TP C#2
My name is Gutenberg

Submission

Archive

You must submit a zip file with the following architecture :

```
rendu-tp-firstname.lastname.zip
    |-- rendu-tpcs2-firstname.lastname/
        |-- AUTHORS
        |-- README
        |-- HelloWorlds/
            |-- HelloWorlds.sln
            |-- HelloWorlds/
                |-- everything except bin/ and obj/
        |-- Add/
            |-- Add.sln
            |-- Add/
                |-- everything except bin/ and obj/
        |-- Skyscraper/
            |-- Skyscraper.sln
            |-- Skyscraper/
                |-- everything except bin/ and obj/
        |-- JollyRoger/
            |-- JollyRoger.sln
            |-- JollyRoger/
                |-- everything except bin/ and obj/
        |-- Morse/
            |-- Morse.sln
            |-- Morse/
                |-- everything except bin/ and obj/
```

- You shall obviously replace *firstname.lastname* with your login (which is *firstname.lastname*).
- The code must compile.
- In this practical, you are allowed to implement any other functions than those specified, they will be considered as bonus. However you must keep the archive's size as light as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline, NOTHING ELSE. 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

You must write in this file any comments on the practical, your work, or more generally on your strengths / weaknesses. An empty README file will be considered as an invalid archive (malus).

1 Introduction

1.1 Objectives

During this TP, we will approach new notions such as:

- The Console,
- ASCII art,
- String manipulation,
- Morse code.

2 Basic Knowledge

2.1 The Console

The terminal

Until now, most of the program you used were usable with your mouse. These applications (called GUI applications), composed of windows, tabs, buttons, images, etc., are as you might think, not the most basic ones. You will never have to implement this type of application during any of your C# practicals this year.

Instead, we will ask you to create CLI programs (for Command Line Interface). These applications work inside a terminal, and are executable using another program called shell. A shell is basically a program which allows an user to interact with his computer using text commands. Once a terminal opens you automatically are running inside a shell. The shell (bash in default on your systems) prints you some basic pieces of information about you, your computer and the directory you are using. These pieces of information are formatted inside a string called *prompt* (see below).

```
[sparrow@sea ~/blackpearl]$
```

In this example, the user of the computer is called `sparrow`, he/she named his/her computer `sea` and is currently inside the directory named `~/blackpearl`.

Using the command line

After printing the prompt, the shell waits for you to enter a command. A valid command can have many forms, but is often the name of a program, followed by arguments. For example, have a look at the command below :

```
1 [sparrow@sea ~]$ ls
2 foo  acdc  sups
3 [sparrow@sea ~]$ ls foo
4 bar  baz
5 [sparrow@sea ~]$
```

In this example, the program `ls` is first executed alone (line 1). `ls`, displays the content of a directory, if no argument is given it prints the content of the current directory (here `foo`, `acdc` and `sups`). On the third line, the `ls` is executed with one argument, `foo`, which content is then printed. As you can see, to give an argument to a program, you just have to append the argument after the name of the program, preceded by a blank space. On line 5, the prompt is printed again and the shell awaits for a new command.

Of course, more than one argument can be passed to a program. Special arguments can be given most of the time to programs to modify the behavior of the program. These special arguments are named options and usually start with a `'-'`. Most of the time, on Unix systems, you can obtain the list of possible arguments of a program using the `--help` argument, or by opening the `man` page.

Standard streams

Now that you understand what the command line is, let's explain about it more. Terminals work with *streams*, a stream is basically a way your system uses to read or write data, including text. For

example, in Unix systems, any file can be used as a stream. They are also 3 main streams that you must know:

- Standard Input: This stream is the one your programs can read in. Everything you type in a terminal is sent to this stream. It is often called `stdin`.
- Standard Output: This stream is the opposite of `stdin`. It's the stream your shell and your programs write in. What you, as a user, read on a terminal is what the standard output sends. It is often called `stdout`.
- Standard Error Output: This stream is very similar to `stdout`, it works exactly the same way, except that in most case, only error messages are sent on this stream. For you, when you read in the console it doesn't change anything. It is often called `stderr`.

It is important that you understand correctly how you can interact with programs using these streams, if you have doubts, ask your beloved ACDCs.

2.2 Get arguments

As you already know, the `Main` method is automatically called at the start of your program. Let's have a look at its prototype:

```
1 static int Main(string[] args);
```

The `static` prefix is a C# keyword (you will learn later about its usage). `int` can be replaced by `void`. It's used to send to your system a code about the state of your program at its end (we can ignore it for now). `Main` is the identifier of the method. And... `string[] args`. As you know, inside the parentheses of a method, we put its arguments. But the `Main` method is automatically called when your program starts, so how do you change those arguments and what is their purpose ? The answer is simple, when your system calls the `Main` method, it puts all the arguments, given to your program on the command line, inside the `args` array. For example, write this basic program and compile it into a program called `MyEcho.exe`:

```
1 static void Main(string[] args)
2 {
3     if (args.Length < 1)
4         Console.WriteLine("Not enough arguments.");
5     else
6         Console.WriteLine(args[0]);
7 }
```

Let's try it into the console:

```
1 [sparrow@sea ~]$ ./MyEcho.exe
2 Not enough arguments.
3 [sparrow@sea ~]$ ./MyEcho.exe Test.
4 Test.
5 [sparrow@sea ~]$ ./MyEcho.exe I love my ACDCs
6 I
7 [sparrow@sea ~]$ ./MyEcho.exe "I love my ACDCs"
8 I love my ACDCs
9 [sparrow@sea ~]$
```

As you see, this program check if arguments are passed to it. If no argument is given, it prints the message `"Not enough arguments."` on `stdout`, if arguments are given, it prints the first one (index 0 in the array).

Notice the behavior of the program on the lines 7 and 8 of the console example. Here, we use double quotes (") when giving arguments to the program. If you do so, the words between the double quotes are seen by the shell as one token and put inside the `args` array as one string. If you don't write the double quotes, the shell splits the arguments on blank spaces, and each token is put alone at its index in the array.

2.3 Console class

You will now learn how to read from and write to the standard streams. To do so, the C# library provides you the Console [1] class. This class contains many methods which allow you to write in the console, read from it, get information about its status, change its properties. There is a summary of what you can do and the main methods to do so. You can of course have further information about it by reading the according MSDN page. All those methods can be called by writing in your code `Console.MethodName(args)`.

Write

Two methods allow you to write in the console :

- `Console.Write(arg)`: This method converts the given `arg` parameter to a `string` and prints it on `stdout`. The argument can be of many types. The method can even take more than one parameter, read the MSDN page to have more information about it.
- `Console.WriteLine(arg)`: This method does the same as `Console.Write`, except that it adds a newline at the end of the printed string.
  => `Console.WriteLine(arg) == Console.Write(arg + "\n")`

[1] `https://msdn.microsoft.com/en-us/library/system.console`

These two methods have variants, `Console.Error.Write` and `Console.Error.WriteLine`. They do the same as the metods listed above, except that they print the result on the standard error output (`stderr`) instead of `stdout`.

Here are some examples of how to use these functions, feel free to test them (the output is written in comment):

```
1  Console.WriteLine("The Code is the Law.");
2  // The Code is the Law.
3
4  Console.Write("Jack");
5  Console.Write("Sparrow\n");
6  // JackSparrow
7
8  Console.WriteLine("a" + "b" + "c" + "d");
9  // abcd
10 Console.WriteLine("42");
11 // 42
12 Console.WriteLine(42);
13 // 42
14 Console.WriteLine(6 * 7);
15 // 42
16
17 string beloved = "ACDC";
18 bool best = !false;
19 Console.WriteLine("My {0}s are the {1} best ones."
20                   , beloved, best);
21 // My ACDCs are the True best ones.
22 Console.Error.Write("An error occurred.\n");
23 // on stderr: An error occurred.
```

Remember that even if, for you, everything is printed the same way on the console, when you use `Console.Error`, the result is sent to an other stream, `stderr`.

Read

Read from the console is the opposite operation, it allows you to get a value typed by a user. They are two main methods to read from standard input:

- `Console.Read()`: This method awaits for the user to type one character. As soon as the user typed one in the console, the method returns the character typed (casted in an `int`) and the program continues.

- `Console.ReadLine()`: This method awaits for the user to type one entire line, ended by a newline (when the key `Enter` is typed). As soon as the user typed a line in the console, the method returns the full line as a string (without the ending `'\n'`) and the program continues.

To try these methods, read and run the following program:

```
1  static void Main(string[] args)
2  {
3      Console.Write("First char: ");
4      int c = Console.Read();
5      Console.WriteLine();
6      Console.WriteLine("You typed: '{0}'", (char)c);
7
8      Console.Write("Second char: ");
9      c = Console.Read();
10     Console.WriteLine();
11     Console.WriteLine("You typed: '{0}'", (char)c);
12
13     Console.WriteLine("Type a line:");
14     string str = Console.ReadLine();
15     Console.WriteLine("You typed: \"{0}\"", str);
16     Console.WriteLine("Length == " + str.Length);
17 }
```

Please notice that you can't do `Console.Error.Read()`. Indeed, this would have no sense because the `stderr` stream is only for output.

Appearance

So, you can read from and write into the console. This is cool. But this is a bit boring. In fact, the Console class allows you to do a lot of things to modify the output, such as cursor positioning, background and foreground color setting, etc.:

Methods:

- `Console.Clear()`: Clears the console, meaning that after a call to it, any text on the console will be deleted and the cursor for the Write method will be set to the top left corner.
- `Console.SetCursorPosition()`: Sets the position of the cursor used to write.

Properties:

- `Console.Title`: Gets or sets the title to display on the console title bar.
- `Console.BackgroundColor`: Gets or sets the background color of the console.

- `Console.ForegroundColor`: Gets or sets the foreground color of the console.

A lot more methods and properties are available to use the console. Again, read the MSDN page[2] of the Console class to get further information about them and have some example on how to use them.

Let's do some exercises to use what you learned about the console!

3 Exercise 0: Hello WorldS

Let's start with basic basics. You sure know the "Hello World". This exercise will ask you to implement some enhanced versions of it.

You must implement your work in a project named `HelloWorlds`.

Write the method `Hello`:

```
1 static void Hello(string str);
```

This method takes a name in the form of a string as parameter and should display a line on the console standard output, saying "Hello" to the name, followed by a exclamation mark. If the name is empty, it should say "Hello" to the "World". You shall not use the "+" operator. Here are in comment examples of expected outputs.

```
1 Hello("Jack");
2 // Hello Jack!
3 Hello("");
4 // Hello World!
```

Then, write the method `HelloWorldErr`:

```
1 static void HelloWorldErr();
```

This method prints "Hello World!" on the console standard *error* output.

```
1 HelloWorldErr();
2 // Hello World!
```

Finally, you will write the method `ImportantHello`:

```
1 static void ImportantHello();
```

This method prints on the console standard output "This is an important announcement: Hello World!" with yellow background and a red text. Once the message has been printed, the colors of the console should be reset to their previous values. You have to use the `Console.BackgroundColor` and

[2] `https://msdn.microsoft.com/en-us/library/system.console`

`Console.ForegroundColor` properties for this. Remember that you can get or set them. These properties are of a special type `ConsoleColor`. `ConsoleColor` is an `enum` type (you should not worry if you don't know what this is, this will be taught in a future practical). Since `ConsoleColor` is a type, you may create variables of this type. Valid values for this type can be obtained with `ConsoleColor.mycolor` where "mycolor" is one of the colors listed in the MSDN page of `ConsoleColor`.

The project of this exercise should NOT contain a `Main` method.

4 Exercise 1: Add

The goal of this exercise is to get and use the arguments passed to your program through the command line interface. Therefore, the project of this exercise MUST contain a `Main` method. You must implement your work in a project named `Add`.

Your program will add the two first arguments passed to your program through command line, and will display the result on standard output. If there are not enough arguments, you should print a message on standard error output. You might want to check the `Int32.Parse` MSDN page. You may also assume that the two first parameters passed will be valid integers.

```
[sparrow@sea ~]$ ./Add.exe
Invalid number of operands
[sparrow@sea ~]$ ./Add.exe 42 24
66
[sparrow@sea ~]$ ./Add.exe -1 -3 8
-4
```

5 Exercise 2: Skyscraper

For the last practical, you had to draw a house. Today we will ask you to draw a skyscraper. Your skyscraper will have a non-constant number of floors. Each floor will be owned by the person living in, and each owner will have random decoration tastes (represented with colors). Of course, the ground and the roof of the building are not owned by anyone and will have usual colors.

You must implement your work in a project named `Skyscraper`.

5.1 Ground and Roof

First, let's do the simple and constant parts.

Write the method `Ground`:

```
static void Ground();
```

This method draws the ground floor, the ground floor has the same color as the default ones of your terminal, you shall not modify them yet. Here is an example of what the ground floor must look like (the long lines are sequences of underscore characters '_'), the width of the two levels is equal to 14:

```
|     _____     |
|____|-||-|____|
```

Once done, write the method `Roof`:

```
static void Roof();
```

This method shall be very similar to the previous one, except that it draws the roof instead of the ground floor. Here is an example of what the ground floor must look like:

```
 _____________
|             |
```

5.2 Beautiful floor

Now that you have the basic parts, let's move to the fun ones !

Implement the method `Floor`:

```
static void Floor();
```

`Floor` prints one floor of the buiding, with random foreground and background colors, a floor must look like this:

```
|  |0|  |0|  |
|             |
```

To get a random `ConsoleColor`, you must use a `Random` object. To do so, we provide you a method:

```
static Random random = new Random();

static ConsoleColor GetRandomColor()
{
    return (ConsoleColor)random.Next(0, 16);
}
```

Copy these 6 lines inside your code, next to your other methods. You don't need to understand this code. The provided method, `GetRandomColor`, returns a random color usable to set the properties `Console.ForegroundColor` or `Console.BackgroundColor`. If you are curious, feel free to read the MSDN page about the `new` keyword, the `Random` class and the `enum` type.

So, you are now able to get a random `ConsoleColor`. Use it in your `Floor` method.

The `Floor` method must behave as described below:

- Save the previous colors of the console,
- Set the colors of the console to random colors,
- Print the 2 lines of the floor, only the 14 first characters, corresponding to the width of the building, must be colored, not the rest of the console, think about the difference between `Console.Write` and `Console.Writeline`,
- Set back the console colors as they were before the call to this method.

5.3 Let's build something

You are now able to write the ground floor, a single colored floor and a roof. Implement now the `Floors` method:

```
1 static void Floors(int n);
```

This method takes an `int` as a parameter and prints `n` colored floors. It must be recursive and must call your `Floor` method.

Implement now the final method, named `Skyscraper`:

```
1 static void Skyscraper(int n);
```

This method simply calls the `Roof`, `Floors` and `Ground` methods.

The project of this exercise should NOT contain a `Main` method.

6 Exercise 3: Jolly Roger

Arr, me hearty! I lost the jack of my ship and I need a new one. If you can craft it for me, I'll give you some doubloons of my last bounty.

In this exercise, you'll have to draw an ASCII Art[3] skull in the middle of your console.

You must implement your work in a project named `JollyRoger`. You will first implement the `PrintSkull` method.

```
1 static void PrintSkull();
```

This method will print an ASCII art Skull. You must clear your console before starting printing. The Skull must be printed from the first line on top of the console, and must be centered in width.

[3] `https://fr.wikipedia.org/wiki/Art_ASCII`

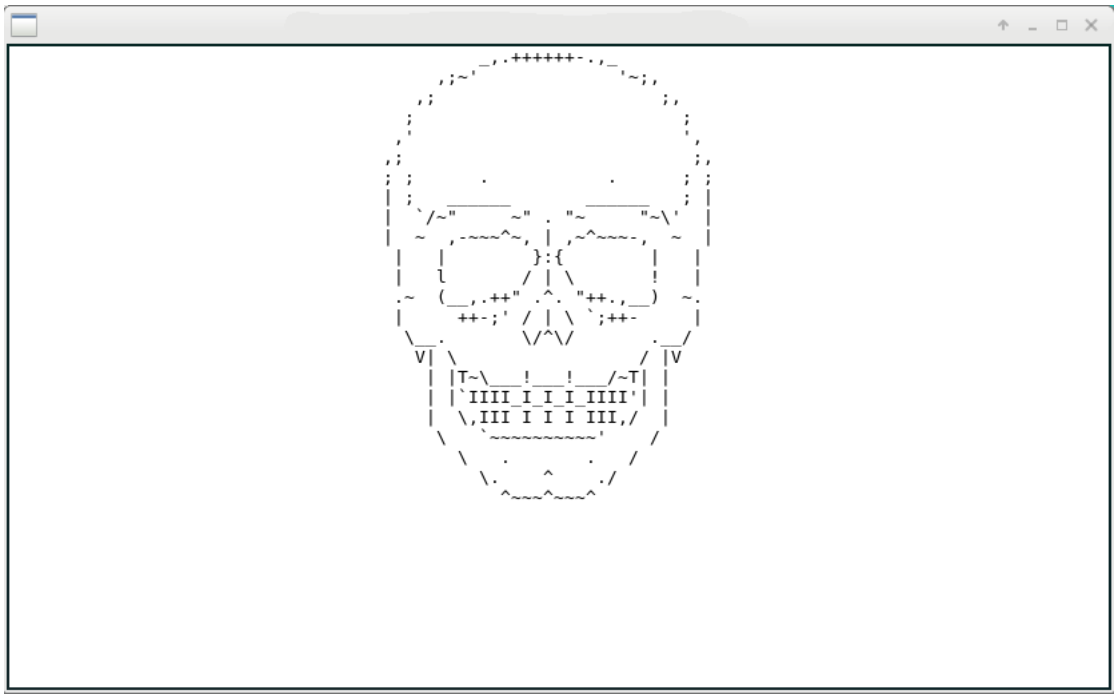

Figure 1: Expected display

You should consider reading MSDN pages about `Console.BufferWidth`, `Console.BufferHeight`, and `Console.SetCursorPosition` to do this. You may find nice Skulls here : `http://www.asciiworld.com/-Death-Co-.html`. If you don't have enough space in your console, nothing should be printed.

Then, you will implement the method `MyJack`.

```
1 static void MyJack();
```

This method will first print your skull. It will then await for an user input to print your skull again, but with white background and black text, rather than the previous console colors. It will then await for another user input before clearing the console and printing (with the colors of the terminal before the black on white skull) "The code is the law.".

The project of this exercise should NOT contain a `Main` method.

7 Exercise 4: -.-. .- -. ... .-.. .- - --- .-.

In this exercise, you will have to write a program which translates a message written with the Latin alphabet (aka. the French one) to Morse code. Your program will take one argument, an integer indicating how many letters your program must translate. Until all the letters are read, the program will wait for the user to write a character. At the end, it must print the translated message on the standard output. If no argument is given, or too many, your program must write a message on the standard *error* output.

You must implement this exercise in a project called `Morse`. It should contain a `Main` method.

Have a look below on an example of how to use the program you must create:

```
[sparrow@sea ~]$ ./Morse.exe
Invalid arguments.
[sparrow@sea ~]$ ./Morse.exe 42 51
Invalid arguments.
[sparrow@sea ~]$ ./Morse.exe 3
SOS
... --- ...
[sparrow@sea ~]$ ./Morse.exe 11
Votai Test.
...- --- - .- .. / - . ... - .
[sparrow@sea ~]$ ./Morse.exe 19
The code is the law
- .... . / -.-. --- -.. . / .. ... / - .... . / .-.. .- .--
[sparrow@sea ~]$
```

7.1 Morse code

"Morse code is a method of transmitting text information as a series of on-off tones, lights, or clicks that can be directly understood by a skilled listener or observer without special equipment. The International Morse Code encodes the ISO basic Latin alphabet, some extra Latin letters, the Arabic numerals and a small set of punctuation and procedural signals (prosigns) as standardized sequences of short and long signals called *dots* and *dashes*." [4]

More basically, the Morse code doesn't change the words, it is just another way of writing / spelling the letters of a message. Each character of a message is encoded as a sequence of *dots* (`'.'`) and *dashes* (`'-'`). To make the resulting message more readable, each encoded character is separated from the next one by a blank *space* (`' '`). The *space* character, separating words in the original message, is a bit particular. Because it gives information about the original message (the words), it can not be ignored, but can not be encoded as a simple *space* neither, because this character is already used to separate letters. You will represent *space* characters with *slash* (`'/'`). For example, the message `"foo bar"` would be printed in Morse code as `"..-. --- --- / -... .- .-."`.

[4] `https://en.wikipedia.org/wiki/Morse_code`

7.2 Convert characters

Implement the method `LetterToMorse`, which converts a given character to the string reprenting it in Morse code:

```
static string LetterToMorse(char c);
// LetterToMorse('f') -> "..-."
// LetterToMorse('F') -> "..-."
// LetterToMorse('7') -> "--..."
// LetterToMorse(' ') -> "/"
// LetterToMorse('#') -> "#"
```

This method takes as parameter the character to translate and returns as a string the encoding of the character. If the character is one of the 26 letters of the alphabet (upper or lower) or a digit, use the table given on the Wikipedia page[5] of the Morse code as a reference for the conversion. If the character is a blank space, returns the string `"/"`. If the character has any other value, it must return a string containing only the given character.

7.3 Read and translate

Implement the `Translate` method:

```
static string Translate(int n);
```

This method must be recursive. It reads `n` characters entered by the user on the standard input and returns the string containing the translation of the full message in Morse code. You of course have to use your `LetterToMorse` method. Do not forget to separate each translation of letter by a space (`' '`).

7.4 Make it runnable

You will now write the rest of your algorithm as a functioning program. For this, you need to code into the `Main` method.

You must first check the number of arguments given to your program, if there is no argument, or if too many are given, you must write a message on the standard *error* output (`stderr`) and stop.

Then, it converts the given argument into an `int` (remember `Int32.Parse`), calls your `Translate` method and then prints on the next line of the console the resulting translation in Morse code.

[5] `https://upload.wikimedia.org/wikipedia/commons/b/b5/International_Morse_Code.svg`

8 Boni:

If you are done with all the exercises and want to try more stuff with the console, you can improve any of the previous exercises the way you want. Here is a list of possible things you can do:

- Make the skull of the exercise 3 animated,
- Change window title depending on which program is launched,
- Use the `Console.Beep` method to play the Morse code message after translation,
- Write a variant of the translator, which translates messages the other way: from Morse code to Latin alphabet,
- Anything else, be creative.

Don't forget to list and explain all your bonii in your README file.

The code is the law.