

COURS DE C#

LES FONDEMENTS



- Premier programme
- Visual Studio
- Déboguer son application
- Avant de commencer
- 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 :
C# documentation (<https://docs.microsoft.com/en-us/dotnet/csharp/>)
Guide .Net Core (<https://docs.microsoft.com/en-us/dotnet/core/>)

PREMIER PROGRAMME

C# - LES FONDEMENTS

- Présentation du langage
- Le Framework .NET Core
- L'outil « dotnet »
- .Net Core 3.1 (LTS) OS supportés
- Avec Visual Studio Code

PREMIER PROGRAMME

PRÉSENTATION DU C#

C# (prononcé "See Sharp") est un langage de programmation apparu en 2002. Il se veut simple, moderne, orienté objet et fortement typé. Il tient ses racines de la famille des langages C et sera immédiatement familier aux programmeurs C, C++, Java et JavaScript.

| Version | Date |
|---------|----------------|
| 1.0 | Janvier 2002 |
| 1.1 | Avril 2003 |
| 2.0 | Novembre 2005 |
| 3.0 | Novembre 2007 |
| 4.0 | Avril 2010 |
| 5.0 | Août 2012 |
| 6.0 | Juillet 2015 |
| 7.0 | Mai 2018 |
| 8.0 | Septembre 2019 |

LE FRAMEWORK .NET CORE

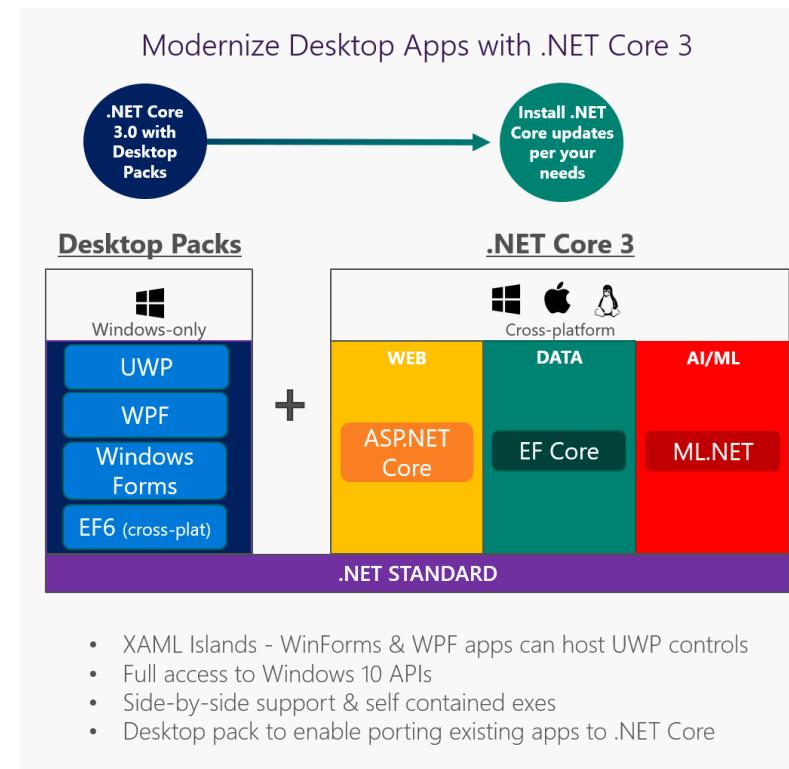
.NET Core est une plateforme de développement généraliste open source qui est tenue à jour par Microsoft et la communauté .NET sur GitHub.

Il est multiplateforme et prend en charge Windows, MacOs et Linux. Elle permet de générer des applications destinées à des appareils, au cloud et à l'IOT,

Actuellement, dans sa version 3.0, le Framework permet de créer des applications console, des sites web en ASP MVC et des API pour ce qui est du multiplateforme et des applications UWP, Windows Forms et WPF pour Windows.

Pour cela, il se repose sur les librairies du .NET Standard. Le .Net Standard est une spécification officielle des API .NET qui sont destinées à être disponibles sur toutes les implémentations de .NET.

L'objectif de .NET Standard est d'établir une meilleure uniformité dans l'écosystème .NET



PROCHAINE VERSION MAJEURE

.NET – A unified platform



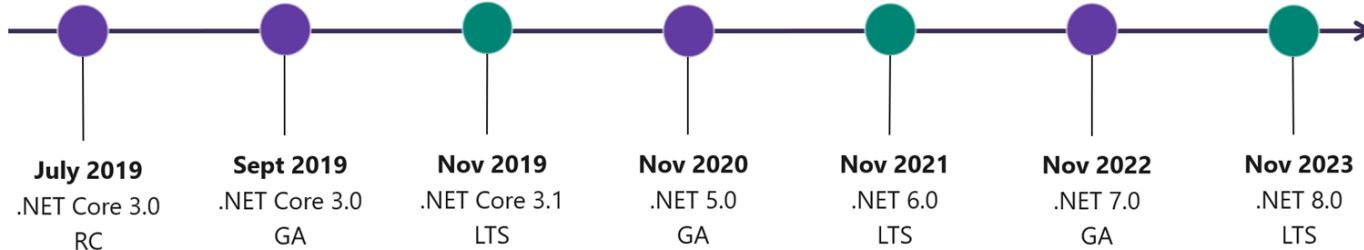
Annoncé pour novembre 2020, cette .Net 5.0 nous donnera la possibilité d'unifier l'univers .Net et rendant accessible pour Windows, Linux, macOS, iOS, Android, tvOS, watchOS, WebAssembly et plus encore.

.Net 5.0 a pour objectifs :

- De créer un environnement d'exécution .NET unique, utilisable partout et offrant des environnements d'exécution et des expériences de développement uniformes.
- De développer les fonctionnalités de .NET en exploitant le meilleur des technologies .NET Core, .NET Framework, Xamarin et Mono.
- De développer des produits à partir d'une base commune, sur laquelle les développeurs (Microsoft et la communauté) pourront travailler et développer ensemble.

PROCHAINE VERSION MAJEURE (5.0)

.NET Schedule



- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

.NET CORE 3.1 (LTS) OS SUPPORTÉS

Windows

| | |
|-------------------|-------------|
| Windows Clients | 7 SPI+, 8.1 |
| Windows 10 Client | 1607+ |
| Nano Server | 1803+ |
| Windows Server | 2012 R2+ |

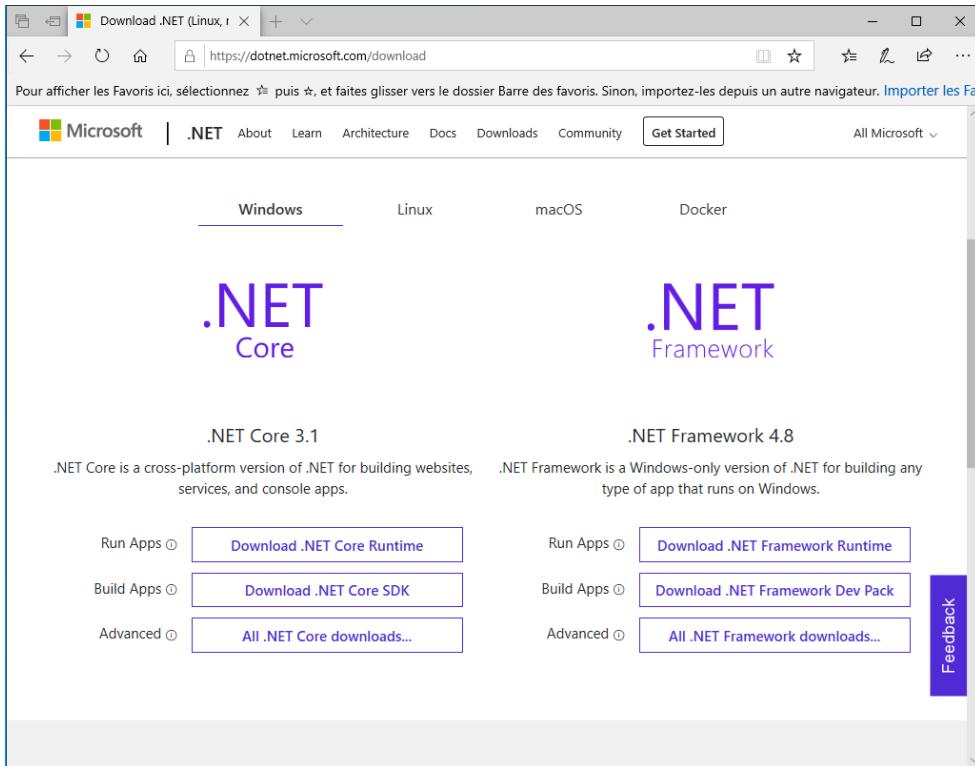
Linux

| | |
|------------------------------|---------|
| Red Hat Enterprise Linux | 7+ |
| CentOS | 7+ |
| Oracle Linux | 7+ |
| Fedora | 29+ |
| Debian | 9+ |
| Ubuntu | 16.04+ |
| Linux Mint | 18+ |
| OpenSUSE | 15+ |
| SUSE Enterprise Linux (SLES) | 12 SP2+ |
| Alpine Linux | 3.8+ |

Mac OS

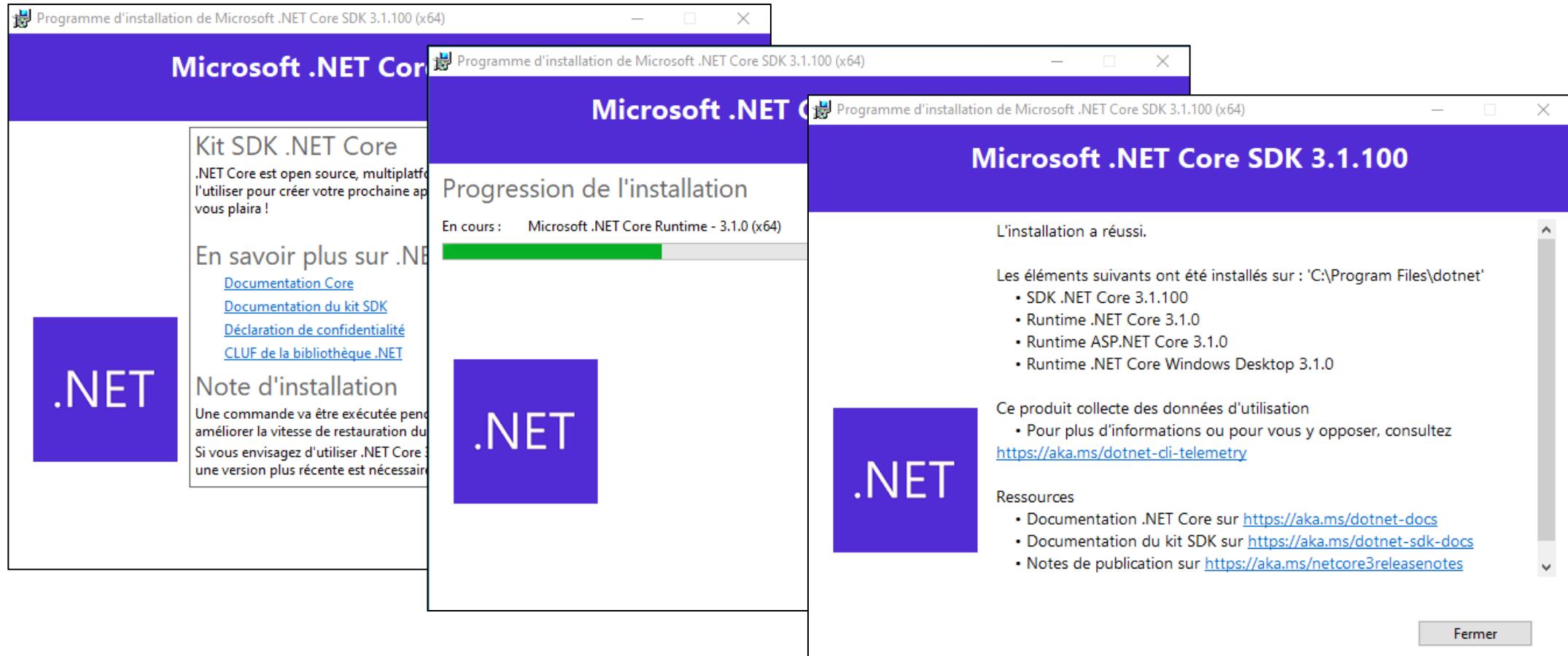
| | |
|----------|--------------|
| Mac OS X | 10.13+ (x64) |
|----------|--------------|

INSTALLATION SOUS WINDOWS



1. Se rendre sur <https://dotnet.microsoft.com/download>
2. Télécharger le .Net Core SDK
3. Installer le logiciel

INSTALLATION SOUS WINDOWS



INSTALLATION SOUS LINUX

Avant toute chose, nous devrons installer le package « Snapd ».

« Snap » est un système de déploiement de logiciels et de gestion de paquets développé par Canonical pour le système d'exploitation Linux.

Il nous servira à installer les packages nécessaire pour faire du développement .Net Core sous Linux.

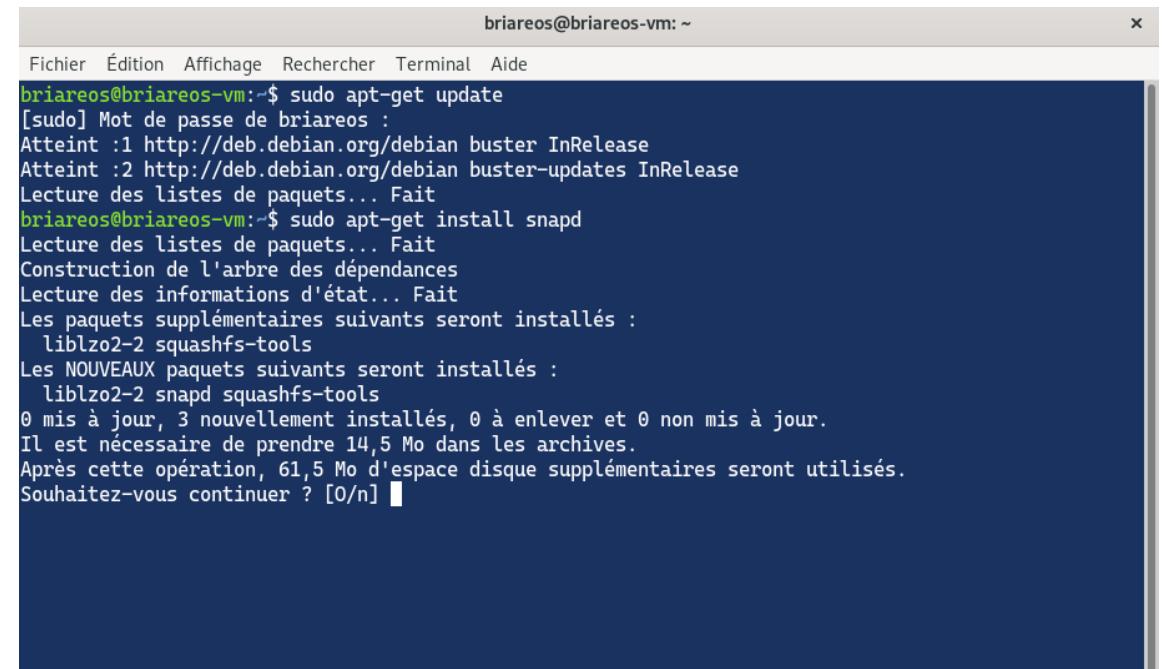
Après s'être assuré d'avoir mis à jours votre liste de paquets disponibles.

« sudo apt-get update »

Nous pouvons installer « Snapd » avec la commande :

« sudo apt-get install snapd »

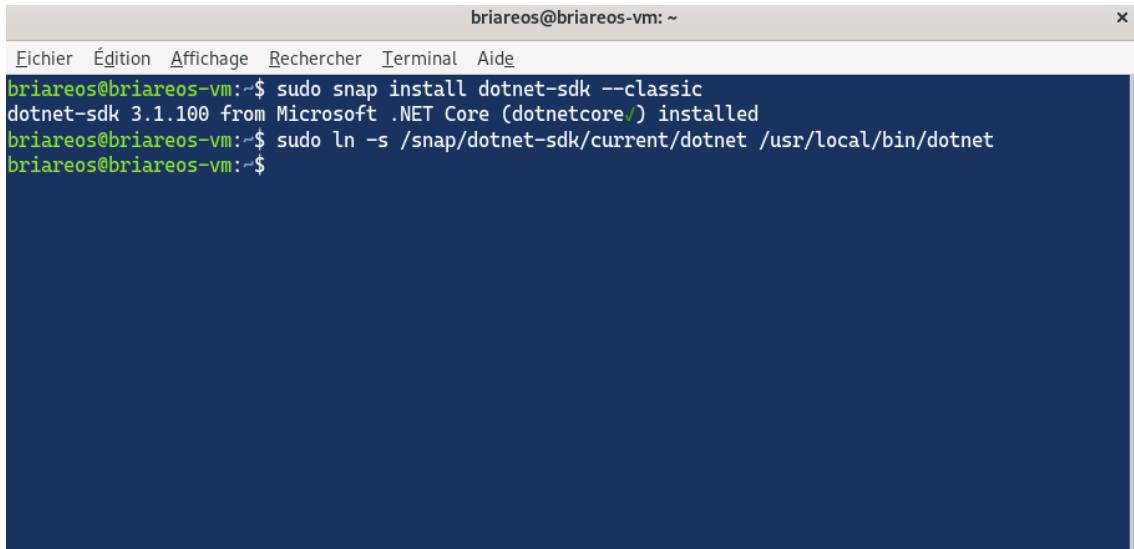
Ensuite, il est conseillé de redémarrer la machine.



The screenshot shows a terminal window titled "briareos@briareos-vm: ~". The window contains the following command and its output:

```
briareos@briareos-vm:~$ sudo apt-get update
[sudo] Mot de passe de briareos :
Atteint :1 http://deb.debian.org/debian buster InRelease
Atteint :2 http://deb.debian.org/debian buster-updates InRelease
Lecture des listes de paquets... Fait
briareos@briareos-vm:~$ sudo apt-get install snapd
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances
Lecture des informations d'état... Fait
Les paquets supplémentaires suivants seront installés :
  liblzo2-2 squashfs-tools
Les NOUVEAUX paquets suivants seront installés :
  liblzo2-2 snapd squashfs-tools
0 mis à jour, 3 nouvellement installés, 0 à enlever et 0 non mis à jour.
Il est nécessaire de prendre 14,5 Mo dans les archives.
Après cette opération, 61,5 Mo d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] ■
```

INSTALLATION SOUS LINUX



A screenshot of a Linux terminal window titled "briareos@briareos-vm: ~". The window shows the following command being run:

```
briareos@briareos-vm:~$ sudo snap install dotnet-sdk --classic  
dotnet-sdk 3.1.100 from Microsoft .NET Core (dotnetcore) installed  
briareos@briareos-vm:~$ sudo ln -s /snap/dotnet-sdk/current/dotnet /usr/local/bin/dotnet  
briareos@briareos-vm:~$
```

Une fois Snap installé, nous sommes dans la possibilité d'installer le .Net Core SDK.

Pour se faire :

1. Tapons la commande « sudo snap install dotnet-sdk --classic ».
2. Ensuite, pour être capable d'utiliser nativement l'outil « dotnet », ajoutons un lien symbolique via la commande :
« sudo ln -s /snap/dotnet-sdk/current/dotnet /usr/local/bin/dotnet ».

L'OUTIL « DOTNET » : PRÉSENTATION

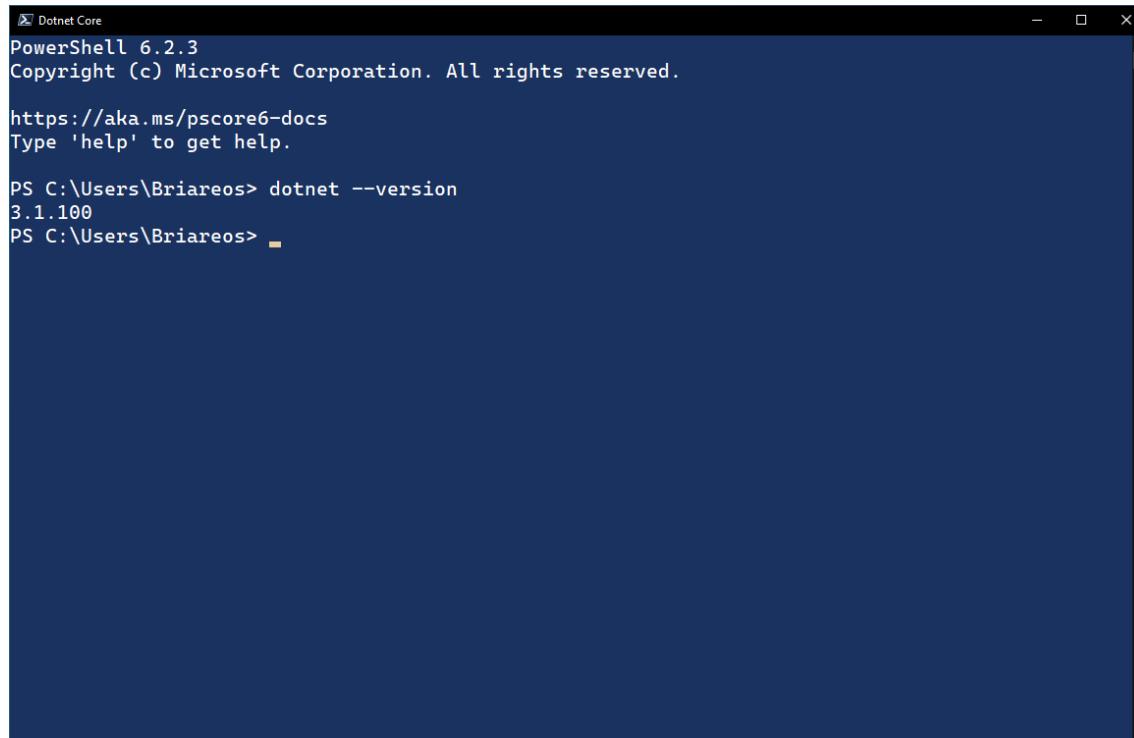
L'interface de ligne de commande (CLI) de .NET Core est un outil multiplateforme pour le développement d'applications .NET.

Cette interface en ligne de commande, est le fondement sur lequel les autres outils de niveau supérieur, tels que les « IDE », peuvent se baser.

L'utilisation de « dotnet CLI » se fait au travers de l'outil « dotnet ».



L'OUTIL « DOTNET » : DÉTERMINER LA VERSION INSTALLÉE



```
Dotnet Core
PowerShell 6.2.3
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/pscore6-docs
Type 'help' to get help.

PS C:\Users\Briareos> dotnet --version
3.1.100
PS C:\Users\Briareos>
```

Nous allons utiliser nos premières commandes pour savoir comment :

- Déterminer la version installée
- Créer un projet
- Compiler un projet
- Et démarrer le projet

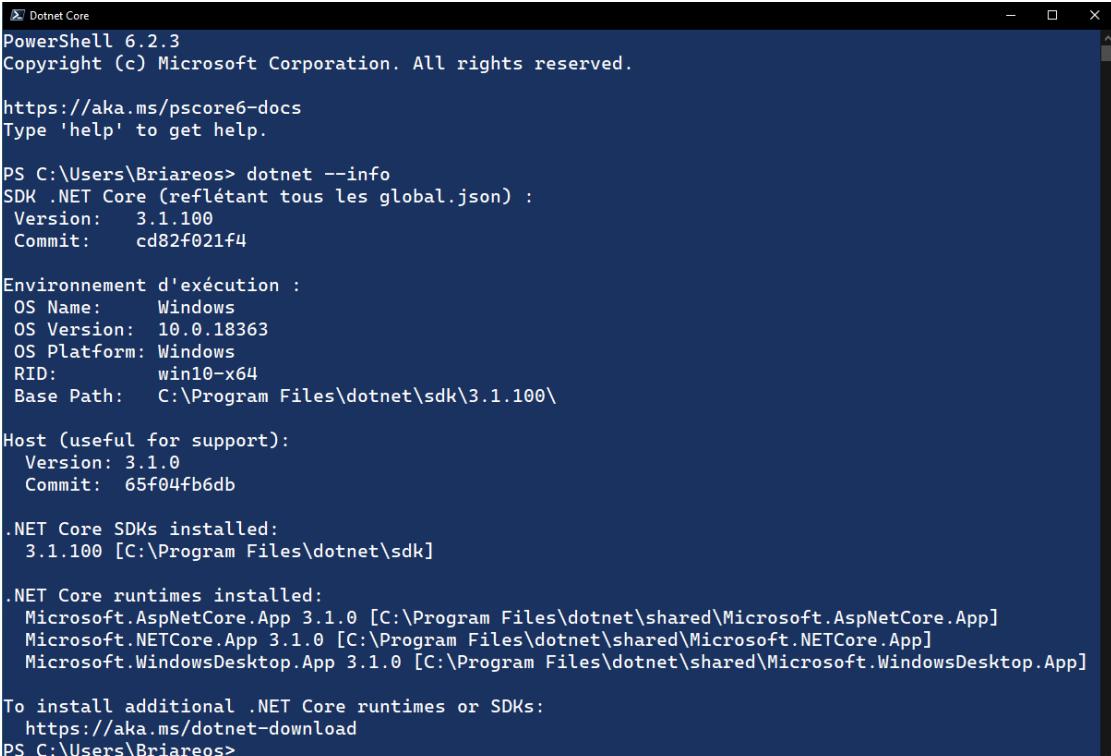
Pour connaître la plus récente version du Framework .Net Core installée, nous pouvons utiliser la commande : « dotnet --version »

L'OUTIL « DOTNET » : DÉTERMINER LA VERSION INSTALLÉE

Nous pouvons aussi utiliser la commande « dotnet --info » pour obtenir plus d'informations.

Comme :

- Les données sur le système sur lequel est installé .Net Core
- Les versions des kits de développement (SDK) installés
- Les environnements d'exécution (Runtime) installés.



```
PS C:\Users\Briareos> dotnet --info
PowerShell 6.2.3
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/pscore6-docs
Type 'help' to get help.

PS C:\Users\Briareos> dotnet --info
SDK .NET Core (réflétant tous les global.json) :
  Version: 3.1.100
  Commit: cd82f021f4

Environnement d'exécution :
  OS Name: Windows
  OS Version: 10.0.18363
  OS Platform: Windows
  RID: win10-x64
  Base Path: C:\Program Files\dotnet\sdk\3.1.100\

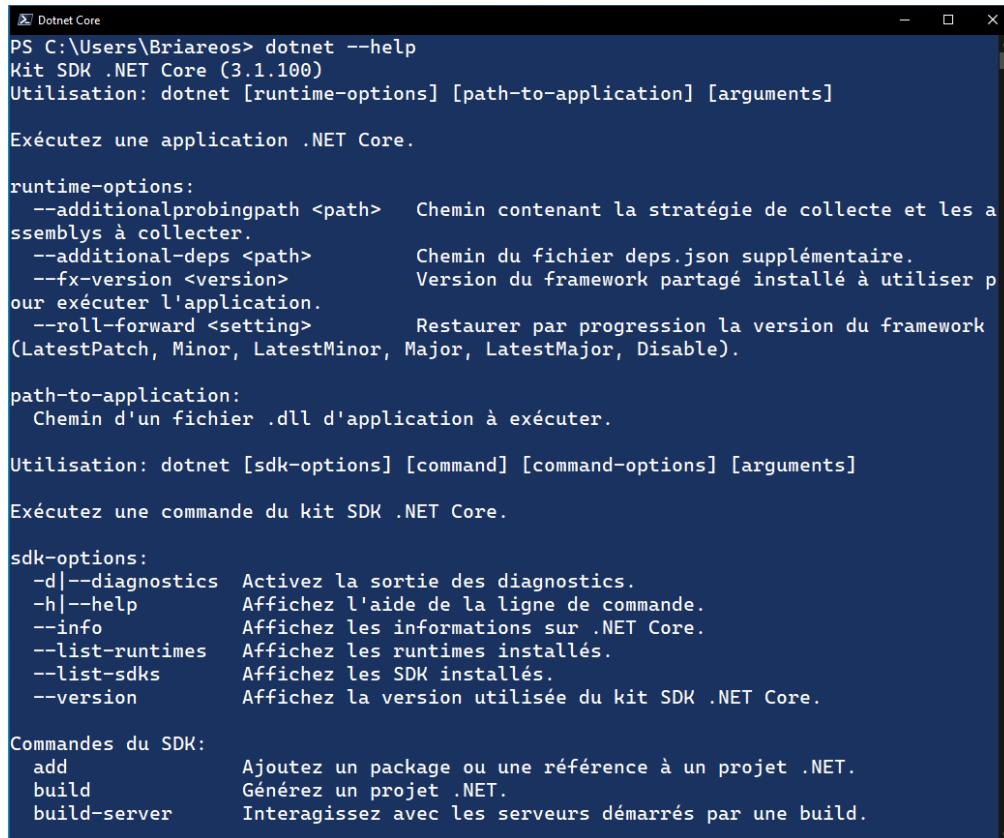
Host (utile pour le support):
  Version: 3.1.0
  Commit: 65f04fb6db

.NET Core SDKs installés:
  3.1.100 [C:\Program Files\dotnet\sdk]

.NET Core runtimes installés:
  Microsoft.AspNetCore.App 3.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 3.1.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.WindowsDesktop.App 3.1.0 [C:\Program Files\dotnet\shared\Microsoft.WindowsDesktop.App]

Pour installer des autres runtimes ou SDKs .NET Core :
  https://aka.ms/dotnet-download
PS C:\Users\Briareos>
```

L'OUTIL « DOTNET » : OBTENIR DE L'AIDE



```
PS C:\Users\Briareos> dotnet --help
Kit SDK .NET Core (3.1.100)
Utilisation: dotnet [runtime-options] [path-to-application] [arguments]

Exécuter une application .NET Core.

runtime-options:
  --additionalprobingpath <path>    Chemin contenant la stratégie de collecte et les a
ssemblys à collecter.
  --additional-deps <path>          Chemin du fichier deps.json supplémentaire.
  --fx-version <version>           Version du framework partagé installé à utiliser p
our exécuter l'application.
  --roll-forward <setting>        Restaurer par progression la version du framework
(LatestPatch, Minor, LatestMinor, Major, LatestMajor, Disable).

path-to-application:
  Chemin d'un fichier .dll d'application à exécuter.

Utilisation: dotnet [sdk-options] [command] [command-options] [arguments]

Exécuter une commande du kit SDK .NET Core.

sdk-options:
  -d|--diagnostics  Activez la sortie des diagnostics.
  -h|--help          Affichez l'aide de la ligne de commande.
  --info             Affichez les informations sur .NET Core.
  --list-runtimes   Affichez les runtimes installés.
  --list-sdks        Affichez les SDK installés.
  --version          Affichez la version utilisée du kit SDK .NET Core.

Commandes du SDK:
  add               Ajoutez un package ou une référence à un projet .NET.
  build             Générez un projet .NET.
  build-server      Interagissez avec les serveurs démarrés par une build.
```

L'outil « dotnet » inclus un système d'aide, quelque soit notre commande, il nous suffira d'ajouter « --help » à notre commande pour obtenir une aide détaillée.

« dotnet --help », par exemple, nous donne accès aux commandes basiques utilisable.

L'OUTIL « DOTNET » : CRÉER UN PROJET

Pour créer un projet, nous devrons utiliser la commande :

« dotnet new »

Mais avant de créer notre premier projet, il serait intéressant de connaître les modèles de projets que nous pouvons créer.

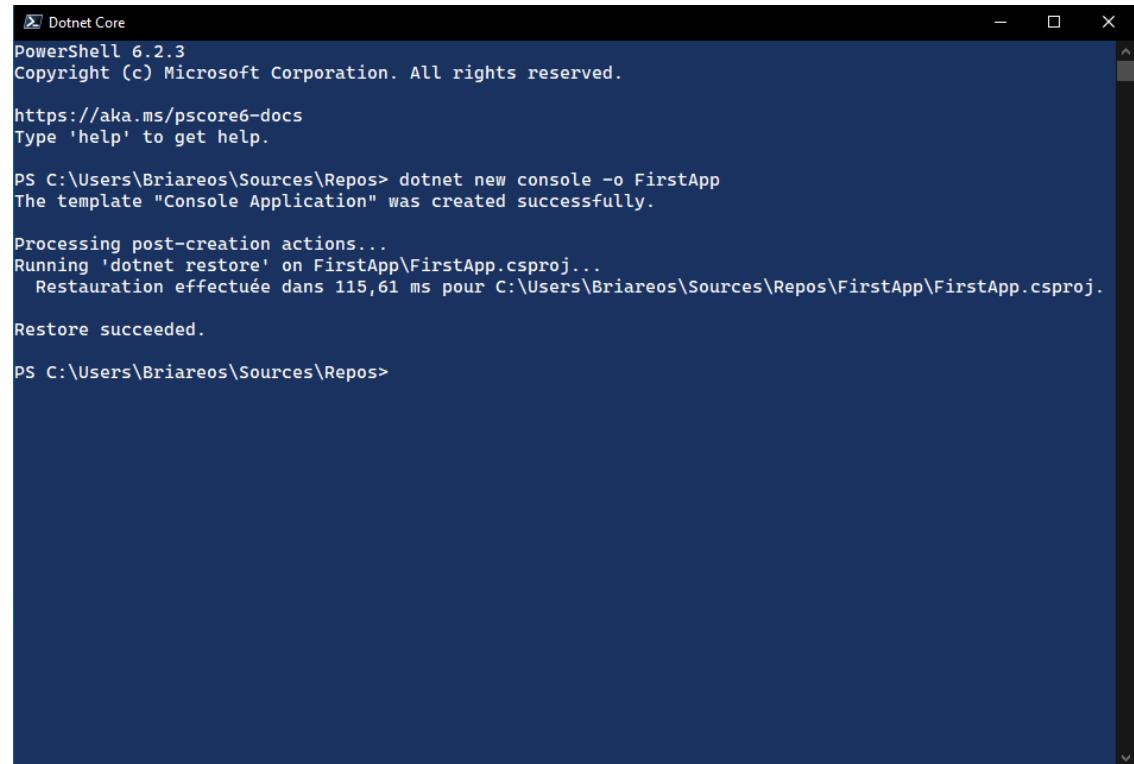
Pour cela, nous pouvons utiliser la commande :

« dotnet new --list »

Enfin, grâce à cette liste, nous allons pouvoir créer notre projet à l'aide du « Short Name ».

| Templates | Short Name | Language | Tags |
|--|---------------------|--------------|---------------------------------------|
| Console Application | console | [C#], F#, VB | Common/Console |
| Class library | classlib | [C#], F#, VB | Common/Library |
| WPF Application | wpf | [C#] | Common/WPF |
| WPF Class library | wpflib | [C#] | Common/WPF |
| WPF Custom Control Library | wpfcustomcontrollib | [C#] | Common/WPF |
| WPF User Control Library | wpfusercontentlib | [C#] | Common/WPF |
| Windows Forms (WinForms) Application | winforms | [C#] | Common/WinForms |
| Windows Forms (WinForms) Class library | winformslib | [C#] | Common/WinForms |
| Worker Service | worker | [C#] | Common/Worker/Web |
| Unit Test Project | mstest | [C#], F#, VB | Test/MSTest |
| NUnit 3 Test Project | nunit | [C#], F#, VB | Test/NUnit |
| NUnit 3 Test Item | nunit-test | [C#], F#, VB | Test/NUnit |
| xUnit Test Project | xunit | [C#], F#, VB | Test/xUnit |
| Razor Component | razorcomponent | [C#] | Web/ASP.NET |
| Razor Page | page | [C#] | Web/ASP.NET |
| MVC ViewImports | viewimports | [C#] | Web/ASP.NET |
| MVC ViewStart | viewstart | [C#] | Web/ASP.NET |
| Blazor Server App | blazorserver | [C#] | Web/Blazor |
| ASP.NET Core Empty | web | [C#], F# | Web/Empty |
| ASP.NET Core Web App (Model-View-Controller) | mvc | [C#], F# | Web/MVC |
| ASP.NET Core Web App | webapp | [C#] | Web/MVC/Razor Pages |
| ASP.NET Core with Angular | angular | [C#] | Web/MVC/SPA |
| ASP.NET Core with React.js | react | [C#] | Web/MVC/SPA |
| ASP.NET Core with React.js and Redux | reactredux | [C#] | Web/MVC/SPA |
| Razor Class Library | razorclasslib | [C#] | Web/Razor/Library/Razor Class Library |
| ASP.NET Core Web API | webapi | [C#], F# | Web/WebAPI |
| ASP.NET Core gRPC Service | grpc | [C#] | Web/gRPC |
| dotnet gitignore file | gitignore | | Config |
| global.json file | globaljson | | Config |
| NuGet Config | nugetconfig | | Config |
| Dotnet local tool manifest file | tool-manifest | | Config |
| Web Config | webconfig | | Config |
| Solution File | sln | | Solution |
| Protocol Buffer File | proto | | Web/gRPC |

L'OUTIL « DOTNET » : CRÉER UN PROJET



```
Dotnet Core
PowerShell 6.2.3
Copyright (c) Microsoft Corporation. All rights reserved.

https://aka.ms/powershell-docs
Type 'help' to get help.

PS C:\Users\Briareos\Sources\Repos> dotnet new console -o FirstApp
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on FirstApp\FirstApp.csproj...
  Restauration effectuée dans 115,61 ms pour C:\Users\Briareos\Sources\Repos\FirstApp\FirstApp.csproj.

Restore succeeded.

PS C:\Users\Briareos\Sources\Repos>
```

Afin de créer notre première application.

Commençons par la exécuter la commande :
« dotnet new console -o FirstApp »

Par cette commande, nous demandons à créer un projet de type
« Console » et nous demandons à mettre ce projet dans un répertoire
qui se nommera « FirstApp » (l'option « -o FirstApp »).

L'OUTIL « DOTNET » : COMPILER UN PROJET

Notre projet étant créé, nous devons le compiler.

Pour cela nous utiliserons l'option « build » de l'outil « dotnet » après s'être placé dans le répertoire du projet :

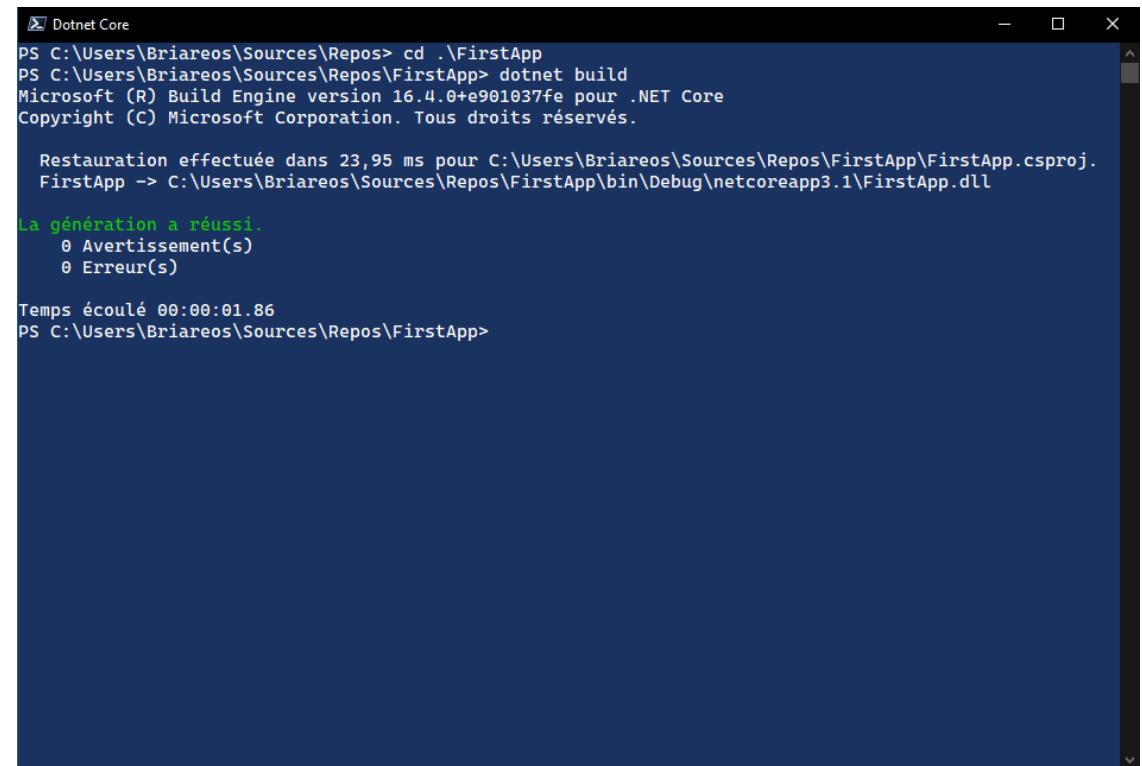
« cd FirstApp »

Suivi de :

« dotnet build »

Il est possible de ne pas se placer dans le projet mais dans ce cas nous devrons spécifier le chemin.

« dotnet build .\FirstApp »



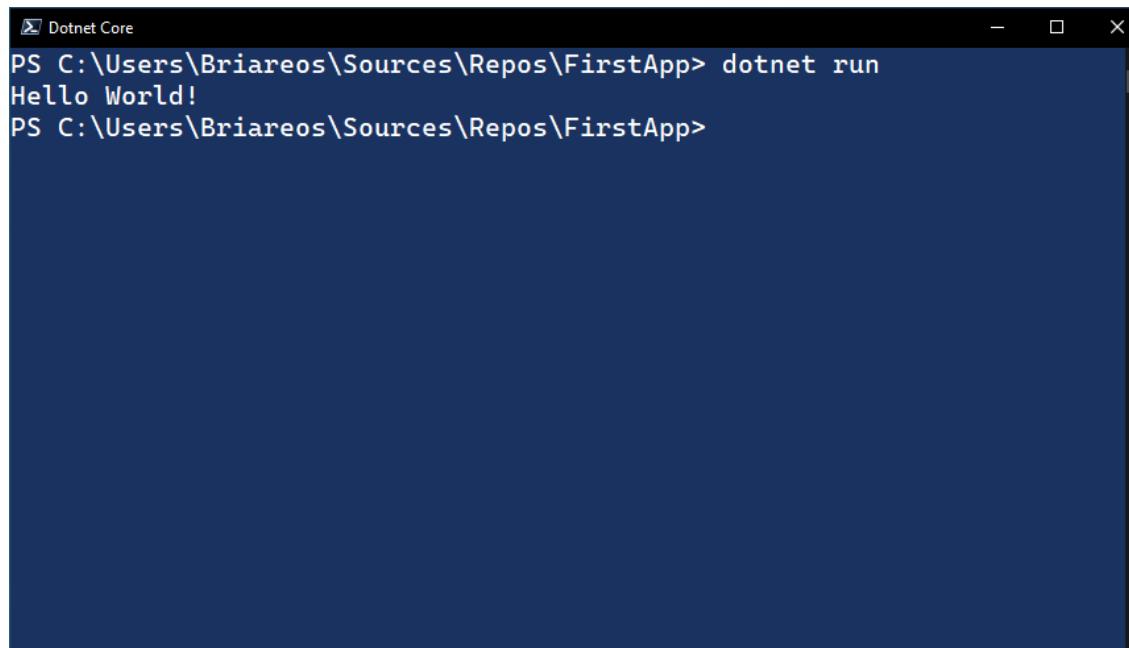
```
PS C:\Users\Briareos\Sources\Repos> cd .\FirstApp
PS C:\Users\Briareos\Sources\Repos\FirstApp> dotnet build
Microsoft (R) Build Engine version 16.4.0+e901037fe pour .NET Core
Copyright (C) Microsoft Corporation. Tous droits réservés.

Restauration effectuée dans 23,95 ms pour C:\Users\Briareos\Sources\Repos\FirstApp\FirstApp.csproj.
FirstApp -> C:\Users\Briareos\Sources\Repos\FirstApp\bin\Debug\netcoreapp3.1\FirstApp.dll

La génération a réussi.
  0 Avertissement(s)
  0 Erreur(s)

Temps écoulé 00:00:01.86
PS C:\Users\Briareos\Sources\Repos\FirstApp>
```

L'OUTIL « DOTNET » : EXÉCUTER UN PROJET



```
Dotnet Core
PS C:\Users\Briareos\Sources\Repos\FirstApp> dotnet run
Hello World!
PS C:\Users\Briareos\Sources\Repos\FirstApp>
```

Enfin, pour terminer, il nous reste à exécuter notre projet au travers de l'outil dotnet.

Via la commande : « dotnet run »

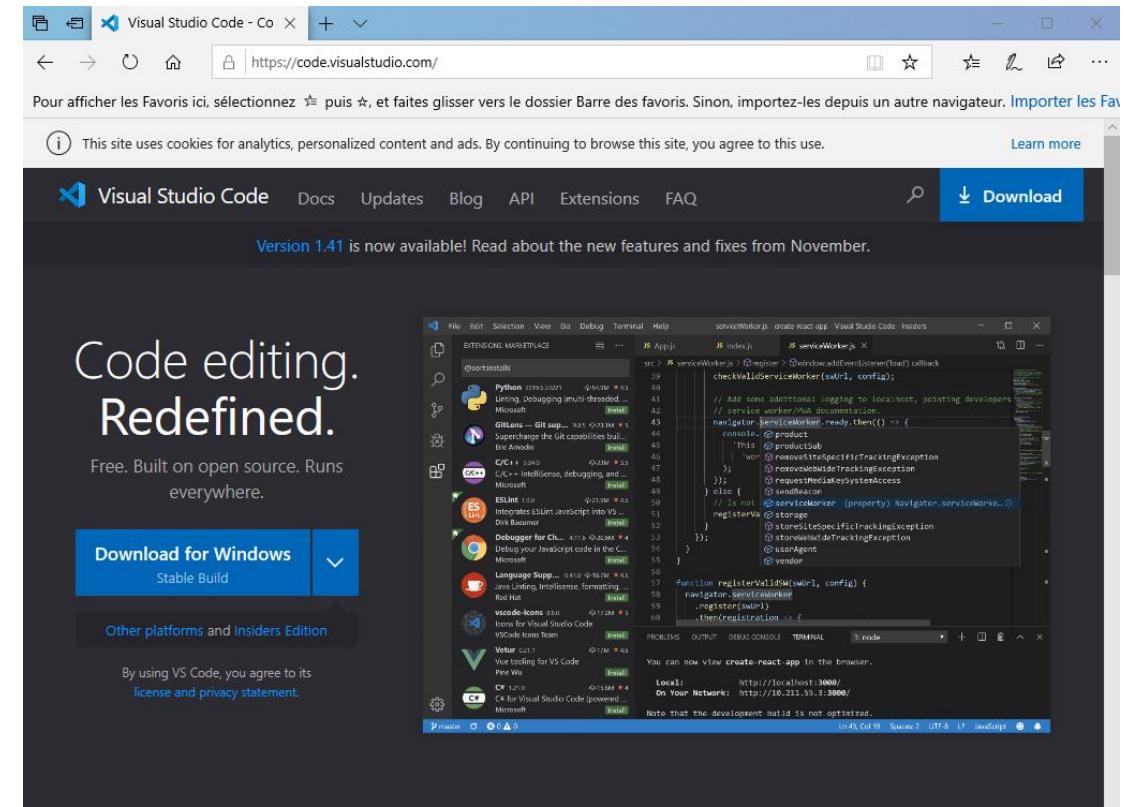
AVEC VISUAL STUDIO CODE : PRÉSENTATION

Visual Studio Code est un éditeur de code source léger mais puissant qui est disponible pour Windows, macOS et Linux.

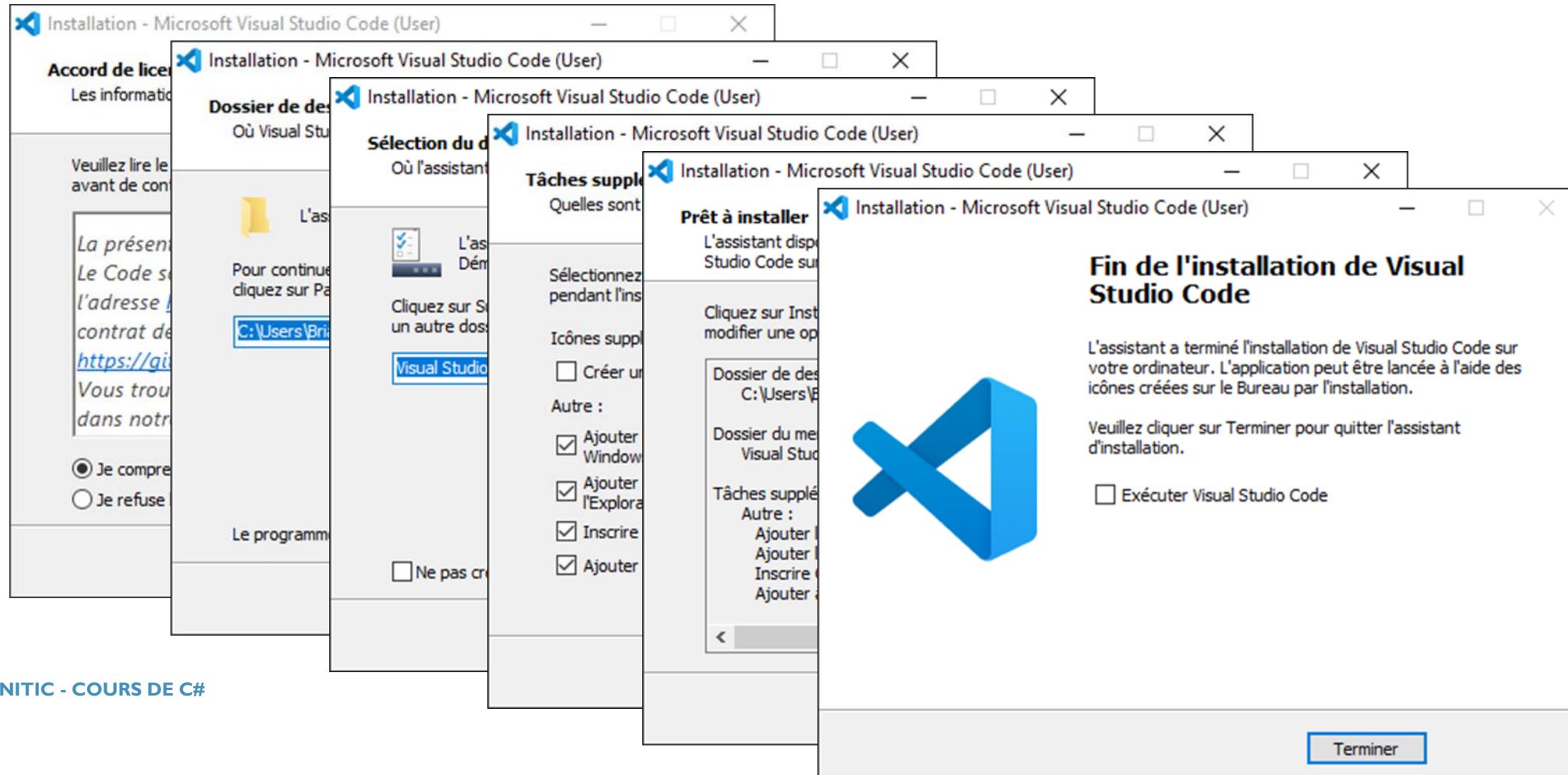
Il est livré avec un support intégré pour JavaScript, TypeScript et Node.js et possède un riche écosystème d'extensions pour d'autres langages (tels que C++, C#, Java, Python, PHP, Go) et des runtimes (tels que .NET et Unity)

Ce dernier peut-être téléchargé sur le site :

<https://code.visualstudio.com/>



AVEC VISUAL STUDIO CODE : INSTALLATION SOUS WINDOWS



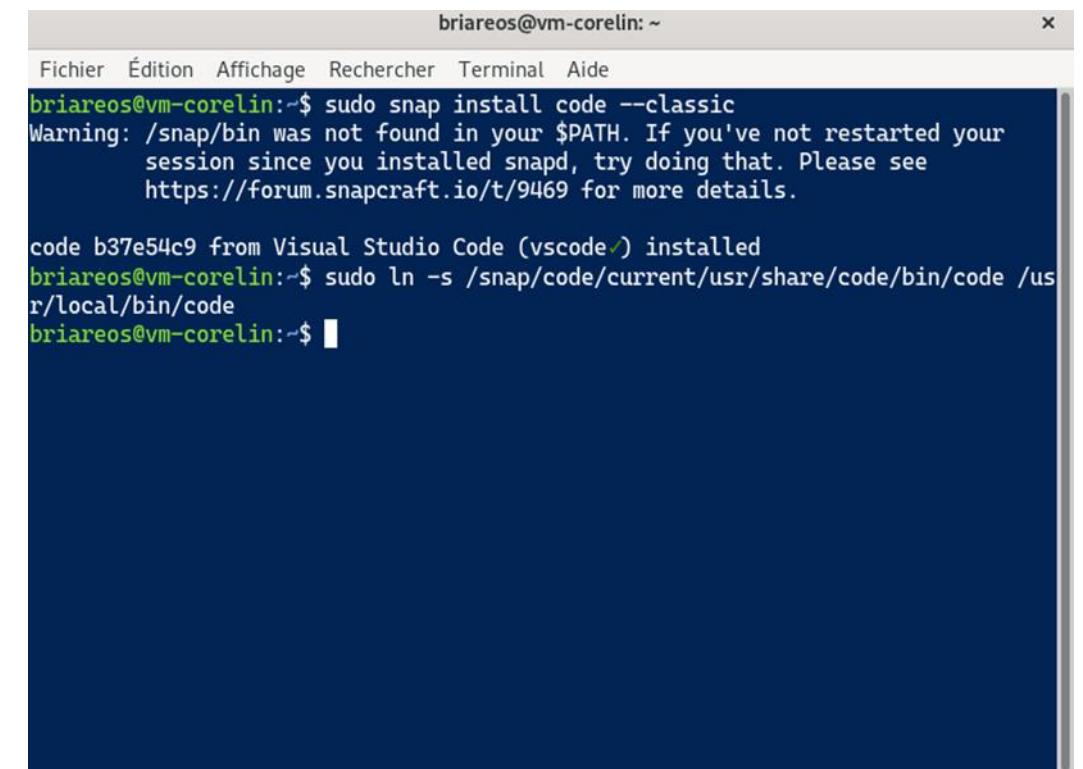
AVEC VISUAL STUDIO CODE : INSTALLATION SOUS LINUX

Sous Linux, nous allons utiliser la commande :

« sudo snap install code --classic »

Une fois installé, nous allons lui créer aussi un lien symbolique en utilisant cette fois la commande :

« sudo ln -s /snap/code/current/usr/share/code/bin/code
/usr/local/bin/code »

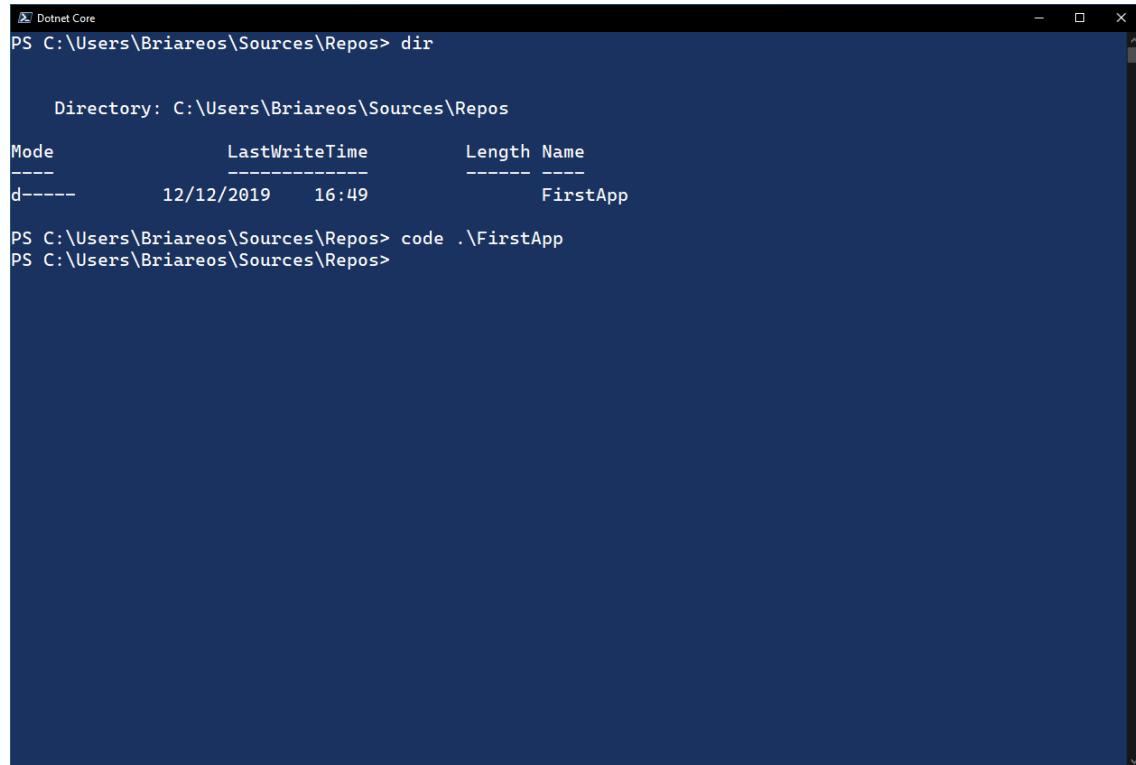


The screenshot shows a terminal window titled "briareos@vm-corelin: ~". The window includes a menu bar with "Fichier", "Édition", "Affichage", "Rechercher", "Terminal", and "Aide". The terminal itself displays the following command-line session:

```
Fichier Édition Affichage Rechercher Terminal Aide
briareos@vm-corelin:~$ sudo snap install code --classic
Warning: /snap/bin was not found in your $PATH. If you've not restarted your
session since you installed snapd, try doing that. Please see
https://forum.snapcraft.io/t/9469 for more details.

code b37e54c9 from Visual Studio Code (vscode✓) installed
briareos@vm-corelin:~$ sudo ln -s /snap/code/current/usr/share/code/bin/code /us
r/local/bin/code
briareos@vm-corelin:~$ █
```

AVEC VISUAL STUDIO CODE : OUVRIR LE PROJET



```
PS C:\Users\Briareos\Sources\Repos> dir

Directory: C:\Users\Briareos\Sources\Repos

Mode                LastWriteTime         Length Name
----              -----           -----    Name
d----       12/12/2019   16:49            FirstApp

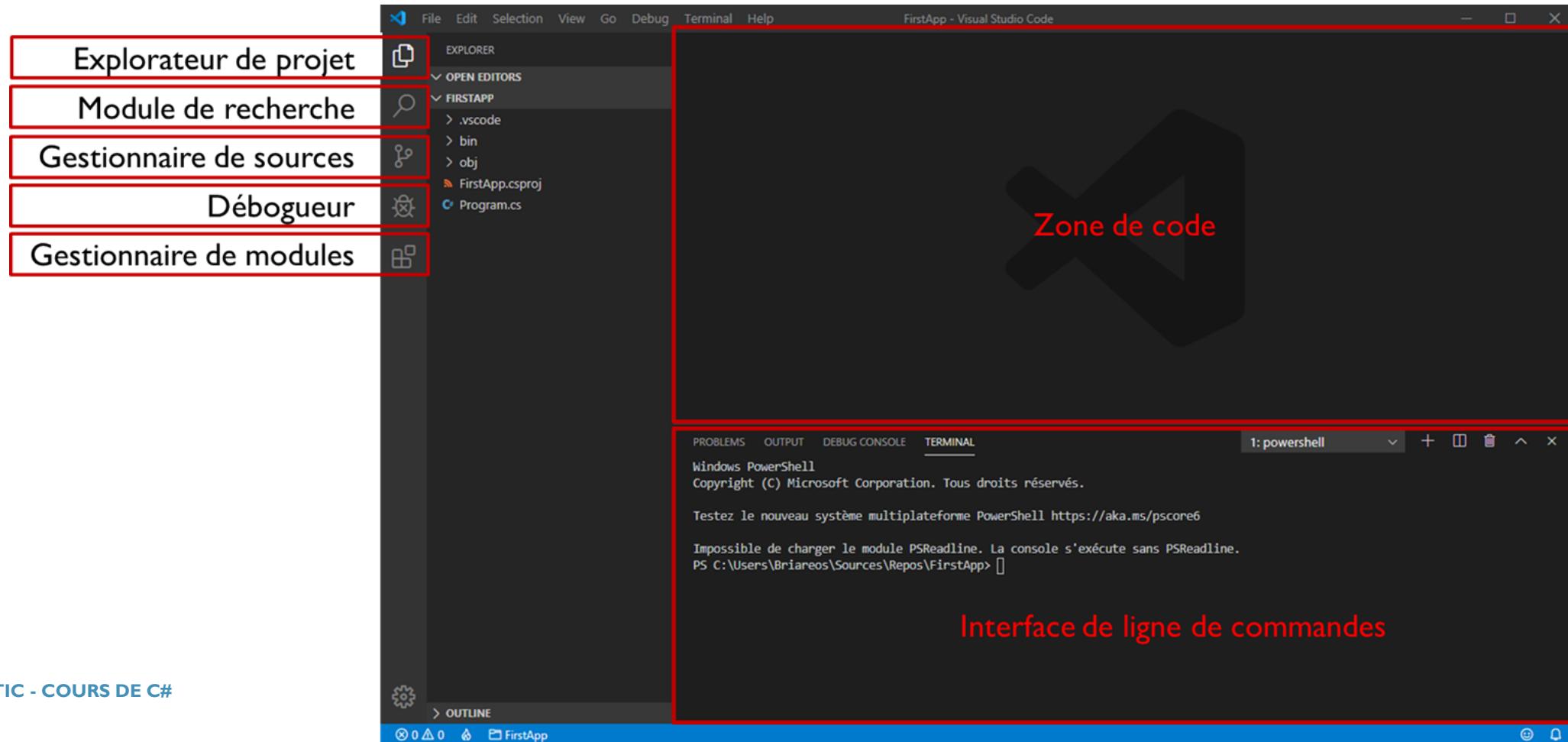
PS C:\Users\Briareos\Sources\Repos> code .\FirstApp
PS C:\Users\Briareos\Sources\Repos>
```

Pour ouvrir notre projet, toujours depuis notre interface ligne de commandes, utilisons la commande :

« code » si nous sommes dans le répertoire du projet
ou

« code [chemin du projet] » si nous sommes en dehors

AVEC VISUAL STUDIO CODE : PRÉSENTATION DE L'INTERFACE

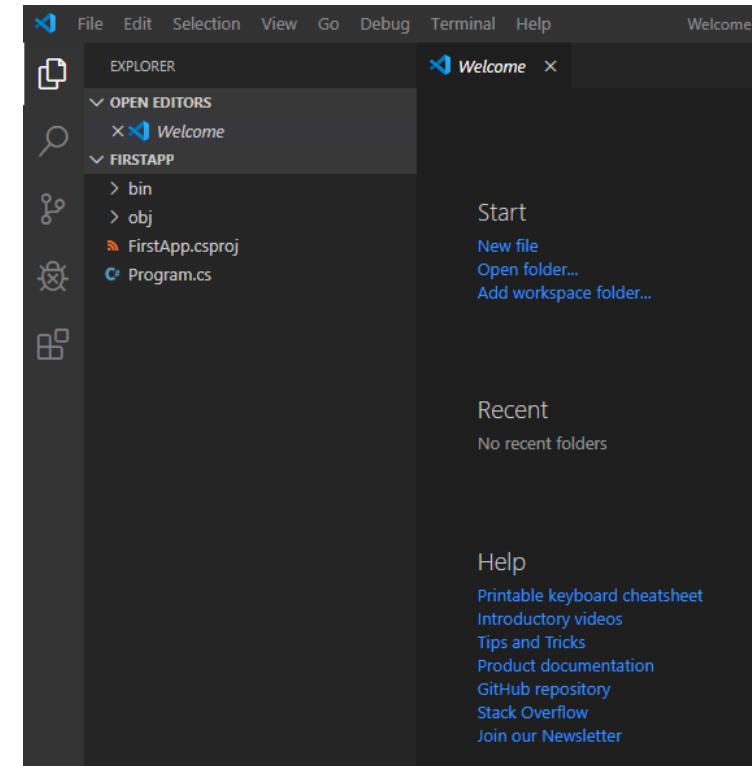


AVEC VISUAL STUDIO CODE : ANATOMIE

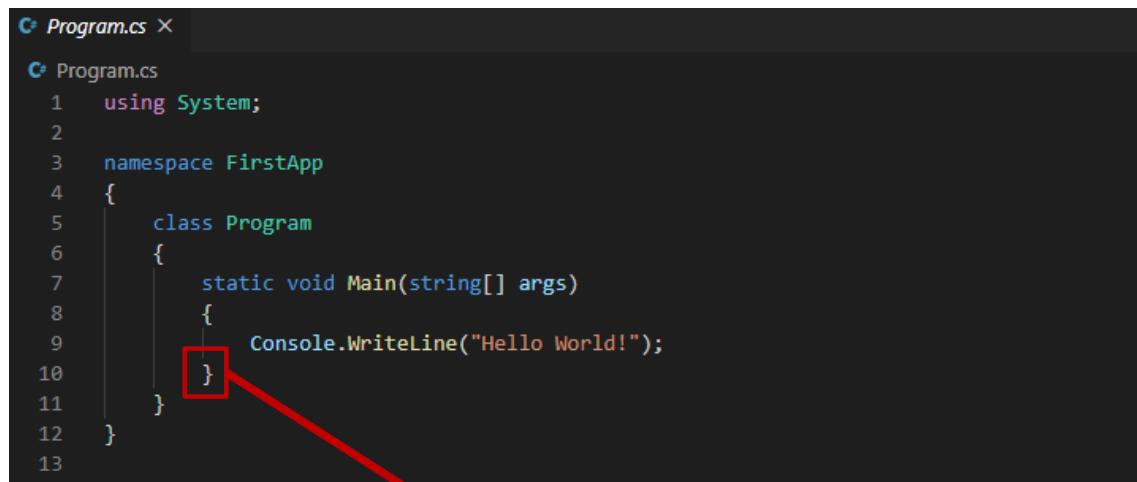
Une fois Visual Studio Code ouvert, dans la partie « explorateur de projet », nous trouvons deux fichiers.

Le premier « FirstApp.csproj » est le fichier de configuration de notre projet, il s'agit d'un fichier XML qui reprend la configuration de notre projet.

```
<Project Sdk="Microsoft.NET.Sdk">  
  
    <PropertyGroup>  
        <OutputType>Exe</OutputType>  
        <TargetFramework>netcoreapp3.1</TargetFramework>  
    </PropertyGroup>  
  
</Project>
```



AVEC VISUAL STUDIO CODE : ANATOMIE



```
C# Program.cs x
C# Program.cs
1  using System;
2
3  namespace FirstApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("Hello World!");
10         }
11     }
12 }
13 }
```

A screenshot of the Visual Studio Code interface. The title bar shows 'C# Program.cs x'. The code editor displays a C# program. A red rectangular box highlights the closing brace '}' on line 10, which is the end of the 'Main' method. The code itself is as follows:

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

Fin du bloc d'instruction

Le deuxième « Program.cs » va contenir les actions à réaliser de notre applicatif.

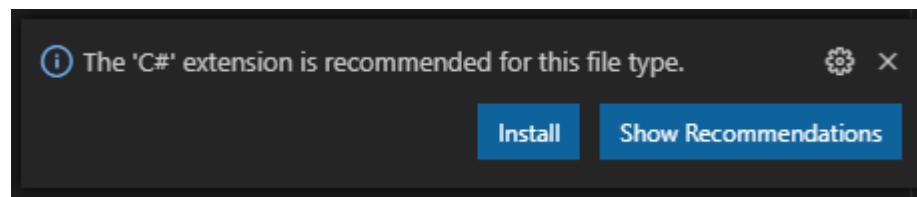
Un programme écrit en C# doit contenir une méthode « Main ».

Lorsque cette méthode est appelée, notre programme commence et lorsque nous atteignons la fin du bloc d'instruction de cette dernière, le programme s'arrête.

C'est donc, dans un premier temps, à l'intérieur de cette méthode que nous spécifierons notre logique métier.

AVEC VISUAL STUDIO CODE : CONFIGURATION

Au moment où nous avons ouvert un fichier, Visual Studio Code a détecté que notre projet était un projet C# et nous propose donc d'installer le Module « C# for Visual Studio Code »



En cliquant sur installer, l'IDE va ouvrir une console pour installer les outils nécessaire au développement d'application .Net Core en C# avec Visual Studio Code.

A screenshot of the Visual Studio Code terminal window. The title bar shows "C#" and the tabs "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", and "TERMINAL", with "OUTPUT" being the active tab. The terminal output shows the following text:

```
Installing C# dependencies...
Platform: win32, x86_64

Downloading package 'OmniSharp for Windows (.NET 4.6 / x64)' (32059 KB)... Done!
Validating download...
Integrity Check succeeded.
Installing package 'OmniSharp for Windows (.NET 4.6 / x64)'

Downloading package '.NET Core Debugger (Windows / x64)' (47489 KB)... Done!
Validating download...
Integrity Check succeeded.
Installing package '.NET Core Debugger (Windows / x64)'

Downloading package 'Razor Language Server (Windows / x64)' (50372 KB)... Done!
Installing package 'Razor Language Server (Windows / x64)'

Finished
```

AVEC VISUAL STUDIO CODE : EXÉCUTION

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Debug, Terminal, Help.
- Terminal:** Program.cs - FirstApp - Visual Studio Code
- Code Editor:** Program.cs (FirstApp) - Main(string[] args)

```
1 using System;
2
3 namespace FirstApp
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             Console.WriteLine("Hello World!");
10        }
11    }
12}
```
- Left Sidebar:**
 - DEBUG AND RUN (.NET)
 - VARIABLES (Locals: args [string[]]: {string[0]})
 - WATCH
 - CALL STACK (PAUSED ON BREAKPOINT: FirstApp.dll!FirstApp.Program.Main [Code externe] Unknown Source)
 - BREAKPOINTS (Toutes les exceptions, Exceptions non prises en charge r... checked, Program.cs checked)
- Bottom Status Bar:** 0 ▲ 0 .NET Core Launch (console) (FirstApp) ⏪ FirstApp Ln 10, Col 10 (1 selected) Spaces: 4 UTF-8 with BOM CRLF C# ⌂

Pour exécuter notre projet, nous appuierons sur la touche « F5 » pour lancer notre projet en mode « Débogage » qui sera vu un peu plus loin dans ce cours.

VISUAL STUDIO

C# - LES FONDEMENTS

- Présentation des outils
- Visual Studio Code
- Visual Studio
- La gestion du code

VISUAL STUDIO

PRÉSENTATION

Visual Studio est un IDE (Integrated Development Environment) prévu pour développer des applications .Net.

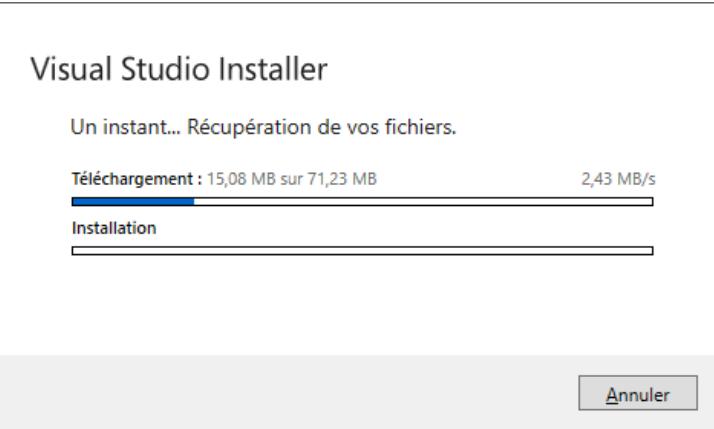
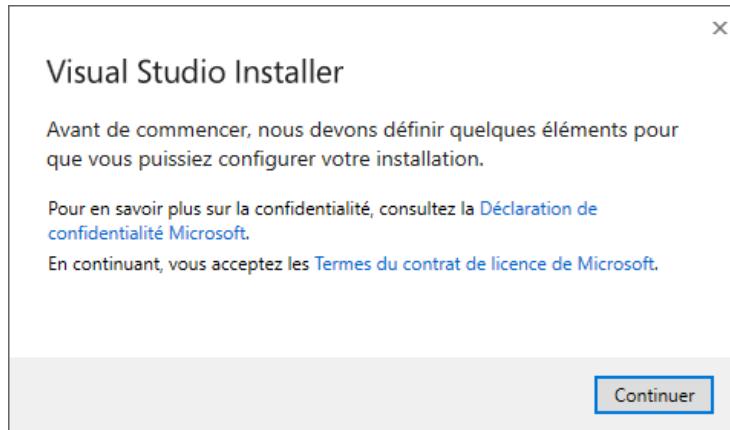
Actuellement la version 2019 se décline en 4 versions :

- Entreprise
- Professionnelle
- Community (Gratuite)
- Online (Environnements de développement dans le cloud)



Visual
Studio

INSTALLATION



L'installation de Visual Studio se fait par l'outil « Visual Studio Installer »,
Cet outil est téléchargeable sur :

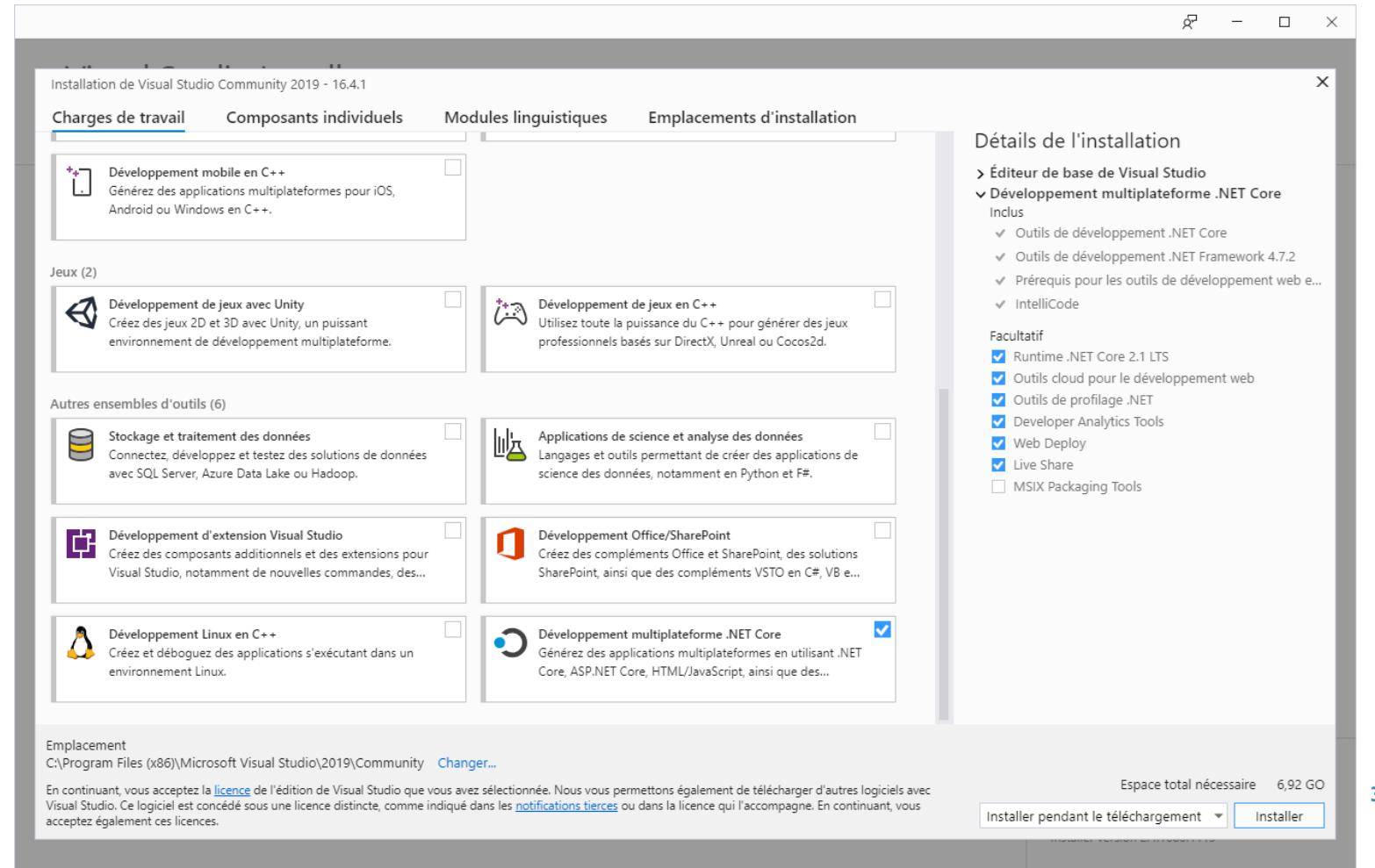
<https://visualstudio.microsoft.com/fr/>

Sur ce site prenons la version « Community ».

INSTALLATION

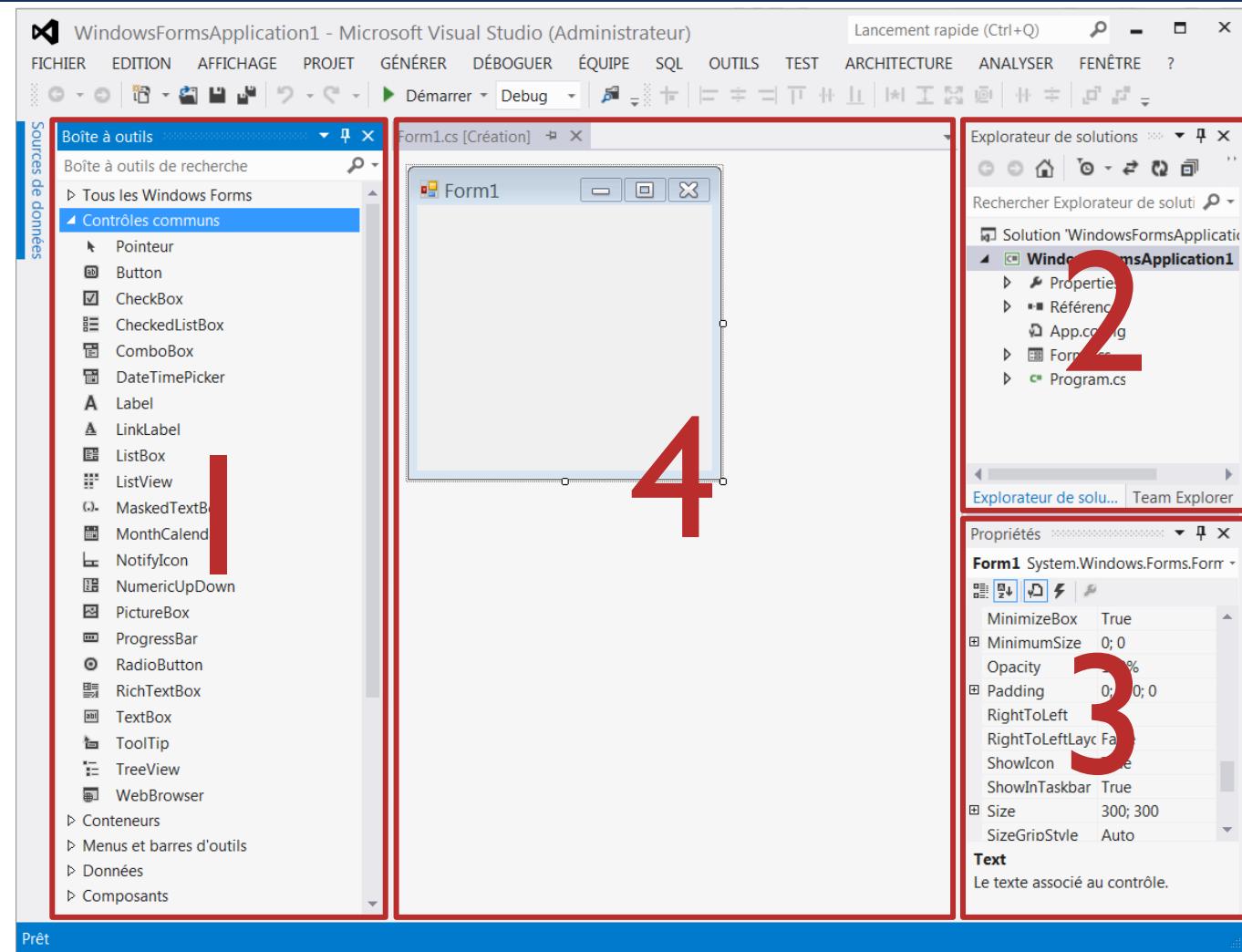
Sélectionnons au minimum le package
« Développement multiplateforme .Net Core »

Ensuite, validons en cliquant sur « Installer »

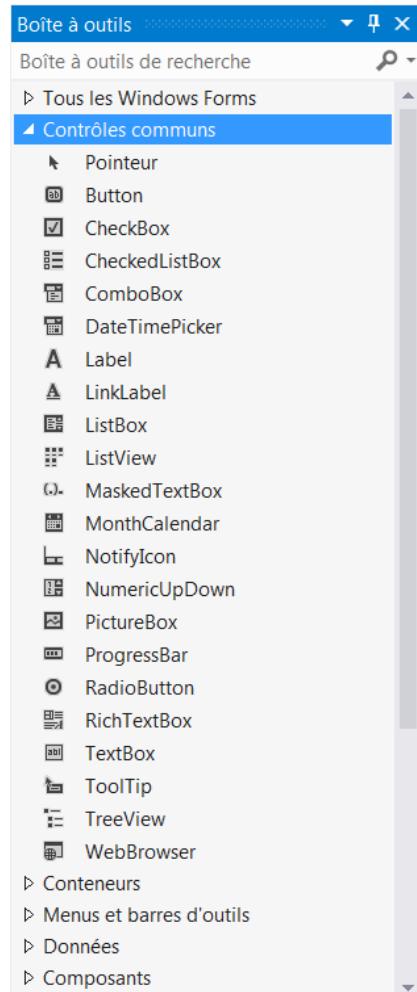


PRÉSENTATION DE L'INTERFACE

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

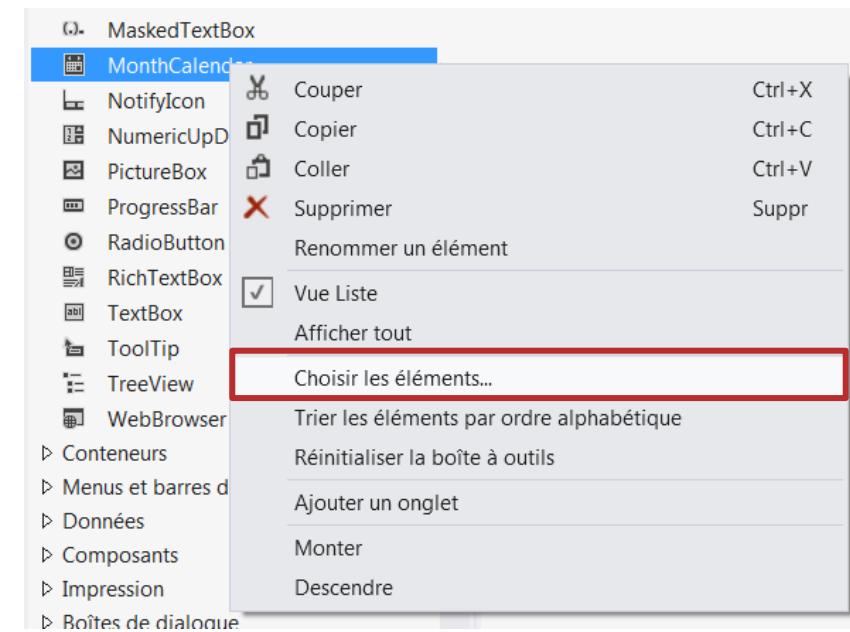


PRÉSENTATION DE L'INTERFACE - LA BOÎTE À OUTILS



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,

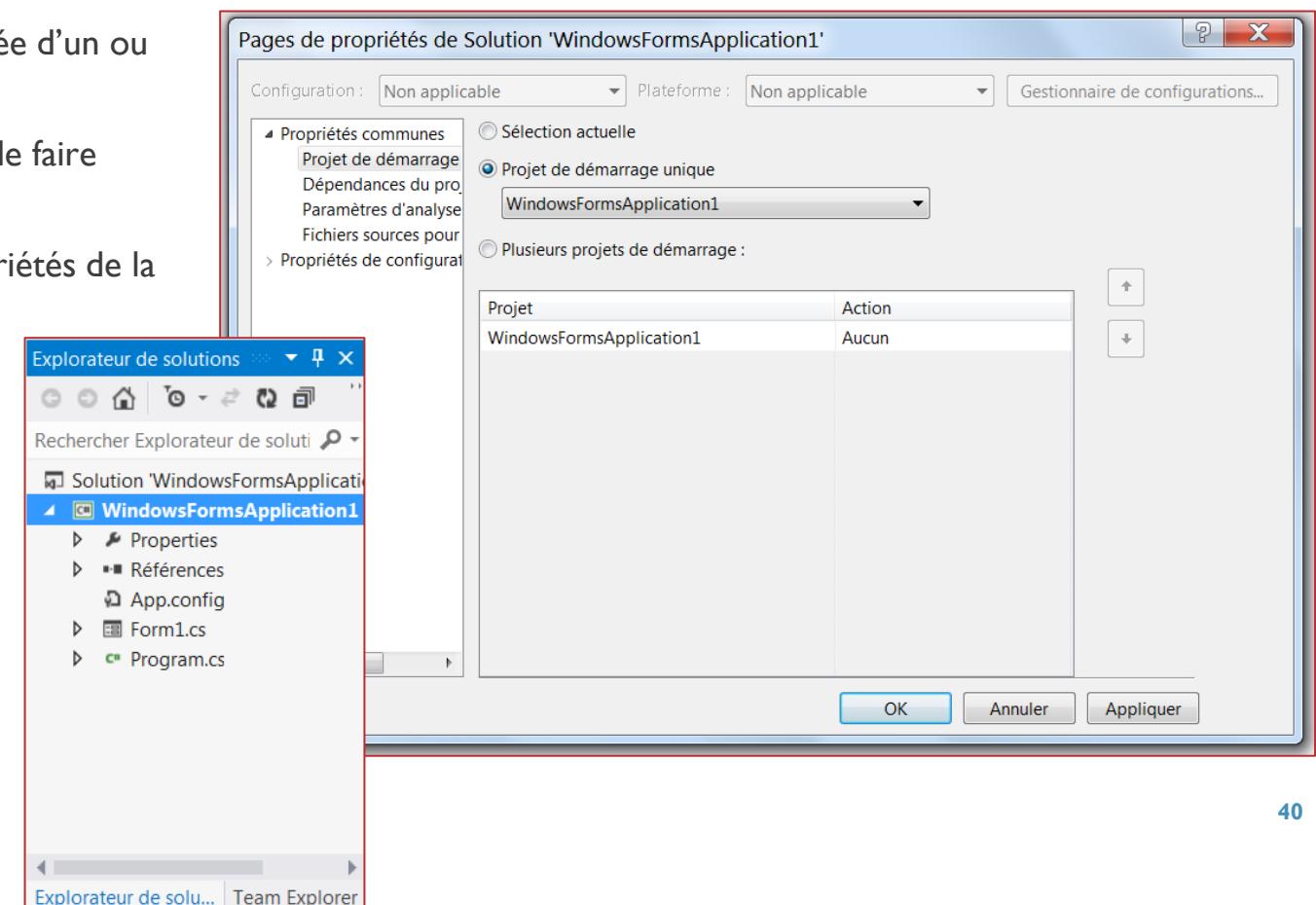


PRÉSENTATION DE L'INTERFACE - L'EXPLORATEUR DE SOLUTION

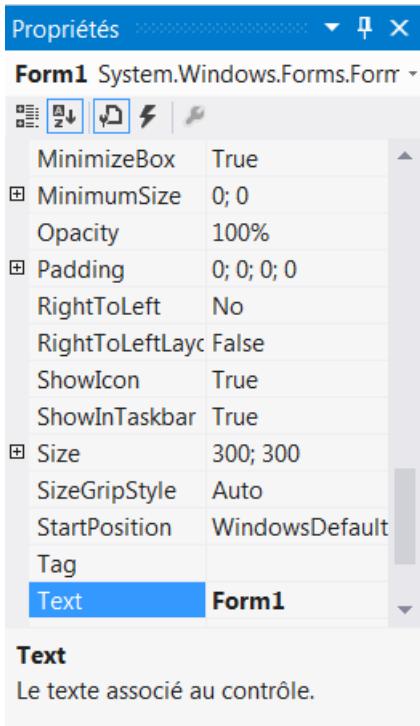
Le concept de solution est propre à notre IDE et est composée d'un ou plusieurs projets.

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.

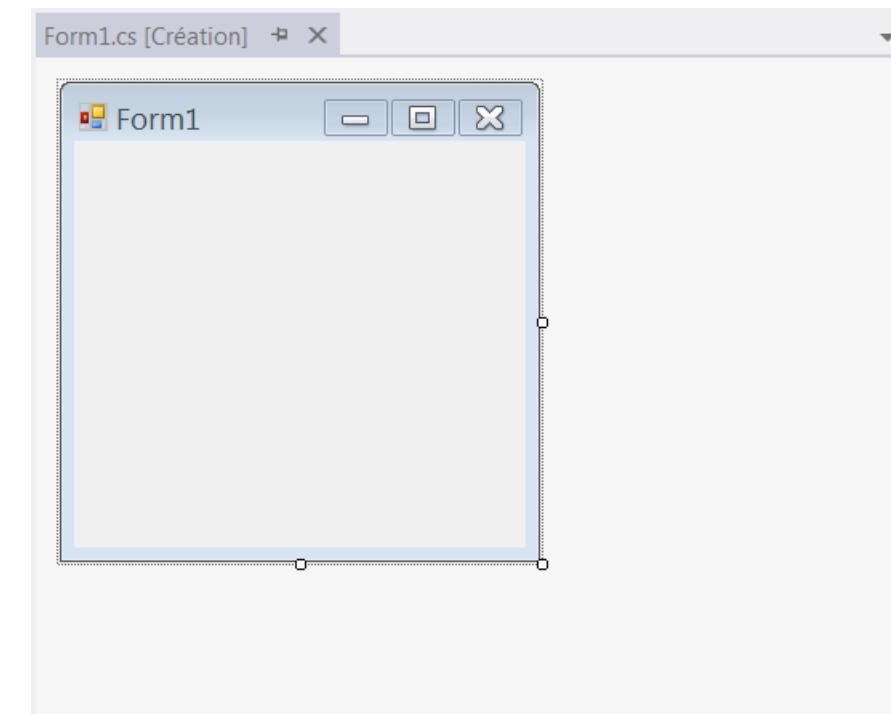
En plus de cela, Visual Studio sauvegarde votre configuration afin d'afficher la même disposition des fenêtres lorsque vous rouvrirez votre projet.

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

Les méthodes de l'objet

Possibilité de réduire certaine partie de code

The diagram shows a code editor window with a C# file named Form1.cs. The code defines a Windows Form application with a single form named Form1.

```
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;

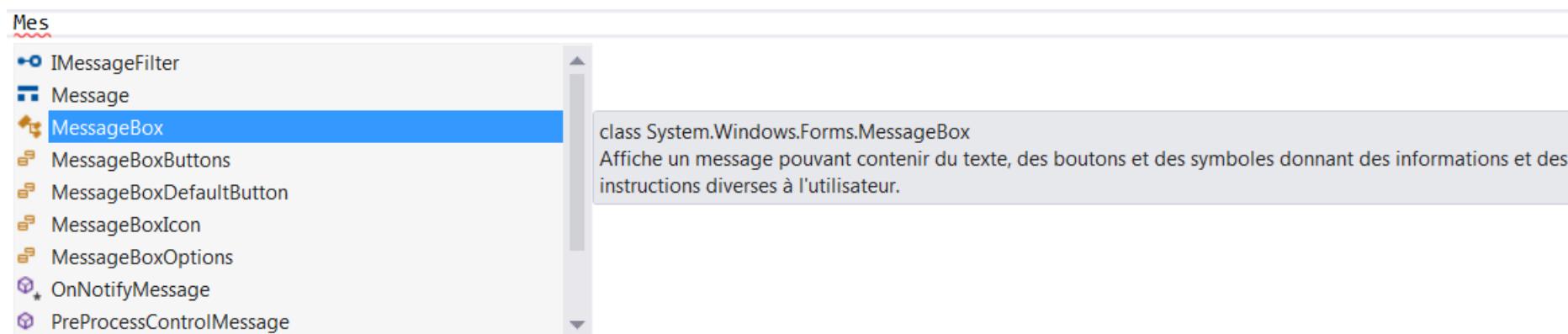
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Annotations in the code editor:

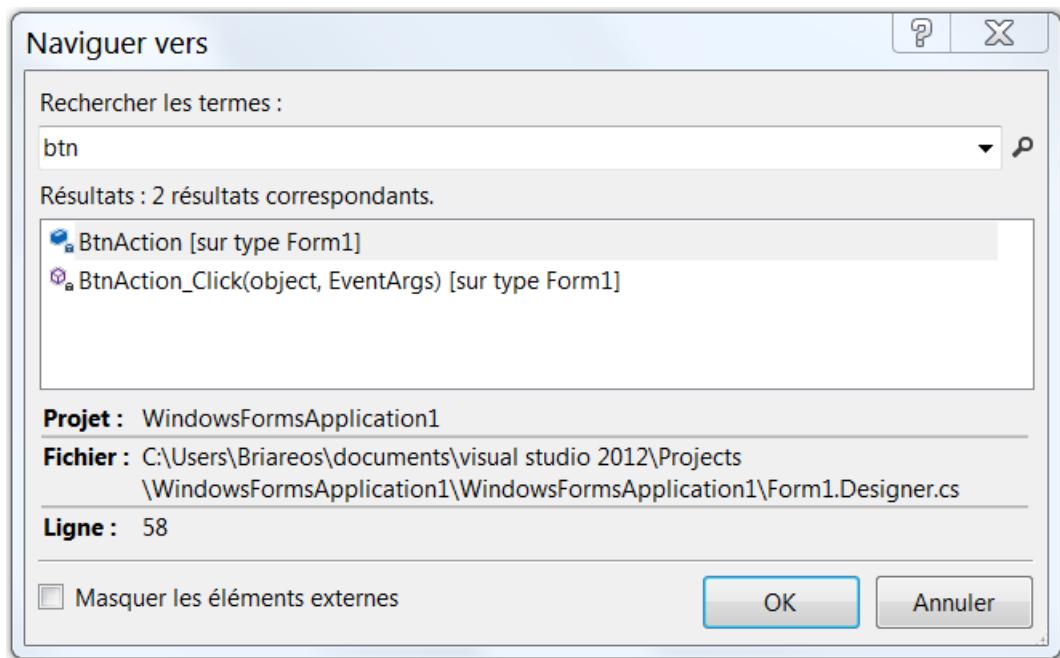
- A red bracket on the left side of the code editor highlights the entire code area, labeled "Possibilité de réduire certaine partie de code".
- A red bracket on the right side of the code editor highlights the "using" statements at the top, labeled "Les « using »".
- A red bracket on the right side of the code editor highlights the class definition and its constructor, labeled "Blocs d'instructions".

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 - ATTEINDRE LA DÉFINITION



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");
}

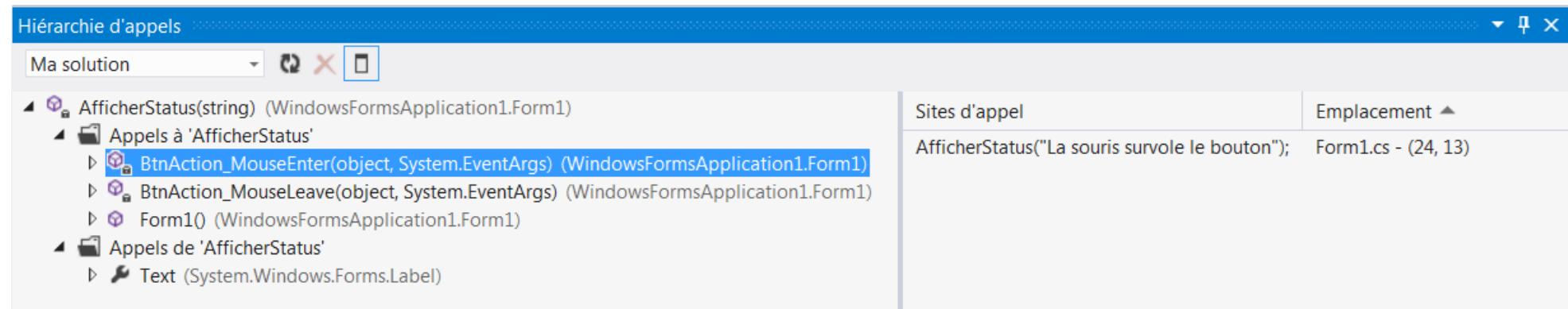
private void AfficherStatus(string Status)
{
    label1.Text = Status;
}
```

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 »).



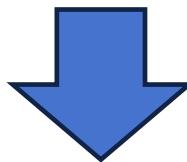
LA GESTION DU CODE - GÉNÉRATION AUTOMATIQUE

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

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



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

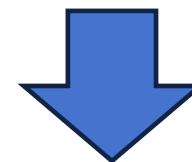


```
private int Addition(int p1, int p2)
{
    throw new NotImplementedException();
}
```

```
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

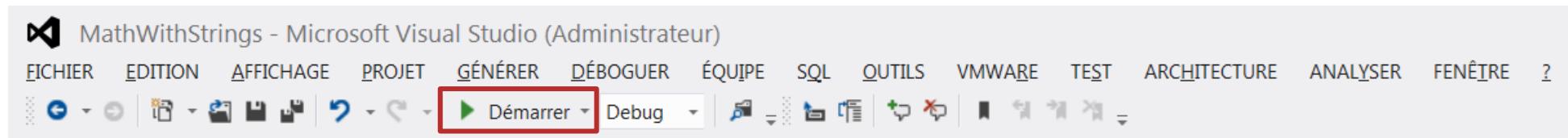
C# - LES FONDEMENTS

- Les raccourcis courants
- Les points d'arrêts
- Les variables locaux
- Les espions

DÉBOGUER SON APPLICATION

LES RACCOURCIS COURANTS

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

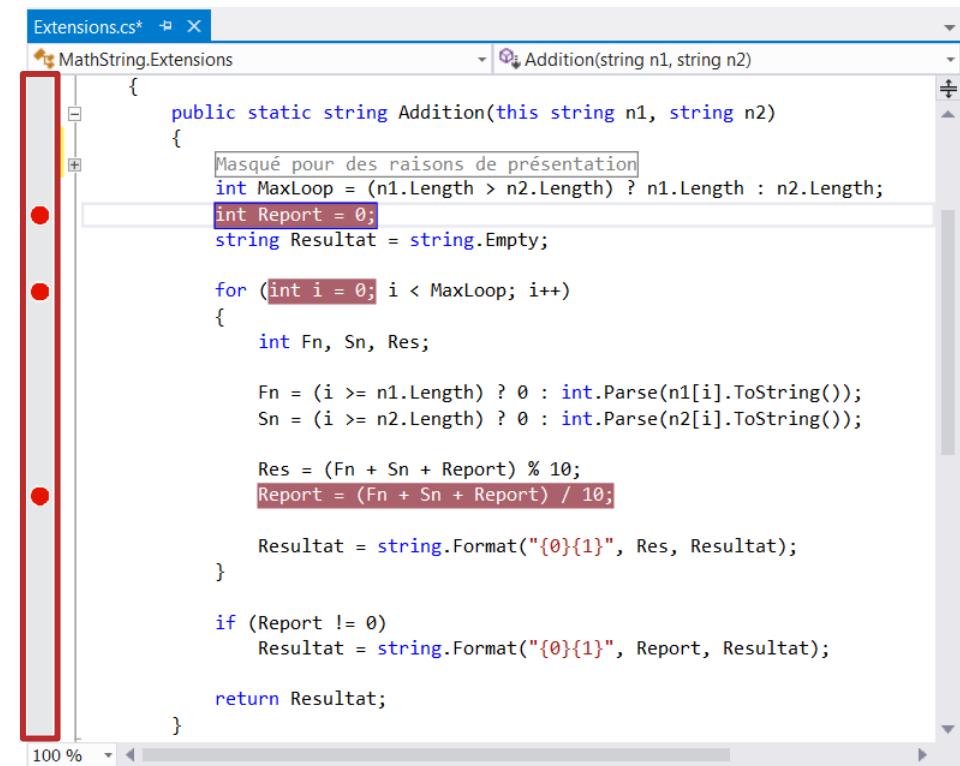
LES POINTS D'ARRÊTS

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.



```
Extensions.cs*  X
MathString.Extensions
Add(string n1, string n2)

{
    public static string Addition(this string n1, string n2)
    {
        // Masqué pour des raisons de présentation
        int MaxLoop = (n1.Length > n2.Length) ? n1.Length : n2.Length;
        int Report = 0;
        string Resultat = string.Empty;

        for (int i = 0; i < MaxLoop; i++)
        {
            int Fn, Sn, Res;

            Fn = (i >= n1.Length) ? 0 : int.Parse(n1[i].ToString());
            Sn = (i >= n2.Length) ? 0 : int.Parse(n2[i].ToString());

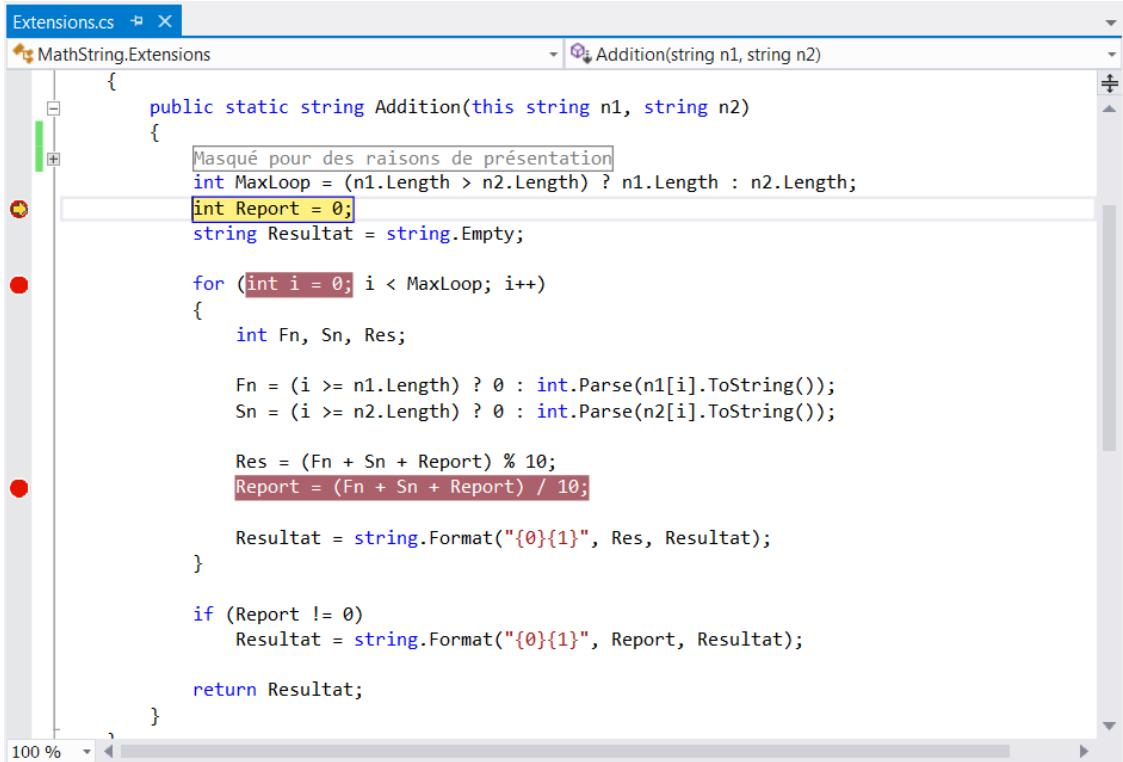
            Res = (Fn + Sn + Report) % 10;
            Report = (Fn + Sn + Report) / 10;

            Resultat = string.Format("{0}{1}", Res, Resultat);
        }

        if (Report != 0)
            Resultat = string.Format("{0}{1}", Report, Resultat);

        return Resultat;
    }
}
```

LES POINTS D'ARRÊTS



The screenshot shows a code editor window for a file named 'Extensions.cs'. The code defines a static method 'Addition' that adds two strings character by character. A yellow arrow-shaped breakpoint is visible on the left margin at line 12, indicating the next point of execution. The code includes comments, variable declarations, and a loop that handles digit addition and carries over.

```
Extensions.cs
MathString.Extensions
    public static string Addition(string n1, string n2)
    {
        int MaxLoop = (n1.Length > n2.Length) ? n1.Length : n2.Length;
        int Report = 0;
        string Resultat = string.Empty;

        for (int i = 0; i < MaxLoop; i++)
        {
            int Fn, Sn, Res;

            Fn = (i >= n1.Length) ? 0 : int.Parse(n1[i].ToString());
            Sn = (i >= n2.Length) ? 0 : int.Parse(n2[i].ToString());

            Res = (Fn + Sn + Report) % 10;
            Report = (Fn + Sn + Report) / 10;

            Resultat = string.Format("{0}{1}", Res, Resultat);
        }

        if (Report != 0)
            Resultat = string.Format("{0}{1}", Report, Resultat);

        return Resultat;
    }

```

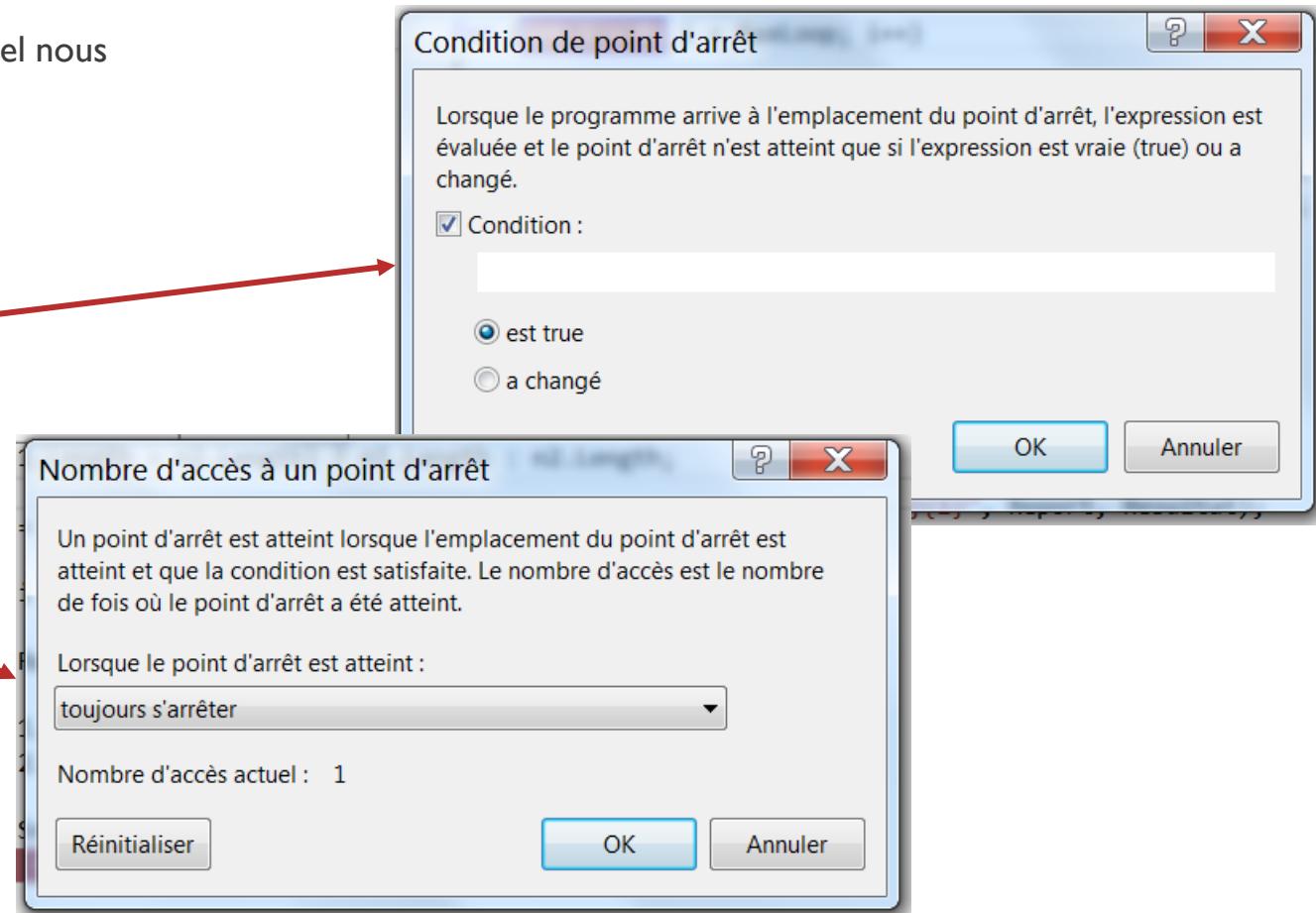
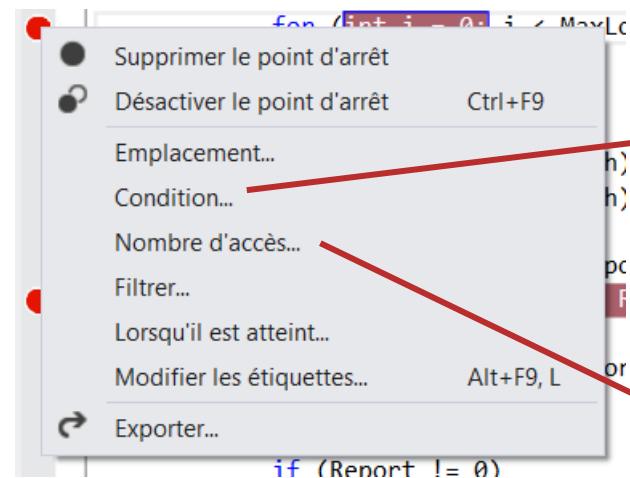
En mode débogage, lorsque le programme s'arrête sur un breakpoint, cela est représenté par une flèche jaune.

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.

LES POINTS D'ARRÊTS

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



LES 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.

| Nom | Valeur | Type |
|-------------------|--|---------------|
| n1 | "987654321987654321987654321987654321987654321987654321" | string |
| n2 | "321987654321987654321987654321987654321987654321987654" | string |
| Fn | 8 | int |
| Sn | 2 | int |
| Res | 2 | int |
| i | 1 | int |
| RegularExpression | "^\\d+\$" | string |
| sb2 | {321987654321987654321987654321987654321987654} | System.String |
| MaxLoop | 45 | int |
| Report | 1 | int |
| Résultat | "2" | string |

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 ».

| Nom | Valeur | Type |
|---|--------|------|
| <input checked="" type="checkbox"/> i < MaxLoop | true | bool |
| <input checked="" type="checkbox"/> i >= n1.Length | false | bool |
| <input checked="" type="checkbox"/> i >= n2.Length | false | bool |
| <input checked="" type="checkbox"/> (Fn + Sn + Report) / 10 | 1 | int |
| <input checked="" type="checkbox"/> (Fn + Sn + Report) % 10 | 1 | int |

AVANT DE COMMENCER

C# - LES FONDEMENTS

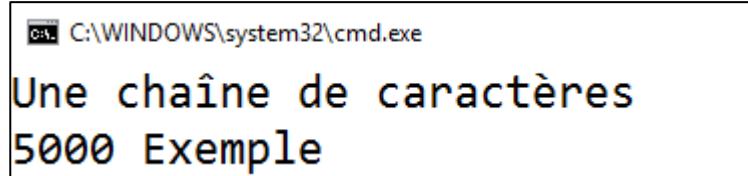
- Écrire dans la console
- Lire depuis la console
- Nettoyer la console

AVANT DE COMMENCER

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.WriteLine(...)` » et « `Console.Write(...)` ».

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

ECRIRE DANS LA CONSOLE

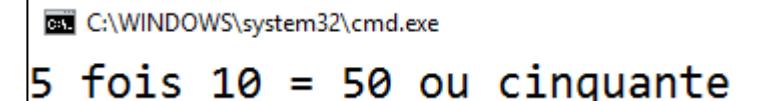
Parfois, nous aurons plusieurs informations à écrire sur une ligne.

Nous pourrons formater la ligne comme en 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:\ C:\WINDOWS\system32\cmd.exe
5 fois 10 = 50 ou cinquante

LIRE DEPUIS LA CONSOLE

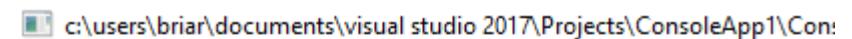
```
using System;

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 devrons déclarer des réceptacles capable de contenir ces informations. Ceux-ci sont appelés « Variable ».

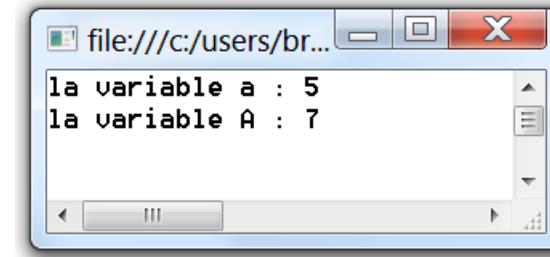
Le C# est un langage fortement typé, ce qui veut dire que nous devrons donner un type (entier, booléen, chaîne de caractères) à nos variables lors de leur déclaration.

De plus, il est à noté 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 égale (« = »).

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 raisons de mise en page

```
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 définie 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 automatique 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
```

PORТАBILITÉ 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 initialisée à la déclaration, et ne peuvent plus être modifiée.
- 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;
            Impossible de convertir null en 'int', car il s'agit d'un type valeur qui n'autorise pas les valeurs null
            Console.ReadLine();
        }
    }
}
```

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 délégues 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 |

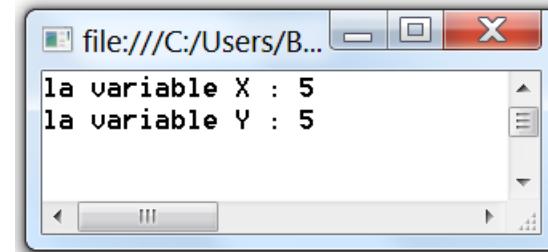
LES TYPES VALEUR

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



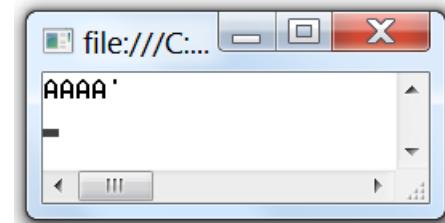
LE CARACTÈRE

| 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 prédéfinis de type référence, ceux-ci font partie de la catégorie « Classe » et sont stockés 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();  
        }  
    }  
}
```



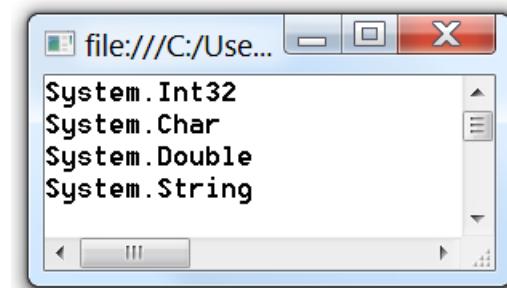
Le mot-clé « var » n'est utilisé qu'à titre d'exemple pédagogique.
Il est déconseillé de l'utiliser dans un premier temps.

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

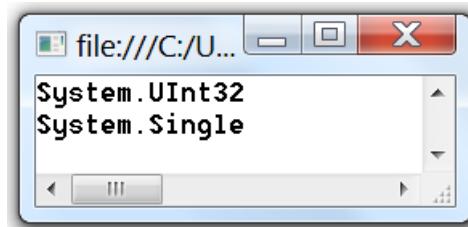
En utilisant les suffixes. En effet, certains types valeurs prédéfinis ont un suffixe spécifique pour la déclaration sous forme littérale.

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

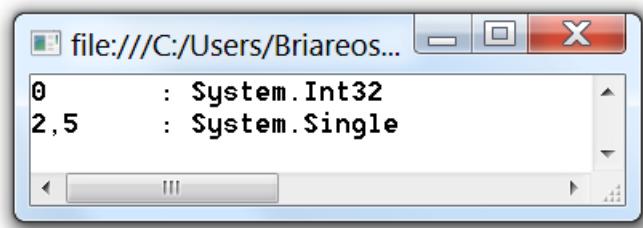
Exercice : Quelles seront les valeurs et les types affichés

```
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();
        }
    }
}
```

LES VALEURS LITTÉRALES NUMÉRIQUES

```
using System;  
  
namespace CoursCSharpFondements  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int x = 2_147_483_647;  
            Console.WriteLine(x); //affiche 2147483647  
            Console.ReadLine();  
        }  
    }  
}
```

Lorsque nous avons de grand nombre à écrire, il devient parfois difficile à les lire ou à les écrire comme par exemple « 1000000000 ».

Pour cela, il existe une annotation spécifique pour ces grands nombres qui consiste à utiliser le caractère ‘_’ en terme de séparateur de milliers.

LA VALEUR LITTÉRALE DE TYPE « STRING »

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

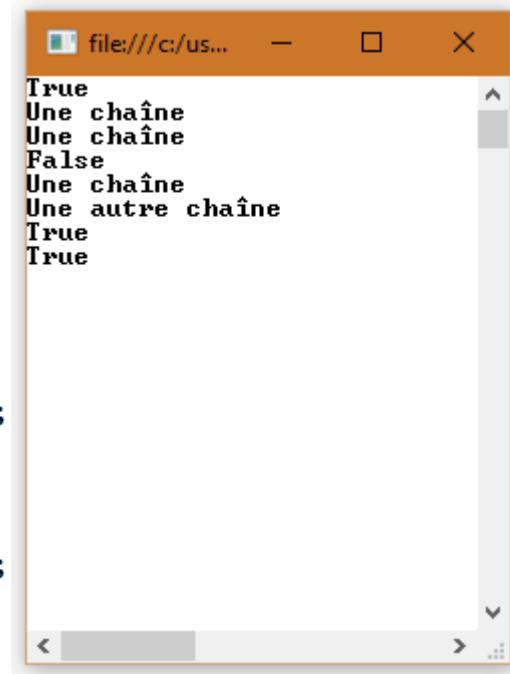
Pour rappel, une valeur littérale est une constante. Ce qui explique pourquoi le type « string » est le seul type référence à pouvoir être utilisé avec le mot clé « const ».

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;

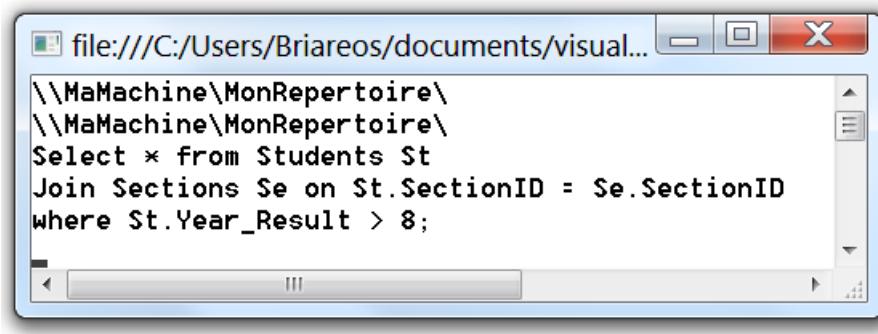
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string NetworkLink1 = "\\\\"MaMachine\\MonReperoire\\";
            string NetworkLink2 = @"\\"MaMachine\MonReperoire\";
            string Query =
                @"Select * from Students 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).



LA CONCATÉNATION DE « STRING »

Nous avons quatre possibilités pour concaténer des chaînes :

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

```
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.ToString("T"));

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

LA CONCATÉNATION DE « STRING »

Afin de simplifier l'utilisation et la lecture de la méthode « string.Format », il est possible également d'utiliser l'« interpolation de chaîne ».

Ce concept implique que l'on utilise le « \$ » avant la chaîne et que l'on remplace les chiffres par les variables ou les valeurs.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            string Prenom = "Michael";
            string Message = $"Bonjour {Prenom}, il est {DateTime.Now.ToString("T")}";

            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 »

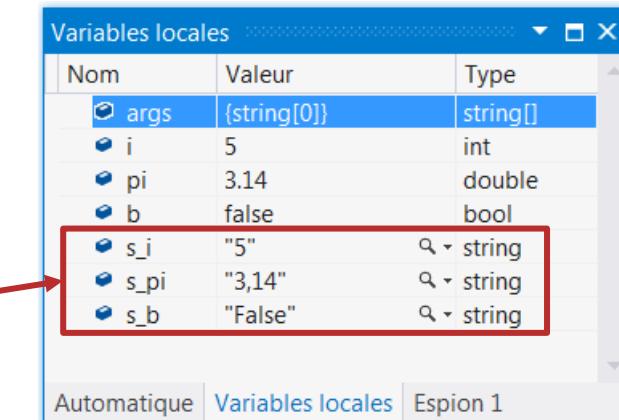
Tout élément en C# possède une méthode « `ToString()` », dans le cadre des types valeur, nous pourrons utiliser cette méthode pour transformer notre type valeur en type string.

```
using System;
```

```
namespace CoursSharpFondements
{
    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();
        }
    }
}
```



The screenshot shows the 'Variables locales' (Local Variables) window in Visual Studio. It displays the following variables:

| 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 |

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();
        }
    }
}
```

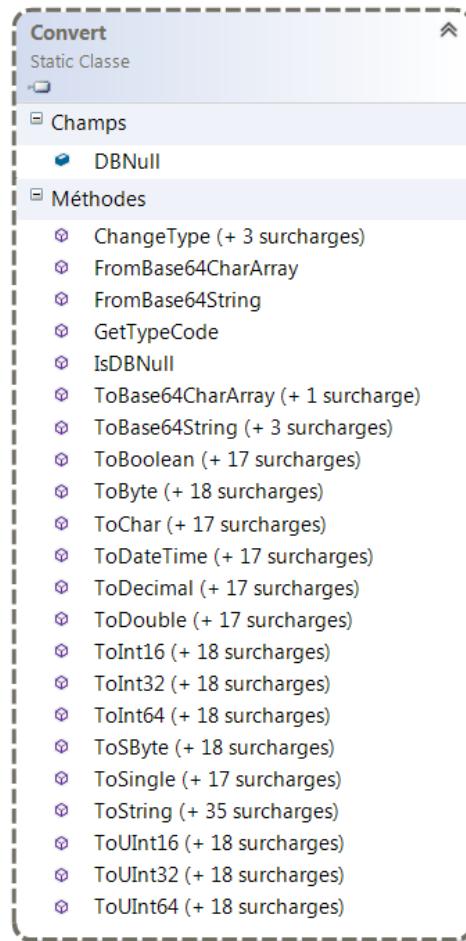
DE « STRING » À UN TYPE VALEUR - LA CLASSE « CONVERT »

```
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();
        }
    }
}
```



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 autorisées 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;
            Masqué pour des raison de mise en page
            Console.ReadLine();
        }
    }
}
```

CONVERSION EXPLICITE

Dans les autres cas nous devrons 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 effectuer par rapport au traitement complet.

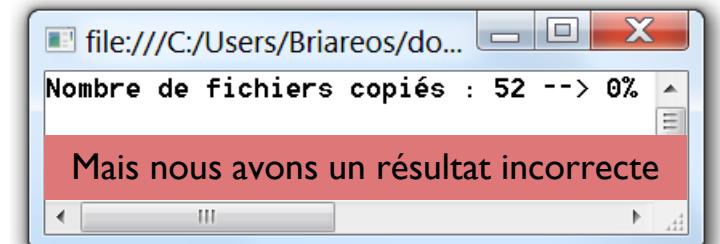
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int NbrFichiersTotal = 208;
            int NbrFichiersCopies = 52;

            int Pourcent = 100 / NbrFichiersTotal * NbrFichiersCopies;

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

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

Le calcul consiste donc à faire :
100 / Nombre de fichier total * 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 ($0,5 > 0 = \text{perte de précision}$)

CONVERSION EXPLICITE

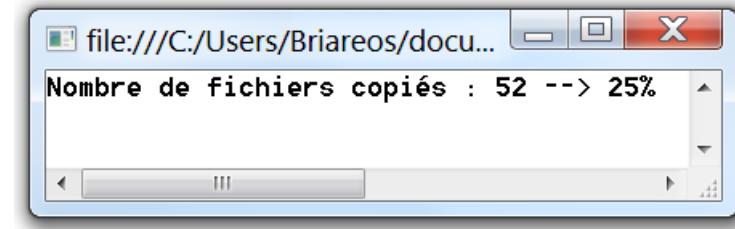
Nous devons spécifier au compilateur que nous voulons transtyper notre « float » en « int » et que par conséquent nous prenons la responsabilité en cas de perte de données (précision).

```
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 FONDEMENTS

- 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 | | |
|--------------------------------|--------------------|------------------------|
| <code>==</code> | Égale | <code>i == 5</code> |
| <code>!=</code> | Différent | <code>i != 5</code> |
| <code><</code> | Plus petit que | <code>i < 5</code> |
| <code><=</code> | Plus petit ou égal | <code>i <= 5</code> |
| <code>></code> | Plus grand que | <code>i > 5</code> |
| <code>>=</code> | Plus grand ou égal | <code>i >= 5</code> |

| Les opérateurs logiques | | |
|-------------------------|-------------|--|
| <code>!</code> | Négation | |
| <code>&&</code> | Et | |
| <code> </code> | Ou | |
| <code>^</code> | 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();
        }
    }
}
```

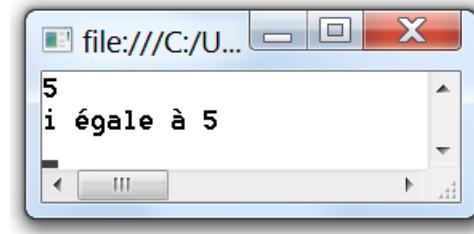
LE BLOC « IF ... ELSE IF ... ELSE »

```
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();
        }
    }
}
```

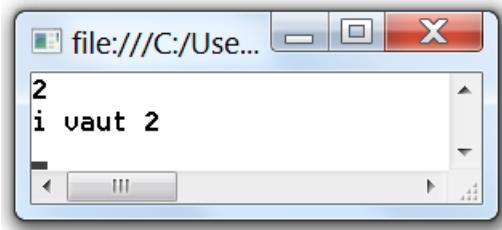
Il arrive, cependant, que nous ayons plusieurs conditions et que nous ayons un traitement spécifique pour chacune d'entre elles. Dans ce cas nous devrons utiliser une variante du bloc « if ... else ».



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();
        }
    }
}
```

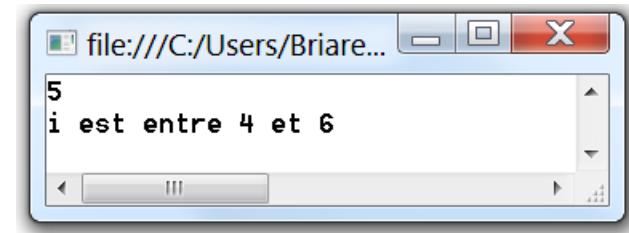
LE BLOC « SWITCH »

```
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 1 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 coalesce
- L'opérateur coalesce et affectation
- L'opérateur « typeof »
- L'opérateur « is »
- Pattern matching
- L'opérateur « as »
- 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
- Incrémentation et décrémentation
- 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.

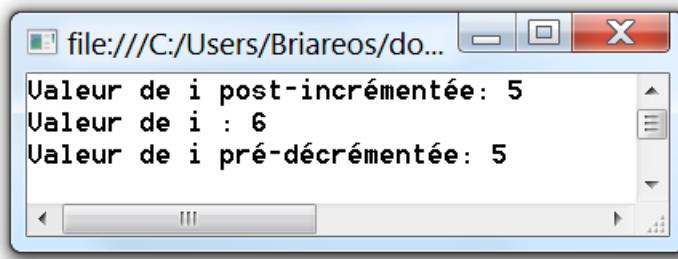
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            int j = i++;
            j += i;

            int k = 7;
            //multiplie k par 2
            k <= 1;
            //divide 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 incréméntation ou décréméntation est définie sur le moment où la variable va changer de valeur.

Dans le cadre de la pré-incréméntation (`++x`) ou de la pré-décréméntation (`--x`), `x` changera de valeur avant d'avoir été utilisée.

Dans le cadre de la post-incréméntation (`x++`) ou de la post-décréméntation (`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 :

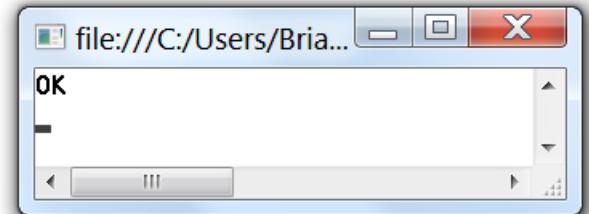
- La condition
- La valeur si vrai
- La valeur si faux

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;

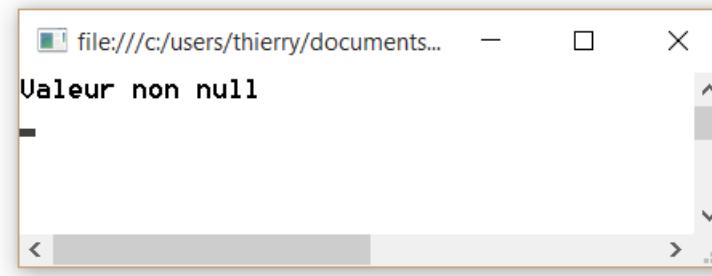
    string result = s ?? t ?? u ?? "Valeur non null";

    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 COALESCE ET AFFECTATION

L'opérateur coalesce possède une version simplifiée « ??= » qui permet l'affectation d'une valeur si la variable est « null ».

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

    //Avant l'opérateur coalesce
    if (s == null)
        s = "Nouvelle valeur";

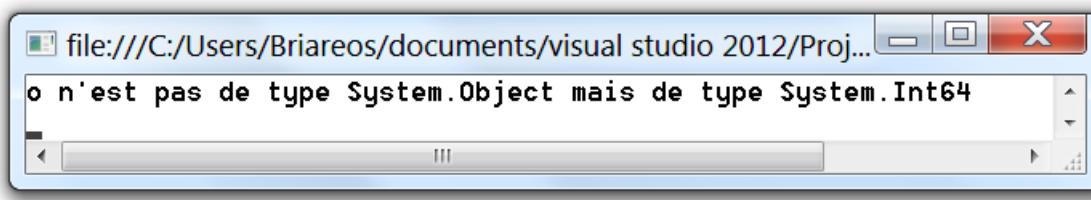
    //Utilisation de l'opérateur coalesce
    s = s ?? "Nouvelle valeur";

    //Utilisation de l'opérateur coalesce avec affectation
    s ??= "Nouvelle valeur";

    Console.WriteLine(s); //affiche nouvelle valeur
}
```

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 « IS »

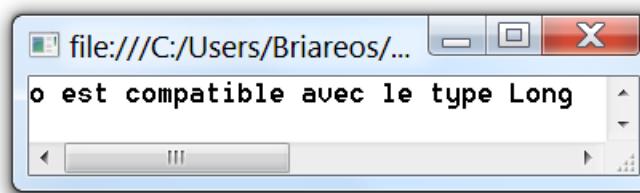
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            object o = 5L; //Boxing

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

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

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

is :
Expression is Type;



PATTERN MATCHING : TYPE PATTERN

L'utilisation répétée de l'opérateur « is » peut-être simplifié par l'utilisation du « pattern matching ». En effet, ce dernier teste si une expression peut être convertie en un type spécifié et, si elle peut l'être, l'affecte à une variable de ce type.

```
static void Main(string[] args)
{
    object o = 5L;

    if (o is int)
    {
        int i = (int)o;
        Console.WriteLine($"o est un {i.GetType()} : {i}");
    }
    else if (o is long)
    {
        long l = (long)o;
        Console.WriteLine($"o est un {l.GetType()} : {l}");
    }
}
```

```
static void Main(string[] args)
{
    object o = 5L;

    if (o is int i)
    {
        Console.WriteLine($"o est un {i.GetType()} : {i}");
    }
    else if (o is long l)
    {
        Console.WriteLine($"o est un {l.GetType()} : {l}");
    }
}
```

PATTERN MATCHING : TYPE PATTERN

L'utilisation du « pattern matching » peut aussi être utilisé avec une instruction conditionnel « switch ».

```
static void Main(string[] args)
{
    object o = 5L;

    switch(o)
    {
        case int i:
            Console.WriteLine($"o est un {i.GetType()} : {i}");
            break;
        case long l:
            Console.WriteLine($"o est un {l.GetType()} : {l}");
            break;
    }
}
```

PATTERN MATCHING : TYPE PATTERN

```
static void Main(string[] args)
{
    object o = 5L;

    switch(o)
    {
        case int i:
            Console.WriteLine($"o est un {i.GetType()} : {i}");
            break;
        case long l when l <= 2_147_483_647 && l >= -2_147_483_648:
            Console.WriteLine($"Cette valeur aurait pu être stockée dans un int");
            Console.WriteLine($"o est un {l.GetType()} : {l}");
            break;
        case long l:
            Console.WriteLine($"o est un {l.GetType()} : {l}");
            break;
    }
}
```

Et cette instruction « switch » peut être conditionnée au niveau des « case » à l'aide du mot clé « when » suivi de la condition.

PATTERN MATCHING : CONSTANT PATTERN

L'opérateur « is » est également utilisé pour tester la valeur d'une variable par rapport à une constante.

En C# « null » est une valeur littérale qui spécifie l'absence de valeur et, comme toute valeur littérale, elle est une « constante ».

```
static void Main(string[] args)
{
    double pi = 3.1415926535897931;

    if(pi is Math.PI)
        Console.WriteLine("C'est PI");

    string s = null;

    if(s is null)
        Console.WriteLine("La variable n'a pas de valeur");
}
```

PATTERN MATCHING : CONSTANT PATTERN

```
static void Main(string[] args)
{
    object o = null;
    string s = "Hello";

    if (o is null)
        Console.WriteLine("La variable 'o' n'a pas de valeur");

    if(o == null)
        Console.WriteLine("La variable 'o' n'a pas de valeur");

    if (!(s is null))
        Console.WriteLine("La variable 's' a une valeur");

    if (s != null)
        Console.WriteLine("La variable 's' a une valeur");
}
```

Alors, dans ce cas, est-il préférable d'utiliser l'opérateur d'égalité « == » ou l'opérateur « is »?

À choisir, il est préférable d'utiliser l'opérateur « is ».

En effet, l'opérateur d'égalité peut-être surchargé, comme nous le verrons dans la partie orienté objet de ce cours, et une mauvaise implémentation pourrait nous renvoyer une réponse erronée.

L'opérateur « is » quand à lui ne peut pas être surchargé.

L'OPÉRATEUR « AS »

```
static void Main(string[] args)
{
    object o = "Hello";

    string s = o as string;
    // est équivalent à
    s = o is string ? (string)o : null;

    o = 5;

    int i = o as int;
    int? j = o as int?;
```

L'opérateur « as » converti explicitement le résultat pour donner une valeur d'un type donné ou « null » s'il ne parvient pas à convertir.

De ce fait, cet opérateur ne peut être utilisé qu'avec des types « nullable ».

Cependant, contrairement à l'opérateur de conversion classique, « (int)x » par exemple, l'opérateur as ne déclenche jamais d'erreur et la variable « o » ne sera évaluée qu'une seule fois.

as :

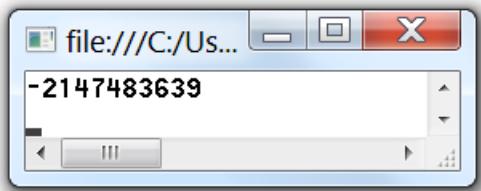
Type variable = Expression as Type;

readonly struct System.Int32
Represents a 32-bit signed integer.

L'opérateur as doit être utilisé avec un type référence ou un type nullable ('int' est un type valeur non-nullble)

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 »

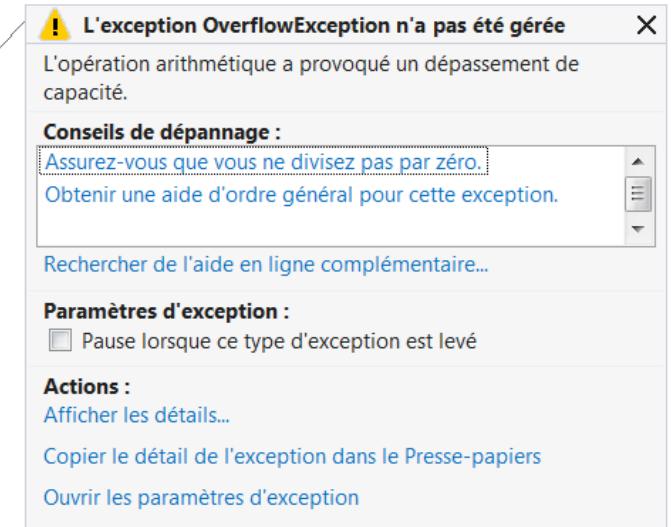
Le mot clé « checked » sert à activer explicitement le contrôle de dépassement pour les opérations arithmétiques de type entier et les conversions.

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

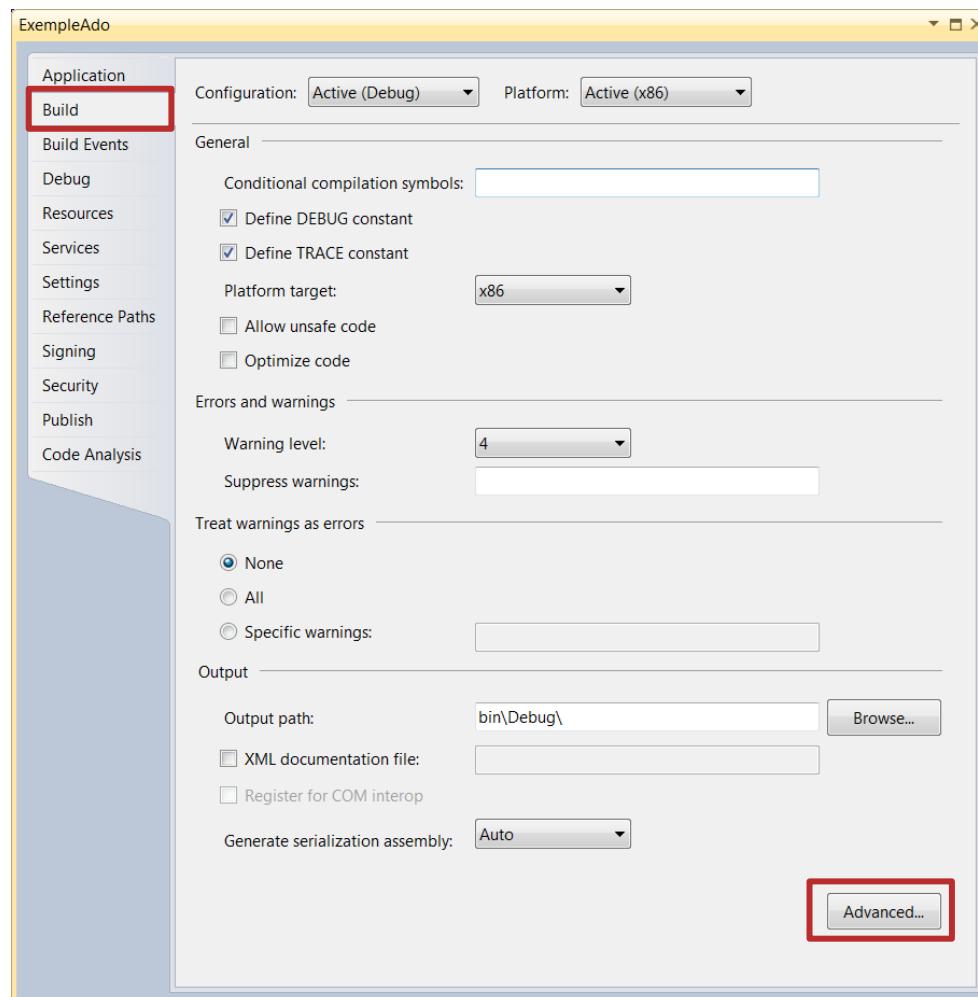
```
using System;
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = int.MaxValue;
            int y = checked(x + 10);

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

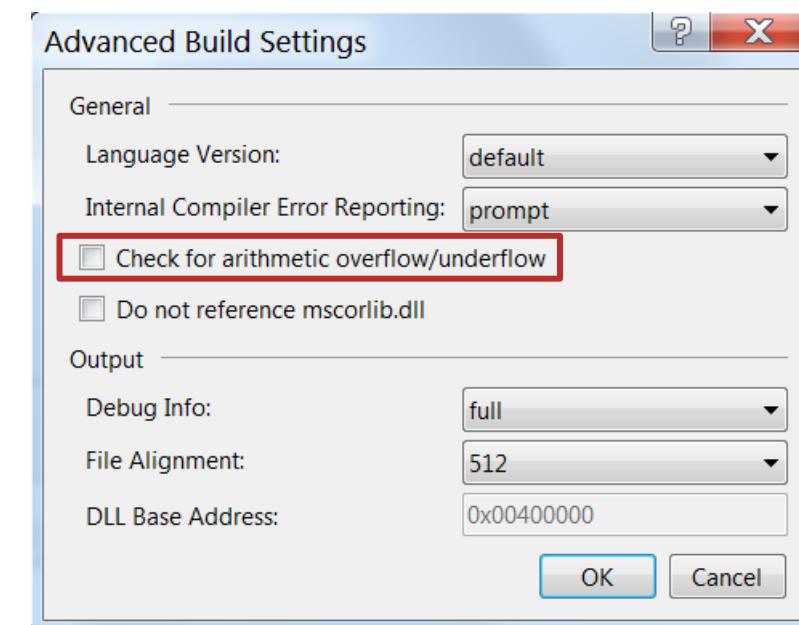
            Console.WriteLine(y);
            Console.ReadLine();
        }
    }
}
```



L'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 |
|---------------------------------|---|--------------------|--|
| Primaire | , , ?, [], f(x), [], x++, x--, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y | ET binaire | & |
| Unaire | +x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false | XOR binaire | ^ |
| Range | 0..25 | OU binaire | |
| Multiplication / Division | *, /, % | ET logique | && |
| Addition / Soustraction | +, - | OU logique | |
| Opérateurs de décalage binaires | <<, >> | Opérateur ternaire | ?: |
| Relationnel | <, <=, >, >=, is, as | Affectation | =, +=, -=, *=, /=, %=, &=, =, ^=, <<=, >>=, ??=, => |
| Comparaison | ==, != | | |



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-xxxxxx-xx) en IBAN (BExx-xxxx-xxxx-xxxx). Trouvez la démarche via un moteur de recherche.

LES BOUCLES

C# - LES FONDEMENTS

- 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 ».

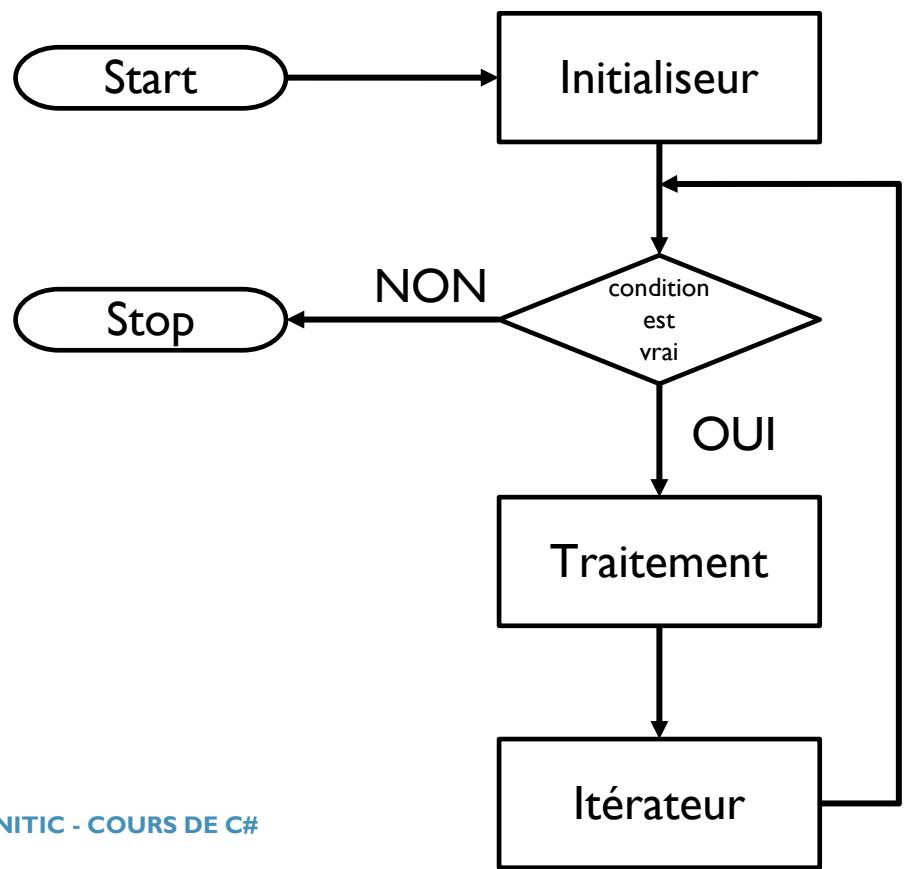
```
//Boucle "for"
for (int i = 0; i < 10; i++)
{
}

//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.

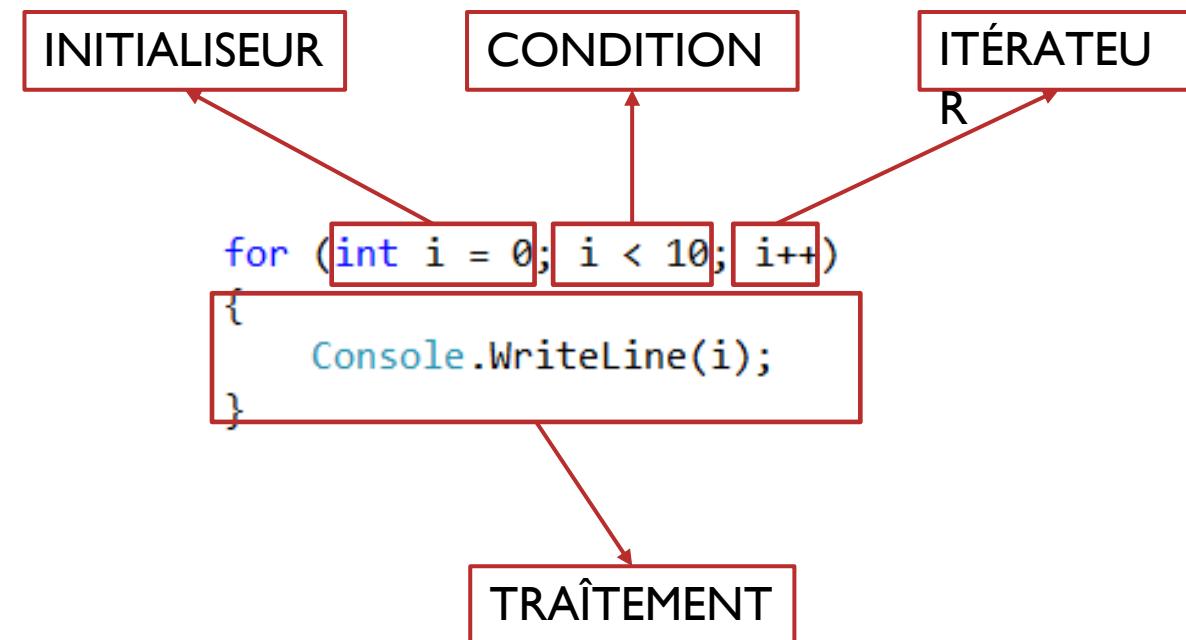
Cette boucle est composée de trois éléments :

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

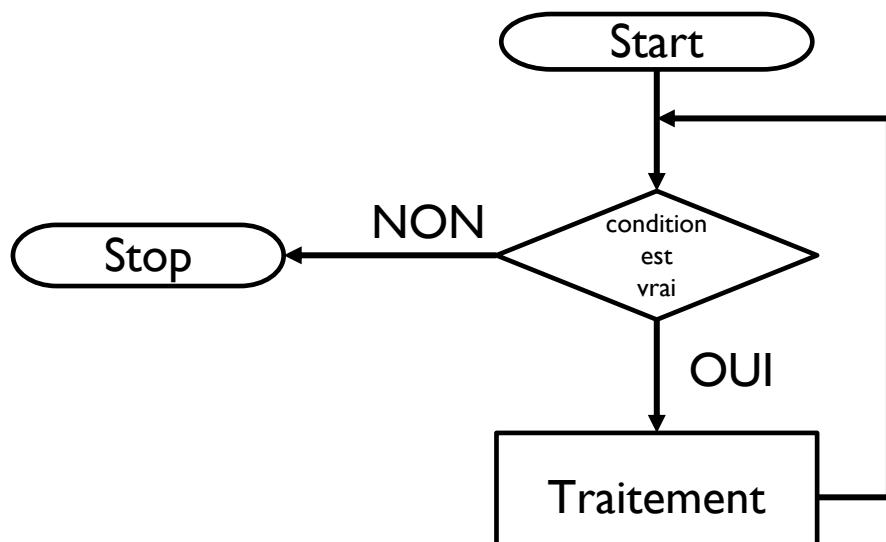
La condition est l'expression vérifiée avant chaque nouvelle itération de la boucle. (Elle doit être égale à « true » pour qu'une nouvelle itération puisse être effectuée)

L'itérateur est une expression qui est évaluée après chaque itération. Ces itérations cessent quant 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.

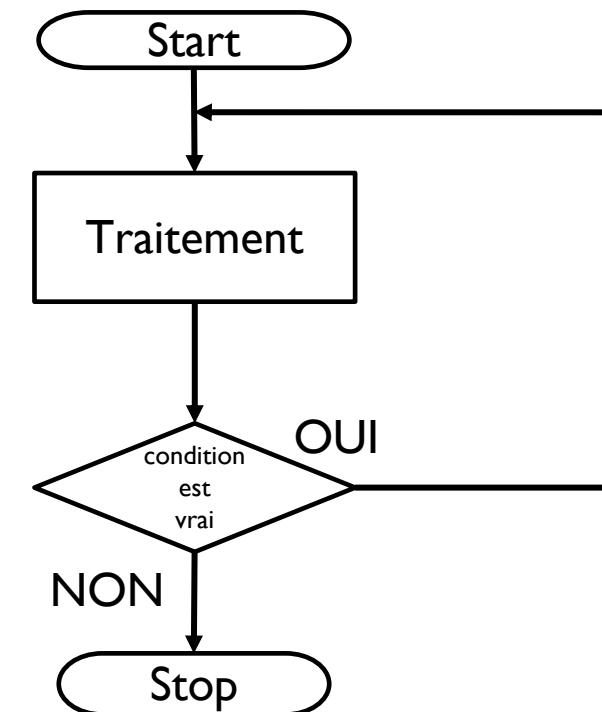
```
int x = 0;  
  
while (x < 10)  
{  
    Console.WriteLine(x);  
    x++;  
}
```

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 ».
 $1 \times 1 = 1$; $2 \times 1 = 2$;
 $2 \times 1 = 2$; $2 \times 2 = 4$;
5. À l'aide d'une boucle « for » comptez de 0, à 20,0 en augmentant de 0,1, en utilisant des doubles, et afficher la valeur à chaque itération.
Remarquez-vous quelque chose de particulier ?
6. Bonus : Calculer la racine carré d'un nombre avec maximum 10 décimales (Math.Sqrt(x) ne peut être utilisée que pour vérifier la réponse),

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 côté du type de la variable que nous déclarerons. Une fois instancié, chaque élément se verra attribué, par défaut, la valeur de « default(T) ».

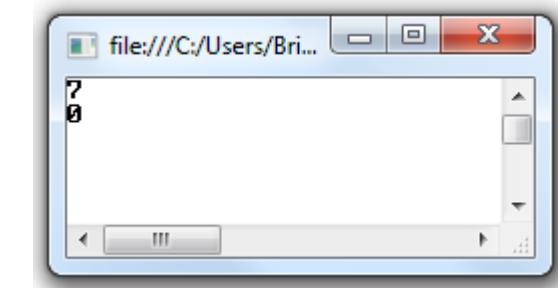
Le premier élément du tableau à pour index 0 (zéro).

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.

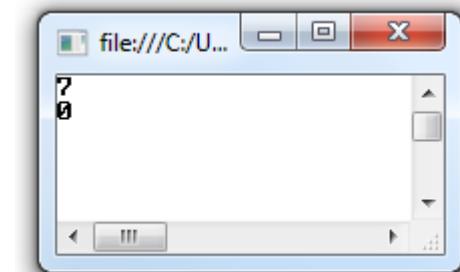
Les tableaux matriciels : où chaque dimension contient le même nombre d'éléments.

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

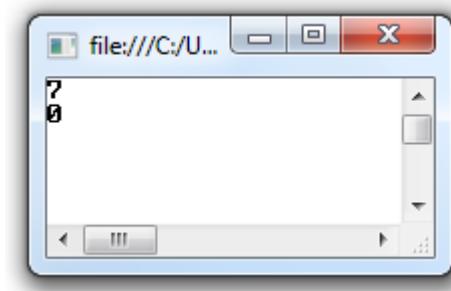
Les tableaux orthogonaux : où chaque dimension peut contenir un nombre différent d'éléments.

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 définit 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.

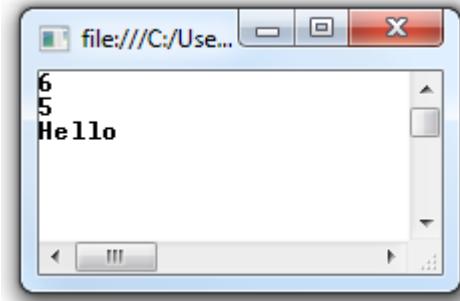
De plus, ces collections peuvent travailler avec n'importe quel type de données puisque qu'elles travaillent avec des objets en paramètres, rendant les conversions implicite et explicite obligatoire.

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);

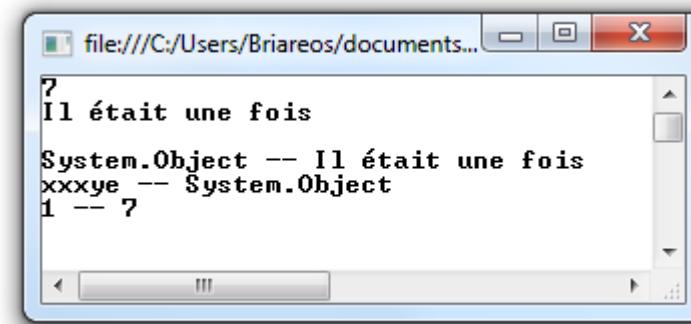
            Console.WriteLine();

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

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

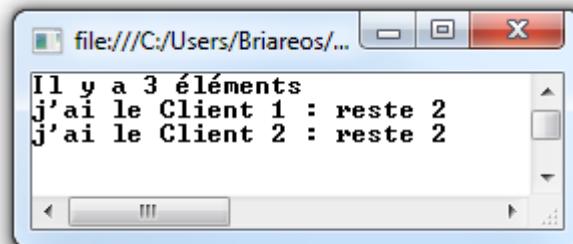
Une « Hashtable » est comparable à un dictionnaire ou l'on accède à une valeur en utilisant une clé. La clé et la valeur sont de type « object ». De plus, la clé doit être unique.

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();

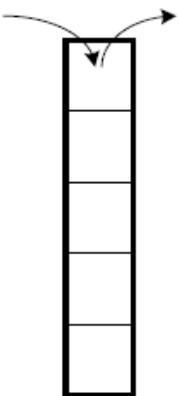
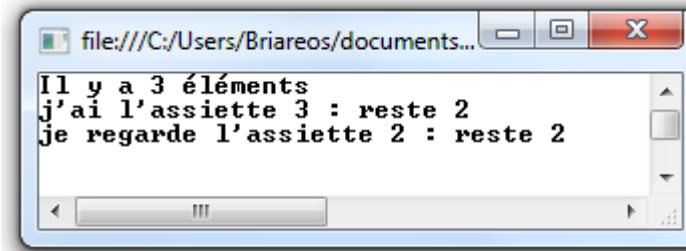
            //La méthode Push ajoute un élément à la pile
            queue.Push("assiette 1");
            queue.Push("assiette 2");
            queue.Push("assiette 3");

            //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)



LIFO

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] = "Mvvm de la découverte à la matrise";

            string livre = livres[1];

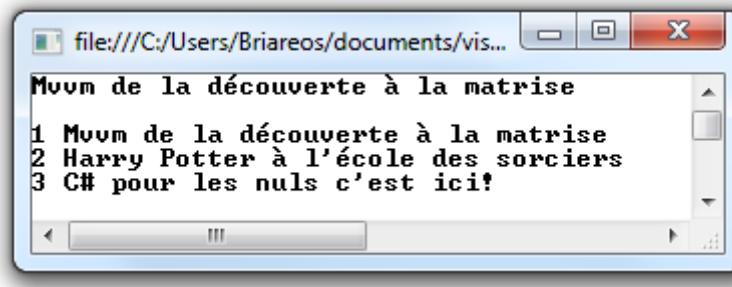
            Console.WriteLine(livre);
            Console.WriteLine();

            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 fonctionne exactement de la même manière que les non générique, 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("Assiete 1");
            Assietes.Push("Assiete 2");
            Assietes.Push("Assiete 3");

            Console.WriteLine(Assietes.Count);
            string Assiete = Assietes.Pop();
            Console.WriteLine("{0} : reste {1}", Assiete, Assietes.Count);
            Assiete = Assietes.Peek();
            Console.WriteLine("{0} : reste {1}", Assiete, 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 faites l'addition de ces deux nombres en ne convertissant que caractère par caractère. (Méthode « ToCharArray() » de la classe « string »).

LES STRUCTURES

C# - LES FONDEMENTS

-
- Déclaration & utilisation
 - Limitations

LES STRUCTURES

DÉCLARATION & UTILISATION

Une structure permet de déclarer un type complexe de type valeur.

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

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

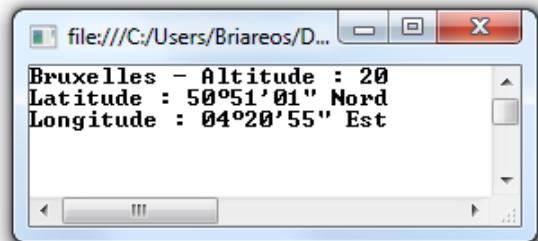
namespace CoursCSharpFondements
{
    public struct Localisation
    {
        public MesureAngulaire Longitude, Latitude;
        public int Altitude;
    }
}
```

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Localisation Bruxelles;
            Bruxelles.Latitude.Degre = 50;
            Bruxelles.Latitude.Minutes = 51;
            Bruxelles.Latitude.Seconde = 1;
            Bruxelles.Latitude.Direction = "Nord";
            Bruxelles.Longitude.Degre = 4;
            Bruxelles.Longitude.Minutes = 20;
            Bruxelles.Longitude.Seconde = 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.Seconde, Bruxelles.Latitude.Direction);
            Console.WriteLine("Longitude : {0:D2}°{1:D2}'{2:D2}\" {3}", Bruxelles.Longitude.Degre,
                Bruxelles.Longitude.Minutes, Bruxelles.Longitude.Seconde, Bruxelles.Longitude.Direction);

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



The screenshot shows a Windows Command Prompt window with the title bar 'file:///C:/Users/Briareos/D...'. The window contains the following text:
Bruxelles - Altitude : 20
Latitude : 50°51'01" Nord
Longitude : 04°20'55" Est

LIMITATIONS

- 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 de constructeur sans paramètre (ceux-ci sont réservés pour l'affectation par copie)
- Une structure ne peut pas déclarer de destructeur.
- Elles fonctionnent comme les types valeurs. 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 l'instance d'origine.
- Un structure ne peut pas utiliser les concepts d'héritage.
- Elles ne peuvent pas recevoir la valeur « null »

LIMITATIONS

Ceci dit, elles peuvent comme les classes :

- Déclarer des constructeurs qui ont des paramètres.
- Implémenter des interfaces.
- 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.



EXERCICES

EXERCICES (MAXIMUM 30')

1. Ecrire une structure pour définir un point possédant deux entier X, Y et créer un tableau deux dimensions de 5 sur 5 de type « Point » (nullable) et remplir une des diagonales, ensuite à l'aide de deux boucles afficher les valeurs dans la console comme suit :
« X : 1 - Y : 1 »
« X : 2 - Y : 2 »
« X : 3 - Y : 3 »
...
2. Ecrire deux structures Celsius et Fahrenheit toutes deux ayant une variable de type double appelée « Temperature ».



LES MÉTHODES

C# - LES FONDEMENTS

- Déclaration
- Mot-clé « return »
- Invocation
- L'opérateur null conditionnel « ?. »
- Les paramètres
- Les paramètres facultatifs
- Paramètres nommés
- Mot-clé « in »
- Mot-clé « params »
- Mot-clé « ref »
- Retour de fonction par référence
- 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 langages 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 UneProcedure()
        {
            //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 pris par notre application.

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



INVOCATION

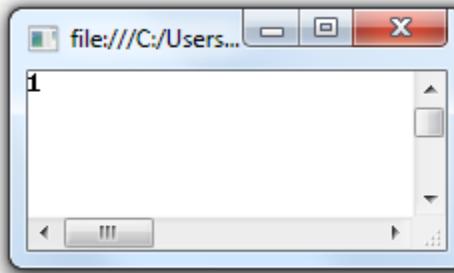
```
using System;

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) et ce en utilisant l'opérateur d'accès aux membres « . ».



L'OPÉRATEUR NULL CONDITIONNEL « ?. »

```
static void Main(string[] args)
{
    string s = null;
    int? length = s?.Length;
    string result = s?.Substring(2);

    Console.WriteLine((length.HasValue) ? length.ToString() : "No value");
}
```

Il existe un variant à l'opérateur d'accès aux membres « . ».

Cette variante est l'opérateur null conditionnel « ?. », ce dernier vérifie que la variable est non null avant d'accéder au membre souhaité.

Si la variable est null, il ne fera rien.

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 devrons spécifier le type et leur donner un nom que nous pourrons utiliser dans la méthode.

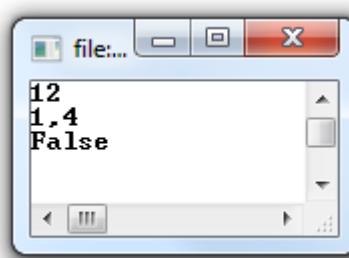
Ensuite, lors de l'appel de nous passerons les valeurs de ses paramètres afin que la méthode effectue son traitement.

```
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;
        }

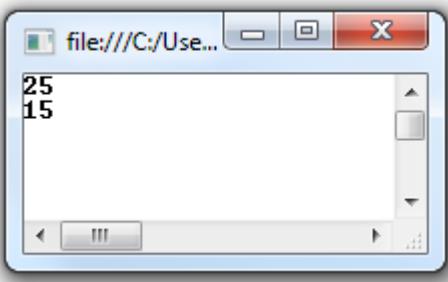
        public bool EstPaire(int i)
        {
            return i % 2 == 0;
        }
    }
}
```

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 valeur 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 « IN »

Utiliser le mot-clé « in » dans les paramètres d'une méthode signifie que ce paramètre est en lecture seul.

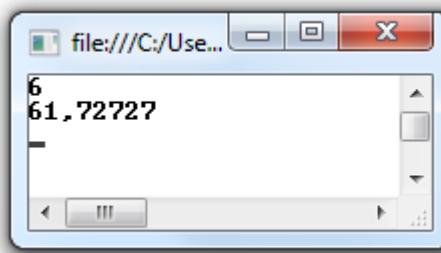
```
O réferences
static void Test(in int x)
{
    x++;
}
[?] (paramètre) in int x
Impossible d'effectuer l'assignation à variable 'in int', car il s'agit d'une variable en lecture seule
```

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 »

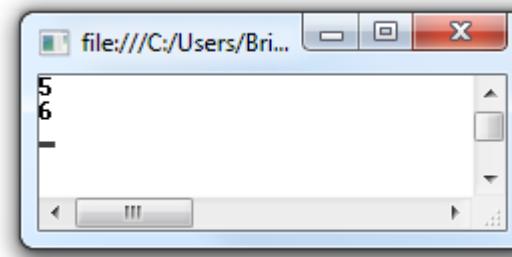
Avec les types valeurs, le passage en paramètre se fait par copie.

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();
        }
    }
}
```

RETOUR DE FONCTION PAR RÉFÉRENCE

```
struct AStruct
{
    string[] firstNames;

    3 références
    public string GetByIndex(int index)
    {
        firstNames ??= new string[]{ "Pierre", "Pauline", "Jacques" };
        return firstNames[index];
    }

    1 référence
    public ref string GetRefByIndex(int index)
    {
        firstNames ??= new string[] { "Pierre", "Pauline", "Jacques" };
        return ref firstNames[index];
    }
}
```

Le mot clé « `ref` » peut être utilisé pour les type de retour, renvoyant la référence d'une variable plutôt que la valeur de cette variable.

RETOUR DE FONCTION PAR RÉFÉRENCE

De ce fait, il sert également lors de la déclaration de variable locale.

```
static void Main(string[] args)
{
    AStruct aStruct = new AStruct();

    string prenom = aStruct.GetByIndex(1);
    prenom = "Paul";
    Console.WriteLine(aStruct.GetByIndex(1));

    ref string refPrenom = ref aStruct.GetRefByIndex(1);
    refPrenom = "Paul";
    Console.WriteLine(aStruct.GetByIndex(1));
}
```

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;
        }
    }
}
```

MOT-CLÉ « OUT »

Implémentation classique

```
static void Main(string[] args)
{
    string val = "ABC";

    int i;
    if (int.TryParse(val, out i))
    {
        Console.WriteLine($"i a pour valeur {i}");
    }
    else
    {
        Console.WriteLine($"Parsing a échoué valeur de i : {i}");
    }
}
```

Implémentation possible depuis la version 7.0 du langage

```
static void Main(string[] args)
{
    string val = "ABC";

    if (int.TryParse(val, out int i))
    {
        Console.WriteLine($"i a pour valeur {i}");
    }
    else
    {
        Console.WriteLine($"Parsing a échoué valeur de i : {i}");
    }
}
```



Ce n'est que du sucre syntaxique

DIFFÉRENCE ENTRE SIGNATURE ET PROTOTYPE

Signature d'un 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')

1. Dans les structures Celsius et Fahrenheit, écrire la fonction de conversion de l'une vers l'autre.
2. 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 déclarent 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 ».

```
public enum TypeCarburant { Essence, Diesel, Gaz }
```

```
public enum TypeCarburant { Essence = 2, Diesel = 4, Gaz = 8 }
```

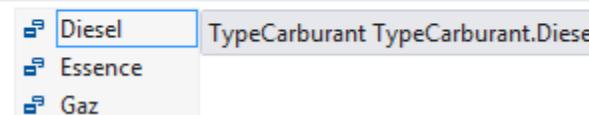
UTILISATION

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.
        }
    }
}
```



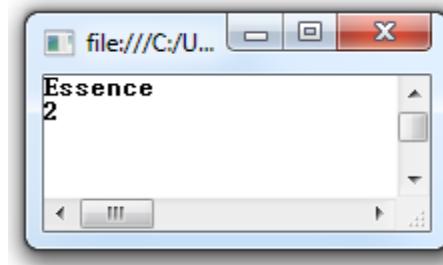
The screenshot shows a portion of a C# program within a code editor. An intellisense dropdown is open at the end of the line 'TypeCarburant Carburant = TypeCarburant.'. The dropdown contains three items: 'Diesel', 'Essence', and 'Gaz'. The item 'Diesel' is highlighted with a blue selection bar. To the right of the dropdown, the full name 'TypeCarburant' and its member 'TypeCarburant.Diesel' are displayed.

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();
        }
    }
}
```

UTILISATION DES « FLAGS »

```
[Flags]
enum Right { Execute, Write, Read }
enum Droit { Executer = 1, Ecrire = 2, Lire = 4 }

class Program
{
    static void Main(string[] args)
    {
        Right right = Right.Execute | Right.Read;

        if(right.HasFlag(Right.Execute))
            Console.WriteLine("j'ai le droit d'exécuter");

        if (right.HasFlag(Right.Write))
            Console.WriteLine("j'ai le droit d'écrire");

        if (right.HasFlag(Right.Read))
            Console.WriteLine("j'ai le droit de lire");
    }
}
```

Il y a deux manières de travailler avec des « flag ».

La première en utilisant l'attribut « Flags »,

Cet attribut indique qu'une énumération peut être traitée comme un champ de bits (0 ou 1).

La deuxième en utilisant des puissances de 2 ($2^0, 2^1, 2^2, \dots$).

Pour attribuer, plusieurs « flags » à une variable, nous les sépareront par un ou binaire « | ».

Pour tester si la valeur possède un « flag » particulier, nous utiliserons sur la variable la méthode « HasFlag(...) »

ENUM.GETNAMES(...)

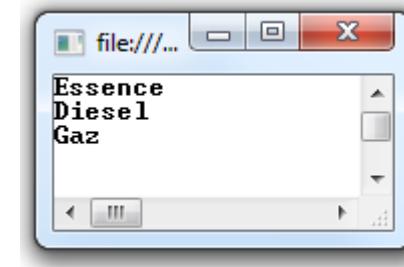
```
using System;

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 arriva 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 }

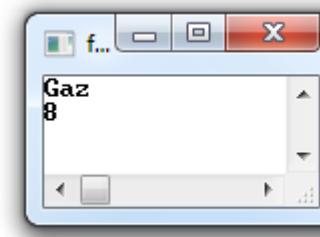
        static void Main(string[] args)
        {
            string s = "Gaz";

            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 ».





EXERCICES

EXERCICES (MAXIMUM 30')

- Créer une énumération pour les couleurs (Coeur, Carreau, Pique, Trefle)
- Créer une énumération pour les valeurs (as = 14, deux = 2, trois = 3, ..., Roi = 13)
- Créer une structure Carte qui contient deux variables publiques :
 - Couleur de type Couleurs
 - Valeur de type Valeurs
- Déclarer un tableau de Carte d'une taille de 52
- À l'aide d'une boucle « foreach » définir les couleurs et les valeurs de chacune des cartes
- Afficher les cartes (Définir si cela fonctionne : si oui pourquoi, sinon pourquoi)

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))