TP C#2 :The bug of the Baskervilles

Submission instructions

At the end of the practical, your repository must follow this architecture :

```
tp-csharp2-sherlock.holmes/
|-- Exercices/
    |-- Exercices.sln
    |-- Loop/
        |-- Everything except bin/ and obj/
    |-- Debugger/
        |-- Everything except bin/ and obj/
    |-- DebugMeDaddy/
        |-- Everything except bin/ and obj/
|-- README
```

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

- Replace `firstname.lastname` with your own login.
- The file `README` is mandatory.
- No folder `bin` or `obj` in the project.
- Follow the given prototypes to the letter.
- Remove all the tests from your submission.
- **The code must compile !**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty README file will be considered as an invalid archive (malus).

1 Introduction

1.1 Goals

This practical will teach you how to use loops and the *debugger*. Loops are an easy way to repeat some portion of your code until a condition is fulfilled, such as reaching a given number starting from zero. The *debugger* is an essential tool in helping you to find and eliminate bugs in your code. However, all great tools are worthless if we don't know how to use them.

2 Lecture

2.1 The other loops

Last week you were introduced to the `while` and the `do...while` loops. Consequently, we will consider that you have read and assimilated all notions regarding them.

In this lecture we will focus on the other fundamental loops that `C#` has to offer : `for` and `foreach` loops.

2.1.1 The `for` loop

With the `while` loop, you can achieve all your needs concerning loops. However, other types of loop exist, such as the `for` loop. This is a condensed version of the `while` loop to repeat a number of instructions a desired number of times.

The syntax is the following :

```
1 for (int count = 0; count < 42; count++)
2 {
3   Console.WriteLine("Excellent! I cried. ");
4   Console.WriteLine("'Elementary,' said he.");
5 }
```

In one line we declared our variable, defined the stopping condition and defined how our variable should be incremented. This is very useful for small and concise code. However, you need to understand that this is only **syntaxic sugar** :

— The first part of the loop (before the first ';') is executed before the loop.
— The middle part is the condition such as in a `while` loop.
— The last part is executed at the end of every iteration loop.

Please take care that the variable defined in the first part of the `for` loop is only accessible between the curly brackets. To understand why, please check the **scope** notion.

2.1.2 The `foreach` loop

The `foreach` loop is a variant of the `for` loop to easily go through the elements of a collection (often an array, but we will see other structures in the following weeks).

The syntax is the following :

```
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 }
```

In each iteration of the loop, the variable `element` will take the value of the next element in the collection `array`.

In this example, the code will print every element of the `array`.

2.1.3 `break` and `continue`

There are other keywords to modify the behaviour of a loop. For instance, we can ignore the current loop iteration or leave the loop.

The first keyword is `break`. It allows us to leave a loop in which it is being used :

```
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  }
```

Which outputs :

```
1  I only took 50 coffees today.
2  I only took 51 coffees today.
3  I only took 52 coffees today.
```

Here the program will leave the `for` loop when `i = 53`.

The keyword `continue` allows the user to skip (and thus ignore) a loop iteration.

```
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  }
```

Which outputs :

```
1  I only took 50 coffees today.
2  I only took 51 coffees today.
3  I only took 53 coffees today.
4  I only took 54 coffees today.
5  ...
```

2.2 The Debugger

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

As you have obviously understood, the process of *debugging* is the act of removing bugs from programs. And let's be honest, it is impossible to create bug-less code (even if you are a genius !). Knowing how to debug is a part of knowing how to program. Thus, knowing how to use the right tool is essential to be efficient.

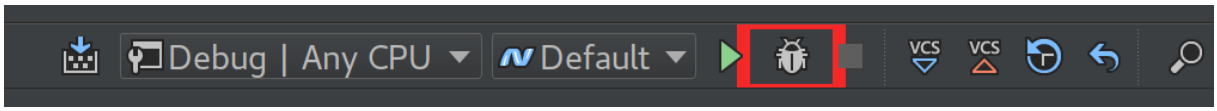
The debugging phase of a program consists of observing carefully everything that happens in your program, and interact correctly depending on the context. Thus, you can check the program execution at your own pace to find the faulty section of code. Here is a non-exhaustive list of debugging operations :

— Pause the program.
— Print variable values.
— Edit the variables if necessary.
— Edit the program if necessary.
— Show the list of called functions to find out where you are.

Different debugging tools exist (some specific to a language, others to an editor) but every tool should have at least the features we will see in this practical. As said before, the debugging notions are essential for your future life as programmers. However, it is impossible to deliver all the know-how in a single practical ; thus, we will ask you for considerable investment in this practical and encourage you to use it in the future practicals.

2.2.1 Rider

Caution : in **Rider**, launching the run button is not enough to launch the debugging tool. We will need to launch what is called the debug mode, which is represented as a bug next to the **Run** button in the upper right corner of the screen.



We also recommend that you start debug mode using the shortcut Alt+F5.

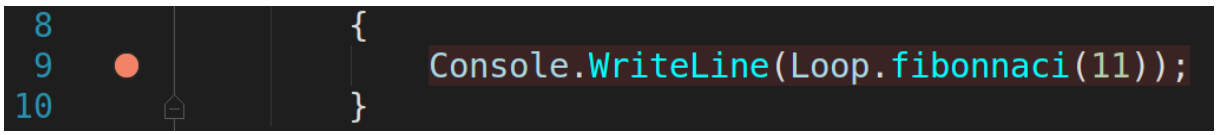
Fun fact

We often assign the origin of the word « bug » to Grace Hopper who in 1945 reported the discovery of a **bug** stuck in the machine causing a short circuit. It was the first bug of modern computer science.

2.2.2 Breakpoints

The **breakpoints** indicate to the debugger where to pause the execution flow of the program. This allows the programmer to check more precisely what is happening in some section of code. You can place as many breakpoints as you want on any line of your choice. For instance, when we set a breakpoint on a loop, the execution flow will stop at every iteration. Identically, a breakpoint on a function will stop the execution flow every time the function is being called.

To set a breakpoint, you simply need to click on the margin of the wanted line. A red dot will show up at the same place and the line will be slightly highlighted in red.



To delete a breakpoint, you simply need to click on the red dot a second time. To disable it without deleting it, you need to right click on the red dot and disable the option *Enabled*.

2.2.3 The steps

We know how to stop the program at some places, but we want to be able to move and keep executing some section of our code. Here are some techniques :

- **Step over** F10 : executes the line and goes directly to the following line.
- **Step into** F11 : executes the line and enters the function if a function call is present on the current line.
- **Step out** ⇧+F11 : continues the execution flow until the end of the current function and returns to the calling method.

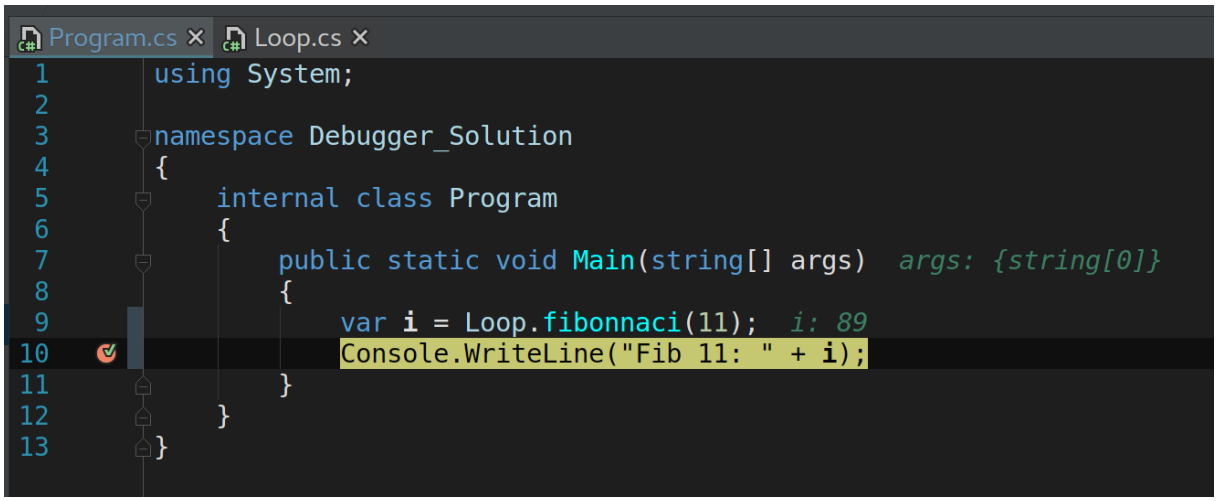
For example, let's suppose we want to place a breakpoint on the first line of this given code where `value` gets assigned.

```
1 int value = calculation();
2 Console.WriteLine(value);
```

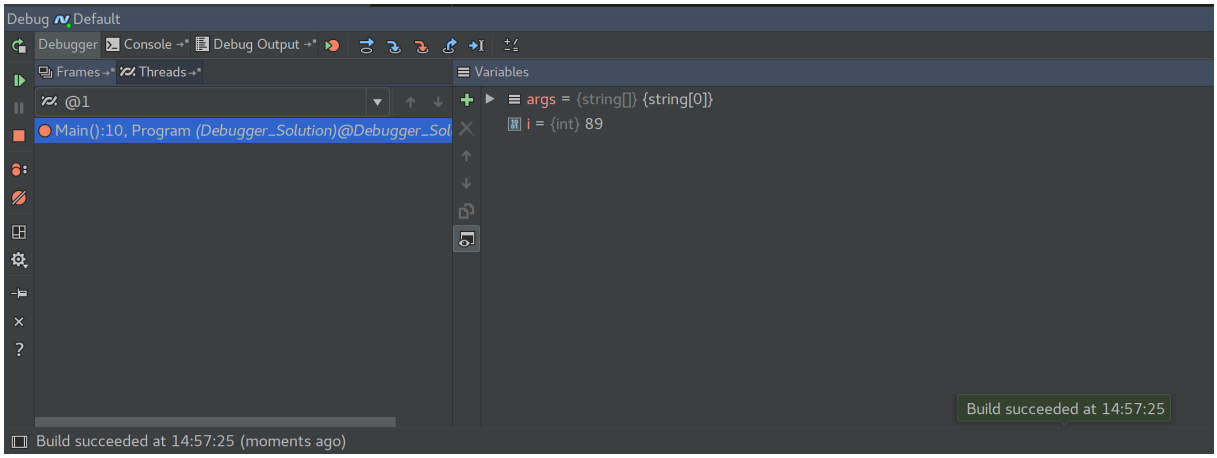
When the debugger arrives on the first line, it sees the breakpoint and stops. If we do a **step over** we will arrive at line 2 without entering in the function `calculation`. If we did a **step into**, then the debugger would have moved us at the first line of the `calculation` function, allowing us to see what will happen in the function. Eventually, a **step out** while we are in the `calculation` function will bring us back to the variable assignation.

2.2.4 The Illuminati's are watching

When we are going through our code, we would like to be able to know the values of our variables. Rider does this automatically by adding a comment next to the declaration of the variable.



If you need more information on a variable (especially an object) you simply need to click on the name of the variable in the list that shows up on the bottom of the screen in debug mode.



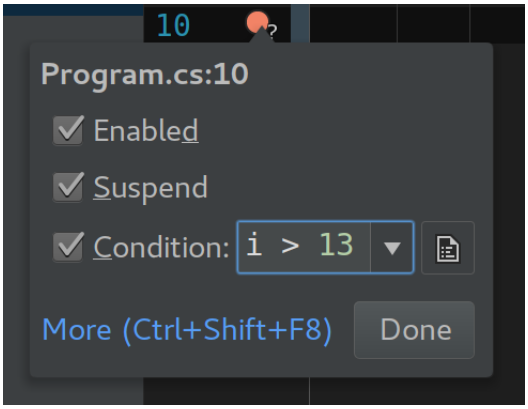
2.2.5 Conditional breakpoint

It's really cool to know all this. However, what if my bug occurs after the millionth iteration. Do I really need to click one million times on the `continue` button ?

Instead, there is a way to do this easily without getting a cramp : conditional breakpoints. As obvious as the name suggests, it allows us to create conditions when a breakpoint should be active or not.

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

If we consider the code seen above, we can either write in the code a condition `if (i > 13370000)` and put a breakpoint in the `if`, or put a conditional breakpoint whose condition is `i > 13370000`.



2.3 That's all folks

Now with all the knowledge you have acquired, it's your turn to impress us.

3 Exercises

3.1 Loops ++

Additional Restrictions

You must use `for` or `foreach`loops while completing the following exercises. The goal of this tp is to familiarize yourself with this new tool, not for you to stay in your comfort zone by using `while`.

3.1.1 You are a natural

Let's start by some nice and well done simple things. You have to display every natural number between 1 and n (included), with white spaces between each number, but not before the first or after the last number displayed. If n is inferior to 1 you must print an error message.

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

3.1.2 Optimus Prime

Write a function that outputs on the console the list of prime numbers between 1 and n (included). Once again, with white spaces between numbers but not after the last one. If n is inferior to 1 you must print an error message.

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

3.1.3 Running gag-acci

Let's compute the terms of the Fibonacci sequence. Each term is the sum of the two previous terms. Of course, you have to implement the iterative version because it is far quicker than the recursive version you already know. Therefore, it will be tested on **very** big numbers (take note of the `long`, and not `int`, as the return value). If n is negative you must return -1 and print an error message.

Reminder : the first terms are 0, 1, 1, 2, 3, 5, 8, and 13.

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

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

3.1.4 I'm Stronkk

An integer can be defined as *strong* if the sum of the factorials of its digits is equal to itself. For example, 145 is *strong* since $1! + 4! + 5! = 145$. You have to write a function which displays every *strong* integer between 1 and n (included), with white spaces between each number, but not before the first or after the last number displayed. If n is inferior to 1 you must print an error message.

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

Pro tip

We strongly recommend writing a separate factorial function :

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

As a reminder, $0! = 1$ and in a more general way :

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

3.1.5 Square Root

You first have to write the function returning the absolute value of a `float` given as parameter.

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

By using the previous function, write a function computing the square root of a number given as parameter, up to a precision of 10^{-3}. You will use Newton's method [1] for that purpose. If n is negative you must print an error message and return -1.

`https://en.wikipedia.org/wiki/Newton%27s_method`.

Formula

The general idea of the formula is the following :

$$\begin{cases} X_0 & = & S \\ X_{n+1} & = & \frac{1}{2}(X_n + \frac{S}{X_n}) \end{cases} \quad (1)$$

S = The number the square root of wich we are searching for.

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

3.1.6 Powerrrrrrr ! ! ! !

Write a function power which computes iteratively a^b.

When a = 0 and b < 0, you must return 0 and print an error message.

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

3.1.7 All I Want For Christmas Is C#

Christmas is right around the corner but you cannot stand so many trees being cut down every year anymore. This is **it**, this Christmas will be celebrated inside the console ! You will have to draw a beautiful Christmas tree, of size n. If $n > 3$, you will need 2 lines for the trunk, in other cases only 1 is needed.

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

1. For your own curiosity, we urge you to study/code other algorithms !

```
// n = 3
  *
 ***
*****
  *

// n = 5
    *
   ***
  *****
 *******
*********
    *
    *
```

3.1.8 $3x + 1$

Do you know of a problem extremely simple to formulate, but awfully complex to solve ? Let us introduce you to the famous **Syracuse problem**, with a wording so simple it can be explained to a child, but the demonstration is still unknown to this day and has obsessed mathematicians for decades.

Take any number $n > 0$, and follow these instructions :

— If n is even, we start again with $\frac{n}{2}$
— Otherwise, we start again with $3n + 1$

By using multiple examples, we can quickly highlight a strange phenomenon :

— $n = 5$: 5, 16, 8, **4, 2, 1, 4, 2, 1,** ...
— $n = 20$: 20, 10, 5, 16, 8, **4, 2, 1, 4, 2, 1,** ...
— $n = 104$: 104, 52, 26, 13, 40, 20, 10, 5, 16, 8, **4, 2, 1, 4, 2, 1,** ...

No matter what the starting number is, it seems the following cycle is always there : 4, 2, 1. This is exactly the conjecture introduced by Lothar Collatz in 1937, the proof of which is totally beyond the reach of our mathematics according to the famous mathematician Paul Erdős.

Your task is to compute the number of iterations needed before encountering the value 1 (and, therefore, the cycle 1, 4, 2, 1, 4, 2, ...).

If n is negative or 0 you must print an error message and return -1.

```
public static int Syracuse(int n);
```

As an example, `Syracuse(5) = 5`.

3.2 The debugger

For these exercises an archive containing the code to debug is given. Each exercise has **one or multiples** bugs. You have to use the debugger to find them ; it would be hard to find them without it anyway.

Once you have found the bugs, fix them and add the project to your submission archive. In order to verify that you used the debugger, you have to explain in the README file what the bugs were, how you found them, and how you fixed them. Once you have fixed all the bugs, a message confirming they have been corrected will be displayed in the console.

Be careful

If it is not the case, the exercise will not be considered as done !

3.2.1 While does it bug ? !

For this exercise, you have to fix the bug(s) of the function `ex1` in the file `Ex1.cs`. Once the exercise is done, « Exercise 1 : OK » will be displayed on the console. It is up to you to find the purpose of this exercise's function.

3.2.2 A FORmidable bug

For this exercise, you have to fix the bug(s) of the function `ex2` in the file `Ex2.cs`. Once the exercise is done, « Exercise 2 : OK » will be displayed on the console. It is up to you to find the purpose of this exercise's function.

3.2.3 A not very efficient sort

For this exercise, you have to fix the bug(s) of the function `ex3`, `ex3_2`, `ex3_3` in the file `Ex3.cs`. Be careful, not every function has bugs. The behaviour of the functions `ex3` and `ex3_3` are described in the comments. It is up to you to find the sorting algorithm used. Once the exercise is done, « Exercise 3 : OK » will be displayed on the console.

3.2.4 Bonus : Cracking

For this exercise you are provided with a source file. When launching the program, a password will be asked. In order to obtain the bonus points, you have to find this password and write it in your `README` file. As for the previous exercises, you have to explain your approach in the `README` file in order to have your exercise validated. A little hint : the debugger might be very useful for this problem.

There is nothing more deceptive than an obvious fact.