TP C#5 : Interrogation Room

Submission instructions

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

```
tp-csharp5-sherlock.holmes/
|-- README
|-- Interrogation_Room/
    |-- IO.sln
    |-- .gitignore
    |-- Basics/
        |-- Everything except bin/ and obj/
    |-- Interrogation/
        |-- Everything except bin/ and obj/
```

Do not forget to check the following requirements before submitting your work:

- You shall obviously replace `firstname.lastname` with your login.
- The `README` file is mandatory.
- There must be no `bin` or `obj` folder in the repository.
- You must respect the prototypes of the given and asked functions.
- Remove all personal tests from your code.
- **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

Murder ! Sherlock Holmes yet again needs your help to crack the case. He has already made a list of witnesses and suspects. Unfortunately, he has his hands full on another case and asks you to find, interrogate each person and gather all the information for him. In order to be efficient, you will obviously learn how to manipulate files in C#.

1.1 Objectives

At the end of this TP, you will be able to perform actions on files such as reading and writing. We have already mentioned the standard streams previously (**stdin, stdout, stderr**) and these notions will be seen in further detail in this TP.

WARNING !

We will ask you for plenty of displays in different outputs. Be careful to respect meticulously the format asked for (e.g: commas, spaces, upper cases...). It is also equally important to write in the correct streams. Any mistake linked to this will be considered invalid and points will be deducted.

2 Lesson

In this section, the notions necessary to finish your TP will be explained. You can also go on MSDN, to study how the different functions seen in this lesson work. It is important to have the reflex to check the documentation before asking for help. It will often help to answer your questions, and it's really complete and includes many useful examples.

2.1 Input/Output Stream

There are 3 standard streams: **stdout, stdin and stderr**.

You have already learned how to read and write in a terminal via your program. When you display an output with **Console.Writeline()**, it's on the **stdout** stream that you write. In the same way, to recover information provided by the user, you use **Console.Readline()**, you actually read in the **stdin** stream. Therefore, you already know how to use streams.

When you want to write to a file, an **input stream** is created. Conversely, if you want to read the contents of a file, an **output stream** is created. Think of streams as gates allowing you to access the contents of a file. Some gates only allow you to read from a file (input stream) and others allow you to write to it (output stream).

So when you open a file, what happens is that you create a stream.

2.2 Path

Files and folders are stored as a tree. A folder can have zero or more children that are files and folders as well. Thus, all files and folders contained in a folder are called children of that folder. The parent of a file is the folder in which the file is contained.

There is a special folder, the root. All the folders and files on the disk are descendants of this folder.

By convention, a '/' is added to the end of the folder name to differentiate it from the file.

Paths are strings of characters that allow you to move around in this tree of files and folders.

1. - "/" is the root.
2. - "./" is the actual folder.
3. - "../" is the parent folder.
4. - "***name***" is the <name> of a file or a folder.
5. - "../test" is, in the parent folder, the file or folder test.

There are two types of paths: relative and absolute.

A relative path is defined in relation to the current location in the tree. For example, if you are executing code in a C project, the current folder is where the executable is located.

An absolute path is a path prefixed by / and which thus starts from the root. Its advantage is that it does not depend on where you are currently located. However, it is often unknown. Here are some examples of correspondences between absolute and relative paths:

```
Example: we are in /tmp/tests/
../ => /tmp/    # The parent of tests/ is tmp/
./../../ => /   # The parent of the parent of tests/ is the root /
../../../ => /  # The parent of the root is the root.
```

We can find two important parts in a file. Information about itself and its content.

The **FileInfo** class allows you to get and modify information about a file (at your own risk).

For example:

1. Creation date
2. Length
3. Parent folder
4. Absolute path
5. ...

On the other hand, the **File** class allows one to manipulate the content of the file. There are many functions that allow the user to read, write, create or delete a file. Let's have a look at an example of code to see it more clearly:

```
1 FileStream fs = File.Create("test");
2 if (File.Exists("test"))
3 {
4     Console.WriteLine("the test file belongs to the working directory !");
5 }
6 fs.Close();
7 File.Delete("test");
```

- **File.Create(path)**: This method returns a **FileStream** instance. If the file does not exist, it is created; otherwise the old one is overwritten.
- **File.Exists(path)**: This method returns a boolean, depending on the existence of the path file.
- **FileStream.Close()**: This method closes a **FileStream** instance, fs in our example.
- **File.Delete(path)**: Deletes the file if it exists.

Reminder

It is important to understand that when you open a file to read or write it, becomes a stream. The return of **File.Create()** is precisely a **FileStream**.

2.3 StreamReader

If you have gone to the documentation of the **FileStream** class, you could see that it is both possible to write and read. In fact the **FileStream** class is very complete and allows you to do many things. However, we will rather use the **StreamReader** and **StreamWriter**, which allows the user to do only one thing at a time : read or write.

```
1  StreamReader myReader = new StreamReader("test");
2  // We can also init a StreamReader with File.OpenText()
3
4  string firstLine = myReader.ReadLine();
5  string secondLine = myReader.ReadLine();
6  int thirdLineFirstChar = myReader.Read();
7  int thirdLineSecondChar = myReader.Read();
8  string rest = myReader.ReadToEnd();
9
10 myReader.Close();
```

- **new StreamReader(path)** : This method allows us to create an **input** stream from a file. Be careful, if the file does not exist, the method returns an exception and your program stops immediately.
- **StreamReader.ReadLine()** : This method allows you to use an **input** stream to read the contents of the file linked to it, line by line. It is not possible to read the same line (unless you use another function made for) since the function takes the trouble of placing a pointer to the next line to read. If there is nothing more to read, the function returns null.
- **StreamReader.Read()** : This method is similar to **ReadLine()**, but instead of reading a line, it reads a character.
- **StreamReader.ReadToEnd()** : This method reads everything from the pointer to the end of the file. However, if the pointer is located at the end of the file (this one has already been read entirely) the function does not return **null** but an empty **string**.
- **StreamReader.Close())** : This method closes the stream; it is important not to forget to close a stream once the processing is finished.

Tip

Don't hesitate to check the documentation to see all the methods associated with the StreamReader class.

2.4 StreamWriter

Let's speak now about **StreamWriter**. It allows the user to write in a files via a stream. Let's look at this example:

```
1  StreamWriter myWriter = new StreamWriter("/test.txt");
2  myWriter.Write('a');
3  myWriter.WriteLine("we don't sleep");
4  myWriter.Write('c');
5  myWriter.Write('d');
6  myWriter.Write('c');
7  myWriter.WriteLine("ACDC");
8  myWriter.Write('>');
9  myWriter.Write('A');
10 myWriter.Write('S');
11 myWriter.Write('M');
12 myWriter.Close();
```

Here, the path given to the **StreamWriter** is absolute and starts with a /. Unlike the **StreamReader**, if the file does not exist it will be created !
- **new StreamWriter(path)** : This method allows us to create an **output** stream from from a file. Be careful, if the file does not exist it will be created.
- **StreamWriter.WriteLine()** : This method allows you to use an **output** stream to write to the file linked to it, line by line. It is not possible to write several times on the same line for the same reason as for the method **ReadLine()**.
- **StreamWriter.Write()** : This method is similar to WriteLine(), but does not add a newline at the end.
- **StreamWriter.Close())** : Closes the stream; it is important not to forget to close a stream once the processing is finished.

Here, the content of the test file would be as follows:

awe don't sleep
cdcACDC
>ASM

Tip

Do not hesitate to go and see the documentation to see all the methods associated with the StreamWriter class.

2.5 Recap

Let's take a look at an example of how **StreamReader** and **StreamWriter** are used to make sure we understand everything:

```
string path = "../../../test";
if (!File.Exists(path))
{
    //Create a file to write to.
    using (StreamWriter sw = File.CreateText(path))
    {
        sw.WriteLine("Hello");
        sw.WriteLine("And");
        sw.WriteLine("Welcome");
    }
}

// Open the file to read from.
StreamReader sr = File.OpenText(path);

string s;
while ((s = sr.ReadLine()) != null)
{
    Console.WriteLine(s);
}
sr.Close();
```

Let's look at what each method does here

- **File.Exists(path)** : This method returns a boolean, depending on the existence of the **path** file.
- **File.CreateText(path)** : This method returns a **StreamWriter** and if the file does not exist, the method will also create it. This stream allows you to send information, in this case to write to the file.
- **StreamWriter.WriteLine("string")** : This method allows you to use a stream to write information, here it is in the file opened with **File.CreateText(path)**. Note the similarity with **Console.WriteLine** which was also a stream, but to the **stdout** and not to a file.
- **File.OpenText(path)** : This method returns a **StreamReader**, which allows us to read the information in the open file.
- **StreamReader.ReadLine()** : This method allows us to read the content of a file line by line. At the first call to this function, the first line of the file will be read, at the second call the second line will be read, and so on. When all the content of the file is read, the function returns **null**.
- **StreamWriter.Close()** : This method allows one to close the stream.

In the end, if the file does not exist, it will be created and will contain :

```
Hello
And
Welcome
```

The terminal should display the same thing.

WARNING !

It is essential to close a stream after using it; otherwise you may get errors, especially if you try to open a new stream from a file without having closed the old one!

However, for the **StreamWriter** we didn't need to close it. You might ask, but why? In fact, the **using** statement did it for us, once out of the scope of it (outside the {}), our stream is automatically closed, a good method to never forget to close our streams !

For more information on the different functions seen above, go read the documentation. There are many more or less useful variants depending on what you want to do. It is also important to check the errors that these functions can return.

2.6 Directory

In the same way as a file, a directory has an information part. The **DirectoryInfo** class allows one to obtain and modify about the same information as a file but for a directory.

The class **Directory** allows you to manipulate folders. It is mainly used to obtain information about its contents (the files and folders it contains). It also allows you to delete, create and move folders.

See the docs of the **Directory** and **DirectoryInfo** classes for more information on the methods available in these classes.

3 Basics

3.1 What am I ?

```
1 public static void FileOrDir(string path);
```

This function should display on the standard output whether the given **path** corresponds to a file or a folder.

If the file (or folder) does not exist, also express it on the terminal.

```
1 // foo = file
2 Console.Writeline("foo is a file !");
3
4 // foo = directory
5 Console.Writeline("foo is a directory !");
6
7 // foo does not exist
8 Console.Writeline("foo is neither a file nor a directory !");
```

3.2 What does the file say ?

```
1 public static void DisplayFile(string path);
```

This function must display on the standard output the content of the file pointed by **path** line by line, prefixed by **"Line {i} :"** where i is the line number as in the example below. If the file does not exist, display on the console "Error: No such file or directory !

```
$ cat file.txt

Elementary
My Dear

Watson
```

```
1 DisplayFile("file.txt");
2 // Line 1:
3 // Line 2: Elementary
4 // Line 3: My Dear
5 // Line 4:
6 // Line 5: Watson
7 DisplayFile("non_existant_file.txt");
8 // Error: No such file or directory !
```

3.3 Learn how to write 101

```
1  public static void WriteInFile(string path);
```

This function must write the contents of the string **<content>** to the file indicated by **<path>**. If the file indicated by **<path>** does not exist, the function must create the file and then write to it. If the file already exists, write to the end of the pre-existing content.

```
$ ls
file1.txt file2.txt
$ cat file1.txt
$ cat file2.txt
What is 9 + 10 ?
```

```
1  WriteInFile("file1.txt", "Hello File !");
2  WriteInFile("file2.txt", "19 of course");
3  WriteInFile("file3.txt", "Is anyone here ?");
```

```
$ ls
file1.txt file2.txt file3.txt
$ cat file1.txt
Hello File !
$ cat file2.txt
What is 9 + 10 ?
19 of course
$ cat file3.txt
Is anyone here ?
```

3.4 Copycat or copy cat ?

```
1  public static void CopyFile(string source, string dest);
```

This function must copy the content in the file indicated by the **source path** and write it in the file indicated by the **dest path**. If the source file does not exist, return -1; otherwise return 0. As before, if the dest file does not exist, the function must create the file and then write to it. If the file already exists, write to the end of the pre-existing content.

```
$ ls
file1.txt file2.txt file3.txt
$ cat file1.txt
Hello,

it's me
$ cat file2.txt
$ cat file3.txt
Hello ?
```

```
1  CopyFile("file1.txt", "file2.txt");
2  CopyFile("file2.txt", "file3.txt");
3  CopyFile("file3.txt", "file4.txt");
```

```
$ ls
file1.txt file2.txt file3.txt file4.txt
$ cat file1.txt
Hello,

it's me
$ cat file2.txt
Hello,

it's me
$ cat file3.txt
Hello ?
Hello,

it's me
$ cat file4.txt
Hello ?
Hello,

it's me
```

3.5 Display all files

```
1  public static void MiniLs(string path);
```

This function is an intermediary step for the next function. However, it is mandatory.

You have to display in the way of "ls" all the files (and only the files!) contained in the path folder. The display order is not important. You must handle this the same way as in DisplayFile if **<path>** does not exist.

```
$ ls
dir0/
$ cd dir0/
$ ls
file1 dir2/ file2 file3 dir2/
```

```
1  MiniLs("dir0");
2  // file1 file2 file3
3  MiniLs("dir0/file1");
4  // file1
5  MiniLs("file1");
6  // Error: No such file or directory
```

3.6 I am (G)root

```
1  public static void MyTree(string path);
```

This function should display the tree of files/folders contained in the **path** folder.
If a folder is present, you also have to display the content inside.

```
$ ls
root
$ cd root
$ ls
file1 file2 leaf.txt dir1/
$ cd dir1/
$ ls
almost_there.md come_on/
$ cd come_on/
$ ls
README
```

```
MyTree("root");
|– root/
|– – file1
|– – file2
|– – leaf.txt
|– – dir1/
|– – – almost_there.md
|– – – come_on/
|– – – – README
```

4 Interrogation

4.1 Introduction

In these exercises you will have to help Sherlock Holmes to solve an investigation by creating a program that parses information stored in different files. Those files will be in two formats: the files types `.profile` containing various information on a suspect, and the `.question` which contain a question and the answers of various suspects.

4.1.1 General information

All the functions concerning the Interrogation exercise are to be done in the Interrogation project and in the Interrogation.cs file. Be careful to respect the requested architecture. Moreover, in these exercises you can consider that if the file passed as a parameter exists, then it will always be in the right format.

4.2 Profile class

In these exercises, all information obtained in the files should be saved in a Profile class as below:

```
1  public class Profile
2  {
3      public string Name; //name of the suspect ex: "Sherlock"
4      public string Sex = ""; //gender of the suspect ex: "Male"
5      public string Address = ""; //address of the suspect
6      public uint Age = 0; //age of the suspect ex: 30
7      public uint Size = 0; //size of the suspect ex: 180
8      public List<string> Question = new List<string>(); //list of question
9      public List<string> Answer = new List<string>(); //list of the answer
10     public Profile(string name)
11     {
12         //todo
13     }
14 }
```

Each field of the class corresponds to information that can be retrieved in the `.profile` files except for the `Question`, `Answer` and `AlreadyAnswer` fields, which will be used to store the information of the `.question` files.

Warning

An instance of the class is considered valid if the `Name` field is not empty or does not contain an empty string. In other words, all fields of the class can be empty except the `Name` field, which must always have a non-null value.

4.3 .profile file

One of the files you will have to parse are the .profile files. Each line will always respect this format: `NameOfTheField:Content`. For example:

```
[sherlock@holmes] cat sherlock.profile
name:Sherlock
age:30
address:221B Baker Street
sex:male
size:180
```

warning

The order of appearance of the fields is not fixed. Some fields may not be present (except `Name`). Others can be present several times.

4.4 .question file

The other type of file you will need to parse are the .question files. They will always respect this format :
Question:The question
SuspectName1:His answer
SuspectNamet2:His answer
...
Example:

```
[sherlock@holmes] cat question1.question
question:How are you ?
Sherlock:Fine
Moriarty:Great
Waston:Bad
```

4.5 ReadProfile

```
1  public static Profile ReadProfile(string path)
```

This function asks you to parse the `path` file, which is a file in the format `.profile`. All the information read in the path file must be stored in an instance of the `Profile` class, which must be returned at the end. If a field occurs more than once, then only the last occurrence must be taken into account. The `path` file will always be valid (it will have at least the `Name` field).

tip

The `Split()` method of the string class can be useful! The method `Int.Parse()` method can also be useful!

4.6 ReadQuestion

```
1  public static void ReadQuestion(List<Profile> allProfiles, string path)
```

This function takes as parameters a string `path`, a list of instances `Profile allProfiles`. This function asks you to parse the path file which respects the `.questions` format. If the name of a person who answered is also in one of the profile instances in `allProfiles`, then you have to update this profile by adding the answer and the question respectively in the fields `Answer` and `Question`.

4.7 PrintInformation

```
1  public static void PrintInformation(Profile profile)
```

This function asks you to display on the console all the information stored in the profile class passed in parameter. You must respect the format of this example:

```
Name:Sherlock
Sex:male
Age:30
Size:180
Address:221B Baker Street
Question:How Are you ?
Answer:Fine
Question:Where do you live ?
Answer:221B Baker Street
```

4.8 SaveProfile

```
1  public static void SaveProfile(Profile profile)
```

This function asks you to write in a file the various information contained in the profile class passed in parameter. The name of the file must be the name contained in the field "name" and with the extension .profile. If a file called already has this name, you must replace it with the name of the class. You have to respect the format of the .profile file.

4.9 CreateProfile

```
1  public static void CreateProfile()
```

This function asks you to code a textual interface that allows the user to create a text file. You must read the user's input as long as he does not write `exit`. The function will ask which field the user wants to modify, then what he wants to write in it. If the user writes `exit` during the request of a field, you must stop reading in the terminal. Moreover, if the class is valid (has at least one non-empty `Name`), you must save this data in a file as in the function `SaveProfile`. If the class is invalid, the function must write "Failed to create this new Profile".

Example:

```
which field ?
sex
Male
which field ?
age
30
which field ?
question
Didn't recognize this field : question
which field ?
exit
Failed to create this new Profile
```

4.10 Bonus

4.10.1 Interrogation

```
1  public static void Interrogation()
```

For this function, you will have to create a textual interface that will allow you to use the `5` functions, which you have coded just before. For this, the function will have to read commands sent by the user. There are `6`:
`read profile` : which execute ReadProfile
`read Interrogation` : which execute ReadQuestion
`print profile` : which execute PrintProfile
`create profile` : which execute CreateProfile
`save profile` : which execute SaveProfile
`exit` : which will stop the function
You must create a list of instances of the profile class to be able to store the instances of profile classes created by `ReadProfile` as well as to update them by `ReadQuestion`. For `PrintProfile` and `SaveProfile`, the user will give the name of the profile they want to print. If the name does not match any profile, you will have to write on the terminal "There is no profile Name that matches with NameEnter", with `NameEnter` which must be replaced by the name entered by the user.

You must respect this format:

```
Enter a command:
read profile
sherlock.profile
Enter a command:
read Interrogation
question1.question
Enter a command:
print profile
sherlock
//execution of PrintProfile with sherlock profile
Enter a command:
create profile
//execution of CreateProfile
Enter a command:
save profile
sherlock
//execution of SaveProfile with sherlock profile
Enter a command:
wrong command
Unknown command
Enter a command:
save profile
Unknown
There is no profile Name that matches with Unknown
exit
```

4.10.2 Other Bonus

You are free to implement as many bonuses as you wish as long as they are specified in the `README` and that they do not modify the expected behaviour of of the functions.

There is nothing more deceptive than an obvious fact.