TP C#7

Submission

Archive

You must submit a zip file with the following architecture :

```
– rendu-tp-firstname.lastname.zip
        |– rendu-tp-firstname.lastname/
                |– AUTHORS
                |– README
                |– Moulinette/
                        |– Moulinette.sln
                        |– Moulinette/
                                |– Program.cs
                                |– Moulinette.cs/
                                |– Rendu.cs
                                |– Exo.cs
                                |– Correction.cs
                                |– everything except bin/ and obj/
```

— You must obviously replace *firstname.lastname* with you name.
— Your code must compile.
— Remove bin/ and obj/ folders (we don't need them).

AUTHORS

This file should contain the following : a star (*), a space, your login, and a linefeed. Here is an example :

```
* firstname.lastname
```

Please note that the filename is AUTHORS with NO extension.

Objectifs

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> collection.

1 Lesson

1.1 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, and so on ...), refer to the official documentation pages (msdn). The following example executes the program "test.exe" in the "folder/" folder.

```
private void Execute()
{
   // Instanciate a new object of type ProcessStartInfo
   ProcessStartInfo pStart = new ProcessStartInfo();
   // Add the path of the executable to this object
   pStart.FileName = "folder/test.exe";
   // Disable window pop-up for the process
   pStart.CreateNoWindow = false;
   // Disable the shell of the OS to start the process
   pStart.UseShellExecute = false;

   // Run the executable
   Process p = Process.Start(pStart);
}
```

1.2 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 :

- **Stdin**, for standard input. You use it with **Console.ReadLine()**.
- **Stdout**, for standard output. You use it with **Console.WriteLine()**.
- **Stderr**, for standard error output. You use it with **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" ;
   pStart.CreateNoWindow = false;
   pStart.UseShellExecute = false;

   //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();
}
```

1.3 List<T>

List<T> is a standard collection in the .NET library. It's an optimized implementation of dynamic lists. 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, and so on ... It's a template, so don't worry, you will encounter this again later. You only need to understand that you can add only 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, and so on ...), refer to msdn.

Here's an example of **List<T>** usage :

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

2 Exercise : Moulinette

2.1 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 will :

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

Don't worry : this practical is not so difficult if you have followed the lectures. Use the example given in the lesson part.

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

```
moulinette.exe
correction/
  |-- ex1/
    |-- stdout.txt
    |-- stderr.txt
  |-- ex2/
    |-- stdout.txt
    |-- stderr.txt
  |-- ...

rendu/
  |-- rendu-tp-login_x/
    |-- ex1/
      |-- exo.exe
    |-- ex2/
      |-- exo.exe
  |-- rendu-tp-login_y/
    |-- ex2/
```

```
        |-- exo.exe
   |-- ...
```

2.2 Class : Correction.cs

Introduction

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

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

```
// Attributes:
private string name;
private string folder;
private string stdout;
private string stderr;
```

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

```
// Getters:
public string getName();
public string getStdout();
public string getStderr();
```

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);
```

Init

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**).

2.3 Class : Exo.cs

Introduction

This class will contain elements associated with an exercise. Here an exercise is the executable file given by the student. Therefore, this class must have an attribute for the path to this executable file.

Attributes and Getters

Here is the attributes expected for this class :

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

A getter is associated to each attribute (just return the value of the private attribute.)

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

Constructor

The constructor of this class only initialize the "name" and "folder" attributes. The attributes "stdout" and "stderr" will be initialized only after the execution of the exercise of the student.

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

Execute

The **Execute** method will be used to execute the exercise of a student. It will then put the standard output and error stream in the associated variable **stdout** and **stderr**.

This method returns a boolean : **true** if no problem was found, and **false** if there was a problem (no executable found, ...).

```
public bool Execute();
```

Use the examples seen in the lesson part to do this exercise!

2.4 Class : Rendu.cs

Introduction

During this practical session, an object of type "Rendu" is associated to the files submitted by the student. Therefore, you will have to instanciate an object of type "Rendu" for each student submission. Every student is supposed to have an executable "exo.exe" in his submission folder.

Attributes

Here are the attributes expected for this class :

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

The "folder" attributes is a string that contains the absolute path of the folder of the student and the attribute "listExo" is a list of object of type "Exo" that you have implemented previously.

Constructor

The constructor of the class Rendu expects only one argument : the name of the student's folder. You must initiliaze the attribute "folder" with the string given as argument, and initialize listExo with an empty list.

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

Init

This method will crawl through each subfolder in order to find the exercise of the student. Each exercise will be added to the "listExo" attribute

```
public bool Init();
```

This method will return **false** if no exercise was found in the given folder, **false** otherwise.

RunCorrection

This method will do the following for each element of the **Exo** list :

— Search in the correction list for the exercise with the same name.

— If no exercise was found, display **ExerciseName** + **" : not found !\n"**.

— If an exercise was found, call the method **Execute()** of the Exo object.

— If **Execute()** returns false, display **ExerciseName** + **" : Exec error !\n"**.

— If **Execute()** returns true, compare **stdout** and **stderr** with the correction.

— If **stdout** and **stderr** are correct, display **ExerciseName** + **" : OK\n"** otherwise, display **ExerciseName** + **" : FAIL\n"**

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

The return value of this method is the number of tests passed.

2.5 Class : Moulinette.cs

Introduction

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

```
// Attributes:
private List<Correction> listCorrection;
private List<Rendu> listRendu;
```

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 "Moulinette v1.0").

```
public Moulinette();
```

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();
```

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();
```

2.6 Main : Program.cs

All classes you will need are now 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();
  }
}
```

Expected result (on a terminal) :

```
--------------------
3 correction found!
3 rendu found!
--------------------
rendu-tp-login_x:
exo_0: OK
exo_1: KO
exo_2: Not found!
Grade: 33% (1/3)
--------------------
rendu-tp-login_y:
exo_0: OK
exo_1: OK
exo_2: Exec error!
Grade: 66% (2/3)
--------------------
rendu-tp-login_z:
exo_0: OK
exo_1: OK
exo_2: OK
Grade: 100% (3/3)
--------------------
```

The code is the law.