



L'École Nationale de l'Intelligence
Artificielle et Digitale **Berkane**

Développement d'applications .NET

Support de cours partie I

Niveau : 2A Cycle Ingénieur

Filières : GI/IA/ROC

Elaboré et enseigné par : Prof. KADDARI Zakaria

Année universitaire

2024/2025

Table des matières

I.	Les instructions de base en C#	3
1.	Les variables :	3
1.1.	Déclaration des variables simples :	3
1.2.	L'instruction d'affectation :	3
2.	Les conversions :	3
2.1.	Les conversions entre les types compatibles :	3
2.2.	Les conversions entre les types incompatibles :	4
3.	L'instruction d'écriture :	4
4.	L'instruction de lecture :	5
5.	Les opérateurs :	5
5.1.	Les Opérateurs arithmétiques :	5
5.2.	Les opérateurs de comparaison :	6
6.	Les structures alternatives :	6
7.	Les structures répétitives :	6
8.	Les tableaux :	7
8.1.	Tableau à une dimension :	7
8.2.	Les tableaux à deux dimensions :	7
8.3.	La taille d'un tableau :	7
8.4.	Tableau de chaînes :	7
8.5.	Trier un tableau :	7
8.6.	Copier entre tableaux :	8
9.	Les énumérations :	8
10.	Les fonctions :	8
II.	Programmation orientée objet en C# :	9
1.	Définitions :	9
2.	Les classes :	10
3.	Propriété :	11
3.1.	Définition :	11
3.2.	Propriété automatique (auto-implémentée) :	12
3.3.	Propriété faisant un calcul :	12
4.	Les méthodes :	12
4.1.	Le mot-clé « this » :	13
4.2.	Passage des paramètres aux méthodes :	13
4.3.	Passage de paramètre en sortie :	14
5.	Les constructeurs :	15
5.1.	Le constructeur par défaut :	15
5.2.	Le constructeur paramétrique :	15
5.3.	Appel d'un constructeur au niveau d'un autre :	15
6.	Déclaration et instanciation d'un objet :	16
6.1.	Initialisation d'un objet à null :	16
6.2.	Initialisation des propriétés lors de l'instanciation :	17
7.	L'héritage :	17
7.1.	Syntaxe :	17
7.2.	Protected :	17
7.3.	Appel de constructeur de parent :	18
7.4.	Manipuler un enfant en tant que son parent :	18
7.5.	La classe « Object » :	18
7.6.	La substitution :	19
7.7.	Le mot clé « base » :	19
7.8.	Le mot-clé « sealed » :	20
7.9.	Masquer une méthode de parent :	20
8.	Le Polymorphisme :	20
9.	Conversion et manipulation d'objets :	21
9.1.	Le casting :	21
9.2.	Le mot clé « is » :	21
9.3.	Le mot clé « as » :	21
9.4.	Le boxing et unboxing :	22
9.5.	La comparaison des objets :	22
10.	Les interfaces :	23
10.1.	Définition :	23
10.2.	Exemple :	23

10.3.	Création d'une interface :	24
10.4.	Héritage entre interfaces :	24
11.	Les classes et les méthodes abstraites :	25
12.	Le mot clé « static » :	26
12.1.	Méthodes et variables statiques :	26
12.2.	Propriété statique :	26
12.3.	Classe statique :	27
13.	Les classes internes (nested class) :	27
14.	Le mot clé « var » :	27
15.	Les collections :	28
15.1.	Les tableaux dynamiques :	28
15.2.	La classe « List » :	28
15.3.	La classe Queue < > :	30
15.4.	La classe Dictionary :	30
16.	La gestion des exceptions :	30
16.1.	Try ... catch :	30
16.2.	Enchaîner les blocs catch :	31
16.3.	Les blocs « try ... catch » imbriqués :	31
16.4.	Le mot clé « Finally » :	31
16.5.	Lever une exception :	31
16.6.	Exception personnalisée :	32
17.	Sérialisation et désérialisation :	33
17.1.	Sérialisation des objets en XML :	33
18.	Ecriture dans un fichier texte :	34
18.1.	Lire le contenu d'un fichier texte :	34
19.	Lire et écrire dans un fichier JSON :	35

I. Les instructions de base en C#

1. Les variables :

1.1. Déclaration des variables simples :

Il s'agit des types simples supportant une seule valeur.

Syntaxe :

```
type nom ;
type nom1, nom2 ;
```

Exemple :

```
int a, b;
```

1.2. L'instruction d'affectation :

```
variable = valeur ;
```

Exemple :

<i>ENTIER</i> : byte-short-int-long	int i, j, k ; System.Int32 a = 10;
<i>DECIMAL</i>	decimal solde = 100;
<i>REEL</i> : float-double	float x=0.1,y=5.5 ;
<i>CARACTERE</i>	char c='r' ;
<i>CHAINE</i>	string nom= "Alaoui" ;
<i>BOOLEEN</i>	bool estVrai = true;

En pratique, on utilise les alias plutôt que les classes qu'ils représentent, ainsi on utilise par exemple « int » au lieu de « System.Int32 »

2. Les conversions :

2.1. Les conversions entre les types compatibles :

Le « **casting** » est simplement l'action de convertir la valeur d'un type à l'autre. Cela fonctionne pour les types qui sont compatibles entre eux (exemple :int <-> short)

Exemple 1 : (types compatibles)

```
int i = 200;
short s = (short)i;      // cast explicite.
double prix = 125.55;
int valeur = (int)prix;   // valeur vaut 125
```

Exemple 2 : (types incompatibles)

La conversion suivante provoque une erreur car les deux types sont trop différents et incompatibles entre eux.

```
string nombre = "123";
int valeur = (int)nombre;
```

2.2. Les conversions entre les types incompatibles :

La conversion entre les types incompatibles est possible si ces types sont sémantiquement proches.

La méthode Convert.ToInt32()	string A = "20"; int age = Convert.ToInt32(A); // age vaut 20
La méthode int.Parse()	string chaineAge = "20"; int age = int.Parse(chaineAge); // age vaut 20

Remarque : dans le cas où la chaîne ne représente pas un entier la conversion va échouer et le C# va renvoyer une erreur au moment de l'exécution

La méthode **int.TryParse()** :

```
string A = "ab20cd";
int age;
if (int.TryParse(A, out age))
    Console.WriteLine("La conversion est possible");
else
    Console.WriteLine("Conversion impossible");
```

Pour les autres types on utilise les méthodes : **double.TryParse()** - **Convert.ToString()** - **Convert.ToDecimal()** ...

3. L'instruction d'écriture :

```
Console.Write(Message);           //Affiche un message
Console.WriteLine(Message);       //Affiche un message + retour à la ligne
int age = 30;
Console.Write(age);               // Affiche la valeur d'une variable
Console.WriteLine("age = " + age); // Affiche : age = 30
int a=1, b=2, c=3;
Console.WriteLine("a = {0} et b = {1} et c = {2}", a, b, c); //afficher les valeurs de plusieurs
                                                                //variables
```

Exemple : Programme qui affiche un message sur la console :

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Message : . . . ");
    }
}
```

Affichage d'un caractère spécial :

Le caractère spécial « \ » permet de mettre des caractères spéciaux dans des variables de type string.

Exemple :

string phrase = "Mon prénom est \"Ahmed\""; Console.WriteLine(phrase);	//Affiche : Mon prénom est "Ahmed"
string fichier = "c:\\repertoire\\fichier.cs"; Console.WriteLine(fichier);	//Affiche : c:\\repertoire\\fichier.cs

« \n » Permet de passer à la ligne suivante.

« \t » Permet d'afficher une tabulation.

4. L'instruction de lecture :

Lire un texte	string saisie = Console.ReadLine();
Lire un char	char c = (char)Console.Read();
Lire un entier	int i=int.Parse(Console.ReadLine());
Lire un réel	double r = double.Parse(Console.ReadLine());

5. Les opérateurs :

5.1. Les Opérateurs arithmétiques :

Les opérateurs « + », « * », « / » ou encore « - » servent bien évidemment à faire les opérations mathématiques qui leur correspondent, à savoir respectivement l'addition, la multiplication, la division et la soustraction.

L'opérateur « + » peut également servir à concaténer des chaînes de caractères.

Exemple :

```
int age = 30;
int age2 = 20;
int moyenne = (age + age2) / 2;
age++;           // age contient 31           (incrément de 1)
age--;           // age contient 30           (décrément de 1)
age += 10;        // équivalent à age = age + 10   (age contient 40)
age /= 2;          // équivalent à age = age / 2 => (age contient 20)
```

Exemple 2 :

```
string nom = "Alaoui";
string prenom = "Ahmed";
string nomCompleto = nom + " " + prenom;
Console.WriteLine(nomCompleto); // affiche : Alaoui Ahmed
```

5.2. Les opérateurs de comparaison

Egalité : == Différence : != ET logique : && OU logique : Négation : !	Supérieur à : > Inférieur à : < Supérieur ou égal : >= Inférieur ou égal : <=
(a < 0 ET b > 1) OU (a > 0 ET b > 3)	(a < 0 && b > 1) (a > 0 && b > 3)

6. Les structures alternatives :

if (condition) { ... } else { ... }	if (cond1) { ... } else if (cond2) { ... } else if (cond3) { ... } else { ... }	if (condition) { ... }
--	--	----------------------------------

Exemple : `if ((a == b && c == d) || e != f)`

```
switch (expression){
    case valeur1: action1;
    break;
    case valeur2: action2;
    break;
    ...
    default : action;
    break;
}
```

7. Les structures répétitives :

for(i= val_Initiale; condition; i=i+...) { ... }	while (condition) { ... }
foreach(type val in tab) { ... }	do{ ... } while (condition);

Remarque : Au moment où l'on rentre dans la boucle « **foreach** » pour parcourir un tableau (ou une liste), alors la variable d'itération utilisée ne permettra pas de modifier le contenu de l'élément parcouru (tableau ou liste).

Les instructions « break » et « continue »

break : permet de sortir de la boucle		continue : permet de passer à l'itération suivante.	
<pre>for (int i = 0; i < 20; i++) { if (i % 2 != 0) { break; } Console.WriteLine(i); }</pre>	Résultat: ?	<pre>for (int i = 0; i < 20; i++) { if (i % 2 == 0) { continue; } Console.WriteLine(i); }</pre>	Résultat: ?

8. Les tableaux :

8.1. Tableau à une dimension :

Déclaration : `type[] nom_tableau = new type[taille] ;`

Exemples :

```
Float[] Note = new float[20] ;
Note [0] = 10 ;
Note [1] = 12 ;
//...
Note [19] = 11 ;
```

8.2. Les tableaux à deux dimensions

nom_tableau : désigne le nom du tableau
i : désigne le nombre de lignes du tableau
j : désigne le nombre de colonnes du tableau
Type : représente le type des éléments du tableau

```
type[,] nom_tableau = new type[i,j];
t[0, 0] = 0; //Affectation de la valeur 0 à la
//cellule (0,0)
```

On peut déclarer aussi des tableaux à plusieurs dimensions (3 dimensions, 4, 5, ...)

```
int[, , , , , , , , , , ] t = new int[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2];
Console.WriteLine(t[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]);
```

8.3. La taille d'un tableau :

```
int[] t={1,2,3}; //déclaration et initialisation d'un tableau d'entiers
float[] notes=new float[20]; //déclaration d'un tableau de réels
//Afficher la taille d'un tableau
Console.WriteLine(t.Length); //affiche: 3
Console.WriteLine(notes.Length); //affiche: 20
```

8.4. Tableau de chaînes :

<pre>string[] jours = new string[] { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };</pre>	Equivalent à :	<pre>string[] jours = new string[7]; jours[0] = "Lundi"; jours[1] = "Mardi"; jours[2] = "Mercredi"; jours[3] = "Jeudi"; jours[4] = "Vendredi"; jours[5] = "Samedi"; jours[6] = "Dimanche";</pre>
---	----------------	--

8.5. Trier un tableau :

La méthode « `Array.Sort(...)` » permet de trier un tableau.

Exemple : trier et afficher le tableau « jours » précédent :

```
Array.Sort(jours);
for (int i = 0; i < jours.Length; i++){
    Console.WriteLine(jours[i]);
}
```

8.6. Copiage entre tableaux :

```
int[] t = new int[4]{1,2,0,-1};
int[] t2 = new int[2];
t.CopyTo(t2, 2); //Copie de tableaux à partir de l'index 2
```

9. Les énumérations

Une énumération permet d'avoir une liste exhaustive et fixée de valeurs constantes.

```
enum Jours
{
    Lundi,    //ou Lundi=0,
    Mardi,    //ou Mardi = 1,
    Mercredi,
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}
```

Création de l'énumération jours en dehors de la méthode main() c.-à-d. : définition de type jours qui contient des valeurs constantes.

Le fait de définir une telle énumération revient en fait à enrichir les types que nous avons à notre disposition, comme les entiers ou les chaînes de caractères (int ou string). Ce nouveau type s'appelle « Jours ».

Exemple :

```
Jours j = Jours.Lundi;
Console.WriteLine(j);
if (j == Jours.Dimanche) //tester la valeur
                        //d'une énumération
```

Accéder à l'élément 'Lundi' de l'énumération.
//Affiche : Lundi
L'écriture : **if (j == Dimanche)** est incorrecte

10. Les fonctions :

Syntaxe :

```
type nom_methode (type Argument1, type Argument2,...) //Signature de la méthode
{ //Bloc de code de la méthode return
    Valeur_Renvoyée ;
}
```

Exemple : La méthode **main()** est le point d'entrée de l'application :

```
static void DireBonjour(string prenom, int age){
    Console.WriteLine("Bonjour " + prenom);
    Console.WriteLine("Vous avez " + age + " ans");
}
```

II. Programmation orientée objet en C# :

La POO est une façon de développer une application informatique sous la forme d'objets Les avantages de la POO :

- La POO est proche de la réalité
- Les personnes non-techniques pourront comprendre et éventuellement participer à cette modélisation
- La POO permet de découper un gros problème en plus petits afin de le résoudre plus facilement
- Elle permet d'améliorer la maintenabilité.
- La réutilisabilité : des objets peuvent être réutilisés ou même étendus grâce à l'héritage

1. Définitions :

Objet	Un objet est un concept abstrait caractérisé par un ensemble de propriétés et d'actions
Classe	Une classe est une manière de représenter la structure d'un objet.
Encapsulation	Permet de protéger l'information contenue dans notre objet et de le rendre manipulable uniquement par ses actions ou propriétés.
Héritage	C'est la transmission de certaines caractéristiques d'un objet dit « père » à un autre objet dit « fils » : l'objet fils hérite le comportement de l'objet père.
Spécialisation	c.-à-d. l'héritage, ainsi l'objet fils est une spécialisation de l'objet père (il dérive de l'objet père). <u>Exemple</u> : Si l'objet « animal » dérive de l'objet « être_vivant » alors : L'objet père est : « être_vivant ». L'objet fils est : « animal »
Polymorphisme	C'est la capacité pour un objet de faire une même action avec différents types d'intervenants (prendre plusieurs formes). C'est ce qu'on appelle aussi la surcharge des méthodes.
Substitution	C'est la capacité d'un objet fils à redéfinir des caractéristiques ou des actions d'un objet père (redéfinir un comportement hérité si celui de père ne convient pas au fils). Exemple : Les objets « homme » et « dauphin » se déplacent différemment. Chaque fils de l'objet « être_vivant » donne sa définition adéquate à l'action « se déplacer » héritée de parent.
Interface	L'interface est un contrat que s'engage à respecter un objet. Il indique en général un comportement. les interfaces ne fournissent qu'un contrat, elles ne fournissent pas d'implémentation (code C#). c'est l'objet implémentant une interface qui sera responsable de coder la fonctionnalité associée au contrat.
Remarques	<ul style="list-style-type: none"> • Le C# ne permet pas ce qu'on appelle l'héritage multiple • Contrairement à l'héritage, un objet est capable d'implémenter plusieurs interfaces.

2. Les classes :

Une classe est une structure qui permet de regrouper des données et des opérations d'une même nature. Par exemple, une classe *Personne* pourrait regrouper les informations personnelles d'une personne comme son nom, son âge, son salaire, etc.

Une classe permet donc de créer de nouveaux types plus complexes.

Une classe peut contenir plusieurs éléments tels que :

- Données membres (« variables »)
- Propriétés (« méthodes spéciales »)
- Constructeurs (« méthodes spéciales »)
- Méthodes

Une classe est définie sur un fichier d'extension « .cs »

Un seul fichier peut contenir la définition de plusieurs classes

Syntaxe :

Définition d'une classe	Déclarer une variable (objet) dont le type est une classe :
<pre>class NomDeLaClasse { // Données et opérations ici }</pre>	<pre>NomDeLaClasse objet1;</pre>

Encapsulation :

Elle consiste à cacher et protéger l'information contenue dans un objet et d'offrir des méthodes pour manipuler les données. Un objet est donc vu comme une « boîte noire » dont on connaît les fonctionnalités, mais pas les détails d'implémentation.

Les indicateurs de visibilité permettant d'encapsuler le contenu d'un objet sont :

- ☐ **Public** : Accès non restreint
- ☐ **Protected** : Accès depuis la même classe ou depuis une classe dérivée
- ☐ **Private** : Accès uniquement depuis la même classe
- ☐ **Internal** : Accès restreint à la même assembly
- ☐ **protected internal** : Accès restreint à la même assembly ou depuis une classe dérivée

Donnée membre :

C'est une variable propre à une classe permettant de stocker l'état d'un objet et qui n'est pas souvent accessibles à l'extérieur de la classe.

Syntaxe de définition d'une donnée membre :

```

class NomDeLaClasse
{
    // donnée membre non accessible en dehors de la
    // classe private int _nomDeLaDonneeMembrePrivee;
    // donnée membre accessible en dehors de la classe
    public int _nomDeLaDonneeMembrePublique;
    ...
}

```

Il est déconseillé de rendre une donnée membre publique. Cela va contre le principe fondamental de l'encapsulation en programmation orientée objet. Il est donc préférable d'utiliser des propriétés (ou des méthodes) pour modifier l'état d'un objet.

3. Propriété :**3.1. Définition :**

C'est une méthode spéciale se comportant comme une donnée membre, et permettant d'implémenter le concept d'accesseurs et de mutateurs.

Accesneur : C'est une sous-routine qui permet d'accéder à l'état d'un objet sans donner un accès direct aux données.

Mutateur : C'est la variante qui permet d'assigner une valeur.

Il ne faut pas utiliser les parenthèses lors de l'appel d'une propriété, par opposition aux appels de méthodes. De plus, les propriétés permettent l'affectation avec l'instruction =.

Syntaxe	Exemple
<pre> private type varPrivée; public type MaProp { // la valeur de retour de la propriété get { return varPrivée; } // l'affectation : valeur permet d'obtenir la //valeur à assigner set { varPrivée = value; } } </pre>	<pre> class Personne { int _age; //private par default public int Age { get { return _age; } set { _age = value; } } } </pre>

Pour rendre la propriété en **lecture seule** il faut omettre le bloc **set**

Pour rendre la propriété en **écriture seule** il faut omettre le bloc **get**

Une propriété « **public** » peut être accessible en dehors de la classe

Une propriété « **private** » ne peut pas être accessible en dehors de classe

Les propriétés sont par défaut privées « **private** ».

Une propriété peut être publique en lecture mais privée en écriture

```

class voiture
{
    private bool _enPanne;
    public bool estEnPanne
    {
        public get { return _enPanne; }
        private set { _enPanne = value; }
    }
}

```

3.2. Propriété automatique (auto-implémentée) :

Elle simplifie grandement l'écriture

```
Class MaClasse
{
    // aucune donnée membre à déclarer, cela est fait implicitement.
    public type _MaPropriété {get ; set ;}    //Propriété automatique
}
```

Nous pouvons accélérer l'écriture des propriétés en utilisant ce qu'on appelle des « **snippets** » qui sont des extraits de code (prop, propfull,...) et la touche tabulation.

Propriété automatique avec affectation privée :

```
public type _MaPropriété {get ; private set ;}    // propriété d'écriture en privée
```

Ce n'est pas intéressant de toujours mettre la propriété auto-implémentée en « **private** ».

3.3. Propriété faisant un calcul :

```
private float Longueur; //donnée membre privée
private float Largeur;  //donnée membre privée
public float Surface    //propriété qui calcule et renvoie la surface de rectangle
{
    get
    {
        return Longueur * Largeur;
    }
    //Remarquer l'absence de bloc set car on ne peut pas affecter une valeur à la surface
}
```

4. Les méthodes :

Elles accèdent aux données membres et permettent de modifier et manipuler un objet.

Une méthode n'accédant pas (au moins) à une donnée membre n'a aucun intérêt et ne devrait pas se trouver à l'intérieur de la classe.

Méthode « private » non accessible en dehors de la classe	Méthode « public » accessible en dehors de la classe
private type NomDeLaMethode(...) {...}	public type NomDeLaMethode(...) {...}

Par défaut, les méthodes sont privées.

Exemple :

```
class cercle
{
    float _Rayon;
    double surface()
    {
        double s = Math.PI * _Rayon * _Rayon;
        return s;
    }
}
```

Accès aux données d'un objet :

4.1. Le mot-clé « this » :

Lorsque nous écrivons le code d'une classe, le mot-clé « **this** » représente l'objet dans lequel nous nous trouvons (l'objet en cours). Ainsi, pour accéder à une variable de la classe ou éventuellement une méthode, nous pouvons les préfixer par « **this.** ».

Exemple :

```
void afficherComplexe()
{
    Console.WriteLine("{0} + {1}i", this.Re, this.Im);    //Accès aux variables de la classe
}
void calcul()
{
    //faire des calculs sur l'objet courant
    this.afficherComplexe();                                //Appel d'une méthode de la classe
}
```

Le mot clé « **this** » est facultatif, on peut utiliser uniquement le « . » pour accéder aux données membre, propriétés et méthodes d'un objet.

Dans le cas où le nom d'un paramètre est identique à celui d'une donnée membre, alors on doit utiliser « **this** » pour éviter l'ambiguïté.

```
class complexe
{
    public float Re;
    public float Im;
    void ModifierPartieRéelle(float Re)
    {
        this.Re = Re;
    }
}
```

4.2. Passage des paramètres aux méthodes :

Lorsque nous passons des types valeur (entier, réel, string, ...) en paramètre d'une méthode, nous utilisons un passage de **paramètre par valeur**.

Pour modifier la valeur d'un paramètre de type simple on utilise le mot clé « **ref** ». (C'est le **passage par référence**)

Le fait de passer un objet (type référence) à une méthode équivaut à passer la référence de l'objet en paramètres (passage par référence).

Exemple :

```
void méthode(int a,ref string s, Personne p)
{
    a += a;                //Changement en local
    s += s;                //Changement sur la variable source
    p.age = 22 ;           //Changement sur la variable source
}

int A = 1;                //A=1
string S = "rr";          //S="rr"
Personne P = new Personne();
P.age = 12;               //P.age = 12
méthode(A,ref S, P);      //Résultat : A=1  mais  S="rrrr"      et      P.age = 22
```

Exemple 2 : Passage d'objet en paramètre avec le mot clé « **ref** »

```
public void méthode(ref Personne p1, Personne p2)
{
```

<pre> p1 = null; // la référence de « p1 » est passée à nulle p2 = null; // une copie de la variable contenant la référence est passée à nulle } </pre>
Méthode(ref p1,p2) ; //Résultat : p1 égale à nullmais p2 est différent de null

4.3. Passage de paramètre en sortie :

Il permet d'initialiser la variable passée en paramètre.

Exemple :

<pre> public void somme(int a, int b, out int sum) { sum = a + b; } </pre>
<pre> int som; somme(2, 3,out som); //Résultat : som=5 </pre>

5. Les constructeurs :

Un constructeur est une méthode (public) spéciale qui permet d'indiquer ce qui se produit lorsqu'un objet est instancié. Typiquement, un constructeur initialise les données membres de l'objet.

Un constructeur ne possède pas de type de retour et son nom est obligatoirement celui de la classe. Une classe peut avoir plusieurs constructeurs (voir la notion de Polymorphisme).

5.1. Le constructeur par défaut :

C'est un constructeur sans paramètres.

Remarque : Si aucun constructeur n'est spécifié, un constructeur par défaut est fourni par le compilateur.

5.2. Le constructeur paramétrique.

C'est un constructeur qui possède des paramètres et qui permet d'initialiser les variables de la classe.

Syntaxe :

```
class MaClasse
{
    //Constructeur par défaut
    public MaClasse()
    {
        // Code qui doit être exécuté lors de l'instanciation d'un objet par le constructeur
        par //défaut
    }
    //Constructeur paramétrique
    public MaClasse(type param1, type param2, ...)
    {
        // Code qui doit être exécuté lors de l'instanciation d'un objet par le constructeur
        //paramétrique
    }
    ...
}
```

Remarque : Si une classe possède un ou plusieurs constructeurs paramétriques, le constructeur par défaut n'existe plus automatiquement. Si un constructeur par défaut est nécessaire, il faut donc le définir explicitement.

5.3. Appel d'un constructeur au niveau d'un autre :

```
class Personne
{
    public string nom { get; set; }
    public int age { get; set; }
    public Personne(){}
    public Personne(int a){ age=a; }
    //L'appel de constructeur Personne(int) permet d'éviter d'écrire du code redondant
    public Personne(int a, string n) : this(a)
    {
        // Code ici
    }
}
```

6. Déclaration et instanciation d'un objet :

Une variable doit être déclarée et instanciée avant d'être utilisée.

Nous utilisons le mot clé « **new** » pour créer une instance d'un objet. C'est lui qui permet la création d'un objet. Il appelle le constructeur correspondant.

Déclaration d'un objet de la classe MaClasse	MaClasse c ;
Instanciation de la variable (objet) avec le constructeur par défaut.	c = new MaClasse() ;
Instanciation avec le constructeur paramétrique.	c = new MaClasse(param1, param2) ;
Déclaration + Instanciation	MaClasse c2 = new MaClasse() ;

Exemple :

```
namespace ConsoleApplication2
{
    class complexe
    {
        public double Re { get; set; }
        public double Im { get; set; }
        public complexe(double r, double i) { Re = r; Im = i; }
        public complexe somme(complexe z1, complexe z2)
        {
            return new complexe(z1.Re + z2.Re, z1.Im + z2.Im);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            complexe z1= new complexe(1,1);//z1=1+i
            complexe z2= new complexe(0,1);//z2=i
            z2 = z2.somme(z1, z2);    //z2=z1+z2=1+2*i
        }
    }
}
```

6.1. Initialisation d'un objet à null :

```
complexe z = null;           //objet null non utilisable
if(z== null)    ...         // tester la nullité d'un objet
z= new compexe() ;          //création d'une référence d'un objet
```

6.2. Initialisation des propriétés lors de l'instanciation :

```
Rectangle r = new Rectangle {Longueur=10, Largeur= 12 };
```

7. L'héritage :

Il permet la spécialisation des classes pour améliorer le découpage et faciliter la réutilisation du code.

Exemple : La classe « Stagiaire » est une spécialisation de la classe « Personne »

Le stagiaire est une personne qui possède des informations que certains n'ont pas : Groupe, Filière....

Dans ce cas la classe stagiaire possède aussi les informations de la classe personne (données membre, propriétés, constructeurs, méthodes).

7.1. Syntaxe :

```
class Parent : Enfant
```

```
{
    ...
}
```

Exemple :

```
class Personne
{
    public int Age { get; set; }
    public string Nom { get; set; }
}
class Stagiaire : Personne
{
    public string Groupe { get; set; }
    public string Filière { get; set; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Stagiaire st = new Stagiaire();
        st.Age = 19;
        st.Filière = "tdi";
        st.Nom = "El Kamali";
        st.Groupe = "101";
    }
}
```

- La classe « Stagiaire » est un enfant de « Personne ».
- La classe « Personne » est le parent de « Stagiaire ».

Remarque : Les méthodes de l'enfant n'accèdent pas aux informations privées de parent !

7.2. Protected :

C'est un niveau d'accessibilité qui se comporte comme **private** mais qui permet aux méthodes de la classe enfant (et ses potentiels enfants) d'accéder aux informations de parent sans restriction.

Exemple :

```
class Parent
{
    public int donnée_Public {get;set;}
    private int donnée_Private {get;set;}
    protected int donnée_Protected {get;set;}
}
class Enfant : Parent
{
    void Méthode_Test()
    {
        donnée_Public = 10; // OK
        donnée_Protected = 10; // OK
        donnée_Private = 0; // KO
    }
}
```

```
class AutreClasse
{
    void Méthode_Test()
    {
        Enfant e = new Enfant();
        e.donnée_Public = 20; // OK
        donnée_Protected = 10; // KO
        donnée_Private = 0; // KO
    }
}
```

7.3. Appel de constructeur de parent :

Lorsqu'un enfant est instancié, alors le code du constructeur du parent est d'abord exécuté, puis celui de l'enfant.

Le mot-clé « base » :

Il permet de choisir le constructeur du parent qui doit être appelé lors de l'instanciation d'un objet enfant.

Syntaxe :

```
class Parent
{
    public Parent(int i)
    {
        . . .
    }
}
```

```
class Enfant : Parent
{
    public Enfant() : base(10)
    {
        . . .
    }
}
```

7.4. Manipuler un enfant en tant que son parent :

La classe « Chien » hérite de la classe « Animal »

```
Animal animal = new Chien { NombreDePattes = 4 }; //Ecriture correcte
```

Remarque : Si aucun constructeur n'est défini au niveau de la classe enfant, alors c'est le constructeur du parent qui sera utilisé lors de la création d'un objet de la classe « enfant ».

```
class animal
{
    public animal() { Console.WriteLine("Construction d'un objet \"animal\"); }
}
class chien : animal
{ }
class Program
{
    static void Main(string[] args)
    {
        chien c = new chien(); //Affiche: Construction d'un objet "animal"
        Console.Read();
    }
}
```

7.5. La classe « Object » :

L'héritage des classes de « **Object** » est automatique, c.-à-d. toutes les classes qu'on définit dérivent de cette classe.

On utilise généralement son alias « **object** »

Parmi les méthodes offerte par « **object** » la méthode **ToString()** qui renvoie pour les types référence le nom de l'espace de nom suivi du nom de sa classe, et pour les types valeur, la valeur du type.

```
chien c = new chien();
int i = 110;
Console.WriteLine(c.ToString()); //Affiche: ConsoleApplication3.chien
Console.WriteLine(i.ToString()); //Affiche: 110
```

```
int A= 85;
string chaine = "A = " + A ;           //Le compilateur demande la représentation de l'entier A sous
                                         //forme de chaine.
                                         //L'instruction est équivalente à :
string chaine = "A = " + A.ToString() ;
```

7.6. La substitution :

Appelé aussi : spécialisation ou redéfinition.

Elle permet de redéfinir un comportement que l'objet a hérité afin que ce comportement corresponde aux besoins de l'objet fils.

Pour pouvoir créer une méthode qui remplace une autre, il faut que la méthode à remplacer soit candidate à la substitution : elle doit être préfixée par le mot-clé « **virtual** », ainsi les classes enfants peuvent redéfinir uniquement les méthodes virtuelles.

Pour redéfinir une méthode on utilise le mot clé « **override** » qui signifie que nous souhaitons substituer la méthode existante afin de remplacer son comportement,

Exemple : redéfinition de la méthode **ToString()** de la classe **object**:

```
class point
{
    public float x { get; set; }
    public float y { get; set; }
    public override string ToString()
    {
        //return base.ToString();
        return "(" + x + "," + y + ")";
    }
}

static void Main(string[] args)
{
    point p = new point();
    Console.WriteLine(p.ToString()); //Affiche (0,0)
    Console.Read();
}
```

Exemple 2 :

```
class parent{
    public virtual void methode()
    {
        //définition de la méthode par le parent
    }
}
class enfant:parent
{
    public override void methode()
    {
        //redéfinition de la méthode par l'enfant
    }
}
```

7.7. Le mot clé « base » :

On utilise ce mot clé pour qu'une méthode du fils fasse appel à une méthode de la classe mère.

```
class enfant : parent
{
    public void methodeFils()
    {
        base.methodeParent();
    }
}
```

Il est possible que la classe fille redéfinisse une méthode tout en conservant la fonctionnalité de la méthode offerte par classe mère. Cela se passe en utilisant le mot clé « **base** » qui représente la classe mère

```
class enfant:parent
{
    public override void methode()
    {
        //redéfinition de la méthode par l'enfant
        base.methode(); //Appel de la méthode de la classe mère
        //Autres traitements ...
    }
}
```

7.8. Le mot-clé « sealed ».

Il permet d'**empêcher une classe d'être héritée**.

Exemple : On ne peut pas créer une classe qui dérive de la classe .NET « String ».

Pour créer une classe scellée il suffit de précéder le mot-clé class du mot-clé «

```
sealed » public sealed class ClasseScellée //Impossible de dériver de cette classe
{ ... }
```

Le mot clé « sealed » peut être utilisé aussi pour **empêcher la substitution d'une méthode :**

```
public sealed void méthodeScellée(..) //Les classe dérivées ne peuvent pas redéfinir cette
{ ... } //Méthode
```

7.9. Masquer une méthode de parent :

```
class parent
{
    public void méthode(){}
}
class enfant : parent
{
    public new void méthode() //C'est pas une substitution, c'est une autre méthode qui a //le
    { } //même nom que la méthode de parent (devenue masquée).
}
```

8. Le Polymorphisme

C'est capacité d'une classe à effectuer la même action sur différents types d'intervenants, c'est ce qu'on appelle aussi la **surcharge** des méthodes.

Par exemple la méthodes prédéfinie **WriteLine()** effectue l'action d'affichage quel que soit le type des paramètres : entier, chaîne, réel, objet ... C'est une méthode surchargée (elle accepte divers signatures).

Les méthodes surchargées doivent se différencier avec les paramètres d'entrée.

Exemple :

```
public int Addition(int a, int b, int c) //méthode qui accepte les entiers
{
    return a + b + c + d;
}
public double Addition(double a, double b) //méthode qui accepte les réels
{
```

```

        return a + b;
    }
    public int Addition(Personne a, Personne b)           //méthode qui accepte des objets
    {
        return a.Age + b.Age;
    }

```

Lorsque nous avons plusieurs signatures possibles pour la même méthode, la complétion automatique nous propose alors plusieurs alternatives.

9. Conversion et manipulation d'objets :

9.1. Le casting :

Il permet de faire la conversion entre deux objets dans le cas où l'un est une sorte (hérite) de l'autre. Si par exemple la classe « Chien » hérite de la classe « Animal », alors un objet « chien » peut être converti en « animal » :

```

Chien ch1 = new Chien();
Animal animal = (Animal)ch1;    //Le casting

```

Exemple : Ajout des objets de type « chien » à une liste d'objets de type « animal » :

```

List<Animal> animaux = new List<Animal>();    //La liste peut contenir aussi des objets qui
                                              //dériver de la classe « animal »
Chien chien1, chien2 = new Chien();          //La classe « chien » hérite de « animal »
animaux.Add((Animal)chien1);                //Conversion en utilisant le cast
animaux.Add(chien2);                        // conversion implicite,

```

9.2. Le mot clé « is » :

Il vérifie si une variable correspond à un objet (à une instance d'une classe).

Exemple : Parcourir la liste des objets personnes en affichant la moyenne s'il s'agit d'un étudiant ou le grade s'il s'agit d'un professeur.

```

foreach (Personne p in personnes)           //personnes est une liste
{
    if (p is Etudiant)                      //si la personne est un étudiant
    {
        Etudiant e = (Etudiant)p;          //conversion explicite de « p » en objet étudiant
        e.AfficherMoyenne();
    }
    if (p is Professeur)
    {
        Professeur f = (Professeur)p;
        f.AfficherGrade();
    }
}

```

9.3. Le mot clé « as » :

Le mot clé « as » est employé pour faire un **cast dynamique** : il vérifie si l'objet est bien convertible, alors il fait un cast explicite pour renvoyer le résultat de la conversion, sinon, il renvoie une référence nulle.

Exemple :

```

foreach (Personne p in personnes)           //personnes est une liste
{
    Etudiant e = p as Etudiant;
    if (e != null)

```

```

    {
        e.AfficherMoyenne();
    }
    Professeur f = p as Professeur;
    if (p != null)
    {
        f.AfficherGrade();
    }
}

```

9.4. Le boxing et unboxing :

Le « boxing » est la conversion d'un type valeur (int, float, string, ..) en type référence.

```

int i = 7;
object o = i; // boxing : « o » est une référence vers une copie de « i »

```

Le « unboxing » est le contraire de « boxing » :

```

int i = 5;
object o = i;      // boxing
int j = (int)o;    // unboxing : cast explicite

```

9.5. La comparaison des objets :

Il n'est pas possible de comparer les objets d'une manière similaire à la comparaison des types valeur. En effet, les variables qui représentent des instances d'objet contiennent en fait une référence vers l'instance. Comparer des références d'objets n'a pas de sens !

L'utilisation de l'opérateur « == » permet de vérifier si deux références pointent vers le même objet.

La méthode Equal() :

C'est une méthode virtuelle définie sur la classe « Object » qui possède un comportement par défaut consistant à comparer les références des objets et les valeurs des types valeur.

On peut utiliser le mot clé « override » pour redéfinir cette méthode pour avoir un comportement approprié à notre classe.

Exemple : deux personnes sont égales s'elles ont les mêmes informations (nom, prénom, cin)

```

class Personne
{
    public string nom { get; set; }
    public string prenom { get; set; }
    public string cin { get; set; }
    public override bool Equals(object obj)
    {
        Personne p = obj as Personne;      // cast dynamique
        if (p != null)
        {
            if (this.nom == p.nom &&
                this.prenom == p.prenom && this.cin == p.cin)
                return true;
        }
        return false;
    }
}

static void Main(string[] args)
{
    Personne p1 = new Personne();
    Personne p2 = new Personne();
    bool b = p1.Equals(p2);
}

```

```

        Console.WriteLine(b);           //affiche true
    }

```

10. Les interfaces :

10.1. Définition :

Une interface est un contrat que s'engage à respecter un objet (la classe qui implémente l'interface). Il est conventionnel de démarrer le nom d'une interface par un I majuscule. C'est le cas de toutes les interfaces du framework .NET.

10.2. Exemple :

L'interface « **Comparable** » permet de définir un contrat de méthodes destinées à la prise en charge de la comparaison entre deux instances d'un objet. Une fois ces méthodes implémentées, on sera capable de dire qu'une instance est supérieure ou inférieure à une autre.

Syntaxe : Pour qu'une classe implémente une interface, on utilise les deux points suivis du type de l'interface (c'est la même syntaxe que celle d'héritage).

Exemple : Comment peut-on comparer deux personnes ? On peut utiliser l'âge (ou bien l'ordre alphabétique des noms ou ...).

```

class Personne:Comparable           //La classe Personne implémente l'interface Comparable
{
    //Ainsi on doit respecter le contrat en définissant la
    // méthode CompareTo

    public string nomComplet { get; set; }
    public int age { get; set; }
    public int CompareTo(object obj)           //Définition de la méthode de tri
    {
        Personne p = obj as Personne;
        if (this.age > p.age) return 1;
        else if (this.age < p.age) return -1;
        else return 0;
    }
}

```

Ainsi on peut comparer des instances de la classe personne.

```

static void Main(string[] args)
{
    Personne p0 = new Personne { age = 25 };
    Personne p1 = new Personne { age = 29 };
    Console.Write(p0.CompareTo(p1));           //Affiche: -1
}

```

On peut aussi trier des tableaux de personnes.

```

static void Main(string[] args)
{
    Personne[] Tab = new Personne[4];
    for (int i = 0; i < 4; i++) Tab[i] = new Personne();
    Tab[0].age = 25;
    Tab[1].age = 30;
    Tab[2].age = 24;
    Tab[3].age = 29;
    Array.Sort(Tab);           //Trier le tableau
    foreach (Personne p in Tab)

```

```

        Console.WriteLine(p.age + " <");           //Affiche: 24<25<29<30
    }

```

10.3. Création d'une interface :

Pour définir une interface on utilise le mot-clé **interface** à la place de **class**.

Syntaxe :

```

interface I_Nom           //les interfaces doivent commencer par un i majuscule,
                           //c'est une convention
{
    int nbr { get; set; }
    int methode1();        //Pas d'implémentation des méthodes
    void methode2();       //Pas d'indicateur de visibilité
}

```

Les classes qui choisiront d'implémenter cette interface seront obligés d'avoir une propriété entière « **nbr** » et une méthode qui renvoie un entier (**methode1()**) et une autre qui ne renvoie rien (**methode2()**).

```

class C : I_Nom           //la classe « C » implémente l'interface « I_Nom »
{
    public int nbr { get; set; }           //Implémentation des données membres
    public int methode1() { return 1; }    //Implémentation des méthodes
    public void methode2() { }
}

```

10.4. Héritage entre interfaces :

Les interfaces peuvent hériter entre elles, comme c'est le cas avec les classes.

```

interface I_Parent
{
    void method_Parent();
}
interface I_Enfant : I_Parent
{
    int methode_Enfant();
}

```

Ainsi chaque classe qui implémente l'interface « I_Enfant » doit définir les méthodes de l'enfant « I_Enfant » et aussi de parent « I_Parent »

```

class C : I_Enfant
{
    public int prop { get; set; }           //Implémentation des propriétés et des
    public int methode_Enfant()             méthodes //des interfaces I_Enfant et I_Parent
    { ...
    }
    public void method_Parent()
    { ...
    }
}

```

Une classe ne peut hériter que d'une seule classe mais elle peut implémenter plusieurs interfaces :

```

interface I_1 { void methode_I_1();}
interface I_2 { void methode_I_2();}
interface I_3 { void methode_I_3();}
class C:I_1,I_2,I_3

```

```
{
    public void methode_I_1(){ }
    public void methode_I_2(){ }
    public void methode_I_3(){ }
}
```

11. Les classes et les méthodes abstraites :

Une **classe abstraite** est une classe particulière qui ne peut pas être instanciée. Concrètement, cela veut dire que nous ne pourrions pas utiliser l'opérateur new.

Une **méthode abstraite** est une méthode qui ne contient pas d'implémentation, c'est-à-dire pas de code.

Remarque :

- Si une classe possède une méthode abstraite, alors la classe doit absolument être abstraite.
- L'inverse n'est pas vrai, une classe abstraite peut posséder des méthodes non abstraites.
- Pour être utilisables, les classes abstraites doivent être héritées et les méthodes redéfinies. On utilise le mot clé « **abstract** » pour définir les classes et les méthodes abstraites.

Exemple :

```
public abstract class EtreVivant           //La classe est abstraite
{
    public abstract void SeDeplacer();    //chaque enfant se déplace à sa manière
}
```

On est obligé de définir la méthode abstraite SeDeplacer() dans chaque classe fille de la classe « EtreVivant ».

```
class Personne : EtreVivant
{
    public override void SeDeplacer()
    {
        //Redéfinition de la méthode
    }
}
```

Une classe abstraite peut contenir des méthodes dont l'implémentation ne dépendra pas des enfants, ces méthodes ne doivent pas être déclarées abstraites (elles doivent avoir une implémentation).

```
abstract class EtreVivant
{
    public DateTime dateNaissance { get; set; }
    public abstract void SeDeplacer();
    public int GetAge()                //l'âge se calcule indépendamment
    {                                  //de l'enfant
    }
```

```

        return DateTime.Now.Year - dateNaissance.Year;
    }
}

```

12. Le mot clé « static » :

12.1. Méthodes et variables statiques :

Une (méthode / variable) statique est une (méthode / variable) accessible en dehors de tout objet (instance de classe), mais n'est pas accessible pour les objets.

Exemple : Pour appeler la méthode « **WriteLine** » on n'est pas obligé de créer une instance de la classe « **Console** », il suffit d'écrire : **Console.WriteLine(...)**.

Pour rendre une variable ou méthode statique on utilise le mot clé « **static** » :

```

class Nomclasse
{
    public static int nbr;    //variable statique
    ....
    public static void methode_static()    //méthode statique
    { ... }
}

```

Utilisation :

```

Nomclasse.methode_static(); // Appel de la méthode statique
int i = Nomclasse.nbr ;    // Lecture de la valeur de la variable statique

```

Remarque :

Les méthodes statiques ne peuvent pas utiliser les données membre et les propriétés non statiques.

Exemple : Les méthodes « addition » et « soustraction » sont complètement indépendantes des instances de la classe « **OperMath** ».

```

class OperMath
{
    public static int addition(int a, int b)
    { return a + b; }
    public static int soustraction(int a, int b)
    { return a - b; }
}

```

12.2. Propriété statique :

On parle aussi de propriétés statiques c.-à-d. indépendantes des instances d'une classe.

Exemple : Propriété qui détermine le nombre des instances créées :

```

class NomClasse
{
    public static int nbr_Objet_Créés { get; }    //variable statique : compteur des instances
    public NomClasse()
    { nbr_Objet_Créés++; }    //incréméntation de compteur des instances
}

```

```

}
NomClasse n1 = new NomClasse(),
n2 = new NomClasse(), n3 = new NomClasse();
Console.Write(NomClasse.nbr_Objet_Crées); //Affiche : 3

```

12.3. Classe statique :

Une classe statique est une classe qui ne peut pas être instanciée.

Une classe statique ne contient que des membres statiques.

Si une classe ne contient que des variable et méthodes statiques alors elle peut devenir également statique.

Syntaxe :

```

static class NomClasse
{
    public static int nbr; //variable statique
    public static void methode_static() //méthode statique
    { }
}

```

13. Les classes internes (nested class) :

Les classes internes sont un mécanisme qui permet d'avoir des classes définies à l'intérieur d'autres classes. Cela permet de limiter l'accès d'une classe uniquement à sa classe mère.

```

class NomClasse
{ ...
    class classeInterne { ... }
    ...
}

```

La classe « classeInterne » ne peut être utilisée que par la classe « NomClasse » car elle est privée. Si la classe interne est définie en « **protected** » alors les classes dérivées de « NomClasse » peuvent aussi utiliser la classe « classeInterne ».

Si nous mettons la classe « classeInterne » en « **public** » ou « **internal** », elle sera utilisable par tout le monde ; comme une classe normale.

```

class NomClasse
{
    internal class classeInterne { }
}
//Instantiation d'un objet:
NomClasse.classeInterne ci = new NomClasse.classeInterne();

```

14. Le mot clé « var » :

Il permet de déclarer les variables implicitement typées, il demande au compilateur de déduire le type de la variable à partir de sa valeur initiale.

```

var v=10 ; //équivalent à : int v = 10 ;

```

Il permet aussi de créer des types anonymes, c'est-à-dire des types dont on n'a pas défini la classe au préalable. Une classe sans nom.

```

var v = new { a = 10, b = true, message = "..." }; //La structure de la classe est déduite à
                                                    //son initialisation
Console.Write(v.b); //Affiche: True

```

15. Les collections :

Les collections sont une alternative aux tableaux. La différence majeure est que la taille d'un tableau est fixée alors que celle d'une collection peut varier.

15.1. Les tableaux dynamiques :

La classe « **ArrayList** » (présente dans l'espace de nom « System.Collections ») implémente un tableau dynamique d'objets. La taille d'un tel tableau est automatiquement ajustée, si nécessaire, lorsque des éléments y sont ajoutés, ce qui n'est pas le cas pour les tableaux traditionnels.

Un tableau dynamique peut contenir n'importe quel objet puisqu'il contient des objets d'une classe dérivée de la classe « **Object** ».

```
ArrayList al = new ArrayList();
al.Add(new Personne("nom", "prenom", 20));
al.Add("texte");
al.Remove(1);
```

La classe « **Hashtable** » (présente dans l'espace de nom « System.Collections ») implémente un conteneur parfois appelé « panier » (bag en anglais) ou dictionnaire. On y insère des objets sans ordre particulier, objets que l'on pourra retrouver plus tard sur la base d'une clé qui doit être unique.

La clé peut être de type entier ou objet de n'importe quelle classe.

Exemple :

```
Hashtable ht = new Hashtable();
Personne p = new Personne { Nom = "Alaoui", Age = 30 };
ht.Add(p.Nom, p);
p = (Personne)ht["Alaoui",];
if (p != null) Console.WriteLine(p.Nom);
else Console.WriteLine("Pas trouvé");
```

15.2. La classe « List »

Elle peut être utilisée avec n'importe quel type, entier, chaîne de caractères, personne, ... Les listes permettent de stocker plusieurs éléments de même type au sein d'une seule entité.

Déclaration :

```
List<string> listeString = new List<string>();
List<int> listeInt = new List<int>();
List<Personne> listePersonne = new List<Personne>();
List<object> L = new List<object>();
```

Pour accéder à un élément en position « index » : **liste[index]**

```
List<type> liste = new List<type>(); //Instanciation
liste.Add(element);                // l'ajoute à la fin de la liste
liste.AddRange(collection);        // ajoute d'une collection d'éléments à la fin de la liste.
liste.Remove(valeur);              // supprime le premier élément de la liste possédant la valeur passée
                                   // en paramètre
liste.RemoveAt(index);             // supprime l'élément contenu à index mentionné
liste.Clear();                     // supprime tous les éléments de la liste
```

```
int[] tableau = { 1, 2, 3, 4, 5 };
List<int> liste = new List<int>();
liste.AddRange(tableau);
```

```
List<int> chiffres = new List<int>();
```

création de la liste

```
chiffres.Add(8);  
chiffres.Add(9);  
chiffres.Add(4);  
chiffres.RemoveAt(1); //Suppression en utilisant l'indice  
foreach (int entier in chiffres) {  
    Console.WriteLine(entier);  
}
```

chiffres contient 8
chiffres contient 8, 9
chiffres contient 8, 9, 4
chiffres contient 8, 4

Affichage de contenu de la liste en
utilisant une boucle

Gestion d'une liste :

Recherche d'un élément : La méthode IndexOf() permet de rechercher un élément dans la liste et de renvoyer son indice	int indice = chiffres.IndexOf(8);
Obtenir le nombre d'éléments contenu dans la liste	int n=chiffres.Count ;
Vérifier si une liste contient un élément	bool existe = liste.Contains(element);
Trier la liste en ordre ascendant.	Liste.sort() ;

15.3. La classe Queue < > :

Elle permet de gérer une file d'attente style FIFO (first in, first out : premier entré, premier sorti). Parmi les méthodes de la classe Queue:

- **Enqueue**(objet) qui ajoute un objet à la fin de la liste.
- **Dequeue**() qui supprime et retourne l'objet au début de la liste.

Queue<int> List_FIFO = new Queue<int>();	
List_FIFO.Enqueue(10);	//le contenu de la liste est : 10
List_FIFO.Enqueue(5);	//le contenu de la liste est : 10 puis 5
int suivant = List_FIFO.Dequeue();	//le contenu de la liste es : 5
List_FIFO.Enqueue(1);	//le contenu de la liste est : 5 puis 1

15.4. La classe Dictionary :

C'est une liste composée d'une clé et d'une valeur et qui permet de stocker plusieurs éléments sous la forme (clef, valeur).

Pour accéder à un élément, il suffit d'utiliser l'opérateur [] avec la clef de l'élément : **liste[clef]**

Déclaration	Dictionary<type_clef, type_elements> liste;
Instanciation	liste = new Dictionary<type_clef, type_elements>();
Insertion d'un élément	liste.Add(clef, element);
Insertion d'un élément	liste[clef] = element;
Supprimer l'élément associé à cette clef.	liste.Remove(clef);
Supprimer tous les éléments de la liste.	liste.Clear();
Obtenir le nombre d'éléments.	liste.Count();
Vérifier si une clef est contenue dans la liste.	liste.ContainsKey(clef);
Vérifier si un élément est contenu dans la liste.	liste.ContainsValue(element);

Exemple:

Dictionary<int, string> listeStagiaires;
listeStagiaires = new Dictionary<int, string>();
listeStagiaires.Add(1, "Ahmadi");
listeStagiaires[2] = "El Mandoubi";
listeStagiaires.Remove(1);

16. La gestion des exceptions :

La classe « Exception » permet la gestion des erreurs qui interrompent l'exécution de programme en produisant un rapport d'erreur.

Une exception est levée lorsqu'une ligne de code rencontre un cas limite qui nécessite d'être géré.

Pour éviter le plantage de l'application, nous devons gérer ces cas limites et intercepter les exceptions.

16.1. Try ... catch :

Il faut encadrer les instructions pouvant atteindre des cas limites avec le bloc d'instruction **try...catch**,

Exemple :

```
float moyenne;
Console.Write("Veuillez saisir un nombre : ");
try
{
    //surveiller l'exécution de code suivant
    //En cas d'erreur interrompre l'exécution de
    //try et aller au catch
    moyenne = float.Parse(Console.ReadLine());
}
catch (Exception)
{
    //La gestion de l'exception levée si
    //la conversion échoue
    Console.WriteLine("La valeur que vous avez saisi est incorrecte !");
}
```

Pour Obtenir les informations sur l'exception on utilise un paramètre dans le bloc catch

```
catch (Exception ex)
{
    Console.WriteLine("Type de l'exception : " + ex.GetType());
    Console.WriteLine("Pile d'appel : " + ex.StackTrace);
    Console.WriteLine("Message d'erreur : " + ex.Message);
    Console.WriteLine("Représentation de l'exception : " + ex.ToString());
}
```

Les exceptions dérivent de la classe « **Exception** ». Il existe une hiérarchie entre les exceptions. Deux grandes familles d'exceptions existent : **ApplicationException** et **SystemException**

16.2. Enchaîner les blocs catch :

Cette méthode permet de gérer les exceptions d'une manière très précise en traitant chaque type d'exception d'une manière différente :

```
catch (NullReferenceException ex){ ... } // Erreur de référence nulle
catch (FormatException ex){ ... } // Erreur de format
catch (ApplicationException ex){ ... } // Erreur d'application
catch (SystemException ex) { ... } // Erreur système
catch (Exception ex){ ... } // Les exceptions restantes
```

16.3. Les blocs « try ... catch » imbriqués :

```
try
{
    try
    {...}
    catch (Exception)
    {...}
}
catch (Exception ex) { ...}
```

16.4. Le mot clé « Finally »

Ce mot clé est utilisé pour exécuter un code dans tous les cas, qu'une exception intervienne ou pas.

```
try { ... }
catch { ... }
finally { ... }
```

16.5. Lever une exception

Il est possible de déclencher soi-même la levée d'une exception. C'est utile si nous considérons que notre code a atteint un cas limite, qu'il soit fonctionnel ou technique.

Pour lever une exception, nous utilisons le mot-clé **throw**, suivi d'une instance d'une exception.

```

static void Main(string[] args)
{
    try
    {
        bool b = NoteValide(21);
    }
    catch (Exception ex)
    {
        Console.WriteLine("La note n'est pas valide : " + ex.Message);
    }
}
public static bool NoteValide(float note)
{
    if(note<0 || note>20)
        throw new ArgumentOutOfRangeException("note","La note doit être comprise entre 0 et 20");
    return true;
}

```

Le résultat :
La note n'est pas valide : La note doit être comprise entre 0 et 20
Nom du paramètre : note

16.6. Exception personnalisée :

On utilise ce principe pour créer une (ou plusieurs) classe qui dérive de la classe « Exception » et qui permet de lever nos propres exceptions correspondant à des cas limites fonctionnels ou techniques.

```

class NoteNonValideException : Exception
{
    // surcharge du constructeur prenant un string en paramètre
    // et modification de message d'erreur
    public NoteNonValideException(float note)
        : base("La note "+note+" n'est pas une note valide !"){ }
}
class Program
{
    static void Main(string[] args)
    {
        try
        {
            bool b = NoteValide(22);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.GetType());
            Console.WriteLine(ex.Message);
        }
    }
    private static bool NoteValide(float n)
    {
        if(n<0 || n>20) throw new NoteNonValideException(n);
        return true;
    }
}

```

Le résultat:
ConsoleApplication4.NoteNonValideException
La note 22 n'est pas une note valide !

17. S rialisation et d s rialisation :

La « **s rialisation** » consiste   rendre un objet susceptible d' tre enregistr  sur disque ou d' tre transmis   une autre machine via une ligne de communication.

La « **d s rialisation** » consiste   cr er et initialiser un objet   partir d'informations provenant d'un fichier. La s rialisation permet aussi de passer des objets d'une machine   l'autre.

Pour s rialiser un objet, il suffit de :

1. Marquer la classe de l'objet avec l'attribut **[Serializable]**.

```
[Serializable]
class Personne
{...}
```

2. Cr er un objet **BinaryFormatter** ainsi qu'un objet **FileStream** pour cr er un fichier. Ajouter les espaces de nom :

- **System.IO;**
- **System.Runtime.Serialization.Formatters.Binary;**

3. Ex cuter la m thode **Serialize(...)**.

Exemple :

```
Personne p = new Personne("nom", "prenom", 20);
BinaryFormatter bf = new BinaryFormatter();
FileStream fs = new FileStream("fichier.txt", FileMode.Create, FileAccess.Write);
bf.Serialize(fs, p);
fs.Close();
```

Pour d s rialiser l'objet, c'est- -dire l'instancier en m moire   partir d'une lecture sur disque, il faut :

1. Cr er un objet **BinaryFormatter** ainsi qu'un objet **FileStream** pour lire un fichier.
2. Ex cuter la m thode **Deserialize(...)** appliqu    l'objet **BinaryFormatter**.

Exemple :

```
BinaryFormatter bf = new BinaryFormatter();
FileStream fs = new FileStream("fichier.txt", FileMode.Open,
FileAccess.Read); Personne p = (Personne)bf.Deserialize(fs);
Console.WriteLine(p.Nom);
```

17.1. S rialisation des objets en XML :

 tapes   suivre :

- 1- Utiliser les espaces de nom « **System.IO** » et « **System.Xml.Serialization** »
- 2- Cr er une instance de **XmlSerializer** dans lequel on sp cifie le type de l'objet   s rialiser. On utilise l'op rateur **typeof** pour cela
- 3- Cr er un objet **FileStream**
- 4- Ex cuter **Serialize(...)** ou **Deserialize(...)**

S rialisation :

```
XmlSerializer xmlS = new XmlSerializer(typeof(Personne));
FileStream fs = new FileStream("fichier.xml", FileMode.OpenOrCreate,
FileAccess.ReadWrite);
```

```
Personne p = new Personne("nom", "prenom", 20);
xmlS.Serialize(fs, p);
fs.Close();
```

Ainsi le r sultat de la s rialisation est un fichier xml contenant les informations suivantes :

```
<?xml version="1.0"?>
<Personne ...>
  <Nom>nom</Nom>
  <Prenom>prenom</Prenom>
```

```
<Age>20</Age>
</Personne>
```

Désérialisation:

```
XmlSerializer xmlS = new XmlSerializer(typeof(Personne));
FileStream fs = new FileStream("fichier.xml", FileMode.Open,
FileAccess.Read);
Personne p = (Personne)xmlS.Deserialize(fs);
Console.WriteLine(p.Nom);
```

18. Ecriture dans un fichier texte :

Ces exemples montrent différentes manières d'écrire du texte dans un fichier.

Exemple 1 :

```
string[] lines = { "First line", "Second line", "Third line"
};
System.IO.File.WriteAllLines(@"C:\fichier1.txt", lines);
```

Exemple 2:

```
string text = "A class is the most powerful data type in C#. Like a structure, "
+ "a class defines the data and behavior of the data type. ";
System.IO.File.WriteAllText(@"C:\fichier2.txt", text);
```

Exemple 3 :

```
using (System.IO.StreamWriter file = new System.IO.StreamWriter(@"C:\fichier3.txt"))
{
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
    }
}
```

Exemple 4 : ajout d'un texte au contenu d'un fichier :

```
using (System.IO.StreamWriter file =
    new System.IO.StreamWriter(@"C:\fichier3.txt", true))
{
    file.WriteLine("Fourth line");
}
```

18.1. Lire le contenu d'un fichier texte :

Exemple 1 :

Cet exemple lit le contenu d'un fichier texte, ligne par ligne, dans une chaîne à l'aide de la méthode **ReadLine** de la classe **StreamReader**. Chaque ligne de texte est stockée dans la chaîne line et affichée à l'écran :

```
int counter = 0;
string line;

System.IO.StreamReader file = new
System.IO.StreamReader(@"c:\test.txt");
while ((line = file.ReadLine()) != null)
{
    System.Console.WriteLine (line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
```

```
// Suspend the screen.
System.Console.ReadLine();
```

Exemple 2 :

Cet exemple lit le contenu d'un fichier texte à l'aide des méthodes statiques **ReadAllText** et **ReadAllLines** de la classe **System.IO.File**.

```
string text = System.IO.File.ReadAllText(@"C:\WriteText.txt");
System.Console.WriteLine("Contents of WriteText.txt = {0}", text);
string[] lines = System.IO.File.ReadAllLines(@"C:\ WriteLines2.txt");

System.Console.WriteLine("Contents of Writelines2.txt = ");
    foreach (string line in lines)
    {
        // Use a tab to indent each line of the
        file. Console.WriteLine("\t" + line);
    }
// Keep the console window open in debug
mode. Console.WriteLine("Press any key to
exit."); System.Console.ReadKey();
```

19. Lire et écrire dans un fichier JSON :

Json et XML sont presque semblables en termes d'utilisation, mais ils sont équivalents au fichier texte en termes d'accessibilité. Vous pouvez utiliser le code suivant pour lire / écrire un fichier json :

Lecture :

```
var stream = File.OpenText("json file.txt");
string st = stream.ReadToEnd();
string result = "";
var jsonArray = JsonArray.Parse(st);
foreach (var item in jsonArray)
{
    JsonObject ob = new JsonObject(item);
    foreach (var t in ob.Values)
    {
        JsonObject oo = new JsonObject(t);
        foreach (var x in oo)
        {
            Result+=(x.Key + " : " + x.Value + " - ");
        }
    }
}
}
```

Ecriture :

```

KeyValuePair<string, JsonValue> pair = new KeyValuePair<string, JsonValue>("FName","Sourabh");
KeyValuePair<string, JsonValue> pair2 = new KeyValuePair<string, JsonValue>("LName", "Sinha");
List<KeyValuePair<string, JsonValue>> list = new List<KeyValuePair<string, JsonValue>>();

list.Add(pair);
list.Add(pair2);
JsonObject jobject = new JsonObject(list);
var stream = new StreamWriter("json out file.txt");
foreach (var x in jobject)
{
    stream.WriteLine(x.ToString() + "\n");
}
JsonArray jarray = new JsonArray("item1","item2","Item3");
foreach (var x in jarray)
{
    stream.WriteLine(x.ToString());
}

```

Exemple 2 :

Le contenu de fichier JSON :

```

[
  { "millis": "1000", "stamp":
    "1273010254", "datetime":
    "2010/5/4 21:57:34", "light":
    "333",
    "temp": "78.32",
    "vcc": "3.54" },
]

```

```

public class Item
{
    public int millis;
    public string stamp;
    public DateTime datetime;
    public string light;
    public float temp;
    public float vcc;
}

```

Récupération de contenu de fichier JSON

```

public void LoadJson()
{
    using (StreamReader r = new StreamReader("file.json"))
    {
        string json = r.ReadToEnd();
        List<Item> items = JsonConvert.DeserializeObject<List<Item>>(json);
    }
}

```

Récupérer les données sans déclaration de la classe Item :

```

dynamic array = JsonConvert.DeserializeObject(json);
foreach(var item in array)
{
    Console.WriteLine("{0} {1}", item.temp, item.vcc);
}

```