisant, nous avons perdu le typage fort des tableaux.

Depuis le Framework .Net 2.0, un nouveau style de collection à fait son apparition, il s'agit des collections génériques, elles ont les avantages d'être de taille dynamique et d'être également typée.

Nous dégageant de devoir gérer les conversions (excepté dans le cadre du polymorphisme).

Nous y retrouvons les quatre collections précédentes :

- `List<T>`
- `Dictionary<T, U>`
- `Queue<T>`
- `Stack<T>`

en plus de nouvelles :

- `ObservableCollection<T>`
- `ReadOnlyCollection<T>`
- Etc.

Ces collections font partie de l'espace de noms
« `System.Collections.Generic` »

LES COLLECTIONS GÉNÉRIQUES - LIST<T>

La « List<T> » remplace l'« ArrayList ». C'est lors de la déclaration de la liste que nous spécifierons avec quel type nous travaillerons.

Une fois ce type spécifié, les éléments insérer et récupérer seront de ce type.

```
using System;
using System.Collections.Generic;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> ints = new List<int>();
            ints.Add(5);
            ints.AddRange(new int[] { 5, 6, 7, 8, 9, 0 });

            List<string> strings = new List<string>();
            strings.Add("Hello");
            strings.Add("Hola");
            strings.Add("Aloha");

            foreach (int i in ints)
            {
                Console.WriteLine(i);
            }

            Console.ReadLine();
        }
    }
}
```


LES COLLECTIONS GÉNÉRIQUES - DICTIONARY<T,U>

```
using System;
using System.Collections.Generic;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Dictionary<int, string> livres = new Dictionary<int, string>();

            livres.Add(1, "Le petit prince");
            livres.Add(2, "Harry Potter à l'école des sorciers");
            livres.Add(3, "C# pour les nuls c'est ici!");

            livres[1] = "Mvum de la découverte à la maîtrise";

            string livre = livres[1];

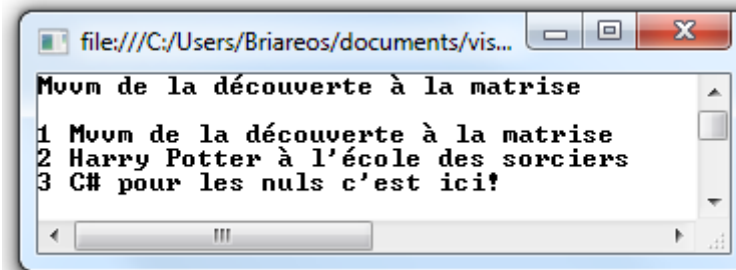
            Console.WriteLine(livre);
            Console.WriteLine();

            foreach (KeyValuePair<int, string> kvp in livres)
            {
                Console.WriteLine("{0} {1}", kvp.Key, kvp.Value);
            }

            Console.ReadLine();
        }
    }
}
```

Un « Dictionary<T, U> » est comparable à une « HashTable ». Cependant nous allons pouvoir spécifier les types de la clé et de la valeur avec lesquels nous nous souhaiterons travailler.

De plus, chaque élément ne sera donc plus de type « DictionaryEntry » mais de type « KeyValuePair<T, U> ».



LES COLLECTIONS GÉNÉRIQUES - QUEUE<T> & STACK<T>

Nous retrouvons également les « Queue » et les « Stack » génériques, ceux-ci fonctionnent exactement de la même manière que les non génériques, excepté que nous spécifions le type des éléments que nous allons manipuler.

```
using System;
using System.Collections.Generic;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue<string> Clients = new Queue<string>();

            Clients.Enqueue("Client 1");
            Clients.Enqueue("Client 2");
            Clients.Enqueue("Client 3");

            Console.WriteLine(Clients.Count);
            string client = Clients.Dequeue();
            Console.WriteLine("{0} : reste {1}", client, Clients.Count);
            client = Clients.Peek();
            Console.WriteLine("{0} : reste {1}", client, Clients.Count);

            Console.ReadLine();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<string> Assietes = new Stack<string>();

            Assietes.Push("Assiette 1");
            Assietes.Push("Assiette 2");
            Assietes.Push("Assiette 3");

            Console.WriteLine(Assietes.Count);
            string Assiette = Assietes.Pop();
            Console.WriteLine("{0} : reste {1}", Assiette, Assietes.Count);
            Assiette = Assietes.Peek();
            Console.WriteLine("{0} : reste {1}", Assiette, Assietes.Count);

            Console.ReadLine();
        }
    }
}
```



EXERCICES

EXERCICES (MAXIMUM 50')

1. Grâce à une boucle « while » et à l'aide d'une collection, calculez les nombres premiers inférieur à un nombre entier entré au clavier.
2. Grâce à une boucle « for » et à l'aide d'une collection générique, calculez les x premiers nombres premiers (version optimisée).
3. Demandez à l'utilisateur d'introduire deux nombres au clavier et faite l'addition de ces deux nombres en ne convertissant que caractère par caractère. (méthode ToCharArray() de la classe « string »).



LES STRUCTURES

C# - LES FONDEMENTS

- Différences en les structures et les classes
- Déclaration & utilisation

LES STRUCTURES

DIFFÉRENCES ENTRE LES STRUCTURES ET LES CLASSES

Les structures ont pour but de regrouper des éléments de données. Leurs champs sont, la plupart du temps, déclarés comme « public ».

Cependant, les structures ne sont pas des classes et elles s'en différencient sur plusieurs points :

- Dans une déclaration de structure, les champs ne peuvent pas être initialisés à moins qu'ils ne soient déclarés comme `const` ou `static`.
- Une structure ne peut pas déclarer constructeur sans paramètre ni de destructeur.
- Les structures sont copiées lors de l'assignation. Lorsqu'une structure est assigné à une nouvelle variable, toutes les données sont copiées et les modifications apportées à la nouvelle copie ne changent pas les données de la copie d'origine.
- Une structure ne peut pas utiliser les concepts d'héritage.
- Contrairement aux classes et dans certains cas, les structures peuvent être instanciées sans avoir recours à l'opérateur « `new` ».

DIFFÉRENCES ENTRE LES STRUCTURES ET LES CLASSES

Ceci dit, elles peuvent comme les classes :

- Déclarer des constructeurs qui ont des paramètres.
- Implémenter des interfaces.
- Etre utilisée comme un type « Nullable » et se voir assigner la valeur « null ».
- Contenir des constantes, des champs, des méthodes, des propriétés, des indexeurs, des opérateurs, des événements et des types imbriqués.

Cependant, si plusieurs de ces membres sont nécessaires, nous devons envisager de faire de notre structure une classe,



Le mot-clé « static » ainsi que les concepts de classes, d'héritage, d'indexeurs, d'interfaces, de constructeurs et du destructeur seront vu en détail dans la partie « Orienté Objet » de ce cours.

DÉCLARATION & UTILISATION

La déclaration d'une structure se fait à l'aide du mot-clé « struct »

```
namespace CoursCSharpFondements
{
    public struct MesureAngulaire
    {
        public int Degre, Minutes, Secondes;
        public string Direction;
    }
}

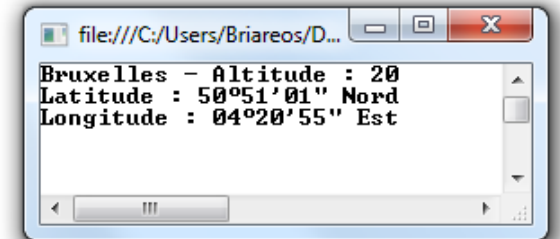
namespace CoursCSharpFondements
{
    public struct Localisation
    {
        public MesureAngulaire Longitude, Latitude;
        public int Altitude;
    }
}
```

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Localisation Bruxelles;
            Bruxelles.Latitude.Degre = 50;
            Bruxelles.Latitude.Minutes = 51;
            Bruxelles.Latitude.Secondes = 1;
            Bruxelles.Latitude.Direction = "Nord";
            Bruxelles.Longitude.Degre = 4;
            Bruxelles.Longitude.Minutes = 20;
            Bruxelles.Longitude.Secondes = 55;
            Bruxelles.Longitude.Direction = "Est";
            Bruxelles.Altitude = 20;

            Console.WriteLine("Bruxelles - Altitude : {0}", Bruxelles.Altitude);
            Console.WriteLine("Latitude : {0:D2}°{1:D2}'{2:D2}\" {3}", Bruxelles.Latitude.Degre,
                Bruxelles.Latitude.Minutes, Bruxelles.Latitude.Secondes, Bruxelles.Latitude.Direction);
            Console.WriteLine("Longitude : {0:D2}°{1:D2}'{2:D2}\" {3}", Bruxelles.Longitude.Degre,
                Bruxelles.Longitude.Minutes, Bruxelles.Longitude.Secondes, Bruxelles.Longitude.Direction);

            Console.ReadLine();
        }
    }
}
```





LES MÉTHODES

C# - LES FONDEMENTS

- Déclaration
- Mot-clé « return »
- Invocation
- Les paramètres
- Les paramètres facultatifs
- Paramètres nommés
- Mot clé « params »
- Mot-clé « ref »
- Mot-clé « out »
- Différence entre signature et prototype
- Surcharge de méthodes

LES MÉTHODES

DÉCLARATION

En programmation, nous essayons au maximum de ne pas répéter le code par copier-coller. Pour y parvenir, nous déclarons des méthodes afin qu'elles réalisent une série d'actions spécifiques afin de simplifier notre code et d'améliorer la maintenabilité.

Une fois ces méthodes créées, nous les invoquons afin qu'elles réalisent leur tâche.

Dans de nombreux langage de programmation, nous retrouvons deux types de méthodes : les procédures et les fonctions. Par définition, on entend qu'une procédure ne retourne rien et qu'une fonction retourne une valeur.

En C#, différencions une fonction d'une procédure au travers de son type de retour, pour toutes les procédures nous utiliserons le mot clé « void ». De plus, nos fonctions devront obligatoirement retourner une valeur grâce au mot clé « return ».

Déclaration d'une méthode

```
[modificateur] return_type method_name ([paramètres])  
{  
    //corps de la méthode  
}
```

```
namespace CoursCSharpFondements  
{  
    public struct MaStructure  
    {  
        //Déclaration d'une procédure  
        public void UneProcédure()  
        {  
            //Code à réaliser  
        }  
  
        //Déclaration d'une fonction retournant un entier  
        public int UneMethode()  
        {  
            //Code à réaliser  
            return 1;  
        }  
    }  
}
```

MOT-CLÉ « RETURN »

Le mot-clé « return » est appelé, le plus souvent, pour retourner une valeur dans nos fonctions, mais son rôle est en réalité de mettre fin à l'appel d'une méthode en retournant une valeur dans le cadre des fonctions.

En effet, y compris dans nos procédures nous pouvons faire appel à ce mot clé pour mettre fin à un appel de méthode.

```
public void UneProcedure()  
{  
    if (!Condition)  
    {  
        //Met fin à l'invocation de la procédure  
        //si la condition n'est pas remplie  
        return;  
    }  
  
    //Lignes de code  
}
```

MOT-CLÉ « RETURN »

```
public string EstValide()  
{  
    if (Condition)  
    {  
        return "OK";  
    }  
    else  
    {  
        return "KO";  
    }  
}
```



```
public string EstValide()  
{  
    if (Condition)  
    {  
        return "OK";  
    }  
  
    return "KO";  
}
```

De plus, dans le cadre des fonctions, nous sommes obligé de retourner une valeur et ce quelque soit le chemin prit par notre application.

```
public string EstValide()  
{  
    if (Condition)  
    {  
        return "OK";  
    }  
}
```



INVOCATION

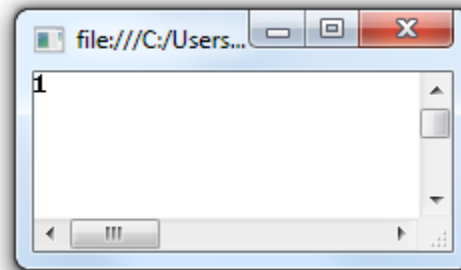
```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaStructure m = new MaStructure();

            m.UneProcedure();
            int retour = m.UneFonction();
            Console.WriteLine(retour);

            Console.ReadLine();
        }
    }
}
```

Une fois que nous avons créés nos méthodes, nous pouvons les invoquer (appeler) grâce à leur nom et avec éventuellement leur(s) paramètre(s).



LES PARAMÈTRES

La plupart des méthodes doivent recevoir des éléments afin de fonctionner correctement, ces éléments sont appelé des paramètres.

Lors de la déclaration des paramètres, nous devons spécifier le type et leur donner un nom que nous pourrons utiliser dans la méthode.

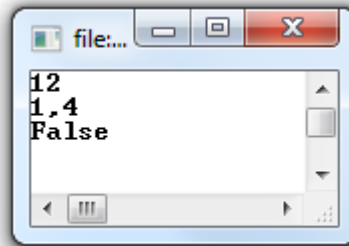
Ensuite, lors de l'appel de nous passerons les valeurs de ses paramètres afin que la méthode effectue son traitement.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Mathematique m = new Mathematique();

            Console.WriteLine(m.Addition(5, 7));
            Console.WriteLine(m.Division(7, 5));
            Console.WriteLine(m.EstPaire(5));

            Console.ReadLine();
        }
    }
}
```



```
namespace CoursCSharpFondements
{
    public struct Mathematique
    {
        public int Addition(int x, int y)
        {
            return x + y;
        }

        public double Division(double x, double y)
        {
            return x / y;
        }

        public bool EstPaire(int i)
        {
            return i % 2 == 0;
        }
    }
}
```

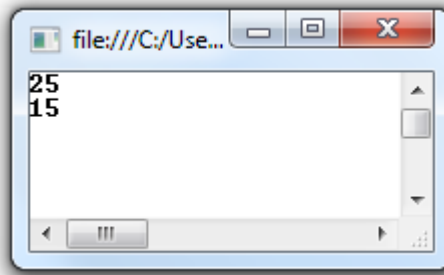

LES PARAMÈTRES FACULTATIFS

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Mathematique m = new Mathematique();

            //Appel de la fonction Addition sans utiliser la valeur
            //par défaut du paramètre y;
            Console.WriteLine(m.Addition(10, 15));
            //Appel de la fonction Addition en utilisant la valeur
            //par défaut du paramètre y;
            Console.WriteLine(m.Addition(10));

            Console.ReadLine();
        }
    }
}
```



Depuis la version 4.0 du Framework .Net, nous pouvons donner des valeurs par défaut à nos paramètres.

Les paramètres ayant une valeurs par défaut sont appelés, paramètre facultatif).

La valeur par défaut est définie à la déclaration du paramètre à l'aide de l'opérateur « = ».

Cependant, les paramètres facultatifs doivent être les derniers paramètres à être définis.

```
namespace CoursCSharpFondements
{
    public struct Mathematique
    {
        public int Addition(int x, int y = 5)
        {
            return x + y;
        }
    }
}
```

PARAMÈTRES NOMMÉS

En invoquant la méthode, nous avons la possibilité de choisir l'ordre dans lequel nous passons les paramètres indépendamment de l'ordre dans lequel ils ont été déclaré.

Cela est encore plus utile lorsque nous avons des paramètres facultatifs.





Pour nommer un paramètre dans son appel, nous utiliserons le schéma suivant : « nom_du_paramètre:valeur »

```
namespace CoursCSharpFondements
{
    public struct Mathematique
    {
        public int Addition(int x = 1, int y = 10, int z = 100)
        {
            return x + y + z;
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Mathematique m = new Mathematique();

        //Appel de la fonction Addition en utilisant
        //la valeur 5 pour z,
        //la valeur 15 pour x et
        //la valeur par défaut pour y;
        Console.WriteLine(m.Addition(z:5, x:15));

        Console.ReadLine();
    }
}
```

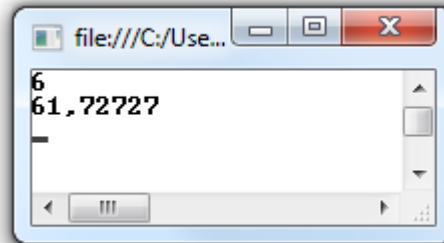
Variables locales	
Nom	Valeur
 this	{CoursCSharpFondements.Mathematique}
 x	15
 y	10
 z	5

MOT-CLÉ « PARAMS »

Lorsque nous ignorons le nombre de paramètres à passer à une fonction, nous pouvons utiliser le mot clé « params ».

```
using System.Linq;

namespace CoursCSharpFondements
{
    public struct Mathematique
    {
        public float Moyenne(params int[] ints)
        {
            float somme = 0;
            foreach (int i in ints)
            {
                somme += i;
            }
            return somme / ints.Count();
        }
    }
}
```



Toute fois, trois règles sont à respecter :

1. Il ne peut y en avoir qu'un.
2. Le paramètre utilisant le mot clé « params » doit être le dernier à être spécifié.
3. Toutes les valeurs passées pour ce paramètre devront être du même type que celui déclaré.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Mathematique m = new Mathematique();

            Console.WriteLine(m.Moyenne(5, 7));
            Console.WriteLine(m.Moyenne(1, 22, 333, 4, 55, 67, 78, 89, 9, 10, 11));

            Console.ReadLine();
        }
    }
}
```

MOT-CLÉ « REF »

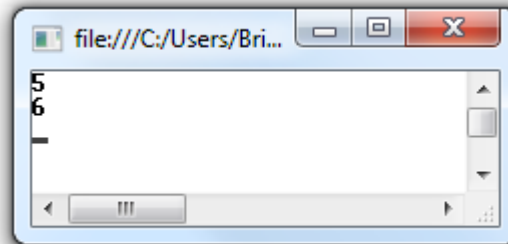
Avec les types valeurs, le passage en paramètre se fait par copie.

Cependant, il nous est parfois nécessaire de modifier la valeur d'une variable passée en paramètre à l'intérieur d'une méthode et que cette modification soit répercutée à la variable source également.

Pour ce faire nous utiliserons « **ref** » (Par référence)

```
namespace CoursCSharpFondements
{
    public struct Mastructure
    {
        public void Methode1(int i)
        {
            i++;
        }

        public void Methode2(ref int i)
        {
            i++;
        }
    }
}
```



```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            Mastructure m = new Mastructure();

            m.Methode1(i);
            Console.WriteLine(i);
            m.Methode2(ref i);
            Console.WriteLine(i);

            Console.ReadLine();
        }
    }
}
```

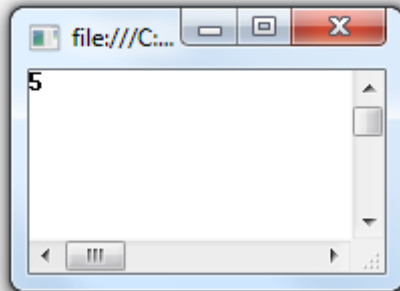
MOT-CLÉ « OUT »

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Mastructure m = new Mastructure();
            int i;

            m.Methode1(out i);
            Console.WriteLine(i);

            Console.ReadLine();
        }
    }
}
```



Le mot clé « out » va fonctionner comme « ref ».

En effet, la variable sera également passée par référence.

Cependant, les différences seront que :

- La variable ne devra pas obligatoirement être initialisée avant l'appel de la méthode,
- La variable passée en paramètre devra être initialisée dans la méthode appelée.

```
namespace CoursCSharpFondements
{
    public struct Mastructure
    {
        public void Methode1(out int i)
        {
            i = 5;
        }
    }
}
```

DIFFÉRENCE ENTRE SIGNATURE ET PROTOTYPE

Signature d'une méthode

La signature d'une méthode est définie par son nom ainsi que du nombre et du type de leur(s) paramètre(s) passé(s) en entrée.

```
public void MaMethode(string s)
{
    //Traitement
}
```

Prototype d'une méthode

Le prototype de méthode quant à lui reprend l'intégralité de l'entête de la méthode.

```
public void MaMethode(string s)
{
    //Traitement
}
```

SURCHARGE DE MÉTHODES

La surcharge de méthode est un principe qui permet de déclarer plusieurs méthodes ayant le même nom pour peu que le reste de leur signature soit différent.

Soit par le type, soit par le nombre de paramètres.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaStructure m;
            m.MaMethode();
        }
    }
}
```

▲ 1 sur 3 ▼ void MaStructure.MaMethode()

```
namespace CoursCSharpFondements
{
    public struct MaStructure
    {
        public void MaMethode()
        {
            //Traitement
        }

        public void MaMethode(int i)
        {
            //Traitement
        }

        public void MaMethode(string s)
        {
            //Traitement
        }
    }
}
```



EXERCICES

EXERCICES (MAXIMUM 30')

Ecrire une structure pour résoudre une équation du second degré.

La structure devra contenir :

- Trois variables membres publiques A, B et C de type double.
- Une méthode publique « Resoudre » retournant une valeur de type « bool » stipulant si une réponse a été trouvée et devra retourner également les valeurs de X1 et de X2 de type double.
- Si aucune solution n'a été trouvée, les valeurs de X1 et de X2 doivent être égale à « null ».



LES ÉNUMÉRATIONS

C# - LES FONDEMENTS

- Déclaration
- Utilisation
- Enum.GetNames(...)
- Enum.Parse(...)
- Enum.TryParse<T>(...)

LES ÉNUMÉRATIONS

DÉCLARATION

Les énumérations nous permettent de stocker de façon simple plusieurs valeurs invariables qui sont liées logiquement les unes aux autres.

Les énumérations se déclare avec le mot clé « enum » et les valeurs se voient attribuer, à chacune et par défaut, une valeur numérique.

- Essence vaudra 0
- Diesel vaudra 1
- Gaz vaudra 2

Cependant nous pouvons définir nous même ces valeurs.

Nous ne pouvons pas déclarer d'énumération à l'intérieur d'une méthode en raison que déclarer une énumération revient à définir un type de variable.

Toute énumération hérite de la classe « Enum ».

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence, Diesel, Gaz }

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

using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }

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

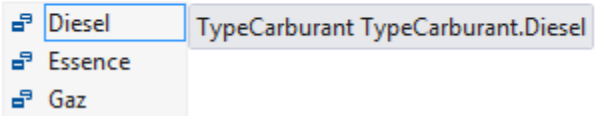
UTILISATION

Une fois définie, nous pouvons utiliser l'énumération comme type de variable et l'« IntelliSense » nous aidera à obtenir les valeurs possibles.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }

        static void Main(string[] args)
        {
            TypeCarburant Carburant = TypeCarburant.
        }
    }
}
```



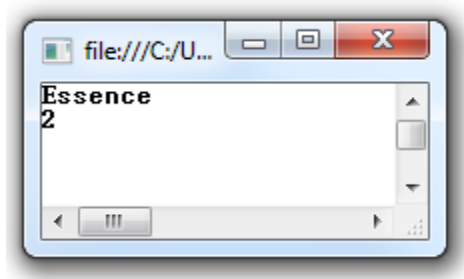
The image shows a screenshot of a code editor with a C# program. The code defines an enumeration `TypeCarburant` with values `Essence = 2`, `Diesel = 4`, and `Gaz = 8`. In the `Main` method, the line `TypeCarburant Carburant = TypeCarburant.` is being typed. An IntelliSense dropdown menu is visible, showing the options `Diesel`, `Essence`, and `Gaz`. The `Diesel` option is selected, and the text `TypeCarburant TypeCarburant.Diesel` is shown to the right of the dropdown.

UTILISATION

Une fois définie, nous pouvons utiliser l'énumération comme type de variable et l'« IntelliSense » nous aidera à obtenir les valeurs possibles.

Une fois la valeur de la variable définie, nous pouvons récupérer les deux types de valeurs :

- Littérale
- Numérique



```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }

        static void Main(string[] args)
        {
            TypeCarburant Carburant = TypeCarburant.Essence;

            Console.WriteLine(Carburant.ToString());
            Console.WriteLine((int)Carburant);
            Console.ReadLine();
        }
    }
}
```

ENUM.GETNAMES(...)

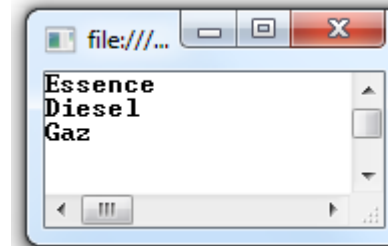
```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }

        static void Main(string[] args)
        {
            foreach (string s in Enum.GetNames(typeof(TypeCarburant)))
            {
                Console.WriteLine(s);
            }
            Console.ReadLine();
        }
    }
}
```

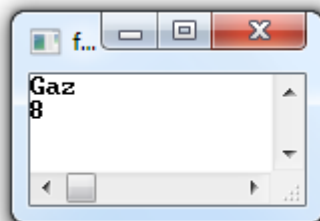
Dans certaines situations, Il nous arrivera d'avoir besoin, en une fois, de toutes les valeurs littérales spécifiées dans une énumération.

Pour ce faire nous pouvons utiliser la méthode statique « GetNames », qui en lui passant le type de l'énumération, va nous retourner un tableau de « string » (string[]).



ENUM.PARSE(...)

À l'inverse maintenant, pour passer de la valeur de type « string » à une valeur de type de l'énumération, nous utiliserons la méthode « Parse ».



```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }

        static void Main(string[] args)
        {
            string s = "Gaz";

            TypeCarburant tc = (TypeCarburant)Enum.Parse(typeof(TypeCarburant), s);
            Console.WriteLine(tc.ToString());
            Console.WriteLine((int)tc);
            Console.ReadLine();
        }
    }
}
```


ENUM.TRYPARSE<T>(...)

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }

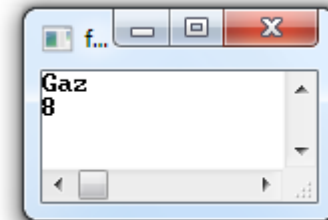
        static void Main(string[] args)
        {
            string s = "Gaz";

            TypeCarburant tc;
            if (Enum.TryParse<TypeCarburant>(s, out tc))
            {
                Console.WriteLine(tc.ToString());
                Console.WriteLine((int)tc);
            }
            Console.ReadLine();
        }
    }
}
```

Une autre méthode consiste d'utiliser la méthode générique « TryParse ».

Cette méthode retourne une valeur de type « bool », qui spécifie si la conversion est réussie ou non.

Le résultat de la conversion sera stocké dans la variable passée en paramètre avec le mot clé « out ».





RÉFÉRENCES

C# - LES FONDEMENTS

- O'Reilly :
C# 4.0 in a nutshell (ISBN-13 : 978-0-596-80095-6)
C# 5.0 in a nutshell (ISBN-13 : 978-1-4493-2010-2)
- Microsoft :
Spécification du langage C# 5.0 ([http://msdn.microsoft.com/fr-fr/library/vstudio/ms228593\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/vstudio/ms228593(v=vs.110).aspx))