TP C#5 : Interrogation Room

Consignes de rendu

À la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

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

N'oubliez pas de vérifier les points suivants avant de rendre :

- Remplacez `sherlock.holmes` par votre propre login.
- Le fichier `README` est obligatoire.
- Pas de dossiers `bin` ou `obj` dans le projet.
- Respectez scrupuleusement les prototypes demandés.
- Retirez tous les tests de votre code.
- **Le code doit compiler !**

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

Au meurtre! Sherlock Holmes a une nouvelle fois besoin de votre aide pour résoudre un mystère. Il a fait une liste des témoins et suspects. Malheureusement, il est trop occupé sur un autre mystère et vous confie la tâche cruciale de trouver, interroger chacune de ces personnes et rassembler toutes ces informations pour lui. Pour agir efficacement, vous allez évidemment apprendre à manipuler des fichiers en C#.

1.1 Objectif

À la fin de ce TP, vous serez capable d'effectuer des actions sur des fichiers telles que les lire et écrire. On vous a déjà mentionné les flux standards précédemment (**stdin, stdout, stderr**) et ces notions seront vues plus en détails durant ce TP.

Durant votre lecture, vous verrez que certaines définitions sont en anglais, comme ***stream*** , en effet c'est généralement comme cela qu'elles sont utilisées, nous allons donc employer le terme de **stream** pour flux dans le reste du TP. Il sera fait de même avec d'autre vocabulaire tel que file, **input/output**...

ATTENTION !

Dans ce TP on va vous demander beaucoup d'affichage dans les différentes sorties. Attention à respecter à la virgule, la majuscule, l'espace près les syntaxes que l'on vous donne. Il est également important d'écrire sur les bons "streams", toute erreur écrite sur la sortie standard sera considérée comme invalide. Des points sont retirés quand les sorties ne sont pas exactement identiques !

2 Cours

Dans cette partie cours, les notions nécessaires pour compléter le TP vont vous être expliquées. Vous devrez aussi vous rendre sur MSDN, pour étudier le fonctionnement des différentes fonctions vu dans ce cours. Il est important d'avoir le réflexe de regarder la documentation avant de demander de l'aide, elle permet souvent de répondre à vos questions car elle est très complète et comprend de nombreux exemples.

2.1 Input/Output Stream

Il existe 3 streams standards: **stdout, stdin et stderr**.

Vous avez déjà appris à lire et écrire dans un terminal via votre programme. Lorsque vous affichez quelque chose sur la console avec **Console.Write()**, c'est sur le stream **stdout** que vous écrivez. De même, pour récupérer des informations données par l'utilisateur avec **Console.Read()**, vous lisez dans le stream **stdin**. De ce fait, vous savez déjà comment utiliser des streams.

Lorsque vous voulez écrire dans un fichier, un **input** stream est créé. À l'inverse, si vous voulez lire le contenu d'un fichier, c'est un **output stream** que vous allez devoir créer. Pensez aux streams comme des portes vous permettant d'accéder aux contenus d'un fichier. Certaines portes vous permettent que de lire dans un fichier (input stream) et d'autres vous autorisent d'écrire dedans (output stream).

Ainsi, quand on ouvre un fichier, ce qui se passe c'est que l'on crée un stream.

2.2 Path

Les fichiers et les dossiers sont stockés sous forme d'arbre. Un dossier peut avoir zéro ou plus fils qui sont des fichiers et des dossiers également. Ainsi, on appelle fils d'un dossier, tous les fichiers et dossiers contenus dans ce dossier. On appelle parent d'un fichier, le dossier dans lequel est contenu le fichier.

Il y a un dossier spécial, la racine. Tous les dossiers et fichiers du disque sont des descendants de ce dossier.

Par convention, on ajoute un '/' à la fin du nom des dossiers pour les différencier des fichiers.

Les chemins (ou path en Anglais) sont des chaînes de caractères qui permettent de se déplacer dans cette arborescence de fichiers et dossiers.

1. - "/" désigne la racine.
2. - "./" désigne le dossier actuel.
3. - "../" désigne le dossier parent.
4. - "***nom***" désigne le <nom> d'un fichier ou dossier.
5. - "../test" désigne, dans le dossier parent, le fichier ou dossier test.

Il existe deux types de chemins : relatif et absolu.

Un chemin relatif est défini par rapport à l'endroit où on se trouve actuellement dans l'arborescence. Par exemple, si on exécute du code dans un projet C, le dossier courant est la où se trouve l'exécutable.

Un chemin absolu est un chemin préfixé par / et qui part donc de la racine. Son avantage est qu'il ne dépend pas de l'endroit où l'on se trouve actuellement. Il est cependant souvent inconnu. Voici quelques exemples de correspondances entre les chemins absolus et relatifs :

```
Exemple: on est dans /tmp/tests/
../ => /tmp/    # Le parent de tests/ est tmp/
./../../ => /   # Le parent du parent de tests/ est la racine /
../../../ => /  # Le parent de la racine est la racine
```

Nous pouvons trouvez deux parties importantes dans un fichier (file en anglais). Des informations sur lui-même et son contenu.
La classe **FileInfo** permet d'obtenir d'obtenir et de modifier des informations sur un fichier (à vos risques et périls).
Par exemple:

1. Date de création
2. Longueur
3. Dossier parent
4. Chemin absolu
5. ...

De l'autre coté, la classe **File** permet quant à elle de manipuler le contenu du fichier. Il y a de nombreuses fonctions qui permettent de lire, écrire, créer ou supprimer un fichier. Regardons un exemple de code pour y voir plus clair :

```
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)**: Cette méthode renvoie une instance **FileStream**. Si le fichier n'existe pas il est créé, sinon l'ancien est écrasé.
— **File.Exists(path)**: Cette méthode renvoie un booléen, en fonction de l'existence du fichier path.
— **<object FileStream>.Close()**: Cette méthode permet de fermer une instance **FileStream**, fs dans notre exemple.
— **File.Delete(path)**: Supprime le fichier s'il existe.

Rappel

Il est important de comprendre qu'au moment d'ouvrir un fichier pour lire ou écrire, celui-ci devient un stream. Le retour de **File.Create()** est justement un **FileStream**.

2.3 StreamReader

Si vous êtes allé voir la documentation de la classe **FileStream**, vous avez pu voir qu'il est a la fois possible d'écrire et de lire. En réalité la classe **FileStream** est très complète et permet de faire beaucoup de choses. Pour ne pas s'y perdre nous allons plutôt utiliser les **StreamReader** et **StreamWriter**, qui ne permettent de ne faire qu'une chose a la fois : lire ou écrire.

```
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)** : Cette méthode nous permet de créer un **input** stream à partir d'un fichier. Attention, si le fichier n'existe pas la méthode renvoie une exception et votre programme s'arrête immédiatement.
— **<object StreamReader>.ReadLine()** : Cette méthode permet d'utiliser un **input** stream pour lire le contenu du fichier qui lui est lié, et ce ligne par ligne. Il n'est pas possible de lire la même ligne (A moins d'utiliser d'autre fonction faite pour) puisque la fonction s'occupe de placer un pointeur vers la prochaine ligne à lire. S'il n'y a plus rien à lire la fonction vous renvoie null.
— **<object StreamReader>.Read()** : Cette méthode est similaire a **ReadLine()**, mais plutôt que de lire une ligne, elle lit un caractère.
— **<object StreamReader>.ReadToEnd()** : Cette méthode lit tout en partant du pointeur jusqu'à la fin du fichier. Cependant, si le pointeur est situé à la fin du fichier (celui-ci a donc déjà été lu entièrement) la fonction ne renvoie pas **null** mais une **string** vide.
— **<object StreamReader>.Close())** : Cette méthode ferme le stream, il est important de ne pas oublier de fermer un stream une fois le traitement terminé.

Conseil

N'hésitez pas à aller voir la documentation pour voir toutes les méthodes associées à la classe StreamReader.

2.4 StreamWriter

Parlons maintenant des **StreamWriter**, comme le nom l'indique il permet d'écrire dans un fichiers via un stream. Regardons cet exemple :

```
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();
```

Ici, le path donné au **StreamWriter** est absolu, il commence par un /. Contrairement au **StreamReader**, si le fichier n'existe pas il sera créé !
— **new StreamWriter(path)** : Cette méthode nous permet de créer un **output** stream à partir d'un fichier. Attention, si le fichier n'existe pas il sera créé.
— **<object StreamWriter>.WriteLine()** : Cette méthode permet d'utiliser un **output** stream pour écrire dans le fichier qui lui est lié et ce ligne par ligne. Il n'est pas possible d'écrire plusieurs fois sur la même ligne pour les mêmes raison que pour la méthode **ReadLine()**.
— **<object StreamWriter>.Write()** : Cette méthode est similaire à WriteLine(), mais n'ajoute pas de newline a la fin.
— **<object StreamWriter>.Close())** : Ferme le stream, il est important de ne pas oublier de fermer un stream une fois le traitement terminé.

Ici, le contenu du fichier test serait le suivant :

```
awe don't sleep
cdcACDC
>ASM
```

Conseil

N'hésitez pas à aller voir la documentation pour vois toutes les méthodes associés à la classe **StreamWriter**.

2.5 Recap

Voyons un exemple d'utilisation de **StreamReader** et de **StreamWriter** pour s'assurer d'avoir tout bien compris :

```
1  string path = "../../../test";
2  if (!File.Exists(path))
3  {
4      //Create a file to write to.
5      using (StreamWriter sw = File.CreateText(path))
6      {
7      sw.WriteLine("Hello");
8      sw.WriteLine("And");
9      sw.WriteLine("Welcome");
10     }
11 }
12
13 // Open the file to read from.
14 StreamReader sr = File.OpenText(path);
15
16 string s;
17 while ((s = sr.ReadLine()) != null)
18 {
19     Console.WriteLine(s);
20 }
21 sr.Close();
```

Regardons un ce que fait chaque méthode ici :

— **File.Exists(path)** : Cette méthode renvoie un booléen, en fonction de l'existence du fichier **path**.
— **File.CreateText(path)** : Cette méthode renvoie un **StreamWriter**, si le fichier n'existe pas, la méthode va également le créer. Ce stream vous permet d'envoyer de l'information, en l'occurrence d'écrire dans le fichier.
— **StreamWriter.WriteLine("string")** : Cette méthode vous permet d'utiliser un flux pour écrire de l'information, ici c'est dans le fichier ouvert avec **File.CreateText(path)**. Noté la ressemblance avec **Console.WriteLine**, celle-ci était également un stream, mais vers la stdout et non un fichier.
— **File.OpenText(path)** : Cette méthode renvoie un **StreamReader**, qui nous permet de lire les informations du fichier ouvert.
— **StreamReader.ReadLine()** : Cette méthode nous permet de lire le contenu dans un fichier ligne par ligne. En effet, au premier appel à la fonction, la première ligne du fichier sera lue, au deuxième appel la deuxième ligne sera lue, etc. Une fois tout le fichier lu, la fonction renvoie null.
— **sr.Close()** : Cette méthode permet de fermer le stream.

Au final, si le fichier n'existe pas, il sera créé et contiendra :

```
1 Hello
2 And
3 Welcome
```

La console devrait afficher la même chose.

ATTENTION !

Il est primordial de fermer un stream après l'avoir utilisé, sinon vous risquez d'avoir des erreurs, notamment si vous essayez d'ouvrir à nouveau un stream à partir d'un fichier sans avoir fermé l'ancien !

Pourtant, pour le **StreamWriter** nous n'avons pas eu besoin de fermer celui-ci me diriezvous, mais pourquoi ? En réalité, c'est l'instruction **using** qui l'a fait pour nous, une fois sorti de la portée de celle-ci (en dehors des {}), notre stream est automatiquement fermé, une bonne méthode pour ne jamais oublier de fermer nos streams !

Pour plus d'informations sur les différentes fonctions vues ci-dessus, allez lire le doc, il y a beaucoup de variantes plus ou moins utiles en fonction de ce que vous souhaitez faire. Il est également important d'aller voir les erreurs que peuvent renvoyer ces fonctions.

2.6 Directory

De la même manière qu'un fichier, un répertoire (directory en Anglais) possède une partie information. La classe **DirectoryInfo** permet d'obtenir et modifier à peu prêt les mêmes informations qu'un fichier mais pour un dossier.

La classe **Directory** permet de manipuler les dossiers. Elle sert surtout à obtenir des informations sur son contenu (les fichiers et dossiers qu'il contient). Elle permet aussi de supprimer, créer et déplacer des dossiers.

Allez voir la doc des classes **Directory** et **DirectoryInfo** pour plus d'information sur les méthodes disponible dans celles-ci.

3 Basics

3.1 What am I ?

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

Cette fonction doit afficher sur la sortie standard si le **path** donné correspond à un fichier ou un dossier.

Si le fichier (ou le dossier) n'existe pas, l'exprimez également sur le 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);
```

Cette fonction doit afficher, sur la sortie standard, le contenu du fichier pointé par **path** ligne par ligne, prefixé par **"Line {i}:"** ou i est le numéro de la ligne comme dans l'exemple ci-dessous. Si le fichier n'existe pas, affichez sur la 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);
```

Cette fonction doit écrire le contenu du string **<content>** dans le fichier indiqué par **<path>**. Si le fichier indiqué par **<path>** n'existe pas, la fonction doit créer le fichier et ensuite écrire dedans. Si le fichier existe déjà, écrire à la fin du contenu préexistant.

```
$ ls
file1.txt file2.txt
$ cat file1.txt
$ cat file2.txt
La question est très
```

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

```
$ ls
file1.txt file2.txt file3.txt
$ cat file1.txt
Hello File !
$ cat file2.txt
La question est très
vite répondue
$ cat file3.txt
Is anyone here ?
```

3.4 Copycat or copy cat ?

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

Cette fonction doit copier le contenu dans le fichier indiqué par le **path source** et l'écrire dans le fichier indiqué par le **path dest**. Si le fichier source n'existe pas, renvoyez -1 sinon renvoyez 0. Comme précedemment, si le fichier dest n'existe pas, la fonction doit creer le fichier et ensuite ecrire dedans. Si le fichier existe déjà, ecrire à la fin du contenue préexistant.

```
$ 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);
```

Cette fonction est une étape intermédiaire pour la fonction suivante, elle est tout de même obligatoire.

Vous devez afficher à la manière de "ls" afficher tout les fichiers (et que les fichiers !) contenus dans le dossier path. L'ordre d'affichage n'a pas d'importance. Vous devez gérer de la même manière que dans DisplayFile si **<path>** n'existe pas.

```
$ 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);
```

Cette fonction doit afficher l'arborescence des fichiers/dossiers contenus dans le dossier **path**. Si un dossier est présent, afficher également le contenu à l'intérieur.

```
$ 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

Dans ces exercices vous allez devoir aider Sherlock Holmes à résoudre une enquête en créant un programme pouvant traiter les informations stockées dans différents fichiers. Il y aura 2 formats de fichier à manipuler: les fichier types `.profile` contenant divers informations sur un suspect et les `.question` qui eux contiennent une question et les réponses de différents suspects.

4.1.1 Information générale

Toutes les fonctions concernant l'exercice Interrogation sont à faire dans le projet Interrogation et dans le fichier `Interrogation.cs`, faite attention à bien respecter l'architecture demandée. De plus dans ces exercices vous pouvez considérer que si le fichier passer en paramètre existe alors il sera toujours du bon format.

4.2 La classe Profile

Dans ces exercices, toutes information obtenue dans les fichiers devront être sauvegardé dans une classe « Profile » comme ci-dessous:

```
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 questions
9      public List<string> Answer = new List<string>(); //list of answers
10     public Profile(string name)
11     {
12         //todo
13     }
14 }
```

Chaque champ de la classe correspond à une information pouvant être récupérée dans les fichiers `.profile` hormis pour les champs `Question`, `Answer` et `AlreadyAnswer` qui serviront à stocker les informations des fichiers `.question`.

Attention

Une instance de la classe est considéré valide si le champ `Name` n'est pas vide ou ne contient pas une chaine de caractères vide. Autrement dit tous les champs de la classe peuvent être vide à part celui de `Name` qui doit toujours avoir une valeur non `null`.

4.3 Les fichier .profile

L'un des fichier que vous devrez traiter sont les fichiers `.profile`. Ils respecteront toujours ce format : `NomDuChamps:Contenue`. Par exemple:

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

Attention

L'ordre d'apparition des champs n'est pas fixe. Certain champs peuvent ne pas être présent (sauf `Name`). D'autre peuvent être présent plusieurs fois.

4.4 Les fichier .question

L'autre type de fichier que vous devrez parser sont les fichiers `.question`. Ils respecterons toujours ce format : Question:La Question
NomDuSuspect1:Sa réponse
NomDuSuspect2:Sa réponse
...
Exemple:

```
[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)
```

Cette fonction vous demande de recupérer les informations stockées dans le fichier `path`, qui est un fichier sous le format `.profile`. Toutes les informations lues dans le fichier `path` doivent être stockées dans une instance de la classe `Profile` qui doit être retourner à la fin. Si un champ est présent plusieurs fois alors seule la dernière occurrence doit être prise en compte. Le fichier passé en paramètre sera toujours valide.

tip

La méthode `Split()` de la classe `String` peut vous être utile ! La méthode `Int.Parse()` peut également l'être !

4.6 ReadQuestion

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

Cette fonction prend en paramètre un string `path`, une liste d'instance `Profile allProfiles`. Cette fonction vous demande de parser le fichier `path` qui suit le format `.questions`. Si le nom d'une personne qui a répondu est aussi dans l'une des intances profile dans `allProfiles`. Alors vous devez mettre à jour ce profile en y ajoutant la réponse et la question respectivement dans les champs `Answer` et `Question`.

4.7 PrintInformation

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

Cette fonction vous demande d'afficher sur la console toutes les informations stockées dans la classe `Profile` passée en paramètre. Vous devez respecter le format de cet exemple:

```
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)
```

Cette fonction vous demande d'écrire dans un fichier les différentes informations contenues dans la classe profile passer en paramètre. Le nom du fichier doit être le nom contenu dans le champ `Name` et avec l'extension `.profile`. Si un fichier possède déjà ce nom, vous devez alors d'abord effacer son contenu, puis le remplacer par celui généré par la fonction. Vous devez respecter le format des fichier `.profile`.

4.9 CreateProfile

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

Cette fonction vous demande de coder une interface textuelle permettant à l'utilisateur de créer un `fichier .profile`. Vous devez lire les entrées de l'utilisateur tant qu'il n'écrit pas `exit`. La fonction demandera quel champs l'utilisateur veux modifier, puis ce qu'il veut écrire dedans. Si l'utilisateur écrit `exit` pendant la demande d'un champ, vous devez arrêter de lire dans le terminal. De plus si la classe est valide (possède au minimum un Name non vide), vous devez enregistrez ces données dans un fichier comme dans la fonction `SaveProfile`. Si la classe est invalide la fonction doit "ecrire Failed to create this new Profile".

Exemple:

```
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()
```

Pour cette fonction vous allez devoir créer une interface textuelle qui permettra d'utiliser les `5` précédentes fonctions. Pour cela la fonction devra lire des commandes envoyées par l'utilisateur. Il existe `6` commandes differentes:
`read profile` : qui va exécuter la fonction `ReadProfile`
`read Interrogation` : qui va exécuter la fonction `ReadQuestion`
`print profile` : qui va exécuter la fonction `PrintProfile`
`create profile` : qui va exécuter la fonction `CreateProfile`
`save profile` : qui va sauvegarder la fonction `SaveProfile`
`exit` : qui va arrêter la fonction
Vous devez créer une liste d'instances de la classe `Profile` pour pouvoir stocker les instances de classes `Profile` créer par `ReadProfile` ainsi que les mettre à jour par `ReadQuestion`.
Pour `PrintProfile` et `SaveProfile` l'utilisateur donnera le nom du `Profile` qu'il veut afficher ou sauvegarder. Si le nom correspond à aucun `profile`, vous devrez alors afficher sur le terminal "There is no profile Name that matches with NameEnter", avec `NameEnter` qui doit être remplacé par le nom entré par l'utilisateur.

Vous devez respecter ce format là :

```
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

Vous êtes libre d'implémenter autant de bonus que vous le souhaitez tant que ces derniers sont précisés dans le `README` et qu'ils ne modifient pas le comportement attendu des fonctions.

There is nothing more deceptive than an obvious fact.