TP C#8: Brainfuck and co

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tp-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- Brainfuck/
        |-- Brainfuck.sln
        |-- Brainfuck/
            |-- 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).

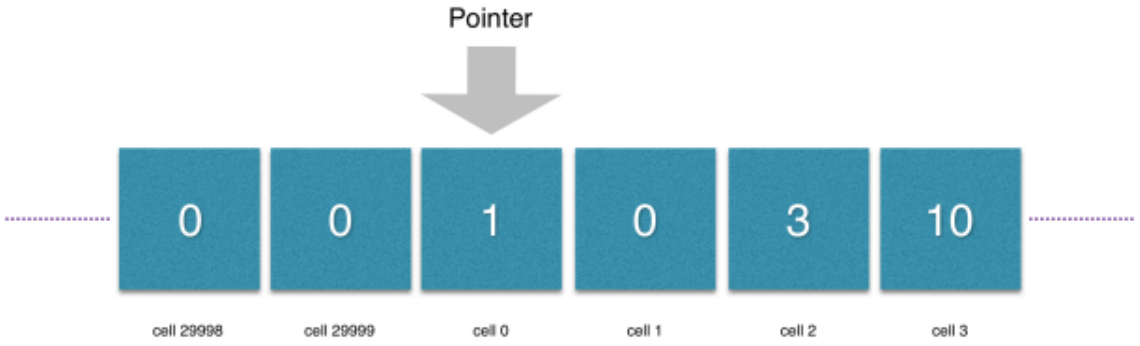
WARNING

This TP needs a week. If you begin it only few days before the deadline you will not be able to get a good grade.

1 Brainfuck

1.1 Generalities

The Brainfuck is an esoteric programming language created by Urban Müller in 1993. The goal of this language is to be as simple as possible but remaining Turing-complete. Therefore, it is theoretically possible to write any program in Brainfuck . Just like the Turing machine, the language consists of a "tape" that stores bytes. You can move from one cell to another. A pointer will keep the position of the current cell of the array.



The array is usually a circular array of 30,000 elements.

1.2 Instructions

The langage has only 8 different symbols, each symbol corresponding to an instruction. Here is the list of these instructions :

- > increment the data pointer
- < decrement the data pointer
- + increment the value of the cell pointed to by the data pointer p
- - decrement the value of the cell pointed to by the data pointer p
- . output the value of the cell pointed by the data pointer p (cell value = ASCII code)
- , accept one byte of input, storing its value in the cell at the data pointer p
- [jump to the instruction after the corresponding] if the byte value is equal to 0
-] jump to the instruction after the corresponding [if the byte value is different from 0

```
1 ++++++++++[>+++++++>++++++++++>+++>+<<<<-]
2 >++>+.+++++++..+++>++.<<+++++++++++++++.
3 >.+++.------.--------.>+.>.
```

This code prints "**Hello World!**" followed by a **newline**. Only the first line is worth to explain. Firstly it initializes a counter to 10 in the first cell of the array. Once done, a loop

initializes the next four cells to the closest value possible of the needed characters. The second cell is initialized to 70 in order to print the uppercase characters. The third one to 100 in order to print the lowercase characters. The fourth one to 30 in order to print the special characters and the fifth one to 10 in order to print a newline.
The other lines are just incrementing/decrementing cells in order to print the exact value of the needed character.

1.3 implementation

You understood the purpose of this TP is to code a Brainfuck interpreter. The array is an array of **Int**. The size of the array is 30,000 and has to be circular. The pointer will be implemented as a Int representing the index of the pointed cell. The stored values will be between **0** and **255** included. Remember to handle the overflow (value **256** passes **0** and the value **-1** passes **255**).

2 Dictionary

2.1 Definition

The Dictionary generic class provides a mapping from a set of keys to a set of values. It is a powerful tool which allows the user to retrieve and insert values through a key in a constant time. This efficiency exists because the class is implemented as a Hash table. To learn more about the implementation or the key comparison system, read the "Notes" part at: `https://msdn.microsoft.com/fr-fr/library/xfhwa508(v=vs.110).aspx`

2.2 Utilisation

You will be asked to use the Dictionary class in order to implement a brainfuck interpreter handling a dynamic set of symbols. You will not need to handle the more difficult aspects of the class and everything you need is present in the following example.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys.
        Dictionary<string, string> acdcs =
            new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        acdcs.Add("Zarakailloux", "The unknown");
        acdcs.Add("Kirszie", "El Major");
        acdcs.Add("Heldhy", "Cuttie.exe");

        // The Add method throws an exception if the new key is
        // already in the dictionary.
        try
        {
            acdcs.Add("Heldhy", "NoOneWillNotice");
        }
        catch (ArgumentException)
        {
            Console.WriteLine("An acdc with Key = \"Heldhy\" already
            exists.");
        }
```

```
1           // The Item property is another name for the indexer, so you
2           // can omit its name when accessing elements.
3           Console.WriteLine("For key = \"Zarakailloux\", value = {0}.",
4               acdcs["Zarakailloux"]);
5
6           // The indexer can be used to change the value associated
7           // with a key.
8           acdcs["Zarakailloux"] = "JSON master";
9           Console.WriteLine("For key = \"Zarakailloux\", value = {0}.",
10              acdcs["Zarakailloux"]);
11
12          // If a key does not exist, setting the indexer for that key
13          // adds a new key/value pair.
14          acdcs["DotH"] = "Source files destroyer";
15
16          // The indexer throws an exception if the requested key is
17          // not in the dictionary.
18          try
19          {
20              Console.WriteLine("For key = \"Gollum\", value = {0}.",
21                  acdcs["Gollum"]);
22          }
23          catch (KeyNotFoundException)
24          {
25              Console.WriteLine("Key = \"Gollum\" is not found.");
26          }
27          // When a program often has to try keys that turn out not to
28          // be in the dictionary, TryGetValue can be a more efficient
29          // way to retrieve values.
30          string value = "";
31          if (acdcs.TryGetValue("Coucou", out value))
32              Console.WriteLine("For key = \"Coucou\", value = {0}.", value);
33          else
34              Console.WriteLine("Key = \"Coucou\" is not found.");
35
36          // ContainsKey can be used to test keys before inserting
37          // them.
38          if (!acdcs.ContainsKey("TheCoon"))
39          {
40              acdcs.Add("TheCoon", "Ninja");
41              Console.WriteLine("Value added for key = \"TheCoon\": {0}",
42                  acdcs["TheCoon"]);
43          }
```

```
        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach (KeyValuePair<string, string> kvp in acdcs)
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }

        // Use the Remove method to remove a key/value pair.
        Console.WriteLine("\nRemove(\"TheCoon\")");
        acdcs.Remove("TheCoon");

        if (!acdcs.ContainsKey("TheCoon"))
            Console.WriteLine("Key \"TheCoon\" is not found.");
    }
}

/* This code example produces the following output:

An acdc with Key = "Heldhy" already exists.
For key = "Zarakailloux", value = The unknown.
For key = "Zarakailloux", value = JSON master.
Key = "Gollum" is not found.
Key = "Coucou" is not found.
Value added for key = "TheCoon": Ninja

Key = Zarakailloux, Value = JSON master
Key = Kirszie, Value = El Major
Key = Heldhy, Value = Cuttie.exe
Key = DotH, Value = Source files destroyer
Key = TheCoon, Value = Ninja

Remove("TheCoon")
Key "TheCoon" is not found.
 */
```

3 JSON

3.1 Introduction

The JSON, or JavaScript Object Notation, is a data format that was derived from Javascript. It is used to represent data in a human-readable format. It is structured as XML or YAML would be for example. The main advantage is the fact that it is really easy to read JSON with bare eyes and to understand it. It is also really easy to handle it as its rules are quite simple to understand. However, JSON has some limitations. For example, the fact that comments are not part of the official JSON definition, or that only simple types exist.

3.2 Behaviour

In JSON, there are 6 different types of values. We are going to present them in a simplified version:

- The character chain. It begins with a double-quote '"' and ends with another one. If we want to put a double quote inside, we must escape it with a backslash '\'
- The number. For this exercise, we will consider that it may be beginning with an optional '-' (minus), and then one or many digits. We will consider that they are only whole number.
- The boolean. It is either "true" or "false", in lowercase and without quotes.
- The null value. It is simply "null", without quotes.
- The list. JSON's lists, unlike C# lists, may contain any type of values mixed in the same list. It begins with a '[' and ends with a ']'. Values are comma separated. We will consider that a list cannot be empty.
- The object, or the dictionary. This is an extension of a list, but every value has a unique key, exactly like the C# dictionary presented in part 2. The key must be a JSON string, while the value can be any JSON element. Dictionaries begin with '{' and end with '}'. Values are also comma separated. Elements in the dictionary follow the syntax "key : value".

A JSON file must contain only one dictionary that will contain the rest.

Look at the following examples:

```
{
    "BEST_ACDC": 2020,
    "Campus": "Villejuif",
    "rooms":
    [
        306,
        "Lab"
    ],
    {
        "TPC#" : true,
        "TPCAML": false,
        "Studends that had 0": null
    }
}
```

Indentation is not taken into account and is used only to make it more readable. That is, the following JSON is also valid:

```
1 {"key":"value",[1,2,3]}
```

For more details, visit the `http://json.org` website

4 Exercises

4.1 Brainfuck

It is much too easy for an intellect like yours to code a brainfuck interpreter and we are aware of this. This is reason why your interpreter will be able to manage a **dynamic** mapping of symbols. In order to do that, every function takes two arguments: a string and an object Dictionary<char, char>. The object dictionary<char, char> represents the mapping <**classic symbol**, **esoteric symbol**>. **Every** operation and comparison between a symbol and a character must be done through the dictionary. As example:

```
1 if (code[i] == symbols['>'])
2     do_something();
```

Arguments are given to the functions through the IDE. For now if you want to have fun with the symbols, check out the function *ClassicBrainfuck* in Program.cs

4.1.1 I did not mean that

```
1 public static string Interpret(string code, Dictionary<char, char> symbols)
```

The method **Interpret** interprets the brainfuck code given as parameter with the mapping in **symbols**. Several steps :

- Declaration of variables according to the specifications of implementation defined above.
- Initialisation of the array.
- For each character execute the corresponding instruction.

If you hit a caracter that is not part of the dictionnary (an error), you must throw an exception. The exception doesn't matter, as long as it makes sense.
Protip : for the management of loops you can use a **Stack** (see MSDN and algo courses) to store their positions in the code.

4.1.2 What do you mean

```
1 public static string GenerateCodeFromText(string text,
2                                            Dictionary<char, char> symbols)
```

This method returns the Brainfuck code to write the sentence contained in text. Of course the returned code must follow the mapping given in symbols.

Protip : The code "[-]" resets the current cell at 0.

4.1.3 The size does count

```
1 public static string ShortenCode(string program,
2                                   Dictionary<char, char> symbols)
```

ShortenCode must return a functional code that is doing the same thing as the original code but with fewer characters. You can use all the techniques you want as long as the product code is shorter than the original code.

For the mandatory part of this exercise you have to spot the long sequences of **symbols**['+'] and replace them with multiplication (with the help of loops). If other techniques are used, please mention it in the Readme to complete the Bonus.
Obviously, the generated code will strictly have the same behaviour.

4.2 JSON

Those exercises will guide you through the implementation of a JSON parser in order to read JSON file, and create the associated object in C#.
We will consider the JSON input file as valid and written without any error.
For the JSON exercises, using any C# JSON library's function is strictly forbidden and will be considered as cheating.

4.2.1 JSONElement

To represent a JSON object (one of the six that exist), we're going to use a class JSONElement, which is provided. Using this class, we will be able to use a List<> of JSONElement which will be helpful and simplify your code a lot. We define an enum JSONType which contains the different types of JSON's elements.
The JSONElement class contains a type, and many public variables. Using the type of the object, we will modify only the associated variable. For example, if our JSONElement object's type is "string", we will only access and modify the string_value variable.

4.2.2 GetJsonType

```
1  public static JSONElement.JSONType GetJsonType(char c)
```

This function is used to determine the type of an element based on the first char (the argument c) of its JSON reprensentation, and to return it.

4.2.3 ParseString

```
1  public static string ParseString(string json, ref int index)
```

This function takes a string and an index by reference, and reads a JSON string. The index will point on the first '"' of the JSON string to be read.

For example:

```
1  int i = 2;
2  string input = "{ \"ACDC\": 2020}";
3  input[i] // '"'
4  ParseString(input, ref i); //returns "ACDC" and i egals 8
```

4.2.4 ParseInt

```
1  public static int ParseInt(string json, ref int index)
```

The same behaviour is expected, but to parse a JSON int this time.

4.2.5 ParseBool

```
1  public static bool ParseBool(string json, ref int index)
```

The same behaviour is expected, but to parse a JSON boolean this time.

4.2.6 EatBlank

```
1 public static void EatBlank(string json, ref int index)
```

This function will increase the index passed by reference until the pointed char is no longer blank (that is, every char that is not a space, a tabulation, etc).

4.2.7 ParseJSONString

```
1 public static JSONElement ParseJSONString(string json, ref int index)
```

The big function that takes a string and an index, and parses the next JSON element.
Remember to use the previously coded functions.

4.2.8 ParseJSONFile

```
1 public static JSONElement ParseJSONFile(string file)
```

This function takes a path to a valid JSON file as input, and returns the associated JSONElement successfully parsed.

4.2.9 PrintJSON

```
1 public static void PrintJSON(JSONElement el)
```

This function is used to display a JSONElement. It will be displayed as a valid JSON, but spaces and tabulations are not mandatory (see the bonus part).

4.2.10 SearchJSON

```
1 public static JSONElement SearchJSON(JSONElement element, string key)
```

Will look in the JSONElement to find an element but its key. Keep in mind that the element might be in a dictionary within another dictionary, etc. If the element is not found, you have to return **null** (the *C#* null value).

4.2.11 Brainfuck with options - BONUS

Now that **SearchJSON** is implemented you can load a JSON containing a mapping of the classic brainfuck symbols as keys, to esoteric ones. The **load brainfuck** button will change the current mapping to the one in the JSON.

4.2.12 Pretty PrintJSON - BONUS

Modify your PrintJSON function to make it display your JSON while respecting the conventional JSON representation (space and tabs). Use the website `http://jsonprettyprint.com` to help you.
Be careful ! You must not modify the function prototype, but you may create subfunctions.

4.2.13 ParserJSON - BONUS

As a bonus, you can implement a JSON parser that handles JSON syntax errors and that throw exceptions. If you detect an incorrect JSON syntax, you may throw any kind of exception that make sense.

4.2.14 Ook ! - BONUS

Adapt your code in order to be able to understand the **Ook!** language (`http://www.dangermouse.net/esoteric/ook.html`) with the right JSON configuration file.

These violent deadlines have violent ends.