TP C#1 : I Can See Sharp.

1 Consignes de rendu

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

```
– rendu-tpcs1-login_x.zip
    |– rendu-tpcs1-login_x/
        |– AUTHORS
        |– README
        |– Calc/
            |– Calc.sln
            |– Calc/
                |– Tout sauf bin/ et obj/
        |– Classes/
            |– Classes.sln
            |– Classes/
                |– Tout sauf bin/ et obj/
```

Bien entendu, vous devez remplacer *login_x* par votre propre login. N'oubliez pas de vérifier les points suivants avant de rendre :

- Le fichier AUTHORS doit être au format habituel : * *login_x$* où le caractère '$' représente un retour à la ligne.
- Le fichier README doit contenir les difficultés que vous avez rencontrées sur ce TP, et indiquera les bonus que vous avez réalisés.
- Pas de dossiers bin ou obj dans le projet.
- **Le code doit compiler !**

2 Introduction

Le Caml c'est fini [1]. À partir de maintenant, et ce jusqu'à la fin de l'année, le C# sera officiellement votre langage de prédilection (de gré ou de force). À l'inverse du Caml, qui est un langage fonctionnel, le C# est quant à lui un langage impératif [2]. Cela sous

1. Cachez votre joie.
2. Ainsi qu'orienté objet, mais ça, c'est pour plus tard...

entend donc une nouvelle façon d'appréhender la programmation où, en l'occurrence, les instructions sont exécutées de façon séquentielle par l'ordinateur.

L'objectif de ce TP sera donc de vous inculquer les bases du langage, ainsi que les notions élémentaires de la programmation impérative. Ainsi, bien que relativement accessibles, il est impératif [3] de bien assimiler les notions abordées ici. Cela comprend donc les variables et les types de base, les structures conditionnelles, ainsi qu'une introduction aux fonctions.

3 Cours

3.1 Variables

Dans les langages impératifs, les notions de variable et de type sont primordiales. Les variables associent un nom à une valeur ou un objet. Comme leur nom l'indique, elles peuvent être amenées à changer de valeur au cours de la vie d'un programme.

En C#, une variable est donc définie en priorité par son nom, son type, et sa valeur. D'autres caractéristiques permettent de la décrire plus précisément, mais pour l'heure, seuls les éléments susnommés nous intéressent.

La déclaration d'une variable s'effectue de la façon suivante :

```
1 // Initialized variable
2 TYPE <name> = <value>;
3
4 // Variable not initialized yet
5 TYPE <name>;
```

Cela peut pour l'instant sembler assez abstrait, mais tout devrait s'éclaircir avec ce qui va suivre. Penchons nous donc en détail sur les types de variable les plus courants, et par extension aux valeurs leurs étant associées.

```
1 int i    = -24; // Integer ( -2 147 483 648 <= i <= 2 147 483 647)
2 uint ui  = 24; // Positive integer ( 0 <= ui <= 4 294 967 295)
```

3. ACDC en représentation chaque semaine au Bataclan, humour et boissons fraîches garanties.

```
3 float f  = 666.42; // Floating point number
4 char c   = 'A'; // Character
5 string s = "myString"; // String
6 bool b   = true; // Boolean (true or false)
```

Il s'agit là d'une liste non exhaustive. Vous êtes, bien évidemment, fortement encouragés à vous renseigner sur les autres types existants en C#, étant donné qu'ils pourront s'avérer utiles un jour ou l'autre.

Vous aurez par ailleurs parfois besoin de convertir une variable d'un type vers l'autre, lorsque cela s'avérera possible : on parle alors de transtypage (ou cast). La conversion peut être implicite (à savoir qu'il suffit d'une simple assignation vers une variable du type souhaité), ou explicite. Dans ce dernier cas, il faut alors spécifier explicitement le type voulu.

```
1 char c   = '*';
2 int  i   = (int)c; // i has the ascii value of '*', which is 42
3 float f  = i; // implicit conversion. f has value 42.0
4 int j    = c; // implicit conversion. j has the ascii value of '*', 42
5 string s = i.ToString(); // value of s : "42"
```

Encore une fois, il ne s'agit là que d'exemples, vous renseigner sur les différentes possibilités ne saurait que vous être bénéfique, étant donné que vous tomberez un jour ou l'autre sur une problématique similaire.

Enfin, vous noterez dans l'exemple ci-dessus la conversion du type *char* vers le type *int*. Cela peut sembler surprenant au premier abord, mais cette conversion a pour origine la norme ASCII [4], qui associe chaque caractère à une valeur numérique différente.

Les opérateurs de base pouvant être utilisés en C# sont les suivants :

```
1     + // Addition and string concatenator
2     - // Substraction
3     * // Multiplication
```

4. cf section 3.4.

```
4       / // Division
5       % // Modulo
```

3.2 Structures conditionnelles

Les informations que retourne un même programme peuvent différer d'une exécution à l'autre. Cela est notamment dû à l'utilisation de branchements, et plus précisément, de branchements conditionnels. Ainsi, en fonction de l'état de différentes variables à un moment donné, le programme se comportera de façon différente. La forme la plus connue de structure conditionnelle (ainsi que celle que vous aurez à utiliser le plus souvent) est celle que vous avez déjà rencontrée jusqu'à présent, à savoir la structure *if... else...*. Elle se présente de la façon suivante :

```
1   if (<condition>)
2   {
3       // Instruction 1;
4       // Instruction 2;
5       // Instruction 3;
6       //  etc...
7   }
8   else if (<altCondition>)
9   {
10      // Alternative ;
11      /* Do something;
12          Do something else; */
13  }
14  else
15  {
16      /* Other instruction;
17          Another one;
18          Again and again; */
19  }
```

Ainsi, selon le cas où la condition est remplie ou non, les différentes instructions, situées dans le bloc délimité par le couple d'accolades correspondant seront exécutées. La condition consiste donc en n'importe quelle expression booléenne, pouvant avoir une

valeur vraie ou fausse. Les expressions booléennes se construisent à l'aide d'opérateurs logiques et d'opérateur de comparaison.

Les opérateurs de comparaison sont les suivants :

```
==  // equal
!=  // different
<   // less
>   // greater
<=  // less or equal
>=  // greater or equal
```

Quant aux opérateurs logiques, on en compte notamment trois :

```
|| // logic OR
&& // logic AND
!  // logic NOT
```

Notez que les opérateurs *OU* et *ET* présentés sont dits conditionnels ou de "court-circuit" : si la première opérande est vraie, le *OU* n'évaluera pas la deuxième opérande ; si la première opérande est fausse, le *ET* n'évaluera pas la deuxième opérande.

Au dessus, vous noterez la présence de l'instruction *else if*, qui permet de rajouter autant d'alternatives que nécessaire. Son utilisation est bien entendu facultative. Il en va par ailleurs de même pour l'instruction *else*, qui peut être omise en fonction des cas, cela sera toujours laissé à votre jugement.

Le deuxième type de branchement conditionnel en C# correspond à l'instruction *switch*. À l'inverse du *if... else...* qui teste le résultat d'une expression booléenne, le *switch* teste la valeur d'une variable, et différentes instructions seront exécutées en fonction de la valeur de la variable.

```
switch (<variable>)
{
    case 1:
        instructions;
```

```
5           break;
6       case 2:
7           instructions;
8           break;
9       case 3:
10          instructions;
11          break;
12      default:
13          instructions;
14          break;
15  }
```

Ici, l'élément placé à la suite du mot-clé *case* doit être du même type que la variable testée. Si, à l'issue des tests, aucun des cas ne correspond à la variable testée, ce sont les instructions placées à la suite de *default* qui seront exécutées. Il est donc nécessaire de toujours prévoir un cas par défaut. Veuillez, enfin, ne pas oublier l'instruction *break* qui est quasiment obligatoire à la fin de chaque cas, de façon à éviter que les instructions correspondant aux cas suivants ne soient exécutées.

3.3 Fonctions

Le C# nous offre la possibilité de découper un programme en différentes petite parties, plus connues sous le nom de fonctions. Nous allons ici aborder les bases en ce qui concerne les fonctions, à savoir les créer, pour ensuite pouvoir y faire appel.

La déclaration d'une fonction s'effectue de la façon suivante :

```
1  public static RETURNTYPE <function>(TYPE1 <param1>, TYPE2 <param2>)
2  {
3       //instructions;
4       int a = 5;
5       int b = a + 7;
6       /* ...
7           ....
8           ...*/
9       string str = "blaba";
```

```
10      <param2name> = <value>;
11      // other instructions;
12
13      return <result>;
14  }
```

Nous allons pour l'instant passer sur les mots *public* et *static*. Vous pouvez vous contenter de les inclure à chaque déclaration de fonction ; à moins bien évidemment de savoir parfaitement ce que vous faites.

```
1  public static RETURNTYPE <functionName>(TYPE1 <param1>, TYPE2 <param2>)
```

La première ligne correspond à ce qu'on appelle le prototype d'une fonction. Elle est constituée du type de retour de la fonction, de son nom, des paramètres si elle en possède, ainsi que du type des dits paramètres. Généralement, on dit qu'une fonction renvoie ou retourne une valeur, dont le type fait partie de ceux évoqués précédemment, à savoir les types de base du C#. On note aussi la présence de paramètres, représentés chacun par leur type et leur nom, qui peuvent ainsi être utilisés dans le corps de la fonction. Cependant, toutes les fonctions ne renvoient pas forcément une valeur. Certaines se contentent en effet d'exécuter une suite d'instructions (par exemple afficher différents éléments à l'écran), sans retourner quoi que ce soit. Dans ce cas là, le type de retour est remplacé par le mot-clé *void*. Cela sert concrètement à signaler qu'aucune valeur de retour n'est attendue de la part de la fonction.

Le corps de la fonction, quant à lui, consiste en une suite d'instructions qui varie bien évidemment selon la tâche que doit remplir la fonction. Ce corps est délimité par le couple d'accolades placées avant et après les instructions.

Concentrons-nous enfin sur la dernière ligne :

```
1  return <result>;
```

Cette instruction est présente dans le cas où la fonction renvoie une valeur (une fonction de type *void* n'en aura donc pas l'utilité). Le cas échéant, le résultat renvoyé sera du même type que le type de retour de la fonction, d'où l'intérêt de le préciser dans son prototype.

Nous avons par exemple, ici, une fonction parfaitement inutile qui prend en paramètres une chaîne de caractères et un entier, convertit ce dernier en chaîne de caractères, la concatène au premier paramètre, et affiche une chaîne de caractères complètement différente. Cette fonction ne renvoie rien.

```
1 public static void GreatFunction (string str, int num)
2 {
3     string sNumber = num.ToString();
4     string newString = str + sNumber;
5     // displays a string
6     Console.WriteLine("This is a useless function.");
7 }
```

De même, les fonctions récursives en C# se déclarent le plus naturellement du monde, nul besoin d'indiquer un mot clé spécifique. En voici un exemple :

```
1 public static int GiveTheAnswer(int n)
2 {
3     if (n < 42)
4         return GiveTheAnswer(n + 1);
5     else if (n > 42)
6         return GiveTheAnswer(n - 1);
7     else
8         return n;
9 }
```

Pour appeler une fonction, il suffit d'entrer son nom et de lui passer ce qu'on appelle des arguments. Les arguments doivent correspondre exactement aux paramètres de la fonction, ce qui implique donc être en même quantité et du même type, et ce dans le même ordre que celui où ont été définis les paramètres. Voici une illustration de ce principe :

```
1 static void Main ()
2 {
3     int answer = GiveTheAnswer(2020);
4     Console.WriteLine(answer); // display value of answer
```

```
5       GreatFunction("Deadline is coming.", 1337);
6   }
```

On remarque la présence de la fonction *Main()* qui correspond au point d'entrée de votre programme, et qui sera donc présente dans tous vos programmes en C#. C'est à l'intérieur de celle-ci que seront appelées toutes vos fonctions, et que seront exécutées les instructions du programme.

3.4 Divers

Commentaires

Comme vous avez pu le voir à plusieurs reprises au dessus, certaines portions de code sont précédées des caractères //, ou encore entourées des caractères /* et */. Il s'agit là tout simplement de la façon d'écrire des commentaires en C#. Les caractères // indiquent que tout ce qui suit sur cette même ligne sera considéré comme des commentaires. Les caractères /* et */ quant à eux servent à délimiter des commentaires répartis sur plusieurs lignes.

Les commentaires sont ignorés par le compilateur et sont là principalement pour documenter votre code, afin de le rendre compréhensible à n'importe quelle personne qui souhaiterait le lire, ou ne serait-ce qu'à vous, si d'aventure vous aviez à reprendre un de vos projets après plusieurs semaines ou mois. Il sera donc primordial de commenter votre code lorsque vous serez amenés à travailler sur des projets en groupe.

Indentation

Tout au long de ce cours, vous avez pu voir différentes portions de code respectant une certaine indentation, c'est-à-dire que chaque bloc de code était décalé en fonction de son niveau d'imbrication logique. Nous vous recommandons fortement d'effectuer des indentations dans votre code afin de le rendre plus lisible.

Généralement, Visual Studio le fera automatiquement pour vous, mais dans le cas où cela ne sera pas fait, il existe différents raccourcis vous permettant de le faire :

— Ctrl + K, Ctrl + F : Indentation de la zone sélectionnée.
— Ctrl + K, Ctrl + D : Indentation de tout le fichier.

ASCII

L'ASCII (American Standard Code for Information Interchange) est une norme de codage de caractères en informatique qui établit une correspondance entre les caractères et une valeur numérique particulière. Les caractères de votre ordinateur sont, en effet, représentés par des nombres, et c'est la façon dont ces nombres sont interprétés qui définit la façon selon laquelle ils seront affichés à l'écran. Ci-dessous la table ASCII des 128 premiers caractères, qui sont aussi les plus utilisés. Elle présente les caractères, associées à leurs valeurs décimales et hexadécimales :

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

4 Exercices

4.1 Calculatrice

Vous allez à présent travailler sur la solution *Calc.sln*. Complétez les fonctions qui se trouvent dans *Function.cs* afin de rendre cette calculatrice fonctionnelle.

```
|– Calc/
    |– Calc.sln
    |– Calc/
        |– Functions.cs
        |– Other files
```

La fonction BasicOp

Cette fonction prend en paramètre deux opérandes A et B, ainsi qu'un opérateur sous la forme d'un caractère, et renvoie le résultat de l'opération sur A et B. Ci-dessous la liste des opérandes à gérer :

- '+' représente l'addition,
- '-' représente la soustraction,
- '*' représente la multiplication,
- '/' représente la division,
- '%' représente le modulo.

```
1 public static double BasicOp(double A, double B, char op);
```

Attention

Toutes les fonctions qui vont suivre sont à faire de manière récursive [5]. Il est évidemment interdit d'appeler les fonctions préexistantes de la bibliothèque *Math* du framework *.NET*, notamment pour les fonctions *Pow* et *Sqrt*.

La fonction Fact

Cette fonction renvoie le résultat de $A!$.

```
1 public static double Fact(double A);
```

La fonction Fibo

Cette fonction renvoie le résultat de $Fibonacci(A)$. On rappelle que $Fibonacci(0) = 0$ et $Fibonacci(1) = 1$.

```
1 public static double Fibo(double A);
```

5. Certes, le Caml c'est fini, mais pas la récursivité. :)

La fonction Pow

Cette fonction renvoie le résultat de A^B.

```
1 public static double Pow(double A, double B);
```

La fonction Gcd

Cette fonction renvoie le plus grand diviseur commun de A et de B. Pensez à l'algorithme d'Euclide.

```
1 public static double Gcd(int A, int B);
```

La fonction RotN

Cette fonction prend en paramètre un caractère représentant un chiffre, et renvoie le chiffre obtenu après n rotations, c'est-à-dire situé n rangs après le chiffre donné. Il faudra bien faire attention à ce que le caractère retourné soit compris entre '0' et '9'.

Par exemple :

— Rotn('4', 3) = '7'
— Rotn('9', 1) = '0'
— Rotn('4', 18) = '2'

```
1 public static char RotN(char C, int N);
```

Bonus : La fonction Sqrt

Cette fonction renvoie une approximation au dix-millième de $\sqrt{A}$. Renseignez-vous sur la méthode de Héron [6]. Vous pouvez par exemple écrire une fonction récursive calculant la valeur de la racine carrée, qui sera appelée dans votre fonction Sqrt.

```
1 public static double Sqrt(double A);
```

4.2 Un petit dicton

La rumeur dit qu'à l'EPITA, les promotions paires (ex : 2000) sont de qualité supérieure aux promotions impaires. Cette information peut éventuellement être soumise à vérification. Néanmoins, nous allons créer un *Estimateur de Qualité des Promotions*.

6. https://fr.wikipedia.org/wiki/Méthode_de_Héron

Vous allez à présent travailler sur la solution *Classes.sln*. Complétez les fonctions qui se trouvent dans *Function.cs* afin de pouvoir estimer la qualité de votre propre promotion !

```
|– Classes/
    |– Classes.sln
    |– Classes/
        |– Functions.cs
        |– Other files
```

La fonction Swap

Cette fonction échange le contenu de deux TextBox A et B.

```
1 public static void Swap(TextBox A, TextBox B);
```

La fonction IsValid

Tout d'abord, nous allons vérifier qu'une année donnée correspond bel et bien à une promotion de l'EPITA. La première promotion de l'école date de 1989. On considérera qu'une promotion après 2020 n'est pas valide. Cette fonction renvoie un booléen indiquant si *year* est effectivement l'année d'une promotion.

```
1 public static bool IsValid(int year);
```

La fonction IsEven

Cette fonction renvoie un booléen indiquant si l'entier n est pair ou non.

```
1 public static bool IsEven(int n);
```

La fonction CoolOrNot

Cette fonction vérifie en premier lieu la validité de l'année passée en paramètre pour ensuite vérifier sa parité. Elle renvoie une chaîne de caractères selon les différents cas :

— "They should be cool !" en cas d'année paire,

— "Well, they are less likely to be cool." en cas d'année impaire,

— "This class doesn't exist." si ce n'est pas une promotion de l'école.

```
1 public static string CoolOrNot(int year);
```

La fonction Older

Cette fonction compare l'ancienneté de deux années, en vérifiant les cas spéciaux des années identiques ou non existantes, et renvoie les chaînes de caractères suivantes selon ces cas :

— "Class of {A or B} is older than class of {B or A}."
— "At least one class does not exist."
— "Those classes are the same."

```
1 public static string Older(int yearA, int yearB);
```

La fonction Better

Cette fonction compare deux années selon les règles suivantes :

— Une promotion paire sera considérée meilleure qu'une promotion impaire.
— Dans le cas où les deux promotion sont de même parité, la plus ancienne sera considérée comme légèrement meilleure.
— N'oubliez pas de vérifier les cas spéciaux (non existence, deux années identiques).

Vous renverrez les chaînes de caractères suivantes selon les cas :

— "Class of {A or B} seems to be better than class of {B or A}."
— "Both are great, but class of {A or B} is better, thanks to the wisdom of the elders."
— "At least one class does not exist."
— "Those classes are the same."

```
1 public static string Better(int yearA, int yearB);
```

Bonus

— Lorsqu'une année supérieure à 2020 est donnée, affichez "This class does not exist... Yet.".
— Lorsqu'une année est divisible par 2018, les règles changent. La promotion est évidemment meilleure, et passe devant toutes les autres.

Brace yourselves, Deadline is coming.