TP C#3 : Loops and Debugger

1 Guidelines for handing in

At the end of this tutorial, you have to submit an archive that respects the following architecture :

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

Don't forget to verify the following points before submitting your work :

- Remplace `firstname.lastname` by your own login and don't forget your `AUTHORS` file.
- The file `AUTHORS` and `README` are mandatory.
- No `bin` or `obj` directory in your project.
- Carefully observe the prototypes required.
- Remove all your tests.
- **Your code must compile!**

2 Introduction

2.1 Objectives

As indicated in the title, the objective of this tutorial is to learn how loops and the debugger work on monodevelop. The debugger is a tool that will make your life easier when searching for bugs in your code. After this tutorial, launching the debugger must become a reflex when you will faced with a bug in your code. However, before starting to use the debugger, an introduction to loops structures is required.

If debugging is the process of removing bugs, then programming must be the process of putting them in. – Dijkstra.

3 Course : The loops

Loops can be seen as structures used to repeat a certain portion of code several times while a certain condition is met. An example of instruction representing a loop would be *Go to the cans machine to see if there are people. And while you're at it, buy a can.*

As a human being, this sentence seems logical and by following it, you should buy a can before returning back. For a robot, this phrase has a different meaning. Indeed, it will go to the cans machine and it will buy cans until there's people at the machine.

That's the structure we will study in this course, the goal is to teach you the different types of loops and how to use them.

3.1 The while loop

This is the most basic loop. It will allow you to execute portions of code while a condition is met. It may remind you of the example we have taken above.

```
int counter = 0;

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

Through this piece of code, the computer understand *As long as the condition is true, repeats the instructions between the brackets.* Here the condition is represented by `counter < 1337`. Therefore, the computer will display the text Yuki ! and increment `counter` while the value is less than 1337. If before the loop `counter` has a value greater or equal to 1337, then we will not enter the loop.

Another variant of the `while` loop is the `do...while` loop. The only difference with the previous is that the program will enter at least once in the loop before testing the condition. Therefore, in the following code, the text will be displayed at least once. Note the semicolon on the right of the condition.

```
int counter = 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 (counter < 42);
```

3.2 The for loop

In theory, `while` loops allows you to create any type of loop you need. However, it may be useful to have a condensed loop to repeat a certain number of times a portion of code. For this, we will use the `for` loops.

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

Thanks to this type of loop, we gather the statement, the condition and the increment in a single line of code. During the execution, the text will appear 24 times in the console. There is also the `foreach` loop which is a variant of the `for` loop. You can learn more about it on the internet...

3.3 Break, Continue

In order to keep control over the loops, there are some key words to remember. The `break` is the keyword that tells the program to exit the current loop.

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

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

In this snippet, the terminal does not display the 5 as the `break` will exit the loop when the variable `i` reaches 3.

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

The keyword `continue` skips the current loop step and moveon to the next one.

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

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

The terminal displays this result because it has moved to the next loop when `i` had the value 4.

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

4 Cours : The 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

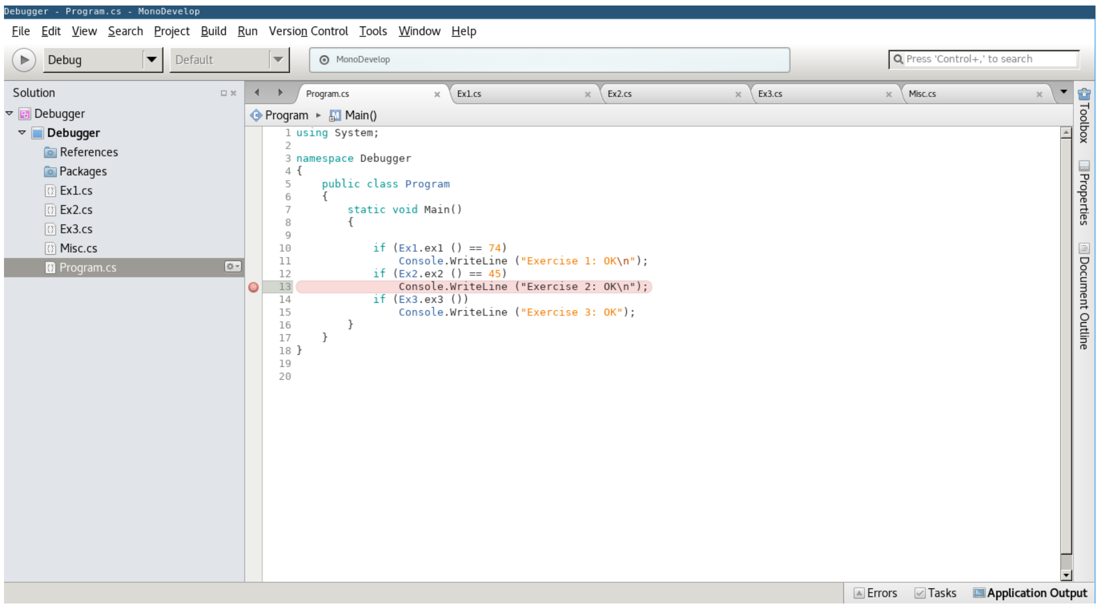
What's the purpose of this tool ? If the name isn't already clear enough, it allows you to fix bugs in your program. Imagine that you have access to all machinery from the inside. It is now possible to follow the program's execution *at your own pace*. Here is a non-exhaustive list of what you'll be able to do at the end of this session :

— Pause your program during runtime.
— Show the content of the variables at a given time.
— Change the content of these variables at runtime.
— Change the behavior of your program at runtime.
— Show a list of the functions called to get to where you are now in the program.
— ...

This list isn't exhaustive at all, but trying to present all those in one session isn't reasonable. Please note, however, that this list covers at least 95% — if not all — of the possible cases that you'll see this year.

4.0.1 Breakpoints

Imagine your program is a book that your processor will read in a non-linear way, as you'd do with a dictionary. A breakpoint is like a bookmark in this book, that tells the processor to pause reading when it reaches it. You can put a breakpoint on a function declaration to pause the program every time the function is called. You can also put a brakpoint on the **while** keyword in a loop to pause the program at each iteration. To put a breakpoint simply click in the left margin, in front of the chosen line. A red circle is now displayed (to let you know that a breakpoint has been placed) and the line should be highligted in red.



```
using System;

namespace Debugger
{
    public class Program
    {
        static void Main()
        {

            if (Ex1.ex1 () == 74)
                Console.WriteLine ("Exercise 1: OK\n");
            if (Ex2.ex2 () == 45)
                Console.WriteLine ("Exercise 2: OK\n");
            if (Ex3.ex3 ())
                Console.WriteLine ("Exercise 3: OK");
        }
    }
}
```

To delete a breakpoint click on it. To deactivate it while keeping it in place, right-click on it and select the corresponding option (Enable/Disable Breakpoint) The circle should now be gray. When your **breakpoint** is set, launch your program as always with the green “play” arrow (you can use the shortcuts *<F5>* or *Shift+<F5>*).

4.0.2 Seek’n Destroy

Alright, you know how to stop the program, but what’s the purpose of all of this ? The goal behind putting a **breakpoint** in your code is to take the time to analyze the program’s context [1] and do what we call *step by step execution*. When you’re not in the debugger all the execution is done in a fraction of a second depending on the complexity of the calculations you’re doing [2] and the speed of the computer [3]. Thanks to the **step-by-step** execution no matter what kind of computer you have, you can decide when to execute the next instruction. If we move forward once (to the next step), the debugger will execute what is on the current line *then* advance to the next one and wait. However, what’s going on if the line isn’t a simple assignment like we have here but a function call ? In fact it’s up to you to choose how the debugger will react next. Any good debugger let you choose three types of jump to the next instruction.

- **step over** : Execute the current line and place the cursor to the next one without displaying the execution of the function that was on the previous line.
- **step into** : It’s the opposite, this time the debugger bring you into the code of the called function.
- **step out** : Continue the execution until the end of the current function and break when returning to the calling function.

It’s up to you to know when it’s better to step over a piece of code or not, depending on what you seek or have already checked. If you want to step over press the key *<F10>*, while the step into is done with the key *<F11>*.

4.0.3 Smile you’re on CCTV

Now that you know how to go through your code at runtime, it would be good to look at what’s hiding beneath our variables and spot potential issues [4]. There are several methods, each with advantages and drawbacks. Please note that it’s difficult to access variables that are not in the current scope of execution. So please don’t try to check values stored inside variables of a previous and terminated function, it does not make sense.

- naive : Place your mouse over a variable and wait a little. There is now a popup that contains the variable’s content. Do it as often times as you need... Or use the post-it method.
- post-it : When you put your mouse over the previous pop-up you can see a pin, click on it to make pop-up persistent.
- watch : You can see them as variables that you can create on the fly at runtime. You can assign them to a variable but this is a limited use of them (i.e this is the post-it method in another window), or you can create more complex variables that are not directly found in the code. A useless but educational example would be creating a watch that display the sum of two other variables “a” and “b” that are available in the body of a function, it avoids mentally computing the sum every time.

1. The function is being executed, the values of the local variables
2. Algorithm complexity
3. Clock speed, amount of RAM...
4. This is a problem : 42 / 0

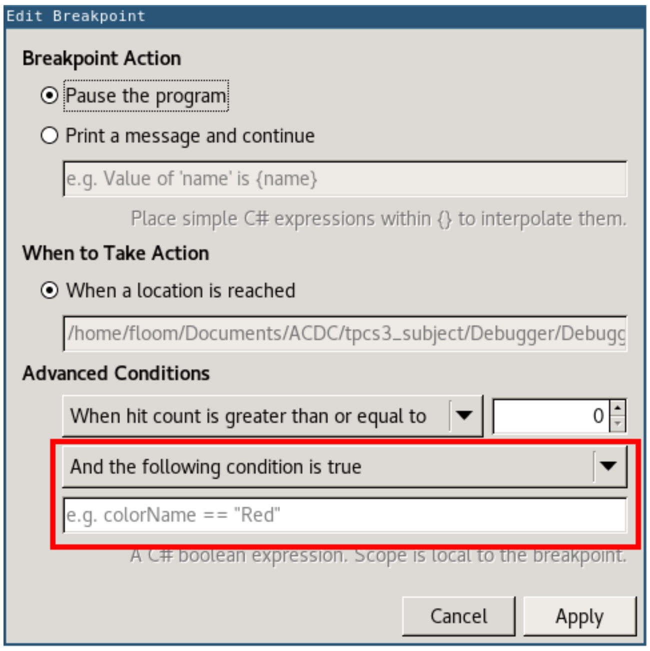
4.1 Maybe, maybe not

Perhaps you're already wondering how to put breakpoints efficiently in big loops. What if your program becomes unstable after the 1337th iteration ? There are two methods, the first one needs a lot of patience and some solid fingers to press 1336 times the <*F5*> key. The second one is far more interesting and allows you to put a conditional breakpoint. Right-click on the desired breakpoint and select « Breakpoint Properties ».

As you can see this window contains a lot of useful options. In the last field at the bottom you can write a condition, in order to only stop the program on the breakpoint when the condition is true (This behavior can be changed with the pulldown menu just above the condition field).

In the following example, if we want to skip the first 1336 iterations, the condition to enter in the input box would be i == 1336.

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


Edit Breakpoint

Breakpoint Action

(●) Pause the program
( ) Print a message and continue

e.g. Value of 'name' is {name}

Place simple C# expressions within {} to interpolate them.

When to Take Action

(●) When a location is reached

/home/floom/Documents/ACDC/tpcs3_subject/Debugger/Debugg

Advanced Conditions

When hit count is greater than or equal to: 0

And the following condition is true

e.g. colorName == "Red"

A C# boolean expression. Scope is local to the breakpoint.

4.2 You're the only master onboard

The window named "locals" displays an overview of all the local variables of the current context in real time. You can modify any variable by changing the correct line in the window locals.

It can be very useful to modify the behavior of your program at runtime.

5 Exercises

In the following exercises, you will use loops and iterative functions to complete the exercises. We let you choose the type of loop to use. The threshold 0 and 1 must be coded in the Loops solution whereas the threshold 2 must be coded in the provided Debug solution. Of course, **you can't** use functions from `System.Math`

5.1 Threshold 0 : To be for or to be while

5.1.1 Exercise 0.1 There will be cake

Let's start with simple goals. The purpose of this exercise is to print n times the text *'The Cake is a Lie'* on the console[5].

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

5.1.2 Exercise 0.2 Multiplication

You will re-code the multiplication function, but just to make this exercise interesting, **you can't** use * in your operations.

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

5.1.3 Exercise 0.3 Pow

You will now implement the function to set a number to the power given in parameter.

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

5.1.4 Exercise 0.4 Square root

We will now begin to complicate things. You're going to code the function that returns the square root and a function that returns the absolute value of a given parameter. However, asking you to get an accurate value would be quite complicated. Therefore, we will ask you for an approximation to at least 10×10^{-5} using Newton's method[6].

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

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

5.2 Threshold 1 : Born to be while

5.2.1 Exercise 1.0 Factorial

You will now code the function that returns the factorial of an integer n. Remember that the factorial of zero is, by convention, equal to 1.

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

5. Aperture Science informs you that cake and psychological support will be available at the end of the test.
6. https://en.wikipedia.org/wiki/Newton%27s_method

5.2.2 Exercise 1.1 Fibonacci

In this exercise, you will rewrite the Fibonacci sequence. It is a sequence of integers in which each term is the sum of the two preceding terms. Of course, as the purpose of this tutorial is to train you to use loops, you need to implement an iterative version of Fibonacci. Furthermore, the iterative version of Fibonacci is much faster than the recursive release.

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

For example, the first terms are 0, 1, 1, 2, 3, 5, 8, and 13.

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

5.2.3 Exercise 1.2 TARDIS

In one episode of *Doctor Who*, the Doctor explains why the TARDIS became gigantic [7] due to the deterioration of its dimension dampeners. However, we also want to see a giant TARDIS [8] in your TP.

Therefore, you will code a function that will draw on the screen a TARDIS of a given size. To fulfill this goal, we provide a TARDIS ~~drawn by ourselves~~ shamefully copied on the Internet.

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

The function takes as parameter an integer n representing the number of times to repeat the TARDIS seamless pattern (lines preceded by a 2 in the example) using a loop. The upper part (lines preceded by a 3) and lower (lines preceded by a 1) are to display a only once.

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

7. A can offered by the ACDC shepard to the first who can explain the cause and give the episode name
8. Time And Relative Dimension In Space

5.3 Threshold 2 : The debugger

For those exercises zip containing the code to debug will be given. Every exercises contain *one or several* bugs. You have to use the debuger to find it/them, anyway il will be way too long without it.

Once you have found the bugs, you have to correct them and should integrate the project in your submission. In order to check that you actually used the debugger you must write in the README a small description of the bugs, how you found them and how you corrected them. Once all the bugs of an exercises a validation message will be displayed in the console.
/ !\ If this step is not done, the exercise will not be considered as done / !\

5.4 Exercise 2.1 : While does it bug ? !

For this exercise you have to correct the bug(s) in the function « ex1 »located in the file « Ex1.cs ». Once the bug(s) is/are corrected « Exercice 1 : OK »will be displayed in the console. You have to find what this function is supposed to do by yourself.

5.5 Exercise 2.2 : Un bug FORmidable

For this exercise you have to correct the bug(s) in the function « ex2 »located in the file « Ex2.cs ». Once the bug(s) is/are corrected « Exercice 2 : OK »will be displayed in the console. You have to find what this function is supposed to do by yourself.

5.6 Exercise 2.3 : A really ineffective sort

For this exercise you have to correct the bug(s) of the functions « ex3 », « ex3_2 »and « ex3_3 »located in the file « Ex3.cs ». Be aware : not all functions have a bug. The behavior of the function ex3 and ex3_3 are given in comment. You have to figure out which sort algorithm is used by yourself. Once the bug(s) is/are corrected « Exercice 3 : OK »will be displayed on the console.

5.7 Bonus : Cracking

For this exercise, you are given a binary with its sources files. When you run it you need to enter a password. You have to find that password and to write it in the `README` file. Like the other exercises, you will not get the points if you don't explain how your method in the `README`. As a protip, the debugger can really be useful on this one.

The code is the Law