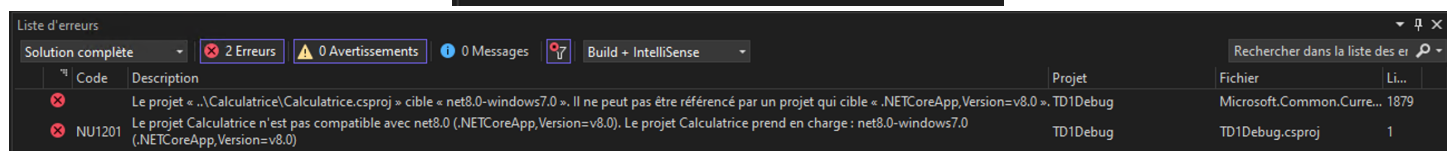
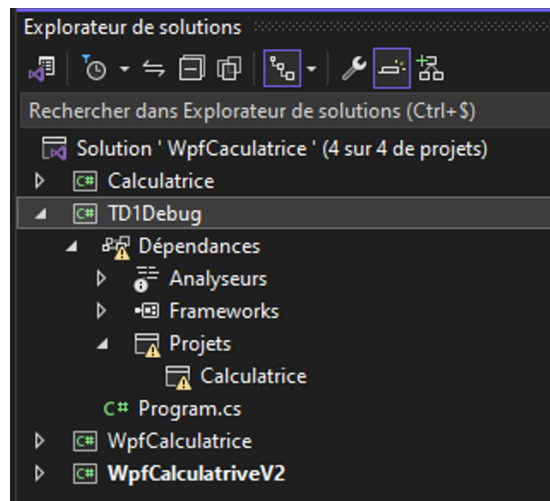


**Ne commencer ce TD que si le TD1 est terminé.**

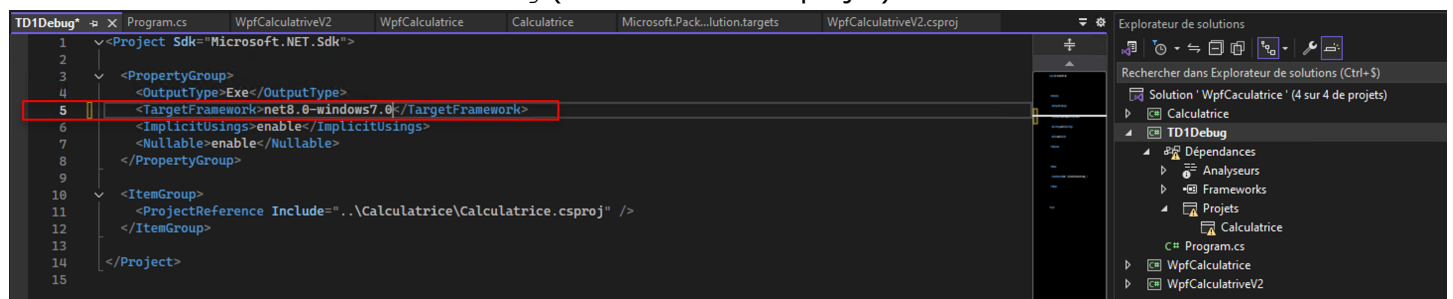
**Bonne pratique : bien maîtriser les différents outils de debugging de votre environnement de développement (Visual Studio).**

## Mise en place

1. Créer un nouveau projet TD1Debug de type « Application console », dans la solution du TD1.  
**Utiliser le framework .NET 8.**
2. Rajouter une référence vers le projet bibliothèque de classes Calculatrice.
3. Définir le projet TD1Debug comme *Projet de démarrage de la solution*.
4. Vous verrez alors les erreurs suivantes :

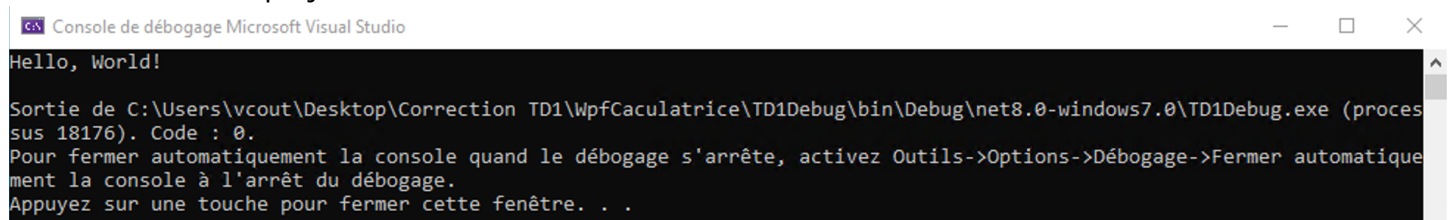


5. Comme le projet bibliothèque de classes Calculatrice utilise le framework net8.0-windows7.0, modifier le framework de TD1Debug (double-clic sur le projet) :



Enregistrer le fichier.

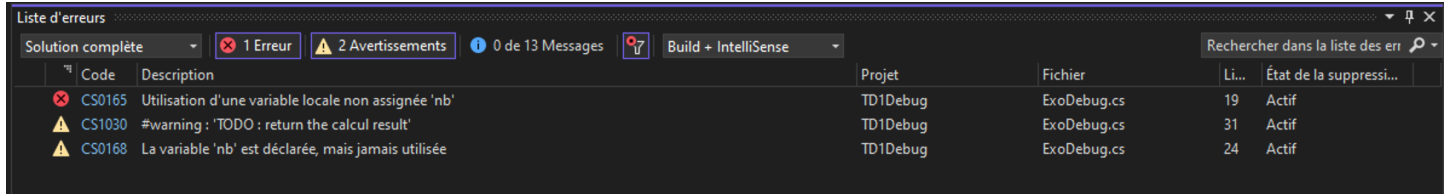
6. Exécuter le projet :



## Exercice 1 - Erreur / Warning

Ajouter le fichier `ExoDebug.cs` au projet `TD1Debug` (fichier disponible sur le serveur).

La première étape du debug est de regarder les erreurs / avertissements indiqués par Visual Studio.



Supprimez ces erreurs / avertissements dans le code (on pourra complètement supprimer ces fonctions ou les mettre en commentaire).

Corriger les erreurs est indispensable pour compiler. Corriger les warnings peut permettre d'éviter des potentielles erreurs d'exécution et rend le code plus propre (suppression du code inutile, etc.)

**Note :** On peut ajouter "artificiellement" des avertissements dans son code grâce à l'instruction `#warning`. Cela peut être intéressant pour indiquer des portions de code à tester particulièrement, à terminer, etc. Ne cependant pas en abuser : si le projet contient trop de warnings, on ne les regarde plus.

## Exercice 2 - Assert

L'assertion permet de vérifier des prédicats. Un cas classique est de vérifier que les préconditions des fonctions sont respectées (regarder le code de l'exercice 2 dans le fichier `ExoDebug`). Les assertions sont très utilisées lors des tests unitaires.

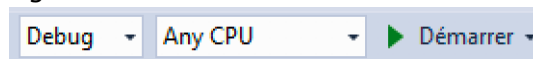
.NET propose des fonctions d'assertion qui sont exécutées uniquement en Debug.

Instancier un objet `ExoDebug` dans le programme principal (`Program.cs`). Appeler la méthode `Exercice2()`.



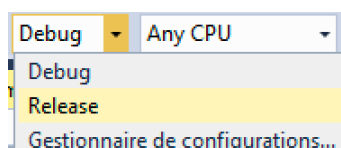
Remarque : la méthode `Main` est maintenant cachée, mais votre code s'exécute bien dans cette méthode contenue dans `Program.cs`. Cliquer sur le lien indiqué en entête de fichier pour en apprendre plus sur cette modification qui date du framework Net 6 (novembre 2021).

Compiler en Debug et lancer le programme.



Que se passe-t-il ? (Une assertion a échoué).

Compiler en mode "Release" :



Relancer le programme. Bien cliquer sur « Désactiver uniquement mon code... ». Que se passe-t-il ?

Rien : en mode d'exécution release qui sera utilisée pour la production, l'assertion sera ignorée.  
Bien revenir en mode Debug.

**Note :** La méthode `System.Diagnostics.Debug.Assert` n'est pas à utiliser dans tous les cas. On peut utiliser cette méthode pour, par exemple, revérifier une saisie utilisateur. Mais en aucun cas elle ne doit être la seule validation des données, car ces vérifications ne seront plus faites en Release.




Plus de détails ici :

<https://learn.microsoft.com/fr-fr/dotnet/api/system.diagnostics.debug.assert?view=net-8.0>

### Exercice 3 - Step By Step / Breakpoint Simple / Call Stack

Le debugging étape par étape est un outil que l'on utilise très souvent pour debugger. Pour être efficace avec, il est nécessaire de maîtriser les différentes opérations :

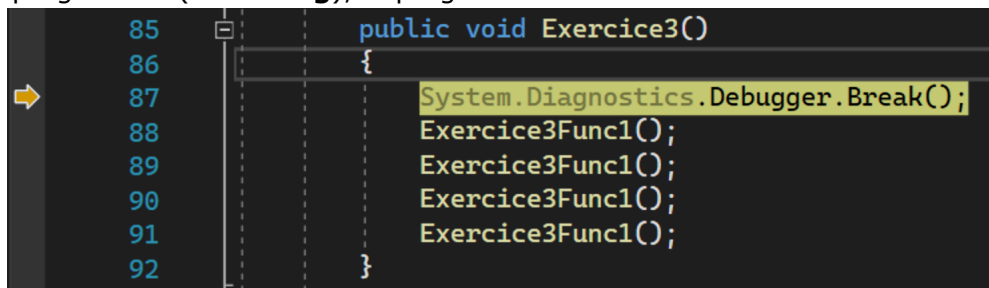


- Pas à Pas Détaillé  : Rentrer dans la fonction
- Pas à Pas Principal  : Aller à la ligne suivante (ne pas rentrer dans la fonction)
- Pas à Pas Sortant  : Aller directement à la fin de la fonction en cours

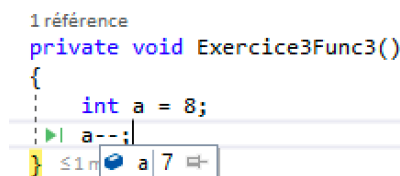
Il est également nécessaire de savoir ajouter des *BreakPoints* (points d'arrêt) et naviguer d'un BreakPoint à l'autre.

Exercice :

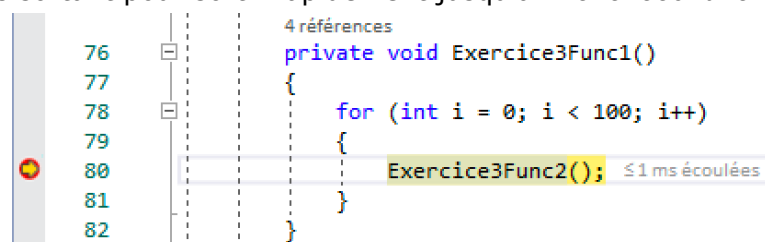
1. Supprimer l'appel à `Exercice2()` dans le Main
2. Ajouter un appel à `Exercice3()`
3. Lancer le programme (**en Debug**), le programme s'arrête sur un BreakPoint "artificiel".



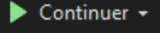
4. Grâce au pas à pas détaillé, aller jusqu'à rentrer dans la fonction `Exercice3Func3()`. En utilisant le pas à pas détaillé et en survolant une variable avec la souris, on peut voir sa valeur. Une fois la ligne exécutée, la valeur change.



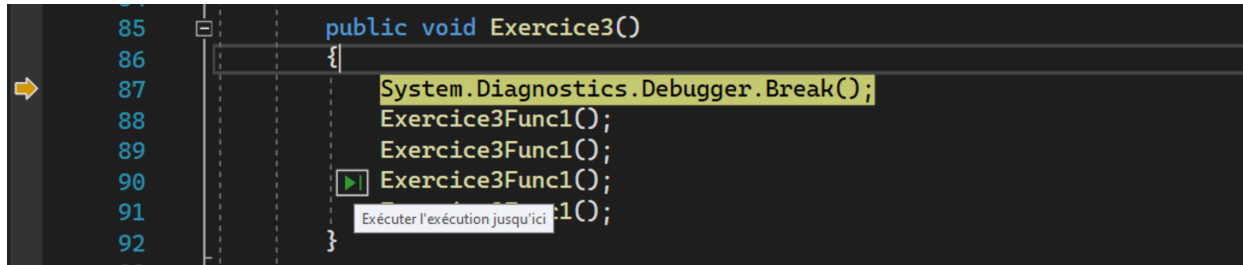
5. Utiliser le pas à pas sortant pour sortir rapidement jusqu'à `Exercice3Func1()` (dans la boucle)



6. Ajouter un point d'arrêt dans la boucle et un dans la fonction `Exercice3Func3` :

7. Utiliser le bouton "Continuer"  pour naviguer entre les 2 BreakPoints.
8. Après quelques tours, supprimer tous les BreakPoint (Menu *Debugger > Supprimer tous les points d'arrêts*).
9. En utilisant le Pas à Pas sortant revenir à la fonction principale (`Exercice3()`).
10. Aller jusqu'à la fin de la fonction en utilisant le Pas à Pas Principal.

Une fois un breakpoint atteint, il est aussi possible d'exécuter du code jusqu'à un point suivant sans ajouter de point d'arrêt supplémentaire, grâce au curseur vert :

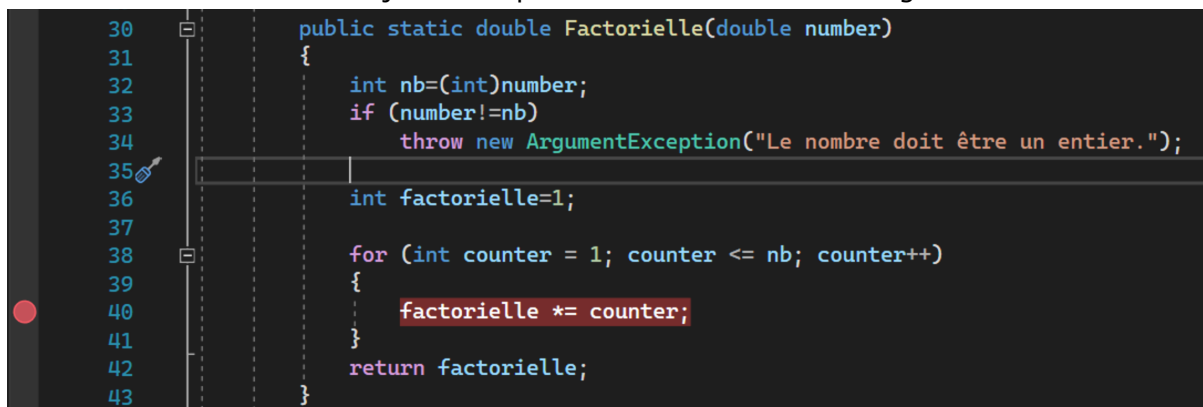


## Exercice 4 - Watch / Breakpoint Conditionnel / Trace

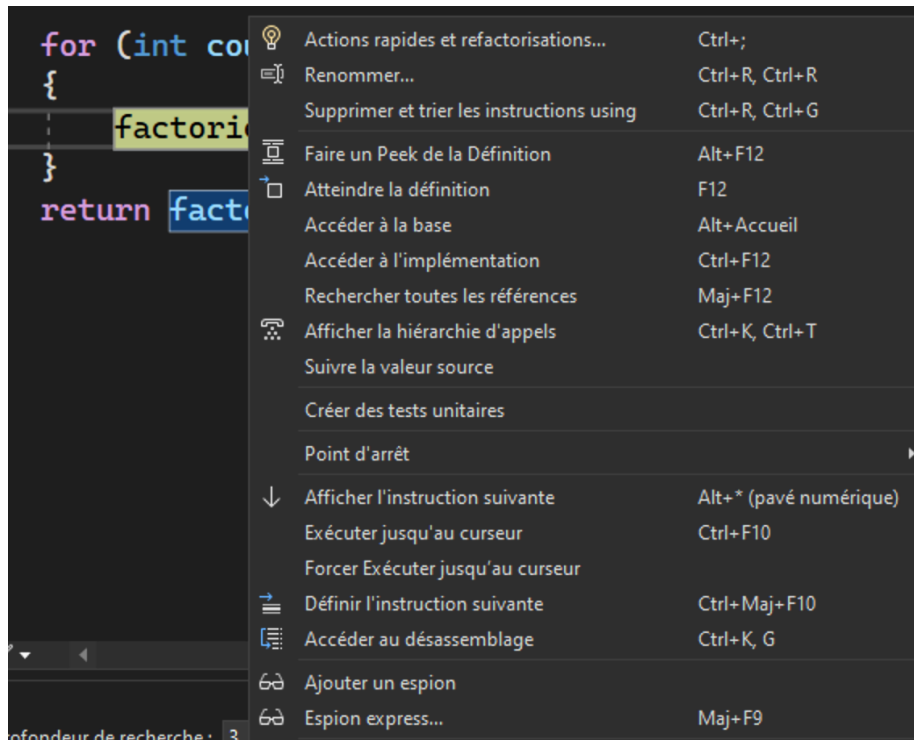
Les espions (ou Watch) permettent d'afficher simplement des variables. Ils sont surtout utiles dans des boucles pour afficher différentes variables.

### Exercice 4.1 - Watch :

- Dans le Main, remplacer `Exercice3()` par `Exercice4_1()`.
- Dans la méthode Factorielle ajouter un point d'arrêt à l'instruction figurant dans la boucle `for`



- Lancer le programme. Il s'arrêtera sur la ligne `System.Diagnostics.Debugger.Break();` Continuer l'exécution.
- Quand on arrive sur le point d'arrêt, ajouter un espion sur la variable à l'intérieure de la boucle `for`, ici `factorielle` (sélectionner la variable, puis *Clic Droit*, puis *Ajouter un espion*)



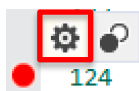
- Continuer l'exécution
- ⇒ On voit la variable `factorielle` évoluer

Espion 1		
Rechercher (Ctrl+E)  Profondeur de recherche : 3		
Nom	Valeur	Type
factorielle	24	int
Ajouter un élément à espionner		

#### Exercice 4.2 - BreakPoint conditionnel

- Remplacer `Exercice4_1()` par `Exercice4_2()`.
- Lancer l'exécution.
- Sans modifier le code, ni utiliser l'avancement pas à pas, retrouver directement à l'aide d'un BreakPoint la valeur de Fibonacci de **12**

**Indice :** utiliser un BreakPoint conditionnel sur la ligne `return result;` de la fonction Fibonacci



```

return n;
int result = Fibonacci(n - 1) + Fibonacci(n - 2);
return result;

```

**Réponse :** 144 !

#### Exercice 4.3 - BreakPoint conditionnel

- Remplacer `Exercice4_2()` par `Exercice4_3()`.
- Lancer l'exécution.
- Sans modifier le code, ni utiliser l'avancement pas à pas, retrouver directement la factorielle de 8

**Indice :** utiliser un BreakPoint par nombre d'accès

#### Exercice 4.4 - Trace

- Remplacer `Exercice4_3()` par `Exercice4_4()`.
- Ajouter 2 points de traces (clic droit de la souris) :

```

30 public static double Factorielle(double number)
31 {
32     int nb=(int)number;
33     if (number!=nb)
34         throw new ArgumentException("Le nombre doit être un entier.");
35
36     int factorielle=1;
37
38     for (int counter = 1; counter <= nb; counter++)
39     {
40         factorielle *= counter;
41     }
42     return factorielle;
43 }

```

Paramètres de point d'arrêt

Emplacement : Calcul.cs, ligne : 34, caractère : 17, doit correspondre à la source

☐ Conditions

☒ Actions

Afficher un message dans la fenêtre Sortie :

☒ Continuer l'exécution du code

☐ Supprimer le point d'arrêt une fois atteint

☐ Activer uniquement quand le point d'arrêt suivant est atteint

Paramètres de point d'arrêt

Emplacement : Calcul.cs, ligne : 40, caractère : 17, doit correspondre à la source

☐ Conditions

☒ Actions

Afficher un message dans la fenêtre Sortie :

☒ Continuer l'exécution du code

- Lancer l'exécution.
- Dans la fenêtre de Sortie (et non la fenêtre d'exécution !), on peut voir les traces (en bas de la fenêtre Visual Studio) :

```

Sortie
Afficher la sortie à partir de : Débugger
'TD1Debug.exe' (CoreCLR: clrhost) : Chargé 'C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.8\System.Collections.Concurrent.dll'.
Je suis passé par la factorielle
Je suis passé par la factorielle
Je suis passé par la factorielle
Je suis passé par la factorielle
Je suis passé par la factorielle
Je suis passé par la factorielle
Je suis passé par l'erreur
Exception levée : 'System.ArgumentException' dans Calculatrice.dll
Une exception non gérée du type 'System.ArgumentException' s'est produite dans Calculatrice.dll
Le nombre doit être un entier.

```

Plus de détails ici :

<https://learn.microsoft.com/fr-fr/visualstudio/debugger/navigating-through-code-with-the-debugger?view=vs-2022&tabs=csharp>

## Exercice 5 - #if DEBUG

En .Net il existe une directive permettant de changer le code que l'on compile selon que le code soit compilé en Debug ou en Release.

Exercice :

- Remplacer l'appel à `Exercice4_4` par l'appel à `Exercice5`
- Lancer le programme en Debug. Qu'affiche la console ?
- Lancer le programme en Release. Qu'affiche la console ?

**Note :**

- Le code qui ne sera pas compilé est affiché en grisé dans Visual Studio
- Utiliser cette directive avec parcimonie (afficher plus de détails pour le debug, changer une config, etc.), mais ne pas changer le comportement du programme. Le code deviendrait beaucoup plus complexe et difficile à debugger.