Microsoft

# Visual C#

## Les fondamentaux

Mars 2019 – Version 1.5

Microsoft .net

Prof. Youness BOUKOUCHI
ENSA d'Agadir

# INTRODUCTION

- C# .NET est le langage de programmation orientée objet, commercialisé par Microsoft depuis 2002.
- Il a été développé dans le but de pouvoir créer facilement divers types d'applications en tirant le meilleur des produits et technologies Microsoft.
- Le couple C# et Framework .NET englobe les dernières avancées des langages de programmation (Generic, Lambda, Inférence de type, Linq...).
- Syntaxe très proche de C++ et Java

# VERSION C#

## L'évolution de C#

| Version | .NET Framework | Visual Studio | Important Features |
|---|---|---|---|
| C# 1.0 | .NET Framework 1.0/1.1 | Visual Studio .NET 2002 | •Basic features |
| C# 2.0 | .NET Framework 2.0 | Visual Studio 2005 | Generics, Partial types, Anonymous methods, Iterators, Nullable types, Private setters (properties), Method group conversions (delegates), Covariance and Contra-variance, Static classes |
| C# 3.0 | .NET Framework 3.0\3.5 | Visual Studio 2008 | Implicitly typed local variables, Object and collection initializers, Auto-Implemented properties, Anonymous types, Extension methods, Query expressions, Lambda expressions, Expression trees, Partial Methods |
| C# 4.0 | .NET Framework 4.0 | Visual Studio 2010 | Dynamic binding (late binding), Named and optional arguments, Generic co- and contravariance, Embedded interop types |
| C# 5.0 | .NET Framework 4.5 | Visual Studio 2012/2013 | Async features, Caller information |
| C# 6.0 | .NET Framework 4.6 | Visual Studio 2013/2015 | Expression Bodied Methods, Auto-property initializer, nameof Expression, Primary constructor, Await in catch block, Exception Filter, String Interpolation |
| C# 7.0 | .NET Core | Visual Studio 2017 | out variables, Tuples, Discards, Pattern Matching, Local functions, Generalized async return types, throw Expressions |

# HELLO WORLD !

- La procédure principale doit être une méthode statique publique d'une classe, elle doit s'appeler « **Main** », c'est le point d'entrée de l'application.
- Les bibliothèques utilisées doivent être référencées dans le projet, elle est fortement structurée:
  - utilisation d'espaces de nommages «namespace»,
  - accès via la directive «using».

```
using System;
namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
            /* commentaire : hollo ENSA */
         Console.WriteLine("Hello ENSA! ");
      }
   }
}
```

# L'INSTRUCTION USING

- Un espace de nom correspond à un endroit où l'on range des méthodes et des objets, c'est un peu comme des répertoires.
- Il est caractérisé par des mots séparés par des points « . »
- On utilise le mot-clé **using** pour inclure un espace de nom comme raccourci dans son programme, ce qui permet de ne pas avoir à préfixer les types de leurs espaces de noms complets.
- Par exemple **using System** indique au compilateur que nous allons utiliser l'espace de nom **System.** Ainsi, il n'est plus obligatoire de préfixer l'accès à DateTime par System.

```
namespace exempleHello
{
class Program
 {
  static void Main(string[] args)
  {
     System.Console.WriteLine(System.DateTime.Now);
   }
 }
}
// affiche : 25/02/2018 14:45:58
```

```
using System;
namespace exempleHello
{
class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine(DateTime.Now);
      }
   }
}
```

# ENVIRONNEMENT VS

TPCSharp (Exécution) - Microsoft Visual Studio

Lancement rapide (Ctrl+Q)

FICHIER EDITION AFFICHAGE PROJET GÉNÉRER DÉBOGUER ÉQUIPE SQL OUTILS TEST ANALYSER FENÊTRE ?

Continuer Debug

Boîte... Boîte à outi

Général

Il n'existe aucun contrôle utilisable dans ce groupe. Faites glisser un élément ici pour l'ajouter à la boîte à outils.

Program.cs

TPCSharp.Program Main(string[] args)

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

namespace TPCSharp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Saisir votre nom : ");
            String nom=Console.ReadLine();
            Console.Write("Saisir votre age : ");
            int age = int.Parse( Console.ReadLine());
            Console.WriteLine("Bienvenue : "+nom+" ( "+age+"ans)");
            Console.ReadKey();
        }
```

100 %

Sortie Fenêtre Exécution

Explorateur de s...

Rechercher Explorate

TPCSharp
- Properties
- Références
- App.config
- Program.cs

# VALEURS ET RÉFÉRENCES

- C# est un langage très fortement typé.
- Il y a deux grandes catégories de valeurs :
  - **Variable de type valeur : La variable contient la valeur de la donnée**
    - *Les types prédéfinis, les structures et les enum sont de type valeur*
    - Exemple: **int mavariable=100;**
  - **Variable de type référence : La variable stocke une référence aux données**
    - object, tableaux, classes utilisateur, etc.
    - Exemple : **Person p=new Person(); object[] tab;**

# TYPES ET VARIABLES

- Une variable est une zone mémoire permettant de stocker une valeur d'un type particulier.
- Le C# possède plein de types prédéfinis, comme les entiers (int), les chaînes de caractères (string), etc.
- On utilise l'opérateur = pour affecter une valeur à une variable.
- Il est possible de faire des opérations entre les variables.

```
int i = 2;

long l = 3L;

float f = 2.12f;

double d = 3.14D;

char c = 'A';

string s = "Hello World!\n";

bool b = true;

int[] tab1 = new int[3] {7,9,42};

tab1[0] = 2;

long[,] tab2 = new long[5,4];

tab2[2,3] = 5L;

object o = new int();
```

# TYPES ET VARIABLES

| Alias | .NET Type | Type | Size (bits) | Range (values) |
|---|---|---|---|---|
| byte | Byte | Unsigned integer | 8 | 0 to 255 |
| sbyte | SByte | Signed integer | 8 | -128 to 127 |
| int | Int32 | Signed integer | 32 | -2,147,483,648 to 2,147,483,647 |
| uint | UInt32 | Unsigned integer | 32 | 0 to 4294967295 |
| short | Int16 | Signed integer | 16 | -32,768 to 32,767 |
| ushort | UInt16 | Unsigned integer | 16 | 0 to 65,535 |
| long | Int64 | Signed integer | 64 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| ulong | UInt64 | Unsigned integer | 64 | 0 to 18,446,744,073,709,551,615 |
| float | Single | Single-precision floating point type | 32 | -3.402823e38 to 3.402823e38 |
| double | Double | Double-precision floating point type | 64 | -1.79769313486232e308 to 1.79769313486232e308 |
| char | Char | A single Unicode character | 16 | Unicode symbols used in text |
| bool | Boolean | Logical Boolean type | 8 | True or False |
| object | Object | Base type of all other types | | |
| string | String | A sequence of characters | | |
| decimal | Decimal | Precise fractional or integral type that can represent decimal numbers with 29 significant digits | 128 | (+ or -)1.0 x 10e-28 to 7.9 x 10e28 |
| DateTime | DateTime | Represents date and time | | 0:00:00am 1/1/01 to 11:59:59pm 12/31/9999 |

# LES CONVERSIONS ENTRE LES TYPES

- Entre les types compatibles **(le casting)** : est l'action de convertir la valeur d'un type dans un autre et qui sont compatibles entre eux. Exemple:
  - int i=200;
  - short s=(short)i; // un cast explicite
  - short s = 200 ;
  - int i = s; // un cast implicite
- Entre les types incompatibles: des types qui ne se ressemblent pas mais qui ont le même sens (par exemple : une chaîne de caractères qui contient uniquement des chiffres en un nombre).
- Il existe plusieurs méthode pour la conversion, par exemple: **chaine ⇨ int**
  - La méthode: **int.Parse()**
  - la méthode: **Convert.ToInt32()**
  - La méthode: **int.TryParse** renvoie vrai si la conversion est bonne, et faux sinon.

```
using System;
class Program
 {
  static void Main(string[] args)
  {
   string chaineAge="ab20cd";
   int age;
   if(int.TryParse(chaineAge,out age))
    {
      Console.WriteLine("La conversion est possible, âge vaut " +age);
    }
   else
   {
      Console.WriteLine("Conversion impossible");
    }
  }
}
```

# LE TYPE NULLABLE

## Initialisation des variables

- En C#, on ne peut pas utiliser une variable tant qu'elle n'est pas initialisé
- Si on désire vraiment déclarer une variable non initialisée, il faut lui donner un type annulable (***nullable***).
- Pour cela, on suffixe le type d'un point d'interrogation.
  - Exemple : `int ? i = null ;`

# LE TYPE VAR

- C# version 3 (VS 2008) a introduit le type **var** qui correspond à un type indéterminé.
- Le véritable **type** de la variable est déterminé dès la compilation, d'après la valeur d'initialisation.
- Le type de la variable ne peut être modifié par la suite. Le compilateur ne fait donc que substituer un véritable **type** à **var**.

```
Pers p = new Pers { ..... };
int q = 15;
string s = "Hello";
int[] tab = new int[] {10, 20, 30};
```

```
var p = new Pers { ..... };
var q = 15;
var s = "Hello";
var tab = new [] {10, 20, 30};
```

# LES TYPES ANONYMES

- Les types anonymes sont des types dont on n'a pas défini la classe au préalable. Une classe sans nom.
- il est possible de créer des types anonymes grâce à l'opérateur new suivi de la description des propriétés de ce type.
- Les instances sont manipulées grâce au mot-clé var.

```
var unePersonneAnonyme = new { Prenom = " ENSA ", Age = 30 };
```

# Les opérations I/O

# OPÉRATIONS D'ENTRÉE/SORTIE

## *Affichages:*

- Des fonctions statiques de la classe Console permettent de mettre en format et d'afficher une chaîne de caractères.
- **Write** affiche une chaîne tandis que **WriteLine** force un retour à la ligne suivante aussitôt après affichage.
- Exemple:

```
Console.Write("Bon");
Console.WriteLine("jour"); //Bonjour
int i=10, j=5;
Console.WriteLine("i = " + i + " et j = " + j); //i = 10 et j = 5
Console.WriteLine("i = {0} et j = {1}", i, j); //i = 10 et j = 5
Console.WriteLine(i + j); //15
Console.WriteLine("Somme = " + i + j); //Somme = 105
Console.WriteLine("Somme = " + (i + j)); //Somme = 15
Console.WriteLine("Somme = {0}", i + j); //Somme = 15
```

# OPÉRATIONS D'ENTRÉE/SORTIE

***De la couleur pour la console*** :

Depuis C# version 2 (Visual Studio 2005), il est possible d'afficher en couleurs, de positionner le curseur avant affichage, et de redimensionner la fenêtre console.

- Console.BackgroundColor = ConsoleColor.Blue;
- Console.ForegroundColor = ConsoleColor.Red;
- Console.Clear();
- Console.Title = "Mon application";
- Etc.

***Des sons pour la console*** :

.NET version 2 a introduit des effets sonores dans la classe Console:

- Console.**Beep()**: Émet un bip dans le haut-parleur de la machine.
- Console.**Beep(int freq, int durée)**: Émet un signal sonore. La durée est exprimée en millisecondes et la fréquence en Hertz (entre 37 et 32 767). Certaines fréquences (surtout dans les fréquences élevées) peuvent être inaudibles pour certaines personnes.

# OPÉRATIONS D'ENTRÉE/SORTIE

## *Lecture de données saisies au clavier*

### ReadLine()

- C'est une fonction statique de la classe Console,
- Elle lit les caractères tapés au clavier, jusqu'à la validation par la touche ENTREE.
- **ReadLine** renvoie alors la chaîne lue sous forme d'un string.

```
Console.Write("Saisissez un entier :");
string s = Console.ReadLine();
int lu = int.Parse(s);
Console.WriteLine("J'ai lu :" + lu.ToString());
```

# OPÉRATIONS D'ENTRÉE/SORTIE

## *Lecture de données saisies au clavier*

Console.ReadKey():

- il permet de saisir qu'un seul caractère.
- **true** indiquer au C# que la saisie apparaisse à l'écran sinon false.
- Elle renvoie en l'occurrence une variable du type **ConsoleKeyInfo** qui est une énumération du type **ConsoleKey**.

```
using System;
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Voulez-vous continuer(O/N)?");
    ConsoleKeyInfo saisie = Console.ReadKey(true);
     if(saisie.Key==ConsoleKey.O)
     {
       Console.WriteLine("On continue... ");
     }
      else
     {
       Console.WriteLine("On s'arrête ... ");
     }
  }
}
```

# Les instructions conditionnelles et les boucles

# L'INSTRUCTION IF

```
if(condition)
{
   bloc d'instructions si condition vraie;
}
else
{
   bloc d'instructions si condition fausse;
}
```

```
if(civilite ==" Mme " || civilite == "Mlle ")
   Console.WriteLine(" Vous êtes une femme") ;
else
   Console.WriteLine("Vous êtes un homme") ;
```

```
decimal compteEnBanque = 300 ;

if (compteEnBanque>= 0)

  Console.WriteLine (" Votre compte est créditeur ");

else

  Console.WriteLine (" Votre compte est débiteur ") ;
```

Une condition se construit grâce à des opérateurs de comparaison.

| Opérateur | Description |
|---|---|
| == | Égalité |
| != | Différence |
| > | Supérieur à |
| < | Inférieur à |
| >= | Supérieur ou égal |
| <= | Inférieur ou égal |
| && | ET logique |
| \|\| | OU logique |
| ! | Négation |

# L'INSTRUCTION SWITCH

```
switch(variable)
{
case valeur_1: instruction(s) pour valeur_1;
                break;
case valeur_2: instructions(s) pour valeur_2;
                break;
...
default: instruction(s) pour cas par défaut;
         break;
}
```

```
string civilite = "M.";
switch ( civilite)
{
 case "M." :Console.WriteLine("Bonjour monsieur");
            break ;
 case "Mme" :Console.WriteLine("Bonjour madame");
            break ;
 case "Mlle":Console.WriteLine("Bonjour mademoiselle");
            break;
 default :Console.WriteLine("Bonjour inconnu");
            break ;
}
```

# LA BOUCLE FOR

```
for ( instructions d'initialisation; condition; instructions de fin de boucle)
{
  // une ou plusieurs instructions qui seront exécutées
  // tant que la condition est vraie
}
```

```
class Program
{
   static void Main(string[] args)
   {
     int compteur;
     for (compteur=0; compteur<10;compteur++)
     {
       Console.WriteLine("Bonjour ENSA" + compteur);
     }
    }
}
```

# LA BOUCLE WHILE

La boucle while consiste à évaluer la condition avant d'exécuter les instructions de la boucle :

```
while ( condition)
{
// une ou plusieurs instructions (formant le corps de la boucle)
// qui sont exécutées tant que la condition est vraie
}
```

```
class Program
{
   static void Main(string[] args)
   {
      int i = 0;
      while (i < 10 )
      {
         Console.WriteLine("Bonjour ENSA" + i);
         i ++;
       }
    }
}
```

# LA BOUCLE DO/WHILE

La forme **do/while** consiste à exécuter les instructions de la boucle puis évaluer la condition, L'utilisation du mot-clé **do** permet d'indiquer le début de la boucle :

```
do
{
  // une ou plusieurs instructions
} while ( condition);
```

```
class Program
{
    static void Main(string[] args)
    {
        int i = 0;
        do
        {
            Console.WriteLine("Bonjour ENSA" + i);
            i++;
        } while (i < 10);
    }
}
```

# LA BOUCLE FOREACH

- L'instruction ***foreach*** permet de parcourir un tableau ainsi qu'une collection.
- la boucle ***foreach*** est une boucle **en lecture seule**. Cela veut dire qu'il n'est pas possible de modifier l'élément de l'itération en cours.
- **Exemple :**
  - `int[] ti = new int[] { 10, 5, 15, 0, 20 };`
  - `foreach (int i in ti) Console.WriteLine(i);`
  - `string[] ts = { "BOUKOUCHI", "ELYAAKOUBI", "NOUHI" };`
  - `foreach (string s in ts) Console.WriteLine(s);`

# BREAK ET CONTINUE

Les instructions break et continue peuvent être exécutées dans une boucle :

- **continue :** fait passer directement à l'itération suivante ;
- **break** : fait quitter la boucle.

```
class Program
  {
     static void Main(string[] args)
     {
        for (int j = 0; j < 5; j++)
        {
           Console.Write(j);
           if (j == 2) continue;
              Console.Write("A");
           if (j == 3) break;
              Console.Write("B");
        }
     }
  } //affiche 0AB1AB23A.
```

# LES SAUTS GOTO

L'instruction **goto** permet de brancher à une étiquette (il s'agit d'un identificateur terminé par **:** ) mais celle-ci doit impérativement faire partie de sa fonction

```
class Program
    {
        static void Main(string[] args)
        {
         Console.WriteLine("AAAA");
         goto làbas;
         ici: Console.WriteLine("BBBB");
         làbas: Console.WriteLine("CCCC");
        }
    }
//Le programme affiche AAAA puis CCCC.
```

# Les classes en C#

# UNE CLASSE

- Les classes sont introduites avec « **class** »
  - Les attributs : ce sont les champs de données.
  - Les méthodes:  autre nom pour les fonctions membres.
- Plusieurs types de mode d'accès (**l'encapsulation**):
  - **private** : visible uniquement par les classes du même espace de noms ***(par défaut)***
  - **public** : visible à tout le monde, d'un autres espace de noms
  - **protected** : pour les classes imbriquées, sera visible uniquement par la classe conteneur et ses héritiers
  - **internal** : visible par les autres classes du même assembly (l'EXE ou la DLL du projet).
- La notion de classe **sealed** en C# correspond strictement à la notion de classe final de Java : ce sont des classes non héritables

# DÉCLARATION D'UNE CLASSE

```
using System;
namespace ExempleClasse
{
    Public class Person
    {
      // les membres
      public String firstName;
      public String lastName;
      public Person()
      {// constructeur par défaut
        }
    }
}
```

```
using System;
namespace ExempleClasse
{
  class Program
  {
   static void Main(string[] args)
   {
    Person p = new Person();
    p.firstName = "Youness";
    p.lastName = "BOUKOUCHI";
    Console.WriteLine("Mr "+p.firstName+" "+ p.lastName);
    Console.ReadKey(true);
    }
   }
}
```

# CONSTRUCTEURS ET DESTRUCTEURS

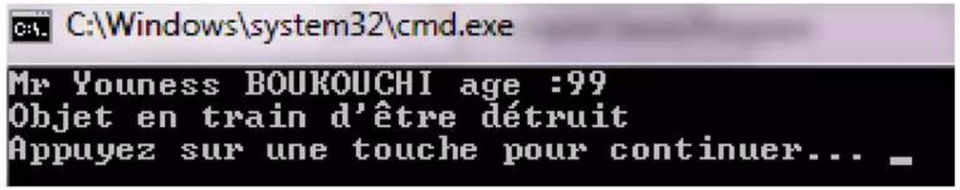
- Un constructeur est appelée automatiquement lors de l'instanciation.
- Même nom que la classe, forcément public.
- Peuvent être plusieurs, se différencient par les paramètres
- On peut réutiliser un des constructeurs dans un autre constructeur par **this**,
- **this()** provoque l'appel du constructeur sans argument.

```
using System;
namespace ExempleClasse
{ public class Person
   {  public String firstName;
      public String lastName;
      public int age;
      public Person()
      { }
      // constructeur personnalisé
      public Person(String firstName, String lastName)
      {  this.firstName = firstName;
         this.lastName = lastName;
      }
      // constructeur appel un autre constructeur
      public Person(String firstName, String lastName, int age)
         : this(firstName, lastName)
      {   this.age = age;  }
   }
}
```

# CONSTRUCTEURS ET DESTRUCTEURS

- Un destructeur porte le nom de la classe préfixé de ~ *(caractère tilde)* et n'accepte aucun argument
- les destructeurs sont bien moins utiles en C#, l'environnement .NET s'occupe de libérer automatiquement les objets qui n'ont plus cours.

```
using System;
namespace ExempleClasse
{ public class Person
   {  public String firstName;
      public String lastName;
      public int age;
      public Person()
      { }
      // constructeur personnalisé
      public Person(String firstName, String lastName)
      {  this.firstName = firstName;
         this.lastName = lastName;
      }
      // constructeur appel un autre constructeur
      public Person(String firstName, String lastName, int age)
         : this(firstName, lastName)
      {   this.age = age; }
     // destructeur
      ~Person  () {
         Console.WriteLine("Objet en train d'être détruit");  }
   }
}
```



# MODE DE PASSAGE DES PARAMÈTRES

*Les trois modes principaux de transmission des paramètres sont :*

***1- Le passage par valeur:*** est valable pour tous les types élémentaires (int, byte, short, long, …) et les objets( c'est en fait la référence de l'objet qui est passée par valeur).

```
using System;
namespace ExempleClasse
{
    class Program
    {
        static void Main(string[] args)
        {
           int a;
           a = 0;
           Console.WriteLine("avant: a="+a);
           f( a);  // appel de la fonction f
           Console.WriteLine("après: a=" + a);
        }
           // la fonction f
        static void f( int aa)
        {
           aa = 10;
           Console.WriteLine("en cours: a=" + aa); }
    }
}
```

C:\Windows\system32\cmd.exe

```
avant: a=0
en cours: a=10
après: a=0
Appuyez sur une touche pour continuer...
```

# MODE DE PASSAGE DES PARAMÈTRES

*Les trois modes principaux de transmission des paramètres sont :*

***2- Le passage par référence***: est valable pour tous les types de C#.

- pour indiquer un passage par référence on précède la déclaration du paramètre formel du mot clef **ref**.
- Lors de l'appel d'un paramètre passé par référence, le mot clef **ref** doit obligatoirement précéder le paramètre effectif *qui doit obligatoirement avoir été initialisé auparavant*.

```
using System;
namespace ExempleClasse
{
    class Program
    {
        static void Main(string[] args)
        {

            int a;
            a = 0;
            Console.WriteLine("avant: a="+a);
            f( ref a);  // appel de la fonction f
            Console.WriteLine("après: a=" + a);
        }
            // la fonction f
        static void f( ref int aa)
        {
            aa = 10;
            Console.WriteLine("en cours: a=" + aa); }
    }
}
```

C:\Windows\system32\cmd.exe

```
avant: a=0
en cours: a=10
après: a=10
Appuyez sur une touche pour continuer...
```

# MODE DE PASSAGE DES PARAMÈTRES

*Les trois modes principaux de transmission des paramètres sont :*

***3- Le passage par résultat***: Le passage par résultat est valable pour tous les types de C#.

- pour indiquer un passage par résultat on précède la déclaration du paramètre formel du mot **out**
- Lors de l'appel d'un paramètre passé par résultat, le mot clef **out** doit obligatoirement précéder le paramètre effectif *qui n'a pas besoin d'avoir été initialisé avant l'appel mais qui doit l'avoir à la fin du traitement*.

```
using System;
namespace ExempleClasse
{
    class Program
    {
        static void Main(string[] args)
        {
            int a;
            //a = 0;
            //Console.WriteLine("avant: a="+a);
            f( out a);  // appel de la fonction f
            Console.WriteLine("après: a=" + a);
        }
            // la fonction f
        static void f( out int aa)
        {
            aa = 10;
            Console.WriteLine("en cours: a=" + aa); }
    }
}
```

C:\Windows\system32\cmd.exe

```
en cours: a=10
après: a=10
Appuyez sur une touche pour continuer...
```

# LES PARAMÈTRES NOMMÉS /PAR DÉFAUT

- C# version 4 (Visual Studio 2010) a introduit les arguments par défaut ainsi que les arguments nommés.
  - double f(int num=10, int den=5) { ..... }
- Les arguments par défaut, s'ils sont présents, doivent apparaître après les arguments sans valeur par défaut.
  - f(13); // L'argument den, ici absent, prend automatiquement la valeur 5
  - f(); // Utilisation des deux valeurs par défaut.
  - Ecrire f(,7) // n'est pas autorisé
- il est également possible d'appeler une fonction avec des arguments nommés (l'ordre des arguments lors de l'appel n'ayant alors plus d'importance).
  - f(den:7, num:21);
  - f(den:17);
  - int A=12, B=2; f(den:ref B, num:ref A);

# LES PARAMÈTRES CONST ET READONLY

- Un champ d'une classe peut être qualifié de **const**.
- Il doit alors être initialisé mais il ne peut être modifié par la suite.
- Un champ d'une classe peut également être qualifié de **readonly**. Il est alors initialisé dans le constructeur de la classe mais ne peut être modifié par la suite.
- C'est donc une constante dont l'initialisation de la valeur peut être retardée jusqu'au moment de l'exécution (et non à la compilation).

# LES MÉTHODES

Pour déclarer une méthode, nous aurons :

***Signature de la méthode***

**{**

***Corps de la méthode***

**}**

- Une méthode regroupe un ensemble d'instructions pouvant prendre des paramètres et pouvant renvoyer une valeur.
- Les paramètres d'une méthode doivent être utilisés avec le bon type.
- Une méthode qui ne renvoie rien est préfixée du mot-clé **void**.
- Le point d'entrée d'un programme est la méthode statique **Main()**.
- Le mot-clé **return** permet de renvoyer une valeur du type de retour de la méthode, à l'appelant de cette méthode.

# LES PROPRIÉTÉS ET LES ACCESSEURS

**Les variables membres:**

- un objet peut avoir une caractéristique sous la forme d'une variable publique qui fait partie de la classe

**Les propriétés**

- Elles sont des variables évoluées, à mi-chemin entre une variable et une méthode.
- Une propriété s'utilise comme un champ d'une classe mais provoque en fait l'appel d'une fonction.
- Une ou deux fonctions, appelées «accesseurs», peuvent être associées à une propriété :
  - fonction ***get*** pour l'accès en lecture de la propriété ;
  - fonction ***set*** pour l'initialisation ou la modification de la propriété.
  - Si la fonction ***set*** est absente, la propriété est en lecture seule.

```
class Voiture
  {  private int vitessePrivee;
     public int Vitesse{
        get {
           return vitessePrivee;
        }
        set  {
           vitessePrivee = value ;
        }
     }
  }
```

```
class Program
{
  static void Main(string[] args)
   {
     Voiture voiture = new Voiture();
      voiture.Vitesse = 500;
      Console.WriteLine("la vitesse : "+voiture.Vitesse);
    }
}
```

# LES PROPRIÉTÉS ET LES ACCESSEURS

## Les propriétés auto-implémentées

- Il s'agit de la définition d'une propriété de manière très simplifiée.
- Le compilateur génère (dans le code compilé) une variable membre qui va servir à stocker la valeur de la propriété.
- Pour avoir de la lecture seule, nous devons indiquer que l'affectation est privée (**private set**), seul la classe peut modifier la valeur de la propriété, que ce soit par une méthode ou par une propriété.

```
class Voiture
  {  private int vitessePrivee;
     public int Vitesse{
        get {
           return vitessePrivee;
        }
        set  {
           vitessePrivee = value ;
        }
     }
  }
```

```
class Voiture
  {
     public int Vitesse{get; set; }
  }
```

# LES PROPRIÉTÉS ET LES ACCESSEURS

```
class Program
{
  static void Main(string[] args)
  {
   Voiture voiture = new Voiture();
   voiture.Vitesse = 500;
   voiture.Couleur = "rouge";
   Console.WriteLine("la vitesse : "+voiture.Vitesse);
   Console.WriteLine("la couleur : "+ voiture.Couleur);
   Console.WriteLine("Le moteur : "+voiture.Moteur);
   Voiture voitureENSA = new Voiture { Vitesse=450, Couleur="Noir"};
   Console.WriteLine("la vitesse : " + voitureENSA.Vitesse);
   Console.WriteLine("la couleur : " + voitureENSA.Couleur);
   Console.WriteLine("Le moteur : " + voitureENSA.Moteur);
   }
}
```

C:\Windows\system32\cmd.exe

```
la vitesse : 500
la couleur : rouge
Le moteur : Diesel
la vitesse : 450
la couleur : Noir
Le moteur : Diesel
Appuyez sur une touche pour continuer...
```

```
class Voiture
  {
     public Voiture()
     {
        Moteur = "Diesel";
     }
     private int vitessePrivee;
     public int Vitesse
     {
        get{
           return vitessePrivee;
        }
        set
        {
           vitessePrivee = value ;
        }
     }
     public string Couleur { get; set; }
     public string Moteur { get; private set; }
  }
```

# ACCÈS AUX CHAMPS ET MÉTHODES

les classiques modificateurs de visibilité des variables et des méthodes dans le langage C#:

- **par défaut** (aucun mot clef): Les variables et les méthodes d'une classe non précédées d'un mot clef sont **private** et ne sont visibles que dans la classe seulement.
- **Public**: Les variables et les méthodes d'une classe précédées du mot clef public sont visibles par toutes les classes de tous les modules.
- **Private** : Les variables et les méthodes d'une classe précédées du mot clef private ne sont visibles que dans la classe seulement.
- **Protected** : Les variables et les méthodes d'une classe précédées du mot clef protected sont visibles par toutes les classes dérivées de cette classe.
- **Internal**: Les variables et les méthodes d'une classe précédées du mot clef internal sont visibles par toutes les classes inclues dans le même assembly.

# CHAMPS, MÉTHODES ET CLASSES STATIQUES

Le mot-clé **static** permet d'indiquer que la méthode/champs d'une classe :

- existe indépendamment de tout objet de la classe et existe même si aucun objet de la classe n'a encore été créé.
- est partagé par tous les objets de la classe ;
- est accessible par le nom de la classe suivi d'un point suivi du nom du champ/méthode;
- peut être initialisé et manipulé dans une méthode de la classe.

Les méthodes statiques et non statiques doivent respecter les règles suivantes.

- Une méthode statique n'a accès qu'aux champs statiques de sa classe.
- Une méthode statique peut appeler une autre méthode statique (de sa classe ou d'une autre) mais ne peut pas appeler une méthode non statique.
- Une méthode non statique d'une classe a accès à la fois aux champs statiques et aux champs non statiques de sa classe.

# Héritage

# HÉRITAGE

- Les classes .NET peuvent dériver d'une autre classe.
- .NET ne prend pas en charge l'héritage multiple
- La classe de base est spécifiée après le nom de la classe dérivée suivi de « : »
- L'accès à la classe de base se fait via le mot-clef « **base** », utile pour initialiser la classe de base dans le constructeur de la classe dérivée

```
public class Person
   {  protected String firstName;
      protected String lastName;
      protected int age;
      public Person() {}
      public Person(String firstName, String lastName, int age)
      {  this.firstName = firstName;
         this.lastName = lastName;
         this.age = age;  }
      public  void sePrésenter() {
         Console.WriteLine("je m'appelle " + firstName + " " +
lastName + ", j'ai " + âge+" ans");  }
   }
```

```
public class Enseignant : Person
{  public int SOM { get; set; }
   public Enseignant(String prenom, String nom, int age)
         : base(prenom, nom, age) { }
   public new void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis enseignant à l'ENSA");  }
}
```

# POLYMORPHISME

Le polymorphisme est la capacité pour une classe d'effectuer la même action sur différents types d'intervenants.

```
public class Person
  {  protected String firstName;
     protected String lastName;
     protected int age;
     public Person() {}
     public Person(String firstName, String lastName, int age)
     {  this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;  }
     public  void sePrésenter() {
        Console.WriteLine("je m'appelle " + firstName + " " +
lastName + ", j'ai " + âge+" ans");  }
  }
```

```
public class Enseignant : Person
{  public int SOM { get; set; }
   public Enseignant(String prenom, String nom, int age)
        : base(prenom, nom, age) { }
   public void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis enseignant à l'ENSA");
}
}
```

```
public class Etudiant : Person
{  public int Apogee { get; set; }
   public Etudiant(String prenom, String nom, int age)
        : base(prenom, nom, age) { }
   public void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis étudiant à l'ENSA");
}
}
```

```
class Program{
 static void Main(string[] args){
  Etudiant etd = new Etudiant("Ahmed", "Ahmed", 24) { Apogee=1188};
  Enseignant ens = new Enseignant("youness", "BOUKOUCHI", 99) { SOM=2991};
  List<Person> liste = new List<Person>();
  liste.Add(etd);
  liste.Add(ens);
  foreach (var p in liste) {  p.sePrésenter(); }
 }
}
```

```
je m'appelle Ahmed Ahmed, j'ai 24 ans
je m'appelle youness BOUKOUCHI, j'ai 99 ans
```

# SURCHARGE DE MÉTHODES

- Les membres des classes de base peuvent être redéfinis
- Ces redéfinitions ne sont accessibles que depuis la classe qui les redéfinit et ses classes dérivées
- La redéfinition provoque un avertissement du compilateur
- On peut lui indiquer que la redéfinition est voulue avec le mot-clé « **new** »

```
public class Person {
 …
 public void sePrésenter()  {
        Console.WriteLine("je m'appelle " + firstName + " " +
lastName + ", j'ai " + age+" ans"); }
 }
```

```
public class Enseignant : Person
{  …
   public void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis enseignant à l'ENSA");  }
}
```

```
public class Etudiant : Person
{  …
   public void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis étudiant à l'ENSA");  }
}
```

Liste d'erreurs

0 erreurs | 2 avertissements | 0 messages | Liste d'erreurs de recherche

| | Description | Fichier | Ligne | Colonne | Projet |
|---|---|---|---|---|---|
| 1 | 'TPCSharp.Etudiant.sePrésenter()' masque le membre hérité 'TPCSharp.Person.sePrésenter()'. Utilisez le mot clé new si le masquage est intentionnel. | Etudiant.cs | 14 | 22 | TPCSharp |
| 2 | 'TPCSharp.Enseignant.sePrésenter()' masque le membre hérité 'TPCSharp.Person.sePrésenter()'. Utilisez le mot clé new si le masquage est intentionnel. | Enseignant.cs | 14 | 22 | TPCSharp |

```
je m'appelle Ahmed Ahmed, j'ai 24 ans
je m'appelle youness BOUKOUCHI, j'ai 99 ans
```

# SURCHARGE DE MÉTHODES

- Les membres des classes de base peuvent être redéfinis
- Ces redéfinitions ne sont accessibles que depuis la classe qui les redéfinit et ses classes dérivées
- La redéfinition provoque un avertissement du compilateur
- On peut lui indiquer que la redéfinition est voulue avec le mot-clé « **new** »

```
public class Person {
 …
 public void sePrésenter()  {
        Console.WriteLine("je m'appelle " + firstName + " " +
lastName + ", j'ai " + age+" ans"); }
 }
```

```
public class Enseignant : Person
{  …
   public new void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis enseignant à l'ENSA");  }
}
```

```
public class Etudiant : Person
{  …
   public new void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis étudiant à l'ENSA");  }
}
```

Liste d'erreurs

0 erreurs | 0 avertissements | 0 messages | Liste d'erreurs de recherche

| Description | Fichier | Ligne | Colonne | Projet |
|---|---|---|---|---|

```
je m'appelle Ahmed Ahmed, j'ai 24 ans
je m'appelle youness BOUKOUCHI, j'ai 99 ans
```

# SURCHARGE DE MÉTHODES

- Les méthodes et propriétés de la classe de base peuvent être déclarées **virtuelles** ou **abstraites**
- Les mots-clefs utilisés sont « **virtual** » et « **abstract** »
- La surcharge dans les classes dérivées se fait avec le mot-clef « **override** »

```
class Program{
 static void Main(string[] args){
  Etudiant etd = new Etudiant("Ahmed", "Ahmed", 24) { Apogee=1188};
  Enseignant ens = new Enseignant("youness", "BOUKOUCHI", 99) { SOM=2991};
  List<Person> liste = new List<Person>();
  liste.Add(etd);
  liste.Add(ens);
  foreach (var p in liste) {  p.sePrésenter(); }
 }
}
```

```
public class Person {
 ...
 public virtual void sePrésenter()  {
        Console.WriteLine("je m'appelle " + firstName + " " +
lastName + ", j'ai " + age+" ans"); }
 }
```

```
public class Enseignant : Person
{  ...
   public override void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis enseignant à l'ENSA");  }
}
```

```
je m'appelle Ahmed Ahmed, j'ai 24 ans
je m'appelle youness BOUKOUCHI, j'ai 99 ans
 et je suis enseignant à l'ENSA
```

```
public class Etudiant : Person
{  ...
   public new void sePrésenter()
   { base.sePrésenter();
     Console.WriteLine(" et je suis étudiant à l'ENSA");  }
}
```

# INTERFACES

- Les classes peuvent implémenter une ou plusieurs interfaces
- Elles en dérivent comme une classe
- Les implémentations des méthodes doivent être publiques.
- Les interfaces se définissent avec le mot-clef « **interface** »
- Il est conventionnel de démarrer le nom d'une interface par un **« I »**.

```
interface ISampleInterface
  {
     void SampleMethod();
  }

  class ImplementationClass : ISampleInterface
  {
     // Explicit interface member implementation:
     void ISampleInterface.SampleMethod()
     {
        // Method implementation.
     }

     static void Main()
     {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
     }
  }
```

# INTERFACES

- Exemple 1

```
public interface IPresnetation
  {
      void sePrésenter();
  }
```

```
public class Person
{  protected String firstName;
   protected String lastName;
   protected int age;
   public Person() { }
   public Person(String firstName, String lastName, int age)
   { this.firstName = firstName;
     this.lastName = lastName;
     this.age = age;}
}
```

```
public class Etudiant : Person,IPresnetation
{    public int Apogee { get; set; }
     public Etudiant(String prenom, String nom, int age)
          : base(prenom, nom, age) { }
      public  void sePrésenter() {
      Console.WriteLine(firstName+" "+lastName+", etudiant à l'ENSA");
      }
}
```

```
public class Enseignant : Person,IPresnetation
{ public int SOM { get; set; }
  public Enseignant(String prenom, String nom, int age)
      : base(prenom, nom, age) { }
  public void sePrésenter() {
  Console.WriteLine(firstName+" "+lastName+", enseignant à l'ENSA");
      }
}
```

```
class Program {
   static void Main(string[] args) {
    Etudiant etd = new Etudiant("Ahmed", "AHMED", 24) { Apogee = 1188 };
    Enseignant ens = new Enseignant("youness", "BOUKOUCHI", 99) { SOM = 2991 };
    List<IPresnetation> liste = new List<IPresnetation>();
    liste.Add(etd);
    liste.Add(ens);
    foreach (var p in liste) { p.sePrésenter(); }
   }
}
```

# INTERFACES

- Les interfaces de .Net:
  - IDisposable
  - IComparable

```
class Voiture:IComparable
  {
      public int CompareTo(object obj)
      {
          Voiture voiture = ( Voiture )obj;
          if(this.Vitesse<voiture.Vitesse)
              return -1 ;
          if(this.Vitesse>voiture.Vitesse)
              return 1;
          return 0;
      }
  }
```

```
class Program
{
  static void Main(string[] args)
   { Voiture v1 = new Voiture() {Couleur="Noir", Vitesse=180 };
     Voiture v2 = new Voiture() { Couleur = "Rouge", Vitesse = 300 };
     Console.WriteLine(v1.CompareTo(v2));
    }
}
```

# CLASSE ABSTRAIT

- Le mot clef **abstract** est utilisé pour représenter une classe ou une méthode abstraite.
- Une méthode abstraite n'est qu'une signature de méthode sans implémentation dans la classe.
- les méthodes abstraites sont automatiquement virtuelles, elles ne peuvent être déclarées que **public** ou **protected**, enfin elles doivent être redéfinies avec le qualificateur **override**.
- Une classe abstract peut contenir des méthodes non abstraites et donc implantées dans la classe.
- Une classe abstract peut même ne pas contenir du tout de méthodes abstraites, dans ce cas une classe fille n'a pas la nécessité de redéfinir les méthodes de la classe mère pour être instanciée.

```
abstract class ShapesClass
  {
     abstract public int Area();
  }
  class Square : ShapesClass
  {  int side = 0;
     public Square(int n)
     {
        side = n;
     }
     // Area method is required to avoid
     // a compile-time error.
     public override int Area()
     {
        return side * side;
     }
  static void Main()
  {
   Square sq = new Square(12);
   Console.WriteLine("Area of the square = {0}", sq.Area());
   }
  }
  // Output: Area of the square = 144
```

# Exceptions

# LES EXCEPTIONS

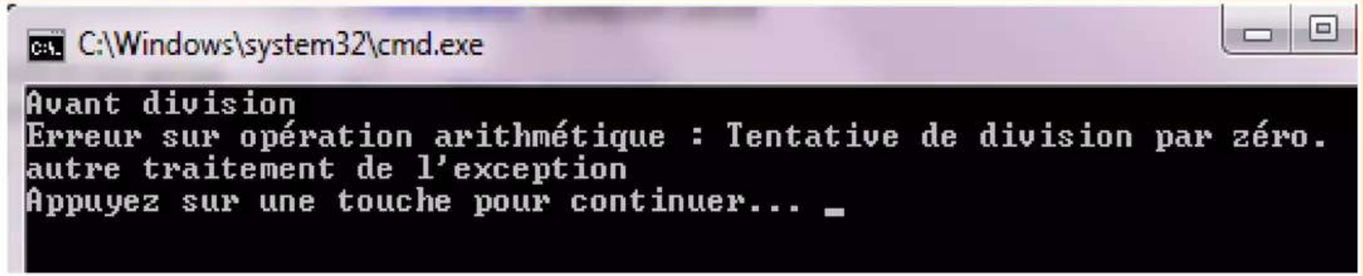
La gestion des erreurs de .NET est basée sur les exceptions, elles permettent de gérer les cas limites d'une méthode.

- Les exceptions contiennent les informations sur l'erreur et son contexte
- On utilise le bloc try...catch pour encapsuler un bout de code à surveiller.
- Disponibilité d'un bloc finally pour la libération des ressources (la mémoire, des données, un fichier de log, etc.)

```
class Program
   {  static void Main(string[] args)
       {  /*Exceptions*/
           int n = 0;
           Console.Write("Taper un nombre :");
           String str=Console.ReadLine();
           try
           {
               n = int.Parse(str); }
           catch
           { Console.WriteLine("Exceprion!");
               n = 0; }
           finally {
               Console.WriteLine("Vous avez tapé
le nombre :" + n);
           }
}
```

# LES EXCEPTIONS

```
using System;
namespace ExempleClasse
{
  class Program {
    static void Main(string[] args){
      int a=10, b=0, c;
      try{
           Console.WriteLine ("Avant division");
           c = a/b;
           Console.WriteLine ("Après division : c vaut " + c);
         }
      catch (ArithmeticException exc)
        { Console.WriteLine("Erreur sur opération arithmétique : "+exc.Message); }
      catch (Exception exc)
        { Console.WriteLine("Exception générale : "+exc.Message); }
      finally
          { Console.WriteLine("autre traitement de l'exception");}
   }
  }
}
```



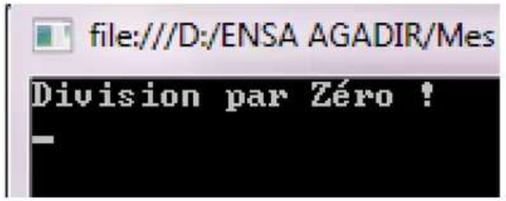
# LES EXCEPTIONS

On peut lever à tout moment une exception grâce au mot-clé **throw**.

```
class Program
    {

        static void Main(string[] args)
        {
            /*Exceptions*/
            int a = 5, b = 0, c;
            try
            {
                if (b != 0)
                    c = a / b;
                else
                    throw new DivideByZeroException("Division par Zéro !");
            }
            catch (Exception e)
            {

                Console.WriteLine(e.Message);
            }
        }
    }
```



# LES EXCEPTIONS

Il est possible de créer des **exceptions personnalisées** en dérivant de la classe de base **Exception**.

```
class MyException : Exception
   {   DateTime dateError;
       static int numError=0;
       public MyException()
           : base("Parametre invalide")
       {   dateError = DateTime.Now;
           numError++;
       }
       public MyException(string message)
           : base(message)
       {
           dateError = DateTime.Now;
           numError++;
       }
       public void PrintError() {
        Console.WriteLine("{0}: Error n°{1}
[{2}].",dateError,numError,base.Message);
       }
   }
```

```
class Program
 {static void Main(string[] args)
   { /*Exceptions*/
     int a = 5, b = 0, c;
     try
     {if (b != 0)
        c = a / b;
      else
        throw new MyException("My Exception : Division par Zéro !");
     }
     catch (MyException e)
       { e.PrintError(); }
  }
}
```

```
08/12/2019 12:11:25: Error n°1 [My Exception : Division par Zéro !].
```

# LES EXCEPTIONS

La gestion des erreurs de .NET est basée sur les exceptions, elles permettent de gérer les cas limites d'une méthode.

- Les exceptions contiennent les informations sur l'erreur et son contexte
- On utilise le bloc **try...catch** pour encapsuler un bout de code à surveiller.
- Disponibilité d'un bloc **finally** pour la libération des ressources (la mémoire, des données, un fichier de log, etc.)
- Il est possible de créer des **exceptions personnalisées** en dérivant de la classe de base **Exception**.
- On peut lever à tout moment une exception grâce au mot-clé **throw**.

# Les structures et le type Enum

# LES STRUCTURES

Les structures sont des types valeur qui sont optimisés par le framework .NET.

- Une structure est un objet qui ressemble beaucoup à une classe, mais qui possède des restrictions.
- Les structures possèdent des propriétés, des méthodes, des constructeurs, etc.
- Il n'est pas possible d'utiliser l'héritage avec les structures.
- Pour créer une structure, nous utiliserons le mot-clé **struct**.

**Étant gérées en mémoire différemment, les structures sont optimisées pour améliorer les performances des petits objets. Comme il n'y a pas de référence, on utilisera directement l'objet sans aller le chercher via sa référence.**

```
using System;
namespace ExempleClasse
{  struct Pers
   {   public string firstName;
       public string lastName;
       public int age;
   }
   class Program
   {
       static void Main(string[] args)
     {
         Pers p;
         p.age = 99;
         p.firstName = "Youness";
         p.lastName = "BOUKOUCHI";
        Console.WriteLine("Mr "+p.firstName+"
"+ p.lastName+ " age :"+p.age);
     }
   }
}
```

# LES STRUCTURES

- Comme nous avons dit, il est impossible qu'une structure hérite d'une autre structure ou d'un objet ; sauf bien sûr du fameux type de base **object**, pour qui c'est automatique.
- Une structure hérite donc des quelques méthodes d'Object (comme ToString()) que nous pouvons éventuellement spécialiser.
- Comme pour les classes, il est possible d'avoir des constructeurs sur une structure, à l'exception du constructeur par défaut qui est interdit.

```
class Program
{ public struct Personne
  { public string Prenom { get; set; }
    public int Age { get; set; }
    public override string ToString()
      { return "Prénom :"+Prenom+" Age:"+Age;}
  }
  static void Main(string[] args)
  {  Personne pers= new Personne () { Prenom = "Youness", Age =25 };
     Console.WriteLine( pers.Prenom+ " a " + pers.Age+ " ans ");
     Console.WriteLine(pers.ToString());
   }
}
```

```
class Program
{ public struct Personne
  {  public string prenom;
     public int age;
     public Personne(String Prenom, int Age)
       {   prenom = Prenom;
           age = Age;  }
     }
     static void Main(string[] args)
      {  Personne pers= new Personne ("Youness",25);
         Console.WriteLine( pers.prenom+ " a " + pers.age+ " ans ");
      }
  }
```

# LES STRUCTURES

Structures et classes présentent néanmoins des différences notables:

- Les variables structurées sont de type «valeur» tandis que les véritables objets sont de type «référence».
- Les structures ne peuvent hériter d'aucune classe ou structure.
- Les champs d'une structure ne peuvent pas être explicitement initialisés dans la définition même du champ.
- Une structure peut contenir zéro, un ou plusieurs constructeurs mais pas de constructeur par défaut.
- Une variable structurée peut être créée par new.

```
struct Pers
  {  public string nom;
     public int age;
     public Pers(String Nom, int Age)
    {  nom = Nom;
       age = Age; }
    public void moi() {
    Console.WriteLine("Mr " + nom + " " + " age :" + age);  }
  }
```

```
class Program
  {  static void Main(string[] args)
     {  Pers p;
        p.age = 99;
        p.nom = "Youness";
        Pers p2 = new Pers("BOUKOUCHI",99);
        p2.moi();
       Console.WriteLine("Mr "+p.nom+" "+ " age :"+p.age);
     }
}
```

# LE TYPE ENUM

- Une variable de type **enum** peut prendre l'une des valeurs listées dans l'énumération.
- Un type **enum** correspond une énumération de valeurs
- Une énumération s'utilise lorsque l'on veut créer un type possédant plusieurs valeurs fixes, comme les jours de la semaine.

```
using System;
namespace ExempleClasse
{
 enum Jours
   {
      Lundi,
      Mardi,
      Mercredi,
      Jeudi,
      Vendredi,
      Samedi,
      Dimanche
   }
   class Program
   {
      static void Main(string[] args)
      {
       Jours lundi = Jours.Lundi;
       Console.WriteLine(lundi);
      }
   }
}
```

# LE TYPE ENUM

- Une énumération est un type dont toutes les valeurs définies sont des entiers. La première vaut 0, et chaque valeur suivante prend la valeur précédente augmentée de 1( par défaut, 0 pour la première valeur et incrémentation de 1 pour chaque valeur suivante).
- Il est également possible d'associer des valeurs bien particulières à chaque valeur de l'énumération.

```
enum EtatCivil
{
    Célibataire=1,
    Marié=4,
    Divorcé=8,
    Veuf=10
}
```

```
using System;
namespace ExempleClasse
{enum EtatCivil
   {  Célibataire,
      Marié,
      Divorcé,
      Veuf }
   class Program
   {static void Main(string[] args)
      {
         EtatCivil ec; // ec est une variable de type EtatCivil
         ec = EtatCivil.Marié;
         ec++;
         switch (ec)
         {
            case EtatCivil.Célibataire:
               Console.WriteLine("Célibataire:"+(int)ec);
break;
            case EtatCivil.Marié:
               Console.WriteLine("Marié"+(int)ec); break;
            case EtatCivil.Divorcé:
               Console.WriteLine("Divorcé"+(int)ec); break;
            case EtatCivil.Veuf:
               Console.WriteLine("Veuf"+(int)ec); break;
         }
      }
   }
}
```

# Les génériques

# LES GÉNÉRIQUES

- Avec les génériques, vous pouvez créer des méthodes ou des classes qui sont indépendantes d'un type. On les appellera des méthodes génériques et des types génériques.
- On utilise **les chevrons <>** pour indiquer le type d'une classe ou d'une méthode générique.
- Le framework .NET possède beaucoup de classes et d'interfaces génériques, notamment dans l'espace de nom **System.Collections.Generic.** (List<>, Queue<>, Dictionary <,>, etc.)

```
class Program
{
    public static void Echanger<T>(ref T t1, ref T t2)
    {
        T temp = t1;
        t1 = t2;
        t2 = temp;
    }
    static void Main(string[] args)
    {
      int i = 5;
      int j = 10 ;
      Echanger(ref i , ref j);
      Console.WriteLine(i + " et " + j);
      String s1 = "Ensa";
      String s2 = "Agadir";
      Echanger(ref s1, ref s2);
      Console.WriteLine(s1 + " et " + s2);
    }
}
```

# LES GÉNÉRIQUES

- Classe générique : Une classe générique fonctionne comme les méthodes, nous pouvons indiquer de 1 à N types génériques.

```
class Program
{
  static void Main(string[] args)
  {
    MaListeGenerique<string> maListe = new MaListeGenerique<string>();
    maListe.Ajouter("BOUKOUCHI");
    maListe.Ajouter("ELYAAKOUBI");
    Console.WriteLine(maListe.ObtenirElement(0));
    Console.WriteLine(maListe.ObtenirElement(1));
  }
}
```

```
public class MaListeGenerique<T>
  {  private int capacite;
     private int nbElements;
     private T[] tableau ;
     public MaListeGenerique()
     {  capacite=10;
        nbElements=0;
        tableau = new T[capacite];
     }
     public void Ajouter(T element)
      {  if(nbElements>= capacite)
         {  capacite*= 2;
            T[] copieTableau=new T[capacite];
            for (int i=0; i<tableau.Length; i++)
              { copieTableau[i]=tableau[i];}
            tableau = copieTableau;
         }
         tableau[nbElements]=element;
         nbElements++;
      }
     public T ObtenirElement(int indice)
     {  return tableau[indice];  }
  }
```

# Les tableaux et les collections

# LES TABLEAUX

**Un tableau:**

- Collection d'éléments de même type
- Accès indicé

la manipulation des tableaux:

- **Length** : indique le nombre d'éléments
- **foreach** : est une boucle spécifique pour les tableaux (collections)

Tableaux statiques –Taille et contenu fixés à la compilation:

Déclaration : type[]identifiant ={énumération des valeurs,…};

```
string[] jours = {"Lundi","Mardi","Mercredi","Jeudi","Vendredi", "Samedi", "Dimanche"};
for(int i = 0; i<jours.Length; i ++)
            {
                Console.WriteLine(jours[i]);
            }
```

# LES TABLEAUX

- Il est possible facilement de faire des opérations sur un tableau, comme un tri. On pourra utiliser la méthode Array.Sort().
- **Tableaux dynamiques –Taille et valeurs définies à l'exécution**
  - Déclaration: **type_de_donnée[] nom_de_variable;**
  - Initialisation: **nom_de_variable = new type_de_donnée [taille];**
  - la méthode Resize() de la classe Array peut redimensionner le tableau
- **Tableaux à multi-dimensions:**
  - type[,,]identifiant =new type[,,];

```
Animal[] tab;
tab=new Animal[2];
tab[0] = new Animal("Pixel", 5, "Noir");
tab[1] = new Animal("Rex", 2, "Noir et blanc");
foreach (var animal in tab)
  { Console.WriteLine(animal.ToString()); }
```

```
string[] jours = new string[] {"Lundi","Mardi","Mercredi","Jeudi","Vendredi", "Samedi", "Dimanche" };
for(int i = 0; i<jours.Length; i ++)
   { Console.WriteLine(jours[i]); }
```

# LES LISTES

- les deux types, tableau et liste, se ressemblent beaucoup.
- Le tableau peut être multidimensionnel.
- le type tableau est de taille fixe alors que la liste est de taille variable.
- On peut ajouter un nouvel élément à une liste grâce à la méthode **Add()**. De même, on peut supprimer des éléments avec la méthode **Remove** alors qu'avec un tableau, on peut seulement remplacer les valeurs existantes.

```
List <string> jours = new List <string>();

jours.Add("Lundi");

jours.Add("Mardi");

jours.Add("Mercredi");

jours.Add("Jeudi");

jours.Add("Vendredi");

jours.Add("Samedi");

jours.Add("Dimanche");

int indice = jours.IndexOf(" Mercredi ");

// indice vaut 2
```

# LES COLLECTIONS

- Le FrameWork .NET fournit des classes de conteneurs dans **System.Collection**s, mais ces classes fournies sont trop abstraites.
- La version 2.0 fournit des conteneurs génériques dans **System.Collections.Generic,** ces classes sont plus orientées structures de données (List, Dictionnary, ...).
- Il existe des versions génériques dans l'espace de noms *System.Collections.Generic* et des versions non génériques dans *System.Collections*.
  - Classe ArrayList,
  - La classe StringCollection,
  - *La classe Stack* (une pile)
  - La classe Queue (pile FIFO)
  - La classe SortedList
  - *La classe Hashtable*
  - La classe StringDictionary
  - La classe NameValueCollection

# LES COLLECTIONS

## using System.Collections.Generic;

- **List<T>** :Tableau dynamique d'objets T, avec ses méthodes Add, AddRange, Remove, Sort, ToArray, etc., maintenant bien connues, comme dans la classe ArrayList.
- **Stack<T>** Pile avec ses méthodes Push, Pop et Peek.
- **SortedList<T>** Liste triée sur clé, avec les méthodes et propriétés de la SortedList vue précédemment.
- **Queue<T>** File (premier entré, premier sorti) avec ses méthodes Enqueue, Dequeue et Peek.
- **LinkedList<T>** File doublement chaînée avec ses méthodes AddAfter, AddBefore, AddFirst, AddLast, Remove ainsi que ses propriétés First et Last.
- **Dictionary<K, V>** Collection de couples clé/valeur (*key* / *value* en anglais).
- **SortedDictionary<K, V>** Même chose mais avec tri sur la clé.

# Les délégués et les évènements

# LES DÉLÉGUÉS

Un délégué (***delegate*** en anglais) est un objet (de la classe **Delegate**) qui permet d'appeler une fonction, qui n'est pas nécessairement toujours la même. Il peut même s'agir d'un appel à une série de fonctions.

```
la Fonction f
la Fonction g et i=5
la Fonction K et i*i=81
Dans fonction anonyme
```

```
class Program
    {   delegate void D1();
        delegate void D2(int i);
        delegate int D3(int i);
        static void f()
        {
            Console.WriteLine("la Fonction f");
        }
        static void g(int i)
        {
            Console.WriteLine("la Fonction g et i="+ i);
        }
        static int k(int i)
        {
          Console.WriteLine("la Fonction K et i*i="+i*i);
            return i * i;
        }
        static void Main(string[] args)
        {
            D1 d1;
            d1 = new D1(f);
            d1();
            D2 d2 = new D2(g);
            d2(5);
            D3 d3 = new D3(k);
            int i = d3(9);
            d1 = delegate { Console.WriteLine("Dans fonction
anonyme"); };
            d1();
        }
}
```

# LES DÉLÉGUÉS

- Un délégué peut faire référence à plusieurs fonctions, et ainsi les exécuter les unes à la suite des autres (on parle alors de ***multicast delegate***).
- Il est possible d'ajouter, ou de retirer, des fonctions à la liste des fonctions référencées par un délégué.
- L'opérateur **-=** retire une fonction de la délégation
- L'opérateur **+=** ajoute une fonction à la délégation

```
class Program
    {   delegate void D1();
        static void f()
        { Console.WriteLine("la Fonction f"); }
        static void n()
        { Console.WriteLine("la Fonction n"); }
  static void Main(string[] args)
   { D1 d1;
     d1 = new D1(f); // d1 contient une référenc
     d1();           //exécution de f
     Console.WriteLine("------1------");
     d1 += new D1(n);// d1 contient une référence à n
     d1();           //exécution de f puis n
     Console.WriteLine("-----2------");
     d1 += new D1(n);// d1 contient une autre référence à n
     d1();           //exécution de f et n puis n
     Console.WriteLine("------3------");
     d1 -= new D1(n);// d1 ne contient pas une référence à n
     d1();           //exécution de f et n
     Console.WriteLine("------4------");
     d1 -= new D1(n); // d1 ne contient pas plus une référence à n
     d1();            //exécution de f
     Console.WriteLine("------5------");
     d1 -= new D1(f);// d1 ne contient plus aucune référence de fonction
     if (d1 != null) d1();//d1 est null
        else Console.WriteLine("Aucune fonction à exécuter");
  }
}
```

```
la Fonction f
------1------
la Fonction f
la Fonction n
-----2------
la Fonction f
la Fonction n
la Fonction n
------3------
la Fonction f
la Fonction n
------4------
la Fonction f
------5------
Aucune fonction à exécuter
```

# LES DÉLÉGUÉS

- Simplification d'écriture à partir de C# 2:

```
D1 d1;

d1 = f; // d1 = new D1(f);

d1();
```

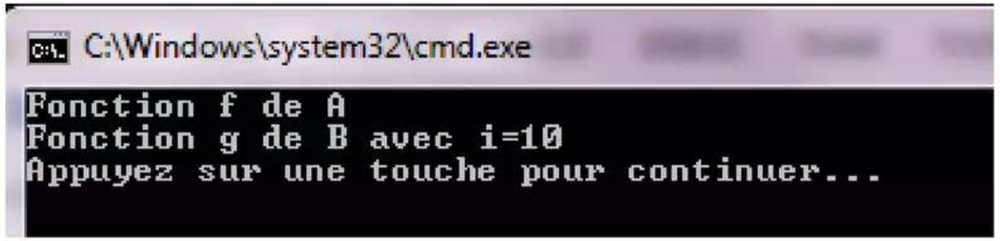
```
class Program
    {   delegate void D1();
        static void f()
        { Console.WriteLine("la Fonction f"); }
        static void n()
        { Console.WriteLine("la Fonction n"); }
  static void Main(string[] args)
   {D1 d1;
    d1 = f; // d1 contient une référence à f
    d1();           //exécution de f
    Console.WriteLine("------1------");
    d1 += n;// d1 contient une référence à n
    d1();           //exécution de f puis n
    Console.WriteLine("-----2------");
    d1 += n;// d1 contient une autre référence à n
    d1();           //exécution de f et n puis n
    Console.WriteLine("------3------");
    d1 -= n;// d1 ne contient pas une référence à n
    d1();           //exécution de f et n
    Console.WriteLine("------4------");
    d1 -= n; // d1 ne contient pas plus une référence à n
    d1();            //exécution de f
    Console.WriteLine("------5------");
    d1 -= f;// d1 ne contient plus aucune référence de fonction
    if (d1 != null) d1();//d1 est null
       else Console.WriteLine("Aucune fonction à exécuter");
  }
}
```

```
la Fonction f
------1------
la Fonction f
la Fonction n
-----2------
la Fonction f
la Fonction n
la Fonction n
------3------
la Fonction f
la Fonction n
------4------
la Fonction f
------5------
Aucune fonction à exécuter
```

# LES DÉLÉGUÉS

Le délégué fait partie d'une classe distincte de celle du programme

```
using System;
namespace ExempleClasse
{
    class A
    { public void f() {
       Console.WriteLine("Fonction f de A"); }
      public delegate void T();
    }
    class B
    { public void g(int i) {
       Console.WriteLine("Fonction g de B avec i=" +i);}
      public delegate void T(int i );
    }
    class Program
    {
        static void Main(string[] args)
        {
            A a = new A();
            A.T d1 = new A.T(a.f);
            d1();
            B b = new B();
            B.T d2 = new B.T(b.g);
            d2(10);
        }
    }
}
```



# LES DÉLÉGUÉS

```
class CArithmetique
    {   public int somme(int i, int j) { return i + j; }
        public int soustraction(int i, int j) { return i - j; }
        public int multiplication(int i, int j) { return i * j; }
        delegate int operation(int i, int j);
        // Tableau d'opérations
        operation[] ops;
        public CArithmetique()
        {   ops= new operation[3];
            ops[0]=new operation(somme);
            ops[1] = new operation(soustraction);
            ops[2] = new operation(multiplication);
        }
        public int calcule(int i, int j, int op)
        {
            return ops[op](i, j);
        }
    }
```

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

```
class Program
    {
        static void Main(string[] args)
        {   CArithmetique op = new CArithmetique();
            Console.WriteLine("{0} + {1} = {2} ", 10, 5, op.calcule(10, 5, 0));
            Console.WriteLine("{0} - {1} = {2} ", 10, 5, op.calcule(10, 5, 1));
            Console.WriteLine("{0} * {1} = {2} ", 10, 5, op.calcule(10, 5, 2));
        }
}
```

# LES ÉVÉNEMENTS

- Les événements sont un mécanisme du C# permettant à une classe d'être notifiée d'un changement (un clic de souris, une modification de prix, etc.).
- On peut voir les événements comme une forme particulière de délégation. Un événement peut déléguer un traitement à une ou plusieurs fonctions (comme pour une délégation).
- Mais seul l'objet propriétaire de l'événement peut déclencher l'événement, c'est-à-dire appeler les fonctions de la délégation. Par contre, n'importe quel utilisateur peut ***abonner/désabonner*** une fonction à l'événement, c'est-à-dire ajouter ou retirer une fonction à la liste de délégation.
- La base des événements est le délégué. ***On pourra stocker dans un événement un ou plusieurs délégués qui pointent vers des méthodes respectant la signature de l'événement.***
- Un événement est défini grâce au mot-clé **event**.

# LES ÉVÉNEMENTS

- Dans l'exemple suivant, la classe Emetteur est propriétaire d'un événement appelé EmetteurEvent.
- Cet événement est déclenchable seulement par l'objet Emetteur.
- Lorsque l'événement est déclenché, les abonnés à cet événement sont appelés.
- A noter, un événement est toujours déclaré à partir d'un type de délégation.
- Le type de délégation décrit la signature des méthodes pouvant s'abonner à l'événement.

```
// classe propriétaire de l'événement EmetteurEvent
class Emetteur
{ // signature des abonnés à l'événement
  public delegate void CallBack();
  public event CallBack EmetteurEvent; // événement
  public void DeclencheEvenement()
  {// appel de tous les délégués (fonctions abonnées).
     if (EmetteurEvent != null) EmetteurEvent();
  }
}
```

```
 class Program
 {
   public static void CB1() { Console.WriteLine("Call
Back 1"); }
   public static void CB2() { Console.WriteLine("Call
Back 2"); }

   static void Main(string[] args)
   {  Emetteur emetteur = new Emetteur();
      emetteur.EmetteurEvent += CB1;
      emetteur.EmetteurEvent += CB2;
      // interdit : emetteur.EmetteurEvent();
      emetteur.DeclencheEvenement();
   }
}
```

```
Call Back 1
Call Back 2
```

# LES ÉVÉNEMENTS

Exemple: un client s'abonne au notification de changement de prix d'une voiture.

```
public class Voiture {
  public delegate void DelegateDeChangementDePrix(double nouveauPrix);
  public event DelegateDeChangementDePrix ChangementDePrix;
  public double Prix { get; set; }
  public void PromoSurLePrix(double reduction)
   {  Prix = Prix *reduction;
      if (ChangementDePrix != null) ChangementDePrix(Prix);
    }
}
```

```
public class Client {
 public string Nom { get; set; }
 public void voiture_ChangementDePrix(double nouveauPrix)
  {
  Console.WriteLine("Client {0} : Le nouveau prix est de :{1} ",
Nom, nouveauPrix);
   }
}
```

```
class Program {
  static void Main(string[] args) {
      Voiture voiture = new Voiture { Prix = 10000 };
      Client client1 = new Client() { Nom="Youness"};
      Client client2 = new Client() { Nom = "Boukouchi" };
      voiture.ChangementDePrix += client1.voiture_ChangementDePrix;
      voiture.PromoSurLePrix(0.8);
      voiture.ChangementDePrix += client2.voiture_ChangementDePrix;
      voiture.PromoSurLePrix(0.7);     }
}
```

```
Client Youness : Le nouveau prix est de :8000
Client Youness : Le nouveau prix est de :5600
Client Boukouchi : Le nouveau prix est de :5600
```

# LES ÉVÉNEMENTS

- Les événements sont beaucoup utilisés dans les applications en C# (**notamment dans les classes System.Windows.Forms**), par exemple, lorsque on clique sur un bouton, un événement est levé.
- Ces événements utilisent en général une construction à base du délégué **EventHandler** standard.
- Ces événements ont généralement une signature (définie par une EventHandler) assez standard qui accepte deux paramètres.
  - Le premier est une référence à l'objet qui représente la source de l'événement, c'est-à-dire l'objet qui a déclenché l'événement.
  - Le second est une référence à l'objet de la classe qui dérive de la classe de base **EventArgs,** censé transporter des données associées à l'événement

# LES ÉVÉNEMENTS

- Les événements sont beaucoup utilisés dans les applications en C# (**notamment dans les classes System.Windows.Forms**), par exemple, lorsque on clique sur un bouton, un événement est levé.
- Ces événements utilisent en général une construction à base du délégué **EventHandler** standard.
- Ces événements ont généralement une signature (définie par une EventHandler) assez standard qui accepte deux paramètres.
  - Le premier est une référence à l'objet qui représente la source de l'événement, c'est-à-dire l'objet qui a déclenché l'événement.
  - Le second est une référence à l'objet de la classe qui dérive de la classe de base **EventArgs,** censé transporter des données associées à l'événement

# LES ÉVÉNEMENTS

L'exemple précédent avec ce nouveau **handler**.

```
public class PrixPromoEventArgs : EventArgs
  {
     public double prixPromo;
     public PrixPromoEventArgs(double prix)
     {prixPromo = prix; }
  }
```

```
public class Voiture
  {
  //public delegate void DelegateDeChangementDePrix(double nouveauPrix);
     public event EventHandler ChangementDePrix;
     public double Prix { get; set; }
     public void PromoSurLePrix(double reduction)
     {
        Prix = Prix * reduction;
        if (ChangementDePrix != null)
          ChangementDePrix(this, new PrixPromoEventArgs(Prix));
     }
  }
```

```
class Program {
static void Main(string[] args) {
    Voiture voiture = new Voiture { Prix = 10000 };
    Client client1 = new Client() { Nom="Youness"};
    Client client2 = new Client() { Nom = "Boukouchi" };
    voiture.ChangementDePrix += client1.voiture_ChangementDePrix;
    voiture.PromoSurLePrix(0.8);
    voiture.ChangementDePrix += client2.voiture_ChangementDePrix;
    voiture.PromoSurLePrix(0.7);
}
}
```

```
public class Client
 { public string Nom { get; set; }
   public void voiture_ChangementDePrix(Object sender,
EventArgs e)
    { Console.WriteLine("Client {0} : Le nouveau prix est de :{1} ",
Nom, ((PrixPromoEventArgs)e).prixPromo);
    }
}
```

```
Client Youness : Le nouveau prix est de :8000
Client Youness : Le nouveau prix est de :5600
Client Boukouchi : Le nouveau prix est de :5600
```

# LES ÉVÉNEMENTS

```
SAMU: Appel de secours urgents de Ali Ahlam à ENSA, CP 8000,Agadir,Maroc
Police: Appel d'urgence de Ali Ahlam à ENSA, CP 8000,Agadir,Maroc
SAMU: Appel de secours urgents de Amal Akram à EST, CP 80150, Agadir, Maroc
Police: Appel d'urgence de Amal Akram à EST, CP 80150, Agadir, Maroc
```

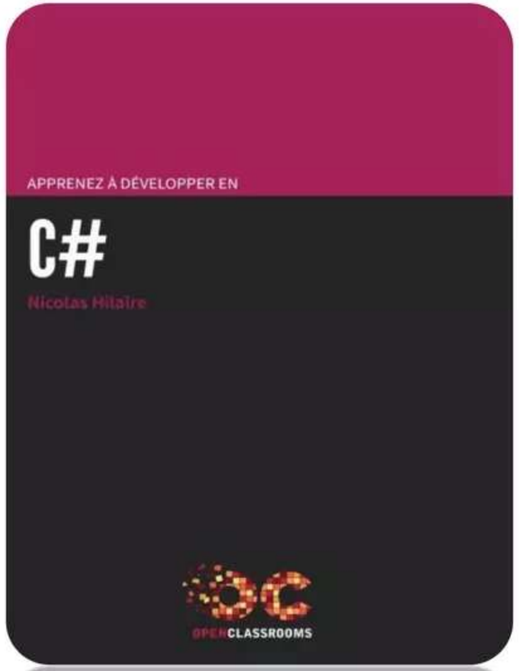
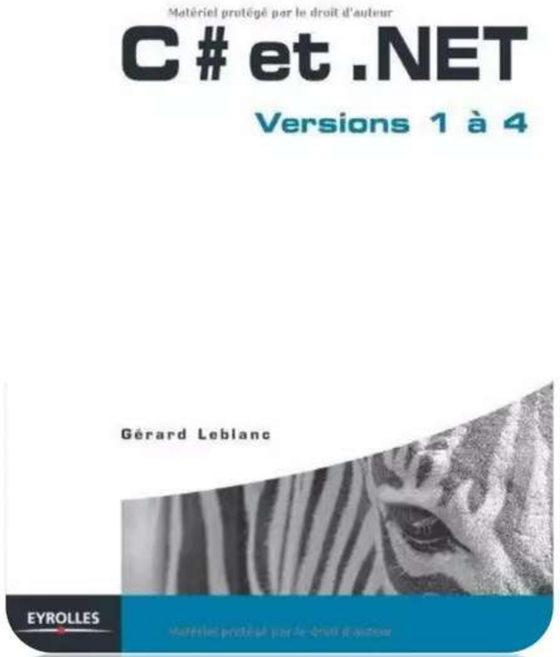
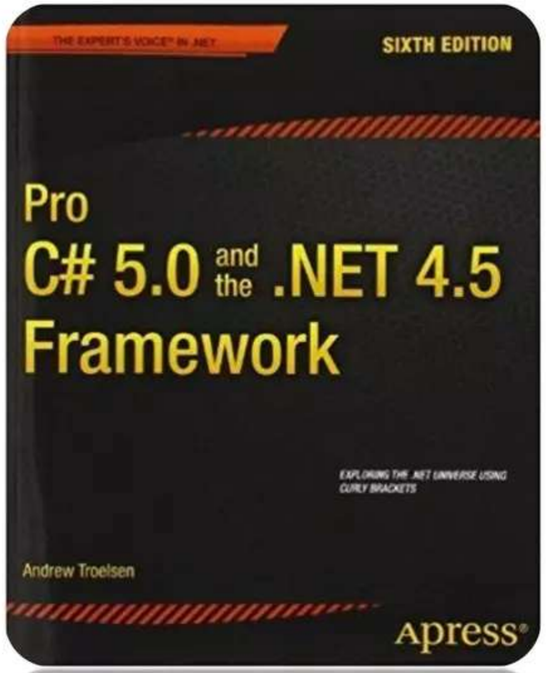
Exemple d'un objet de type **event** qui peut faire référence à plusieurs fonctions de traitement d'événements.

```
class AccidentEventArgs:EventArgs
  {  public string adresse;
     public string responsable;
     public AccidentEventArgs(string adr, string _responsable )
     {  adresse = adr;
        responsable = _responsable;    }
}
```

```
public class Surveillant {
 public event EventHandler accident;
 public string Nom { get; set; }
 public void Alerter(String adr)
   { if (accident != null) accident(this, new AccidentEventArgs(adr, Nom)); }
}
```

```
class Program {
 static void Main(string[] args) {
 Surveillant surveillantENSA = new Surveillant() { Nom="Ali Ahlam"};
 Surveillant surveillantEST = new Surveillant() { Nom = "Amal Akram" };
 SAMU samu = new SAMU();
 Police police = new Police();
 surveillantENSA.accident += samu.appelSecoursUrgents;
 surveillantENSA.accident += police.appelAccident;
 surveillantEST.accident += samu.appelSecoursUrgents;
 surveillantEST.accident += police.appelAccident;
 surveillantENSA.Alerter(" ENSA, CP 8000,Agadir,Maroc");
 surveillantEST.Alerter("EST, CP 80150, Agadir, Maroc ");
 }
}
```

```
public class SAMU {
 public void appelSecoursUrgents(Object sender, EventArgs e)
   { AccidentEventArgs ev = (AccidentEventArgs)e;
     Console.WriteLine("SAMU: Appel de secours urgents de {0} à
{1}",ev.responsable ,ev.adresse);  }
 }
```

```
public class POLICE {
public void appelAccident(Object sender, EventArgs e)
  {  AccidentEventArgs ev = (AccidentEventArgs)e;
     Console.WriteLine("Police: Appel d'urgence de {0} à {1}",
ev.responsable, ev.adresse); }
}
```

# LA MAÎTRISE D'UN LANGAGE PASSE TOUJOURS PAR UNE **PRATIQUE IMPORTANTE.**