



TP2

Introduction C#

Melekhova O. - *melekhova@gmail.com*,

Langages de POO : Python, JEE, C#

Création

Java	<p>créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld.</p> <p>La société Sun a été ensuite rachetée en 2009 par la société Oracle qui détient et maintient désormais Java.</p>
C#	<p>commercialisé par Microsoft depuis 2002 et destiné à développer sur la plateforme Microsoft .NET.</p>
Python	<p>créé par Guido van Rossum et a été rendu public pour la première fois en 1991.</p> <p>Le langage Python est placé sous une licence libre proche de la licence BSD</p>

Langages de POO : Python, JEE, C#

Performance

Java	Les deux sont compilés dans un Bytecode, qui est produit par le compilateur. Le Bytecode est compilé dans un code machine au moment de l'exécution avec l'interpréteur Just-In-Time, également connu sous le nom de JIT.
C#	
Python	Python est un langage interprété. L'interpréteur exécute directement chaque ligne de code à l'exécution, l'une après l'autre. Cela signifie qu'un code peut s'exécuter pendant un certain temps, puis s'arrête au moment où il trouve une erreur. En Python, la vérification de type se produit au moment de l'exécution, ce qui a tendance à affecter les performances. Il n'est pas rare qu'un programme Python prenne X 2 ou plus de temps pour exécuter le même programme que de nombreux autres langages.

Langages de POO : Python, JEE, C#

Verbosité & formatage

Java	<pre>public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } }</pre>
C#	<pre>using System; namespace HelloWorld { class Program { static void Main(string[] args) { Console.WriteLine("Hello World!"); } } }</pre>
Python	<pre>print ("Hello World")</pre>

Langages de POO : Python, JEE, C#

Fortement typé VS faiblement typé

Java	Java et C# sont des langages à typage fort. Lorsque vous définissez une variable dans ces langues vous devrez définir explicitement son type.
C#	- pas possible de changer le type de variable en le déclarant à nouveau, il sera considéré comme une variable en double
Python	Python est un langage à typage dynamique. Vous pouvez déclarer une variable, lui attribuer un numéro, puis lui attribuer une valeur de chaîne à un autre moment sans problème.

Exemple :

```
Valeur = 5      # La valeur sera traitée comme un  
                entier/nombre à partir de maintenant  
Valeur = 'Salut' # La valeur est une chaîne maintenant
```

Langages de POO : Python, JEE, C#

Rétrocompatibilité

Java	La machine virtuelle Java peut virtuellement exécuter n'importe quelle application écrite et compilée sur les versions précédentes.
C#	C# et .NET Framework sont généralement rétrocompatibles.

Python

Python a un problème de compatibilité, car Python 2.0 est assez différent de Python 3.0, ce qui a causé un grand fossé entre les bases de code et les développeurs des deux.

Exemple :

```
print "Hihi"    # fonctionne uniquement avec Python 2
print("Hihi")   # fonctionne avec Python 2 et 3
```

Langages de POO : Python, JEE, C#

Facilité d'apprendre

Java	Java et C# ont leurs propres avantages. Il est beaucoup plus facile d'entrer dans la programmation orientée objet avec eux, ce qui est un concept très courant dans les langages de programmation aujourd'hui.
C#	
Python	Python est le langage le plus facile car il est beaucoup moins strict si l'objectif est d'apprendre à écrire des algorithmes, ce qui est le but de la programmation.

Langages de POO : Python, JEE, C#

Application

Java	Applications Web Applications et logiciels scientifiques Applications Android Centres de données Applications basées sur le cloud Les entreprises célèbres qui utilisent Java sont Google, Netflix, Airbnb, Instagram, Amazon.
C#	Applications Windows Jeux vidéo des applications Web Software d'entreprise Applications basées sur le cloud Les principales entreprises qui utilisent C# sont Microsoft, Alibaba, Stack Overflow, Intuit.
Python	Systèmes d'exploitation Data Science & Data Visualization Extraction de données ou Data Mining Machine learning Développement de jeux et graphiques 3D Les principales entreprises qui utilisent Python sont Google, Facebook, Instagram, Yahoo, Dropbox, Youtube, Nasa

C#

C# est un langage de programmation orienté objet.
Similaire à Java

70% Java, 10% C++, 5% Visual Basic, 15% new

Comme en Java

- Orienté objet (héritage simple)
- Interfaces
- Exceptions
- Threads
- namespace (comme package)
- Typage forte
- Ramasse-miettes
- Réflexion
- Chargement dynamique du code

Comme en C++

- Surcharge d'opérateur
- Pointeur et code unsafe
- Quelques détails syntaxiques

C#

C# est un langage de programmation orienté objet.
Similaire à Java

En plus de Java

- Reference and output parameters
- Objects on the stack (structs)
- Matrices
- Enumerations
- Système de type unifié
- goto
- Gestion de versions

« Sucre syntaxique »

- Programmation à la base de composants
 - Properties
 - Events
- Delegates
- Indexers
- Operator overloading
- foreach statements
- Boxing/unboxing
- Attributes
- etc

Fichier Hello.cs :

```
using System;

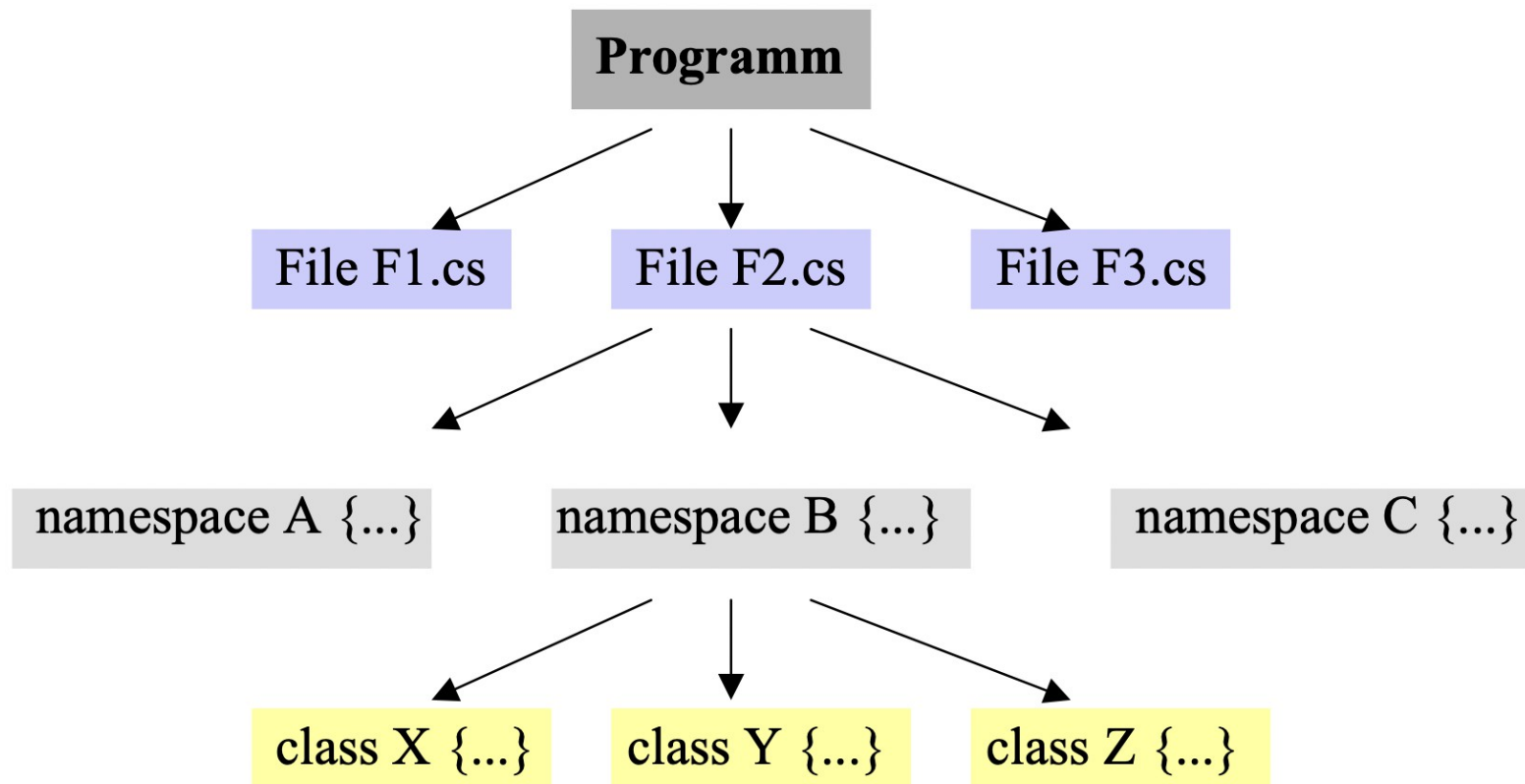
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)    {
            Console.WriteLine("Hello World!");
        }
    }
}
```

- utilise namespace *HelloWorld*
- le point d'entrée doit être appelé *Main*
- le nom de fichier et le nom de classe ne sont *pas* identiques

Compilation (in the Console window)

csc Hello.cs

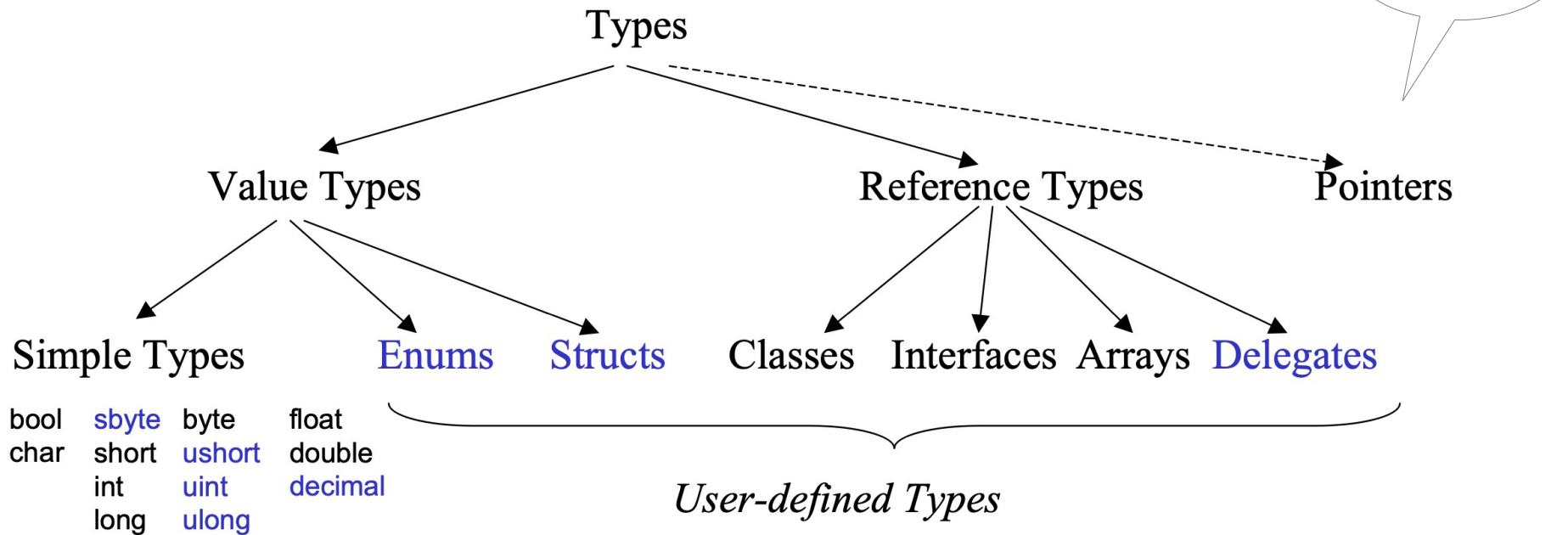
Structure de C# Projet



- Si aucun namespace n'est spécifié => par défaut anonyme
- Les namespaces peuvent également contenir de structures, de interfaces, de délégués et d'énumérations
- namespace peut être « ouvert » dans d'autres fichiers
- Cas le plus simple : fichier unique, namespace par défaut, classe unique

Systeme de type unifié

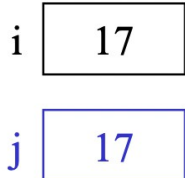
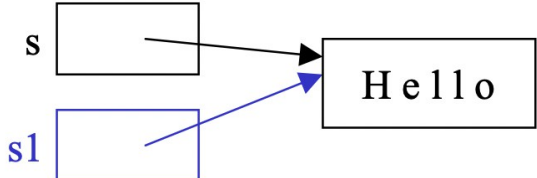
Système de type unifié



Tous les types sont compatibles avec l'objet

- peut être affecté à des variables de type objet
- toutes les opérations de type objet sont applicables

Système de type unifié

	Value Types	Reference Types
variable contains	value	reference
stored on	stack	heap
initialisation	0, false, '\0'	null
assignment	copies the value	copies the reference
example	<pre>int i = 17; int j = i;</pre> 	<pre>string s = "Hello"; string s1 = s;</pre> 

Value type/Variables

Une variable est un **espace mémoire** dans lequel une valeur **y** est stockée. Une variable possède un **type** et un **nom**.

Le **type**

- détermine la **place mémoire** occupée par la variable
- détermine les **valeurs possibles/acceptables** qui pourront être stockées (une à la fois !) dans la variable.

e.g. une variable de type short peut stocker, sur 2 octets, uniquement des valeurs entières comprises entre -32 768 et +32 767

- permet **d'interpréter** la valeur

e.g. 0|0|0|0|0|0|0|0 0|1|1|0|0|0|0|1

- correspond à l'entier (type short) 97
- correspond au caractère (type char) 'a'

Le **nom** permet de faire référence de manière symbolique à cet espace mémoire dans le code.

- identifieur unique.

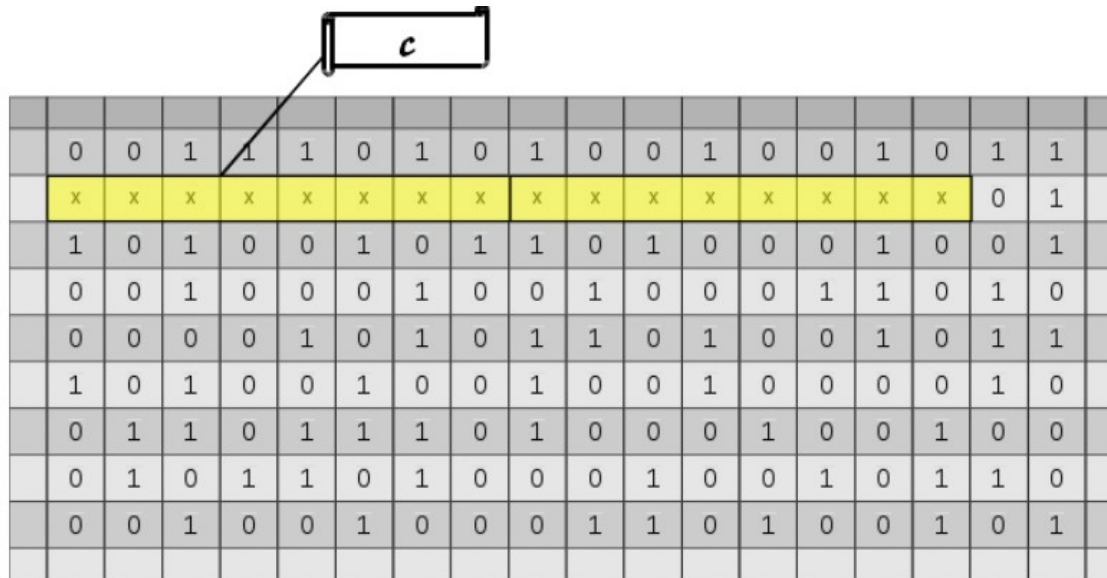
Value type/Variables

Déclaration, affectation et initialisation

CODE :

```
1 char c; // variable de type « caractère »
```

zone en mémoire réservée (on dit allouée)



Value type/Variables

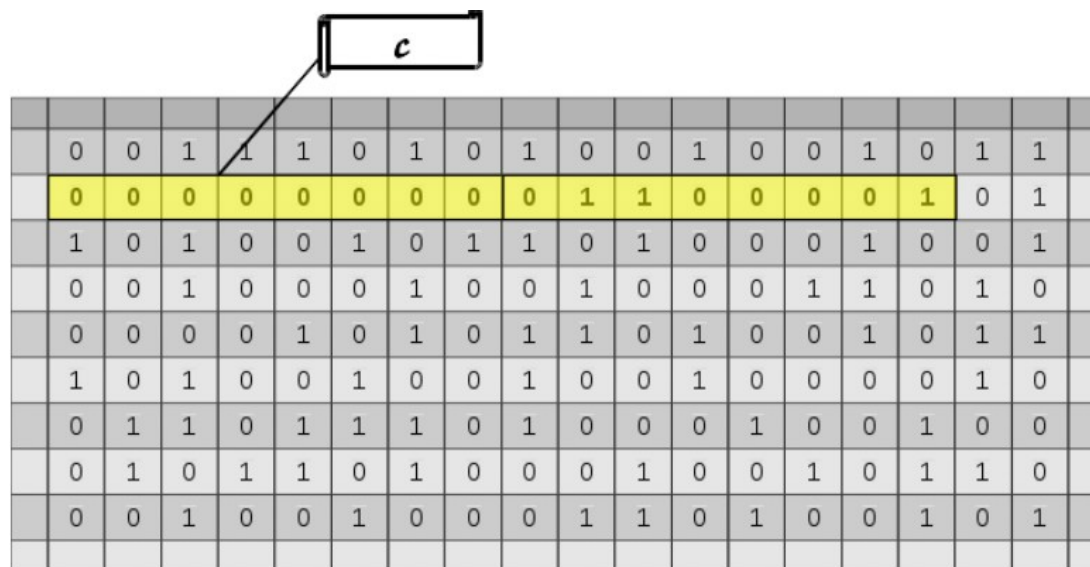
Déclaration, **affectation** et initialisation

CODE :

```
1 char c; // variable de type « caractère »
```

2 // ...

```
3 c = 'a'; // affectation de la valeur 'a' dans la variable c
```



Value type/Variables

Déclaration, affectation et **initialisation**

Déclarer c'est bien... en **initialisant** c'est mieux !

maîtrise de la valeur **dès la création** d'une variable

CODE :

```
1 // on préférera :  
2 int x = 1 ;  
3 int y = 2 ;  
4  
5 // plutôt que :  
6 int x ;  
7 int y ;  
8 x = 1 ;  
9 y = 2 ;
```

Value type/Variables

Déclaration, affectation et initialisation

Déclarer c'est bien... en **initialisant** c'est mieux !

maîtrise de la valeur **dès la création** d'une variable

CODE :

```
char c = 'a';    // variable type «caratère» (2 octets, unicode)
short e = 13;    // variable de type « entier » (2 octets)
int e2 = 42;     // autre variable type « entier » (4 octets)
double r = 1.23; // variable de type « réel » (8 octets)
```

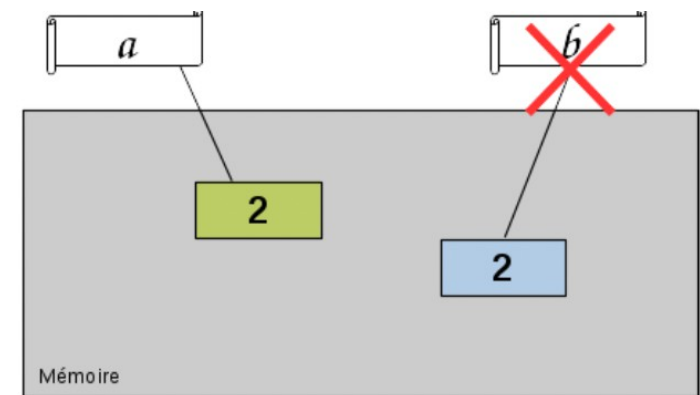
Value type/Variables

La portée des variables

Une variable est :

- **locale au bloc** dans lequel elle est déclarée ;
i.e. accessible dans les instructions qui suivent sa déclaration,
au sein du **bloc où elle est déclarée et des sous-blocs** ;
- « **détruite** » en sortie du bloc où elle est déclarée.
devient inaccessible, la zone mémoire est de nouveau libre...

```
1 // ...
2 {
3  int a = 1 ;
4  {
5  int b = 2 ;
6  a = b ;
7  }
8  Console.WriteLine ( a ) ;
9  Console.WriteLine( b ); // ERREUR, variable détruite
10 }
```



Value type/Variables

Entier : **int** ; opérateurs : +, - , *, / , %

Réel : **double**, op : +, - , *, /

Booléen : **bool** (**true**, **false**); op : !, &&, ||

Chaîne de caractères : **string** ; op : méthodes associées au type String [ex. Substring(), etc.]

Remarque 1 : TRANSTYPAGE (type vers chaîne et inversement)

- nom_de_variable.**ToString**()
- type_destination.**Parse**(chaîne de caractères)

Remarque 2 : Opérateurs de comparaison

- Compare des éléments de même type, et renvoie un booléen • **==**, **!=** , **>** , **>=** , **<** , **<=**

Value type/Enums

Un type **enum** est un type valeur distinct (types valeur) qui déclare un ensemble de constantes nommées.

Déclaration (directement dans namespace) :

```
enum Color {red, blue, green} // values: 0, 1, 2
enum Access {personal=1, group=2, all=4}
enum Access1 : byte {personal=1, group=2, all=4}
```

Utilisation :

```
Color c = Color.blue; // enumeration constants must be qualified
```

```
Access a = Access.personal | Access.group;
if ((Access.personal & a) != 0) Console.WriteLine("access granted");
```

Value type/Enums

Les **opérateurs** suivants peuvent être utilisés sur des valeurs de types enum :

== , != , < > <= , >=	: comparaison d'énumération ;
+, -	: addition, soustraction binaires ;
^ , & , 	: opérateurs logiques d'énumération ;
~	: opérateur de complément au niveau du bit ;
++ et --	: incrémentation et décrémentation.....

Rémarques :

- Les énumérations ne peuvent pas être affectées à **int** (sauf après un transtypage).
- Les types d'énumération héritent de l'objet (*Equals*, *ToString*, ...).
- La classe *System.Enum* fournit des opérations pour énumérations (*GetName*, *Format*, *GetValues*, ...).

Reference type/Array

Déclarer une **matrice**

```
type[ , ] nomDeLaMatrice2D;  
type[ , , ] nomDeLaMatrice3D;  
etc. pour N dimensions
```

=> après ça, la matrice n'existe toujours pas, i.e. la valeur est **null**.

Code :

```
int [ , ] matriceEntiers ; // déclaration d'une matrice à 2D, qui pourra  
                           contenir des entiers
```

```
double[ , , ] matriceReels = null; // déclaration d'une matrice à 3D, qui  
                                     pourra contenir des réels ; ici la  
                                     valeur null est explicitement renseignée.
```

Reference type/Array

Allouer de la mémoire

```
nomDeLaMatrice2D = new type[tailleDim1, tailleDim2];
```

Code :

```
int [ , ] matriceEntiers = null ; // déclaration  
matriceEntiers = new int [2 ,3]; // allocation en mémoire de 6 cases,  
                                sur 2 lignes et 3 colonnes, pouvant  
                                chacune contenir (la valeur d')un entier.
```

Formule deux-en-un : **déclaration + allocation mémoire =>**

« **Création** » d'une matrice

```
type[ , ] nomMatrice2D = new type[tailleDim1, tailleDim2];
```

Code :

```
int [ , ] matriceEntiers = new int [2 ,3];
```

Reference type/Array

Création avec **initialisation** de chaque élément

Code

```
int [,] matrice = new int [2 ,3] { {2,4,6} , {3,5,7} };
```

Code (version simplifiée)

```
int [,] matrice = new int [ ,] { {2,4,6} , {3,5,7} };
```

Code (version encore plus simplifiée)

```
int [,] matrice = { {2,4,6} , {3,5,7} };
```

Reference type/Array

Accéder aux éléments d'une matrice

Code

```
int [ , ] matrice = new int [2 ,3]; // création d'une matrice de int de taille 6
                                   // cases sur 2 lignes et 3 colonnes
Matrice[0, 0] = 3;                // accès en écriture à l'élément (0,0)
Console.WriteLine( matrice[0,0] ); // accès en lecture
```

Reference type/Array

Nombre d'éléments et dimensions d'une matrice

- **Nombre** d'éléments
→ propriété Length
- **Taille** de chaque **dimension**
→ méthode `GetLength(choixDimension)`
les dimensions sont identifiées de 0 à N - 1

Code :

```
Int [ , ] matrice = new int [2, 3];  
Console.WriteLine("GetLength(0) : " + matrice.GetLength(0));  
Console.WriteLine("GetLength(1) : " + matrice.GetLength(1));  
Console.WriteLine("Length : " + matrice.Length);
```

Exemple (Affichage) :

```
GetLength(0) : 2  
GetLength(1) : 3  
Length : 6
```

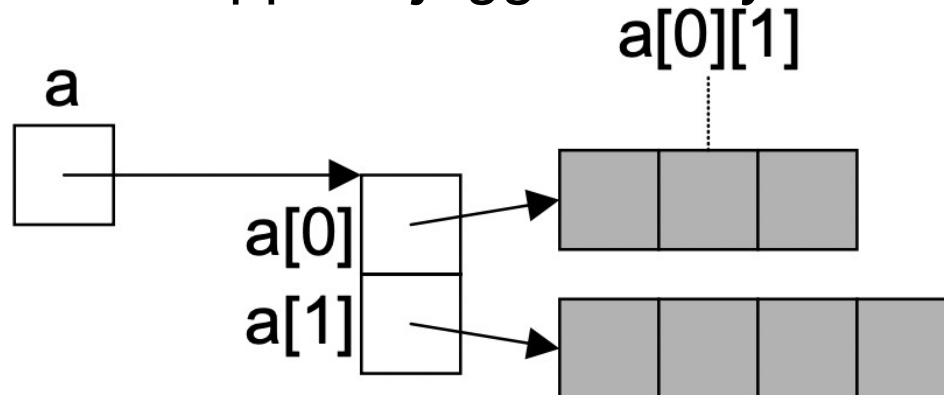
Reference type/Array

La plupart des langages de programmation ne disposent pas de cette notion de matrice.

Par contre, rien n'empêche que chaque élément (i.e. case) d'un tableau soit lui même un tableau... et ainsi de suite.

Il est donc « naturel » de pouvoir manipuler des **tableaux de tableaux**.

=> En anglais, on les appelle *jagged arrays* ou *ragged arrays*



Reference type/Array

Exemple d'un tableau de tableaux d'entiers

Code (Déclaration)

```
int [][] tab2D;           // <=> int[][] tab2D = null;
```

Code (Allocation mémoire du tableau principal)

```
tab2D = new int [2][];
```

Code (Création du 1^{er} élément (i.e. du 1^{er} tableau imbriqué)

```
tab2D[0] = new int[3];  
tab2D[0][0] = 10;  
tab2D[0][1] = 20;  
Tab2D[0][2] = 30;
```

Code (Création du 2nd élément (i.e. du 2nd tableau imbriqué)

```
tab2D[1] = new int [4] {40, 50, 60, 70};
```

Reference type/Classe *System.String*

Classe **System.String** peut être utilisé comme chaîne de type standard

Code :

```
string s = "Dubois";
```

- Les chaînes sont **immuables** (utilisez **StringBuilder** si vous souhaitez modifier les chaînes)
- Peut être concaténé avec + : **"Monsieur " + s**
- Peut être indexé : **s[i]**
- Longueur de chaîne : **s.Length**
- Les chaînes sont des types de référence => sémantique de référence dans les affectations
- mais leurs valeurs peuvent être comparées avec == et != : if (s == "Dubois") ...
- Class String définit de nombreuses opérations utiles : **CompareTo, IndexOf, StartsWith, Substring, ...**

Value type/Structs

Un type de **structure** (ou type **struct**) est un type valeur qui peut encapsuler des données et des fonctionnalités associées. Le struct mot clé vous permet de définir un type de structure :

Déclaration (directement dans namespace) :

```
struct Point {  
    public int x, y;                // fields  
    public Point (int x, int y) { this.x = x; this.y = y; } // constructor  
    public void MoveTo (int a, int b) { x = a; y = b; }    // methods  
}
```

Utilisation :

```
Point p = new Point(3, 4); // constructeur initialise l'object dans stack  
p.MoveTo(10, 20);         // appel fonction
```

Reference type / Classes

Déclaration :

```
class Rectangle {  
    Point origin;  
    public int width, height;  
    public Rectangle() { origin = new Point(0,0); width = height = 0; }  
    public Rectangle (Point p, int w, int h) { origin = p; width = w; height = h; }  
    public void MoveTo (Point p) { origin = p; }  
}
```

Utilisation :

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);  
int area = r.width * r.height;  
r.MoveTo(new Point(3, 3));
```

Expressions

Opérateurs et Priorités

Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ~ ! ++x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	
Conditional AND	&&
Conditional OR	
Conditional	c?x:y
Assignment	= += -= *= /= %= <<= >>= &= ^= =

Contrôle de débordement

Le débordement **n'est pas vérifié par défaut**

Exemple :

```
int x = 1000000;  
x = x * x; // -727379968, pas d'erreur
```

Le contrôle de débordement **peut être activé :**

```
x = checked(x * x); //System.OverflowException
```

```
checked {  
    ...  
    x = x * x; //System.OverflowException  
    ... }  

```

Le contrôle de débordement peut également être activé **au moment de compilation** : `csc /checked Test.cs`

Type et taille

L'opérateur **typeof** obtient l'instance System.Type pour un type.

Exemple :

```
Type t = typeof(int);  
Console.WriteLine(t.Name); // Int32
```

L'opérateur **sizeof** retourne le nombre d'octets occupés par une variable d'un type donné.

Exemple :

```
unsafe {  
    Console.WriteLine(sizeof(int));  
    Console.WriteLine(sizeof(MyEnumType));  
    Console.WriteLine(sizeof(MyStructType));  
}
```

Déclarations

Déclaration

Les entités peuvent être déclarées dans un :

- **namespace** : Déclaration de classes, interfaces, structs, enums, delegates
- **classe , interface, struct** : Déclaration de fields, methods, properties, events, indexers, ...
- **enum** : Déclaration de constantes d'énumération
- **block** : Déclaration de variables locales

Règles de portée :

- Un nom ne doit pas être déclaré deux fois dans le même espace de déclaration.
- Les déclarations peuvent se produire dans un ordre arbitraire.

Exception : les variables locales doivent être déclarées avant d'être utilisées

Règles de visibilité :

- Un nom n'est visible que dans son espace de déclaration (les variables locales ne sont visibles qu'après leur point de déclaration).
- La visibilité peut être restreinte par des modificateurs (private, protected, ...)

Namespaces (imbriqué)

Color.cs

```
namespace Util {  
    public enum Color {...}  
}
```

Figures.cs

```
namespace Util.Figures {  
    public class Rect {...}  
    public class Circle {...}  
}
```

Triangle.cs

```
namespace Util.Figures {  
    public class Triangle {...}  
}
```

```
using Util.Figures;
```

```
class Test {  
    Rect r;           // without qualification (because of using Util.Figures)  
    Triangle t;  
    Util.Color c;     // with qualification  
}
```

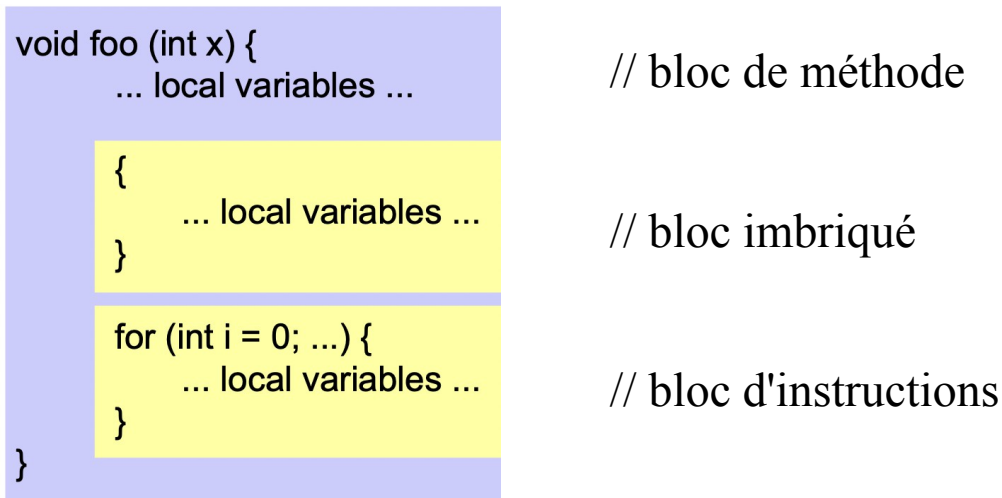
Namespaces externes :

- doivent soit être importé (par exemple, *using Util ;*)
- ou spécifié dans un nom qualifié (par exemple, *Util.Color*)

La plupart des programmes ont besoin de namespace System => *using System;*

Blocks

Différents types de blocks :



Notes :

- Un block comprend des blocks imbriqués.
- Les paramètres formels appartiennent à l'espace de déclaration du block méthode.
- La variable de boucle dans une instruction *for* appartient au bloc d'instruction *for*.
- La déclaration d'une variable locale doit précéder son utilisation

Variables locales

```
void foo(int a) {  
    int b;  
    if (...) {  
        int b;  
        int c;  
        int d;  
        ...  
    } else {  
        int a;  
        int d;  
    }  
    for (int i = 0; ...) {...}  
    for (int i = 0; ...) {...}  
    int c;  
}
```

// erreur : ***b*** déjà déclaré dans le bloc externe

// erreur : ***a*** déjà déclaré dans le bloc externe
// ok : pas de conflit avec ***d*** du bloc précédent

// ok : pas de conflit avec ***i*** de la boucle précédente

// ok : pas de conflit avec ***c*** de la boucle précédente

Structures conditionnelles,
Boucles...

; // ; est un terminateur, pas un séparateur

Affectation :

$x = 3 * y + 1$

Appel de méthode :

```
string s = "a, b, c";  
string[] parts = s.Split(','); // invocation d'une méthode  
objet (non statique)  
s = String.Join(" + ", parts); // invocation d'une méthode  
de class (static)
```

Structure conditionnelle

```
if ('0' <= ch && ch <= '9')  
    val = ch - '0';  
else if ('A' <= ch && ch <= 'Z')  
    val = 10 + ch - 'A';  
else {  
    val = 0;  
    Console.WriteLine("invalid character {0}", ch);  
}
```

Structure conditionnelle/switch

```
switch (country) {  
    case "Germany": case "Austria": case "Switzerland":  
        language = "German";  
        break;  
    case "England": case "USA":  
        language = "English";  
        break;  
    case null:  
        Console.WriteLine("no country specified");  
        break;  
    default:  
        Console.WriteLine("don't know language of {0}", country);  
        break;  
}
```

Type d'expression switch

numeric, char, enum ou string.

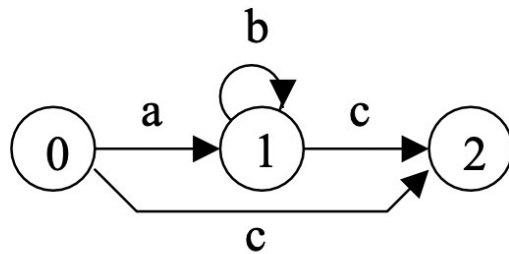
Pas de chute !

Chaque séquence d'instructions dans un cas doit se terminer par break (ou return, goto, throw).

Si aucune étiquette de cas ne correspond → par défaut

Si aucune valeur par défaut n'est spécifiée → continuation après l'instruction switch

Switch avec goto



Ex, mise en place
d'un automate

```
int state = 0;
int ch = Console.Read();
switch (state) {
    case 0: if (ch == 'a') { ch = Console.Read(); goto case 1; }
           else if (ch == 'c') goto case 2;
           else goto default;
    case 1: if (ch == 'b') { ch = Console.Read(); goto case 1; }
           else if (ch == 'c') goto case 2;
           else goto default;
    case 2: Console.WriteLine("input valid");
           break;
    default: Console.WriteLine("illegal character {0}", ch);
            break;
}
```


Boucles

while

```
while (i < n) {  
    sum += i;  
    i++;  
}
```

do while

```
do {  
    sum += a[i];  
    i--;  
} while (i > 0);
```

for

```
for (int i = 0; i < n; i++)  
    sum += i;
```

Boucles / foreach

Pour parcourir les **collections** et les **tableaux**

```
int[] a = {3, 17, 4, 8, 2, 29};  
foreach (int x in a) sum += x;
```

```
string s = "Hello";  
foreach (char ch in s) Console.WriteLine(ch);
```

```
Queue q = new Queue();  
q.Enqueue("John"); q.Enqueue("Alice"); ...  
foreach (string s in q) Console.WriteLine(s);
```

POO

POO

Programmation orientée objet (ou POO) est une technique visant à faire interagir des **objets** entre eux, permettant une meilleure modularité et une plus grande souplesse de la programmation.

Un **objet** va être constitué par l'association d'une quantité d'information organisée en champs appelés aussi **variables d'instance** ou **attributs** (nom, prénom, âge, notes pour un étudiant; marque, modèle, cylindrée, vitesse pour une voiture) et d'un **ensemble de méthodes** (plus ou moins équivalentes à des fonctions) permettant d'interagir avec lui (calculer la moyenne d'un étudiant ou accélérer pour une voiture).

POO

Cette technique de programmation a principalement **deux objectifs** :

- Faciliter l'écriture des applications par une structuration en termes d'objets.
- Favoriser la réutilisation de code, en composant des programmes à partir d'objets existants.

Deux notions essentielles sont caractéristiques de la POO :

- La notion de **classe** : schématiquement, une classe représente le type d'un objet, tout comme int représente un entier. Elle peut être vue comme une sorte de super-structure.
- La notion de **méthode** : schématiquement, une méthode est une fonction appartenant à une classe et permettant d'agir sur et avec l'objet.

POO

Les **quatre principes fondamentaux** de la programmation orientée objet sont les suivants :

Abstraction Modélisation des attributs et interactions pertinents des entités en tant que classes pour définir une représentation abstraite d'un système.

Encapsulation Masquer l'état interne et les fonctionnalités d'un objet et autoriser uniquement l'accès via un ensemble public de fonctions.

Héritage Possibilité de créer de nouvelles abstractions basées sur des abstractions existantes.

Polymorphisme Possibilité d'implémenter des propriétés ou des méthodes héritées de différentes façons sur plusieurs abstractions.

Concept de Classe

```
class C {  
... fields, constants ...    // pour POO  
... methods ...  
... constructors, destructors ...  
  
... properties ...          // pour la programmation orientée composants  
... events ...  
  
... indexers ...             // for amenity  
... overloaded operators ...  
  
... types imbriqués (classes, interfaces, structs, enums, delegates) ... }
```

Classe

```
class Stack {  
    int[] values;  
    int top = 0;  
  
    public Stack(int size) { ... }  
  
    public void Push(int x) {...}  
    public int Pop() {...}  
}
```

- Les objets sont alloués dans le tas (les classes sont des types référence)
- Les objets doivent être créés avec de *new*
Stack s = new stack (100);
- Les classes peuvent hériter d'*une* autre classe (héritage de code unique)
- Les classes peuvent implémenter plusieurs interfaces (héritage de type multiple)

Visibilité

(Quatre) niveaux de visibilité :

private : accessible seulement à l'intérieur de la classe (par défaut)

public : accessible par tout (à l'intérieur et à l'extérieur de la classe)

protected : accessible seulement à l'intérieur de la classe où à partir d'une classe dérivée

internal : accessible uniquement à l'assembly actuel.

```
public class Stack {  
    private int[] val;  
    private int top;  
    public Stack() {...}  
    public void Push(int x) {...}  
    public int Pop() {...}  
}
```

Champs et constantes

classe C {

int value = 0 ; **Champ/Field**

- L'initialisation est facultative
- L'initialisation ne doit pas accéder à d'autres Champs ou méthodes du mêmes types

Les champs
représentent les
«variables»
traduisant l'état de
l'objet

const long size = ((long)int.MaxValue + 1) / 4;

Constant

- La valeur doit être calculable au moment de la compilation

readonly DateTime date ;

Champ en lecture seule

- Doit être initialisé dans leur déclaration ou dans un constructeur
- La valeur n'a pas besoin d'être calculable au moment de la compilation
- Consomme un emplacement mémoire (comme un champ)

}

Accès dans la classe C

... value ... size ... date ...

Accès depuis les autres classes

C c = new C();
... c.value ... c.size ... c.date ...

Méthodes/exemples

```
class C {  
    int sum = 0, n = 0;
```

```
    public void Add (int x) {           // procedure  
        sum = sum + x; n++;  
    }
```

```
    public float Mean() {              // function (must return a value)  
        return (float)sum / n;  
    }
```

```
}
```

Accès dans la classe C

```
this.Add(3);  
float x = Mean();
```

Accès depuis les autres classes

```
C c = new C(); c.Add(3);  
float x = c.Mean();
```

Méthodes statiques

Mot clef **static** qui indique que ce champ est un champ de classe

```
class Rectangle {  
    static Color defaultColor;  
  
    public static void ResetColor() {  
        defaultColor = Color.white;  
    }  
}
```

Accès dans la classe C
ResetColor();

Accès depuis les autres classes
Rectangle.ResetColor();

Paramètres

Value Parameters (input values)

```
void Inc(int x) {x = x + 1;}  
void f() {  
    int val = 3;  
    Inc(val); // val == 3  
}
```

ref Parameters (transition values)

```
void Inc(ref int x) { x = x + 1; }  
void f() {  
    int val = 3;  
    Inc(ref val); // val == 4  
}
```

out Parameters (output values)

```
void Read (out int first, out int next) {  
    first = Console.Read(); next = Console.Read();  
}  
void f() {  
    int first, next;  
    Read(out first, out next);  
}
```

- "appel par valeur"

- le paramètre formel est une copie du paramètre réel ;
- le paramètre réel est une expression.

- "appel par référence"

- le paramètre formel est un alias du paramètre réel (l'adresse du paramètre réel est transmise) ;
- le paramètre réel doit être une variable.

similaire aux paramètres de référence mais aucune valeur n'est transmise par l'appelant ;

- ne doit pas être utilisé dans la méthode avant qu'il a une valeur

Surcharge de méthode

Les méthodes d'une classe peuvent avoir le même nom

- s'ils ont des nombres de paramètres différents, ou
- s'ils ont des types de paramètres différents, ou
- s'ils ont des types de paramètres différents (valeur, ref/out)

Exemple :

```
void F (int x) {...}  
void F (char x) {...}  
void F (int x, long y) {...}  
void F (long x, int y) {...}  
void F (ref int x) {...}
```

Appels :

```
int i; long n; short s;  
F(i);      // F(int x)  
F('a');    // F(char x)  
F(i, n);   // F(int x, long y)  
F(n, s);   // F(long x, int y);  
F(i, s);   // ne peut pas distinguer F(int x,  
long y) et F(long x, int y); => erreur de  
compilation  
F(i, i);   // ne peut pas distinguer F(int x, long  
y) et F(long x, int y); => erreur de  
compilation
```

Les méthodes surchargées ne doivent pas être différenciées uniquement par leurs types de fonctions !

Constructeur

Un **constructeur** est une fonction appelée au moment de la création d'un objet à partir d'une classe. Il permet d'initialiser correctement cet objet, éventuellement à partir de paramètres. Plusieurs constructeurs peuvent être définis.

Exemple :

```
public class Personne {  
    //Champs  
    private string nom;  
    private string prenom;  
    private int age;  
    private int telephone;  
    //Constructeurs  
    public Personne() {} //Constructeur par défaut qui existe si aucun autre n'est défini mais doit  
                        // être redéfini dans l'autre cas  
    public Personne (string nom, string prenom, int age, int telephone)  
    {  
        this.nom = nom;  
        this.prenom = prenom;  
        this.age = age;  
        this.telephone = telephone;  
    }  
}
```

Le mot clef **this** est une référence sur l'instance courante. Elle permet d'accéder à l'objet appelant depuis la méthode et aussi d'éviter les confusions lorsque deux variables ont le même nom.

Constructeur statique

Un **constructeur statique** est utilisé pour initialiser toutes les données statiques ou pour exécuter une action particulière qui ne doit être exécutée qu'une seule fois. Il est automatiquement appelé avant la création de la première instance ou le référencement d'un membre statique.

```
class Rectangle {  
    ...  
    static Rectangle() {  
        Console.WriteLine("Rectangle initialized");  
    }  
}
```

- Un constructeur statique ne prend pas de modificateur d'accès ou a des paramètres.
- Les constructeurs statiques ne peuvent pas être hérités ou surchargés.
- Un constructeur statique ne peut pas être appelé directement et est uniquement destiné à être appelé par le Common Language Runtime (CLR). Il est appelé automatiquement.
- L'utilisateur n'a aucun contrôle sur le moment d'exécution du constructeur statique dans le programme.
- Un constructeur statique est appelé automatiquement.

Finaliseurs

```
class Test {  
  
    ~Test() {  
        ... finalization work ...  
        // automatically calls the destructor of the base class  
    }  
  
}
```

- Les finaliseurs ne peuvent pas être définis dans des structs. Ils sont utilisés uniquement avec les classes.
- Une classe ne peut avoir qu'un seul finaliseur.
- Les finaliseurs ne peuvent pas être hérités ou surchargés.
- Les finaliseurs ne peuvent pas être appelés. Ils sont appelés automatiquement.
- Un finaliseur ne prend pas de modificateur et n'a pas de paramètre.

Propriétés

Une **propriété** permet de définir des accès en consultation (get) et en modification (set). Un accès en consultation ou en modification est possible uniquement si le get ou set correspondant est défini.

La **syntaxe** d'une propriété se définit de la manière suivante :

```
public int Age {  
  get { return this.age; }  
  set {this.age = value;}  
}
```

Le mot clef **public** indique bien que la propriété peut être utilisée dans n'importe quelle partie du code et que le mot clef **int** indique bien que **get** renvoie une valeur de type **int** et que **set** attend une valeur de type **int**.