

## Secteur Tertiaire Informatique Filière étude - développement

### Développer des composants d'interface

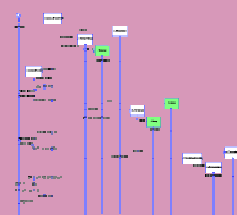
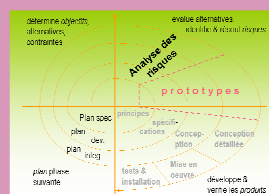
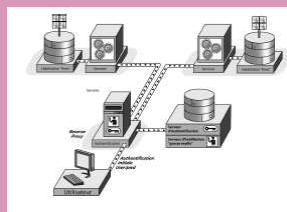
#### Initiation C#

Accueil

Apprentissage

Période en  
entreprise

Evaluation



# SOMMAIRE

I	LANGAGE C# : STRUCTURE D'UN PROGRAMME .....	6
I.1	Les objectifs.....	6
I.2	Introduction .....	6
I.3	Exercices .....	12
II	LES TYPES DE BASE ET LES ENTREES SORTIES.....	13
II.1	Les objectifs.....	13
II.2	Les types intégrés.....	13
II.3	Le type chaîne .....	17
II.4	Les opérations d'entrée sortie .....	21
II.5	Exercices .....	23
III	INSTRUCTIONS CONDITIONNELLES ET ALTERNATIVES .....	24
III.1	Les objectifs.....	24
III.2	Les structures conditionnelles .....	24
III.3	Expression de la condition .....	31
III.4	Exercices .....	32
IV	LES BOUCLES .....	33
IV.1	Les objectifs.....	33
IV.2	La nécessité des structures répétitives.....	33
IV.3	Les structures répétitives .....	34
IV.4	Les rupteurs.....	37
IV.5	Exercices .....	38
V	FONCTIONS ET PARAMETRES .....	39
V.1	Les objectifs.....	39
V.2	Définition.....	39
V.3	Utilisation des fonctions .....	40
V.4	Le corps de méthode .....	42
V.5	Le passage de paramètres .....	43
V.6	La récursivité .....	46
V.7	Les méthodes surchargées .....	47
V.8	Exercices .....	48
VI	LES TABLEAUX.....	49
VI.1	Les objectifs.....	49
VI.2	Définition.....	49
VI.3	Déclaration et création.....	49
VI.4	Utilisation .....	51
VI.5	Opération et tri sur les tableaux.....	53
VI.6	Exercices .....	55

<b>VII</b>	<b>AUTRES TYPES DE DONNEES.....</b>	<b>56</b>
VII.1	Les objectifs.....	56
VII.2	Le type enum.....	56
VII.3	Le type struct.....	57
VII.4	Dates et heures .....	57
VII.5	Exercices .....	60
<b>VIII</b>	<b>LE TRAITEMENT DES ERREURS .....</b>	<b>62</b>
VIII.1	Les objectifs.....	62
VIII.2	Vue d'ensemble.....	62
VIII.3	Les objets Exception.....	62
VIII.4	Les clauses try et catch .....	63
VIII.5	Le groupe finally .....	65
VIII.6	L'instruction throw.....	66
VIII.7	Exercices .....	66
<b>IX</b>	<b>LES FICHIERS.....</b>	<b>67</b>
IX.1	Les objectifs.....	67
IX.2	Vue d'ensemble .....	67
IX.3	Les classes de flux .....	68
IX.4	Utilisation d'un fichier texte .....	69
IX.5	Gestion des fichiers sur disque.....	75
IX.6	Exercices .....	79

# I LANGAGE C# : STRUCTURE D'UN PROGRAMME

## I.1 LES OBJECTIFS

Comprendre les généralités de l'environnement .NET et savoir créer une première application C# avec Visual Studio .NET.

Connaître les règles et les conventions de syntaxe et de présentation d'un programme écrit en langage C#.

- Namespace et directive Using
- La fonction Main, point d'entrée de l'application.
- Comment déclarer des variables et des constantes, comment les utiliser dans le corps d'un programme.
- Comment commenter vos programmes.

## I.2 INTRODUCTION

### I.2.1 Le namespace

Même si les concepts objets ne seront traités que dans une phase ultérieure de l'apprentissage C#, il est nécessaire d'introduire ici le concept de classe ainsi que le mot clé du langage C# qui permet de l'implémenter.

Ainsi, un programme C# est d'abord constitué par un **namespace** (espace de nom en français) Un namespace est une collection de types, qui se déclare par le mot clé **namespace** de la façon suivante

```
namespace TPCSharp
{
    //Déclaration des classes et fonctions de l'application
}
```

Le nom du namespace (ici **TPCSharp**) est choisi par le programmeur. La construction d'un nom de namespace obéit à la règle standard régissant tous les identificateurs du langage C# (voir ci-dessous).

Ce namespace est un bien un simple regroupement de classes avec leurs méthodes (fonctions) ayant un lien logique ou fonctionnel entre elles. Par exemple, le namespace prédéfini **system** de C# regroupera l'ensemble des classes ayant trait au système lui-même.

Selon le standard C# (ECMA-334), il est fortement déconseillé de mettre le même nom à un namespace et à une classe

Les noms de classe et de namespace commencent par une **Majuscule** : notation **PascalCase**

([http://msdn2.microsoft.com/en-us/library/893ke618\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/893ke618(VS.71).aspx))

.

### I.2.2 La directive USING

Le langage C# est un langage peu implémenté. C'est-à-dire qu'il n'existe que très peu de mots réservés, une cinquantaine, dont la liste est publiée dans la documentation du compilateur.

Par contre, il existe plusieurs bibliothèques de classes ou namespaces (qui correspondent aux packages de Java) qui offrent au programmeur un grand nombre de fonctionnalités. Ces namespaces sont utilisés grâce à l'instruction `using`.

Pour pouvoir utiliser les fonctions ou les objets des namespaces, il faut que celles-ci soient reconnues par le compilateur (qui ne les connaît pas par défaut). C'est le rôle de la directive `using`. C'est pourquoi on commence très souvent (voire toujours) un programme C# par la directive suivante :

```
using System;
```

Encore une fois, ceci n'est pas obligatoire, mais facilite grandement la tâche du développeur.

En l'absence de directive `using` et à chaque fois que l'on souhaite utiliser une classe ou une fonctionnalité d'un namespace, il faudrait préfixer cette classe ou cette fonctionnalité par le nom du namespace où elle est définie

Par exemple : `System.Console.ReadLine()` en l'absence de la directive `using System`.

En utilisant la directive `using`, le préfixe n'est plus obligatoire. Le compilateur ira chercher les fonctionnalités dans les différents namespaces des directives `using`. Dans notre exemple, en mettant `using System`, on pourra utiliser directement et simplement `Console.ReadLine()` dans l'ensemble du programme.

### I.2.3 Le programme principal

Une application C# possède toujours au moins une classe dite « classe application » et une fonction **Main**.

La classe application est simplement la dénomination donnée à la classe contenant la fonction **Main**.

La fonction **Main** est obligatoire et constitue le point d'entrée du programme principal de l'application. Elle se déclare dans la classe de l'application de la façon suivante :

```
namespace TPCsharp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Instructions
        }
    }
}
```

La fonction **Main** est le point d'entrée du programme. Elle est donc obligatoire pour toute application C# et doit être définie de façon unique (il ne peut pas y avoir deux fonctions **Main** dans un programme). Elle doit s'appeler obligatoirement **Main** et doit être suivie par des parenthèses ouvrante et fermante encadrant les arguments (comme toutes les fonctions en C#).

Par défaut, la fonction **Main** reçoit du système d'exploitation un seul argument (**args**) qui est un tableau de chaînes de caractères, chacune d'elles correspondant à un paramètre de la ligne de commande (unix ou MS-DOS).

Le corps du programme suit la fonction **Main** entre deux accolades ouvrante et fermante. Les instructions constituant le programme sont à insérer entre ces deux accolades.

Les mots clés **static** et **void** sont obligatoires pour la fonction **Main**. Ces notions seront plus faciles à comprendre lors de la partie objet du cours (concepts objets proprement dits), mais on peut simplement préciser que :

- **static** : permet au système d'exploitation de ne pas avoir besoin de créer une instance (un objet) de la classe définie ;
- **void** : ce mot clé sera également plus longuement explicité dans la suite du cours. Pour résumer, il s'agit de la déclaration du type de retour de la fonction, à savoir : aucun retour pour le type **void**.

#### I.2.4 Les variables

Les données en mémoire sont stockées dans des variables. Il peut par exemple s'agir de données saisies par l'utilisateur ou de résultats obtenus par le programme, intermédiaires ou définitifs.

Pour employer une image, une variable est une **boîte**, que le programme (l'ordinateur) va repérer par une **étiquette**. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette

Comme tous les autres identificateurs (identificateurs de classes, entre autres) en langage C#, les noms de variables peuvent être choisis librement par le programmeur (sauf parmi les *mots-clés*). La liste des mots clés se trouve dans la documentation en ligne, mais dans un problème, une variable est choisie pour jouer un rôle. Il est souhaitable que l'auteur de l'algorithme s'en souvînt en choisissant des noms évocateurs et en respectant, autant que faire se peut, l'unicité du rôle pour chaque variable.

Les caractères suivants peuvent être utilisés :

- de **A** à **Z**
- de **a** à **z** (les minuscules sont considérées comme des caractères différents des majuscules)
- de **0** à **9**
- le caractère souligné **\_** et le caractère dollar **\$** (même en initiale).

Exemple : nom des variables à retenir dans un problème de calcul de moyenne de notes

totalNotes	: total des notes d'un(e) élève
nombreNotes	: nombre des notes de cet(te) élève
moyenneNotes	: moyenne des notes de cet(te) élève

Remarquez le style de nommage des variables : (convention CamelCase) ([http://msdn2.microsoft.com/en-us/library/x2dbyw72\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/x2dbyw72(VS.71).aspx))

Une variable est typée: elle peut représenter un nombre (entier ou réel), une chaîne de caractères, etc.

Une variable peut être déclarée à tout moment à l'intérieur du corps du programme. Cependant, la déclaration et l'initialisation d'une variable doivent impérativement précéder la première utilisation de celle-ci. Il est également et généralement préférable de déclarer les variables en début de bloc (juste après l'accolade ouvrante) pour une question de lisibilité

Chaque déclaration de variable est construite sur le modèle suivant :

```
type      nomDeLaVariable ;
```

Une variable peut être initialisée lors de sa déclaration :

```
type      nomDeLaVariable = valeurInitiale;
```

Les variables ne sont visibles (reconnues par le compilateur) que dans le bloc d'instructions (défini par { et }) dans lequel elles sont définies.

Comme toutes les instructions du langage C#, chaque déclaration de variable DOIT absolument être terminée par un point-virgule ; Le point-virgule ne constitue pas un séparateur, mais plutôt un *terminateur* d'instructions.

Le chapitre suivant vous donnera plus de détails sur l'ensemble des types de données C# et .NET.

### I.2.5 Les constantes

Une constante est une variable dont la valeur ne peut changer. Son rôle est de noter des repères, des dimensions, des références invariants au cours d'un programme.

Les constantes sont définies par un identificateur, et par une valeur. Le nom représente la manière de faire référence à la valeur. La valeur représente le contenu de notre constante. Cette valeur est **invariante**.

Les constantes se déclarent comme les variables initialisées précédées du mot-clef **const**. Leur valeur ne pourra pas être modifiée pendant l'exécution du programme.

```
const Type nomDeLaConstante = valeurInitiale;
```

### I.2.6 Les instructions

Une instruction est une ligne de traitement

Le caractère `;` est un *terminateur*. TOUTES les instructions doivent se terminer par un `;`.

Les différents identificateurs sont séparés par un *séparateur*.

Les *séparateurs* peuvent être indifféremment l'espace, la *tabulation* et le *saut de ligne*.

### I.2.7 Les commentaires

Les caractères compris entre `/*` et `*/` ne seront pas interprétés par le compilateur.

Les caractères compris entre `//` et un *saut de ligne* ne seront pas interprétés par le compilateur.

Les caractères compris entre `///` et un *saut de ligne* ne seront pas interprétés par le compilateur. Le texte de commentaire peut être utilisé pour générer une documentation de façon automatique.

### I.2.8 Les conventions d'écriture

Ces conventions ne sont pas des contraintes de syntaxe du langage C#. Elles n'existent que pour en faciliter la lecture et font partie d'une sorte de norme implicite que tous les bons développeurs s'obligent à respecter.

Une seule instruction par ligne. Même si tout un programme en langage C# peut être écrit sur une seule ligne.

Les délimiteurs d'un bloc `{` et `}` doivent se trouver sur des lignes différentes et être alignés sur la première colonne de sa déclaration.

A l'intérieur d'un bloc `{ }` les instructions sont indentées (décalées) par un caractère *tabulation*.

A l'intérieur d'un bloc `{ }` la partie *déclaration des variables* et la partie *instructions* sont séparées par une ligne vide.

### I.2.9 Les entrées sorties standard

Les fonctions de lectures et d'écritures standard (clavier/écran) sont définies dans le namespace **System**.

La classe **Console** est définie dans ce namespace et dispose, notamment de trois fonctionnalités :



- `Console.ReadLine()` : qui retourne **une chaîne de caractères** (de type `string`) saisie au clavier jusqu'à la frappe de la touche "Entrée";
- `Console.Write()` : qui affiche sur l'écran a priori la chaîne de caractères (de type `string`) envoyée en paramètre ;
- `Console.WriteLine()` : qui réalise la même opération en ajoutant un caractère retour-chariot après l'affichage de la chaîne de caractères (de type `string`) envoyée en paramètre.

### I.2.10 Un exemple de code

Le code ci-dessous calcule et restitue la valeur de la circonférence d'un cercle à partir de la saisie à la console de son rayon.

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

namespace TPCSharp
{
    class Cercle
    {
static void Main() ❶
    {
        //Declaration de variables. ❷
        string saisie;    // variable recevant la saisie de
                          //l'utilisateur
        double rayon ;    // rayon dans une unité
        double perimetre; // périmètre dans cette même unité

        // Etape 1 : lecture du rayon
        Console.WriteLine("Entrez la valeur du rayon : "); ❸
        saisie = Console.ReadLine(); ❹
        // Etape 2 : calcul et affichage du périmètre
        rayon = Convert.ToDouble(saisie); ❺
        perimetre = 2 * Math.PI * rayon; ❻ //Calcul du perimetre
        Console.WriteLine("Le cercle de rayon " + rayon); ❼
        Console.WriteLine("      a pour périmetre : " + perimetre); ❽

        // Permet de conserver l'affichage de la console
        Console.ReadLine(); ❾
    }
}
}
```

Le programme décrypté, instruction après instruction...

- ❶ La fonction Main, point d'entrée du programme
- ❷ La variable **saisie** est déclarée de type **chaîne de caractères**, les variables **rayon** et **perimetre** de type **réel**
- ❸ Affichage à la console du texte « Entrez la valeur du rayon »
- ❹ Dans la variable de nom saisie, sera stockée la chaine de caracteres entrée par l'utilisateur

- ⑤ Les variables faisant partie d'une opération (+, -, \*, /) doivent être des nombres, entier ou réels. Cette instruction a pour but de **convertir le contenu de la variable saisie de type chaîne en un nombre réel de type double**.
- ⑥ La formule magique .... La valeur de PI est obtenue à partir d'une bibliothèque du framework, la classe Maths, dont le rôle est de fournir des constantes et des fonctions mathématiques et trigonométriques.
- ⑦ Affichage à la console du texte « Le cercle de rayon », suivi de la valeur de la variable **rayon** : notez l'opérateur + qui est ici l'opérateur de concaténation.
- ⑧ Affichage à la console du texte « a pour périmètre : », suivi de la valeur de la variable **périmetre** .
- ⑨ Permet de ne pas terminer le programme pour visualiser l'affichage.

### I.3 EXERCICES

Saisissez le code fourni et exécutez le.

Rajouter une fonctionnalité de calcul et d'affichage de la surface du cercle.

Afficher la fonctionnalité et la version du programme, dès son démarrage :

```
**** Périmètre et surface du Cercle (V1.0, XX/XX/XX) ****
```

On conservera cette saine habitude dans tous les développements suivants, pour éviter de mettre au point avec le code d'un autre programme, ou avec le code du mois précédent !

Aérer la présentation du résultat de l'exécution du programme, en insérant des retours à la ligne : *System.Writeline()*

## II LES TYPES DE BASE ET LES ENTREES SORTIES

### II.1 LES OBJECTIFS

L'objectif de ce chapitre est l'approfondissement de ce que nous venons de découvrir au travers de ce premier exemple.

- La déclaration et l'utilisation des variables et des constantes.
- La lecture et l'écriture des données numériques sur les flux standard et l'utilisation des manipulateurs.
- Les opérateurs de calcul.

Le système de types communs (CTS) définit deux types de variables :

**Les types valeur** : variables qui contiennent directement leurs données : ce sont les types intégrés, les structures, les variables énumération.

**Les types référence** : variables qui stockent des références à des données : leurs données sont stockées dans une zone séparée de la mémoire (classe String, classes).

### II.2 LES TYPES INTEGRES

#### II.2.1 Définition

Les types de données simples - de type scalaire (entiers) ou virgule flottante (réels) - sont identifiés par des mots clés réservés, ou en utilisant le type struct prédéfini.

Type C#	Type .NET	Taille	Val. Min.	Val. Max.	Préfixe
bool	System.Boolean	1	False	True	<b>b</b>
byte	System.Byte	1	0	255	
sbyte	System.SByte	1	-128	127	
char	System.Char	2	0	65 535	<b>c</b>
short	System.Int16	2	-32 768	32 767	<b>s</b>
ushort	System.UInt16	2	0	65 535	<b>us</b>
int	System.Int32	4	-2 147 483 648	2 147 483 647	<b>i</b>
uint	System.UInt32	4	0	4 294 967 295	<b>ui</b>
long	System.Int64	8	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	<b>l</b>
ulong	System.UInt64	8	0	18 446 744 073 709 551 615	<b>ul</b>
float	System.Single	4	$-1.5 \times 10^{-45}$	$3.4 \times 10^{38}$	<b>f</b>
double	System.Double	8	$-5.0 \times 10^{-324}$	$1.7 \times 10^{308}$	<b>lf</b>
decimal	System.Decimal	16	$-1.0 \times 10^{-28}$	env. $7.9 \times 10^{28}$	<b>d</b>

La première colonne indique le type de données que l'on peut utiliser dans un programme écrit en C#.

La seconde colonne indique le type de donnée correspondant dans l'environnement .NET. A noter qu'il est également possible, bien entendu, de les utiliser directement en C# (en omettant **system**. Si la directive **using system** est présente dans le programme).

La troisième colonne donne la taille en octets (place prise en mémoire) d'une variable déclarée avec ce type.

La quatrième colonne indique la valeur minimum admise pour une donnée de ce type.

La cinquième colonne indique la valeur maximum admise pour une donnée de ce type.

La sixième colonne indique le préfixe qu'il est préférable de mettre devant l'identificateur d'une variable de ce type. Ce préfixe n'est pas obligatoire mais renseigne précieusement tout développeur et à chaque niveau d'un programme de quel type est une variable. Dans la littérature, certains ouvrages utiliseront les préfixes, d'autres non. Il est tout de même fortement conseillé de les utiliser.

Dans le code, il est possible de déclarer la variable de 2 façons :

```
double rayon
```

ou

```
System.Double rayon
```

Qui définiront toutes deux, un réel signé sur 8 octets.

Les constantes numériques entières peuvent être écrites sous 3 formes :

- en décimal
- en hexadécimal (base 16)
- en caractère.

Ainsi, 65 (décimal), 0x41 (hexadécimal), 'A' désignent la même constante numérique entière (le code ASCII de A est 65).

### Exemples :

```
long grandEntier ;           // entier signé sur 8 octets
float leReel ;               // réel en notation virgule flottante (mantisse, exposant)
double dblPrecision ;       // idem en double précision
const int N1=65 ;
const int N2=0x41 ;
const int N3='A';
const int N4 =N1 + N2 ;
const int N5 = N1 + 10 ;
```

## II.2.2 Attribution de valeurs aux variables

Une valeur peut être attribuée à une variable lors de sa déclaration.

```
double pi = 3.14159 ;
```

ce qui est équivalent à

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

On parle d'instruction d'**affectation** ou d'**assignation**, et on lit **PI prend pour valeur 3.14159**.

Pour attribuer une valeur à une variable de type caractère, on écrira :

```
char lettre;  
lettre = 'A' ;
```

Types d'affectation :

```
int resultat, Y = 3 ;           //seule la variable Y est initialisée à 3  
resultat = 5 ;                 // résultat prend la valeur 5  
resultat = Y ;                 // résultat prend la valeur 3  
                                // Y reste à 3  
resultat = Y + 2 ;             // résultat prend la valeur 5  
resultat = resultat + 2 ;      // résultat prend la valeur 7
```

On pourra utiliser la syntaxe abrégée, quelque soit l'opérateur arithmétique :

```
Y = Y + 2 ;                     // Y prend la valeur 5  
Y += 2 ;                       // syntaxe abrégée
```

Attention :

```
double d = 2.5 ;                //syntaxe OK  
float f1 = 2.5 ;                //pas bon : 2.5 est au format double  
float f2 = 2.5f ;  
float f3 = (float)12.5 ; } // syntaxe OK  
float f4 = 2.5e10f;
```

## II.2.3 Les opérateurs

### Arithmétiques

+	Addition	<b>a + b</b>
-	Soustraction	<b>a - b</b>
-	Changement de signe	<b>-a</b>
*	Multiplication	<b>a * b</b>
/	Division	<b>a / b</b>
%	Reste de la division entière	<b>a % b</b>

---

<sup>1</sup> Voir le paragraphe relatif au casting

Lorsqu'une expression contient plusieurs opérateurs, l'ordre dans lequel sont effectués les calculs dépend de l'ordre de priorité des opérateurs.

L'expression  $x + y * z$  est évaluée sous la forme  $x + (y * z)$  car l'opérateur de multiplication a une priorité supérieure à celle de l'opérateur d'addition.

**Règle** : Les opérateurs  $-$  et  $+$  ont une priorité plus basse que celle des opérateurs  $*$ ,  $/$  et  $\%$ . On peut contrôler la priorité à l'aide de parenthèses.

### D'affectation

=	Affectation	<code>a = 5;</code>
+=	Incrémentation	<code>a += b;</code>
	Les deux exemples sont équivalents	<code>a = a + b;</code>
--	Décrémentation	<code>a -= b;</code>
	Les deux exemples sont équivalents	<code>a = a - b;</code>

### D'incrémentation

++	Pré-incrémentation (+1)	<code>++a</code>
++	Post-incrémentation (+1)	<code>a++</code>
--	Pré-décrémentation (-1)	<code>--a</code>
--	Post-décrémentation (-1)	<code>a--</code>

### Exemples :

```
int    a, b;

a = 5;
b = a++;
// a = 6 et b = 5;

a = 5;
b = ++a;
// a = 6 et b = 6;
```

`a++` est incrémenté après affectation, alors que `++a` est incrémenté avant affectation.

## II.2.4 Conversion implicite et casting

Quand deux opérandes de part et d'autre d'un opérateur binaire (qui a deux opérandes) sont de types différents, une conversion implicite est effectuée vers le type "le plus fort" en suivant la relation d'ordre suivante :

**bool < byte < char < short < int < long < float < double**

**Exemple** : La conversion d'un type de données **int** en un type de données **long** est implicite : cette conversion réussit toujours et n'entraîne jamais de perte d'informations.

```
int    a = 78;
long   b = a;
```

Par contre, le compilateur ne voudra pas exécuter l'inverse, le code suivant

```
long a = 78;
int b = a;
```

Il est impossible de convertir implicitement un type **long** en un type **int**, car le risque de perdre de l'information existe. Une conversion explicite doit être mise en place.

```
long a = 78;
int b = (int) a;
```

Un **casting** se fait en mentionnant le type entre parenthèse avant l'expression à convertir.

Par ailleurs, un opérateur binaire dont les deux opérandes sont de même type opère dans ce type. Ce qui semble donner parfois des résultats curieux. Pour avoir le résultat attendu, il faut forcer la conversion par un **casting** (changement de type) afin de préciser dans quel référentiel on opère.

**Exemples :**

```
int n1 = 5;
int n2 = 2;
double x = n1 / n2;
```

Aussi bizarre que cela paraisse, la valeur de x est de 2.0. En effet, 5 et 2 sont des entiers. Le résultat de la division entière de ces deux nombres est 2 (et il reste 1) et non pas 2.5. Pour obtenir cette dernière valeur, il faut forcer la conversion de l'un des opérandes en un nombre virgule flottante :

```
double x = (double)n1 / n2;
```

Le fait de faire le **casting** (**double**) devant 5 force la conversion de 5 (type **int**) en 5.0 (type **double**). L'opérateur / va donc devoir opérer sur un **double** et un **int**. Il y a alors une conversion implicite de 2 (type **int**) en 2.0 (type **double**). L'opérateur peut maintenant opérer sur deux **double** et le résultat (2.5) est un **double**.

## II.3 LE TYPE CHAÎNE

### II.3.1 Définition

En C#, une chaîne de caractères est un objet de la classe **String**, de l'espace de nom **System**.

Ainsi une chaîne peut être déclarée :

```
string strChaine ;
```

ou

```
String strChaine;
```

La variable `strChaine` ne *contient* pas la chaîne de caractères : elle est la référence d'une chaîne de caractères. Telle qu'elle est déclarée ci-dessus, la valeur de `strChaine` est **null**.

On pourra l'initialiser :

```
strChaine = "Bonjour" ; // si elle est préalablement déclarée
```

ou

```
string strChaine = "Bonjour" ;
```

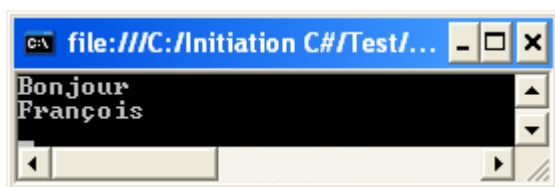
Entre les doubles-quotes, le caractère \ est utilisé comme modificateur et interprète le caractère immédiatement après pour coder des caractères non-affichables comme le saut de ligne ou une tabulation :

\n	Saut de ligne
\r	Retour de chariot
\t	Tabulation
\b	Retour arrière
\v	Tabulation verticale
\f	Saut de page
\\	Le caractère \ lui-même
\'	Le caractère ' (quote) lui-même
\"	Le caractère " (double quote) lui-même
\xHH	Caractère de code HH en hexadécimal
\NNN	Caractère de code ASCII NNN en décimal

**Exemple :** Le code suivant

```
string strNom = "François";  
Console.WriteLine("Bonjour \n" + strNom);
```

produira l'affichage suivant :



La **concaténation** de chaînes de caractères s'obtient grâce à l'opérateur +. Le résultat de cette opération est une nouvelle chaîne de caractères qui peut être affectée à une variable de type **string** :

```
string strMsg = "Bonjour " + strPrenom;
```

On accède à un caractère de rang fixé d'une chaîne par l'opérateur [ ] (la chaîne est lue comme un tableau de char, le premier caractère est indicé par 0) :

```
char lettre = strNom [3] ; // lettre contient le caractère n
```

par contre

```
strNom [3] = "B"; // provoque une erreur de compilation
```

Le texte d'une chaîne ne peut pas être modifié après sa création (immuable).

### II.3.2 Manipulation de chaîne de caractères

La classe **String** possède intrinsèquement les méthodes (fonctions membres) nécessaires à la manipulation des chaînes de caractères. Comme beaucoup de méthodes définies dans une classe d'objet, elles s'utilisent conjointement à une variable de type **String** avec le caractère . (point).

Toutes les méthodes de la classe **String**, ne seront pas traitées ici. Pour plus de précisions la documentation en ligne.



## Longueur d'une chaîne de caractères

La longueur d'une chaîne de caractères est calculée par la propriété **Length** d'un objet de la classe **String**.

```
string strTitre = "Support de cours C#";  
int iLen = strTitre.Length;    // iLen vaudra donc 19
```

## Position d'une chaîne dans une autre

La méthode d'instance **IndexOf** permet de calculer la position d'une séquence de caractères dans une chaîne. La valeur 0 de cette position correspond au premier caractère de la chaîne. La valeur -1 est retournée si la séquence de caractères n'a pas été trouvée dans la chaîne. Il existe plusieurs versions de cette fonction dont les plus utilisées sont les suivantes :

```
int pos1 = str1.IndexOf(str2);  
int pos2 = str1.IndexOf(str2, iPos);
```

Dans le premier cas **pos1** contiendra la position de **str2** dans **str1**. Dans le second cas, la position de **str2** sera cherchée à partir du **iPos**<sup>ième</sup> caractère de **str1**. Ceci permet de repérer plusieurs occurrences de **str2** dans **str1** en utilisant une boucle de parcours.

Exemples:

```
string str1 = "Pascal disait: je pense, donc je suis";  
int pos1 = str1.IndexOf( "je" );           // pos1 = 15  
int pos2 = str1.IndexOf( "je", pos1 + 1 ); // pos1 = 30
```

## Extraction d'une sous-chaîne

La méthode d'instance **Substring** permet d'extraire une sous-chaîne d'une chaîne de caractères :

```
string strPrenom = strNom.Substring(0, 6);  
// Extrait les 6 premiers caractères de la chaîne strNom
```

## Conversion Majuscules / Minuscules

Les méthodes d'instance **ToUpper** et **ToLower** permettent de retourner une chaîne dont tous les caractères sont convertis respectivement en majuscules et en minuscules.

```
string strSalut = "Bonjour Tout Le Monde";  
Console.WriteLine(strSalut.ToUpper()); // BONJOUR TOUT LE MONDE  
Console.WriteLine(strSalut.ToLower());  // bonjour tout le monde
```

## Suppression d'espaces

La méthode d'instance **Trim** supprime tous les espaces de début et fin de chaîne.

```
string strSalut = "    Bonjour    ";  
Console.WriteLine(strSalut.Trim()); // Bonjour
```

## Comparaison de chaînes de caractères

La méthode **Equals** permet de comparer l'égalité de 2 chaînes : elle renvoie un booléen positionné à **true / false** si les 2 chaînes sont identiques/différentes ; elle peut être employée de 2 façons :

### Exemple :

```
string str1 = "abc";  
string str2 = "abc";
```

- En tant que méthode d'instance

```
bool bIden = str1.Equals(str2);           // bIden = true
```

- En tant que méthode de classe

```
bool bIden = String.Equals(str1,str2);    // bIden = true
```

Notez la différence entre une méthode de classe et une méthode d'instance : nous reviendrons sur le sujet dans un prochain chapitre.

La méthode de classe **Compare** permet de comparer les chaînes en fonction de leur ordre de tri : elle renvoie **0** si les chaînes sont identiques, **-1** si la 1ere chaîne précède la 2eme, **1** si la deuxième précède la 1ere.

```
string str1 = "abc";  
string str2 = "def";  
int iIden = String.Compare(str1,str2);    // iIden = -1
```

Une autre version de la méthode permet de positionner un booléen en 3eme paramètre qui spécifiera si la casse doit être ignorée ou non.

```
string str1 = "abc";  
string str2 = "Abc";  
int iIden = String.Compare(str1,str2, true); // iIden = 0
```

La casse est ignorée.

## Remplacement de sous chaînes dans une chaîne

La méthode **Replace** permet de remplacer toutes les occurrences d'une sous chaîne de la chaîne de base par une autre.

```
string str1 = "Bonjour";  
string str2 = str1.Replace("on","ON ") ;  
// str2 contient "BON jour", str1 est inchangé
```

## Eclatement d'une chaîne en tableau de sous chaînes

La méthode **Split** permet de retourner un tableau de mots constituant la chaîne de base en spécifiant le(ou les) séparateur(s).

```
string strSalut = "Bonjour Tout Le Monde";  
string[] tsc = strSalut.Split( ' ' ) ; // un seul séparateur  
// tsc[1] contient "Bonjour", tsc[2] contient "Tout",  
// tsc[3] contient "Le", tsc[4] contient "Monde",
```

## II.4 LES OPERATIONS D'ENTREE SORTIE

L'affichage des données numériques se fait à travers le flux de sortie **Console**. à l'aide des méthodes **Write()** ou **WriteLine()**.

Exemples :

```
Console.WriteLine("Bon")
```

L'argument est une chaîne de caractères affichée avec un retour et un saut de ligne.

```
int i=5 ;
Console.WriteLine(i);
```

L'argument est un entier : une des formes de **WriteLine** ou **Write** accepte un entier en argument ; l'entier est converti en une chaîne de caractères pour l'affichage ; d'autres formes acceptent les autres types (bool, char, double ...)

```
int i=5 , j=4 ;
Console.WriteLine(" i = " + i + " , j= " + j);    (équivalent à)
Console.WriteLine(" i = {0}, j= {1} ", i, j);
```

Les valeurs de i et j sont respectivement positionnées à la place des paramètres 0 et 1

### Mise en format de chaînes de caractères

Chaque spécificateur a la forme { N [ , M ] : chaîne de format } où :

- N désigne le numéro de l'argument (0,1)
- M, facultatif désigne le nombre de positions d'affichage , valeur positive pour un alignement à droite, ou négative pour un alignement à gauche
- La chaîne de format, facultative peut représenter un format standard

```
double unDouble = 123456789;
Console.WriteLine(" le double = {0: e} ", unDouble);
// rend le double = 1.2345678e+008
Console.WriteLine(" le double = {0: f} ", unDouble);
// rend le double = 123456789.00
Console.WriteLine(" le double = {0:n}", unDouble);
// rend le double = 123 456 789.00
```

ou personnalisé, où les caractères suivants ont une signification particulière :

- 0 représente un chiffre
- # représente un chiffre ou rien
- % le nombre sera multiplié par 100
- . représente le séparateur de milliers
- E indique une représentation scientifique

```
double unDouble = 34.5;
Console.WriteLine(" le double = {0:##.##0} ", unDouble);
// le double = 34.50

double unDouble = 34567.89;
Console.WriteLine(" le double = {0:##,###.##} ", unDouble);
// le double = 34 567,89

int unEntier = 34;
Console.WriteLine(" l'entier = {0,5:000} ", unEntier);
// l'entier = 034,00 La valeur 34 est précédée de 2 espaces.

int unEntier = 34;
Console.WriteLine(" l'entier = {0:000.00} ", unEntier);
// l'entier = 034,00
```

(Consultez l'aide en ligne pour plus de détails)

**La lecture des données numériques** se fait à travers le flux d'entrée **Console**. à l'aide de la méthode **ReadLine**. Cette méthode pose un problème principal : Elle ne permet de ne lire qu'une chaîne de caractères **string** Il faut donc convertir explicitement cette chaîne de caractères pour obtenir l'objet numérique souhaité (**Double**, **Float**, **Long**, **Int32**, etc...):

**Exemples :**

```
// Pour lire l'entier k déclaré de type int
string strLine;
strLine = Console.ReadLine();
int k = Convert.ToInt32( strLine );
//ou encore
k = Int32.Parse( strLine );
```

La classe **System.Convert** fournit un jeu complet de méthodes pour les conversions prises en charge. On peut, par exemple convertir des types **string** en types numériques, des types **DateTime** en types **string** et des types **string** en types **boolean**.

D'autre part, tous les types numériques disposent d'une méthode **Parse** statique pouvant être utilisée pour convertir une représentation sous forme de chaîne d'un type numérique en un type numérique réel. Les deux méthodes sont équivalentes.

**Attention :** Le programme se plantera si l'utilisateur introduit autre chose qu'un entier correct. Le problème sera résolu dans un chapitre ultérieur.

```
// Pour lire le nombre dSalaire déclaré de type double
string strLine;
strLine = Console.ReadLine();
double dSalaire = Convert.ToDouble( strLine );
// ou encore
dSalaire = Double.Parse( strLine );
```

Comme dans l'exemple précédent, le programme *se plantera* si l'utilisateur introduit autre chose qu'un réel. Le séparateur de décimales saisi doit être conforme aux caractéristiques régionales.

## II.5 EXERCICES

### **Exercice 1 : Calcul de la moyenne pondérée des notes d'un étudiant dans une matière**

Il s'agit de calculer une moyenne sur la base de trois notes sachant que :

- une note de devoir surveillé a un coefficient de 3
- une note d'interrogation écrite a un coefficient de 2
- une note de travaux pratique a un coefficient de 1

### **Exercice 2 : Conversion d'une durée exprimée en secondes en heures, minutes et secondes**

Il s'agit pour un nombre de secondes entré au clavier d'en déduire, son expression en nombre d'heures de minutes et de secondes.

### **Exercice 3 :**

Ecrire un programme qui saisit un code Unicode en décimal et affiche le caractère correspondant.

Exemple la saisie de l'entier 65 donne le caractère « A ».

### **Exercice 4 :**

Afficher les caractères de contrôle de tabulation, beep, retour en début de ligne...qui seront utiles pour réaliser une présentation simple en mode texte.

## III INSTRUCTIONS CONDITIONNELLES ET ALTERNATIVES

### III.1 LES OBJECTIFS

L'objectif de ce chapitre est d'apprendre à utiliser les instructions **if**, **switch**.

### III.2 LES STRUCTURES CONDITIONNELLES

#### III.2.1 L'action conditionnée

L'action conditionnée est une instruction élémentaire ou une suite d'instructions **exécutées** en séquence **si l'état du système l'autorise**. Le(s) critère(s) à respecter pour exécuter l'action s'exprime(nt) à l'aide d'une condition (ou **prédicat**) évaluable au moment précis où l'action doit, le cas échéant, intervenir.

Lors de l'exécution du programme, le processeur est donc amené à évaluer la condition. La condition évaluée constitue alors un énoncé (ou **proposition**) vrai ou faux.

**Schéma :**

```
si prédicat alors
    Instruction 1
    Instruction 2
    ...
    Instruction N
Fin si
```

**Exemples :**

<b>Si</b> Température > 38 <b>alors</b> <b>Écrire</b> "Le patient a de la fièvre" <b>Fin si</b>	<b>Si</b> Température > 41 <b>et</b> Tension > 25 <b>alors</b> <b>Écrire</b> "Le patient va perdre patience" <b>Fin si</b>
<b>Si non</b> Patient <b>alors</b> <b>Écrire</b> "Éconduire l'olibrius" <b>Fin si</b>	<b>Si</b> Température > 42 <b>ou</b> (Tension < 25 <b>et</b> Pouls > 180) <b>alors</b> <b>Écrire</b> "Prévenir la famille" <b>Fin si</b>
<b>Si</b> Température > 40 <b>ou</b> Tension ≥ 25 <b>alors</b> <b>Écrire</b> "Hospitaliser le patient" <b>Fin si</b>	<b>Si</b> Patient <b>et</b> Pouls = 0 <b>alors</b> Patient ← non Patient <b>Fin si</b>

**Syntaxe C# :**

```
if (prédicat)
{
    Instruction 1
    Instruction 2
    ...
    Instruction N
}
```

**Exemple 1:** réaliser le programme qui après lecture d'un entier au clavier, affiche un message si l'entier est pair.

```
class Program
{
    static void Main(string[] args)
    {
        string strLigne;
        int entier;
        int reste;
        // saisie utilisateur
        Console.WriteLine("Saisissez un entier :");
        strLigne = Console.ReadLine();
        // conversion en entier
        entier = Int32.Parse(strLigne);
        // Reste de la division par 2
        reste = entier % 2;
        if (reste == 0)
        {
            Console.WriteLine("Entier pair !");
        }
        Console.ReadLine();
    }
}
```

Notez l'opérateur relationnel == qui teste l'égalité. Tous les opérateurs relationnels seront vus dans le prochain paragraphe.

Remarquez l'indentation (retrait de paragraphe) qui permet de voir facilement la portée de la condition.

A noter : si une seule instruction est à exécuter dans le IF, les accolades ne sont pas obligatoires ...

```
if (reste == 0)
    Console.WriteLine("Entier pair !");
```

Mais le code est tellement plus lisible et évite des problèmes futurs de rajout d'instructions!

Les conditions peuvent être imbriquées :

**Exemple 2:** Modifier le programme précédent qui après lecture d'un entier au clavier, affiche un message si l'entier est pair ; indiquez également si l'entier est multiple de 4.

```
...
// Reste de la division par 2
if (entier % 2 == 0)
{
    Console.WriteLine("Entier pair !");
    // Reste de la division par 4
    if (entier % 4 == 0)
    {
        Console.WriteLine("Entier multiple de 4 !");
    }
}
```

Pourquoi les conditions sont-elles imbriquées ? Tout simplement parce qu'un nombre impair ne peut pas être un multiple de 4... le traitement ne concerne donc que les nombres pairs.

Notez la notation différente du prédicat dans cet exemple ... Les 2 notations sont équivalentes, l'important étant que le prédicat puisse être évalué.

**Exemple 3:** Et de la même façon, si on avait voulu également savoir si l'entier était également multiple de 6, on aurait pu écrire :

```
...
// Reste de la division par 2
if (entier % 2 == 0)
{
    Console.WriteLine ("Entier pair !");
    // Reste de la division par 4
    if (entier % 4 == 0)
    {
        Console.WriteLine("Entier multiple de 4 !");
    }

    // Reste de la division par 6
    if (entier % 6 == 0)
    {
        Console.WriteLine("Entier multiple de 6 !");
    }
}
```

Un entier pair peut être à la fois multiple de 4 et de 6 (12), mais peut être multiple de 4 sans être multiple de 6 (8) ou l'inverse (18).

### III.2.2 La structure alternative

La structure alternative traduit la nécessité d'exécuter, à un instant donné, une action, élémentaire ou composée, ou une autre action, elle-même élémentaire ou composée, exclusivement l'une de l'autre.

**Schéma :**

```
si prédicat alors
    Instructions_si_vrai;
sinon
    Instructions_si_faux;
Fin si
```

**Exemples :**

<b>Si</b> Nombre = 0 <b>alors</b> <b>Écrire</b> "Impossible de diviser par 0" <b>sinon</b> Moyenne = Total / Nombre <b>Écrire</b> Moyenne <b>Fin si</b>	<b>Si</b> C= "A" <b>ou</b> C= "E" <b>ou</b> C= "I" <b>ou</b> C= "O" <b>ou</b> C= "U" <b>alors</b> <b>Écrire</b> C, " est une voyelle" <b>sinon</b> <b>Écrire</b> C, " est une consonne" <b>Fin si</b>
--	---



## Syntaxe C# :

```
if (prédicat)
{
    Instructions_si_vrai;
}
else
{
    Instructions_si_faux;
}
```

**Exemple :** Modifier le programme précédent qui après lecture d'un entier au clavier, affiche un message si l'entier est pair, pour qu'il affiche un message si l'entier est impair.

```
...
// Reste de la division par 2
if (entier % 2 == 0)
{
    Console.WriteLine ("Entier pair !");
    // Reste de la division par 4
    if (entier % 4 == 0)
    {
        Console.WriteLine("Entier multiple de 4 !");
    }

    // Reste de la division par 6
    if (entier % 6 == 0)
    {
        Console.WriteLine("Entier multiple de 6 !");
    }
}
else
{
    Console.WriteLine("Entier impair !");
}
```

### III.2.3 Le choix multiple

La sélection (ou **choix multiple**) permet de présenter une solution claire à des problèmes où un nombre important de cas, mutuellement exclusifs, sont à envisager en fonction des valeurs prises par un nombre réduit de variables (généralement une seule). Puisque chaque action est exclusive des autres, la structure sélective correspond à une **imbrication d'alternatives**.

Cette structure vise en fait à réduire la complexité engendrée par la présence d'un nombre élevé de cas à envisager. Elle privilégie ainsi l'essentiel (la recherche des différents cas et des actions à leur associer) sans négliger l'accessoire (la lisibilité de la solution).

**Exemple 1:** En fonction du caractère saisi à l'écran, affichez le statut de la personne en clair.

```
namespace Test
{
    class Statut
    {
        static void Main(string[] args)
        {

            string strStatut;
            // saisie utilisateur
            Console.WriteLine("Saisissez le statut :");
            strStatut = Console.ReadLine();

            if (strStatut == "C")
            {
                Console.WriteLine("Célibataire !");
            }
            else
            {
                if (strStatut == "M")
                {
                    Console.WriteLine("Marié !");
                }
                else
                {
                    if (strStatut == "V")
                    {
                        Console.WriteLine("Veuf !");
                    }
                    else
                    {
                        if (strStatut == "D")
                        {
                            Console.WriteLine("Divorcé !");
                        }
                    }
                }
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

Traité avec une imbrication de if, la solution est peu lisible, mais peut être améliorée en utilisant le **If étendu** :

## Exemple 2:

```
namespace Test
{
    class Statut
    {
        static void Main(string[] args)
        {

            string strStatut;
            // saisie utilisateur
            Console.WriteLine("Saisissez le statut :");
            strStatut = Console.ReadLine();

            if (strStatut == "C")
            {
                Console.WriteLine("Célibataire !");
            }
            else if (strStatut == "M")
            {
                Console.WriteLine("Marié !");
            }
            else if (strStatut == "V")
            {
                Console.WriteLine("Veuf !");
            }
            else if (strStatut == "D")
            {
                Console.WriteLine("Divorcé !");
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

L'utilisation de l'instruction **switch** va encore apporter une autre lisibilité :

## Syntaxe C#

```
switch (expression)
{
    case valeur1:
        Instructions_Valeur1;
        break;
    case valeur2:
        Instructions_Valeur2;
        break;
    case valeur3:
        Instructions_Valeur3;
        break;
    default:
        Instructions_Default;
}
```

Si **expression** est égale à **valeur1** on exécute les instructions **Instructions\_Valeur1**.

Si **expression** est égale à **valeur2** on exécute les instructions **Instructions\_Valeur2**.

Si **expression** est égale à **valeur3** on exécute les instructions **Instructions\_Valeur3**.

Si **expression** n'est égale à aucune des valeurs énumérées, on exécute **Instructions\_Default**.

Pour éviter d'exécuter tous les pavés d'instructions en séquence à partir du moment où l'égalité a été trouvée, il est obligatoire de mettre une instruction **break** (rupteur) à la fin de chaque pavé : cette instruction provoque un débranchement à l'instruction suivant l'accolade fermante.

### Exemple 3:

```
namespace Test
{
    class Statut
    {
        static void Main(string[] args)
        {

            string strStatut;
            // saisie utilisateur
            Console.WriteLine("Saisissez le statut :");
            strStatut = Console.ReadLine();
            switch (strStatut)
            {
                case "C":
                    Console.WriteLine("Célibataire !");
                    break;
                case "M":
                    Console.WriteLine("Marié !");
                    break;
                case "V":
                    Console.WriteLine("Veuf !");
                    break;
                case "D":
                    Console.WriteLine("Divorcé !");
                    break;
                default :
                    Console.WriteLine("Saisie incorrecte !");
                    break;
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

### III.3 EXPRESSION DE LA CONDITION

#### Opérateurs relationnels

Si les opérandes peuvent être des expressions arithmétiques quelconques, le résultat de ces opérateurs est de type `bool`.

<code>==</code>	Egalité	<code>a == b</code>
<code>!=</code>	Différence	<code>a != b</code>
<code>&lt;</code>	Inférieur	<code>a &lt; b</code>
<code>&gt;</code>	Supérieur	<code>a &gt; b</code>
<code>&lt;=</code>	Inférieur ou égal	<code>a &lt;= b</code>
<code>&gt;=</code>	Supérieur ou égal	<code>a &gt;= b</code>

Ce résultat peut, bien sur, être utilisé comme opérande des opérateurs booléens

#### Opérateurs booléens

Les opérateurs booléens ne reçoivent que des opérandes de type `bool`. Ces opérandes peuvent avoir la valeur vraie (`true`) ou faux (`false`).

<code>&amp;</code>	ET logique Le résultat donne vrai (valeur <code>true</code> ) si les deux opérandes ont pour valeur vrai Le résultat donne faux (valeur <code>false</code> ) si l'un des deux opérandes a pour valeur faux.	<code>a &gt; b &amp; a &lt; c</code>
<code>&amp;&amp;</code>	ET logique Cet opérateur fonctionne comme le précédent, à la différence que le deuxième opérande (à droite) n'est pas évalué (calculé) si le premier a pour valeur faux. Car quelle que soit la valeur du deuxième opérande, le résultat est forcément faux.	<code>a &gt; b &amp;&amp; a &lt; c</code>
<code> </code>	OU logique Le résultat donne vrai (valeur <code>true</code> ) si l'un des deux opérandes a pour valeur vrai. Le résultat donne faux (valeur <code>false</code> ) si les deux opérandes ont pour valeur faux.	<code>a == b   a == c</code>
<code>  </code>	OU logique Cet opérateur fonctionne comme le précédent, à la différence que le deuxième opérande (à droite) n'est pas évalué (calculé) si le premier a pour valeur vrai. Car quelle que soit la valeur du deuxième opérande, le résultat est forcément vrai.	<code>a == b    a == c</code>
<code>!</code>	NON logique Le résultat est faux si l'expression est vraie et inversement	<code>!(a &lt;= b)</code>

Il existe, en C#, un opérateur à trois opérandes :

? : Structure alternative.

```
n = k>3 ? 5 : 6;
```

L'exemple sera interprété :

Si  $k$  est supérieur à 3,  $n$  vaudra 5 sinon 6.

### Example :

```
if ( i ==1 && j !=2 )
{
    ...
}
```

Signifiera si  $i$  vaut 1 et  $j$  est différent de 2, alors ...

### III.4 EXERCICES

## Exercice 1 : Réalisation d'une calculette

Faire la saisie de 2 nombres entiers, puis la saisie d'un opérateur '+', '-', '\*' ou '/'. Si l'utilisateur entre un opérateur erroné, le programme affichera un message d'erreur. Dans le cas contraire, le programme effectuera l'opération demandée (en prévoyant le cas d'erreur "division par 0"), puis affichera le résultat.

## Exercice 2 : Calcul d'une remise

A partir de la saisie du prix unitaire d'un produit PU et de la quantité commandée QTECOM, afficher le prix à payer PAP, en détaillant le port PORT et la remise REM, sachant que :

- le port est gratuit si le prix des produits TOT est supérieur à 500 €. Dans le cas contraire, le port est de 2% de TOT.
- la valeur minimale du port à payer est de 6 €.
- la remise est de 5% si TOT est compris entre 100 et 200 € et de 10% au delà.

## IV LES BOUCLES

### IV.1 LES OBJECTIFS

L'objectif de ce chapitre est d'apprendre à coder les instructions **répétitives**.

### IV.2 LA NECESSITE DES STRUCTURES REPETITIVES

L'itération (ou structure répétitive ou **boucle**) permet d'obtenir une action composée par la répétition d'une action élémentaire ou composée, répétition qui continue tant qu'une condition n'est pas remplie, ou cesse lorsqu'une condition donnée est remplie

**Exemple** : La table de multiplication par 5

Avec les instructions définies à ce stade, la seule possibilité d'écrire la table en totalité est donnée par le programme ci-dessous.

```
namespace Repetitives
{
    class Test1
    {
        static void Main()
        {
            Console.WriteLine("Table de multiplication par 5");
            Console.WriteLine("=====");
            Console.WriteLine("{0} * 5 = {1}", 1, 1 * 5);
            Console.WriteLine("{0} * 5 = {1}", 2, 2 * 5);
            Console.WriteLine("{0} * 5 = {1}", 3, 3 * 5);
            Console.WriteLine("{0} * 5 = {1}", 4, 4 * 5);
            Console.WriteLine("{0} * 5 = {1}", 5, 5 * 5);
            Console.WriteLine("{0} * 5 = {1}", 6, 6 * 5);
            Console.WriteLine("{0} * 5 = {1}", 7, 7 * 5);
            Console.WriteLine("{0} * 5 = {1}", 8, 8 * 5);
            Console.WriteLine("{0} * 5 = {1}", 9, 9 * 5);
            Console.WriteLine("{0} * 5 = {1}", 10, 10 * 5);

            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

A la lecture de ce programme, on s'aperçoit vite que la même action élémentaire (moyennant paramétrage) est répétée un certain nombre de fois.

En généralisant, on peut écrire que l'instruction suivante

```
Console.WriteLine("{0} * 5 = {1}", i, i * 5);
```

Se répète pour une valeur de i, variant de 1 à 10, la condition d'arrêt pouvant s'énoncer

Pour **i** variant de **1** à **10**  
Ou  
Tant que **i** **<= 10**  
Ou  
Jusqu'à **i** **> 10**

On distingue généralement plusieurs types de structures répétitives.

## IV.3 LES STRUCTURES REPETITIVES

### IV.3.1 Structure répétitive for

**Syntaxe C# :**

```
for ( expression_a; expression_b; expression_c )  
{  
    // Instructions;  
}
```

**expression\_a** représente l'initialisation des itérateurs ;

**expression\_b** représente la condition d'itération ;

**expression\_c** représente l'actualisation des itérateurs ;

**Dans l'exemple de la table de multiplication :**

```
{  
    class Test1  
    {  
        static void Main()  
        {  
            Console.WriteLine("Table de multiplication par 5");  
            Console.WriteLine("=====");  
            for (int i =1; i <= 10 ; i++)  
            {  
                Console.WriteLine("{0} * 5 = {1}", i, i * 5);  
            }  
            // Affichage persistant  
            Console.ReadLine();  
        }  
    }  
}
```

**Déroulement de l'exécution :**

- Lors de la première exécution de l'instruction **for**, **i** est initialisée à **1** ;  
A chaque exécution, la condition d'itération (**i** <= 10) est évaluée ; si **i** > 10, la boucle s'arrête, et l'instruction suivant l'accolade fermante est exécutée.
- Lorsque la condition d'itération est vraie, les instructions entre accolades sont exécutées.
- Sur l'accolade fermante, **i** est incrémenté de **1**  
Retour sur l'instruction **for**



### IV.3.2 La structure While

#### Syntaxe C# :

```
while ( condition )
{
    // Instructions;
}
```

**condition** est une expression booléenne (type `bool`). Les **instructions** sont exécutées plusieurs fois tant que le résultat de l'expression **condition** est vraie (valeur `true`).

La condition doit pouvoir être évaluée à la première exécution de l'instruction **while**, ce qui nécessite toujours l'initialisation de la (des) variable(s) intervenant dans la condition.

Si à la première exécution du **while**, le résultat de l'expression **condition** est faux (valeur `false`), les **instructions** ne sont jamais exécutées.

Les instructions seront donc exécutées de 0 à n fois.

#### Exemple : La table de multiplication par 5

```
namespace Repetitives
{
    class Test2
    {
        static void Main()
        {
            int i;
            Console.WriteLine("Table de multiplication par 5");
            Console.WriteLine("=====");
            i = 1;
            while (i <= 10)
            {
                Console.WriteLine("{0} * 5 = {1}", i, i * 5);
                i++;
            }
            // Affichage persistant
            Console.ReadLine();
        }
    }
}
```

#### Déroulement de l'exécution :

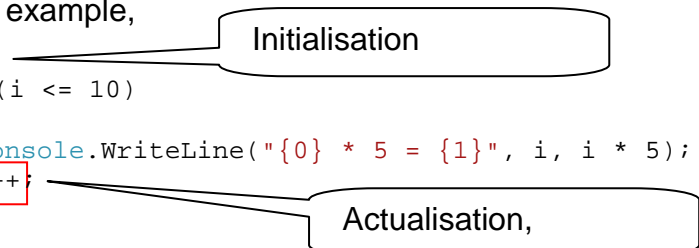
- A chaque exécution de l'instruction **while**, la condition d'itération (`i <= 10`) est évaluée ; si `i > 10`, la boucle s'arrête, et l'instruction suivant l'accolade fermante est exécutée.  
Lorsque la condition d'itération est vraie, les instructions entre accolades sont exécutées.
- Sur l'accolade fermante, retour sur l'instruction **while**

L'instruction **while** nécessite une attention soutenue : Sa syntaxe complète est :

```
initialisation
while ( condition )
{
    // Instructions;
    // actualisation
}
```

Dans notre exemple,

```
i = 1;
while (i <= 10)
{
    Console.WriteLine("{0} * 5 = {1}", i, i * 5);
    i++;
}
```



**Surtout, ne pas oublier la partie Actualisation ....**

L'instruction **while** par rapport à l'instruction **for** présente l'intérêt de pouvoir évaluer une condition d'itération complexe, par exemple :

```
while ((i <= 10) && (j != 2)){...}
```

ou

```
while (!trouve) {...} // bool trouve
```

### a) La structure do

**Syntaxe C# :**

```
do
{
    // Instructions;
} while ( condition );
```

**condition** est une expression booléenne (type `bool`). Les **Instructions** sont exécutées plusieurs fois tant que le résultat de l'expression **condition** est vraie (valeur `true`).

L'instruction **do** est toujours accompagnée d'une instruction **while**.

Elle est similaire à l'instruction **while**, sauf que l'évaluation de la condition d'itération s'effectue en fin de boucle, et non pas au début, ce qui signifie que, contrairement à l'instruction **while** qui est exécutée de **0** à **n** fois, une instruction **do** est exécutée **au moins une fois**.

```

namespace Repetitives
{
    class Test3
    {
        static void Main()
        {
            int i;
            Console.WriteLine("Table de multiplication par 5");
            Console.WriteLine("=====");
            i = 1;
            do
            {
                Console.WriteLine("{0} * 5 = {1}", i, i * 5);
                i++;
            } while (i <= 10)
            // Affichage persistant
            Console.ReadLine();
        }
    }
}

```

#### Déroulement de l'exécution :

- Les instructions entre accolades sont exécutées.
- A chaque exécution de l'instruction **while**, la condition d'itération ( $i \leq 10$ ) est évaluée ; si  $i > 10$ , la boucle s'arrête, et l'instruction suivant l'accolade fermante est exécutée. Lorsque la condition d'itération est vraie, retour sur l'instruction **do**

## IV.4 LES RUPTEURS

Les rupteurs, en C#, sont des instructions de branchement inconditionnel à des points stratégiques du programme.

- continue** arrêt de la boucle en cours et débranchement à l'instruction responsable de la boucle.
- break** arrêt de la boucle en cours et débranchement derrière l'accolade fermante.
- return** dans une fonction retour au programme appelant quel que soit le niveau d'imbrication des structures. La valeur **n** constitue le résultat de la fonction. Si l'instruction **return** est trouvée dans la fonction **Main**, on retourne au système d'exploitation. La valeur **n** constitue alors la valeur de retour du programme (*status*) qui peut être utilisé par le système d'exploitation (**IF ERRORLEVEL** sous DOS).

Les rupteurs sont parfois dangereux pour une structuration correcte d'un programme. Ils doivent être utilisés à bon escient. Ce sont des GOTO déguisés

## IV.5 EXERCICES

### Exercice 1 : Suite de la calculette

Faire la saisie de 2 nombres entiers, puis d'un opérateur + - \* ou / .

Tant que l'opérateur n'est pas correct, le programme recommencera sa saisie. Enfin le programme effectuera l'opération demandée (en prévoyant le cas d'erreur "division par 0"), puis affichera le résultat.

### Exercice 2 : Multiples

Ecrire un programme qui calcule les N premiers multiples d'un nombre entier X, N et X étant entrés au clavier.

Il est demandé de choisir la structure répétitive (for, while, do...while) la mieux appropriée au problème.

On ne demande pas pour le moment de gérer les débordements (overflows) dus à des demandes de calcul dépassant la capacité de la machine.

### Exercice 3 : Somme et moyenne.

Ecrire le programme qui permet de saisir des entiers et en affiche la somme et la moyenne (on arrête la saisie avec la valeur 0)

Il est demandé de choisir la structure répétitive (for, while, do...while) la mieux appropriée au problème.

### Exercice 4 : Nombre de voyelles.

Ecrire le programme qui compte le nombre de voyelles d'un mot saisi au clavier, en utilisant :

- **Length** qui rend le nombre de lettres d'une chaîne donnée.
- **Substring(p,n)** qui extrait d'une chaîne donnée une sous chaîne de **n** caractères à partir de la position **p**.
- **IndexOf(schaîne)** qui restitue le rang de la première occurrence de **schaîne** dans chaîne donnée (si non trouvé : -1).

### Exercice 5 : Calcul du nombre de jeunes, de moyens et de vieux.

Il s'agit de dénombrer les personnes d'âge inférieur strictement à 20 ans, les personnes d'âge supérieur strictement à 40 ans et celles dont l'âge est compris entre 20 ans et 40 ans (20 ans et 40 ans y compris).

Le comptage est arrêté dès la saisie d'un centenaire. Le centenaire est compté.

Donnez le programme C# correspondant qui affiche les résultats.

## V FONCTIONS ET PARAMETRES

### V.1 LES OBJECTIFS

- Structuration d'un programme en fonctions élémentaires.
- Syntaxe et présentation des fonctions.
- Passage de paramètres.
- Surcharge des fonctions

### V.2 DEFINITION

Une fonction est une partie de programme comportant un ensemble d'instructions qui a besoin d'être utilisé plusieurs fois dans un programme ou dans différents programmes.

Parce que C# est un langage orienté objet, les fonctions ne peuvent être déclarées qu'à l'intérieur d'une classe. Il est donc impossible de déclarer des fonctions isolées.

Une fonction peut ou non renvoyer un résultat. Ce résultat n'est pas forcément exploité. Certaines fonctions (déclarée `void`) ne renvoient pas de résultat.

#### Exemple :

```
namespace Fonctions
{
    class Test1
    {

        // définition des fonctions
        static void AfficheMessage()
        {
            Console.WriteLine("Ceci est un message");
        }
        static bool ChercheEtTrouve()
        {
            bool btrouve;
            // ....
            return btrouve;
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            AfficheMessage();
            // ....
            AfficheMessage();
            // ....
            bool btrouve = ChercheEtTrouve()
        }
    }
}
```

Dans cet exemple, 4 fonctions ou méthodes sont nommées :

- `Main` est le point d'entrée de l'application
- `WriteLine` est une méthode de la classe `System.Console` du Framework.
- `AfficheMessage` est une méthode de la classe `Fonctions.Test1` qui ne renvoie aucune valeur
- `ChercheEtTrouve` est une méthode de la classe `Fonctions.Test1` qui renvoie une valeur booléenne.

Une fonction ou méthode est définie par

- Un nom
- Une liste de paramètres entre parenthèses
- Des instructions codées entre accolades, formant le corps de la méthode,

Exemple :

```
static void AfficheMessage ()  
{  
    // Corps de la méthode  
}
```

The diagram illustrates the components of the `AfficheMessage` method signature. A box labeled "Nom" points to the method name `AfficheMessage`. Another box labeled "Liste de paramètres" points to the empty parentheses `()`.

### V.3 UTILISATION DES FONCTIONS

Après avoir été définie, une fonction peut être appelée :

- A partir de la même classe

```
static void Main()  
{  
    Console.WriteLine("Programme principal");  
    // ....  
    AfficheMessage();  
    // ....  
    AfficheMessage();  
    // ....  
    bool btrouve = ChercheEtTrouve()  
}
```

La méthode est appelée par son nom.

- A partir d'une autre classe

Le nom de la méthode doit être précédé du nom de la classe qui contient la méthode qui aura été déclarée avec le mot clé **public**.

Une méthode non déclarée **public** est une méthode privée (**private**) pour la classe ; cette méthode ne pourra être appelée que de la classe ou elle est définie.

```
static void AfficheMessage ()  
private static void AfficheMessage ()
```

Les 2 notations sont équivalentes.

```

namespace Fonctions
{
    class Test1
    {
        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            Test2.AfficheMessage();
            // ....
            Test2.AfficheMessage();
        }
    }
    class Test2
    {
        // définition de la fonction
        public static void AfficheMessage()
        {
            Console.WriteLine("Ceci est un message");
        }
    }
}

```

Appel de la méthode de la classe Test2

Déclaration de la méthode **public**

- A partir d'une autre méthode (méthodes imbriquées)

Il est possible également d'appeler une méthode à partir d'une autre méthode.

```

namespace Fonctions
{
    class Test1
    {
        // définition des fonctions
        static void AfficheMessage()
        {
            Console.WriteLine("Ceci est un message");
        }
        static bool ChercheEtTrouve()
        {
            bool btrouve;
            // ....
            AfficheMessage();
            // ....
            return btrouve;
        }
    }

    // programme principal
    static void Main()
    {
        Console.WriteLine("Programme principal");
        // ....
        AfficheMessage();
        // ....
        AfficheMessage();
    }
}

```

AfficheMessage est appelée à partir de ChercheEtTrouve

```

    // ....
    bool btrouve = ChercheEtTrouve()
}
}

```

## V.4 LE CORPS DE METHODE

### V.4.1 Les variables

Des variables peuvent être déclarées dans le corps de méthode : elles sont locales, créées au début de la méthode et détruites à la sortie

```

static void ChercheEtTrouve ()
{
    bool btrouve = false;
    // ....
}

```

Pour partager une des informations entre méthodes, on peut utiliser une variable de classe.

```

namespace Fonctions
{
    class Test1
    {
        // définition de la variable de classe
        static string strMess;
        // définition les fonctions
        static void EtablirMessage()
        {
            strMess = "Ceci est un message";
        }

        static void AfficheMessage()
        {
            Console.WriteLine(strMess);
        }
        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            EtablirMessage();
            // ....
            AfficheMessage();
            // ....
        }
    }
}

```

### V.4.2 . L'instruction return

L'instruction **return** dans une fonction permet le retour immédiat à l'appelant.



Dans le cas de l'exemple précédent :

```
static void Main()  
{  
    Console.WriteLine("Programme principal");  
    // ....  
    EtablitMessage();  
    return;  
    AfficheMessage();  
    // ....  
}
```

L'ajout de l'instruction **return** entre l'appel des 2 fonctions provoque un avertissement du compilateur « impossible d'atteindre le code détecté » ; le programme n'exécute jamais la fonction `AfficheMessage`.

Il est plus courant d'utiliser l'instruction **return** dans une expression conditionnelle : le retour au niveau supérieur se fait que si la condition est vraie.

Dans le cas de méthode renvoyant un résultat, l'instruction **return** suivie d'une expression termine la méthode immédiatement en retournant l'expression comme valeur de retour de la méthode.

```
static bool ChercheEtTrouve ()  
{  
    bool btrouve = false;  
    // ....  
    return btrouve;  
}
```

L'instruction **return** est obligatoire dans ce cas.

## V.5 LE PASSAGE DE PARAMETRES

Les paramètres permettent de passer des informations à l'intérieur et à l'extérieur d'une méthode.

### V.5.1 . Déclaration des paramètres

Chaque paramètre se caractérise par un type et un nom. Chaque paramètre est séparé de l'autre par une virgule.

```
static void AfficheMessage(string strMess, int nbfois)  
{  
    for (int i = 1; i <= nbfois; i++)  
    {  
        Console.WriteLine(strMess);  
    }  
}
```

### V.5.2 . Appel de la méthode

Le code appelant doit fournir les valeurs des paramètres lors de l'appel de la méthode, dans l'ordre de définition.

Dans le cas de l'exemple précédent :

```
static void Main()
{
    Console.WriteLine("Programme principal");
    // ....
    string strMess = "Ceci le premier message affiché 2 fois";
    AfficheMessage(strMess, 2);
    // ....
    strMess = "Ceci est le 2eme message";
    AfficheMessage(strMess, 1);
    // ....
}
```

**C# utilise par défaut le mécanisme de passage des paramètres par valeur** : les données sont transférées de l'extérieur de la méthode vers l'intérieur : ce sont des **paramètres d'entrée**.

### V.5.3 Méthode de passage de paramètres

Il existe trois façons de passer des paramètres :

- **Par valeur** (paramètres **d'entrée**)

Les données sont transférées de l'extérieur vers l'intérieur de la méthode.

La valeur du paramètre est copiée : la variable peut être modifiée à l'intérieur de la méthode sans influence sur sa valeur à l'extérieur.

```
namespace Fonctions
{
    class Test2
    {
        // définition de la fonction
        static void Incremente(int n)
        {
            n++ ;
            Console.WriteLine("n= {0}",n); // Affiche 2
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            int x = 1;
            Incremente(x);
            Console.WriteLine("x= {0}", x); // Affiche 1

            Console.ReadLine();
        }
    }
}
```

Le paramètre doit être initialisé avant l'appel de la fonction ; Après l'appel de la fonction, la valeur affichée est toujours 1 : Il n'y a pas de retour de la valeur de la fonction vers l'appelant.

- **Par référence** (paramètres **d'entrée/sortie**)

Les données peuvent être transférées de l'extérieur vers l'intérieur, puis vers l'extérieur de la méthode.

On utilise le mot clé **ref** par paramètre, pour spécifier un appel par référence, dans la méthode et lors de son appel.

```
namespace Fonctions
{
    class Test2
    {
        // définition de la fonction
        static void Incremente (ref int n)
        {
            n++ ;
            Console.WriteLine("n= {0}",n); // Affiche 3
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            int x = 1;
            Incremente (ref x);
            Console.WriteLine("x= {0}", x); // Affiche 2

            Console.ReadLine();
        }
    }
}
```

Le paramètre doit être initialisé avant l'appel de la fonction ; Après l'appel de la fonction, la valeur affichée est maintenant 2 : Il y a retour de la valeur de la fonction vers l'appelant.

- **Paramètres de sortie**

Les données ne peuvent être transférées que de l'intérieur vers l'extérieur de la méthode.

On utilise le mot clé **out** par paramètre, pour spécifier un appel avec paramètre de sortie, dans la méthode et lors de son appel.

```

namespace Fonctions
{
    class Test3
    {
        // définition de la fonction
        static void Incremente (out int n)
        {
            n=3 ;
            Console.WriteLine("n= {0}",n); // Affiche 2
        }

        // programme principal
        static void Main()
        {
            Console.WriteLine("Programme principal");
            // ....
            int x ;
            Incremente (out x);
            Console.WriteLine("x= {0}", x); // Affiche 3

            Console.ReadLine();
        }
    }
}

```

Dans la fonction, le paramètre doit être initialisé avant d'être utilisé ; Après l'appel de la fonction, la valeur affichée est 3 : Il y a retour de la valeur de la fonction vers l'appelant.

## V.6 LA RECURSIVITE

Le langage C# permet les fonctions récursives. C'est-à-dire que l'une des instructions d'une fonction peut être un appel à la fonction elle-même. C'est très pratique pour coder certains algorithmes comme la factorielle :

$\text{factorielle}(n) = n * \text{factorielle}(n - 1)$

ou l'algorithme d'Euclide :

si  $n_2 > n_1$ ,  $\text{pgcd}(n_1, n_2) = \text{pgcd}(n_1, n_2 - n_1)$

Ce principe est basé sur une notion mathématique : la *réurrence* :

Pour démontrer qu'une propriété est vraie quelle que soit la valeur de  $n$ , on démontre que :

- La propriété est vraie pour  $n=1$ .
- Si la propriété est vraie pour  $n-1$ , elle est vraie pour  $n$ .

Ainsi, si les deux théorèmes précédents sont démontrés, on saura que la propriété est vraie pour  $n=1$  (1er théorème). Si elle est vraie pour  $n=1$  elle est vraie pour  $n=2$  (2ème théorème). Si elle est vraie pour  $n=2$ , elle est vraie pour  $n=3$ ... Et ainsi de suite.

La création d'une fonction récursive risque d'engendrer un phénomène sans fin. C'est pourquoi, on prévoira toujours une fin de récursivité. Cette fin correspond en fait au codage du premier théorème :

```

static long SommeDesNombres(long n)
{
    if (n == 1)
    {
        return 1;           // 1er théorème
    }
    else
    {
        return n + SommeDesNombres(n - 1); // 2ème théorème
    }
}

```

## V.7 LES METHODES SURCHARGEES

Le langage C# permet que deux fonctions différentes aient le même nom à condition que les paramètres de celles-ci soient différents soit dans leur nombre, soit dans leur type, soit dans le mode de passage du paramètre (out ou ref).

Par exemple, les trois fonctions prototypées ci-dessous sont trois fonctions différentes. Cette propriété du langage C# s'appelle la *surcharge*.

```

double maFonction(double par1);
double maFonction(double par1, double par2);
double maFonction(int par1);

```

Contrairement aux autres langages de programmation, ce qui permet au compilateur d'identifier et de distinguer les sous-programmes, ce n'est pas le nom seul de la fonction, mais c'est la *signature*. Deux fonctions sont identiques si elles ont la même signature, c'est-à-dire le même **nom**, le même **nombre de paramètres** de même **type** et de même **mode** de passage.

Le nom du paramètre et le type de retour de la fonction n'affectent pas la signature par exemple, ces 2 fonctions sont identiques, et ne pourront donc pas être déclarées dans la même classe.

```

double maFonction(double par1);
int maFonction(double parx);

```

### Exemple :

```

class Exemple
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return Add(a,b)+ c;
    }
    static void Main()
    {
        Console.WriteLine(Add(1, 2) + Add(1, 2, 3));
    }
}

```

La méthode **Add** a été surchargée pour pouvoir traiter l'addition de 3 entiers. On notera que dans le corps de la méthode surchargée, on utilise la méthode initiale, pour éviter la redondance de code.

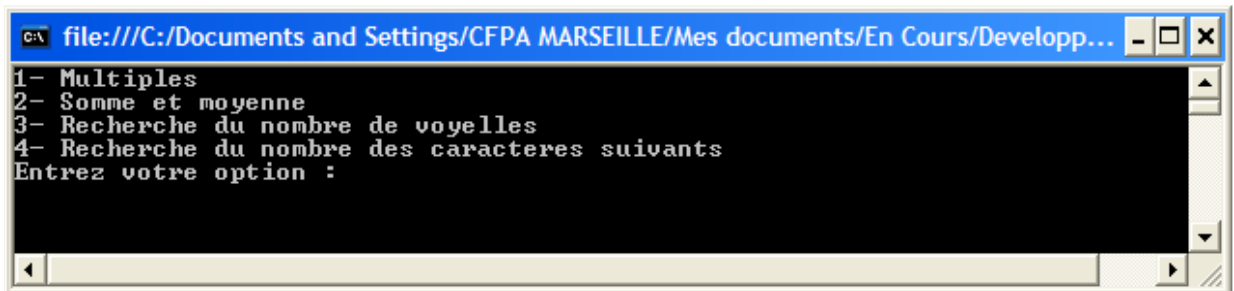
Il est évident que l'abus de cette possibilité peut amener à écrire des programmes difficiles à maintenir.

Cependant, cela permet de donner le même nom à des fonctions qui jouent le même rôle mais dont le nombre et le type de paramètres doivent être différents. C'est le cas, par exemple, de la fonction **IndexOf** de la classe **String** qui peut recevoir 1, 2 voire 3 paramètres. Dans tous les cas, le premier paramètre est la portion de chaîne (ou le caractère) que l'on veut repérer dans la chaîne globale (voir paragraphe II.3.b du présent support).

## V.8 EXERCICES

### Exercice 1 : Menu

A partir du menu affiché à l'écran



Vous exécuterez, par les 3 premières options, les exercices réalisés, appelés sous forme de fonction (transformation de vos TP).

L'option 4 est une généralisation de la recherche du nombre de voyelles dans un mot : elle permet de rechercher la présence de n'importe quel caractère dans une chaîne.

La recherche de voyelles dans une chaîne constitue une surcharge de cette fonction, dans la mesure où les caractères à rechercher seront fournis sous forme de chaîne.

Créer 2 fonctions supplémentaires pour automatiser la lecture de données standard au clavier, que vous appellerez systématiquement pour toute lecture clavier (modification de vos TP):

- une fonction **GetString** pour lire et restituer une chaîne de caractères au clavier,
- une fonction **GetInteger** pour lire et restituer un entier au clavier,

## VI LES TABLEAUX

### VI.1 LES OBJECTIFS

L'objectif de ce chapitre est de créer, initialiser et utiliser des tableaux.

### VI.2 DEFINITION

Imaginons que dans un programme, nous ayons besoin simultanément de 12 valeurs (par exemple, des notes pour calculer une moyenne). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions de lecture distinctes, cela donnera obligatoirement quelque chose comme :

$$\text{Moy} = (N1 + N2 + N3 + N4 + N5 + N6 + N7 + N8 + N9 + N10 + N11 + N12) / 12$$

Ce qui est laborieux. Et pour un peu que nous soyons dans un programme de gestion avec quelques centaines ou quelques milliers de valeurs à traiter, cela se révèle fortement problématique.

Si, de plus, on est dans une situation où l'on ne peut pas savoir d'avance combien il y aura de valeurs à traiter, on se retrouve là face à un mur.

C'est pourquoi la programmation nous permet **de rassembler toutes ces variables en une seule**, au sein de laquelle chaque valeur sera désignée par un numéro. En bon français, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ».

Un tableau est une **suite d'éléments de même type**. On peut accéder à un élément d'un tableau en utilisant sa position : l'index.

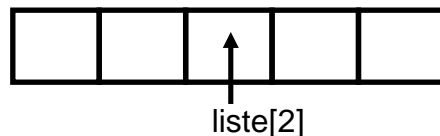
En C#, le premier élément se trouve à l'**index 0**.

### VI.3 DECLARATION ET CREATION

Un tableau est déclaré en spécifiant son type, sa dimension (ou rang) et son nom.

```
type [ ] nom ;
```

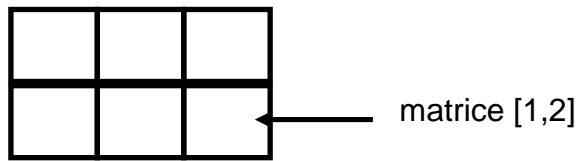
```
int [ ] liste; // tableau à une dimension
```



liste[2] désignera le 3<sup>ème</sup> élément du tableau liste.

Un seul index est nécessaire pour repérer un élément de tableau.

```
string [ , ] matrice ; // tableau à deux dimensions
```



matrice [1,2] désignera l'élément situé sur la 2<sup>ème</sup> ligne, 3<sup>ème</sup> colonne du tableau matrice.

Deux index sont nécessaires pour repérer un élément de tableau, l'indice des lignes et l'indice des colonnes

La déclaration d'une variable tableau n'entraîne pas sa création: il faut obligatoirement utiliser **new** pour créer explicitement l'instance du tableau et spécifier la taille de toutes les dimensions.

```
int[] liste; // déclaration
liste = new int[5]; // création d'une instance
```

que l'on peut résumer en :

```
int[] liste = new int[5]; // création d'une instance
int [,] matrice= new string [2,3] ;
```

Le compilateur C# initialise implicitement chaque élément du tableau à sa valeur par défaut (en fonction du type).

que l'on peut initialiser comme suit :

```
int[] liste = new int[5]{4,2,3,1,5};
int [,] matrice= new int[2,3] {
    {1,2,3},
    {10,20,30}};
```

Chaque élément doit obligatoirement être initialisé.

La taille d'un tableau n'est pas nécessairement une constante; elle peut également être spécifiée par une valeur entière au moment de la compilation.

```
int taille = int.Parse(Console.ReadLine()); // lecture
int []liste= new int [taille]; // création
int [,]matrice= new int [2,taille]; // création
```

En C#, lors d'une tentative d'accès au tableau, l'index est automatiquement contrôlé de manière à garantir sa validité.

Un index hors limite – inférieur à 0 ou supérieur ou égal à sa taille- renvoie une exception **IndexOutOfRangeException**.



## VI.4 UTILISATION

Soit 2 tableaux ainsi déclarés :

```
int[] liste = new int[5] {4,2,3,1,5}; // création d'une instance
string [,] matrice= new string [2,3] {
    {1,2,3},
    {10,20,30}};
```

### VI.4.1 . Propriétés

La propriété **Length** renvoie le nombre total de postes d'un tableau.

liste.Length rend la valeur 5  
matrice.Length rend la valeur 6 (2 \* 3)

La propriété **Rank** renvoie le nombre de dimensions du tableau

liste.Rank rend la valeur 1  
matrice.Rank rend la valeur 2.

### VI.4.2 . Méthodes

La méthode **Sort** permet de trier un tableau.

System.Array.Sort(liste); rend dans liste un tableau trié.

La méthode **Clear** permet de réinitialiser un tableau à sa valeur par défaut.

```
// initialise p éléments à partir de la position i
System.Array.Clear(liste, i, p);
```

La méthode **GetLength** permet de renvoyer la longueur d'une dimension.

```
matrice.GetLength(0) ; //rend la valeur 2.
matrice.GetLength(1) ; //rend la valeur 3.
```

La méthode **Clone** crée une nouvelle instance de tableau dont les éléments sont des copies des éléments du tableau cloné.

```
int[] listeCopie = (int [])liste.Clone();
```

alors que l'instruction suivante ne copie pas l'instance du tableau, mais se contente de référencer la même instance de tableau.

```
int[] listeCopie = liste ;
```

### VI.4.3 Utilisations particulières

Un tableau peut être paramètre de retour de méthode, ou passé en paramètre dans une méthode.

```
namespace Tableaux
```

```
{
```

```
class Test1
```

```
{
```

```
    static void Main()
```

```
    {
```

```
        // création nouveau tableau
```

```
        int[] nouveauT = CreeTableau(5);
```

```
        Console.WriteLine("Nb de postes = {0}", nouveauT.Length);
```

```
        // ....
```

```
        // Affichage du tableau
```

```
        AfficheTableau(nouveauT);
```

```
        //
```

```
        Console.ReadLine();
```

```
    }
```

```
    static int[] CreeTableau(int taille)
```

```
    {
```

```
        return new int[taille];
```

```
    }
```

```
    static void AfficheTableau(int [] unTableau )
```

```
    {
```

```
        for (int i = 0; i < unTableau.Length; i++)
```

```
        {
```

```
            Console.WriteLine("Poste {0}: {1}", i, unTableau[i]);
```

```
        }
```

```
    }
```

```
}
```

```
}
```

Tableau en retour de méthode

Passage d'un tableau en parametre

C'est ainsi que Main peut accepter un tableau de chaînes comme paramètre, réceptionnant les arguments de la ligne de commande.

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

```
    {
```

```
        // ....
```

```
    }
```

### VI.4.4 L'instruction foreach

L'instruction **foreach** simplifie le parcours des éléments d'un tableau.

```
foreach (int entier in unTableau)
```

```
{
```

```
    Console.WriteLine(entier);
```

```
}
```

Les éléments d'index (initialisation, contrôle et incrémentation) sont inutiles.

## VI.5 OPERATION ET TRI SUR LES TABLEAUX

### VI.5.1 Opérations diverses sur un tableau non trié

Sur une structure de données de type tableau, il est possible de faire plusieurs types de traitement, comme par exemple **la recherche du minimum ou du maximum, la somme ou le produit ou la moyenne des postes du tableau.**

On peut également vouloir **rechercher un élément donné** dans un tableau.

Par exemple, sur un tableau de 35 postes numériques, on désire calculer la somme des postes, et rechercher le plus petit élément.

Pour le calcul de la somme, on ajoutera le contenu des cases une à une, depuis la première jusqu'à la trente cinquième.

Pour la recherche du minimum :

- On va supposer que la première case contient le minimum relatif
- On va comparer le contenu de la deuxième case avec le minimum relatif : si celui-ci est inférieur, il deviendra le minimum relatif.
- On recommencera l'opération avec les postes restants

### VI.5.2 Tri d'un tableau

Un tableau est ordonné lorsqu'il existe une relation d'ordre entre les différentes cases : On parle de :

- tri croissant si le contenu de la case d'indice  $i$  est inférieur ou égal au contenu de la case d'indice  $i + 1$
- tri décroissant si le contenu de la case d'indice  $i$  est supérieur ou égal au contenu de la case d'indice  $i + 1$

Plusieurs méthodes de tri existent ; en voici deux exemples sur la base d'un tableau de 30 valeurs numériques VALNUM

#### Tri croissant par recherche successive des minima

Le principe est le suivant :

- Recherche du minimum dans le tableau de 30 valeurs, et échange du contenu des cases d'indice 1 et d'indice correspondant à la valeur du minimum.
- Application du même principe sur 29 valeurs (30 - première), puis sur 28, puis 27 .... jusqu'au tableau de deux cases.

Visualisation du traitement sur 4 valeurs :

Tableau initial	8	1	7	5
Après le premier passage	1	8	7	5
Après le deuxième passage	1	5	7	8
Après le troisième passage	1	5	7	8

Lors du premier passage, on a inversé les cases d'indices 1 et 2.  
Lors du deuxième passage, le minimum de (5, 7,8) étant 5, on a inversé les cases d'indices 2 et 4.  
Lors du troisième passage, rien ne s'est passé.

### **Tri à bulle**

Le principe est le suivant :

Le tableau est parcouru en comparant les éléments consécutifs.  
S'ils sont mal ordonnés, ces deux éléments sont permutés. On recommence jusqu'à ce qu'il n'y ait plus d'échange.

Visualisation du traitement sur 5 valeurs :

Tableau initial	5	18	14	4	26
Premier passage	5	14	18	4	26
	5	14	4	18	26
Deuxième passage	5	4	14	18	26
Troisième passage	4	5	14	18	26
Quatrième passage	4	5	14	18	26

Comme aucune permutation n'a été réalisée, l'algorithme s'arrête.

### **Méthode :**

Les éléments sont comparés deux à deux, et on affecte une variable booléenne à vraie si un échange est réalisé.  
La condition d'arrêt du traitement est que la variable booléenne soit restée à faux.

### **VI.5.3 Recherche d'un élément sur un tableau trié**

Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier.

Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20000 tours de boucle. C'est beaucoup, même pour un ordinateur.

Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, un être humain qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots soient triés.

Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve pile poil au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans la première moitié du dico. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.

A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.

Regardons ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire :

Au départ, on cherche le mot parmi 40 000.

Après le test n°1, on ne le cherche plus que parmi 20 000.

Après le test n°2, on ne le cherche plus que parmi 10 000.

Après le test n°3, on ne le cherche plus que parmi 5 000.

etc.

Après le test n°15, on ne le cherche plus que parmi 2.

Après le test n°16, on ne le cherche plus que parmi 1.

Et là, on sait que le mot n'existe pas. Moralité : on a obtenu notre réponse en 16 opérations contre 40 000 précédemment ! Il n'y a pas photo sur l'écart de performances entre la technique barbare et la technique fûtée. Attention, toutefois, même si c'est évident, **la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.**

## VI.6 EXERCICES

### Exercice 1 : Manipulations de base

Créer le programme C# qui fournira un menu permettant d'obtenir les fonctionnalités suivantes à partir d'un tableau à une dimension contenant 10 entiers :

- Affichage du contenu de tous les postes du tableau,
- Affichage du contenu d'un poste dont l'index est saisi au clavier,
- Affichage du maximum et de la moyenne des postes du tableau

Ce programme sera structuré de la manière suivante :

- une fonction **GetInteger** pour lire un entier au clavier,
- une fonction **InitTab** pour créer et initialiser l'instance de tableau de type entier : le nombre de postes souhaité sera entré au clavier,
- une fonction **AfficheTab** pour afficher tous les postes du tableau,
- une fonction **RechercheTab** pour afficher le contenu d'un poste de tableau dont le rang est saisi au clavier
- une fonction **InfoTab** qui affichera le maximum et la moyenne des postes.

### Exercice 2 :

On recherche dans un tableau contenant 20 prénoms, un prénom saisi au clavier. Lorsque cet élément est trouvé, on l'élimine du tableau en décalant les cases qui le suivent, et en mettant à blanc la dernière case

### Exercice 3 : Tri d'un tableau

Ecrire le programme C# qui réalise le tri à bulles.

## VII AUTRES TYPES DE DONNEES

### VII.1 LES OBJECTIFS

- Créer et utiliser des types de données **enum** et **struct** définis par l'utilisateur
- Utiliser les classes de manipulation de date et heure

### VII.2 LE TYPE ENUM

Les énumérateurs sont utiles lorsqu'une variable ne peut prendre qu'un ensemble spécifique de valeurs. Leur but est de rendre le code source plus compréhensible.

Un type énuméré est un entier défini par le programmeur à l'aide du mot `enum` ; il s'agit d'une liste de constantes symboliques qui possèdent un identificateur et formant un ensemble de valeurs appartenant à un nouveau type.

**Définition d'un type enum :**

```
enum EtatCivil {Celibataire, Marie, Divorcé, Veuf} ;
```

**Utilisation d'un type enum :**

On pourra déclarer une variable `ec` de type `EtatCivil`, en utilisant la syntaxe suivante :

```
EtatCivil ec ;
```

On pourra l'affecter en codant :

```
ec = EtatCivil.Marie ;
```

Ou

```
ec =(EtatCivil)1; // cast d'un type int en EtatCivil
```

On pourra l'afficher en codant :

```
Console.WriteLine("Statut :{0}",ec);
```

Par défaut, le compilateur associe la valeur 0 à célibataire, la valeur 1 à marié, la valeur 2 à divorcé et la valeur 3 à Veuf.

La déclaration peut être placée dans la classe, mais en dehors d'une fonction, ou en dehors de la classe ; le point virgule n'est pas obligatoire.

On pourrait définir des valeurs spécifiques pour chaque valeur de l'énumération :

```
enum EtatCivil
{
    Celibataire=10,
    Marie =20,
    Divorcé= Marie + 1,
    Veuf= 40
}
```

Il est cependant prudent de prévoir une valeur zéro pour la plus commune des valeurs pour des problèmes d'initialisation.

On peut également convertir une chaîne de caractères en l'une des valeurs possibles de l'énumération :

```
string strStatut = "Veuf";  
ec = (EtatCivil)Enum.Parse(typeof(EtatCivil),strStatut,false) ;
```

le 3<sup>ème</sup> argument signale que la casse est ignorée.

Lorsque le 2<sup>ème</sup> argument ne correspond à aucune des valeurs possibles, une erreur de type **ArgumentException** est générée.

### VII.3 LE TYPE STRUCT

Dans sa forme la plus simple, une structure comprend plusieurs informations regroupées dans une même entité.

Les objets créés à partir de structures se comportent comme des types prédéfinis.

**Exemple :** Simulation d'un point en géométrie

```
struct Point  
{  
    public int x;  
    public int y;  
}
```

La structure contient 2 membres de type `int` spécifiant la position horizontale et verticale d'un point.

La structure peut être définie à l'intérieur ou à l'extérieur d'une classe, mais en dehors d'une fonction; le point virgule n'est pas obligatoire.

**Utilisation d'un type struct :**

On pourra déclarer une variable *p* de type `Point`, en utilisant la syntaxe suivante :

```
Point point ;
```

Les champs de *p* seront accessibles par `point.x` et `point.y` .et ne sont pas initialisés.

**Exemple:**

```
static void Main()  
{  
    Point point;  
    point.x = 0; Point.y = 0;  
    Console.WriteLine("Coordonnées : {0},{1}",point.x, point.y);  
}
```

### VII.4 DATES ET HEURES

Le type valeur **DateTime** représente des dates et des heures dont la valeur est comprise entre 12:00:00 (minuit), le 1er janvier de l'année 0001 de l'ère commune et 11:59:59 (onze heures du soir), le 31 décembre de l'année 9999 après J.C. (ère commune).

Il permet d'effectuer diverses opérations sur les dates : déterminer le jour de la semaine, déterminer si une date est inférieure à une autre, jouter un nombre de jours à une date.

## Déclaration d'une date :

```
// crée un objet DateTime
DateTime d1= new DateTime();
//crée un objet DateTime pour le 01/01/2008
DateTime d2 = new DateTime(2008, 01, 01);
// crée un objet DateTime pour le 01/01/2008 12h 0mn 0s 0 ms
DateTime d3 = new DateTime(2008, 01, 01, 12, 0, 0, 0);
```

## Utilisation d'un type DateTime :

La structure **DateTime** possède quelques propriétés intéressantes :

- **Date** permet de rendre un nouveau DateTime avec la partie heure initialisée à 0  
`Console.WriteLine(d3.Date); // rend 2008/01/01 à 0h`
- **Day, Month, Year** permettent de rendre respectivement jour, mois année.  
**Hour, Minute, Second, Millisecond** permettent de rendre respectivement heures, minutes, secondes et millisecondes.  
`Console.WriteLine("{0},{1},{2}", d2.Day, d2.Month, d2.Year);  
// rend 1,1,2008`
- **DayOfWeek** permet de rendre le jour de la semaine  
`Console.WriteLine("{0}", d2.DayOfWeek); // rend Tuesday`
- **DayOfYear** permet de rendre le quantième correspondant à la date.  
`Console.WriteLine("{0}", d2.DayOfYear); // rend 1`
- **Now** donne la date et l'heure du jour  
`// crée un objet DateTime initialisé avec date et heure du jour  
DateTime d1 = DateTime.Now;`
- **Today** donne la date du jour : l'heure est initialisée à 0  
`// crée un objet DateTime initialisé avec date du jour  
DateTime d1 = DateTime.Today;`

Et possède également un certain nombre de méthodes :

- **AddDays, AddMonths, AddYears** permettent d'ajouter respectivement jour, mois, et année.  
**AddHours, AddMinutes, AddSeconds, AddMilliseconds** permettent d'ajouter respectivement heures, minutes, secondes et millisecondes.
- **Compare, CompareTo** permettent de comparer deux dates  
`int res1 = DateTime.Compare(d2, d3);  
int res2 = d2.CompareTo(d3);`  
et renvoient 0 si d2=d3, 1 si d2> d3, -1 sinon.  
Les opérateurs de comparaison peuvent également être employés.
- **ToString** permet de mettre en forme une date  
En utilisant des formats généraux  
`// Format date longue  
Console.WriteLine(d2.ToString("D")); // mardi 1 janvier 2008  
// Format date courte  
Console.WriteLine(d2.ToString("d")); // 01/01/2008  
// Jour + Mois en clair  
Console.WriteLine(d2.ToString("M")); // 1 janvier`



En utilisant des formats personnalisés, par exemple :

d, M	Jour ( 0 à 31), mois (01 à 12)
dd ,MM	Jour (01 et 31), mois (01 à 12)
ddd, MMM	Abréviation du jour/ mois
dddd, MMMM	Nom complet du jour / mois
y	Année (de 1 à 99)
yy	Année (de 01 à 99)
yyyy	Année (de 1 à 9999)

```
Console.WriteLine(d2.ToString("d-MMMM-yy")); // 1-janvier-08
```

(Voir l'aide en ligne pour les autres méthodes, et tous les formats).

### **DateTime et TimeSpan :**

Les types valeur **DateTime** et **TimeSpan** se distinguent par le fait que **DateTime** représente un instant, tandis que **TimeSpan** représente un intervalle de temps. Cela signifie, par exemple, que vous pouvez soustraire une instance **DateTime** d'une autre pour obtenir l'intervalle de temps les séparant. De la même façon, vous pouvez ajouter un **TimeSpan** positif au **DateTime** en cours pour calculer une date ultérieure.

Vous pouvez ajouter ou soustraire un intervalle de temps d'un objet **DateTime**. Les intervalles de temps peuvent être négatifs ou positifs ; ils peuvent être exprimés en unités telles que les graduations ou les secondes ou comme un objet **TimeSpan**.

### **Déclaration d'un TimeSpan :**

```
// crée un nouveau TimeSpan
TimeSpan ts1 = new TimeSpan();
//crée un nouveau TimeSpan en heure, mn, secondes
TimeSpan ts2 = new TimeSpan(10,20,30);
```

La structure **TimeSpan** possède des propriétés:

- **Days, Minutes, Seconds, Milliseconds** permettent de rendre respectivement le nombre de jour, minutes, secondes et millisecondes de l'intervalle de temps.

Et aussi des méthodes :

- **Add, Subtract** permettent d'ajouter ou de soustraire un intervalle de temps passé en argument.

### **Exemple : Calcul du nombre de jours écoulés depuis le début du 21eme siècle**

```
DateTime dj = DateTime.Today; // aujourd'hui
DateTime ds = new DateTime(2001, 01, 01); // début du 21eme
siècle
TimeSpan ts = dj - d;
Console.WriteLine("Nb de jours : {0}", ts.Days);
```

## VII.5 EXERCICES

### **Exercice : Majorité**

A partir de la saisie de sa date de naissance, affichez si le candidat est majeur ou non.

Plusieurs solutions sont possibles ...

## **Synthèse : Le jeu du pendu**

Ecrire le programme du jeu du pendu.

Le principe est le suivant :

Un premier joueur choisit un mot de moins de 10 lettres.

Le programme affiche \_ \_ \_ \_ \_ (un \_ par lettre.)

Le deuxième joueur propose des lettres jusqu'à ce qu'il ait trouvé le mot ou qu'il soit pendu (11 erreurs commises).

A chaque proposition le programme réaffiche le mot avec les lettres découvertes ainsi que les lettres déjà annoncées et le nombre d'erreurs.

Le sujet est ardu, si on se jette sur le programme la tête la première ... !!!

Sollicitez tout d'abord un de vos collègues de formation, et jouez avec lui. Puis recherchez ensemble les grandes fonctions qui constituent ce programme, et leurs liens.

L'objectif est plus clair désormais ...

Une fois l'application découpée en tâches élémentaires, il est plus facile de coder !

## VIII LE TRAITEMENT DES ERREURS

### VIII.1 LES OBJECTIFS

- Repérer les instructions qui sont susceptibles de générer des exceptions
- Capturer des exceptions pour afficher les messages d'erreur adéquats.
- Générer des exceptions

### VIII.2 VUE D'ENSEMBLE

Des erreurs d'exécution peuvent se produire lors de l'exécution d'une instruction : Si aucun traitement particulier n'est effectué, l'application s'arrête brutalement, ce qui en général, n'est guère apprécié des utilisateurs, et peut provoquer des pertes de données.

Il faut donc identifier les parties de programme où des erreurs d'exécution peuvent se produire, et écrire du code spécifique pour les traiter.

Les techniques de traitement d'erreurs permettent en cours d'exécution de programme de détecter et de réagir à :

- Des erreurs générées par le système, comme les divisions par 0, l'utilisation d'un index de tableau invalide ou la non correspondance d'une valeur d'une énumération.
- Des erreurs générées par une fonction du programme, par exemple une valeur négative ou trop élevée pour l'âge d'une personne.

**Exemple :**

```
static void Main(string[] args)
{
    int a = 10, b = 0, c;
    Console.WriteLine("La division");
    c = a/b;
    Console.WriteLine(c);
}
```

Le programme est interrompu par une erreur,

**L'exception DivideByZeroException n'a pas été gérée.**

Le programme s'arrête brutalement, et le deuxième affichage n'est jamais exécuté.

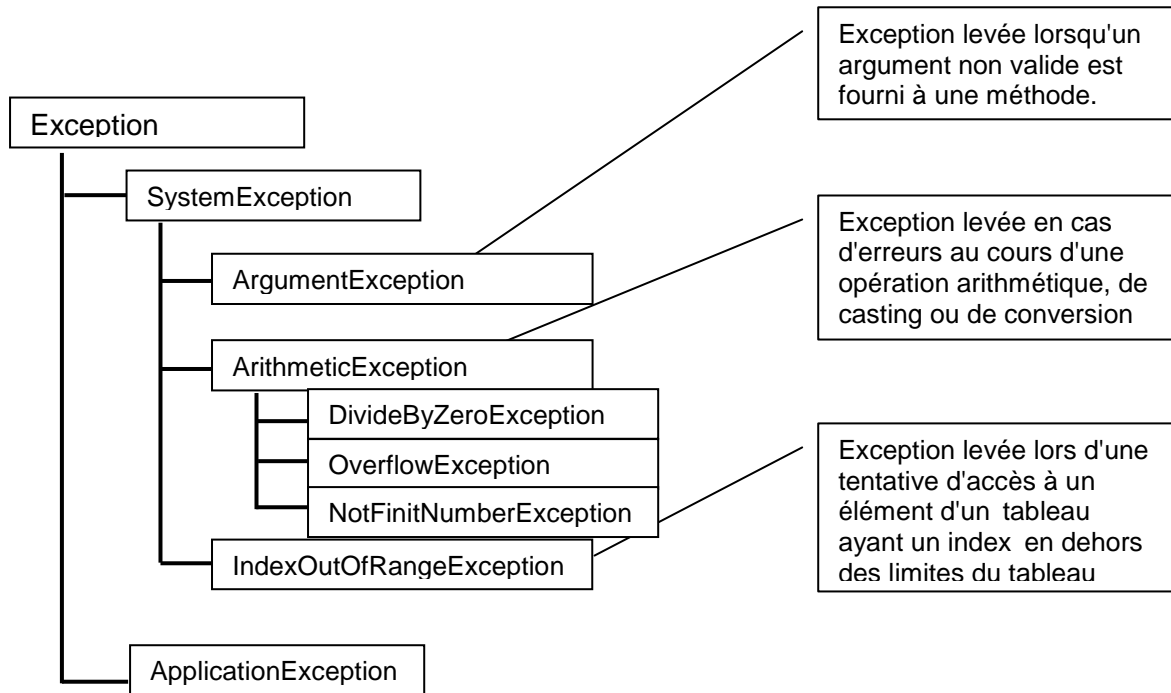
### VIII.3 LES OBJETS EXCEPTION

Le framework définit une hiérarchie de classes d'exception, dont la classe de base est la classe **Exception**.

Il existe deux catégories d'exceptions sous la classe de base **Exception** :

- les classes d'exceptions prédéfinies du Common Language Runtime, dérivées de **SystemException**.
- les classes d'exceptions de l'application définies par l'utilisateur, dérivées de **ApplicationException**.

Chaque classe d'exception décrit clairement l'erreur qu'elle représente.



## VIII.4 LES CLAUSES TRY ET CATCH

L'idée est de séparer les instructions essentielles du programme, des instructions de gestion d'erreur.

Les parties de code susceptibles de lever des exceptions sont placées dans un bloc **try**, et le code de traitement de ces exceptions est placé dans un bloc **catch**.

A l'intérieur du groupe **try**, les instructions sont écrites, sans se soucier du traitement d'erreurs.

**Dans notre exemple :**

```

static void Main(string[] args)
{
    int a = 10, b = 0, c;
    try
    {
        Console.WriteLine("La division");
        c = a/b;
        Console.WriteLine(c);
    }
    catch (Exception)
    {
        Console.WriteLine("Erreur arithmétique");
    }
}

```

Le programme rentre dans le bloc **try**, comme si l'instruction **try** n'existait pas, et les instructions sont exécutées en séquence.

Dés qu'une erreur est détectée par le système, un objet de la classe `Exception`, ou classe dérivée est automatiquement créé, et le programme se débranche à la première instruction du bloc **catch** correspondant.

A l'intérieur des parenthèses de la clause catch, un nom d'objet est souvent spécifié, de manière à obtenir des informations supplémentaires au sujet de l'erreur, par exemple le champ Message de la classe Exception, qui délivre en clair le message de l'erreur.

```
catch (Exception e)
{
    Console.WriteLine("Erreur arithmétique \n" + e.Message);
}
```

Le message affiché sera :

*Erreur arithmétique  
Tentative de division par zéro.*

Nous nous sommes servi de la classe générale d'exception pour traiter l'erreur ; de ce fait, toutes les erreurs détectées par n'importe quelle instruction afficheront ce même message... ce qui paraît absurde !

Il est possible, et recommandé d'enchaîner les blocs catch chacun correspondant à un type d'exception traité.

```
static void Main(string[] args)
{
    int a = 10, b = 0, c;
    try
    {
        Console.WriteLine("La division");
        c = a/b;
        Console.WriteLine(c);
    }
    catch (ArithmeticException ae)
    {
        Console.WriteLine("Erreur arithmétique\n" + ae.Message);
    }
    catch (Exception e)
    {
        Console.WriteLine("Erreur générale\n" + e.Message);
    }
}
```

De ce fait, si aucun bloc catch spécifique n'existe, l'exception est interceptée par un bloc catch général, le cas échéant.

Il faut veiller à toujours **classer les exceptions dans les blocs catch de la plus spécifique à la plus générale**.

Cette technique permet de gérer l'exception spécifique avant qu'elle ne passe à un bloc catch plus général.

**Attention :** vous vous apercevrez rapidement que la mise en place d'une gestion d'erreurs affecte les performances de manière considérable .... Réservez ces traitements aux erreurs imprévisibles (vous préférerez ainsi contrôler préventivement la saisie d'un nombre en mémoire plutôt que de mettre en place un bloc try ... catch).

## VIII.5 LE GROUPE FINALLY

C# fournit la clause **finally** pour entourer les instructions qui doivent être exécutées, quoi qu'il arrive : ces instructions seront donc exécutées, si le programme se termine normalement, à l'issue du bloc **try**, ou anormalement, à l'issue d'un **catch**.

Le bloc **finally** est utile dans deux cas :

- éviter la duplication d'instructions  
des instructions devant s'exécuter à la fois en cas de fin normale et anormale trouveront leur place dans un bloc **finally**
- libérer des ressources après la levée d'une exception.

De manière générale, et sur un exemple :

```
using System;
...
try
{
    // Séquence d'instructions correspondant à la saisie de données
    // numériques, à la gestion de tableau et à l'exécution
    // d'opérations arithmétiques.
}
catch (FormatException fe)
{
    Console.WriteLine("Erreur de saisie \n" + fe.Message);
}
catch (ArithmeticException ae)
{
    Console.WriteLine("Erreur arithmétique \n" + ae.Message);
}
catch (IndexOutOfRangeException ioore)
{
    Console.WriteLine("Index de tableau non valide \n" + ioore.Message);
}
catch (Exception e)
{
    // Ce message est affiché pour toutes les autres
    // exceptions susceptibles d'être générées.
    Console.WriteLine("Erreur générale \n" + e.Message);
}
finally
{
    // instructions à exécuter quelque soit l'issue
    // du programme
}
```

## VIII.6 L'INSTRUCTION THROW

Dans le traitement d'une fonction, il peut être intéressant de générer une exception. Par exemple, dans l'utilisation d'un tableau, on peut générer exception lorsque l'index champ est inférieur à 1 ou supérieur au nombre de postes contenus dans le tableau.

Pour générer une exception, on utilise le mot-clé **throw** :

```
Console.WriteLine("Entrez un chiffre compris entre 1 et 9:");  
if (i < 0 || i >= 9)  
    throw new IndexOutOfRangeException(i + "n'est pas un choix  
    valide");
```

L'exécution séquentielle normale du programme est interrompue, et le contrôle d'exécution est transféré sur le premier bloc **catch** capable de gérer l'exception, en fonction de sa classe.

Les instructions qui suivent le **throw** ne seront jamais exécutées.

Le choix de l'exception dépend bien sûr du type de problème à traiter.

## VIII.7 EXERCICES

### Exercice :

Reprendre l'exercice 1 (manipulations de base) du chapitre VI- Les tableaux, et sécuriser le code.



## IX LES FICHIERS

### IX.1 LES OBJECTIFS

- Créer, accéder et mettre à jour un fichier texte.
- Gérer les fichiers sur disque.

### IX.2 VUE D'ENSEMBLE

Un **fichier** est une collection ordonnée et nommée d'une séquence particulière d'octets ayant un stockage persistant.

Bien que l'emploi des bases de données se soit généralisé, il est encore nécessaire de savoir accéder aux fichiers « classiques », dits fichiers « plats » (fichiers à organisations séquentielle et relative). Le cas se présente, par exemple, pour récupérer des anciens fichiers ou bien pour assurer l'interface entre deux applications.

Un fichier séquentiel est typiquement un fichier Texte, constitué de lignes de texte séparées par des paires de caractères Chr(13) + Chr(10) (Retour chariot - changement de ligne ou CR-LF) qui indiquent un changement de ligne.

Exemple :

L'utilisation d'un fichier dans un programme comportera trois phases :

- Ouverture du fichier avant la première utilisation ;  
Lors de l'ouverture, on spécifiera le mode d'utilisation du fichier :
  - Si on ouvre un fichier **pour lecture**, on ne pourra que récupérer les informations qu'il contient, sans les modifier en aucune manière.
  - Si on ouvre un fichier **pour écriture**, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes, si elles existent, seront **intégralement écrasées**.
  - Si on ouvre un fichier **pour ajout**, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra ajouter de nouveaux **enregistrements**)
- Traitement du fichier (Effacement total, positionnement selon un index, lecture, écriture, re-écriture ...)
  - Une instruction de **lecture** copie du disque vers la mémoire, les informations d'un enregistrement.
  - Une instruction d'**écriture** copie les informations en mémoire sur une place libre du support.
  - Une instruction de **réécriture** recopie les informations en mémoire à la place des dernières informations lues.
  - Une instruction de **suppression** supprime une ligne du support.
- Fermeture du fichier après la dernière utilisation

En C#, toutes les entrées/sorties de données sont vues comme des **flux** (ou flot de données). Un flux permet d'effectuer des lecture/écriture d'octets à d'un **fichier**, de la **mémoire**, du **réseau** ou de la **console**.

L'espace de noms **System.IO** contient un certain nombre de classes permettant d'effectuer des opérations sur les fichiers. Ces diverses classes peuvent être regroupées en différentes catégories :

- Les classes **Directory** et **DirectoryInfo** pour manipuler les répertoires (créer un répertoire, en afficher le contenu...),
- Les classes **File** et **FileInfo** qui fournissent des informations sur les fichiers et permettent diverses manipulations (suppression, changement de nom, copie) mais sans permettre de lire ou d'écrire des enregistrements,
- Les classes **Stream** qui permettent de lire et d'écrire dans les fichiers,
- Des classes qui fournissent des informations sur les répertoires.

### IX.3 LES CLASSES DE FLUX

La classe abstraite de base **Stream** prend en charge la lecture et l'écriture d'octets en modes synchrone et asynchrone.

Toutes les classes qui représentent des flux héritent de la classe **Stream**. La classe **Stream** et ses classes dérivées donnent une vue générique des sources de données et des référentiels, isolant ainsi le programmeur des détails propres au système d'exploitation et aux périphériques sous-jacents.

Les flux impliquent trois opérations fondamentales :

- Il est possible de **lire les flux**. La lecture est le transfert de données d'un flux vers une structure de données tel qu'un tableau d'octets
- Il est possible d'**écrire les flux**. L'écriture désigne le transfert de données d'une source de données vers un flux
- Les flux peuvent prendre en charge la **recherche**. La recherche consiste à envoyer une requête concernant la position actuelle dans un flux et à modifier cette dernière.

Selon la source de données sous-jacente ou le référentiel, les flux peuvent prendre en charge certaines de ces fonctionnalités. Par exemple, **NetworkStreams** ne prend pas en charge la recherche. Les propriétés **CanRead**, **CanWrite** et **CanSeek** de **Stream** et de ses classes dérivées déterminent les opérations prises en charge par les différents flux

## Classes de Flux

Classes	Types de flux
<b>Stream</b>	Classe de base abstraite des flux
<b>BufferedStream</b>	Flux bufférisé (données mises en cache)
<b>FileStream</b>	Classe à tout faire pour les <b>fichiers</b> : - Prend en charge les opérations de lecture et d'écriture synchrones et asynchrones. - Prend en charge l'accès aléatoire via la méthode Seek. Comme Stream <b>ne travaille que sur des tableaux d'octets</b> (byte), le développeur a souvent <b>recours à d'autres classes spécialisées</b> sur les traitements de flux : <b>StreamReader/StreamWriter, BinaryReader/BinaryWriter</b>
<b>MemoryStream</b>	Flux non bufférisé, directement accessible en mémoire (Ce flux n'a pas de magasin de sauvegarde et peut servir de mémoire tampon temporaire)
<b>TextReader</b>	Classes de base abstraites de <b>StreamReader/ StringReader</b>
<b>StreamReader</b>	Lecture de fichier texte
<b>StringReader</b>	Lecture de chaînes de caractères
<b>TextWriter</b>	Classes de base abstraites de <b>StreamWriter/ StringWriter</b>
<b>StreamWriter</b>	Ecriture de fichier texte
<b>StringWriter</b>	Ecriture de chaînes de caractères
<b>BinaryReader et BinaryWriter</b>	Lecture et écriture de flux binaires

## IX.4 UTILISATION D'UN FICHIER TEXTE

### IX.4.1 Ouverture d'un fichier

La classe **FileStream** (dérivée de **Stream**) permet d'ouvrir ou de créer un fichier en spécifiant

- le mode d'ouverture (les intentions du programmeur) au moyen de l'énumération **FileMode**
- le mode d'accès (lecture uniquement, écriture uniquement ou lecture/écriture) au moyen de l'énumération **FileAccess**
- le mode de partage (ce que peuvent faire les autres utilisateurs de ce fichier) au moyen de l'énumération **FileShare**

**FileMode** : énumération des modes d'ouverture de fichier.

Les paramètres **FileMode** vérifient si un fichier est remplacé, créé ou ouvert ou une combinaison de ces actions.

Utilisez **Open** pour ouvrir un fichier existant Pour ajouter à un fichier, utilisez **Append**. Pour tronquer un fichier ou le créer s'il n'existe pas, utilisez **Create**.

Nom de membre	Description
<b>Append</b>	Ouvre le fichier s'il existe et accède à la fin du fichier, ou crée un nouveau fichier. <b>FileMode.Append</b> peut seulement être utilisé conjointement avec <b>FileAccess.Write</b> . Toute tentative de lecture échoue et lève un <b>ArgumentException</b> .
<b>Create</b>	<i>Spécifie que le système d'exploitation doit créer un fichier. Si le fichier existe, il est remplacé. Cela nécessite <b>FileIOPermissionAccess.Write</b> et <b>FileIOPermissionAccess.Append</b>.</i> <b>System.IO.FileMode.Create</b> équivaut à demander que si le fichier n'existe pas, utilisez <b>CreateNew</b> ; sinon, utilisez <b>Truncate</b> .
<b>CreateNew</b>	Spécifie que le système d'exploitation doit créer un fichier. Cela nécessite <b>FileIOPermissionAccess.Write</b> . Si le fichier existe, un <b>IOException</b> est levé.
<b>Open</b>	Spécifie que le système d'exploitation doit ouvrir un fichier existant. La possibilité d'ouvrir le fichier dépend de la valeur spécifiée par <b>FileAccess</b> . <b>System.IO.FileNotFoundException</b> est levé si ce fichier n'existe pas.
<b>OpenOrCreate</b>	Spécifie que le système d'exploitation doit ouvrir un fichier s'il existe ; sinon, un nouveau fichier doit être créé. Si le fichier est ouvert avec <b>FileAccess.Read</b> , <b>FileIOPermissionAccess.Read</b> est requis. Si l'accès au fichier est <b>FileAccess.ReadWrite</b> et si le fichier existe, <b>FileIOPermissionAccess.Write</b> est requis. Si l'accès au fichier est <b>FileAccess.ReadWrite</b> et si le fichier n'existe pas, <b>FileIOPermissionAccess.Append</b> est requis en plus de <b>Read</b> et <b>Write</b> .
<b>Truncate</b>	Spécifie que le système d'exploitation doit ouvrir un fichier existant. Une fois ouvert, le fichier doit être tronqué de manière à ce que sa taille soit égale à zéro octet. Ceci nécessite <b>FileIOPermissionAccess.Write</b> . Toute tentative de lecture d'un fichier ouvert avec <b>Truncate</b> entraîne une exception.

#### FileAccess : énumération d'autorisation de lecture/écriture.

Nom de membre	Description	Valeur
<b>Read</b>	Accès en lecture au fichier. Les données peuvent être lues à partir de ce fichier. Combinez avec <b>Write</b> pour l'accès en lecture/écriture.	1
<b>ReadWrite</b>	Accès en lecture et en écriture au fichier. Les données peuvent être écrites dans le fichier et lues à partir de celui-ci.	3
<b>Write</b>	Accès en écriture au fichier. Les données peuvent être écrites dans le fichier. Combinez avec <b>Read</b> pour l'accès en lecture/écriture.	2

## FileShare : énumération d'autorisation de partage.

Cette énumération est généralement utilisée pour définir si deux processus peuvent simultanément lire à partir du même fichier. Par exemple, si un fichier est ouvert et si **FileShare.Read** est spécifié, les autres utilisateurs peuvent ouvrir le fichier en lecture mais pas en écriture.

Nom de membre	Description	Valeur
<b>Delete</b>	Autorise la suppression ultérieure d'un fichier.	
<b>Inheritable</b>	Crée le handle de fichier hérité par les processus enfants. Ceci n'est pas pris en charge par Win32.	16
<b>None</b>	Refuse le partage du fichier en cours. Toute demande d'ouverture du fichier (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier.	0
<b>Read</b>	Permet l'ouverture ultérieure du fichier pour la lecture. Si cet indicateur n'est pas spécifié, toute demande d'ouverture du fichier pour la lecture (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier. Cependant, si cet indicateur est spécifié, des autorisations supplémentaires peuvent toujours être nécessaires pour accéder au fichier.	1
<b>ReadWrite</b>	Permet l'ouverture ultérieure du fichier pour la lecture ou l'écriture. Si cet indicateur n'est pas spécifié, toute demande d'ouverture du fichier pour la lecture ou l'écriture (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier. Cependant, si cet indicateur est spécifié, des autorisations supplémentaires peuvent toujours être nécessaires pour accéder au fichier.	3
<b>Write</b>	Permet l'ouverture ultérieure du fichier pour l'écriture. Si cet indicateur n'est pas spécifié, toute demande d'ouverture du fichier pour l'écriture (par ce processus ou un autre) échouera jusqu'à la fermeture du fichier. Cependant, si cet indicateur est spécifié, des autorisations supplémentaires peuvent toujours être nécessaires pour accéder au fichier.	2

On peut ouvrir un fichier :

- En spécifiant son nom et son mode d'ouverture

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open);
```

Ouverture du fichier **Exemple.txt** existant (**FileMode.Open**)

- En spécifiant son nom, son mode d'ouverture et un mode d'accès

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open, FileAccess.Read);
```

Ouverture du fichier **Exemple.txt** existant (**FileMode.Open**), en lecture seule (**FileAccess.Read**).

- En spécifiant son nom, son mode d'ouverture, un mode d'accès et un mode de partage

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open, FileAccess.Read, FileShare.Read);
```

Ouverture du fichier **Exemple.txt** existant (**FileMode.Open**), en lecture seule (**FileAccess.Read**) partagée avec les autres utilisateurs (**FileShare.Read**).

### IX.4.2 Traitement des enregistrements du fichier

Les classes **StreamReader** (spécialisée dans la lecture de fichier texte) et **StreamWriter** (spécialisée dans l'écriture de fichier texte) qui dérivent des classes abstraites de base **TextReader** et **TextWriter** prennent en charge respectivement la lecture et l'écriture des flux de caractères – encodés **UTF-8\*** par défaut.

La classe **StreamReader** permet de lire un fichier texte, ligne par ligne. Elle permet également de spécifier le type d'encodage, important pour nous, qui utilisons des caractères accentués.

*\***UTF-8** (UCS Transformation Format 8 bits). UTF-8 est un codage **Unicode\*** multi-octet des caractères. Il est compatible ASCII. Chaque caractère est codé sur une suite de un à six mots de 8 bits (il n'existe pas actuellement de caractères codés avec plus de 4 mots).*

*\***Unicode** est un codage de caractères international 16 et permet d'encoder 64000 caractères.*

Pour lire le contenu du fichier texte `Exemple.txt` (supposé existant et sans lettres accentuées), on codera :

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open); ❶
StreamReader sr = new StreamReader (fs); ❷

string strLine = sr.ReadLine(); ❸
while (strLine != null) ❹
{
    // traitement de la ligne lue
    // ...
    strLine = sr.ReadLine(); ❺
}
sr.Close(); ❻
fs.Close(); ❼
```

- ❶ Ouverture du fichier `Exemple.txt` existant
- ❷ Déclaration d'un objet `StreamReader` à partir de l'objet `FileStream`
- ❸ Lecture de la première ligne du fichier
- ❹ Boucle de lecture de tous les enregistrements du fichier  
lire un fichier séquentiel de bout en bout suppose de programmer une **boucle**. ;  
Comme on sait rarement à l'avance combien d'enregistrements comporte le fichier, il faut tester la fin de fichier : si aucun enregistrement n'est lu, la méthode **ReadLine** renvoie `null`
- ❺ Lecture de l'enregistrement suivant
- ❻ Fermeture du lecteur en cours.
- ❼ Fermeture du fichier

## Membres de StreamReader

Méthodes publiques	Description
<b>Close</b>	Ferme le lecteur en cours et le flux sous-jacent.
<b>DiscardBufferedData</b>	Permet à <b>StreamReader</b> d'ignorer ses données en cours.
<b>Peek</b>	Substitué. Retourne le prochain caractère disponible, mais ne le consomme pas.
<b>Read</b>	Surchargé. Substitué. Lit le caractère ou le jeu de caractères suivant dans le flux d'entrée.
<b>Read(char [] buffer, int index, int count)</b>	Lecture d'un jeu de caractères suivant dans le flux d'entrée.
<b>ReadBlock</b>	Lit un maximum de caractères à partir du flux en cours et écrit les données dans <i>buffer</i> , en commençant par <i>index</i> .
<b>ReadLine</b>	Substitué. Lit une ligne de caractères à partir du flux en cours et retourne les données sous forme de chaîne. Retourne <b>null</b> en fin de fichier.
<b>ReadToEnd</b>	Substitué. Lit le flux entre la position actuelle et la fin du flux.

On aurait pu aussi écrire :

```
StreamReader sr = new StreamReader("Exemple.txt");

string strLine = sr.ReadLine();
while (strLine != null)
{
    // traitement de la ligne lue
    // ...
    strLine = sr.ReadLine();
}
sr.Close();
```

Le lecteur peut être créé directement en spécifiant le nom du fichier. Les caractères sont supposés être encodés **UTF-8** par défaut.

Ou encore:

```
StreamReader sr = new StreamReader("Exemple.txt");

string strLine;
while ((strLine = sr.ReadLine()) != null)
{
    // traitement de la ligne lue
    // ...
}
sr.Close();
```

Pour lire correctement un fichier comprenant des lettres accentuées (cas des fichiers créés par le bloc-notes sous Windows 9X), il faut créer l'objet **StreamReader** de la manière suivante :

```
StreamReader sr = new StreamReader("Exemple.txt", ASCIIEncoding.Default);
```

Ou

```
FileStream fs = new FileStream("Exemple.txt", FileMode.Open);  
StreamReader sr = new StreamReader (fs, ASCIIEncoding.Default);
```

La classe **StreamWriter** permet d'écrire dans un flot ; elle est symétrique de **StreamReader**.

**Exemple** : lecture du fichier texte `Exemple.txt` et écriture des lignes lues dans un nouveau fichier `Out.txt`.

```
FileStream fsI = new FileStream("Exemple.txt", FileMode.Open);  
FileStream fsO = new FileStream("Out.txt", FileMode.CreateNew); ❶  
  
StreamReader sr = new StreamReader (fsI);  
StreamWriter sw = new StreamWriter(fsO); ❷  
  
string strLine = sr.ReadLine();  
while (strLine != null)  
{  
    // traitement de la ligne lue  
    sw.WriteLine(strLine); ❸  
    strLine = sr.ReadLine();  
}  
sr.Close();  
sw.Close(); ❹  
fsI.Close();  
fsO.Close();
```

- ❶ Création et ouverture du fichier `Out.txt`
- ❷ Déclaration d'un objet `StreamWriter` `sw` à partir de l'objet `FileStream` de sortie.
- ❸ Ecriture de la ligne lue dans le flux de sortie ;
- ❹ Fermeture du lecteur.

## Membres de StreamWriter

Méthodes publiques	Description
<b>Close</b>	Ferme le lecteur en cours et le flux sous-jacent.
<b>Flush</b>	Force l'écriture sur le disque.
<b>Write</b>	Surchargé. Écrit dans le flux.
<b>WriteLine</b>	Surchargé. Écrit des données de la manière spécifiée par les paramètres surchargés, suivies d'un terminateur de ligne.



### IX.4.3 Fermeture du fichier

On a vu dans les exemples précédents l'appel systématique après traitement de la méthode **Close** sur les objets **FileStream**, **StreamReader** ou **StreamWriter**.

## IX.5 GESTION DES FICHIERS SUR DISQUE

### IX.5.1 Gestion des répertoires

La classe **Directory** fournit des méthodes de classe pour la création, le déplacement et l'énumération dans les répertoires et les sous répertoires.

La classe **DirectoryInfo** fournit des informations semblables à celles de la classe **Directory**, mais les méthodes sont des méthodes d'instance : pour les utiliser, il faut d'abord créer un objet **DirectoryInfo**.

#### Membres de Directory (extrait)

Méthodes	Description
directory <b>CreateDirectory</b> (string path) *	Crée un répertoire. L'exception <b>IOException</b> est générée si ce nom (répertoire ou fichier) existe déjà
void <b>Delete</b> (string rep) *	Supprime le répertoire de nom <i>rep</i>
bool <b>Exists</b> (string rep) *	Renvoie <b>true</b> si le répertoire de nom <i>rep</i> existe
string <b>GetCurrentDirectory</b> ()	Renvoie le nom du répertoire courant.
bool <b>Exists</b> (string rep)	Renvoie <b>true</b> si le répertoire de nom <i>rep</i> existe
string[ ] <b>GetDirectories</b> (string rep) *	Renvoie les noms des sous répertoires du répertoire <i>rep</i>
string[ ] <b>GetFiles</b> (string rep) *	Renvoie les noms des fichiers du répertoire <i>rep</i>
string[ ] <b>GetLogicalDrives</b> () *	Renvoie les noms des unités du système
void <b>Move</b> (string repS, string repD)	Déplace le répertoire <i>repS</i> vers <i>repD</i> L'exception <b>IOException</b> est générée si ce nom existe déjà
void <b>SetCurrentDirectory</b> (string repS)	Renvoie les noms des fichiers du répertoire <i>rep</i>

\* surchargé

#### Exemple 1: Affichage de tous les fichiers du répertoire courant (utilisation de Directory)

```
// affichage de tous les fichiers du répertoire courant
string curDir = Directory.GetCurrentDirectory() ; ❶
foreach (string f in Directory.GetFiles(curDir)) ❷
{
    Console.WriteLine(f);
}
```

- ❶ Recherche du répertoire courant
- ❷ Boucle d'affichage du nom des fichiers

L'avantage d'utiliser la classe **DirectoryInfo** est d'obtenir des informations supplémentaires grâce aux propriétés fournies par la classe.

### Propriétés de DirectoryInfo

Propriétés	Description
<b>Attributes</b>	Obtient ou définit des attributs de répertoire( valeur de l'énumération <b>FileAttributes</b> )
<b>CreationTime</b>	Date de création du répertoire.
<b>Exists*</b>	Indique si le répertoire existe.
<b>FullName</b>	Nom complet du répertoire(y compris l'unité).
<b>LastAccessTime</b>	Date de dernier accès au répertoire.
<b>LastWriteTime</b>	Date de dernière écriture dans le répertoire.
<b>Name</b>	Nom du répertoire.
<b>Parent</b>	Répertoire père.

**Exemple 2:** Affichage de tous les sous répertoires du répertoire courant avec leur date de création.

```
// affichage de tous les fichiers du répertoire courant
string curDir = Directory.GetCurrentDirectory() ;❶
DirectoryInfo cdi = new DirectoryInfo(curDir); ❷
foreach (DirectoryInfo di in cdi.GetDirectories())❸
{
    Console.WriteLine(fi.Name + fi.CreationTime.ToString("d"))❹;
}
```

- ❶ Recherche du répertoire courant
- ❷ Création d'une instance DirectoryInfo
- ❸ Parcours des sous répertoires
- ❹ Affichage des informations du sous-répertoire.

## IX.5.2 Gestion des fichiers

La classe **File** fournit des méthodes de classe pour créer, copier, supprimer, déplacer et ouvrir des fichiers et facilite la création d'objets **FileStream**. La classe **FileInfo** fournit des méthodes d'instance identiques.

### Membres de File (extrait)

Méthodes	Description
void <b>Copy</b> (string f1, string f2, bool bRep) *	Copie le fichier <i>f1</i> vers le fichier <i>f2</i> : si <i>bRep</i> = <b>true</b> , <i>f2</i> est remplacé.
void <b>Move</b> (string f1, string f2) ;*	Déplace le fichier <i>f1</i> vers le fichier <i>f2</i> : (le renomme dans le même répertoire)
void <b>Delete</b> (string f) ;	Supprime le fichier <i>f</i> .
bool <b>Exists</b> (string f) ;	Renvoie <b>true</b> si le fichier existe.
string <b>GetCurrentDirectory</b> () ;	Renvoie le nom du répertoire courant.
FileAttributes <b>GetAttributes</b> (string f) ;*	Renvoie les attributs du fichier (Archive, Directory, Hidden, Normal, ReadOnly, System...)
void <b>SetAttributes</b> (string f, FileAttributes) ;	Modifie les attributs du fichier.

**Exemple 1:** Copie d'un fichier sans vouloir écraser le fichier de destination s'il existe déjà.

```
try
{
    File.Copy("ancien.txt", "nouveau.txt", false) ;
}
catch (FileNotFoundException)
{
    Console.WriteLine("Le fichier source n'existe pas ") ;
}
catch (IOException)
{
    Console.WriteLine(" Le fichier de destination existe déjà");
}
```

L'avantage d'utiliser la classe **FileInfo** est semblable à l'utilisation de **DirectoryInfo** par rapport à **Directory**.

### Propriétés de FileInfo

Propriétés	Description
<b>Attributes</b>	Obtient ou définit des attributs de répertoire( valeur de l'énumération <b>FileAttributes</b> )
<b>CreationTime</b>	Date de création du fichier.
<b>Exists*</b>	Indique si le fichier existe.
<b>FullName</b>	Nom complet du fichier(unité, répertoire,nom, extension).
<b>LastAccessTime</b>	Date de dernier accès.
<b>LastWriteTime</b>	Date de dernière écriture.
<b>Length</b>	Taille du fichier en octets.
<b>Name</b>	Nom du fichier(sans le répertoire mais avec l'extension).
<b>DirectoryName</b>	Nom du répertoire du fichier

**Exemple 2:** Renommer un fichier.

```
FileInfo fi = new FileInfo("Exemple.txt ");  
if (fi.Exists) fi.MoveTo("ExempleAnc.txt");
```

L'exception **IOException** est générée si le fichier de destination existe déjà.

## IX.6 EXERCICES

### Exercice n°1 : Liste simple

A partir du fichier séquentiel à créer (*Clients.txt*) dont la structure est la suivante :

Code Client	type entier,
Nom Client	type string,
Adresse	type string,
Code postal	type string,
Ville	type string,
Telephone	type string,
Adresse mail	type string,

chaque champ étant séparé de l'autre par une virgule,

afficher une liste formatée donnant pour chaque client, son code, son nom, son téléphone et son adresse mail.

### Exercice n°2 : Liste avec rupture, fichier en sortie

A partir du fichier séquentiel (*Commandes.txt*) de structure :

Nom Client	type string,
Numéro de commande	type entier,
Code Produit	type entier,
Quantité	type entier,
Prix total	type double.

calculer les totaux par client (CA) et générer une ligne par client dans un nouveau fichier de nom CAClient.txt, de structure

Nom Client	type string,
CA	type double.

dans un répertoire de nom **Repdata**, créé à la première utilisation dans le répertoire courant. Si ce fichier existe déjà, on en gardera toujours la dernière sauvegarde de nom CAClientAncien.txt et on créera toujours un nouveau fichier.

Un jeu d'essai est fourni.

### Exercice n°3 : Liste avec ruptures

A partir du fichier séquentiel (*Commandes.txt*) de structure

calculer les sous totaux (prix) par numéro de commande, et les totaux par client, et afficher ces résultats sous la forme:

Nom Client 1	N°Commande 1	Sous Total
Nom Client 1	N°Commande 2	Sous Total
Nom Client 1		Total
Nom Client 2	N°Commande 3	Sous Total
Nom Client2	N°Commande 4	Sous Total
Nom Client2		Total

**Etablissement référent**  
*Direction de l'ingénierie Neuilly*

**Equipe de conception**  
*Groupe d'étude de la filière étude – développement*

**Remerciements :**

## Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.  
« toute représentation ou reproduction intégrale ou partielle faite sans le  
consentement de l'auteur ou de ses ayants droits ou ayants cause est  
illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction  
par un art ou un procédé quelconques. »

Date de mise à jour 19/02/2009  
afpa © Date de dépôt légal février 09



**afpa / Direction de l'Ingénierie** 13 place du Générale de Gaulle / 93108 Montreuil  
Cedex  
association nationale pour la formation professionnelle des  
adultes  
Ministère des Affaires sociales du Travail et de la  
Solidarité