TP C#3 : Boucles et Debugger

1 Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tp-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- Debugger/
    |   |-- Debugger.sln
    |   |-- Debugger/
    |       |-- Tout sauf bin/ et obj/
    |-- Loops/
        |-- Loops.sln
        |-- Loops
            |-- Tout sauf bin/ et obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

2 Introduction

2.1 Objectifs

Comme indiqué dans le titre, l'objectif de ce TP est d'aborder le fonctionnement des boucles et du debugger de monodevelop. Le debugger est un outil qui va vous faciliter la tâche lors de la recherche de bugs dans votre code. À l'issue de ce TP, lancer le debugger doit devenir un réflexe lorsque vous ferez face à un bug dans votre code. Cependant, avant de commencer à se servir du debugger, un cours sur les boucles s'impose.

> *I*f debugging is the process of removing bugs, then programming must be the process of putting them in. – Dijkstra.

3 Cours : Les boucles

Les boucles peuvent être vues comme des techniques permettant de répéter plusieurs fois une certaine portion de code tant qu'une certaine condition est remplie. Un exemple d'instruction représentant une boucle serait *Va au distributeur voir s'il y a du monde. Et tant que t'y es, achète une canette.*

En tant qu'être humain, cette phrase ou parait logique et en la suivant, on achète une canette avant de revenir. Pour un robot, cette phrase a un tout autre sens. En effet, celui-ci va comprendre d'aller au distributeur et tant qu'il se trouve au distributeur, il achètera une canette.

C'est ce type de structure que nous allons étudier dans ce cours, le but étant de vous enseigner les différents types de boucles qui existent et comment elles fonctionnent.

3.1 La boucle while

Il s'agit de la boucle la plus basique. Elle va vous permettre d'exécuter des portions de code tant qu'une condition est vérifiée. Elle peut être apparentée à l'exemple que nous avons pris au-dessus.

```
int compteur = 0;

while (compteur < 1337)
{
    Console.WriteLine("Yuki !");
    compteur = compteur + 1;
}
```

À travers ce bout de code, l'ordinateur comprends *Tant que la condition est vraie, répète les instructions entre les deux accolades.* Ici la condition est représentée par `compteur < 1337`. Par conséquent, l'ordinateur va afficher le texte `Yuki !` et incrémenter `compteur` tant que celui-ci est inférieur à 1337. Si avant la boucle, `compteur` a une valeur supérieure ou égale à 1337, dans ce cas on ne rentrera pas dans la boucle.

Une autre variante de la boucle `while` est la boucle `do...while`. La seule différence avec la précédente est que le programme entrera au moins une fois dans la boucle avant de tester la condition. Par conséquent, dans le code ci-dessous, le texte sera affiché au moins une fois. Notez le point virgule qui vient se greffer à la condition.

```
int compteur = 0;

do
{
    Console.WriteLine("Never gonna give you up");
    Console.WriteLine("Never gonna let you down");
    Console.WriteLine("Never gonna run around and desert you");
    compteur = compteur + 1;
} while (compteur < 42);
```

3.2 La boucle for

En théorie, les boucles `while` permettent de réaliser toutes les boucles que l'on souhaite. Cependant, il peut s'avérer utile d'avoir un type de boucle condensé permettant de répéter un certain nombre de fois une portion de code. Pour cela, on utilise les boucles `for`.

```
for (int compteur = 0; compteur < 24; compteur++)
{
    Console.WriteLine("I want to be the very best");
}
```

Grâce à ce type de boucle, on regroupe, la déclaration, la condition et l'incrémentation en une seule ligne de code. Lors de l'exécution, le texte s'affichera 24 fois dans la console. Il existe également la boucle `foreach` qui est une variante de la boucle `for`. Les plus curieux pourront se renseigner sur son fonctionnement sur internet...

3.3 Break, Continue

Afin de contrôler le fonctionnement des boucles, il existe certains mots-clés à retenir. Le `break` est le mot-clé qui indique au programme de sortir de la boucle dans laquelle il se trouve.

```
for (int i = 0; i < 5; i++)
{
    if (i == 3)
         break;

    Console.WriteLine("i is " + i);
}
```

Dans ce bout de code, le terminal n'affichera pas le 5 car le `break` aura indiqué de sortir de la boucle lorsque la variable `i` aura atteint 3.

```
i is 0
i is 1
i is 2
```

Le mot-clé `continue` permet de sauter un tour de boucle et de passer au suivant.

```
for (int i = 0; i < 5; i++)
{
    if (i == 4)
        continue;

    Console.WriteLine("i is " + i);
}
```

Le terminal affichera ce résultat car il a sauté un tour de boucle lorsque `i` a eu pour valeur 4.

```
i is 0
i is 1
i is 2
i is 3
i is 5
```

4 Cours : Le debugger

> *E*veryone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it ? – Brian Kernighan

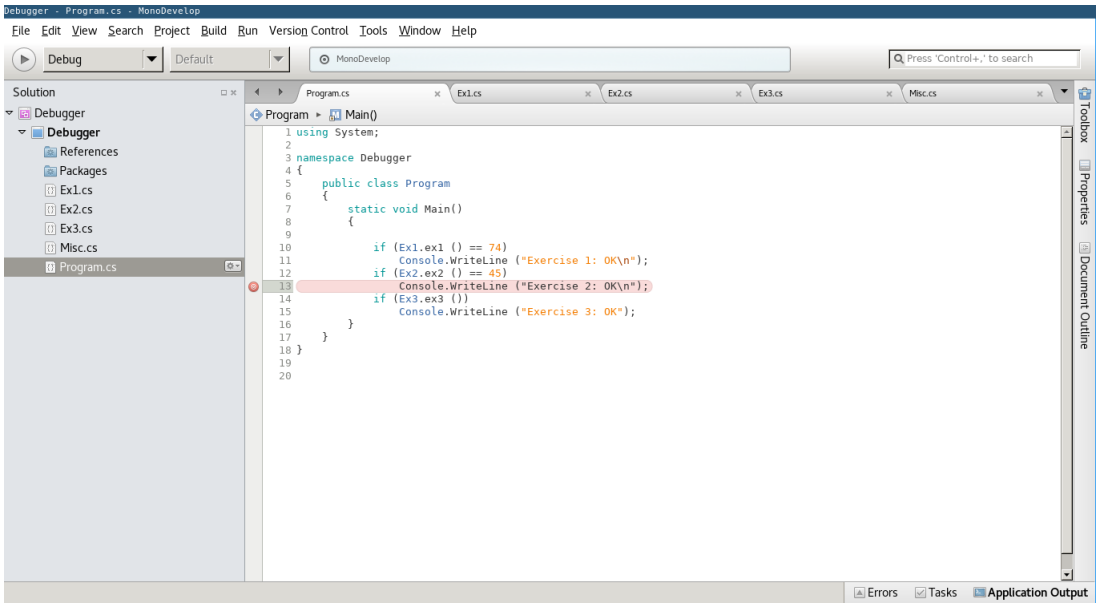
À quoi sert l'outil dont nous parlons depuis tout à l'heure ? Si son nom n'est pas déjà suffisamment explicite il permet de « retirer les bugs » de votre programme. Imaginez que vous ayez maintenant accès à toute la machinerie depuis l'intérieur, il vous est alors possible de suivre l'exécution de votre programme *à votre rythme*. Voici une liste non exhaustive de ce que vous saurez faire à la fin de cette séance :

— Mettre en pause votre programme pendant son exécution.
— Mettre en évidence le contenu de toutes vos variables à un moment donné.
— Modifier le contenu de ces variables pendant l'exécution.
— Modifier le comportement de votre programme pendant son exécution.
— Mettre en évidence la série de fonctions appelées pour arriver où vous vous trouvez actuellement dans le programme.
— ...

Cette liste est loin d'être exhaustive mais prétendre à présenter efficacement l'ensemble des fonctionnalités d'un outil aussi complexe en une seule séance n'est pas raisonnable. Notez cependant que cette liste couvre 95% — pour ne pas dire la totalité — des cas d'utilisations que vous en ferez cette année.

4.1 Les Breakpoints

Imaginez votre programme comme un livre que votre processeur lirait de façon non linéaire comme vous le feriez avec un dictionnaire, le *breakpoint* ou « point d'arrêt » (vous ne voulez pas utiliser ce terme) serait alors un marque page dans ce livre indiquant au processeur de s'arrêter dès qu'il tombe dessus. Vous pouvez placer un breakpoint sur n'importe quelle ligne de votre programme, ainsi une fois que cette ligne sera executée, il s'arrêtera et vous pourrez exécuter la suite à votre rythme. Par exemple, lorsque vous placez un *breakpoint* sur une déclaration de fonction pour mettre en pause l'exécution du programme à chaque fois qu'un appel vers cette fonction est fait, sur le keyword *while* pour vous arrêter à chaque tour d'une boucle... Vous avez compris le principe. Pour placer un *breakpoint* cliquez dans la marge gauche en face de la ligne souhaitée. Il s'affichera un rond rouge indiquant que le *breakpoint* est en place et la ligne concernée sera surlignée en rouge.



Pour supprimer un *breakpoint* il suffit de faire un clique gauche sur le rond rouge qui le représente. Pour le désactiver sans le supprimer lors de vos différents tests « cliquez droit » dessus et sélectionnez l'action correspondante (Enable/Disable Breakpoint), le pictogramme devrait passer à un rond cette fois gris. Après avoir placé correctement vos *breakpoints* vous lancez l'exécution du programme comme d'habitude avec la flèche « play », préférez utiliser le raccourci clavier en pressant simplement la touche *<F5>*.

4.2 Seek'n Destroy

Arrêter le programme oui, mais pour quoi faire ? Le but de placer un *breakpoint* est de pouvoir prendre le temps d'analyser le contexte du programme [1] et de faire ce qu'on appelle de l'exécution pas à pas. Quand vous exécutez un programme hors du debugger cela se fait en une fraction de seconde selon la lourdeur des calculs [2] et la puissance de votre machine [3]. Grâce à l'exécution pas à pas, peu importe la puissance de votre machine c'est vous qui décidez quand passer à la prochaine instruction à exécuter. Sur le listing de la dernière image, le *breakpoint* est placé sur la ligne 21, dire que l'on avance une fois revient à exécuter ce qui se trouve sur la ligne en cours *puis* avancer sur la ligne suivante et attendre. Cependant, que se passe-t-il si la ligne en cours n'est pas une simple assignation de valeur comme ici mais un appel vers une fonction ? Et bien c'est à vous de choisir, en général tout bon debugger propose trois types de saut vers la prochaine instruction :

— Le step over : Exécute la ligne en cours et passe à la suivante sans entrer dans le possible appel à une fonction se trouvant sur la ligne.
— Le step into : C'est l'inverse, cette fois le debugger vous amène au coeur de la fonction appelée.
— Le step out : Continue l'exécution jusqu'à la fin de la fonction et retourne à la méthode appelante.

1. Dans quelle fonction se trouve-t-on, quelles sont les valeurs contenues dans les variables accessibles à ce moment ?
2. Complexité algorithmique
3. La vitesse de l'horloge, la quantité de RAM etc

C'est à vous de savoir si il est préférable de sauter au dessus d'un bout de code ou pas suivant ce que vous cherchez ou avez déjà vérifié. Le step over se fera avec la touche *<F10>*, alors que le step into avec la touche *<F11>* et le step out avec *Shift+<F11>*.

4.3 Smile you're on CCTV

Maintenant que l'on sait avancer dans notre programme à la vitesse que l'on veut il serait bon de pouvoir analyser notre environnement et déceler les éventuels problèmes ; Il existe plusieurs méthodes, toutes ont des avantages et inconvénients et doivent être utilisées au bon moment. Avant toute chose sachez qu'il est difficile d'accéder à des variables qui ne sont pas dans le scope courant. Aussi ne cherchez pas à connaître la valeur contenue dans une variable d'une fonction précédemment appelée et terminée, cela n'a pas de sens.

— La méthode naïve : Survolez la variable avec votre souris, attendez un cours instant et observez une fenêtre vous indiquant la valeur contenue dans la variable. Recommencez ceci autant de fois que nécessaire... Ou utilisez la méthode *post-it*.
— La méthode *post-it* : Quand vous survolez la pop-up cliquez sur la punaise se trouvant sur la partie droite de la fenêtre pour la rendre persistante une fois que votre souris ne la survolera plus.
— Les *watch* Ce sont des variables que vous pouvez créer à la volée pendant l'exécution de votre programme. Vous pouvez les assigner à une variable ce qui est un peu réducteur (revient à faire du post-it) ou alors vous pouvez créer des variables plus complexes ou qui ne se trouvent pas directement telles quelles dans votre code. Un exemple sans intérêt mais néanmoins didactique serait de surveiller la somme de deux variables « a » et « b » disponibles dans le corps de la fonction, ce qui évite d'avoir à faire le calcul de tête à chaque fois.

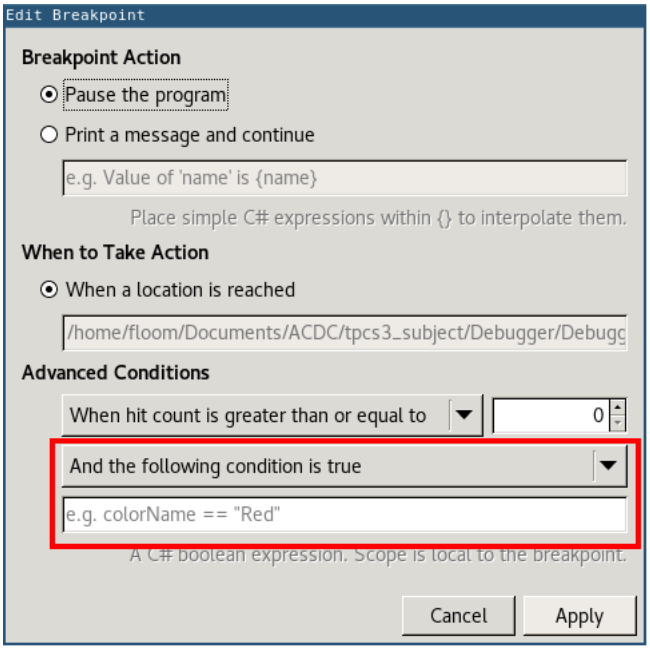
4.4 Maybe, maybe not

Vous vous êtes peut être déjà demandé comment procéder si nous mettions un *breakpoint* dans une boucle et que le programme devenait instable à partir de la 1337ème itération ? Il existe deux méthode, la première est d'avoir beaucoup de patience et de solides doigts pour marteler 1336 fois la touche *<F5>*. La seconde méthode plus intéressante permet de mettre une condition sur un *breakpoint*. Pour cela « cliquez droit » sur le *breakpoint* puis sélectionnez l'item « Breakpoint Properties ».

Comme vous pouvez le voir que cette fenêtre contient beaucoup d'options utiles. Lorsqu'une condition est écrite dans le champ tout en bas de la fenêtre, le programme ne s'arrêtera dessus que lorsque la condition sera vraie (il est possible de changer ce comportement grace au menu déroulant juste au dessus).

Dans l'exemple suivant la condition de *breakpoint* pour atteindre la 1336 ème itération, c'est à dire celle qui précède l'itération de la boucle contenant le bug serait « i == 1336 ».

```
int i = 0;
while (i < 4242)
{
  if (i > 1336)
    // BUG HERE
  i++;
}
```



4.5 You're the only master onboard

Via la fenêtre nommée « locals » vous avez un aperçu en temps réel de l'ensemble de votre contexte. Et rien ne vous empêche de le modifier quand bon vous semble en éditant la ligne correspondante à la variable à modifier.

Cela peut s'avérer très utile pour modifier le comportement de votre programme au *runtime*[4].

4. Pendant son exécution

5 Exercices

Dans les exercices qui vont suivre, vous devrez utiliser des boucles et des fonctions itératives afin de parvenir à vos fins. Nous vous laissons le choix du type de boucle à utiliser. Les paliers 0 et 1 sont à coder dans une solution Loops tandis que le palier 2 est à coder dans la solution Debug fournie. Bien entendu, vous **ne pouvez pas** utiliser de fonctions de `System.Math`

5.1 Palier 0 : To be for or to be while

5.1.1 Exercice 0.1 There will be cake

Commençons avec des choses simples. Le but de cet exercice est d'afficher n fois le texte *'The Cake is a Lie'* sur la console [5].

```
static void GlaDOS (int n);
```

5.1.2 Exercice 0.2 Multiplication

Vous allez recoder la fonction multiplication, mais histoire de rendre ça intéressant, vous avez **interdiction d'utiliser *** dans vos opérations sur cette fonction.

```
static long MyMult (long a, long b);
```

5.1.3 Exercice 0.3 Puissance

Vous allez maintenant implémenter la fonction permettant de mettre un nombre à la puissance donnée en paramètre.

```
static long MyPow (long n, long pow);
```

5.1.4 Exercice 0.4 Racine carrée

On va commencer à corser les choses. Vos allez coder la fonction qui renvoie la racine carrée ainsi que la fonction valeur absolue. Cependant, vous demander d'obtenir une valeur précise serait assez compliqué. Par conséquent, nous vous demanderons une approximation à au moins 10×10^{-5} près à l'aide de la méthode de Newton [6].

```
static float MyAbs (float n);
```

```
static float MySqrt (float n);
```

5.2 Palier 1 : Born to be while

5.2.1 Exercice 1.0 Factorielle

Vous allez maintenant coder la fonction qui renvoie la factorielle d'un entier n. Il est à rappeler que la factorielle de l'élément nul est par convention égal à 1.

```
static long MyFactIter (int n);
```

5. Aperture Science vous informe que les participants se verront offrir du gâteau et un soutien psychologique.
6. https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Newton

5.2.2 Exercice 1.1 Fibonacci

Dans cette exercice, vous allez réécrire la suite de Fibonacci. Il s'agit d'une suite d'entiers dont chaque terme est la somme des deux termes qui le précèdent. Bien entendu, le but de ce TP étant de vous former à l'utilisation des boucles, vous devrez implémenter la version itérative de Fibonacci. En effet, la version itérative de Fibonacci est bien plus rapide à l'exécution que la version récursive.

$$F(n) = F(n-1) + F(n-2)$$

A titre d'exemple, ses premiers termes sont 0, 1, 1, 2, 3, 5, 8, et 13.

```
static long MyFiboIter (int n);
```

5.2.3 Exercice 1.2 TARDIS

Dans un des épisodes de *Doctor Who*, le Docteur explique pourquoi le TARDIS est devenu gigantesque [7] en raison de la détérioration de ses amortisseurs dimensionnels. Cependant, les ACDC que nous sommes souhaitons également voir un TARDIS [8] géant dans vos TP.

Par conséquent, vous allez coder une fonction qui va dessiner à l'écran un TARDIS d'une taille donnée en paramètre. Pour mener à bien cet objectif, nous vous fournissons un TARDIS ~~dessiné de nos propres mains~~ honteusement copié sur Internet.

```
static void MyGiantTardis (int n);
```

La fonction prend pour paramètre un entier n qui représente le nombre de fois qu'il faut répéter le motif seamless du TARDIS (les lignes précédées d'un 2 dans l'exemple) à l'aide d'une boucle. La partie supérieure (lignes précédées d'un 3) et inférieures (lignes précédées d'un 1) sont à afficher une unique fois.

```
3           ___
3           | |
3           | |
3   ====================
3   ====================
3    |  ___  |  ___  |
3    | | | | | | | | |
3    | |-+-| | |-+-| |
3    | |_|_| | |_|_| |
2    |  ___  |  ___  |
2    | |   | | |   | |
2    | |   | | |   | |
2    | |___| | |___| |
1    |       |       |
1   ====================
```

7. Une canette offerte par l'ACDC shepard au premier qui saura en expliquer la cause en citant l'épisode
8. Time And Relative Dimension In Space

5.3 Palier 2 : Le debugger

Pour ces exercices une archive contenant le code à débuger vous est fournie. Chaque exercice contient un ou plusieurs bugs. Il vous est demandé d'utiliser le débugger pour trouvez ces bugs, de toute façon il vous sera beaucoup plus long de les trouver sans.

Une fois les bugs trouvés, corrigez-les et intégrez le projet dans votre archive de rendu. Afin de vérifier que vous ayez bien utilisé le débugger vous devez expliquer dans le README quel(s) étai(t/ent) le(s) bug(s), comment vous (le/les) avez trouvé(s), et comment vous (le/les)avez corrigé(s). Une fois le(s) bug(s) d'un exercice corrigé(s un message confirmant que l'exercice à bien été résolu s'affichera dans la console.

/ !\ Si ce n'est pas fait les exercices seront considérés comme non fait / !\

5.4 Exercice 2.1 : While does it bug ? !

Pour cet exercice veuillez corriger le/les bugs de la fonction « ex1 »située dans le fichier « Ex1.cs ». Une fois l'exercice résolu, « Exercice 1 : OK »s'affichera dans la console. C'est à vous de trouver ce que fait la fonction de cet exercice.

5.5 Exercice 2.2 : Un bug FORmidable

Pour cet exercice veuillez corriger le/les bugs de la fonction « ex2 »située dans le fichier « Ex2.cs ». Une fois l'exercice résolu, « Exercice 2 : OK »s'affichera dans la console. C'est à vous de trouver ce que fait la fonction de cet exercice.

5.6 Exercice 2.3 : Un tri pas très efficace

Pour cette exercice veuillez corriger le/les bugs des fonctions « ex3 », « ex3_2 »et « ex3_3 »situées dans le fichier « Ex3.cs ». Attention, toutes les fonctions ne contiennent pas nécessairement un bug. Les comportements des fonctions ex3 et ex3_3 sont précisés en commentaire. C'est à vous de trouver quel algorithme de tri est utilisé. Une fois l'exercice résolu, « Exercice 3 : OK »s'affichera dans la console.

5.7 Bonus : Cracking

Pour cet exercice un binaire et le code source correspondant vous sont donné. Lorsque vous lancez le programme, un mot de passe vous est demandé. Afin d'obtenir les poins bonus vous devez trouver ce mot de passe et l'écrire dans votre fichier `README`. Comme pour les exercices précédents, vous devez expliquer votre démarche dans le `README` afin que l'exercice soit validé. Un conseil, le debugger devrait s'avérer très utile ici.

The code is the Law