

Lab: Manual String Processing

Implementing the command interpreter of our Bash

Now is probably the key moment for the application we are building, because our app is a stack of different functionality, that is coupled to the class with the Main method and to be more specific to the commands we have written there. However, our application has no predefined order of the commands and the main idea is that it should be able to interpret such at runtime. So now our job is to **build an interpreter that calls the functionality** that we have.

We are going to need **two public static classes** that **handle the input** and the **commands**. The **first one** is called **InputReader** and the **second one** is called **CommandInterpreter**.

Now that you have these classes we are going to have to write some code in order for them to do their jobs.


Problem 1. Implement InputReader class

First we are going to start with the **InputReader** because it **uses the command interpreter to do some of its job**.

The only method for now will be a method that will be called from the main that starts to **listen for commands** and **executes them if the syntax is correct**. We will name this **method StartReadingCommands()** and its return type will be void.

```
public static void StartReadingCommands()
{
}
}
```

You've probably opened the Command Prompt and you've seen that you do not write your commands on empty lines, instead the folder that you are currently in is in the beginning of the line.



So in order to add this functionality and our bash to be able to **look like the command prompt**, we will **write a message on the OutputWriter** and our message will be the **current path from the SessionData class followed by '>'**.

Now it's time to read an input and trim it from all white spaces.

```
OutputWriter.WriteMessage($"{SessionData.currentPath}> ");
string input = Console.ReadLine();
input = input.Trim();
```

However, we want to **continue reading once we've interpreted the current command**, so maybe here will be a good time to **add a while loop** and **read a new input at the end of the loop**. Note that we

repeat the **code above** in our **while loop**, however we **do** the **first read out of the loop**, because even the first command can be the command for terminating the BashSoft.

```
while (true)
{
    //Interpret command
    OutputWriter.WriteMessage($"{SessionData.currentPath}> ");
    input = Console.ReadLine();
    input = input.Trim();
}
```

Now we have only two things left to do in the while loop, to finish with its implementation. Firstly, we should **set** some **condition** for which the **while loop** has to be true. A good way of doing this is to **make** a **constant** for the **command for termination** (which is "quit") and **then check** in the **condition** of the **loop** whether the input is different from the termination command.

The declaration of a constant looks like this:

```
private const string endCommand = "quit";
```

and it is **private**, because **we do not want other classes** to be able to **see it and use it**. Your task now it to **make** the **check between** the **end command** and the **input**.

Problem 2. Interpreting commands

Once you've done that it is time we move on to the **interpreting** of the **command**, but in order to change the comment with code, we have to **write** the **functionality for interpreting a command**. This functionality is somewhat a **different task** from reading input and for this reason we will **use another class** you've already made and **write** the **method that interprets a command**.

It can be called exactly as its purpose and its declaration should be similar to this:

```
public static void InterpretCommand(string input)
```

However, in order **to write** an **implementation** for this **method** we need to **know all** the **commands** that our **interpreter can do**.

The declaration of a command will be given in the following format:

Description of the command – actual command and possible parameters

Here is a list of all of them:

Commands list:

- mkdir directoryName – make directory in current directory
- ls (depth) – traverse current directory in given depth
- cmp absolutePath1 absolutePath2 – comparing two files by given two absolute paths
- changeDirRel relativePath – change current directory by a relative path
- changeDirAbs absolutePath – change current directory by an absolute path
- readDb dataBaseFileName – read students data base by given the name of the data base file which is searched in the current folder

- filter courseName poor/average/excellent take 2/10/42/all – filter students from some course by given filter and add quantity for the number of students to take, or all, if you want to take all the students matching the current filter
- order courseName ascending/descending take 3/26/52/all – order student from given course by ascending or descending order and then taking some quantity of the filter, or all that match it
- download (path of file) – download file
- downloadAsynch: (path of file) – download file asynchronously
- help – get help
- open – opens a file

An easy approach is to **check if** the **input command** corresponds to the ones given in the **commands set**. And **if** the given command **exists**, to **check** for the **input parameters**. The main check you may have to make for the input parameters in each command is whether the **number of commands corresponds to** the given **number of parameters**. So you'll probably need this piece of code in each method for calling a given operation (data is all the commands given on the current line, split by a space):

```
if (data.Length == )
{
    //Add code here after the check for
    params
}
else
{
    DisplayInvalidCommandMessage(input);
}
```

An approach to **checking whether** the **command is** one of the **possible** can be achieved if we **split** the **input** by a **space** and **check** the **element** with index **0** in a **switch-case** and **if** it **enters** one of the cases, we **call** the **corresponding method**, that **executes** the **wanted command**. **If no command matches** the input, then the default is a method that **displays** a **message** for an **invalid command**.

InterpretCommand method should look something like this:

```

public static void InterpretCommand(string input)
{
    string[] data = input.Split(' ');
    string command = data[0];
    switch (command)
    {
        case "open":
            TryOpenFile(input, data);
            break;
        case "mkdir":
            TryCreateDirectory(input, data);
            break;
        case "ls":
            TryTraverseFolders(input, data);
            break;
        case "cmp":
            TryCompareFiles(input, data);
            break;
        case "cdRel":
            TryChangePathRelatively(input, data);
            break;
        case "cdAbs":
            TryChangePathAbsolute(input, data);
            break;
        case "readDb":
            TryReadDatabaseFromFile(input, data);
            break;
        case "help":
            TryGetHelp(input, data);
            break;
        case "filter":
            //TODO: implement after functionality is implemented
            break;
        case "order":
            //TODO: implement after functionality is implemented
            break;
        case "decOrder":
            //TODO: implement after functionality is implemented
            break;
        case "download":
            //TODO: implement after functionality is implemented
            break;
        case "downloadAsynch":
            //TODO: implement after functionality is implemented
            break;
        default:
            DisplayInvalidCommandMessage(input);
            break;
    }
}

```

Here in the cases we have a lot of methods that we call which are not yet known and we haven't talked about them, however almost **every single one** of them **contains** the **check for the number of parameters**. First we are going to **look at the implementation of the method that displays an invalid command message**. Actually the only thing that we do in this **function** is to **display an exception** in the **following format**: **"The command '{input}' is invalid"** (*Display exception using the **OutputWriter***). We are going to use this method a lot if something with the commands or parameters is not ok, because it is a good way of notifying the user that something is wrong.

Now we have to look at the implementations of the other methods and implement them in the order they were given above.

1. **Open file** – all we need here is to know the name of the file that we have to open, and then we use the **current path** from the **Session Data** to **generate the absolute path** of the **file**. The **length of the data must be 2** elements. Finally, we need to know how to open files with their

default program, using C# and this is done using the following code:

```
string fileName = data[1];  
Process.Start(SessionData.currentPath + "\\\" + fileName);
```

2. **Make directory** – for making a directory again we need to check if the length of the data array is 2 and then take the folder name and create such a folder using the functionality in the **UIManager**:

```
string folderName = data[1];  
IOManager.CreateDirectoryInCurrentFolder(folderName);
```

3. **Traverse current folder** – here can have no parameters (only **ls**, which displays the files and subfolders in the current folder), or just one parameter (the depth to go in [**ls 4**]). If the number of elements in the data array is 1, we call the **TraverseDirectory** from the **IOManager** with depth of 0 and if the elements are 2, then the second element should be the depth and we try to parse it and in case of success use it to pass it to the method for traversal. **If the parameter can't be parsed**, we print an exception message on the output writer using its method **DisplayException**. We should first add the exception we talked about to the **ExceptionsMessages** class with the name **UnableToParseNumber** and a message: "The sequence you've written is not a valid number." The code inside the check for whether the elements are two looks something like this:

```
else if (data.Length == 2)  
{  
    int depth;  
    bool hasParsed = int.TryParse(data[1], out depth);  
    if (hasParsed)  
    {  
        IOManager.TraverseDirectory(depth);  
    }  
    else  
    {  
        OutputWriter.DisplayException(ExceptionMessages.UnableToParseNumber);  
    }  
}
```

4. **Compare content of two files** – if the input corresponds to this command, two parameters are expected, **which are the absolute path of the first and the absolute path of the second file** and if there are any mismatches, a new log file is created in the same folder as the second file path. The way we compare two files is already implemented in the **Tester** class, so we just need to call

it if all conditions are true:

```
if (data.Length == 3)
{
    string firstPath = data[1];
    string secondPath = data[2];

    Tester.CompareContent(firstPath, secondPath);
}
```

5. **Change directory relative** – here the path given should be appended to the current path in the **SessionData** and then it is passed to the **change directory absolute**, because an actual absolute path is generated, but we have all of this implemented in the **IOManager** so we are going to use it to change the current directory by a relative path...

```
string relPath = data[1];
IOManager.ChangeCurrentDirectoryRelative(relPath);
```

6. **Change directory absolute** – the approach now is pretty much the same as in the previous command.

```
string absolutePath = data[1];
IOManager.ChangeCurrentDirectoryAbsolute(absolutePath);
```

7. **Read database** – the parameter needed here for the initialization of the database is a file name from which to read the database of SoftUni. Note that only the name is wanted, which should mean that the file will be searched in the current folder. So maybe we can use the **StudentRepository** and make a few changes so that our new input comes from a file and not from the console.

First thing you might want to add is a parameter for the public method **InitializeData()** from the student repo so it should look something like this :

```
public static void InitializeData(string fileName)
```

However **InitializeData** is just a **façade** for the **method that does the actual reading** of the data, so we need to **add the same parameter in this method and then pass the filename to the ReadData call**:

```
private static void ReadData(string fileName)
```

Now it's time for only a little change in the read data method. First we need to remove the while loop and all the places where we read from the console and finally the input variable. After that you can make a new variable to generate the absolute path for the file and check if it exists.

```
string path = SessionData.currentPath + "\\\" + fileName;
if (Directory.Exists(path))
{
```

If the path exists we are going to do all the processing of the input, so you may **copy all the code that was in the while loop and paste it in the body of the if statement**. Now that you know that

there is such a file, you may read it. And after that **wrap everything that was in the while loop in a for loop, iterating through all the lines of the file and processing them one by one**. Your code in the if should begin with something like this:

```
string[] allInputLines = File.ReadAllLines(path);

for (int line = 0; line < allInputLines.Length; line++)
{
    if (!string.IsNullOrEmpty(allInputLines[line]))
    {
        string[] data = allInputLines[line].Split(' ');
```

If the path does not exist however, an exception with the name InvalidPath from the ExceptionsMessages is displayed on the OutputWriter.

Now that we've done all these changes, we can easily call the method from the command interpreter like this.

```
string fileName = data[1];
StudentsRepository.InitializeData(fileName);
```

8. **Get help** – does not need any parameters. Displays some information about all of the commands, so that we can use them easily. We've given the whole code for the get help method in the file appended with this lecture. Use it to copy all the printing and not lose time in doing such things. The file is called getHelp.txt.
9. **For the rest of the commands** – you may leave the body unimplemented, because we do not have the functionality implemented yet.

So now that we've written the functionality for the command interpreter, we can link it to the **InputReader** and we should be finally done. All we have to do is go back to the input reader and **change the comment for interpreting the command with the method** that interprets a command from the command interpreter.

```
CommandInterpreter.InterpretCommand(input);
```

Now we should be done with the functionality for interpreting commands and we will only extend it further on in future pieces in order to implement the full functionality of our BashSoft. And we should also be ready with the whole piece. The only thing left is to call the StartReadingCommands from the main method, and test all the functionality that we have by now. We'll leave the part with the testing to you, but we'll show a few pictures of the current state of the program:

```
static void Main()
{
    InputReader.StartReadingCommands();
}
```



```

E:\> cd ..\StoryMode\Executor\bin\Debug> ls
E:\> cd ..\StoryMode\Executor\bin\Debug
-----\dataNew.txt
-----\Executor.exe
-----\Executor.exe.config
-----\Executor.pdb
-----\Executor.vshost.exe
-----\Executor.vshost.exe.config
-----\Executor.vshost.exe.manifest
-----\new 8.txt
E:\> cd ..\StoryMode\Executor\bin\Debug> cdRel ..
E:\> cd ..\StoryMode\Executor\bin> cdRel ..
E:\> cd ..\StoryMode\Executor> ls
E:\> cd ..\StoryMode\Executor
-----\App.config
-----\BashSoft.csproj
-----\CommandInterpreter.cs
-----\Program.cs
-----\Tester.cs
E:\> cd ..\StoryMode\Executor> _

```

```

E:\bojo\Labs\StoryMode\Executor\bin\Debug> cdAbs D:\
D:\> ls
D:\
D:\> System Volume Information
The folder/file you are trying to get access needs a higher level of rights than you c
currently have.
D:\> cdAbs C
Folder does not exist!
D:\> cdAbs C:\
C:\> help

|make directory - mkdir path
|
|traverse directory - ls depth
|
|comparing files - cmp path1 path2
|
|change directory - cdRel relativePath
|
|change directory - cdAbs absolutePath
|
|read students data base - readDb fileName
|
|filter students from given course by given criteria - filter {courseName} excelent/av

```

In the next piece we are going to learn how to make more restricted, pattern following data and filter it easily.

Congratulations! You've successfully completed the lab for Manual String Processing.