LPDIM– Debug – création de dll

INFO

Partie 1 : Debug

Mise en place

1. Créer un nouveau projet `TP1Debug` de type application console **(.Net Core 3.1**).
2. Définir le projet `TP1Debug` comme Projet de Démarrage de la solution

Exercice 1 – Création de la classe Pays

Créez la classe `Pays(Id,Nom)` l'id est un entier supérieur ou égal à 1 et le nom doit être non vide et non nul.

En mode sans débogage (F5) déclenchez les deux types d'exception (Id et Nom) et vérifiez l'affichage obtenu. En mode débogage (CTRL+F5) déclenchez les deux types d'exception (Id et Nom) et vérifiez que le débogueur vous renvoie vers la bonne ligne de code à l'origine du problème.
Le mode débogage est donc à privilégier !!

Créez une liste de 3 pays dans le main afin de préparer le suite du TP :

```
Pays p1 = new Pays(1, "France");
Pays p2 = new Pays(2, "Irlande");
Pays p3 = new Pays(3, "Allemagne");
```

Exercice 2 - Erreur / Warning

Mise en place : Ajouter le fichier `TdDebug.cs` au projet.

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

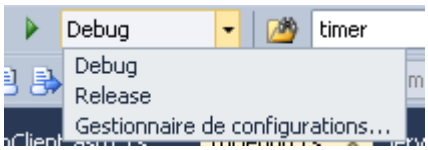
Supprimez ces erreurs / avertissements dans le code. (On pourra complètement commenter des fonctions).
Corriger les erreurs est indispensable pour compiler. Corriger les warnings peut permettre d'éviter des potentiels erreurs d'exécution, et rend le code plus propre (Suppression du code inutile, etc.)
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 3 - Assert

L'assertion permet de vérifier des prédicats. Un cas classique est de vérifier que les préconditions des fonctions sont respectées. Les assertions sont très utilisées lors des Tests Unitaires.

Instancier un objet `TdDebug` dans le `Main`. Appeler la méthode `Exercice3()` depuis le `Main`. Compiler en Debug et lancer le programme. Que se passe-t-il ? (Une assertion a échoué).

Compiler en mode “Release” :



Relancer le programme en mode « sans débogage ». Que se passe-t-il ? (Rien, en mode release, version 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.

Exercice 4 - Step By Step / Breakpoint Simple / Call Stack

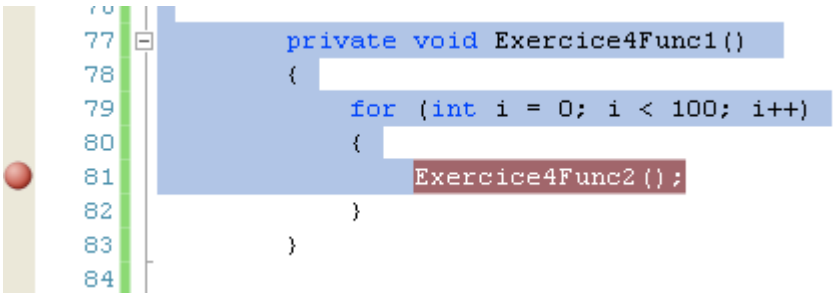
Le debugging étape par étape est un outil que l’on utilise très souvent pour debugger. Pour être efficace avec, il est nécessaire de maîtriser les différentes opérations :

- 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 BreakPoint (points d’arrêt) et naviguer d’un BreakPoint à l’autre.

Exercice :

1. Supprimer l’appel à `Exercice3();`
2. Ajouter un appel à `Exerice4();`
3. Lancer le programme, le programme s’arrête sur un BreakPoint “artificiel” :
   `System.Diagnostics.Debugger.Break();`
4. Grace au pas à pas détaillé, aller jusqu’à rentrer dans la fonction `Exercice4Func3()`
5. Utiliser le pas à pas sortant pour sortir rapidement jusqu’à `Exercice4Func1()` (dans la boucle)



6. Ajouter un point d’arrêt dans la boucle et un dans la fonction `Exercice4Func3` :
7. Utiliser le bouton “Continuer” (Fleche verte) pour naviguer entre les 2 BreakPoints.
8. Après quelques tours, supprimer tous les BreakPoint (Menu Deboguer ⇒ Supprimer tous les points d’arrêts) ;
9. En utilisant le Pas à Pas sortant revenir à la fonction principale (`Exercice4()`).
10. Aller jusqu’à la fin de la fonction en utilisant le Pas à Pas Principal.

Exercice 5 - Watch / Breakpoint Conditionnels

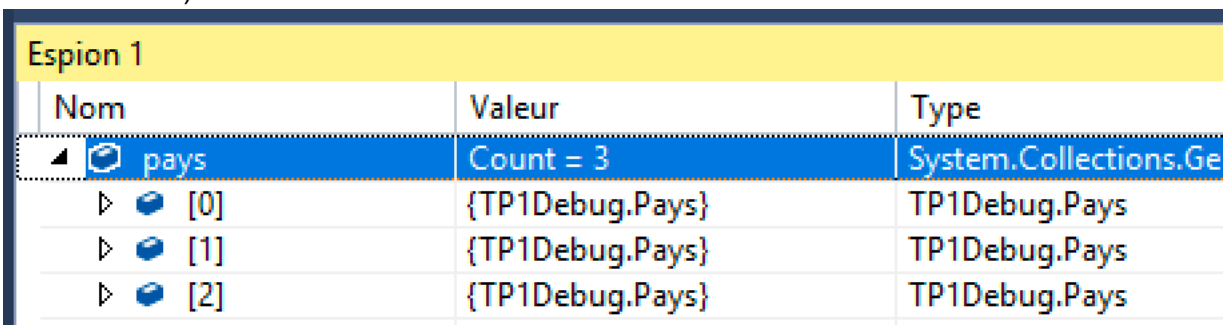
Les espions (ou Watch) permettent d'afficher simplement des variables. Ils sont surtout utiles dans des boucles, pour afficher différentes variables.

Exercice 5.1 - Watch :

- Remplacer `Exercice4()` par `Exercice5_1()`.
- Lancer le programme.
- Ajoute un point d'arrêt à l'instruction `string nom = p.Nom;`
- Continuer l'exécution
- Quand on arrive sur le point d'arrêt, ajouter un espion sur la variable "nom" (Sélectionner la variable ⇒ Clic Droit ⇒ Ajouter un espion)
- Continuer l'exécution
- ⇒ On voit la variable "nom" évoluer

Exercice 5.2 - intérêt de la méthode `ToString()` pour le debug :

- Remplacer `Exercice5_1()` par `Exercice5_2()`.
- Lancer le programme
- Lors de l'arrêt, regarder la variable pays. (Soit avec un espion, soit simplement en survolant la variable avec la souris).



- L'affichage n'est pas très pratique :
- Rajouter une méthode `ToString()` dans la classe Pays qui renvoie le nom et l'id. (ne pas oublier le mot clef `override`)
- Relancer le programme.
- Regarder une nouvelle fois la variable pays.

Exercice 5.3 - BreakPoint conditionnel

- Remplacer `Exercice5_2()` par `Exercice5_3()`.
- 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 (clic droit sur le point d'arrêt → Conditions)

Réponse : 144 !

Exercice 5.4 - BreakPoint conditionnel

- Remplacer `Exercice5_3()` par `Exercice5_4()`.
- Lancer l'exécution
- Sans modifier le code, ni utiliser l'avancement pas à pas, retrouver directement le nom du 2ème pays de la liste triée.

Indice : utiliser un BreakPoint par nombre d'accès (Conditions → nombre d'accès) sur l'instruction après "`string nom = p.Nom;`" (le `}`)

Réponse : C'est la France

Partie 2 : Création de dll

Afin de fiabiliser des saisies vous allez créer une bibliothèque de méthodes statiques en utilisant la méthode statique TryParse :

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

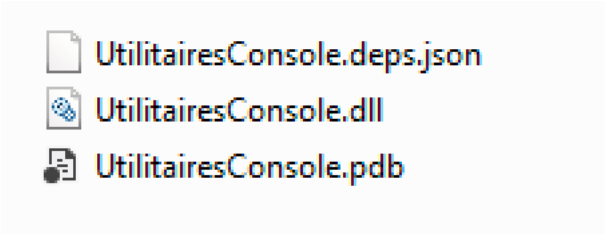
out : passage par reférence

Créez une bibliothèque de classes .NET Core 3.1 nommée `UtilitairesConsole` :

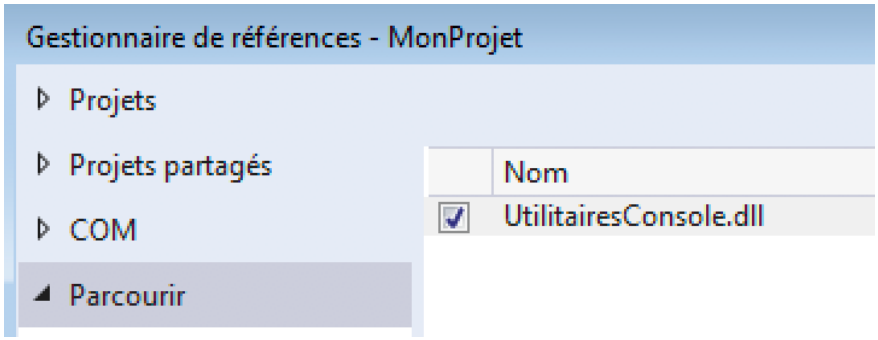
Créez la méthode suivante :

```
public static int SaisieInt() {
    int res;
    bool reussi = int.TryParse(Console.ReadLine(), out res);
    while ( ! reussi ) {
        Console.WriteLine("Erreur de saisie. Il faut un entier : ");
        reussi = int.TryParse(Console.ReadLine(), out res); }
    return res; }
```

Elle a pour but de vérifier que l'utilisateur fait bien une saisie d'un entier. Elle évitera ainsi tout problème de conversion et de plantages.
Générer la solution et observez la création de la dll dans le répertoire **bin/debug**



Créez un projet MonProjet pour tester cette méthode. Vous devez faire une référence à cette dll (ajouter référence de projet → parcourir).



Faites un essai avec une saisie d'un entier.

Surchargez la méthode précédemment définie afin d'afficher un message au sein de la méthode

```
public static int SaisieInt(String message)
```

Testez à nouveau dans MonProjet

Enrichissez la classe « Saisie », définissez et testez les méthodes suivantes :

- **SaisieUInt(string message)** : fait la saisie d'un entier non signé (>=0) puis le retourne.
- **SaisieDouble(string message)** : fait la saisie d'un nombre réel puis le retourne. Testez avec des valeurs comme 5.5
- **SaisieDoublePositif(string message)** : fait la saisie d'un nombre réel positif puis le retourne.
- **SaisieDateTime(string message)** : fait la saisie d'une date au format JJ/MM/AAAA et renvoie un objet DateTime. Vous utilisez une expression régulière pour le format de saisie.
- une méthode **SaisieUnCaracParmiDeuxChoix** (`String message, String choix1, String choix2`)