



TD - Découverte de Visual Studio

Ressources Célène : DI4.S8.PLATEFORMES .NET / « Lg.Net »

Machine virtuel disponible : VMWARE Windows 10 - .Net - Nicolas DAGNAS

Intervenant : Nicolas DAGNAS

Durée total du TD : 6h

Visual Studio

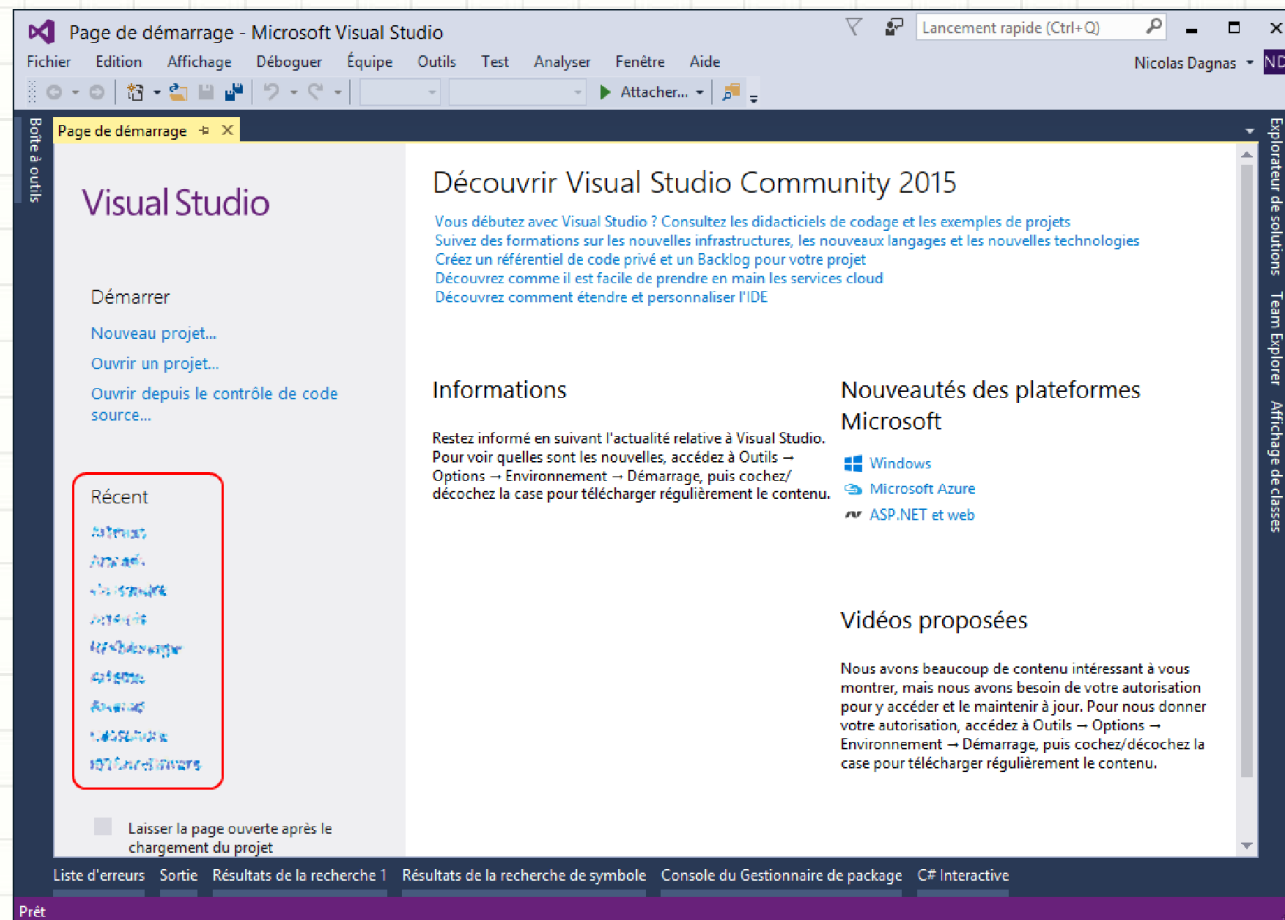
- Ce TD a pour but de vous faire découvrir Visual Studio par l'exemple. Vous allez coder un objet permettant la lecture et l'écriture d'un fichier texte dans un premier temps (avec présentation), puis la création d'une fenêtre permettant sa visualisation autre qu'un mode console (en mode autonome).
- Les *slides* suivants vous fourniront les grandes lignes de ce qu'il faut savoir pour utiliser cet E.D.I. qui reste très similaire aux autres existants (Eclipse, Netbeans, etc...), ainsi que les indications nécessaires pour coder le projet.
- Ce TD traitera donc principalement (en dehors de l'E.D.I. lui-même), du passage de paramètres (ligne de commande), de l'utilisation des flux et de la création d'interfaces simples.

Pour commencer

- Le TD peut être effectué avec Microsoft Visual Studio de 2015 à 2019, voir les précédents (Visual Studio Code n'est pas abordé ici).
- Pour ceux n'ayant pas de Visual Studio à disposition, une machine virtuelle est disponible avec Visual Studio 2019 :
 - VMWARE Windows 10 - .Net - Nicolas DAGNAS
- Pour ceux qui l'installeront, faites une installation basique avec uniquement C# Desktop et sans la mobilité.
- Ce TD ayant pour but la découverte de l'environnement, n'hésitez pas à poser des questions si quelque chose n'est pas clair.

VS – Page de démarrage

- En démarrant « Microsoft Visual Studio », on arrive par défaut sur la page de démarrage. La zone « Récents » permet d'accéder aux projets ouverts récemment d'un simple clic.

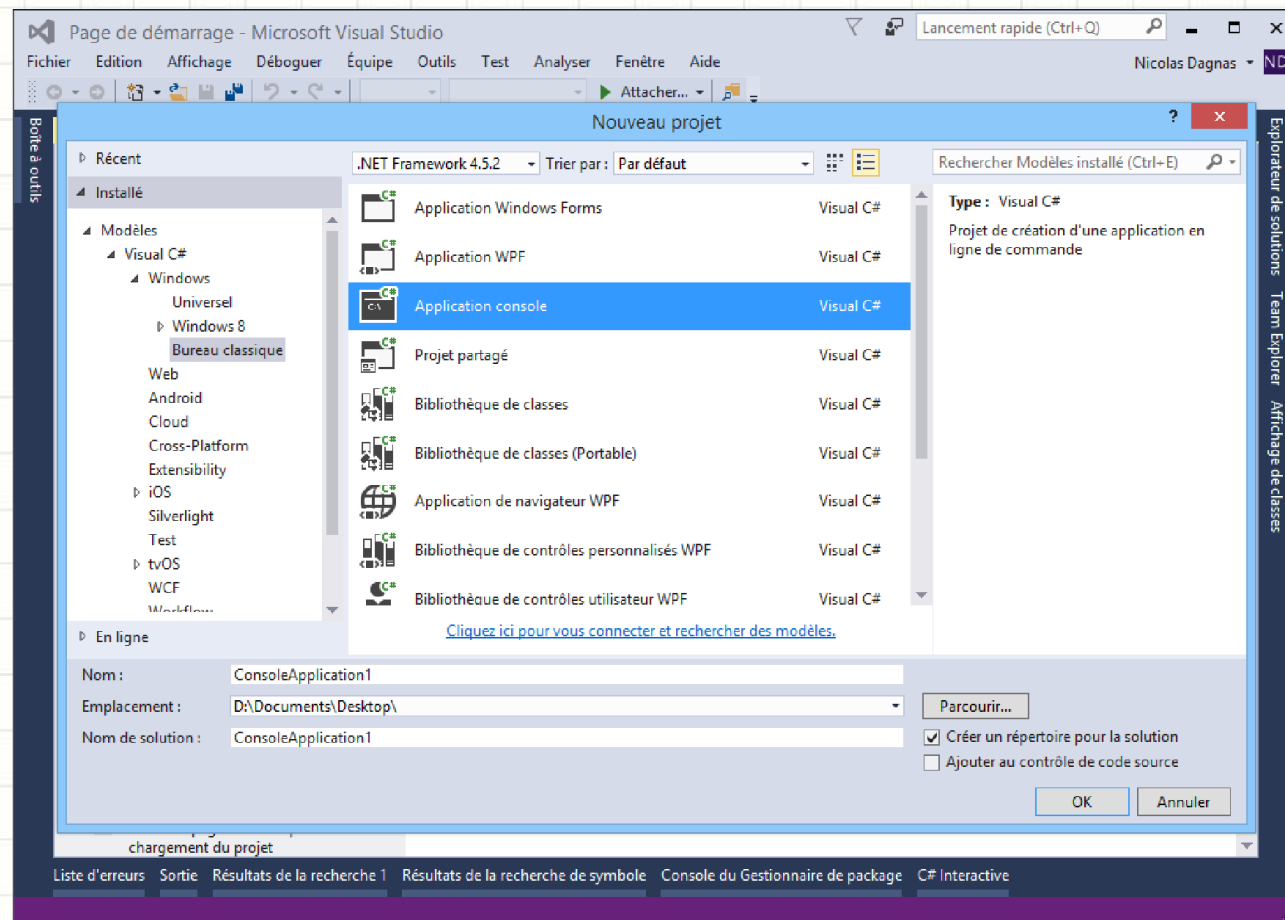


VS – Types de Projets

- Pour le TD, nous nous concentrerons sur ces deux types de projets :
 - Windows > Application console (.NET Framework)
 - Fenêtre DOS, interactions minimalistes avec l'utilisateur
 - Windows > Application Windows Form (.NET Framework)
 - Fenêtre Windows, conceptions d'écrans complexes type IHM
- Les projets de ce types sont suffisamment basique pour vous permettre de vous faire une bonne idée générale. Une fois maîtrisés, les autres types de projets deviennent accessibles, y compris les projets de type WinRT, voir Xamarin.

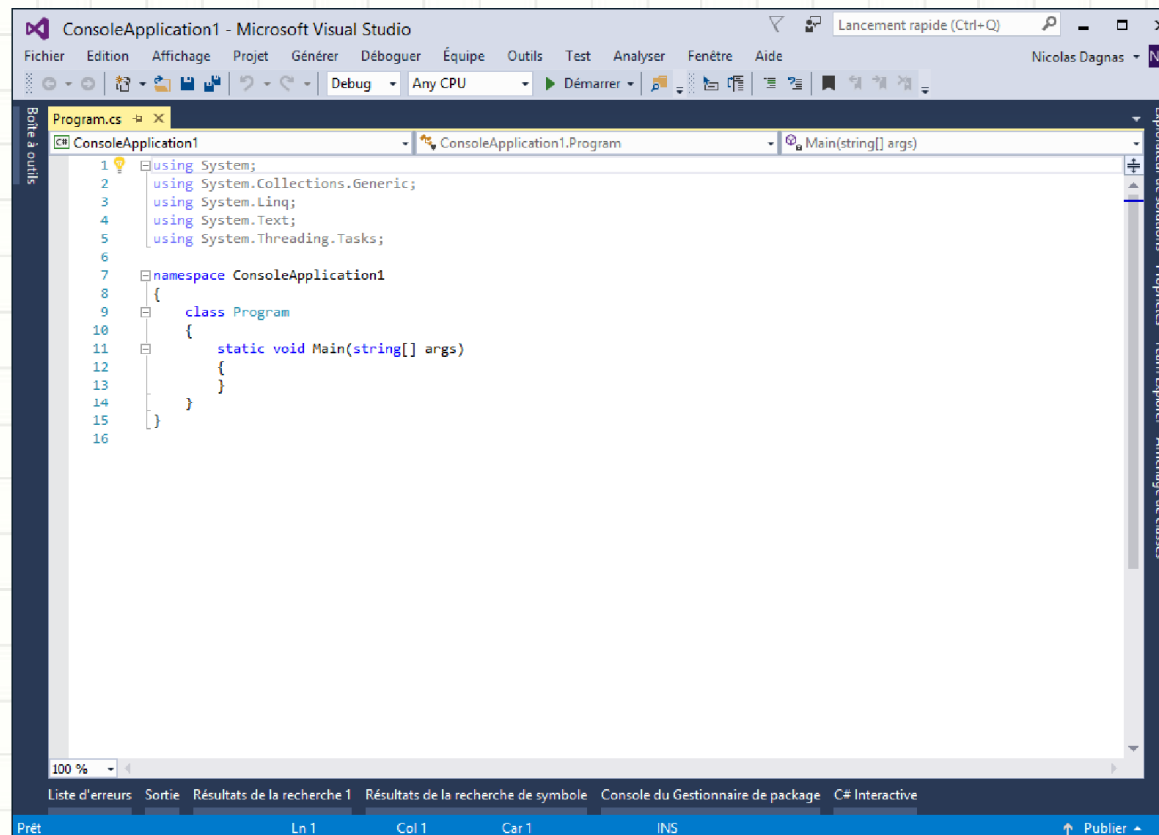
VS – Projet Console (1^{ère} partie)

- Nous allons commencer par créer un nouveau projet de type « Application console (.NET Framework) », en « C# » en qui sera le langage le plus proche de « Java » que vous connaissez déjà.



VS – Projet Console

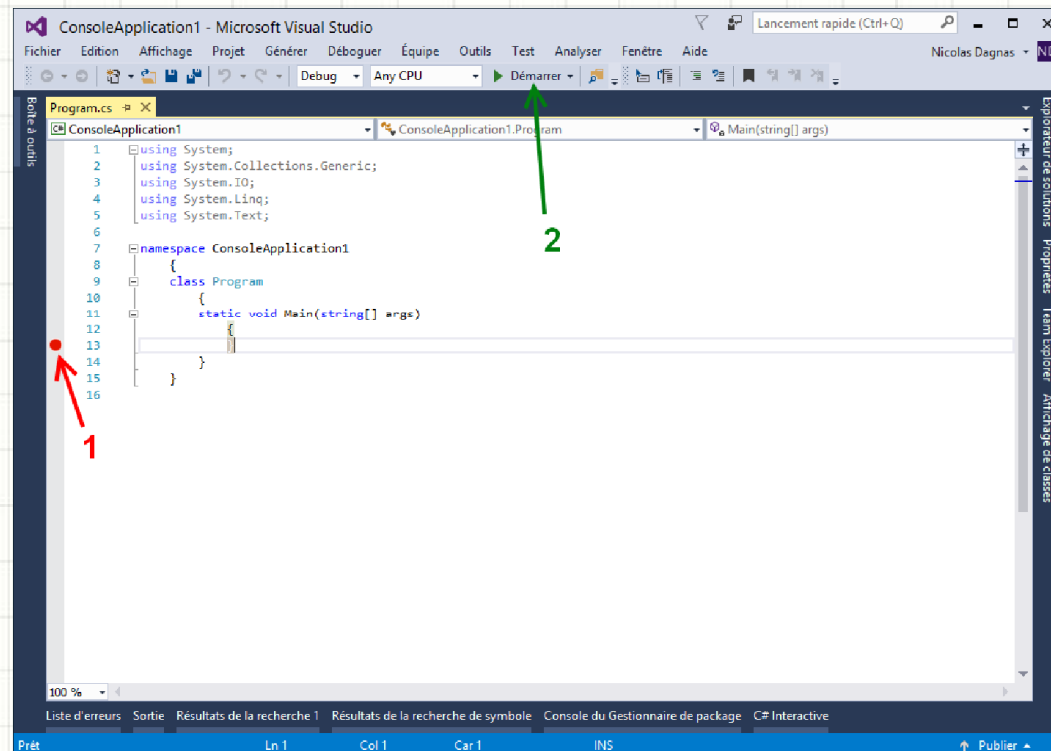
- On se retrouve avec le strict minimum, une procédure principale qui sera appelée lors de l'exécution du projet et c'est tout. Attardons nous tout de même sur « args » qui contient les éventuels paramètres passés à l'application. Mais comment y passer quelque chose ?



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ConsoleApplication1
8 {
9     class Program
10     {
11         static void Main(string[] args)
12         {
13         }
14     }
15 }
16
```

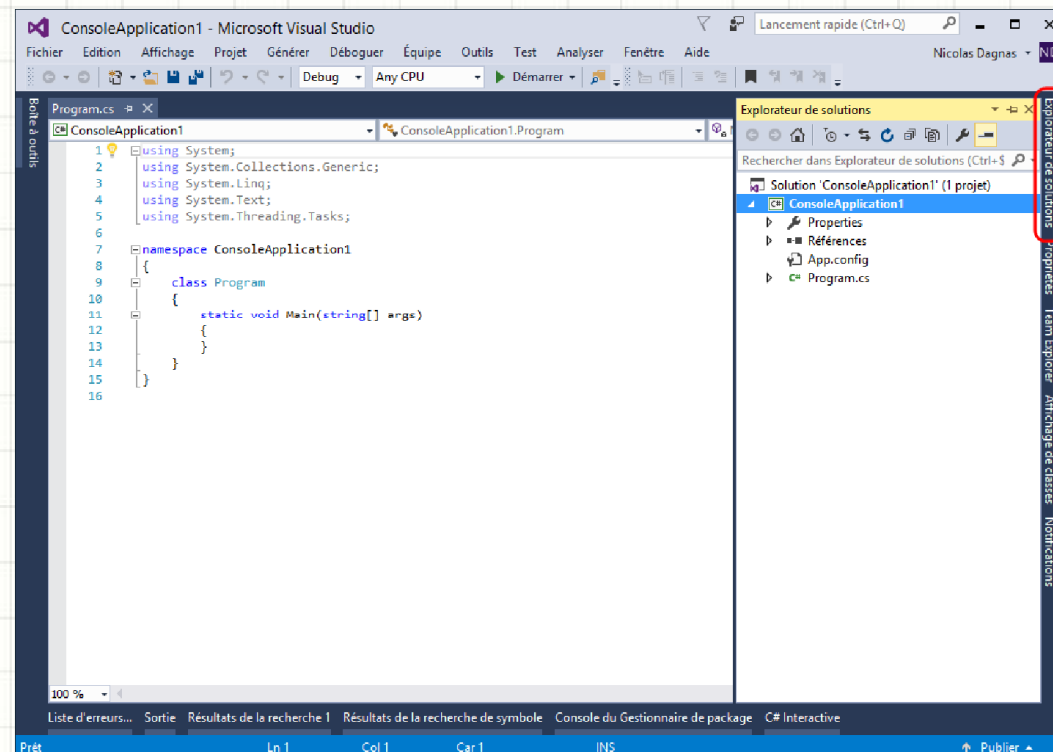
VS – Projet Console

- Pour commencer, nous allons placer un point d'arrêt comme dans la représentation ci-dessous en cliquant dans la marge en face de la ligne correspondante à l'accolade fermante de la procédure « Main ». Puis lancer le projet (Nous reviendrons plus tard sur le débogage). Une fois le projet arrêté sur cette ligne, passez la souris sur « args » qui est évidemment vide.



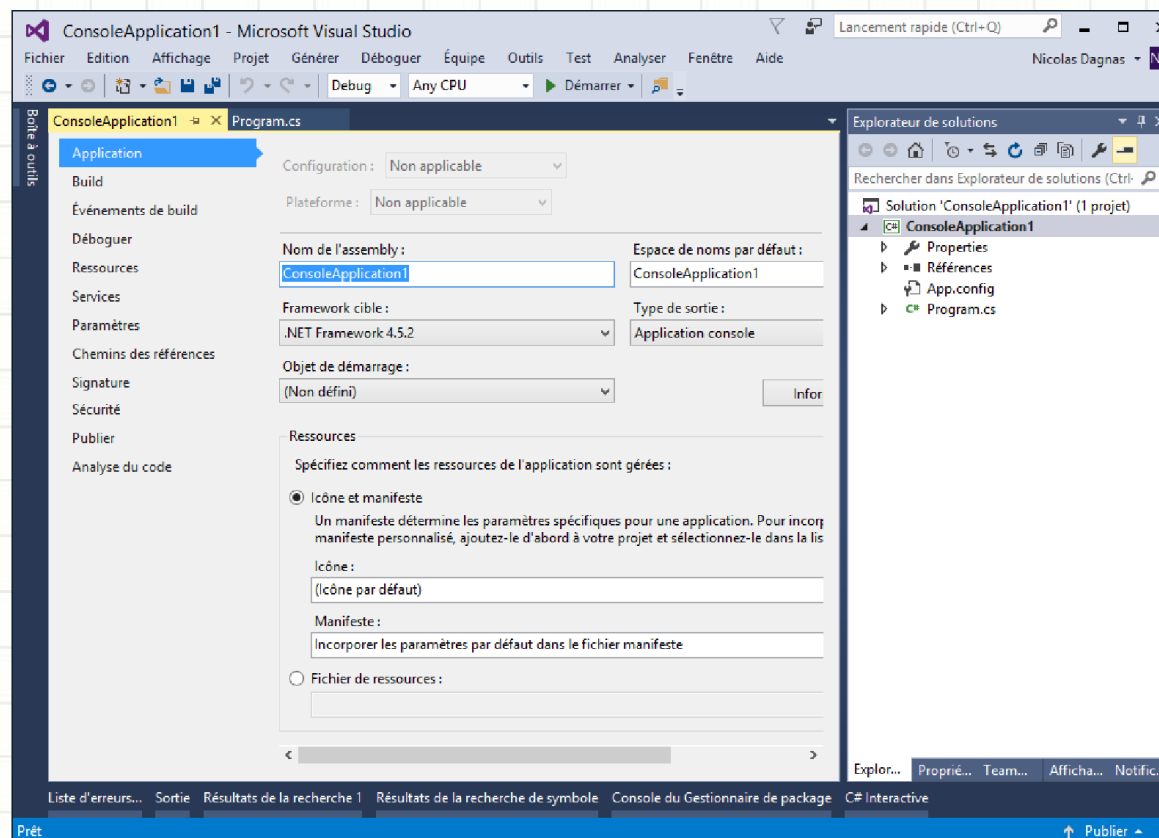
VS – Propriétés du projet

- Pour passer des valeurs en mode déboguage, cela se passe via les propriétés de l'application. Pour cela il suffit de cliquer sur « Explorateur de solution » puis de faire un clic-droit sur le projet (pas la solution) et simplement de cliquer sur « Propriétés ».



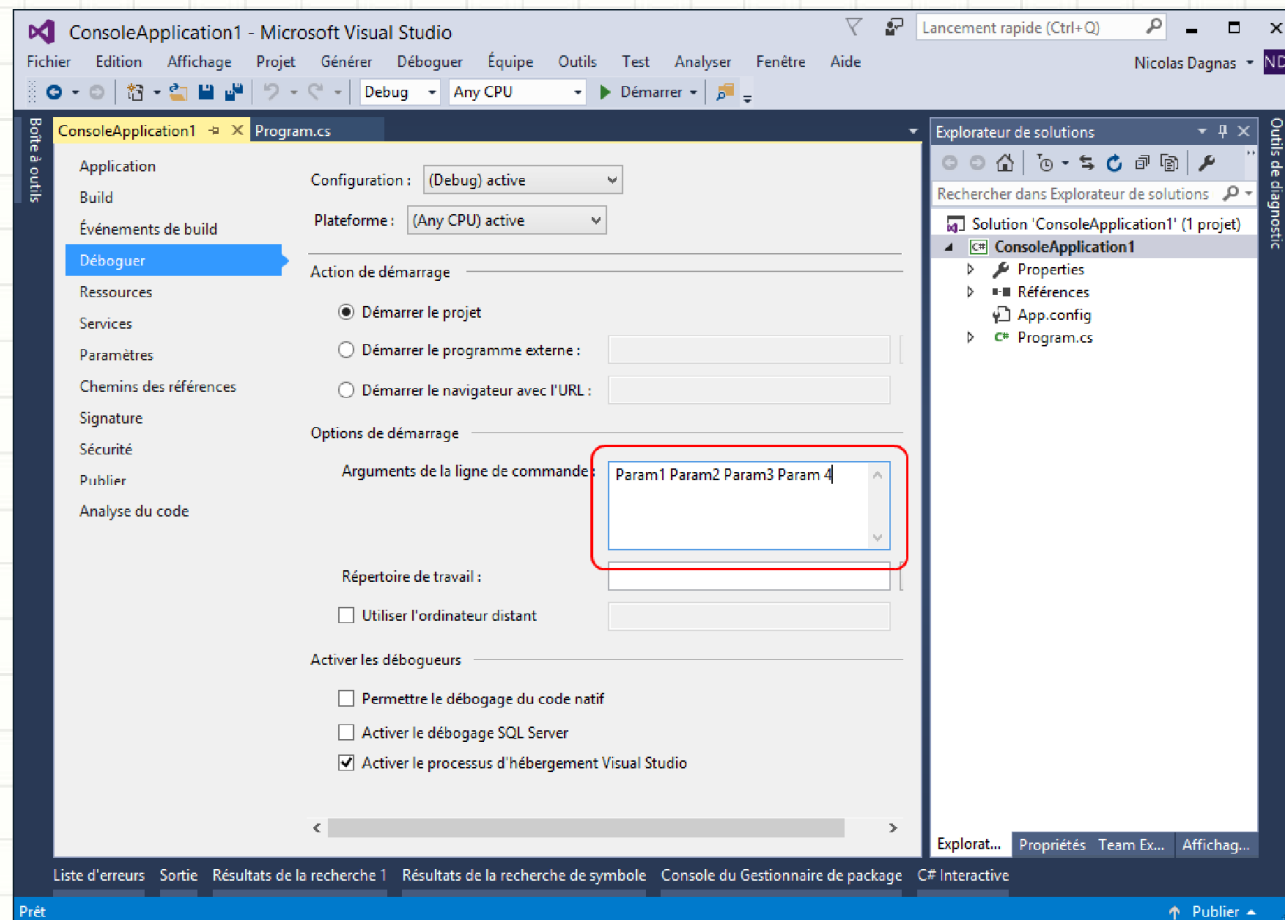
VS – Propriétés - Application

- On se retrouve par défaut sur l'onglet « Application » qui nous permet de modifier le nom de l'assembly (binaire générée), la version du framework à utiliser et l'icône de notre application. Pour l'instant c'est l'onglet « Déboguier » qui nous intéresse.



VS – Propriétés - Déboguer

- Nous trouvons dans l'onglet « Déboguer » la section « Options de démarrage », c'est ici que nous renseignerons les valeurs à passer à la variable « args » de notre méthode « Main ».



VS – Ligne de commande

- Maintenant, saisissez cette chaîne dans le champ « Arguments de la ligne de commande » : « Param1 Param2 Param3 Param 4 » avec les espaces entre les valeurs (n'hésitez pas à faire un copier - coller).
- Retournez dans votre méthode « Main » retirez le point d'arrêt précédent puis saisissez ce code :

```
class Program
{
    static void Main(string[] args)
    {
        for ( int Index = 0 ; Index < args.Length ; Index ++ )
        {
            Console.WriteLine ( string.Format ( "{0}: {1}" , Index, args[Index]) );
        }

        Console.WriteLine ( "Appuyez sur une touche pour quitter..." );

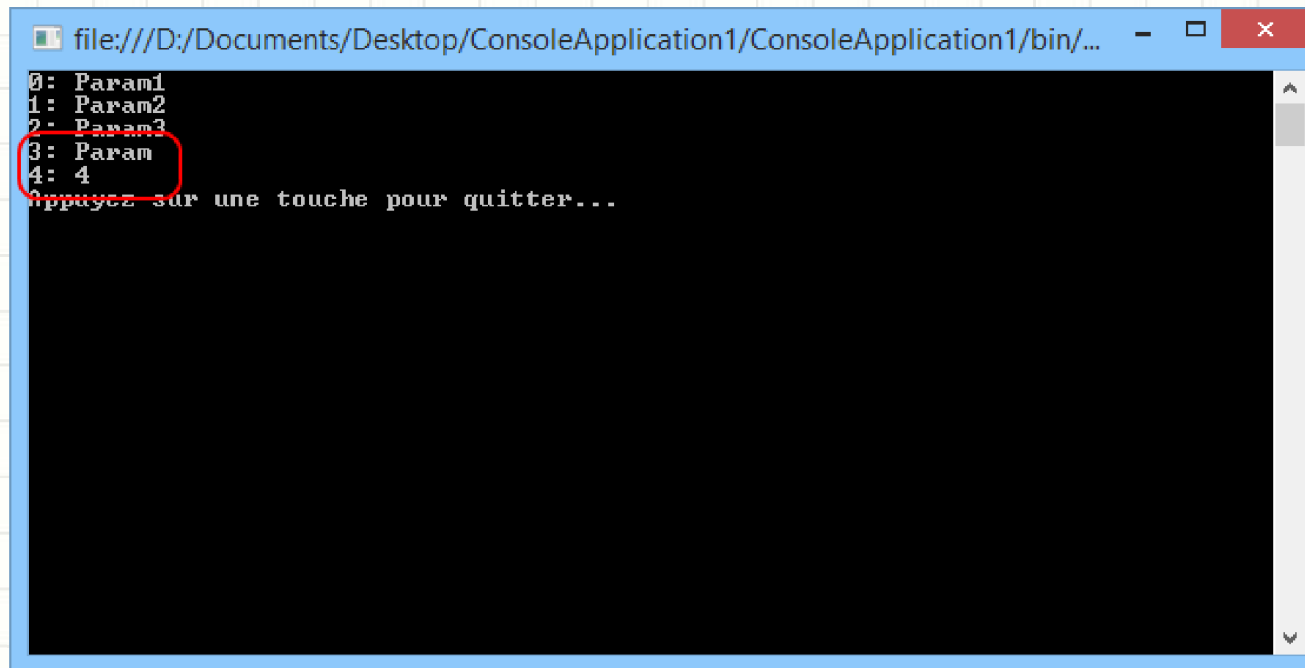
        Console.In.Read ();
    }
}
```


VS – Ligne de commande

- Dans le code que vous venez de saisir, vous avez sûrement remarqué la présence des deux lignes commençant par « Console ». Ces lignes permettent de communiquer avec notre fenêtre console.
- Un autre moyen nous permet de faire remonter des informations mais cette fois dans Visual Studio directement. On utilise l'objet « Debug » qui s'utilise de la même manière que « Console » pour la partie « Write » sans la partie « In » ou « Out ». Nous y reviendrons plus tard.

VS – Ligne de commande

- Exécutons maintenant notre bout de code et voyons ce que l'on obtient.



A screenshot of a Windows command prompt window. The title bar shows the file path: file:///D:/Documents/Desktop/ConsoleApplication1/ConsoleApplication1/bin/... The command prompt displays the following text: 0: Param1, 1: Param2, 2: Param3, 3: Param, 4: 4. The line '3: Param' is highlighted with a red rectangle. Below the arguments, the text 'Appuyez sur une touche pour quitter...' is visible.

```
file:///D:/Documents/Desktop/ConsoleApplication1/ConsoleApplication1/bin/...
0: Param1
1: Param2
2: Param3
3: Param
4: 4
Appuyez sur une touche pour quitter...
```

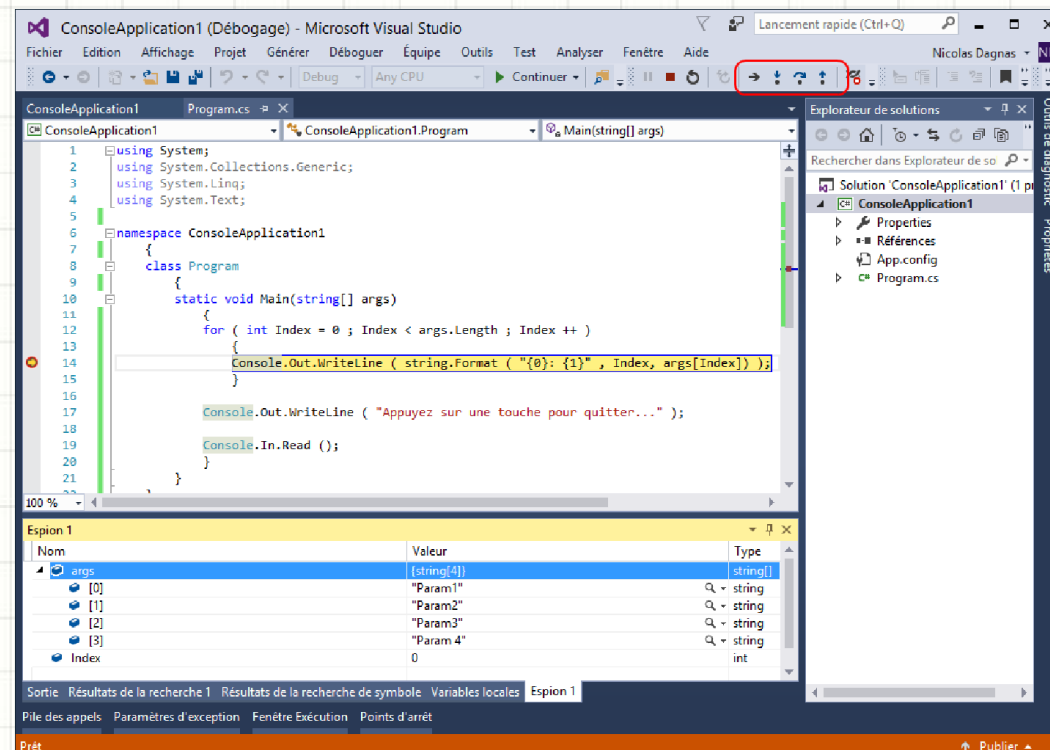
- Nous remarquons que « Param 4 » est découpé comme si c'était 2 paramètres. C'est normal car le système découpe la chaîne en détectant les espaces.
- Retournez dans les paramètres de votre projet et entourez « Param 4 » avec des doubles quotes. Puis relancez votre projet. Nous avons maintenant bien 4 paramètres.

VS – L'exécution Pas à Pas

- Regardons maintenant comment déboguer efficacement notre projet.
- Commençons par ajouter un point d'arrêt au niveau du « Console.Out... ». Puis lançons notre projet...
- Pendant que nous sommes en cours d'exécution, nous allons tout d'abord ajouter une fenêtre nommée « Espion ». Pour cela, faites :
 - Déboguer > Fenêtres > Espion (menu et fenêtre disponible uniquement à l'exécution).
- Faites un clic droit sur la variable « args » puis faites « Ajouter un espion ».
- Faites de même avec la variable « Index ».

VS – L'exécution Pas à Pas

- L'exécution va bien évidemment s'arrêter au point d'arrêt et nous permettre de voir le contenu de nos variables dans la fenêtre « Espion ».
- Les menus entourés permettent de continuer le débogage pas à pas.



VS – Namespace

- Maintenant que nous avons testé l'exécution, attardons nous un peu sur ce que nous voyons à l'écran. En haut à gauche, nous trouvons des lignes commençant par « using ».
- Ces lignes sont similaires aux « import » de « Java » en mode « .* ». En effet, il n'est pas utile en « C# » de préciser l'élément utilisé. L'option n'existe même pas.
- Par contre, comment faire pour connaître le « namespace » d'un objet ? Heureusement, Visual Studio (tout comme la plupart des E.D.I. pour Java) permet de le retrouver d'un clic droit.

VS – L'objet « Debug »

- Comme brièvement abordé précédemment, il est possible d'envoyer des informations « textes » à Visual Studio pendant l'exécution d'un projet. Pour se faire, il suffit d'écrire la ligne suivante :
 - `Debug.WriteLine ("...") ;`
- « Debug » reste en noir et c'est normal car nous n'avons pas intégré le « namespace » correspondant. Pour ce faire rien de plus simple, laissez le curseur sur « Debug » puis faites « Ctrl + ; ».
- Cette appel ajoutera le texte entre parenthèses à la fenêtre « Sortie » que vous devriez trouver sans difficulté aux cotés de la fenêtre « Espion 1 » en bas.
 - Déboguer > Fenêtres > Sortie (menu et fenêtre disponible uniquement à l'exécution).
- Un fait important est à prendre en compte cependant. Cet appel peut être gourmand en ressources et donc dans le cas de projets lourds en calculs algorithmique par exemple, les temps de traitement que vous pourriez souhaitez mesurer, seront faussés. Donc à utiliser avec parcimonie.

VS – Modes d'exécutions

- Pour l'instant, nous avons systématiquement exécuté notre projet en mode « Debug ». Il existe aussi un mode dit « Release ». Qu'est ce que cela implique ?
- Nous allons commencer par ajouter une ligne de code comme suit et placer un point d'arrêt devant la ligne for (si vous avez supprimés votre point d'arrêt précédent).

```
class Program
{
    static void Main(string[] args)
    {
        string VarInutilise = "";
        for ( int Index = 0 ; Index < args.Length ; Index ++ )
        {
            Console.Out.WriteLine ( string.Format ( "{0}: {1}" , Index, args[Index]) );
        }

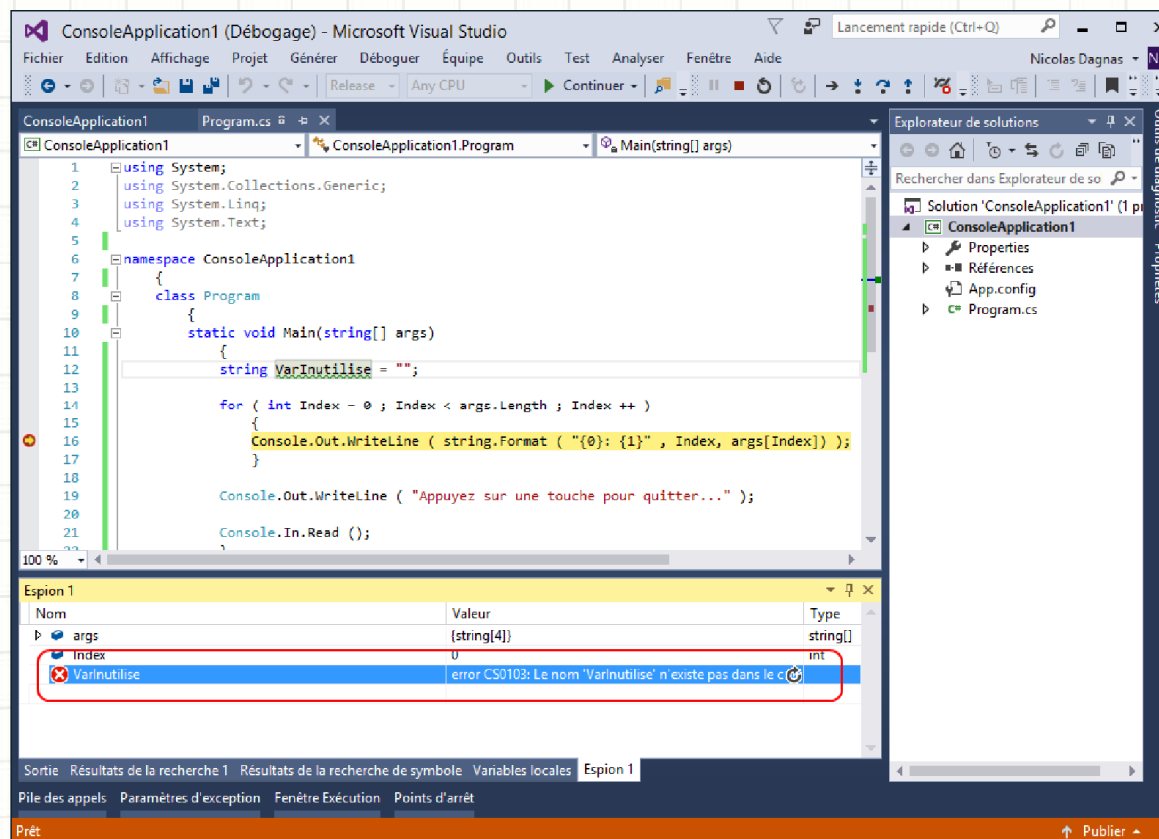
        Console.Out.WriteLine ( "Appuyez sur une touche pour quitter..." );

        Console.In.Read ();
    }
}
```

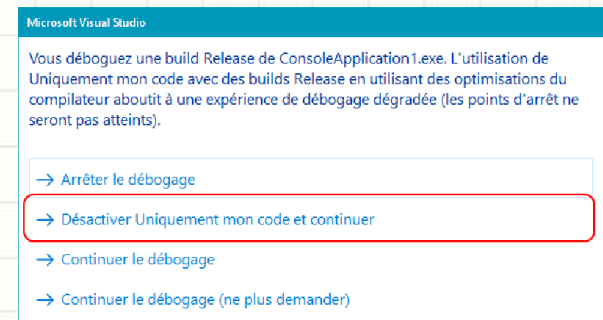
- Puis nous allons passer en mode « Release » et voir ce qui se passe quand on lance le déboguage. Il ne faudra pas oublier de renseigner dans ce mode, les arguments de la ligne de commande. Si vous avez un message de VS, lisez le slide suivant).

VS – Le Mode Release !

- Le mode « release » optimise le code. C'est-à-dire que les variables non utilisées ne sont plus lisibles lors du débogage. Cela se confirme par le fait que laisser la souris sur notre nouvelle variable ne donne rien, et si nous l'ajoutons à « Espion », le message est clair :



Si Visual Studio vous indique que vous déboguez une build Release et vous propose 4 options, sélectionnez : « Désactiver Uniquement mon code et continuer ».

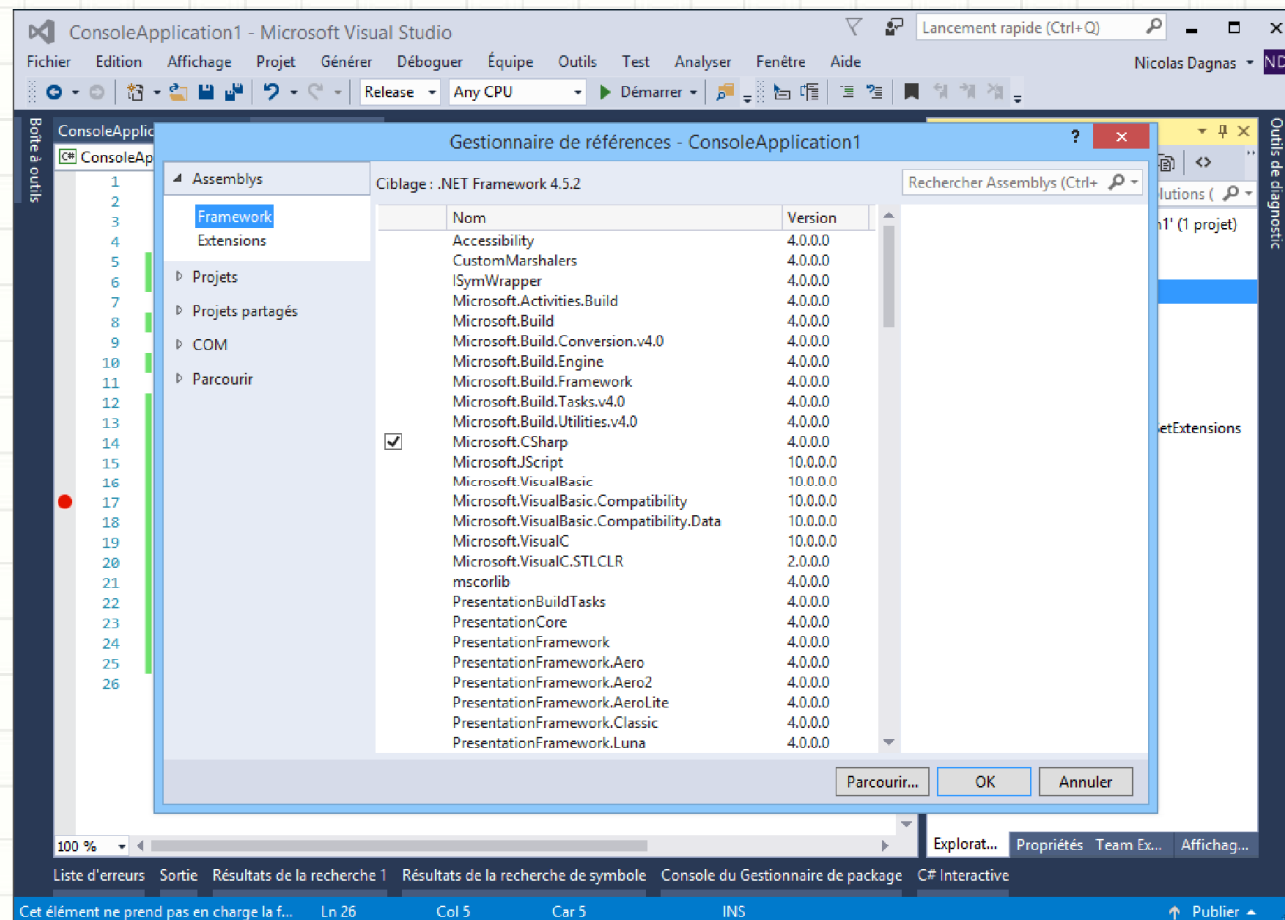


VS – Références

- Malheureusement, il se peut que l'E.D.I. ne connaisse pas le « namespace » d'un objet. C'est le cas quand une référence est manquante.
- Mais qu'est ce qu'une référence ? Une référence est un ensemble de *namespaces* contenant un nombre variable d'objets. Ce sont en fait des librairies (Des dlls principalement, comme un « .jar » en « java »).
- Mais où les trouver ? Il y a 3 sources principales de références :
 - Les références systèmes, les plus classiques bien sûr.
 - Les références fournies par un tiers.
 - Et enfin les grands classiques Nuget/GitHub et autres.

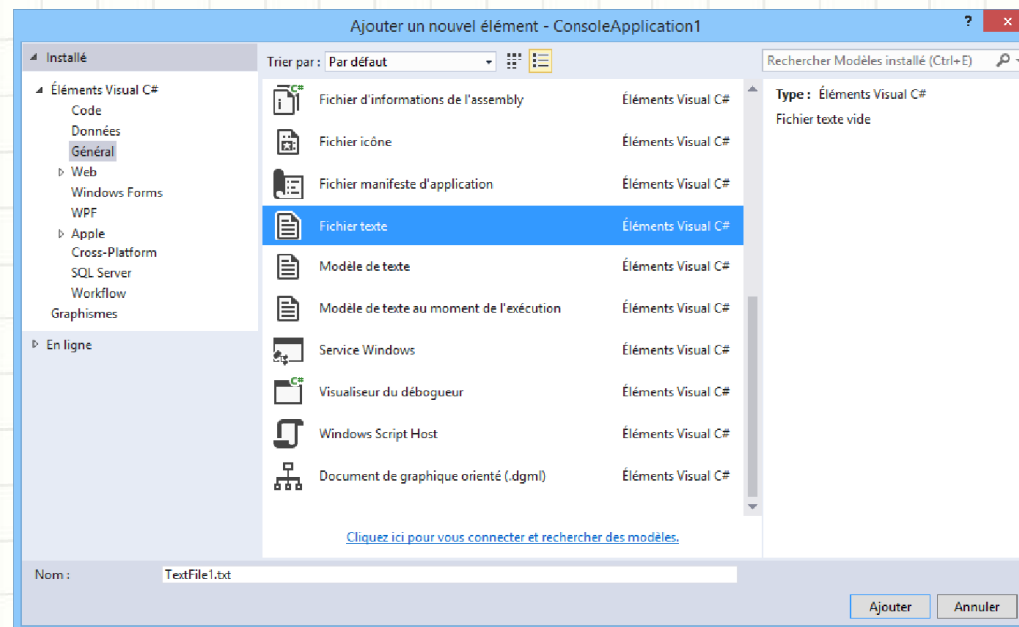
VS – Références

- Pour ajouter une référence, il suffit de faire un clic droit sur le projet et de faire « Ajouter une référence » :



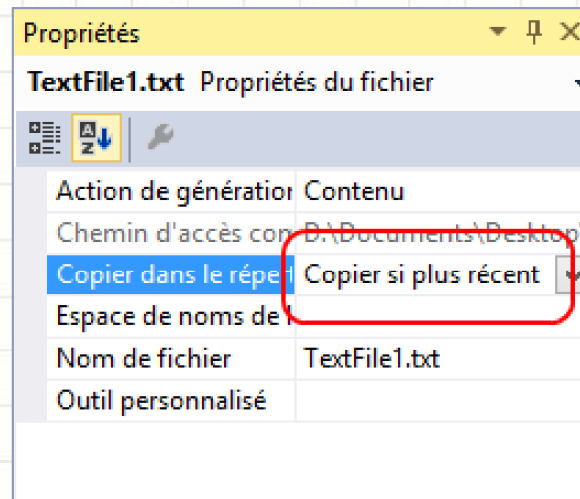
VS – Les Flux (lecture)

- Maintenant que nous connaissons les « namespaces » et les « références », nous allons commencer à coder notre lecture de fichier texte.
- Pour commencer, nous allons créer un fichier texte directement dans notre projet en faisant un clic droit sur notre projet puis « ajouter » et « nouvel élément » et rechercher la ligne « Fichier texte », nommez-le « MonFichier.txt ».



VS – Les Flux (lecture)

- Après avoir ajouté notre fichier texte, nous allons modifier ses propriétés.
- Faites un clic droit sur l'élément ajouté dans « l'explorateur de solution » (zone à droite), puis cliquer sur « Propriétés » et modifier la propriété « Copier dans le répertoire de sortie » comme suit (cette propriété nous sera utile dans le TP, donc gardez-la en mémoire) :



- Cela va indiquer à l'E.D.I. qu'il doit copier notre fichier texte dans le dossier d'exécution, mais seulement si nous modifions son contenu.

VS – Les Flux (lecture)

- Après avoir saisi du texte sur plusieurs lignes dans votre fichier texte, modifiez le contenu de votre méthode « Main » comme suit, et exécutez votre projet.

```
class Program
{
    static void Main(string[] args)
    {
        // Est ce qu'un paramètre a été passé à l'application ?

        if ( args.Length > 0 && File.Exists ( args[0] ) )
        {
            // Ouvrons le fichier

            StreamReader Sr = File.OpenText ( args[0] );

            string Ligne;

            while ( ( Ligne = Sr.ReadLine () ) != null )
            {
                // Affichons dans notre console, le contenu du fichier

                Console.Out.WriteLine ( Ligne );
            }

            Sr.Close (); // <= Fermons le fichier

            Sr.Dispose (); // Assurons nous que toutes les ressources sont libérées
        }

        Console.Out.WriteLine ( "Appuyez sur une touche pour quitter..." );

        Console.In.Read ();
    }
}
```

VS – Les Flux (lecture)

- Comme vous le constaterez, le contenu de votre fichier n'apparaît pas. Il y a 2 possibilités pour faire afficher le contenu de votre fichier :
 - Renseigner la section « Options de démarrage » avec le nom court de votre fichier comme il apparaît dans l'explorateur de solution : « MonFichier.txt ».
 - **Attention, cette pratique est très dangereuse car le programme cherchera le fichier dans le dossier d'exécution et l'utilisation de certains objets peuvent le modifier, comme l'objet « OpenFileDialog » et rendre cette pratique inopérante voir critique dans le cas de suppression de fichiers.**
 - Soit en renseignant cette même section mais avec le chemin complet. Pour cela récupérez le chemin de votre projet, ajoutez-y « bin » et « debug » puis le nom de votre fichier. Par contre, rappelez-vous que si le chemin dispose d'espace, il faudra entourer le chemin complet avec des « " ».

VS – Les ressources

- Comme vous avez pu le constater, il y a une ligne après le « Close » contenant un « Dispose ».
- Malheureusement de nombreux objets C# nécessite ce genre d'appel car un close ne libère pas toujours les ressources. Normalement le « Garbage collector » est censé nettoyer ces ressources, mais cela n'est pas toujours le cas.
- Heureusement, nous disposons de plusieurs optimisations dont l'utilisation du « Using Disposeur ».
- Certains objets dérivent donc de l'interface « IDisposable » permettant de réduire le code et surtout de ne pas oublier la ligne « .Dispose () », nécessaire à la libération des ressources.

VS – « Using disposeur »

- Ce « Using », à ne pas confondre avec le « Using » des namespaces, permet de créer un bloc indiquant au compilateur qu'il faudra « libérer » l'objet s'y trouvant une fois sortie de ce bloc.
- Si vous tentez d'y déclarer un objet qui n'implémente pas l'interface « IDisposable », vous aurez une erreur de compilation.
- Modifiez maintenant votre code comme suit, pour gagner en clarté.

```
using ( StreamReader Sr = File.OpenText ( args[0] ) )
{
    string Ligne;

    while ( ( Ligne = Sr.ReadLine () ) != null )
    {
        // Affichons dans notre console, le contenu du fichier

        Console.Out.WriteLine ( Ligne );
    }
}
```


VS – Les Flux (écriture)

- Maintenant que nous savons lire notre fichier texte, nous allons y écrire. Pour cela, nous utiliserons logiquement l'objet « StreamWriter » en opposition à l'objet « StreamReader » utilisé précédemment.
- Voici une partie du code que nous utiliserons :

```
if ( args.Length > 0 && File.Exists ( args[0] ) )
{
    // Ouvrons le fichier

    using ( StreamReader Sr = File.OpenText ( args[0] ) )
    {
        string Ligne;

        while ( ( Ligne = Sr.ReadLine () ) != null )
        {
            // Affichons dans notre console, le contenu du fichier

            Console.Out.WriteLine ( Ligne );
        }
    }

    using ( StreamWriter Sw = File.AppendText ( args[0] ) )
    {
        // Petit exercice de découverte ...
    }
}
```

VS – Les Flux (écriture)

- Et si on faisait un petit exercice ?
 - Ajoutez une méthode statique (comme « Main ») que vous nommerez « Demander_Et_Ecrire » prenant en paramètre d'entrée un objet « StreamWriter » et sans valeur de retour.
 - Le début de cette méthode affichera sur la console le texte :
 - Saisissez votre texte puis appuyez sur 'ENTER' ('exit' pour sortir).
 - Ajouter ensuite une boucle permettant à un utilisateur de saisir du texte.
 - Utilisez Console.In...
 - Lorsque l'utilisateur appuiera sur 'ENTER', ajoutez le texte saisi dans le fichier texte à l'aide de l'objet passé en paramètre, retour chariot compris. Indiquez par un texte dans la console que le texte saisi a été ajouté au fichier.
 - Lorsque l'utilisateur saisira 'exit' (ou même 'Exit' pour complexifier un peu) puis appuiera sur 'ENTER', vous sortirez de la boucle pour continuer normalement l'exécution de votre projet.
- Cette méthode est à appeler dans la section « using » que vous avez précédemment ajoutée.

VS – Les Flux (écriture)

- Une fois que vous avez codé votre méthode, testez là sans hésiter à recourir au déboguage comme vu précédemment.
- Pour voir le fichier texte modifié, allez dans le dossier « bin » puis « debug » de votre projet, passez par l'explorateur.
- Ensuite, rappelez-vous, quand nous avons ajouté notre fichier texte à notre projet, nous avons modifié une propriété : « Copier dans le répertoire de sortie ». Modifiez votre fichier encore ouvert dans l'E.D.I., enregistrez le, puis exécutez votre projet.
- Le contenu du fichier se trouvant dans « bin/debug » s'en retrouve écrasé par le contenu du fichier que vous venez de modifier.

VS – Intellisense

- Maintenant que nous avons créé et testé notre méthode, nous allons la documenter. Pour ce faire, nous utiliserons la fonctionnalité « Intellisense ». C'est très similaire à ce que l'on trouve dans « java » en un peu plus assisté.
- Pour commencer, Visual Studio permet de générer le bloc en une seule fois. Pour cela, il suffit de se placer sur la ligne (vide) juste au dessus de notre méthode et de taper le caractère « / » 3 fois de suite.

```
/// <summary>  
/// |  
/// </summary>  
/// <param name="Sw"></param>  
private static void Demander_Et_Ecrire ( StreamWriter Sw )  
{  
    // ...  
}
```


VS – Intellisense

- Renseignez les éléments « summary » et « param » comme suit :

```
/// <summary>
/// Demande à l'utilisateur de saisir du texte pour l'écrire dans le fichier.
/// </summary>
/// <param name="Sw">Objet <b>StreamWriter</b> permettant l'écriture.</param>
private static void Demander_Et_Ecrire ( StreamWriter Sw )
{
    // ...
}
```

- Puis en laissant la souris sur l'appel à cette méthode, nous voyons apparaître notre résumé :

```
using ( StreamWriter Sw = File.AppendText ( args[0] ) )
{
    // Petit exercice de découverte ...

    Demander_Et_Ecrire ( Sw );
}

void Program.Demander_Et_Ecrire(StreamWriter Sw)
    Demande à l'utilisateur de saisir du texte pour l'écrire dans le fichier.
```

- C'est très pratique pour s'y retrouver dans son projet !

VS – Les Exceptions

- Les exceptions sont très présentes en « C# ». Elles sont très (trop) souvent utilisées pour retourner une information, au lieu d'une valeur de retour.
- Devant faire avec, nous allons voir comment les utiliser.
- Pour cela, nous allons modifier notre procédure « Demander_Et_Ecrire » en générant une exception quand la valeur du champ « Sw » est vide. Cela prend cette forme :

```
private static void Demander_Et_Ecrire ( StreamWriter Sw )  
{  
    // Est ce que Sw est renseigné ?  
  
    if ( Sw == null )  
        throw new ArgumentNullException ( "Sw", "Sw vaut null !" );  
  
    // ...  
}
```

VS – Les Exceptions

- Pour intercepter une exception, et bien c'est comme partout, c'est avec un « Try ... Catch ».

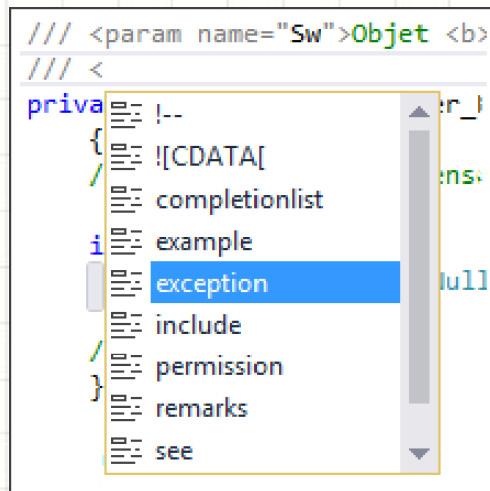
```
try
{
    using ( StreamWriter Sw = File.AppendText ( args[0] ) )
    {
        // Petit exercice de découverte ...

        Demander_Et_Ecrire ( Sw );
    }
}
catch ( ArgumentNullException )
{
    // Si on arrive ici, c'est qu'un argument vaut NULL
    // et que cela a généré une exception !
}
catch ( Exception Err )
{
    // Ici une autre exception a été générée :/
}
}
```

Vous remarquerez que le bloc « `ArgumentNullException` » n'a pas de variable déclarée. Cela est autorisé en « C# ». Si cette variable n'est pas utilisée, cela permet de compiler le projet sans avoir de « Warning » comme nous avons actuellement pour le second bloc.

VS – Intellisense (Suite)

- « Intellisense » va nous être utile aussi pour les exceptions car il permet de les documenter. Placez-vous à la fin de la dernière ligne de notre bloc permettant de décrire notre méthode, faites « Entrée » puis saisissez le caractère « < », cela vous propose une liste de possibilités et celle qui nous intéresse est « exception » :



- Cela va générer la ligne qu'il faudra renseigner du type de l'exception et de sa description.
- Ensuite, si nous replaçons la souris sur l'appel à notre méthode, une ligne supplémentaire est apparue.

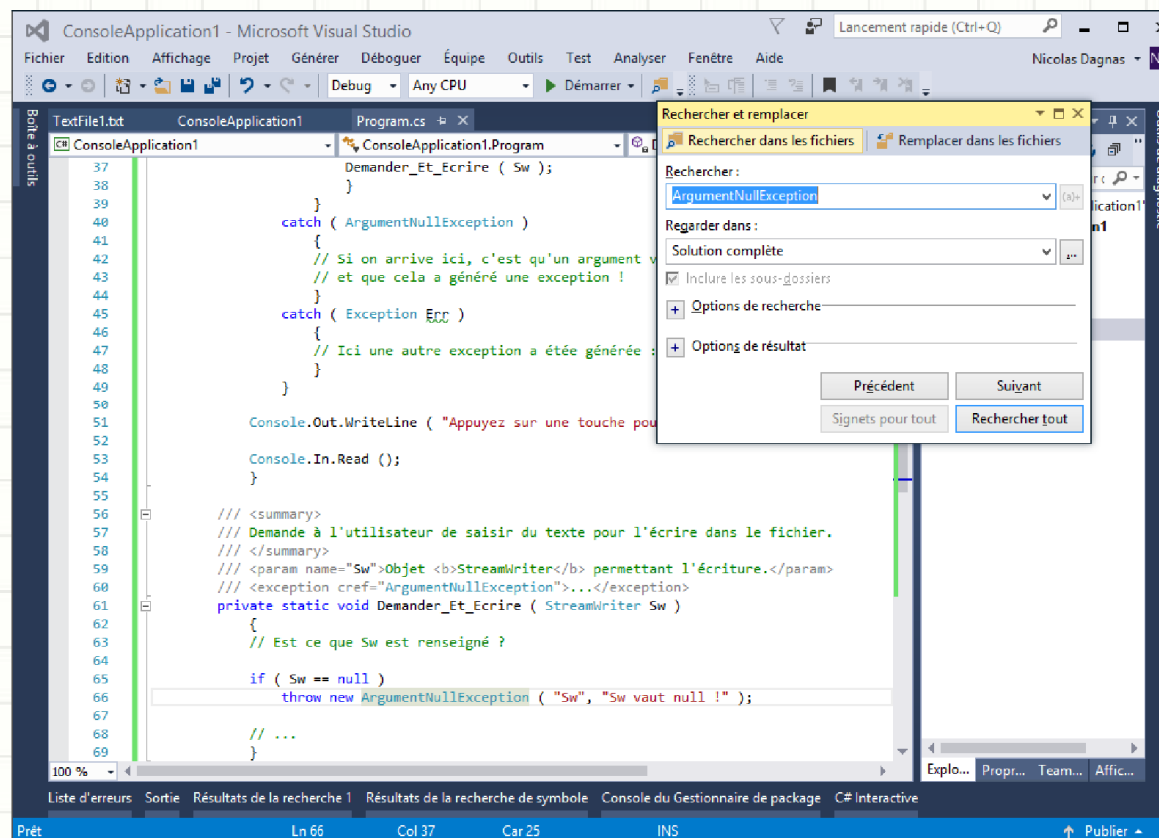
```
/// <summary>
/// Demande à l'utilisateur de saisir du texte pour l'écrire dans le fichier.
/// </summary>
/// <param name="Sw">Objet <b>StreamWriter</b> permettant l'écriture.</param>
/// <exception cref="ArgumentNullException">...</exception>
```


VS – Raccourcis clavier

- Quelques raccourcis claviers dans Visual Studio :
 - F5 : Lancer l'application
 - F10 : Exécuter jusqu'à la ligne suivante
 - F11 : Exécuter en entrant dans les fonctions
 - F12 : Permet d'aller à la déclaration de l'objet sélectionné
 - Ctrl+C : Copier la ligne entière avec le retour chariot
 - Ctrl+X : Couper la ligne entière avec le retour chariot
 - F6 : Compiler le projet (Sans l'exécuter)
 - Ctrl+F : Effectuer une recherche dans la page en cours
 - Ctrl+Alt+F : Effectuer une recherche globale au projet
 - Ctrl+K > Ctrl+C : Placer la ligne en commentaire
 - Etc...

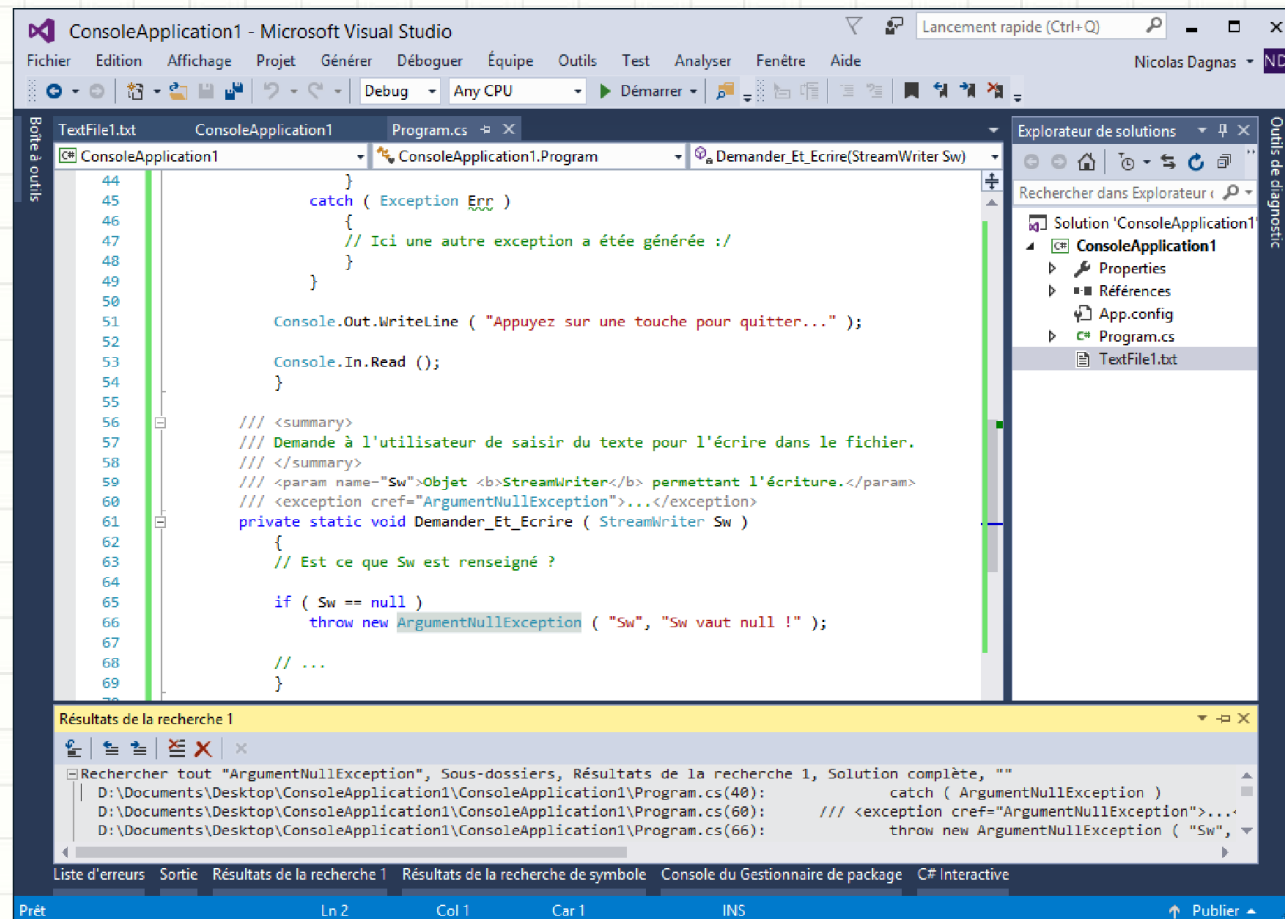
VS – La Recherche classique

- « Ctrl+F » permet comme partout de lancer une recherche locale (fichier en cours).
- « Ctrl+Maj+F » permet de lancer une recherche globale, c'est-à-dire, dans tout le projet.



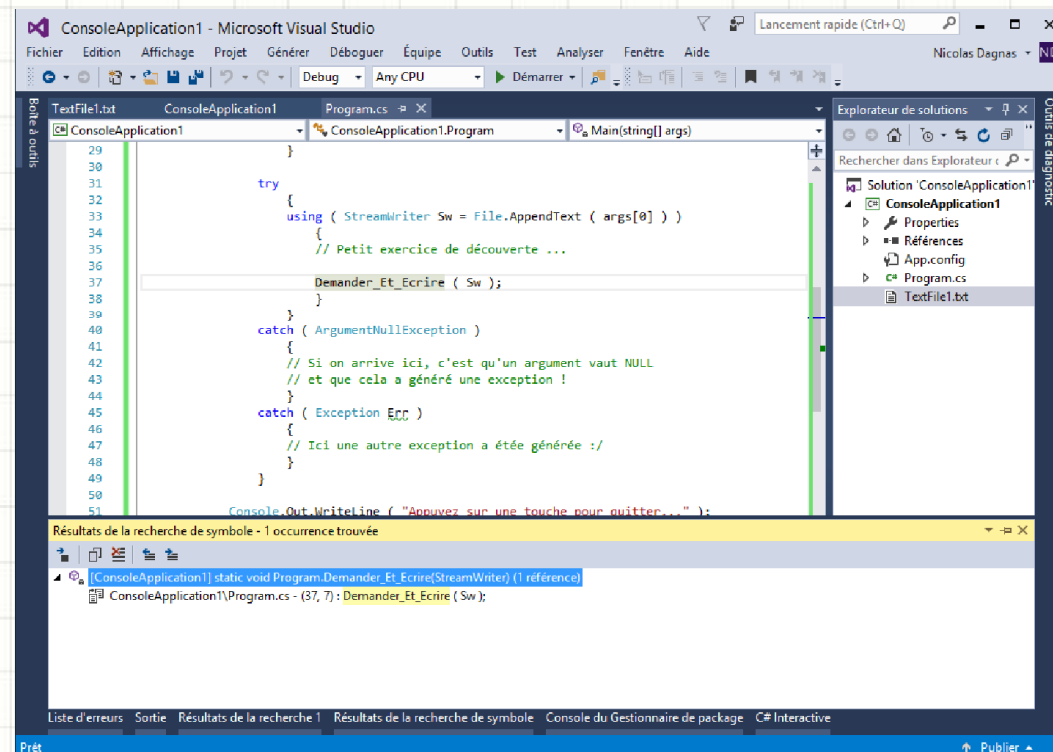
VS – Résultat de la recherche

- À l'instar de Netbeans ou d'autre E.D.I., le résultat est affiché dans une zone à part mais bien plus détaillée, quoique !



VS – La Recherche de référence

- En faisant un clic droit sur un élément du code, nous avons la ligne « Rechercher toutes les références » qui va nous permettre de rechercher tous les appels ou utilisations de cet élément, en excluant les noms similaires (2 méthodes d'objets différents portant le même nom).
- Cela peut s'avérer très pratique lors de modifications du contenu d'une méthode pour s'assurer de gérer les différents cas et éviter les effets de bord.

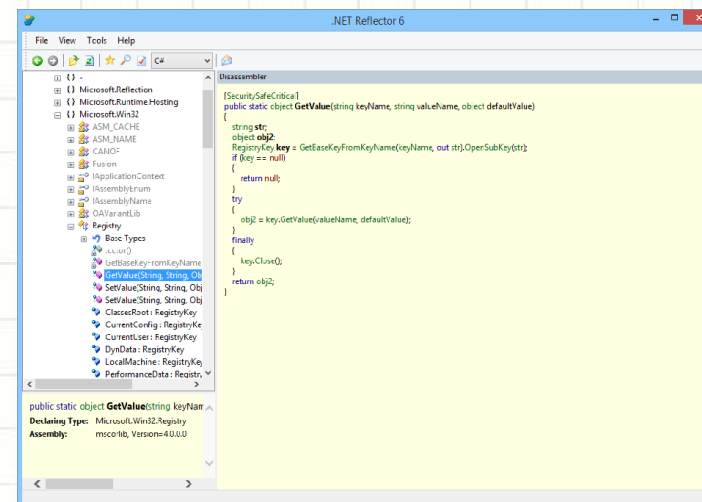


VS – Mes Fichiers !

- Petit rappel des fichiers composant le projet :
 - Le fichier « Solution » => .sln, pouvant référencer plusieurs projets.
 - Le fichier « Projet » => .csproj, permet de constituer notre projet.
 - Le fichier « User » => .user, contient les paramètres d’affichage du projet en cours, comme les éléments ouverts.
 - Les fichiers « Sources » => .cs, ...
 - Le dossier « Properties » => en général, il n’est jamais modifié.
 - Les dossiers « bin » et « obj » => contient les fichiers compilés.

VS – Dé-compilateurs

- Le langage C# étant un langage semi compilé, on trouve des utilitaires très pratiques permettant de voir le code d'un binaire.
- La plupart sont payant mais vous trouverez dans le section 1 du cours (célène) une archive contenant une version gratuite (ancienne version) de « Reflector » qui encore aujourd'hui fonctionne très bien.
- Utilisez-là sur votre projet pour voir comment le code est remanié lors de la compilation.
- Depuis Vs 2019, on dispose d'un outil intégré équivalent : Outils - Options - C# - Avancé.

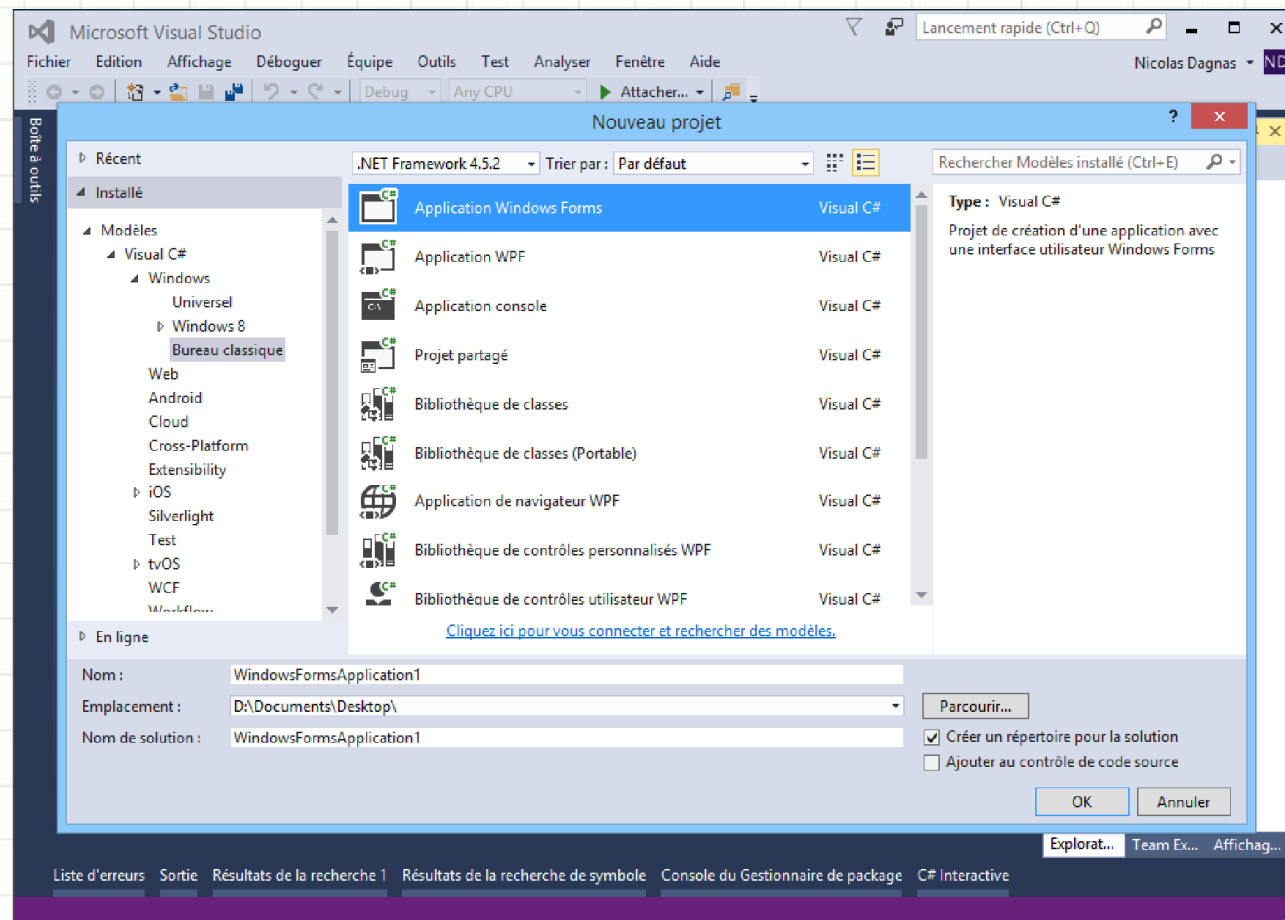


VS – Fin de la première partie

- Maintenant vous savez globalement utiliser Visual Studio. Vous aurez compris qu'en lui-même il permet déjà de faire pas mal de choses et ce même sans « add-on ».
- Vous avez également vu l'utilisation de « .Net Framework ». Vous avez probablement entendu parler de « .Net Core » et comme cité plus haut de « Xamarin ». Ils ne sont pas concurrents, car chacun correspond à un besoin particulier :
 - « .Net Framework » : destiné aux applications et services basiques windows.
 - « .Net Core » : se destine plus à l'environnement UWP, c'est-à-dire le Store Windows par exemple et la création d'applications pour Windows IOT.
 - « Xamarin » : pour tout ce qui sera cross-platform « Android », « iOS » et « OS-X ».
- Mais tous se basent sur « .Net Standard » qui en est la base.

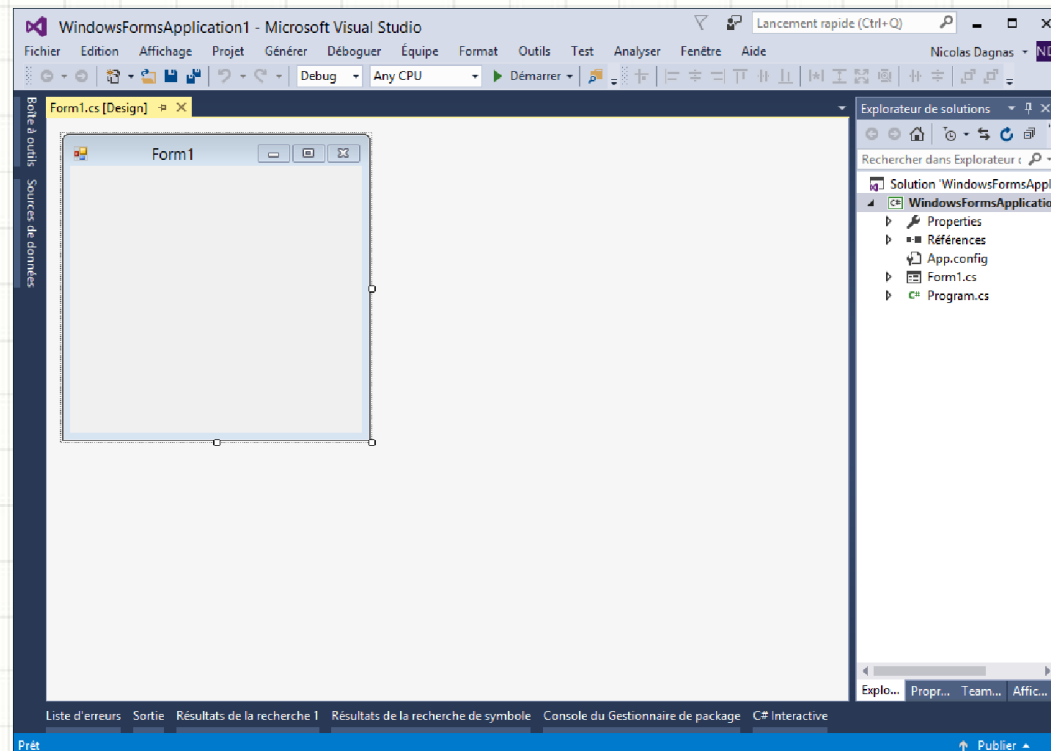
VS – Projet WinForm (2ème partie)

- Nous allons maintenant créer un nouveau projet mais cette fois du type « Application Windows Form (.NET Framework) », toujours en « C# » bien sûr.



VS – Projet WinForm

- Une fois validé, nous arrivons sur un écran contenant la fenêtre principale de notre application.
- Dans l'explorateur de solution, nous retrouvons à peu de choses près les mêmes éléments que pour notre projet console. Les composants comme les zones d'éditations, les boutons, etc se trouvent dans la boîte à outils à gauche.



VS – Projet WinForm – Exercices

- Exercice 1 :
 - Configurez votre fenêtre principale pour qu'elle mesure 700*500 pixels, qu'elle s'affiche systématiquement (au lancement de notre application) centrée par rapport à l'écran (via les propriétés), et pour finir donnez-lui comme titre « Éditeur de fichier texte ».
 - Ajoutez un menu « Fichier » contenant deux sous-menus : « Ouvrir » et « Enregistrer sous... ».
 - Ajoutez un objet « TextBox » configuré en mode multi-ligne et qui devra recouvrir l'écran principal avec une marge de 10 pixels.
 - Lancez votre projet. En redimensionnant votre fenêtre, vous constatez que l'objet « TextBox » ne se redimensionne pas.
- Exercice 2 :
 - Configurez l'élément « TextBox » pour qu'il se redimensionne quand vous modifiez la taille de votre fenêtre (à l'exécution) ; regardez du côté de « Anchor » et « Dock », mais lequel choisir ?

VS – Projet WinForm – Exercices

- Exercice 3 :
 - Codez votre sous-menu « Ouvrir » (double cliquez sur votre sous-menu pour générer l'appel) pour qu'il affiche une fenêtre de sélection de fichier (mode ouverture) vous permettant de sélectionner un fichier texte (masque '*.txt' à configurer).
 - À la validation de la fenêtre de sélection, chargez le fichier sélectionné dans votre objet « TextBox ».
- Exercice 4 :
 - Codez votre sous-menu « Enregistrer sous... » pour qu'il affiche une fenêtre de sélection de fichier (mode enregistrement) vous permettant de sélectionner un fichier texte (masque '*.txt' à configurer) et demandant confirmation quand le fichier de destination existe.
 - À la validation de la fenêtre de sélection, enregistrez le contenu de votre objet « TextBox » dans le fichier sélectionné.

VS – Projet WinForm – Exercices

- Exercice 5 :
 - Lorsque vous chargez un fichier, faites apparaître dans le titre de votre fenêtre principale, le nom du fichier (sans le chemin) entre crochets après le titre, sous la forme « Éditeur de fichier texte [...] ».
 - Quand vous modifiez le contenu chargé dans votre « TextBox » en tapant du texte, ajoutez une « * » à la fin du titre de votre fenêtre, étoile qui devra disparaître quand vous enregistrez.
 - Quand vous fermez votre fenêtre sans avoir enregistré et que votre contenu a été modifié et non enregistré dans votre éditeur (étoile présente), affichez une fenêtre de confirmation de type Oui/Non/Annuler demandant si vous souhaitez vraiment quitter l'application sans enregistrer.

VS – Projet WinForm – Exercices

- Exercice 6 :
 - Détectez, à la prise de focus de votre fenêtre principale, si le fichier ouvert a été modifié depuis une autre source (autre éditeur), puis demandez à l'utilisateur s'il souhaite recharger le contenu.
 - Ne lisez pas le fichier pour vérifier, contentez-vous de la date de dernière modification.
 - Vous pouvez utiliser également jouer avec « FileSystemWatcher ».
 - Ensuite, prévoyez le cas où le contenu a été modifié et non enregistré, en demandant si l'utilisateur souhaite remplacer son contenu actuel.

VS – Projet WinForm – Exercices

- Exercice 7 :
 - Maintenant qu'on a notre éditeur de texte, passons à ce que l'on trouve généralement dans tout logiciel fenêtré : « L'enregistrement du contexte ».
 - Ici rien de trop compliqué. Utilisez simplement l'objet « Settings ».
 - Accédez à la section « Paramètres » des propriétés de votre projet.
 - Ajoutez les différents valeurs nécessaires : Taille, Position.
 - Accédez ensuite à ces valeurs depuis votre code. Il faudra ensuite les mettre à jour également depuis votre code et s'assurer de la prise en compte des modifications.
 - Enfin, faites en sorte que votre fenêtre principale se retrouve systématiquement dans son état précédent lors de son lancement, position, taille et état agrandie ou non. Attention, si vous fermez votre application lorsqu'elle est en partie en dehors de l'écran, il faut qu'au re-lancement elle se trouve intégralement dans l'écran. Utilisez l'objet « Screen » pour accéder aux informations de l'écran.

VS – Projet WinForm – Exercices

- Exercice 8 :
 - Modifiez votre méthode « Main » du fichier « Program.cs » pour ajouter le paramètre « args ».
 - Maintenant, utilisez ce paramètre pour passer un nom de fichier (Utilisez les propriétés de votre projet pour y mettre une valeur, comme vu dans la première partie).
 - Vous devrez donc vérifier si la chaîne passée est bien un nom de fichier et que le fichier existe. Puis ouvrez le fichier dans votre fenêtre principale comme si vous aviez fait « Ouvrir ».
 - Quand vous aurez finit cela, allez dans le dossier de sortie de votre projet « bin » puis « debug », vous y trouverez votre application. Prenez un fichier texte puis glissez le sur son icône et constatez ce qu'il se passe.

VS – Dernière ligne droite

- Exercice 9 :
 - Pour finir, nous allons permettre un peu de personnalisation.
 - Placez un séparateur en dessous du menu « Enregistrer sous ... ».
 - Ajouter en dessous du séparateur un menu « Personnaliser la police ». Ce menu devra utiliser l'objet « FontDialog » pour permettre à l'utilisateur de spécifier la police du contenu de champ « TextBox ». Cette personnalisation devra bien entendu survivre au redémarrage de l'application.
 - Faire de même avec un menu « Personnaliser la couleur du texte » à l'aide de l'objet « ColorDialog ». Même punition concernant la sauvegarde du contexte ;)