SUPPORT DE COURS C#

SI4 - SLAM2

Date	Révision
07/07/2017	Publication initiale BTS SIO
19/08/2017	Correction erreurs et ajout encapsulation
05/11/2017	Correction diverses et ajout classes abstraites et interfaces et ajout des fichiers
14/11/2017	Ajout des chapitres généralités, opérations booléennes, langages, etc.
22/11/2017	Ajout "les fonctions" et "interfaces graphiques".

TABLE DES MATIÈRES

1 GÉNÉRALITÉS

1.1 INTRODUCTION

La programmation implique une idée simple : **faire** effectuer des tâches répétitives ou complexes.

Cela signifie décrire des actions, des étapes et des conditions à un système capable de les exécuter.

1.2 EXEMPLES

Il y a de nombreux exemples de programmes, dans la nature, dans les objets humains et dans l'univers.

Les exemples classiques :

- une chaîne ADN est un programme du corps humain
- une réaction chimique est un programme atomique
- une recette de cuisine est un programme culinaire
- un moulin à vent est un programme de conversion (blé en farine, vent en énergie)
- un plan est un programme géographique (suivre un chemin)

1.3 OBJECTIFS D'UN PROGRAMME

Un programme a donc un but. Les programmes d'origine humaine, ont généralement comme objectif :

- d'effectuer automatiquement les tâches répétitives
- sécuriser les actions (éviter les oublis au bout de 1000 fois par exemple)
- calculer des opérations complexes (puissance de calcul)

2 FONCTIONNEMENT DES SYSTÈMES PROGRAMMABLES

La plupart des systèmes sont facilement visibles : munis d'un écran et d'un moyen de saisie, ils sont rapidement identifiables et souvent indispensables.

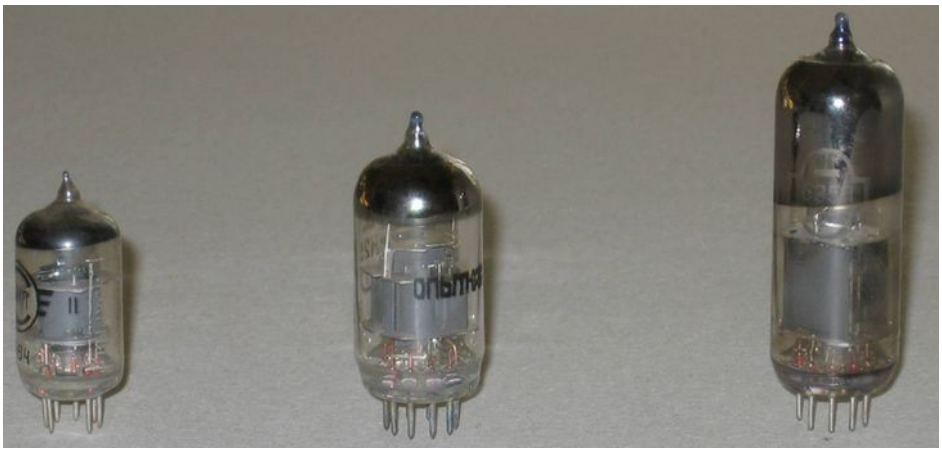
PC, smartphones mais aussi télévisions, autoradios… les systèmes programmés sont partout.

Pourtant, la structure est toujours la même que le premier ordinateur !

2.1 HISTORIQUE

On attribue la paternité de l'informatique à Alan Turing : en 1936, il propose un concept permettant à une machine d'interpréter du code.

En 1946, le premier ordinateur s'appelle ENIAC : c'est un acronyme qui signifie "Electronic Numerical Integrator And Calculator". Il utilise des lampes, car les transistors n'existent pas encore.



L'invention du transistor (1947) révolutionne l'électronique : l'UNIVAC (1951) qui occupe maintenant une surface de 25m² pour une mémoire de 1000 mots.

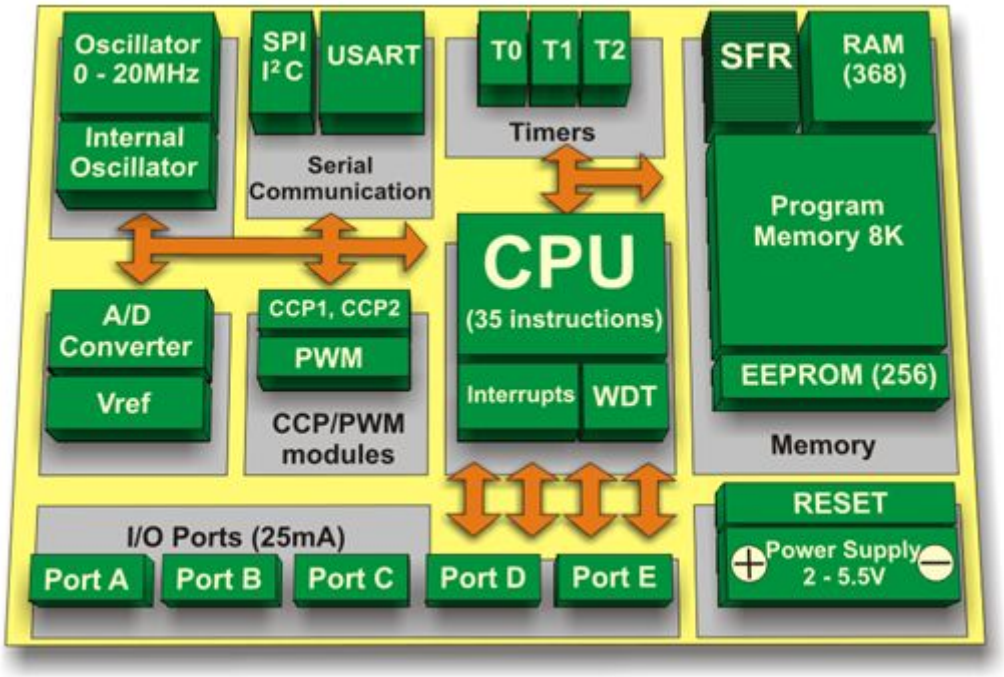
Finalement, Intel se lance dans l'informatique et vend le premier microprocesseur en 1971… mais il faut attendre 1981 pour voir arriver le premier PC (Personal Computer) d'IBM.

2.2 ARCHITECTURE MATÉRIELLE

Pour pouvoir fonctionner, un ordinateur nécessite au minimum 4 composants :

- Une **mémoire** pour contenir le programme, les données et d'autres informations
- Un **processeur** capable de lire les informations dans la mémoire et les traiter
- Une **horloge** pour faire avancer le programme : sans horloge, le processeur resterait sur la même instruction, indéfiniment
- Un **bus** (et même plusieurs bus) pour permettre à tous les éléments et périphériques de communiquer ensemble. Il s'agit d'un ensemble de pistes (de fils électriques) qui relient la mémoire et le processeur (bus principal) ou d'autres bus pour le clavier, les cartes vidéos, etc.

Il faut également ajouter des **ports** d'entrées et sorties (ils fonctionnent de manière similaire aux ports marins) pour relier l'écran, le clavier et les autres périphériques.



Architecture d'un contrôleur PIC

Pour résumer, l'horloge cadence les vitesses de la CPU, de la mémoire et des ports de communications. Elle permet d'incrémenter la lecture des instructions pour passer aux suivantes.

A noter : on différencie les architectures RISC (Reducted Instruction Set Computing) et CISC (Complex Instruction Set Computing).

Dans l'architecture RISC, le processeur exécute une instruction par cycle d'horloge. Dans l'architecture CISC, certaines instructions sont plus longues et le processeur bloque l'avancement du programme pour les terminer.

3 LES DIFFÉRENTS LANGAGES

Il existe plusieurs langages de programmation, certains sont propres à une technologie, d'autres n'existent plus tandis que de nouveaux langages naissent pour répondre à de nouveaux besoins.

3.1 LE LANGAGE MACHINE

Il s'agit du langage le plus proche du binaire. Le microprocesseur ne peut comprendre que des codes binaires mais les programmeurs préfèrent des mots.

Le langage machine utilise des codes binaires, est l'assembleur utilise donc un dictionnaire dans lequel chaque mot correspond à un code binaire compréhensible par le processeur.

`MOV EAX, 0x01020304` *(déplace EAX dans mémoire 01020304)*	A101020304

Si c'est le langage le plus rapide, il a comme inconvénient que chaque processeur utilise ses propres codes (EAX est un registre sur les processeurs Intel, qui n'existe pas sur ARM, Motorola, ...)

3.2 LES LANGAGES COMPILÉS

Pour éviter l'inconvénient de devoir taper plusieurs codes pour faire le même programme sur plusieurs processeurs, sont apparus les langages compilés.

Le langage de développement utilise des instructions proches des mots humains (en anglais) : for, if, else, print, etc.

Exemple de programme en Delphi (Pascal)

```
procedure TForm1.Draw();
var
  i, j, k: Integer;
begin
  for j:=0 to 4 do
    for i:=0 to 4 do
      begin
        k:=tab_int[i,j];
        img_screen.Canvas.Draw(i*100,j*100,tab_img[k].Picture.Graphic);
      end;
end;
```

Il faut ensuite utiliser un programme appelé compilateur, qui va traduire ces instructions en code machine. Il est donc évident qu'il faut créer autant de compilateur qu'il existe de processeur. Pour le développeur, c'est transparent (c'est la société qui vend les compilateurs qui fait le travail).

Il reste cependant un inconvénient : chaque machine utilise du matériel différent (résolution d'écran différente, emplacement mémoire différent, etc) et il reste difficile de créer un programme compatible avec plusieurs architectures.

3.3 LES LANGAGES INTERPRÉTÉS

Pour contourner l'inconvénient des différences entre les machines, il existe récemment les machines virtuelles : Java Runtime Environnement (Oracle) et framework.net (Microsoft) sont des exemples de machines virtuelles.

Le code écrit dans le langage correspondant (Java ou C#) est exécuté en temps réel sur la VM (Virtual Machine). Le code est donc interprété.

On parle parfois de compilation "bytecode" pour indiquer la réduction des instructions, mais ce n'est pas une compilation en langage machine.

3.4 LES EXEMPLES DE LANGAGES

Langage	Compilé Interprété	VM	Commentaire
Assembler	compilé	non	Langage très rapide (vitesse) mais compatible à un type de processeur.
Pascal / Delphi	Compilé	non	Langage avec une syntaxe simple, utilisant un compilateur.
C / C++	Compilé	non	Langage compilé, orienté objet, très rapide mais restant proche du processeur.
Java	Interprété	oui	Langage très sécurisé, orienté objet utilisant une machine virtuelle
Csharp (C#)	Interprété	oui	Langage très sécurisé, orienté objet utilisant une machine virtuelle
PHP	Interprété	Non (serveur)	Langage principalement utilisé sur Internet. Nécessite un serveur (Apache, IIS, NGINX, ...)
JavaScript	Interprété	Non (client)	Langage principalement utilisé dans les navigateurs web (Firefox, Chrome, IE, Safari).
Python, Ruby, Perl	Interprété	Non (script)	Langages de scripting dans certaines applications (ex : Blender)

A Noter : on estime que le fonctionnement d'une machine virtuelle fait perdre un peu de puissance mais que cela reste négligeable (de l'ordre de quelques pourcents)

4 LES SYSTÈMES NUMÉRIQUES

Le système de comptage décimal ne correspond pas au système des machines : ces dernières comptent en binaire : 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, etc.

4.1 COMPTAGE DÉCIMAL

Pour rappel, en décimal il y a dix symboles : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. Ce sont les **chiffres**.

Pour compter au-delà de ces dix symboles, on ajoute une colonne à gauche et les symboles contenus dans cette colonne sont multipliés à 10^1, 10^2, … comme dans le tableau suivant :

milliers	centaines	dizaines	unités
* 10^3	* 10^2	* 10^1	* 10^0
1	9	8	4

Ainsi le nombre dans le tableau est : $1*10^3 + 9*10^2 + 8*10^1 + 4 (*10^0)$

Pour rappel, un nombre à la puissance 0 est toujours égal à 1, dont un nombre multiplié par un est toujours égal à lui-même.

Il est plus facile pour les humains de pouvoir décrire le nombre plutôt que d'énumérer la suite de chiffres (dire "mille neuf cent quatre-vingt-quatre" est plus facile que "un – neuf – huit – quatre")

4.2 COMPTAGE BINAIRE

Les règles sont exactement les mêmes sauf qu'il n'y a que deux symboles : 0 et 1. La notion de **bit** remplace la notion de chiffre.

*** 2^3**	*** 2^2**	*** 2^1**	*** 2^0**
Décimal = 8	*Décimal = 4*	*Décimal = 2*	*Décimal = 1*
1	1	0	1

Ainsi le nombre dans le tableau est : $1*2^3 + 1*2^2 + 0*2^1 + 1 (*2^0) \Rightarrow 13_{(10)}$ soit treize en décimal.

4.2.1 ENTRAÎNEZ-VOUS

Retrouver la valeur en décimal des nombres binaires suivants :

00000000	11111111	10000000	00001111	0101

4.3 COMPTAGE HEXADÉCIMAL

Sur les systèmes informatiques, il est fréquent de travailler sur un groupe de bits : précisément, on utilise une unité appelée octet (en anglais, byte) qui regroupe 8 bits.

Un octet peut contenir 256 valeurs différentes : de 0 à 255 (mais parfois on parle d'octet signé pour compter de -127 à +128).

Cependant, l'écriture en binaire est longue, sujette à erreur et peu intuitive. Les informaticiens ont pris l'habitude d'utiliser l'hexadécimal pour représenter les octets, mais aussi des valeurs plus grandes, comme les couleurs dans une page web. En effet, n'avez-vous jamais rencontré une valeur de la forme #FFC7BA dans les logiciels de retouche d'images ?

L'hexadécimal comporte 16 symboles : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, **A, B, C, D, E et F.**

Hormis cela, le fonctionnement reste le même, d'un point de vue comptage :

*** 16^3**	*** 16^2**	*** 16^1**	*** 16^0**
Décimal = 4096	*Décimal = 256*	*Décimal = 16*	*Décimal = 1*
1	0	A	7

Ainsi le nombre dans le tableau est : $1*16^3 + 0*16^2 + A*16^1 + 7\ (*16^0) \Rightarrow 4263_{(10)}$ soit quatre mille deux cent soixante-trois en décimal.

Si ce système paraît complexe pour le calcul, il simplifie l'écriture et la conversion :

$10A7_{(16)}$ utilise 2 octets : $10_{(16)}$ et $A7_{(16)}$. L'écriture en binaire est

0001	0000	1010	0111
00010000		10100111	

4.3.1 ENTRAINEZ-VOUS

Retrouver la valeur en binaire des nombres hexadécimaux suivants :

F0	**0F**	**FF**	**77**	**10**

4.4 Notations

Les systèmes d'écriture pouvant être équivoques (10 signifie-t-il $10_{(2)}$ ou $10_{(10)}$ ou $10_{(16)}$?) vous rencontrerez les notations suivantes dans les langages de programmation :

décimal	binaire	hexadécimal
10 ou 2.5, etc.	**0b**0110010	**0x**FE1A
	%0110010	**$**FE1A
		#FE1A

5 Opérations booléennes

Les opérations booléennes sont des opérations logiques qui ne s'appliquent que sur du binaire. Cependant, un nombre décimal peut toujours être convertit en binaire, ce qui signifie que ces opérations peuvent être écrites avec une notation décimale : l'ordinateur convertira les valeurs en binaire puis effectuera l'opération demandée et renvoyer un résultat décimal.

La plupart du temps (ET, OU, OU exclusif) il s'agit de superposer deux nombres binaires et d'appliquer le filtre. Voici les opérations :

5.1 ET logique (AND ou &)

Le ET logique ne place un 1 que lorsque il y a un 1 dans chaque valeur sur le même emplacement.

Exemple : 111001 AND 110011 ⇒ 110001 (avec ET, les zéros gagnent toujours)

	1	**1**	**1**	**0**	**0**	**1**
ET	**1**	**1**	**0**	**0**	**1**	**1**
	1	**1**	**0**	**0**	**0**	**1**

5.2 OU logique (OR ou |)

Le OU logique place un 1 chaque fois qu'il y a un 1 dans une des valeurs sur le même emplacement.

Exemple : 111001 OR 110011 ⇒ 111011 (avec OU, les uns gagnent toujours)

	1	**1**	**1**	**0**	**0**	**1**
OU	**1**	**1**	**0**	**0**	**1**	**1**
	1	**1**	**1**	**0**	**1**	**1**

5.3 OU EXCLUSIF (XOR)

Le OU logique exclusif ne place un 1 que lorsque il n'y a qu'un seul 1 sur le même emplacement.

Exemple : 111001 XOR 110011 ⇒ 001010

	1	**1**	**1**	**0**	**0**	**1**
XOU	**1**	**1**	**0**	**0**	**1**	**1**
	0	**0**	**1**	**0**	**1**	**0**

5.4 NON LOGIQUE (NOT OU !)

Le NON logique correspond à l'inversion de tous les bits : 1 devient 0 et 0 devient 1.

Exemple : NOT 111001 ⇒ 000110

NON	**1**	**1**	**1**	**0**	**0**	**1**
	0	**0**	**0**	**1**	**1**	**0**

5.5 ENTRAÎNEZ-VOUS AUX OPÉRATIONS BOOLÉENNES

A) $FE AND $EF

B) $FE OR $EF

C) 00111100 XOR 11001100

D) NOT $F0

E) $A AND $5

F) 173 AND 255

Réponses :

A) $EE *B) $FF* *C) 11110000* *D) $0F* *E) $00* *F)173*

6 LES ÉLÉMENTS D'UN PROGRAMME

La programmation reprend des éléments de fonctionnement de la pensée. La difficulté est moins d'apprendre un langage que de pouvoir exprimer ses réflexions.

Nous verrons que rendre la monnaie avec des pièces et des billets est un programme simple mais qui nécessite de décortiquer le fonctionnement de notre pensée.

Il y a cependant quelques principes que nous pouvons expliquer : les conditions et les répétitions.

6.1 CONDITIONS

Les conditions permettent à un programme de faire un choix, de la même manière que notre esprit juge et décide à chaque instant de prendre des décisions pour notre vie.

6.1.1 CONDITIONS SIMPLES

Par exemple, s'il fait beau, j'irai à la piscine :

- Cas 1 : il fait beau ⇒ je vais à la piscine
- Cas 2 : il ne fait pas beau ⇒ *rien n'est précisé*

En programmation l'écriture sera similaire :

```
SI (beau == Vrai) alors
    Piscine = Vrai
SINON
FINSI
```

Comme vous pouvez le voir, l'ordinateur ne peut pas déduire seul que piscine est Faux, il faut donc l'écrire explicitement. C'est là où votre esprit a déduit seul que s'il ne fait pas beau, alors vous n'irez pas à la piscine.

Il faudra alors écrire à l'ordinateur, tous les comportements à avoir :

```
SI (beau == Vrai) alors
    Piscine = Vrai
SINON
    Piscine = Faux
FINSI
```

6.1.2 CONDITIONS COMPLEXES

Dans la réalité, nous sommes capables de gérer plusieurs cas de figures, notamment, s'il y a plusieurs conditions.

Prenons l'exemple suivant :

S'il fait beau et que j'ai assez d'argent, j'irai à la piscine.

Pour l'ordinateur, notre programme indique que Piscine sera toujours Faux, sauf lorsque Beau sera

Vrai ET argent > prixTicket

```
SI (beau == Vrai ET argent > prixTicket) alors
    Piscine = Vrai
SINON
    Piscine = Faux
FINSI
```

Mais nous sommes habitués à des sélections plus complexes :

S'il fait beau et que j'ai assez d'argent, j'irai à la piscine, sinon, si j'ai assez d'argent j'irai au cinéma, sinon je reste à la maison.

Pour notre ordinateur, cela donne :

```
Piscine = Faux
Cinema = Faux
Maison = Faux

SI (beau == Vrai ET argent > prixTicket) alors
    Piscine = Vrai
SINON SI (argent > prixTicket)
    Cinema = Vrai
SINON
    Maison = Vrai
FINSI
```

Les conditions mesurent des égalités :

- nombreA > nombre B, nombreA >= nombreB (signifie plus grand ou égal)
- nombreA == nombreB (on n'écrit pas = tout seul car cela signifie une affectation)

Lorsque l'égalité est vraie, on exécute la première partie du code.

Lorsque l'égalité est fausse, on exécute la partie de code après le premier SINON.

6.1.3 CONDITIONS IMBRIQUÉES

Le programme ci-dessus peut se faire en deux temps : on vérifie la première condition et si elle est vraie, on vérifie la seconde :

```
SI (argent > prixTicket) alors
    SI ( beau == Vrai )
       Piscine = Vrai
   SINON
       Cinema = Vrai
   FINSI
SINON
    Maison = Vrai
FINSI
```

6.2 RÉPÉTITIONS

6.2.1 RÉPÉTITIONS SIMPLES

Les répétitions consistent à faire plusieurs fois les mêmes actions. A chaque boucle, nous parlerons d'itération : une boucle qui compte de 1 à 10 aura donc 10 itérations.

Comme il est très rare de faire des boucles à l'infinie, il y a généralement une condition de sortie.

Exemple : *je dois écrire dix fois "je ne bavarde pas en classe"*

Cette répétition se traduit en langage informatique :

```
COMPTER à partir de 1 et jusqu'à 10
    écrire "je ne bavarde pas en classe"
FIN COMPTER
```

Dans ce cas, la condition de sortie est l'usage du compteur "jusqu'à 10" (la valeur du compteur doit être inférieure à 11).

Nous verrons qu'il y a 3 sortes de boucles :

- FOR (*condition*) FAIRE { actions }
- DO { actions } TANTQUE (*condition*)
- TANTQUE (*condition*) FAIRE { actions }

Seule la dernière boucle permet de ne pas faire d'action si la condition est fausse. Les deux autres exécuteront au moins une fois les actions avant de tester la condition ;

6.2.2 RÉPÉTITIONS IMBRIQUÉES

C'est un cas particulier, car il est possible dans une première boucle, d'exécuter une autre boucle (et encore une autre, etc.). En revanche, les boucles ne doivent pas se croiser.

Bon	Mauvais

`COMPTER à partir de 1 et jusqu'à 10`
`  TANTQUE FinPage = 0`
`  écrire "je ne bavarde ..."`
`  FIN TANTQUE`
`FIN COMPTER`
	`COMPTER à partir de 1 et jusqu'à 10`
	`  TANTQUE FinPage = 0`
	`  écrire "je ne bavarde ..."`
	`  FIN COMPTER`
	`FIN TANTQUE`

7 PROGRAMMATION C#

7.1 VUE GÉNÉRALE "HELLO WORLD"

Le premier code de base en C# se présente comme suit :

```
/* Voici le code d'exemple de base : Hello World */
using System;
using System.Collections.Generic;      // voici un commentaires de fin de ligne
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args) {
            Console.WriteLine("Hello World");
            Console.ReadKey();
         }
    }
}
```

Ce code est constitué de plusieurs parties, dont certaines resteront faiblement décrites pour le moment :

7.1.1 COMMENTAIRES /* ET */

Pour C#, les lignes comprises entre /* et */ ne sont pas lues : cela permet au programmeur de placer des commentaires avant un code complexe, ou bien décrire rapidement ce que fait l'application.

Il est important de comprendre qu'il faut une convention d'écriture commune à tous les programmeurs. On utilise donc des balises ouvrantes et fermantes, un peu comme en français, les symboles " et " indique que le texte contenu entre les deux symboles et dit à haute voix par exemple.

7.1.2 BLOCS { ET }

L'utilisation de blocs, permet à l'interpréteur C# de comprendre que l'ensemble des instructions comprises entre { et } vont ensemble et constituent un bloc non-sécable.

Ici, **namespace** contient tout le programme qui n'est constitué que d'une seule **classe** 'Program' qui elle-même ne contient qu'une **fonction** 'Main'.

Cette structure permet de découper un programme en section logique (on regroupe ensemble, les classes et fonctions qui agissent de manière similaire ou qui ont un lien commun sur les données ou les interfaces). Si vous ne comprenez pas cette phrase, c'est tout à fait normal lorsque l'on commence à programmer avec un langage comme celui-ci…

7.1.3 Bibliothèques (using)

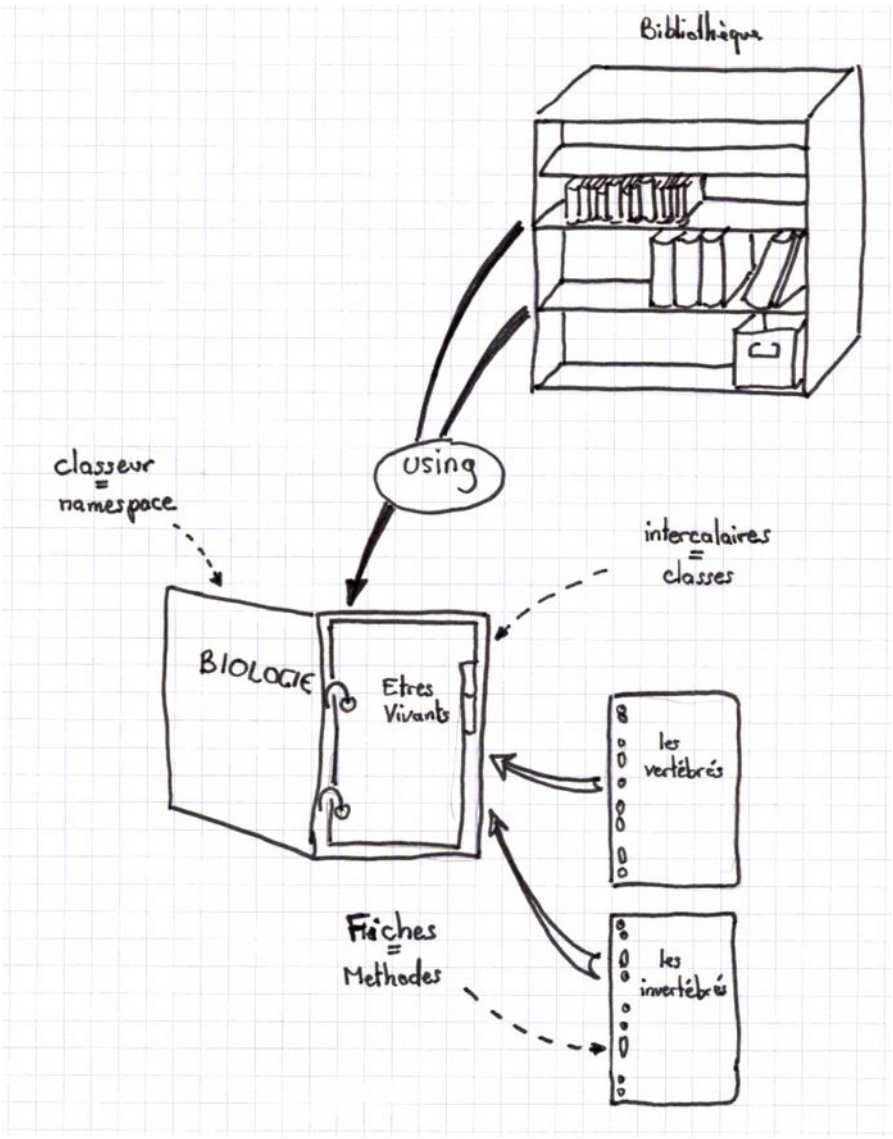
En programmation, il existe de nombreuses instructions et fonctions qui ne nécessitent pas d'être écrites, car les programmeurs ont déjà fait ce travail. En bricolage, on utilise des outils existants et un bricoleur n'a pas à ré-inventer le tournevis ou la pince.

En programmation, ces outils sont des bibliothèques de fonctions (ou méthodes). Pour éviter d'avoir une boite à outils trop lourde, C# demande de préciser en début de programme les bibliothèques qui seront utilisées.

Par défaut Visual Studio propose d'utiliser 5 bibliothèques, mais ce n'est pas nécessaire pour un programme aussi simple.

7.2 En résumé

Le schéma ci-après donne un exemple d'analogie possibles



7.3 Syntaxe d'écriture

C# utilise une syntaxe simple mais très codifié :

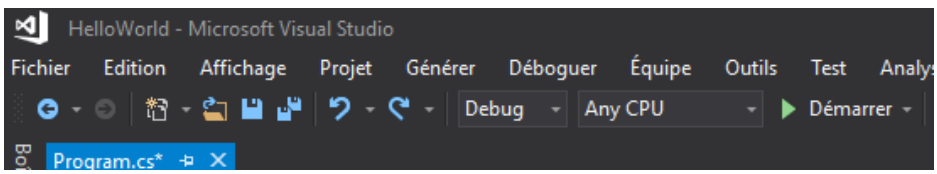
- Les instructions se terminent par un point-virgule (;) comme une phrase en français se termine par un point.
- Les noms de classes commencent par une majuscule. Pour le moment, une classe est un regroupement de commandes, un peu comme une pochette transparente dans un classeur, qui contient plusieurs feuilles
- Les fonctions (appelées aussi méthodes) et variables ne commencent pas par une majuscule. Considérez qu'une fonction est un regroupement d'instructions qui seront exécutées lignes après lignes, jusqu'à la fin du bloc fonction.
- C# utilise la notation "camel" à l'exception des règles sur la première lettre. Ainsi, une méthode (fonction) pour colorier un rectangle s'écrira "colorierRectangle". *Seules les classes s'écrivent avec la première lettre en majuscule*
- Les variables peuvent utiliser tous les caractères alphabétiques (accentués compris) mais ne peuvent commencer par un chiffre, ou porter le nom d'un mot-clé de C#
- Les conditions seront encadrées par des parenthèses ().
- Les blocs d'instructions seront encadrés par des crochets {}
- Un commentaire d'une ligne commencera par deux 'slash' //
- Un bloc de commentaires (plusieurs lignes) s'écrit entre /* et */
- Un bloc de commentaires pour C#doc s'écrit entre /** et */
- Les crochets carrés sont utilisés pour les tableaux [ ]
- Le point (.) est généralement utilisé pour lier les méthodes (fonctions) aux variables (objets), c'est le lien entre le sujet et le verbe (moi.Écrire(quelquechose) ou document.Imprimer(), etc.)

7.4 Les instructions

Voici enfin les instructions :

```
Console.WriteLine("Hello World");
Console.ReadKey();
```

En français, nous pourrions dire "*sur la console, écrire 'Hello Word' puis sur la console, lire une touche du clavier*".



En cliquant sur ► Démarrer, une console doit s'ouvrir et afficher notre message.

7.5 AUTO-ÉVALUATION CHAPITRE 1

Voici quelques questions pour valider votre compréhension et approfondir vos connaissances…

Question ou affirmation	Vrai ou Faux
À la fin de chaque ligne, on ajoute un point	
Les instructions sont regroupées dans un paquet	
L'espace de nom contient les classes qui contiennent les méthodes (fonctions)	
La variable RdV_de_l'Heure est possible en C#	
Le mot-clé "using" est une instruction d'affichage	
Console.ReadKey() applique la fonction ReadKey() sur la console ouverte	

Réponses :

Faux c'est point-virgule ; Faux c'est regroupé dans un bloc ; Vrai ; Faux car contient un symbole (') ; Faux using intègre les bibliothèques ; Vrai.

8 TYPES DE DONNÉES

Dans les langages de programmation typés, c'est le programmeur qui choisit l'espace de valeurs pour les variables. En Python ou PHP (qui ne sont pas typés), l'interpréteur suppose le type automatiquement. En C#, C++, Java et de nombreux autres langages, le programmeur peut optimiser son code (pour le rendre plus performant en vitesse et occupation mémoire) en choisissant lui-même le type de ses variables.

8.1 LES TYPES PRIMITIFS

Seules les données numériques sont des types primitifs, avec deux exceptions : le caractère et le booléen.

Les types primitifs sont déclarés avec un mot-clé en minuscule devant le nom de la variable.

Type	Représente	Plage	Valeur par défaut
bool	bololéen	True ou False	False
byte	8 bits ~~non~~ signés	0 à 255	0
char	16 bits Unicode	U +0000 à U +FFFF	'\0'
decimal	128 bits	-7.9×10^{28} à $+7.9 \times 10^{28}$	0.0M
double	64 bits	$+/- 5 \times 10^{-324}$ à $+1.7 * 10^{308}$	0.0D
float	32 bits	-3.4×10^{38} à $+3.4 \times 10^{38}$	0.0F
int	32 bits signés	-2147483648 à +2147483647	0
long	64 bits signés	+/- 9223372036854775808 environ	0L
sbyte	8 bits signés	-128 à +127 (0x7F)	0
short	16 bits signés	-32768 à +32767 (0x7FFF)	0
uint	32 bits non signés	0 à 4294967295 (0xFFFFFFFF)	0
ulong	64 bits non signés	0 à 18446744073709551615	0
ushort	16 bits non singés	0 à 65535	0

Différences Java :
Les types non-signés n'existent pas en Java

Les types qui sont rayés n'existent pas en C# mais sont présents en C#.

8.1.1 DÉCLARATION

```
int unEntier;
char uneLettre;
```

8.1.2 AFFECTATION

```
unEntier = 25;
uneLettre = 'c';
```

Il est possible de faire la déclaration et l'affectation sur la même ligne :

```
double unDecimal = 2.5;
```

8.2 LES CLASSES CONTENEURS

Pour manipuler plus facilement les types primitifs ou permettre l'utilisation de chaînes de caractères, C# propose les classes conteneurs (ou classes références, en anglais "wrapper classes"). On les reconnaît par la présence d'une majuscule dans le type : cela indique que le type est un objet.

Les classes conteneurs usuelles sont :

- Byte, Sbyte, Int16, Int32, Int64
- Double
- Char, Boolean

8.2.1 DÉCLARATION

```
Int32 unEntier;
String unMot;
Boolean monDrapeau;
```

8.2.2 AFFECTATION

```
unEntier = 25;
unMot = "Bonjour le monde";
monDrapeau = true;    // ne peut prendre que true ou false
```

En effet, certaines méthodes ne permettent que l'utilisation d'objet (comme les méthodes de la classe ArrayList). Il était nécessaire d'encapsuler les types primitifs dans un objet. Par exemple :

```
int myInt = 5 ;
list.add(new Int32(myInt)) ;   // ce mécanisme de boxing est relativement lourd. Les classes conteneurs facilitent tout.
```

8.3 LES CONSTANTES

Pour déclarer n'importe quelle variable (on dit aussi attribut), il suffit d'ajouter le mot-clé final devant la déclaration. Cette variable n'est alors plus modifiable ailleurs.

```
const double pi = 3.14159;
```

8.4 DÉCLARATION D'UNE VARIABLE SANS TYPE : VAR

C# permet (avantage sur Java) de déclarer une variable, sans préciser le type. Lors de la première affectation, le compilateur essayera de deviner son type.

```
var maVariable = 1.68 ;  // variable de type float ou double
```

8.5 LES CONVERSIONS

L'interpréteur C# refuse les conversions qui risquent de faire perdre de l'information.

8.5.1 CONVERSIONS IMPLICITES

Ainsi, l'opération suivante ne pose aucun problème, car un entier est inclus dans les décimaux :

```
double monNombre = 0;
int monAutreNombre = 5;
monNombre = monAutreNombre;
```

8.5.2 CONVERSIONS EXPLICITES

En revanche, l'inverse nécessite que le programmeur 'confirme' de manière explicite le risque de perte d'information, par l'ajout du type souhaité entre parenthèse avant le nom de variable :

```
double monNombre = 7.35;
int monAutreNombre = 5;
monAutreNombre = (int)monNombre;    // on appelle cela un cast
```

*Pour synthétiser, un **byte** est inclus dans un **short** qui est inclus dans un **int** qui est inclus dans un **long** qui est inclus dans un **float** qui est inclus dans un **double**.un **char** sera inclus uniquement à partir d'un **int** car il n'a pas de bit de signe.*

8.5.3 CONVERSIONS DE CHAÎNE VERS NOMBRE

La conversion d'une chaîne de caractères vers un nombre est possible par l'utilisation de méthodes existant dans les classes correspondantes. Il faut donc utiliser les classes conteneurs vues précédemment.

8.5.3.1 LA MÉTHODE 'CONVERT.'

```
String maChaine="5.68";
Double d;
d = Convert.ToDouble(maChaine);
```

8.5.4 CONVERSIONS DE NOMBRE VERS CHAÎNE

L'opération permettant de convertir un nombre dans une chaîne existe aussi, en utilisant là encore des méthodes dans les classes conteneurs.

8.5.4.1 LA MÉTHODE 'TOSTRING'

```
String maChaine = "";
Double d = 5.68;
maChaine = d.toString();
```

9 LES OPÉRATIONS

9.1 LES OPÉRATIONS ARITHMÉTIQUES

9.1.1 OPÉRATIONS SIMPLES

Les opérations d'addition, soustraction, multiplication, soustraction et modulo sont supportées :

```
Double d = 5.68;
Int32 e = 120;
d = d * 1.20 / 5 + 3.14 - 6.222;
e = e % 100;         // retourne 20 car 120/100 = 1 et il reste 20
```

Les priorités sont respectées (multiplication et division passent avant l'addition et la soustraction)

9.1.2 OPÉRATIONS COMPLEXES

Les opérations trigonométriques, la génération de nombres aléatoires, l'utilisation de racines carrés ou puissances nécessitent les méthodes contenues dans la classe Math.

```
Double a, b, c, d;
a = Math.Sin(3.14/2);
b = Math.Pow(5,2);           // 5² ou 5 à la puissance 2
c = Math.Sqrt(c);            // √c (racine, en anglais, square)
```

Pour les nombres aléatoires, il faut utiliser la classe Random

9.1.3 OPÉRATIONS PARTICULIÈRES

Quelques opérations sont pratiques et doivent êtres connues.

- Incrémentation : d**++** (équivalent à d = d + 1)
- Décrémentation : d**--** (équivalent à d = d - 1)
- Cumul : total**+=** d (équivalent à total = total + d)

9.2 LES OPÉRATIONS BINAIRES

Il s'agit d'opérations sur les bits des variables : ET, OU, NON, << et >>, etc.

```
int a, b, c, d, e, f;
a = 0b1100 & 0b1010;    // ET : renvoie 1000 (notez la notation binaire '0b')
b = 0b1100 | 0b1010;    // OU : renvoie 1110
c = 0b1100 ^ 0b1010;    // OU exclusif : renvoie 0110
d = !0xF0;              // NON : renvoie 0F
e = 0x0F << 8;          // Décalage par 8 à gauche : renvoie F0
```

Attention à ne pas confondre opérations binaires et comparateurs (&&, ||, !=, ==, ≥=, <=...)

9.3 LES OPÉRATIONS DE CHAÎNES

Ce sont les opérations disponibles pour le traitement de chaînes de caractères ou de caractères : il faut cependant se souvenir que C# travaille en Unicode ! La taille d'un caractère n'est donc pas un octet comme en ASCII.

Différences Java :
En Java, SubString utilise PDepart et pFin
On utilise .toLowerCase et .toUpperCase

- Méthodes toLower() et toUpper() changent la casse de la chaîne
- Méthode length renvoie la taille de la chaîne (en nombre de caractère)
- Méthodes Substring(pDepart, pLongueur) affiche un ou plusieurs caractères
- Méthode Replace(chOrigine,chFinale) remplace une chaîne par une autre
- Méthode de recherche Contains(strRecherchee) pour trouver une chaîne dans une autre
- Méthode Split(strSepare) pour séparer une chaîne en plusieurs chaînes

Quelques exemples d'utilisations sont données ci-dessous :

```
String maChaine = "Bonjour";
String monPrenom = "Coco";
char maLettre = ' ';

Console.writeLine(maChaine.toLower());          // affiche 'bonjour'
Console.writeLine(maChaine.toUpper());          // affiche 'BONJOUR'
Console.writeLine(maChaine.length);             // affiche 7
Console.writeLine(maChaine.SubString(4,1));     // affiche 'j'
String messageCool = maChaine+maLettre+monPrenom; //contient 'Bonjour Coco'
String message2 = maChaine.concat(monPrenom);   // contient BonjourCoco
```

Voir https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx

9.3.1 FORMATAGE DES AFFICHAGES

L'affichage en C# n'est pas aussi puissant que la méthode printf() de Java.

Il faut donc passer par une conversion en une chaîne avec string.Format

Un exemple (utilisant les arguments {*numéro*}) :

```
double a = 5.6d;
double b = 2d;
double mult = a * b;
string s = string.Format("{0:0.00}", mult);

Console.WriteLine("{0} fois {1} égal {2}", a, b , mult);
Console.WriteLine(a.toString()+" fois "+b.toString()+" égal "+mult);
```

affichera : 5,6 fois 2 égal 11,20
5,6 fois 2 égal 11,20

10 LES TABLEAUX

Un tableau est un ensemble de variables de même type dans une quantité définie.

Chaque case du tableau est numérotée, en partant de 0 : un tableau contenant 10 valeurs aura les numéros de cases de 0 à 9 (on parle de l'indice de cellule).

10.1.1 DÉCLARATION

```
int[] monTableau =  { 1995, 1996, 1997, 1998, 2000, 2001 };
```

0	1	2	3	4	5
1995	1996	1997	1998	2000	2001

Une autre déclaration (pour un tableau de chaîne vide par exemple) est la suivante :

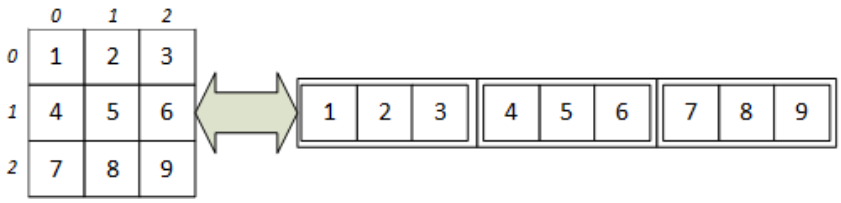
```
String[] monTableau = new String[9];
String[] autreTableau = new String[9];
```

Ces deux tableaux contiendront donc 10 chaînes de caractères (l'index commence à 0 !).

```
int tailleTableau = autreTableau.length;
```

Enfin, il est possible d'utiliser des tableaux à plusieurs dimensions (comme pour la bataille navale) :

```
byte[,] monTableau = new byte[5,4];
int[,] autreTableau2D = {{1,2,3},{4,5,6},{7,8,9}};
```



Le premier tableau est une grille de 10x10 éléments tandis que le deuxième tableau contient 3x3 éléments. Il faut donc voir les tableaux à dimensions multiples comme des tableaux de tableau.

10.1.2 AFFECTATION

Dans l'exemple ci-dessus, la case autreTableau2D[0] contient le tableau {1, 2, 3}

Il existe des opérations très pratiques sur les tableaux, notamment le tri (.sort()) ou le remplissage d'un tableau avec

var items = Enumerable.Repeat<bool>(true, 1000).ToArray();

```
AutreTableau2D[0,1] = 666;
Console.writeLine("la case coordonnées 0,1 a pour valeur "+autreTableau[0,1]);
```

11 LES ENTRÉES-SORTIES STANDARDS

En C#, les entrées et sorties concernent autant le mode console, que les fichiers et connexions. Seule la gestion des interfaces graphiques est différente (utilise les notions d'événements). Les notions utiles sont :

- Stream : un flux qui ne s'arrête pas, comme une émission en direct à la télévision
- Buffer : une zone mémoire pour ne pas perdre d'information (notamment si le flux n'est pas lu suffisamment souvent).

11.1 MODE CONSOLE

Pour afficher et récupérer du texte, les méthodes de la classe System et C#.util sont nécessaires.

11.1.1 AFFICHAGE

Voici comment comprendre le code d'affichage suivant :

```
Console.writeLine("Bonjour !");
```

- **System** est la classe utilisée pour l'affichage sur une console. Il n'est pas nécessaire de la déclarer si "using System" est placé en entête du programme.
- **Console** est l'objet dans cette classe, qui gère les sorties : ici, l'objet **console** est l'écran principal de la console. Il existe un objet **error** qui permet de noter les erreurs. L'objet **in** est utilisé pour récupérer les flux du clavier.
- **WriteLine()** est la méthode qui s'applique sur l'objet out : cette méthode envoie à cet objet les données comprises entre ses parenthèses. .

Exemple :

```
Console.writeLine("Coco !");      // affichera 'Bonjour Coco !' sur la même ligne
Console.Error.WriteLine("Un problème"); // affichera 'Un problème'
```

11.1.2 SAISIE

La console est ouverte par défaut, ainsi que le flux en provenance du clavier (plus pratique qu'en Java).

```
String reponse = Console.ReadLine(); // attend [Entrée] et stocke les données
ConsoleKeyInfo car = Console.ReadKey();   // attend l'appui d'une touche
```

Le type pour récupérer le résultat de ReadKey() est un type objet. Nous verrons cela plus tard.

Il n'existe pas de lecture de clavier caractère par caractère qui ne serait pas bloquant.

11.1.3 DEBUG

Il existe une classe en C# qui permet d'afficher des informations dans la console du débogueur. Pour cela, il faut activer la bibliothèque correspondante :

```
using System.Diagnostics ;
```

Ensuite, l'écriture vers cette console se fait par l'instruction suivante :

```
Debug.WriteLine("L'attribut resultat a pris la valeur "+resultat);
```

Enfin, pour visualiser la console dans Visual Studio, utilisez la séquence de touches CTRL+Alt+O

D'autres options sont disponibles pour une meilleure mise en forme, comme :

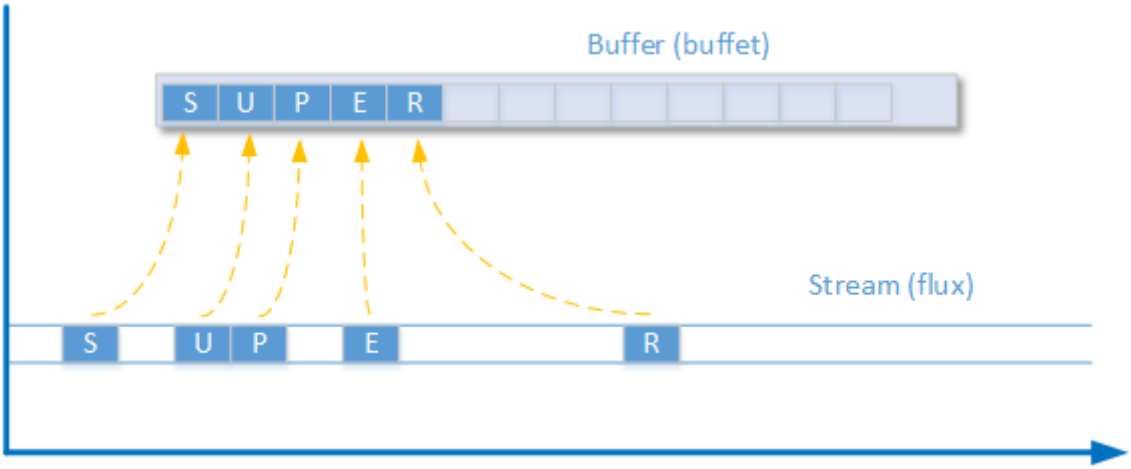
```
Debug.Indent() ;
Debug.Unindent() ;
```

L'intérêt de cette solution, est de ne pas polluer la console du programme principale, ou bien d'être utilisable dans les programmes de type WinForm.

11.2 FICHIERS

L'astuce ci-dessus nous amène à nous intéresser à la lecture et l'écriture de fichier.

Comme évoqué précédemment, pour C#, la plupart des périphériques sont considérés comme des flux entrants et sortants. Cela signifie que si le flux n'est pas mémorisé, les données transmises sont perdues (un peu comme une chaîne en direct : si vous êtes absent de l'écran, les informations sont perdues).



Comme la plupart des périphériques acceptent les entrées et les sorties, cela multiplie par deux les flux et buffets à déclarer.

Attention, la lecture d'un flux vide entraîne la génération d'une exception qui bloque l'exécution du programme.

Ainsi, la plupart du temps (et c'est aussi vrai en Java), il faudra ouvrir le flux et ouvrir un buffet :

1. Ouvrir le flux du fichier
2. Ouvrir le buffet du flux du fichier
3. Lire/Écrire dans le buffet
4. Fermer le buffet
5. Fermer le flux

Voyons cela avec un exemple, à la page suivante.

Voici donc un exemple de code pour lire un fichier texte, à copier tel quel pour le moment :

```
using System;
using System.IO;
using System.Text;

namespace FileReadExample
{
    class Program
    {
        static void Main(string[] args) {
            FileStream monFlux = null;       // Créer un objet flux (en direct)
            StreamReader reader = null;      // Créer un objet lecteur de flux (bufferise)
            String fileContent = "";
            try {
                // Ouverture
                monFlux = File.OpenRead(@"e:\test.txt");
                reader = new StreamReader(monFlux, Encoding.UTF8);

                // Lecture
                Console.WriteLine("Votre fichier contient : ");
                  fileContent = reader.ReadToEnd();
                  Console.WriteLine(fileContent);
             }
            catch (Exception ex) {
                Console.WriteLine("Erreur de lecture !"+ex.Message);
             }
            finally {
                // Fermeture
                if (reader != null) {
                    reader.Dispose();       // Fermer le buffer (libère sa mémoire)
                  }

                if (monFlux != null) {
                    monFlux.Dispose();      // Fermer le fichier (libère le canal)
                  }
                Console.Write("** Appuyez sur une touche **");
                  Console.ReadKey();
             }
         }
    }
}
```

Le programme se déroule en 3 temps (ouverture, lecture, fermeture) et nécessite un fichier texte présent sur le disque e:\ ! Vous pouvez donc créer un fichier texte ailleurs, en modifiant l'emplacement dans le programme.

Pour écrire dans un fichier, il faut utiliser

```
monFlux = File.Open(@"c:\testwrite.txt", FileMode.Append);
```

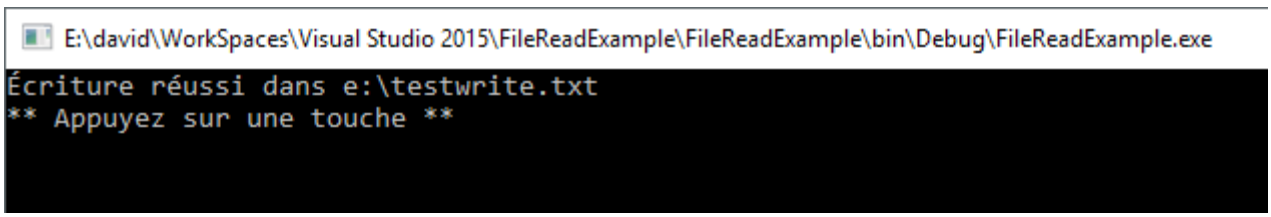
Et pour écrire

```
writer.WriteLine("Hello World pour les fichiers :");
```

Voici le code d'écriture :

```
using System;
using System.IO;

namespace FileReadExample
{
    class Program
    {
        static void Main(string[] args) {
            FileStream monFlux = null;              // Créer un objet flux (en direct)
            StreamWriter writer = null;          // Créer un objet de flux (bufferise)

            try {
                // Ouverture
                monFlux = File.Open(@"e:\testwrite.txt", FileMode.Append);
                writer = new StreamWriter(monFlux);

                // Lecture
                Console.WriteLine("Écriture réussi dans e:\\testwrite.txt ");
                writer.WriteLine("Hello World pour les fichiers :");
             }
            catch {
                Console.WriteLine("Erreur d'écriture !");
             }
            finally {
                // Fermeture
                if (writer != null) {
                    writer.Dispose();       // Fermer le buffer (libère sa mémoire)
                  }

                if (monFlux != null) {
                    monFlux.Dispose();      // Fermer le fichier (libère le canal)
                  }
                Console.Write("** Appuyez sur une touche **");
                  Console.ReadKey();
             }
        }
    }
}
```

et le résultat :



12 EXERCICE

12.1 ENREGISTRER UN FICHIER CONTENANT UNE FICHE CONTACT

Votre programme doit demander quelques renseignements à l'utilisateur puis les enregistrer sous forme de texte dans un fichier avec l'extension .txt

12.2 SOURCE FICHIER

https://openclassrooms.com/courses/apprenez-a-programmer-en-c-sur-net/lire-et-ecrire-dans-un-fichier-2

http://www.java2s.com/Tutorials/CSharp/System.IO/File/C_File_OpenRead.htm

13 BOUCLES ET CONDITIONS

Comme dans d'autres langages, C# propose un ensemble d'instructions pour créer des boucles et permettre des choix à l'aide de conditions.

13.1 BOUCLES

Il existe 3 boucles connues (répétition, test avant de commencer la boucle et test en fin de boucle).

13.1.1 BOUCLE FOR OU FOREACH

La boucle for permet un comptage ou une énumération : elle s'écrit avec 3 paramètres séparés par un point-virgule.

- Déclaration et initialisation de la variable de comptage
- Condition de sortie de boucle (tant que la condition n'est pas remplie, on reste)
- Choix de l'incrément (on peut aller de 2 en 2 ou bien partir dans le sens inverse en décrémentant la variable)

```
for (int t=0 ; t < 7 ; t++) {
    Console.writeLine("boucle N°"+t) ;
}
```

On peut aussi créer une boucle infinie comme ceci (*sortie possible avec une instruction 'break'*) :

```
for (;;) {
    Console.writeLine("ne s'arrête jamais... ahahah !") ;
}
```

Enfin, lorsqu'on a un tableau ou une liste d'objet, on peut la parcourir automatiquement (foreach)

```
int[] maTable1 = { 3, 14, 1, 59, 2, 65 };
Console.WriteLine("Taille de la table = " + maTable1.Length);
foreach (int cpt in maTable1) {
    Console.Write(":" + cpt + " ");     //cpt prend les valeurs 3 puis 14 puis 1, 59...
}
```

13.1.2 BOUCLE WHILE

Cette boucle classique propose **un test en début de boucle**. La boucle s'exécute tant que la condition reste vraie. Si la condition est fausse dès le début, les instructions du bloc ne sont pas exécutées.

```
int t=0;
while (t < 10) {
    // action à faire...
    t++;
}
```

13.1.3 Boucle Do… while

La condition se trouve à la fin de la boucle. Les instructions seront exécutées au moins une fois. Cependant, contrairement à une boucle repeat… until(), la sortie de la boucle se fait lorsque la condition est fausse (comme dans la boucle while classique).

```
do {
    nbreHasard = (int) (Math.random() * 49 + 1);
} while (grilleLoto.contains(nbreHasard));
```

Il est possible de sortir d'une boucle sans que la condition ne soit modifiée, par l'utilisation de l'instruction break !

```
int t=0;
while (t < 10) {
    if (t = 5) {
          break;
    }
    t++;
}
```

13.2 Conditions

Il y a deux types de conditions :

- les IF, ELSE IF, ELSE
- les SWITCH, CASE

13.2.1 Conditions IF - ELSE

Le contenu du bloc est exécuté si l'ensemble de la condition est vrai : s'il y a plusieurs conditions (dans la même parenthèse), elles doivent toutes être vraies. C'est le modèle le plus connu.

```
int testscore = 76;
char grade;
boolean work = true;

if (testscore >= 90 && work) {
  grade = 'A';
} else if (testscore >= 80) {
    grade = 'B';
} else if (testscore >= 70) {
  grade = 'C';
} else if (testscore >= 60) {
   grade = 'D';
} else {
grade = 'F';
}
Console.writeLine("Grade = " + grade);
```

13.2.2 CONDITIONS SWITCH - CASE

Un peu moins visible, ce système de conditions est très structuré :

```
a = 1;
switch(a) {
    case 0 :
          Console.writeLine("nul"); break;
    case 1 :
          Console.writeLine("un"); break;
    case 3 :
          Console.writeLine("trois"); break;
}
```

Il y a cependant un impératif : ajouter une instruction 'break' avant chaque nouveau bloc 'case', sinon dès qu'une condition est réalisée, tous les blocs suivants sont exécutés.

Essayez le programme précédent en enlevant les mots-clés 'break'. Le résultat est-il cohérent ?

D'autre part, en C#, la gestion des plages (intervalles) n'est pas intuitive ! Le code le plus proche est le suivant :

```
switch (num) {
  case 1: case 2: case 3: case 4: case 5:
     Console.writeLine("testing case 1 to 5");
     break;
  case 6: case 7: case 8: case 9: case 10:
     Console.writeLine("testing case 6 to 10");
     break;
  default:
     //
}
```

Les utilisateurs de Pascal et Purebasic seront surpris du manque de souplesse mais PHP et C# fonctionnent de la même manière que C#. Enfin, il n'y a pas de switch/case en Python.

13.2.3 PORTÉE DE VARIABLES

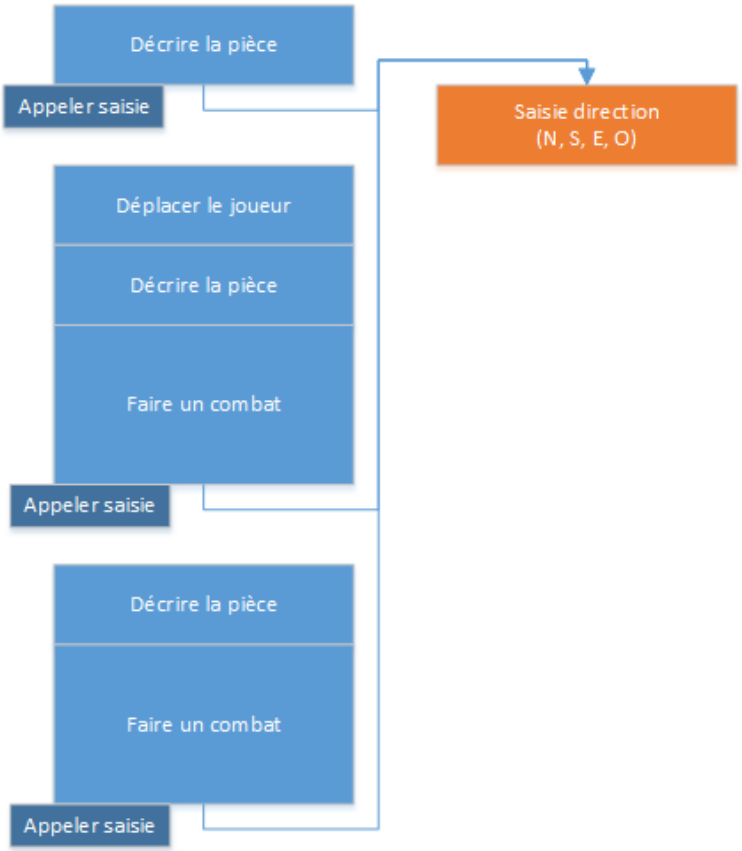
La portée est une notion qui signifie simplement que plus un attribut est déclaré dans un bloc supérieur, plus il est visible. Ce qui entraîne qu'un attribut déclaré dans une boucle, n'est visible que dans cette boucle. Pour rappel, un bloc est généralement représenté entre { }.

```
for (int x=0; x < 10; x++) {
    for (int y=0; y < 10; y++) {
        Console.writeLine(x+"-"+y);
        int z = x*y;
    }
    Console.writeLine("Cette ligne entraine une erreur. Z = "+z);
}
```

14 LES FONCTIONS

Les fonctions simplifient et fiabilisent le codage. En effet, effectuer les mêmes actions plusieurs fois à la suite peut se faire dans une boucle. Effectuer les mêmes actions à des moments différents nécessite de pouvoir exécuter le même morceau de code à plusieurs endroits du programme.

Voici les deux possibilités :

Avoir plusieurs fois du code identique	Appeler plusieurs fois un seul code
	

Il est évident que la deuxième solution est plus fiable, plus élégance et moins encombrante.

Une fonction est donc un bout de code, utilisable à plusieurs endroits.

14.1 Création d'une fonction

La syntaxe est la suivante :

portée static **type** nom(paramètre1, paramètre2…) { code et r**eturn** valeur }

dit comme ceci, ce n'est pas clair mais en C#, cela donne :

```
public static int maximum(int nombreA, int nombreB) {
    if (nombreA > nombreB) {
        return nombreA ;
    } else {
        return nombreB ;
    }
}
```

Le travail que va faire la fonction Maximum est d'évaluer quel est le nombre le plus grand et de renvoyer sa valeur.

Évidemment, il est possible de ne pas avoir de paramètre et/ou de ne pas renvoyer de valeur. Une fonction qui imprime sur l'imprimante par défaut serait (cas fictif) :

```
public void Imprimer() {
    SortirImprimanteVeille() ;
    LancerImpressionSpooler() ;
}
```

Cette fonction – qui appelle deux fonctions – ne renvoie pas de valeur, d'où le mot-clé 'void' qui signifie "vide". L'appel à une fonction sans valeur de retour se fait comme pour la fonction SortirImprimanteVeille() ou LancerImpressionSpooler() ;

14.2 Appel d'une fonction

Pour utiliser les fonctions ainsi créées, il suffit de les appeler par leurs noms, comme nous le faisons avec la fonction WriteLine() par exemple :

```
int A, B, Resultat ;
A = SaisirNombre() ;
B=SaisirNombre() ;
Resultat = Maximum(A, B) ;
Console.WriteLine(Resultat) ;
```

Comme Maximum() est une fonction, on peut l'utiliser partout :

```
monNombre = 100 ;
if (Maximum(25, monNombre) > 25) {
   Console.WriteLine("Le maximum est : "+Maximum(25, monNombre)) ;
}
```

14.3 ENTRAÎNEZ-VOUS : FONCTION SAISIRNOMBRE(INT MIN, INT MAX)

L'idée est de créer une fonction qui vérifie la saisie et n'accepte que des nombres entre nombreMin et nombreMax. Tant que le nombre saisi n'est pas dans l'intervalle, on recommence la saisie.

Aide N°1 :

Il faut utiliser une boucle, dont on ne sort que si le résultat est un nombre.

Aide N°2 :

La conversion d'une chaîne de caractère en entier utilise la fonction C# existante :

monNombre = Convert.ToInt16(maChaine)

Aide N°3 :

Il faudra utiliser les exceptions (try… catch) au moment de la conversion (voir chapitre suivant de ce livre)

Réponse

```
using System;

namespace SaisirNombre
{
    class Program
    {
        public static Int32 SaisirNombre(Int32 min, Int32 max) {
            String maChaine = "";
            Int32 nombre=0;
            do {
                Console.WriteLine("Saisissez un nombre entre "+min+" et "+max);
                  maChaine = Console.ReadLine();
                if (maChaine != "") {
                    try {
                        nombre = Convert.ToInt32(maChaine);
                      }
                    catch {
                        maChaine = "";
                      }
                  }
            } while ((maChaine == "") | (nombre > max) | (nombre < min));
            return nombre;
         }

        static void Main(string[] args) {
             SaisirNombre(0, 10);
         }
    }
}
```

15 LES EXCEPTIONS

C# fournit un moyen de contrôle des erreurs lors du déroulement des applications. Cette solution se faisant au détriment de la performance et nécessitant du code supplémentaire, il est fréquent que seules les parties sensibles du code soient protégées.

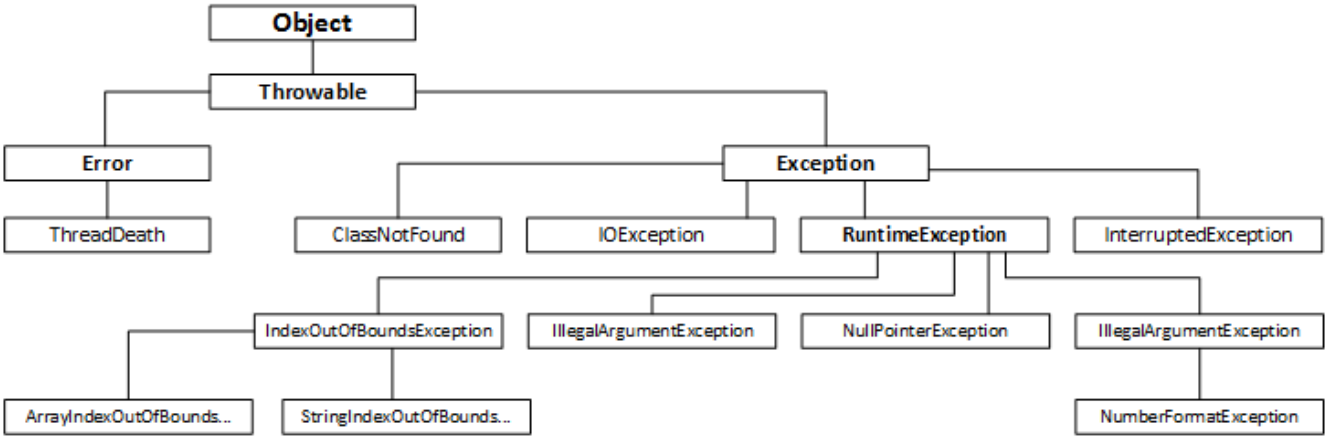
15.1 ARBORESCENCE DES EXCEPTIONS

Il existe plusieurs exceptions qui peuvent survenir par différentes situations :

- Lecture de données dans un fichier vide
- Lecture d'un fichier sur un média inexistant (retrait de clé USB par exemple)
- Division par zéro dans un calcul
- etc.

Une grande partie de ces erreurs sont habituelles et ne doivent pas bloquer le programme ou le planter. C# peut donc décider d'avertir le programme du problème, et si le programmeur n'a rien prévu, arrêter le programme.

Voici les classes des objets liés aux erreurs fréquentes (par exemple, la lecture d'un index de tableau au-delà de la taille maximale du tableau générera une exception "ArrayIndexOutOfBounds").



15.1.1 EXEMPLE DE CODE AVEC ERREUR

L'exemple le plus simple est la gestion d'une division par zéro.

```
public static void main(String[] args) {
   int a=20;
   int b=0;
   Console.writeLine(a+" divisé par "+b+" égal "+a/b);
}
```

Ce code entraînera l'erreur ArithmeticException :

```
Exception in thread "main" C#.lang.ArithmeticException: / by zero
    at net.roumanet.Main.main(Main.C#:19)

Process finished with exit code 1
```

15.2 GESTION DES EXCEPTIONS RENCONTRÉES

La méthode est relativement simple et repose sur quelques instructions :

- try { … } : le bloc contenant le code contenant un risque d'exceptions
- catch { … } : un des blocs dédié au traitement des erreurs. Il peut y avoir plusieurs catch {}
- finally { … } : le bloc qui traite la sortie du code normal (try) ou des codes de traitements des erreurs (pour fermer un fichier par exemple).

15.2.1 EXEMPLE DE CODE AVEC GESTION D'ERREUR

Dans le code précédent, il suffit d'ajouter try/catch sur l'erreur ArithmeticException pour pouvoir afficher un message d'erreur plutôt qu'arrêter l'application.

```
public static void main(String[] args) {
   // Préparation des attributs
   int a=20;
   int b=0;
   try {
      Console.writeLine(a + " divisé par " + b + " égal " + a / b);
   }
   catch (ArithmeticException e) {
      Console.writeLine("le diviseur est nul : division par zéro impossible");
   }
}
```

Ici, le programme ne traitera que cette erreur, mais il est possible de traiter chaque erreur (ou exception) différemment : c'est l'ordre des catch {…} qui déterminera le premier traitement.

Il est également possible de récupérer le message d'erreur de l'exception, comme le montre le code ci-dessous :

```
catch (Exception ex) {
    Console.WriteLine("Erreur ! " + ex.Message);
}
```

16 LES INTERFACES GRAPHIQUES

Jusqu'à maintenant, nous avons programmé dans le mode console : il s'agit d'une interface homme-machine austère mais facile à utiliser !

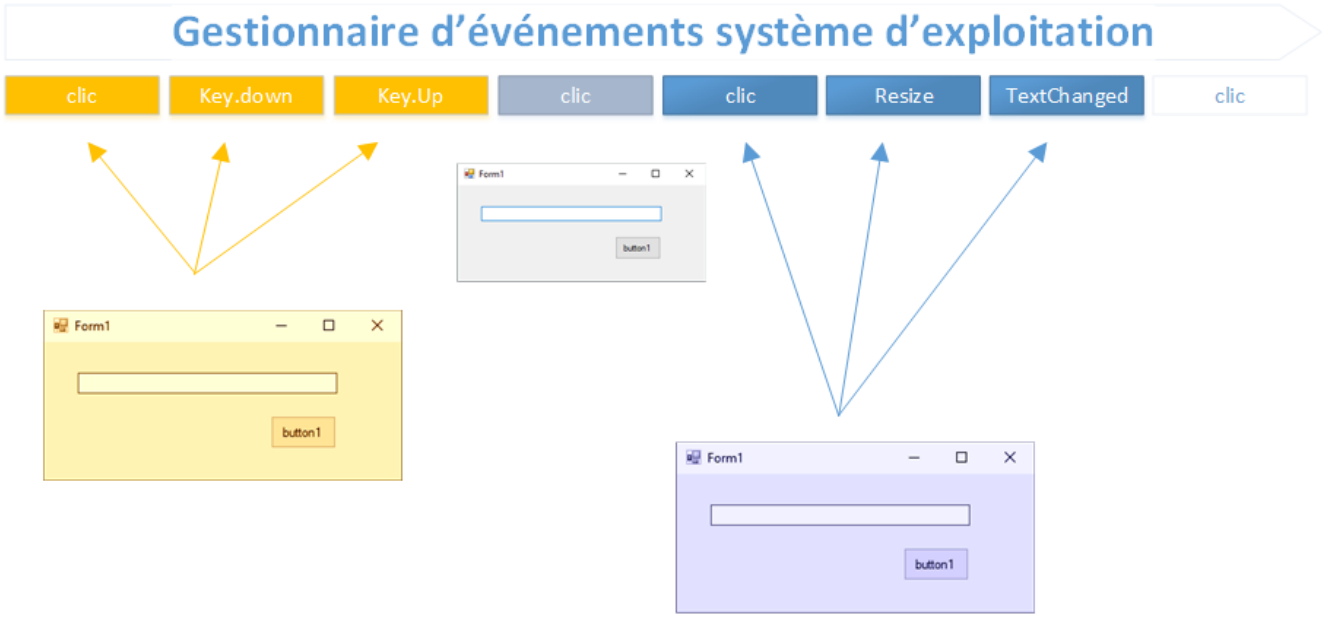
- L'ordinateur présente une information
- L'ordinateur attend une saisie… et ne fait rien d'autre
- Une fois la saisie effectuée, l'ordinateur continue le traitement

Cette programmation très séquentielle n'est pas celle utilisée sur les systèmes d'exploitation graphiques.

16.1 FONCTIONNEMENT

Dans les OS modernes, c'est ce dernier qui gère tous les événements : le système "prête" des ressources aux programmes. Une fenêtre, un champ de saisie, un bouton… ces éléments sont dessinés et gérés par l'OS !

Dès lors, chaque clic, chaque action de l'utilisateur est enregistré par le système :



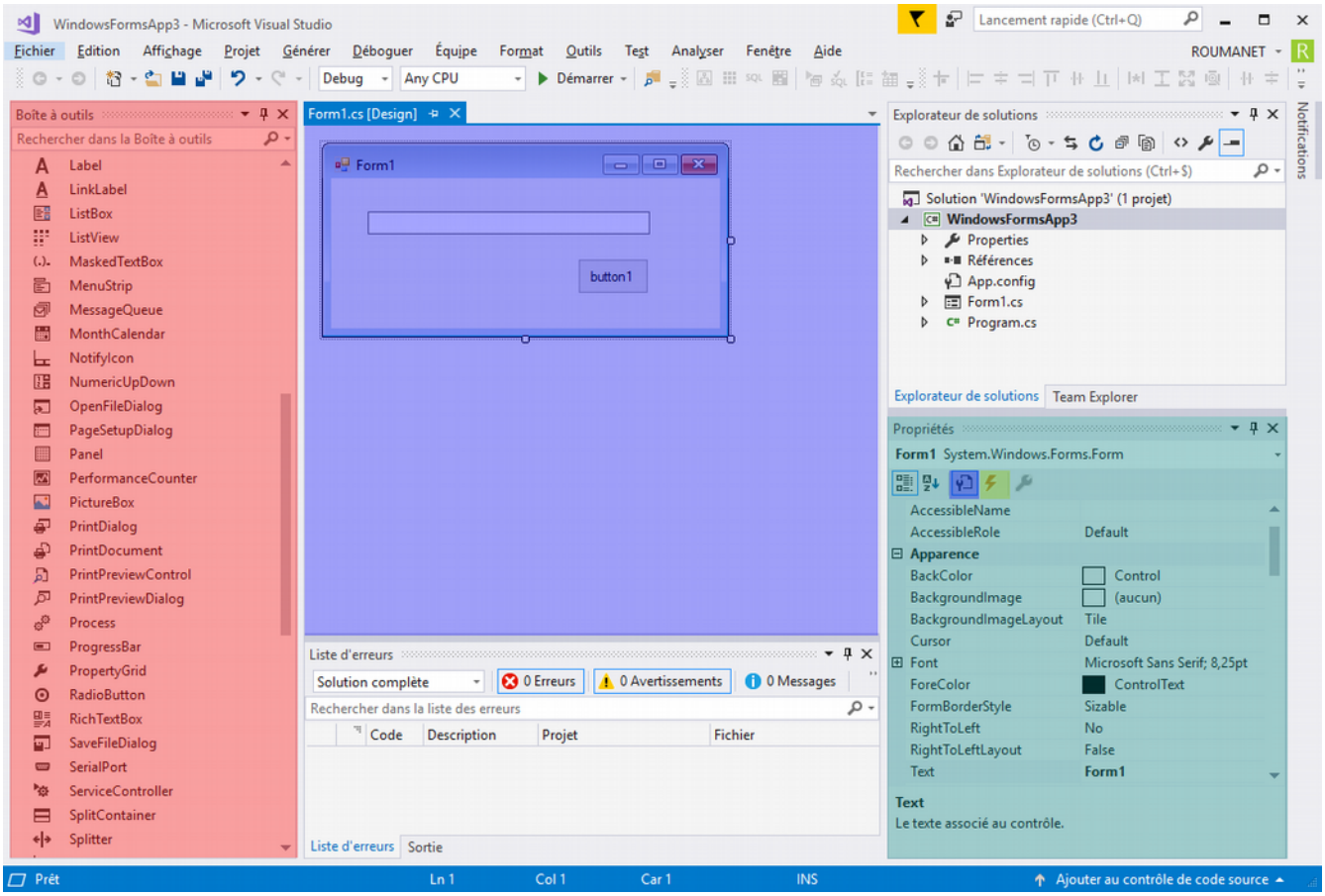
Lorsqu'il y a un événement qui concerne l'application, l'OS lui envoie le numéro de l'événement : le programmeur décide s'il doit réagir à cet événement. L'avantage est qu'une application qui n'est pas utilisée ne consommera pas de puissance à sa propre surveillance.

En C#, Visual Studio nous propose un éditeur graphique de fenêtre et masque une partie du codage.

16.2 Visual Studio et projet Windows Forms

Dans Visual Studio, créez un nouveau projet de type Windows Forms Application (Net Framework).

L'IDE s'affiche avec 3 parties importantes :



Voici l'utilité des différentes parties :

- La zone bleue affiche la forme sur laquelle vous travaillez : grâce aux poignées de sélection, vous pouvez redimensionner tous les éléments.
- La zone verte contient les propriétés de l'élément sélectionné : c'est ici qu'on peut renommer un champ, changer sa couleur, bloquer ses propriétés (éditable ou non, etc.
  - Dans la zone verte, l'icône verte représentant un éclair permet d'éditer les actions à effectuer en fonction des événements. Un clic, un touche enfoncé, un changement…
  - Pour revenir aux propriétés statiques de l'élément, il faut cliquer sur l'icône ayant une clef à molette (champ bleu dans la zone verte sur l'image).
- La zone rouge contient la boite à outils : tous les éléments utilisables dans la fenêtre peuvent être sélectionnés et placés sur celle-ci. Pour afficher la boite à outils : CTRL+Alt+X

16.2.1 LES FICHIERS

La création d'une application en mode console générait un seul fichier, Program.cs.

La création d'une application en mode fenêtre génère trois fichiers :

- Program.cs qui reste le fichier principal. Cependant, la fonction Main() est désormais différente.
- **Form1.cs (ou MainForm.cs sous SharpDevelop). Ce fichier contient les actions et fonctions que le programmeur peut gérer et modifier**.
- Form1.Designer.cs (ou MainForm.Designer.cs) : ce fichier contient les composants qui sont inclus dans la fenêtre : positions, tailles, propriétés y sont définis. Il est préférable de ne pas modifier manuellement ce fichier.

Il faut comprendre la philosophie d'usage : et pour cela, parler légèrement du modèle MVC. Il s'agit d'un modèle qui permet de simplifier le développement, en séparant la vue (les fenêtres, l'affichage), le contrôleur (les interactions de l'utilisateur : clic, saisie au clavier, déplacement de souris…) et le modèle de stockage des données (une base, un fichier, la mémoire…). MVC signifie Modèle – Vue – Contrôleur.

16.2.2 LES MORCEAUX DE CODE

Voici quelques explications sur le contenu de chaque fichier. Tout ne sera pas expliqué ici, mais cette partie devrait suffire pour vous permettre de créer des applications graphiques.

16.2.2.1 PROGRAM.CS

Il contient l'appel de la fenêtre graphique. Comme c'est une programmation événementielle, le programme devient multi-tâche (il accepte notamment d'être géré par le système).

Visual Studio

```
[STAThread]
static void Main() {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

Sharp Develop

```
[STAThread]
private static void Main(string[] args) {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainForm());
}
```

La ligne importante est :

```
Application.Run(new Form1()) ;
```

"Run" indique un fonctionnement en thread et "new Form1()" la demande de création de la fenêtre au système.

16.2.2.2 FORM.DESIGNER.CS

C'est le fichier qui contient la construction de la fenêtre. On y trouve notamment la fonction (on dira "méthode") qui génère les composants de **manière procédurale**.

En effet, si vous devez créer 40 composants identiques (imaginez une application contenant un champ par jour de la semaine), dessiner ces composants manuellement est fastidieux. La syntaxe utilisée ici, permettra de générer dans une méthode conçue par le développeur, les composants répétitifs.

```
        private void InitializeComponent() {
             this.textBox1 = new System.Windows.Forms.TextBox();
             this.button1 = new System.Windows.Forms.Button();
             this.SuspendLayout();
             //
             // textBox1
             //
             this.textBox1.Location = new System.Drawing.Point(32, 29);
             this.textBox1.Name = "textBox1";
             this.textBox1.Size = new System.Drawing.Size(246, 20);
             this.textBox1.TabIndex = 0;
             //
             // button1
             //
             this.button1.Location = new System.Drawing.Point(215, 70);
             this.button1.Name = "button1";
             this.button1.Size = new System.Drawing.Size(62, 31);
             this.button1.TabIndex = 1;
             this.button1.Text = "Cliquez ici";
             this.button1.UseVisualStyleBackColor = true;
             //
             // Form1
             //
             this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
             this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
             this.ClientSize = new System.Drawing.Size(339, 131);
             this.Controls.Add(this.button1);
             this.Controls.Add(this.textBox1);
             this.Name = "Form1";
             this.Text = "Form1";
             this.ResumeLayout(false);
             this.PerformLayout();

          }
```

Notez que les attributs des composants sont suivis par les propriétés puis les valeurs :

```
this.button1.Text = "textBox1";
```

Cette ligne signifie que pour le composant button1, on place la valeur "Cliquez ici" dans la propriété "Text".

16.2.2.3 FORM1.CS

C'est ce fichier qui contiendra les interactions entre l'utilisateur et l'interface du programme. Ainsi, les lignes suivantes méritent d'être expliquées :

```
        public Form1() {
             InitializeComponent();
         }

        private void button1_Click(object sender, EventArgs e) {
            MessageBox.Show("Bravo, vous avez cliqué sur le bouton !");
         }
```

La méthode "Form1()" est publique, donc visible de partout. Elle ne contient qu'une ligne qui appelle la construction de la fenêtre, dans le fichier Form1.Designer.cs.

La méthode "button1_Click()" correspond aux actions qui seront faites lorsque l'événement correspondant est appelé. Ici, on affiche une simple boite d'alerte. Notez cependant, que cette méthode reçoit deux paramètres : l'appelant (sender) et le type d'événement (e).

en effet, lors de l'initialisation des composants, on indique au système (dans le fichier Form1.Designer.cs) que le programme surveillera les événements qui touchent à ce bouton :

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Ces informations dépassent le niveau du module SI4 : il s'agit cependant d'une culture générale pour comprendre comment fonctionne les applications dans les systèmes graphiques fenêtrés.

16.3 CE QUI FAUT RETENIR EN PROGRAMMATION ÉVÉNEMENTIELLE

L'IDE permet de générer automatiquement les formes (fenêtres) et composants nécessaires à vos applications.

Le déroulement de l'application n'est plus linéaire : l'utilisateur peut cliquer sur n'importe quel composant et ce, dans le désordre. En tant que programmeur, vous devez donc vous assurer que les opérations à réaliser sont possibles au moment de l'appel à vos fonctions.

L'OS met à la disposition du programmeur des bibliothèques de composants et leurs comportements associés : il faut donc connaître les composants principaux[1] : textBox, button, checkBox, listBox, …

1 https://www.codeproject.com/Articles/1465/Beginning-C-Chapter-Using-Windows-Form-Controls

17 LES OBJETS

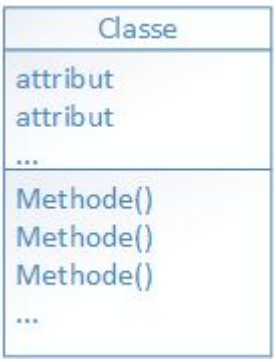
En C#, les objets s'utilisent un peu comme des "super" variables.

Un objet en C#, correspond à un ensemble de variables et des fonctions intégrées. Grâce à une notion importante appelé encapsulation, il est possible de n'accéder à ces variables qu'au travers de ces fonctions.

Ainsi, une classe est donc un bloc autonome, et les objets créés à l'aide de cette classe ne peuvent pas être mis en "pièces détachées".

17.1 VOCABULAIRE

Le langage orienté objet propose un vocabulaire à connaître impérativement. Ci-contre, la représentation symbolique d'une classe.



Une **classe** est constituée de **membres** :

- les **attributs** : ce sont les variables de la classe. Sauf s'ils sont publics, les attributs ne sont pas visibles depuis une autre classe (étrangère).
- Les **méthodes** : ce sont les fonctions ou procédures de la classe. Elles aussi peuvent être publiques ou privées.

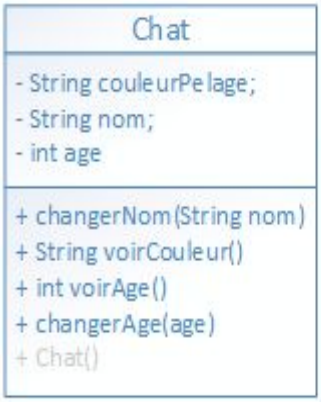
Une classe n'est qu'un patron, un plan d'objet : il faut construire les objets à partir de ce plan, en objet le terme utilisé : **instancier**.

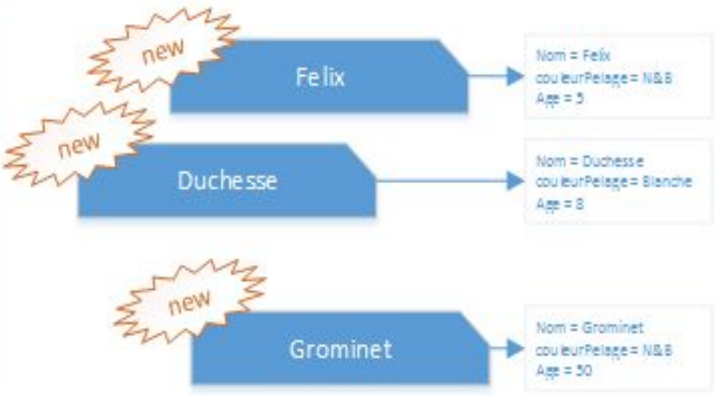
Pour instancier une classe, on utilise une méthode de la classe particulière, qui porte le nom de la classe, et qu'on appelle un **constructeur**.

17.2 REPRÉSENTATION

Voici un exemple graphique :

La classe Chat contient les attributs privés





couleurPelage, nom et age (un signe – les précède).

Elle contient les méthodes publiques (signe +) permettant de modifier ou lire les attributs : cela

signifie qu'il n'est possible de modifier ces attributs que par ces méthodes, ce qui est très sécurisant.

17.2.1 INSTANCIATION

Pour créer le chat Félix, il suffit d'instancier la classe Chat et de stocker le résultat dans une "variable" de type Chat (ce n'est pas un entier, une chaîne… c'est un chat).

La syntaxe d'instanciation est la suivante :

```
Chat felix = new Chat();

felix.changerNom("Felix");
felix.changerAge(5);
```

le mot-clé **new** indique à C# de construire un objet, mais comme pour une maison, l'objet est initialement vide.

Notez qu'il n'est pas nécessaire de nommer l'objet avec le véritable nom du chat.

```
Chat chatRoumanet = new Chat() ;
chatRoumanet.changerNom("Pirouette") ;
…
```

A noter : si le constructeur de la classe n'est pas déclaré explicitement, C# va le créer automatiquement, avec tous les attributs par défaut.

Astuce : il est possible d'avoir plusieurs constructeurs, à la condition que le nombre d'arguments soit différents pour chaque constructeur et qu'il soit public (donc visible par tous).

Le mot-clé **this** permet de préciser à C# qu'il s'agit de l'attribut de l'objet en cours de création, permettant de ne pas mélanger avec le nom de l'argument transmis :

```
public Chat() {}

public Chat(String nom) {
    this.nom = nom;      //this.nom correspond à l'objet, nom à l'argument méthode
}

public Chat(int age, String nomChoisi) {
    this.age = age;
    this.nom = nomChoisi ;
}
```

17.2.2 Accesseurs et mutateurs

Les méthodes qui permettent de lire et modifier les attributs privés dans la classe, sont appelées des **accesseurs** et **mutateurs**. En anglais, les termes pour…

- les accesseurs qui permettent de lire : les **getters**
- les mutateurs qui permettent de modifier : les **setters**

Dans la classe Chat, Les méthodes voirAge() est un accesseur et changerAge(int age) est un mutateur.

17.2.3 Portées et modificateurs

Plusieurs fois présenté pour certaines, les portées servent à protéger les attributs et les méthodes. Les 3 modificateurs importants pour le moment sont private, public et static.

Dans le détail,

17.2.3.1 Pour les classes

static	La classe ne peut pas être instanciée : elle est unique et définie une seule fois en mémoire.
abstract	La classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe et qui implémente les méthodes nécessaires.
final (sécurité)	La classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private (accessibilité)	La classe n'est accessible qu'à partir du fichier où elle est définie.
public (accessibilité)	La classe est accessible partout. (C'est la valeur par défaut, mais il faut la préciser pour que la classe soit accessible en dehors de son package.)

17.2.3.2 Pour les méthodes

static	La méthode static déclarée statique ne peut agir que sur les attributs de la classe et pas sur les attributs des instances. La méthode main() est une méthode statique car il ne peut y avoir qu'un point de démarrage du programme.
Abstract	La méthode n'a pas de code. En cas d'héritage par une classe fille, celle-ci devra définir sa propre méthode.
private (accessibilité)	La méthode n'est accessible qu'à l'intérieur de la classe où elle est définie.
public (accessibilité)	La méthode est accessible partout.

17.2.3.3 POUR LES ATTRIBUTS

static	L'attribut est unique pour l'ensemble des instances. Il est généralement utilisé pour un comptage
final	L'attribut est une constante : il ne peut être modifié après sa première initialisation
private (accessibilité)	L'attribut n'est accessible qu'à l'intérieur de la classe où elle est définie.
Protected (accessibilité)	L'attribut est accessible par sa classe mais aussi les classes filles (héritage)
public (accessibilité)	L'attribut est accessible partout. Exemple : maVoiture.puissance = 215;

17.2.4 ANALYSE D'UN PROGRAMME

Le premier programme saisie (HelloWorld) peut donc maintenant être totalement analysé :

```
/* voici un exemple de code */
public class HelloWorld {
    public static void main(String[] args) {
        // Prints "Hello, World" to the terminal window.
        Console.writeLine("Hello, World");
    }
}
```

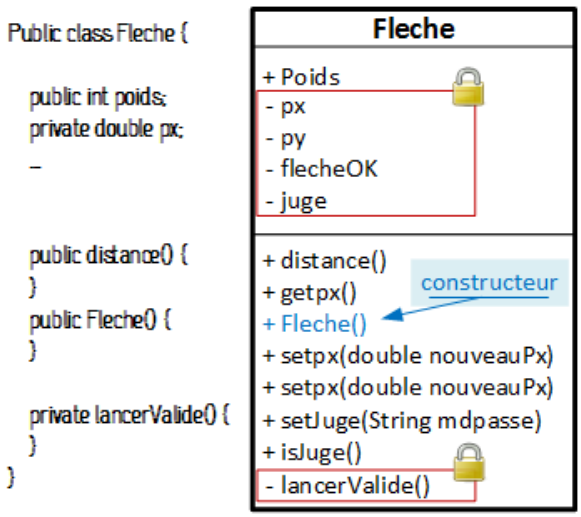
La classe publique HelloWorld n'a pas d'attribut.

Elle ne contient qu'une méthode publique dont voici les arguments :

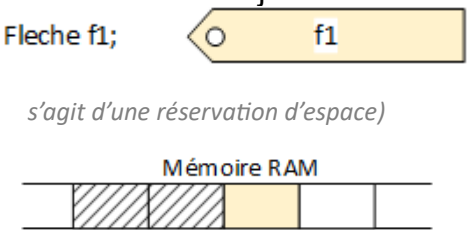
public	Il s'agit d'une méthode publique, utilisable depuis une autre classe
static	Cette méthode n'est pas instanciable : c'est toujours le cas pour main() qui est la méthode de lancement des applications (il ne peut en exister qu'une seule officielle par programme)
void	La méthode ne renvoie aucun argument (le type void signifie 'vide')
main	Le nom de la méthode
String[] args	La méthode accepte des arguments en entrée : un tableau de chaîne de caractères.

17.3 RÉSUMÉ

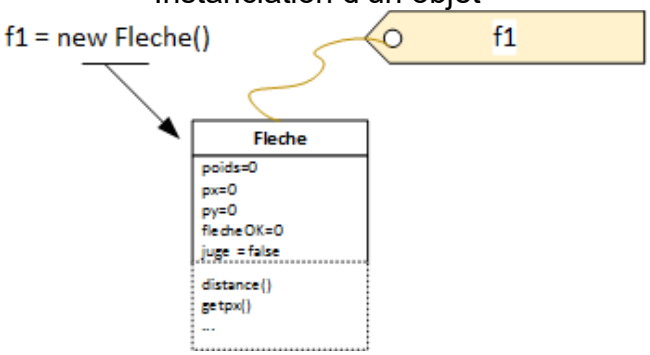
Création d'une classe



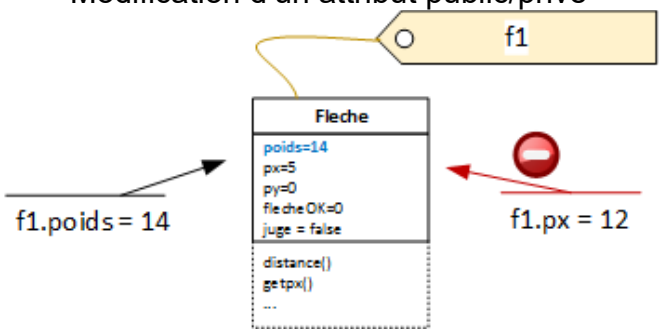
Déclaration d'un objet de la classe



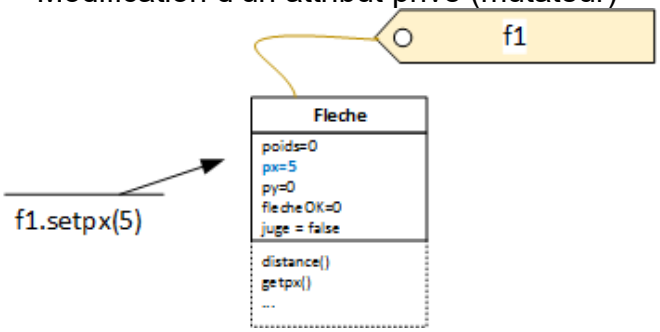
Instanciation d'un objet



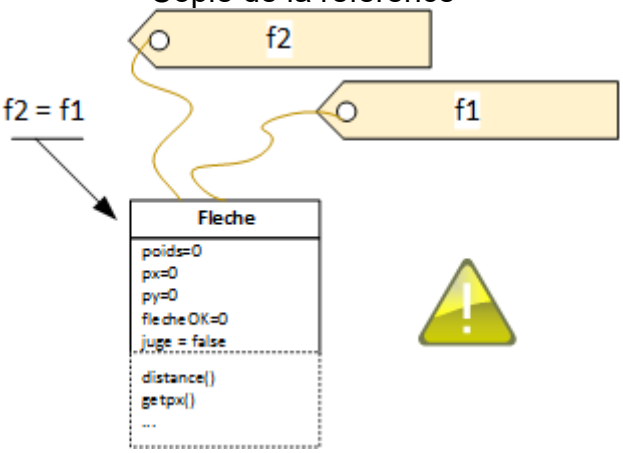
Modification d'un attribut public/privé



Modification d'un attribut privé (mutateur)



Copie de la référence

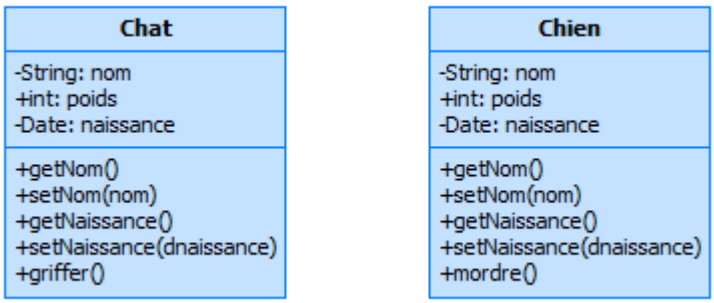


17.4 HÉRITAGE

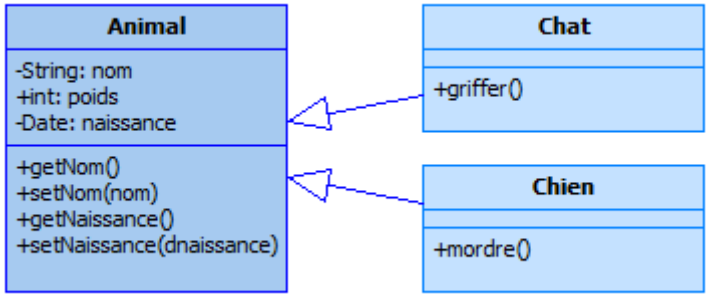
L'intérêt majeur de la programmation objet est probablement l'héritage : cette notion permet en effet de profiter des propriétés d'une classe existante et de pouvoir étendre ses capacités.

17.4.1 EXEMPLE

Une clinique vétérinaire qui doit enregistrer des chats et des chiens pourrait avoir deux classes, comme dans l'image suivante :



Cependant, on constate que de nombreux éléments sont communs, hormis la particularité des chats pour griffer et celle des chiens pour mordre.
L'héritage permet de simplifier le développement en ne créant qu'une seule fois les membres (attributs et méthodes) communs et en créant les classes propres aux deux espèces.



17.4.2 DÉCLARATION

La déclaration d'un héritage de classe se fait par l'utilisation du mot 'extends' :

```
public class Chat : Animal {
    public void griffer() {
        Console.writeLine("Griffé !") ;
    }
}
public class Chien : Animal {
    public void mordre() {
        Console.writeLine("Mordu !") ;
    }
}
```

Contrairement au C++, En C# (et même en Java) une classe ne peut hériter que d'une seule classe.

17.4.3 UTILISATION

Une fois déclarée, la classe s'utilise normalement.

```
Chat monChat = new Chat();
Chien monChien = new Chien();
Animal maTortue = new Animal();
```

17.4.4 EXEMPLE DE CODAGE (TD À REPRODUIRE)

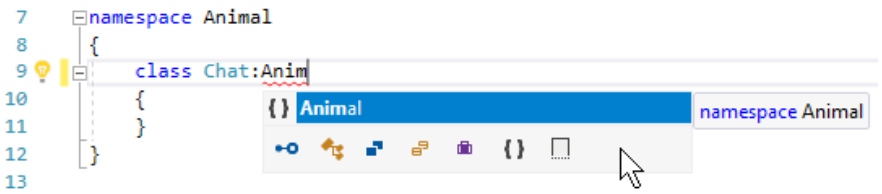
Dans la pratique, il n'est pas possible de placer les classes filles dans le même fichier que la classe mère. Il faut donc créer un fichier par classe.

Voici la classe mère Animal (nommer le fichier Animal.cs) :

```
/**
* Created by david on 09/04/2017.
*/
using System;

namespace Animal
{
    public class Animal
    {
        protected String nom;
        protected Int16 poids;
        protected DateTime naissance;

        // constructeur explicite pour créer et affecter les attributs en même temps
        public Animal(String nom, Int16 poids, DateTime dnaissance) {
            this.nom = nom;
            this.poids = poids;
            this.naissance = dnaissance;
         }
        public Animal() {
            this.nom = "(inconnu)";
            this.poids = 0;
            this.naissance = DateTime.Now.Date;
         }
        public String getNom() {
            return nom;
         }
        public void setNom(String nom) {
            this.nom = nom;
         }
        public DateTime getNaissance() {
            return naissance;
         }
        public void setNaissance(DateTime naissance) {
            this.naissance = naissance;
         }
        static void Main(string[] args) {
         }
    }
}
```

Toujours dans le même namespace (package), clic droit, Ajouter ► Class

Il suffit d'écrire les quelques lignes ci-contre et de valider que l'IDE présente les attributs de la classe mère lorsque vous créez le constructeur de la classe.



Ici, naissance, nom et poids (ordre alphabétique) appartiennent bien à la classe Animal dans le fichier Animal.cs.

La classe Chat étend la classe Animal en ajoutant le miaulement (méthode Miaule()) :

```
using System;

namespace Animal
{
    class Chat : Animal
    {
        public Chat() {
            this.nom = "Minou inconnu";
         }
        public String Miaule() {
            return "Miaou";
         }
    }
}
```

Enfin, éditer la méthode main() de votre programme comme suit :

```
using System;

namespace Animal
{

    public static void main(String[] args) {
           // Préparation des attributs
           Chat leMinou = new Chat();
           Console.writeLine("Le chat ayant pour étiquette 'leMinou' s'appelle
"+leMinou.nom);
    }
}
```

Le résultat sera :

```
"C:\Program Files (x86)\Java\jdk1.8.0_121\bin\java" ...
Le chat ayant pour étiquette 'leMinou' s'appelle Minou inconnu

Process finished with exit code 0
```

17.4.5 EXERCICE

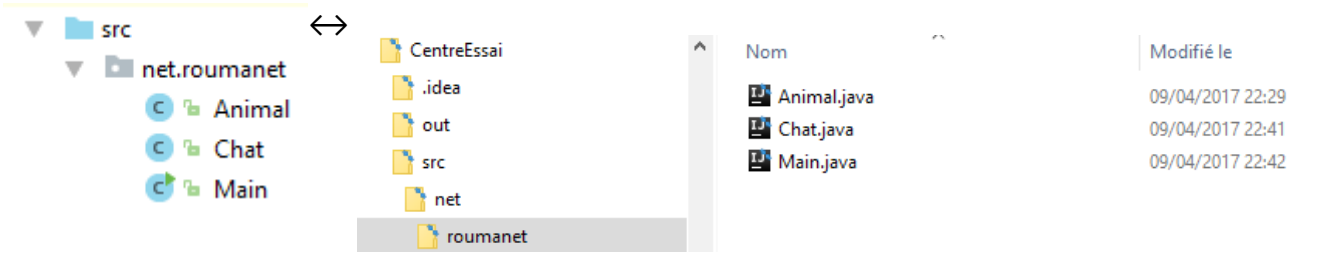
Finissez la classe Chat en ajoutant la méthode griffer(). Cette méthode contient un tirage au hasard entre 0 et 0.999 et affichera

"*le-nom-du-chat* a essayé de vous griffer" si le tirage est supérieur à 0.8, sinon

"*le-nom-du-chat* vous aime bien et ne vous griffe pas".

Créez également la classe Chien en y intégrant la méthode mordre() ayant un comportement similaire que la méthode griffer().

Pour validation auprès du professeur, envoyez seulement le contenu du répertoire 'src' dans un fichier ZIP.



18 LES CLASSES ABSTRAITES ET INTERFACES

Les classes sont des modèles pour créer des objets. Coder une classe ne permet pas de créer des objets tant que l'on instancie pas la classe (xxx = new Class()) et que la classe ne contient pas de constructeur.

Il est pourtant possible de créer des modèles de classe ou des modèles de méthodes. L'utilité est de s'assurer que les utilisateurs de ces modèles proposeront bien des classes ayant des critères communs, par exemple :

- Il existe plusieurs formes géométriques (en deux dimensions), pour chacune on veut obtenir la surface et le périmètre. Les formules de calculs étant différentes, il faut fixer 2 méthodes Aire() et Perimetre() que chaque codeur de forme devra utiliser
- Le remplissage d'une forme est commun à toutes les formes. Il n'est pas nécessaire que chaque codeur ré-écrive une méthode de remplissage.

18.1 CLASSE ABSTRAITE

Les classes abstraites sont une solution à l'exemple ci-dessus : le développeur de la classe 'Forme' va créer les méthodes communes et laisser "vide" les méthodes de calculs. On dit que l'implémentation est **partielle**.

```
namespace ClassesAbstraites
{
    public abstract class Formes {
        protected int couleurForme;
        protected float aireForme;
        protected float perimetreForme;

        public Formes() {
             couleurForme = 0;
             aireForme = 0;
         }
        public void Remplissage(int color) {
            this.couleurForme = color;
         }

        public abstract void Aire(float[] valeurCalcul);
        public abstract void Perimetre(float[] valeurCalcul);
    }
    // voici comment utiliser la classe abstraite
    public class Carre : Formes {
        public override void Aire(float[] valeurCalcul) {
            this.aireForme = valeurCalcul[0] * valeurCalcul[0];
         }
        public override void Perimetre(float[] valeurCalcul) {
            this.aireForme = valeurCalcul[0] * 4;
         }
    }
}
```

L'exemple donné montre comment la classe 'Carre' implémente la classe 'Formes' :

- Dans la classe Formes ► Les méthodes Aire() et Perimetre() sont déclarées comme étant abstraites, grâce au mot-clé '**abstract**'.
- Dans la classe Carre ► On écrit les méthodes Aire() et Perimetre() en *recouvrant* la définition d'origine, grâce au mot-clé '**override**'.

18.1.1 EXERCICE

Ajoutez une classe 'Disque' dont on calculera :

- le périmètre par la formule 2 * PI * valeurCalcul[0]
- l'aire par la formule PI * r^2

Pour rappel, valeurCalcul est un tableau mais pour un disque, on n'utilise que le premier élément pour y placer le rayon.

Le calcul d'un nombre élevé au carré (x^2) se fait avec Math.Pow(x, 2)

18.1.2 EXERCICE

Ajoutez une classe 'Rectangle' dont on calculera :

- le périmètre par la formule (valeurCalcul[0] + valeurCalcul[1])* 2 (P=(a+b)*2)
- l'aire par la formule valeurCalcul[0] * valeurCalcul[1] (A = a * b)

18.2 LES INTERFACES

Les interfaces vont plus loin : c'est un contrat que le codeur s'engage à respecter.

Chaque attribut et chaque méthode devra être implémenté en utilisant les mêmes noms.

La classe aura un nom qui commencera conventionnellement par un 'i' majuscule. Le mot-clé 'class' est remplacé par 'interface'. Enfin, comme il s'agit d'un contrat public, les éléments sont toujours publics. Exemple :

```
interface IAnimal {
    string Name { get; }
    void Move();
}
```

Ici, l'interface 'Ianimal' contient un atribut (public) 'Name' et une méthode (publique) Move().

Pour utiliser l'interface, il faut écrire le code suivant :

```
class Dog : IAnimal
{
    private string m_name;
    // On implémente la propriété Name accessible en lecture.
    public string Name
    {
        get { return m_name; }
    }

    public Dog(string name) {
         m_name = name;
    }

    // On implémente la méthode Move.
    public void Move() {
        Console.WriteLine("{0} bouge.", m_name);
    }
}
```

Cet exemple est inspiré de celui donné sur https://openclassrooms.com/courses/apprenez-a-programmer-en-c-sur-net/les-interfaces-6

On crée notre propre attribut de stockage du nom en privé. Les utilisateurs de la classe 'Dog' ne connaîtront jamais cet attribut et utiliseront 'Name' à la place : cela garantit le même usage par tous !

Notez que la modification du nom du chien ne se fera qu'à l'instanciation de la classe : une fois nommé, le chien ne changera plus de nom.

L'implémentation de la méthode Move() est similaire à une classe abstraite mais n'utilise pas 'override'. L'oubli de cette implémentation donne lieu à un message clair du compilateur :

Enfin, voici comment utiliser la classe 'Dog' :

```
class Program
{
    public static void Main() {
        IAnimal chienTintin = new Dog("Milou");
         chienTintin.Move();
    }
}
```

L'avantage principal est qu'**une classe peut hériter de plusieurs interfaces** (alors qu'une classe ne peut hériter que d'une seule classe).

19 ANNEXES