

# C#, DÉVELOPPER EN .NET AVEC VISUAL STUDIO

INTRODUCTION À MICROSOFT .NET



# SOMMAIRE JOUR 1

## MODULE 1 Introduction à Microsoft .NET

- Leçon 1 Présentation Générale et tour d'horizon
- Leçon 2 Historique et aperçu des types d'application
- Leçon 3 Structure d'une application .NET

## MODULE 2 Prise en main de Visual Studio

- Leçon 1 Les solutions dans Visual studio
- Leçon 2 Les projets dans Visual studio
- Leçon 3 Les propriétés
- Leçon 4 Génération et débogage

## MODULE 3 Les bases du langage

- Leçon 1 premiers pas
- Leçon 2 Conventions de nommage
- Leçon 3 Les types et les variables
- Leçon 4 conversion de type

# SOMMAIRE JOUR 2

## Module 4 Les bases du langage Part 2

- Leçon 1 Les opérateurs
- Leçon 2 Les boucles
- Leçon 3 Les conditions
- Leçon 4 Passage d'arguments aux méthodes
- Leçon 5 Directives de compilation et attributs

## Module 5 Les collections

- Leçon 1 Les tableaux
- Leçon 2 Les enumérateurs
- Leçon 3 Les collections génériques
- Leçon 4 Utilisation des flux

## Module 6 Gestion d'erreurs et Les évènements

- Leçon 1 Les exceptions
- Leçon 2 Les délégués
- Leçon 3 Les évènements

# SOMMAIRE JOUR 3

## Module 7 La programmation orienté objet

- Leçon 1 La Programmation procédurale
- Leçon 2 La programmation Orienté objet (POO)
- Leçon 3 Les classes, représentation du monde objet
- Leçon 4 La notion d'héritage
- Leçon 5 Les modificateurs d'accès
- Leçon 6 Interfaces et classes abstraites

## Module 8 La programmation OO avec c#

- Leçon 1 Composition d'une classe
- Leçon 2 Les méthodes
- Leçon 3 Utilisation
- Leçon 4 L'héritage
- Leçon 5 Les classes abstraites
- Leçon 6 L'encapsulation en csharp
- Leçon 7 Types de classes

## Module 9 Autres éléments du langage

- Leçon 1 Les méthodes d'extension
- Leçon 2 Les paramètres nommés et optionnels
- Leçon 3 Le typage dynamique

# SOMMAIRE JOUR 4

## Module 10 Accès aux données

Leçon 1 ADO.NET

Leçon 2 Lecture des données

Leçon 3 Mise à jour des données

Leçon 4 Introduction à LINQ

Leçon 5 Entity Framework

## Module 11 Les frameworks client lourd

Leçon 1 Introduction à WinForm

Leçon 2 Le rendu et la disposition en WPF

Leçon 3 Les contrôles en WPF

Leçon 4 Les ressources

## MODULE 12 Les frameworks web

Leçon 1 Présentation d'ASP.NET MVC

Leçon 2 Les contrôleurs

Leçon 3 Les vues

# SOMMAIRE JOUR 5

## MODULE 13 .NET Core

- Leçon 1 Introduction
- Leçon 2 Démarrage
- Leçon 3 ConfigureServices

## MODULE 14 Injection de dépendances

- Leçon 1 Introduction
- Leçon 2 Durées de vie du service

## PRÉSENTATIONS ET TOUR DE TABLE



# **MODULE 1**

INTRODUCTION À MICROSOFT .NET



# LEÇON 1

HISTORIQUE ET APERÇU DES TYPES D'APPLICATION

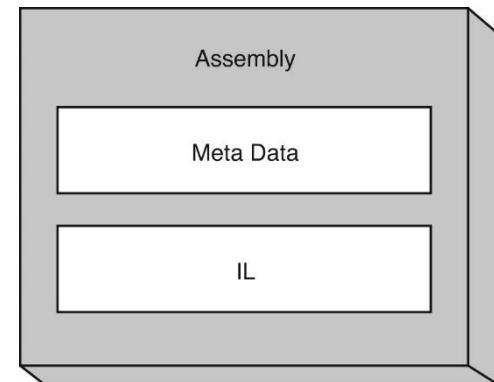
## FRAMEWORK .NET

- Infrastructure de développement basé sur CLI
- Indépendant du language de programmation
- Common Language Infrastructure (CLI)
  - Spécification ouverte de Microsoft
  - Décrit l'environnement d'exécution basé sur CIL
  - > Commun Language Runtime
    - La CTS (*Common type system*)
    - La CLS (*Common Langage Spécification*)
- Environ 30 langages actifs compatibles .NET (C++/CIL, VB.net, F#, Ironpython, Ironruby)

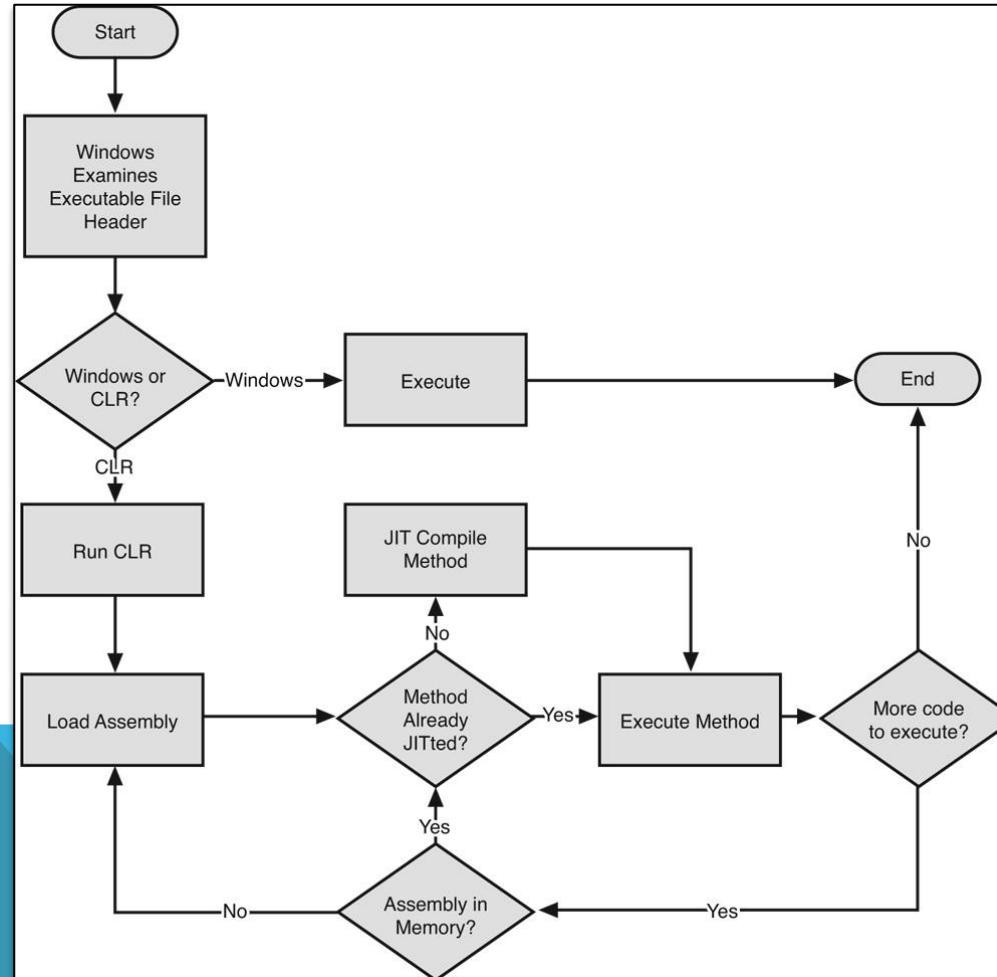
# LE RÔLE DE LA CLR

## Le rôle de la CLR

- sécurité
- Interopérabilité
- Gestion de la mémoire (Garbage Collector)
- BCL (Base Class Library)
- Exécution du code
- Gestion des métadonnées (Réflexion)



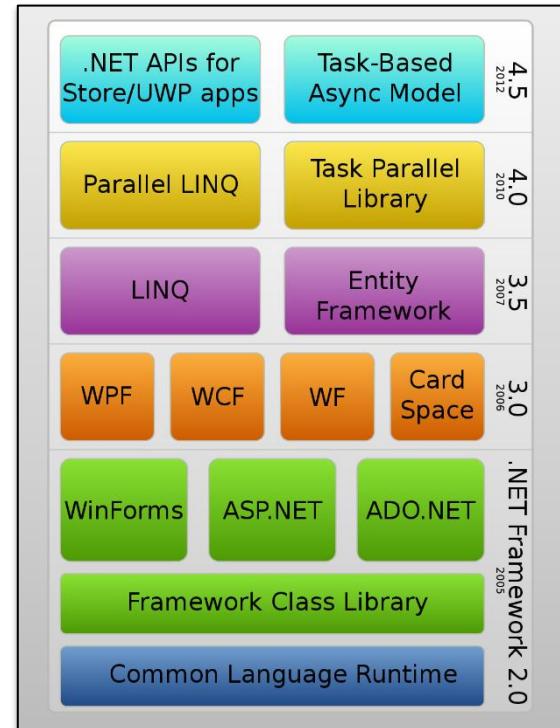
# PRINCIPE D'EXÉCUTION DE LA CLR



# LA PLATEFORME .NET

## Historique du Framework

Version	CLR	Sortie	Livré avec Visual Studio	Préinstallé avec Windows		Comprend
				Client	Serveur	
1.0	1.0	13 février 2002	.NET 2002	NC	NC	
1.1	1.1	24 avril 2003	.NET 2003	NC	2003	
2.0		7 novembre 2005	2005	NC	2003 R2	
3.0		6 novembre 2006	NC	Vista	NC	2.0
3.5	2.0	19 novembre 2007	2008	NC	NC	3.0 SP1 (2.0 SP1)
		4 février 2008	NC	NC	2008	
		11 août 2008	2008 SP1	NC	NC	3.0 SP2 (2.0 SP2)
		22 juillet 2009	NC	7	2008 R2	
4	4.0	12 avril 2010	2010	NC	NC	NC (mise à jour sur place)
		15 août 2012	2012	8	2012	
		17 octobre 2013	2013	8.1	2012 R2	
		5 mai 2014	NC	NC	NC	
		20 juillet 2015	2015	10 v1507	NC	
		17 novembre 2015	2015 U1	10 v1511	NC	
		2 août 2016	NC	10 v1607	2016	
		5 avril 2017	2017 v15.3	10 v1703	NC	
		17 octobre 2017	2017 v15.5	10 v1709	v1709	
		30 avril 2018	NC	10 v1803	v1803	
		18 avril 2019	NC	10 v1903	2019	



En parallèle, Xamarin a créé le projet Mono iso fonctionnel aux versions de .NET  
MICROSOFT a créé .NET Standard dont l'implémentation est .NET Core

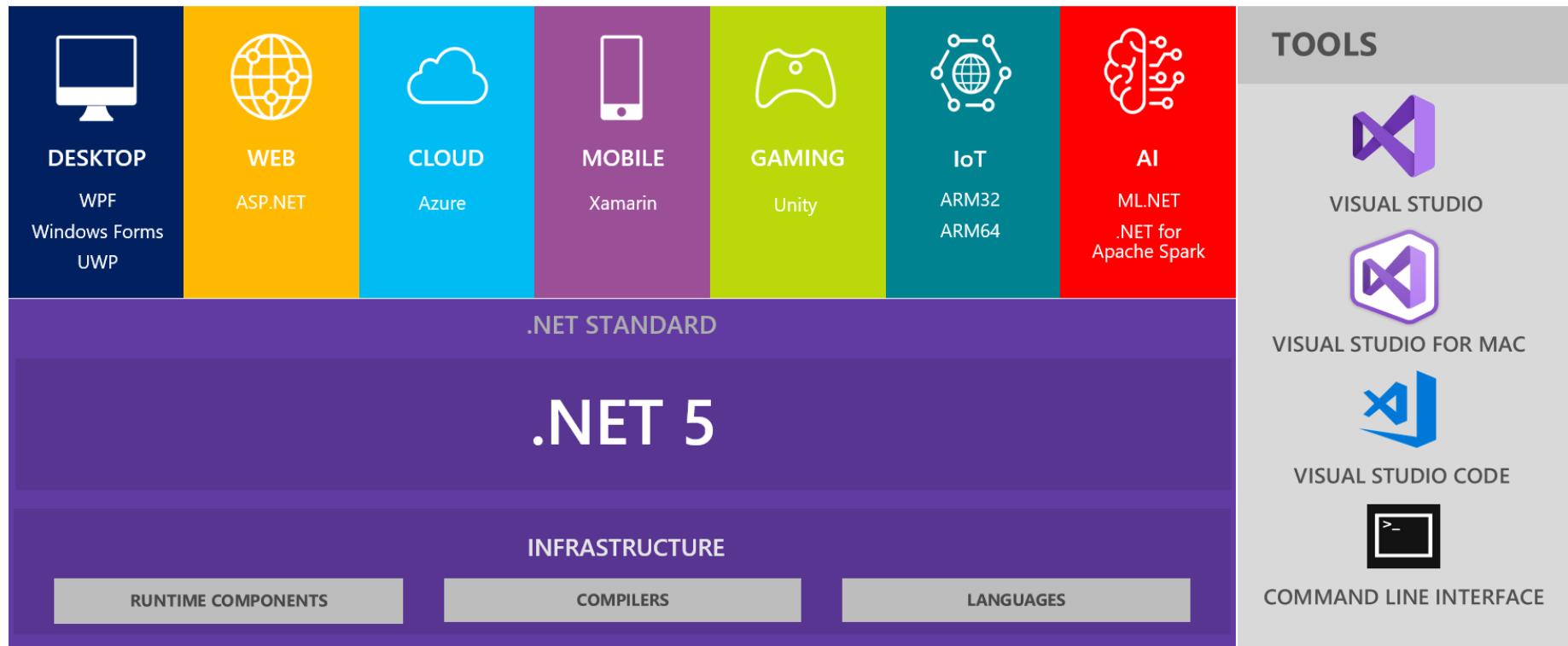
# ETAT DES LIEUX

<b>ASP.Net 4.x</b>	<b>Desktop</b>	<b>ASP.Net Core 1.x</b>
Web Form MVC WebAPI	Windows Form WPF	MVC Core WebAPI Entity Framework Core
<b>C#/VB Compiler</b>		<b>C#/VB Compiler(Roslyn)</b>
<b>.Net Framework</b>		<b>.Net Core</b>
<b>FCL/BCL</b> ( <i>System.Web, System.Data, System.Xml</i> ) Net Framework Class Library		<b>CoreFX</b> ( <i>System.Collections, System.IO, System.Xml</i> ) Net Core Class Library, NuGetPackages
<b>Common Language Runtime(CLR)</b> <i>JIT Compiler/GC/CAS</i>		<b>Common Language Runtime(CoreCLR)</b> <i>RyuJIT Compiler(for 64-bit systems)/GC/SIMD</i>
Operating System 		Operating System   

# UNIFICATION AVEC .NET5

@Rémy => ajout .Net 6 ?

# .NET – A unified platform



QUESTIONS ? REMARQUES ?



# LEÇON 3

STRUCTURE D'UNE APPLICATION .NET

## LES DIFFÉRENTS TYPES DE LANGAGES

- C++, Pascal, C sont des langages natifs
  - > code machine, très rapide mais spécifique à une plateforme (intel, arduino, etc ...)
- JavaScript, VBA.. sont des langages interprétés
  - > nécessite un interpréteur pour lire le code. Le code est exécuté au fur et à mesure que c'est lu. Code plutôt lent
- Les langages .NET et Java sont managés
  - > mix entre natif et interprété = précompilé puis interprété. Code presque aussi rapide que natif

## COMPARAISON DES LANGAGES

**VB.NET = Le langage historique de Microsoft**

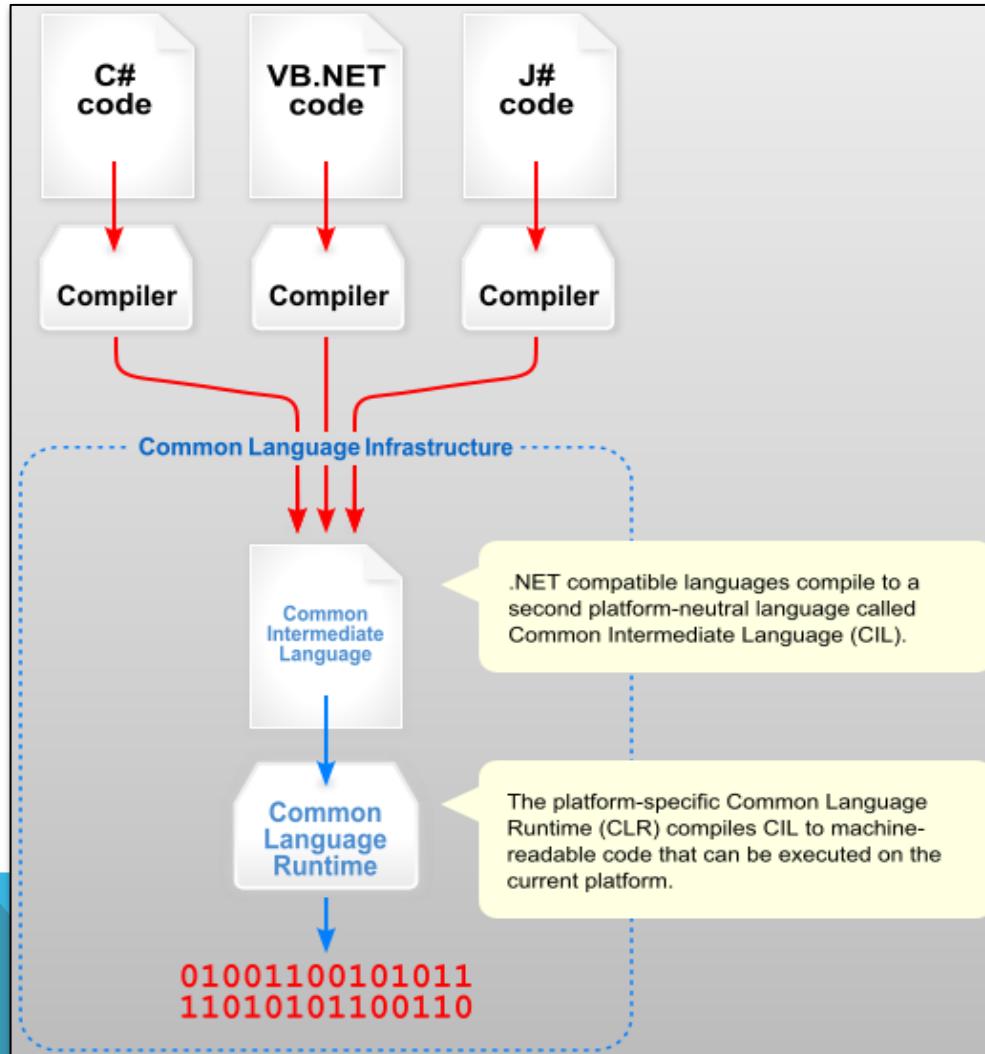
- Très verbeux
- Insensible à la casse
- Prise en main facile pour les utilisateurs de VB6 et ou VBA
- Moins « productif » que C#
  - Tout est explicite, plus de code à écrire
- Certaines classes ont été faites spécifiquement pour VB (même si elles peuvent être utilisées en c#)
- Moins sécurisé que C#
  - Plus laxiste (déclaration non typé)

## COMPARAISON DES LANGAGES

### C# *Le langage phare de Microsoft*

- Même syntaxe que Java ou C++
- Langage le plus populaire de la plateforme .NET
- Langage jeune donc non soumis aux problème de rétrocompatibilité lors de sa création
- Productivité maximale car peu verbeux
- Plus difficile à apprécier
  - Utilisation d'une notation moins transparente
- Très sécurisé
  - Tout est fait pour éviter les erreurs

# FINALEMENT, C# OU VB.NET ?



## FINALEMENT, C# OU VB.NET ?

- 2 langages respectant le CLS
- Compilation du code en CIL
- Utilisation de la CLR pour la compilation à la volé.
- VB.NET et C# utilise .NET et ne sont donc pas limité en terme de fonctionnalités.

QUESTIONS ? REMARQUES ?



# **MODULE 2**

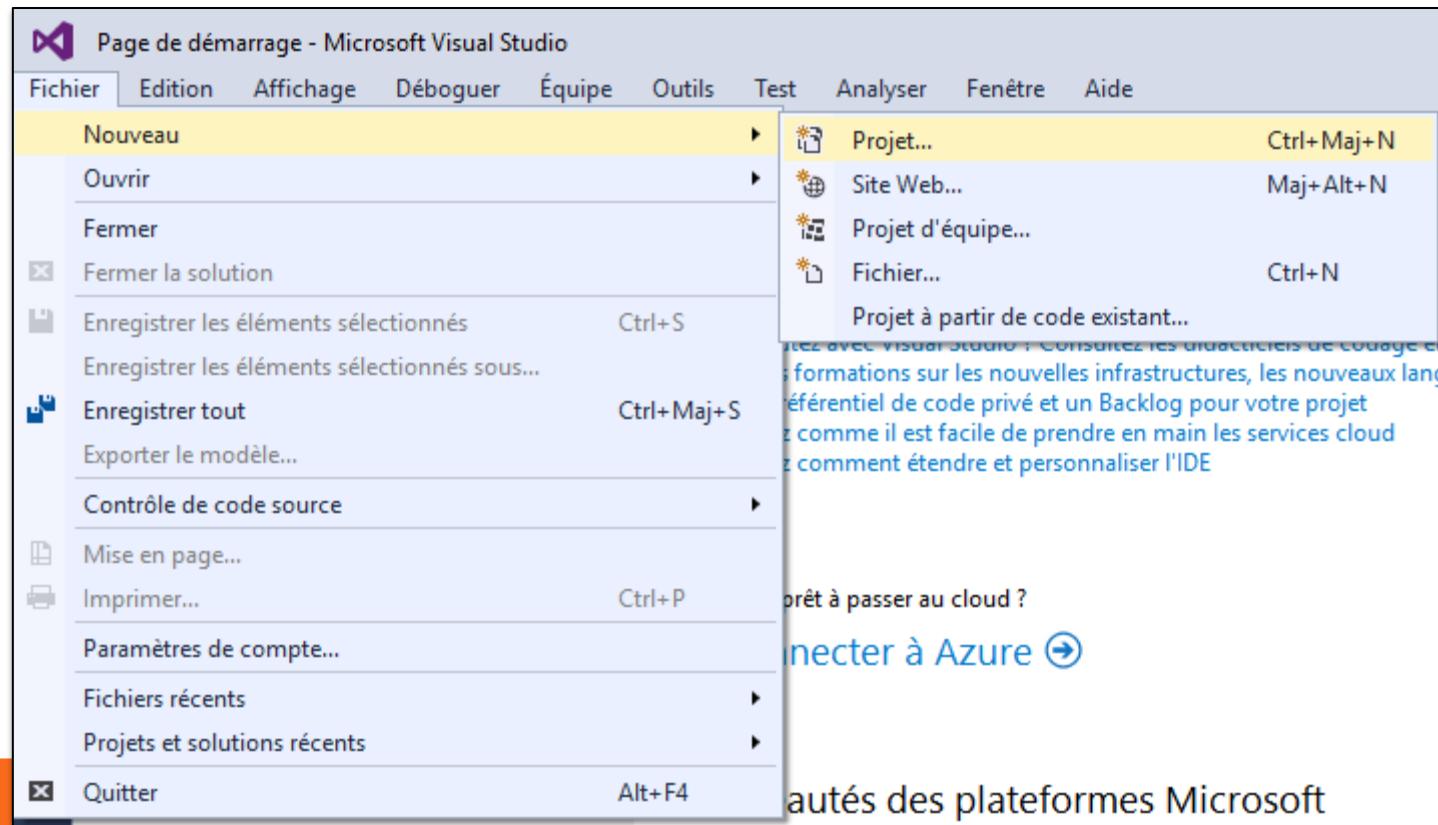
PRISE EN MAIN DE VISUAL STUDIO



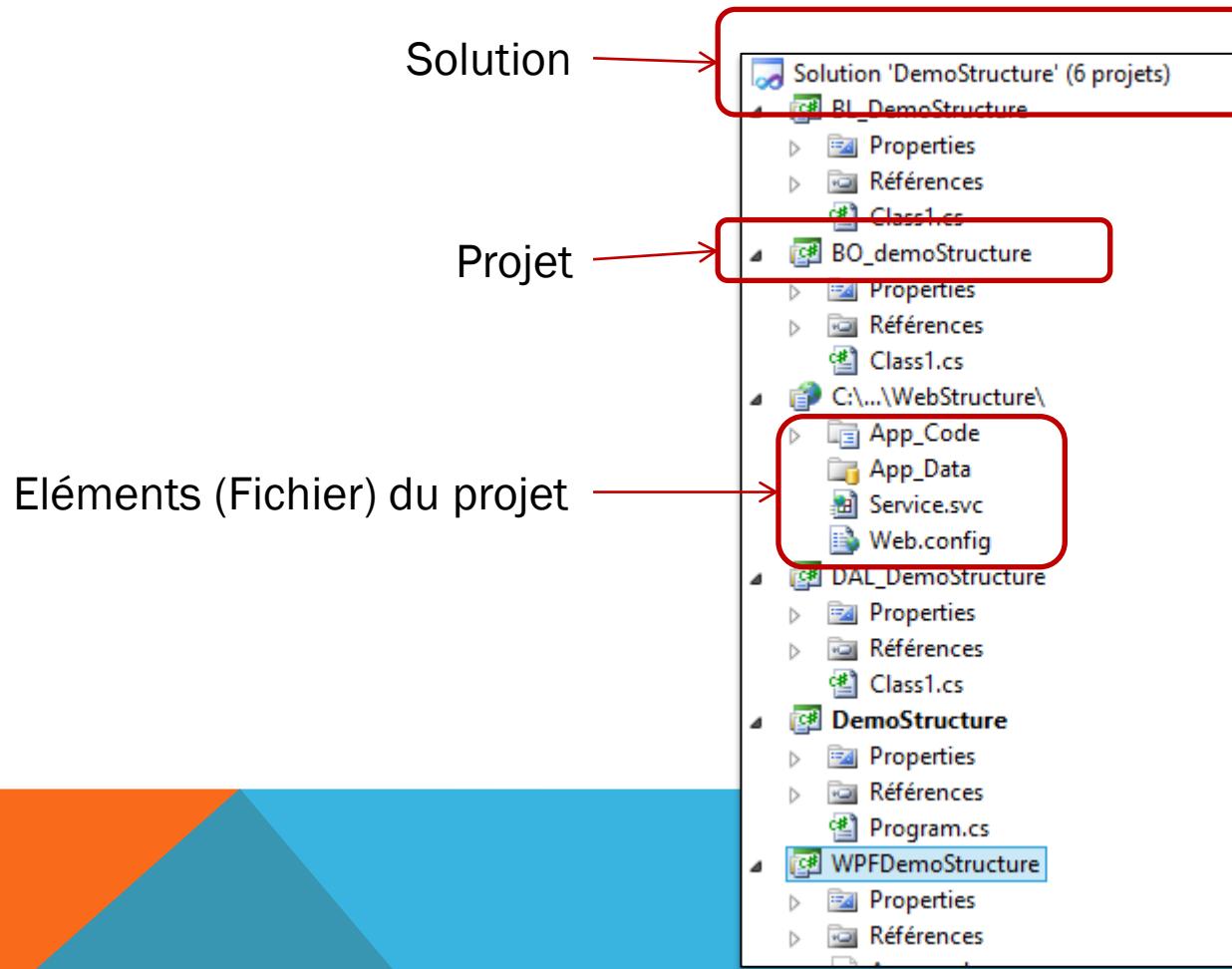
# LEÇON 1

LES SOLUTIONS DANS VISUAL STUDIO

# CRÉATION ET OUVERTURE D'UNE SOLUTION



# STRUCTURE D'UNE SOLUTION



# DÉMONSTRATION



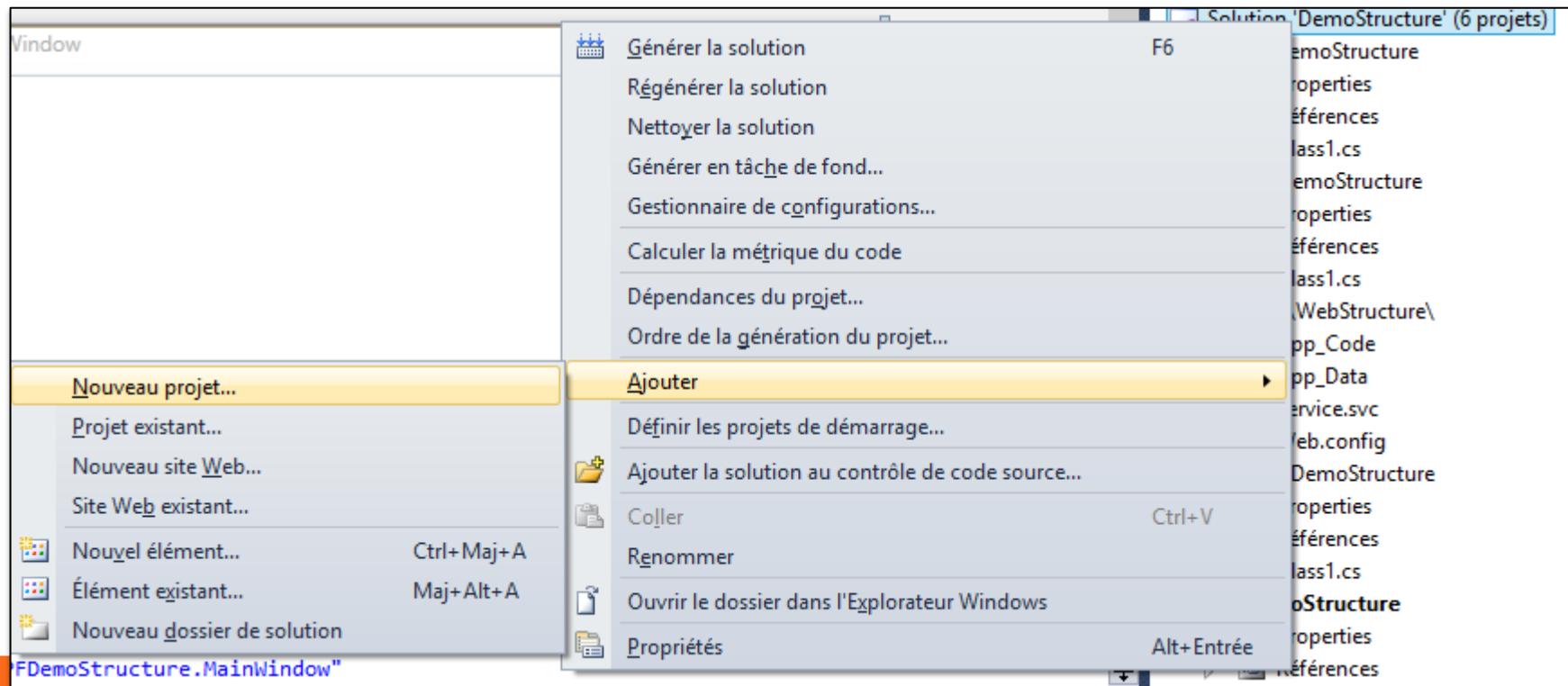
QUESTIONS ? REMARQUES ?



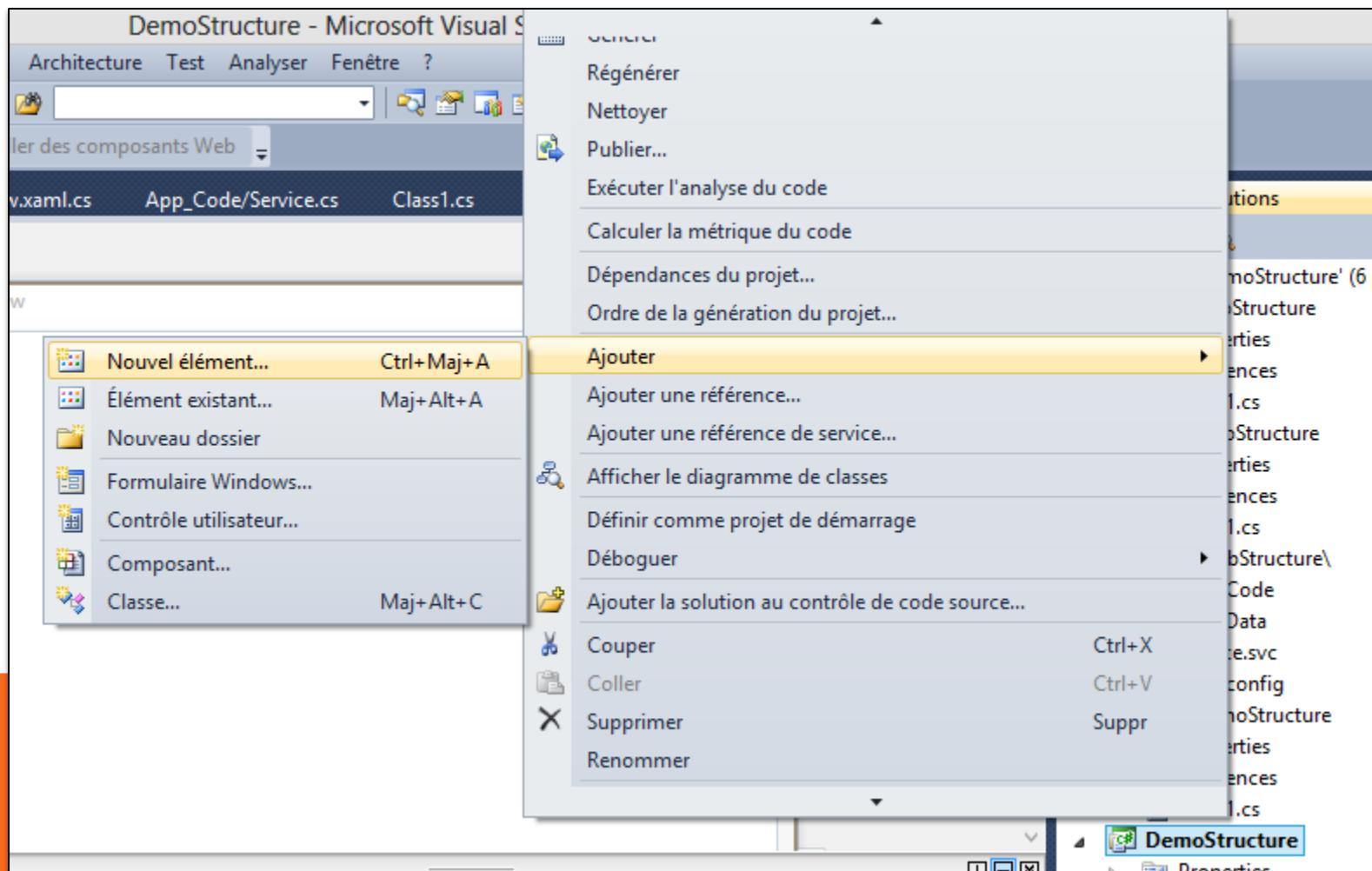
# LEÇON 2

LES PROJETS DANS VISUAL STUDIO

# AJOUT D'UN PROJET À LA SOLUTION



# AJOUT D'UN ÉLÉMENT AU PROJET



# DÉMONSTRATION



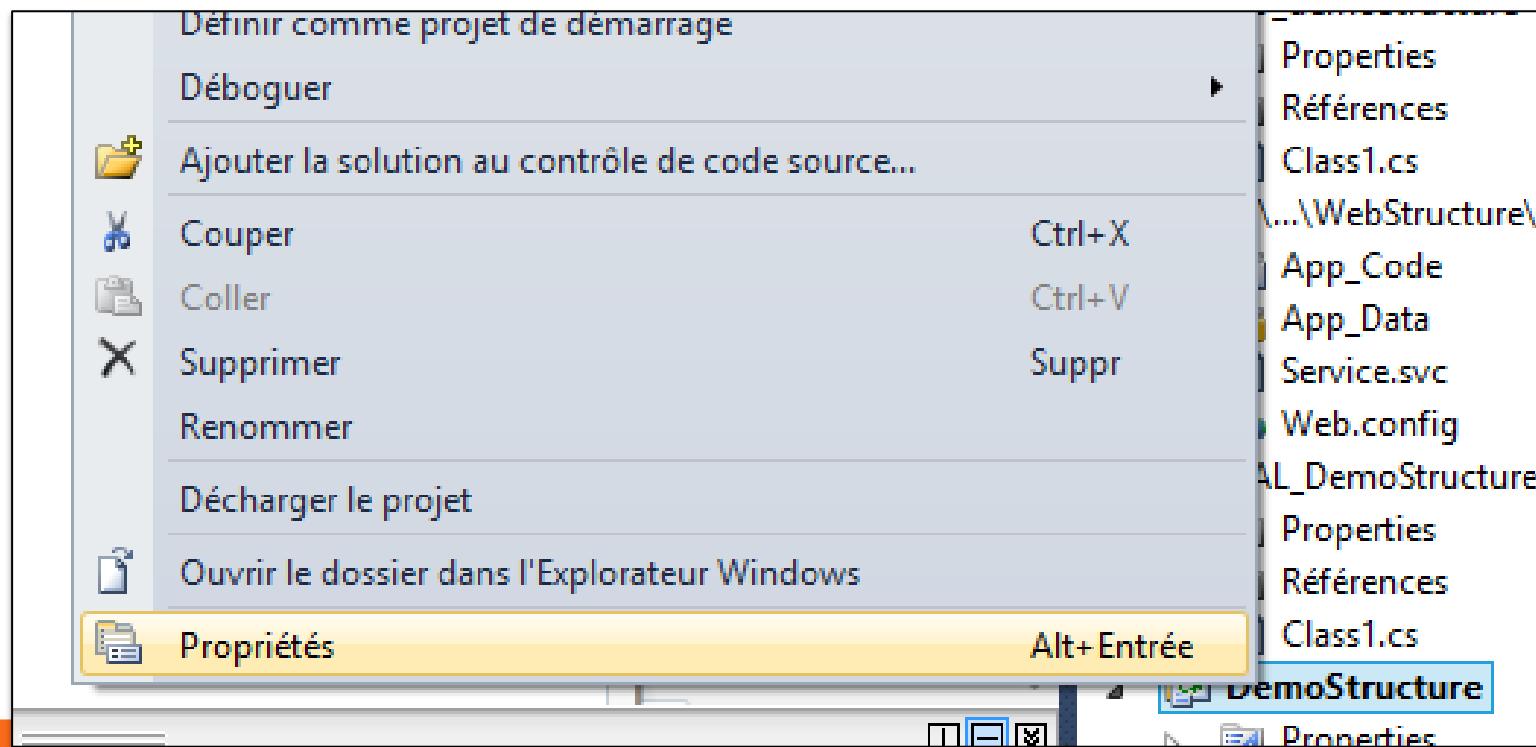
QUESTIONS ? REMARQUES ?



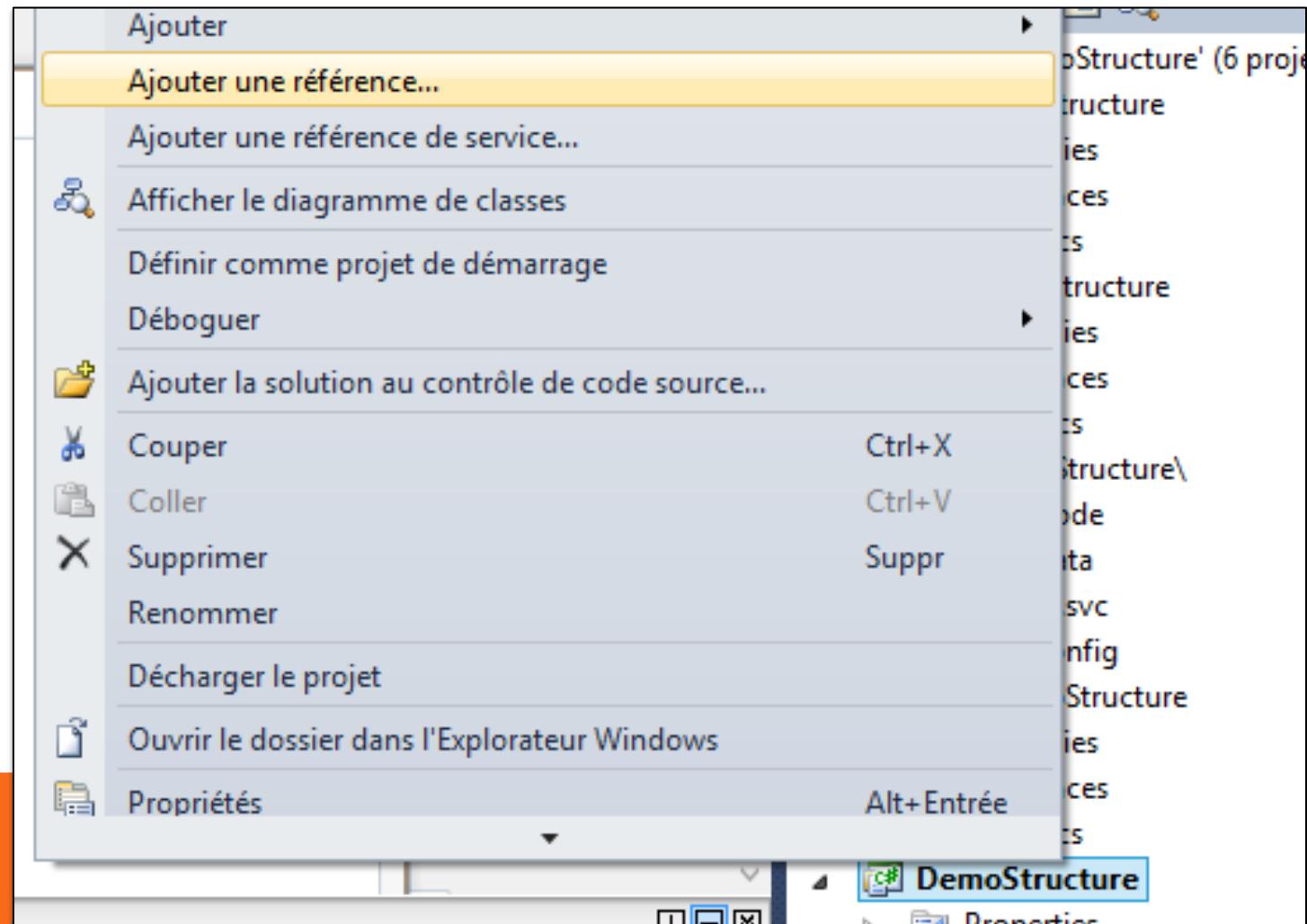
# LEÇON 3

LES PROPRIÉTÉS

# PROPRIÉTÉ D'UN PROJET



## AJOUT DE RÉFÉRENCES



# DÉMONSTRATION



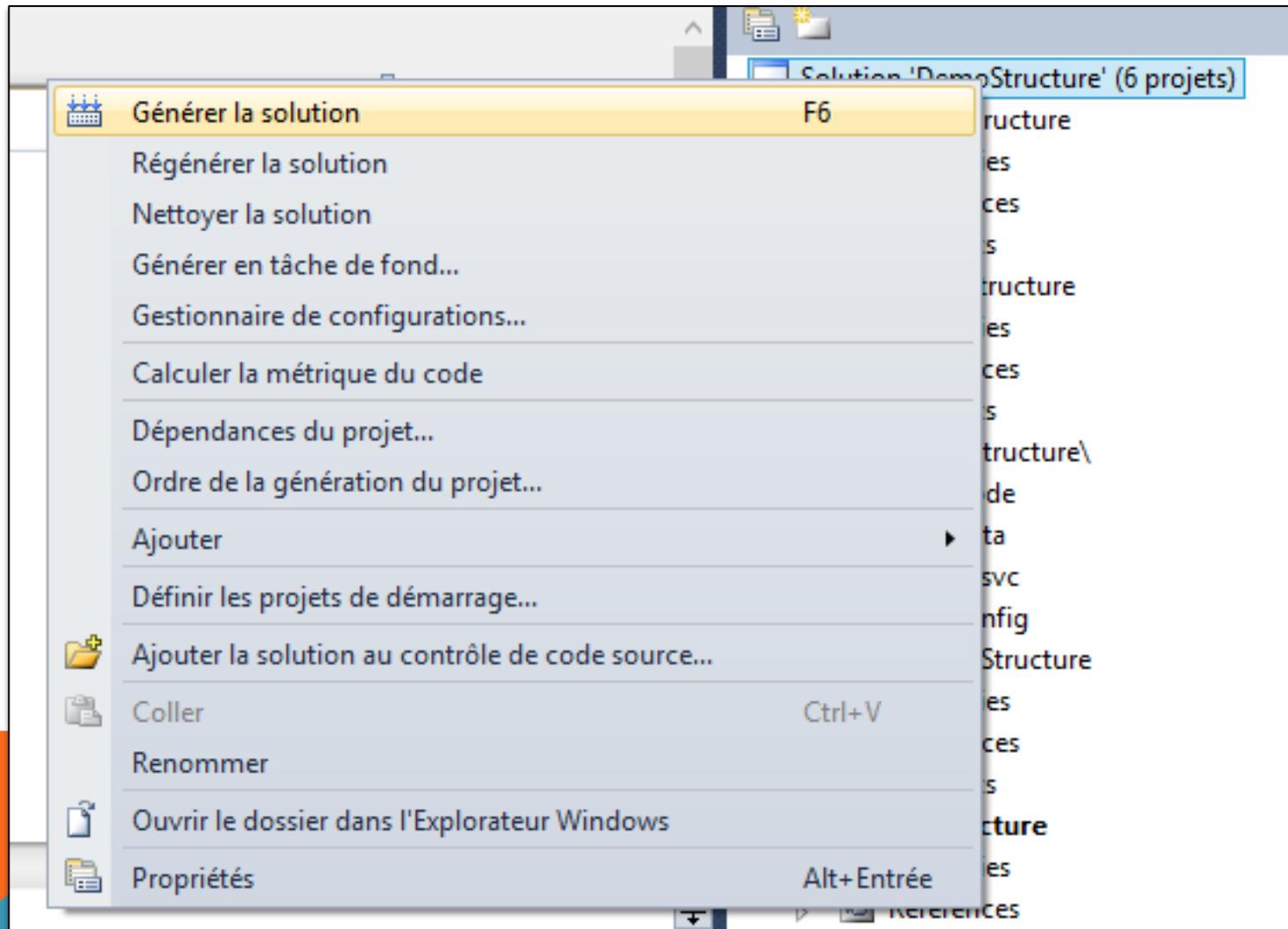
QUESTIONS ? REMARQUES ?



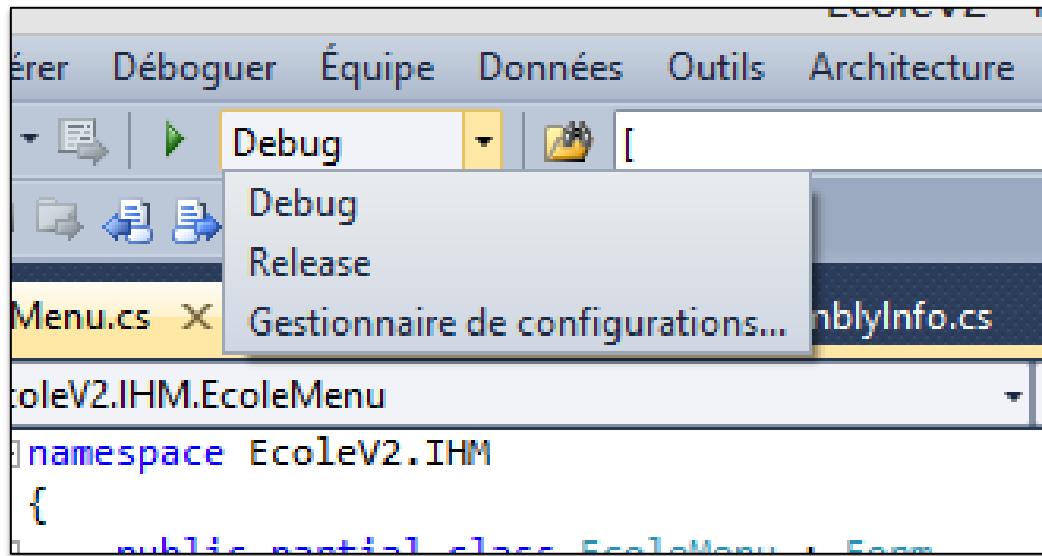
# LEÇON 4

GÉNÉRATION ET DÉBOGAGE

# RÉGÉNÉRATION ET EXÉCUTION

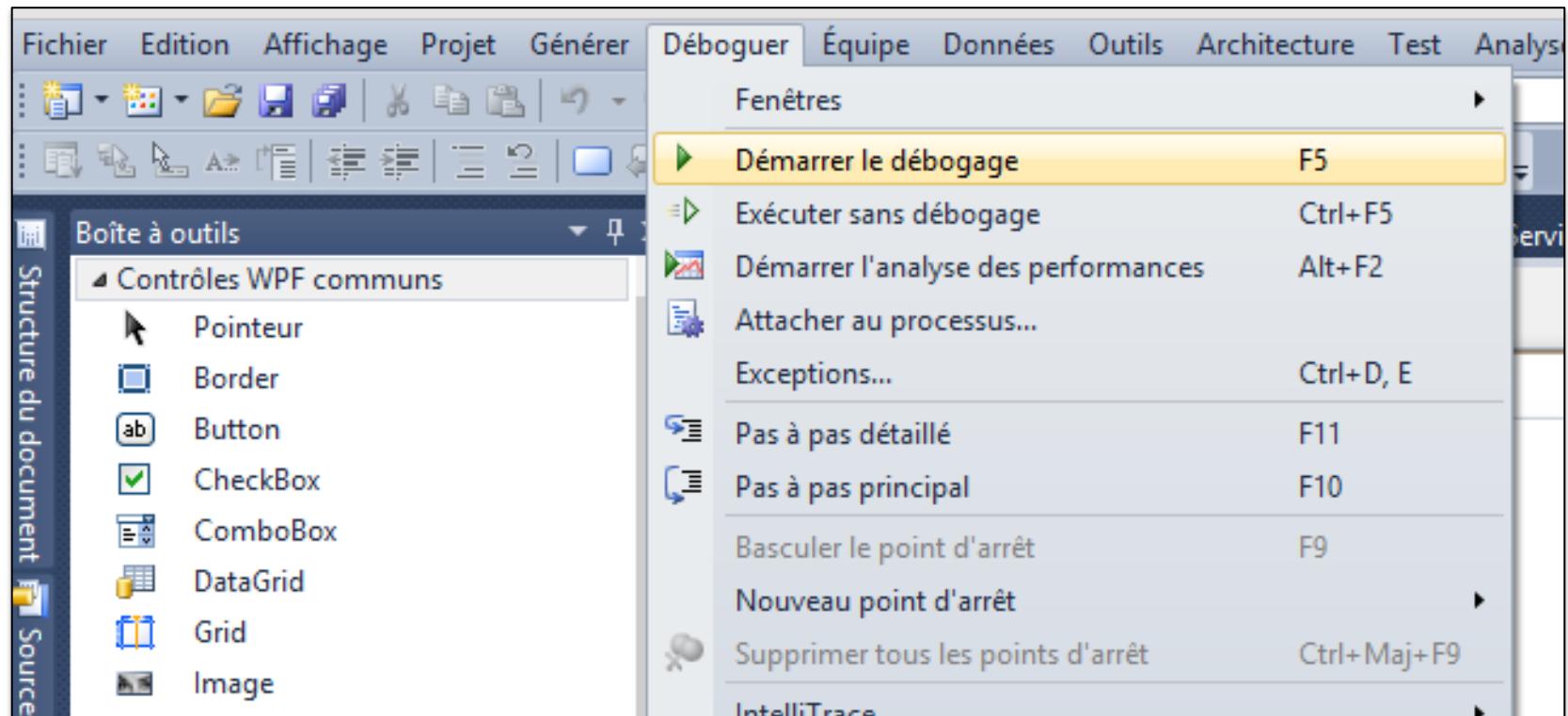


# MODES D'EXÉCUTION



Mode débuge : Non optimisé, mais débogage facilité  
Mode Release : Optimisé, mais pas de débogage

# LE DÉBOGAGE



QUESTIONS ? REMARQUES ?



## EXERCICES

Le premier exercice consiste à créer une première application console demandant à l'utilisateur son nom grâce à la console.

L'application devra alors répondre "Bonjour Mr [nom]"



# **MODULE 3**

LES BASES DU LANGAGE



# LEÇON 1

PREMIERS PAS

## PREMIERS PAS

*C # philosophie entièrement objet*

- Chaque type, variable, classe sont des objets.
- Toutes les méthodes appartiennent obligatoirement à une classe.
- Le point d'entrée d'un programme porte le nom Main

# PREMIERS PAS

```
using System;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
        }
    }
}
```

QUESTIONS ? REMARQUES ?





# LEÇON 2

CONVENTIONS DE NOMMAGE

## LES CONVENTIONS DE NOMMAGE

- C# est sensible à la casse, VB est insensible à la casse.
- Les noms de méthode et de classe et des namespaces commencent par une majuscule

*Program, Module1, Main*

- Les paramètres et les variables locales sont en minuscules.
- Les noms composés de plusieurs mots sont en majuscules

*DemarrerVoiture(int quantiteCarburant)*

## LES CONVENTIONS DE NOMMAGE

### Conventions moins formelles :

- Les variables privés peuvent commencer par un Under score,

***private int \_voiture;***

- Les collections d'objet sont aux pluriels.

***public Voiture MaVoiture***

***public Voiture[] MesVoitures***

- Les variables statiques peuvent commencer par un s suivi d'un Under score,

***int s\_ageLegal;***

## LES ESPACES DE NOM

- Organisation des classes du Framework et de votre projet
- Possibilité de créer deux classes de même nom, mais de contexte différent

*Convention de nommage :*

`<NomSociete>.<NomApplication>.<NomModule>.<BriqueApplicative>.<...>`

*Exemple :*

`Contoso.SuperCRM.GestionClient.BusinessLayer`

## LES ESPACES DE NOM

- Soit en utilisant la syntaxe de nom entièrement qualifié.

```
System.IO.File.Exists(MonFichier)
```

- Soit en utilisant dans le code les mot clés `using`.

The screenshot shows a C# code editor with two examples of qualified name usage:

```
using System;
using System.IO;

namespace ConsoleApp77
{
    class Program
    {
        static void Main(string[] args)
        {
            string MonFichier = "c:\\\\test.txt";
            File.Exists(MonFichier);
        }
    }
}
```

The first example at the top uses fully qualified names (`System.IO.File.Exists`). The second example in the `Main` method uses the `using` directive to import the `File` class from the `System.IO` namespace, allowing the use of the shorter name `File.Exists`.

QUESTIONS ? REMARQUES ?



# LEÇON 3

LES TYPES ET LES VARIABLES

## LES IDENTIFICATEURS

- Ne peuvent pas commencer par un chiffre
- Ne peuvent pas contenir d'espace
- Peuvent contenir des chiffres
- Peuvent commencer ou contenir un underscore
- Peuvent contenir des accents  
*String prénom;*

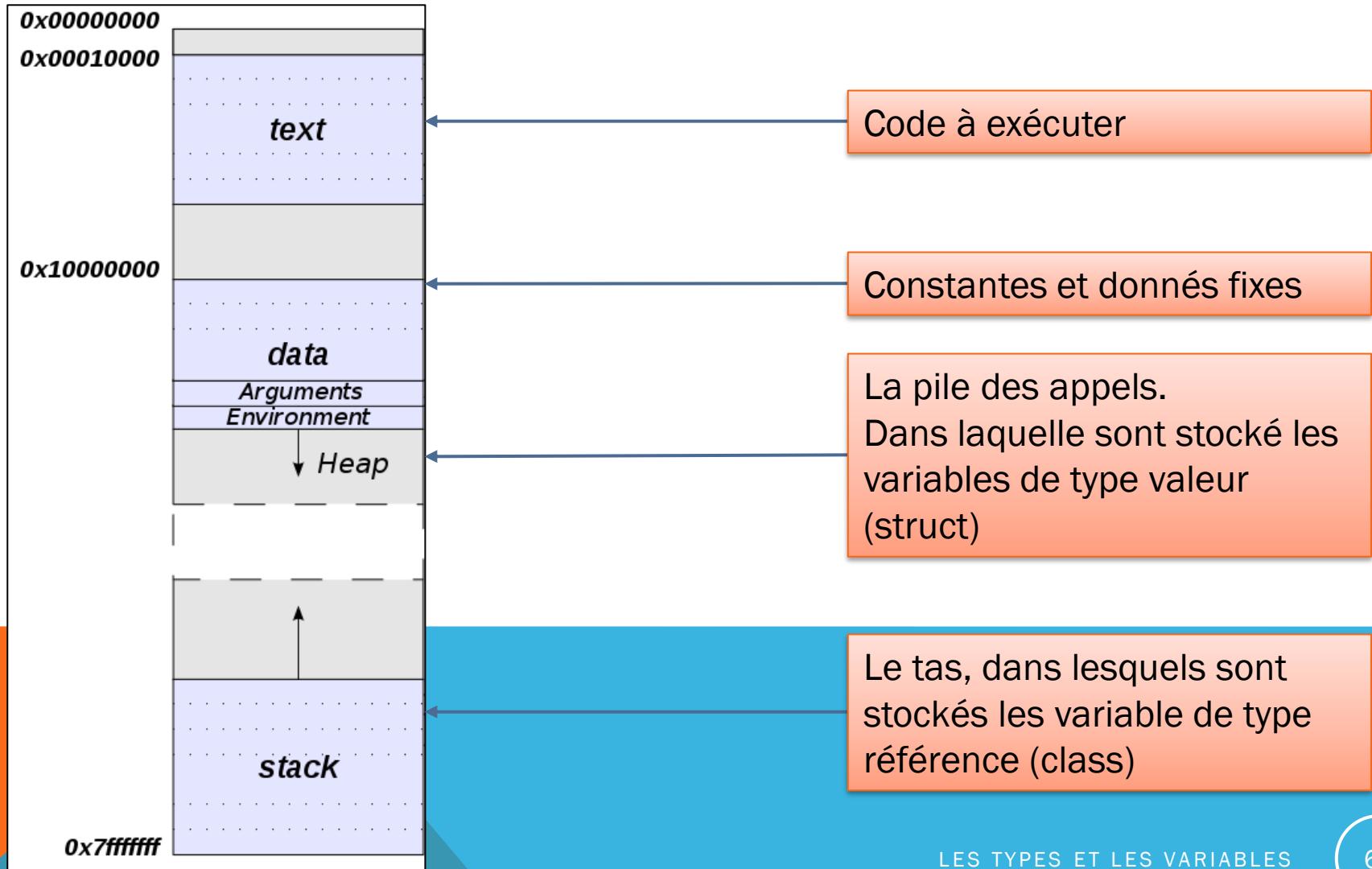
**Rem :** En c#, il est possible d'utiliser un mot réservé comme identificateur

*Ex : int @int = 5;*

## TYPES VALEUR ET TYPES RÉFÉRENCE

- Les types valeurs sont créés directement sur la pile.  
*System.Int32*  
*System.DateTime*  
*System.Drawing.Point*
- Les types références n'ont qu'une référence dans la pile.  
*System.IO.File*  
*System.Configuration.ApplicationSettingsBase*

# ORGANISATION DE LA MÉMOIRE



## LES CONSTANTES

- Les constantes sont des informations non modifiables.

*Math.Pi*

```
const double PI = 3.14159;
```

- Les constantes sont des données en mémoire et doivent respecter leurs types

```
const short tinynum = 3.14100000000000000059;
```

## LES STRUCTURES

- Équivalent aux classes mais elles sont stockées dans la pile (type valeur)
- Pas d'héritage possible
- Impossible d'affecter null à un objet de type structure
- Un constructeur doit forcément affecter toutes les variables de la structure

```
struct Simple
{
    public int Position;
    public bool Exists;
    public double LastValue;
}
```

# LES ÉNUMÉRÉS

- Enuméré = type

```
3 références
enum JourDeLaSemaine
{
    Lundi,
    Mardi,
    Mercredi,
    Jeudi,
    Vendredi,
    Samedi,
    Dimanche
}
```

- Facilité d'utilisation

```
JourDeLaSemaine jour = JourDeLaSemaine.Lundi;

if (jour == JourDeLaSemaine.Lundi)
{
    Console.WriteLine("Nous sommes un Lundi");
}
```

QUESTIONS ? REMARQUES ?





# LEÇON 4

CONVERSION DE TYPE

## LE TRANSTYPAGE

- Il existe 2 types de transtypage en C#:
  - Implicit:

```
int a = 5;  
  
long b = a; // transtypage implicite OK car long est plus grand que int
```

- Explicit:

```
double a = 5.5;  
  
int b = (int)a; // transtypage explicite car perte d'information
```

## CONVERSION DE CHAINE DE CARACTÈRE

- Conversion d'un objet en chaîne de caractère
  - Méthode `.ToString()`;

```
double d = 13.34345;  
  
string s = d.ToString();
```

- Conversion d'une chaîne en objet
  - À l'aide de `Convert`
  - À l'aide de `Parse()` ou `TryParse()`

```
string str = "34.56";  
  
double d = Convert.ToDouble(str);
```

## CONVERSION DE CHAINE DE CARACTÈRE

- Exemple de conversion d'une string en date en utilisant le formatage

```
DateTime madate = new DateTime(2022, 1, 11);  
  
string str = madate.ToString("dd/MM/yyyy");
```

## INFÉRENCE DE TYPE

- Mot clé **var** en c#
- Le compilateur déduit le type de la variable grâce à son affectation.

```
var mavariable = 13.89;  
  
string str = mavariable.ToString();
```

- Utilisation de type anonyme

```
var mavoiture = new {modele="Saxo", Année=1990,marque="Citroen" };  
  
string str = mavoiture.ToString();
```

## BOXING ET UNBOXING

- Boxing :
  - Conversion d'un type **valeur** en type **Object**
  - Enregistrement de la valeur dans une instance de **System.Object**
  - **Conversion implicite**
- Unboxing :
  - Récupération de la valeur de l'objet
  - **Conversion explicite**

## LA CONSOLE

- L'objet **Console** est très utile pour le débogage ou pour les traces.
- 3 propriétés :
  - In: Lecture sur le flux d'entrée
  - Out : Ecriture sur le flux de sortie
  - Error: Ecriture sur le flux d'erreurs

```
Console.In.ReadLine();
Console.Out.WriteLine("HelloWorld");
Console.Error.WriteLine("Error");
```

QUESTIONS ? REMARQUES ?



## EXERCICES

En utilisant la console, demander à l'utilisateur de créer un article composé d'un nom, d'un prix unitaire et d'une quantité.

Afficher le nom de l'article et le prix total (prix \* qte)



# **MODULE 4**

LES BASES DU LANGAGE PART 2



# LEÇON 1

LES OPERATEURS

# OPÉRATEURS

- Opérateurs arithmétiques
  - Type du résultat définit de manière à ne perdre le moins d'information possible  
*int / double -> double*
- Opérateurs de comparaison
  - Opérateurs : <, >, <=, >=, ==, !=
- Opérateurs binaires
  - Travaillent sur la représentation binaire des valeurs
  - Opérateurs : &, |, ~, >>, <<, ^

# OPÉRATEURS

- Opérateurs booléens
  - Opérateurs : &&, ||, !
- Opérateur ternaire :

```
s2 = (s == "") ? "chaîne vide" : s + " la suite de s2";
```

- Pré/pos incrémentation/décrémentation
  - i++, ++i, i--, -i
- Les opérateurs sont aussi des méthodes d'objet:

```
int sum = 5+5;           // -----> 10
string strSum = "5"+ "5"; // -----> "55"
```

QUESTIONS ? REMARQUES ?



# LEÇON 2

LES BOUCLES

## STRUCTURES DE BOUCLE

- Boucle **while** ( tant que ... faire ... )

```
int i = 10;
while( i-- > 0 )
    Console.WriteLine( i );
```

- Boucle **do/while** ( faire ... tant que ... )

```
int i = 10;
do
{
    Console.WriteLine( i );
} while (i-- > 0);
```

# STRUCTURES DE BOUCLE

- Boucle *for*

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

- Boucle *foreach*

```
int[] tblei = new int[50];

foreach(int i in tblei)
    Console.WriteLine( i );
```

## STRUCTURE DE BOUCLE

- Sortie anticipée d'une boucle grâce aux mots clés ***break***
- Passage direct à la prochaine itération grâce au mot clé ***continue***

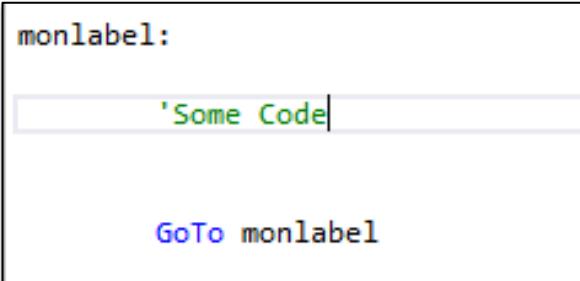
```
for (int i=0; i<10; i++)
{
    if (i % 2 == 0) continue;
    if (i == 7) break;
    Console.WriteLine(i);
}
//1,3,5
```

Résultat : 1,3,5

# STRUCTURE DE BOUCLE

+ d'infos

- Les labels



```
for (int x = 0; x < 100; x++)
{
    for (int y = 0; y < 100; y++)
    {
        if (x == 5 && y == 5) goto finBoucle;
    }
}
finBoucle:;
```

Utilisation déconseillé car cela nuit à la lecture du code.

QUESTIONS ? REMARQUES ?



# LECON 3

LES CONDITIONS

# LES CONDITIONS

- If... then ... else

```
int i=10;  
  
if (i == 10)  
{  
    Console.WriteLine( i );  
}
```

- Notation ternaire

```
if (i == 10)  
{  
    Console.WriteLine( i );  
}else  
{  
    Console.WriteLine( "i est différent de 10" );  
}
```

```
Console.WriteLine( (i == 10)?i.ToString():"i est différent de 10" );
```

## CHOIX MULTIPLE : SWITCH/SELECT

```
switch( combienOnEst ) {
    case 1:
        Console.Out.WriteLine("FaireUnDemineur");
    break;
    case 2:
        Console.Out.WriteLine("JouerAuxDames");
    break;
    case 3:
        Console.Out.WriteLine("EssayerDeTrouverUnQuatrieme");
    goto case 4; // Fall through explicite nécessaire
    case 4: // Fall through implicite autorisé
    case 5:
        Console.Out.WriteLine("FaireUnPoker");
    break;
    case 22:
        Console.Out.WriteLine("FaireUnFoot");
    break;
    default:
        Console.Out.WriteLine("BoireDeLaBiere");
    break;
}
```

QUESTIONS ? REMARQUES ?



# LEÇON 4

PASSAGE D'ARGUMENTS AUX MÉTHODES

# PASSAGE D'ARGUMENTS PAR VALEUR OU RÉFÉRENCE

- Par défaut c'est la valeur de l'objet qui est recopié en argument de méthode

```
static void MethodeDeTest(int firstarg,Maclass secondarg)
{
    secondarg.Libelle = "tata";
    firstarg = 45;
}

static void Main(string[] args)
{
    Maclass cl = new Maclass();
    cl.Libelle = "toto";
    int param = 6;

    MethodeDeTest(param,cl);

    Console.Out.WriteLine(param);
    Console.Out.WriteLine(cl.Libelle);

    Console.ReadLine();
}
```

param = 6  
cl.Libelle = tata

param vaut 6 car la valeur est copié dans le paramètre. La valeur d'origine ne change pas.

# PASSAGE D'ARGUMENTS

- Mot clé **ref** pour un passage par référence

```
static void MethodeDeTest(ref int firstarg, ref MaClass secondarg)
{
    secondarg.Libelle = "tata";
    firstarg = 45;
}

static void Main(string[] args)
{
    MaClass cl = new MaClass();
    cl.Libelle = "toto";
    int param = 6;

    MethodeDeTest(ref param, ref cl);

    Console.Out.WriteLine(param);
    Console.Out.WriteLine(cl.Libelle);

    Console.ReadLine();
}
```

param = 45  
cl.Libelle = tata

param vaut 45 car la référence est copié dans le paramètre. La valeur d'origine change.

## PASSAGE D'ARGUMENTS

- Le mot clé **out**
  - Spécifique à c#
  - Assignment obligatoire

```
static void MethodeDeTest(out int firstarg,ref MaClass secondarg)
{
    // Le paramètre de sortie 'firstarg' doit être assigné avant que le contrôle quitte la méthode actuelle

    secondarg.Libelle = "tata";
}

static void Main(string[] args)
{
    MaClass cl = new MaClass();
    cl.Libelle = "toto";
    int param = 6;

    MethodeDeTest(out param,ref cl);

    Console.Out.WriteLine(param);
    Console.Out.WriteLine(cl.Libelle);

    Console.ReadLine();
}
```

## LES ARGUMENTS VARIABLE

- Arguments de type tableaux d'objets

```
static void MethodeDeTest(params object[] mesparametres)
{
    Console.Out.WriteLine(mesparametres[0]);
    Console.Out.WriteLine(mesparametres[1]);
    Console.Out.WriteLine(mesparametres[2]);
}

static void Main(string[] args)
{
    MethodeDeTest("1 textes",34,new System.IO.FileInfo("c:\test.txt"));
}
```

QUESTIONS ? REMARQUES ?



## EXERCICES

**Exercice 1 :** Grace aux boucles, il est désormais possible de créer des algorithmes plus complexes et d'améliorer notre mini gestion de stock.

A partir de l'exercice précédent, rajouter la possibilité de demander à l'utilisateur s'il souhaite continuer avec un autre article ou pas.



## EXERCICES

**Exercice 2 :** concevoir un jeu demandant à l'utilisateur de saisir une valeur entre 0 et 100, l'ordinateur compare la valeur à un chiffre tiré au hasard et informe l'utilisateur si le nombre est plus grand ou plus petit.

La partie continue tant que l'utilisateur n'a pas trouvé le chiffre, ou si plus de 10 essais ont été réalisés



# LEÇON 5

DIRECTIVES DE COMPILEATION ET ATTRIBUTS

# DIRECTIVES DE COMPILATION

Permet de spécifier un comportement particulier du compilateur

```
#define TEST

using System;
using System.IO;
using ConsoleApplication1;

class Program
{
    static void Main(string[] args)
    {
#define DEBUG
        Console.WriteLine("Message de debug");
#define TEST
        Console.WriteLine( "Message de test" );
#define else
        Console.WriteLine( "Message normal" );
#define endif
#define endif
    }
}
```

## LES ATTRIBUTS

- Les attributs sont fortement liés à la réflexion
- Les attributs enrichissent la structure du code par des métadonnées.

```
[DllImport("Kernel32.dll")]
public static extern void Beep(int f, int duree);

[Serializable]
public string MonText;
```

## LES ATTRIBUTS

- Il aussi est possible de créer ces propres attributs en héritant de la classe de base **Attribute**

```
[MonAttribut()]
    MonAttribut(Properties: [MaString = string], [MaValeur = int])
```

```
1 reference
public class MonAttribut : Attribute
{
    0 references
    public int MaValeur { get; set; }
    0 references
    public string MaString { get; set; }
}
```

QUESTIONS ? REMARQUES ?



# MODULE 5

LES COLLECTIONS





# LEÇON 1

LES TABLEAUX

## LES TABLEAUX SIMPLES

- Les tableaux sont des objets comme les autres
  - Utilisation de la classe System.Array

```
int[] tabInt;  
System.Array t = new int[13];
```

- Initialisation direct possible

```
int[] t = {1,2,3,4,5};
```

- Affectation

```
int i = t[3];  
t[3] = 45;
```

## TABLEAUX MULTIDIMENSIONNELS

- Un tableau simple est une liste
- Un tableau à 2 dimensions est une grille

```
int[,] tabInt;  
int[,] t = new int[13,4];
```

```
int i = t[3,2];  
t[3,8] = 45;
```

# TABLEAUX IMBRIQUÉS (JAGGED ARRAYS)

Tableaux contenant des tableaux

```
int [][] t = new int[2][]; // Deux dimensions
t[0] = new int [2]; // 1ere ligne de 2 cases
t[1] = new int [4]; // 2nde ligne de 4 cases
t[1][3] = 30;
```

QUESTIONS ? REMARQUES ?



## EXERCICES

**Exercice 1 :** A partir de l'exercice précédent avec les articles, rajouter la possibilité de stocker des articles dans un tableau, tant que l'utilisateur le souhaite.

Un tableau étant fixe, il est nécessaire de définir un tableau suffisamment grand (1000) et de gérer un index pour savoir où insérer l'article dans le tableau.

Lorsque l'utilisateur ne souhaite plus ajouter d'articles dans le tableau, on sort de la boucle et on parcourt les articles du tableau pour afficher le Libelle et le prix total de chaque article.



## EXERCICES

***Exercice 2 : Le jeu du pendu***  
on définit un tableau de mots, on tire au hasard un mot.  
L'utilisateur doit proposer une lettre, si la lettre appartient au  
mot, on ajoute la lettre à un tableau de caractères  
permettant de savoir si le mot a été trouvé ou pas. Si la lettre  
n'appartient pas au mot, un compteur s'incrémente jusqu'à  
un maximum de 10 (dessin du pendu)



# LEÇON 2

LES ÉNUMÉRATEURS

## ENUMÉRATEURS

- = la base de toute collections.
- Utilisation de **foreach** pour parcourir un **IEnumerator**
- Implémentation de **GetEnumerator()**
- System.Collections.IEnumerator

```
// Lit l'élément courant
object Current {get;}

// Première méthode invoquée par foreach,
// Rend vrai si l'élément courant est valide
bool MoveNext();

// Le premier élément de la collection devient le prochain élément
void Reset();
```

# TABLEAUX DYNAMIQUES

## Les arrayList

- Un tableau n'est pas assez souple car sa taille est fixe.
- La classe ArrayList permet de stocker des objets sans spécifier de taille
- Taille dynamique
- Hérite de **IEnumerable** (peut donc être utilisé avec foreach)
- Inconvénients :
  - On peut stocker tous types de données
  - Problème de Boxing/Unboxing

```
ArrayList array = new ArrayList();
array.Add(5);
array.Add("Toto");
array.Add(new Personne());

int a = (int)array[0];
```

# TABLEAUX DYNAMIQUES

## Fonction de tri .Sort()

- Fonctionne avec des classes héritant de **IComparable**
- Ou avec un paramètre de type IComparer

```
ArrayList array = new ArrayList();
array.Add(new Personne(){Nom = "Dupond"});
array.Add(new Personne(){Nom = "Durand"});
array.Add(new Personne() {Nom = "Martin" });

array.Sort(new PersonneComparer());
```

```
class PersonneComparer : IComparer
{
    public int Compare(object x, object y)
    {
        Personne p1 = (Personne)x;
        Personne p2 = (Personne)y;
        return p1.Nom.CompareTo(p2.Nom);
    }
}
```

QUESTIONS ? REMARQUES ?





# LEÇON 3

LES COLLECTIONS GÉNÉRIQUES

## C'EST QUOI UN GÉNÉRIQUE ?

Une classe générique est une classe permettant des traitements sur des types inconnues à la conception.

- Ce sont des Meta-class (classes qui génèrent des classes)
- Lors de la compilation, le compilateur transforme ces classes en une classe avec le bon type avant de compiler.
- Avantage du langage fortement Typé.
- Evite le boxing/unboxing
- Utilisation de `<T>`

```
List<Personne> array = new List<Personne>();  
array.Add(new Personne(){Nom = "Dupond"});  
array.Add(new Personne(){Nom = "Durand"});  
array.Add(new Personne() { Nom = "Martin" });  
  
array.Add("Martin");
```

Erreur, car le type  
n'est pas celui défini  
(ici Personne)

# LISTE GÉNÉRIQUE

Les listes génériques *List<T>*

- Classe générique équivalent à *ArrayList*

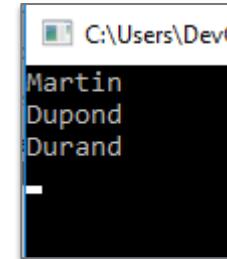
```
List<Personne> lst = new List<Personne>();  
  
lst.Add(new Personne("Jean"));  
lst.Add(new Personne("Jacques"));  
lst.Add(new Personne("Marcel"));  
lst.Add(new Personne("Jeanne"));  
lst.Add(new Personne("Fred"));  
  
foreach (Personne p in lst)  
    Console.Out.WriteLine(p);
```

## LISTE TRIÉE

Les listes triées **SortedList<TKey, TValue>**

- Liste trié avec table de hachage.
- Pas de doublons dans la clé possible
- Performance accrue pour le tri
- Utilisable avec une clé, et pas un indice

```
SortedList<int, Personne> ListTrie = new SortedList<int, Personne>();  
  
ListTrie.Add(4,new Personne(){Nom = "Dupond"});  
ListTrie.Add(5,new Personne(){Nom = "Durand"});  
ListTrie.Add(3,new Personne() { Nom = "Martin" });  
  
foreach (var p in ListTrie)  
    Console.WriteLine(p.Value.Nom);
```



# DICTIONNAIRE

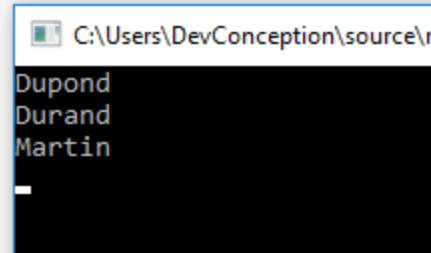
Les dictionnaires **Dictionary<TKey, TValue>**

- Permet un stockage de données par clé. La récupération d'une valeur a partir de la clé est quasi instantané, quelque soit le nombre d'éléments dans le dictionnaire.
- Fournit un conteneur clé, valeur
- La clé est unique, les doublons ne sont pas autorisé
- Utilisation de KeyValuePair dans le foreach

```
Dictionary<int, Personne> Dico = new Dictionary<int, Personne>();

Dico.Add(4,new Personne(){Nom = "Dupond"});
Dico.Add(5,new Personne(){Nom = "Durand"});
Dico.Add(3,new Personne() { Nom = "Martin" });

foreach (var p in Dico)
    Console.WriteLine(p.Value.Nom);
```

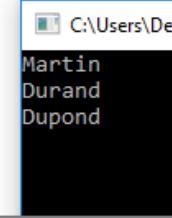


# LES PILES ET LES FILES

## Les piles *Stack<T>*

- FILO (First In Last Out)

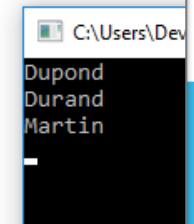
```
Stack< Personne> Dico = new Stack< Personne>();  
  
Dico.Push( new Personne(){Nom = "Dupond"});  
Dico.Push( new Personne(){Nom = "Durand"});  
Dico.Push( new Personne() { Nom = "Martin" });  
  
while (true)  
{  
    Personne p;  
    if (!Dico.TryPop(out p)) break;  
    Console.WriteLine(p.Nom);  
}
```



## Les files *Queue<T>*

- FIFO (First In First Out)

```
Queue< Personne> Dico = new Queue< Personne>();  
  
Dico.Enqueue( new Personne(){Nom = "Dupond"});  
Dico.Enqueue( new Personne(){Nom = "Durand"});  
Dico.Enqueue( new Personne() { Nom = "Martin" });  
  
while (true)  
{  
    Personne p;  
    if (!Dico.TryDequeue(out p)) break;  
    Console.WriteLine(p.Nom);  
}
```



QUESTIONS ? REMARQUES ?



## EXERCICES

**Exercice 1 :** A partir de l'exercice précédent avec les articles, transformer le tableau en List<T>



## EXERCICES

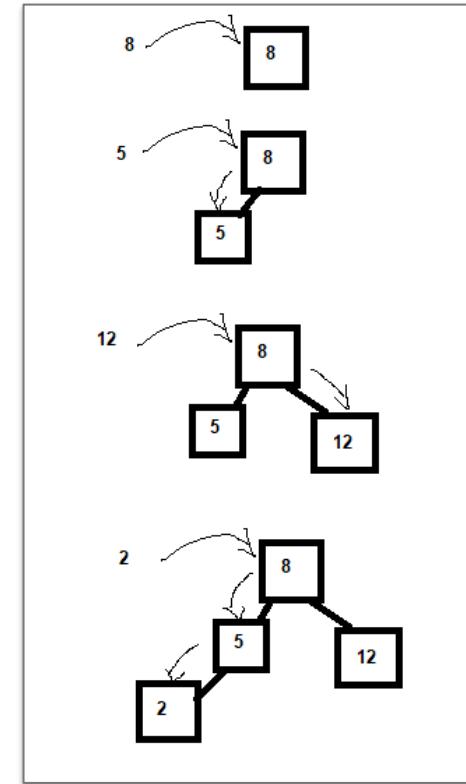
### Exercice 2 : Création d'un algorithme d'arbre binaire

Un arbre binaire et une structure permettant de stocker une valeur et 2 sous structures de même type.

Lorsqu'on rajoute une valeur au travers une fonction, celle ci compare la valeur de l'objet avec la valeur en paramètre, si c'est plus grand la valeur se répercute à droite, si c'est plus petit, elle se répercute à gauche.

Cela va constituer un arbre dont les éléments seront pré-triés.

Par récursivité, il est possible de récupérer les valeurs dans l'ordre.





# LEÇON 4

UTILISATION DES FLUX

# APPROCHE CLASSIQUE PAR LES LISTES

```
static void Main(string[] args)
{
    List<string> liste = new List<string>() { "Bateau", "maison", "velo", "voiture" };

    liste = Majuscule(liste);
    liste = First2(liste);

    foreach (var s in liste)
        Console.WriteLine(s);
}

private static List<string> Majuscule(List<string> liste)
{
    List<string> lst = new List<string>();
    foreach(var s in liste)
    {
        Console.WriteLine("Majuscule");
        lst.Add(s.ToUpper());
    }
    return lst;
}

private static List<string> First2(List<string> liste)
{
    List<string> lst = new List<string>();
    foreach (var s in liste)
    {
        Console.WriteLine("First2");
        lst.Add(s.Substring(0,2));
    }
    return lst;
}
```

```
C:\Users\DevConception\sources>
Majuscule
Majuscule
Majuscule
Majuscule
First2
First2
First2
First2
BA
MA
VE
VO
```

Dans cette approche,  
3 listes sont créées et 3  
boucles sont faites,  
les 1 derrière les  
autres

# APPROCHE PAR FLUX

```
static void Main(string[] args)
{
    I Enumerable<string> liste = new List<string>() { "Bateau", "maison", "velo", "voiture" };

    liste = Majuscule(liste);
    liste = First2(liste);

    foreach (var s in liste)
        Console.WriteLine(s);
}

private static I Enumerable<string> Majuscule(I Enumerable<string> liste)
{
    foreach(var s in liste)
    {
        Console.WriteLine("Majuscule");
        yield return (s.ToUpper());
    }
}

private static I Enumerable<string> First2(I Enumerable<string> liste)
{
    foreach (var s in liste)
    {
        Console.WriteLine("First2");
        yield return (s.Substring(0,2));
    }
}
```

```
C:\Users\DevConception\source\re
Majuscule
First2
BA
Majuscule
First2
MA
Majuscule
First2
VE
Majuscule
First2
VO
```

Dans cette approche,  
1 seul liste crée et 1  
seul boucle est faite  
avec les 3 traitements

Le compilateur va  
imbriquer l'ensemble  
des foreach pour créer  
un flux de données

# DÉMONSTRATION



QUESTIONS ? REMARQUES ?



# **MODULE 6**

GESTION D'ERREURS ET LES ÉVÉNEMENTS





# LECON 1

LES EXCEPTIONS

# TRAITEMENT DES ERREURS

## Les exceptions

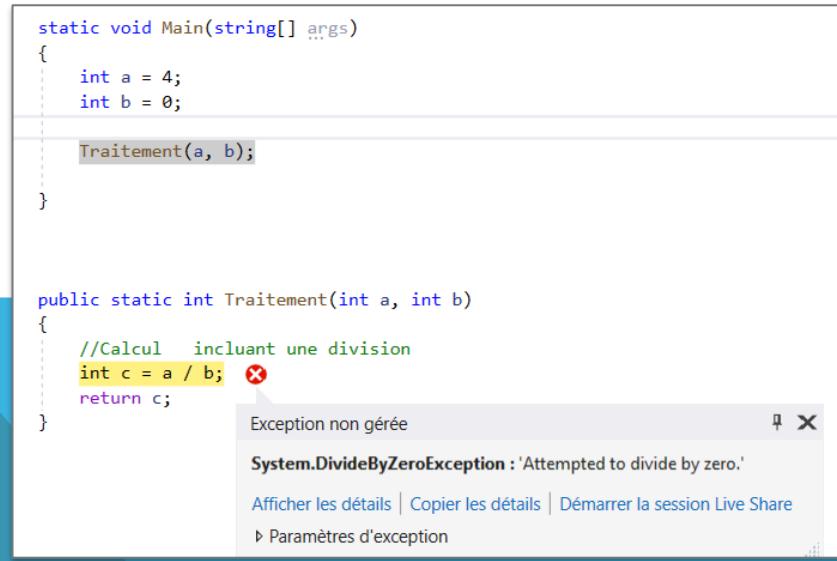
- Evite les multiples tests.
- Système d'exceptions hiérarchisé
- Utilisation des exceptions au plus bas niveau de .NET
- Utilisation de ***try...catch...finally***
- Possibilité de créer ces propres classes d'exception en dérivant ***System.exception***

# LES EXCEPTIONS

En C, l'approche pour la gestion des erreurs consiste à renvoyer un code en retour d'une fonction. Cette approche limite les valeurs possibles en sorti et elle est source d'erreur du fait de la mauvaise interprétation du code d'erreur.

➤ Une approche plus moderne, consiste à répercuter l'erreur dans la pile des appels et permettre aux appelants de traiter cette erreur.

## C'est ce qu'on appelle une exception



The screenshot shows a C# code editor with the following code:

```
static void Main(string[] args)
{
    int a = 4;
    int b = 0;

    Traitement(a, b);
}

public static int Traitement(int a, int b)
{
    //Calcul incluant une division
    int c = a / b; ✖
    return c;
}
```

A tooltip window is displayed over the line of code where the division occurs, containing the following information:

Exception non gérée

System.DivideByZeroException : 'Attempted to divide by zero.'

Afficher les détails | Copier les détails | Démarrer la session Live Share  
▶ Paramètres d'exception

## LES EXCEPTIONS

- Une exception peut être attrapée (**catch{}**) par un bloc de code **try{}**:
- Si une erreur est rencontré, l'exécution s'arrête et le code du bloc **catch{}** est immédiatement exécuté.

```
public static int Traitement(int a, int b)
{
    try
    {
        //Calcul incluant une division
        int c = a / b;
        return c;
    }catch (System.Exception)
    {
        return 0;
    }
}
```

## LES EXCEPTIONS

- Il est possible de re-lever un autre type d'exception pour apporter des précisions sur l'erreur

```
public static int Traitement(int a, int b)
{
    try
    {

        //Calcul incluant une division
        int c = a / b;
        return c;
    }catch (System.Exception)
    {
        throw new ArgumentException("le parametre b ne peut pas valoir 0");
    }
}
```

# LES EXCEPTIONS

- Il est possible de préciser quelles exceptions peuvent être attrapées

```
public static int Traitement(int a, int b)
{
    try
    {

        //Calcul incluant une division
        int c = a / b;
        return c;
    }
    catch (System.DivideByZeroException)
    {
        throw new ArgumentException("le parametre b ne peut pas valoir 0");
    }
    catch (System.Exception)
    {
        throw new ApplicationException("Erreur de calcul");
    }
}
```

# LES EXCEPTIONS

- Le bloc finally permet de libérer les ressources même si une erreur à eu lieu

```
public static int Traitement(int a, int b)
{
    try
    {
        //ouverture d'un fichier
        int c = a / b;
        //traitement sur le fichier
        return c;
    }
    catch (System.Exception)
    {
        throw new ApplicationException("Erreur de calcul");
    }
    finally
    {
        //Fermeture du fichier
    }
}
```

Ce code est  
toujours exécuté

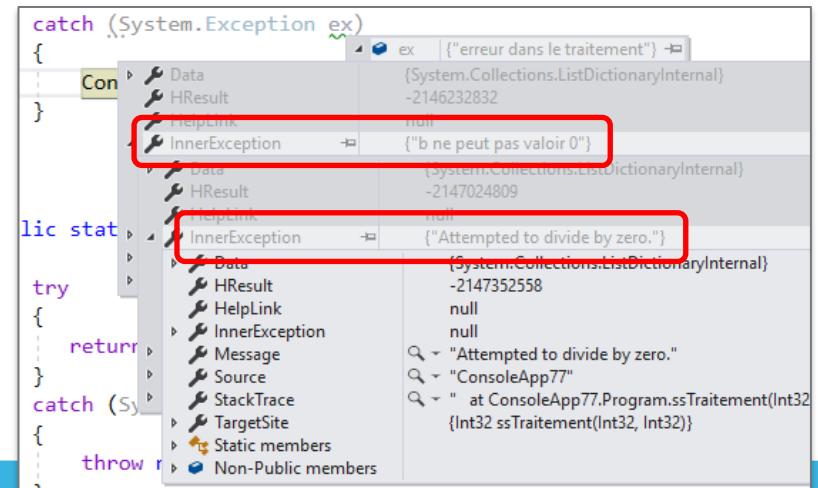
# LES EXCEPTIONS

- Les exceptions peuvent être chainées pour faciliter le débogage

```
try
{
    Traitement(a, b);
}
catch (System.Exception ex)
{
    Console.WriteLine("ERREUR");
}

public static int Traitement(int a, int b)
{
    try
    {
        return ssTraitement(a, b);
    }
    catch (System.Exception ex)
    {
        throw new ApplicationException("erreur dans le traitement", ex);
    }
}

public static int ssTraitement(int a, int b)
{
    try
    {
        return a / b;
    }
    catch (System.DivideByZeroException ex)
    {
        throw new ArgumentException("b ne peut pas valoir 0", ex);
    }
}
```



# DÉMONSTRATION

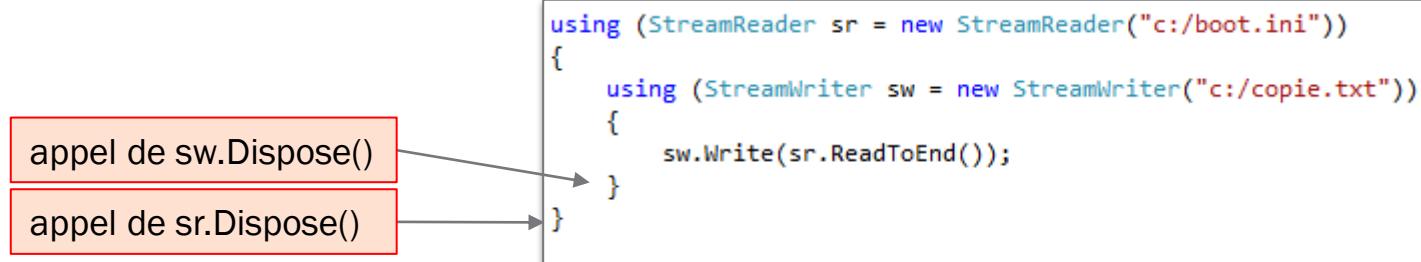


## USING ET IDISPOSABLE

- Les classes **IDisposable** sont des classes qui nécessite une libération explicite des ressources
- Le mot clé **using** est accompagné d'un bloc de code
- using fait appelle a **Dispose()**
- Dispose permet d'effectuer des opérations de nettoyage et de libération (socket, connexion DB)

```
using (StreamReader sr = new StreamReader("c:/boot.ini"))
{
    using (StreamWriter sw = new StreamWriter("c:/copie.txt"))
    {
        sw.Write(sr.ReadToEnd());
    }
}
```

appel de sw.Dispose()  
appel de sr.Dispose()



Le using inclut un mécanisme de try...catch...finally, garantissant l'appel au dispose dans tous les cas

QUESTIONS ? REMARQUES ?



## EXERCICES

***Exercice :*** A partir de l'exercice précédent sur les articles :

- Gérer les erreurs de saisie des utilisateurs concernant le prix de l'article. (L'utilisateur est en mesure de saisir du texte à la place du prix)
- Gérer cela avec des exceptions





# LEÇON 2

LES DÉLEGUÉS

# DÉLÉGUÉ

## Délégué = Pointeur de méthode

- Un délégué est un type permettant de déclarer une variable qui pointe sur une méthode.
- Mot clé **delegate**
- Respect de la signature

```
delegate double Operation(int a, int b);

class Program
{
    static void Main(string[] args)
    {
        Operation op = Addition;
        double resultat = op(5, 6);
    }

    public static double Addition(int a, int b)
    {
        return a + b;
    }

    public static double Soustraction(int a, int b)
    {
        return a - b;
    }
}
```

# DÉLÉGUÉ

- Possibilité de chainer les délégués.
- Possibilité d'effectuer des opération sur les délégués.

Les délégués sont utilisés dans les évènements.  
Dans l'utilisation des méthodes de callback, avec LINQ...

```
delegate double Operation(int a, int b);

class Program
{
    static void Main(string[] args)
    {
        CalculEtAffiche(5, 6, Addition);
        CalculEtAffiche(5, 6, Soustraction);
    }

    public static void CalculEtAffiche(int a, int b, Operation fct)
    {
        double resultat = fct(a, b);
        Console.WriteLine("Le résultat est " + resultat);
    }

    public static double Addition(int a, int b)
    {
        return a + b;
    }

    public static double Soustraction(int a, int b)
    {
        return a - b;
    }
}
```

# DÉLÉGUÉS

## Les expressions lambda

```
class Program
{
    static void Main(string[] args)
    {
        CalculEtAffiche(5, 6, (a, b) => a + b);
        CalculEtAffiche(5, 6, (a, b) => a + b);
    }

    public static void CalculEtAffiche(int a, int b, Func<int, int, double> fct)
    {
        double resultat = fct(a, b);
        Console.WriteLine("Le résultat est " + resultat);
    }
}
```

Une expression lambda est une représentation d'une fonction par le sigle « => »

Tout ce qui se trouve devant => correspond aux paramètres de la méthode et tout ce qui se trouve derrière représente le corps de la méthode

# DÉMONSTRATION



QUESTIONS ? REMARQUES ?



## EXERCICES

*Exercice :* A partir de l'exercice précédent sur les arbres binaires

Plutôt que de trier des entiers (ce n'est pas très utile), il serait préférable de trier des objets,

Il faut donc remplacer la valeur de type int, en valeur de type Object et utiliser une lambda pour gérer la comparaison permettant d'être à gauche ou à droite.



# LEÇON 3

LES ÉVÉNEMENTS

# EVÈNEMENTS

Un évènement est un délégué avec des droits d'accessibilités particuliers

- Mot clé **event** (Event)
- Evènement = Délégué
- Affectation impossible
- Abonnement à un évènement grâce à **+ =**
- Désabonnement grâce à **- =**

Les évènements ont quasiment toujours la même signature :

```
public delegate void EventHandler(object? sender, EventArgs e);
```

Celui qui lève l'évènement

Paramètres de l'évènement

# ÉVÈNEMENTS

## Utilisation de *EventHandler* et *EventArgs*

- Appelle d'un évènement

```
if (e != null) OnTextChanged(this, e);
```

```
class TicTac
{
    public event EventHandler OnTic;

    Timer tmr;

    public TicTac()
    {
        tmr = new Timer(o =>
        {
            if (OnTic != null)
                OnTic(this, new EventArgs());
        }, null, 0, 1000);
    }
}
```

*Rem : Les interfaces peuvent inclure des évènements*

QUESTIONS ? REMARQUES ?



# MODULE 7

LA PROGRAMMATION ORIENTÉ OBJET



# LEÇON 1

LA PROGRAMMATION PROCÉDURALE

# L'APPROCHE PROCÉDURALE

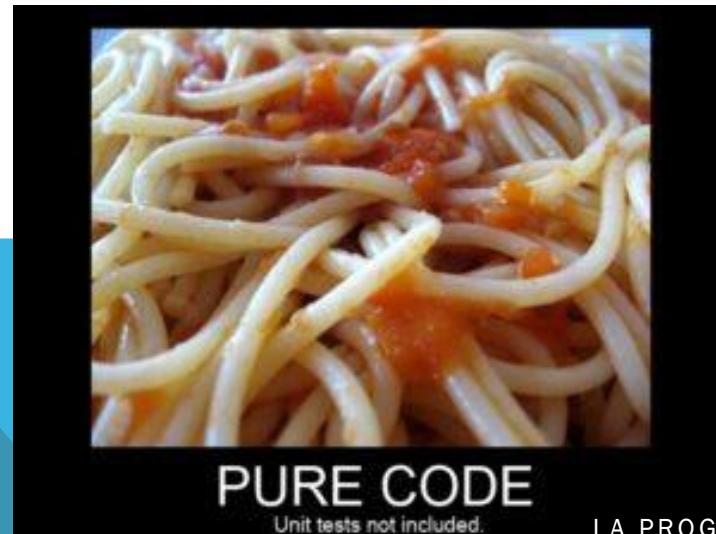
## Utilisation de fonctions et de procédures

- Lisibilité
  - Plus compréhensible
  - Effet de « cloisonnement » (un seul traitement métier)
- Réutilisabilité
  - Les fonctions peuvent être utilisés dans différents traitements, et/ou différents projets.
- Modularité
  - Les fonctions appelant d'autres fonctions peuvent être remaniés

# LA PROGRAMMATION STRUCTURÉE

## Inconvénients

- Il faut être rigoureux et méthodique afin d'éviter le code spaghetti
- Impose une connaissance quasi-globale de l'application pour éviter de redévelopper une fonction existante
- Les fonctions ne sont pas suffisamment représentatives du monde que l'on cherche à reproduire



**PURE CODE**

Unit tests not included.

LA PROGRAMMATION PROCÉDURALE

QUESTIONS ? REMARQUES ?



# LEÇON 2

LA PROGRAMMATION ORIENTÉ OBJET (POO)

# LA PROGRAMMATION OBJET

## L'Objet

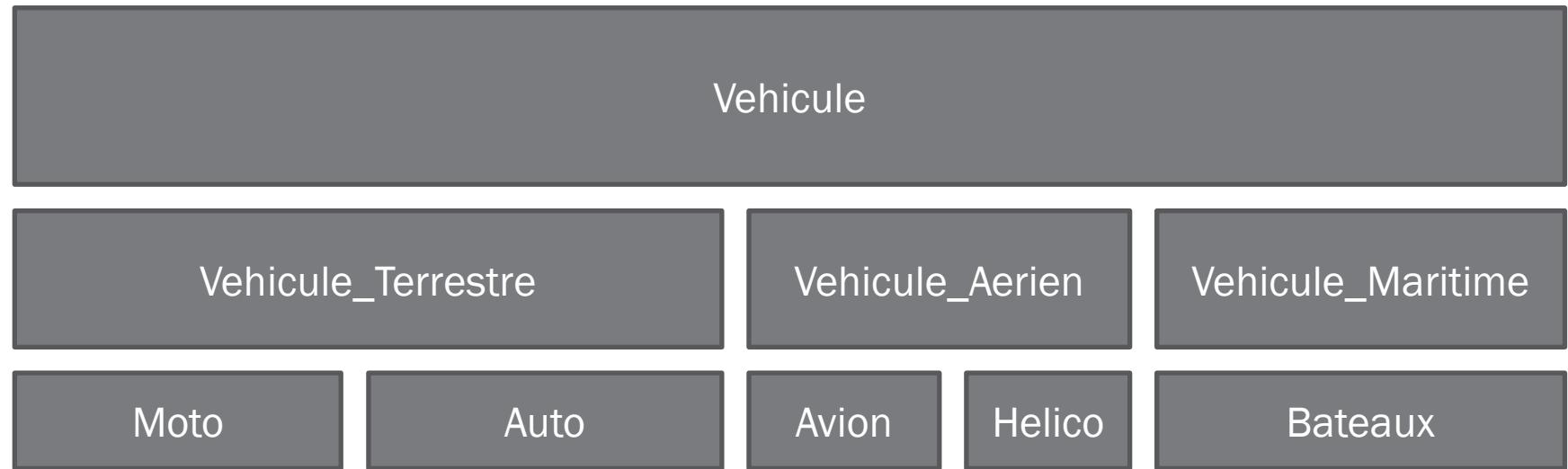
- Regroupement de données et d'actions propre à un concept.
- Quelques exemples :
  - *Voiture, chat, maison, collection, tableaux, Socket, message SOAP...*
- En programmation objet, tout est objet!
  - *Par exemple un entier (integer) est un objet.*

# LA PROGRAMMATION OBJET

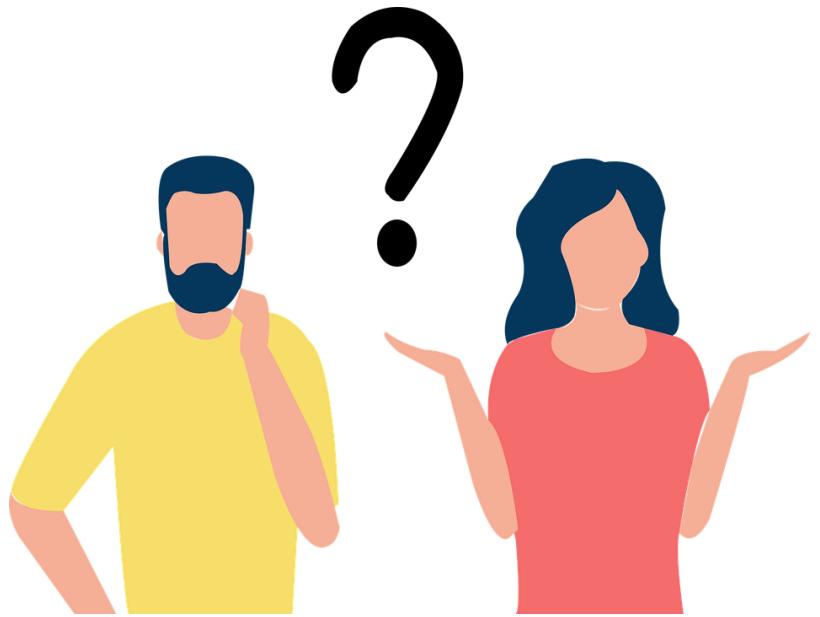
- L'effet boite noire
  - Permet de gérer la façon dont sera utilisé les fonctions de l'objet
- Lisibilité
  - Les objets représentent un concept, donc tous les éléments de l'objet sont en lien avec le concept
- Réutilisation du code
  - Les objets sont réutilisables dans plusieurs traitements et/ou projets

Programmation Objet = Représentation beaucoup plus claire du concept à développer

# EXEMPLE D'ARCHITECTURE OBJET



QUESTIONS ? REMARQUES ?



# LEÇON 3

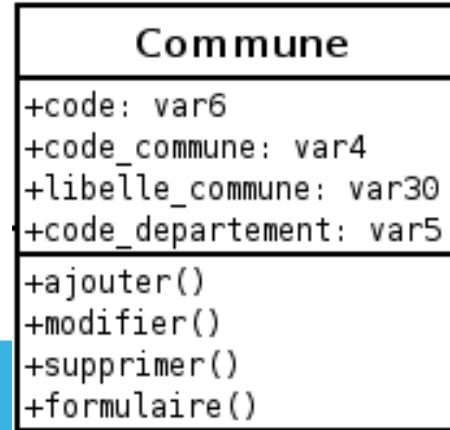
LES CLASSES, REPRÉSENTATION DU MONDE OBJET

## LA CLASSE

Classe = *représentation abstraite de la structure d'un objet*

- Un objet est *l'instanciation* d'une classe
- Un objet est donc la concrétisation d'une classe
- La classe contient la définition des données (les attributs de classes) mais aussi des actions (fonctions/méthodes) utiles à l'objet

- Exemple :



## LE CONSTRUCTEUR

*Constructeur = fonction appelée obligatoirement au moment de l'instanciation d'un objet.*

- Syntaxe différente suivant les langages.
- Permet d'initialiser correctement l'objet.
- Allocation des ressources
- Exemples :
  - Affectation de valeurs par défaut.
  - Lancement d'une action particulière à l'initialisation
  - ...

## LE DESTRUCTEUR

*Destructeur = fonction appelée obligatoirement au moment de la libération d'un objet*

- Syntaxe différente suivant les langages.
- Permet de libérer les ressources utilisées par l'objet.
- Exemples :
  - Fermeture d'un fichier ouvert .
  - Synchronisation de Thread
  - ...

## LE DESTRUCTEUR

### Durée de vie des objets

- Les objets ont une durée de vie dépendant de leur emplacement dans le programme. Cela dépend de la porté de l'objet instancié
- En C++ par exemple, les objets doivent être libérés par le développeur
- En .NET, un objet est libéré de la mémoire grâce au Garbage Collector

QUESTIONS ? REMARQUES ?



# LEÇON 4

LA NOTION D'HÉRITAGE

# HÉRITAGE ET POLYMORPHISME

## Héritage = spécialisation

- L'héritage permet de définir des attributs et des méthodes spécifiques à un sous type d'objet.
- Exemples:
  - L'attribut **NombreDeRoue** est spécifique aux classes **Vehicule\_Terrestre**.
  - Par contre, **VitesseDeDeplacement** est commun à tous les véhicules.
  - **Vehicule\_Terrestre** hérite de **Véhicule**, pour avoir les attributs **NombreDeRoue** et **VitesseDeDeplacement**

## HÉRITAGE ET POLYMORPHISME

- Dans cette exemple, **Vehicule\_Terrestre** est la classe dérivé de **Vehicule**.
- **Vehicule** est la classe de base (super classe) de **Vehicule\_Terrestre**

```
public class Vehicule_Terrestre : Vehicule
{
    public int NombreDeRoue { get; set; }
}
```

# HÉRITAGE ET POLYMORPHISME

Polymorphisme = changement de classe

- Un objet instancié pourra être utilisé comme étant de sa classe, ou de sa classe de base.
- Exemple :
  - Un tableau de **Vehicule** pourra contenir des **Vehicule\_Terrestre**, ou des **Vehicule\_Maritime**

```
List<Vehicule> vehicules = new List<Vehicule>()
{
    new Vehicule_Maritime(),
    new Vehicule_Terrestre()
};
```

QUESTIONS ? REMARQUES ?



# LEÇON 5

LES MODIFICATEURS D'ACCÈS

# LES MODIFICATEURS D'ACCÈS

## *public/private/protected*

- Renforce le cloisonnement en sécurisant l'accès aux données.
- Autorise ou masque l'accès à certains éléments de l'objet.

Les modificateurs d'accès est le mécanisme permettant l'encapsulation

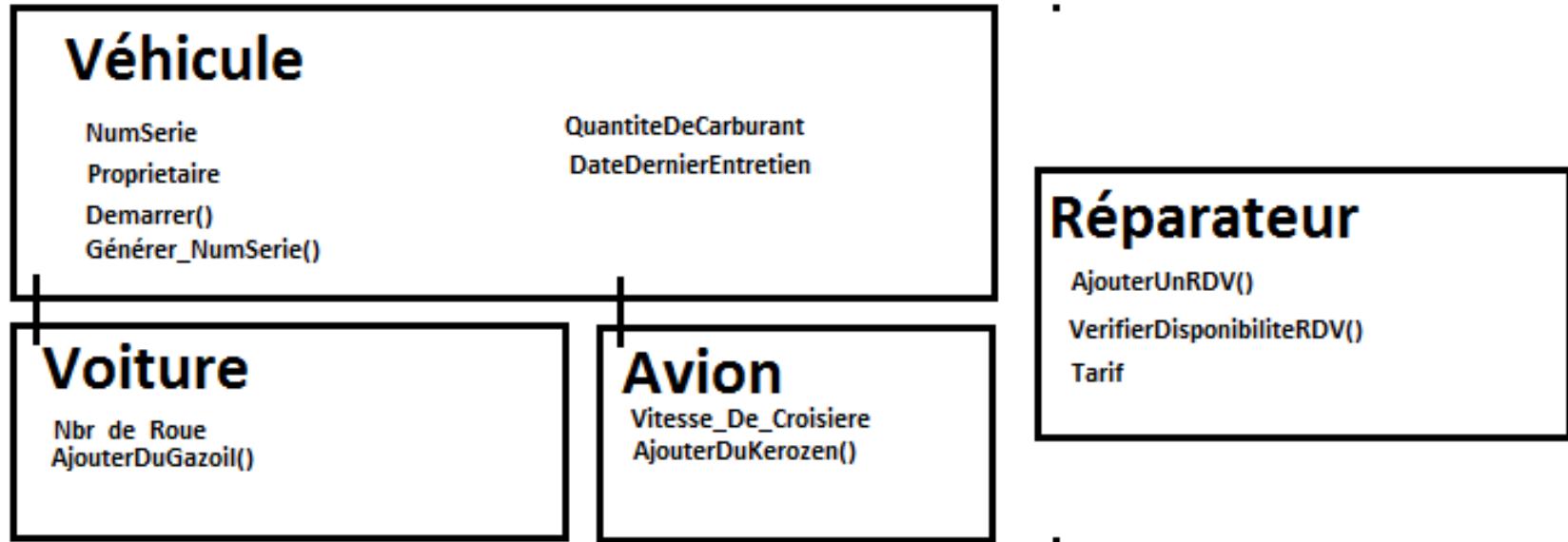
# LES MODIFICATEURS D'ACCÈS

- *public* :
  - L'élément est visible de tous les objets externe à la classe
- *private* :
  - L'élément n'est accessible qu'à l'intérieur d'une même classe
- *protected* :
  - L'élément est visible par les objets dérivés de la classe
- *internal*
  - Membre accessible dans l'Assembly uniquement
  - Valeur par défaut

QUESTIONS ? REMARQUES ?



# EXERCICE



## EXERCICE

- Seule la classe **Véhicule** peut générer un numéro de série lors de la construction de l'objet.
- Tout le monde peut **Demarrer()** un véhicule.
- Les réparateurs ont besoin de savoir le nombre de roues d'une voiture
- Les réparateurs n'ont pas besoin de connaître la vitesse de croisière d'un avion
- Les fonctions **AjouterDuKerozen()** et **AjouterDuGazoil()**, ont besoin de connaître la **QuantiteDeCarburant**, mais les réparateurs ne doivent pas le savoir.
- Par contre, les réparateurs peuvent ajouter du carburant
- Tout le monde peut prendre un RDV auprès d'un réparateur.
- Par contre seul la fonction **AjouterUnRDV()** peut vérifier les disponibilités
- Tous le monde peut vérifier les tarifs d'un réparateur
- Les réparateurs peuvent modifier la date d'entretien d'un véhicule.

# LECON 6

INTERFACES ET CLASSES ABSTRAITES

# INTERFACES

## Interface = Contrat

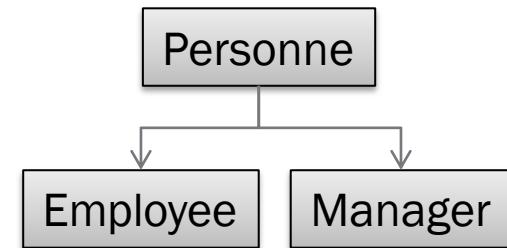
- Les classes qui héritent d'une interface ont obligatoirement les méthodes définies dans cette interface.
- Exemple :
  - On peut définir une interface *IGestionDesRoues*, avec la méthode *CalculerPressionDesPneus()*
  - Cette interface sera appliquée sur *Moto*, *Voiture*, *Avion* mais pas *Helico* ni *Bateau*.

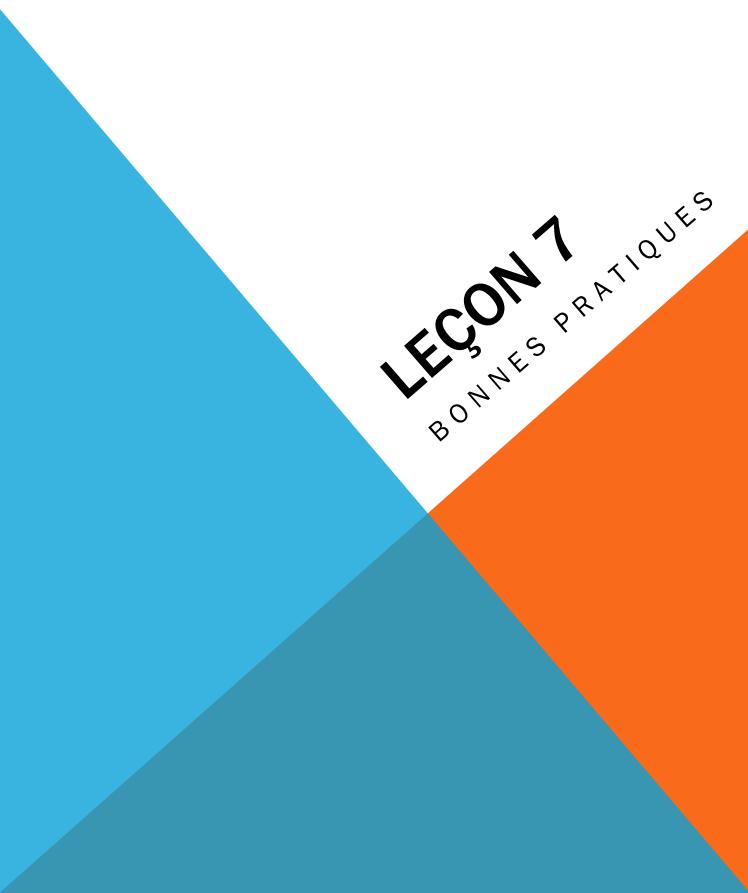
## CLASSE ABSTRAITE

Classe abstraite = classe qui ne peut pas être instanciée.

- Une classe abstraite est une classe qui peut contenir des méthodes non implémentées.
- Obligation aux classes dérivant d'une classe abstraite à l'implémentation.

Ex : La classe personne est une classe abstraite. Car une personne est forcement un employé ou un manager





# LEÇON 1

BONNES PRATIQUES

# PRINCIPE SOLID

	Définition
S	<u>Responsabilité unique</u> ( <u>Single Responsibility Principle</u> ) : une classe, une fonction ou une méthode doit avoir une et une seule responsabilité
O	<u>Ouvert/fermé</u> ( <u>Open/closed principle</u> ) : une classe doit être ouverte à l'extension, mais fermée à la modification
L	<u>Substitution de Liskov</u> ( <u>Liskov substitution Principle</u> ) : une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme
I	<u>Ségrégation des interfaces</u> ( <u>Interface segregation principle</u> ) : préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale
D	<u>Inversion des dépendances</u> ( <u>Dependency Inversion Principle</u> ) : il faut dépendre des abstractions, pas des implémentations

## AUTRES BONNES PRATIQUES

- DRY : Don't repeat yourself
- KISS : Keep it simple and stupid
- YAGNI : You aren't gonna need it

QUESTIONS ? REMARQUES ?



## EXERCICES

**Ecole :** Concevoir un projet objet permettant de représenter les différents acteurs gérés par une école.

Nous voulons ajouter dans une école des élèves, des profs et des administratifs.

- Les élèves ont un nom, un prénom, une moyenne et peuvent s'afficher.
- Les profs ont un nom, un prénom, une matière et peuvent s'afficher.
- Les Administratifs ont un nom, un prénom, une fonction et peuvent s'afficher.
- Une école a un nom.

Il est possible d'ajouter dans une école et d'afficher tous les membres de l'école.

Décrire l'architecture objet du projet : graphe d'héritage, composition et interface de chaque classe.

Rajouter ensuite à notion de Salarie, (un élève n'est pas un salarié)



# MODULE 8

LA PROGRAMMATION ORIENTÉE OBJET AVEC C#

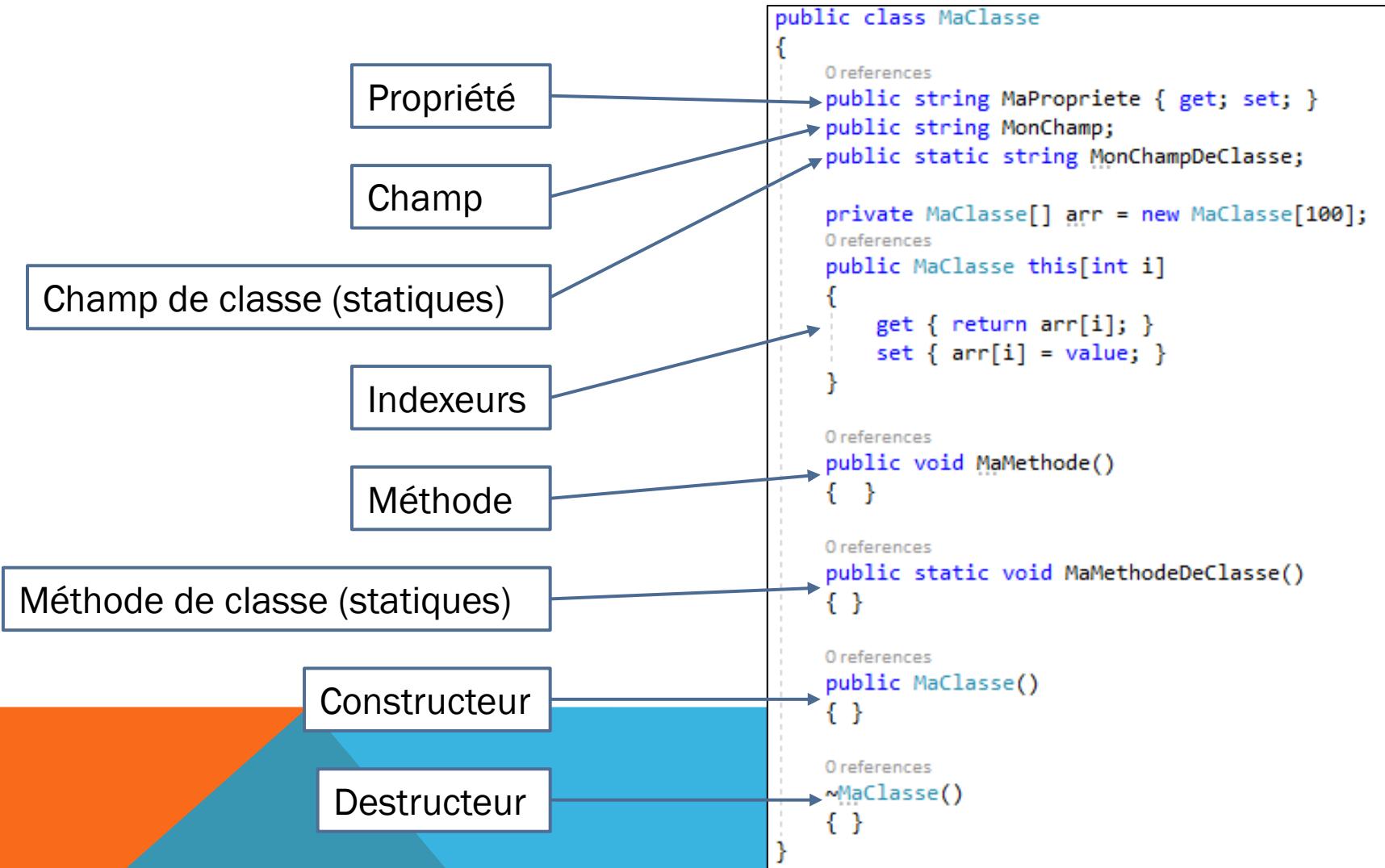




# LEÇON 1

COMPOSITION D'UNE CLASSE

# COMPOSITION D'UNE CLASSE EN CSHARP



## DÉFINITION D'UNE CLASSE

- Utilisation des champs/Méthodes statiques directement sur la classe.
- Utilisation des champs/Méthodes sur un objet instancié.
- Exemple :

```
public class Personne
{
    public DateTime DateDeNaissance;

    public static double Getage(Personne p)
    {
        return DateTime.Now.Year-p.DateDeNaissance.Year;
    }
}
```

```
Personne p = new Personne();
p.DateDeNaissance = new DateTime(1980, 04, 04);

double age = Personne.Getage(p);
```

QUESTIONS ? REMARQUES ?





# LEÇON 2

LES MÉTHODES

# LES MÉTHODES

- Les méthodes d'instance permettent les traitements dépendant des données de l'objet

```
public string GetInitials()
{
    return this.Nom[0].ToString() + this.Prenom[0].ToString();
}
```

- Les méthodes de classe sont précédées par le mot clé **static**
  - Indépendant des données de l'instance

```
class Article
{
    public string nom;
    public double prixUnitaire;

    public static double GetTVA()
    {
        return 0.20;
    }
}
```

Ici la TVA est commune à tous les articles.  
Alors que le prix est spécifique à un article

## SURCHARGE DE MÉTHODES

- Surcharger une méthode = possibilité de mettre 2 méthodes avec le même nom mais avec des paramètres différents

```
public Article CreationArticle()
{
    return new Article();
}

public Article CreationArticle(string nom)
{
    Article art = new Article();
    art.Nom = nom;

    return art;
}
```

QUESTIONS ? REMARQUES ?





# LEÇON 3

UTILISATION

## UTILISATION D'UNE CLASSE

- Le *Garbage Collector* s'occupe de la libération de la mémoire

```
static void Main(string[] args)
{
    Personne p = new Personne();
    p.DateDeNaissance = new DateTime(1980, 04, 04);

    double age = Personne.Getage(p);

    p = null;
}
```

- Il est possible d'aider le Garbage collector en déréférençant les objets au plus tôt.

La bonne pratique en csharp est de créer un objet juste avant son utilisation, et de le libérer dès que l'on s'en sert plus

## LES CONSTRUCTEURS

- Initialisation lors de l'instanciation.
- Permet d'initialiser et de sécuriser la création d'objets
- Un constructeur peut être statique
- Il peut y avoir plusieurs constructeurs

```
public class Personne
{
    public Personne()
    {
        Nom = "Toto";
        Prenom = "Jean";
    }

    public Personne(string nom)
    {
        Nom = nom;
        Prenom = "Jean";
    }
}
```

# EXPRESSION D'INITIALISATION

## Nouveauté C#3

- Seul les membres publiques sont accessibles
- Exemple :

```
Personne p1 = new Personne();
p1.DateDeNaissance = new DateTime(1980, 04, 04);
p1.Nom = "Toto";
p1.Prenom = "Jean";

Personne p2 = new Personne() { Nom = "Toto", Prenom = "Jean" };
```

## TYPES ANONYMES

### Nouveauté C#3

- Crée pour LINQ, très utiles pour manipuler des données d'un certain type temporairement.
- Exemple :

```
var p3 = new { LastName = "Toto", FirstName = "Jean", DateOfBirth = "05/05/1990" };
```

## DESTRUCTEUR

- Méthode particulière appelée lors de la libération d'un objet par le *Garbage Collector*

```
~Personne()
{
    Console.WriteLine(Nom + " vous dit bye bye");
}
```

## LES ÉGALITÉS

Qu'est ce qui doit être égal ?

- L'objet ? -> Utilisation de `==`
- La valeur de l'objet? -> Utilisation de `Object.Equals()`
- La méthode `Object.Equals()` utilise la méthode `GetHashCode()` pour calculer une valeur de comparaison

Un exemple de  
ça ?

# VISIBILITÉ DES CLASSES

## *public*

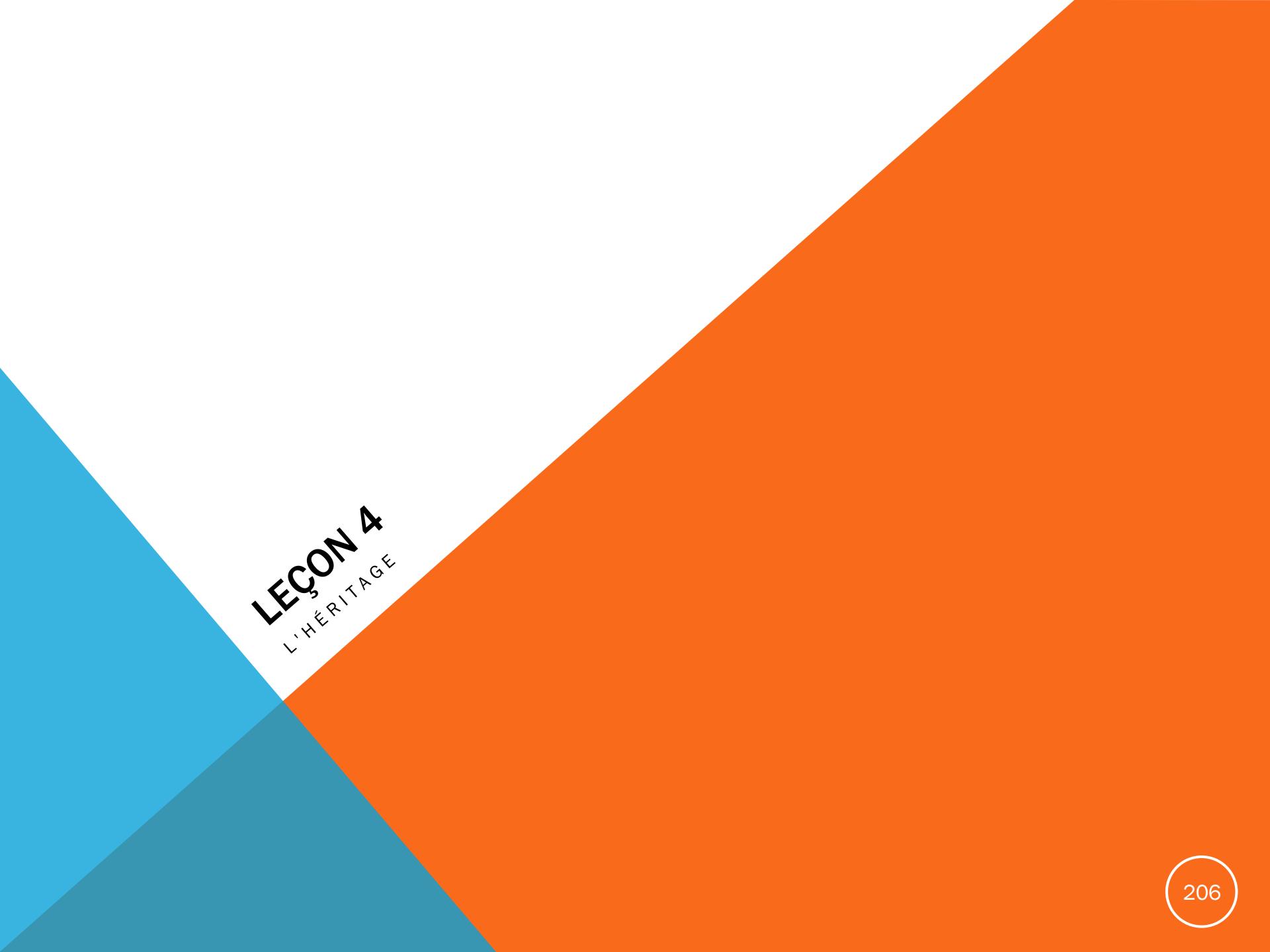
- La classe est visible par toutes les autres classes

## *internal*

- La classe est visible uniquement par les classes de son assembly (valeur par défaut)

QUESTIONS ? REMARQUES ?





# LEÇON 4

L'HÉRITAGE

## HÉRITAGE

- Une classe ne peut hériter que d'une seule autre classe
- La classe héritée acquiert les membres (publiques et protégés) de la classe de base
- Une classe héritée est de type de la classe de base
- Une classe de base n'est pas considérée comme étant d'un type hérité

```
class Employee:Personne
{
    double Salaire;
}
```

```
Personne John = new Employee("John",2300);

Employee personne = new Personne();
    Personne.Personne() (+ 1 surcharge(s))

Erreur :
Impossible de convertir implicitement le type 'ConsoleApplication2.Personne' en 'ConsoleApplication2.Employee'.
```

## HÉRITAGE DU CONSTRUCTEUR

- Initialisation de la classe parent lors de l'instanciation d'une classe dérivée.

```
Employee(string nom, double salaire)
    : base(nom)
{
    Salaire = salaire;
}
```

## SURCHARGE ET POLYMORPHISME

- Méthode de même signature dans la classe dérivée.
- Utilisation du mot clé **new** optionnel mais conseillé
- Utilisation de méthodes virtuels

```
public class Personne
{
    virtual void Afficher()
    {
        //Affiche les infos de la personne
    }
}
```

```
class Employee:Personne
{
    public override void Afficher()
    {
        Console.Out.WriteLine("infos de l'employee");
    }
}
```

## TEST DU TYPE

- Le type de l'objet peut être testé avec le mot clé **is**

```
object[] datas= new object[100];  
  
//....  
  
for(int i=0;i<100;i++)  
    if (datas[i] is Article)  
    {  
        //Do stuff  
    }
```

- Il est également possible de connaître le type par réflexion

```
for(int i=0;i<100;i++)  
    if (datas[i].GetType() == typeof(Article))  
    {
```

## INTERDIRE L'HÉRITAGE

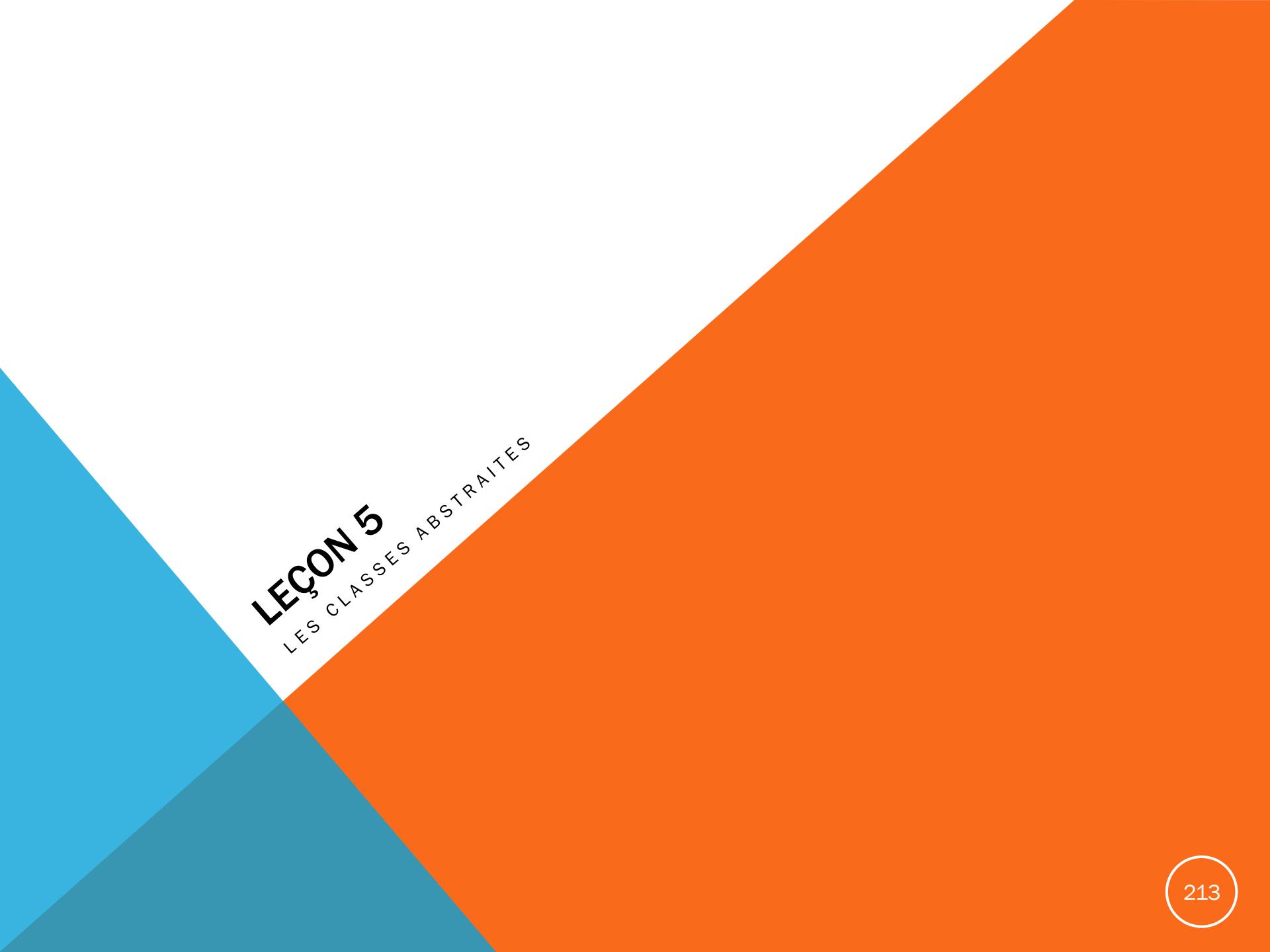
- Utilisation du mot clé **sealed**

```
public sealed class Personne  
{
```

```
class Employee:Personne  
{  
    'ConsoleApplication2.Employee' : impossible de dériver du type sealed 'ConsoleApplication2.Personne'
```

QUESTIONS ? REMARQUES ?





# LEÇON 5

LES CLASSES ABSTRAITES

## LES CLASSES ABSTRAITES

- = Classes dont certains membres ne sont pas implémentés.
- Une classe abstraite ne peut pas être directement instanciée.
- Obligation de définir les membres abstraits lors de la dérivation.
- Mots clé **abstract** et **override**

```
public abstract class Personne
{
    public abstract double GetSalaire();
```

```
class Employee:Personne
{
    public override double GetSalaire()
    {
        return 1200;
    }
}
```

## INTERFACES

- = Contrats pour les classes
- Ne peut pas être instancié
- Pas de champs
- Pas de corps de méthode
- Une interface définit des membres public uniquement
- Le nom commence par un « **I** » par convention

# INTERFACES

- Utilisation par héritage comme pour une classe abstraite

```
interface ISalarie
{
    double GetSalaire();
}
```

```
class Employee:Personne,ISalarie
{
    public double GetSalaire()
    {
        return 1200;
    }
}
```

QUESTIONS ? REMARQUES ?



# LECON 6

L'ENCAPSULATION EN CSHARP

## NOTION D'ENCAPSULATION

- L'encapsulation permet de protégé la donnée en évitant qu'une variable ne prenne des valeur non souhaitées
- Exemple :

```
public class Eleve : Personne
{
    private int note; ←
    public int GetNote()
    {
        return note;
    }

    public void SetNote(int nte)
    {
        if (nte >= 0 && nte <= 20) note = nte;
        else throw new ArgumentException("La note doit être compris en 0 et 20");
    }
}
```

La note sera toujours entre 0 et 20

# PROPRIÉTÉS

- Exécute une action lors de l'affectation ou la récupération d'une variable.
- Une propriété améliore la lecture et l'usage de la donnée

```
private int note;
public int Note
{
    get {
        return note;
    }
    set
    {
        if (value >= 0 && value <= 20) note = value;
        else throw new ArgumentException("La note doit être compris en 0 et 20");
    }
}
```

Une propriété permet d'encapsuler un champ. Et donc de protéger les valeurs possibles prisent par ce champ

## VISIBILITÉ DES PROPRIÉTÉS

### Nouveauté C# 2.0

- Les accesseurs **get** et **set** peuvent également avoir des modificateurs d'accès.

```
public int Id { get; private set; }
public string Nom { get; set; }
public string Prenom { get; set; }
```

# PROPRIÉTÉS AUTOMATIQUES

Nouveauté de C# 3.0

- Propriété automatique :

```
0 references
public string MaPropriete { get; set; }
```

- Équivalent à :

```
private string maPropriete;
0 references
public string MaPropriete
{
    get { return maPropriete; }
    set { maPropriete = value; }
}
```

- Permet aux instances d'une **classe** ou d'une **struct** d'être indexés comme des tableaux.
- Se comporte comme un tableau

```
public class ListPersonnes
{
    List<Personne> lst = new List<Personne>();

    public Personne this[int idx]
    {
        get
        {
            return lst[idx];
        }
    }
}
```

```
ListPersonnes lst = new ListPersonnes();
Personne p = lst[4];
```

QUESTIONS ? REMARQUES ?





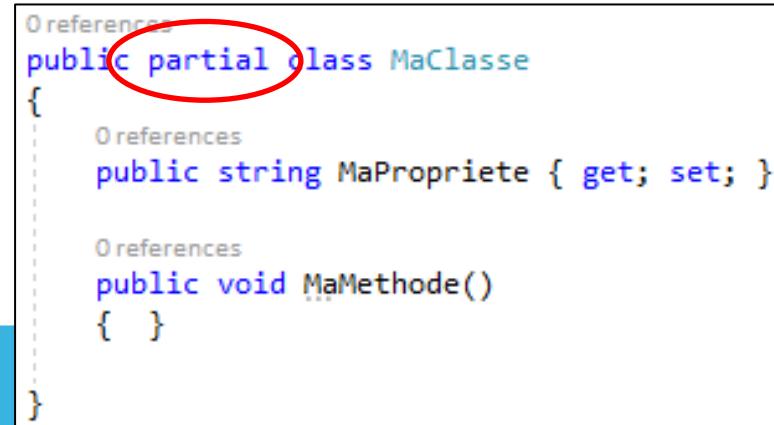
# LEÇON 1

TYPES DE CLASSES

## CLASSES PARTIELLES

### Nouveauté C# 2.0

- Très utilisé par Visual Studio pour la génération de code
- Facilite le travail en équipe
- Permet l'extension d'une classe
- Mot clé **partial** (Partiel)



The image shows a screenshot of a C# code editor. A red circle highlights the word "partial" in the first line of the code. The code defines a partial class named "MaClasse" with a single method "MaMethode".

```
public partial class MaClasse
{
    public string MaPropriete { get; set; }

    public void MaMethode()
    {
    }
}
```

# LES CLASSES STATIQUES

Nouveauté de C# 2.0

- Instanciation impossible.
- Tous les membres sont également statiques
- Mots clés **static**

```
public static class MgmtPersonnes
{
    static void AddPersonne(Personne p)
    {
        //Ajout d'une personne
    }
}
```

QUESTIONS ? REMARQUES ?



## EXERCICES

**Ecole 2 :** Création d'un projet console, et d'un projet bibliothèque de classe. Le projet console faisant référence au projet de bibliothèque de classe.

Création d'un dossier de **BusinessObject** et d'un dossier **BusinessLayer** dans le projet **Bibliothèque** de classe.

Implémentation des classes de l'école vu précédemment en C# dans le dossier **BusinessObject**

Implémentation de la classe **EcoleService** contenant la méthode **Inscrire()** et **Embaucher()** permettant d'inscrire un élève et d'embaucher un prof ou un administratif.

Ces méthodes se contentent d'ajouter les membres dans une liste contenue dans **EcoleService**



# **MODULE 9**

AUTRES ÉLÉMENTS DU LANGAGE





# LEÇON 1

LES MÉTHODES D'EXTENSION

## MÉTHODES D'EXTENSION

- Utiliser pour étendre plusieurs classes dérivant d'une interface
- Utiliser pour étendre une classe dont on a pas les sources.
- Méthode statique
- Utilisation du mot clé **this**
- Le corps de la méthode d'extension peut être paramétrable avec des expressions lambda

# MÉTHODES D'EXTENSION

Exemple de méthode d'extension :

- Ajout de la fonction ‘ToFullString()’ au type ‘int’

```
public static class IntExtension
{
    public static string ToFullString(this int nombre)
    {
        return nombre.ToString() + " est un entier";
    }
}
```

```
private static void Exemple_Methode_Extender()
{
    int entier = 5;
    entier.
}
```

An IntelliSense dropdown menu is displayed over the variable 'entier'. It lists several methods: CompareTo, Equals, GetHashCode, GetType, GetTypeCode, ToFullString, and ToString. The 'ToFullString' method is highlighted with a blue selection bar, and its tooltip '(extension) string int.ToFullString()' is visible to the right of the menu.

# MÉTHODES D'EXTENSION

Exemple de méthode d'extension :

- Méthode d'extension avec valeur par défaut paramétrable

```
public static class IDictionnaryExtention
{
    public static TValue GetValue<TKey, TValue>(
        this IDictionary<TKey, TValue> aList, TKey key, TValue defaultvalue)
    {
        TValue valeurretour;

        if (aList.ContainsKey(key))
            valeurretour = aList[key];
        else
            valeurretour = defaultvalue;

        return valeurretour;
    }
}
```

```
Dictionary<int, string> dico = new Dictionary<int, string>();

dico.Add(1, "Formation .NET");
dico.Add(2, "Formation WinForm");
dico.Add(3, "Formation WebForm");
dico.Add(4, "Formation MVC");

Console.Out.WriteLine(dico.GetValue(34,"Formation inconnue !"));
```

Formation inconnue !

## MÉTHODES D'EXTENSION

*Cas particulier :*

- Méthode étendue masqué par une méthode de la classe de même signature.
- 1 classe ayant 2 interfaces possédant chacun 1 extension de même signature, provoque une erreur.

## MÉTHODES D'EXTENSION

Quelques méthodes d'extension incluses au Framework.

- All
- Any
- Concat
- Count
- Select
- Where
- Distinct
- First
- FirstOrDefault
- GroupBy
- Intersect
- Join
- OrderBy(Descending)/ThenBy(Descending)
- ToList

QUESTIONS ? REMARQUES ?



# LEÇON 2

LES PARAMÈTRES NOMMÉS ET OPTIONNELS

# PARAMÈTRES NOMMÉS ET OPTIONNELS

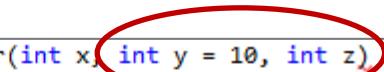
- Permet une souplesse d'écriture du code.
- Ils sont disponibles aux :
  - Méthodes
  - Constructeurs
  - Indexeurs

```
public static void Calculer(int x, int y = 10, int z = 20)
{
    Console.Out.Write(x + " +y+ " +z+ " = ");
    Console.Out.WriteLine((x + y + z).ToString());
}
```

## PARAMÈTRES NOMMÉS ET OPTIONNELS

- Si un paramètre est optionnel, tous les autres paramètres après (à droite) doivent l'être également

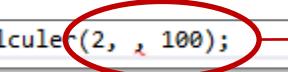
```
public static void Calculer(int x, int y = 10, int z)
{
    Console.Out.WriteLine(x + " + " + y + " + " + z + " = ");
    Console.Out.WriteLine((x + y + z).ToString());
}
```



- L'utilisation des 'options' se fait en omettant le paramètre lors de l'appel.

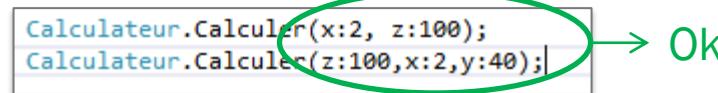
```
Calculateur.Calculer(2);
Calculateur.Calculer(2, 40);
Calculateur.Calculer(2, 40, 100);
```

```
Calculateur.Calculer(2, , 100);
```



## PARAMÈTRES NOMMÉS ET OPTIONNELS

- Il est possible d'appeler une fonction avec les paramètres dans le désordre si on les nomme
- Dans ce cas il est possible d'omettre des paramètres optionnels.



```
Calculateur.Calculer(x:2, z:100);
Calculateur.Calculer(z:100,x:2,y:40);
```

# PARAMÈTRES NOMMÉS ET OPTIONNELS

*La résolution des conflits :*

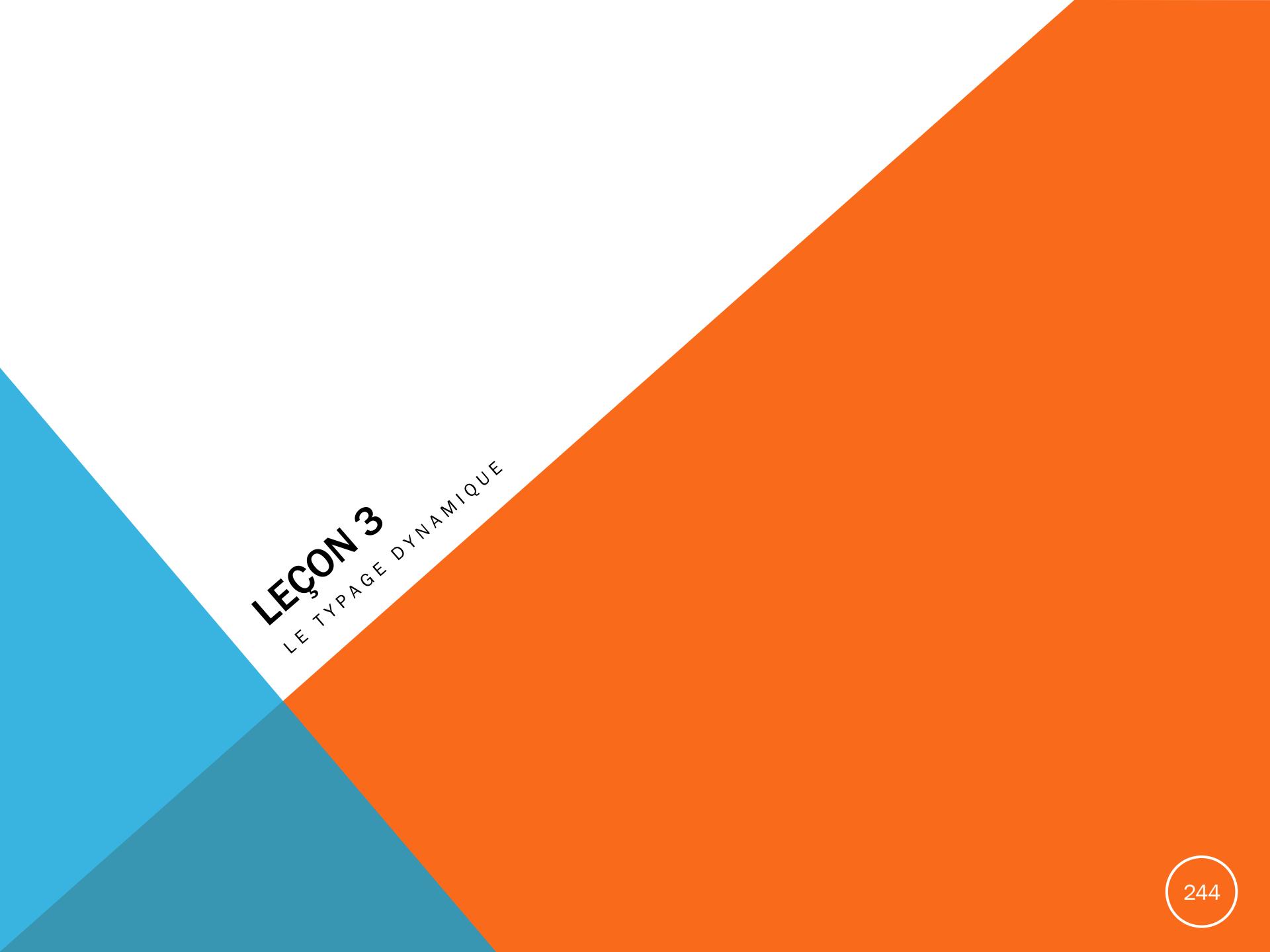
- La méthode choisie est la méthode qui correspond le mieux à la signature de la fonction

```
public static void LaunchMe(string s, int i = 1) { }
public static void LaunchMe(object obj) { }
public static void LaunchMe(int i, string s = "default") { }
public static void LaunchMe(int i) { }

public static void EnConflit()
{
    LaunchMe(14);
}
```

QUESTIONS ? REMARQUES ?



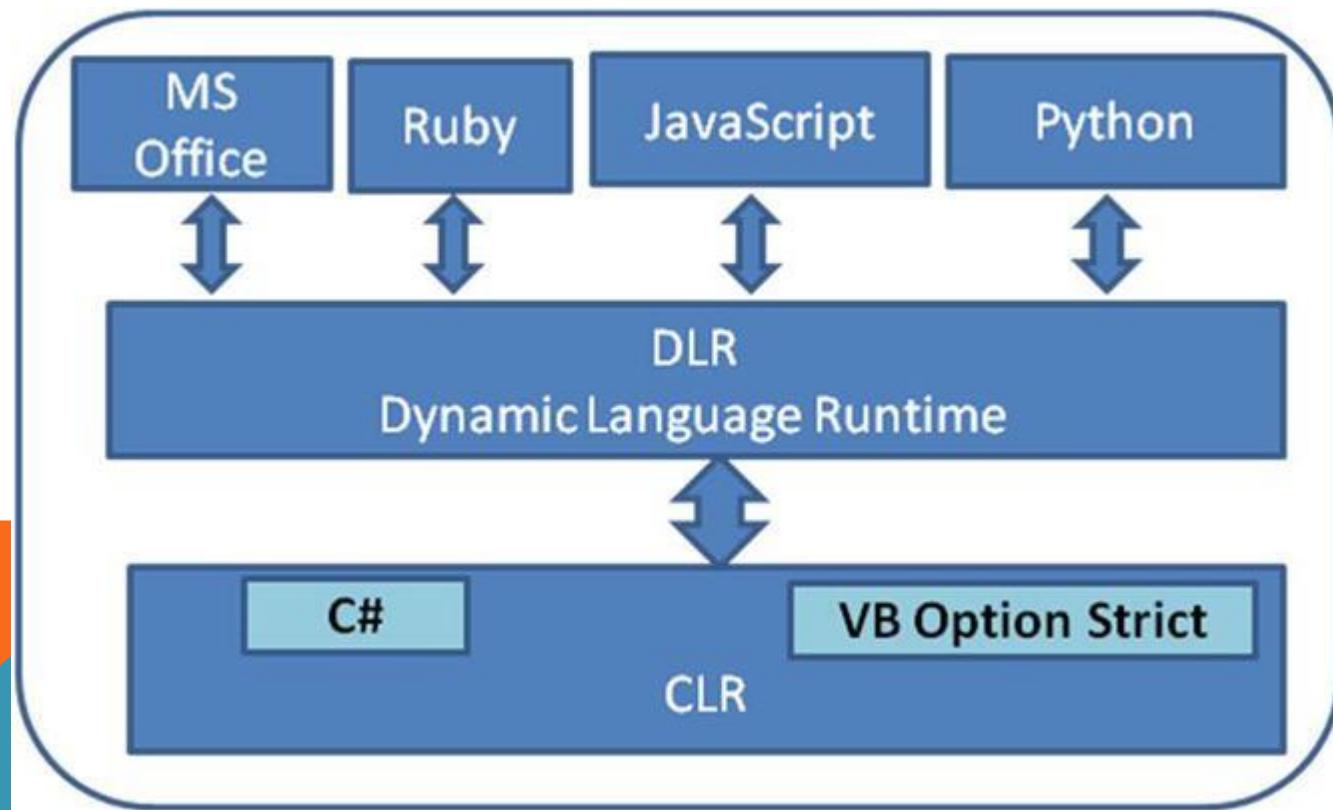


# LEÇON 3

LE TYPAGE DYNAMIQUE

## TYPAGE DYNAMIQUE ET LA DLR

- Modification de la CLR pour y inclure la couche DLR
- Permet l'ouverture du Framework.NET à des langages non fortement typés (IronPython ou IronRuby)
- Avancé majeur du Framework pour l'ouverture vers le Web



# TYPAGE DYNAMIQUE ET LA DLR

## Mot clé *dynamic*

- Le type sera évalué à l'exécution

The diagram illustrates the execution flow of a dynamic variable. On the left, a code snippet is shown in a light gray box:

```
dynamic var1 = 5;  
  
var1 = var1 + "Hello";  
  
Console.Out.WriteLine("test = " + var1);  
  
var1 = 9.5f;  
  
var1 = var1 + 7;  
  
Console.Out.WriteLine("total : " + var1);
```

Three red arrows point from the code to the output on the right, which is enclosed in a black box:

```
test = 5Hello  
total : 16,5
```

# TYPAGE DYNAMIQUE ET LA DLR

*Exemple d'utilisation en tant que paramètre :*

```
public class Personne
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int Age { get; set; }
}

public class Animal
{
    public string Nom { get; set; }
    public string Age { get; set; }
}

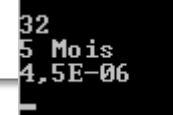
public static class AgeClass
{
    public static void ShowAge(dynamic obj)
    {
        Console.Out.WriteLine(obj.Age);
    }
}
```

```
Personne p = new Personne() { Nom = "DOE", Prenom = "John", Age = 32 };
AgeClass.ShowAge(p);

Animal a = new Animal() { Nom = "Medor", Age = "5 Mois" };
AgeClass.ShowAge(a);

var anonymObject = new { Type = "Photon", Age = 0.0000045f };
AgeClass.ShowAge(anonymObject);

Console.ReadLine();
```



```
32
5 Mois
4.5E-06
```

## TYPAGE DYNAMIQUE ET LA DLR

*dynamic*, mais :

- Utilisation déconseillé
- Aucune vérification du compilateur
- Engendre de fort risque de bug non prévu

QUESTIONS ? REMARQUES ?



# **MODULE 10**

ACCÈS AUX DONNÉES



# LEÇON 1

ADO.NET

## INTRODUCTION

- L'accès au donnée avec .NET se fait grâce à ADO.Net
- ADO.Net = System.Data
- Ensemble de classes indépendantes + ensemble de classes surchargée dépendantes de la Base de donnée

## PRINCIPALES CLASSES

**Connection** : Connection à la base de donnée

**Command** : Création de la requête SQL

**CommandBuilder** : Constructeur de command

**DataReader** : Lecteur sous forme de flux des données retournées

**DataAdapter** : Mise en forme des données retournées

*Version OLEDb, version SQLServer*

*ADO.NET est extensible*

## LES CLASSES « CLIENT »

**DataSet** :Représentation mémoire des données de la base

**DataTable** :représente les tables d'un DataSet

**DataColumn** :représente les colonnes d'un DataSet

**DataRow** :représente une ligne d'une table

**DataRelation** :représente une relation existant entre les deux tables d'un DataSet

QUESTIONS ? REMARQUES ?



# LEÇON 2

LECTURE DES DONNÉES

## LECTURE PAR FLUX

Accès en lecture seul, lecture en avant seulement

- Connection
- Command
- DataReader

## EXEMPLE DE LECTURE PAR FLUX

```
public static void GetElements()
{
    OleDbConnection cnx = new OleDbConnection(
        @"Provider=Microsoft.Jet.OLEDB.4.0;" +
        @"Data source=c:\temp\Northwind.mdb");
    OleDbCommand cmd = cnx.CreateCommand();
    cmd.CommandText = "select * from Customers where Country='France' and [PostalCode] like '75%'";
    cnx.Open();
    OleDbDataReader dr = cmd.ExecuteReader();
    cmd.Dispose();
    while (dr.Read())
    {
        Console.WriteLine("Enregistrement de la société \'{0}\'", dr["CompanyName"]);
        for (int i = 0; i < dr.FieldCount; i++)
            Console.WriteLine("{0} = {1}", dr.GetName(i), dr[i]);
        Console.WriteLine();
    }
    dr.Close();
    cnx.Close();
    cnx.Dispose();
}
```

## CHARGEMENT EN MÉMOIRE

- Utilisation des DataSets
- Accès en lecture/écriture
- Moins Performant mais plus souple que DataReader  
(schéma complet récupéré)
- Indépendant de la BD
- Utilisation de DataAdaptater pour Remplir un DataSet

# CHARGEMENT EN MÉMOIRE

```
public static void GetElements()
{
    OleDbConnection cnx = new OleDbConnection(
        @"Provider=Microsoft.Jet.OLEDB.4.0;" +
        @"Data source=c:\temp\Nwind.mdb");
    OleDbCommand cmd = cnx.CreateCommand();
    cmd.CommandText = "select * from Clients where Pays='France' " +
        "and [Code postal] like '75%'";
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    DataSet ds = new DataSet();
    cnx.Open();
    da.Fill(ds, "Clients");
    da.Dispose();
    cmd.Dispose();
    cnx.Dispose();
    for (int n = ds.Tables["Clients"].Rows.Count; n > 0; n--)
    {
        DataRow record = ds.Tables["Clients"].Rows[n - 1];
        Console.WriteLine("Enregistrement de la société \"{}\"", record["Société"]);
        for (int i = 0; i < ds.Tables["Clients"].Columns.Count; i++)
            Console.WriteLine("{} = {}", ds.Tables["Clients"].Columns[i].ColumnName,
                record[i]);
        Console.WriteLine();
    }
    ds.Dispose();
}
```

QUESTIONS ? REMARQUES ?



# LECON 3

MISE À JOUR DES DONNÉES

## MISE A JOUR DES DONNÉES

- Utilisation de CommandBuilder
- Utilisation du schema du DataSet pour générer la commande
- Fonction Update pour mettre à jour

```
public static void Update()
{
    OleDbConnection cnx = new OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;Data source=c:\temp\NWIND.MDB");
    cnx.Open();
    OleDbCommand cmd = cnx.CreateCommand();
    cmd.CommandText = "select * from Customers where Country='France' and [PostalCode] like '75%'";
    OleDbDataAdapter da = new OleDbDataAdapter(cmd);
    OleDbCommandBuilder cb = new OleDbCommandBuilder(da);
    cb.QuotePrefix = "[";
    cb.QuoteSuffix = "]";
    DataSet ds = new DataSet();
    da.Fill(ds, "Customers");
    for (int n = 0; n < ds.Tables["Customers"].Rows.Count; n++)
    {
        DataRow record = ds.Tables["Customers"].Rows[n];
        if ((string)(record["CustomerID"])) == "PARIS")
        {

            Console.WriteLine("Avant :{0}", record["ContactName"]);
            record["ContactName"] = "Gérard Mensoif";
            da.Update(ds, "Customers");
            Console.WriteLine("Après :{0}", record["ContactName"]);
            Console.WriteLine("Je viens d'exécuter la commande : {0}",
                cb.GetUpdateCommand().CommandText);
            break;
        }
    }
    cb.Dispose();
    ds.Dispose();
    da.Dispose();
    cnx.Close();
    cnx.Dispose();
}
```

```
Dim cnx = New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data source=c:\temp\nwind.mdb")
cnx.Open()
Dim cmd = cnx.CreateCommand()
cmd.CommandText = "select * from Customers where Country='France' and (PostalCode) like '75%'"
Dim da = New OleDbDataAdapter(cmd)
Dim cb = New OleDbCommandBuilder(da)
cb.QuotePrefix = "("
cb.QuoteSuffix = ")"
Dim ds = New DataSet()
da.Fill(ds, "Customers")
For n As Integer = 0 To ds.Tables("Customers").Rows.Count
    Dim record = ds.Tables("Customers").Rows(n)
    If (CType(record("CustomerID"), String) = "PARIS") Then
        Console.WriteLine("Avant :{0}", record("ContactName"))
        record("ContactName") = "Gérard Mensoif"
        da.Update(ds, "Customers")
        Console.WriteLine("Après :{0}", record("ContactName"))
        Console.WriteLine("Je viens d'exécuter la commande : {0}",
            cb.GetUpdateCommand().CommandText)

    End If
Next
cb.Dispose()
ds.Dispose()
da.Dispose()
cnx.Close()
cnx.Dispose()
```

QUESTIONS ? REMARQUES ?





# LECON 4

INTRODUCTION À LINQ

## PRINCIPE DE LINQ

- Ensemble de fonctionnalité du langage permettant de manipuler facilement un ensemble de données
- Fonctionne autour des interfaces :
  - ***IEnumerable*** : manipulation de la donnée
  - ***IQueryable*** : manipulation de la requête (expression)
- Méthode d'extension de Linq :
  - ***Select()***, ***Where()***, ***First()***, ***FirstOrDefault()***, etc ...

## LINQ TO SQL

- Permet donc également de manipuler directement la base sans SQL.
- Fortement typé
- DataContext = Classe « masquant » toutes la mécanique de LINQ to SQL
  - Gestion des connections
  - Gestion de la lecture/écriture des données
  - Contient les entités (représentation des objets d'une table de la base)

QUESTIONS ? REMARQUES ?





# LECON 5

ENTITY FRAMEWORK CORE

# PRÉSENTATION ENTITY FRAMEWORK

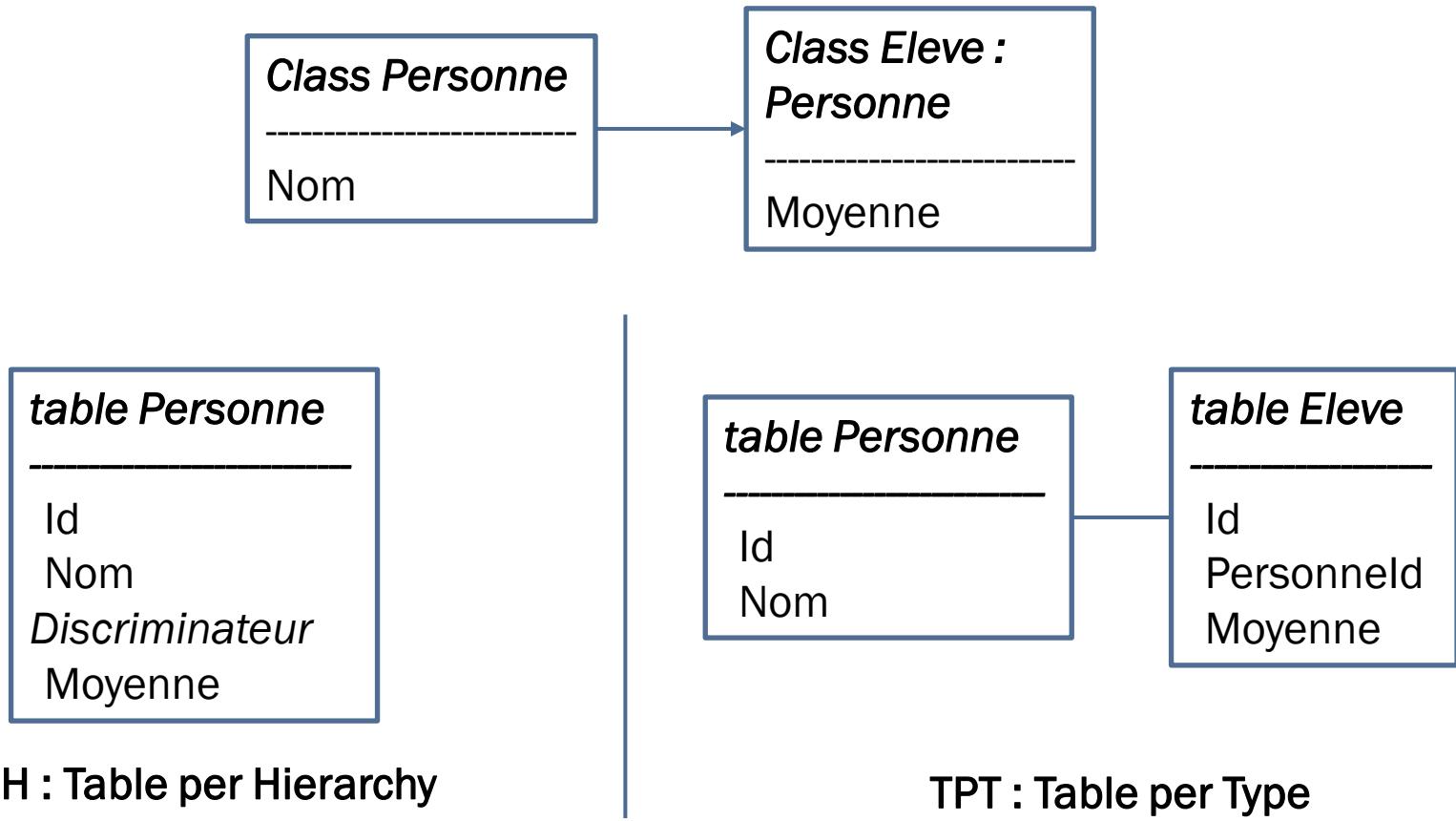
- ORM = Object-Relationnal Mapping
  - Fait le lien entre le monde relationnel (table, colonne, ligne => BDD) et le monde objet (classe, héritage, etc)
- Historiquement 2 modes de liaison à la base de données
  - Database First
    - N'existe plus en version Core
  - Code First
    - Permet les migrations

# PRÉSENTATION ENTITY FRAMEWORK

- Mécanisme d'attribut
- OnModelCreating
- ModelSeed
- Fonctionne de manière transactionnelle (SaveChanges())
- Gestion des accès concurrentiels via attributs

```
[ConcurrencyCheck]
0 references
public DateTime UpdateDate { get; set; }
```

# GESTION DE L'HÉRITAGE DANS EF



## CRUD EFCORE

- CRUD
  - **Create** : Ajoute des données en base (Insert)
  - **Read** : Lit des données en base (Select)
  - **Update** : Met à jour des données dans la base (Update)
  - **Delete** : Supprimer des données en base (Delete)
  - = 4 action de base de manipulation de données

```
public static void Insert(Personne pers)
{
    using (ProjEcoleDbContext ctx = new ProjEcoleDbContext())
    {
        pers.DateUpdate = DateTime.Now;
        ctx.Personnes.Add(pers);
        ctx.SaveChanges();
    }
}
```

# BONNE PRATIQUE EFCORE

- Utilisation d'un repository
- Une instance *DbContext* unique par fonction
  - A sortir de l'injection de dépendance
  - Valable pour une majorité de cas
  - Attention au mécanisme de tracking d'EF

```
public class Repository
{
    1 reference
    public void Insert<T>(T entity) where T : class
    {
        using var _dbctx = new EcoleDbContext();
        _dbctx.Set<T>().Add(entity);
        _dbctx.SaveChanges();
    }

    1 reference
    public List<T> GetAll<T>() where T : class
    {
        using var _dbctx = new EcoleDbContext();
        return _dbctx.Set<T>().ToList();
    }

    1 reference
    public void Delete<T>(T entity) ...
}

1 reference
public T GetById<T>(int id) ...
}
```

# DÉMONSTRATION DE EF CORE CODEFIRST



## EXERCICES

*Ecole :*

Création d'un dossier **DataAccessLayer** dans le projet de bibliothèque de classe.

Création d'une base de donnée via EF Core Code First pour stocker les élèves.

Création d'une classe permettant de faire un CRUD (**Create, Retrieve, Update, Delete**) sur les élèves en base de donnée.



# MODULE 11

LES FRAMEWORKS CLIENT LOURD



# LEÇON 1

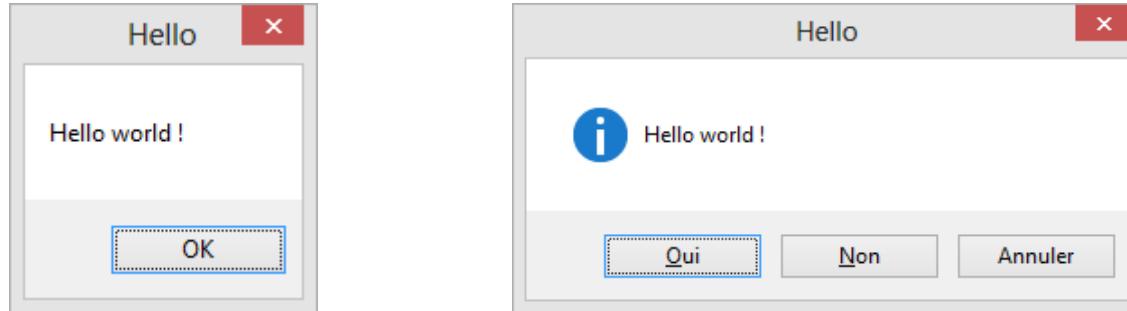
PRÉSENTATION DE WINFORM

## DÉFINITIONS

- Plateforme de développement .NET pour les applications fenêtrées.
- Existe depuis .NET 1.0
- *Windows Form* s'articule autour des formulaires (**Forms**)
- Le formulaire n'est pas l'application !!
- System.Windows.Form

# MESSAGEBOX

- Affichage de la boite de dialogue standard Windows



- Utilisation de la méthode Show
- Surcharge de la méthode pour affiner
  - Boutons, icônes, texte...

```
MessageBox.Show("Hello world !", "Hello");
```

```
MessageBox.Show("Hello world !", "Hello",
    MessageBoxButtons.YesNoCancel,
    MessageBoxIcon.Asterisk,
    MessageBoxDefaultButton.Button1);
```

## FENÊTRES ET ÉVÉNEMENTS

- C'est l'utilisateur qui exécute le programme
- L'utilisateur envoie des évènements au programme.
- Les évènements et délégués sont au cœur des formulaires.
- Un délégué lié à un évènement est un Handler
- Les évènements se gèrent comme des délégués
  - *Sender* est l'objet qui a signalé l'évènement
  - *e* contient les infos supplémentaires liées à l'évènement

```
public Form1()
{
    InitializeComponent();

    this.Load += new System.EventHandler(this.Form1_Load);
}

private void Form1_Load(object sender, EventArgs e)
{
}
```

## LES CONTRÔLES

- Hérite de *System.Windows.Form.Control*
- Comportements graphique
- Interactions avec le système et l'utilisateur grâce aux évènements
- Les contrôles appartiennent à un formulaire
- Designer.cs

# DÉMONSTRATION



QUESTIONS ? REMARQUES ?



# LECON 2

LE RENDU ET LA DISPOSITION EN WPF

# INTRODUCTION À WPF

WPF est une évolution de WinForm

- Utilisation du XAML
- Graphiquement plus riche
- Permet plusieurs types d'applications:
  - Boites de dialogue
  - Navigateur
  - Hébergé (XBAP)
  - Silverlight

# MODÈLE DE MISE EN PAGE WPF

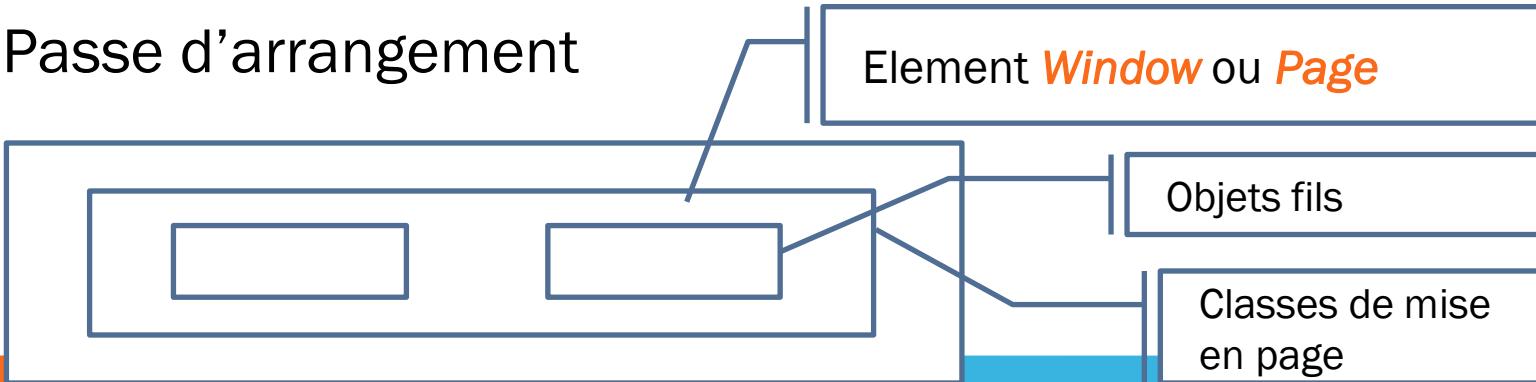
## 1 Passe de mesure

Hello World!

- cadre de délimitation rectangulaire supposé
- Recupéré par un appel à **GetLayout** sur le **FrameworkElement**

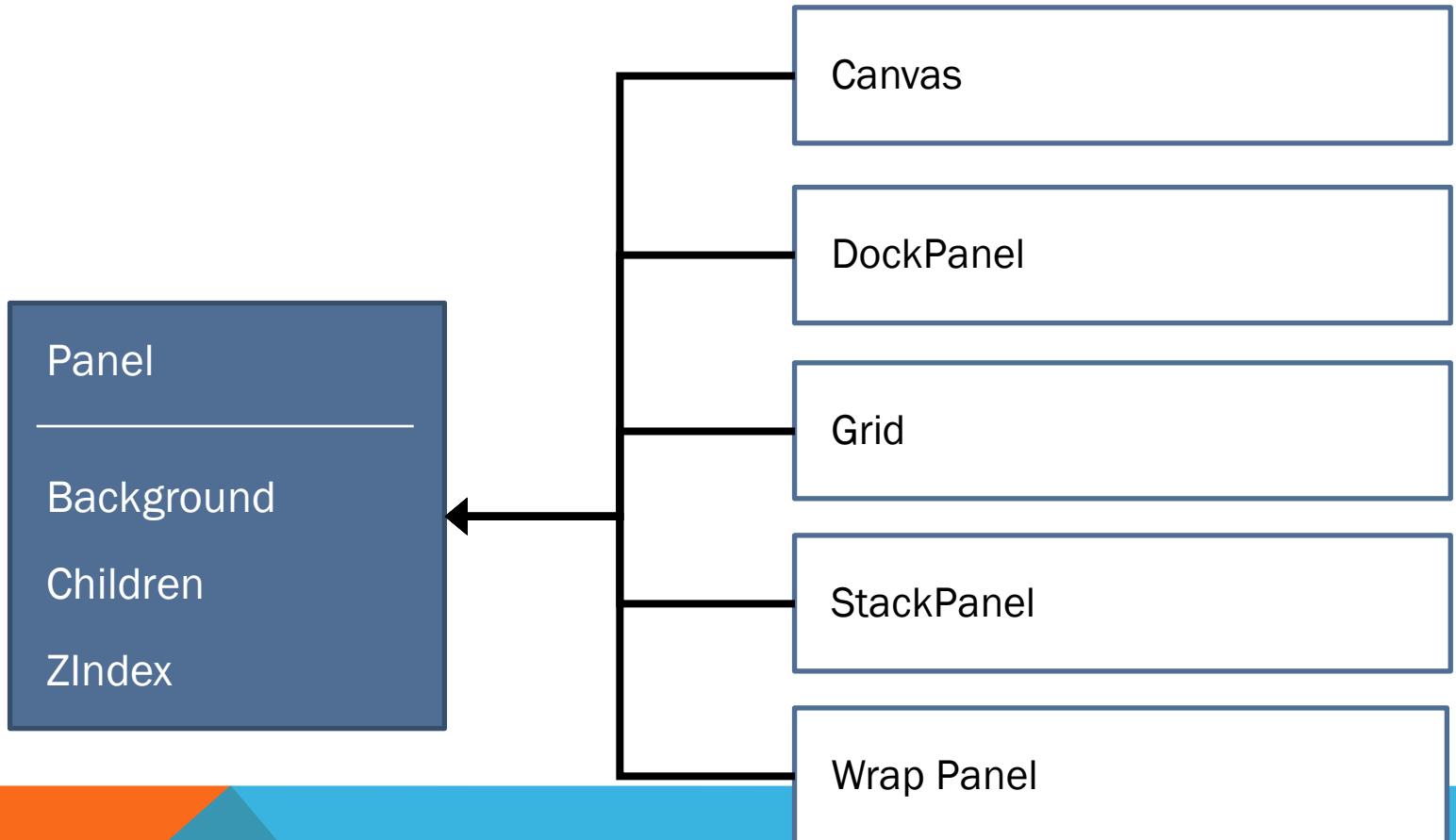
Evaluation de chaque membre de la collection des enfants pour déterminer la “**DesiredSize**”

## 2 Passe d'arrangement



Determination de la taille finale de chaque élément enfant et placement dans l'élément de mise en page.

# LES CLASSES DE MISE EN PAGE



## DÉMONSTRATION

- Classe *Canvas*
- Classe *StackPanel*
- Classe *WrapPanel*
- Classe *DockPanel*
- Classe *Grid*

QUESTIONS ? REMARQUES ?

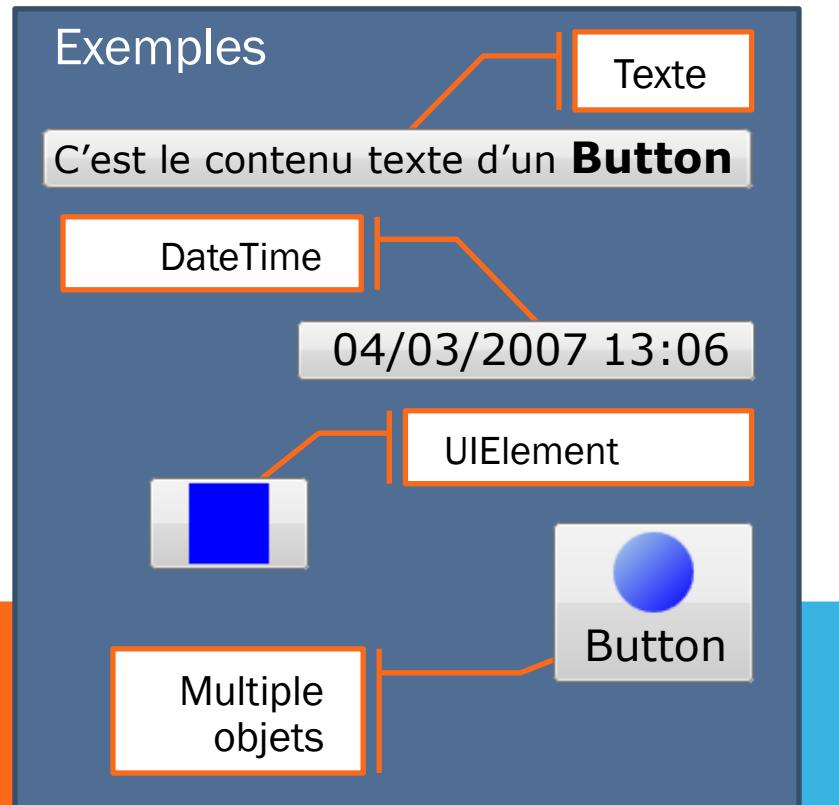


# LEÇON 3

LES CONTRÔLES EN WPF

# LES CONTRÔLES DE CONTENEUR SIMPLE

- Contient un seul élément
- Contient une propriété “Content”



## Controles communs:

- Button
- CheckBox
- GroupItem
- Label
- RadioButton
- RepeatButton
- ToggleButton
- ToolTip

# LES CONTRÔLES CONTENEUR À EN-TÊTE

- Contrôles conteneur spécialisés
- Contient une propriété “Content”
- Contient une propriété “Header”

Les contrôles conteneur à en-tête :

- Expander
- GroupBox
- TabItem

TabItem  
header

• Exemple

Page 1 • Page 2

GroupBox Header

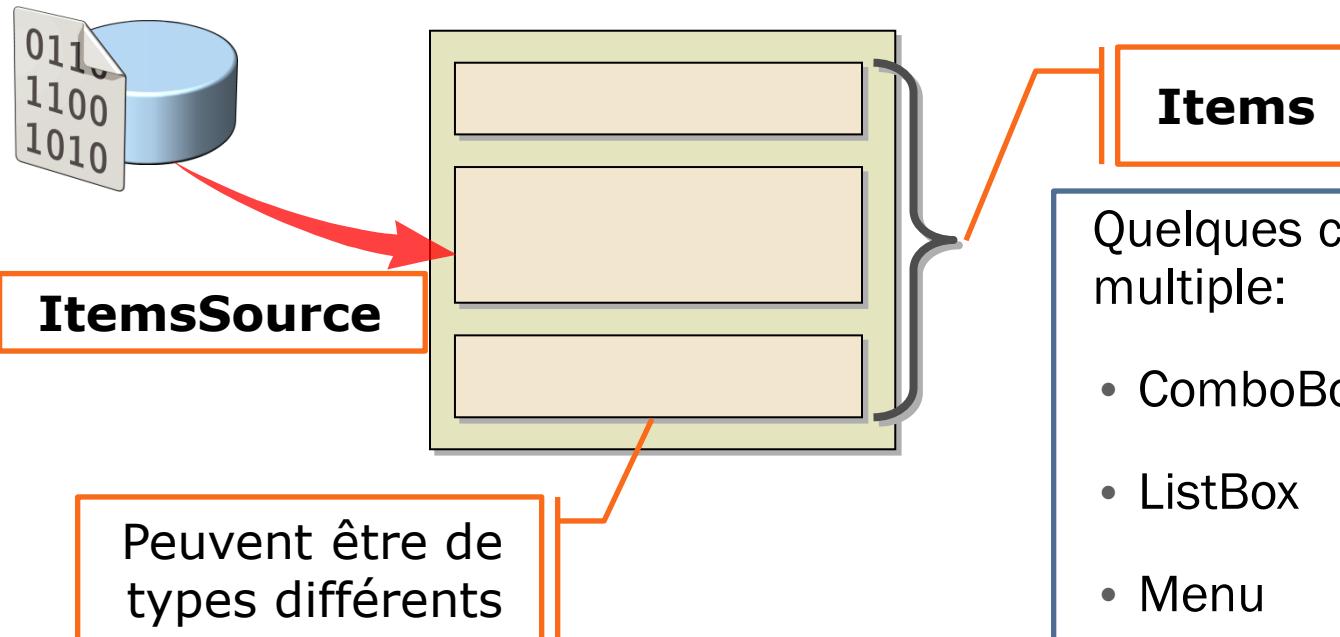
My Expander

Lorem ipsum dolor sit amet,  
consectetur adipisicing elit, sed  
do eiusmod tempor incididunt ut  
labore et dolore magna aliqua

Expander

GroupBox

# LES CONTRÔLES À ÉLÉMENTS MULTIPLE



- Contient de multiple objets
- Contient une propriété **Items**
- Contient une propriété **ItemsSource**

Quelques contrôles à élément multiple:

- ComboBox
- ListBox
- Menu
- StatusBar
- TabControl
- ToolBar
- TreeView

## LA SELECTION D'ÉLÉMENTS

Attacher un évènement

```
<ListBox  
    SelectionChanged="ListBox1_SelectionChanged">  
    ...  
</ListBox>
```

Définition de l'évènement

```
public void ListBox1_SelectionChanged(  
    object sender,  
    SelectionChangedEventArgs e)  
{  
    ...  
}
```

QUESTIONS ? REMARQUES ?



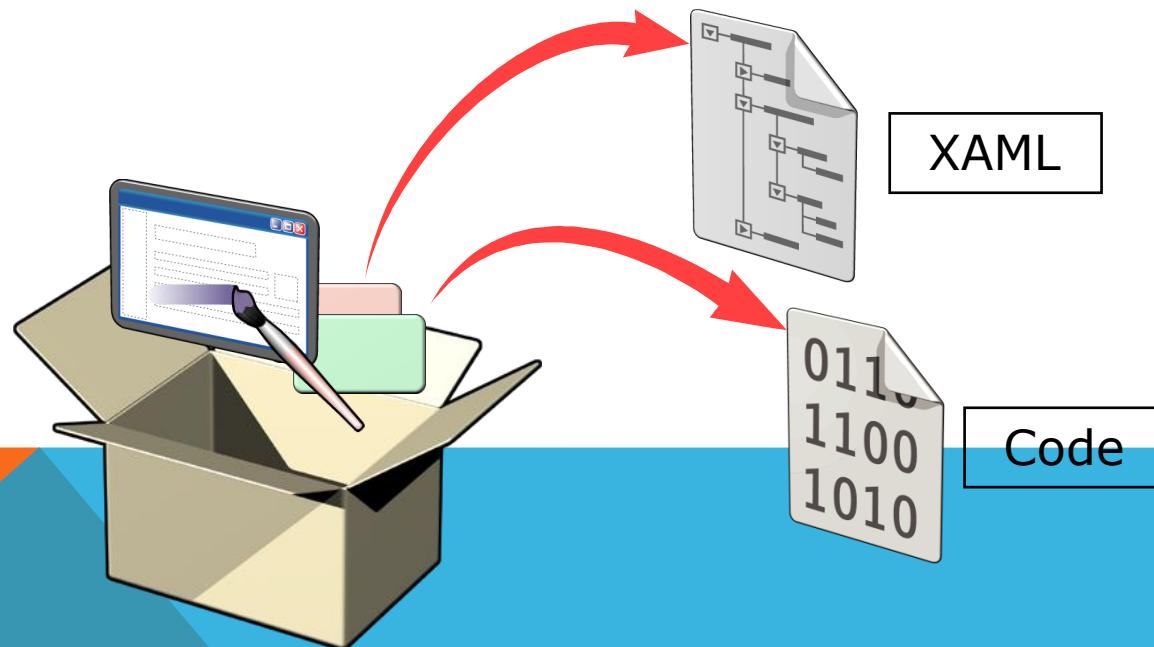
# LEÇON 4

LES RESSOURCES

## LES RESSOURCES

Les ressources permettent la réutilisation simple d'objets et de valeurs.

Par exemple: brushes, styles, et modèle de contrôle



## DÉFINIR UNE RESSOURCE

Les ressources peuvent être définies à différents niveaux :

- Au niveau de l'application
- Au niveau de la fenêtre ou de la page
- Au niveau de l'élément

### XAML

```
<Window.Resources>
    <SolidColorBrush x:Key="blueBrush" Color="Blue"/>
    <SolidColorBrush x:Key="whiteBrush" Color="White"/>
    <sys:Double x:Key="myValue">100</sys:Double>
</Window.Resources>
```

## RÉFÉRENCEMENT DES RESSOURCES EN XAML

Pour référencer une ressource statique

```
PropertyName="{StaticResource ResourceKey}"
```

```
<Button Background="{StaticResource blueBrush}"
        Foreground="{StaticResource whiteBrush}>
    Text
</Button>
```

Pour référencer une ressource dynamique

```
PropertyName="{DynamicResource ResourceKey}"
```

```
<Button Background="{DynamicResource blueBrush}"
        Foreground="{DynamicResource whiteBrush}>
    Text
</Button>
```

# RÉFÉRENCEMENT DES RESSOURCES PAR LE CODE

Methode **FindResource**

```
SolidColorBrush brush = (SolidColorBrush)  
    this.FindResource("whiteBrush");
```

Methode **SetResourceReference**

```
this.myButton.SetResourceReference(  
    Button.BackgroundProperty, "whiteBrush");
```

Ou **TryFindResource**

Propriété **Resources**

```
SolidColorBrush brush = (SolidColorBrush)  
    this.Resources["whiteBrush"];
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE 12

LES FRAMEWORKS WEB



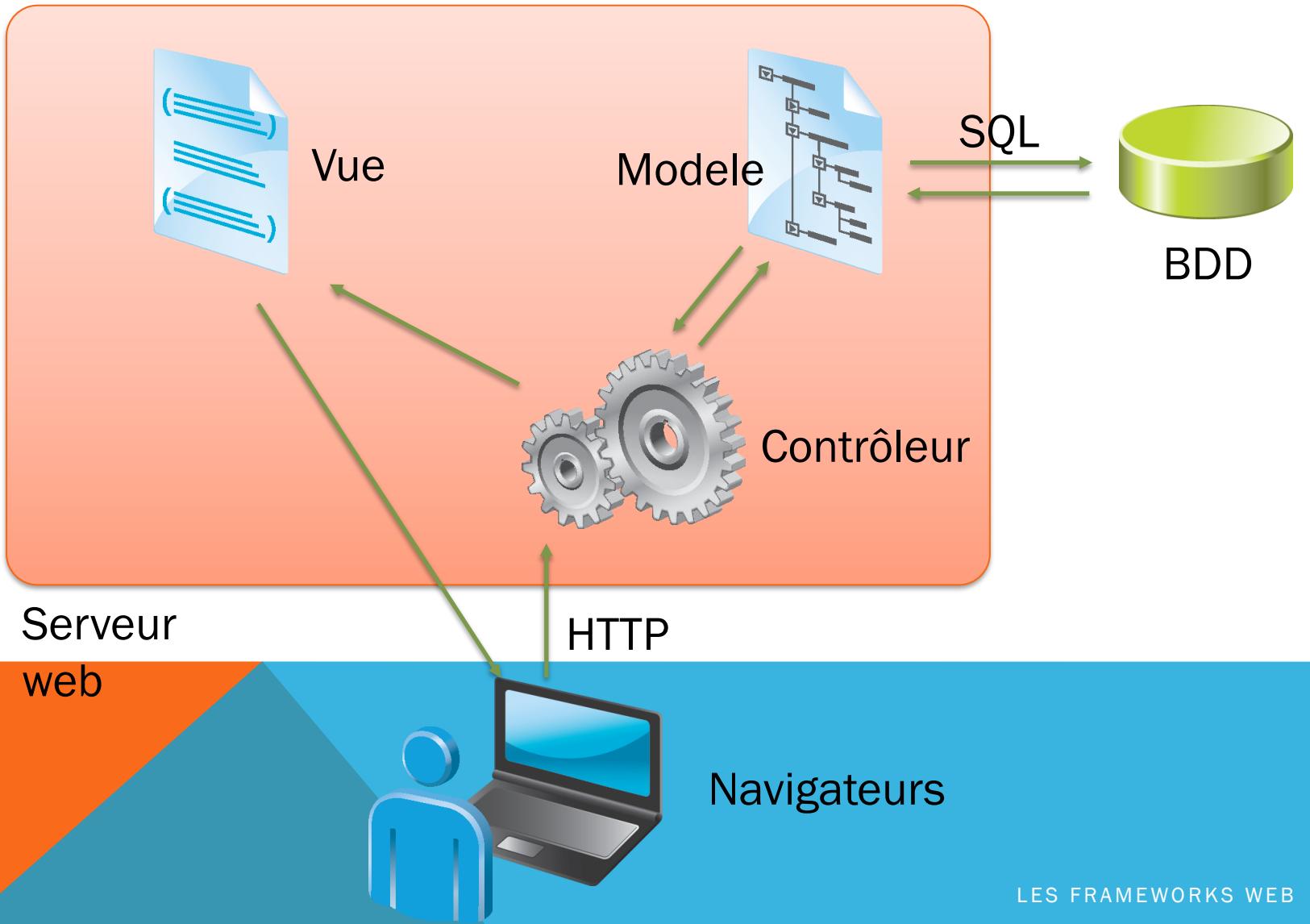
# LEÇON 1

PRÉSENTATION D'ASP.NET MVC

## ASP.NET MVC

- Seulement Visual Studio
- Le modèle englobe les objets et les données.
- La vue génère l'interface utilisateur.
- Les contrôleurs interagissent avec les actions utilisateur
- Le code peut être dans les fichiers .cshtml ou .cs
- Contrôle précis du HTML et des URLs
- Facilite les tests unitaires.

# MVC = MODELS, VIEWS, CONTROLLERS



QUESTIONS ? REMARQUES ?





# LEÇON 2

LES CONTRÔLEURS

## RÉPONDRE AUX REQUÊTES UTILISATEUR

Lorsque MVC reçoit une requête de l'utilisateur, les événements suivants sont levés :

Le **MvcHandler** crée une usine à contrôleurs.

L'usine à contrôleurs crée l'instance de contrôleur en fonction de la requête et le **MvcHandler** appelle la méthode '**execute**'.

Le **MvcHandler** examine la requête (**MvcHandler**) et détermine l'action à appeler.

Le **MvcHandler** détermine les valeurs à passer aux paramètres de l'action

Le **MvcHandler** exécute l'action

# ÉCRIRE DES ACTIONS DE CONTRÔLEUR

- Ecrire des actions peut impliquer :
  - La création d'action simples
  - Utilisation des verbes HTTP GET et POST dans l'action
- Exemple de contrôleur :

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

    public IActionResult Index()
    {
        return View();
    }

    public IActionResult Privacy()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

## PASSAGE D'INFORMATIONS À LA VUE

- Pour passer une information à une vue fortement typée, il est possible d'utiliser une surcharge à l'appel de la méthode **View()**
- Pour passer une information à une vue qui n'est pas fortement typée, il est possible d'utiliser:
  - La propriété **ViewBag**: objet dynamique contenant n'importe quel objet.
  - La propriété  **ViewData**: permet de passer n'importe quel objet sous forme de dictionnaire.

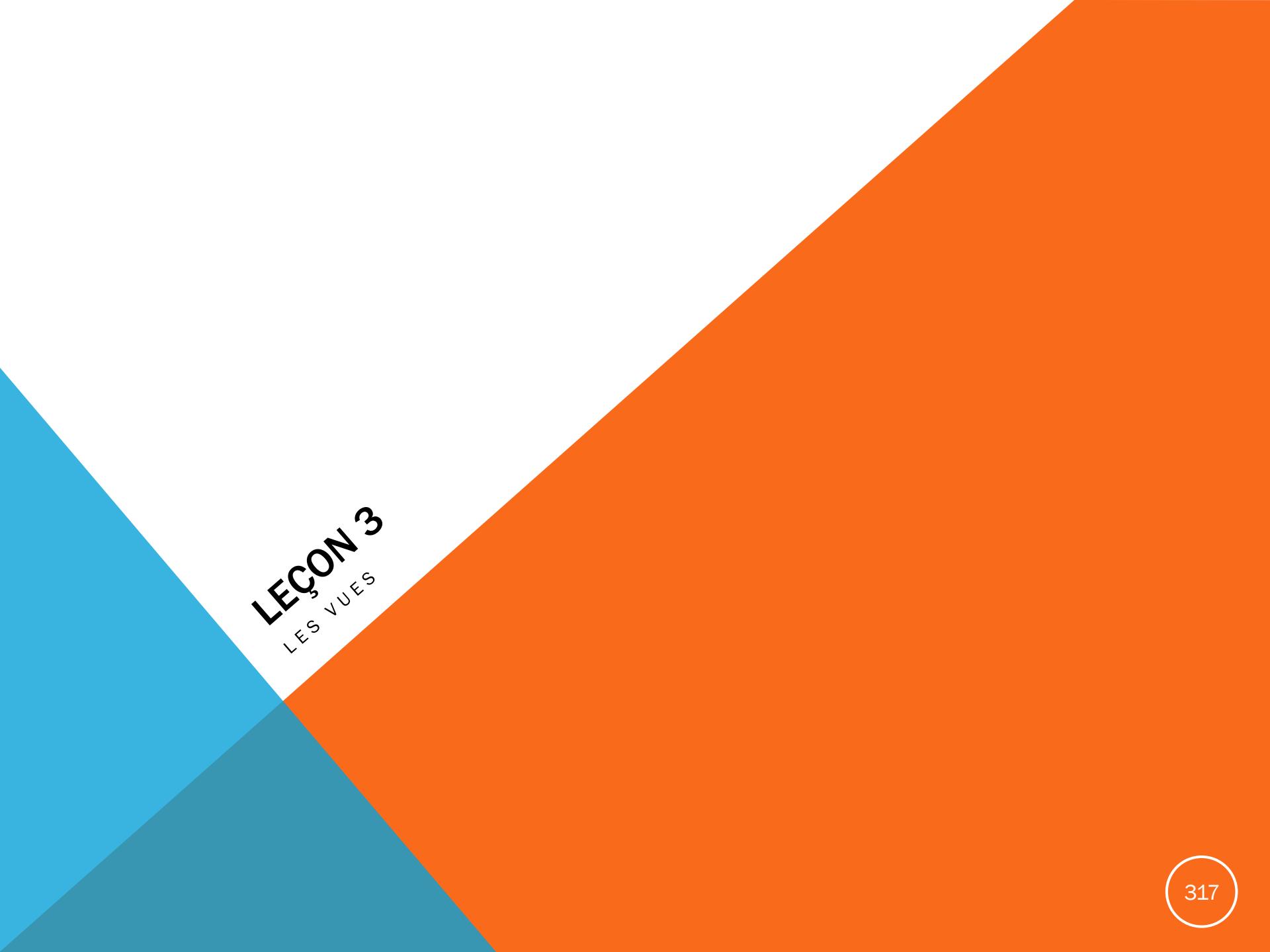
## UTILISATION DE L'OBJET VIEWBAG

- Ajout d'informations :
  - **ViewBag.Message** = "This text is not in the model object";
  - **ViewBag.ServerTime** = **DateTime.Now**;
- Récupération de l'information :

```
<p>
    The message of the day is: @ViewBag.Message
</p>
<p>
    The time on the server is: @ViewBag.ServerTime.ToString()
</p>
```

QUESTIONS ? REMARQUES ?

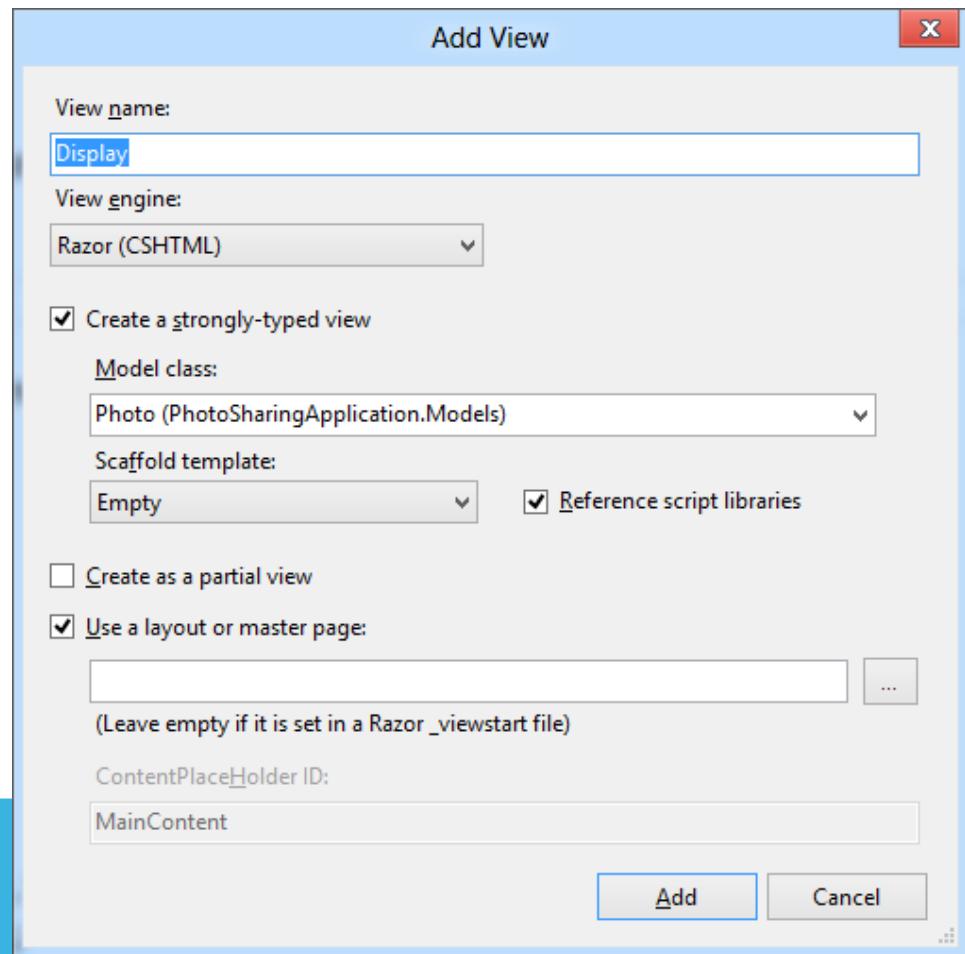
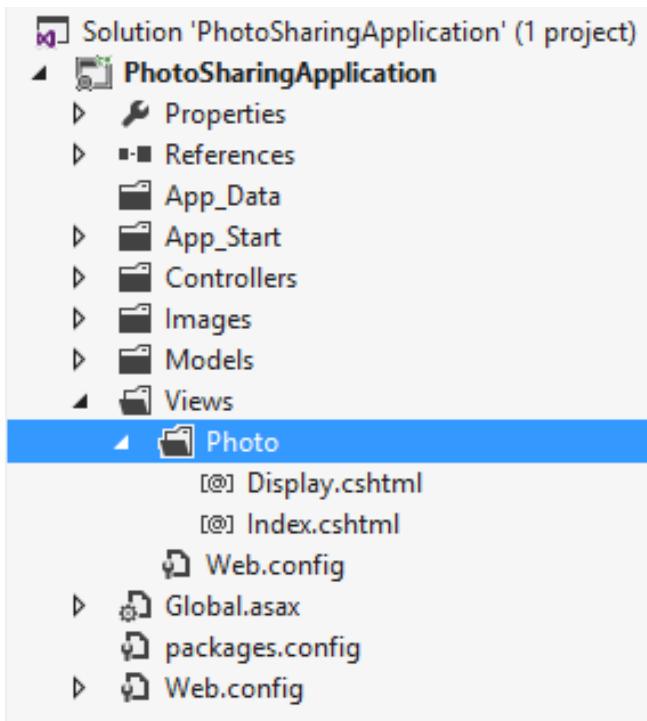




# LEÇON 3

LES VUES

# AJOUT D'UNE VUE



## DIFFÉRENTIER LE CODE SERVEUR DU CODE HTML

*Razor identifie le code serveur en se servant du symbole @.*

La syntaxe Razor utilise le symbole @ de différentes façons :

- Utiliser @ pour identifier le code serveur en c#
- Utiliser @@ pour rendre le symbole @
- Utiliser @: pour définir une ligne de texte est non du code sur une ligne
- Utilisation de <text> pour spécifier plusieurs lignes de texte
- Pour rendre du texte sans encodage HTML, il est possible d'utiliser la fonction **Html.Raw()**

# FONCTIONNALITÉS DE LA SYNTAXE RAZOR

Quelques exemples de syntaxes :

```
@* Ceci est un commentaire *@

<span>
    Price including Sale Tax: @Model.First().Price * 1.2
</span>
<span>
    Price including Sale Tax: @(Model.First().Price * 1.2)
</span>

@if (Model.Count() > 5)
{
    <ol>
        @foreach (var item in Model)
        {
            <li>@item.Name</li>
        }
    </ol>
}
```

## LIAISON DE LA VUE AUX CLASSES MODÈLES ET AFFICHAGE DES PROPRIÉTÉS

- Il est possible de créer des vues fortement typées en incluant la déclaration de la classe du modèle.
- Exemple de liaison avec une liste **IEnumerable** :

```
@model IEnumerable<ProjEcole.DataModel.BO.Personne>



# Liste de personne


@foreach (var personne in Model)
{
    <div>Name: @personne.Name</div>
}
```

## UTILISATION DES HELPERS D'ACTION

### Html.ActionLink()

```
@Html.ActionLink("Cliquez ici pour voir la photo 1",  
"Display", new { id = 1 })
```



```
<a href="/photo/Display/1">  
    Cliquez ici pour voir la photo 1  
</a>
```

### Url.Action()

```

```



```

```

## UTILISATION DES HELPERS D'AFFICHAGE

### Html.DisplayNameFor()

```
@Html.DisplayNameFor(model => model.CreatedDate)
```



Created Date

### Html.DisplayFor()

```
@Html.DisplayFor(model => model.CreatedDate)
```



03/12/2012

## LE HELPER “BEGINFORM”

### Html.BeginForm()

```
@using (Html.BeginForm("Create", "Photo",
    FormMethod.Post,
    new { enctype = "multipart/form-data" }))
{
    /* Placer les éléments de saisie ici */
}
```



```
<form action="/Photo/Create" method="post"
      enctype="multipart/form-data">
</form>
```

## UTILISATION DES HELPERS D'EDITION

### Html.LabelFor()

```
@Html.LabelFor(model => model.ContactMe)
```



```
<label for="ContactMe">  
    Contact Me  
</label>
```

### Html.EditorFor()

```
@Html.EditorFor(model => model.ContactMe)
```



```
<input type="checkbox"  
      name="ContactMe">
```

## LES HELPERS DE VALIDATION

### Html.ValidationSummary()

```
@Html.ValidationSummary()
```



```
<ul>
    <li>Veuillez saisir votre nom</li>
    <li>Veuillez spécifier une adresse valide</li>
</ul>
```

### Html.ValidationMessageFor ()

```
@Html.ValidationMessageFor(model => model.Email)
```



```
Veuillez spécifier une adresse valide
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION



# MODULE 13

.NET CORE



# LEÇON 1

INTRODUCTION

# INTRODUCTION

*Framework multiplateforme à hautes performances et open source*

- Applications et Services web, applications IoT et backends mobiles.
- Windows, MacOs et Linux.
- Environnement cloud ou local
- Exécution sur .NET Core et .Net Standard

## AVANTAGES

- Un scénario unifié pour créer une interface utilisateur web et des API web.
- Conçu pour la testabilité.
- Razor Pages permet de coder des scénarios orientés page de façon plus simple et plus productive.
- Capacité à développer et à exécuter sur Windows, MacOs et Linux.
- Open source et centré sur la communauté.
- L'intégration de Framework modernes, côté client et des workflows de développement

## AVANTAGES

- Un environnement prêt pour le cloud et basé sur des fichiers de configuration système.
- Une Injection de dépendances intégrée.
- Un pipeline de requête HTTP léger, haute performance et modulaire.
- La capacité à héberger sur IIS, Nginx, Apache, Docker, ou d'un auto-hébergement dans votre propre processus.
- La gestion de version des applications côte à côte lorsque la cible est .NET Core.
- SignalR (WebSocket) / WASM (Blazor)

## NOUVEAUTÉS DE ASP.NET CORE

### *SignalR*

- SignalR a été réécrit pour ASP.NET Core 2.1. ASP.NET Core
- SignalR inclut un certain nombre d'améliorations :
  - Un modèle simplifié de montée en puissance parallèle.
  - Un nouveau client JavaScript sans dépendance de jQuery.
  - Un nouveau protocole binaire compact basé sur MessagePack.
  - Prise en charge des protocoles personnalisés.
  - Un nouveau modèle réponse de streaming.
  - Prise en charge des clients basés sur des WebSocket nus.

## NOUVEAUTÉS DE ASP.NET CORE

### *Bibliothèques de classes Razor*

- Le temps de démarrage de l'application est nettement plus rapide.
- Les mises à jour rapides des pages et vues Razor au moment de l'exécution sont toujours disponibles dans le cadre d'un flux de travail de développement itératif.

### *HTTPS*

- Https est actif par défaut

### *RGPD*

- ASP.NET Core fournit des API et des modèles qui aident à satisfaire à certaines des exigences du RGPD

## NOUVEAUTÉS DE ASP.NET CORE

### *Tests d'intégration*

- Un nouveau package est introduit qui simplifie la création et l'exécution de tests. Le package  
***Microsoft.AspNetCore.Mvc.Testing***

### ***[ApiController], ActionResult<T>***

- NET Core ajoute de nouvelles conventions de programmation qui facilitent la génération d'API web propres et descriptives

## NOUVEAUTÉS DE ASP.NET CORE

### *IHttpClientFactory*

- ASP.NET Core 2.1 inclut un nouveau service `IHttpClientFactory` qui facilite la configuration et l'utilisation d'instances de `HttpClient` dans les applications

### *Configuration du transport Kestrel*

- Dans ASP.NET Core 2.1, le transport par défaut de Kestrel n'est plus basé sur Libuv, mais sur des sockets managés

## NOUVEAUTÉS DE ASP.NET CORE

### *Générateur d'hôte générique*

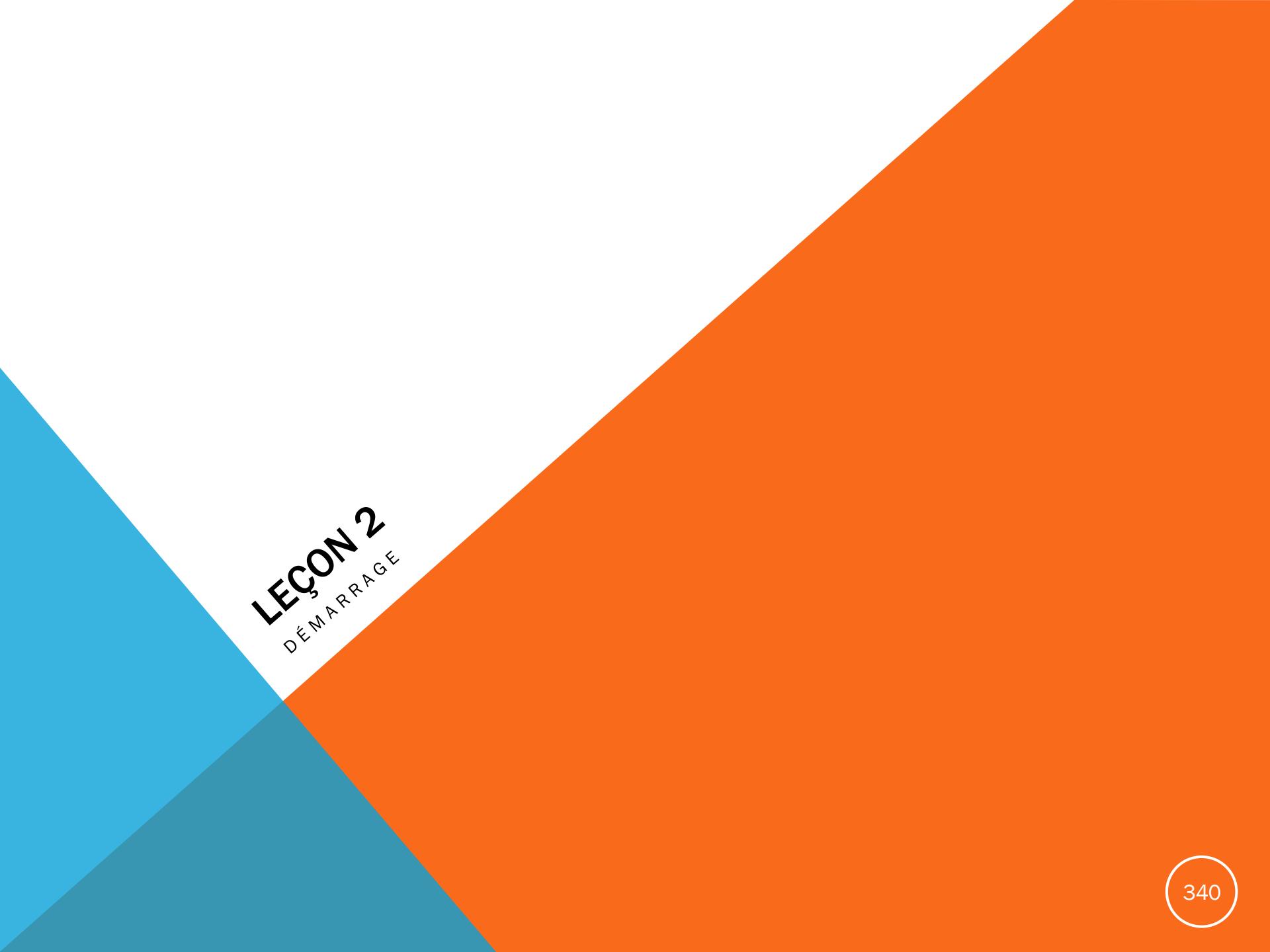
- Le générateur d'hôte générique (HostBuilder) a été introduit. Ce générateur peut être utilisé pour les applications qui ne traitent pas les requêtes HTTP (messagerie, les tâches en arrière-plan, etc.).

### *Modèles SPA mis à jour*

- Les modèles d'applications monopages pour Angular, React et React avec Redux sont mis à jour pour utiliser les systèmes de génération et les structures de projet standard pour chaque Framework.

QUESTIONS ? REMARQUES ?





# LEÇON 2

DÉMARRAGE

## CLASSE STARTUP

Utilisation de la classe **Startup** pour la configuration et l'exécution de l'application

- **ConfigureServices** permettant de configurer les services de l'application.
- **Configure** pour créer le pipeline de traitement de demande de l'application.

```
public class Startup
{
    // References
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    // Reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    // References
    public void ConfigureServices(IServiceCollection services)...
```

```
    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    // References
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)...
```

## ENREGISTREMENT DE LA CLASSE STARTUP

La classe Startup doit être enregistré dès le démarrage de l'application

```
public class Program
{
    0 references
    public static void Main(string[] args)...

    1 reference
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

# UTILISATION COURANTE DE L'INJECTION DE DÉPENDANCE

Une utilisation courante de l'injection de dépendances :

- *IWebHostEnvironment* pour les informations lié à l'environnement d'exécution
- *IConfiguration* pour lire la configuration de l'application
- *ILoggerFactory* pour créer un enregistreur d'événements

```
public class UserService
{
    private readonly ILoggerFactory loggerFactory;
    private readonly IConfiguration configuration;
    public UserService(ILoggerFactory factory, IConfiguration config)
    {
        loggerFactory = factory;
        configuration = config;
    }

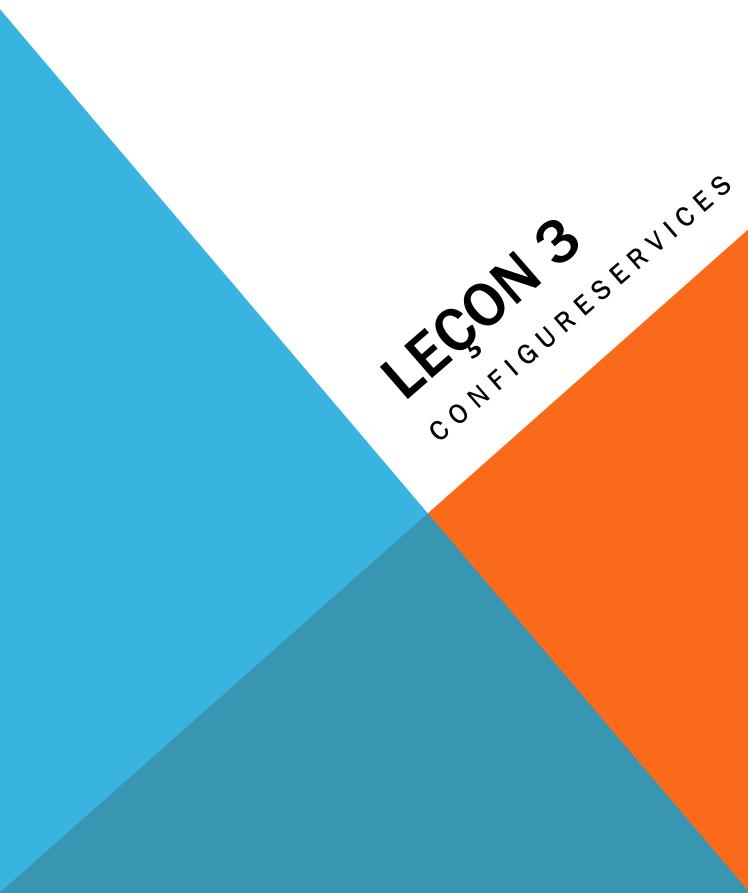
    public User GetUser()
    {
        var logger = loggerFactory.CreateLogger<UserService>();

        logger.LogInformation("Récupération de l'age dans la configuration");
        var age = configuration.GetValue<int>("Age");
        logger.LogInformation("Création d'un utilisateur");
        return new User() { Age = age, Firstname = "Jean", Lastname = "Martin" };
    }
}
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }
}
```

QUESTIONS ? REMARQUES ?





# LECON 3

CONFIGURE SERVICES

# MÉTHODE CONFIGURESERVICES

La méthode **ConfigureServices** est :

- Appelée par l'hôte web avant la méthode Configure pour configurer les services de l'application.
- L'emplacement où les options de configuration sont définies.

```
public void ConfigureServices(IServiceCollection services)
{
    //Add framework services
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDatabaseDeveloperPageExceptionFilter();

    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationContext>();
    services.AddControllersWithViews();

    //Add application services
    services.AddTransient<EmailService>();
    services.AddTransient<UserService>();
}
```

Par default, on appelle toutes les méthodes d'ajout de services **Add{service}** puis leurs configurations **Configure{service}**

# MÉTHODE CONFIGURE

La méthode **Configure** est utilisée pour spécifier la façon dont l'application répond aux requêtes HTTP

Composants intergiciels  
(Middleware)

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseMigrationsEndPoint();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
        endpoints.MapRazorPages();
    });
}
```

## MÉTHODES PRATIQUES

Des méthodes pratiques comme `ConfigureServices()` ou `ConfigureLogging()` peuvent être utilisées au lieu de spécifier une classe `Startup`

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    })
    .ConfigureServices(services => {
        services.AddRazorPages();
        services.AddAuthentication();

        services.AddScoped<UserService>();
    });
}
```

QUESTIONS ? REMARQUES ?



# MODULE 14

INJECTION DE DÉPENDANCES





# LEÇON 1

INTRODUCTION

# INTRODUCTION

ASP.NET Core prend en charge le modèle de conception de logiciel avec injection de dépendances (DI).

Une technique permettant d'obtenir une inversion de contrôle (IoC) entre les classes et leurs dépendances.

- Une *dépendance* est un objet qui nécessite un autre objet

```
public class Calculator
{
    public Produit Produit { get; set; }

    O references
    public decimal Calculate()
    {
        return Produit.Prix * Produit.Qte;
    }
}
```

La dépendance est injectée par le constructeur

```
public class Calculator
{
    public IProduit Produit { get; set; }

    O references
    public decimal Calculate()
    {
        return Produit.Prix * Produit.Qte;
    }
}

O references
public Calculator(IProduit prod)
{
    Produit = prod;
}
```

## IOC (INVERSION OF CONTROL)

ASP.NET Core fournit un conteneur de service intégré, `IServiceProvider`. Les services sont inscrits dans la méthode `Startup.ConfigureServices` de l'application.

```
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)...
    2 references
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddTransient<IProduit, Produit>();
```

La dépendance, peut avoir elle même des dépendances.

ASP.NET core va se charger d'injecter les dépendances dans le constructeur

```
public class Produit : IProduit
{
    private readonly ILogger<Produit> logger;

    0 references
    public Produit(ILogger<Produit> log)
    {
        logger = log;
    }

    0 references
    public string Libelle { get; set; }
    2 references
    public int Prix { get; set; }
    2 references
    public int Qte { get; set; }
}
```

# SERVICES FOURNIS PAR LE FRAMEWORK

## Type de service

Microsoft.AspNetCore.Hosting.Builder.IApplicationBuilderFactory

Microsoft.AspNetCore.Hosting.IApplicationLifetime

Microsoft.AspNetCore.Hosting.IHostingEnvironment

Microsoft.AspNetCore.Hosting.IStartup

Microsoft.AspNetCore.Hosting.IStartupFilter

Microsoft.AspNetCore.Hosting.Server.IServer

Microsoft.AspNetCore.Http.IHttpContextFactory

Microsoft.Extensions.Logging.ILogger<T>

Microsoft.Extensions.Logging.ILoggerFactory

Microsoft.Extensions.ObjectPool.ObjectPoolProvider

Microsoft.Extensions.Options.IConfigureOptions<T>

Microsoft.Extensions.Options.IOptions<T>

System.Diagnostics.DiagnosticSource

System.Diagnostics.DiagnosticListener

## Durée de vie

*Transient*

*Singleton*

*Singleton*

*Singleton*

*Transient*

*Singleton*

*Transient*

*Singleton*

*Singleton*

*Transient*

*Singleton*

*Singleton*

*Singleton*

*Singleton*

QUESTIONS ? REMARQUES ?





# LEÇON 2

DURÉES DE VIE DU SERVICE

## LES DIFFÉRENTS TYPES DE SERVICE

### *Transient*

- Des services à durée de vie temporaire (Transient) sont créés chaque fois qu'ils sont demandés. Cette durée de vie convient parfaitement aux services légers et sans état.

### *Scoped*

- Les services à durée de vie délimitée (Scoped) sont créés une seule fois par requête.

### *Singleton*

- Les services avec une durée de vie singleton sont créés la première fois qu'ils sont demandés. Chaque requête ultérieure utilise la même instance.

# INSCRIPTIONS

L'inscription des services se fait dans la classe **ConfigureServices**

```
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)...

    2 references
    public IConfiguration Configuration { get; }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddTransient<IProduit, Produit>();
        services.AddScoped<IProduit, Produit>();
        services.AddSingleton<IProduit, Produit>();
    }
}
```

ou

ou

## APPELER DES SERVICES À PARTIR DE MAIN

Créez un *IServiceScope* avec *IServiceScopeFactory.CreateScope* pour résoudre un service Scoped dans le scope de l'application. Cette approche est pratique pour accéder à un service Scoped au démarrage pour exécuter des tâches d'initialisation

```
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();

    using (var serviceScope = host.Services.CreateScope())
    {
        var services = serviceScope.ServiceProvider;

        try
        {
            var serviceContext = services.GetRequiredService<IProduit>();
            //Use the context here
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred");
            throw;
        }
    }

    host.Run();
}
```

## CONCEPTION DE SERVICES

Les bonnes pratiques :

- Concevoir des services afin d'utiliser l'injection de dépendances pour obtenir leurs dépendances.
- Éviter les appels de méthode statiques avec état (une pratique appelée static cling).
- Éviter une instanciation directe de classes dépendantes au sein de services. L'instanciation directe associe le code à une implémentation particulière.

# CONCEPTION DE SERVICES

Les bonnes pratiques :

- Respecter les principes SOLID
  - S – Single Responsibility Principle
  - O – Open/Closed Principle
  - L – Liskov Substitution Principle
  - I – Interface Segregation Principle
  - D – Dependency Inversion Principle

# SUPPRESSION DES SERVICES

Le conteneur d'injection de dépendance ne supprime pas les instances ajouté manuellement par le code utilisateur (développeur).

```
public class Service1 ...
1 reference
public class Service2 ...
1 reference
public class Service3 ...
2 references
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)...
    1 reference
    public IConfiguration Configuration { get; }

    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        services.AddAuthentication();

        //The container creates the following instances and disposes them automatically.
        services.AddScoped<Service1>();
        services.AddSingleton<Service2>();

        //The container doesn't create the following instances, so it doesn't dispose the instance
        services.AddSingleton(new Service3());
    }
}
```

QUESTIONS ? REMARQUES ?



# DÉMONSTRATION

