TP C#1 : My First Bernard

Consignes de rendu

A la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
rendu-tp1-prenom.nom.zip
|-- prenom.nom/
    |-- AUTHORS
    |-- README
    |-- TP1/
        |-- TP1.sln
        |-- TP1/
            |-- Everything except bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login et n'oubliez pas le fichier `AUTHORS`.
— Les fichiers `AUTHORS` et `README` sont obligatoires.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (*), un espace, votre login et un retour à la ligne. Voici un exemple (où `$` est un retour à la ligne et ␣ un espace) :

```
*␣firstname.lastname$
```

Notez que le nom du fichier est `AUTHORS` sans extension. Pour créer simplement un fichier `AUTHORS` valide, vous pouvez taper la commande suivante dans un terminal :

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

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

1 Introduction

Ce TP a pour but de vous apprendre d'une part ce qu'est une console ainsi que son mode d'utilisation, et d'autre part à utiliser la classe Console de la bibliothèque C# pour interagir avec la console.

2 La console Linux

2.1 Différence CLI/GUI

Durant votre vie d'utilisateur d'ordinateur, la plupart d'entre vous n'ont utilisé que des programmes graphiques. Ces programmes se contrôlent en général grâce à la souris à travers une interface graphique, aussi appelée GUI (pour Graphical User Interface), avec laquelle on peut interagir grâce à des boutons, des onglets, des champs de texte, etc.

Il existe néanmoins une autre manière d'interagir avec son ordinateur. Une manière plus ancienne, pouvant paraître archaïque aujourd'hui mais étant en général plus efficace : la ligne de commande, aussi appelée CLI.

Certains programmes sont fait pour être utilisé avec la ligne de commande. Ils s'exécutent dans ce que l'on appelle un terminal.

2.2 La ligne de commande

2.2.1 Interpreteur Bash

Maintenant que nous avons ouvert un terminal, nous pouvons examiner ce qu'il y a à l'intérieur. Au démarrage du terminal, celui-ci affiche quelques informations.

```
[42sh]$
```

Cet affichage est ce qu'on appelle le prompt. C'est un exemple, il est possible que votre prompt ne soit pas le même que celui-ci.

Le programme exécuté dans le terminal est en fait un interpréteur shell. Ce programme attend des commandes de l'utilisateur pour ensuite pouvoir les exécuter (vous avez déjà rencontré un interpréteur, repensez au Caml !).

2.2.2 Les commandes de base

Dans son état actuel, l'interpréteur shell est en attente de commandes de la part de l'utilisateur. Ça tombe bien, l'utilisateur ici c'est vous !

Une commande est généralement composée d'un nom de programme suivi par zéro ou plusieurs arguments.
Exercice : essayer d'entrer la commande `echo "Hello World!"` dans votre terminal. Quel a été l'effet de la commande ?

- `ls` (LiSt) :
  `ls` est une commande permettant de lister les fichiers contenus dans le dossier courant (par défaut, votre shell s'ouvre dans votre dossier `home/`).
  `ls` peut prendre comme argument des dossiers. `ls` listera alors le contenu de chacun des fichiers.
- `cd` (Change Directory)
  `cd` est une commande vous permettant de se déplacer vers le fichier donné en argument.
- `echo`
  `echo` affiche ses arguments dans le terminal.

— `cat` (conCATenate)
`cat` permet d'afficher le contenu des fichiers lui étant donné en argument.
— `mkdir` (MaKe DIRectory)
`mkdir` est une commande vous permettant de créer un dossier. Le nom du dossier est donné par l'argument. Si on donne plusieurs arguments, plusieurs dossiers seront créés en conséquence.
— `man` (MANual page)
`man` permet d'afficher la page du manuel de la commande donnée en argument. Cette commande est votre MEILLEURE AMIE en ce qui concerne le shell (et elle vous suivra tout au long de vos études à Epita).

Nous vous conseillons de lire les pages du manuel de chacune des commandes ayant été décrites plus haut. Vous pourrez découvrir que ces commandes disposent de beaucoup d'options permettant de faire facilement ce qui serait considéré comme de la magie noire sur n'importe quel explorateur graphique de fichier.

2.2.3 Les flux standards

La console utilise des canaux pour communiquer avec l'utilisateur, notamment trois canaux que l'on appelle les canaux standards.

— `stdin` :
L'entrée standard (ou Standard Input) est le canal qui est lu par la console. C'est par ce canal que l'utilisateur peut entrer des informations (comme des commandes par exemple. Un programme en ligne de commande pourra lire des informations à partir de ce canal.
— `stdout` :
La sortie standard (ou Standard Output) est le canal ou la console va écrire des informations. Ce canal peut ensuite être lu par l'utilisateur (notons que l'utilisateur peut être un autre programme !). Un programme en ligne de commande pourra écrire des informations dans ce canal.
— `stderr` :
La sortie d'erreur (ou Standard Error) est un canal très similaire à la sortie standard. Il permet à la console d'écrire des informations relatives à des erreurs d'exécution. Un programme en ligne de commande pourra également écrire des informations à propos de ses erreurs dans ce canal.

2.3 AUTHORS comme les pros

Pour pouvoir un peu vous exercer à l'utilisation de la console, nous vous proposons d'essayer de faire votre fichier `AUTHORS` comme un pro. Suivez les étapes, et n'hésitez pas à questionner vos ACDC en cas de problème.

2.3.1 Déplacement

Pour rappel, l'AFS (pour Andrew File System) est un système de fichier distribué, qui vous permet d'accéder à vos fichiers depuis n'importe quelle machine du parc informatique d'EPITA. À chaque démarrage, votre partie de l'AFS est montée sur votre système de fichier (un peu comme une clé USB). Faites donc bien attention : toutes données qui ne sont pas enregistrées sur votre AFS seront perdues quand vous éteindrez votre machine !

Pour commencer, allez dans votre AFS en se servant de la commande `cd`.

```
cd afs
```

Vous êtes maintenant dans votre dossier afs. Vous pouvez voir ce qu'il contient en exécutant la commande `ls`.

```
ls
```

2.3.2 Création du dossier

Créez votre dossier de rendu. Pour cela, vous allez créer un dossier avec la commande `mkdir`.
Nous vous faisons confiance pour remplacer "prenom.nom" par votre prénom et votre nom.
Si vous exécutez la commande `ls`, vous pourrez voir votre nouveau dossier. Entrez dans ce dossier avec la commande `cd` :

```
cd prenom.nom
```

Ca y est, vous êtes dans votre dossier de rendu !

2.3.3 Création du fichier AUTHORS

Maintenant que vous êtes dans votre dossier de rendu, créez votre fichier `AUTHORS`. Pour cela, exécutez la commande suivante :

```
echo "* prenom.nom" > AUTHORS
```

C'est une commande un peu complexe. Pour faire simple, la commande echo va écrire dans la sortie standard, sortie standard qui est redirigée dans le fichier `AUTHORS` (qui est crée vu qu'il n'existe pas encore).
Vous n'avez pas besoin de comprendre cette commande, mais vous pouvez déjà vous rendre compte de la puissance de la console. Faire la même opération avec une interface graphique aurait pris d'une part plus de ressources (ouvrir un éditeur...), d'autre part plus de temps.

2.3.4 Vérification du AUTHORS

Pour vérifier que vous n'avez pas fait d'erreur, vous pouvez afficher le contenu du fichier `AUTHORS` avec la commande `cat`.

```
cat AUTHORS
* prenom.nom
```

Néanmoins, vous voulez aussi vérifier qu'il n'y a pas de caractères invisibles indésirables dans votre AUTHORS. Nous allons donc utiliser une option de la commande `cat`, l'option `-e` :

```
cat -e AUTHORS
* prenom.nom$
```

Cette option permet d'afficher certains caractères qui sont invisibles normalement. Ici, le retour à la ligne est représenté par un symbole `"$"`.

3 La console et le C#

Durant la plupart des TPs C#, nous allons vous demander de développer des programmes pour la console. Il existe en C# plusieurs moyens pour interagir avec son utilisateur via la ligne de commande. Nous allons vous en présenter quelques uns.

3.1 Récupérer les arguments donnés à son programme

Comme vous avez pu le voir plus haut, vous pouvez lancer à partir de la console des programmes. Certains de ces programmes ont besoin d'arguments (pensez à `mkdir` par exemple, qui a besoin d'un argument pour nommer le dossier crée).

Vos programmes peuvent aussi récupérer les arguments que l'on leur donne dans la console. On peut récupérer ces arguments grâce à l'argument "`args`" de la fonction `Main`.

```
1  public static void Main(string[] args)
```

L'argument "args" de la fonction Main est un tableau, c'est une notion que vous verrez dans quelques semaines.

3.2 La classe Console

Pour interagir avec la console en C#, vous allez utiliser une classe de la bibliothèque C# permettant d'utiliser les flux standards : la classe `Console`.

Cette classe contient des méthodes permettant d'écrire ou de lire dans la console. Elle contient aussi des méthodes permettant de modifier les propriétés de celle-ci.

Nous vous invitons à lire la page MSDN de cette classe.

3.3 Écrire dans la sortie standard

En général, les premiers pas d'un apprenti programmeur est d'afficher un message dans la sortie standard. Vous avez déjà tous écrit un programme "Hello World !" lors du TP C#0.

La classe Console de C# contient deux fonctions permettant d'écrire dans la console.

— `Console.Write(arg)`
Cette méthode convertit `arg` en chaîne de caractère, puis l'affiche dans la sortie standard. Lisez la page MSDN pour plus d'informations.

— `Console.WriteLine(arg)`
Pour rappel, cette méthode convertit `arg` en chaîne de caractère, puis l'affiche dans la sortie standard suivi d'un saut de ligne. Lisez la page MSDN pour plus d'informations.

Exemple :

```
1  using System;
2
3  namespace TP1
4  {
5      internal class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("Hello Bernard.");
10             Console.WriteLine(42);
11             Console.Write("Hello");
12             Console.Write("Hello\n");
13             Console.WriteLine("What door?");
14         }
15     }
16 }
```

Résultat :

```
Hello
42
HelloHello
What door?
```

3.4 Lire dans l'entrée standard

La prochaine étape dans l'interaction avec l'utilisateur est de pouvoir lire des informations entrées par celui-ci.

La classe Console de C# contient deux fonctions permettant de lire dans la console.

— Console.Read()
Cette méthode lis le prochain caractère disponible dans l'entrée standard, et le renvoie sous forme d'un `int`.
Dans la pratique, cette méthode attend que l'utilisateur entre un caractère. Une fois le caractère entré par l'utilisateur, le programme reprend.

— Console.ReadLine()
Cette méthode lit la prochaine ligne disponible dans l'entrée standard et la renvoie sous la forme d'une chaîne de caractères, sans le caractère de retour à la ligne finale.
Dans la pratique, cette méthode attend que l'utilisateur tape du texte et appuie sur entrée, puis le programme reprend.

Exemple :

```
using System;

namespace TP1
{
    internal class Program
    {
        public static void Main(string[] args)
        {
            Console.Write("Please enter a char: ");
            int c = Console.Read();
            Console.WriteLine("\nYou typed the char: " + (char)c);
        }
    }
}
```

3.5 Autres options de la console

La classe `Console` contient d'autres méthodes et attributs qui vous seront utiles dans la suite du TP. Nous vous invitons à lire les pages MSDN correspondantes.

- `Console.Clear()`
- `Console.Beep()`
- `Console.ResetColor()`
- `Console.BackgroundColor`
- `Console.ForegroundColor`

4 La boucle While

4.1 Le principe de la boucle itérative

Un des principes essentiels de la programmation est de répéter une action un certain nombre de fois.

Jusqu'à maintenant, vous utilisiez (et êtes censé maîtriser) le principe de la récursion pour réaliser des actions répétées.

Néanmoins, il existe une autre manière de répéter des actions en algorithmique : utiliser les boucles itératives.

Une boucle est un bloc de code qui peut être répété. Le principe est simple : une fois arrivé à la fin de la boucle, l'exécution du code continue au début de la boucle.

```
i = 0
begin loop
        i = i + 1
        print "hello"
end loop
print i
```

Dans cet exemple en pseudo-code, le code commence à être exécuté à partir de "begin loop". On parcourt la boucle ligne par ligne. La première ligne de la boucle ajoute 1 à la variable i, qui est donc maintenant égale à 1. La deuxième ligne affiche "hello". On arrive à la fin de la boucle. À ce moment là, on va revenir au début de la boucle. Et on recommence à la première ligne de la boucle. La première ligne ajoute 1 à la variable i, qui est maintenant égale à 2. La deuxième ligne affiche "hello". On arrive à la fin de la boucle, on revient au début de celle-ci, et ainsi de suite.

4.2 La boucle conditionnelle

Ce que l'on peut remarquer dans ce premier exemple, c'est que cette boucle ne va jamais s'arrêter. La variable i va augmenter à l'infini, et "bonjour" sera affiché en boucle sur la console. La variable i ne sera jamais affichée.

Ce genre de comportement n'est en général pas voulu. C'est l'équivalent d'une récursion infinie. Or, quand une fonction récursive part en récursion infinie, c'est souvent le signe que l'on a oublié d'implémenter un cas d'arrêt.

Il nous faut donc une boucle disposant d'un cas d'arrêt.

```
i = 0
begin while i < 10
        i = i + 1
        print "hello"
end while
print i
```

Dans l'exemple ci-dessus, nous utilisons une boucle "while" (ou boucle "tant que" en français). Quand on arrive pour la première fois sur la ligne "begin while", la variable i est égale à 0, ce qui est plus petit que 10. Le programme rentre donc dans la boucle. On ajoute 1 à la variable i et on affiche "hello". En arrivant sur la ligne "end while", on revient sur la ligne "begin while", et on vérifie la condition : i est égal à 1, la condition est respectée, le programme rentre dans la boucle une deuxième fois.

Accélérons un peu les choses et reprenons quand la variable i est égale à 10. On arrive sur la ligne "end while", on revient sur la ligne "begin while" et on teste la condition. La variable i est égale à 10. Or $10 < 10$ est faux. La condition n'est pas respectée. Dans ce cas là, le programme ne rentre pas dans la boucle et reprend à la ligne suivant "end while".

Au final, la variable i est affichée, on peut voir "10" dans la sortie standard. Nous venons d'écrire un programme affichant 10 fois "hello" et affichant "10".

4.3 Le While en C#

En C#, la boucle "tant que" est écrite de la manière suivante.

```
1  while (condition)
2  {
3      //code à exécuter;
4  }
```

Ici, la condition est une expression booléenne, comme celles que l'on utilise dans les "if". L'accolade ouvrante ('{') indique le début du bloc de code contenu dans la boucle. L'accolade fermante ('}') indique la fin de la boucle.

Reprenons l'exemple précédent :

```
1  int i = 0;
2  while (i < 10)
3  {
4      i = i + 1;
5      Console.WriteLine("bonjour");
6  }
7  Console.WriteLine(i);
```

5 Exercices

5.1 Exercice 0 : Hello (West)World !

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

Écrivez une fonction qui quand elle est appelée affiche n fois "Hello (West)World !" dans la sortie standard.

Attention : la chaîne de caractères imprimée doit être EXACTEMENT "Hello (West)World !". Cette rigueur sur les consignes s'applique à tout le TP.

Exemple avec $n = 10$:

```
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
Hello (West)World!
```

5.2 Exercice 1 : Echo

```
1  static void Echo();
```

Écrivez une fonction qui quand elle est appelée attend une entrée de l'utilisateur, puis écrit cette entrée dans la sortie standard.

Exemple :

```
Doesn't look like anything to me.                // User input

Doesn't look like anything to me.                // Program output
```

5.3 Exercice 2 : Print Reverse

```
1  static void Reverse();
```

Écrivez une fonction qui quand elle est appelée attend une entrée de l'utilisateur, puis écrit cette entrée dans la sortie standard à l'envers. Vous devez utiliser la récursivité pour cette fonction.

Exemple :

```
what door?                  // User input

?rood tahw                  // Program output
```

Rappel : on accède au caractère d'index i d'une string s ainsi :

```
1  s[i]
```

Rappel : on accède à la longueur d'une string s ainsi :

```
1  s.Length
```

5.4 Exercice 3 : Love ACDC

```
1  static void LoveAcdc();
```

Écrivez une fonction qui quand elle est appelée affiche dans l'entrée standard la chaîne de caractère suivante :

```
  * * *   * * *
* *   * * *   * *
*       *       *
* *  <3 ACDC  * *
  * *       * *
    * *   * *
      * * *
        *
```

Cette chaîne de caractères doit être affichée en blanc sur fond rose.

Attention : n'oubliez pas de réinitialiser les couleurs du terminal à la fin de votre fonction !

5.5 Exercice 4 : Encore des QCMs ?

```
1 static void MCQ(string question, string aws1, string aws2,
2                 string aws3, string aws4, int aws);
```

Écrivez une fonction qui, quand elle est appelée, génère un QCM.

La fonction doit d'abord afficher dans l'ordre :
— La chaîne de caractère `question`
— La chaîne de caractère `aws1` précédée de "1) "
— La chaîne de caractère `aws2` précédée de "2) "
— La chaîne de caractère `aws3` précédée de "3) "
— La chaîne de caractère `aws4` précédée de "4) "

La fonction lit ensuite une entrée utilisateur.

Si cette entrée est égale à `aws`, afficher la chaîne de caractère suivante :

```
Good job bro!
```

`[aws]` doit être remplacé par la valeur de `aws`.

Si cette entrée n'est pas égale à `aws` mais que $0 < entrée < 5$, afficher la chaîne de caractère suivante :

```
You lose... The answer was [aws].
```

Si cette entrée ne respecte pas les conditions précédentes, ou que l'entrée est invalide, afficher la chaîne de caractère suivante :

```
So counting is too hard, n00b...
```

5.6 Exercice 5 : Best Years

```
1 static void BestYears();
```

Écrivez une fonction qui doit quand on l'appelle afficher si les promos sont bonnes ou mauvaises.

Voici les règles à respecter : les promos **paires** sont de **bonnes** années, les promos **impaires** sont de **mauvaises** années.

Exceptions : la promo 2020 est la **meilleure** promo. La promo 2022 est une **mauvaise** promo.

Votre fonction doit afficher l'état des promos en commençant par 1989 jusqu'à 2022 inclus. Pour chaque promo, une chaîne de caractères devra être affichée.

Si la promo est la **meilleure** promo, afficher :

```
Best Year
```

Si la promo est une **bonne** promo, afficher :

```
Good Year
```

Si la promo est une **mauvaise** promo, afficher :

```
Bad Year
```

Voilà un exemple d'affichage. Attention, l'exemple est faux :

```
Good Year
Bad Year
Bad Year
Best Year
Good Year
Good Year
```

5.7 Exercice 6 : Morse

```
1  static void Morse(string s, int i = 0);
```

Écrivez une fonction qui quand on l'appelle est capable de "lire" du morse, c'est-à-dire d'émettre du son selon une chaîne de caractères donnée en paramètre.

Dans la chaîne de caractères, trois caractères pourront être présents et devront être interprétés ainsi :

— '.' (point) :
Quand un point est rencontré, un son d'une fréquence de 900 Hz et d'une durée de 0.15 secondes devra être généré.

— '_' (underscore) :
Quand un underscore est rencontré, un son d'une fréquence de 900 Hz et d'une durée de 0.45 secondes devra être généré.

— ' ' (espace) :
Quand un espace est rencontré, votre programme doit se mettre en pause durant 0.45 secondes.

Astuce : Vous pouvez aller voir la page MSDN de la méthode Thread.Sleep().

Exemple de chaîne de caractère à lire :

```
_ ·· __ · _ ___ ·__· ·_·· ·_ _·__ _ ···· · __· ·_ __ · ·_·_·_
```

5.8 Exercice 7 : LightHouse

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

Écrivez une fonction qui, quand on l'appelle, affiche un phare en ASCII-art de hauteur variable.

Le phare est composé au minimum d'une base et d'un toit.

```
// base :
 =====
_||_||_
-------

// toit :
  /^\
  |#|
```

Entre cette base et ce toit, on peut trouver n étages.

```
// 1 étage :
 |===|
  |0|
  | |
```

Voici des exemples d'affichage :

```
// n = 0

  /^\
  |#|
 =====
_||_||_
-------
```

```
// n = 1

  /^\
  |#|
 |===|
  |0|
  | |
 =====
_||_||_
-------
```

```
// n = 3

  /^\
  |#|
 |===|
  |0|
  | |
 |===|
  |0|
  | |
 |===|
  |0|
  | |
 =====
_||_||_
-------
```

These violent deadlines have violent ends.