

# Dot Net Programmation Orientée Objet C#



Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : [med@youssfi.net](mailto:med@youssfi.net)

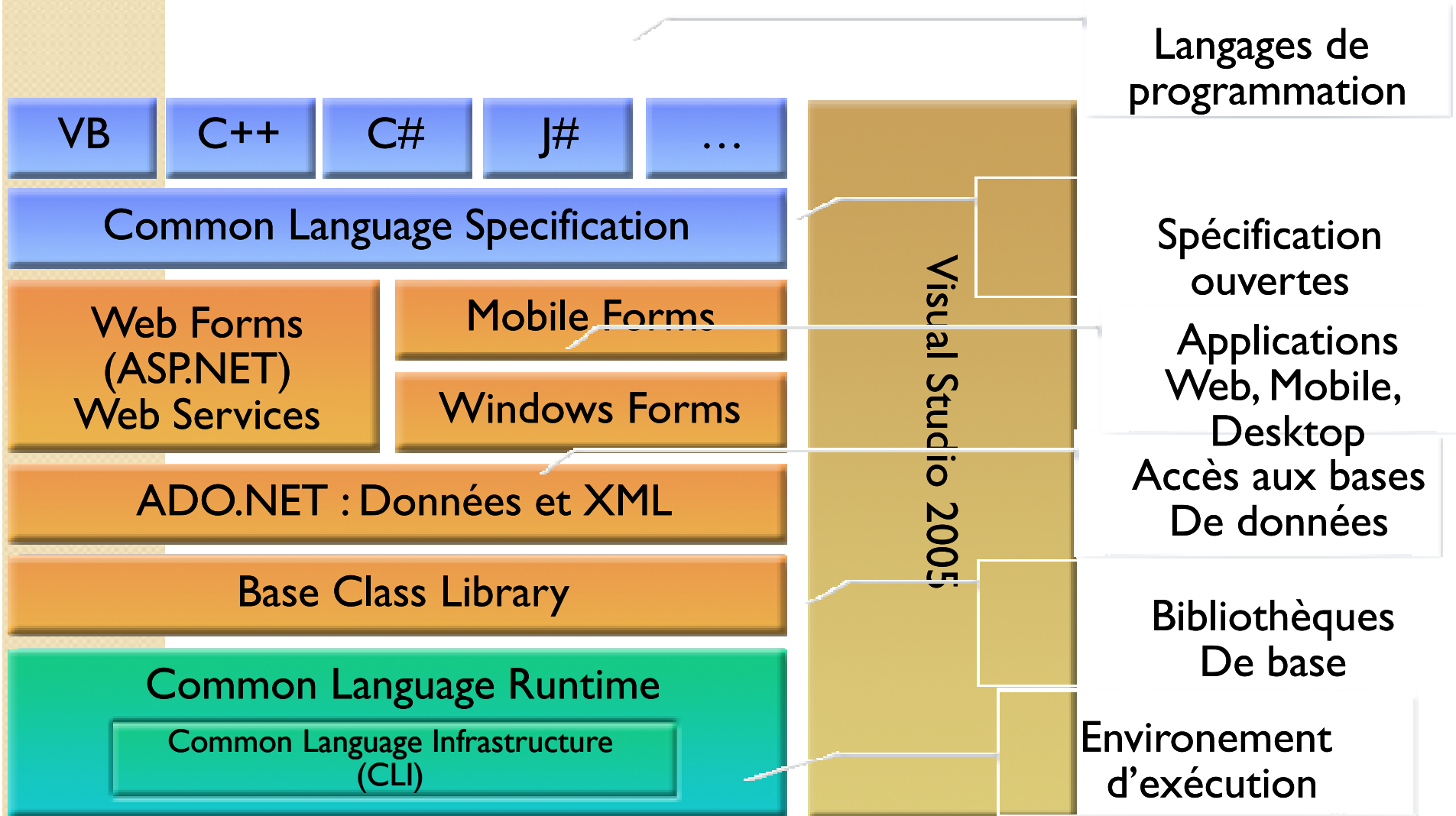
Supports de cours : <http://fr.slideshare.net/mohamedyoussfi9>

Chaîne vidéo : <http://youtube.com/mohamedYoussfi>

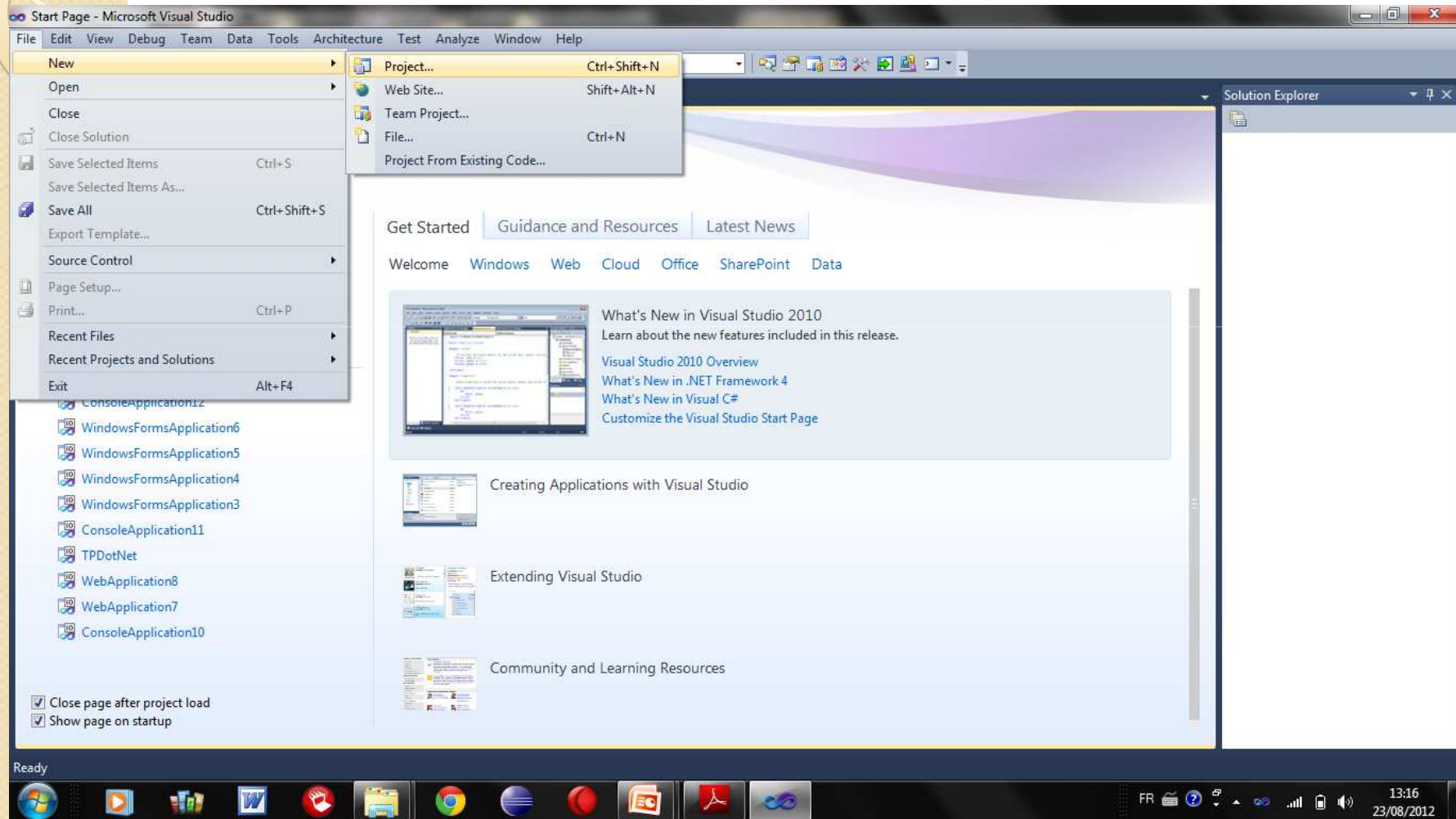
Recherche : [http://www.researchgate.net/profile/Youssfi\\_Mohamed/publications](http://www.researchgate.net/profile/Youssfi_Mohamed/publications)

# .NET Framework

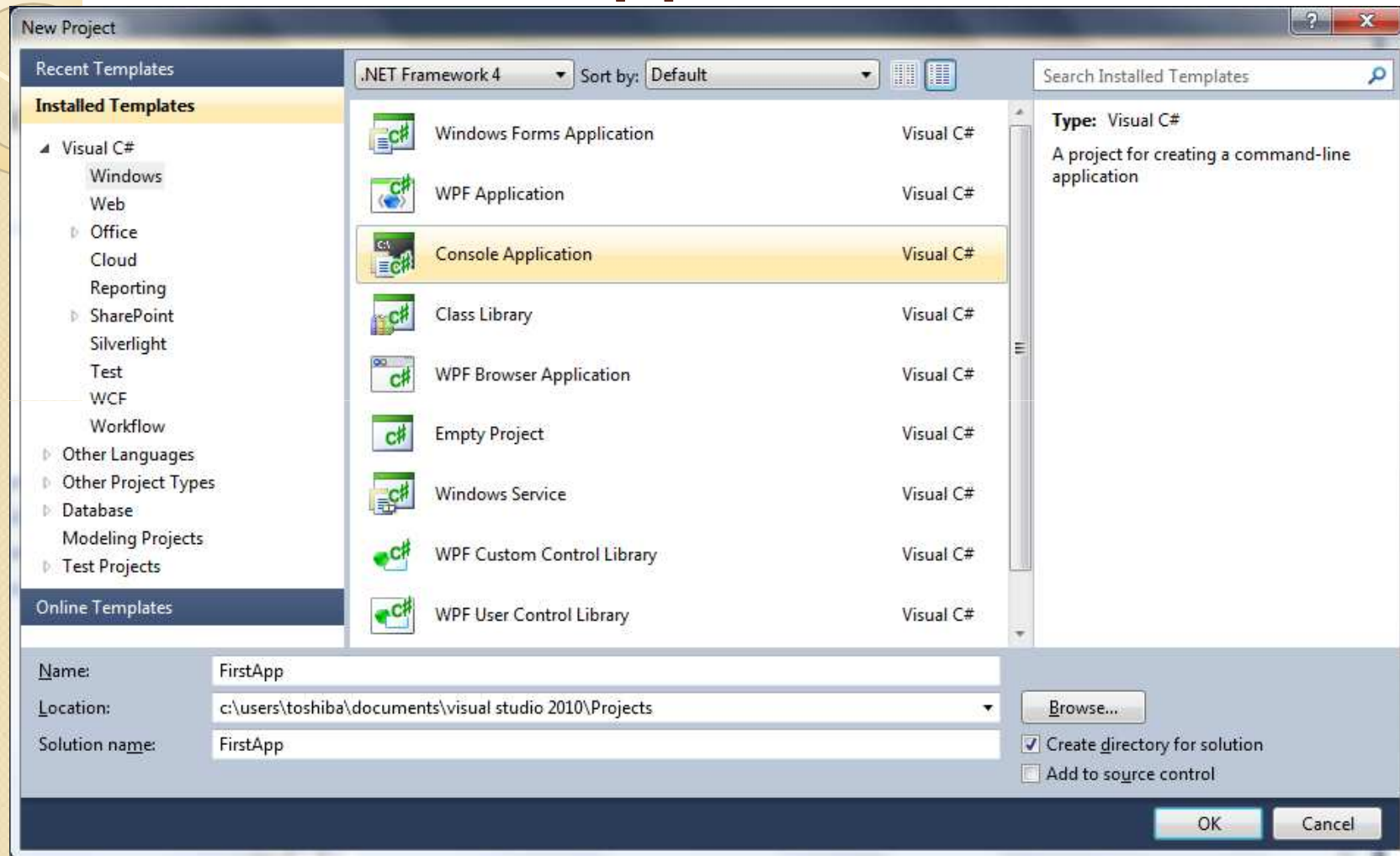
Microsoft  
.NET Framework



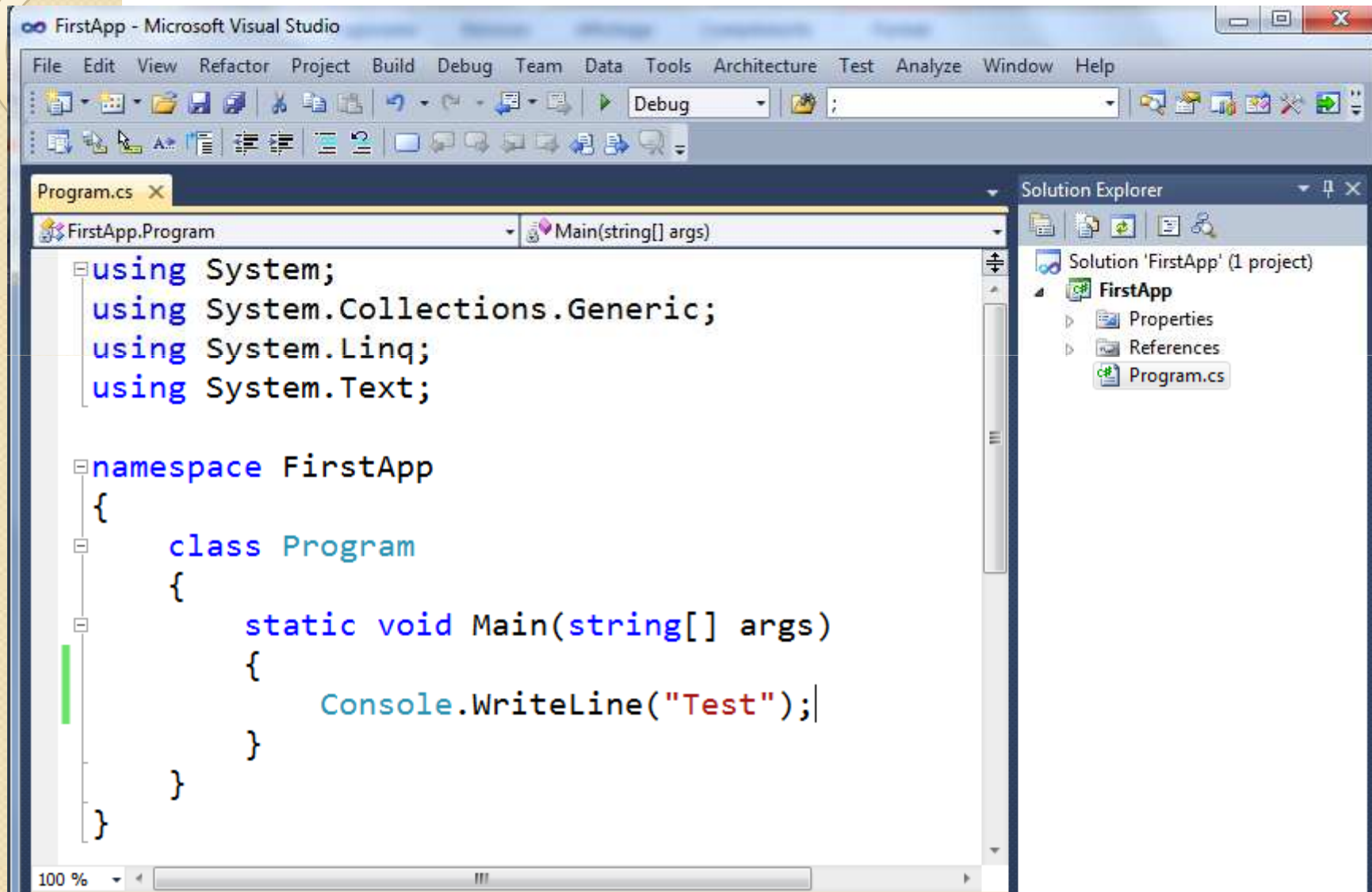
# Première Application



# Première Application

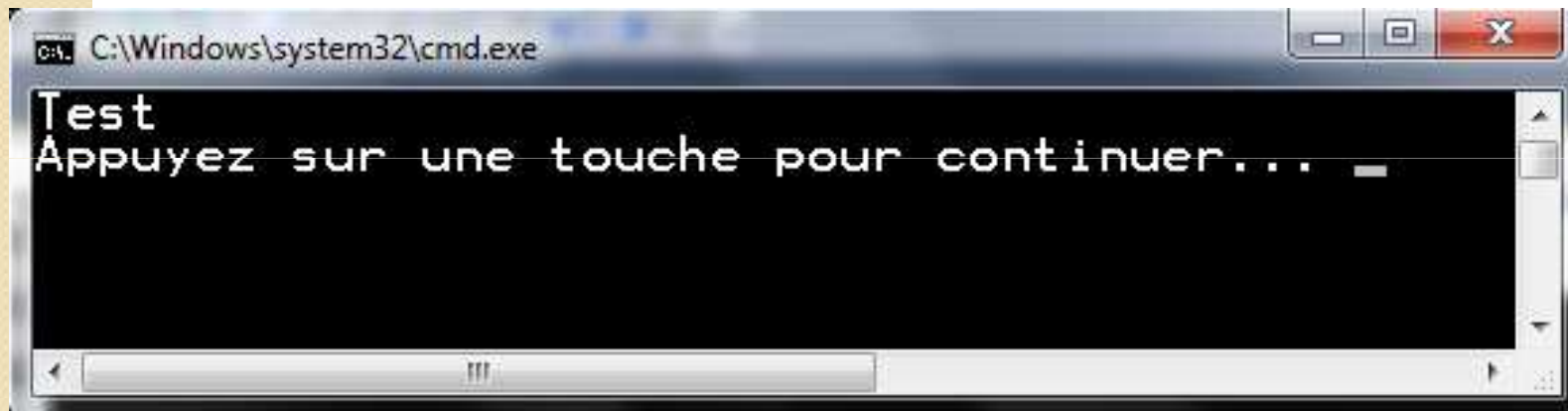


# Première Application

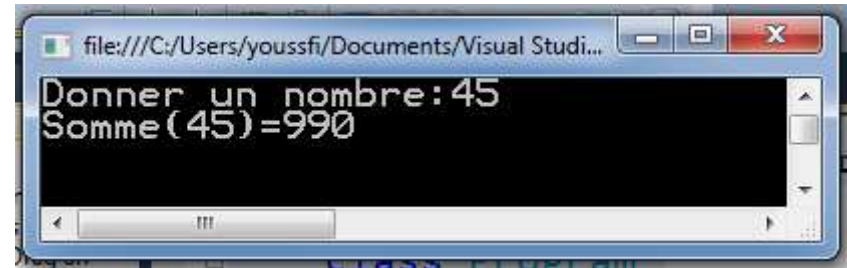




# Exécution

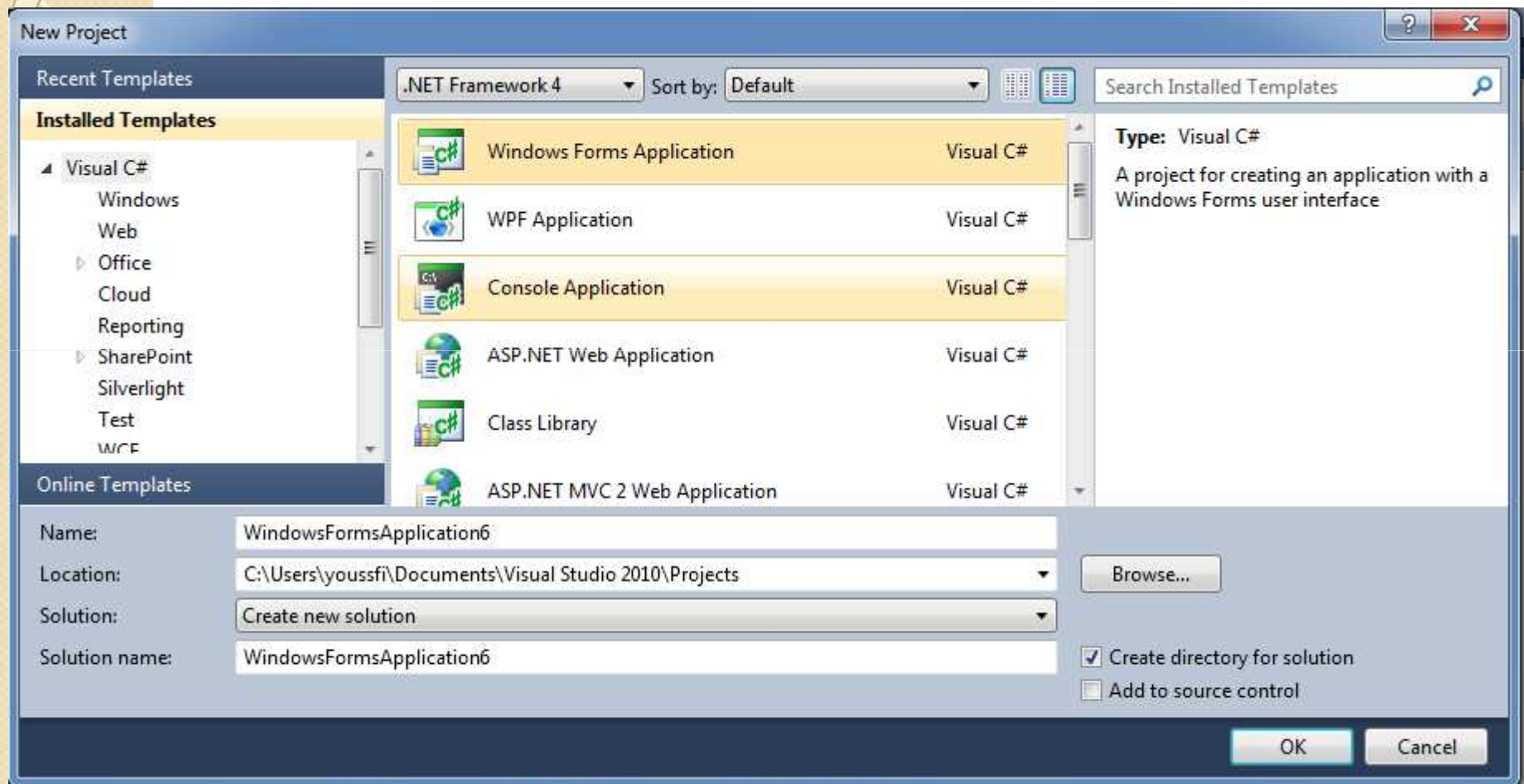


# Exemple 2



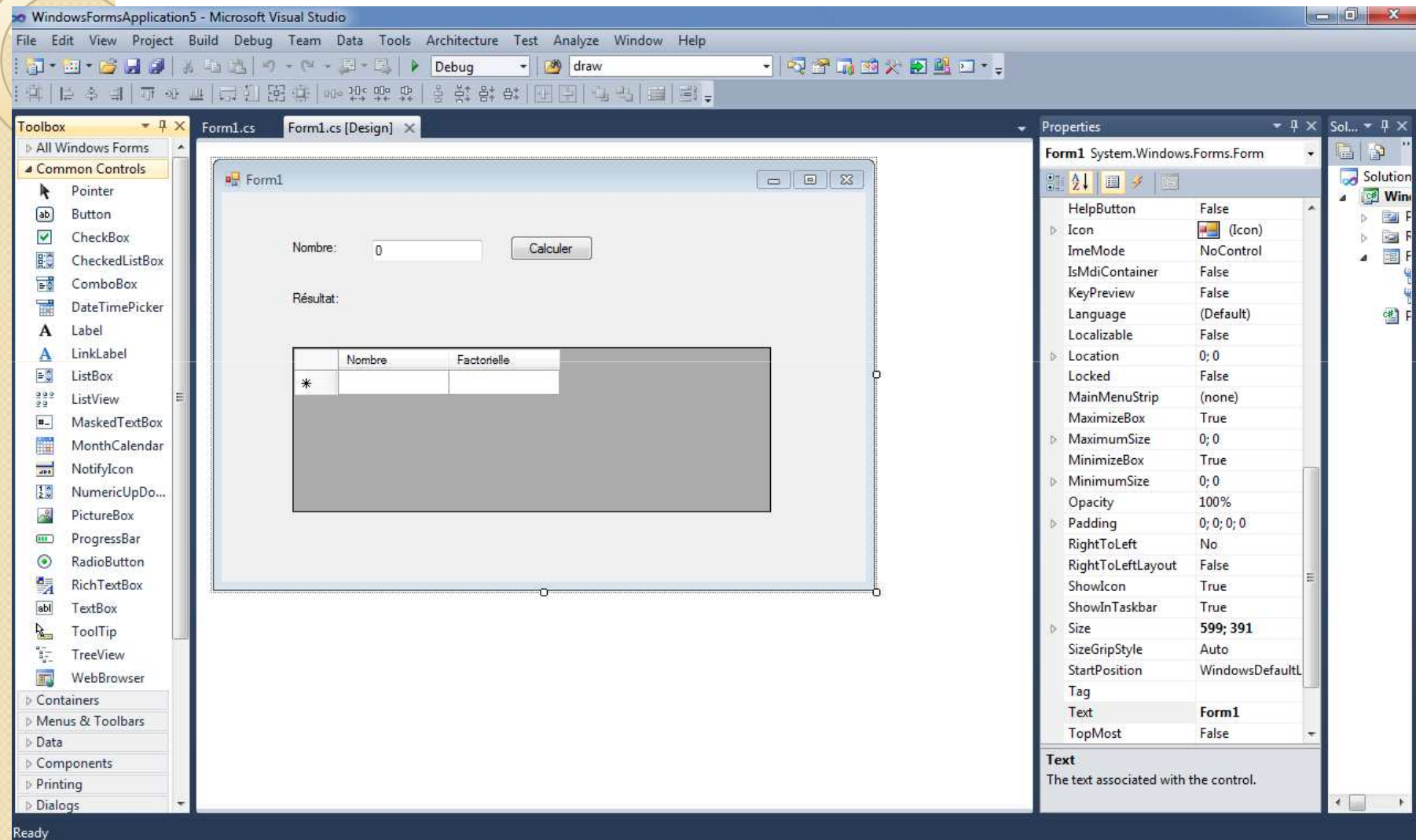
```
using System; using System.Collections.Generic;
using System.Linq; using System.Text;
namespace ConsoleApplication7 {
    class Program {
        public static double somme(int nb) {
            double somme = 0;
            for (int i = 0; i < nb; i++) {
                somme += i;
            }
            return somme;
        }
        static void Main(string[] args) {
            Console.Write("Donner un nombre:");
            int nb = Int32.Parse(Console.ReadLine());
            Console.WriteLine("Somme({0})={1}", nb, somme(nb));
            Console.ReadLine();
        }
    }
}
```

## Exemple 3 :Windows Form Application





# Exemple 3 : Windows From Application



## Exemple 3 : Windows Form Application

```
using System; using System.Collections.Generic;using System.ComponentModel;
using System.Data; using System.Drawing; using System.Linq;
using System.Text; using System.Windows.Forms;
namespace WindowsFormsApplication5 {
    public partial class Form1 : Form {
        public Form1() {
            InitializeComponent();
        }
        private void buttonCalculer_Click(object sender, EventArgs e) {
            int nb = Int32.Parse(textBoxNombre.Text);
            dataGridViewRes.Rows.Add(nb, factorielle(nb));
        }
        public double factorielle(int nb) {
            double f = 1;
            for (int i = 1; i < nb; i++)
                f = f * i;
            return f;
        }
    }
}
```

## Exemple 3 : Windows Form Application

Nombre: 12

Résultat:

	Nombre	Factorielle
▶	3	6
	9	362880
	12	479001600
*		



# Structures fondamentales du langage C#

# Les primitives

Type C#	Type .NET	Donnée représentée	Suffixe des valeurs littérales	Codage
char	Char (S)	caractère		2 octets
string	String (C)	chaîne de caractères		
int	Int32 (S)	nombre entier		4 octets
uint	UInt32 (S)	..	U	4 octets
long	Int64 (S)	..	L	8 octets
ulong	UInt64 (S)	..	UL	8 octets
sbyte		..		1 octet
byte	Byte (S)	..		1 octet
short	Int16 (S)	..		2 octets
ushort	UInt16 (S)	..		2 octets
float	Single (S)	nombre réel	F	4 octets
double	Double (S)	..	D	8 octets
decimal	Decimal (S)	nombre décimal	M	16 octets
bool	Boolean (S)	..		1 octet
object	Object (C)	référence d'objet		

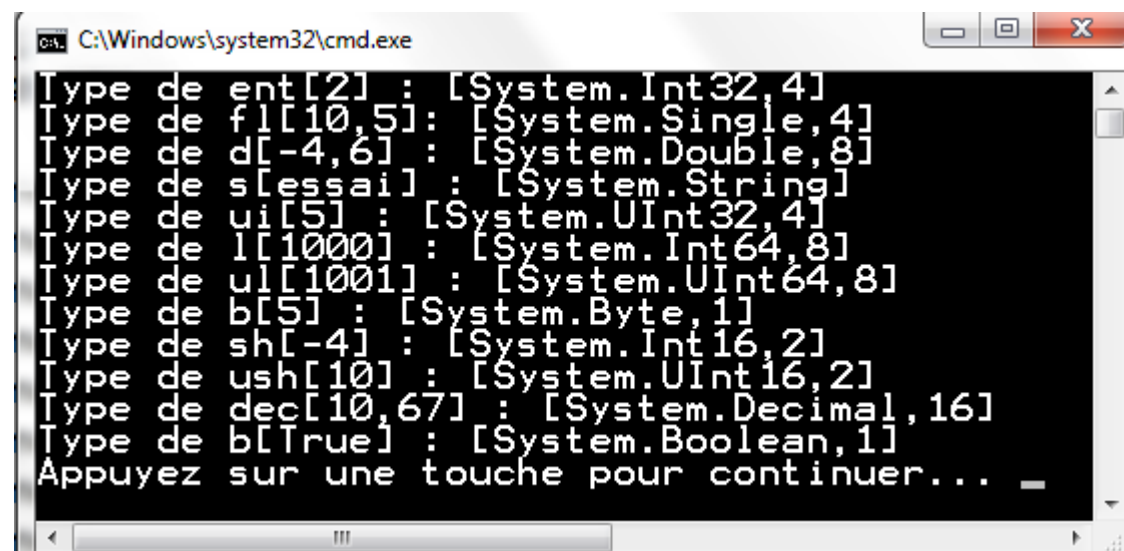


# Exemple de déclaration

- `int ent = 2;`
- `float fl = 10.5F;`
- `double d = -4.6;`
- `string s = "essai";`
- `uint ui = 5;`
- `long l = 1000;`
- `ulong ul = 1001;`
- `byte octet = 5;`
- `short sh = -4;`
- `ushort ush = 10;`
- `decimal dec = 10.67M;`
- `bool b = true;`

# Affichage des types et des valeurs

```
Console.WriteLine("Type de ent[{1}] : [{0},{2}]", ent.GetType().FullName,  
    ent, sizeof(int));  
Console.WriteLine("Type de fl[{1}]: [{0},{2}]", fl.GetType().FullName, fl,  
    sizeof(float));  
Console.WriteLine("Type de d[{1}] : [{0},{2}]", d.GetType().FullName, d, sizeof(double));  
Console.WriteLine("Type de s[{1}] : [{0}]", s.GetType().FullName, s);  
Console.WriteLine("Type de ui[{1}] : [{0},{2}]", ui.GetType().FullName, ui,  
    sizeof(uint));  
Console.WriteLine("Type de l[{1}] : [{0},{2}]", l.GetType().FullName, l, sizeof(long));  
Console.WriteLine("Type de ul[{1}] : [{0},{2}]", ul.GetType().FullName, ul,  
    sizeof(ulong));
```



```
C:\Windows\system32\cmd.exe  
Type de ent[2] : [System.Int32,4]  
Type de fl[10,5]: [System.Single,4]  
Type de d[-4,6] : [System.Double,8]  
Type de s[essai] : [System.String]  
Type de ui[5] : [System.UInt32,4]  
Type de l[1000] : [System.Int64,8]  
Type de ul[1001] : [System.UInt64,8]  
Type de b[5] : [System.Byte,1]  
Type de sh[-4] : [System.Int16,2]  
Type de ush[10] : [System.UInt16,2]  
Type de dec[10,67] : [System.Decimal,16]  
Type de b[True] : [System.Boolean,1]  
Appuyez sur une touche pour continuer...
```

# Déclaration des constantes

- Une constante se déclare en utilisant le mot réservé `const`
- Exemple de déclaration :
  - `const float myPI = 3.141592F;`

# Tableaux de données

- Un tableau C# est un objet permettant de rassembler sous un même identificateur des données de même type. Sa déclaration est la suivante :
  - `Type[] tableau[] = new Type[n]`
  - **n est le nombre de données que peut contenir le tableau. La syntaxe `Tableau[i]` désigne la donnée n° i où i appartient à l'intervalle**
- Un tableau peut être initialisé en même temps que déclaré :
  - `int[] entiers = new int[] { 0, 10, 20, 30 };`
- ou plus simplement :
  - `int[] entiers = { 0, 10, 20, 30 };`
- Les tableaux ont une propriété **Length** qui est le nombre d'éléments du tableau.
- La boucle suivante permet d'afficher tous les éléments du tableau:

```
for (int i = 0; i < entiers.Length; i++)  
{  
    Console.WriteLine("I=" + i + " tab(" + i + ")=" + entiers[i]);  
}
```

# Tableaux à deux dimensions

- Un tableau à deux dimensions peut être déclaré et initialisé de la manière suivante:

```
double[,] réels = new double[,] { { 0.5, 1.7 }, { 8.4, -6 } };
```

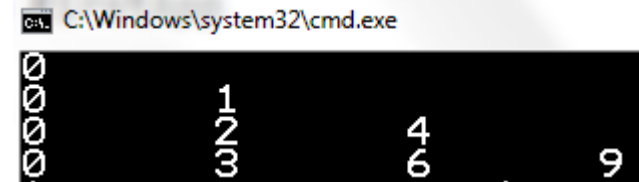
```
for (int i = 0; i < 2; i++)  
{  
    for (int j = 0; j < réels.GetLength(i); j++)  
        Console.Write(réels[i,j]+"\\t");  
    Console.WriteLine();  
}
```



# Tableaux de tableau

- Un tableau de tableaux peut être déclaré de la manière suivante :

```
double[][] tab = new double[4][] ;
for (int i = 0; i < tab.Length; i++)
{
    tab[i] = new double[i+1];
    for (int j = 0; j < tab[i].Length; j++)
        tab[i][j] = i*j;
}
for (int i = 0; i < tab.Length; i++)
{
    for (int j = 0; j < tab[i].Length; j++)
        Console.Write(tab[i][j] + "\t");
    Console.WriteLine();
}
```



0	1	2	3
0	2	4	6
0	3	6	9
0	4	8	12

# Opérateurs

- Opérateur d'affectation:
  - **x=3;** // x reçoit 3
  - **x=y=z=w+5;** // z reçoit w+5, y reçoit z et x reçoit y
- Les opérateurs arithmétiques à deux opérandes:
  - **+** : addition
  - **-** : soustraction
  - **\*** : multiplication
  - **/** : division
  - **%** : modulo (reste de la division euclidienne)

# Opérateurs

- Les opérateurs arithmétiques à deux opérandes (Les raccourcis)

`x = x + 4;` ou `x+=4;`

`z = z * y;` ou `z*=y;`

`v = v % w;` ou `v%=w;`

- Les opérateurs relationnels:

- `==` : équivalent
- `<` : plus petit que
- `>` : plus grand que
- `<=` : plus petit ou égal
- `>=` : plus grand ou égal
- `!=` : non équivalent

- Les opérateurs d'incrémentations et de décrémentation:

- `++` : Pour incrémenter (`i++` ou `++i`)
- `--` : Pour décrémentation (`i--` ou `--i`)

# Opérateurs

- Les opérateurs logiques

- && Et (deux opérandes)
- || Ou (deux opérandes )
- ! Non (un seul opérande)

- L'opérateur à trois opérandes ?:

- **condition ? expression\_si\_vrai : expression\_si\_faux**
- exemple : `x = (y < 5) ? 4 * y : 2 * y;`

**Equivalent à :**

if (y < 5)

    x = 4 \* y;

else

    x = 2 \* y;

# Structures de contrôle

- **L'instruction conditionnelle if**

La syntaxe de l'instruction **if** peut être décrite de la façon suivante:

```
if (expression) instruction;
```

ou :

```
if (expression) {  
    instruction1;  
    instruction2;  
}
```

- **L'instruction conditionnelle else**

```
if (expression) {  
    instruction1;  
}  
else {  
    instruction2;  
}
```



# Structures de contrôle

- **Les instructions conditionnelles imbriquées**

Java permet d'écrire ce type de structure sous la forme :

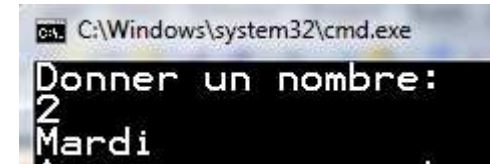
```
if (expression1) {  
    bloc1;  
}  
else if (expression2) {  
    bloc2;  
}  
else if (expression3) {  
    bloc3;  
}  
else {  
    bloc5;  
}
```

# Structures de contrôle:

- **L'instruction switch**

Syntaxe :

```
switch( variable) {  
    case valeur1: instr1;break;  
    case valeur2: instr2;break;  
    case valeurN: instrN;break;  
    default: instr;break;
```



## ■ Exemple:

```
Console.WriteLine("Donner un nombre:");  
int nb = Int32.Parse(Console.ReadLine());  
switch (nb)  
{  
    case 1: Console.WriteLine("Lundi"); break;  
    case 2: Console.WriteLine("Mardi"); break;  
    case 3: Console.WriteLine("Mercredi"); break;  
    default: Console.WriteLine("Autrement"); break;  
}
```

# Structures de contrôle

- **La boucle for**

```
for (initialisation;test;incrémentation) {  
    instructions;  
}
```

Exemple :

```
for (int i = 2; i < 10;i++) {  
    Console.WriteLine ("I="+i);  
}
```

- **foreach**

```
foreach (Type variable in collection)  
    instructions;  
}
```

Exemple :

```
string[] amis = { "A", "B", "C", "D" };  
foreach (string nom in amis) {  
    Console.WriteLine(nom);  
}
```

# Structures de contrôle

- **Sortie d'une boucle par `return`**

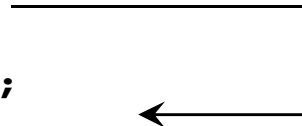
```
int[] tab=new int[]{4,6,5,8};  
for (int i = 0; i < tab.Length; i++) {  
    if (tab[i] == 5) {  
        return i;  
    }  
}
```

- **Branchement au moyen des instructions `break` et `continue`**

- **break:**

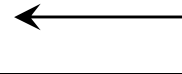
```
int x = 10;  
for (int i = 0; i < 10; i++) {  
    x--;  
    if (x == 5) break;  
}
```

System.out.println(x);



- **continue:**

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) continue;  
    Console.WriteLine(i);  
}
```



# Structures de contrôle

- **L 'instruction While**

```
while (condition){  
    BlocInstructions;  
}
```

- **L 'instruction do .. while**

```
do{  
    BlocInstructions;  
}  
while (condition);
```

## Exemple :

```
int s=0;int i=0;  
while (i<10){  
    s+=i;  
    i++;  
}  
Console.WriteLine("Somme="+s);
```

## Exemple :

```
int s=0;int i=0;  
do{  
    s+=i;  
    i++;  
}while (i<10);  
Console.WriteLine("Somme="+s);
```



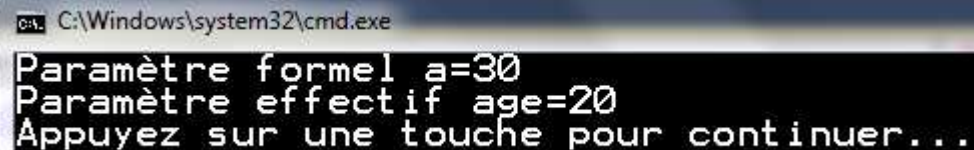
# Passage de paramètres à une fonction

- Nous nous intéressons ici au mode de passage des paramètres d'une fonction. Considérons la fonction statique suivante :

```
private static void ChangeInt(int a) {  
    a = 30;  
    Console.WriteLine("Paramètre formel a=" + a);  
}
```

- Considérons une utilisation de cette fonction:

```
static void Main(string[] args)  
{  
    int age = 20;  
    ChangeInt(age);  
    Console.WriteLine("Paramètre effectif age=" + age);  
}
```



C:\Windows\system32\cmd.exe  
Paramètre formel a=30  
Paramètre effectif age=20  
Appuyez sur une touche pour continuer...

# Passage de paramètres à une fonction

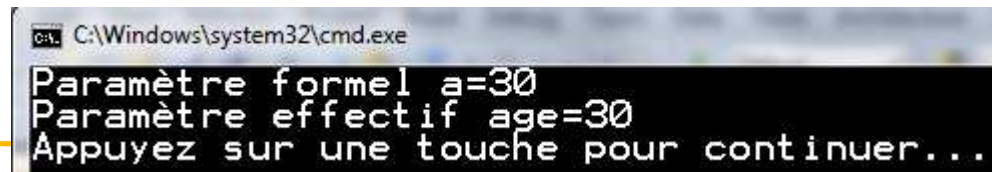
- Passage des paramètres par valeur :
  - L'exemple précédent nous montre que les paramètres d'une fonction sont par défaut passés par valeur, c'est à dire que la valeur du paramètre effectif est recopiée dans le paramètre formel correspondant. On a deux entités distinctes. Si la fonction modifie le paramètre formel, le paramètre effectif n'est lui en rien modifié.

# Passage de paramètres à une fonction

- Passage des paramètres par référence :
  - Dans un passage par référence, le paramètre effectif et le paramètre formel sont une seule et même entité. Si la fonction modifie le paramètre formel, le paramètre effectif est lui aussi modifié. En C#, ils doivent être tous deux précédés du mot clé **ref** :
  - Voici un exemple :

```
using System;
namespace FirstApp {
    class Program    {
        static void Main(string[] args)    {
            int age = 20;
            ChangeInt(ref age);
            Console.WriteLine("Paramètre effectif age=" + age);
        }

        private static void ChangeInt(ref int a) {
            a = 30;
            Console.WriteLine("Paramètre formel a=" + a);
        }
    }
}
```



```
C:\Windows\system32\cmd.exe
Paramètre formel a=30
Paramètre effectif age=30
Appuyez sur une touche pour continuer...
```



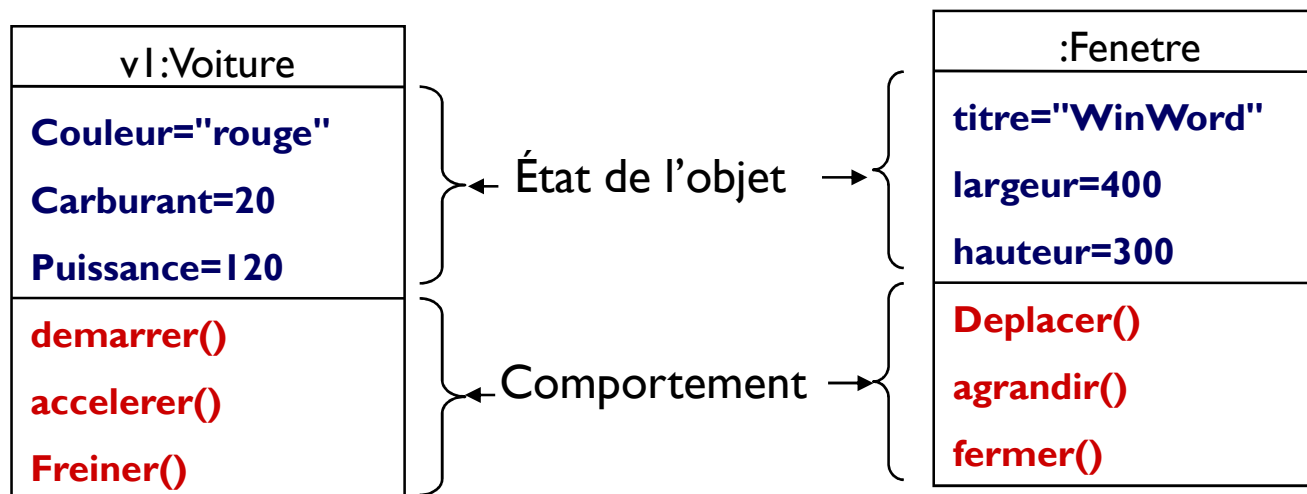
# Programmation orientée objet avec C#

# Méthode orientée 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:
  - **Objet et classe**
  - **Héritage**
  - **Encapsulation (Accessibilité)**
  - **Polymorphisme**

# Objet

- Un objet est une structure informatique définie par un état et un comportement
- Objet=état + comportement
  - L'état regroupe les valeurs instantanées de tous les attributs 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:





# 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.





# Classes

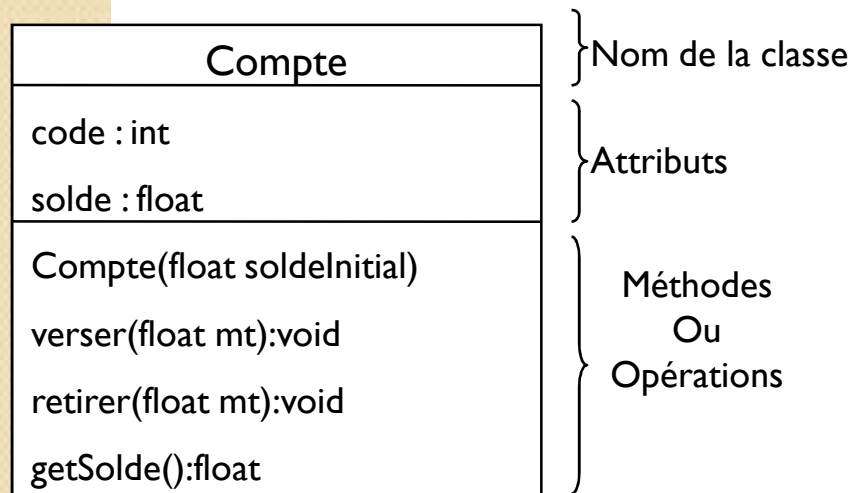
- Les objets qui ont des caractéristiques communes sont regroupés dans une entité appelé classe.
- La classe décrit le domaine de définition d'un ensemble d'objets.
- Chaque objet appartient à une classe
- Les généralités sont contenues dans les classe et les particularités dans les objets.
- Les objets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

# Caractéristique d'une classe

- Une classe est défini par:
  - Les attributs
  - 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 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 méthode 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**
  - Une méthode qui est appelée au moment de la destruction d'un objet. Cette méthode s'appelle le **DESTRUCTEUR**

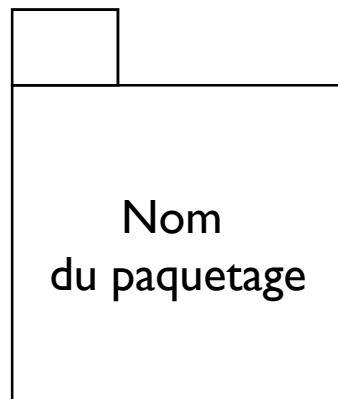
# Représentation UML d'une classe

- Une classe est représenté par un rectangle à 3 compartiments:
  - Un compartiment qui contient le nom de la classe
  - Un compartiment qui contient la déclaration des attributs
  - Un compartiment qui contient les méthodes
- Exemples:



## Les classes sont regroupées dans des namespaces (packages)

- Les packages offrent un mécanisme général pour la partition des modèles et le regroupement des éléments de la modélisation
- Chaque package est représenté graphiquement par un dossier
- Les packages divisent et organisent les modèles de la même manière que les dossier organisent le système de fichier



# Accessibilité au membres d'une classe

## ■ **private:**

- ❑ L'autorisation **private** est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes et classes internes).
- ❑ Les éléments déclarés **private** ne sont accessibles que depuis la classe qui les contient.
- ❑ Ce type d'autorisation est souvent employé pour les variables qui ne doivent être modifiées ou lues qu'à l'aide d'un *getter* ou d'un *setter*.

## ■ **public:**

- ❑ Un membre public d'une classe peut être utilisé par n'importe quelle autre classe.
- ❑ En UML les membres public sont indiqués par le signe +

## ■ **protected:**

- ❑ Les membres d'une classe peuvent être déclarés **protected**.
- ❑ Dans ce cas, l'accès en est réservé aux classes dérivées

## ■ **Internal** : L'accès est restreint à l'assembly en cours. (dll contenant la classe)

## ■ **protected internal** : L'accès est restreint à l'assembly en cours ou aux types dérivés de la classe conteneur.

## ■ Autorisation par défaut : en l'absence de l'un des modificateurs précédents, l'autorisation d'accès par défaut est **private**

# Exemple d'implémentation d'une classe en C#

metier

## Compte

- code : int

# solde : float

+ Compte(int code, float solde)

+ verser(float mt):void

+ retirer(float mt):void

+ toString():String

```
namespace metier {  
    public class Compte {  
        // Attributs  
        private int code; protected float solde;  
        // Constructeur  
        public Compte(int c, float s) {  
            this.code = c; this.solde = s;  
        }  
        // Méthode pour verser un montant  
        public void Verser(float mt) {  
            solde = solde + mt;  
        }  
        // Méthode pour retirer un montant  
        public void Retirer(float mt) {  
            solde = solde - mt;  
        }  
        // Une méthode qui retourne l'état du  
        compte  
        public string ToString() {  
            return (" Code=" + code + " Solde=" +  
solde);  
        }  
    }  
}
```

# Création des objets dans C#

- pour créer un objet d'une classe , On utilise la commande new suivie du constructeur de la classe.
- La commande new Crée un objet dans l'espace mémoire et retourne l'adresse mémoire de celui-ci.
- Cette adresse mémoire devrait être affectée à une variable qui représente la référence de l'objet. Cette référence est appelée aussi handle.

```
using System;
using metier;
namespace test
{
    class Application
    {
        static void Main(string[] args)
        {
            Compte c1 = new Compte(1, 5000);
            Compte c2 = new Compte(2, 6000);
            c1.Verser(3000);
            c1.Retirer(2000);
            Console.WriteLine(c1.ToString());
        }
    }
}
```

**Code=1 Solde= 6000**

c1:Compte
<b>code=1</b>
<b>solde=6000</b>
<b>verser(float mt)</b>
<b>retirer(float mt)</b>
<b>toString()</b>

c2:Compte
<b>code=2</b>
<b>solde=6000</b>
<b>verser(float mt)</b>
<b>retirer(float mt)</b>
<b>toString()</b>



# Constructeur par défaut

- Quand on ne définit aucun constructeur pour une classe, le compilateur crée le constructeur par défaut.
- Le constructeur par défaut n'a aucun paramètre et ne fait aucune initialisation

Exemple de classe :

```
public class Personne {  
    // Les Attributs  
    private int code;  
    private String nom;  
    // Les Méthodes  
    public void SetNom(String n){  
        this.nom=n;  
    }  
    public String GetNom(){  
        return nom;  
    }  
}
```

Instanciation en utilisant le constructeur par défaut :

```
Personne p=new Personne();  
p.SetNom("AZER");  
Console.WriteLine(p.GetNom());
```

# Getters et Setters

- Les attributs privés d'une classe ne sont accessibles qu'à l'intérieur de la classe.
- Pour donner la possibilité à d'autres classes d'accéder aux membres privés, il faut définir dans la classes des méthodes publiques qui permettent de :
  - lire la variables privés. Ce genre de méthodes s'appellent les **accesseurs** ou **Getters**
  - modifier les variables privés. Ce genre de méthodes s'appellent les **mutateurs** ou **Setters**
- Les getters sont des méthodes qui commencent toujours par le mot **Get** et finissent par le nom de l'attribut en écrivant en majuscule la lettre qui vient juste après le get. Les getters retourne toujours le même type que l'attribut correspondant.
  - Par exemple, dans la classe CompteSimple, nous avons défini un attribut privé : **private string nom;**
  - Le getter de cette variable est :

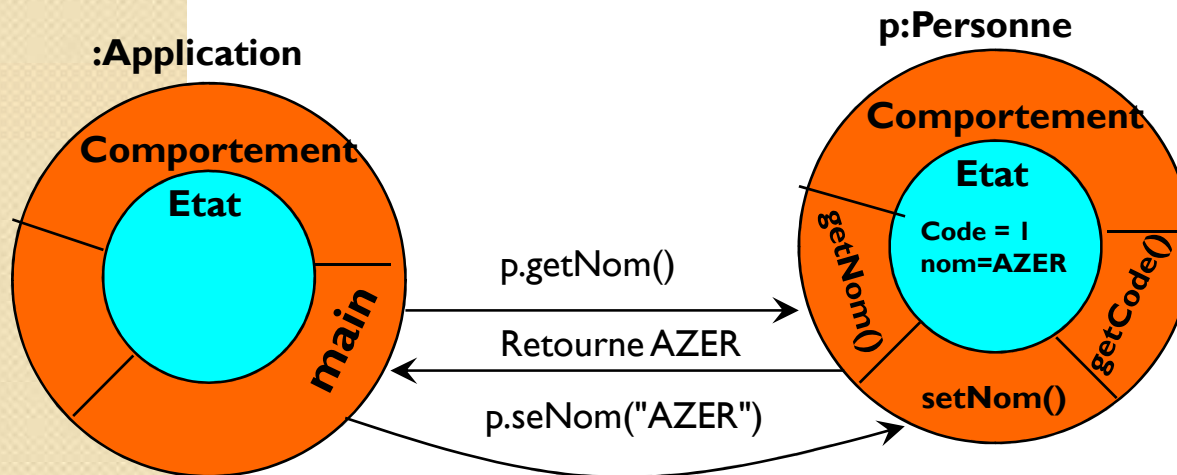
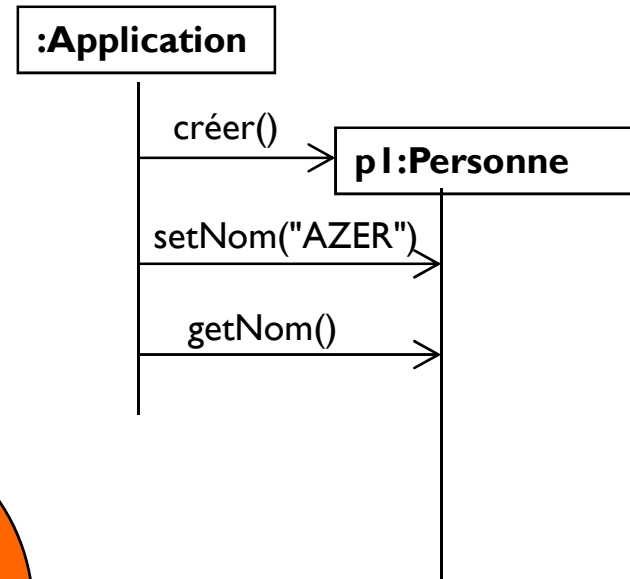
```
public string GetNom(){  
    return nom;  
}
```
- Les setters sont des méthodes qui commencent toujours par le mot set et finissent par le nom de l'attribut en écrivant en majuscule la lettre qui vient juste après le set. Les setters sont toujours de type void et reçoivent un paramètre qui est de meme type que la variable:
  - Exemple:

```
public void SetNom( string n ){  
    this.nom=n;  
}
```

# Encapsulation

```
public class Application {  
    static void Main(string[] args) {  
        Personne p=new Personne();  
        p.SetNom( "AZER" );  
        Console.WriteLine(p.GetNom() );  
    }  
}
```

## Diagramme de séquence :



- Généralement, l'état d'un objet est privé ou protégé et son comportement est publique
- Quand l'état de l'objet est privé Seules les méthode de ses qui ont le droit d'y accéder
- Quand l'état de l'objet est protégé, les méthodes des classes dérivées et les classes appartenant au même package peuvent également y accéder

# Les propriétés

- Il existe une autre façon d'avoir accès aux attributs d'une classe, c'est de créer des propriétés.
- Celles-ci nous permettent de manipuler des attributs privés comme s'ils étaient publics.
- Une propriété permet de lire (**get**) ou de fixer (**set**) la valeur d'un attribut. Une propriété est déclarée comme suit :

```
public Type Propriete{  
    get {...}  
    set {...}  
}
```

# Les propriétés

- Exemple de classe :

```
namespace metier {  
    class Personne {  
        private int code;  
        private string nom;  
  
        public string Nom  
        {  
            get { return nom; }  
            set { this.nom = value; }  
        }  
        public int Code  
        {  
            get { return code; }  
            set { this.code = value; }  
        }  
    }  
}
```

Exemple d'utilisation:

```
static void Main(string[] args)  
{  
    Personne p = new Personne();  
    p.Code = 7;  
    p.Nom = "AZER";  
    Console.WriteLine(p.Code);  
    Console.WriteLine(p.Nom);  
}
```

# Membres statiques d'une classe.

- Dans l'exemple de la classe Compte, chaque objet Compte possède ses propres variables code et solde. Les variables code et solde sont appelées variables d'instances.
- 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.
- Dans la notation UML, les membres statiques d'une classe sont soulignés.

# Exemple:

- Supposant nous voulions ajouter à la classe Compte une variable qui permet de stocker le nombre le comptes créés.
- Comme la valeur de variable nbComptes est la même pour tous les objets, celle-ci sera déclarée statique. Si non, elle sera dupliquée dans chaque nouveau objet créé.
- La valeur de nbComptes est au départ initialisée à 0, et pendant la création d'une nouvelle instance (au niveau du constructeur), nbCompte est incrémentée et on profite de la valeur de nbComptes pour initialiser le code du compte.

Compte
- code : int # solde : float - <u>nbComptes:int</u>
+ Compte(float solde) + verser(float mt):void + retirer(float mt):void + toString():String + <u>getNbComptes():int</u>

```
using System;
namespace metier
{
    public class Compte
    {
        private int code; protected float solde;
        private static int nbComptes;
        public Compte(float s)
        {
            this.code = ++nbComptes;
            this.solde = s;
        }
        public void Verser(float mt)
        {
            solde = solde + mt;
        }
        public void Retirer(float mt)
        {
            solde = solde - mt;
        }
        public static int GetNbComptes()
        {
            return nbComptes;
        }
        public override string ToString()
        {
            return (" Code=" + code + " Solde=" +
solde);
        }
    }
}
```



# Application de test

```
using System;
using metier;
namespace test
{
    class Application
    {
        static void Main(string[] args)
        {
            Compte c1 = new Compte(5000);
            Compte c2 = new Compte(6000);
            c1.Verser(3000); c1.Retirer(2000);
            Console.WriteLine(c1);
            Console.WriteLine("NB Comptes="+
Compte.GetNbComptes());
        }
    }
}
```

Classe Compte
<b><u>nbCompte=2</u></b>
<b><u>getNbComptes()</u></b>

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

**Code=1 Solde= 6000**  
**2**  
**2**



# Destructeur

- Le destructeur est appelé au moment de la destruction de l'objet.
- Une classe peut posséder un seul destructeur.
- Les destructeurs ne peuvent pas être hérités ou surchargés.
- Les destructeurs ne peuvent pas être appelés. Ils sont appelés automatiquement.
- Un destructeur n'accepte pas de modificateurs ni de paramètres.

# Destruction des objets : Garbage Collector

- Dans certains langages de programmation, le programmeur doit s'occuper lui-même de détruire les objets inutilisables.
- L'environnement d'exécution Dot Net détruit automatiquement tous les objets inutilisables en utilisant ce qu'on appelle le **garbage collector (ramasseur de miettes)**. Qui s'exécute automatiquement dès que la mémoire disponible est inférieure à un certain seuil.
- Tous les objets qui ne sont pas retenus par des références seront détruits.
- Le garbage collector peut être appelé explicitement en faisant appel à la méthode statique Collect de la classe GC : **GC.Collect()**

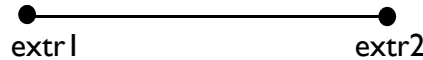
# Exemple

```
namespace metier {  
    class Personne{  
        private int code;  
        private string nom;  
        // Constructeur  
        public Personne(){  
            System.Console.WriteLine("Constructeur");  
        }  
        // Destructeur  
        ~Personne()  
        {  
            System.Console.WriteLine("Destructeur");  
        }  
    }  
}
```

# Exemple

```
using metier;
namespace test{
    class Application    {
        static void Main(string[] args)        {
            Personne p = new Personne();
            Console.WriteLine("Objet Personne Créé : Appuyer sur une touche!");
            Console.ReadLine();
            p = null;
            Console.WriteLine("L'objet est libéré : Appuyer sur une touche!");
            Console.ReadLine();
            GC.Collect();
            Console.WriteLine("Le garbage collector est appelé explicitement!");
            Console.ReadLine();
        }
    }
}
```

# Exercice I : Modélisation d'un segment



- On souhaite créer une application qui permet de manipuler des segments.
- Un segment est défini par la valeur de ses deux extrémités `extr1` et `extr2`.
- Pour créer un segment, il faut préciser les valeurs de `extr1` et `extr2`.
- Les opérations que l'on souhaite exécuter sur le segment sont :
  - `ordonne()` : méthode qui permet d'ordonner `extr1` et `extr2` si `extr1` est supérieur à `extr2`
  - `getLongueur()` : méthode qui retourne la longueur du segment.
  - `appartient(int x)` : retourne si `x` appartient au segment ou non.
  - `toString()` : retourne une chaîne de caractères de type `SEGMENT[extr1,extr2]`
- Faire une représentation UML de la classe `Segment`.
- Implémenter en `C#` la classe `Segment`
- Créer une application `TestSegment` qui permet de :
  - Créer objet de la classe `Segment` avec les valeurs `extr1=24` et `extr2=12`.
  - Afficher l'état de cet objet en utilisant la méthode `toString()`.
  - Afficher la longueur de ce segment.
  - Afficher si le point `x=15`, appartient à ce segment.
  - Changer les valeurs des deux extrémités de ce segment.
  - Afficher à nouveau la longueur du segment.

# Diagramme de classes

<b>Segment</b>
+ extr1 : int + extr2 : int
+ Segment (int e1,int e2) + ordonne() + getLongueur() : int + appartient(int x) : boolean + toString() : String

<b>TestSegment</b>
<u>+ main(String[] args):void</u>



# Solution : Segment.java

```
using System;
namespace metier {
    public class Segment    {
        private int extr1; private int extr2;
        public Segment(int a, int b) {
            extr1 = a; extr2 = b; ordonne();
        }
        public void ordonne() {
            if (extr1 > extr2) {
                int z = extr1; extr1 = extr2; extr2 = z;
            }
        }
        public int getLongueur() {
            return (extr2 - extr1);
        }
        public bool appartient(int x){
            return ((x > extr1) && (x < extr2))
        }
        public override string ToString() {
            return ("segment[" + extr1 + "," + extr2 + "]");
        }
    }
}
```

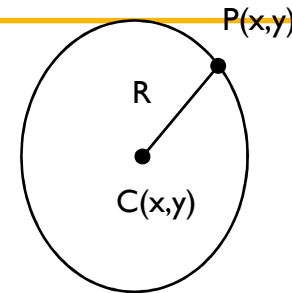
# Application

```
using System;
using metier;
namespace test
{
    class Application
    {
        static void Main(string[] args)
        {
            Console.Write("Donner Extr1:");
            int e1 = Int32.Parse(Console.ReadLine());
            Console.Write("Donner Extr2:");
            int e2 = Int32.Parse(Console.ReadLine());
            Segment s = new Segment(e1, e2);
            Console.WriteLine("Longueur du "+s.ToString()+" est:"+s.getLongueur());
            Console.Write("Donner X:");
            int x = Int32.Parse(Console.ReadLine());
            if (s.appartient(x))
                Console.WriteLine(x + " appartient au " + s);
            else
                Console.WriteLine(x + " n'appartient pas au " + s);
        }
    }
}
```

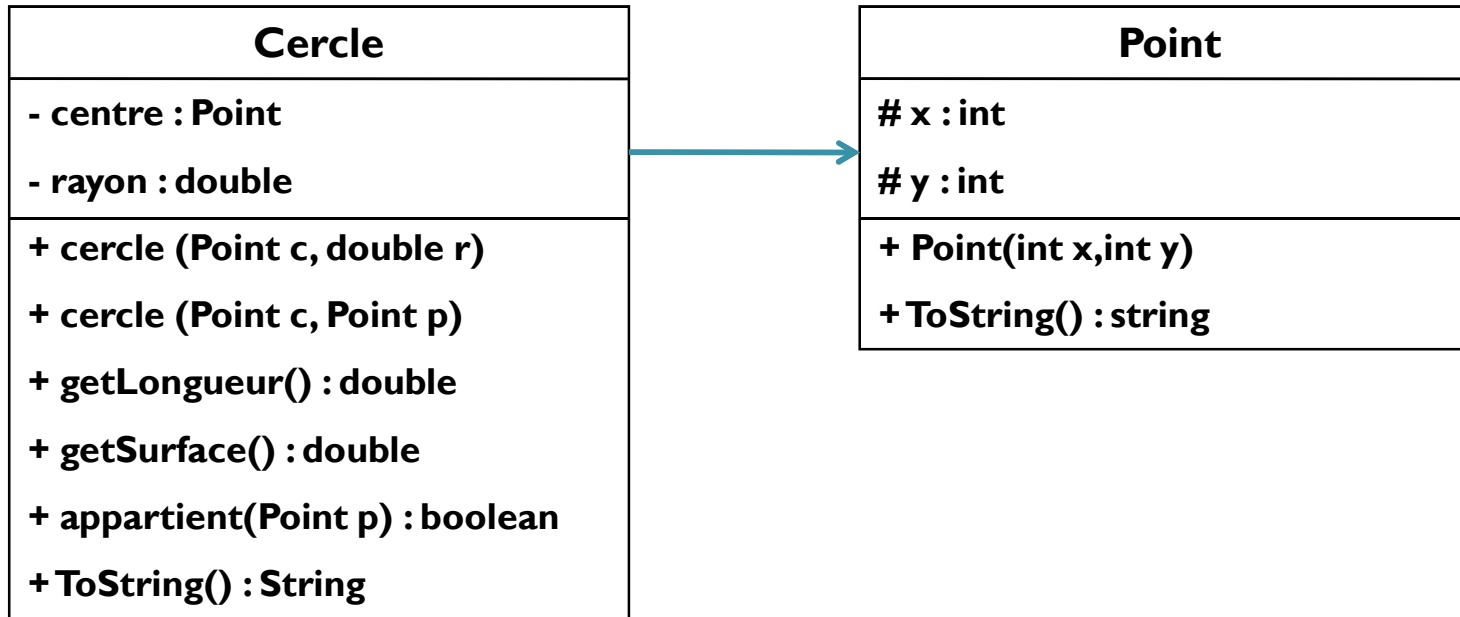
```
Donner Extr1:67
Donner Extr2:13
Longueur dusegment[13,67] est :54
Donner X:7
7 N'appartient pas au segment[13,67]
```

## Exercice 2

- Une cercle est défini par :
    - Un point qui représente son centre :  $\text{centre}(x,y)$  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 :
    - $\text{getPerimetre}()$  : retourne le périmètre du cercle
    - $\text{getSurface}()$  : retourne la surface du cercle.
    - $\text{appartient}(\text{Point } p)$  : retourne si le point  $p$  appartient ou non à l'intérieur du cercle.
    - $\text{toString}()$  : retourne une chaîne de caractères de type  $\text{CERCLE}(x,y,R)$
1. Etablir le diagramme de classes
  2. Créer la classe Point définie par:
    - Les attributs  $x$  et  $y$  de type  $\text{int}$
    - Un constructeur qui initialise les valeurs de  $x$  et  $y$ .
    - Une méthode  $\text{toString}()$ .
  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.



# Diagramme de classe





# **HÉRITAGE ET ACCESSIBILITÉ**



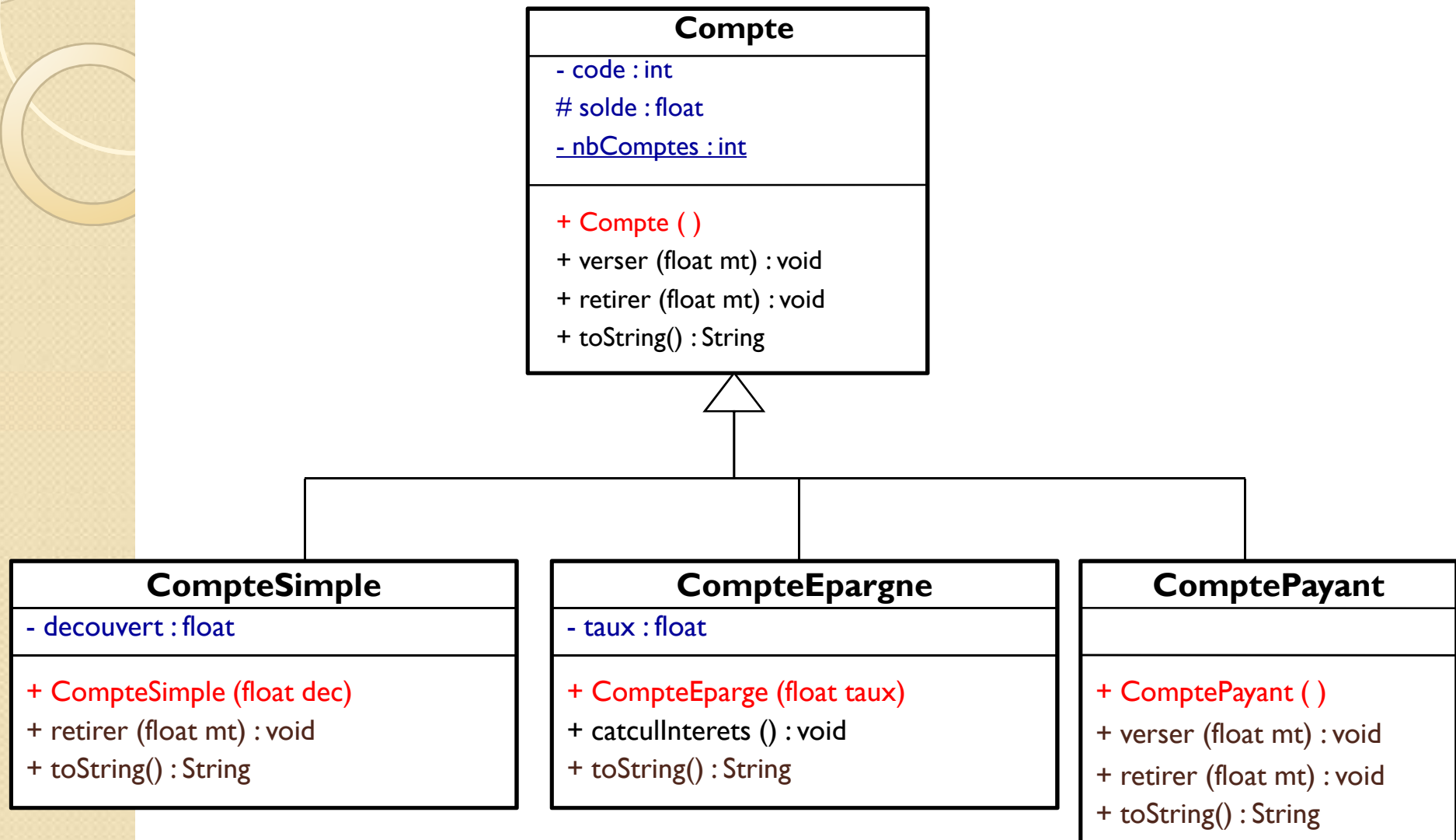
# Héritage

- 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 souhaitions 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 et un solde
  - 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 C# de la classe Compte

```
using System;
namespace metier {
    public class Compte{
        private int code; protected float solde;
        private static int nbComptes;
        public Compte(float s){
            this.code = ++nbComptes; this.solde = s;
        }
        public Compte() {
            this.code = ++nbComptes; this.solde = 0;
        }
        public void Verser(float mt){
            solde = solde + mt;
        }
        public void Retirer(float mt) {
            solde = solde - mt;
        }
        public static int GetNbComptes() {
            return nbComptes;
        }
        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 les deux points « : »

```
namespace metier
```

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

- La classe CompteSimple hérite de la classe CompteBancaire 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.

# Définir les constructeurs de la classe dérivée

- Le constructeur de la classe dérivée peut faire appel au constructeur de la classe parente en utilisant le mot clé **base()** :

```
namespace metier
{
    class CompteSimple :Compte
    {
        private float decouvert;
        public CompteSimple(float dec):base()
        {
            this.decouvert = dec;
        }
        public CompteSimple(float s,float dec): base(s)
        {
            this.decouvert = dec;
        }
    }
}
```

# 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.
- Une méthode redéfinie est précédée par le mot clé **new** si la méthode de la classe parente n'est pas virtuelle.
- Si la méthode de la classe parente est déclarée virtual, la redéfinition de la méthode doit utiliser le mot clé **override**

```
namespace metier
{
    class CompteSimple :Compte
    {
        private float decouvert;
        // Constructeurs
        // Redéfinition de la méthode retirer
        public new void Retirer(float mt)
        {
            if (mt - decouvert <= solde)
                solde -= mt;
        }
    }
}
```

# Redéfinition des méthodes

- 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().

```
namespace metier{
    class CompteSimple :Compte {
        private float decouvert;
        // Redéfinition de la méthode retirer
        public new void Retirer(float mt)
        {
            if (mt - decouvert <= solde)
                solde -= mt;
        }
        // Redéfinition de la méthode ToString
        public override string ToString()
        {
            return base.ToString()+" Découvert="+decouvert;
        }
    }
}
```

# 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
- Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot **this()** avec des paramètres éventuels

# Surcharge de constructeurs

```
class CompteSimple :Compte
{
    private float decouvert;

    public CompteSimple(float s,float dec) : base(s)
    {
        this.decouvert = dec;
    }
    public CompteSimple(float dec) : this(0, dec) { }
}
```

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

```
CompteSimple cs1=new CompteSimple(6000,5000);  
CompteSimple cs2=new CompteSimple(5000);
```



# Accessibilité



# 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 :
  - Connaître le package où se trouve la classe (*Où*)
    - Importer la classe en spécifiant son package.
  - 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
  - Connaître qui a le droit d'accéder aux membres de cette instance.

# Ce qui peut être fait(Quoi)

- Pour qu'une classe puisse être utilisée (directement ou par l'intermédiaire d'un de ses membres), il faut non seulement être capable de la trouver, mais aussi qu'elle soit adaptée à l'usage que l'on veut en faire.
- Une classe peut servir à plusieurs choses :
  - Créer des objets, en étant **instanciée**.
  - Créer de nouvelles classes, en étant **étendue**.
  - On peut utiliser directement ses membres statiques (sans qu'elle soit instanciée.)
  - On peut utiliser les membres de ses instances.

# Classe abstraite

- 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.

```
using System;
namespace metier
{
    public abstract class Compte {
        private int code; protected float solde;
        private static int nbComptes;
        // Suite
    }
}
```

# 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.
- Pour redéfinir une méthode abstraite dans une classe dérivée, on utilise le mot clé **override**
- 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

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

```
public class CompteSimple:Compte {  
    // ...  
    // Obligation de redéfinir la méthode afficher  
    public override void afficher(){  
        Console.WriteLine("Solde="+solde+"  
        Découvert="+decouvert);  
    }  
}
```

# Interfaces

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

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

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

```
public class CompteSimple : Compte, Solvable {  
    private float decouvert;  
    // Redefinir la méthode abstraite de la classe Compte  
    public override void afficher() {  
        Console.WriteLine("Solde="+solde+" Découvert="+decouvert);  
    }  
    // Redefinir les méthodes abstraites de l'interface Solvable  
    public double getSolde() {  
        return solde;  
    }  
    public void solver() {  
        this.solde=0;  
    }  
}
```

# Classe de type sealed

- Une classe de type **sealed** est une classes qui ne peut pas être dérivée.
- Autrement dit, on ne peut pas hériter d'une classe final.
- La classe CompteSimple peut être déclarée final en écrivant:  

```
public sealed class CompteSimple : Compte {  
  
}
```

# méthodes sealed

- Une méthode sealed est une méthode qui ne peut pas être redéfinie dans les classes dérivées.
  - Exemple : La méthode verser de la classe suivante ne peut pas être redéfinie dans les classes dérivées car elle est déclarée final

```
public class Compte {  
    private int code; protected float solde;  
    private static int nbComptes;  
  
    public sealed void verser(float mt) {  
        solde+=mt;  
    }  
    public void retirer(float mt) {  
        if(mt<solde) solde-=mt;  
    }  
}
```



# Variable readonly

- Pour définir une constante en C#, utilisez le modificateur **const** ou **readonly** au lieu du mot clé **final** de Java.
- Le facteur qui fait la distinction entre les deux modificateurs en C# est que les éléments **const** sont gérés au moment de la compilation, alors que les valeurs de champs **readonly** sont spécifiées au moment de l'exécution.
- Cela signifie que l'assignation aux champs **readonly** peut se produire dans le constructeur de classe aussi bien que dans la déclaration.
- Exemple :

```
class Parametrage{  
    public readonly float taux=5;  
    public Parametrage(int t ){  
        taux = t;  
    }  
}
```

# Qui peut le faire (Qui):

- **private:**
  - L'autorisation **private** est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes et classes internes).
  - Les éléments déclarés **private** ne sont accessibles que depuis la classe qui les contient.
  - Ce type d'autorisation est souvent employé pour les variables qui ne doivent être modifiées ou lues qu'à l'aide d'un *getter* ou d'un *setter*.
- **public:**
  - Un membre public d'une classe peut être utilisé par n'importe quelle autre classe.
  - En UML les membres publics sont indiqués par le signe +
- **protected:**
  - Les membres d'une classe peuvent être déclarés **protected**.
  - Dans ce cas, l'accès en est réservé aux classes dérivées
- **Internal** : L'accès est restreint à l'assembly en cours. (dll contenant la classe)
- **protected internal** : L'accès est restreint à l'assembly en cours ou aux types dérivés de la classe conteneur.
- Autorisation par défaut : en l'absence de l'un des modificateurs précédents, l'autorisation d'accès par défaut est **private**



# Polymorphisme

# Polymorphisme

- Le polymorphisme offre aux objets la possibilité d'appartenir à plusieurs catégories à la fois.
- En effet, nous avons certainement tous appris à l'école qu'il était impossible d'additionner des pommes et des oranges
- Mais, on peut écrire l'expression suivante:

**3    pommes    +    5    oranges    =    8**  
**fruits**

# Polymorphisme

- **Le sur-casting des objets:**

- Une façon de décrire l'exemple consistant à additionner des pommes et des oranges serait d'imaginer que nous disons pommes et oranges mais que nous manipulons en fait des fruits. Nous pourrions écrire alors la formule correcte :

```
3 (fruits) pommes
+ 5 (fruits) oranges
```

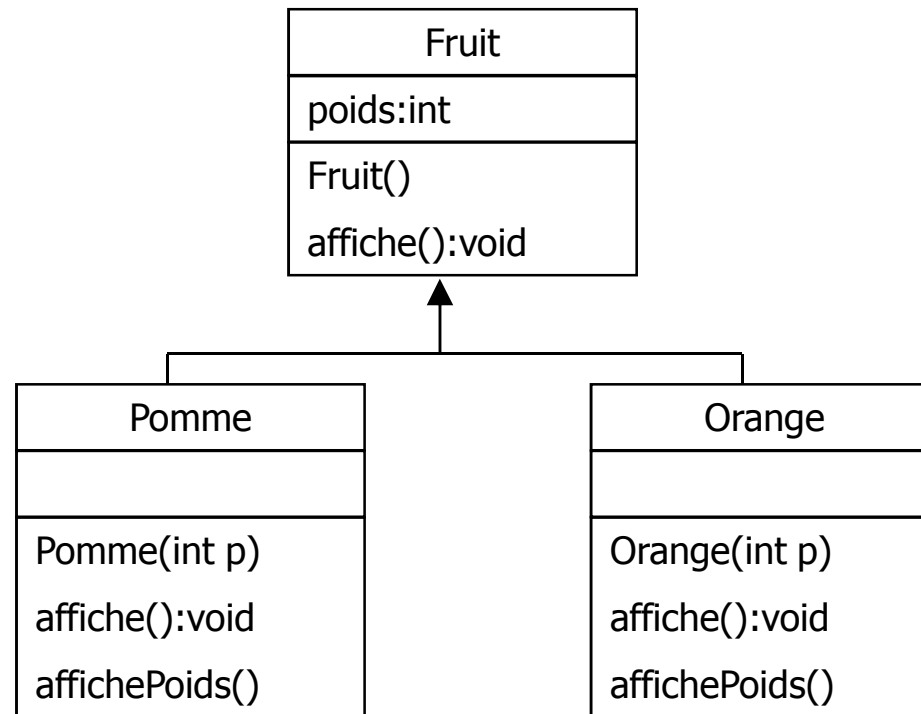
```
-----
--
```

```
= 8 fruits
```

- Cette façon de voir les choses implique que les pommes et les oranges soient "transformés" en fruits préalablement à l'établissement du problème. Cette transformation est appelée *sur-casting*

# Instanciation et héritage

- Considérons l'exemple suivant:



# Instanciation et héritage

```
using System;
namespace metier {
    public abstract class Fruit {
        protected int poids;
        public Fruit(){
            Console.WriteLine("Création d'un fruit");
        }
        public virtual void affiche() {
            Console.WriteLine("C'est un Fruit");
        }
    }
}
```

```
using System;
namespace metier {
    public class Pomme : Fruit {
        public Pomme(int p) {
            this.poids = p;
        }
        Console.WriteLine("Création d'une Pomme de " +
            poids + " grammes");
        public override void affiche() {
            Console.WriteLine("C'est une pomme");
        }
        public void affichePoids(){
            Console.WriteLine("Le poids de cette pomme
est +" + poids + " grammes");
        }
    }
}
```

```
using System;
namespace metier {
    public class Orange : Fruit {
        public Orange(int p) {
            this.poids = p;
        }
        Console.WriteLine("Création d'une Orange
de " + poids + " grammes");
        public override void affiche() {
            Console.WriteLine("C'est une Orange");
        }
        public void affichePoids(){
            Console.WriteLine("Le poids de cette
orange est +" + poids + " grammes");
        }
    }
}
```

```
using metier;
class Application
{
    static void Main(string[] args)
    {
        Pomme p = new Pomme(72);
        Orange o = new Orange(80);
    }
}
```

# Instanciación et héritage

- Le résultat affiché par le programme est:  
Création d'un fruit  
Création d'une pomme de 72 grammes  
Création d'un fruit  
création d'une orange de 80 grammes
- Nous constatons qu'avant de créer une Pomme, le programme crée un Fruit, comme le montre l'exécution du constructeur de cette classe. La même chose se passe lorsque nous créons une Orange



# Sur-casting des objets

- Considérons l'exemple suivant:

```
using metier;
class Application
{
    static void Main(string[] args)
    {
        // Sur-casting implicite
        Fruit f1 = new Orange(40);
        // Sur-casting explicite
        Fruit f2 = (Fruit)new Pomme(60);
        // Sur-casting implicite
        f2 = new Orange(40);
    }
}
```

# Sur-casting des objets

- Un objet de type Pomme peut être affecté à un handle de type fruit sans aucun problème :
  - Fruit f1;
  - f1=new Pomme(60);
- Dans ce cas l'objet Pomme est converti automatiquement en Fruit.
- On dit que l'objet Pomme est sur-casté en Fruit.
- Dans java, 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 :
  - f2=(**Fruit**)new Orange(40);

# Sous-Casting explicite des objets

- Considérons l'exemple suivant:

```
using metier;  
class Application  
{  
    static void Main(string[] args)  
    {  
        Fruit f1;  
        Fruit f2;  
        f1 = new Pomme(60);  
        f2 = new Orange(40);  
        f1.affichePoids(); // Erreur de compilation  
        ((Pomme)f1).affichePoids(); // Solution  
    }  
}
```

- Solution : Sous-casting explicite

# Sous-casting explicite des objets

- Ce message indique que l'objet `f1` qui est de type `Fruit` ne possède pas la méthode `affichePoids()`.
- Cela est tout à fait vrai car cette méthode est définie dans les classes `Pomme` et `Oranges` et non dans la classe `Fruit`.
- En fait, même si le handle `f1` pointe un objet `Pomme`, le compilateur ne tient pas en considération cette affectation, et pour lui `f1` est un `Fruit`.
- Il faudra donc convertir explicitement l'objet `f1` qui de type `Fruit` en `Pomme`.
- Cette conversion s'appelle Sous-casting qui indique la conversion d'un objet d'une classe vers un autre objet d'une classe dérivée.
- Dans ce cas de figure, le sous-casting doit se faire explicitement.
- L'erreur de compilation peut être évitée en écrivant la syntaxe suivante :
  - **`((Pomme)f1).affichePoids();`**
- Cette instruction indique que l'objet `f1`, de type `Fruit`, est converti en `Pomme`, ensuite la méthode `affichePoids()` de l'objet `Pomme` est appelé ce qui est correcte.

# Sous casting implicite des objets

```
using metier;  
class Application {  
    static void Main(string[] args) {  
        Fruit f1;Fruit f2;  
        f1 = new Pomme(60); f2 = new Orange(40);  
        f1.affiche();// Sous casting implicite  
        f2.affiche(); // Sous casting implicite  
    }  
}
```

L'exécution de ce programme donne :

```
Création d'un fruit  
Création d'une pomme de 60 grammes  
Création d'un fruit  
Création d'une orange de 40 grammes  
C'est une pomme  
C'est une Orange
```

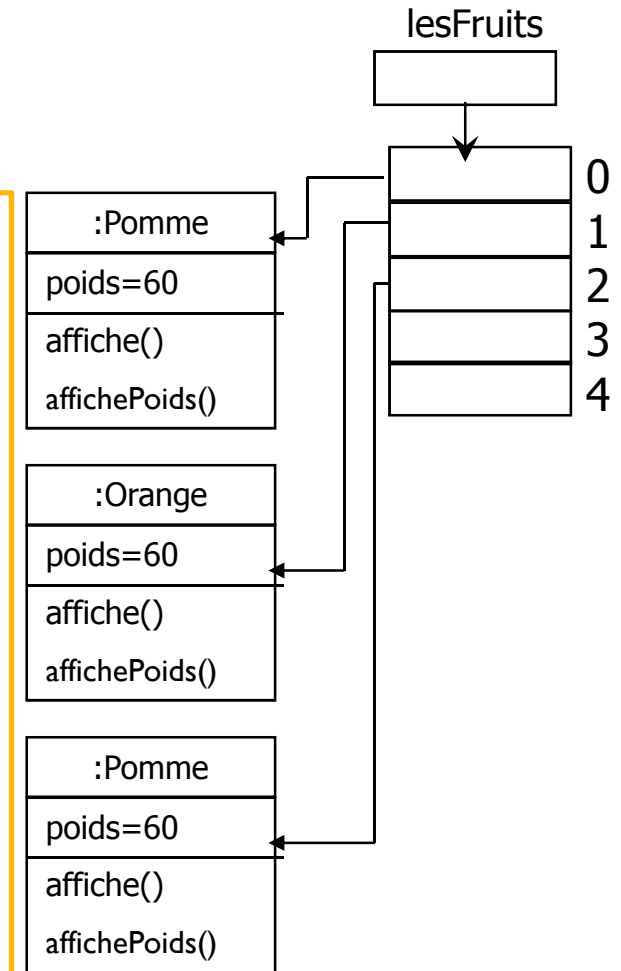


# Tableaux d'objets et Collections

# Tableaux d'objets

- Déclaration :
  - Exemple : Tableau d'objets Fruit
  - `Fruit[] lesFruits;`
- Création du tableau
  - `lesFruits = new Fruit[5];`
- Création des objets:
  - `lesFruits[0]=new Pomme(60);`
  - `lesFruits[1]=new Orange(100);`
  - `lesFruits[2]=new Pomme(55);`
- Manipulation des objets:

```
for(int i=0;i<lesFruits.length;i++){
    lesFruits[i].affiche();
    if(lesFruits[i] is Pomme)
        ((Pomme)lesFruits[i]).affichePoids();
    else
        ((Orange)lesFruits[i]).affichePoids();
}
```
- Un tableau d'objets est un tableau de références



# Collections

- Une collection est un tableau dynamique d'objets de type Object.
- Une collection fournit un ensemble de méthodes qui permettent:
  - D'ajouter un nouveau objet dans le tableau
  - Supprimer un objet du tableau
  - Rechercher des objets selon des critères
  - Trier le tableau d'objets
  - Contrôler les objets du tableau
  - Etc...
- Dans un problème, les tableaux peuvent être utilisés quand la dimension du tableau est fixe.
- Dans le cas contraire, il faut utiliser les collections
- C# fournit plusieurs types de collections:
  - List
  - ArrayList
  - Dictionary
  - HashSet
  - Etc...
- Dans cette partie du cours, nous allons présenter uniquement comment utiliser les collections ArrayList, Vector, Iterator et HashMap
- Vous aurez l'occasion de découvrir les autres collections dans les prochains cours



# Collection générique List

- ArrayList est une classe du package java.util, qui implémente l'interface List.
- Déclaration d'une collection de type List qui devrait stocker des objets de type Fruit:
  - `List<Fruit> fruits;`
- Création de la liste:
  - `fruits=new List<Fruit>();`
- Ajouter deux objets de type Fruit à la liste:
  - `fruits.Add(new Pomme(30));`
  - `fruits.Add(new Orange(25));`
- Faire appel à la méthode affiche() de tous les objets de la liste:
  - En utilisant la boucle for :  

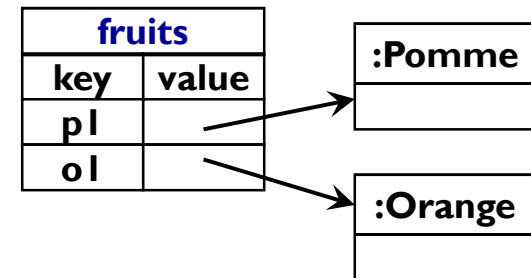
```
for (int i = 0; i < fruits.Count; i++) { fruits[i].affiche(); }
```
  - En utilisant la boucle foreach  

```
foreach (Fruit f in fruits) { f.affiche(); }
```
- Supprimer le deuxième Objet de la liste
  - `fruits.RemoveAt(1);`

# Collection de type Dictionary

- La collection Dictionary est une classe qui permet de créer un tableau dynamique d'objet de type Object qui sont identifiés par une clé.
- Déclaration et création d'une collection de type Dictionary qui contient des fruits identifiés par une clé de type string :
- `Dictionary<string, Fruit> fruits=new Dictionary<string, Fruit>();`
- Ajouter deux objets de type Fruit à la collection
  - `fruits.Add("p1", new Pomme(40));`
  - `fruits.Add("o1", new Orange(60));`
- Récupérer un objet de cette collection:
  - `Fruit f1 = fruits["o1"];`
  - `f1.affiche();`
- Parcourir toute la collection:

```
foreach (Fruit f in fruits.Values)
{
    f.affiche();
}
```



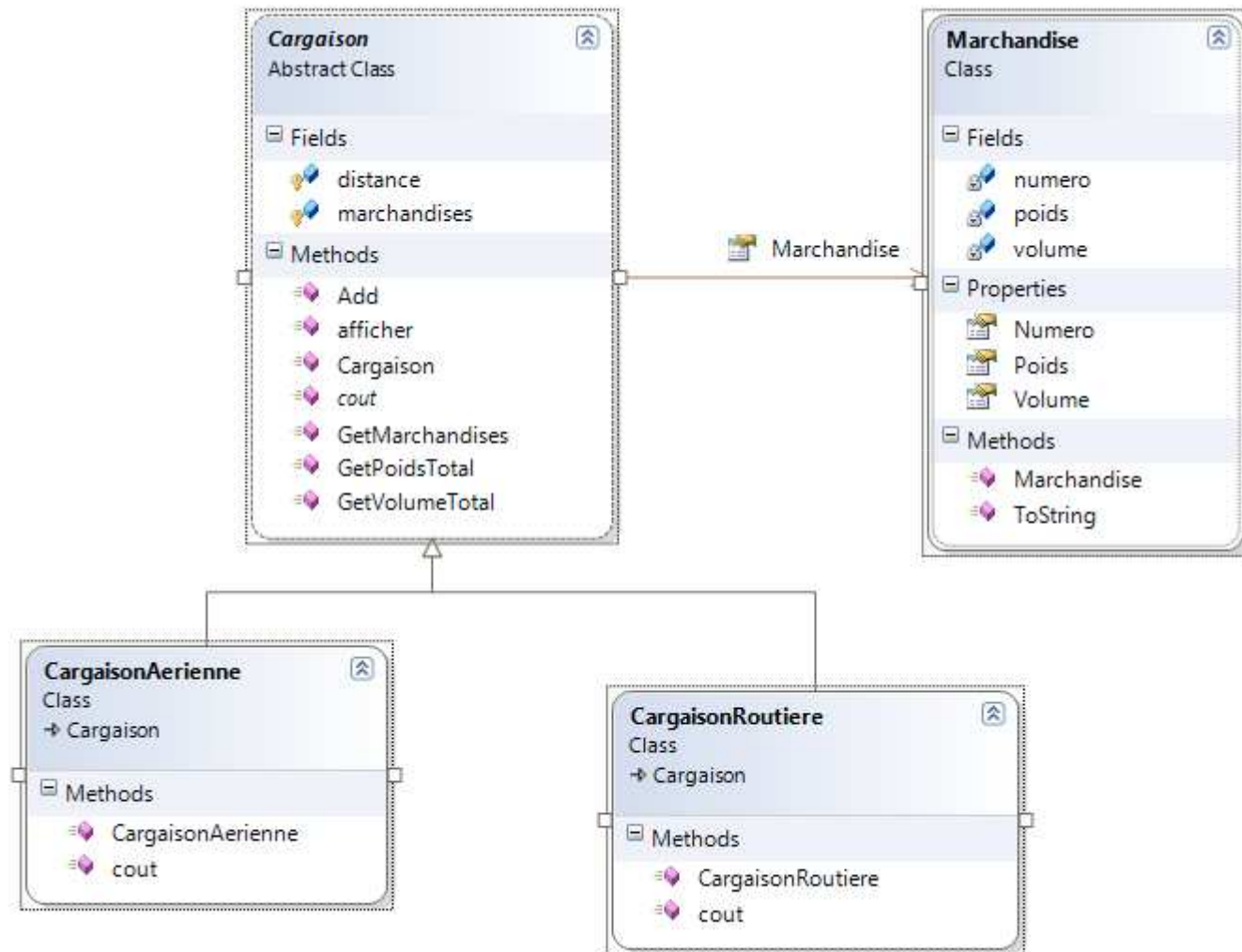
# Application

- 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

# Diagramme de classes



# Classe Marchandise

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

namespace transport {
    public class Marchandise {
        private int numero;
        public int Numero {
            get { return numero; } set { numero = value; }
        }
        private double poids;
        public double Poids {
            get { return poids; } set { poids = value; }
        }
        private double volume;
        public double Volume {
            get { return volume; } set { volume = value; }
        }
        public Marchandise(int num, double p, double v)
        {
            this.numero = num; this.poids = p; this.volume = v;
        }
        public override string ToString() {
            return "Num=" + numero + " Poids=" + poids + " vol=" + volume;
        }
    }
}
```

# Classe Cargaison

```
using System;using System.Collections.Generic; using System.Linq; using
    System.Text;
using System.Collections;
namespace transport{
    public abstract class Cargaison    {
        protected int distance;
        protected List<Marchandise> marchandises = new List<Marchandise>();
        public List<Marchandise> getMarchandises() { return marchandises; }
        public Cargaison(int d) { this.distance = d; }
        public void Add(Marchandise m) {  marchandises.Add(m); }
        public void afficher() {
            foreach (Marchandise m in marchandises) {
                Console.WriteLine(m.ToString());
            }
        }
        public Marchandise GetMarchandises(int num) {
            foreach (Marchandise m in marchandises) {
                if (m.Numero == num) return m;
            }
            return null;
        }
    }
```

# Classe Cargaison

```
public double GetPoidsTotal() {  
    double P = 0;  
    foreach (Marchandise m in marchandises) {  
        P += m.Poids;  
    }  
    return P;  
}  
  
public double GetVolumeTotal() {  
    double V = 0;  
    foreach (Marchandise m in marchandises) {  
        V += m.Poids;  
    }  
    return V;  
}  
  
public abstract double cout();  
  
}  
  
}
```



# Classe CargaisonAerienne

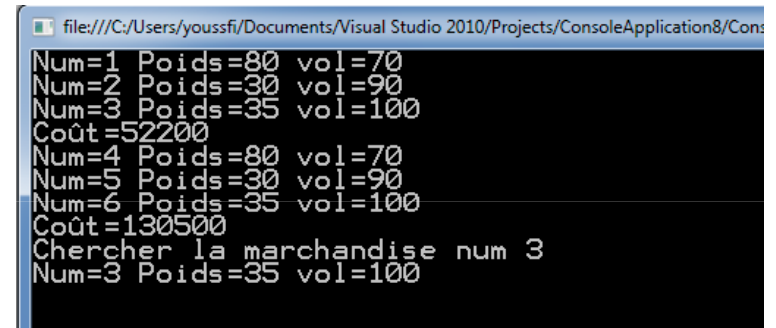
```
using System; using System.Collections.Generic;using System.Linq;
using System.Text;
namespace transport {
    class CargaisonAerienne : Cargaison
    {
        public CargaisonAerienne(int d): base(d) { }
        public override double cout()
        {
            if (GetVolumeTotal() < 80000)
                return 10 * distance * GetVolumeTotal();
            else
                return 12 * distance * GetPoidsTotal();
        }
        public override string ToString(){
            return "Cargaison Aérienne Distance =" + distance;
        }
    }
}
```

# Classe CargaisonRoutiere

```
using System; using System.Collections.Generic;
using System.Linq; using System.Text;
namespace transport {
    class CargaisonRoutiere :Cargaison
    {
        public CargaisonRoutiere(int d):base(d) { }
        public override double cout() {
            if (GetVolumeTotal() < 380000)
                return 4 * distance * GetVolumeTotal();
            else
                return 6 * distance * GetPoidsTotal();
        }
        public override string ToString(){
            return "Cargaison Routière Distance =" + distance;
        }
    }
}
```

# Application

```
using System; using System.Collections.Generic; using System.Linq;
using System.Text; using transport;
namespace test {
    class Program {
        static void Main(string[] args) {
            Cargaison cr = new CargaisonRoutiere(90);
            cr.Add(new Marchandise(1, 80, 70));
            cr.Add(new Marchandise(2, 30, 90));
            cr.Add(new Marchandise(3, 35, 100));
            cr.afficher();
            Console.WriteLine("Coût="+cr.cout());
            Cargaison ca = new CargaisonAerienne(90);
            ca.Add(new Marchandise(4, 80, 70));
            ca.Add(new Marchandise(5, 30, 90));
            ca.Add(new Marchandise(6, 35, 100));
            ca.afficher();
            Console.WriteLine("Coût=" + ca.cout());
            Console.WriteLine("Chercher la marchandise num 3");
            Marchandise m = cr.GetMarchandises(3);
            Console.WriteLine(m);
            Console.ReadLine();
        }
    }
}
```



```
file:///C:/Users/youssfi/Documents/Visual Studio 2010/Projects/ConsoleApplication8/Cons:
Num=1 Poids=80 vol=70
Num=2 Poids=30 vol=90
Num=3 Poids=35 vol=100
Coût=52200
Num=4 Poids=80 vol=70
Num=5 Poids=30 vol=90
Num=6 Poids=35 vol=100
Coût=130500
Chercher la marchandise num 3
Num=3 Poids=35 vol=100
```

# Application Windows Forms

The application window, titled "Cargaisons", contains the following elements:

- Distance:** A text box containing the value "23".
- Type:** A dropdown menu currently showing "Aérienne".
- Ajouter:** A button to add the current entry.
- Num:** A text box containing "81".
- Poids:** A text box containing "9000".
- Volume:** A text box containing "80".
- Add:** A button to add the new entry.
- Table:** A table with 4 columns: NUM, Poids, and Volume. The first three rows contain data, and the fourth row is a summary row marked with an asterisk (\*).
- Coût de la cargaison:** A text box at the bottom showing the calculated cost "16376".

Below the "Ajouter" button, there is a list box showing the following text:

Cargaison Routière Distance =23  
transport.CargaisonAerienne

	NUM	Poids	Volume
	79	89	45
▶	80	89	45
	81	9000	80
*			

# Windows Form : Form I

```
using System; using System.Collections.Generic; using System.ComponentModel;
using System.Data; using System.Drawing; using System.Linq;
using System.Text; using System.Windows.Forms; using transport;
namespace ConsoleApplication8 {
    public partial class Form1 : Form {
        public Form1() { InitializeComponent();}
        private void button1_Click(object sender, EventArgs e)
        {
            int distance = Int32.Parse(textBoxDistance.Text);
            string type = (string)comboBoxType.SelectedItem;
            if (type.Equals("Routière"))
            {
                Cargaison c = new CargaisonRoutiere(distance);
                listBoxCargaisons.Items.Add(c);
            }
            else {
                Cargaison c = new CargaisonAerienne(distance);
                listBoxCargaisons.Items.Add(c);
            }
        }
    }
}
```

# Windows Form : Form I

```
private void button2_Click(object sender, EventArgs e)
{
    object o = listBoxCargaisons.SelectedItem;
    if (o != null)
    {
        Cargaison c = (Cargaison)o;
        int num = Int32.Parse(textBoxNum.Text);
        double poids = Double.Parse(textBoxPoids.Text);
        double volume = Double.Parse(textBoxVol.Text);
        Marchandise m = new Marchandise(num, poids, volume);
        c.Add(m);
        //listBoxMarchandises.Items.Add(m);
        dataGridViewMdises.Rows.Add(m.Numero, m.Poids, m.Volume);
    }
}
```

# Windows Form : Form I

```
private void listBoxCargaisons_SelectedIndexChanged(object sender,
    EventArgs e) {
    object o=listBoxCargaisons.SelectedItem;
    if(o!=null){
        dataGridViewMdises.Rows.Clear();
        Cargaison c = (Cargaison)o;
        if (c != null) {
            List<Marchandise> mdises = c.getMarchandises();
            foreach (Marchandise m in mdises)
            {
                dataGridViewMdises.Rows.Add(m.Numero, m.Poids, m.Volume);
            }
            textBoxCout.Text = c.cout().ToString();
        }
    }
}
```

# Application

```
using System;
using System.Collections.Generic;
using System.Linq; using System.Text;
using System.Windows.Forms;
namespace ConsoleApplication8
{
    class Application2
    {
        static void Main()
        {
            Application.Run(new Form1());
        }
    }
}
```

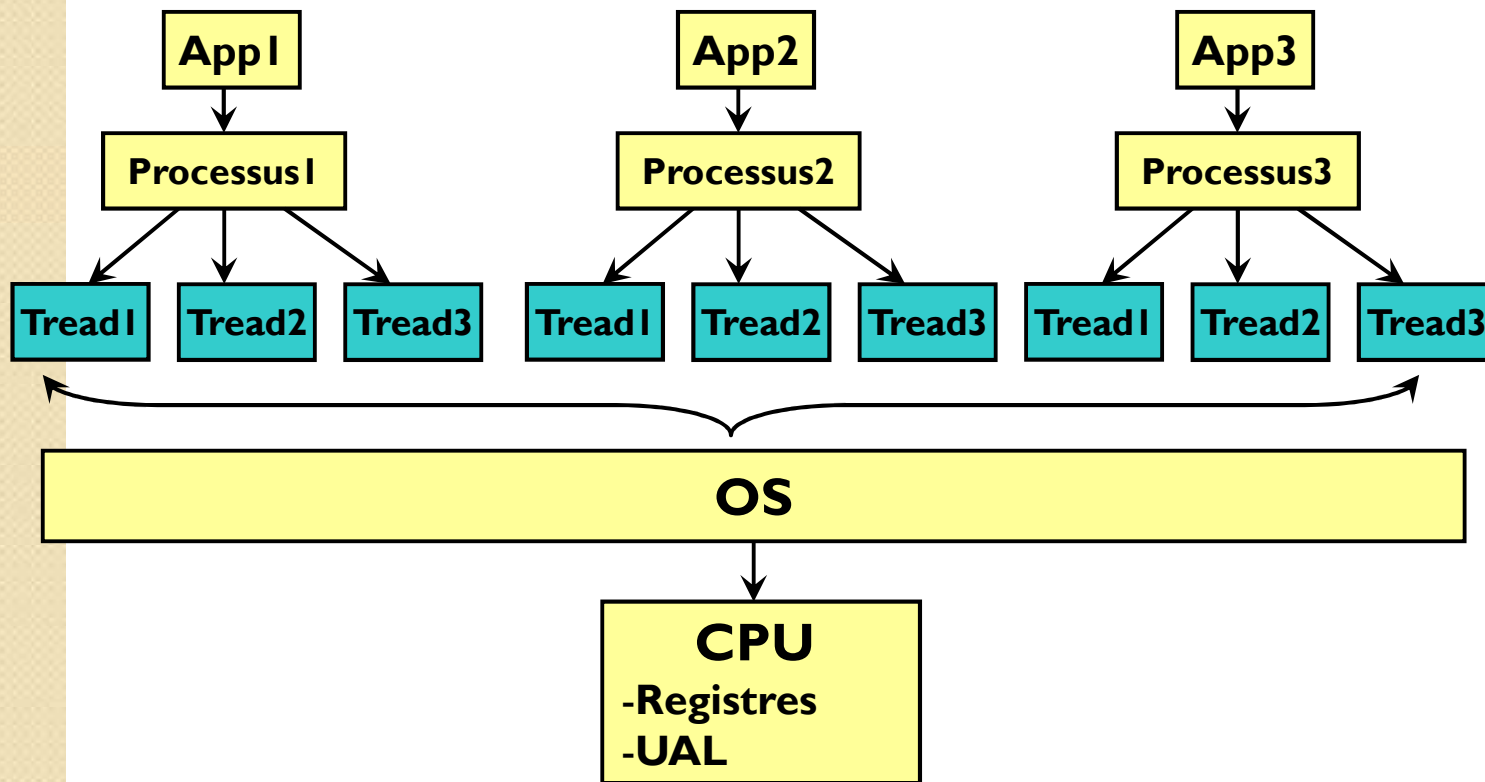




# Threads en C#

# Qu'est ce qu'un thread

- C# est un langage multi-threads
- Il permet d'exécuter plusieurs blocs d'instructions à la fois.
- Dans ce cas, chaque bloc est appelé Thread ( tâche ou fil )

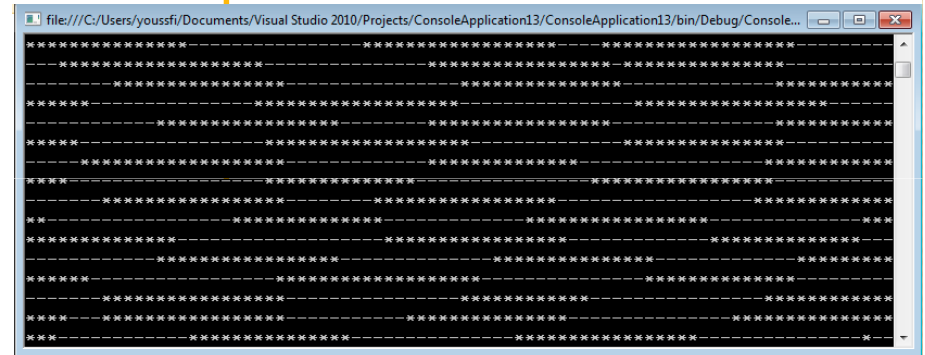


# Création d'un thread en C#

```
Thread t=new Thread(fonction);  
t.Start();
```

Exemple :

```
using System; using System.Collections.Generic;  
using System.Linq;using System.Text;  
using System.Threading;  
namespace Test {  
    class Program{  
        static void Main(string[] args){  
            Thread t1=new Thread(afficher);  
            Thread t2 = new Thread(afficher);  
            t1.Start("*");  
            t2.Start("-");  
            Console.ReadLine();  
        }  
        public static void afficher(object arg){  
            for (int i = 0; i < 1000; i++){  
                Console.Write(arg);  
                //Thread.Sleep(1000);  
            }  
        }  
    }  
}
```



# Gestion de la priorité d'un thread.

- Vous pouvez, en C#, jouer sur la priorité de vos threads.
- Sur une durée déterminée, un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread. Il exécutera donc, globalement, plus de code.
- Des constantes existent dans la classe **ThreadPriority** et permettent d'accéder à certains niveaux de priorités :
  - Lowest=0 , Normal =2, Highest=4, AboveNormal =3 et BelowNormal =1
- Pour spécifier la priorité d'un thread, il faut faire appel à la propriété Priority de la classe Thread
- Exemple :

```
Thread t1=new Thread(afficher);  
Thread t2 = new Thread(afficher);  
Thread t3 = new Thread(afficher);  
t1.Priority = ThreadPriority.Normal;  
t2.Priority = ThreadPriority.Lowest;  
t3.Priority = ThreadPriority.Highest;  
t1.Start(); t2.Start(); t3.Start();
```

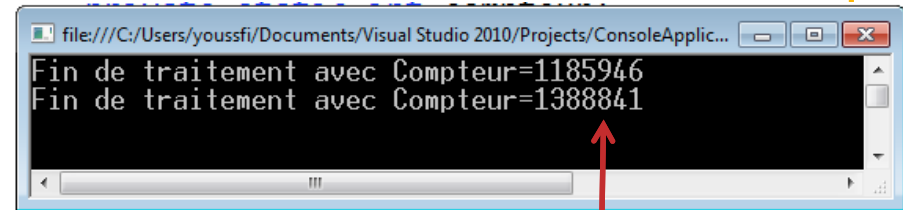


# Synchronisation des threads

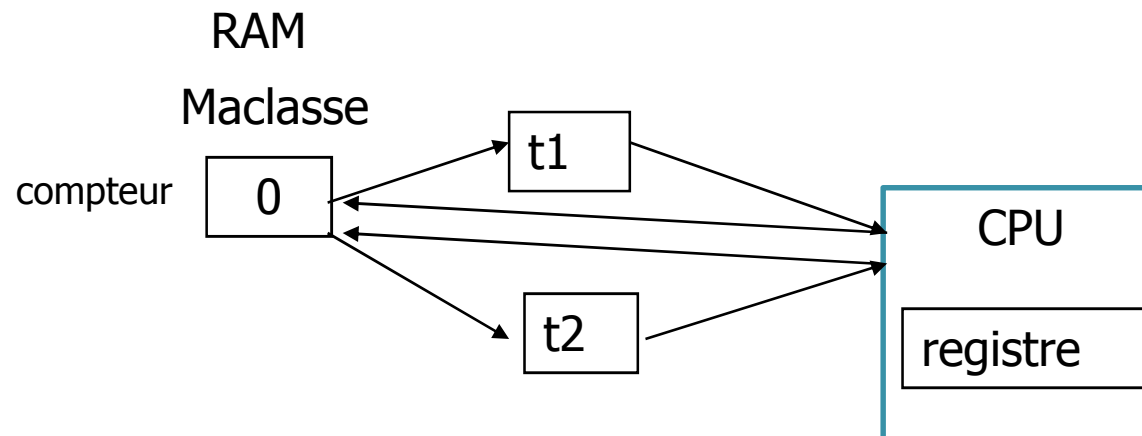
- Même si un thread s'exécute en deçà de votre programme principal, il reste que la méthode qu'il exécute fait partie de la classe à laquelle la méthode appartient.
- Cela signifie que l'accès aux variables globales et membres de votre classe lui seront accessibles sans problème.
- Là où le problème se pose, c'est lorsque plusieurs threads devront accéder à la même variable en mode modification.

# Synchronisation de threads et accès aux ressources partagées.

```
class Program {  
    private static int compteur;  
    static void Main(string[] args) {  
        Thread t1=new Thread(afficher);  
        Thread t2 = new Thread(afficher);  
        t1.Start(); t2.Start();  
        Console.ReadLine();  
    }  
    public static void afficher() {  
        for (int i = 0; i < 1000000; i++) { ++compteur; }  
        Console.WriteLine("Fin de traitement avec Compteur="+compteur);  
    }  
}
```



Résultat Anormal

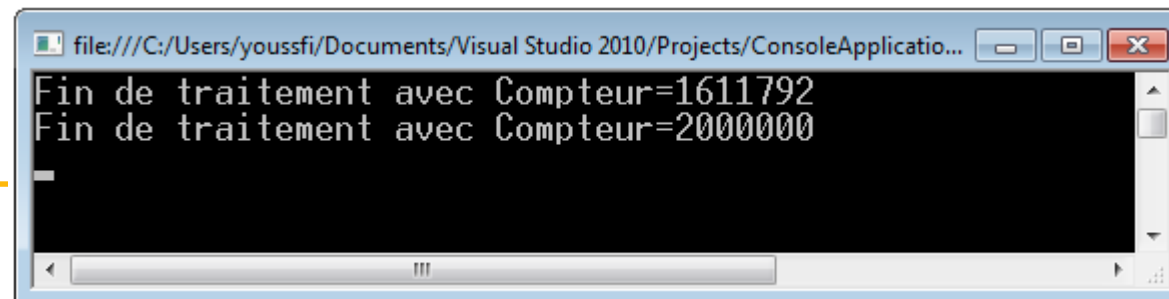


# Notions de verrous

- L'environnement C# offre un premier mécanisme de synchronisation : les verrous (locks en anglais).
- Chaque objet possède un verrou et seul un thread à la fois peut verrouiller un objet.
- Si d'autres threads cherchent à verrouiller le même objet, ils seront endormis jusqu'à que l'objet soit déverrouillé.
- Cela permet de mettre en place ce que l'on appelle plus communément une **section critique**.
- Pour verrouiller un objet par un thread, on peut utiliser le mot clé **lock**.

# Synchronisation avec l'instruction **lock**

```
class Program {  
    private static int compteur;  
    static object verrou = new object();  
    static void Main(string[] args) {  
        Thread t1=new Thread(afficher);  
        Thread t2 = new Thread(afficher);  
        t1.Start(); t2.Start();  
        Console.ReadLine();  
    }  
    public static void afficher(){  
        for (int i = 0; i < 1000000; i++){  
            lock (verrou){++compteur;}  
        }  
        Console.WriteLine("Fin de traitement avec Compteur="+compteur);  
    }  
}
```



The screenshot shows a console window titled "file:///C:/Users/youssfi/Documents/Visual Studio 2010/Projects/ConsoleApplication...". The output of the program is displayed as two lines of text: "Fin de traitement avec Compteur=1611792" and "Fin de traitement avec Compteur=2000000". The window has a standard Windows interface with a title bar, minimize, maximize, and close buttons, and a scrollbar on the right side.




# Synchronisation avec SemaphoreSlim

- Le SemaphoreSlim sert à contrôler l'accès d'une ressource limitée.
- Jusqu'à maintenant, les mécanismes de synchronisation dont nous avons parlé ont surtout servi à limiter une ressource à un accès mutuellement exclusif entre des threads concurrents.
- Qu'en est-il si l'on veut partager une ressource à travers plusieurs threads simultanément tout en gardant un nombre maximal d'accès concurrent ?
- Les sémaphores existent pour cette raison.
- En C# .NET, il existe deux types de sémaphores. Le classique Semaphore et le SemaphoreSlim.

# Exemple de synchronisation avec SemaphoreSlim

```
class Program {  
    private static object verrou = new object();  
    private static SemaphoreSlim salleAttente = new SemaphoreSlim(3);  
    static void Main(string[] args) {  
        Thread[] threads=new Thread[10];  
        for (int i = 0; i < 10; i++)    {  
            threads[i] = new Thread(afficher); threads[i].Name = "T" + i;  
            threads[i].Start();  
        }  
    }  
    public static void afficher(){
```

*Seuls 3 Threads sont autorisés à travailler en même temps dans cette zone critique*



```
        salleAttente.Wait();
```

```
        for (int i = 0; i < 10; i++){  
            Console.Write(Thread.CurrentThread.Name + "-");  
            Thread.Sleep(1000);  
        }  
        Console.WriteLine("Fin Du thread " + Thread.CurrentThread.Name);
```

```
        salleAttente.Release();
```

```
    }
```

```
}
```

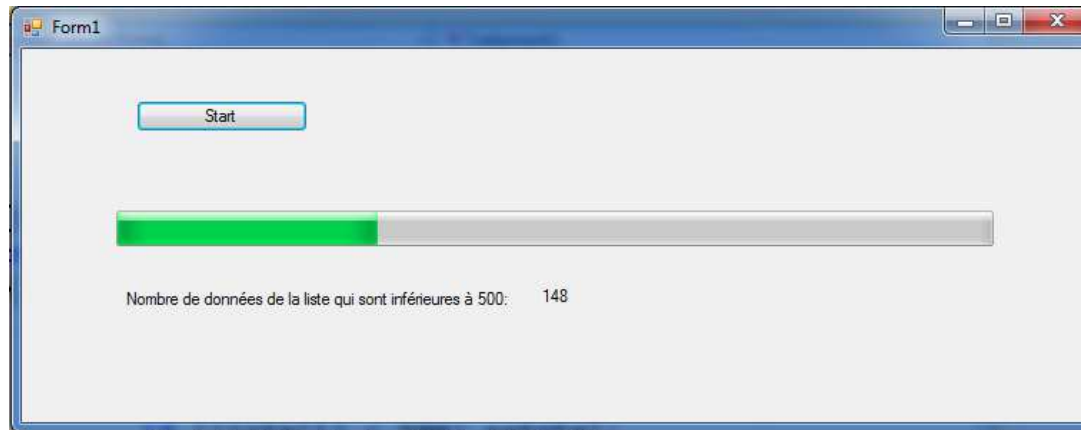
# Exemple de synchronisation avec lock

```
class Program {  
    private static object verrou = new object();  
    private static SemaphoreSlim salleAttente = new SemaphoreSlim(3);  
    static void Main(string[] args) {  
        Thread[] threads=new Thread[10];  
        for (int i = 0; i < 10; i++)    {  
            threads[i] = new Thread(afficher); threads[i].Name = "T" + i;  
            threads[i].Start();  
        }  
    }  
    public static void afficher(){  
        lock(verrou){  
            for (int i = 0; i < 10; i++){  
                Console.Write(Thread.CurrentThread.Name + "-");  
                Thread.Sleep(1000);  
            }  
            Console.WriteLine("Fin Du thread " + Thread.CurrentThread.Name);  
        }  
    }  
}
```

*Un seul Thread est autorisé à travailler en même temps dans cette zone critique*



# Exemple d'utilisation des threads dans une application WindowsForm



```
using System;using System.Collections.Generic; using System.ComponentModel;using System.Data;
using System.Drawing;using System.Linq;using System.Text;using System.Windows.Forms;
using System.Threading;
namespace WindowsFormsApplication1 {
    public partial class Form1 : Form    {
        private Random random = new Random((int)DateTime.Now.Ticks);
        private int[] liste = new int[50000];    private bool termine = true;
        private delegate void ProgressDelegate(int v);
        public Form1() {
            InitializeComponent();
            for (int i = 0; i < liste.Length; i++) {
                liste[i] = random.Next(50000);
            }
        }
    }
}
```

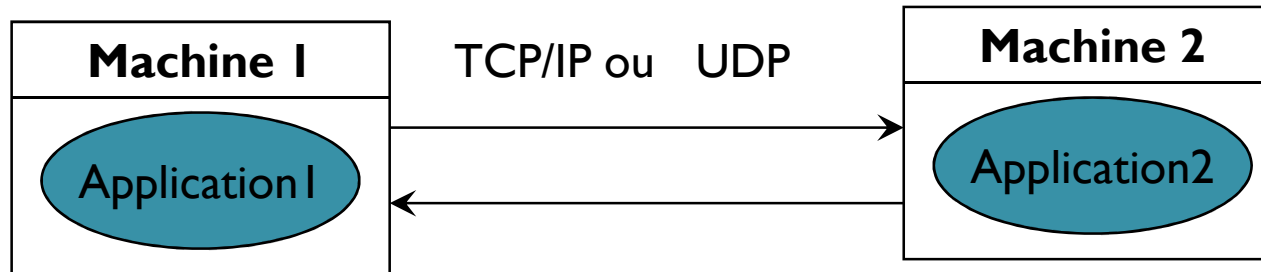
# Exemple d'utilisation des threads dans une application WindowsForm

```
private void buttonStart_Click(object sender, EventArgs e) {  
    if (termine) {  
        Thread t = new Thread(Traitement);    termine = false;    t.Start();  
    }  
}  
  
public void Traitement() {  
    int total = 0;  
    for (int i = 0; i < liste.Length; i++) {  
        if (liste[i] < 500) ++total;  
        int v = (int)(i / (double)liste.Length*100);  
        //progressBar1.Value=v; // Interdit  
        Invoke(new ProgressDelegate(Progress), v);  
        Invoke(new Action<int>(Resultat), total);  
    }  
}  
  
public void Progress(int v) {    progressBar1.Value = v;    }  
public void Resultat(int v) {    res.Text = v.ToString();    }  
}  
}
```



# **PROGRAMMATION RESEAUX : SOCKETS ET DATAGRAM**

# Applications distribuées



Technologies d'accès :

- Sockets ou DataGram



# Protocole TCP/IP

- Le protocole TCP fonctionne en mode connecté
- Lorsqu'une machine A envoie des données vers une machine B, la machine B est prévenue de l'arrivée des données, et témoigne de la bonne réception de ces données par un accusé de réception.
- Ici, intervient le contrôle CRC des données. Celui-ci repose sur une équation mathématique, permettant de vérifier l'intégrité des données transmises.
- Ainsi, si les données reçues sont corrompues, le protocole TCP permet aux destinataires de demander à l'émetteur de renvoyer les données corrompues.
- C'est un protocole fiable.
- Pour créer des applications réseaux, le protocole TCP offre l'API Socket.



# Sockets

- Les sockets sont une API de communication basée sur TCP/IP qui a été développée pour le langage C en 1983.
- Depuis tout ce temps, les sockets sont portés d'une plateforme et d'un langage à l'autre et ce n'est pas étonnant qu'ils existent encore en C#.
- En fait, un socket est totalement indépendant du langage.
- Un programme qui joue le rôle de serveur peut communiquer avec un autre programme jouant le rôle de client si ceux-ci utilisent des sockets, peu importe les langages impliqués.

# Sockets :

- L'espace de nom **System.Net.Sockets** contient tout ce qu'il faut pour manipuler des sockets.
- La classe **Socket** (de ce même espace de nom) est évidemment la principale.
- Un socket doit d'abord être créé pour pouvoir être utilisé.
- Les 3 paramètres que nous devons fournir à son constructeur sont les suivants:
  - Une famille d'adresses (qui représente en fait un type d'adressage),
  - Un type de socket (qui détermine la façon qu'il utilisera pour communiquer)
  - et un type de protocole.
- Il existe justement un enum pour chacun de ces paramètres; il nous suffira simplement de sélectionner les bonnes valeurs dans chacun d'entre eux :

```
Socket sock = new Socket(AddressFamily.InterNetwork,  
SocketType.Stream, ProtocolType.Tcp);
```

# Sockets :

- `Address.Family.InterNetwork` correspond à IPv4. Notez qu'il existe un `InterNetworkV6` et tout un tas d'autres modes d'adressage.
- `SocketType.Stream` permet des communications I à I et exige qu'une connexion soit établie avant que la communication puisse débuter. La communication pourra alors se faire dans les deux sens, en utilisant un flot d'octets. C'est la méthode à utiliser en TCP.
- `ProtocolType.Tcp` correspond au protocole TCP
- Dans une communication client/serveur, chacune des deux parties devra créer un socket.
- Les deux sockets formeront ultimement les deux bouts d'un canal de communication.

# Serveur

- Du côté du serveur, le socket, une fois créé, doit être attaché à un point de communication (*endpoint*).
- Un point de communication correspond à une adresse et un port TCP sur lequel écouter.
- Il existe un type `EndPoint` défini pour cet usage.
- On peut le créer en lui passant une adresse et un port au constructeur (l'adresse sera une instance de `IPAddress`, et le port sera simplement un entier).
- On utilisera généralement l'adresse IP du serveur
- La méthode statique `Parse` de `IPAddress` qui accepte un string et qui retourne un objet `IPAddress` si elle est valide (sinon, une exception de type `FormatException` sera lancée).

```
Socket sock = new Socket( AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
EndPoint iep = new EndPoint(IPAddress.Parse("192.168.1.2"),234);
sock.Bind(iep);
```

# Serveur

- Une fois le socket attaché à un point de communication, on le place en mode "écoute":

```
sock.Listen(1);
```

- Le nombre passé en paramètre représente le nombre de connexions simultanées maximal.
- Lorsqu'un client tentera de se connecter, si le nombre maximal est déjà atteint, le client recevra une exception de type `SocketException` (le champ `ErrorCode` de l'exception donnera les détails appropriés)
- Lorsque le socket est en mode "écoute", il pourra recevoir autant de demandes que l'indiquera le paramètre.
- Ces demandes seront placées en file et devront être traitées en les acceptant de façon synchrone ou asynchrone. La façon synchrone est la plus simple:

```
Socket clientSocket = sock.Accept();
```

- Remarquez que la méthode `Accept()` retourne un socket "connecté". C'est dans celui-là qui va permettre au serveur de communiquer avec le client

# Le client

- Du côté du client, c'est encore plus simple. On doit créer un socket de la même façon que pour le serveur. Le socket tentera ensuite une connexion au point de communication correspondant au serveur:

```
IPEndPoint iep = new IPEndPoint(IPAddress.Parse("192.168.1.2"),234);  
sock.Connect(iep);
```

- Si la connexion n'a pas pu être établie, une SocketException sera levée.

# Pour le client et le serveur :

- On peut écrire dans le socket en envoyant un tableau d'octets (byte[]) dans le flot. Par exemple:

```
byte[] buffer = new byte[512];  
sock.Send(buffer, 0, buffer.length, SocketFlags.None);
```

- Il s'agit de la surcharge la plus complexe de Send:
- on envoie les données contenues dans le buffer, en partant de 0 et en lisant toute la longueur (on pourrait spécifier des bornes différentes),
- puis on définit des options pour l'envoi en donnant une valeur de l'enum SocketFlags (ici, None, donc rien de spécial).
- Une combinaison d'options peut être faite en combinant les valeurs au niveau binaire...
- Si on veut simplifier, on peut enlever un ou plusieurs de ces paramètres additionnels et ne passer que le buffer, ce qui reviendrait au même dans notre exemple).

# Echanger des chaînes de caractères

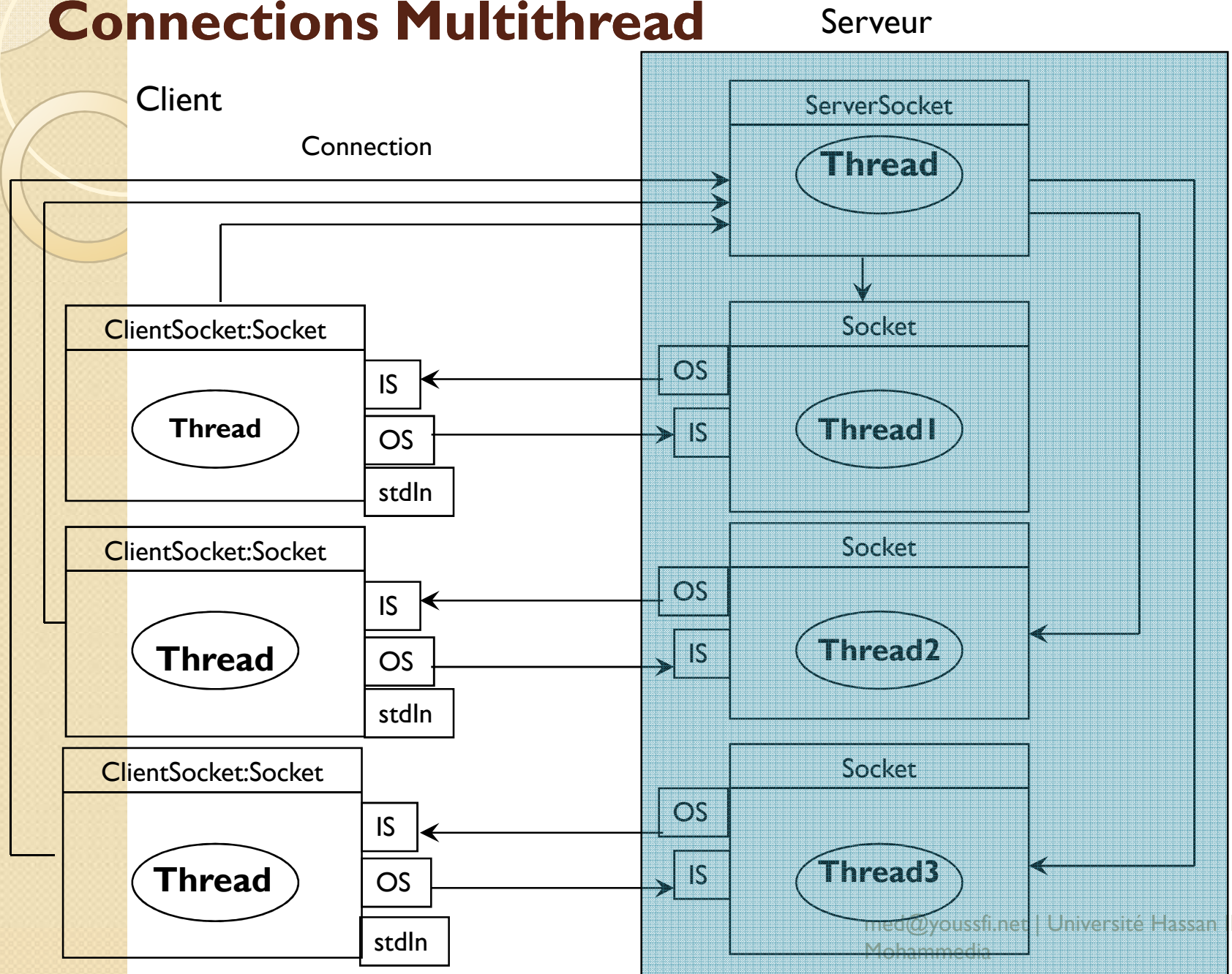
- On peut convertir un `string` en `byte[]` en utilisant la fonction `System.Text.Encoding.ASCII.GetBytes()`.
- L'inverse peut être fait grâce à sa fonction miroir `System.Text.ASCIIEncoding.ASCII.GetString()`.



# Serveur Multithreads

- Pour qu'un serveur puisse communiquer avec plusieurs client en même temps, il faut que:
  - Le serveur puisse attendre une connexion à tout moment.
  - Pour chaque connexion, il faut créer un nouveau thread associé à la socket du client connecté, puis attendre à nouveau une nouvelle connexion
  - Le thread créé doit s'occuper des opérations d'entrées-sorties (read/write) pour communiquer avec le client indépendamment des autres activités du serveur.

# Connections Multithread



# Exemple de serveur multi threads

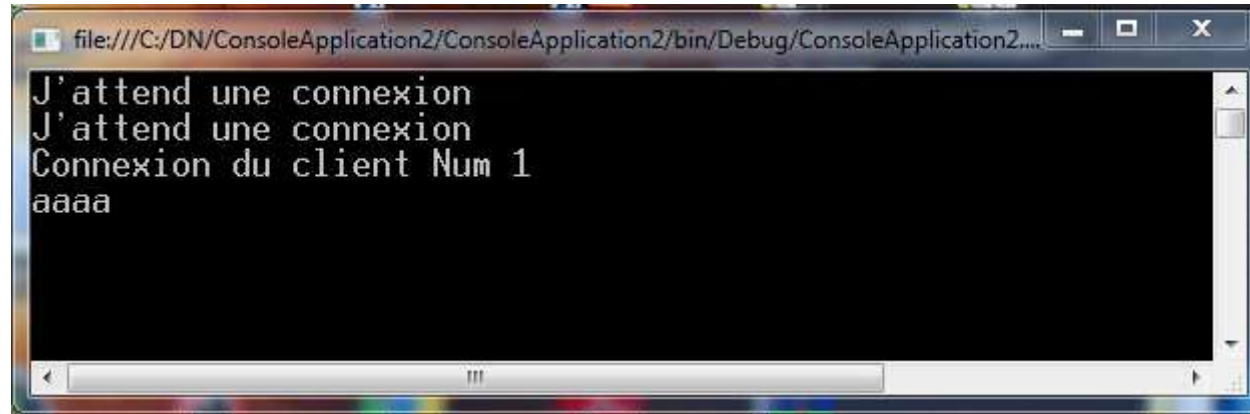
```
using System; using System.Collections.Generic;
using System.Linq; using System.Text; using System.Threading;
using System.Net.Sockets; using System.Net; using System.IO;
namespace ConsoleApplication2 {
    class Program {
        private int nbClients;
        public Program() {
            Thread th = new Thread(Ecoute);
            th.Start();
        }
        static void Main(string[] args) {
            new Program();
        }
        public void Ecoute(){
            Socket ss = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
            IPEndPoint iep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1234);
            ss.Bind(iep); ss.Listen(1);
            while (true) {
                Console.WriteLine("J'attend une connexion");
                Socket s = ss.Accept(); ++nbClients;
                ClientCommunication cc = new ClientCommunication(s, nbClients);
                new Thread(Communication).Start(cc);
            }
        }
    }
}
```

# Exemple de serveur multi threads

```
class ClientCommunication{
    public Socket socket { get; set; }
    public int numero { get; set; }
    public ClientCommunication(Socket s, int num){
        this.socket = s; this.numero = num;
    }
}

public void Communication(object o) {
    ClientCommunication cc = o as ClientCommunication;
    Socket s = cc.socket;
    NetworkStream nws = new NetworkStream(s);
    TextReader tr = new StreamReader(nws,Encoding.ASCII);
    TextWriter tw = new StreamWriter(nws,Encoding.ASCII);
    tw.WriteLine("Vous êtes le client num " + cc.numero);
    tw.Flush();
    Console.WriteLine("Connexion du client Num " + cc.numero);
    while (true)
    {
        string req = tr.ReadLine();
        Console.WriteLine(req);
        tw.WriteLine("Response To : " + req);
        tw.Flush();
    }
}
}
```

# Tester le serveur avec un client telnet




```
file:///C:/DN/ConsoleApplication2/ConsoleApplication2/bin/Debug/ConsoleApplication2....  
J'attends une connexion  
J'attends une connexion  
Connexion du client Num 1  
aaaa
```

Serveur



```
C:\Windows\system32\cmd.exe  
C:\Users\youssfi>telnet 127.0.0.1 1234
```

Client  
Telnet



```
C:\ Telnet 127.0.0.1  
Vous ?tes le client num 1  
aaaa  
Response To : aaaa
```

Client  
Telnet

# Client Console C#

```
using System; using System.Collections.Generic; using System.Linq; using System.Text;
using System.Threading; using System.IO; using System.Net; using System.Net.Sockets;
namespace ClientTelnet {
    class Client{
        private TextReader tr;    private TextWriter tw;
        public Client() {
            Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
            IPEndPoint ipep = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1234);
            s.Connect(ipep);
            NetworkStream nws = new NetworkStream(s);
            tr = new StreamReader(nws, Encoding.ASCII);
            tw = new StreamWriter(nws, Encoding.ASCII);
            Thread t = new Thread(EcouteReponses);
            t.Start();
            while (true)    {
                string req = Console.ReadLine();
                tw.WriteLine(req);
                tw.Flush();
            }
        }
    }
}
```

# Client Console C#

```
public void EcouteReponses()  
{  
    while (true)  
    {  
        string rep = tr.ReadLine();  
        Console.WriteLine(rep);  
    }  
}  
static void Main(string[] args)  
{  
    new Client();  
}  
}
```



```
file:///C:/DN/ClientTelnet/ClientTelnet/bin/Debug/ClientTelnet.EXE  
Vous ?tes le client num 1  
aaaa  
Response To : aaaa  
bbbbbb  
Response To : bbbbbb
```

# Client Java

```
import java.io.BufferedReader; import java.io.InputStreamReader; import
    java.io.PrintWriter;

import java.net.Socket; import java.util.Scanner;

public class ClientJava extends Thread {
    private BufferedReader br;          private PrintWriter pw;
    public ClientJava() {
    try {
        Socket s=new Socket("127.0.0.1",1234);
        br=new BufferedReader(new InputStreamReader(s.getInputStream()));
        pw=new PrintWriter(s.getOutputStream(),true);
        Scanner clavier=new Scanner(System.in);
        this.start();
        while(true){
            String cmd=clavier.nextLine();
            pw.println(cmd);
        }
    } catch (Exception e) { e.printStackTrace(); }
    }
```



# Client Java

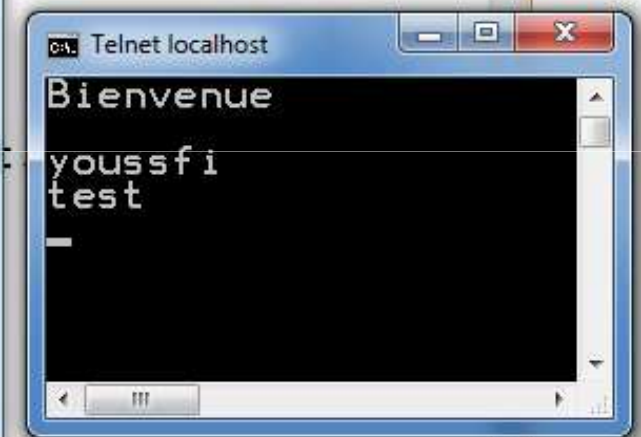
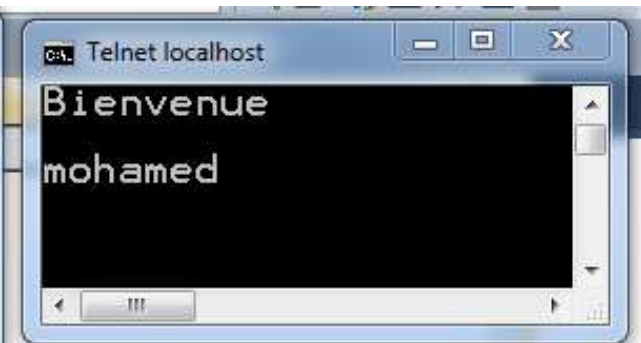
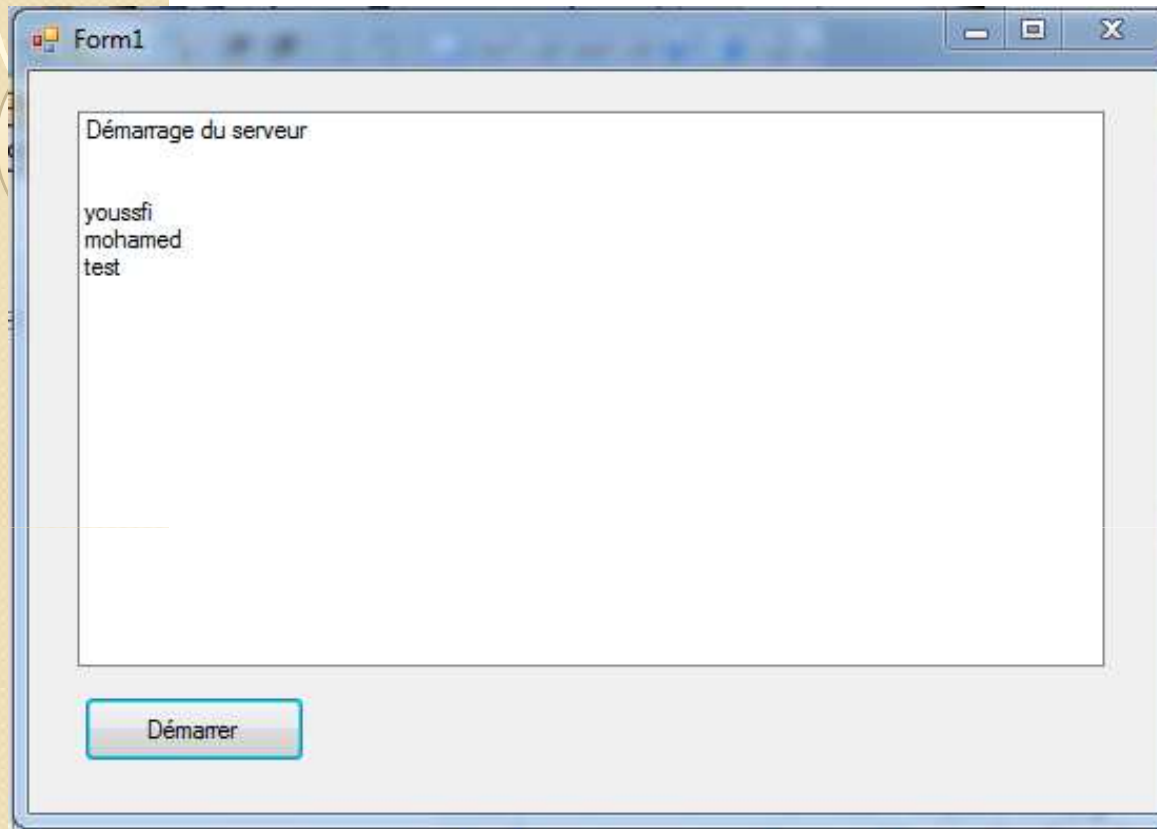
```
@Override
public void run() {
    while(true){
        try {
            String rep=br.readLine();
            System.out.println(rep);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public static void main(String[] args) {
    new ClientJava();
}
```

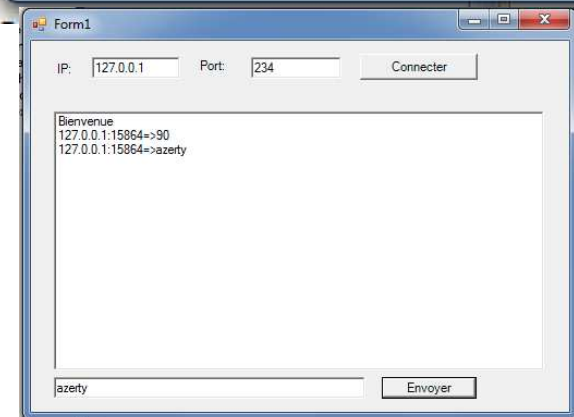
# Exemple de serveur multithreads

Serveur

Clients Telnet



Clients Graphique



# Exemple de serveur :

```
using System;using System.Collections.Generic;
using System.ComponentModel;using System.Data;
using System.Drawing;using System.Linq;
using System.Text;using System.Windows.Forms;
using System.Threading;using System.Net;
using System.Net.Sockets;
namespace SocketServer
{
    public partial class Form1 : Form
    {
        private Socket serverSocket;
        private Thread serverThread;
        private List<Socket> connexions = new List<Socket>();
        bool ecoute=true;
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

# Exemple de serveur :

```
private void buttonDemarrer_Click(object sender, EventArgs e){
    serverThread = new Thread(ServiceEcoule);
    serverThread.Start();
}

public void ServiceEcoule(){
serverSocket = new
    Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 234);
    serverSocket.Bind(ipe);
    serverSocket.Listen(1);
    Invoke(new Action<string>(Log), "Démarrage du serveur");
    while(ecoule){
        Socket clientSocket=serverSocket.Accept();
        connexions.Add(clientSocket);
        new Thread(Communication).Start(clientSocket);
    }
}
```

# Exemple de serveur :

```
public void Communication(object o)      {
    Socket clientSocket=o as Socket;
    string msg="Bienvenue";
    clientSocket.Send(Encoding.Default.GetBytes(msg));
    string s="";
    while (true)
    {
        byte[] data =new byte[1];
        clientSocket.Receive(data);
        string req = Encoding.Default.GetString(data);
        if (data[0] != (byte)'\n')
            s += req;
        else
        {
            Invoke(new Action<string>(Log), s);
            Braodcast(clientSocket,s);
            s = "";
        }
    }
}
```

# Exemple de serveur :

```
public void Log(string msg)
{
    listBoxMessages.Items.Add(msg);
}

public void Braodcast(Socket s,string msg){
    string rep = string.Format("{0}=>{1}\n",
        s.RemoteEndPoint.ToString(), msg);
    foreach (Socket sock in connexions) {
        sock.Send(Encoding.ASCII.GetBytes(rep));
    }
}
}
```

# Client Console

```
using System; using System.Collections.Generic;
using System.Linq; using System.Text; using System.Net;
using System.Net.Sockets; using System.Threading;
namespace TelNet {
    class Program {
        static Socket sock;
        static void Main(string[] args) {
            sock = new
Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
            IPEndPoint iep=new IPEndPoint(IPAddress.Parse("127.0.0.1"),234);
            sock.Connect(iep);
            // sock.Send(Encoding.ASCII.GetBytes("slauttttt\n"));
            Thread th = new Thread(Ecouteur);
            th.Start();
            new Thread(Saisie).Start();
        }
    }
}
```

# Client Console

```
public static void Saisie() {  
    while (true) {  
        string req = Console.ReadLine();  
        sock.Send(Encoding.ASCII.GetBytes(req + "\n"));  
    }  
}  
  
public static void Ecouleur(){  
    while (true)  
    {  
        byte[] data = new byte[256];  
        sock.Receive(data);  
        string rep = Encoding.ASCII.GetString(data);  
        Console.WriteLine(rep);  
    }  
}  
}
```





# Protocole UDP

- Le protocole UDP, soit *User Datagram Protocol*, est le *plus simple* des deux protocoles.
- On peut le comparer à un service d'envoi de lettres, car il s'agit d'un protocole dit *sans connexion*.
- En UDP, il n'y a aucune garantie que les messages arriveront à destination dans l'ordre.
- En fait, il n'y a même pas de garantie que les messages arriveront du court.
- On assume que la couche Application se chargera de vérifier l'intégrité des données.
- Cependant, ce manque de rigueur s'accompagne de bien meilleures performances. Vous l'aurez deviné,
- on utilise l'UDP dans les systèmes de communication où la performance est critique.
- Par exemple, le logiciel de téléphonie Skype. Peu importe si quelques paquets manquent, l'essentiel est que les paquets se rendent le plus rapidement possible afin de constituer une conversation fluide.



# Protocole UDP

- ***Inconvénients***

- Aucune rigueur dans l'ordre d'arrivée des paquets.
- Aucune vérification quant à l'arrivée à destination des paquets.
- Protocole non-fiable pour ce qui est de la qualité des communications.

- ***Avantages***

- Communication sans connexion (plus facile).
- Parfait pour les services où la vitesse est critique.
- Possibilité de communiquer à plusieurs clients à la fois à l'aide du **broadcasting et du multicasting**.
- Permet l'envoi des données en **broadcast très facilement (à l'aide de l'adresse réservée à cette fin)**.

- ***Utilisations***

- Téléphonie IP, Streaming Vidéo
- Protocole DNS (résolution de noms de domaine).
- Protocole DHCP (assignation d'adresse IP de manière dynamique).

# Exemple de Serveur UDP

```
using System; using System.Collections.Generic; using System.Linq;
using System.Text; using System.Net; using System.Net.Sockets; using System.Threading;
namespace serveur {
    class ServeurUDP {
        private static Thread thread;
        static void Main(string[] args) {
            thread = new Thread(new ThreadStart(Ecoute));    thread.Start();
            Console.ReadLine();
        }
        public static void Ecoute() {
            Console.WriteLine("Préparation de l'écoute");
            UdpClient serveur = new UdpClient(1234);
            while (true) {
                IPEndPoint client=null;
                Console.WriteLine("Ecoute");
                byte[] data = serveur.Receive(ref client);
                Console.WriteLine("Réception des données en provenance de {0}:{1}",
client.Address, client.Port);
                string message = Encoding.Default.GetString(data);
                Console.WriteLine("Contenu du message:"+message);
            } } } }
```

# Exemple de Serveur UDP

```
using System; using System.Collections.Generic; using System.Linq;
using System.Text; using System.Net; using System.Net.Sockets; using System.Threading;
namespace serveur {
    class ServeurUDP {
        private static Thread thread;
        static void Main(string[] args) {
            thread = new Thread(new ThreadStart(Ecoute));    thread.Start();
            Console.ReadLine();
        }
        public static void Ecoute() {
            Console.WriteLine("Préparation de l'écoute");
            UdpClient serveur = new UdpClient(1234);
            while (true) {
                IPEndPoint client=null;
                Console.WriteLine("Ecoute");
                byte[] data = serveur.Receive(ref client);
                Console.WriteLine("Réception des données en provenance de {0}:{1}",
client.Address, client.Port);
                string message = Encoding.Default.GetString(data);
                Console.WriteLine("Contenu du message:"+message);
            } } } }
```

# Exécution de l'application client serveur

```
C:\Windows\system32\cmd.exe - TPUDP.exe

C:\DN\TPUDP\TPUDP\TPUDP\bin\Debug>TPUDP.exe
Préparation de l'écote
Ecoule
Réception des données en provenance de 127.0.0.1:61208
Contenu du message:azerty
Ecoule
Réception des données en provenance de 127.0.0.1:61211
Contenu du message:2345
Ecoule
```

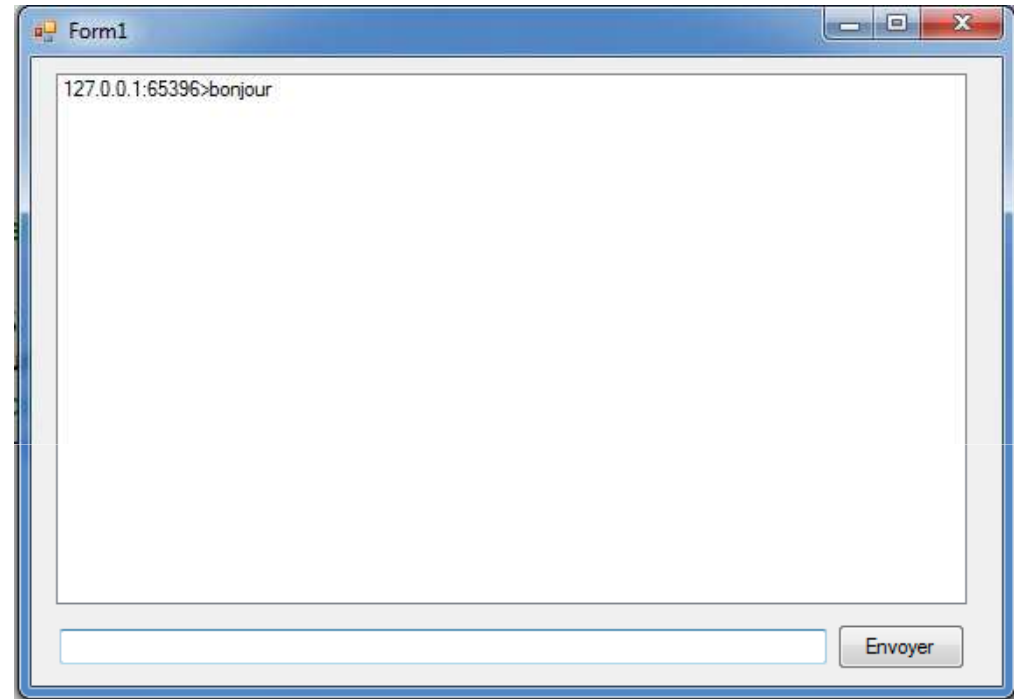
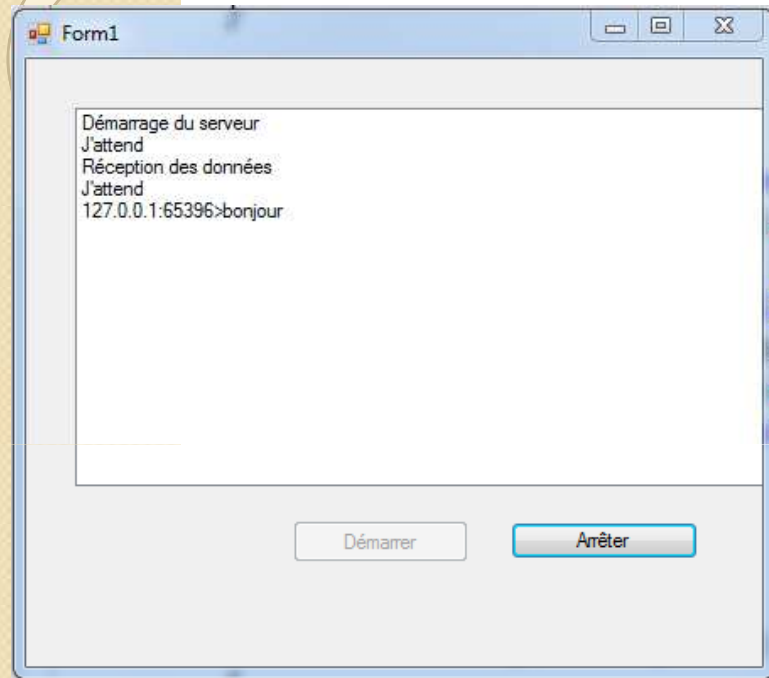
```
C:\Windows\system32\cmd.exe - ClientUDP.exe

C:\DN\CLIENTUDP\ClientUDP\ClientUDP\bin\Debug>ClientUDP.exe
Entrez un message : azerty
Continuer ? (O/N)
```

```
C:\Windows\system32\cmd.exe

C:\DN\CLIENTUDP\ClientUDP\ClientUDP\bin\Debug>ClientUDP.exe
Entrez un message : 2345
Continuer ? (O/N)
C:\DN\CLIENTUDP\ClientUDP\ClientUDP\bin\Debug>
```

# Application : Un chat classique



# Serveur

- Variables de Globales :
  - Contrôle de la boucle d'écoute
    - `bool ecoute = true;`
  - Le thread qui gère le service d'écoute du serveur.
    - `Thread serverThread;`
  - Le Datagram qui permet au serveur de broadcaster un message aux clients
    - `UdpClient broadCatser;`
  - Le Datagram qui permet de créer le service d'écoute du serveur.
    - `UdpClient serveur;`

# Serveur

- Initialisation :

```
public Form1()
{
    InitializeComponent();
    broadCatser = new UdpClient();
    broadCatser.EnableBroadcast = true;
    broadCatser.Connect(new IPEndPoint(IPAddress.Broadcast,6666));
}
```

- Démarrage du serveur :

```
private void buttonDemarrer_Click(object sender, EventArgs e) {
    buttonDemarrer.Enabled = false;
    buttonArreter.Enabled = true;
    serverThread=new Thread(new ThreadStart(EcouteurReseau));
    serverThread.Start();
}
```



## Serveur

- Méthode d'écoute réseau:

```
private void EcouteurReseau(){
    try {
        serveur = new UdpClient(1234);
        Invoke(new Action<string>(log), "Démarrage du serveur");
        while (ecoute){
            IPEndPoint ip=null;
            Invoke(new Action<string>(log), "J'attend");
            byte[] data = serveur.Receive(ref ip);
            Invoke(new Action<string>(log), "Réception des données");
            CommunicationData cd = new CommunicationData(ip,data);
            new Thread(TraiterMessage).Start(cd);
        }
        catch (Exception e){Invoke(new Action<string>(log),e.Message); }
    }
}
```

- La classe Interne : CommunicationData

```
private class CommunicationData {
    public IPEndPoint client{get;set;}
    public byte[] data { get; set; }
    public CommunicationData(IPEndPoint ip,byte[] data){
        this.client = ip;
        this.data = data;
    }
}
```

## Serveur

- Méthode de traitement de la requête du client :

```
public void TraiterMessage(object arg){  
    CommunicationData cd = arg as CommunicationData;  
    string message = string.Format("{0}:{1}>{2}"  
        , cd.client.Address.ToString(), cd.client.Port,  
        Encoding.Default.GetString(cd.data));  
    byte[] donnees = Encoding.Default.GetBytes(message);  
    broadCatser.Send(donnees, donnees.Length);  
    Invoke(new Action<string>(log),message);  
}
```

- La méthode qui permet de logger :

```
private void log(string msg) {  
    listBoxLog.Items.Add(msg);  
}
```

- Arrêt du serveur :

```
private void buttonArreter_Click(object sender, EventArgs e) {  
    ecoute = false;  
    serverThread.Abort();  
    Invoke(new Action<string>(log), "Arrêt");  
}
```

## Client

- Variables de Globales :
  - Contrôle de la boucle d'écoute
    - `bool ecoute = true;`
  - Le thread qui gère le service.
    - `Thread ecouteurThread;`
  - Le Datagram qui permet au client de se connecter au serveur
    - `UdpClient client;`

# Serveur

- Initialisation :

```
public Form1(){  
    InitializeComponent();  
    new Thread(Connector).Start();  
}
```

- La méthode Connceter:

```
public void Connector(){  
    client = new UdpClient();  
    client.Connect("127.0.0.1", 1234);  
    ecouteurThread = new Thread(Ecouteur);  
    ecouteurThread.Start();  
}
```

- Logger des messages dans l'objet ListBox :

```
public void Log(string str) {  
    listBoxMessages.Items.Add(str);  
}
```

# Serveur

- Méthode Ecouteur:

```
public void Ecouteur() {  
    UdpClient ecouteur = new UdpClient(6666);  
    while(ecoute){  
        IPEndPoint ip = null;  
        byte[] data = ecouteur.Receive(ref ip);  
        string message = Encoding.Default.GetString(data);  
        Invoke(new Action<string>(Log), message);  
    }  
    ecouteur.Close();  
}
```

- Envoyer le message au serveur:

```
private void buttonEnvoyer_Click(object sender, EventArgs e) {  
    byte[] data = Encoding.Default.GetBytes(textBoxMessage.Text);  
    client.Send(data,data.Length);  
    textBoxMessage.Clear();  
    textBoxMessage.Focus ();  
}
```



# Programmation C#

## Accès aux bases de données ADO.net

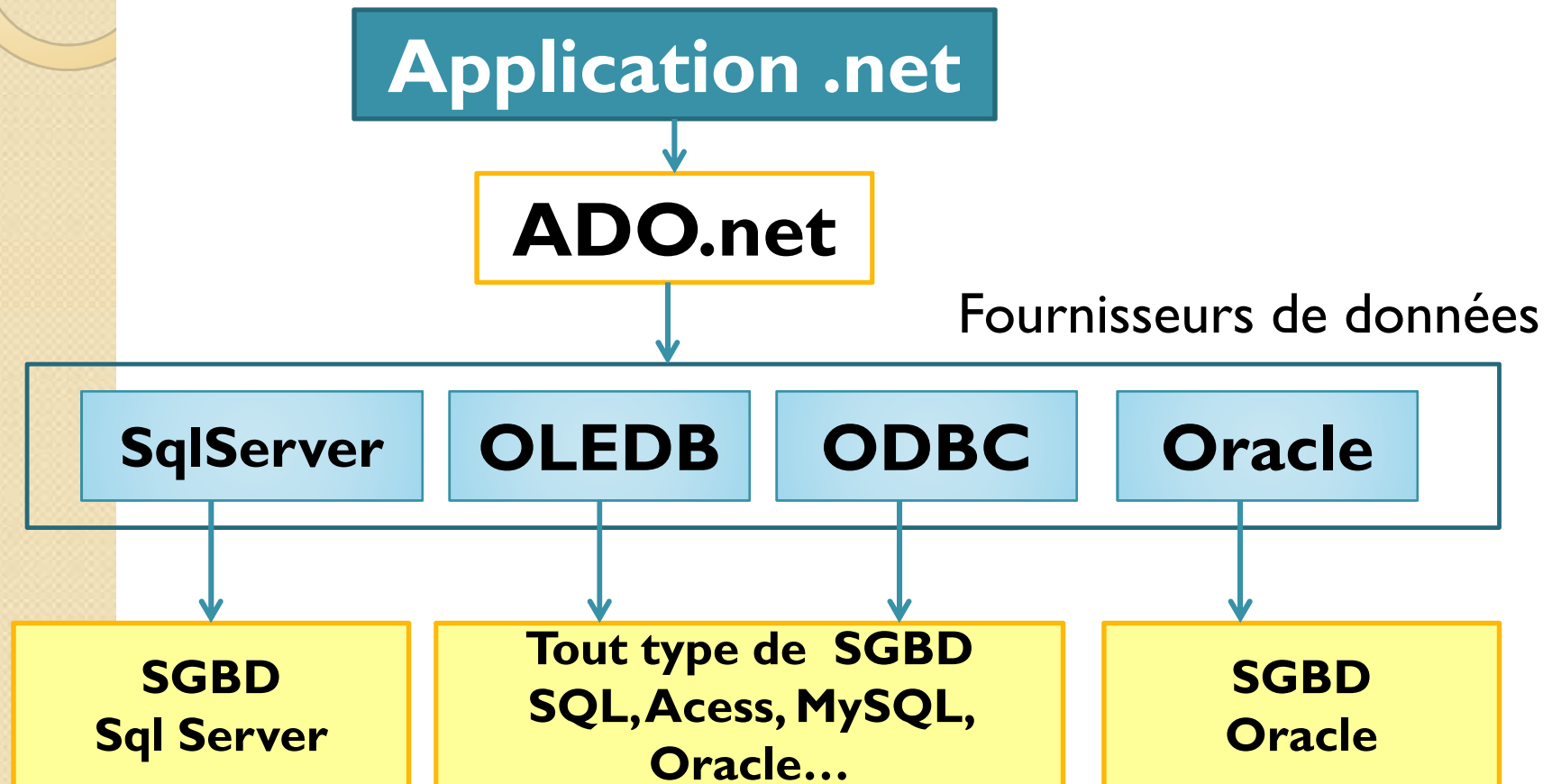
Youssfi Mohamed

ENSET

Université Hassan II Mohammedia

[med@youssfi.net](mailto:med@youssfi.net)

# Fournisseurs de Données





# Fournisseurs de données

- Chaque fournisseur de données permet la communication avec un type de base de données au travers d'une API.
- Une API (*Application Programming Interface*) est l'interface qui permet l'accès de logiciel par un autre.
- Ces fournisseurs permettent de récupérer et de transférer des modifications entre l'application et une base de données.
- Toutes les classes permettant d'utiliser ces fournisseurs se trouvent dans l'espace de nom *System.Data*. Sur le Framework 3.5, il existe quatre types de fournisseurs :
  - Sql Server
  - OLE DB
  - ODBC
  - Oracle



# Fournisseurs de données

- **SqlServer :**
  - Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.SqlClient*, chaque nom de ces classes est *préfixé par Sql*. Ce pilote a accès au serveur sans utiliser d'autres couches logicielles le rendant plus performant.
- **OLE DB :**
  - Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.OleDb*, chaque nom de ces classes est *préfixé par OleDb*. Ce fournisseur exige l'installation de MDAC (Microsoft Data Access Components). L'avantage de ce fournisseur est qu'il peut dialoguer avec n'importe quelle base de données le temps que le pilote OLE DB est installé dessus, mais par rapport à SQL server, OLE DB utilise une couche logicielle nommée OLE DB ; il requiert donc plus de ressources diminuant par conséquent les performances.

# Fournisseurs de données

- ODBC :

- Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.Odbc*, chaque nom de ces classes est préfixé par *Odbc*. Tout comme l'OLE DB, ODBC exige l'installation de MDAC. Il fonctionne avec le même principe qu'OLE DB mais au lieu d'utiliser une couche logicielle, il utilise le pilote ODBC.

- Oracle :

- Les classes de ce fournisseur se trouvent dans l'espace de nom *System.Data.OracleClient*, chaque nom de ces classes est préfixé par *Oracle*. Il permet simplement de se connecter à une source de données Oracle.

# Accéder à la base de données

- Pour dialoguer avec la base de données, tous ces fournisseurs implémentent six classes de base :
  - **Command** :
    - Stocke les informations sur la commande SQL et permet son exécution sur le serveur de base de données.
  - **CommandBuilder**
    - Permet de générer automatiquement des commandes ainsi que des paramètres pour un *DataAdapter*.
  - **Connection** :
    - Permet d'établir une connexion à une source de données spécifiée.

# Accéder à la base de données

- DataAdapter :
  - Permet le transfert de données de la base de données vers l'application et inversement (par exemple pour une mise à jour, suppression ou modification de données). Il est utilisé en mode déconnecté
- DataReader :
  - Permet un accès en lecture seule à une source de données.
- Transaction :
  - Représente une transaction dans le serveur de la base de données.

Remarque : La classe *Connection* n'apparaît pas dans le Framework 3.5. En effet, les classes des fournisseurs managés ont leur propre classe tel que *SqlConnection*.

# Premier Exemple

```
// Chaîne de connection qui identifie la base de donnée SQL Server
String strConn="Data Source=localhost\\SQLEXPRESS;Initial Catalog= DB_CATALOGUE;Integrated Security=True";
// Créer une connexion vers la base de données
SqlConnection conn = new SqlConnection(strConn);
// Ouvrir la connexion
conn.Open();
// Créer L'objet Command pour une requête SQL paramétrée
SqlCommand cmd = new SqlCommand("select * from PRODUITS where DESIGNATION like @x", conn);
// Créer le paramètre x avec sa valeur
SqlParameter p=new SqlParameter("@x","%P%");
// Ajouter le paramètre à l'objet command
cmd.Parameters.Add(p);
// Executer la commande SQL et récupérer le résultat dans un objet DataReader
SqlDataReader dr = cmd.ExecuteReader();
// Parcourir le DataReader ligne par ligne
while(dr.Read()){
    // Afficher la valeur de la deuxième colonne
    Console.WriteLine(dr.GetString(1));
}
// Fermer les objets DataReader et Connection
dr.Close(); conn.Close();
```



# Mode Connecté et Mode déconnecté

- L'ADO.NET permet de séparer les actions d'accès ou de modification d'une base de données.
- En effet, il est possible de manipuler une base de données sans être connecté à celle-ci, il suffit juste de se connecter pendant un court laps de temps afin de faire une mise à jour.
- Ceci est possible grâce au *DataSet*.
- *C'est pourquoi, il existe deux types de fonctionnements :*
  - Le mode connecté
  - Le mode déconnecté



# Mode Connecté

- Avantages :
  - Avec un mode connecté, la connexion est permanente, par conséquent les données sont toujours à jour. De plus il est facile de voir quels sont les utilisateurs connectés et sur quoi ils travaillent. Enfin, la gestion est simple, il y a connexion au début de l'application puis déconnexion à la fin.
- Inconvénients :
  - L'inconvénient se trouve surtout au niveau des ressources. En effet, tous les utilisateurs ont une connexion permanente avec le serveur. Même si l'utilisateur n'y fait rien la connexion gaspille beaucoup de ressource entraînant aussi des problèmes d'accès au réseau.



# Mode Déconnecté

- Avantages :
  - L'avantage est qu'il est possible de brancher un nombre important d'utilisateurs sur le même serveur. En effet, ils se connectent le moins souvent et durant la plus courte durée possible. De plus, avec cet environnement déconnecté, l'application gagne en performance par la disponibilité des ressources pour les connexions.
- Inconvénients :
  - Les données ne sont pas toujours à jour, ce qui peut aussi entraîner des conflits lors des mises à jour. Il faut aussi penser à prévoir du code pour savoir ce que va faire l'utilisateur en cas de conflits.



# Etablir Une connexion

- Chaines de connexion :
  - Dans un mode connecté, il faut tout d'abord connecter l'application à la base de données. Nous utiliserons SQL Server 2005 pour la suite. Pour ouvrir cette connexion il faut d'abord déclarer une variable, ici ce sera « conn » :

```
// Chaine de connection qui identifie la base de donnée SQL Server
String strConn="Data Source=localhost\\SQLEXPRESS;Initial Catalog=
    DB_CATALOGUE;Integrated Security=True";
// Créer une connexion vers la base de données
SqlConnection conn = new SqlConnection(strConn);
// Ouvrir la connexion
conn.Open();

.....
// Fermer la Connection
conn.Close();
```

# Etablir Une connexion

- Quelques paramètres de la chaine de connexion :
  - Data Source : Indique le nom ou l'adresse réseau du serveur.
  - Initial Catalog : Indique le nom de la base de données où l'application doit se connecter.
  - Integrated Security : Indique s'il faut un nom et un mot de passe. Si la valeur est sur False, un login et password seront demandés.
  - Pwd : Indique le mot de passe associé au compte SQL Server.
  - User ID : Indique le nom du compte SQL Server.
  - .....



# Pools de Connexion

- Afin de réduire le coût en ressource engendré par les connexions à des bases de données, l'ADO.NET propose une technique d'optimisation : le pool de connexion.
- Lorsque qu'une application ouvre une nouvelle connexion, un pool est créé.
- Les pools permettent de stocker toutes les requêtes récurrentes.
- Chaque fois qu'un utilisateur ouvre une connexion avec la même `ConnectionString` qu'un pool, le dispositif de connexion vérifie s'il y a une place disponible dans ce pool, si le `MaxPoolSize` n'est pas atteint, la connexion rentre dans l'ensemble.
- Un pool est effacé lorsqu'une erreur critique est levée. Les pools sont paramétrables dans le `ConnectionString` et une connexion est retirée d'un pool lorsqu'elle est inactive depuis une certaine durée.

# Mode Connecté : Objet Command

- Objet Command :
  - Les requêtes SQL et les procédures stockées sont exécutées à partir de commandes.
  - Les commandes contiennent toutes les informations nécessaires à leur exécution et effectuent des opérations telles que créer, modifier ou encore supprimer des données d'une base de données.
  - Chaque fournisseur de base de données possède leurs propres objets *Command* qui sont les suivantes :
    - SqlCommand SQL Server
    - OleDbCommand OLE DB
    - OdbcCommand ODBC
    - OracleCommand Oracle

# Mode Connecté : Objet Command

- Objet Command (Suite):
  - Il existe plusieurs propriétés et méthodes communes à chaque fournisseur pour gérer des commandes, voici les principales :
    - CommandText : Permet de définir l'instruction de requêtes SQL ou de procédures stockées à exécuter. Lié à la propriété *CommandType*.
    - CommandTimeout : Permet d'indiquer le temps en secondes avant de mettre fin à l'exécution de la commande.
    - CommandType : Permet d'indiquer ou de spécifier la manière dont la propriété *CommandText* doit être exécutée.
    - Parameters : C'est la collection des paramètres de commandes. Lors de l'exécution de requêtes paramétrées ou de procédures stockées, vous devez ajouter les paramètres objet dans la collection.
    - Transaction : Permet de définir la *SqlTransaction* dans laquelle la *SqlCommand* s'exécute.
    - ExecuteReader : Permet d'exécuter des commandes et les retourne sous forme de tableau de données (ou des lignes).
    - ExecuteNonQuery : Permet d'exécuter des requêtes ou des procédures stockées qui ne retournent pas de valeurs.

# Mode Connecté : Objet Command

- Utilisation de l'objet Command :

```
// Créer L'objet Command pour une requête SQL paramétrée
SqlCommand cmd = new SqlCommand("select * from PRODUITS where
    NOM_PRODUIT like @x", conn);
// Créer le paramètre x avec sa valeur
SqlParameter p=new SqlParameter("@x","%P%");
// Ajouter le paramètre à l'objet command
cmd.Parameters.Add(p);
// Executer la commande SQL et récupérer le résultat dans un objet
// DataReader
SqlDataReader dr = cmd.ExecuteReader();
// Parcourir le DataReader ligne par ligne
while(dr.Read()){
// Afficher la valeur de la deuxième colonne
    Console.WriteLine(dr.GetString(1));
}
// Fermer les objets DataReader et Connection
dr.Close(); conn.Close();
```



# Mode Connecté : Objet DataReader

- Le DataReader permet un accès en lecture seule à des enregistrements, c'est-à-dire qu'il est impossible de revenir en arrière sur les enregistrements lus.
- Il n'a été créé que pour la lecture pure et simple de données.
- Le DataReader doit toujours être associé à une connexion active.
- Il existe plusieurs Datareader suivant le fournisseur utilisé, par exemple nous avons SqlDataReader ou encore OracleDataReader.
- Le DataReader comprend plusieurs méthodes : *GetBytes*, *GetChars* ou *GetString* qui permettent de récupérer différents types de données des colonnes d'une table.

# Application I

- Base de données SQL Server : DB\_CAT5
  - Table Categories

WIN-7PA448PU10... dbo.CATEGORIES			
	Nom de la colonne	Type de données	Autoriser l...
PK	ID_CAT	bigint	<input type="checkbox"/>
	NOM_CAT	nchar(25)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

- Table Produits

WIN-7PA448PU10... - dbo.PRODUITS			
	Nom de la colonne	Type de données	Autoriser l...
PK	ID_PROD	bigint	<input type="checkbox"/>
	DESIGNATION	nchar(25)	<input checked="" type="checkbox"/>
	PRIX	float	<input checked="" type="checkbox"/>
	QUANTITE	int	<input checked="" type="checkbox"/>
	ID_CAT	bigint	<input checked="" type="checkbox"/>
			<input type="checkbox"/>



# Exemple de consultation

```
using System; using System.Collections.Generic; using System.Linq;
using System.Text; using System.Data.SqlClient;
namespace ConsoleApplication7 {
    class Program {
        static void Main(string[] args) {
            String strConn = "Data Source=localhost\\SQLEXPRESS;Initial Catalog=
DB_CAT5;Integrated Security=True";
            SqlConnection conn = new SqlConnection(strConn);
            conn.Open();
            SqlCommand cmd = new SqlCommand("select * from PRODUITS where designation
like @x", conn);
            SqlParameter p = new SqlParameter("@x", "%P%");
            cmd.Parameters.Add(p);
            SqlDataReader dr = cmd.ExecuteReader();
            while (dr.Read()){
                Console.WriteLine(dr.GetString(1));
            }
            dr.Close();
            conn.Close();
        }
    }
}
```

# Les Transactions

- Les transactions permettent de regrouper des commandes SQL dans une même entité.
- Si l'une des commandes échoue alors l'opération sera arrêtée et la base de données retrouvera son état initial.
- Pour créer une transaction, il suffit d'instancier votre *Transaction* puis de l'assigner en appelant la méthode *BeginTransaction* à la connexion.
- De plus les transactions reposent sur le principe de quatre caractéristiques appelé ACID qui apporte plus de clarté sur la définition d'une transaction :
  - - **Atomicité**, qui signifie que la mise à jour de la base de données doit être totale ou nulle, c'est le principe du "tout ou rien".
  - - **Cohérence**, qui indique que les modifications apportées doivent être valides.
  - - **Isolation**, qui définit que les transactions lancées au même moment ne doivent pas s'interférer entre elles.
  - - **Durabilité**, qui assure que toutes les transactions sont lancées de manière définitive.

# Les Transactions

- Les transactions sont managées au niveau de la connexion.
- Par conséquent nous pourrons commencer une transaction en ouvrant une connexion avec une base de données pour ensuite commencer les transactions en appelant la méthode *BeginTransaction* issues d'une instance de la classe *SqlTransaction*.
- Puis vous devez définir quelle commande nécessite une transaction. Enfin à la fin du traitement des données vous avez la possibilité soit de valider vos transactions grâce à la méthode *Commit* soit de les annuler grâce à la méthode *Rollback*.

# Les Transactions

- Niveaux d'isolation :

- Les niveaux d'isolation vous permettent de gérer les problèmes d'intégrité des données ainsi que des accès simultanés à celles-ci par le biais des transactions.
- Vous trouverez ci-dessous la liste des propriétés *IsolationLevel* associés à l'objet *Transaction* :
  - **Chaos** : Les modifications en attente de transactions très isolés ne peuvent être écrasées.
  - **ReadCommitted** : Les verrouillages partagés sont maintenus pendant que les données sont en lecture pour éviter tout défaut, mais les données sont modifiables durant la transaction, entraînant une lecture non répétable ou des données "fantôme".
  - **ReadUncommitted** : Il n'y a pas de verrouillage entraînant la possibilité de tout défaut de lecture.
  - **RepeatableRead** : Toutes les données utilisées dans une requête sont verrouillées. Par conséquent les lectures ne sont pas répétables mais des données "fantôme" peuvent exister et d'autres utilisateurs ne peuvent mettre à jour les données.

# Les Transactions

- Niveaux d'isolation (Suite) :
  - **Serializable** : Les verrouillages sont placés sur toutes les données qui sont utilisées dans une requête, ce qui empêche d'autres utilisateurs de mettre à jour les données.
  - **Snapshot** : Réduit le blocage par le stockage de données qu'une seule application peut lire pendant qu'une autre est train de les modifier. Cela indique que d'une seule transaction vous ne pouvez voir les modifications faites dans d'autres transactions.
  - **Unspecified** : Aucun niveau ne peut être déterminé. Ainsi si le niveau d'isolation est défini sur celui-ci alors la transaction exécute selon le niveau d'isolation par défaut du sous-jacent du type de base de données.

# Utilisation des transactions

```
// Commencer une transaction
SqlTransaction tx = conn.BeginTransaction();
    try
    {
        // Exécution des commandes
        .....
        // Valider la transaction
        tx.Commit();

    }
    catch (Exception e)
    {
        // En cas d'exception, Annuler la transaction
        tx.Rollback();
        Console.WriteLine(e.StackTrace);
    }
```

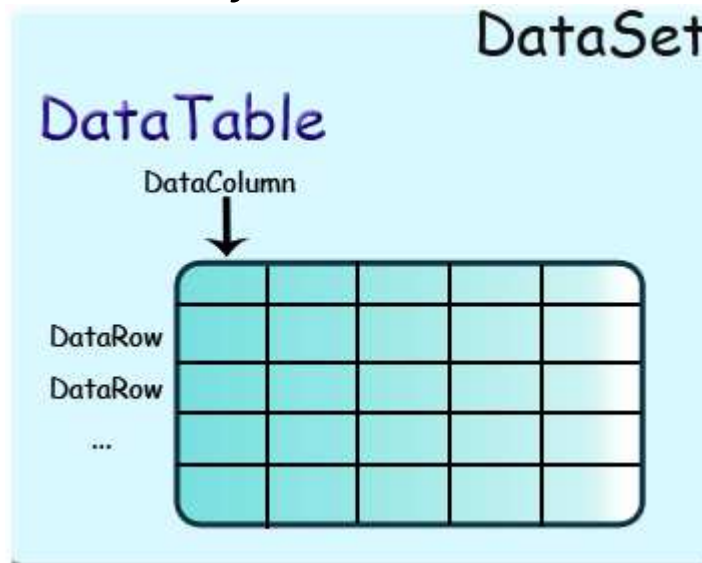
# Exemple d'utilisation des transactions

```
using System; using System.Collections.Generic; using System.Linq;
using System.Text; using System.Data.SqlClient;
namespace ConsoleApplication1 {
    class Program    {
        static void Main(string[] args)        {
            String strConn = "Data Source=localhost\\SQLEXPRESS;Initial Catalog= DB_CAT5;Integrated
Security=True";
            SqlConnection conn = new SqlConnection(strConn);
            conn.Open();
            SqlTransaction tx = conn.BeginTransaction();
            try {
                SqlCommand cmd1 = new SqlCommand("insert into CATEGORIES(NOM_CAT) values(@x)", conn);
                SqlParameter p = new SqlParameter("@x", "BB");
                cmd1.Parameters.Add(p);
                cmd1.Transaction = tx;
                cmd1.ExecuteNonQuery();
                tx.Commit();
            } catch (Exception e) {
                tx.Rollback();
                Console.WriteLine(e.Message);
            }
            conn.Close();
        } } }
```



# Mode Déconnecté : DataSet

- L'objet DataSet :
  - Le DataSet est stocké dans l'espace de nom System.Data. C'est un cache de données en mémoire, c'est-à-dire qu'il permet de stocker temporairement des données utilisées dans votre application.
  - Le DataSet contient la collection d'objets DataTable qui peuvent être liés avec les objets DataRelation.
  - Dans le cas du mode déconnecté, cet objet va nous permettre d'importer la partie désirée de la base de données (fonction de la requête de sélection) en local.
  - Ainsi grâce à des objets nécessaires à la connexion classique (commande select, connections string...) et un DataAdapter, nous pourrions relier ("Binder") un DataSet sur une base de donnée (en lecture et en écriture grâce à une méthode de mise à jour de la base de donnée).







## Mode Déconnecté : DataAdapter

- L'objet *DataAdapter* permet de relier un *DataSet* à une base de données.
- En revanche le *DataAdapter* change suivant le fournisseur, c'est-à-dire, par exemple, pour une base de données SQL, ce sera *SqlDataAdapter*. C'est grâce à cela que votre application pourra communiquer avec la base de données et par exemple mettre à jour celle-ci.

# Utilisation d'un DataSet et DataAdapter

```
String strConn="Data Source=TOSHIBA-PC\\SQLEXPRESS;Initial
    Catalog=DB_CATALOGUE;Integrated Security=True";
String req = "select * from PRODUITS";
// Créer Une connexion
IDbConnection conn = new SqlConnection(strConn);
// Créer Une commande
IDbCommand cmd = conn.CreateCommand();
// Définir la requête de cette commande
cmd.CommandText=req;
// Définir le type de commande SQL
cmd.CommandType=CommandType.Text;
// Créer Un DataSet
DataSet ds = new DataSet();
// Créer Un DataAdapter
IDbDataAdapter da = new SqlDataAdapter();
// Associer la commande à l'objet DataAdapter
da.SelectCommand=cmd;
// Exécuter la commande SQL et stocker les résultats de l'objet DataSet
da.Fill(ds);
```

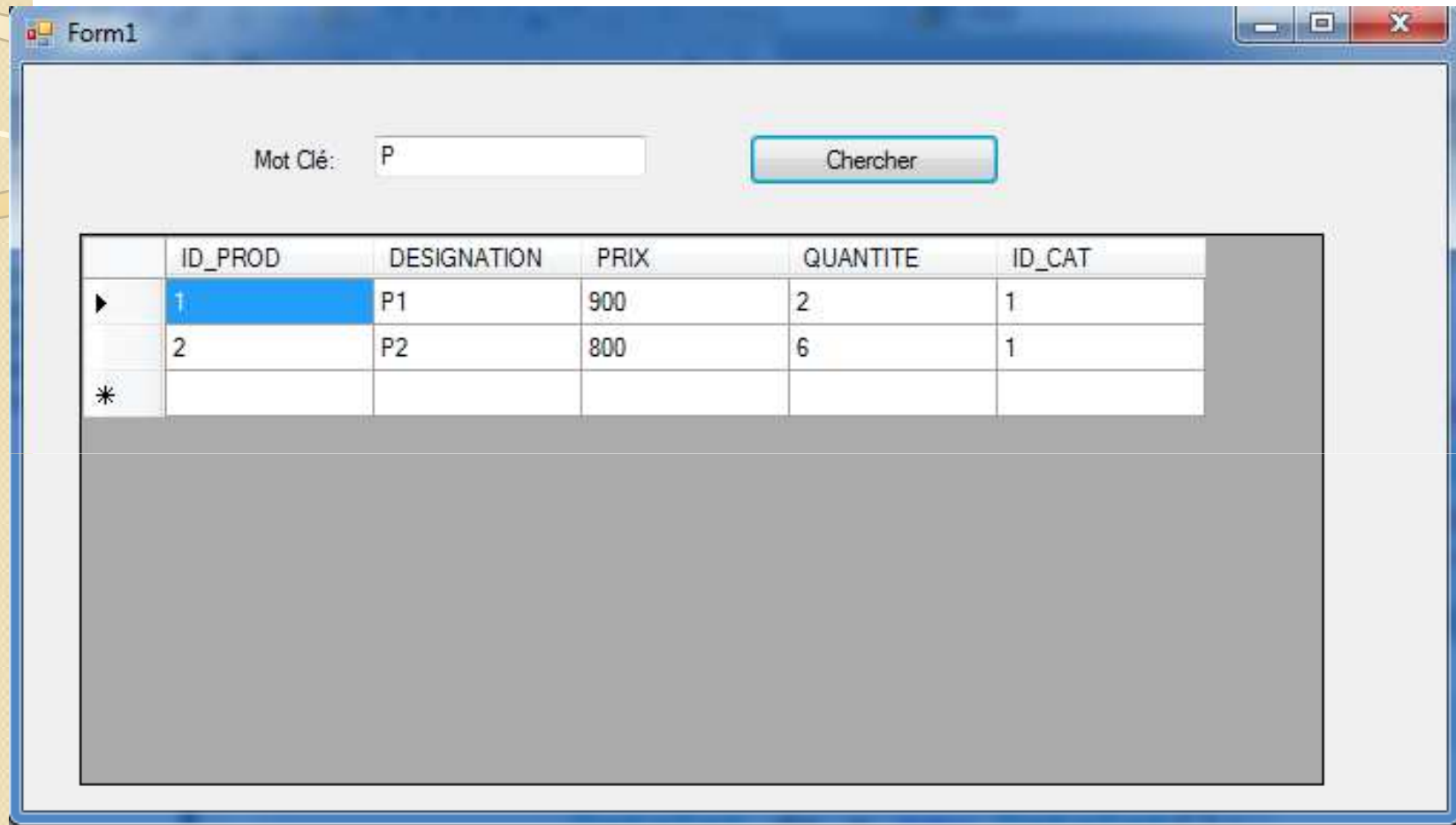
# Parcourir un DataSet

```
// Afficher le nom de chaque colonne de la première table
foreach(DataColumn dc in ds.Tables[0].Columns )
{
    Console.Write(dc.ColumnName+"\t");
}
Console.WriteLine();
// Afficher les lignes de la table
foreach (DataRow dr in ds.Tables[0].Rows)
{
    foreach (DataColumn dc in ds.Tables[0].Columns)
    {
        Console.Write(dr[dc.ColumnName]+"\t\t");
    }
    Console.WriteLine();
}
```

# Exemple d'application en mode déconnecté

```
using System; using System.Collections.Generic; using System.Linq;
using System.Text; using System.Data.SqlClient; using System.Data;
namespace ConsoleApplication2{
    class Program{
        static void Main(string[] args){
            String strConn = "Data Source=localhost\\SQLEXPRESS;Initial Catalog= DB_CAT5;Integrated
Security=True";
            IDbConnection conn = new SqlConnection(strConn);
            string req = "select * from produits";
            IDbCommand cmd = conn.CreateCommand();
            cmd.CommandText = req;    cmd.CommandType = CommandType.Text;
            DataSet ds = new DataSet();  IDbDataAdapter da = new SqlDataAdapter();
            da.SelectCommand = cmd;  da.Fill(ds);
            foreach (DataColumn dc in ds.Tables[0].Columns){
                Console.Write(dc.ColumnName + "\t");
            }
            Console.WriteLine();
            foreach (DataRow dr in ds.Tables[0].Rows){
                foreach (DataColumn dc in ds.Tables[0].Columns){
                    Console.Write(dr[dc.ColumnName] + "\t");
                }
                Console.WriteLine();
            }
        }
    }
}
```

# Exemple d'application WindowsForm



Mot Clé: P

	ID_PROD	DESIGNATION	PRIX	QUANTITE	ID_CAT
▶	1	P1	900	2	1
	2	P2	800	6	1
*					

# Exemple d'application WindowsForms

```
using System; using System.Collections.Generic; using System.ComponentModel;
using System.Data; using System.Drawing; using System.Linq; using System.Text;
using System.Windows.Forms; using System.Data.SqlClient;
namespace WindowsFormsApplication1 {
    public partial class Form1 : Form {
        public Form1() { InitializeComponent(); }
        private void button1_Click(object sender, EventArgs e) {
            string mc = textBoxMC.Text;
            String strConn = "Data Source=localhost\\SQLEXPRESS;Initial Catalog= DB_CAT5;Integrated
Security=True";
            IDbConnection conn = new SqlConnection(strConn);
            string req = "select * from produits where DESIGNATION like @x";
            SqlParameter p=new SqlParameter("@x", "%" + mc + "%");
            IDbCommand cmd = conn.CreateCommand();
            cmd.CommandText = req;
            cmd.CommandType = CommandType.Text;
            cmd.Parameters.Add(p);
            DataSet ds = new DataSet();
            IDbDataAdapter da = new SqlDataAdapter();
            da.SelectCommand = cmd;    da.Fill(ds);
            dataGridView1.DataSource = ds.Tables[0];
        } } }
```