TP C#5: Input/Output

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tp5-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- TP5/
        |-- TP5.sln
        |-- Exercise1/
            |-- Everything except bin/ and obj/
        |-- Maze/
            |-- Everything except bin/ and obj/
```

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

In this tutorial, we will study the interaction between our program and the files stored on your disk.

We will use the *System.IO namespace* to handle files in C#. I/O is the abbreviation for Input / Output and it refers to any communication between a program and the outside world. Hence the name *System.IO*.

This *namespace* contains a number of classes that give us access to many features that we will see in the classes section.

The goal of this tutorial will be to discover some of the most important functions of *System.IO*. Then you will have to use them in concrete cases.

1.1 Notice

This course often refers to MSDN, so you have to click on the links in blue before asking about the usage of the functions of *System.IO*.

An archive is available on the intra-net. It contains the submission architecture, the project you need to submit, as well as some files that will be useful for the exercises. All you need to do is unzip the archive and open the projects in Rider.

2 Classes

In this course, the concepts necessary to complete the tutorial will be explained. You will also have to go to MSDN to study how the various functions of *System.IO* work. Any fields that have *Exception* in their name can be ignored, they do not interest you for now.

2.1 File

There are two important parts in a file: information about itself and its content. The *FileInfo* class gives functions to get and modify the information about the file, such as:

- Its creation date
- Its length
- Its parent directory
- Its absolute file
- ...

The *File* class allows the user to manipulate the content of the file. There is a number of functions that enable reading, writing and appending in a file. Read MSDN for more information.

Here is a non-exhaustive list of such functions that could be useful:

- **ReadAllText** reads the entire content of the file and returns it as a string.

- **ReadAllLines** reads the entire content of the file line by line and returns it as a list of strings. Each element in the list is a line from the file.
- **WriteAllText** removes the entire content of the file and write the text given as parameter instead.
- **AppendAllText** Adds at the end of the file the text given as parameter.
- **Delete** deletes the file.

2.2 Directory

Just like a file, a directory has two parts, information and data. The *DirectoryInfo* class gives access to the same kind of information as *FileInfo* but for directories.

The *Directory* class enables the user to manipulate the directories. Its main uses are to obtain information about their content (the files and directories it contains). Some useful edition functions are to delete, create and move directories with respectively **CreateDirectory**, **Delete** and **Move**. The **Move** is also able to move files.

2.3 Path

Files and directories are stored in a tree. A directory can have zero or more children that are files and directories themselves. Thus, a directory's child is a file or a directory inside it. The parent of a file or directory is the directory in which the file or directory is.

There is a special directory, the root. All folders and files of the disk are descendants of the root.

A '/' character is usually added at the end of a directory's name so as to differentiate them from files.

Paths are strings that enable one to move across the file system.

- **/** is the path to the root.
- **./** is the path to the current directory.
- **../** is the path to the parent directory.
- **name** is the path to the file or directory named <name>.
- **../test** is the path to the *test* file or directory that is in the parent directory of the current directory.

There are two ways to define a path, relative and absolute.

A relative path is relative to the directory the user is currently in. For example, when executing this tutorial's code, the current directory is set to the location of the executable file (wether in *<PojectPath>/bin/Debug/* or in *<PojectPath>/bin/Release/*). The path to access the code files is: **../../*.cs** (with *.cs being the c# code files of the project).

An absolute path is prefixed by **/** so it starts from the root. Its advantage is that it doesn't depend on where the user currently is.

Here are some examples of matching between absolute path and relative path:

```
Example: in /tmp/tests/
../       => /tmp/ # The parent of tests/ is tmp/
./../../  => /     # The parent of the parent of tests/ is the root /
../../../ => /     # The parent of the root is the root
```

For this tutorial, you need to understand how paths work and how to manipulate them. Fortunately, there is a class provided by C# that manipulates the paths for you, you just have to call the right functions: *Path* from *System.IO*. Look into the part called *Methods* and ignore anything with *Exception* in its name. *Indice*, *Combine*, *ChangeExtension* and *GetExtension* may be useful in this tutorial.

2.4 Projects in C#

2.4.1 Solution

A solution is a structure that organises projects. It can contain several projects and is stored in an architecture such as the following:

```
<SolutionName>/
|-- <SolutionName>.sln
|-- <Project1>/
|-- <Project2>/
|-- <Projectn>/
|-- Random files
|-- Random folders/
```

The *.sln* file defines the solution. The *Project* directories are the different projects of the solution. There can also be some directories and files with no link to the solution or projects. They have no influence.

The given solution in the archive has the following architecture:

```
TP5/
|-- Exercise1/
|-- Maze/
|-- tests/
|-- .idea/
|-- TP5.sln
```

The solution's name is *TP5*, with the file *TP5.sln*. It has two projects *Exercise1* and *Maze*. *tests/* is a directory that contains some files to test your functions. *.idea/* is a file generated by Rider that stores the solution's configuration. It can be deleted at any moment, Rider will always regenerate it.

For more information about solutions, see: MSDN Solution.

2.4.2 Project

There are two projects in this tutorial, but what exaclty are they?

A project is a directory that has:

- Two directories bin/ and obj/ are created at each execution of the project. The compiled code is put in these two folders. They can thus be deleted at any time without any problem.

- A .csproj file that defines the project.
- An undefined number of sub directories and code files. The .cs files for example contain the code in C# language.

Each project can be compiled independently. To execute a project easily: select it in the solution explorer, right click, run *<project>*.

Here is the architecture of Exercise1 project:

```
Exercise1/
|-- bin/
|-- obj/
|-- Exercise1.csproj
|-- Architecture.cs
|-- CopyFile.cs
|-- PrintFile.cs
|-- Test.cs
```

The directories *bin/* and *obj/* might not exist. *Exercise1.csproj* defines the project *Exercise1*. Les files *.cs* are the code files to complete in this tutorial.

3 Exercises

It is now your turn to play with files.

3.1 Exercise 1: Basics

This exercise will teach you how to use the different functions of the following classes:

- *File*
- *Directory*
- *Path*

You have to modify the functions in the project named **Exercise1**. To test a function, call it in the Main function that is in the file *Test.cs*. Also, files to test your code are given in archive, in the *tests* directory.

The *Test.cs* file will be deleted during the evaluation of your code. Therefore, you should not write any important code in it. The *tests* directory will also be deleted.

Don't forget to add at the beginning of each file:

```
1  using System.IO
```

With this, you won't have to type *System.IO.<class>.<function>()* at each use of a function of the namespace. You will be able to write *<class>.<function>()*.

3.1.1 Part 1: Print a file

Find the easiest way to print the entire content of a file in the console.

You must complete the following function in the file *PrintFile.cs*.

```
1  public static void PrintAllFile(string path)
2  {
3      //TODO
4  }
```

Warning! You must handle the case where the file given as parameter doesn't exist. You then have to output an error message. You can print for example:

```
could not open file: <file given as parameter>
```

3.1.2 Part 2: Keep half for yourself

In this part you should print only one line out of two from the file given as argument. The first line of the file should thus be printed but not the second one.
Lines number: 0, 2, 4, 6, 8, 10, ... should be printed.
Lines number: 1, 3, 5, 7, 9, 11, ... should **NOT** be printed.

You have to complete the following function in the file *PrintFile.cs*.

```
1  public static void PrintHalfFile(string path)
2  {
3      //TODO
4  }
```

Warning! You must handle the case where the file given as parameter doesn't exist. You then have to output an error message. You can print for example:

```
could not open file: <file given as parameter>
```

3.1.3 Part 3: Copy-paste is BAD!

You have to read all text from the source file, then write it in the destination file.
If the destination file doesn't exist, create it.
If the destination file already exists, overwrite it.

You must complete the following function in the file *CopyFile.cs*.

```
1  public static void CopyAllFile(string source, string destination)
2  {
3      //TODO
4  }
```

Warning! You must handle the case where the file given as parameter doesn't exist. You then have to output an error message. You can print for example:

```
could not open file: <file given as parameter>
```

3.1.4 Part 4: Too lazy to finish

You have to read half of the text from the source file, then write it in the destination file.
If the destination file doesn't exist, create it.
If the destination file already exists, overwrite it.

If the source file has 11 lines, for example:
Lines 0 to 4 will be copied to destination file (first 5 lines)
Lines 5 to 10 will be ignored (last 6 lines)

If the source file has 10 lines for example:
Lines 0 to 4 will be copied to destination file (first 5 lines)
Lines 5 to 9 will be ignored (last 5 lines)

You have to complete the following function in the file *CopyFile.cs*.

```
1  public static void CopyHalfFile(string source, string destination)
2  {
3      //TODO
4  }
```

Warning! You must handle the case where the file given as parameter doesn't exist. You then have to output an error message. You can print for example:

```
could not open file: <file given as parameter>
```

3.1.5 Part 5: Godlike architect

This function will create a file architecture as explained below:

```
1. Create a directory with the path given as parameter.
2. Create a "AUTHORS" file inside the previously created directory.
3. Fill the file with "* firstname.lastname\n"
('\n' is a newline character)
4. Create a "README" file inside the previously created directory.
5. Fill the file with
"Everything in programming is magic... except for the programmer\n"
6. Create a "TP5" directory inside the previously created directory.
7. Create a "useless.txt" file inside the "TP5" directory.
```

If the given path is */tmp/archi/*, the architecture should be:

```
/tmp/archi/
|-- AUTHORS
|-- README
|-- TP5/
    |-- useless.txt
```

Complete the following function in the file *Architecture.cs*.

```
1  public static void Architect(string path)
2  {
3      //TODO
4  }
```

Warning! You must handle two cases:
If the path given as parameter is an existing file, remove it. If the path given as parameter is an existing directory, remove it and all its content.

3.2 Exercise 2: Maze

Now that you know how to use *System.IO*, it is time to get a little serious.

In this exercise, you will have to create functions in the file named **Maze.cs** in the project *Maze*.

Don't forget to add at the beginning of each file:

```
1 using System.IO
```

With this, you won't have to type *System.IO.<class>.<function>()* at each use of a function of the namespace. You will be able to write *<class>.<function>()*.

3.2.1 introduction

The program you will create has three steps:

You have to first load a maze from a file. You need to ask the user for the path to the file, then load it and parse it into a grid.

Once the maze loaded, you need to solve it. You have to find a path between the start point and end point of it. The grid must be modified to insert the path you found in it.

Now that the maze is solved, you just have to save the modified grid into a new file.

3.2.2 instructions

It is now time to explain exactly what you will do in this exercise. Your program will have to stick strictly to the instructions.

Ask for a file

At the beginning of your program, you will ask the user which maze he would like to load. You will next save his answer.

However, since you don't trust the user, you NEVER trust the user, you have to check that his input was not complete bullshit.

So, you need to test two things: check if the file exists at the given path and check that it actually ends with a *.maze* extension. Look into *File* and *Path* from *System.IO* to test easily.

Exemples:

```
> Which file should be loaded ?
/tmp/tests/map1.txt         # this is not a .maze, restart
> Which file should be loaded ?
/tmp/tests/map1.maz  # this is not a .maze, restart
> Which file should be loaded ?
/tmp/tests/map1.maze        # it is a .maze and it exists
> Thank you, bye
```

The output file

You have a valid maze file. So, now you need to deduce the name of the output file from it.

To get it, you just need to switch the extension of the input file *.maze* into the extension *.solved*.

For example, if there is the input file */tmp/tests/map1.maze*:

```
input:  /tmp/tests/map1.maze
output: /tmp/tests/map1.solved
```

.maze file format

The *.maze* files are saved as an array of characters.

Here is an example of *.maze*:

```
> cat tests/map1.maze
SBOOOBOOOBOOO
OBOBOBOBOBOBO
OBOBOBOBOBOBO
OBOBOBOBOBOBO
OBOBOBOBOBOBO
OBOBOBOBOBOBO
OOOBOOOBOOOBF
```

A *.maze* file is a rectangle with each character representing a kind of frame. Here is the meaning of each character:

- **S** is the *START* square, the start of the maze.
- **F** is the *FINISH* square, the end of the maze.
- **B** is a *BLOCK* square, a wall that cannot be crossed.
- **O** is a *EMPTY* square, a path (that can be walked trough).

The goal is to walk through **O** squares, to go from start **S** to finish **F**.

.solved file format

In the maze, it is possible to move only horizontally and vertically. Diagonal movements are forbidden.

If a path is found, all the squares of this path should be set to **P**.

If no path has been found, the *.solved* file should not be created.
Here is an example:

```
> cat tests/map3.maze
SOBOBOBOBOBOBOBOBF
OOOOOOOOOOOOOOOOOO
BOBOBOBOBOBOBOBOBB
BBBBBBBBBBBBBBBBBB
> cat tests/map3.solved
PPBOBOBOBOBOBOBOBF
OPPPPPPPPPPPPPPPPP
BOBOBOBOBOBOBOBOBB
BBBBBBBBBBBBBBBBBB
```

The Point class

In the *Maze.cs* file there is a second class, different from the *Maze* where the exercise's code should be put. This class is named *Point*.

It enables you to store the coordinates of a square in the maze's grid. Instead of moving around with two values for the x axis and y axis, the Point class allows you to have only one variable.

Here are some usage examples:

```
1  public static void Main(string[] args)
2  {
3      // Create a new point with X = 0 et Y = 1.
4      Point p = new Point(0, 1);
5
6      Console.Writeline(p.X); // prints 0
7      Console.Writeline(p.Y); // prints 1
8
9      p.X = 2;
10     p.Y = 10;
11
12     Console.Writeline(p.X); // prints 2
13     Console.Writeline(p.Y); // prints 10
14 }
```

3.2.3 Guide

You should implement the following functions to complete this exercise.

They should all be put in the file *Maze.cs* in the class *Maze*.

Ask for the .maze

Ask the user for the *.maze* file and check its validity. Create a function that returns the path to a valid *.maze* file:

```
1  private static string AskMazeFile()
2  {
3      // TODO
4  }
```

- Ask the user for the file.
- Wait for an input from him.
- Check if the file exists and is *.maze*.
- If not, start over from the beginning.

Name of the .solved

Find the name of the *.solved* file from the *.maze* file. Create a function that takes as parameter the name of the *.maze* file and returns the name of the *.solved* file:

```
1  private static string GetOutputFile(string fileIn)
2  {
3      // TODO
4  }
```

Find the function in *System.IO* that does exactly that in one line.

Get the maze from the .maze file

Read the *.maze* file given as parameter. Then create a two dimensional array from it and then return.

```
1  private static char[][] ParseFile(string file)
2  {
3      // TODO
4  }
```

Here is what the *map3.maze* file should look like after being parsed:

```
SOOB
OOBF
BOOO
```

```
1  {
2      { 'S','O','O','B' },
3      { 'O','O','B','F' },
4      { 'B','O','O','O' }
5  }
```

Find the START square

To solve the maze you need to find the starting point. So, you need the coordinates of the **S** square in the grid. Look at the entire grid, when the *S* square is found, store the coordinates in a *Point* object. The case where the grid has no *START* square will not be tested.

```
1  private static Point FindStart(char[][] grid)
2  {
3      // TODO
4  }
```

Solving the maze

The solving method given in this tutorial is the backtracking. It is a simple method that works well. There are, however, many other ways (often better) to solve this maze.

The function is recursive and takes as parameters:

- The maze's grid called grid.
- A grid of the same size as grid called processed. It is filled with zeros before calling the function for the first time. It tells which squares have already been visited.
- A point object. start the algorithm with the *START* square's coordinates.

```
1  private static bool SolveMazeBackTracking(char[][] grid, int[][] processed,
2  Point p)
3  {
4      //TODO
5  }
```

What the backtracking algorithm does: it follows a path as long as it is unknown whether it is correct or wrong. If it is wrong, you stop following this path, you go back one step and follow a new path. If it is correct, you stop, you found the *FINISH* and you tell to every previous step that this path is the chosen one.

```
1. If the point is outside the grid, return FALSE.
2. If the point has already been visited, return FALSE.
3. This point is now set as visited.
4. If the point is the FINISH square, return TRUE.
5. If the point is a BLOCK square, return FALSE.
6. Recursively call the algorithm on the square above.
7. If it returned TRUE, set the current square as PATH and return TRUE
8. Recursively call the algorithm on the square on the left.
9. If it returned TRUE, set the current square as PATH and return TRUE
10. Recursively call the algorithm on the square on the right.
11. If it returned TRUE, set the current square as PATH and return TRUE
12. Recursively call the algorithm on the square below.
13. If it returned TRUE, set the current square as PATH and return TRUE
14. Return FALSE
```

Saving the maze

You must now save the maze you just solved into the *.solved* file:

```
1  private static void SaveSolution(char[][] grid, string fileOut)
2  {
3      // TODO
4  }
```

You just need to do the reverse from what you did in *ParseFile*.

3.2.4 Bonus

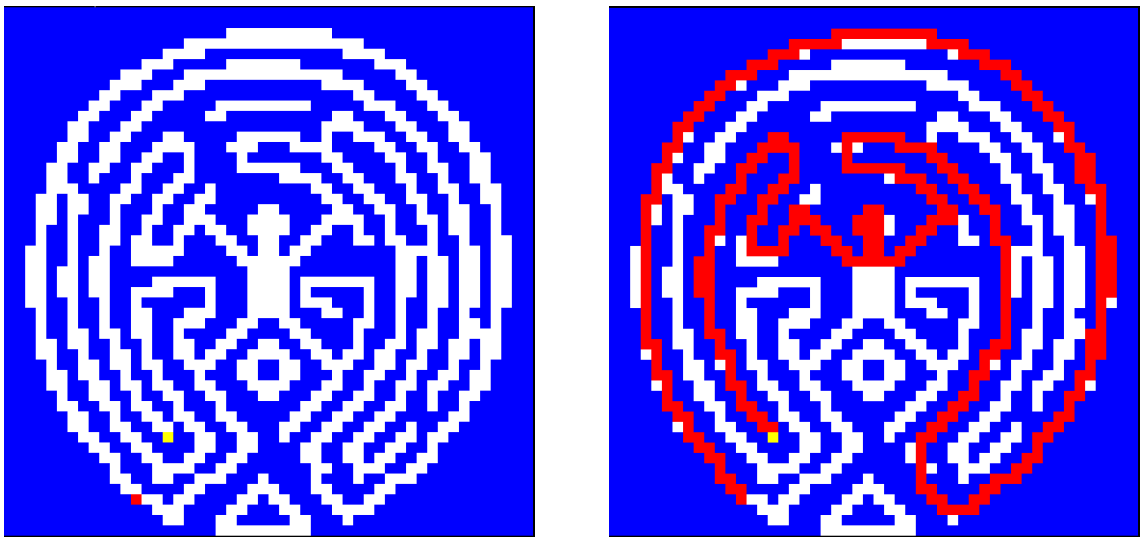
If you add bonuses, put them in new functions with logical name. Also, describe what they do in your *README*.

BONUS 1: Improvement

Improve the path finding algorithm. The backtracking technique is slow and doesn't search for the best path. Implement a better algorithm able to find the best path possible and fast. You can look on the internet if you want (Wikipedia path finding).

BONUS 2: Printing

Print the maze in the console. Here is an example of what you could obtain.



(a) map5.maze (b) map5.solved

Figure 1: Example of printing in the console

You can check out the class *System.Console* to do this bonus.

BONUS n: Others

Any other bonus that you think of is welcome. If you add something, describe it in your *README*.

These violent deadlines have violent ends.