Dawan
Training is our way

Langage C#

Christophe Fontaine
cfontaine@dawan.fr

05/10/2020

Objectifs

- Apprendre à développer avec C#
- Créer des interfaces de gestion de bases
- Manipuler les objets de la plate-forme .NET

Bibliographie

- **C# 8 et Visual Studio 2019**
  Les fondamentaux du langage
  Sébastien Putier
  Éditions ENI - janvier 2020

- **Documentation .Net**
  https://docs.microsoft.com/fr-fr/dotnet/

- **Documentation C#**
  https://docs.microsoft.com/fr-fr/dotnet/csharp/

- **Documentation Visual Studio**
  https://docs.microsoft.com/fr-fr/visualstudio/ide/?view=vs-2019

Plan

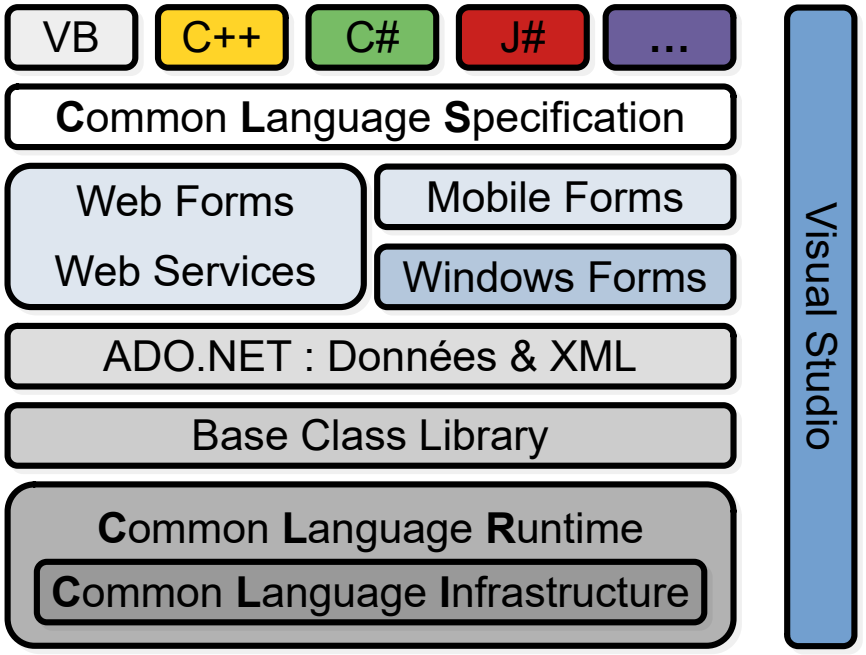
- Présentation
- Syntaxe de base
- Tableaux
- Méthodes et paramètres
- Exceptions
- Bibliothèque de classes .Net
- Interfaces Graphiques : WinForms
- Programmation Orienté Objet
- Accès à une base de données : ADO.NET
- Threads

Dawan
Training is our way

Présentation

Plateforme .Net

- **Utilisation** : Développement – Déploiement – Exécution
- **Applications** : Web, Windows, Mobile, serveurs, jeux
- **Langages supportés** : VB .NET, J#, C#, etc.
- **Gratuite**
- **Installation** : Intégrée à certaines éditions Windows
  Téléchargeable via Windows Update

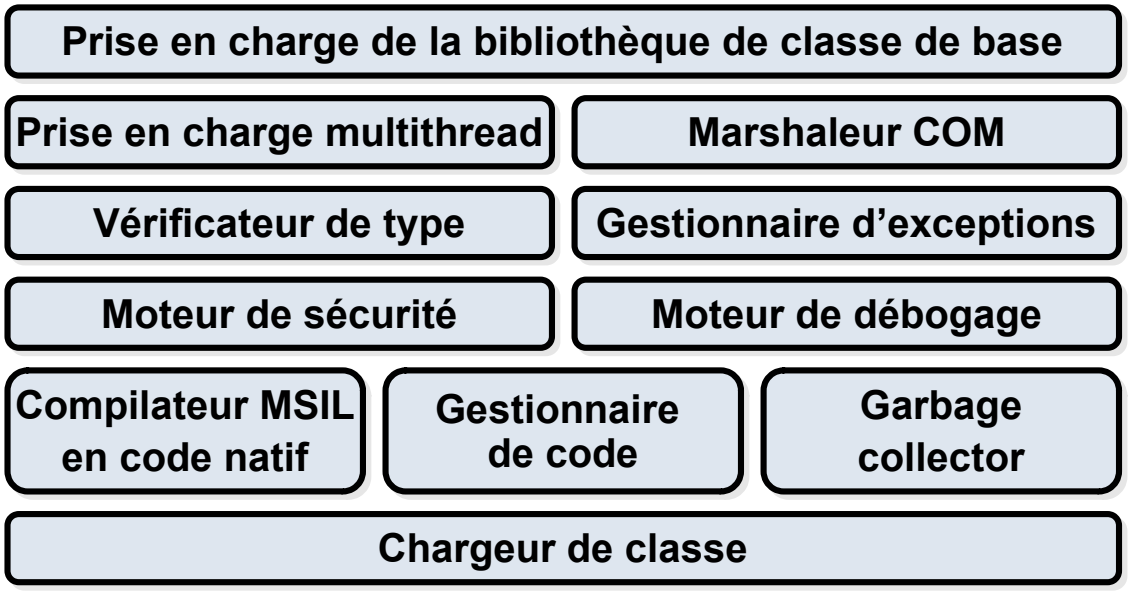


Historique

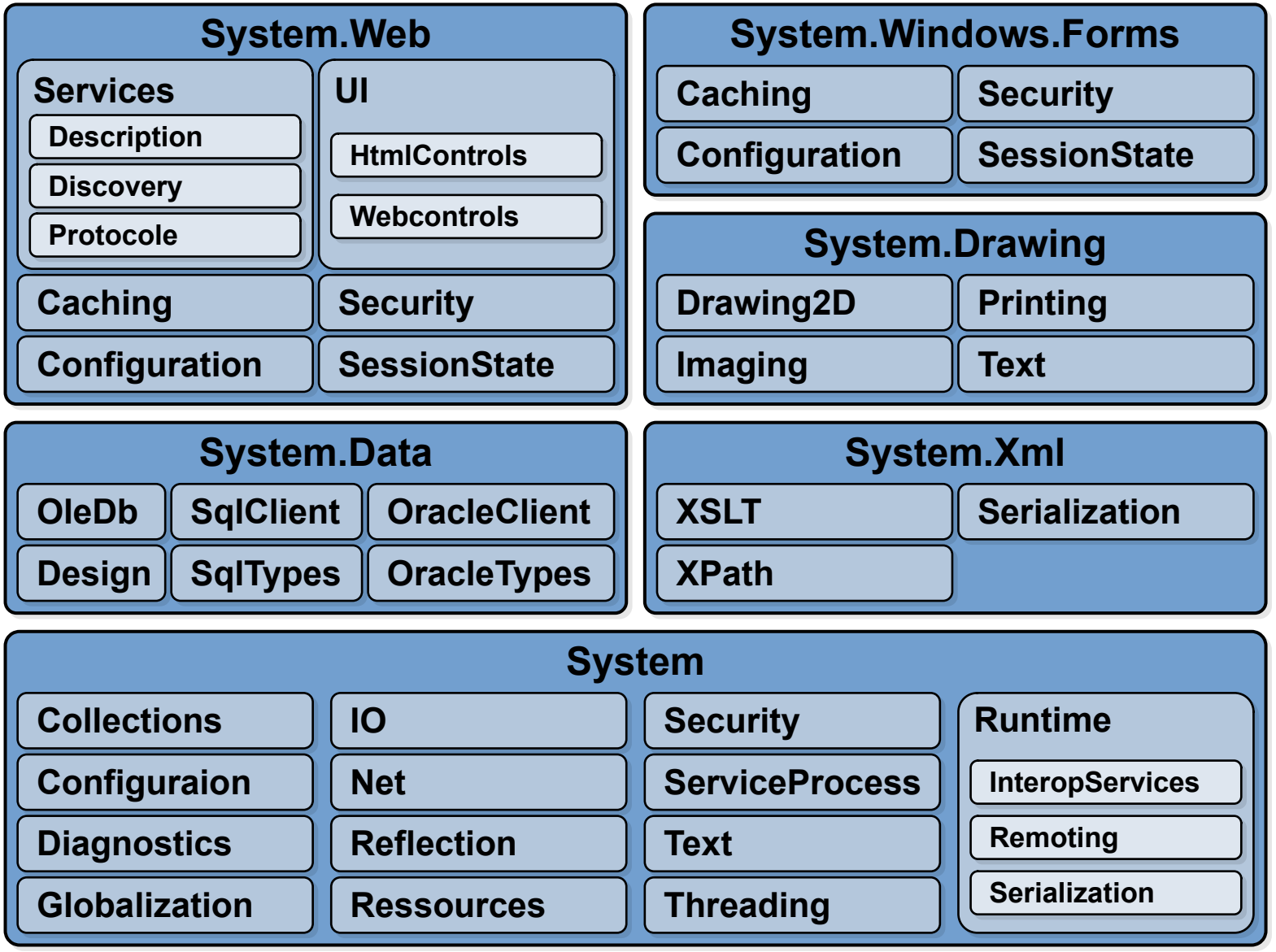
2000	Lancement du développement	**concepteur → Anders Hejlsberg**
2002	.NET Framework 1	**Winform, ASP.Net**
2003	.NET Framework 1.1	**Intégration ASP.Net, IPV6**
2005	.NET Framework 2.0	**64 bits, Type générique**
2006	.NET Framework 3.0	**WPF, WCF, WF, CardSpace**
2008	.NET Framework 3.5	**Linq, Entity Framework**
2010	.NET Framework 4.0	**Parallel Linq , DRL**
2012	.NET Framework 4.5	
2015	.NET Framework 4.6	**RyuJIT**
2017	.NET Framework 4.7	
2019	.NET Framework 4.8	
2020	.NET 5.0	**Fusion de .NET Framework et .NET Core**

Common Language Runtime

- Implémentation du standard Common Language Infrastructure
- Concepts : Debug, Typage (Common Type System), Exceptions ...
- Fonctions : Gestion du contexte d'exécution et de la mémoire
  Versioning des applications
- Sécurité et intégrité des applications (signatures)
- Interopérabilité COM



Bibliothèque de classes



Présentation C#

- Langage orienté objet de type sécurisé
- Très proche du C++ et du Java
- RAD
- Multi-plateformes (IL)
- Plusieurs versions successives (actuellement 8.0)

Développement C#

- Applications Windows
- Pages ASP.NET
- Services Web
- Services Windows
- C#.Net est Multi-plateformes

IDE :

- **Microsoft Visual Studio** payant ou version Community
- **Rider** payant et multi-plateforme
- **SharpDevelop** ou **MonoDevelop** open source mais moins performants



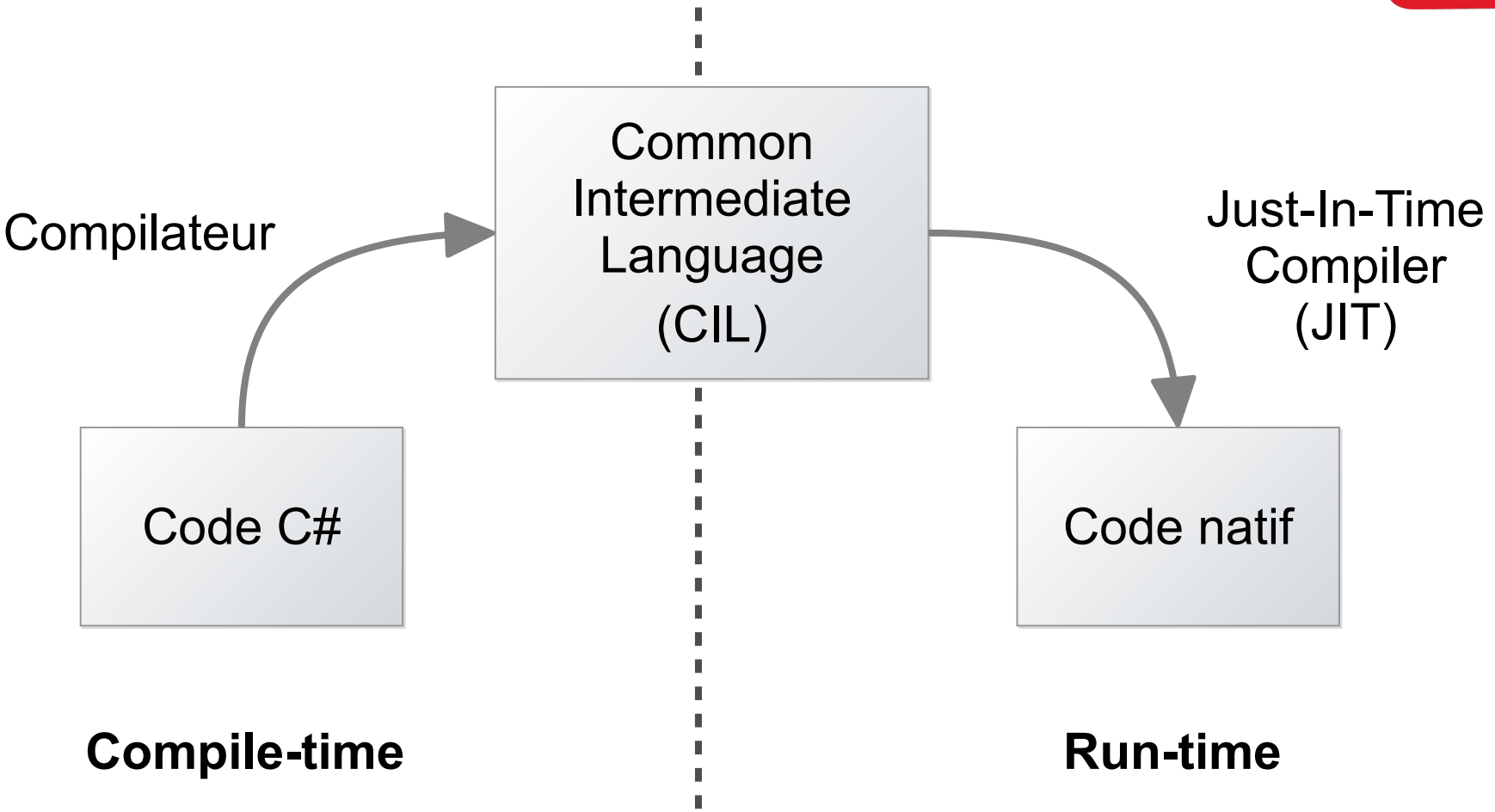

Programme C#

- Espace de noms
- Classes
- Méthode main
- Classe Console (ReadLine et WriteLine)

```
using System;

namespace MyProgram
{
    class HelloWord
    {
        static void Main()
        {
            Console.WriteLine("Hello World !");
        }
    }
}
```

Étapes de compilation



En ligne de commande :

csc.exe msbuild.exe

Débogage et Exécution

- **Localisation et correction des erreurs**
  - Erreurs et débogage JIT
  - Points d'arrêts et pas-à-pas
  - Examen et modifications des variables
- **Exécution**
  - IDE (Start Without Debugging)
  - Ligne de commande (nom de l'application)

Dawan
Training is our way

Syntaxe de base

Base du langage

- Les instructions se terminent par un **;**
- Différences entre **minuscule** et **MAJUSCULE**
- Espaces / Tabulations / CR / LF sans conséquences
- Les fichiers sources sont encodés en **UTF-8**
- Bloc de code : suite d'instructions entre **{ }**
- Commentaire

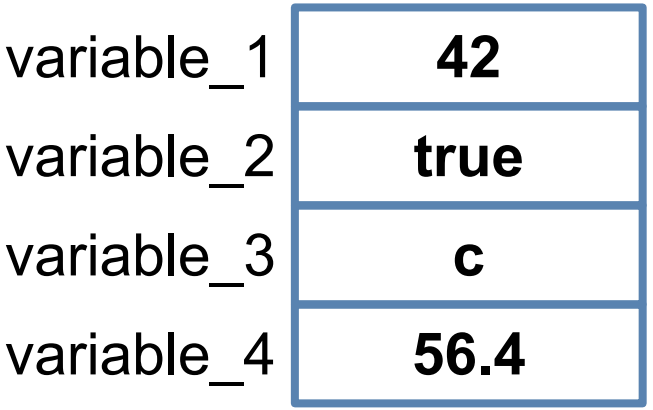
```
// Commentaire fin de ligne

/* Commentaire
   sur plusieurs lignes
*/
```

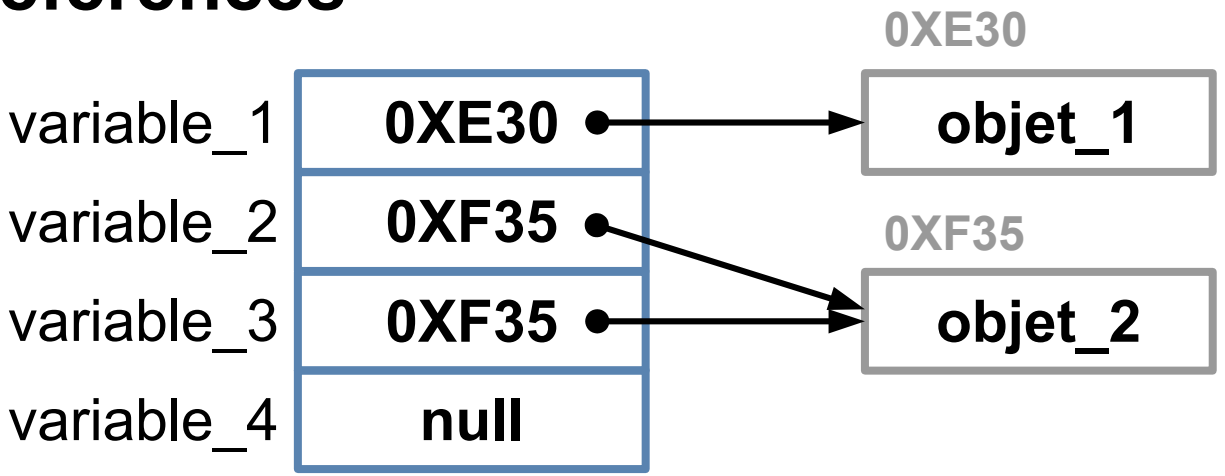
```
/// <summary>
///   Commentaires interprétés
/// </summary>
/// <param name="val"></param>
/// <returns></returns>
```

Types de données

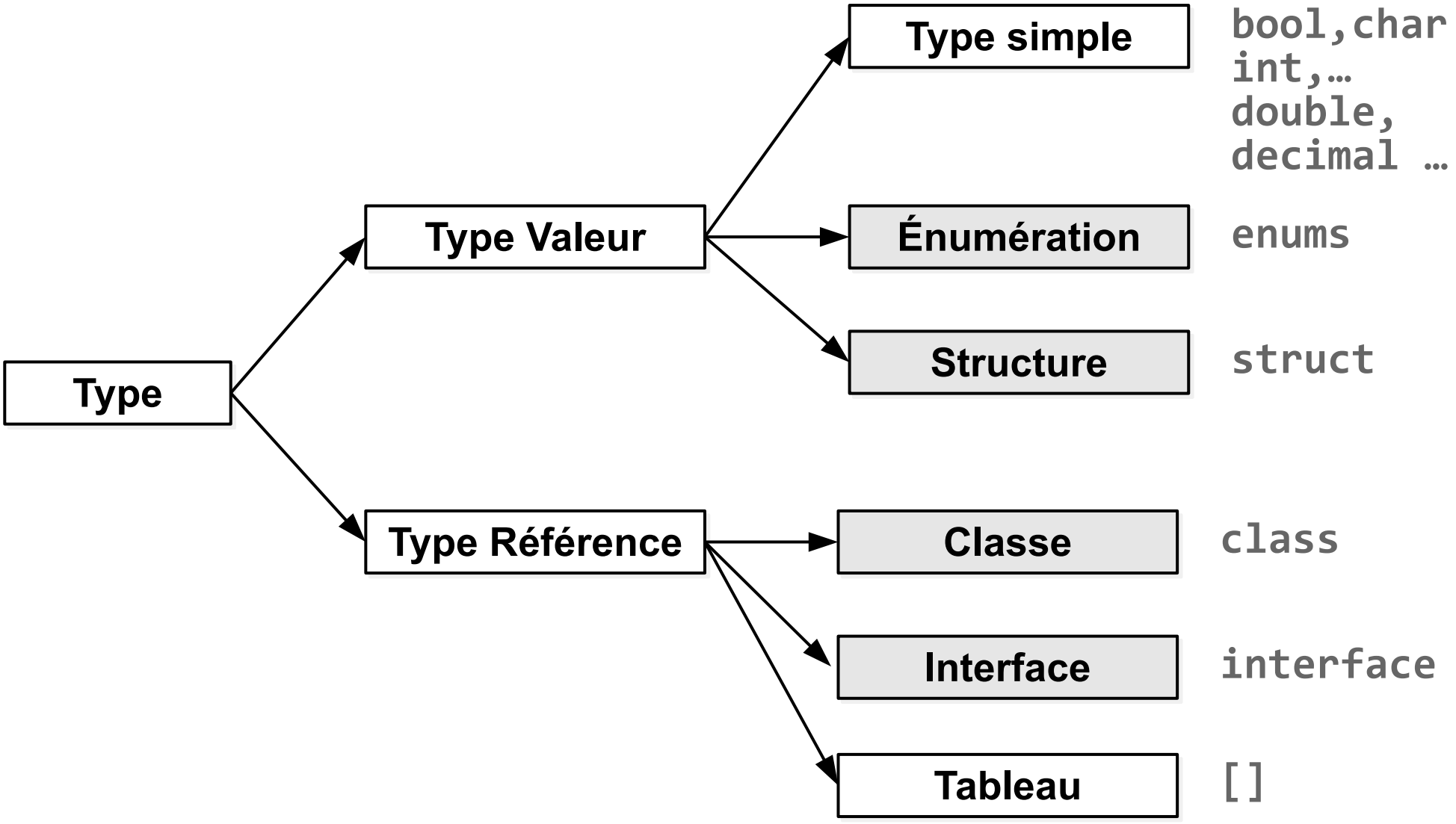
- **Types Valeurs**



- **Types Références**



Types de données



Types simples

Type	Type .NET	Description	Plage de valeurs
bool	System.Bool	booléen	true ou false
char	System.Char	caractère unicode sur **16** bits	'\u0000' à '\uFFFF'
sbyte	System.SByte	entier signé sur **8** bits	-128 à 127
byte	System.Byte	entier non signé sur **8** bits	0 à 255
short	System.Int16	entier signé sur **16** bits	-2^{15} à $2^{15}-1$
ushort	System.UInt16	entier non signé sur **16** bits	0 à $2^{16}-1$
int	System.Int32	entier signé sur **32** bits	-2^{31} à 2^{31}
uint	System.UInt32	entier non signé sur **32** bits	0 à $2^{32}-1$
long	System.Int64	entier signé sur **64** bits	-2^{63} à $2^{63}-1$
ulong	System.UInt64	entier non signé sur **64** bits	0 à $2^{64}-1$
float	System.Single	réel signé sur **32** bits	±1,5 x 10^{-45} à ±3,4 x 10^{38} **précision** → 6 à 9 chiffres
double	System.Double	réel signé sur **64** bits	±5,0 × 10^{-324} à ±1,7 × 10^{308} **précision** → 15 à 17 chiffres
decimal	System.Decimal	réel signé sur **128** bits	±1,0 x 10^{-28} to ±7,9228 x 10^{28} **précision** → 28 à 29 chiffres

Variables

- **Déclaration**

```
type nomVariable;
```

```
double valeur;
int i, j;
```

- **Initialisation**

```
nomVariable = valeur;
```

```
valeur = 134.8;
i = 42;
j = 0;
```

- **Initialisation pendant la déclaration**

```
char c = 'a';
double hauteur = 1.25, d = 1.26;
```

Variables

- **Typage déterminé par le compilateur**

  `var nomVariable = valeur`

  Pour les variables locales, avec le mot-clef **var** le compilateur peut déduire le type de la variable à partir du type de la valeur

  La variable doit être obligatoirement initialisée à la déclaration

```
int ie = 10;     // typé explicitement
var it = 10;     // typé implicitement
var x = ie;                      // x → int
var str = "type implicite";  // str → string
var hauteur = 123.65F;           // hauteur → float
```

Règle de nommage des identifiants

- Le nom doit commencer par : une **lettre** ou _
- Les **nombres** sont autorisés **sauf en tête**
- Ne doit pas être un mot réservé, sauf s'il est préfixé avec **@**

Correct	→	identifier	conv2Int	_test	@if
Faux	→	**3**dPoint	**public**	*$coffe	**while**

- Par convention on utilise le :
  - **P**ascal**C**ase pour les noms de membre, de type, d'espace de noms ou tous les éléments publics
  - **c**amel**C**ase pour les noms de paramètres ou tous les éléments privés

Porté d'une variable locale

- Sa portée se limite au bloc où elle est définie. Leur espace mémoire est libéré lorsque le bloc se termine (pile LIFO)

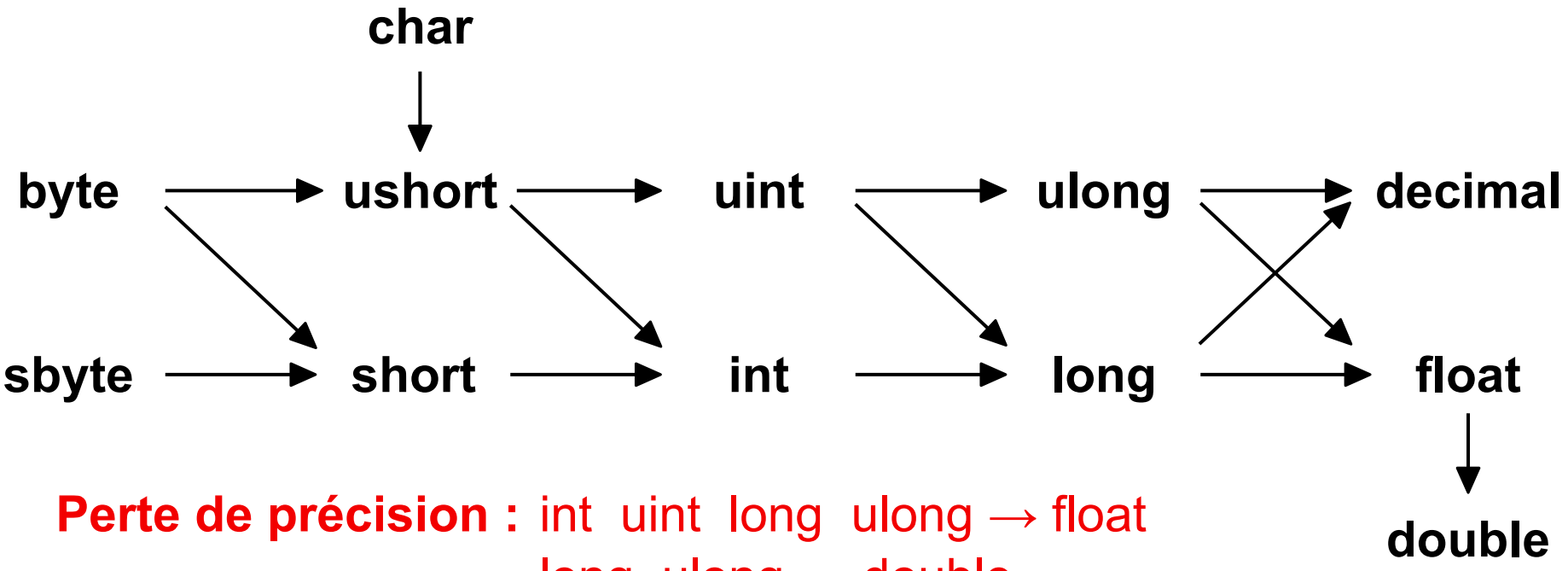
  ↳ **Chaque bloc de code a sa propre portée**

- Quand des blocs contiennent d'autre blocs
  Les blocs contenus peuvent faire référence aux variables du bloc conteneur mais pas l'inverse

```
int a = 10;
if (a > 0)
{
    int somme = a + 20;        // OK
}
Console.WriteLine(somme);      // erreur
```

Transtypage implicite (Automatique)

- Type inférieur vers un type supérieur
- Pour les types réel uniquement : float → double



Perte de précision : int uint long ulong → float
long ulong → double

```
int i = 3;
double d = 2 * i;
```

Transtypage explicite (cast)

- Type supérieur vers un type inférieur
- Réel vers un entier
- double ou float → decimal

```
type variable = (type) variableToCast;
```

```
int i = 123;
short s = (short)i;

double d = 44.95;
decimal dec = (decimal)d;

float f = 3.45f;
sbyte b1 = (sbyte)f;

// Dépassement de capacité
int j = 130;
sbyte b2 = (sbyte)j; // b2 vaut -126
```

Fonctions de conversions

- La classe **Convert** contient des méthodes statiques permettant la conversion de type

  Les types de base pris en charge sont **Boolean**, **Char**, **SByte**, **Byte**, **Int16**, **Int32**, **Int64**, **UInt16**, **UInt32**, **UInt64**, **Double**, **Decimal**, **DateTime** et **String**

```
string s = "2.81";
double d = Convert.ToDouble(s);
int i = Convert.ToInt32(d);
int j = Convert.ToInt32("42");
```

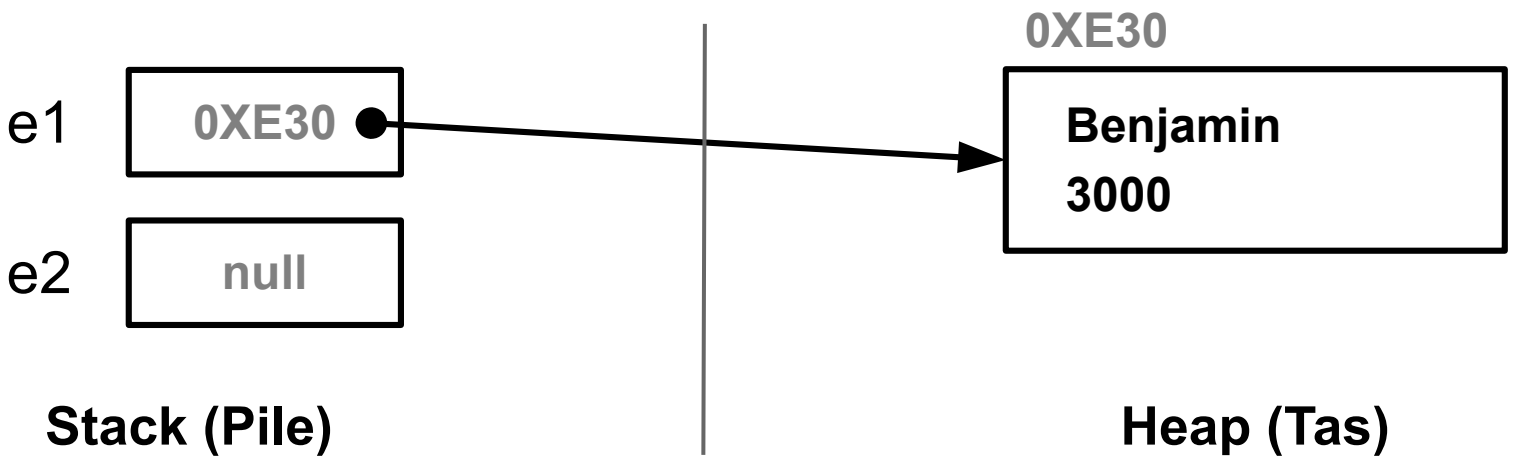
- Les classes **Int32**, **Double**... ont des méthodes **Parse** et **TryParse** qui permettent la conversion d'une chaîne de caractère vers ce type

```
int i = Int32.Parse("1258");
double d ;
bool b= Double.TryParse("1258", out d)
```

Type référence

- Il existe des types complexes :
  - string (chaîne de caractère)
  - tableau
  - …

```
Employe e1 = new Employe("Benjamin", 3000);
Employe e2 = null;
```



Nullable

- Un type **nullable** permet d'utiliser une valeur **null** dans un type valeur

  `type? nomVariable`

```
int? i = null;      // autorisé
```

- La propriété **HasValue** permet de tester si la variable contient une valeur. On peut tester aussi l'égalité à **null**

```
if (i.HasValue)     // équivaut à i!=null
{    // …
```

- La conversion d'un type vers un type nullable est implicite
- La conversion d'un type nullable vers un type est explicite ou obtenu avec la propriété **Value**

```
int? i = 1;
int j = (int)i;     // équivaut i.Value ;
```

Constantes

```
const type nom-constante = valeur;
```

- Une constante doit obligatoirement être initialisée à la déclaration
- Seul les **types valeurs** et les **string** peuvent être des constantes

```
const int A = 26;
const float Pi = 3.14159;
const string Constante = "constante";
const int X;// erreur, une constante doit être initialisée
A = 4;      // erreur, une constante ne peut être modifiée
```

Énumérations

- Une énumération est un type de données, dans lequel une variable ne peut prendre qu'un nombre restreint de valeurs qui sont des constantes prédéfinies

```
enum non : type { enumerator = constexpr,… }
```

```
enum Direction { NORD, EST, SUD, OUEST };
Direction dir = NORD;
```

- Par défaut, une énumération est de type **int** , on peut définir un autre type qui ne peut-être que de type **intégral**
- Par défaut, Les membres d'une énumération sont numéroté séquentiellement et commence à **0**
  On peut définir pour chaque membre une valeur à la déclaration

```
enum Direction : byte { NORD=1, EST=15, SUD=30, OUEST=45 }
```

Énumérations

- **Énumération comme indicateurs binaires**
  - Par convention marquée avec l'attribut **[Flags]**
  - Il faut assigner explicitement tous les membres, pour éviter toute ambiguïté (généralement puissance de 2)

```
[Flags]
enum Options
{
    ToitOuvrant = 1, Climatisation = 2,
    FeuxAntiBrouillard = 4, JantesAlu = 8
}

static void Main()
{
    Options opts = Options.ToitOuvrant |
    Options.JantesAlu;
    Console.WriteLine(opts);
}
```

Structures

- Une structure est un type de données de type valeur
- Elle permet de créer une seule variable contenant des données liées à différents types de données
- Le mot-clé **struct** est utilisé pour créer une structure

```
struct Automobile
{
    public int puissance;
    public string couleur;
    public string marque;
};
static void Main(string[] args)
{
   Automobile maVoiture;
   maVoiture.puissance = 6;
   // …
}
```

Opérateurs

- Arithmétiques : **- + * / %**
- Incrémentation : **++**
  Décrémentation : **--**
  } pré : ++var et post : var++
- Affectations : **= += -= *= /= %= &= |= ^= ~= <<= >>=**
- Relationnels : **== != < > <= >=**
- Logiques : **! || &&**
- Binaires (bit à bit) : **~** (complément) **|** (ou) **&** (et) **^** (ou exclusif)
  (décalage) : **<< >>**
- Autre : **??** (fusion de valeur nulle)

Promotion numérique

La promotion numérique rend compatible le type des opérandes avant qu'une opération arithmétique ne soit effectuée

- **Opérateur unaire** (+, - , ~)
  - **sbyte**, **byte**, **short**, **ushort**, **char** sont promus en **int**
  - **uint** est promu **long**
- **Opérateur Binaire**
  - Le type le **+ petit** est promu vers le **+ grand** type des deux sauf pour les cas suivants, où l'on obtiendra une **erreur** :
    - **decimal** si l'autre opérande est de type : **float** ou **double**
    - **ulong** si l'autre opérande est de type : **sbyte**, **short**, **int** ou **long**
  - **uint** avec **sbyte**, **short** ou **int** : les 2 sont promus en **long**
  - **sbyte**, **byte**, **short**, **ushort**, **char** sont promus en **int**

Condition: if

```
if (condition) {
    // bloc d'instructions 1 (condition vrai)
}
else {
    // bloc d'instructions 2 (condition fausse)
}
```

- **else** n'est pas obligatoire
- On peut **imbriquer** les **if / else**

```
int i = 25;
if (i == 22) {
    // traitement 1
}
else if (i == 25) {
    // traitement 2
}
else {
    // traitement par défaut
}
```

Condition: switch / case

```
switch (variable)
{
    case valeur1:
        // si variable à pour valeur valeur1
        break;
    case valeur2:
    case valeur3:
        // si variable à pour valeur valeur2 ou valeur3
        break;
    default:
        // si aucune valeur des cases ne correspond
        break;
}
```

- La variable peut être de type:
  char, **bool**, valeur intégrale(**int**, **long** …), **string** ou **enum**
- La valeur de case doit avoir une valeur constante
- **default** n'est pas obligatoire

Condition: switch / case

- **Exemple**

```
int jours = 7;
switch (jours)
{
    case 1:
        Console.WriteLine("Lundi");
        break;
    case 6:
    case 7:
        Console.WriteLine("Week end !");
        break;
    default:
        Console.WriteLine("Un autre jour");
        break;
}
```

Condition: opérateur ternaire

condition? condition_vraie **:** condition_fausse**;**

Utilisation : affectation conditionnelle

```
int i = 25;
string resultatTest = (i < 25) ? "Inf à 25":"Sup à 25";
```

équivaut à

```
int i = 25;
string resultat;
if (i < 25)
{
    resultat = "Inf à 25";
}
else
{
    resultat = "Sup à 25";
}
```

Boucle: while

```
while (condition)
{
    // instructions à exécuter
}
```

Tant que la condition est vérifiée, le bloc d'instructions est exécuté

```
int i = 0;
int somme = 0;
while (i <= 10)
{
    somme = somme + i;
    i = i + 1;
}
Console.WriteLine("Somme= " + somme);
```

Boucle: do while

```
do
{
   // instructions à exécuter
} while (condition);
```

Identique à **while**, sauf que le test est réalisé après l'exécution du bloc

```
int i = 0;
int somme = 0;
do
{
    somme = somme + i;
    i = i + 1;
} while (i <= 10);
Console.WriteLine("Somme= " + somme);
```

Boucle: for

```
for (initialisation; condition; itérateur)
{
    // instructions à exécuter
}
```

1. **initialisation** est exécutée
2. si **la condition** est fausse ⟶ on sort de la boucle
3. le bloc d'instruction est exécuté
4. **itérateur** est exécuté

```
for (int i = 0; i < 10; i = i + 1)
{
    Console.WriteLine("i= " + i);
}
```

Instructions de saut

- **break**
  termine le traitement de la boucle ou du switch courant
- **continue**
  passe à l'itération suivante dans un traitement de boucle
- **goto**
  permet de se brancher sur une instruction étiquetée
  utilisé pour :
  - transférer le contrôle à un case ou à l'étiquette par défaut d'une instruction **switch**

```
goto case 1;
```

  - quitter des boucles fortement imbriquées

```
goto label;
//..
label:
```

Dawan
Training is our way

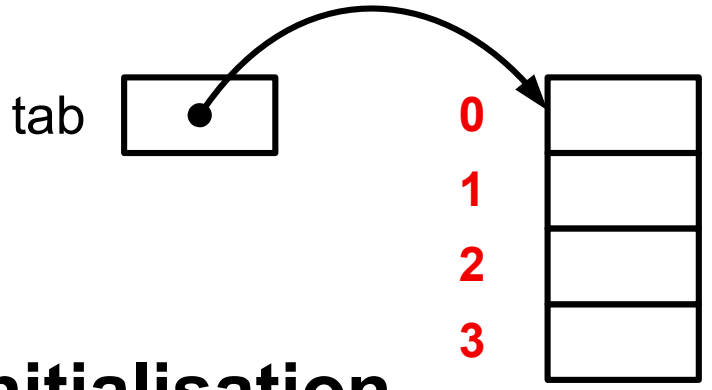
Tableaux

Tableaux

- **Déclaration d'un tableau**

```
type[] nom_tableau = new type[taille_tableau];
```

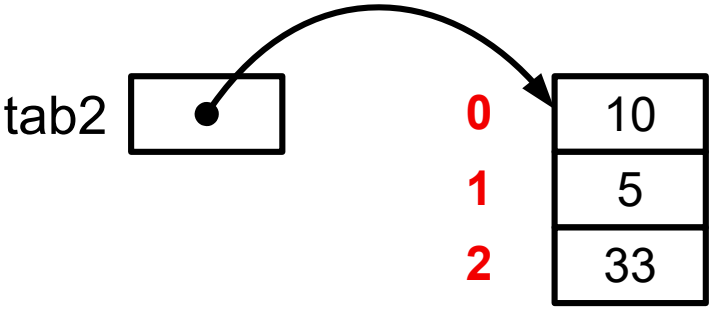
```
int[] tab = new int[4];
```



- **Déclaration d'un tableau avec initialisation**

```
type[] nom_tableau = { valeur1, valeur2, … };
```

```
int[] tab2 = { 10, 5, 33 };
```



Tableaux

- **Accès à un élément d'un tableau**

  `nom_tableau[indice]`

  L'indice d'un tableau commence à 0

```
int[] tab = { 10, 30, 40 };
Console.WriteLine(tab[0]);   // affiche 10
```

- **Taille d'un tableau**

  `nom_tableau.Length`

```
int[] tab = new int[20];
int n = tab.Length;        // n a pour valeur 20
```

Tableaux à 2 dimensions

```
type[,] nom_tableau = new type[ligne,colonne];

type[,] nom_tableau = {  {val00,val01,… },
                         {val10,val11,… },… }
```

```
int[,] tab2d = new int[4, 2];
int[,] tab2dInit = { { 10, 3 }, { 12, -9 },
                     { 1, -1 }, { 4, 1 } };
Console.WriteLine(tab2dInit[1, 1]);  // affiche -9
```

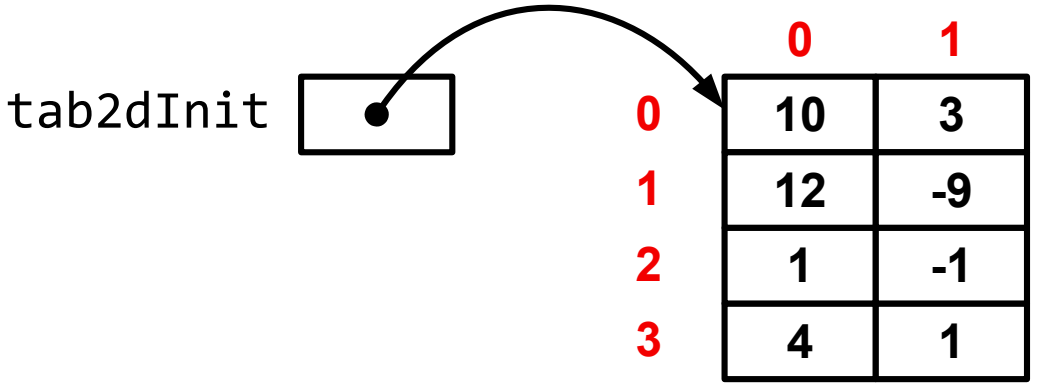


Tableau multidimensionnels


```
type[, , …,] nom_tableau = new type[d1,d2 … ,dn];
```

```
int[,,] cube = new int[5, 10, 5];
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
array3D[1, 0, 2] = 5; // remplace 9 par 5
```

- **Taille d'un tableau multidimensionnel**

  `int GetLength(int dimension);`

  retourne le nombre d'éléments en fonction de la dimension (commence à 0)

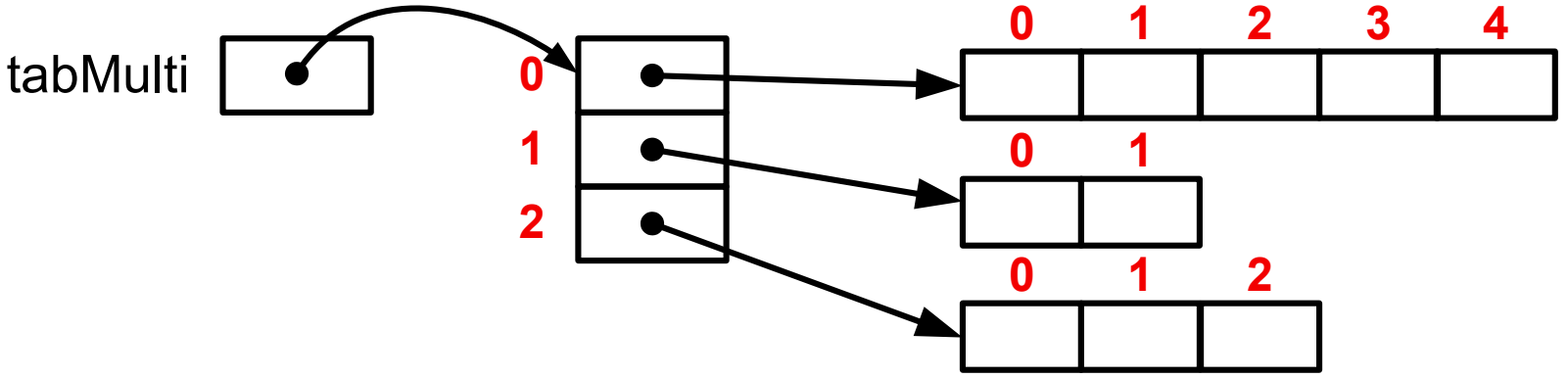
  La propriété **Rank** contient le nombre dimension

```
int[,,] cube = new int[5, 2, 3];
cube.GetLength(2); // retourne 3
cube.Length; // retourne 30, le nombre d'éléments du tableau
```

Tableaux de tableaux (Tableaux en escalier)

```
type[][] nom_tableau = new type[nb_ligne][];
nom_tableau[0] = new type[nb_colonne1];
        :              :              :
nom_tableau[nb_ligne - 1] = new type[nb_colonneN];
```

```
int[][] tabMulti = new int[3][];
tabMulti[0] = new int[5];
tabMulti[1] = new int[2];
tabMulti[2] = new int[3];
int[][] tabMulti2 = new int[][] { new int[] { 10, 3, 4, 1, 5 },
                  new int[] { 12, -9 }, new int[] { 1, -1, 12 }};
```



Tableaux: itération complète (foreach)


```
foreach (type variable in tab)
{
    // instructions à exécuter
}
```

```
int[] tab = { 2, 4, 6 };
foreach (int val in tab)      // → affiche :  2
{                             //              4
   Console.WriteLine(val);    //              6
}

int[,] tab2dInit = { { 10, 3 }, { 12, -9 },
                     { 1, -1 }, { 4, 1 } };
foreach( int a in tab2dInit)
{
   Console.Write(a + ",");  // → affiche : 10,3,12,
                                          -9,1,-1,4,1,
}
```

Dawan
Training is our way

Méthodes et paramètres

Méthodes

- **Une méthode permet de**
  - diviser le code en morceaux (réutilisabilité, clarté)
  - factoriser le code
- **Déclaration**

```
typeDeRetour nom(type param1, type param2, …) {
    // instructions à exécuter
}
```

```
int somme(int a, int b)
{
    return a + b;
}
```

- **Appel**

```
nomMéthode(paramètres);
```

```
int num = 20;
int res = somme(num, 22);
```

Méthodes

- **Type de retour**
  - Il est obligatoire. S'il n'y en a pas → **void**
- **Corps de la méthode**
  - au minimum **{ }**
  - doit contenir au moins une instruction **return**
  - pour **void**: **return;** ou il **peut être omis**
- **Arguments**
  - Ils sont séparés par **,**
  - Pour leurs portés, ils sont considérés comme des variables locales à la méthode

Passage de paramètres

- **Passage par valeur (en entrée)**

```
void Inc(int x) {
    x ++;
}

void f() {
    int val = 3;
    Inc(val);          // val == 3
}
```

- **Passage par référence avec ref (en entrée, en sortie)**

```
void Inc(ref int x) {
    x ++;
}

void f()      {
    int val = 3;
    Inc(ref val);    // val == 4
}
```

Passage de paramètres

- **Passage par référence avec out (en sortie)**

```
void Method(int val, out int next)
{
    next = val * 2;
}

void f()
{
    int i = 1, next;
    Method(i, out next);
}
```

  - On peut de déclarer la variable de retour dans les arguments pendant l'appel de la fonction

```
Method(i, out int next);
```

  - On peut ignorer un paramètre **out** en le nommant _

```
Method(i, out _);
```

Passage de paramètres

- **Paramètre optionnel**
  - Ils ne peuvent pas être **ref** ou **out**
  - Ils doivent se trouver en fin de liste des arguments

```
void MethodOptParams(int required, string optStr =
                                "default", int optInt = 10)
{
   Console.Write("{0} {1} {2}", required,optStr,optInt);
}
MethodOptParams(4) ;              // affiche → 4 default 10
MethodOptParams(6, "other") ; // affiche → 6 other 10
```

- **Paramètre nommées**
  On identifie les arguments par leur noms

```
MethodOptParams(4, optInt: 4);
MethodOptParams(4, optStr: "param");
MethodOptParams(optStr: "param", required: 4);
```

Nombre d'arguments variable (params)

- Un paramètre avec le mot clef **params**
  - accepte un nombre de paramètre variable d'un même type
  - doit toujours être le dernier paramètre
  - doit être déclaré en tant que tableau

```
TypeRetour nomMethode(params type[] nomParam)
```

```
static void MaFonction(string sts, params string[] strs) {
        // …
 }

MaFonction("Test", "Nb", "De", "Parametres", "Inconnu");

string[] tableauParametres = new string[]{"Nb","De",
                                          "Parametres","Inconnu"};
MaFonction("Test", tableauParametres);
```

Surcharge de méthode

- Plusieurs fonctions peuvent avoir le même nom et des arguments différents en nombre ou/et de type
- Le type de retour n'est pas pris en compte

```
void MaFonction(int param1)
{
    // ...
}
int MaFonction(int param1, string param2)
{
    // ...
}

void MaFonction(int param1, int param2)
{
    // ...
}
```

Récursivité

- **Capacité d'une méthode à s'appeler elle-même**

```
int factorial(int n)      // factoriel= 1* 2* … n
{
    if (n <= 1)           // condition de sortie
    {
        return 1;
    }
    else
    {
        return factorial(n - 1) * n;
    }
}
-------------------------------------------------------------------------
int f = factorial(3);     // f vaut 6
```

Appels successifs

```
factorial(3)= factorial(2) * 3
    factorial(2)= factorial(1) * 2
        factorial(1)
```

Remontée des résultats

```
factorial(3) = (2) * 3 = 6
    factorial(2) = (1) * 2 = 2
        factorial(1) = 1
```

Condition de sortie n=1

Méthode main

```
static void Main(string[] args)
{
   // instructions à exécuter
}
```

- point d'entrée du programme
- doit être statique
- ne doit pas être public
- peut avoir comme type de retour **void** ou **int**
- peut être déclarée avec ou sans paramètre **string[]**, qui contient des arguments de ligne de commande
- dans un programme, il ne peut y avoir qu'une seule classe contenant un main. Dans le cas contraire, il faut compiler avec l'option **-main** pour préciser le main utilisé