TP C#1: My First Bernard

Assignment

You must submit a zip file with the following architecture:

```
rendu-tp1-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- TP1/
        |-- TP1.sln
        |-- TP1/
            |-- Everything except bin/ and obj/
```

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified, they will be considered as bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

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

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

This practical session aims to teach you what a console is and how to use it. You will also learn to use the Console class of the C# library, to interact with the console.

2 The Linux console

2.1 CLI/GUI Difference

As a good Windows user, you like to click on buttons with your mouse, and most of you have used only graphical programs so far. Those programs can be used with the mouse through a GUI (for Graphical User Interface), in which you can interact thanks to some buttons, tabs, text fields etc.

However, you may know that another way to interact with your computer exists, a darker and faster way, the command line, also called CLI.

Some programs are designed to be used with the command line. They are running in what we call a terminal.

2.2 The command line

2.2.1 Bash Interpreter

Now that we have opened a terminal, we can look at what's inside. When a terminal starts, it displays some information:

```
[42sh]$
```

This display is what we call the prompt. It is just an example, your own shell may not look like this. The program executed in the terminal is a shell interpreter. This program waits for commands from the user, to run them (you have already met an interpreter, think back to Caml!).

2.2.2 Basic commands

In it's current state, the shell interpreter is waiting for commands from the user. What a coincidence! You are the user. A command is most of the time composed of a program name, followed by 0 or more arguments.

`Exercise:` try to run the command: `'echo "Hello World!"'` in your terminal. What happens?

- `ls` (LiSt):

  `ls` is a command that can list the files contained in the current directory (the default directory of your shell is `home/`).

  `ls` can take directories' names as arguments. It will then list the content of each directory.

- `cd` (Change Directory):

  `cd` is a command allowing you to move to the directory given as argument.

- `echo`:

  `echo` displays its arguments in the terminal.

- `cat` (conCATenate):

  `cat` displays the content of the files given as arguments.

- `mkdir` (MaKe DIRectory):

  With `mkdir`, you can create a folder. The name of the new folder is given as an argument. If you give to `mkdir` several arguments, several folders will be created.

- `man` (MANual page):

  `man` displays the man page of the command given as argument. This command is your BEST FRIEND as far as the shell is concerned (it will follow you throughout your studies at Epita).

We advise you to read the man pages of each of those commands. You will discover that those commands have many options allowing you to perform what could be considered black magic in any graphical file explorer.

2.2.3 Standard streams

The console uses streams to communicate with the user, there are 3 main streams which are called the standard streams.

- `stdin`

  The Standard Input is the canal which is read by the console. It is by this canal that the user can enter information (commands, for example). A command line program will read information from this canal.

- `stdout`

  The Standard Output is the canal where the console will write information. This canal can then be read by the user (note that the user can be another program!). A command line program will write information in this canal.

- `stderr`

  The Standard Error is very similar to the Standard Output. It allows the console to write messages about run-time errors. A command line program can also write information about errors during the execution of the program in this canal.

2.3 AUTHORS like a boss

To practice a bit with the console, we propose that you try to create your `AUTHORS` like a pro. Follow the steps and don't hesitate to ask your ACDCs if you have a problem.

2.3.1 Change Directory

As a reminder, the AFS (Andrew File System) is a distributed file system, which allows you to access your files from any machine in EPITA. Every time you boot a machine, your part of the AFS is mounted on your filesystem (like a USB key). Be careful: if data is not saved in your AFS, it will be lost next time you connect to your session!

You first need to go to your afs using the `cd` command.

```
cd afs
```

You are now in your afs main directory. You can see its content using the `ls` command.

```
ls
```

2.3.2 Creation of your submission folder

Create your submission directory. You need to create a directory using the `mkdir` command.

If you run the `ls` command, you will see your new folder. Change your current directory to this one using the `cd` command.

```
cd prenom.nom
```

Here you are, In your submission directory!

2.3.3 Creation of the AUTHORS file

Now that you are in the correct directory, the serious business begins, create your `AUTHORS` file by running the following command:

```
echo "* prenom.nom" > AUTHORS
```

This command may look a bit complex. The `echo` command will write in the Standard Ouput, which is redirected (> symbol) to the `AUTHORS` file (which is created as it does not exist yet).

You do not need to understand this command (yet), but you can already figure out that the console is a very powerful tool. Indeed, doing the same operation with a GUI would have taken many more ressources (open a text editor, save the file, etc.), and time.

2.3.4 Who am I?

To check if your `AUTHORS` file is correct, you can display its content using the `cat` command:

```
cat AUTHORS
* prenom.nom
```

However, you also want to make sure there are no hidden unwanted characters in your `AUTHORS`. You can use the `-e` option of the `cat` command:

```
cat AUTHORS
* prenom.nom
```

This command displays all the characters, even the invisible ones. Here, the line break is displayed as a "$".

3 C# and the console

In most of the C# practical sessions, we will ask you to develop console programs. In C#, there are several ways to interact with the user using the command line. Let's see some of them.

3.1 Get the arguments given to the program

As you already know, you can run programs from the console. Some of these programs need arguments (think about `mkdir` for instance).

Your C# programs can also get arguments that are given in the console. You can get them thanks to the "args" argument in the `Main` function.

```
1  public static void Main(string[] args)
```

The "args" argument of the Main function is an array, you will see this data structure in more detail in a few weeks.

3.2 The Console class

To interact with the console in C#, you will use a class of the C# library that will allow you to use the Standard Streams: the Console class.

This class contains methods to write or read in the console. It also contains methods to modify the properties of the console.

We strongly recommend that you read the MSDN page of this class.

3.3 Write in the Standard Output

The first steps of a novice programmer are often to display a message on the Standard Output. Some of you have already written a "Hello World!" program (if not, do not worry, this will get fixed in a few minutes).

The C# Console class contains two functions to write in the console:

- `Console.Write(arg)`

  This method converts `"arg"` in a string, and displays it on the Standard Output. Read the MSDN page for more information.

- `Console.WriteLine(arg)`

  As a reminder, this method converts `"arg"` in a string, and displays it on the Standard Output, followed by a line break. Read the MSDN page for more information.

Exemple :

```
1  using System;
2
3  namespace TP1
4  {
5      internal class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("Hello Bernard.");
10             Console.WriteLine(42);
11             Console.Write("Hello");
12             Console.Write("Hello\n");
13             Console.WriteLine("What door?");
14         }
15     }
16 }
```

Result:

```
Hello
42
HelloHello
What door?
```

3.4 Read the Standard Input

The C# Console class contains two functions to read in the console:

- `Console.Read()`

  This method reads the next available character in the Standard Input, and sends it to the program as an int.

  In a more practical way, this method waits for the user to enter a character. Once the character is sent, the program continues.

- `Console.ReadLine()`

  This method reads the next available line in the Standard Input, without the final line break character, and sends it as a string to the program.

  In a more practical way, this method waits for the user to enter some text and then press enter. Then, the program continues.

```
1  using System;
2
3  namespace TP1
4  {
5      internal class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.Write("Please enter a char: ");
10             int c = Console.Read();
11             Console.WriteLine("\nYou typed the char: " + (char)c);
12         }
13     }
14 }
```

3.5 Other console options

The console class contains many other methods and attributes that will be useful for the rest of the exercises. We strongly recommend that you read the corresponding MSDN pages:

- `Console.Clear()`
- `Console.Beep()`
- `Console.ResetColor()`
- `Console.BackgroundColor`
- `Console.ForegroundColor`

4 The While loop

4.1 The iterative loop

One of the core principles of programming is to repeat an action several times.

Until now, you have been used to the principle of recursion to repeat actions.

There is another way to repeat actions in algorithms, the use of iterative loops.

A loop is a part of code that can be repeated. It is quite straightforward: once the loop has run its course, the code's execution starts over at the beginning of the loop.

```
i = 0
begin loop
        i = i + 1
        print "hello"
end loop
print i
```

In this example, the code starts to be executed from the "start loop" statement. Let's go through the loop line by line. The first line adds 1 to the i variable, which is now equal to 1. The second line displays "hello". We reach the end of the loop. Then, we go back to the beginning of the loop. The first line adds 1 to the i variable, which is now equal to 2. The second line displays "hello". We reach the end of the loop, and so on.

4.2 The conditional loop

As you may have guessed, the loop in the first example will never stop. The i variable will get higher and higher, and tons of "hello" will be displayed on the console.

This kind of behavior is often unwanted. It is similar to an infinite recursion. As for the recursion, a loop needs a condition to stop its repetition.

```
i = 0
begin while i < 10
        i = i + 1
        print "hello"
end while
print i
```

In this example, we use a "while" loop. When we first encounter the "begin while", the i variable is equal to 0, which is less than 10. The program enters the loop. We add 1 to the i variable and we display "hello". When we are at the "end while" line; we jump back on the line "begin while", and we check the condition again: i is equal to 1, the condition is true, the program enters the loop for a second time.

Let's go a bit faster and see what the behavior of our program is when I reach the value of 10. We jump back to the "begin while" line and we test the condition. The i variable is equal to 10. The statement $10 < 10$ is false. The condition isn't respected. In this case, the program does not go into the loop and starts again at the line right after the "end while" line.

At the end, the i variable is displayed, 10 is written on the Standard Output. We have just written a program displaying 10 times "hello" and one time "10".

4.3 The while loop in C#

In C#, the while loop has the following syntax:

```
1 while (condition)
2 {
3     //code to execute;
4 }
```

Here, the condition is a boolean expression, as the one you use in the "if". The opening brace indicates the beginning of the block contained in the loop. The closing brace indicates the end of the loop.

Let's take the previous example:

```
1 int i = 0;
2 while (i < 10)
3 {
4     i = i + 1;
5     Console.WriteLine("hello");
6 }
7 Console.WriteLine(i);
```

5 Exercises

5.1 Exercise 0 : Hello (West)World!

```
1  static void HelloWorlds(int n);
```

Write a function that displays n times "Hello (West)World!" on the Standard Output when it's called.

Be Careful: The displayed string must be EXACTLY "Hello (West)World!".

Example, with n = 10:

```
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
```

5.2 Exercise 1 : Echo

```
1  static void Echo();
```

Write a function that waits for an input from the user, before displaying it on the Standard Output.

Example:

```
Doesn't look like anything to me.                // User input

Doesn't look like anything to me.                // Program output
```

5.3 Exercise 2 : Print Reverse

```
1  static void Reverse();
```

Write a function that waits for an input from the user, before displaying it reversed on the Standard Output. You MUST use recursion for this function.

Example:

```
what door?                  // User input

?rood tahw                  // Program output
```

Help: you can access the i index character in the string with the following code:

```
1  s[i]
```

Help: you can get the length of a string with the following code:

```
1  s.Length
```

5.4 Exercise 3 : Love ACDC

```
1  static void LoveAcdc();
```

Write a function that displays the following message on the Standard Output:

```
  * * *   * * *
* *   * * *   * *
*       *       *
* *  <3 ACDC  * *
  * *       * *
    * *   * *
      * * *
        *
```

It must be displayed in white with a pink background color.

Be Careful: Don't forget to reset the colors of the terminal at the end of your function!

5.5 Exercise 4 : More MCQs

```
1 static void MCQ(string question, string aws1, string aws2,
2                 string aws3, string aws4, int aws);
```

Write a function that generates an MCQ.

The function must display, in this order:

- The string `question`
- The string `aws1` preceded by "1) "
- The string `aws2` preceded by "2) "
- The string `aws3` preceded by "3) "
- The string `aws4` preceded by "4) "

Then, the function takes an input from the user:

If this entry is equal to `aws`, display the following message.

```
Good job bro!
```

`[aws]` must be replaced by the value of `aws`.
If this entry is not equal to `aws` but $0 <$ input < 5, display the following message:

```
You lose... The answer was [aws].
```

If the input is invalid, display:

```
So counting is too hard, n00b...
```

5.6 Exercise 5 : Best Years

```
1  static void BestYears();
```

Write a function that displays by year whether Epita graduating classes are good or bad.
Here are the rules to respect: the **even** years are **good** years, and the **odd** years are **bad** years.

Special rule: Year 2020 is the **best** year, and year 2022 is a **bad** year.
Your function must display the years starting from 1989 to 2022.
If the class is the **best** year, display:

```
Best Year
```

If the class is a **bad** year, display:

```
Bad Year
```

If the year is a **good** year, display:

```
Good Year
```

Here's a display example, be careful: it is not the one we expect.

```
Good Year
Bad Year
Bad Year
Best Year
Good Year
Good Year
```

5.7 Exercise 6 : Morse

```
1  static void Morse(string s, int i = 0);
```

Write a function able to "read" Morse, i.e make noise according to a given string.

In the string, you can find 3 different characters. The function must interpret them this way:

- '.' (dot) :
  When a dot is encountered, a 900Hz sound of 0.15 seconds must be played.
- '_' (underscore) :
  When a dot is encountered, a 900Hz sound of 0.45 seconds must be played.
- ' ' (space) :
  When a space is encountered, your program must pause for 0.45 seconds.

Pro tip: Read the MSDN page of the Thread.Sleep() method.

Example:

```
_ .. __ . _ ___ .__. ._.. ._ _.__ _ .... . __. ._ __ . ._._._
```

5.8 Exercise 7 : LightHouse

```
1  static void Lighthouse(int n);
```

Write a function that displays an ASCII-ART lighthouse. The height of the lighthouse depends of the parameter of the function.

The lighthouse must have at least a roof and a base.

```
// base :
 =====
_||_||_
-------

// roof :
  /^\
  |#|
```

Between the base and the roof, we can find n floors:

```
// 1st floor :
 |===|
  |0|
  | |
```

Here are some examples:

```
// n = 0

  /^\
  |#|
 =====
_||_||_
-------

// n = 1

  /^\
  |#|
 |===|
  |0|
  | |
 =====
_||_||_
-------
```

```
// n = 3

  /^\
  |#|
 |===|
  |0|
  | |
 |===|
  |0|
  | |
 |===|
  |0|
  | |
 =====
_||_||_
-------
```

These violent deadlines have violent ends.