Practical C#10 :

1 Guidelines

At the end of this tutorial, you will submit an archive that respects the following architecture :

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- tiny42sh.sln*
    |-- tiny42sh*
        |-- Everything but bin/ and obj/
```

Replace "`login_x`" by your own login. All the files marked with an asterisk are *MANDATORY*.

Verify the following before submitting your work :

— The `AUTHORS` file must be in the usual format (a *, a space, your login and a line feed).
— No `bin` or `obj` directory in your project.
— **The code must compile !**

2 Introduction

This year you had the chance to work with a powerful IDE such as Visual Studio on Windows.
You could simply navigate from window to window, click on buttons, etc. Well, you should know that starting next year, you will leave Windows for good and only work on a Linux distribution in order to taste the wonderful world of the command line.

The flagship project of ING1 is the 42SH, a team project that spans several weeks. It will consist in recoding your entirely own shell in C.
You are obviously still far from starting, but we think that you will enjoy getting a glimpse ahead of time.

3 Course

You have already seen most of the functions we will use in this tutorial. The only new thing is the use of enumerations. For the rest, you'll have to use **Directory**, **File** and **StreamReader** classes.

3.1 Enumerates

This type is very simple to use and quite useful.
Imagine you have a variable named **Days**. Logically, it should receive as values only the days of the week : Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday (and a default value like None for any other value).

This kind of type is both practical and more efficient :
- Convenient because we do not have to test the values all the time to check if they are valid (if it's a day of the week or not)

- Efficient because, as **Days** can contain only a finite number of constant values, an enum type can give a numerical value to each possible value. And despite the great abstraction that brings you C#, you will agree that a comparison between two strings is significantly more expensive than a comparison between two ints.

Thus, in our case, to define the enum **Days** we would do as follow :

```
enum Days { Monday, Tuesday, Wednesday, Thursday,
            Friday, Saturday, Sunday };
```

Then, to use it you would just do :

```
Days today = Days.Friday;

if (today == Days.Friday)
   Console.WriteLine("It's Friday! Friday!");
```

As already said, the enum gives a numerical value to each possible value, the first enumerator with the default value 0, and the next one with the value n+1.
If this does not suit you, you can give your own values :

```
enum Days {Monday=5, Tuesday, [...] };
```

Mondays will have the value 5, Tuesday will be 6, etc.

4 My Tiny 42sh

4.1 The Interpretor

Your tiny42sh will be split in 3 static classes :
— **Interpreter** will read the user's input
— **Execution** will execute the different commands
— **Program** will call the two previous classes

As a reminder, a static class will allow you to call its methods directly :

```
MyClass.themethod()
```

without having to instanciate it beforehand

```
MyClass AnInstance = new MyClass();
AnInstance.themethod();
```

You will implement the method **parse_intput** which takes the user input.
Knowing that the **semicolon allows you to chain multiple commands**, start by splitting the input wherever there is a semicolon and then cut it by the spaces in order to split the tokens' command.

Therefore, you will return a matrix whose first dimension will be the command to recover, and the 2nd dimension the token of this command.

```
static public string[][] parse_input(string input)
```

is_keyword will take a token as argument and returns the value of the correct enum Keyword.

```
static public Keyword is_keyword(string word)
```

This means that you will have to define your **enum Keyword**, just above the class, with all the commands you will code (at least ten for the mandatory part).

Consider adding a default value for any unmatching token.

4.2 The Execution

For all these methods, you must return 1 if there was an error, 0 if everything went well. Always send back the result of the last action. For instance, by running :

```
ls existingFolder inexistentFolder
```

you will send the number resulting from the action on the last parameter (here it is 1 and if we reverse the arguments it will be 0).
Remember to inquire about the **File, Directory, StreamReader** classes (if you have already forgotten them) and, of course, about these classic Unix commands to reimplement.

For the examples, we will follow this structure :

```
folder/
    file2
    file3
file1
```

4.2.1 Execute_command

Execute_command takes as argument an array of tokens (the command)
You will call the Interpreter to check that the first token is valid and get its Keyword
According to the Keyword, you will call the corresponding function that you will see below.

```
static private int execute_command(string[] cmd)
```

4.2.2 Execute_input

Execute_input will take as argument the matrix returned by the Interpreter.
You will execute each command it contains with **execute_command** and will return the result value of the last executed command.

```
static public int execute_input(string[][] input)
```

4.2.3 ls

Execute_ls Handles the arguments and calls **show_content** on each one of them. If there are no arguments, show the current directory's content
show_content Shows the content of the given folder. If it's a file, shows its name and if the folder/file doesn't exist, show the given error message.

```
static private int show_content(string entry)
static private int execute_ls(string[] cmd)

tiny42sh$ ls
folder/
file1

tiny42sh$ ls file1 folder
file1
file2
file3

tiny42sh$ ls folder/file2 blabla
folder/file2
ls: blabla: No such file or directory
```

4.2.4 cd

cd move through directories.

```
static private int execute_cd(string[] cmd)

tiny42sh$ cd;cd folder folder // We want exactly 1 argument
cd: Invalid number of argument
cd: Invalid number of argument

tiny42sh$ cd file1
cd: file1: Is not a directory

tiny42sh$ cd blabla
cd: blabla: No such file or directory

tiny42sh$ cd folder // Move into the folder and shows nothing
```

4.2.5 cat

cat Shows the file content.

```
static private int execute_cat(string[] cmd)

tiny42sh$ cat
cat: Invalid number of arguments

tiny42sh$ cat folder
cat: folder: Is not a file

tiny42sh$ cat blabla
cat: blabla: No such file or directory
```

```
tiny42sh$ cat file1 folder/file2
I'm file1 content!
I'm file2 content!
```

4.2.6 touch

touch Creates a new file if it doesn't exist.
If it exists or if it is a folder, updates its **last access time**.

```
static private int execute_touch(string[] cmd)

tiny42sh$ touch
touch: Invalid number of arguments

tiny42sh$ touch file4
tiny42sh$ ls
folder/
file1
file4

tiny42sh$ touch file1 folder
// Updates the last access time of file1 and folder
```

4.2.7 rm

rm Deletes a file.

```
static private int execute_rm(string[] cmd)

tiny42sh$ rm
rm: Invalid number of arguments

tiny42sh$ rm folder
rm: folder is a directory

tiny42sh$ rm blabla
rm: blabla: No such file or directory

tiny42sh$ rm file4
tiny42sh$ ls
folder/
file1
```

4.2.8 rmdir

rmdir Deletes empty folders.

```
static private int execute_rmdir(string[] cmd)

tiny42sh$ rmdir
rmdir: Invalid number of arguments

tiny42sh$ rmdir folder
rmdir: folder: is not an empty directory
```

```
tiny42sh$ rmdir file1
rmdir: file1: is not a directory

tiny42sh$ rmdir blabla
rm: blabla: No such file or directory
```

4.2.9 mkdir

mkdir Creates a new folder.

```
static private int execute_mkdir(string[] cmd)

tiny42sh$ mkdir folder2;ls
folder/
folder2/
file1

tiny42sh$ mkdir;mkdir folder2;mkdir file1
mkdir: invalid number of arguments
mkdir: folder2: Directory already exists
mkdir: file1: File already exists
```

4.2.10 pwd

pwd Shows current path.

```
static private int execute_pwd(string[] cmd)

tiny42sh$ pwd blabla
pwd: Invalid number of arguments

tiny42sh$ pwd
C:\Users\ [...] \bin\Debug
```

4.2.11 clear

```
// Clears the console
static private int execute_clear(string[] cmd)
```

4.2.12 whoami

```
// Who are you?
static private int execute_whoami(string[] cmd)

tiny42sh$ whoami
ADC 2017 -- MeltedPenguin && Eliams
```

4.2.13 unkwown command

Default action if the input is not recognized.

```
static private int execute_unknown_cmd(string[] cmd)

tiny42sh$ launch Windows Vista
launch: Unknown command
```

4.3 Program

In the **Main** you will only have to do an infinite loop in which you will print.

```
tiny42sh$
```

If the user input is empty, just go to the next loop. Otherwise parse the input with the Interpreter and send the result as argument to the right method in the Execution class.

The Bullshit is strong with this one...