



Programmation Orientée Objet C#:

M.WANE

**Ingénieur Informaticien
Analyste, Concepteur
Développeur d'Applications**

Email : douvewane85@gmail.com

Programme

❑ **S'initier à C#**

- Comprendre l'histoire de C# et de .NET
- Découvrir l'environnement .NET
- Installer Visual Studio
- Créer sa première application

❑ **Programmation orientée Objet C#**

- Les espaces de nom
- Les variables, le type de valeur (structures, énumérations) et le type de Reference
- **Fondamentaux de l'orienté objet**
 - Les propriétés, Classes et les objets
 - Héritage, Polymorphisme et accessibilité
 - classes abstract et sealed
 - Interface

❑ **Bases de la syntaxe**

- Les structures Conditionnelles if ,Switch
- Les tableaux
- Les boucle while ,for ,foreach
- Les opérateurs en C#

Programme

❑ **Les classes du FCL (Framework Class Library)**

- **Le string**
- **Comprendre les types collections**
- **Définir et utiliser des listes**
- **Définir et utiliser des dictionnaires**
- **Manipuler des dictionnaires**

❑ **Comprendre les fonctionnalités du langage C#**

- **Gérer les exceptions**
- **Aller plus loin dans la gestion des exceptions**
- **Aborder les types nullable**
- **Utiliser le typage automatique**
- **Découvrir les variables dynamiques**
- **Appliquer les types anonymes**
- **Avoir une approche des délégués**
- **Créer un délégué pour la recherche**
- **Poursuivre la création d'un délégué pour la recherche**
- **Utiliser des fonctions anonymes**
- **Se servir d'expressions lambda**

Programme

❑ **Découvrir LINQ**

- Travailler sur une première requête LINQ
- Utiliser l'énumérateur
- Découvrir les clauses group et orderby
- Utiliser les fonctions dans LINQ
- Appliquer l'équivalent de IN en LINQ
- Se servir d'une syntaxe LINQ sous forme de méthode

S'initier à C#



Comprendre l'histoire de .Net et C#

.Net

Microsoft .NET (prononcé « dot net ») est le nom donné à un ensemble de produits et de technologies informatiques de l'entreprise Microsoft pour rendre des applications facilement portables sur Internet.

Le but est de fournir un serveur web local permettant de gérer des services et évitant d'externaliser des données privées sur un service web de stockage ou un hébergement web tiers.

La plate-forme .NET se base sur plusieurs technologies :

- Les systèmes d'exploitation propriétaires Microsoft Windows ;
- **Des protocoles de communication** basés sur le Framework .NET;
- **Une bibliothèque logicielle** compatible Framework .NET ;
- **Un environnement d'exécution** de code basé sur la **CLI multi-langage(C#, Visual Basic .NET, J#...)** ;
- **MSBuild** : un outil de gestion de projet avec plusieurs compilateurs ;
- **Visual Studio** : un IDE de développement utilisant la métaprogrammation et compatible avec Visual C++ ;
- **Framework .NET** : un ensemble de bibliothèques de haut niveau (**actuellement on est a la version 4.6**);
- **Une portabilité** pour les systèmes d'exploitation Windows et Windows Mobile ;
- **Des composants** facilitant le développement d'applications locales ou web (ASP.NET) ;

S'initier à C#



Le Langage C#

- ❑ C# est un langage de programmation orienté objet, commercialisé par Microsoft depuis 2002 et destiné à développer sur la plateforme Microsoft .NET. Il est dérivé du C++ et très proche du Java dont il reprend la syntaxe générale ainsi que les concepts.
- ❑ Langage de **programmation orienté objet** (Classe, Objet, Héritage, Encapsulation et Polymorphisme)
- ❑ Avec C# on peut créer des applications **Windows, Web et Mobiles** sous Windows.
- ❑ Un Programme C# est formé :
 - ❖ **De bibliothèques ou Assemblies**: est un ensemble de Namespace, de classe externes utilisables dans un projet.
 - ❖ **FLC(Framework Class Library)**: Les classes de bases du Framework .Net
- ❑ Exécution d'un Programme C#:

Le C# est un langage compilé (**compilateur csc.exe**) en un format **MSIL(Microsoft intermédiaire Language)**. Le MSIL est un langage binaire intermédiaire qui est fait pour être exécuté non pas sur la machine mais sur le **CLR(Common Language Runtime)**. Le CLR est un environnement d'exécution ou **une machine virtuelle**.

La compilation génère deux types de Fichiers:

- ❖ **.exe**, des fichiers exécutables
- ❖ **.dll**, des assemblies réutilisables dans d'autres projets

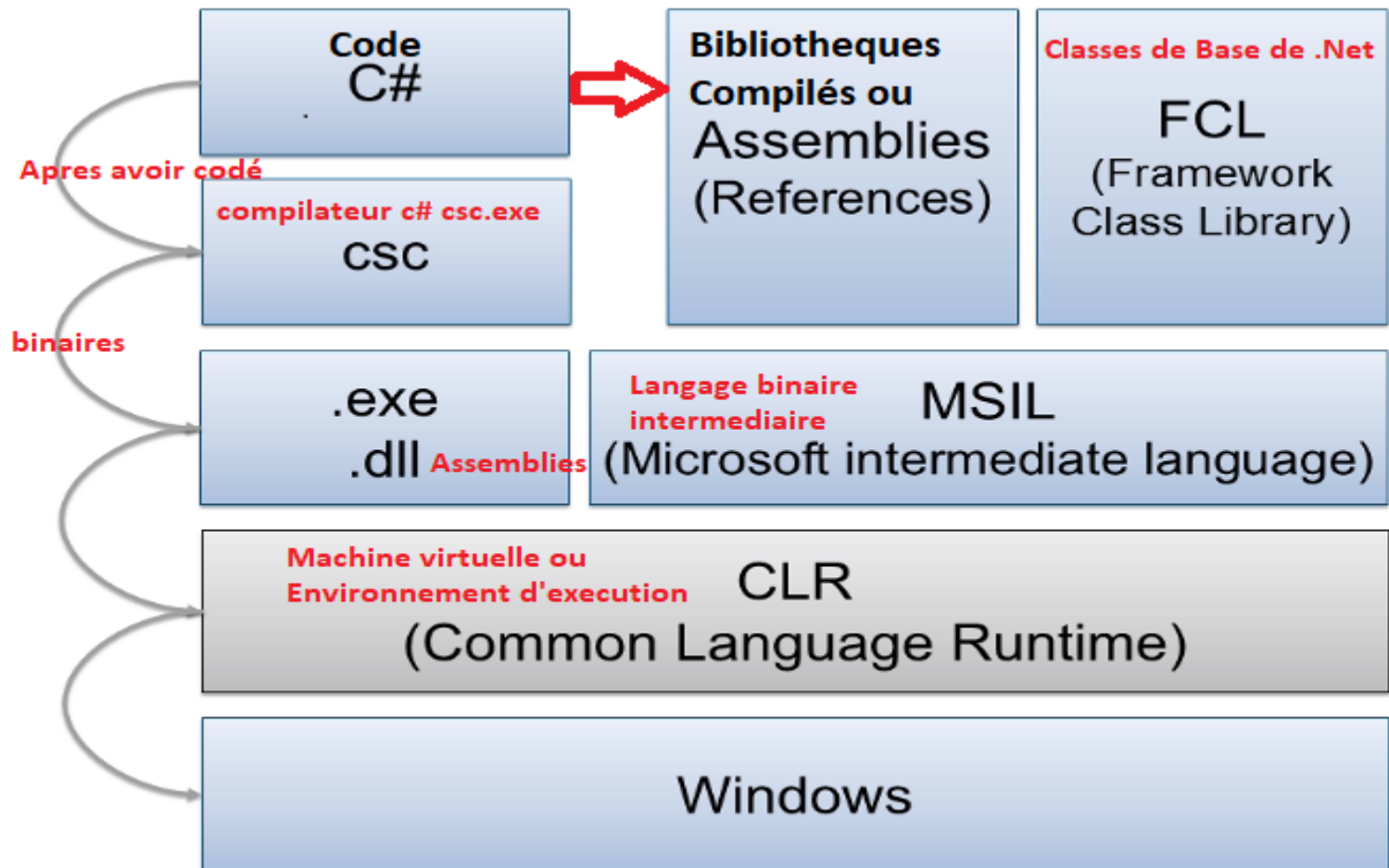
Ainsi le code compile en MSIL sera exécuté par cette couche le CLR qui fait appel au système d'exploitation Windows.

On dit que le Framework est multiplateforme dans l'environnement Windows.

S'initier à C#



Principe d'exécution du Langage C#



S'initier à C#

❑ Découvrir l'environnement .NET

Pour écrire du code **C#** on utilise les IDE ou environnement de développement suivant:

❖ **Visual Studio** disponible en *Community, Professionnel et Enterprise*.

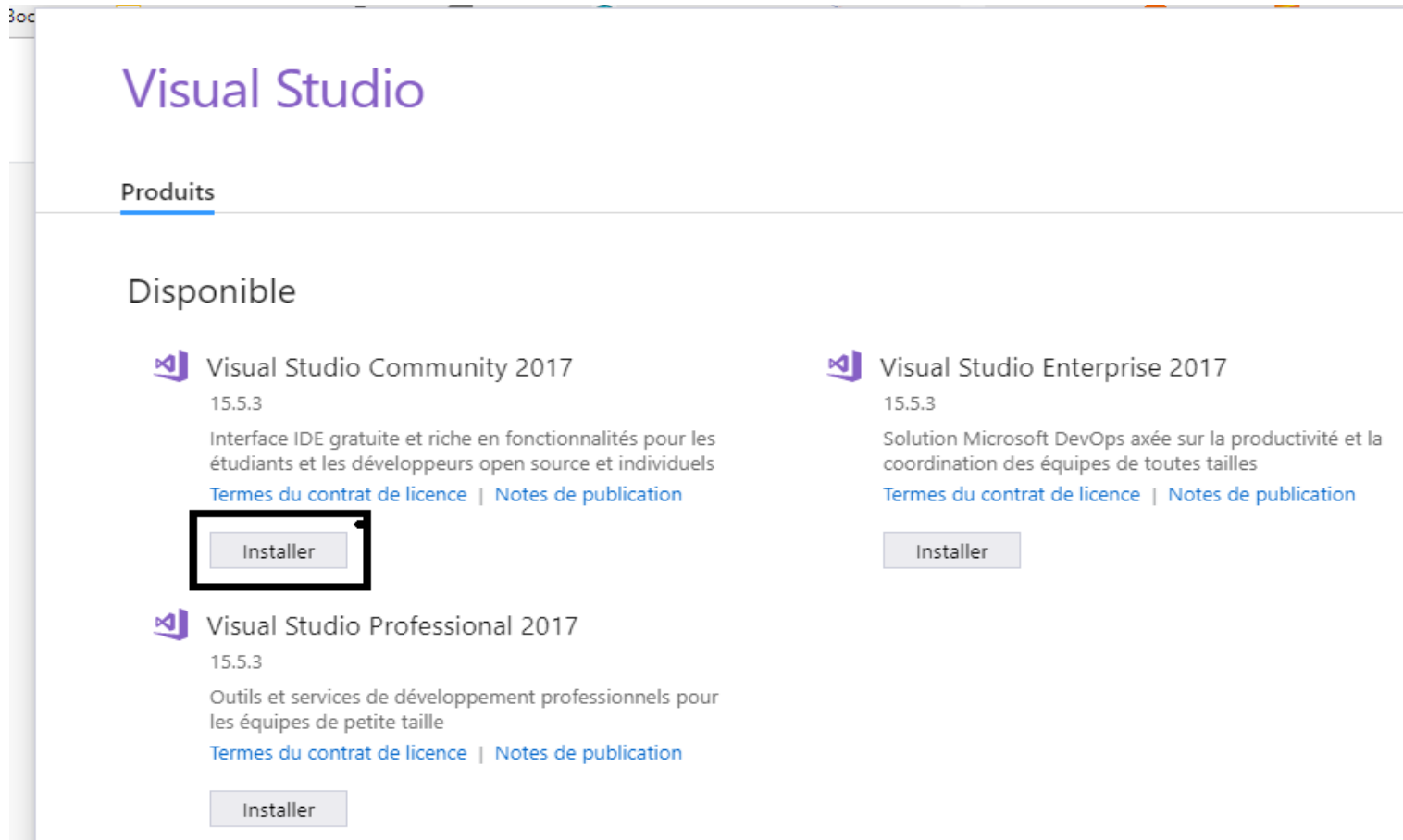
*Cet IDE permet de faire du développement **multiplateformes** :*

- **Le Développement Desktop(WinForm ,WPF)**
- **Le Développement Web(ASP.NET)**
- le développement **multiplateforme mobile (IOS,Android,Windows Phone)**
- ❖ **#develop (abréviation de SharpDevelop)** est un IDE gratuit pour les projets C #, VB.NET et Boo sur la plate-forme .NET de Microsoft.
- ❖ **MonoDevelop** permet aux développeurs d'écrire rapidement des applications bureautiques et Web sous Linux, Windows et Mac OS X. Il facilite également le portage des applications .NET créées avec Visual Studio vers Linux et Mac OS X en maintenant une base de code unique pour toutes les plateformes.

NB: Dans ce cours nous utiliserons **Visual Studio** comme IDE dans sa version 2015 et nous aborderons le **développement Desktop(WinForm ,WPF)** .

S'initier à C#

❑ Installer Visual Studio





The screenshot shows the Visual Studio download page. At the top, the 'Visual Studio' logo is displayed in purple. Below it, the 'Produits' (Products) section is highlighted with a blue underline. Underneath, the word 'Disponible' (Available) is shown. There are three product cards: 'Visual Studio Community 2017', 'Visual Studio Enterprise 2017', and 'Visual Studio Professional 2017'. Each card includes the version number '15.5.3', a brief description, and links for 'Termes du contrat de licence' and 'Notes de publication'. An 'Installer' button is present on each card. The 'Installer' button for Visual Studio Community 2017 is highlighted with a black rectangular box.


Visual Studio

Produits

Disponible

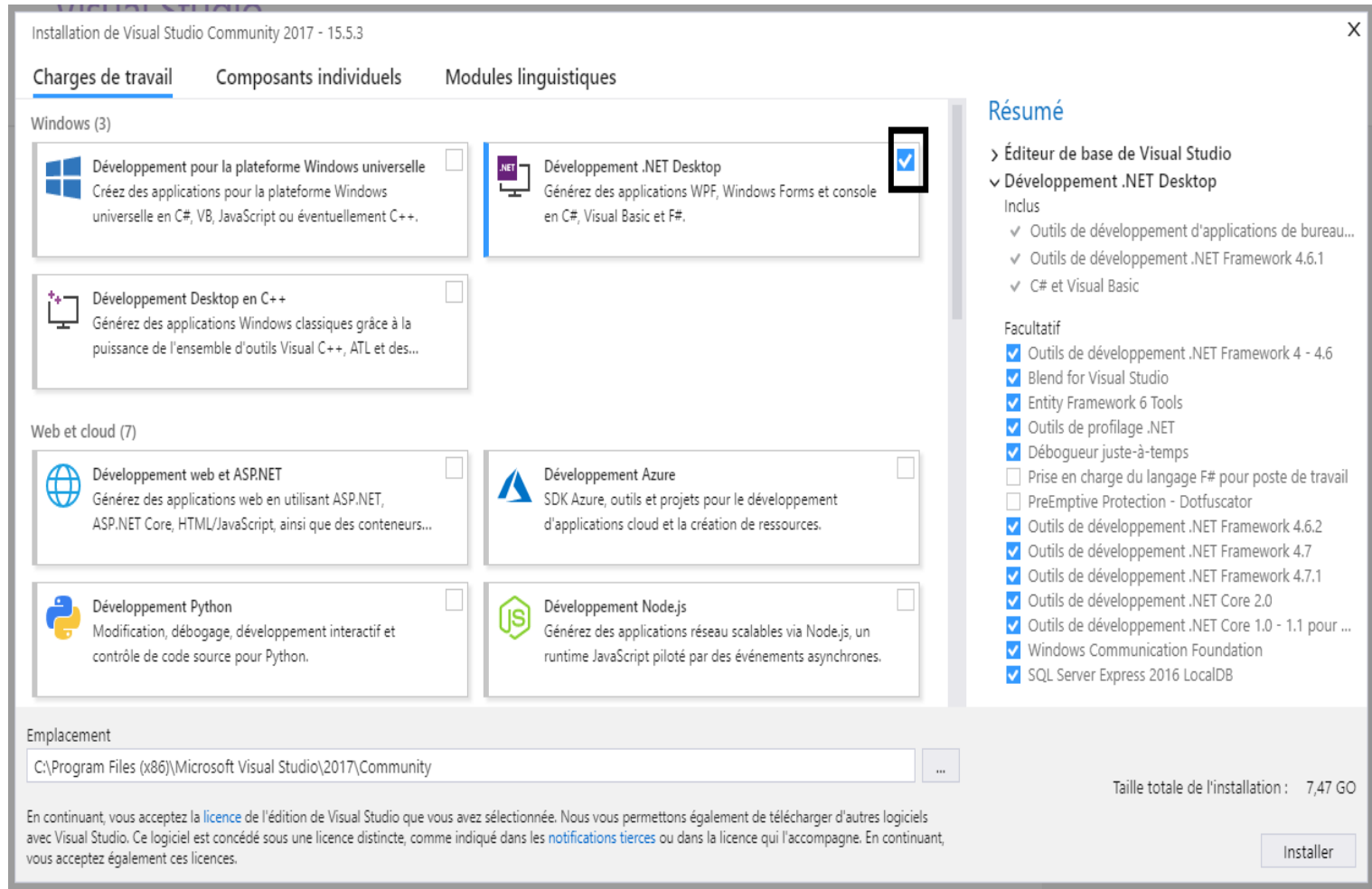
 **Visual Studio Community 2017**
15.5.3
Interface IDE gratuite et riche en fonctionnalités pour les étudiants et les développeurs open source et individuels
[Termes du contrat de licence](#) | [Notes de publication](#)
Installer

 **Visual Studio Enterprise 2017**
15.5.3
Solution Microsoft DevOps axée sur la productivité et la coordination des équipes de toutes tailles
[Termes du contrat de licence](#) | [Notes de publication](#)
Installer

 **Visual Studio Professional 2017**
15.5.3
Outils et services de développement professionnels pour les équipes de petite taille
[Termes du contrat de licence](#) | [Notes de publication](#)
Installer

S'initier à C#

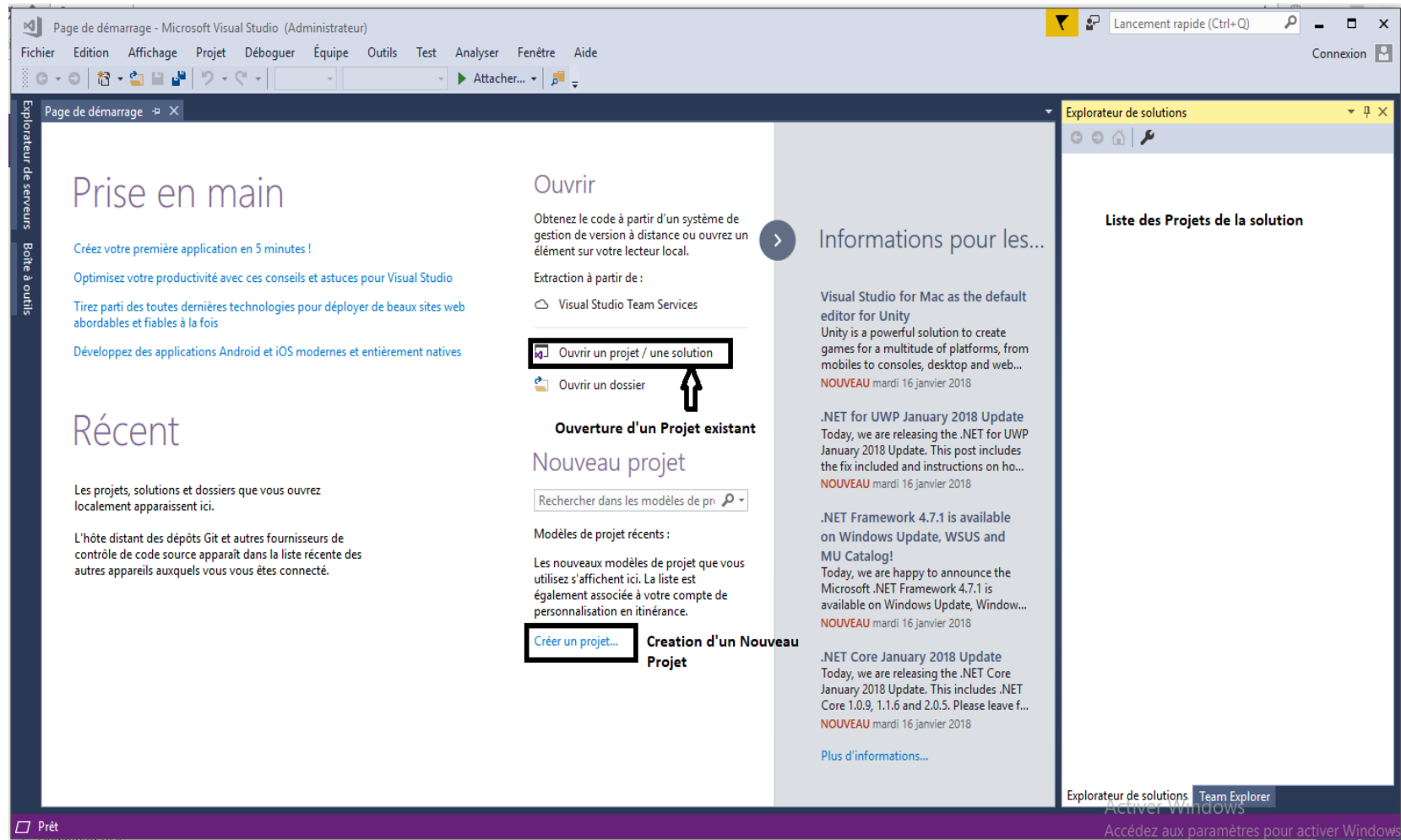
❑ Installer Visual Studio



S'initier à C#



IDE ou environnement de développement

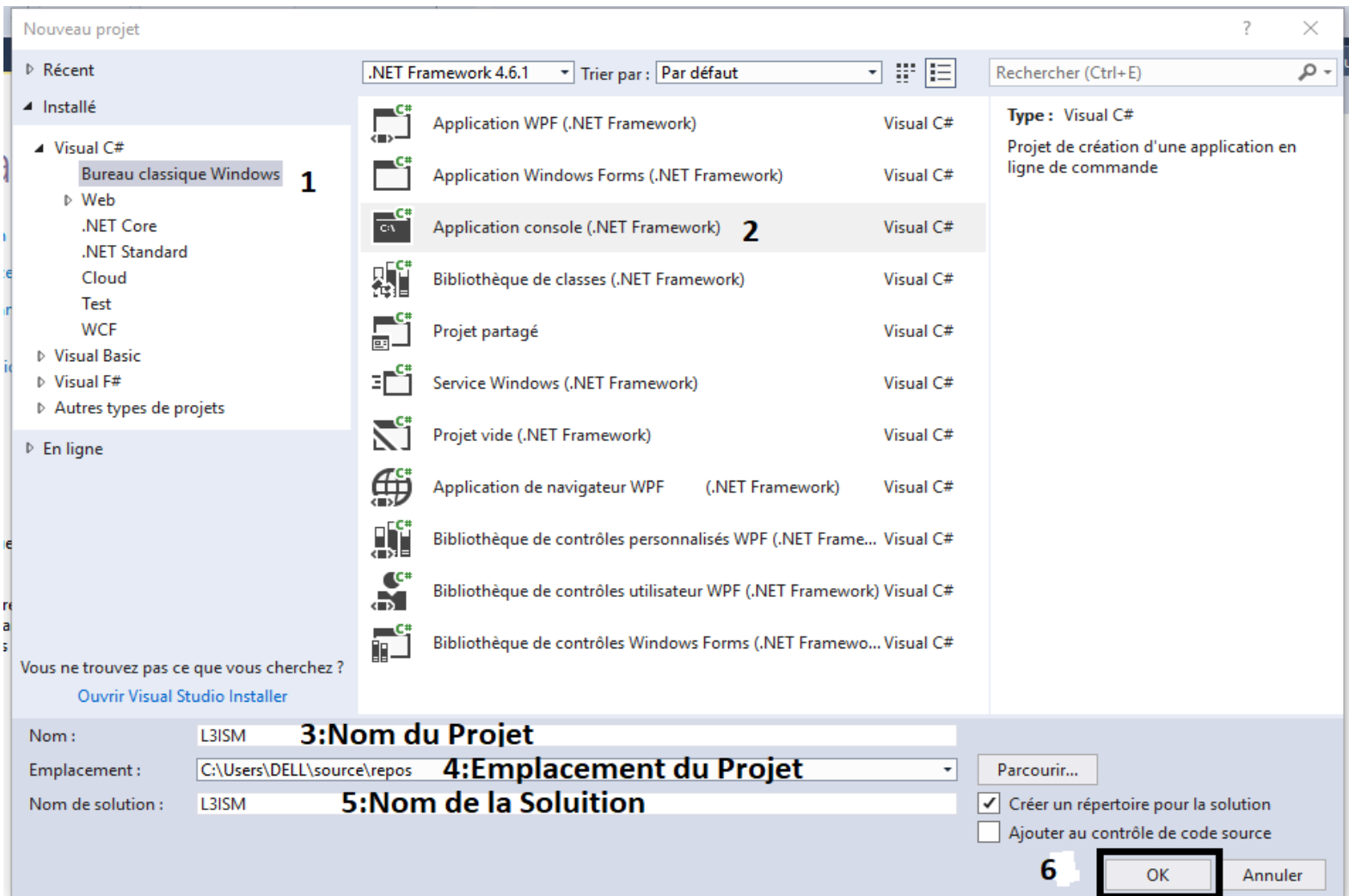


S'initier à C#

- ❑ **Une application Desktop en C# peut être formée:**
 - ❖ **Une application console:** une application sans interface graphique
 - ❖ **Une application Windows Forms:** une application avec interface graphique avec les API Windows Traditionnels.
 - ❖ **Une application Windows WPF:** une application avec interface graphique avec les API Windows Modernes ,définie sur des interfaces graphiques en XML.
- ❑ **Solution:** est un concept d'organisation sur Visual Studio qui peut contenir plusieurs projets. Dans une Solution on peut avoir:
 - **Des Bibliothèques(.dll)**
 - **Des Applications Windows ou Console**
 - **une Couche d'accès au Données ou DAO**

S'initier à C#

❑ Créer sa première application Console



II.) Programmation orientée Objet C#

II.1) Namespace ou Espaces de Nom: est une étendue qui contient un ensemble d'objets connexes. Vous pouvez utiliser un espace de noms pour organiser les éléments de code et créer des types uniques globalement. Dans un espace de noms, vous pouvez déclarer un ou plusieurs des types suivants: **un autre espace de noms, Classe, Interface, Struct, Enum, Déléguer**

Pour déclarer un espace de nom on utilise la syntaxe:

using NomNamespace;

```
namespace NomNamespace
{
    //Classe
    class NomClass { }
    //Interface
    interface NomInterface { }
    //Structure
    struct NomStruct { }
    //enumeration
    enum NomEnum { a, b }
    //Delegate
    delegate void NomDelegate(int i);
    //Namespace interne
    namespace NomNamespace.Nested
    {
        //Classe interne
        class NomClass2 { }
    }
}
```

■ Exemples de Namespaces

❖ **System** contient les classes

- **Console avec les méthodes:**

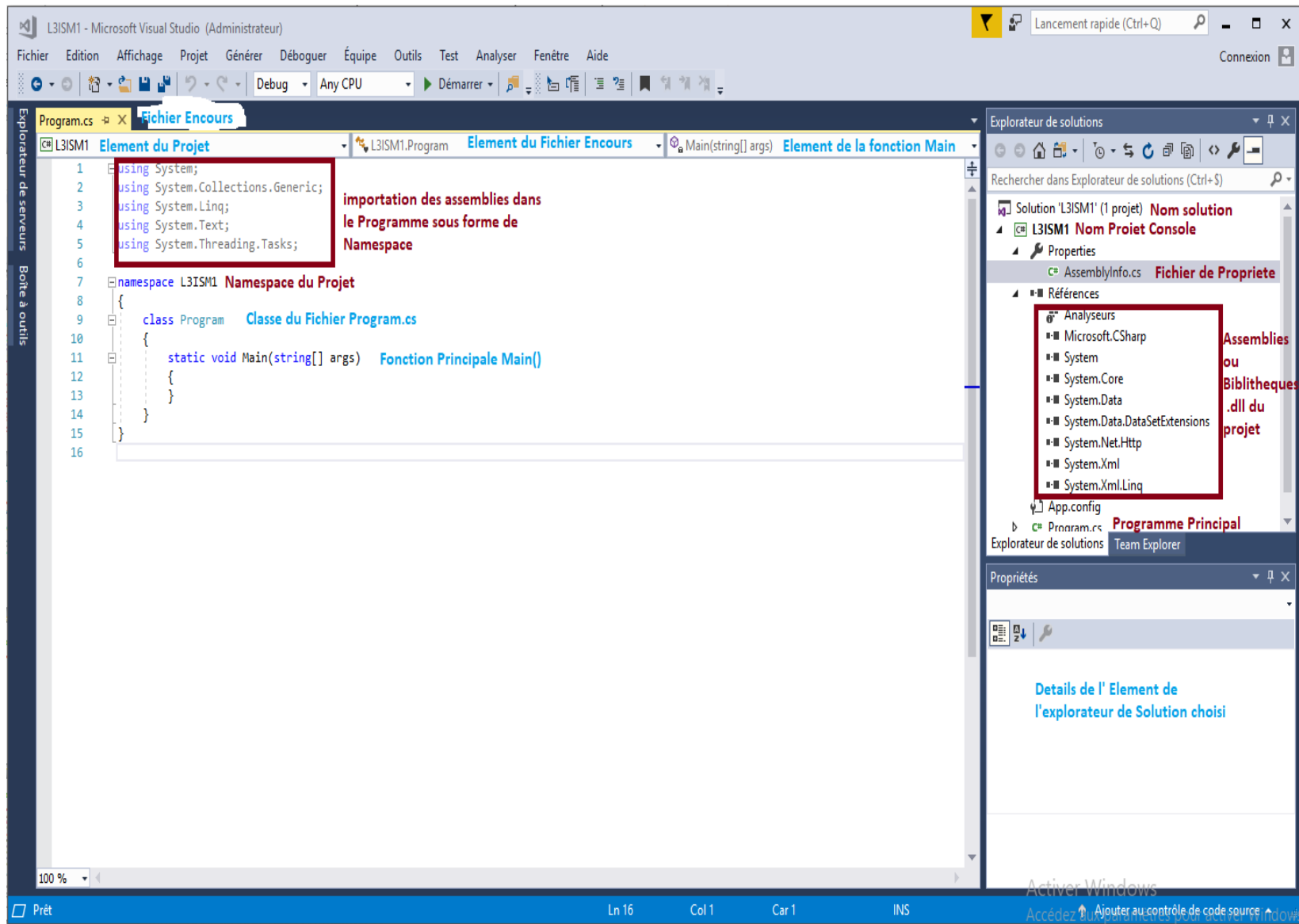
- **WriteLine()**: qui permet d' écrire une information sur la console

- **readKey()**: qui permet de lire un caractère au clavier

- **Clear()**: Efface la mémoire tampon et la fenêtre correspondante de la console contenant les informations d'affichage.

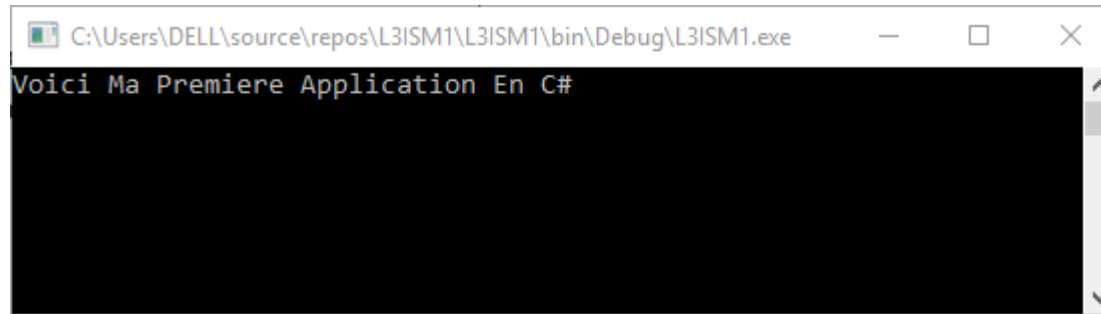
- **ReadLine()**: Lit la ligne de caractères suivante du flux d'entrée standard.

Exemple d'une Première application Console



Exercice d'Application 2 : Structures

Ecrire un Programme qui affiche sur la Console **Voici Ma Première Application En C#.**



A screenshot of a Windows console window. The title bar at the top shows the file path "C:\Users\DELL\source\repos\L3ISM1\L3ISM1\bin\Debug\L3ISM1.exe" and standard window controls (minimize, maximize, close). The console area has a black background with white text. The first line of text is "Voici Ma Première Application En C#". A vertical scrollbar is visible on the right side of the console window.

```
C:\Users\DELL\source\repos\L3ISM1\L3ISM1\bin\Debug\L3ISM1.exe
Voici Ma Première Application En C#
```

II.2) Les types de valeur , les types de référence et les variables

En C # ,on distingue 4 catégories types:

A. **Types de valeur:** stockent des données. Les types de valeur se composent de deux catégories principales:

- Les Structures, sont formées :

+Types entiers: **sbyte,byte,char,short,ushort,int,uint,long,ulong**

+Types à virgule flottante: **float,double**

+Type décimal: **decimal**

+Type booléen: **bool**

+Définition de nos Structures: on peut définir une structure pour représenter des **objets légers**.

Pour définir une structure:

```
Definition de Structure
public struct NonStructure
{
    public type champ;
    public type champI;
    .
    .
    public type champn;
}
Declaration de Variable
NonStructure variable=new NonStructure()
Initialisation
variable.champ="Valeur"
```



Exemple

```
public struct Livre
{
    public decimal prix;
    public string titre;
    public string auteur;
}
```

Exemple:


```
Livre monLivre=new Livre()
```

Exemple

```
monLivre.prix=1200;
monLivre.titre="Une Si longue lettre";
monLivre.auteur="Mariama Ba";
```

Exercice d'Application 2 : Structures

Définir la structure **Personne** avec les champs **Prénom** et **Age**. Déclarer une variables ,l'initialiser puis l'afficher comme suit.



```
Modou à 30 an(s)
```

Remarques:

- 1) Les structures peuvent également contenir des constructeurs , des constantes , des champs , des méthodes , des propriétés , des indexeurs , des opérateurs , des événements et des types imbriqués. Cependant , si plusieurs de ces membres sont requis, vous devriez envisager de créer une classe.
- 2) Pour comparer deux variables de type Structures on utilise la Syntaxe **varStruct1.equals(varStruct2)** et on a comme résultat **True si les deux structures ont la même valeur** sinon **False**;

-Enumérations: est un type distinct ,constitué en un ensemble de constantes nommées appelées liste d'énumérateurs. Pour déclarer une énumération, on utilise la syntaxe:

```
enum NomEnum {CONST1, CONST2....., CONSTn};
```

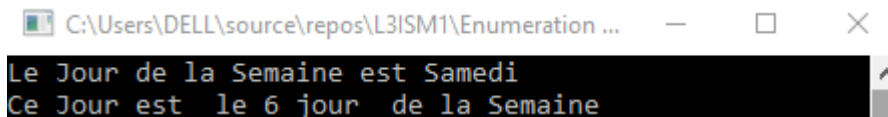
Exemple:

```
enum JourDeLaSemaine {Lundi=1, Mardi,Mercredi,Jeudi,Vendredi, Samedi, Dimanche }
```

NB: Par défaut, le premier énumérateur a la valeur 0 et la valeur de chaque énumérateur successif est augmentée de 1.

Exercice d'Application 3: Enumération

Définir énumération **JourDeLaSemaine** .Déclarer deux constantes, initialiser les puis afficher Les comme la Figure ci-dessous.



```
C:\Users\DELL\source\repos\L3ISM1\Enumeration ...  
Le Jour de la Semaine est Samedi  
Ce Jour est le 6 jour de la Semaine
```

B. Types de référence, stockent des références ou adresses mémoires des données réelles. Une variable de type référence n'est rien d'autre qu'un pointeur vers une donnée réelle. Les variables de type référence sont aussi appelées aussi **Objets**. Les types de référence sont:

-classe, interface, delegate

```
//Definition d'une Classe
class TestClass
{
    // Methodes,
    //propertes,champs,
    //evenements, delegates
}
```

```
//Definition d'une Interface
interface INomInterface
{
    void NomMethod();
}
```

```
//Definition Delagate
public delegate void TestDelegate(string message);
public delegate int TestDelegate(MyType m, long num);
```

Un delegate est un type de référence qui peut être utilisé pour encapsuler une méthode nommée ou anonyme
Les délégués sont la base des événements .

- **dynamic**: permet aux opérations dans lesquelles il se produit de contourner la vérification de type à la compilation .
Déclaration d'un Type Dynamique, **dynamic NomVar= Valeur;**

-**object**: est un alias pour Object dans le .NET Framework. Dans le système de type unifié de C #, tous les types, prédéfinis et définis par l'utilisateur, les types de référence et les types de valeur, héritent directement ou indirectement de Object.

Déclaration d'un Type **object**, **object NomVar= Valeur;**

-**string** : représente une séquence de zéro ou plusieurs caractères Unicode, **string est un alias global de la classe String** (classe du Framework .NET) **en C#**.

Exemple

Déclaration d'un Type **string** , **string NomVar= Valeur;**

string a = "hello"; string b = "h";

On peut créer des alias locales **using str= System.String et pour déclarer une variable on mettra str S;**

NB:

Pour comparer deux variables de type référence on utilise la Syntaxe **varTypeRef1.equals(varTypeRef2)** et on a comme résultat **True si les deux variables pointent sur la même adresse** sinon **False**. **Donc la méthode equals pour les types référence teste les adresses et non les valeurs.**

NB:

1) Le C# est un Langage Fortement Typé c'est-à-dire chaque variable a un type.

Pour connaître le type d'une variable, on peut utiliser **NomVar.GetType().FullName**.

2) Pour connaître si une variable appartient au type de référence ou a un type de valeurs, on peut utiliser respectivement **NomVar.GetType().IsByRef** ou **NomVar.GetType().IsValueType**.

En C#, ils existent les Types pointeurs qui peuvent être utilisés uniquement en mode non sécurisé.

C. Fonctionnement d'un mémoire d'exécution en C#

Un programme en C# est exécuté dans deux zones de la mémoire centrale appelée **Stack** ou **heap**.

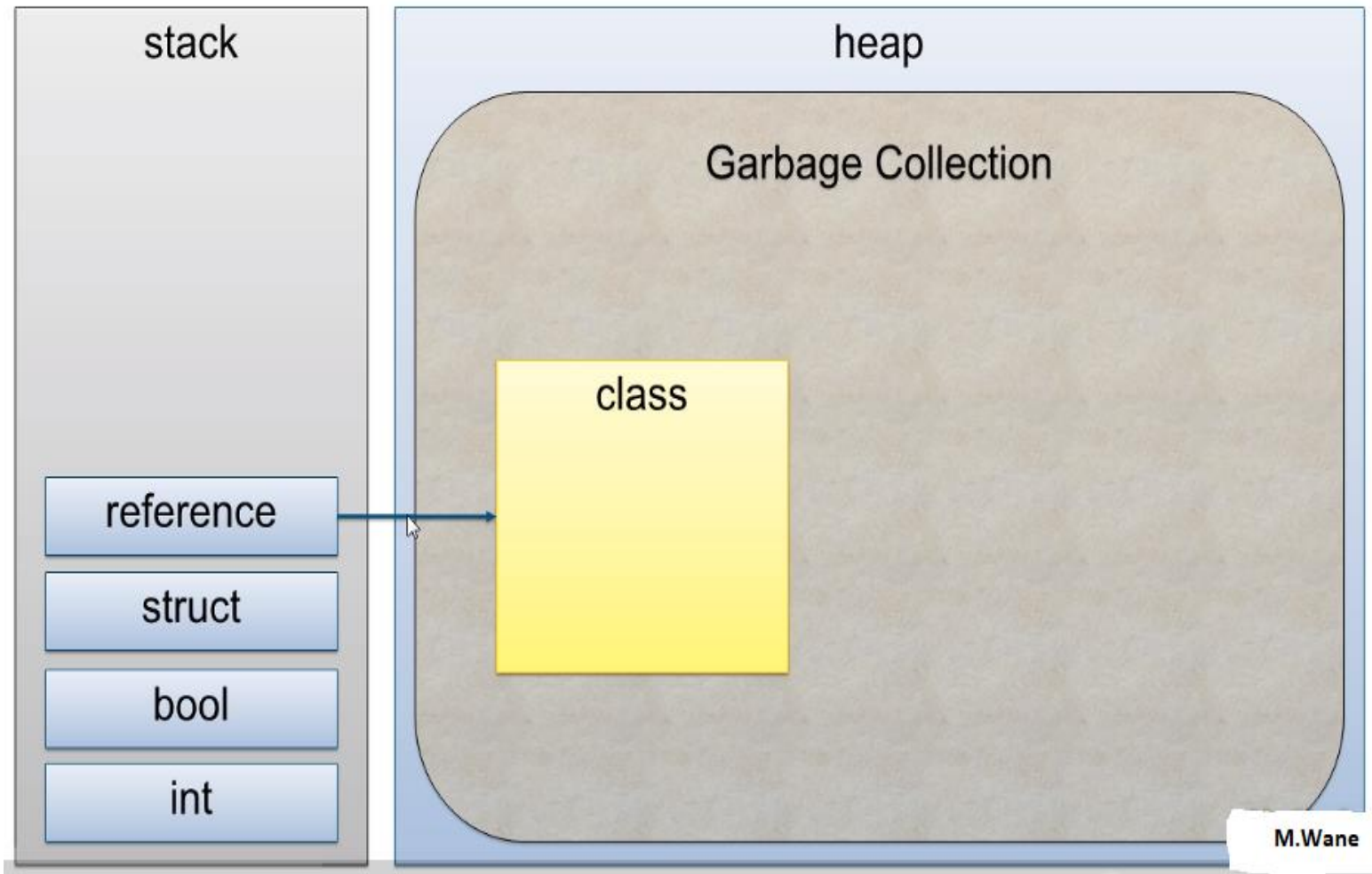
Stack : partie de la Mémoire RAM sous Forme de Pile qui permet de stocker les variables locales d'une fonction lorsqu'elle appelée. Le stack se vide automatique lorsque l'appel de la fonction est terminée. La Limite du slack est de 1Mo en C#.

Un type de Valeur c'est-à-dire qu'une structure est stockée dans le stack alors que pour un type de référence on y stocke la référence ou adresse de la valeur.

heap ou le Tas: partie de la Mémoire RAM plus grande que le stack et élastique qui permet de stocker les valeurs des types références.

Dans les Langages natifs tels que le C ou le C++, c'est aux programmeurs d'allouer ou de libérer les références ou adresses des variables de type référence alors que pour le C# possède un module nommée le **Garbage Collection** qui permet d'effacer toutes les valeurs qui n'ont plus de référence dans le heap.

Division de la Mémoire centrale lors de l' exécution d'un programme en C#



II.3) Fondamentaux de l'orienté objet

La méthode orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux par des relations

□ Lorsque que l'on programme avec cette méthode, la première question que l'on se pose plus souvent est :

□ **«qu'est-ce que je manipule ? »,**

□ **Au lieu de « qu'est-ce que je fait ? ».**

□ L'une des caractéristiques de cette méthode permet de concevoir de nouveaux objets à partir d'objets existants.

On peut donc réutiliser les objets dans plusieurs applications.

□ La réutilisation du code fut un argument déterminant pour venter les avantages des langages à objets.

□ Pour faire la programmation orientée objet il faut maîtriser les fondamentaux de l'orienté objet à savoir:

□ **Propriété, Objet et classe**

□ **Héritage**

□ **Encapsulation (Accessibilité)**

□ **Polymorphisme**

A) Propriété, Objet et classe

A .1) Objet

Un objet est une structure informatique définie par un état et un comportement

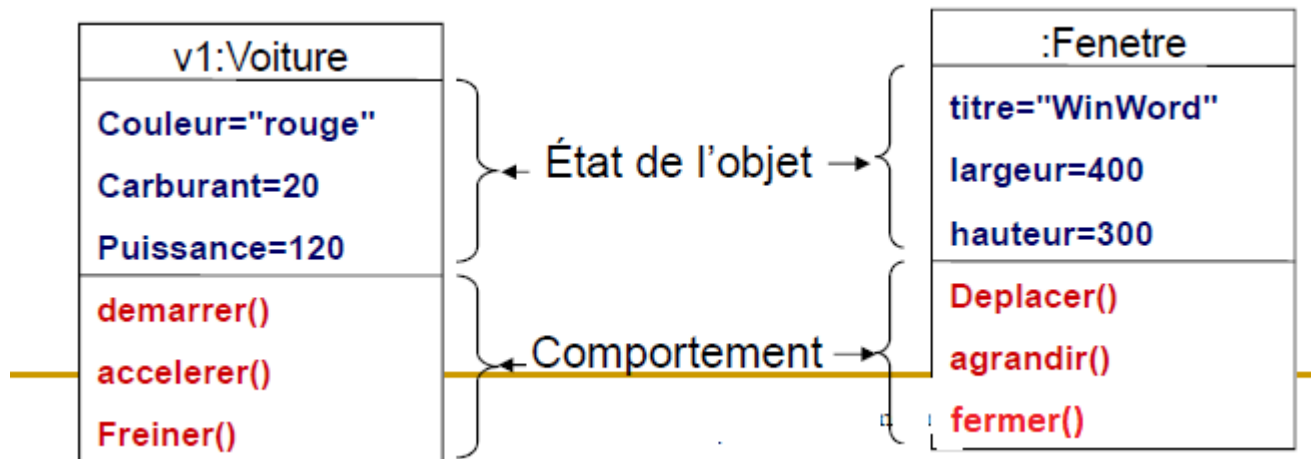
Objet=état + comportement

L'état regroupe les valeurs **a un instant t** de tous les attributs ou propriétés de l'objet.

Le comportement regroupe toutes les compétences et décrit les actions et les réactions de l'objet. Autrement dit le comportement est défini par les opérations que l'objet peut effectuer. **L'état d'un objet peut changer dans le temps.**

Généralement, c'est le comportement qui modifie l'état de l'objet

Exemples:



A .2) Identité d'un objet

En plus de son état, un objet possède une identité qui caractérise son existence propre. Cette identité s'appelle également **référence ou handle** de l'objet

En terme informatique de bas niveau, **l'identité d'un objet représente son adresse mémoire.**

Deux objets ne peuvent pas avoir la même identité : c'est-à-dire que deux objet ne peuvent pas avoir le même emplacement mémoire.

A .3) Classes

- ❑ Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelé classe.
- ❑ La classe décrit le domaine de définition d'un ensemble d'objets.
- ❑ Chaque objet appartient à une classe
- ❑ Les généralités sont contenues dans les classe et les particularités dans les objets.
- ❑ Les objets informatique sont construits à partir de leur classe par un processus qui s'appelle l'instanciation .Tout objet est une instance d'une classe.

A .4) Caractéristique d'une classe

- ❑ Une classe est définie par:
 - **Les attributs**
 - **Les propriétés**
 - ❑ - **Les méthodes**
- ❑ Les attributs permettent de décrire l'état de des objets de cette classe. Chaque attribut est défini par:
 - Son nom**
 - Son type**
 - Éventuellement sa valeur initiale**
- ❑ Les propriétés se comportent comme des attributs lors de leur accès. Cependant, contrairement aux attributs , les propriétés sont implémentées avec des accesseurs qui définissent les instructions exécutées lorsqu'une propriété est accédée ou attribuée. Les méthodes permettent de décrire le comportement des objets de cette classe .
 - Une méthode représente une procédure ou une fonction qui permet d'exécuter un certain nombre d'instructions.
- ❑ Parmi les méthodes d'une classe, existe deux méthodes particulières:
 - ❑ - Une méthode qui est appelée au moment de la création d'un objet de cette classe. Cette méthode est appelée **CONSTRUCTEUR**.

A.5) Représentation C# d'une classe

Une classe est représentée par un rectangle à 4 compartiments:

- **Un compartiment qui contient le nom de la classe**
- **Un compartiment qui contient la déclaration des attributs**
- **Un compartiment qui contient la déclaration des propriétés**
- **Un compartiment qui contient les méthodes**

A.6) Accessibilité aux membres d'une classe

Dans Java, il existe 4 niveaux de protection :

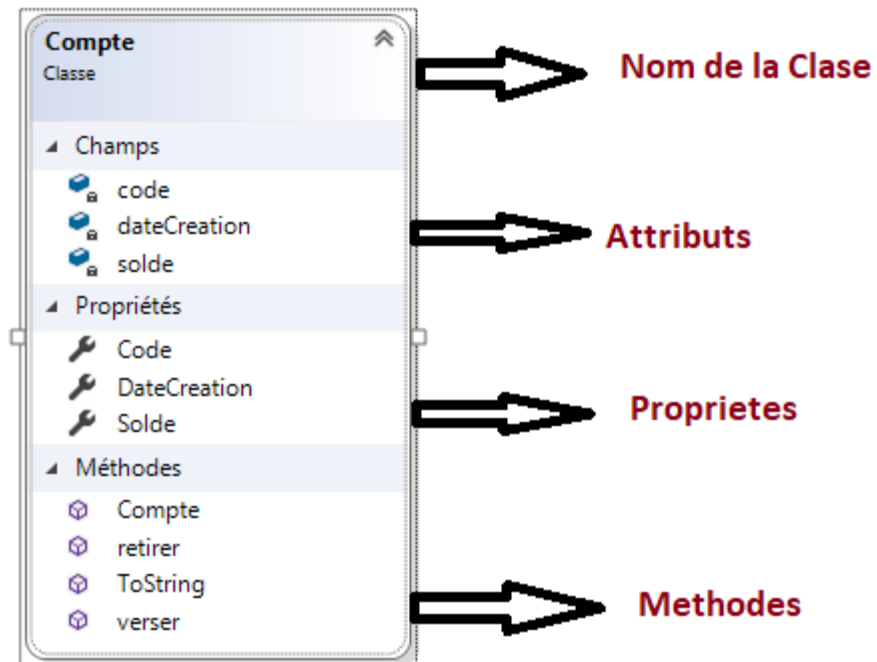
- ❑ **private** : Un membre privé d'une classe n'est accessible qu'à l'intérieur de cette classe.
- ❑ **protected** : un membre protégé d'une classe est accessible à :
 - L'intérieur de cette classe
 - Aux classes dérivées de cette classe.
- ❑ **Internal**: Un membre internal d'une classe est accessible à :
 - L'intérieur de cette classe
 - Aux classes qui appartiennent au même namespace
- ❑ **public** : accès à partir de toute entité interne ou externe à la classe

NB:

Par défaut l'accessibilité des membres d'une classe est:

- **Un attribut est a private**
- **Une propriété ou une méthode est a public**

Exemples: Soit le diagramme classe compte ci-dessous



Exercice d'application: Définir la classe ci-dessus

Implémentation de la Classe Compte

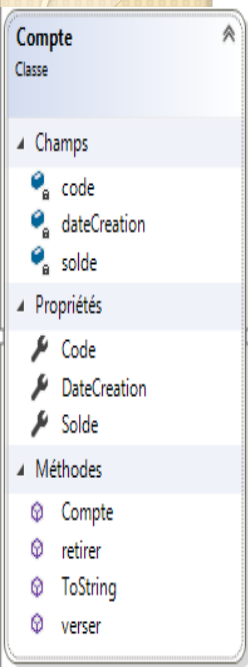


Diagramme de Classe

Definition de la Classe

```
namespace P00
{
    class Compte
    {
        //Attributs
        private int code;
        private float solde;
        private DateTime dateCreation;
        //Proprietés
        public int Code { get; private set ; }
        public float Solde { get => solde; set => solde = value; }
        public DateTime DateCreation { get => dateCreation; set => dateCreation = value; }

        // Constructeur
        public Compte(float solde, DateTime dateCreation)
        {
            this.solde = solde;
            this.dateCreation = dateCreation;
        }

        // Méthode pour verser un montant
        public void verser(float mnt)
        {
            Solde += mnt;
        }

        // Méthode pour retirer un montant
        public void retirer(float mnt)
        {
            if (Solde >= mnt)
                Solde -= mnt;
        }

        // Une méthode qui retourne l'état du compte

        public override String ToString()
        {
            return (" Code=" + Code + " Solde=" + Solde);
        }
    }
}
```

Program d'exécution

```
namespace P00
{
    class Program
    {
        static void Main(string[] args)
        {
            //Initialisation du Compte
            Compte c1 = new Compte(12000, new DateTime());

            Console.WriteLine("Avant Versement {0} ", c1.ToString());
            c1.verser(1000);
            Console.WriteLine("Après Versement {0} ", c1.ToString());

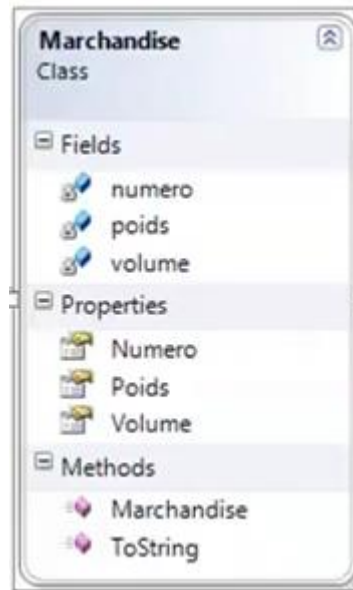
            //Erreur car le set du Code est a private
            c1.Code = 1;

            Console.ReadKey();
        }
    }
}
```

Resultat Execution

```
C:\Users\DELL\source\repos\L3ISM1\Classes\bin\Debug\Classes.exe
Avant Versement Code=0 Solde=12000
Après Versement Code=0 Solde=13000
```

Exercice 1 : soit le Diagramme de classe, définir la classe correspondante



A .7) Membres d'instances ou statiques d'une classe

❑ Membres d'instances d'une classe

Dans l'exemple de la classe Compte, chaque objet Compte possède ses propres variables **code**, **dateCreation** et **solde**.

Les variables **code**, **dateCreation** et **solde** sont appelées **variables d'instances**.

❑ Membres statiques d'une classe

Les objets d'une même classe peuvent partager des mêmes variables qui sont stockées au niveau de la classe. Ce genre de variables, s'appellent **les variables statiques ou variables de classes** .

Un attribut statique d'une classe est un attribut qui appartient à la classe et partagé par tous les objets de cette classe.

Comme un attribut, **une méthode peut être déclarée statique**, ce qui signifie qu'elle appartient à la classe et partagée par toutes les instances de cette classe.

Exercice d'application:

- Ajouter la propriété **nbreDecompteCree** dans la classe Compte comme un membre statique.
- Définir une méthode statique **getNbreCompte()** qui retourne le nombre de comptes crée
- Utiliser la variable static **nbreDecompteCree** pour initialiser le code du compte

• Correction de l'exercice d'application

```
class Compte
{
    //Attributs
    private static int nbreCompteCree;
    private int code;
    private float solde;
    private DateTime dateCreation;
    //Proprietes
    public int Code { get; private set ; }
    public float Solde { get => solde; set => solde = value; }
    public DateTime DateCreation { get => dateCreation; set => dateCreation = value; }

    // Constructeur
    public Compte(float solde, DateTime dateCreation)
    {
        this.code= ++Compte.nbreCompteCree;
        this.solde = solde;
        this.dateCreation = dateCreation;
    }

    // Méthode qui retourne le nombre de Compte Cree
    public static int getNbreCompte()
    {
        return nbreCompteCree;
    }
}
```

Programme d'execution

```
class Program
{
    static void Main(string[] args)
    {
        //Initialisation du Compte
        Compte c1 = new Compte(6000, new DateTime());
        Console.WriteLine("Le Nombre de Comptes Cree est {0} ", Compte.getNbreCompte());
        Compte c2 = new Compte(6000, new DateTime());

        Console.WriteLine("Le Nombre de Comptes Cree est {0} ", Compte.getNbreCompte());

        Console.ReadKey();
    }
}
```

Execution

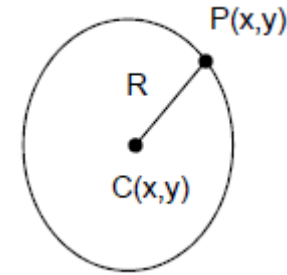
C:\Users\DELL\source\repos\L3ISM1\Classes\bin\Debug\Classes.exe
 Le Nombre de Comptes Cree est 1
 Le Nombre de Comptes Cree est 2

Classe Compte
<u>nbreCompteCree=2</u>
getNbreCompte()

c1:Compte	c2:Compte
code=1	code=2
solde=6000	solde=6000
verser(float mt)	verser(float mt)
retirer(float mt)	retirer(float mt)
toString()	toString()

Exercice 2

- ❑ Un cercle est défini par :
 - Un point qui représente son centre : centre(x , y) et un rayon.
- ❑ On peut créer un cercle de deux manières :
 - Soit en précisant son centre et un point du cercle.
 - Soit en précisant son centre et son rayon
- ❑ Les opérations que l'on souhaite exécuter sur un cercle sont :
 - **getPerimetre()** : retourne le périmètre du cercle ($P=2*\pi*R$).
 - **getSurface()** : retourne la surface du cercle ($S=2*R*R$).
 - **appartient(Point p)** : retourne si le point p appartient ou non à l'intérieur du cercle.
 - **ToString()** : retourne une chaîne de caractères de type CERCLE(x , y , R)



1. Etablir le diagramme de classes

2. Créer la classe Point définie par:

- Les attributs x et y de type int
- Un constructeur qui initialise les valeurs de x et y.
- Une méthode toString().

3. Créer la classe Cercle

4. Créer une application qui permet de :

- a. Créer un cercle défini par le centre c(100,100) et un point p(200,200)
- b. Créer un cercle défini par le centre c(130,100) et de rayon r=40
- c. Afficher le périmètre et le rayon des deux cercles.
- d. Afficher si le point p(120,100) appartient à l'intersection des deux cercles ou non.

B) Héritage, Polymorphisme et accessibilité

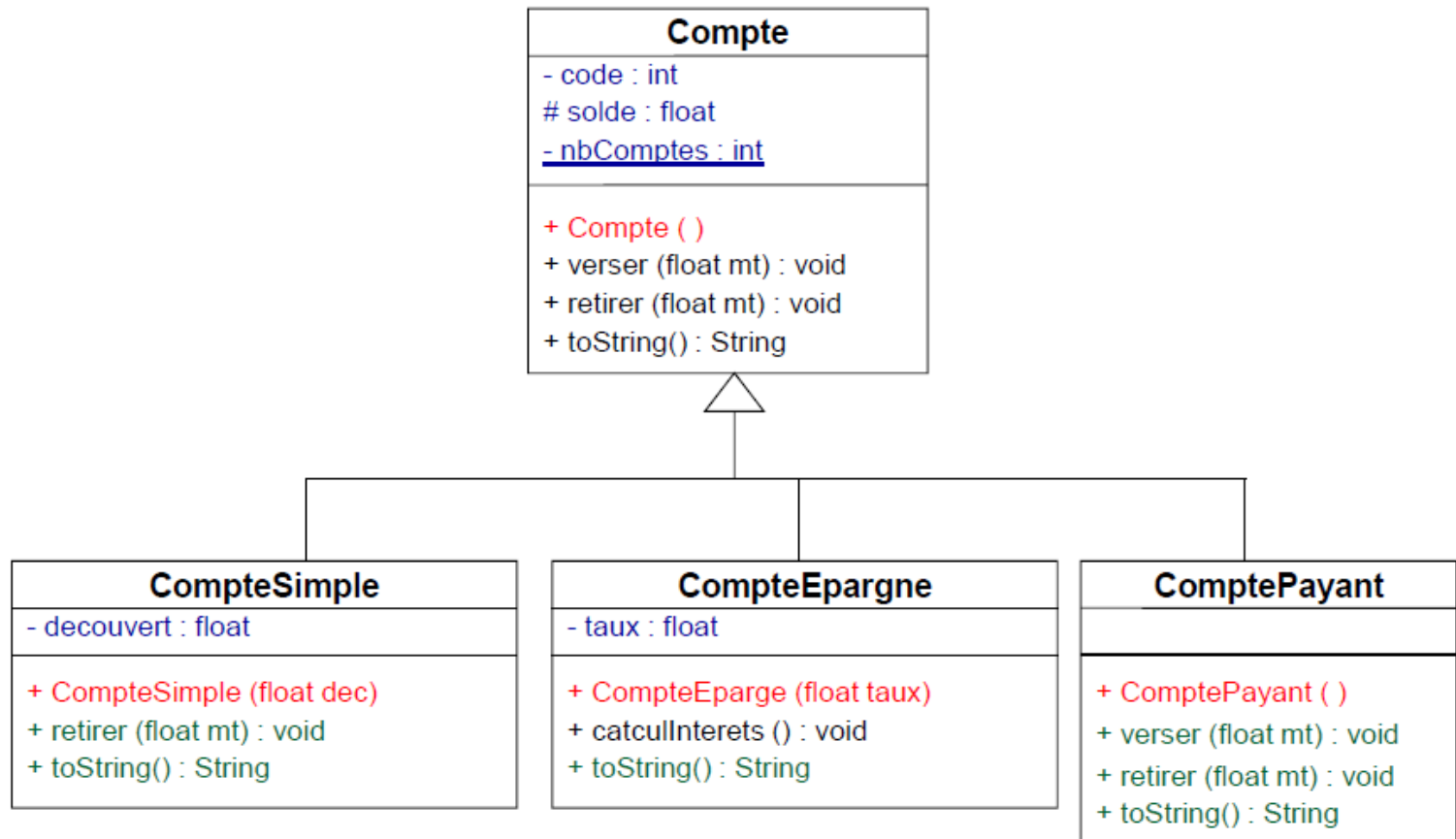
B .I) Héritage

- ❑ Dans la programmation orientée objet, l'héritage offre un moyen très efficace qui permet la réutilisation du code.
- ❑ En effet une classe peut hériter d'une autre classe des attributs et des méthodes.
- ❑ L'héritage, quand il peut être exploité, fait gagner beaucoup de temps en terme de développement et en terme de maintenance des applications.
- ❑ La réutilisation du code fut un argument déterminant pour venter les méthodes orientées objets.

Exemple de problème

- ❑ Supposons que nous souhaitons créer une application qui permet de manipuler différents types de comptes bancaires: **les compte simple, les comptes épargnes et les comptes payants.**
- ❑ Tous les types de comptes sont caractériser par:
 - Un code , un solde et la date de creation
 - Lors de la création d'un compte, son code qui est défini automatiquement en fonction du nombre de comptes créés;
 - Un compte peut subir les opérations de versement et de retrait. Pour ces deux opérations, il faut connaître le montant de l'opération.
 - Pour consulter un compte on peut faire appel à sa méthode toString()
- ❑ Un **compte simple** est un compte qui possède un **découvert**. Ce qui signifie que ce compte peut être débiteur jusqu'à la valeur du découvert.
- ❑ Un **compte Epargne** est un compte bancaire qui possède en plus un champ «**tauxInterêt**» et une méthode **calculIntérêt()** qui permet de mettre à jour le solde en tenant compte des intérêts.
- ❑ Un **ComptePayant** est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 5 % du montant de l'opération.

Diagramme de classes



Implémentation java de la classe Compte

namespace POO

```
{
    class Compte
    {
        //Attributs
        private static int nbreCompteCree;
        private int code;
        private float solde;
        private DateTime dateCreation;
        //Proprietes
        public int Code { get; private set ; }
        public float Solde { get => solde; set => solde = value; }
        public DateTime DateCreation { get => dateCreation; set => dateCreation = value; }
        // Constructeur
        public Compte(float solde, DateTime dateCreation)
        {
            this.code= ++Compte.nbreCompteCree;
            this.solde = solde;
            this.dateCreation = dateCreation;
        }

        // Méthode qui retourne le nombre de Compte Cree
        public static int getNbreCompte()
        {
            return nbreCompteCree;
        }
        // Méthode pour verser un montant
        public void verser(float mnt)
        {
            Solde += mnt;
        }

        // Méthode pour retirer un montant
        public void retirer(float mnt)
        {
            if (Solde>= mnt)
                Solde -= mnt;
        }
        // Une méthode qui retourne l'état du compte
        public override String ToString()
        {
            return (" Code=" + Code + " Solde=" + Solde);
        }
    }
}
```

Héritage

- ❑ La classe **CompteSimple** est une classe qui hérite de la classe **Compte**. Pour désigner l'héritage dans java, on utilise deux points (:) .

```
class CompteSimple : Compte {  
}
```

- ❑ La classe **CompteSimple** hérite de la classe **Compte** tout ses membres sauf le constructeur.
- ❑ Dans **C#** une classe hérite toujours d'une seule classe.
- ❑ Si une classe n'hérite pas explicitement d'une autre classe, elle hérite implicitement de la classe **Object**.
- ❑ La classe **Compte** hérite de la classe **Object**.
- ❑ La classe **CompteSimple** hérite directement de la classe **Compte** et indirectement de la classe **Object**.
- ❑ La classe **Compte** est appelée classe **Mère ou parente**
- ❑ La classe **CompteSimple** est appelée classe **Fille ou dérivée**

B .2) Définir les constructeur de la classe dérivée

- ❑ Le constructeur de la classe dérivée peut faire appel au constructeur de la classe parente en utilisant le mot **base()** suivi de ses paramètres.
- ❑ Les propriétés héritées par les classes filles doivent être déclarées dans la classe mère avec l'accessibilité **protected**.

```
namespace P00
{
    class CompteSimple:Compte
    {
        private float decouvert;
        public float Decouvert { get => decouvert; set => decouvert = value; }

        public CompteSimple(float solde, DateTime dateCreation,float decouvert):base(solde, dateCreation)
        {
            this.decouvert = decouvert;
        }

        public CompteSimple()
        {
        }
    }
}
```

B .3) Redéfinition des méthodes

- ❑ Quand une classe hérite d'une autre classe, elle peut redéfinir les méthodes héritées.
- ❑ Dans notre cas la classe **CompteSimple** hérite de la classe **Compte** la méthode **retirer()**. Nous avons besoin de redéfinir cette méthode pour prendre en considération la valeur du découvert.
- ❑ Dans la classe **Compte**, on va précéder la méthode **retirer()** du mot clé **virtual** pour indiquer que cette méthode doit être redéfinie.
- ❑ Dans la classe **CompteSimple**, on va précéder la méthode **retirer()** du mot clé **override** pour indiquer que cette méthode est redéfinie.
- ❑ Dans la méthode redéfinie de la nouvelle classe dérivée, on peut faire appel à la méthode de la classe parente en utilisant le mot **base** suivi d'un point et du nom de la méthode.
- ❑ Dans cette nouvelle classe dérivée, nous allons redéfinir également la méthode **toString()**.


```
class Compte
```

```
{  
    //Attributs  
    private static int nbreCompteCree;  
    private int code;  
    protected float solde;  
    protected DateTime dateCreation;  
    //Propriétés  
    public int Code { get; private set ; }  
    public float Solde { get => solde; set => solde = value; }  
    public DateTime DateCreation { get => dateCreation; set => dateCreation = value; }  
    // Constructeur  
    public Compte(float solde, DateTime dateCreation)  
    {  
        this.code= ++Compte.nbreCompteCree;  
        this.solde = solde;  
        this.dateCreation = dateCreation;  
    }  
  
    public Compte()  
    {  
        this.code = ++Compte.nbreCompteCree;  
    }  
  
    // Méthode pour retirer un montant  
    public virtual void retirer(float mnt)  
    {  
        if (Solde >= mnt)  
            Solde -= mnt;  
    }  
}
```

Heritage

```
namespace P00
```

```
{  
    class CompteSimple:Compte
```

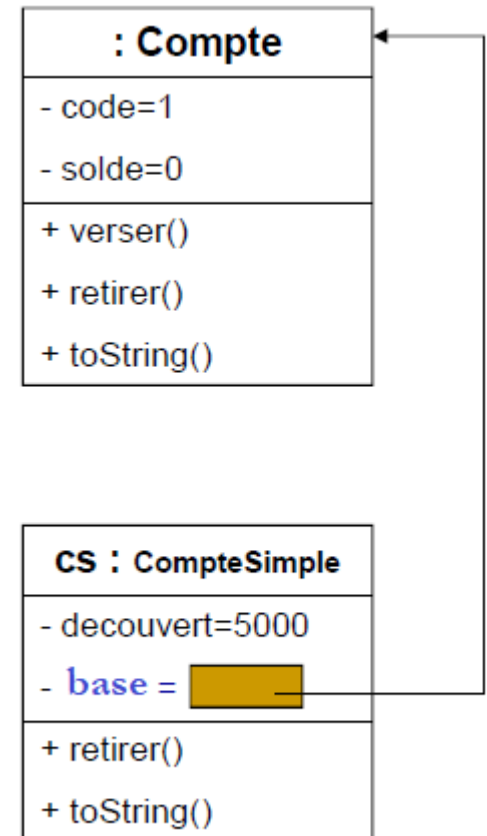
```
{  
    private float decouvert;  
    public float Decouvert { get => decouvert; set => decouvert = value; }  
  
    public CompteSimple(float solde, DateTime dateCreation, float decouvert):base(solde, dateCreation)  
    {  
        this.decouvert = decouvert;  
    }  
  
    public CompteSimple(float decouvert)  
    {  
        this.decouvert = decouvert;  
    }  
  
    // Redéfinition de la méthode retirer  
    public override void retirer(float mnt)  
    {  
        if (mnt - decouvert <= solde)  
            solde -= mnt;  
    }  
  
    public override string ToString()  
    {  
        return ("Compte Simple " + base.ToString() + "Découvert = " + Decouvert);  
    }  
}
```

B .4) Héritage à la loupe : Instanciation

❑ Quand on crée une instance d'une classe, la classe parente est automatiquement instanciée et l'objet de la classe parente est associé à l'objet créé à travers la référence « **base** » injectée par le compilateur.

CompteSimple cs= new CompteSimple(5000);

❑ Lors de l'instanciation, l'héritage entre les classes est traduit par une composition entre un objet de la classe instanciée et d'un objet de la classe parente qui est créé implicitement.



B .5) Surcharge

- ❑ Dans une classe, on peut définir plusieurs constructeurs. Chacun ayant une signature différentes (paramètres différents)
- ❑ On dit que le constructeur est surchargé
- ❑ On peut également surcharger une méthode. Cela peut dire qu'on peut définir, dans la même classe plusieurs méthodes qui ont le même nom et des signatures différentes;
- ❑ La signature d'une méthode désigne la liste des arguments avec leurs types.
- ❑ Dans la classe CompteSimple, par exemple, on peut ajouter un autre constructeur sans paramètre

```
namespace P00
{
    class CompteSimple:Compte
    {
        private float decouvert;
        public float Decouvert { get => decouvert; set => decouvert = value; }

        public CompteSimple(float solde, DateTime dateCreation, float decouvert):base(solde, dateCreation)
        {
            this.decouvert = decouvert;
        }

        public CompteSimple(float decouvert)
        {
            this.decouvert = decouvert;
        }

        public CompteSimple()
        {
        }
    }
}
```

↑ Constructeur avec argument surchargé

← Constructeur sans argument ou par défaut

On peut créer une instance de la classe CompteSimple en faisant appel à l'un des trois constructeur :

```
CompteSimple cs1=new CompteSimple(5000,30000,new DateTime());
CompteSimple cs2=new CompteSimple(5000);
CompteSimple cs3=new CompteSimple();
```

B .6) Polymorphisme

- ❑ Le polymorphisme offre aux objets la possibilité d'appartenir à plusieurs catégories à la fois.

- ❑ on peut écrire l'expression suivante:

3 CompteSimple + 2 CompteEpargne + 3 CompteBancaire = 8 Compte

- ❑ Le sur-casting des objets:

Une façon de décrire l'exemple consistant à additionner des **CompteSimple**, des **CompteEpargne** et des **CompteBancaire** serait d'imaginer que nous disons **CompteSimple**, des **CompteEpargne** et des **CompteBancaire** mais que nous manipulons en fait des **Compte**.

Nous pourrions écrire alors la formule correcte :

3 (Compte) CompteSimple

+

2 (Compte) CompteEpargne

+

3 (Compte) CompteBancaire

= 8 Compte

Cette façon de voir les choses implique que les disons **CompteSimple**, des **CompteEpargne** et des **CompteBancaire** soient "transformés" en **Compte** préalablement à l'établissement du problème. Cette transformation est appelée **sur-casting**.

Compte C1=new CompteSimple();

Compte C2=new CompteEpargne();

Compte C3=new ComptePayant();

```

namespace P00
{
    class Compte
    {
        //Attributs
        private static int nbreCompteCree;
        private int code;
        protected float solde;
        protected DateTime dateCreation;
        //Proprietes
        public int Code { get; private set; }
        public float Solde { get => solde; set => solde = value; }
        public DateTime DateCreation { get => dateCreation; set => dateCreation = value; }
        // Constructeur
        public Compte(float solde, DateTime dateCreation)
        {
            this.code= ++Compte.nbreCompteCree;
            this.solde = solde;
            this.dateCreation = dateCreation;
        }

        public Compte()
        {
            Console.WriteLine("Creation d'un Compte");
            this.code = ++Compte.nbreCompteCree;
        }
        public virtual void affiche()
        {
            Console.WriteLine("C'est un compte");
        }
    }
}

```

Definition CompteEpargne

```

namespace P00
{
    class CompteEpargne : Compte
    {
        public CompteEpargne()
        {
            Console.WriteLine("Creation d'un Compte Epargne");
        }

        public override void affiche()
        {
            Console.WriteLine("C'est un compte Epargne");
        }
    }
}

```

Definition d'un Compte Bancaire

```

namespace P00
{
    class CompteBancaire:Compte
    {
        public CompteBancaire()
        {
            Console.WriteLine("Creation d'un Compte Bancaire");
        }

        public override void affiche()
        {
            Console.WriteLine("C'est un compte Bancaire");
        }
    }
}

```

Execution du Programme

```

namespace P00
{
    class Program
    {
        static void Main(string[] args)
        {
            //Initialisation du Compte
            Compte c1 = new Compte();
            c1.affiche();
            Console.WriteLine("-----");
            Compte c2 = new CompteSimple();
            c2.affiche();
            Console.WriteLine("-----");
            Compte c3 = new CompteEpargne();
            c3.affiche();
            Console.WriteLine("-----");
            Compte c4 = new CompteBancaire();
            c4.affiche();

            Console.ReadKey();
        }
    }
}

```

Definition d'un Compte Simple

```

namespace P00
{
    class CompteSimple:Compte
    {
        private float decouvert;
        public float Decouvert { get => decouvert; set => decouvert = value; }

        public CompteSimple(float solde, DateTime dateCreation, float decouvert):base(solde, dateCreation)
        {
            this.decouvert = decouvert;
        }

        public CompteSimple(float decouvert)
        {
            this.decouvert = decouvert;
        }

        public CompteSimple()
        {
            Console.WriteLine("Creation d'un Compte Simple");
        }

        public override void affiche()
        {
            Console.WriteLine("C'est un compte Simple");
        }
    }
}

```

- ❑ Un objet de type CompteSimple peut être affecté à un handle de type Compte sans aucun problème :

Compte C1;

C1 =new CompteSimple ();

- ❑ Dans ce cas l'objet CompteSimple est converti automatiquement en Compte .
- ❑ On dit que l'objet CompteSimple est sur-casté en Compte .
- ❑ Dans C#, le sur-casting peut se faire implicitement.
- ❑ Toutefois, on peut faire le sur-casting explicitement sans qu'il soit nécessaire.
- ❑ La casting explicit se fait en précisant la classe vers laquelle on convertit l'objet entre parenthèse. Exemple :
Compte C2 =(Compte)new CompteSimple ();

```

7 namespace P00
8 {
9     class Program
10    {
11        static void Main(string[] args)
12        {
13            //Initialisation du Compte
14            Compte c1 = new Compte();
15            c1.affiche();
16            Console.WriteLine("-----");
17
18            Compte c2 = (Compte) new CompteSimple();
19            c2.affiche();
20            c2.affiche1();
21            ((CompteSimple)c2).affiche1();
22
23            Console.ReadKey();
24        }
25    }
26 }

```

sur-casting

Pas de Probleme la methode affiche se dans la classe **Compte**

Probleme car la Methode affiche l () ne se trouve pas dans **Compte**

Pour corriger on utilise le **Sous Casting**

Sous-casting des objets

- ❑ Ce message indique que l'objet C2 qui est de type **Compte** ne possède pas la méthode **affiche1()**.
- ❑ Cela est tout à fait vrai car cette méthode est définie dans la classe **CompteSimple** non dans la classe **Compte**.
- ❑ En fait, même si le handle C2 pointe sur un objet **CompteSimple**, mais le compilateur ne tient pas en considération cette affectation, et pour lui C2 est un **Compte**.
- ❑ Il faudra donc convertir explicitement l'objet C2 qui est de type **Compte** en **CompteSimple**.

- ❑ Cette conversion s'appelle **Sous-casting** qui indique la **conversion d'un objet d'une classe Mère vers un autre objet d'une classe Fille**.
- ❑ Dans ce cas de figure, le sous-casting doit se faire explicitement.
- ❑ L'erreur de compilation peut être évité en écrivant la syntaxe suivante :

((CompteSimple)c2).afficheI();

- ❑ Cette instruction indique que l'objet **C2** , de type **Compte** , est converti en **CompteSimple**, ensuite la méthode **afficheI()** de l'objet **CompteSimple** est appelé ce qui est correcte.

B .7) Accessibilité

- ❑ Les trois critères permettant d'utiliser une classe sont **Qui, Quoi, Où**. Il faut donc :
 - Que l'utilisateur soit autorisé (*Qui*).
 - Que le type d'utilisation souhaité soit autorisé (*Quoi*).
 - Que l'adresse de la classe soit connue (*Où*).
- ❑ Pour utiliser donc une classe, il faut :
 - Connaitre le namespace ou se trouve la classe (*Où*)
 - Importer la classe en spécifiant son namespace .
 - Qu'est ce qu'on peut faire avec cette classe:
 - Est-ce qu'on a le droit de l'instancier
 - Est-ce qu'on a le droit d'exploiter les membres de ses instances
 - Est-ce qu'on a le droit d'hériter de cette classe.
 - Est-ce qu'elle contient des membres statiques
 - Connaitre qui a le droit d'accéder aux membres de cette instance.

C) Mot clé abstraite et sealed

C.1) Les classes abstraites

- ❑ Une **classe abstraite** est une classe qui ne peut pas être instanciée.
- ❑ La classe Compte de notre modèle peut être déclarée **abstract** pour indiquer au compilateur que cette classe ne peut pas être instancié.
- ❑ Une classe abstraite est généralement créée pour en faire dériver de nouvelle classe par héritage.

abstract class Compte {

//Attributs

private static int nbreCompteCree;

private int code;

protected float solde;

protected DateTime dateCreation;

// Propriétés

// Constructeurs

// Méthodes

}

~~Compte c1=new Compte();~~//Interdit car la class compte est **abstract**

C .2) Les méthodes abstraites

- ❑ Une méthode abstraite peut être déclarée à l'intérieur d'une classe abstraite.
- ❑ Une méthode abstraite est une méthode qui n'a pas de définition.
- ❑ Une méthode abstraite est une méthode qui **doit être redéfinie dans les classes dérivées**.
- ❑ Exemple :

On peut ajouter à la classe Compte une méthode abstraite nommée **afficher()** pour indiquer que **tous les comptes doivent redéfinir cette méthode**.

Les méthodes abstraites

```
abstract class Compte {  
    // Membres  
    ...  
    // Méthode abstraite  
    public abstract void afficher();  
}
```

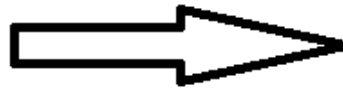
```
class CompteSimple : Compte {  
    // Membres  
    ...  
    public void afficher() {  
        Console.WriteLine("Solde="+solde+"  
        Découvert="+decouvert);  
    }  
}
```

C.3) Les classes sealed

- ❑ Une classe scellée est une classe qui ne peut pas être héritée.

Exemple:

```
sealed class Classesealed {  
  //Attributs  
  // Propriétés  
  // Constructeurs  
  // Méthodes  
}
```



```
class compteFille:Classesealed {  
  //Attributs  
  // Propriétés  
  // Constructeurs  
  // Méthodes  
}
```

Heritage interdit

C.4) Les Méthodes sealed

- ❑ Une Méthode scellée est une méthode qui ne peut pas être redéfinie.

Exercice 2 : soit le Diagramme de classe, définir la classe correspondante

- ☐ Implémenter la classe Compte
- ☐ Implémenter la classe CompteSimple
- ☐ Implémenter la classe CompteEpargne
- ☐ Implémenter la classe ComptePayant
- ☐ Créer une application pour tester les
- ☐ différentes classes

D) Interface, Polymorphisme

D.1) Interface

- ❑ Une interface est une sorte de classe abstraite qui ne contient que des méthodes abstraites.
- ❑ En C# une classe hérite d'une seule classe et peut hériter en même temps de plusieurs interfaces.
- ❑ On dit qu'une classe implémente une ou plusieurs interface.
- ❑ Une interface peut hériter de plusieurs interfaces.

Exemple d'interface:

```
public interface Solvable {  
    public void solver();  
    public double getSolde();  
}
```

Pour indiquer que la classe CompteSimple implémente cette interface on peut écrire:

```
public class CompteSimple : Compte ,Solvable {  
    private float decouvert;  
    public void afficher() {  
        Console.WriteLine("Solde={0} Découvert={1} " ,solde ,decouvert);  
    }  
    public double getSolde() {  
        return solde;  
    }  
    public void solver() {  
        this.solde=0;  
    }  
}
```

III) Bases de la syntaxe

III.1) Les Structures Conditionnelles if ,Switch

❑ Les Structures Conditionnelles if

if (condition1)

Instruction à exécuter si la condition1 est vraie.

else if (condition2)

Instruction à exécuter si la condition2 est vraie.

else if (condition3)

Instruction à exécuter si la condition3 est vraie.

...

...

else

Instruction à exécuter si les conditions 1, 2, et 3 sont toutes les trois fausses.

❑ Les Structures Conditionnelles Switch

```
switch (caseSwitch) {  
case Valeur1:  
    Console.WriteLine("Case 1"); break;  
case Valeur2:  
    Console.WriteLine("Case 2"); break;  
default:  
    Console.WriteLine("Default case"); break;  
}
```

Remarques

- ❑ En C # 6, l'expression de correspondance **caseSwitch** doit être une expression qui renvoie une valeur des types suivants:
char ou string ou un bool ou valeur entier(int long) ou une valeur enum .
- ❑ À partir de C # 7, l'expression de correspondance **caseSwitch** peut être n'importe quelle expression **non nulle**.

III.2) Les boucle while ,for ,foreach

❑ La boucle while

Instruction D'initialisation

while (condition I) {

Instruction à exécuter si la condition I est vraie

}

❑ La boucle for

for (initialisation; condition; iterator) {

Instruction à exécuter si la condition est vraie

}

❑ La boucle foreach

L'instruction foreach répète un groupe d'instructions incorporées pour chaque élément d'un tableau ou d'une collection d'objets qui implémente l'interface

System.Collections.IEnumerable ou **System.Collections.Generic.IEnumerable<T>**.

foreach (type element in Collection) {

System.Console.WriteLine(element);

}

III.2) Les tableaux

❑ Déclaration

- Une Dimension

type[] NomTableau;

- Deux Dimension

type[,]NomTableau;

❑ Initialisation

NomTableau= new type [Taille1];

NomTableau = new type [Taille1, Taille2];

Exemples:

I) int[] scores ;

scores =new int[4] ;

Ou

int[] scores =new int[] { 97, 92, 81, 60 };

II) int[,] scores = new int [5, 10];

Ou

int[,] scores = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };

III)

Compte [] TabCompte=new Compte [5];

ou

Compte [] TabCompte=new Compte [] {

{new CompteSimple() , new CompteEpargne() };

III.3) Les opérateurs en C#

❑ Les opérateurs arithmétiques :

+ - * / %

❑ Les opérateurs d'affectation et d'affectation composée :

= += -= *= /= . =

❑ Les opérateurs d'incrémentation et de décrémentation

++ (a++ ou ++a)

-- (a -- ou -- a)

❑ Les opérateurs relationnels

== === != <> !== > >= < <=

❑ Les opérateurs logiques

&& or

❑ L'opérateur ternaire "?" est une alternative

Syntaxe

Condition ? Expression1 :Expression2

Si la condition est vraie on exécute l'Expression1 sinon Expression2

❑ Opérateurs conditionnels Null

Ces opérateurs sont utilisés pour rechercher les valeurs Null avant d'effectuer une opération d'accès au membre (?) ou d'indexation (?[]). Ils permettent d'écrire moins de code pour gérer les vérifications Null, notamment pour l'exploration des structures de données.

Syntaxe Type ? VariableResult=Variable ?.membre

Exemple : Soit le compte C2

int? code= C2?.Code // null if C2 est null

IV) Les classes du FCL (Framework Class Library)

Les classes du **FCL** représentent les classes de bases du Framework .Net

❑ Les string

Une chaîne est un objet de type String dont la valeur est du texte. En interne, le texte est stocké sous la forme d'une collection séquentielle en lecture seule d'objets Char. Il n'existe aucun caractère de fin Null à la fin d'une chaîne C# ; par conséquent, une chaîne C# peut contenir n'importe quel nombre de caractères Null incorporés ("\0").

Le type string est une chaîne immuable c'est-à-dire non modifiable et dispose d'un ensemble de méthodes statiques ou d'instances.

méthodes statiques

String.méthode

méthodes d'instances

VariableString.méthode

Remarques : On peut utiliser des chaînes Littéraires en précédant la chaîne du caractère @. Les chaînes Littéraires seront affichés sans modification

Exemple

```
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";
```

❑ Les Génériques

Les génériques ont été ajoutés à la version 2.0 du langage C # et au Common Language Runtime (CLR). Les génériques introduisent dans le .NET Framework le concept des paramètres de type, qui permettent de concevoir des classes et des méthodes qui diffèrent la spécification d'un ou plusieurs types jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code client. Par exemple, en utilisant un paramètre de **type générique T**, vous pouvez écrire une seule classe qu'un autre code client peut utiliser sans encourir le coût ou le risque de lancements d'exécution ou d'opérations de boîte, comme indiqué ici:

// Déclaration de la classe Générique

public class GenericList<T>

{

void Add(T input) { }

}

class TestGenericList

{

private class ExampleClass {

}

static void Main()

{

// Déclaration d'une liste de type int.

GenericList<int> list1 = new GenericList<int>();

// Déclaration d'une liste de type string

GenericList<string> list2 = new GenericList<string>();

// Déclaration d'une liste de type ExampleClass.

GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();

}

}

❑ 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 : vous pouvez ajouter et enlever des éléments.

Les collections utilisent la notion de généricité ;

En C#, il existe les collections suivantes:

listes, les dictionnaires, etc.

La Classe Générique List<T> :

située dans l'espace de noms System.Collections.Generic.

une **liste** regroupe un ensemble d'éléments de même nature. Une variable de type "liste" permet de stocker plusieurs éléments partageant le même **type**.

Cependant, une liste offre plus de fonctionnalités qu'un tableau. En particulier, sa taille (nombre d'éléments stockables) n'est pas fixe. Si besoin, une liste se redimensionne automatiquement lors de l'ajout d'un nouvel élément.

Une liste est un **objet** et dispose de plusieurs **propriétés** et **méthodes** pour gérer les éléments qu'elle contient.

Déclaration et utilisation

■ **Déclaration**

List<**type**> valeurs;

Exemple

List<**double**> valeurs;

List<**CompteBancaire**> valeurs;

■ **Instanciation d'une liste**

List<**type**> valeurs = new List<**type**> ();

valeurs = **new** List<**double**>();

List<CompteBancaire> listeComptes = **new List**<CompteBancaire>();

- Ajouter d'Éléments
valeurs.add (Eléments);
- Parcours d'une Liste

foreach (typeElement unElement in listeElements)

{

<instructions utilisant unElement>

}

- La Classe Générique **Dictionary< Tkey,Tvalue >** :située dans l'espace de noms System.Collections.Generic. Un dictionnaire est une collection de paires clef/valeur, chaque clef étant unique. C'est une classe **doublement générique** : il faut préciser le type des clefs et le type des valeurs.
 - Déclaration

Dictionary< Tkey,Tvalue > valeurs;

Exemple

Dictionary<string, string> fichiers;

Dictionary <string, CompteBancaire> valeurs;

- Instanciation d'une liste

valeurs=new Dictionary < Tkey,Tvalue > ();

Exemple

fichiers = new Dictionary <string, string> ();

- **Ajouter d'Eléments**
valeurs.add (valKeys,Eléments);

Exemple

```
fichiers.Add("txt", "notepad.exe");
```

```
fichiers.Add("bmp", "paint.exe");
```

```
fichiers.Add("dib", "paint.exe");
```

Ou **on peut utiliser les indexeurs**

fichiers.["rtf "]="wordpad.exe"; si la clé existe on modifie la valeur associée sinon elle est créée.

//Déclaration puis initialisation

```
valeurs = new Dictionary <string, CompteBancaire>(){  
    {"001",new CompteBancaire() }, {"002",new CompteBancaire() },  
};
```

```
Valeurs. Add({"003", new CompteBancaire() });
```

- **Parcours d'un Dictionnary**

```
foreach (KeyValuePair< Tkey,Tvalue > element in DictionaryElements)  
{  
    <instructions utilisant unElement>  
}
```

```
foreach (var element in DictionaryElements)  
ou {  
    <instructions utilisant unElement>  
}
```

Exemple:

```
foreach(KeyValuePair<string, string> element in fichiers)
{
    Console.WriteLine("L'element dont la clé est {0} est {1}", element.Key, element.Value);
}
```

Ou

```
foreach(var element in fichiers)
{
    Console.WriteLine("L'element dont la clé est {0} est {1}", element.Key, element.Value);
}
```

NB:

Ajout de quelques éléments. Il ne peut pas y avoir deux clefs identiques mais les valeurs peuvent l'être.

Le rangement des éléments dans le Dictionary se fait le calcul d'une clé de hachage et donc peut être différent de l'ordre d'insertion.

TP I Résumé POO

Application

- On souhaite créer une application POO avec C# qui permet de calculer le coûts de transport des marchandises transportées dans des cargaisons.
- Une marchandise est définie par son numéro, son poids et son volume
- Une cargaison peut transporter plusieurs marchandises. Il existe deux types de cargaisons : Routière et Aérienne.
- Chaque cargaison est définie par la distance de du parcours. Les opérations de cette classe sont :
 - Ajouter une marchandise.
 - Afficher toutes les marchandises.
 - Consulter une marchandise sachant son numéro.
 - Consulter le volume total des marchandises
 - Consulter le poids total des marchandises
 - Calculer le cout () de la cargaison qui dépend du type de cargaison.
- Une cargaison aérienne est une cargaison dont le cout est calculé selon la formule suivante :
 - $\text{cout} = 10 \times \text{distance} \times \text{poids total des marchandises}$ si le volume total est inférieur à 80000
 - $\text{cout} = 12 \times \text{distance} \times \text{poids total des marchandises}$ si le volume total est supérieur ou égal à 80000
- Une cargaison routière est une cargaison dont le cout est calculé selon la formule suivante :
 - $\text{cout} = 4 \times \text{distance} \times \text{poids total}$ si le volume total est inférieur à 380000
 - $\text{cout} = 6 \times \text{distance} \times \text{poids total}$ si le volume total est supérieur ou égale à 380000

Application

- Questions :
 - Créer un diagramme de classe simplifié.
 - Ecrire le code C# de la classe Marchandise
 - Ecrire le code C# de la classe Cargaison
 - Ecrire le code C# de la classe CargaisonRoutière
 - Ecrire le code C# de la classe CargaisonAérienne
 - Ecrire le code C# d'une application qui permet de :
 - Créer une cargaison routière
 - Ajouter la cette cargaison 3 marchandises
 - Afficher toutes les marchandises de cette cargaison
 - Afficher le cout de cette cargaison
 - Créer une cargaison aérienne
 - Afficher le cout de cette cargaison
 - Créer une application graphique





-Cargaison

Distance

120000000

Type

Routiere

ADD

Liste Des Categories

Cargaison 12000

Cargaison 120000000

— Merchandise

Poids

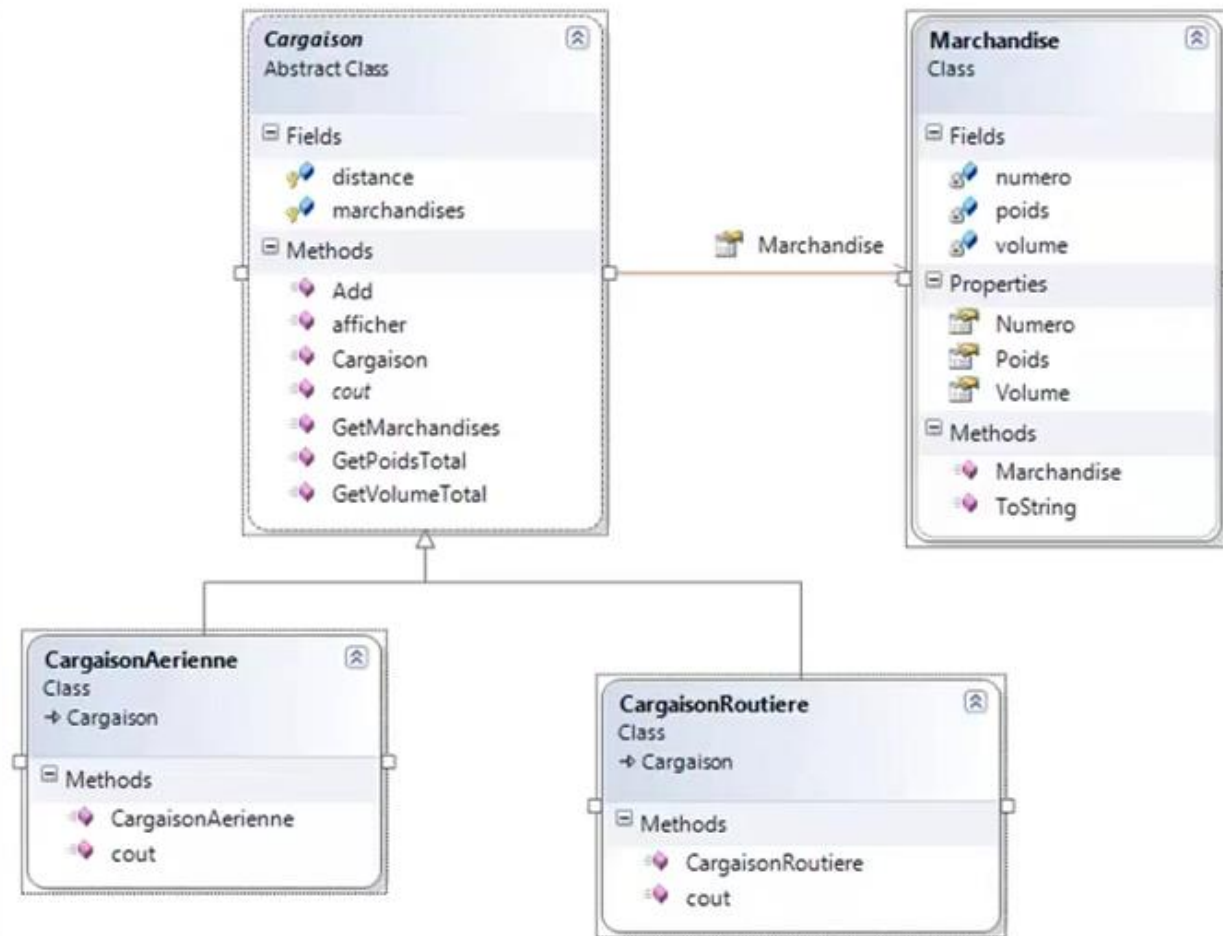
Volume

ADD

Liste Des Marchandises

	Num	Poids	Volume
►	1	1200	10
*			

Diagramme de classes



V) Comprendre les Fonctionnalités du langage C#

V.1) Gérer les exceptions

Les exceptions sont des erreurs de type particulier qui se produisent lors de l'exécution du code et non pas à la compilation. Elle peut se faire à l'aide du bloc

```
try {  
    //Succes  
    }  
    catch (Exception ex)  
    {  
        Console.WriteLine( ex.Message);  
    } finally {  
        //Toujours  
    }
```

Exemple

```
int y = 3;
    int x = 0;
    try
    {
        Console.WriteLine(" Un ");
        int q = y / x;
    }
    catch (Exception ex)
    {
        Console.WriteLine(" Deux");
        Console.WriteLine( ex.Message);
    } finally {
        Console.WriteLine(" Trois");
    }
}
```

NB:

Le bloc try ... catch() permet de capturer une Exception mais pour lever une Exception on peut utiliser

Throw new Exception(" Message");

V.II) Aborder les types nullables

Les types de valeurs ne peuvent pas prendre la valeur **null**, car null est une référence sur aucun objet. Pour corriger ce Problème on utilise la Classe générique **System.Nullable**.

Ainsi

~~int i=null;~~//Erreur Nullable <int >i=null;//Correct ou int ? i=null;

NB: Avec une variable type Nullable on peut utiliser les propriétés suivantes

VarNullable.HasValue: qui retourne true si VarNullable a une valeur sinon false

VarNullable.GetValueOrDefault(): permet de Convertir d'un type nullable en type valeur

Exemple:

```
Int? x=null;
```

```
Int y= x.GetValueOrDefault();//y=0
```

V.III) Utiliser le typage automatique et statique

Le mot clé **var** permet de déclarer une variable sans préciser son type c'est-à-dire le type va s'adapter en fonction de son contenu. Mais ce typage est statique c'est-à-dire une fois qu'une valeur est affectée à la variable donc son type est connu du compilateur et ne peut plus changer.

Exemple:

```
var s="Mon Typage Statique avec var" ;//Le type de s est string
```

```
s=1;//Interdit car s est un string
```

V.IV) Découvrir les variables dynamiques

A partir de C# 4.0 ,il existe des types de variables dynamiques dont le type sera préciser à l' exécution automatiques et il sera dynamiques.

Exemple:

```
dynamic s="Mon Typage Statique avec var" ;//Le type de s est string
```

```
s=1;//Possible
```

V.V)Appliquer les types anonymes

A partir de C# 3.0, on peut créer une variable de type classe sans définir la classe, ce type de variable est appelé **type anonyme ou non défini**.

Exemple

```
var maPersonne = new { nom= "Diop ", prenom= " Amadou" , age= 12};
```

```
Console.WriteLine(" Le Nom est " + maPersonne.nom);
```

NB: Les propriétés d'un **type anonyme** sont **accessibles seulement en lecture seule** c'est-à-dire on ne peut pas les modifier. Ces types de classe ne peuvent pas contenir des méthodes.

V.VI)Avoir une approche des délégués

Un délégué est un type en C# 2 qui représente une référence à une méthode avec une liste de paramètres.

Exemple Utilisons le **délégués Predicate**

Fonction de Recherche

```
public static bool recherche(string critere)
{
    if (critere.Contains("jour")) return true;
    return false;
}
```

Définition du Delege

```
List<String>valeurs=new List<String>(){ "Bonjour", "jour",
"Au revoir" };
var p = new Predicate<string>(recherche);
```

Execution du Delegué

```
var s=valeurs.Find(p); //Retourne Bonjour
Console.WriteLine("J'ai Trouvé "+s);

var s=valeurs.FindAll(p); //Retourne Bonjour et Jour
Console.WriteLine("Nombre d'elements est "+s.count);
```

V.VI) Utiliser des fonctions anonymes

Les fonctions anonymes sont des fonctions qui n'ont pas de nom.
On utilise le mot clé delegate.

```
delegate(type parametre1,,,, typen parametren)  
{  
  
//Instructions  
  
}
```

Fonctions anonymes

```
List<String>valeurs=new List<String>(){ "Bonjour", "jour", "Au revoir" };  
var p = new Predicate<string>(delegate(string critere)  
{  
    if (critere.Contains("jour")) return true;  
    return false;  
});  
var s = valeurs.FindAll(p);  
Console.WriteLine("J'ai Trouvé " + s.Count);
```

V.VII)Se servir d'expressions lambda

Une expression lambda une fonction anonyme qui peut créer des délégués.

(param1,,,, paramn)=>{ instructions};

Exemple

```
List<String>valeurs=new List<String>(){ "Bonjour", "jour", "Au revoir" };  
var s = valeurs.FindAll((critere)=> critere.Contains("jour"));  
Console.WriteLine("J'ai Trouvé " + s.Count);
```

Découvrir LINQ (Language Integrated Query)

❑ Travailler sur une première requête LINQ

- Une *requête* est une expression qui extrait des données d'une source de données.
- Les requêtes sont généralement exprimées dans un **langage de requête spécialisé**.
- Par exemple
 - **SQL pour les bases de données relationnelles et**
 - **XQuery pour XML.**
- **Inconvénients** : Les développeurs ont dû apprendre un nouveau langage de requête pour chaque type de source de données ou de format de données qu'ils doivent prendre en charge.
- **Avantages**: LINQ simplifie cette situation en offrant un modèle cohérent pour travailler avec des données sur différents types de sources de données et de formats.
- Dans une requête LINQ, on travaille toujours avec des objets.
- Mêmes modèles de codage de base pour interroger et transformer des données :
 - **dans des documents XML,**
 - **des bases de données SQL,**
 - **des ensembles de données ADO.NET (Modèle d'accès aux données sur .Net),**
 - **des collections .NET**
 - **et tout autre format pour lequel un fournisseur LINQ est disponible.**

- Toutes les opérations de requête LINQ comportent trois actions distinctes:
 - **Obtenez la source de données.**
 - **Créer la requête**
 - **Exécutez la requête.**
- LINQ a deux type de Syntaxes:
 - **Syntaxe Déclarative comme SQL**
 - **Syntaxe utilisant les fonctions se basant sur l'appel de Méthodes d'extensions**

A) Syntaxe de Méthode Déclarative

Exemple d'une Requête SQL

Select *
From Table
Where condition

Execution de la Requete



From Table
Where Condition
Select *

Exemple

List<String>valeurs=new List<String>(){**"Bonjour"**, **"jour"**, **"Au revoir"** };

Exeprression Lamda

```
var s = valeurs.FindAll((
critere)=> critere.Contains("jour"));
```

Exeprression LINQ

```
var re= From s in valeurs
        where s.Contains("jour")
        select s;
```

Remplacer

IEnumerable<string> re ou var re

Group BY

SQL

```
Select
critere,min(valeur)
from Table
Where Condition
Group by Critere
Having Condition
```

LINQ

```
var result=From s in valeurs
            where s.contains("jour")
            group s by s into s2
            select s2.Min()
```

- **Order By**

```
List<String>valeurs=new List<String>(){ "Bonjour", "jour", "Au revoir" };
```

```
var result= from s in valeurs
```

```
Order by descending
```

```
select s;
```

```
//Après on fait un Parcours avec foreach
```

- **Utiliser les fonctions dans LINQ**

Soit la Requête LINQ suivante

```
var result= from s in valeurs
```

```
Order by descending
```

```
Where s.contains("jour")
```

```
select s;
```

Ou on peut définir une fonction

```
bool recherche(string critere)
```

```
{
```

```
    return (critere.Contains("jour"));
```

```
}
```

Appeler cette fonction dans une Requête LINQ

```
var result= from s in valeurs
```

```
Order by descending
```

```
Where recherche(s )
```

```
select s;
```

- **Appliquer l'équivalent de IN en LINQ**

```
List<String>valeurs=new List<String>(){"Bonjour", "jour", "Au revoir" };
```

```
String [] tab =new String[]={"Bonjour", "Au revoir" };
```

```
var result= from s in valeurs
```

```
Order by descending
```

```
Where tab.Contains(s)
```

```
select s;
```

//Définir le Tableau dans la Requête

```
var result= from s in valeurs
```

```
Order by descending
```

```
Where new [] {"Bonjour", "Au revoir" }. tab.Contains(s))
```

```
select s;
```

B) Syntaxe de Méthode sous forme de méthode

On peut utiliser LINQ dans une syntaxe sous forme de fonctions avec l'utilisation des délégués (prédicat ou Expression Lambda).

Exemple

```
var result= valeurs. Where(s=>new [] {"Bonjour", "Au revoir" }.  
tab.Contains(s)).OrderByDescending(si=>si);
```