TP C#6 : Tiny 42sh

Consignes de rendu

A la fin de ce TP, votre rendu devra respecter l'architecture suivante :

```
|-- csharp-tp6-{login}/
    |-- AUTHORS
    |-- README
    |-- Tiny42sh.sln
    |-- Tiny42sh
        |-- Tout sauf bin/ et obj/
```

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

- Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
- 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 !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣prenom.nom$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

```
echo "* prenom.nom" > AUTHORS
```

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

Ce TP a pour objectif d'aller un peu plus loin dans la manipulation de fichiers. En effet, en plus de pouvoir lire ou écrire des fichiers, vous allez pouvoir les effacer, créer des dossiers, changer de répertoire... Ce TP constitue également une mini-initiation à la ligne de commande Linux (que vous utiliserez en S3 et pour le reste du cursus à EPITA). Vous allez donc devoir coder un interpréteur de commande similaire à celui de Linux qui va permettre de réaliser les différentes actions décrites ci-dessus.

2 Cours

Avant de commencer, nous allons vous résumer le fonctionnement de la ligne de commande Linux ainsi que le comportement des commandes que vous aurez à implémenter. Voici un exemple de ce à quoi ressemblera ce TP :

```
1  Tiny42sh $ ls
2  LoveACDC/
3  foo
4  bar
5  CorrectionTPCS11/
6  Tiny42sh $ ls CorrectionTPCS11
7  Execution.cs
8  Interpreter.cs
9  Program.cs
10 Tiny42sh $ cd CorrectionTPCS11
11 Tiny42sh $ cat Execution.cs
12 cat: Did you really think it could be so easy ?
```

2.1 Format des commandes

Afin d'appeler une commande sous Linux, il suffit de taper le nom de la commande et de taper "Entrée". Les commandes peuvent prendre des arguments, ces arguments sont placés les uns à la suite des autres après le nom de la commande et séparés par des espaces.

2.2 Erreurs

Lorsqu'une erreur se produit sur une commande ou que la commande n'est pas reconnue, votre interpréteur doit afficher un message d'erreur pertinent.

2.3 Valeurs de retour

Si une erreur se produit, cette même commande doit renvoyer un entier différent de 0. Sinon elle devra renvoyer 0.

2.4 Prompt

Le prompt est le message qui est affiché avant que l'interpréteur demande une entrée utilisateur. Cela signifie qu'à chaque fois que votre interpréteur attend une commande, il faudra afficher un prompt. Dans notre cas le prompt sera "Tiny42sh $ ".

2.5 Commandes

Dans le nom des commandes, [args] signifie un seul et unique argument. [args]+ signifie 1 ou plusieurs arguments. [args]* signifie O ou plusieurs arguments.

2.5.1 ls [args]*

`ls` est une commande qui permet d'afficher le contenu d'un dossier. Elle peut prendre autant d'arguments que l'on veut. Si 0 arguments sont envoyés à `ls`, elle affiche le contenu du dossier courant. Si un des arguments n'est pas un dossier, `ls` affiche simplement le nom du fichier. On considère qu'une erreur se produit lorsque l'un des arguments ne correspond à aucun dossier ou fichier.

2.5.2 cd [args]

La commande `cd` permet de se déplacer dans le système de fichier. C'est-à-dire que cette commande change le dossier dans lequel on se trouve actuellement. Ainsi, `cd` prend exactement un seul argument : le dossier dans lequel on veut aller. Une erreur se produit lorsque le nombre d'argument diffère de 1 ou que l'argument ne correspond à aucun dossier.

2.5.3 cat [args]+

`cat` permet simplement d'afficher le contenu d'un fichier sur la console. Cette commande prend au moins un argument. Chaque argument représente le fichier que l'on souhaite afficher. Une erreur se produit lorsqu'aucun argument n'a été spécifié ou qu'un des arguments est un dossier.

2.5.4 touch [args]+

`touch` créé un fichier si celui-ci n'existe pas. Si le fichier existe déjà, `touch` met à jour sa date de dernier accès (LastAccessTime). Une erreur se produit si aucun argument n'a été passé.

2.5.5 rm [args]+

La commande `rm` supprime purement et simplement un fichier. Elle doit prendre au moins un argument. Une erreur se produit si aucun argument n'est spécifié, si l'argument est un dossier ou si le fichier n'existe pas.

2.5.6 mkdir [args]*

Cette commande permet de créer un dossier. Elle prend au moins un arguments. Une erreur se produit si aucun argument n'a été passé ou si le dossier existe déjà

2.5.7 pwd

`pwd` affiche le chemin du dossier courant. Elle ne prend aucun argument. Une erreur se produit si au moins argument est passé à la commande.

2.5.8 clear

La commande `clear` efface tout le contenu de la console. Elle ne nécessite aucun argument. Une erreur se produit si des arguments sont passés.

3 Exercice

Ce TP nécessitera de faire appel à différentes classes de C#. Vous devriez donc vous renseigner sur ces classes :

- `Directory` : toutes les méthodes nécessaires à la manipulation de dossier
- `File` : toutes les méthodes nécessaires à la manipulation de fichier
- `StreamReader` : permet de lire les fichiers.

Votre TP sera divisé en 3 fichiers :

- Program.cs
- Execution.cs
- Interpreter.cs

3.1 Program.cs

Ce fichier servira à appeler toutes les méthodes des classes `Interpreter` et `Execution`. Entre autre, la méthode `Main` devra boucler à l'infini, afficher le prompt, récupérer l'entrée utilisateur depuis la classe `Interpreter`, et l'exécuter avec la classe `Execution`. Implémentez la méthode `Main` au fur et à mesure du TP. Contentez vous pour le moment d'afficher le prompt.

```
1 static void Main(string[] args)
2 {
3     /* Contenu de la méthode Main */
4 }
```

3.2 Interpreter.cs

3.2.1 `readline`

La première méthode à implémenter se contentera de récupérer l'entrée de la console sous forme de string et la renverra.

```
1 static public string readline()
2 {
3     /* Attend une entrée untilisateur et la renvoie */
4 }
```

3.2.2 `parse_input`

Cette méthode prend en paramètre la commande entrée par l'utilisateur. La commande devra être découpée selon le caractère espace. Le découpage résultant sera stocké dans un tableau de string et sera retourné.

```
1 static public string[] parse_input(string input)
2 {
3     /* Découpe la string input en un tableau de string */
4 }
```

3.3 `enum` Keyword

Afin que la classe `Execution` reconnaisse plus facilement les commandes, vous allez créer une énumération qui contiendra toutes les commandes à implémenter ainsi qu'une valeur spéciale qui servira à dire qu'une commande n'existe pas. Cette énumération devra être déclarée dans le fichier Interpreter.cs (mais en dehors de la classe `Interpreter`).

```
enum Keyword
{
    /* Toutes les commandes et la valeur NOT_A_KEYWORD */
}
```

3.3.1 `is_keyword`

Cette méthode renverra la valeur de l'énumération `Keyword` correspondant à la commande entrée par l'utilisateur. Il s'agit tout simplement d'une bête suite de conditions comparant l'entrée avec la liste des commandes.

```
static public Keyword is_keyword(string word)
{
    /* Comparaisons avec toutes les commandes disponibles */
}
```

3.4 Execution.cs

Cette classe sera chargée d'exécuter la commande que la classe `Interpreter` aura découpée.

3.4.1 `execute_command`

Cette méthode prend en paramètre la commande pré-découpée par l'interpréteur et exécutera l'action correspondante. Il faudra utiliser la méthode `is_keyword` de la classe `Interpreter` afin de reconnaître la commande.

```
static private int execute_command(string[] cmd)
{
    /* Exécute la commande pré-découpée */
}
```

3.4.2 Exécution des commandes

Vous allez maintenant implémenter toutes les méthodes permettant d'exécuter la commande entrée par l'utilisateur. Servez-vous du résumé des commandes de la partie cours afin de connaître le fonctionnement des commandes et les implémenter.

```
static private int execute_ls(string[] cmd)
static private int execute_cd(string[] cmd)
static private int execute_cat(string[] cmd)
static private int execute_touch(string[] cmd)
static private int execute_rm(string[] cmd)
static private int execute_mkdir(string[] cmd)
static private int execute_pwd(string[] cmd)
static private int execute_clear(string[] cmd)
```

Rappel

Dans ce TP, toutes les méthodes à implémenter sont marquées comme static. Cela signifie qu'elles peuvent être appelées sans avoir instancié au préalable la classe à laquelle elles sont associées. Ainsi, vous ne devriez pas avoir besoin de créer un constructeur pour Execution ou Interpreter. Pour appeler une méthode static il suffit d'écrire

```
1 NomDeLaClasse.NomDeLaMéthode(...);
```

3.5 Méthodes recommandées

Voici une liste des méthodes de C# que nous vous recommandons afin d'implémenter toutes les commandes.

```
1 Directory.Exists(string)
2 File.Exists(string)
3 Directory.EnumerateFileSystemEntries(string)
4 Directory.GetCurrentDirectory()
5 Directory.SetCurrentDirectory(string)
6 Directory.SetLastAccessTime(DateTime)
7 File.Delete(string)
```

3.6 Bonus

De nombreux bonus peuvent être faits sur ce TP :
— Personnalisation du prompt
— Ajout de couleurs
— Implémentation d'autres commandes
— Exécution de plusieurs commandes à la suite séparées par ";"
— Autocomplétion en appuyant sur Tab
— Exécution d'un script (suite de commandes contenues dans un fichier)
— ...

If you knew time as well as I do, you wouldn't talk about wasting it