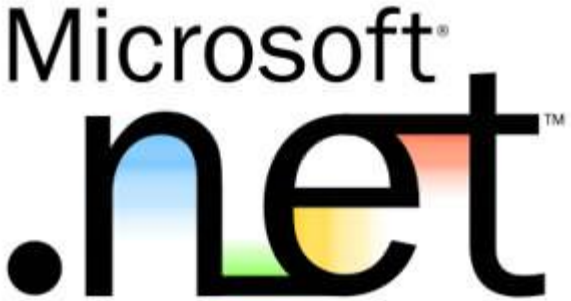



# C# 2.0

## Introduction à C# et au framework .Net

ISIMA ZZ2F2

Luc TOURAILLE

Source : "*C# 3.0 in a Nutshell*", Joseph & Ben ALBAHARI

# Introduction

- Langage développé spécifiquement pour le framework .Net
- A l'origine, presque une copie conforme de Java
- Depuis, apparition de spécificités propres à chacun de ces langages (propriétés, indexers, delegates, events, unsafe...)

# Plan

I – Les bases du langage

II – Création de types

III – Fonctionnalités avancées

IV – Framework .Net

# Plan

I – Les bases du langage

II – Création de types

III – Fonctionnalités avancées

IV – Framework .Net

# Premier programme

```
using System;

class PremierProgramme
{
    static void Main()
    {
        string s = "Hello " + "World!";
        Console.WriteLine(s);
    }
}
```

# Compilation

- Ensemble de fichiers .cs compilés dans un assembly.
- Assembly =
  - Application (.exe), avec un point d'entrée Main
  - Bibliothèque (.dll), destinée à être utilisée dans une application ou dans d'autres bibliothèques
    - Exemple : BCL (Base Class Library)
- Compilation avec csc.exe, ou un IDE (Visual Studio .Net)

# Syntaxe

- Identifiants = noms choisis par le(s) développeur(s)
  - Case-sensitive, commence par une lettre ou un underscore, etc.
  - Exemples : `System PremierProgramme Main s Console WriteLine`
- Mots-clés = noms réservés par le langage
  - On peut éviter les conflits avec @<keyword>
  - Exemples : `using class static void string`

# Premier programme

```
using System;

class PremierProgramme
{
    static void Main()
    {
        string s = "Hello " + "World!";
        Console.WriteLine(s);
    }
}
```

# Syntaxe

- Littéraux = valeurs "en dur"
  - Exemples : `"Hello "  "World!"`
- Ponctuation = structure le programme
  - Exemples : `{ } ;`
- Opérateur = transforme et combine des expressions
  - Exemples : `. ( ) + =`

# Syntaxe

- Commentaires
  - `int i = 2; // Sur une ligne`
  - `int i = 2; /* Si on a beaucoup à dire, sur plusieurs lignes */`

# Les types : définitions

- Un type définit le patron, le modèle d'une valeur.
- Une valeur est une zone de stockage dénotée par une *variable* ou une *constante*.
- Toute valeur est une instance d'un type donné.

# Les types : définitions

- Un type définit le patron, le modèle d'une valeur.
- Une valeur est une zone de stockage dénotée par une *variable* ou une *constante*.
- Toute valeur est une instance d'un type donné.

# Les types : définitions

- Un type définit le patron, le modèle d'une valeur.
- Une valeur est une zone de stockage dénotée par une *variable* ou une *constante*.
- Toute valeur est une instance d'un type donné.

# Les types prédéfinis

- Types supportés de base par le compilateur
- Types numériques
  - `sbyte short int long byte ushort uint ulong float double decimal`
- Caractères
  - `char string`
- Booléen
  - `bool`
- Classe mère de tous les autres types
  - `object`

# Conversions

- Conversion implicite :
  - ```
    int  x = 123456;
    long y = x;
    ```
  - Possible si le compilateur peut garantir la conversion, et si aucune information n'est perdue
- Sinon, conversion explicite :
  - ```
    short z = (short) x;
    ```

# Opérations sur les entiers

- Division entière :
  - `int a = 2 / 3; // a = 0`
  - `int b = 0;`
    `int c = 5 / b; // DivisionByZeroException`
- Overflow :
  - `int a = int.MinValue;`
    `a--; // a = int.MaxValue --> wraparound`
- Possibilité de vérification, avec l'opérateur `checked`

# Float et Double

- Valeurs spéciales :
  - Not a Number (Nan) --> 0.0/0.0, ∞-∞
  - +∞ (PositiveInfinity) --> 1.0/0.0, -1.0/-0.0
  - -∞ (NegativeInfinity) --> -1.0/0.0, 1.0/-0.0
  - MaxValue, MinValue, Epsilon

- Erreurs d'arrondi dues à la représentation en base 2
  - `float f = 0.1f * 10f - 1f; //f=1.490116E-08`

# Strings

- Chaînes de caractères immuable (non modifiable)
- Concaténation :
  - Opérateur +
    - `string s = "a" + "b"; // ab`
    - `string s = "a" + 5;  // a5`
  - StringBuilder
- Comparaison : `CompareTo()`

# Les types personnalisés

- Définis à partis de types primitifs ou d'autres types personnalisés
- Exemple :

```
class Point
{
   int x, y;                                  // Champs
   public Point(int inx, int iny)    // Constructeur
      { x = inx; y = iny; }
   public void move(int depx, int depy) // Méthode
      { x += depX; y += depY; }
}
```

# Instanciation

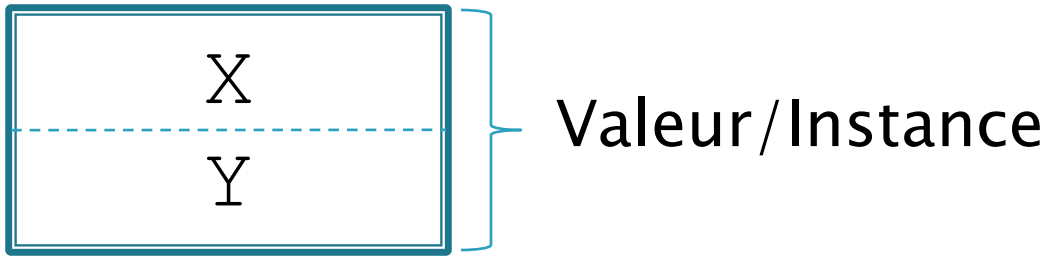
- Pour créer des données, on *instancie* un type.
- Pour les types primitifs, un littéral suffit
  - `int i = 15 * 23; // Trois instanciations`
- Pour les types personnalisés, il faut utiliser l'opérateur `new`
  - `Point p = new Point(15, 23);`
  - Instanciation, puis appel du constructeur

# Valeur vs Référence

- Tous les types C# appartiennent à une de ces catégories :
  - Types *valeur*
  - Types *référence*
  - Types *pointeur* (peu utilisés en C#)
- Types valeurs : tous les types prédéfinis (sauf `string` et `object`) + structures + énumérations
- Types références : classes + interfaces + tableaux + délégués

# Types valeur

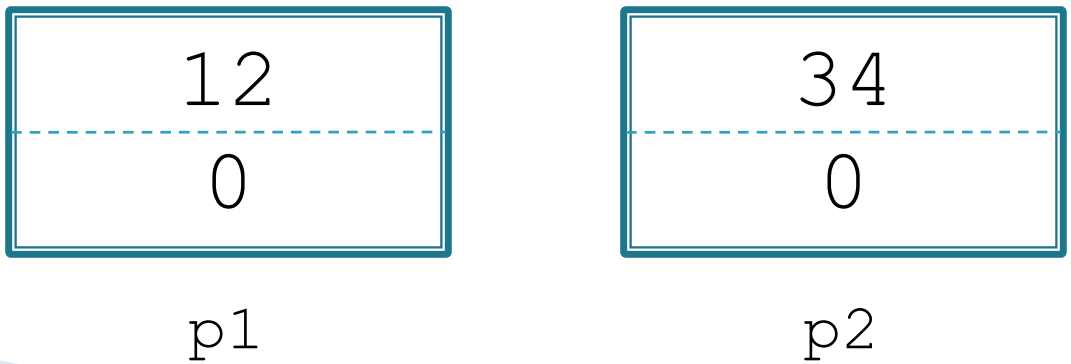
- Le contenu d'une variable ou d'une constante de type valeur est une simple…valeur !
- Exemple :
  - `struct Point { public int X, Y; }`



# Types valeur

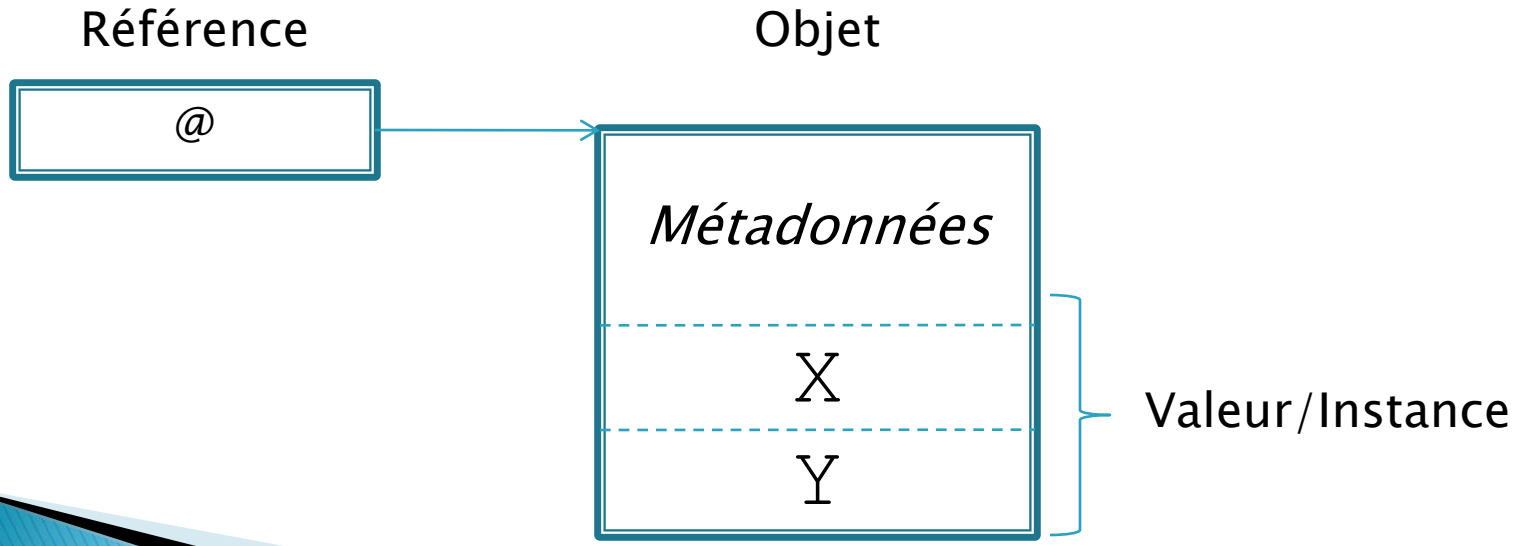
- Lors d'une affectation, la valeur est copiée.
- Exemple :

```
Point p1 = new Point();
p1.X     = 12;
Point p2 = p1; // p1.X = p2.X = 12
p2.X     = 34; // p1.X = 12, p2.X = 34
```



# Types référence

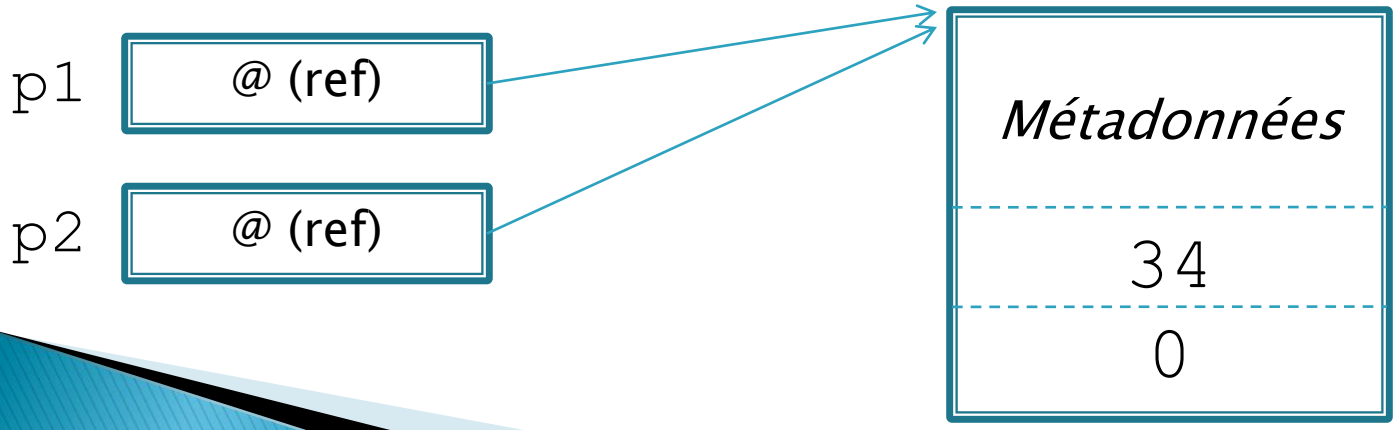
- Deux parties distinctes : l'*objet* et la *référence* vers l'objet.
- Exemple :
  - `class Point { public int X, Y; }`



# Types référence

- Lors d'une affectation, seule la référence est copiée, pas l'objet.
- Exemple :

```
Point p1 = new Point();
p1.X     = 12;
Point p2 = p1; // p1.X = p2.X = 12
p2.X     = 34; // p1.X = 34, p2.X = 34
```



# Null

- On peut affecter le littéral `null` à une référence, pour indiquer qu'elle ne pointe vers aucun objet.
- On ne peut pas affecter `null` à une variable de type valeur (il existe cependant des types nullables).

# Tableaux

- Eléments d'un même type stockés dans un bloc mémoire contigu
- Eléments initialisés à leur valeur par défaut (`0` pour les types valeur, `null` pour les types référence)

```
int[] tab = new int[5];
tab[3] = 3;
for (int i = 0 ; i < tab.Length ; ++i)
    Console.Write(tab[i]); // 00030
```

# Tableaux

- Initialisation :
  - `char[] voy = new char[] {'a','e','i','o','u','y'};`
  - `char[] voy = {'a','e','i','o','u','y'};`
- Vérification de bornes
  - `voy[6] = z; // IndexOutOfRangeException`

# Tableaux rectangulaires

- Matrice MxN(xOxPxQxRxSxT…)
  - `int[,] matrice = new int[2,3];`
    `matrice[0,1] = 23;`
- Dimensions accessibles par GetLength
  - `int dim1 = matrice.GetLength(1); // dim1 = 3`
- Initialisation :
  - 
```
int[,] matrice = new int[]
{
  {0,1,2},
  {3,4,5}
};
```

# Tableaux rectangulaires

- Matrice MxN(xOxPxQxRxSxT…)
  - `int[,] matrice = new int[2,3];`
    `matrice[0,1] = 23;`
- Dimensions accessibles par GetLength
  - `int dim1 = matrice.GetLength(0); // dim1 = 3`
- Initialisation :
  - 
```
int[,] matrice =
{
  {0,1,2},
  {3,4,5}
};
```

# Tableaux "en escalier"

- ⇔Tableaux de tableaux
  - ```
    int[][] tabtab = new int[2][];
    tabtab[0]      = new int[3];
    tabtab[1]      = new int[2];
    ```

- Initialisation :
  - ```
    int[][] tabtab = new int[][]
    {
      new int[] {0,1,2},
      new int[] {3,4}
    };
    ```

# Tableaux "en escalier"

- ⇔Tableaux de tableaux
  - ```
    int[][] tabtab = new int[2][];
    tabtab[0]      = new int[3];
    tabtab[1]      = new int[2];
    ```

- Initialisation :
  - ```
    int[][] tabtab =
    {
      new int[] {0,1,2},
      new int[] {3,4}
    };
    ```

# Variables

- Peuvent être de plusieurs sortes :
  - Variables locales
  - Paramètres (valeur, ref, out)
  - Champ (de classe ou d'instance)
  - Elément de tableau
- Forcément assignées
  - par l'utilisateur pour les variables locales et les paramètres
  - par le runtime pour les champs et les éléments de tableau, avec leur valeur par défaut
- Sont stockées soit dans la *pile*, soit dans le *tas*.

# Pile et Tas

- Pile (stack) = bloc de mémoire où sont empilés les variables locales, les paramètres et les adresses de retour de fonction.
  - Grossit et diminue au fil des appels et des retours de fonction
- Tas (heap) = bloc de mémoire où résident les *objets* (types références).
- Le ramasse-miettes (Garbage Collector) collecte la mémoire qui n'est plus référencée.

# Paramètres

- Par défaut, passage par *valeur* --> une copie est effectuée (copie de la valeur ou de la référence)
- Exemple :

```
void Foo(int i) { i++; }
…
int i = 1;
Foo(i);
Console.WriteLine(i); // 1
```

# Paramètres

- Par défaut, passage par *valeur* --> une copie est effectuée (copie de la valeur ou de la référence)
- Exemple :

```
void Foo(Point p) { p.X = 12; p = null}
…
Point p = new Point();
Foo(p);
Console.WriteLine(p.X); // 12
```

# Paramètres

- Passage par référence avec le modificateur `ref`
- Exemple :

```
void Swap(ref int a, ref int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
…
int i = 1, j = 9;
Swap(ref i, ref j);
Console.WriteLine(i + " " + j); // 9 1
```

# Paramètres

- Paramètre de sortie avec le modificateur `out`
- Exemple :

```
void DivEuclid(    int a        ,      int b,
               out int quotient, out int reste)
{
   quotient = a / b;
   reste    = a % b;
}
…
int quotient, reste;
DivEuclid(13, 4, out quotient, out reste);
Console.WriteLine(quotient + " " + reste); // 3 1
```

# Expressions

- Une expression dénote une valeur (ou `void`).
- Peuvent être transformées/combinées par des *opérateurs*.
- Expressions primaires = intrinsèques au langage
  - ```
    Math.Log(1) // Lookup de membre + appel
                // de méthode
    ```
- Assignation
  - `y = 5 * (x = 2)`
- Composable avec d'autres opérateurs : `*=`, `-=`, `<<=`, etc.
    ```

# Instructions

- Déclarations (avec éventuellement initialisation)
  - `int i, j = 12;`
  - `const double = 2.0;`
- Expressions (doit changer l'état du programme)
  - `5+2; // Erreur de compilation`
- Conditionnelles
  - Sélection : `if`, `switch`
  - Opérateur ternaire : `? :`
  - Boucles : `while`, `do…while`, `for`, `foreach`
    - `foreach (char c in "vodka") Console.WriteLine(c);`

# Instructions

- Sauts :
  - `break;` `// sort de la boucle courante`
  - `continue;` `// retourne au début de la boucle`
  - `label:`

```
// do stuff
goto label; // va au label "label"
```

  - `return res;` `// quitte la méthode en renvoyant une valeur`
  - `throw new UneException;` `// indique une erreur`

- `lock` (threading) et `using` (libération de ressources)

# Namespaces

- Domaine dans lequel chaque nom de type doit être unique
- Permet d'éviter les conflits de nommages et d'organiser les types selon leurs fonctionnalités

```
namespace Machin.Truc { class Classe1 {}}

namespace Bidule
{
  class Classe2 { Machin.Truc.Classe1 c; }
}
```

# Namespaces

- Domaine dans lequel chaque nom de type doit être unique
- Permet d'éviter les conflits de nommages et d'organiser les types selon leurs fonctionnalités

```
namespace Machin.Truc { class Classe1 {}}

namespace Bidule
{
  using Machin.Truc;        // importe le namespace
  class Classe2 { Classe1 c; }
}
```

# Namespaces

- Domaine dans lequel chaque nom de type doit être unique
- Permet d'éviter les conflits de nommages et d'organiser les types selon leurs fonctionnalités

```
namespace Machin.Truc { class Classe1 {}}

namespace Bidule
{
  using MT = Machin.Truc;      //alias le namespace
  class Classe2 { MT.Classe1 c; }
}
```

# Namespaces

- Domaine dans lequel chaque nom de type doit être unique
- Permet d'éviter les conflits de nommages et d'organiser les types selon leurs fonctionnalités

```
namespace Machin.Truc { class Classe1 {}}

namespace Bidule
{
  using Machin.Truc.Classe1;      //importe le type
  class Classe2 { Classe1 c; }
}
```

# Namespaces

- Domaine dans lequel chaque nom de type doit être unique
- Permet d'éviter les conflits de nommages et d'organiser les types selon leurs fonctionnalités

```
namespace Machin.Truc { class Classe1 {}}

namespace Bidule
{
  using MTC = Machin.Truc.Classe1; //alias le type
  class Classe2 { MTC c; }
}
```

# Namespaces

- Namespace `global` : contient tous les namespaces "racines", ainsi que les types n'appartenant à aucun namespace

- Portée des noms :
  - Types d'un namespace sont visibles dans les namespaces imbriqués
  - Pour se référer à un type dans une branche différente de la hiérarchie, on peut utiliser un nom qualifié partiel

- Masquage de noms : on prend toujours le type le plus proche dans la hiérarchie