# TP C#9: We have ways of making you talk!

## 1 Guidelines for handing in

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

```
rendu-tp-login_x.zip
+-- strtok_r/
    +-- AUTHORS*
    +-- README
    +-- tp15.sln*
    +-- tp15*
        +-- Everything but bin/ and obj/
```

Of course, you must replace "`login_x`" with your own login. All the files marked with an asterisk are *MANDATORY*.

Don't forget to 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, otherwise you will get a 0!**

# Contents

# 2 Introduction

You should read the whole subject carefully before starting. **No function is allowed unless explicitly indicated in the guidelines or by assistants.**

## 2.1 Goals

The purpose here is to make apply use all the notions you have seen so far, but also make you think by yourself about some problems without assistants handling your hand.

In this practical, you will connect to a server, ask for a question, solve it, and send it back. This kind of box will appear along the subject:

```
=> "login_x|Ex0:"
<= "OK:Hello World!"
=> "login_x|Ex0:Salut le Monde !"
<= "OK:Well Done!"
```

This illustrates the communication between you and the server.

- => : Stands for a request you send
- <= : Stands for an answer you receive

Here is the protocol you use:

- A request begins with you login, followed by "|", the exercise name, ":" and possibly a message.
- Each server answer begins with "OK" or "KO", followed by ":" and the message.
- First send a query in order to receive the question of the exercise
- If the server answers "OK", the message contains the question, otherwise it is an error message.
- Send back your solution
- If the server sends back "OK" as the 2 first characters, your answer is correct.

Here is a teaser:

**Palier 0** First contact with the server

**Palier 1** Basic exercises

**Palier 10** *Brainfuck*, a new language

**Palier 11** Your best tool ever is your brain

**Palier 100** Ask for some help the server

# 3 Level 0

## 3.1 Exercise 0.a : Client

### Goals

Setup a connection between you and the server.
Prerequisite:

**TcpClient** This object represents the client over the network according to the TCP protocol, which guarantees the integrity and the order in which the message reaches the server.

**IPEndPoint** This object is the IP address and the port of the server.

**NetworkStream** This object is a network stream (Captain Obvious struck again) that you shared with the server in which you can read and write messages.

**Stopwatch** Does Captain Obvious need to explain?

**byte[ ]** A *byte* is composed by 8 bits, a *byte[]* is a byte array. A *char* is encoded on a byte.

Example :

- The server is local, listening on port 80
- We are going to send a message and receive the answer
- If the server takes more than 5 seconds to answer, the exchange ends.

```
public void main()
{
    TcpClient tcpClient = new TcpClient();
    IPEndPoint serverAddress = new IPEndPoint(
        IPAddress.Parse("127.0.0.1"), 80);

    tcpClient.Connect(serverAddress);

    byte[] msg = some_function_you_have_to_code(
        "My string message");

    NetworkStream ns = tcpClient.GetStream();

    ns.Write(msg, 0, msg.Length);
    ns.Flush();

    Stopwatch clock = new Stopwatch();
    s.Start();

    byte[] ans = new byte[4096];
    int ans_size = 0;

    while (clock.ElapsedMillisecond < 5000)
    {
        if (ns.DataAvailable)
        {
            ans_size = ns.Read(ans, 0, 4096);
            break;
        }
    }

    Console.WriteLine("Message received (length = {0}): {1}",
        ans_size, some_function_you_have_to_code(ans));
}
```

```
public string some_function_you_have_to_code(byte[] msg)
{
    /* FIXME: return a string corresponding to the byte array */
}

public byte[] some_function_you_have_to_code(string msg)
{
    /* FIXME: return a byte array corresponding to the string */
}
```

You should be able to code the *Client* class.

Be careful and manage errors (*try ... catch*).

```
Class Client
{
    #region Attributes
    /* FIXME */
    /* Login */
    /* TcpClient */
    /* IPEndPoint */
    /* ... */
    #endregion

    #region Constructor
    public Client(/* FIXME */)
    {
        /* FIXME: initialize attributes */
    }
    #endregion

    #region Methods
    public bool connect()
    {
        /* FIXME: Connect the client to the serveur */
    }

    public void send(/* FIXME */)
    {
        /* FIXME: Send the request to the server */
        /* accroding to the protocol */
    }

    public String receive(/* FIXME */)
    {
        /* FIXME: Receive and return the server response */
    }

    public bool process(Ex ex)
    {
        /* FIXME: Solve the exercise */
        /* return true or false, according to the result */
        /* given by the server */

        /* return true  : The answer is correct */
        /*        false : The answer is wrong */
    }

    public bool process(List<Ex> exs)
    {
        /* FIXME: Solve all the exercises */
        /* return true  : All the answers are correct */
        /*        false : One of the answers is wrong */
    }
    #endregion
}
```

The *process* method solves the exercise accordingly to the protocol:

1. Ask the server the question

2. Call *solve*
3. Send the answer back to the server

### 3.2 Exercise 0.b : ExX

#### Goals

For each exercise you have to create a class. In order to factorize the code, create a parent class, which contains the skeleton of the exercises. Then each exercise will inherit this parent class.

**Property** Exercise name

**Constructor** Takes the exercise name as parameter

**Methode** One abstract method *solve* which takes a question (*string*) as parameter.

***INFO*** *: Note that the abstract keyword means we provide only the prototype of the method but not the implementation. The content depends on the exercise inherited from the* ExX *abstract class.*

```
abstract class ExX
{
    #region Attributes
    /* FIXME */
    /* Exercise's name*/
    #endregion

    #region Constructors
    public Ex(/* FIXME */)
    {
        /* FIXME: Initialize attributes */
    }
    #endregion

    #region Methods
    public abstract string solve(string question);
    /* FIXME */
    #endregion
}
```

### 3.3 Exercise 0.c : Ex0

#### Goals

Create the *Ex0* class, inherited from *ExX*.
Override the *solve* method, which takes the question *"Hello World!"* as parameter and must answer *"Salut le Monde !"*.
Connect to the server, ask for *"Ex0"*, answer the result of the *solve* method.

#### Example

Instanciate an *Ex0* and solve it:

```
Ex1 e = new Ex1();
client.process(e);
```

Here is an example of communication.

```
=> "login_x|Ex0:Some text"
<= "KO:Fail!"

=> "login_x|Ex0:"
<= "OK:Hello World!"
=> "login_x|Ex0:Some answer"
<= "KO:Fail!"

=> "login_x|Ex0:"
<= "OK:Hello World!"
=> "login_x|Ex0:Salut le Monde !"
<= "OK:Success!"
```

# 4 Level 1

## 4.1 Exercise 1 : Banana Splif

### Goals

You create the class Ex1 and redefine the function solve.

The server sends you a string, you send the message between two ';'.

### Example

Here is an example of communication.

```
=> "login_x|Ex1:Some text"
<= "KO:Fail!"

=> "login_x|Ex1:"
<= "OK:abc;def;ghi"
=> "login_x|Ex1:def"
<= "OK:Success!"
```

## 4.2 Exercise 2 : Banana Spliff

### Goals

You create the class Ex2 and redefine the function solve.

The server sends you a string, you send the part between both delimiters. The first char from the message is the delimiters.

### Example

Here is an example of communication.

```
=> "login_x|Ex2:Some text"
<= "KO:Fail!"

=> "login_x|Ex2:"
<= "OK:.abc.def.ghi"
=> "login_x|Ex1:def"
<= "OK:Success!"

=> "login_x|Ex2:"
<= "OK:*Bmoc*AcDc*SKORM"
=> "login_x|Ex1:AcDc"
<= "OK:Success!"
```

### 4.3 Exercise 3 : 4701+

#### Goals

You create the class Ex3 and redefine the function solve.

Transform both parts of the string (split with an "+") into integers then send their sum.

#### Example

Here is an example of communication.

```
=> "login_x|Ex3:Some text"
<= "KO:Fail!"

=> "login_x|Ex3:"
<= "OK:37+5"
=> "login_x|Ex3:42"
<= "OK:Success!"

=> "login_x|Ex3:"
<= "OK:13+37"
=> "login_x|Ex3:1337"
<= "KO:Fail!"
```

### 4.4 Exercise 4 : Calculator 2006

#### Goals

You create the class Ex4 and redefine the function solve.

You will receive a string in which the first char is the delimiter. When both parts are split into two integers (cf. Exercise 3), you apply the operation fixed by the delimiter, then send the result.

Operation/Delimiter used: "+", "-", "*", "/", "%"

#### Example

Here is an example of communication.

```
=> "login_x|Ex4:Some text"
<= "KO:Fail!"

=> "login_x|Ex4:"
<= "OK:-84-42"
=> "login_x|Ex4:42"
<= "OK:Success!"

=> "login_x|Ex4:"
<= "OK:/127/3"
=> "login_x|Ex4:42"
<= "OK:Success!"

=> "login_x|Ex4:"
<= "OK:%42%2"
=> "login_x|Ex4:42"
<= "KO:Fail!"
```

### 4.5 Exercise 5 : sentence! the Reverse

#### Goals

You create the class Ex5 and redefine the function solve.

Reverse word order (each one separated by one *space*, but not the letters order) in the message sent the server.

#### Example

Here is an example of communication.

```
=> "login_x|Ex5:Some text"
<= "KO:Fail!"

=> "login_x|Ex5:"
<= "OK:The sentence is reversed!"
=> "login_x|Ex5:reversed! is sentence The"
<= "OK:Success!"

=> "login_x|Ex5:"
<= "OK:Reverse me!"
=> "login_x|Ex5:!em esreveR"
<= "KO:Fail!"
```

### 4.6 Exercise 6 : My name is Bound, Boundaries

#### Goals

You create the class Ex6 and redefine the function solve.

From the message sent by the server, fill an *int[]*, whose values are between delimiters (first char from the string) in the message, sort in ascending order, then send its representation keeping the same delimiter.

#### Example

Here is an example of communication.




```
=> "login_x|Ex6:Some text"
<= "KO:Fail!"

=> "login_x|Ex6:"
<= "OK:.5.4.3.2.1"
=> "login_x|Ex6:.1.2.3.4.5"
<= "OK:Success!"

=> "login_x|Ex6:"
<= "OK:/45/0/98461"
=> "login_x|Ex6:.0.45.98461"
<= "KO:Fail!"

=> "login_x|Ex6:"
<= "OK: 1 3 5 3 1 0 1 3 5 3 1"
=> "login_x|Ex6: 0 1 1 1 1 3 3 3 3 5 5"
<= "OK:Success!"
```

# 5 Level 10

## 5.1 Brainfuck ? WutzIt ?

*Brainfuck* is a "fun" language not easy to read. It uses only 8 instructions (we will implement only 7) and an *Hello_World* in *Brainfuck* looks something like this:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+
[<]<-]>>.>---.+++++++..+++.>>.<-.<.+++.----
--.--------.>>+.>++.
```

***INFO** : The full interpreter is split into three parts (Exercise 7, 8 and 9).*

## 5.2 Exercise 7 : +-.-+

### Goals

Implement the class *Ex7* (inheriting from *ExX*). You will redefine the solve method.

In this exercise, the *Brainfuck* interpreter *Brainfuck* will only understand :

'+' Increment the memory cell pointed by the pointer

'-' Decrement the memory cell pointed by the pointer

'.' Add the current memory cell to the output string (ASCII representation).

```
class Ex7 : ExX
{
    #region Attributes
    byte[] memory;  /* Program memory */
    int    pointer; /* Index in memory */
    int    ip;      /* Instruction pointer (Index in code) */
    string output;  /* Program output */
    string code;    /* Brainfuck code to process */
    /* FIXME */
    #endregion

    #region Constructor
    public Ex7(string name) /* FIXME */
    {
        /* FIXME */
    }
    #endregion

    #region Methods
    public override string solve(string question)
    {
        /* FIXME: Some initialization, memory = 0, ip = 0, pointer = 0
           */

        /* FIXME: Loop */ (/* FIXME */)
        {
            /* FIXME */
            /* Store the instruction */
            /* Move forward the instruction pointer */
            /* Process the instruction */
        }

        return output;
    }

    private void process_instr(/* FIXME */)
    {
        /* FIXME: Do something according to */
        /* the current instruction */
    }

    /* FIXME */
    #endregion
}
```

### Example

Here is an example of communication.

```
=> "login_x|Ex7:Some text"
<= "KO:Fail!"

=> "login_x|Ex7:"
<= "OK:+++++++++++++++++++++++++++++++++++++++.-.+."
=> "login_x|Ex7:*)*"
<= "OK:Success!"
```

### 5.3 Exercise 8 : Shift, reduce

#### Goals

Some of you have seen that only one memory cell is used.
Add these two instructions to your interpreter :

'>' Increments the memory index

'<' Decrements the memory index

#### Example

Here is an example of communication.

```
=> "login_x|Ex8:Some text"
<= "KO:Fail!"

=> "login_x|Ex8:"
<= "OK:+++++++++++++++++++++++++++++++++++++++.>++++++++++++++++++++++++
++++++++++++++++++++.<."
=> "login_x|Ex8:*.*"
<= "OK:Success!"
```

### 5.4 Exercise 9 : LOOPLOOPLOOPLOOP...

#### Goals

$$8 - 3 - 2 = 2$$

Add these two last instructions to your interpreter :

'[' Move the instruction pointer after the corresponding ']' if the memory value pointed by the memory pointer is 0. In other cases nothing happens

']' Move the instruction pointer after the corresponding '[]' if the memory value pointed by the memory pointer is not 0. In other cases nothing happens

***INFO*** *: These instructions can be nested*

### Example

Here is an example of communication.

```
=> "login_x|Ex9:Some text"
<= "KO:Fail!"

=> "login_x|Ex9:"
<= "OK:++++++++[>+++++<-]>++<+++[>.<-]"
=> "login_x|Ex9:***"
<= "OK:Success!"
```

# 6 Level 11

## 6.1 Exercise A : + || -

### Goals

Here you have the function that the server will use to test your answer.
Find the mistake that allows the server to send you "OK".
You will explain your steps in the *README*.
***INFO*** *::* answer *is the message that you send to the server.* "validation" *is it answer.*

```
public static string solve_1(string answer)
{
    int nb = 0;

    try { nb = int.Parse(answer); }
    catch { return "KO:Wrong integer format"; }

    string[] tab = {"KO:You lose!", "KO:Nope!", "KO:LOLNO", "OK:GGWP"};

    string validation = "KO:Nice Tried!";

    if (nb >= tab.Length - 1)
        return validation;

    for (int i = 0; i < tab.Length && nb != 0; i++, nb--)
        validation = tab[i];

    return validation;
}
```

### Example

Here is an example of communication.

```
=> "login_x|ExA:Some text"
<= "KO:Wrong integer format!"

=> "login_x|ExA:8"
<= "KO:Nice Tried!"

=> "login_x|ExA:1"
<= "KO:You lose!"
```

### 6.2 Exercise B : return 4

#### Goals

Here you have the function that the server will use to test your answer.
Find the mistake that allows the server to send you "OK".
You will explain your steps in the *README*.
***INFO*** *:* answer *is the message that you send to the server.* secret *et* incr *change at each connection to the server.*

```
public static string solve_2(string answer, ref int secret, int incr)
{
    int nb = 0;

    try { nb = int.Parse(answer); }
    catch { return "KO:Wrong integer format"; }

    int current = secret;
    secret += incr;

    if (current == nb)
        return "OK:EZ";
    else
        return string.Format("KO:{}", current);
}
```

#### Example

Here is an example of communication.

```
=> "login_x|ExB:Some text"
<= "KO:Wrong integer format!"

=> "login_x|ExB:42"
<= "KO:27"
=> "login_x|ExB:27"
<= "KO:42"
```

## 7 Level 100

### 7.1 Exercise C : THE GAME

#### Goals

Only the server can help you.
You will explain your steps in the *README*.

**The bullshit is strong with this one...**