TP C#6
My Explorer

Submission

Archive

Vous devez suivre l'architecture suivante:

```
rendu-tp-firstname.lastname.zip
    |-- rendu-tp-firstname.lastname/
        |-- AUTHORS
        |-- README
        |-- Crypto/
            |-- Crypto.sln
            |-- Crypto/
                |-- tout sauf bin/ and obj/
        |-- MyExplorer/
            |-- MyExplorer.sln
            |-- MyExplorer/
                |-- tout sauf bin/ and obj/
```

- Bien entendu, vous devez remplacer *prenom.nom* par votre nom.
- Votre code doit compiler, et ne doit contenir aucun warning.

AUTHORS

Ce fichier doit contenir Une astérisque (*), un espace et votre login.
Voici un exemple:

```
* prenom.nom
```

Notez bien que le nom du fichier est AUTHORS SANS extension.

README

Ce fichier doit contenir au moins les informations suivantes Où en êtes-vous dans l'avancée du tp, qu'est-ce que vous n'avez pas fait, **Quels bonii avez-vous fait**...
Notez bien que le nom du fichier est README SANS extension.

1 Introduction

1.1 Objectifs

Pendant ce TP, vous allez apprendre de nouvelles notions comme:

- Manipuler des fichiers
- Le mot clé "Using"
- Se déplacer à travers le file system

2 Cours

2.1 Manipuler des fichiers en C#

Jusqu'à aujourd'hui, vous avez toujours créé des programmes qui n'avaient pour but de n'être lancés qu'une seule fois (deux fois au moment où ils étaient corrigés), puis étaient misérablement oubliés à jamais, pauvres programmes, ils n'ont jamais eu la chance d'interagir avec le monde exterieur, de manipuler tous ces merveilleux fichiers sur votre disque dur (ou votre RAM dans le cas des PCs de l'école).
Ce TP est là pour changer tout cela: Apres cela vous pourrez enfin communiquer avec le reste du monde[1].

La partie facile: La classe `File`

La classe `File` [2] est une classe qui vous permet de manipuler les fichiers (étonnement). Son but principal est de manipuler les fichiers en eux-même, et non pas leur contenu. Voici un courte liste des méthodes les plus utilisées:

```
1 public static void Delete(string path);
2 public static FileStream Create(string path);
3 public static void Copy(string sourceFileName, string destFileName);
4 public static FileStream Open(string path, FileMode mode);
5 public static void Move(string sourceFileName, string destFileName);
```

Vous avez sûrement remarqué que la deuxième méthode, `File.Create`, renvoie un objet que vous ne connaissez pas : `FileStream`.
Le `FileStream` est la façon qu'un programme C# utilise pour représenter un fichier : Dans un programme, vous ne pouvez pas vraiment prendre un fichier, et le charger dans votre mémoire pour qu'il fasse partie du programme, et ensuite faire ce que vous voulez avec. Les fichiers ont des permissions qui limitent votre capacité à faire ce que vous voulez avec, en fonction de qui vous êtes. Et encore plus important, les fichiers ont une taille. Devoir charger entièrement un fichier d'1Go du disque en RAM est assez inimaginable tellement ce serait long. Si tous les programmes faisaient ca, rien que regarder un film prendrait toute la RAM de votre ordinateur.
Du coup, au lieu de charger complètement un fichier, C# utilise un `FileStream`, qui est un objet qui

[1] Limité à votre machine locale, il va falloir attendre pour toucher à internet!
[2] https://msdn.microsoft.com/en-us/library/system.io.file(v=vs.110).aspx

va interagir avec le fichier de la façon que vous voulez. Beaucoup de méthodes prennent en argument un `FileStream` pour savoir de quel fichier vous parlez.
Savoir tout cela est bien beau, mais vous n'allez pas aller très loin si vous ne savez pas écrire et lire dans un fichier. Pour cela vous avez deux choix :

- Vous pouvez utiliser `File.ReadAllLines` ou `File.ReadAllText`, sauvegarer le contenu dans une variable, faire ce que vous voulez avec, et ensuite réécrire dans le fichier en utilisant de la même façon `File.WriteAllLines` ou `File.WriteAllText`.
- Vous pouvez utiliser un `StreamReader` pour lire le fichier, et un `SreamWriter` pour écrire dedans.

La première méthode a l'air beaucoup plus simple, mais n'est pas très pratique lorsque l'on y réfléchit un peu : Vous êtes en train de sauvegarder dans une variable tout le contenu du fichier. Cette méthode peut être utilisée pour les fichiers très courts, ou pour les fichiers de configurations, mais c'est beaucoup moins acceptable pour les fichiers de tailles arbitraires. Comme vous l'avez deviné, vous allez utiliser la deuxième méthode pendant ce tp : Les `StreamReader` et `StreamWriter`

La partie fun: Les classes `StreamReader` et `StreamWriter`

La classe `StreamReader`[3] gère la lecture dans un fichier dans n'importe quel format (Cela lit octet par octet le fichier), mais ne vous inquiétez pas, c'est aussi parfaitement adapté pour lire dans un fichier texte, vu qu'un caractère est seulement un octet. Par exemple, considérons un dossier contenant l'exécutable du programme et un fichier `mytreasuredfile.gold` contenant seulement la ligne suivante :

```
This is either madness... or brilliance...
```

Voici une partie de code assez basique qui pourrait lire ce fichier:

```
// Nous créons un nouvel objet StreamReader appelé sr, qui va lire dans
// le fichier mytreasuredfile.gold
StreamReader sr = new StreamReader("mytreasuredfile.gold");
// Nous pouvons lire un caractère en utilisant sr.Read()
// vu que le fichier est lu dans l'ordre, du début à la fin.
// Ici, nous lisons le premier caractère
char character = (char) sr.Read();
// -> character == 'T'
// Nous pouvons lire toute la ligne d'un coup
// Vu que nous avons déjà lu le premier caractère, il est manquant
string line = sr.ReadLine();
// -> line == "his is either madness... or brilliance..."
// sr.EndOfStream renvoie true si nous avons tout lu, false si non.
// Vu que nous avons lu la seule ligne du fichier, sr.EndOfStream
// renverra true.
if (sr.EndOfStream)
  Console.WriteLine("The file has been read completely");
```

[3] https://msdn.microsoft.com/en-us/library/system.io.streamreader(v=vs.110).aspx

```
else
  Console.WriteLine("There is still something to read");
// N'oubliez pas de fermer le StreamReader !
sr.Close();
```

N'hésitez pas à demander à vos ACDCs si vous n'avez pas compris quelque chose au code juste au dessus, il est vraiment important que vous le compreniez. Il vous est aussi conseillé de vous renseigner sur les méthodes suivantes qui pourraient vous être utiles dans ce TP :
`Peek()`[4] et `Read(char[],int, int)`[5].

Comme vous avez parfaitement compris comment marchent les `StreamReader`, la classe `StreamWriter` ne devrait pas être très compliquée à comprendre. Elle fonctionne de la même manière que `StreamReader`. Voici un exemple:

```
// Nous créons un nouvel objet StreamWriter appelé sw, qui va lire dans
// le fichier stuff
StreamWriter sw = new StreamWriter("stuff");
// Nous pouvons écrire un caractère en utilisant sw.Write()
// Cette méthode fonctionne exactement comme Console.Write()
sw.Write('a');
sw.Write(14);
sw.Write(13.37);
// Nous pouvons aussi utiliser sw.WriteLine() pour écrire une ligne.
// Cette méthode fonctionne exactement comme Console.WriteLine();
sw.WriteLine("The code is the law");
sw.WriteLine("The answer is {0}", 42);
// N'oubliez pas de fermer le StreamWriter !
sw.Close();
```

Si nous regardons le fichier stuff après ce code :

```
a1413.37The code is the law
The answer is 42
```

Comme précédemment, vous devez avoir parfaitement compris cette notion avant de continuer à lire ce TP.

2.2 Utilise moi !

Vous avez peut-être remarqué que vous devez absolument fermer les streams après utilisation, mais pourquoi ? Tout ce que vous faites dans les streams n'est pas effectué immédiatement par votre ordinateur. Pour minimiser les chargements du disque vers la RAM, l'ordinateur va d'abord écrire dans un buffer, puis tout "flush" d'un coup, mais vous ne savez pas quand. Du coup, si votre programme

[4] https://msdn.microsoft.com/en-us/library/system.io.streamreader.peek(v=vs.110).aspx
[5] https://msdn.microsoft.com/en-us/library/9kstw824(v=vs.110).aspx

se finit sans avoir fermé proprement les streams, tout n'aura pas été écrit dans votre fichier. C'est pourquoi vous devez absolument appeler la méthode `Close()`.

Devoir écrire `Close()` après chaque utilisation peut-être assez ennuyant et vous pouvez facilement l'oublier. C'est pourquoi vous devez tout le temps utiliser `using`[6]. Le mot clé `using` va simplement dire au compilateur d'automatiquement "libérer" la variable à la fin du scope. Voici un exemple d'utilisation :

```
using (StreamWriter sw = new StreamWriter("file"))
{
  // Vous pouvez faire ce que vous voulez avec sw
  sw.WriteLine("Gentlemen!");
  //...
  // sw.Close() sera appelé automatiquement
  // en quittant le scope
}
// sw n'est plus disponible
```

2.3 Se balader dans le Filesystem

Etre capable de créer, lire et écrire des fichiers est cool, mais ce n'est pas assez. Vous êtes toujours limités à votre dossier courant : celui d'où vous avez lancé votre programme. Si nous voulons accéder aux autres dossiers, nous avons besoin de certaines méthodes, définies dans la classe `Directory`[7] :

```
public static string GetCurrentDirectory();
public static string[] GetDirectories(string path);
public static string[] GetFiles(string path);
public static string[] GetFileSystemEntries(string path);
public static void SetCurrentDirectory(string path);
```

Leur utilisation ne va pas être détaillée ici, vu que le nom de ces méthodes est déjà assez explicite. Si vous avez quand même des doutes, vous pouvez regarder MSDN, ou demander à vos assistants.

Notez bien que si vous souhaitez utiliser les méthodes évoquées dans toute la partie cours de ce TP, vous devez rajouter la ligne suivante en haut de votre fichier `Program.cs` :

```
using System.IO;
```

Maintenant que vous avez tout lu et tout parfaitement compris, vous pouvez commencer les exercices, bonne chance.

[6] https://msdn.microsoft.com/en-us/library/yh598w02.aspx
[7] https://msdn.microsoft.com/en-us/library/system.io.directory(v=vs.110).aspx

3 Exercice 1: Crypto

La cryptographie est un procédé rendant un message non compréhensible pour toute personne ne possédant pas les informations nécessaires au déchiffrage de ce message. La cryptographie est donc un procédé à double sens : un message encrypté peut être décrypté, à l'inverse des hash comme `md5` ou `SHA`. Les méthodes les plus basiques de crypto sont le **rotn**, et la méthode **Vigenère**. Rien de vraiment nouveau, ces méthodes ont déjà été faites en Caml, vous allez devoir les recoder en C#. Les deux fonctions décrites ci-dessous doivent être présentes dans le même fichier.

3.1 Rotn

La méthode **Rotn**, aussi appelé César, est une méthode qui décale tous les caractères d'un texte de n positions. La **rotn** que nous vous demandons de faire va seulement décaler les lettres et les chiffres d'une string passée en argument.

```
1 static string Rotn(string msg, int n);
2 //Rotn("abCz18", 3) -> "deFc41"
3 //Rotn("deF ;c41", -3) -> "abC ;z18"
4 //Rotn("ACaC1", 101) -> "XZaZ2"
```

Tips: Servez vous de l'opérateur modulo ('%') et de la table `ASCII`.

3.2 Vigenère

Le chiffre de **Vigenère** est aussi une méthode de chiffrement qui va substituer chaque lettre d'un texte en une autre lettre grâce à une clé donnée. La particularité de cette méthode est qu'une lettre ne va pas toujours donner la même lettre, en fonction de sa position dans le texte. Cette méthode nécessite la "table de Vigenère", mais pour vous épargner la recherche Wikipédia, nous vous précisons les formules d'encryptage et de décryptage. Le Vigenère que nous vous demandons de coder doit respecter les règles suivantes :

- La clé et le message sont de longueur variable et peuvent être constitués de n'importe quel caractère de la table ASCII.
- **Dans le message**, seuls les caractères étant des lettres seront chiffrés. Une majuscule donne une majuscule, une minuscule donne une minuscule. Les autres caractères seront passés, et la position dans la clé restera inchangée.
- **Dans la clé**, seuls les caractères étant des lettres serviront pour le chiffrement/déchiffrement. Les lettres minuscules seront considérées comme des majuscules (voir les exemples ci-dessous)
- `Chiffré[i] = (Texte[i] + Clé[j]) modulo 26`
- `Texte[i] = (Chiffré[i] - Clés[j]) modulo 26`

```
static string VigenereEncode(string msg, string key);
//VigenereEncode("AB xza ,;48ACk", "Un petit post-scriptum pour vouS")
//  -> "UO mdt ,;48IVz"

static string VigenereDecode(string msg, string key);
//VigenereDecode("AB xza ,;48ACk", "Quatoorze ;z")
//  -> "KH xgm ,;48MLl"
```

4 Exercice 2: MyExplorer

Cet exercice va mettre en application toute la partie cours vue au début de ce TP. Vous allez devoir lire des fichiers, les manipuler, et les réécrire. Le but sera d'avoir un programme interactif très basique qui va prendre en compte des commandes telles que : `ls`, `cd` ou `cat`, et de voir un peu comment parser des fichiers d'un certain format.

Note: Nous allons seulement tester vos fonctions avec des fichiers au format correct, sauf pour palier 4. Vous pouvez assumer que le texte que contient chaque fichier correspond à exactement ce qui est décrit dans les consignes du palier en question. De plus, pour tout cet exercice, aucune clé ou valeur ne peut contenir les caractères ':' ou ','. Des exemples de fichiers qui seront utilisés pour tester votre exercice sont donnés sur l'intranet `acdc.epita.it`. Pour tout l'exercice il vous est interdit d'utiliser la classe `RegExp`, cependant vous pouvez (et il vous est même fortement conseillé de) utiliser la classe `String`[8].

Pour tout l'exercice, si, pour une n'importe quelle raison, vous n'arrivez pas à ouvrir un fichier (par exemple parce qu'il n'existe pas), vous devez écrire sur le standard d'erreur `"Could not open file: "` suivi par le nom du fichier.

4.1 Palier -1 : MiniCat

Ce Palier est un palier d'introduction au `MyExplorer`. La fonction prend en argument le nom d'un fichier. Elle a pour but de l'ouvrir, de le lire, et ensuite d'écrire son contenu sur la console. Vous fournirez aussi une deuxième fonction ayant le même nom que la première mais qui, à la place d'écrire son contenu dans la console, l'écrira dans un fichier, si celui ci n'existe pas vous devez le créer, et si il existe déjà alors vous devez écrire à la suite du contenu de celui ci.

```
static void MiniCat(string InputFile);

static void MiniCat(string InputFile, string OutputFile);
```

4.2 Palier 0: GetValue

Maintenant que vous êtes familier avec la lecture dans un fichier, commençons le vrai exercice. Dans cet exercice, vous aurez pour but de récupérer (et potentiellement de les modifier) une valeur qui correspond à une clé. Pour ce palier, vous devez fournir la fonction `GetValue` qui prendra en argument

[8] https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx

le nom du fichier et le nom de la clé. Chaque ligne de ce fichier sera composé de la même façon : Une clé, suivie de ':', suivie d'une valeur, suivie du retour à la ligne. Ce fichier ne peut pas contenir de ligne vide. Il n'y a pas d'espace entre chaque élément.

```
1 static string GetValue(string InputFile, string key);
```

4.3 Palier 1.0: MiniLs

Pour valider ce palier, vous devez fournir la fonction `MiniLs`. Cette fonction écrira sur la console la liste des fichiers/dossiers dans le dossier courant.

```
1 static void MiniLs();
```

4.4 Palier 1.5: MiniCd

Pour valider ce palier, vous devez fournir la fonction `MiniCd`. Cette fonction prendra le nom d'un dossier en argument, et changera le dossier courant pour ce dossier. Si l'argument passé à cette fonction n'est pas un dossier, la fonction écrira sur la console un message d'erreur.

```
1 static void MiniCd(string FolderName);
```

4.5 Palier 1.8: Main function

Pour valider ce palier, vous allez devoir rendre votre programme utilisable. Votre fonction `Main` lira les entrées de l'utilisateur dans une boucle infinie, à l'image d'un vrai terminal (le gros carré noir, pour ceux qui ne savent toujours pas ...). L'utilisateur va donc pouvoir écrire `cat`, `ls`, `cd` ou `get` suivi d'arguments séparés par des espaces. Les arguments ne contiendront pas d'espaces ni de guillemets (simples ou doubles). Pour ce palier, nous supposons qu'il n'y a pas d'erreurs au niveau du nombre d'argument pour les trois commandes. La commande `ls` lance `MiniLs`. La commande `cd`, suivie d'un argument lance `MiniCd`. La commande `get` suivie de deux arguments lance `GetValue`.

4.6 Palier 2: Ajoutons quelques espaces

Pour valider ce palier, vous devez gérer les fichiers contenant des lignes contenant des espaces avant et après chaque élément dans la fonction `GetValue`. Regardez le fichier donné sur l'intra pour voir exactement où les espaces peuvent se situer. Les paliers précédents doivent toujours être fonctionnels après avoir rajouté cette amélioration.

4.7 Palier 3: Plusieurs valeurs par clé

Pour valider ce palier, vous devez coder la fonction `GetValue` qui possèdera un argument en plus, qui est la position de la valeur demandée. Les fichiers donnés à cette fonction auront toujours une seule clé par ligne, mais il peut y avoir plusieurs valeurs par clé, séparée par des virgules. La première valeur a pour position 0.
La fonction `Main` doit aussi gérer `get` avec 3 arguments.

```
1 static string GetValue(string InputFile, string key, int position);
```

4.8 palier 4: Gestion d'erreur

Pour valider ce palier, vos deux fonctions `GetValue` doivent écrire sur la console un message d'erreur lorsqu'il y a une erreur évidente dans le fichier. Les caractères interdits restent interdits, vous ne devez pas gérer ces erreurs. Une erreur doit être renvoyée quand :

- La clé n'est pas présente.
- Il n'y a aucune valeur après la clé demandée.
- Vous essayez d'accéder à une valeur qui n'existe pas (par exemple la clé a 3 valeurs et vous essayez d'accéder à la valeur d'index 3)

4.9 Palier 5: Le changement c'est maintenant

Pour le dernier palier, vous devez pouvoir changer une valeur associée à une clé dans un fichier d'input. La fonction `ChangeValue(InputFile, "ACDC", 'Bad", "Good")`, appliquée à un fichier contenant la ligne "ACDC : Acdc, are, Bad, guys", changera "Bad" en "Good". Le nombre d'espaces avant et après doit être le même que dans la ligne originale, seule `OldValue` est changé.

```
1 static void ChangeValue(string InputFile, string key,
2   string OldValue, string NewValue);
3 //OldValue et NewValue ne démarrent ni ne finissent par des espaces.
```

La fonction `Main` doit pouvoir lancer cette fonction avec la commande `change`.

5 Bonii

Voici une liste des bonii conseillés pour ce TP. Il est conseillé de la faire dans l'ordre. Vu que les bonii reposent sur l'ensemble du tp, il faut que vous ayez fini entièrement le tp pour pouvoir passer aux bonii. N'oubliez pas de préciser ce que vous avez fait dans le README.

- Gérer les cas de mauvais nombre d'arguments dans la fonction `Main`.
- Faire la fonction `MiniCatCrypto` : Cette fonction va lire dans le fichier donné en argument, regarder la première ligne et agir en conséquence. Si le premier mot est "ROTN", elle va rotate tout le fichier (première ligne exclue) grâce au nombre donné dans le deuxième mot, et écrire le résultat dans la console. Si le deuxième mot n'est pas un nombre vous renverrez une erreur. Si le premier mot est "VIGENERE", vous **déchiffrerez** le fichier grâce à la clé constituée par tout le reste de la première ligne. Si le premier mot est autre chose, vous aurez le comportement normal de `MiniCat`.

```
1 static void MiniCatCrypto(string InputFile);
2 // Première ligne de InputFile:
3 //   "ROTN 4z" -> erreur
4 //   "ROTN 42 4" -> rotn(42) sur tout le fichier sauf première ligne
5 //   "VIGENERE Slam du rible"
6 //      -> vigenere avec comme clé "Slam du rible"
7 //   "ACDC are the best" -> MiniCat(InputFile);
```

- Refaire ce tp sans utiliser les méthodes de la classe `String`.
- Refaire ce tp sans utiliser le type `string` (hint : Tableau de char) (hint2 : Have fun).

The code is the law.