TP C#2 : Debugging for a while

1 Consignes de rendu

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

```
rendu-tp-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- Debugger/
        |-- Debugger.sln
        |-- Debugger/
            |-- Everything except bin/ and 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 !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où '$' est un retour à la ligne et ' ' un espace) :

```
* firstname.lastname$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* firstname.lastname" > AUTHORS
```

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les bonus que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

2 Introduction

2.1 Objectifs

Ce TP a pour objectif de vous apprendre l'utilisation des boucles ainsi que du débugger. Les boucles sont un moyen de répéter une portion de code, que ce soit un nombre donné de fois ou bien tant qu'une condition est satisfaite. Le débugger est quant à lui un outil indispensable dans la recherche et l'élimination de bugs dans votre code. Mais tout bon outil est inutile tant que l'on ne sait pas s'en servir.

3 Cours

3.1 Les boucles

L'affichage en C# du numéro de la ligne courante, sans boucles, ressemblerait à ceci :

```
1 using System;
2 Console.WriteLine("ceci est la ligne 1");
3 Console.WriteLine("ceci est la ligne 2");
4 Console.WriteLine("ceci est la ligne 3");
5 Console.WriteLine("ceci est la ligne 4");
6 Console.WriteLine("ceci est la ligne 5");
7 [...]
```

Au delà de l'évidente perte de temps causée par l'écriture de ce code, si l'on décide de numéroter ces lignes de 0 à n-1 la correction doit se faire sur chaque ligne. De plus, la portion de code dupliquée dans cet exemple ne fait qu'une seule ligne, mais pourrait en faire beaucoup plus.

Les boucles permettent de remédier à deux problèmes :

- Répéter une ou plusieurs actions tant qu'une condition est vraie.
  Ceci est la fonction de **while**.

- Répéter une ou plusieurs actions un certain nombre de fois. Une variable est généralement utilisée comme compteur. Ce type de répétition est appelé itération en informatique ;
  Ceci est la fonction de **for**.

3.1.1 La boucle while

C'est la boucle la plus simple. Elle permet d'exécuter des lignes de code tant qu'une condition est vraie. La syntaxe ressemble à ça.

```
1 while (condition)
2 {
3     // do things
4 }
```

Ici, l'ordinateur comprend : *tant que la 'condition' est vraie, répète les instructions entre les accolades*. Évidemment, si la 'condition' est fausse dès le début, on ne rentrera jamais dans la boucle.

Ainsi, un exemple basique de cette boucle avec une compteur s'écrit comme suit :

```
int counter = 0;
while (counter < 42)
{
    Console.WriteLine("If at first you don't succeed, redefine success!");
    counter = counter + 1;
}
```

La 'condition' est donc **counter < 42**. Vu que **counter** commence à 0, il va rentrer dans la boucle, afficher le texte, et incrémenter celle-ci. Ainsi de suite, 42 fois.

Petite parenthèse sur le counter = counter + 1 de l'exemple. On utilise des variables comme cela pour compter les tours de boucle qu'on a fait. Si on ajoute 1 chaque tour, on dit qu'on incrémente la variable et si on retire 1, on la décrémente. Comme vous pouvez vous en douter, vous allez en créer souvent et écrire counter = counter + 1 à chaque fois n'est ni efficace ni beau à voir. Il existe deux variantes. Pour incrémenter des variables, on peut écrire **variable += 1** ou **variable++**. Pour décrémenter, c'est **variable -= 1** ou **variable−−**.

Il existe une alternative à **while**, la boucle **do...while**, dans laquelle la vérification de la condition s'effectue à la fin de la boucle, *après chaque itération.* Cela implique que cette boucle est TOUJOURS parcourue au moins une fois, même si la condition est basiquement fausse. Attention, il ne faut pas oublier le point-virgule à la fin de la boucle.

```
var i = 0;
do {
    Console.WriteLine("Palms are sweaty");
    Console.WriteLine("Knees weak, arms are heavy");
    Console.WriteLine("Vomit on my sweater already");
    Console.WriteLine("Mom's spaghetti");
    i++;
} while (i != 42); // N'oubliez pas de fermer la boucle
```

3.1.2 La boucle for

En théorie, les **while** permettent de répondre à tous vos besoins en ce qui concerne les boucles. Néanmoins, il existe un autre type de boucle : les **for**. C'est une version condensée du while pour répéter des instructions un nombre voulu de fois. La syntaxe basique est ci-dessous :

```
for (int counter = 0; counter < 42; counter++)
{
    Console.WriteLine("I am nobody");
    Console.WriteLine("Nobody is perfect");
    Console.WriteLine("Therefore I am perfect");
}
```

En une seule ligne, on déclare notre variable, on lui impose une condition et on l'incrémente. C'est très pratique pour du code concis et clair. Mais il faut voir que ce n'est que du sucre syntaxique. La première partie de la boucle (avant le premier ';') s'exécute avant la boucle. La partie du milieu est la condition comme dans une boucle while et la dernière partie s'exécute à la fin d'un tour de boucle. Mais attention, aucune variable déclarée dans la ligne du for n'est accessible en dehors des accolades (regardez le scope si cela vous dit rien).

3.1.3 La boucle foreach

La boucle **foreach** n'est pas dans tous les langages de programmation. C'est une variante du **for** qui permet de ne même plus écrire de condition. Attention, cette boucle ne s'applique que à des tableaux et ou des constructions similaires que vous allez voir la semaine prochaine. La syntaxe basique est :

```
1  int[] array = {0, 1, 2, 3, 4, 5, 42, 1337};
2  foreach(int element in array)
3  {
4      Console.WriteLine("Element is:" element);
5  }
```

Ce code va simplement imprimer tous les éléments contenus dans **array**. Si vous voulez en savoir plus, vous pouvez facilement aller sur internet pour satisfaire votre curiosité.

3.1.4 Break, Continue

En plus du fonctionnement des boucles, il existe des mots-clés qui permettent de modifier le comportement normal d'une boucle. Par exemple, on peut ignorer un tour de boucle, ou bien quitter la boucle courante, à n'importe quel tour de la boucle.
Le premier mot-clé est **break**. Il permet de sortir de la boucle dans lequel **break** est appelé.

```
1  for (int i = 50; i < 1337; i++)
2  {
3      if (i == 53)
4          break;
5
6      Console.WriteLine("I only took " + i + " coffees today");
7  }
8  // output
9  // I only took 50 coffees today.
10 // I only took 51 coffees today.
11 // I only took 52 coffees today.
```

Ici, le programme va s'arrêter au tour de boucle où i est égal à 53 vu que break est appelé.
Le mode-clé **continue** permet de sauter un tour de boucle.

```
1  for (int i = 50; i < 1337; i++)
2  {
3      if (i == 52)
4          continue;
5
6      Console.WriteLine("I only took " + i + " coffees today");
7  }
8  // output
9  // I only took 50 coffees today.
10 // I only took 51 coffees today.
11 // I only took 53 coffees today.
12 // ...etc...
```

3.2 Le Debugger

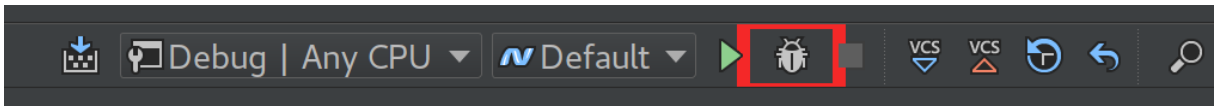
> If debugging is the process of removing bugs, then programming must be the process of putting them in - Edsger Dijkstra

Vous l'aurez compris, 'debugging' est le processus qui permet de retirer des bugs de vos programmes. Il est presque impossible de créer du code sans bugs, même en étant le génie des hôtes de ses bois. Savoir débugger correctement est donc primordial, et il faut absolument connaître et utiliser les bons outils pour pouvoir le faire plus simplement et surtout plus vite.
La phase de debugging d'un programme consiste à pouvoir voir tous ce qu'il se passe avec toutes les choses que vous utilisez dans le contexte actuel. Vous pouvez donc suivre votre programme à votre rythme pour voir où se trouve le mauvais code. Vous aller pouvoir :

- Mettre en pause votre programme.
- Voir les valeurs de vos variables.
- Modifier lesdites variables si besoin.
- Modifier le programme si besoin.
- Connaître la liste des fonctions appelées pour savoir où on en est.
- etc...

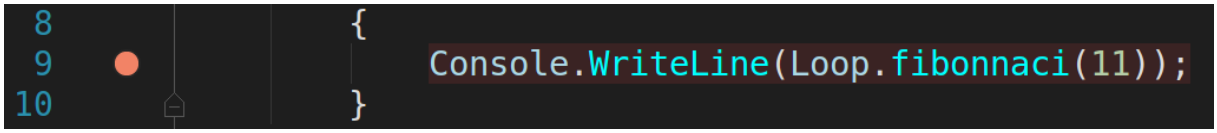
Vous apprendre à débugger parfaitement en un TP est impossible (pour vous surtout, nous ACDC aurions bien sûr été content de le faire). Sachez juste que l'on couvre un grand nombre des cas d'utilisations dont vous aurez besoin durant votre année.

Attention : dans Rider, il ne suffit pas d'exécuter votre code pour qu'il détecte que vous voulez débugger votre code. Il faut lancer ce qu'on appelle le mode debug et c'est simplement le petit insecte à coté du bouton run en haut à gauche de l'écran.



3.2.1 Les breakpoints

Les breakpoints sont tous simplement des lignes de code où lorsque le débugger passe dessus, il suspend l'exécution du programme. Ils peuvent être placés n'importe ou dans votre programme. Par exemple, lorsqu'on place un breakpoint sur une boucle, l'exécution va s'arrêter à chaque boucle. Si on le place sur une fonction, l'exécution va s'arrêter à chaque appel de cette fonction. Pour placer un breakpoint, il suffit de de cliquer dans la marge au niveau de la ligne. Un point rouge va apparaître à cet endroit et la ligne sera surligné légèrement en rouge.



Pour supprimer un breakpoint, il suffit de cliquer sur le point rouge une seconde fois. Pour le désactiver sans le supprimer, il faut clique droit sur le point rouge et désactiver l'option *Enabled*.
On vous conseille aussi de lancer le mode debug avec le raccourci clavier **shift+<F9>**.

3.2.2 Les steps

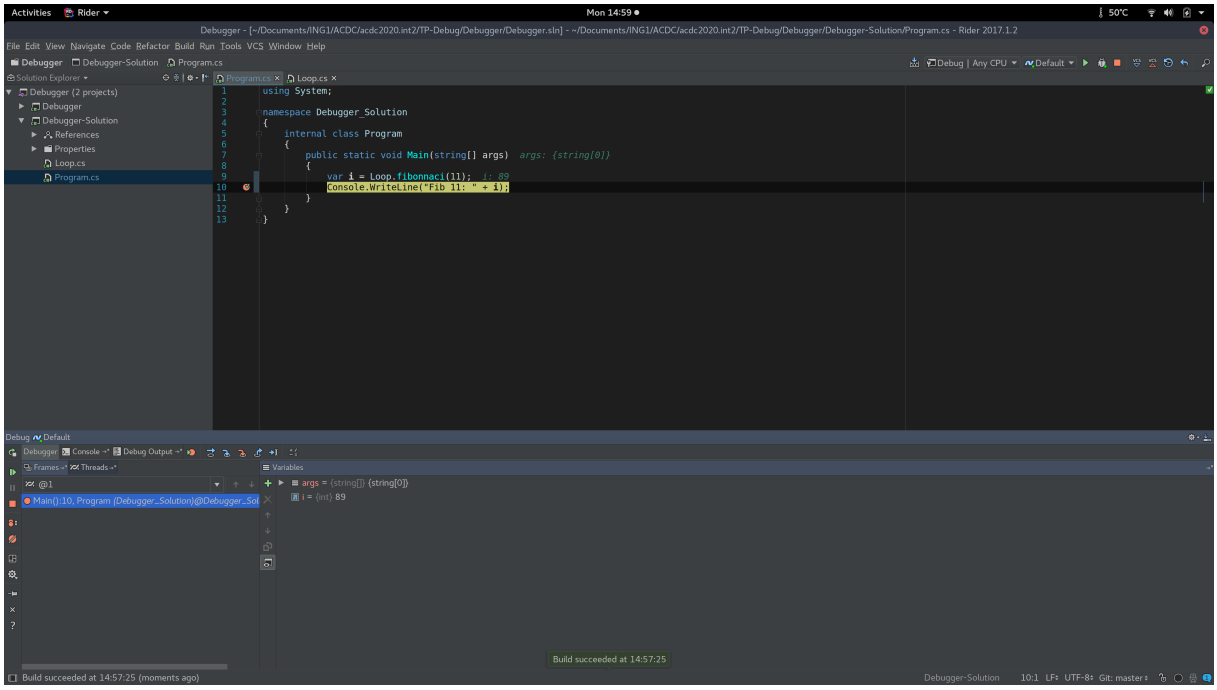
Arrêter le programme mais pourquoi ? Une fois le programme arrêté, c'est à vous de jouer. Il existe plein de manières de voir le code et de l'exécuter ligne par ligne après le breakpoint. Par

exemple, le breakpoint est sur la ligne d'une assignation d'une variable par un retour de fonction. On sait que le bug se trouve quelque part après ou sur la ligne actuelle. Pour pouvoir avancer et aller dans la fonction, on fait ce qu'on appelle un *step-in*. Pour avancer sans passer par la fonction, c'est un *step-over*. Si on est déjà dans une fonction, et qu'on veut finir son exécution et sortir, on fait un *step-out*. Rien avec ces trois la, on peut se déplacer comme on veut tout en avançant dans le code. Voici une liste des opérations possible dans Rider :

- **Step Over** : Executer la ligne et passer à la suivant sans aller dans la fonction si il y en à une dans la ligne : <F8>
- **Step Into** : Executer la ligne et renter dans la fonction si il y en à une dans la ligne : <F7>
- **Force Step Into** : Executer la ligne et renter dans une fonction dans lequel on aurait pas le droit de step into (il y en a) : alt+shift+<F7>
- **Step Out** : Continuer l'execution jusqu'à la fin de la fonction et retourne à la méthode appelante : shift+<F8>

3.2.3 Les Illuminati surveillent

Vous savez parcourir votre code, maintenant il faut peut être savoir comment connaître la valeur des variables durant le debug. En fait, c'est déjà tout fait pour voir. Rider va automatiquement rajouter en commentaire la valeur des variables à la fin des lignes. Si vous voulez plus de détails sur une variable ou un objet, il suffit de cliquer sur la variable dans la liste qui s'affiche en bas en mode debug.
Vous pouvez aussi watch une variable si vous voulez.



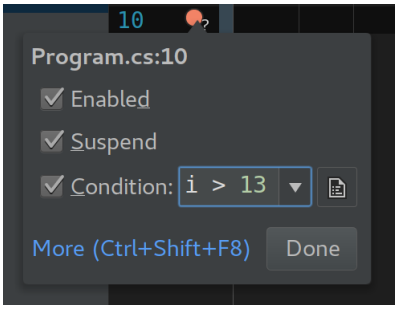
3.2.4 Breakpoints conditionnels

C'est bien sympa de pouvoir faire tout ça mais si mon bug n'est qu'après la 1 000 000ième itération, vais-je devoir cliquer un million de fois sur le continue ? On vous laisse essayer, mais il parait qu'il y a eu des morts.

Il existe un moyen de faire cela facilement sans se faire une crampe au doigt voir au cerveau en même temps. Cela s'appelle les breakpoints conditionnels. Comme son nom l'indique, cela permet de mettre des conditions sur l'activation d'un breakpoint.

```
1  for (var i = 0; i < 42 000 000; i++)
2  {
3      ...
4      //bug here if i > 13 370 000
5      ...
6  }
```

Si on prends le code du dessous, on peut soit écrire un if avec condition i > 13 370 000 et mettre un breakpoint dans le if soit mettre un breakpoint conditionnel avec pour condition i > 13 370 000.



3.3 That's all folks

Et voila, à vous de jouer maintenant.

4 Exercices

4.1 Boucles

4.1.1 Tu es un naturel

Commençons par les choses simples et bien faites. Il faut afficher tous les entiers naturels entre 1 et n, avec des espaces entre les nombres mais pas d'espace après le dernier.

```
1  public static void Print_Naturals(int n);
```

4.1.2 Optimus Prime

Il faut imprimer dans la console la liste des entiers premiers entre 1 et n. Encore une fois, avec des espaces entre les nombres mais pas après le dernier.

```
1  public static void Print_Primes(int n);
```

4.1.3 Fibonacci toujours

Ici, il faut recoder la suite de Fibonacci. C'est une suite dont chaque terme est la somme des deux termes précédents. Bien sur il faut absolument implémenter la version itérative vu que celle ci est bien plus rapide que la version récursive que vous connaissez. On testera donc sur des nombres plutôt grands. Petit rappel : les premiers termes sont 0, 1, 1, 2, 3, 5, 8, et 13.

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

```
1  public static long Fibonacci(long n);
```

4.1.4 Factorielle encore

Il faut recoder la fonction factorielle en version itératif. On rappelle que 0 ! est égal a 1.

$$n! = \prod_{1 \leqslant i \leqslant n} i = 1 * 2 * 3 * ... * (n-1) * n$$

```
1  public static long Factorial(long n);
```

4.1.5 I'm Stronk

Il faut coder une fonction qui imprime tous les entiers dit 'strong' entre 1 et n. Ce sont des entiers pour lesquels la somme des factorielles de leurs chiffres est égale a eux-même. Un exemple serait 145 car $1! + 4! + 5! = 145$.

```
1  public static void Print_Strong(int n);
```

4.1.6 Racine Carré

Il faut d'abord recoder la fonction qui donne la valeur absolue d'un float donné en argument.

```
1  public static float Abs(float n);
```

En utilisant la fonction précédente, il faut coder une fonction qui donne la racine carré d'un nombre passé en argument. Il faudra utiliser la méthode de Newton [1].

```
1  public static float Sqrt(float n);
```

1. https ://math.mit.edu/ stevenj/18.335/newton-sqrt.pdf

4.1.7 Powerrrrrrr ! ! ! !

Il faut recoder la fonction puissance en itératif, ou a est à la puissance de b.

```
1  public static long Power(long a, long b);
```

4.1.8 Vic-tree is ours

Il faut coder une fonction qui imprime un arbre dans la console. Le n correspond à la hauteur de l'arbre. Si n > 3, il faut 2 lignes de tronc, sinon 1 seule.

```
1  public static void Print_Tree(int n);
```

```
1  // n = 3
2     *
3    ***
4   *****
5     *
6
7  // n = 5
8        *
9       ***
10     *****
11    *******
12   *********
13       *
14       *
```

4.2 Debugging

Pour ces exercices, une archive contenant le code à débugger vous est fournie. Chaque exercice contient un ou plusieurs bugs. Il vous est demandé d'utiliser le débugger pour trouver 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 avez 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). ATTENTION : si ce n'est pas fait, les exercices seront considérés comme non faits.

These violent deadlines have violent ends !