C# Practical 7: The Empire Strikes Back...

1 Before starting

At the end of this subject, you will need to follow special rules for giving in your work.

```
rendu-tp-login_x.zip
|-- rendu-tp-login_x/
        |-- AUTHORS
        |-- README
        |-- Moulinette/
                |-- Moulinette.sln
                |-- Moulinette/
                        |-- Program.cs
                        |-- Moulinette.cs
                        |-- Rendu.cs
                        |-- Exo.cs
                        |-- Correction.cs
                        |-- Everything except bin/ and obj/
        |-- Bonus/
                |-- Moulinette.sln
                |-- Moulinette/
                        |-- Everything except bin/ and obj/
```

Replace "`login_x`" by your own login.

Don't forget to check the following rules before giving in :
- `AUTHORS` file must be good. (a star, a space, your login and a newline : "* login_x\n"). Else, you will get a 0.
- No `bin` or `obj` folders in your project. Else, you will get a 0.
- **Your code `ABSOLUTELY` must compile !** Else, you will get a 0.

2 Introduction

2.1 Goal

During this practical class, we will study the following concepts:
- Executing external commands in your program.
- Standard streams (stdout and stderr).
- Using the List<T> container.

3 Lesson

3.1 Process Execution

One of the goals of this subject is to execute an external program or command through your C# program. To do that, you are going to use the classes Process and ProcessStartInfo (System.Diagnostics). In order to thoroughly learn everything about these classes (memory size limit, setting a timeout, etc...), refer to the official documentation pages (msdn). The following example executes the program "test.exe" in the "folder/" folder.

```
private void execute()
{
        //Create a process configuration
        ProcessStartInfo pStart = new ProcessStartInfo();
        //Add the path of the executable
        pStart.FileName = "folder/test.exe";
        //Run the executable
        Process p = Process.Start(pStart);
}
```

3.2 Standard Streams

In computer programming, standard streams are communication channels between a program and its environment in which it is executed. We count three types of these streams : the input (stdin, you use it when you use Console.ReadLine), the output (stdout (Console.WriteLine) and stderr (Console.Error.WriteLine)).

Here's an edited version of the previous example that lets us get the two last streams:

```
private void execute()
{
        ProcessStartInfo pStart = new ProcessStartInfo();
        pStart.FileName = "folder/test.exe" ;

        //Enable the stdout getter of the process
        pStart.RedirectStandardOutput = true;
        //Enable the stderr getter of the process
        pStart.RedirectStandardError = true;

        Process p = Process.Start(pStart);
        //Get the process stdout stream
        string stdout = p.StandardOutput.ReadToEnd();
        //Get the process stderr stream
        string stderr = p.StandardError.ReadToEnd();
}
```

3.3 List<T>

List<T> is a standart container in the .NET library. It's an optimized implementation of dynamic lists (list vectors in C++). The "T" is the type of the list's content: List<int> is a list for which all elements are integers, List<string> is a list for which all elements are of type string, etc... It's a template, so don't worry, you will encounter this again later. You only need to understand that you can only add integers on a List<int>. Like for the process execution part, to thoroughly examine the List<T> class (getting the number of elements, accessing an element, etc...), refer to msdn.

Here's an examble of List<T> usage:

```
private void fun()
{
        //Initialize a new list of integers
        List<int> list = new List<int>();
        //Add 1 to the list
        list.Add(1);
        //Add 2 to the list
        list.Add(2);
        for (int i = 0; i < list.Count; ++i){
                //Get element at index i
                int n = list[i];
                //Print the element on stdout
                Console.WriteLine(n.ToString());
        }
}
```

4 Moulinette

4.1 Introduction

The Moulinette is a program used by your ACDCs (and will be used by your future ASMs, ACUs and YAKAs), to check if your source code works. It compiles, executes and compares the result of your program with the correction. This week, you will implement a simple Moulinette in C#. Your Moulinette must:

1. Get all corrections.
2. Get all repositories and exercises for each repository.
3. Execute exercises (not compile) and compare the output streams (stdout and stderr) with the correction.

Don't worry: this practical isn't too difficult if you master classes and all the lesson part (what they are and how to use them). You have to use the source code given in the lesson part.

Consider that the correct folder hierarchy to execute your Moulinette is as follows :

```
moulinette.exe
correction/
|-- nameOfTheFirstExercice/
        |-- stdout.txt
        |-- stderr.txt
|-- nameOfTheSecondExercice/
        |-- stdout.txt
        |-- stderr.txt
|-- [...]
rendu/
|-- rendu-tp-login_x/
        |-- nameOfTheFirstExercice/
                |-- exo.exe
        |-- nameOfTheSecondExercice/
                |-- exo.exe
        |-- [...]
|-- rendu-tp-login_y/
        |-- nameOfTheFirstExercice/
                |-- exo.exe
        |-- nameOfTheSecondExercice/
                |-- exo.exe
        |-- [...]
|-- [...]
```

Before starting to program, you must read the subject in its entirety once. Then, read it a second time. If and only if you have understood the subject can you start, otherwise keep reading.

4.2 Correction.cs

4.2.1 Introduction

This class will contain the name, the folder path and the correct standard output and error streams of an exercise.

4.2.2 Attributes and Getters

This is the list of attributes that you'll have in your Correction class:

```
private string name;
private string folder;
private string stdout;
private string stderr;
```

And now, the list of getter methods (simple return value of attributes) that you'll have in your Correction class:

```
public string getName();
public string getStdout();
public string getStderr();
```

4.2.3 Constructor

The constructor of the Correction class initializes ONLY the name and the folder path of the object, and not stdout and stderr). The name of the exercise is the folder name.

```
public Correction(string folderName);
```

4.2.4 Initialization

This method initializes the stdout and stderr attributes of the object. For this practical, the correction of the stdout stream is the contents of stdout.txt and the correction of the stderr stream is the contents of stderr.txt file. This method returns true if successful, false otherwise.

```
public bool init();
```

Hint : look up System.IO on msdn to be able to read in a folder (DirectoryInfo).

4.3 Exo.cs

4.3.1 Introduction

This class will contain the name, the folder path and stdout/stderr (after execution) of an exercise from a student repository.

4.3.2 Attributes and Getters

This is the list of attributes that you'll have in your Exo class:

```
private string name;
private string folder;
private string stdout;
private string stderr;
```

And now, the list of getter methods (simple return value of attributes) that you'll have in your Exo class:

```
public string getName();
public string getStdout();
public string getStderr();
```

4.3.3 Constructor

The constructor of the Exo class initializes ONLY the name and the folder path of the object, and not stdout and stderr. The name of the exercise is the folder name.

```
public Exo(string folderName);
```

4.3.4 Execution

This method executes the exercice. At the end of the execution, the method will get the output streams (stdout and stderr). For this practical, we consider that the binary file name must be "exo.exe". If any binary files with this name is found, this method must return false. If any problems appear, return true, otherwise false.

```
public bool execute();
```

4.4 Rendu.cs

4.4.1 Introduction

In your project, an object with the type Rendu will represent the repository of a student. Each repository contains 0 or more folders. Each folder represents an exercise. Each exercise contains a binary file named "exo.exe".

4.4.2 Attributes

This is the list of attributes that you'll have in your Rendu class:

```
private string folder;
private List<Exo> listExo;
```

4.4.3 Constructor

The constructor of the Rendu class initializes the folder path and the list of exercises (listExo) as an empty list.

```
public Rendu(string folderName);
```

4.4.4 Initialization

This method will initialize the list of exercises (listExo) of the object. Remember : listExo is a list of Exo objects. You must add elements to the list only when the initialization method of the Exo object returns true. This method returns false only if no exercise is found.

```
public bool init();
```

4.4.5 The runCorrection Method

This method, for each element of the Correction object list input, must :
- Search the exercise in the listExo for the name.
- If it's found, print nameOfExercice + ": not found!\n", else call the execute() method of the Exo object.
- If execute() returns false, print nameOfExercice + ": error execute()!\n", else compare stdout and stderr with the correction.
- If stdout and stderr are correct, print nameOfExercice + ": OK\n", else print nameOfExercice + ": FAIL\n"

At the end, return the number of successful tests (correct stdout and stderr).

```
public int runCorrection(List<Correction> listCorrection);
```

4.5 Moulinette.cs

4.5.1 Attributes

This is the list of attributes that you'll have in your Moulinette class:

```
private List<Correction> listCorrection;
private List<Rendu> listRendu;
```

4.5.2 Constructor

The constructor of the Moulinette class initializes the list of corrections (listCorrection) and the list of repositories (listRendu) as empty lists. It can print some information (like "Moulnette v1.0").

```
public Moulinette();
```

4.5.3 Initialization

This method initializes the list of corrections (listCorrection) with the contents of the "correction" folder and the list of repositories (listRendu) with the contents of the "rendu" folder. You must add elements to these lists only when the initialization method of the object returns true. This method returns true if the list of corrections and the list of repositories aren't empty. You can print the number of elements from each list at the end.

```
public bool init();
```

4.5.4 Execution

The execution method of the Moulinette class calls the runCorrection method and prints the percentage of successful tests of each element from the repository list (listRendu).

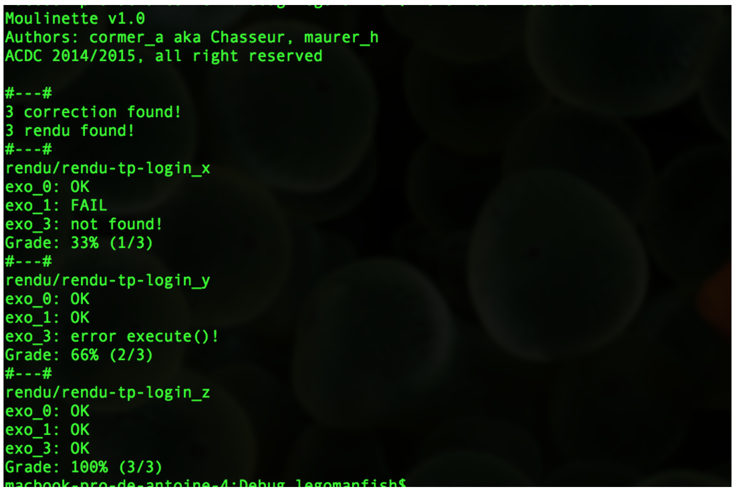
```
public void execute();
```

4.6 Program.cs

Now, all classes you will need are written. You just have to write your Main function. The following code MUST work:

```
static void Main(string[] args){
        Moulinette moulinette = new Moulinette();
        if (moulinette.init()){
                moulinette.execute();
        }
}
```

When your ACDC executes your code, he must have something like this below picture:



```
Moulinette v1.0
Authors: cormer_a aka Chasseur, maurer_h
ACDC 2014/2015, all right reserved

#---#
3 correction found!
3 rendu found!
#---#
rendu/rendu-tp-login_x
exo_0: OK
exo_1: FAIL
exo_3: not found!
Grade: 33% (1/3)
#---#
rendu/rendu-tp-login_y
exo_0: OK
exo_1: OK
exo_3: error execute()!
Grade: 66% (2/3)
#---#
rendu/rendu-tp-login_z
exo_0: OK
exo_1: OK
exo_3: OK
Grade: 100% (3/3)
macbook-pro-de-antoine-4:Debug legomanfish$
```

5 Bonus

Before starting to work on the bonus, you must have completed all the previous sections. If all is working well, you can go ahead and give the bonus a shot. You can let your imagination speak, but you have to log and specify everything you add as a bonus to the practical in a README file. If you do not specify what you have done in addition, you won't be accorded the bonus points.

The bullshit is strong with this one...