Environnement de programmation C#

Objectif: Maitriser les concepts objets

Plan

- **Introduction**
- Types et opérations de base
- Les instructions de base
- Types valeur et types référence
- Gestion des classes et des objets
- Composition, Héritage, polymorphisme
- Abstraction, Interfaces
- Les classes du Framework.NET
- Mécanismes utilisés en C#

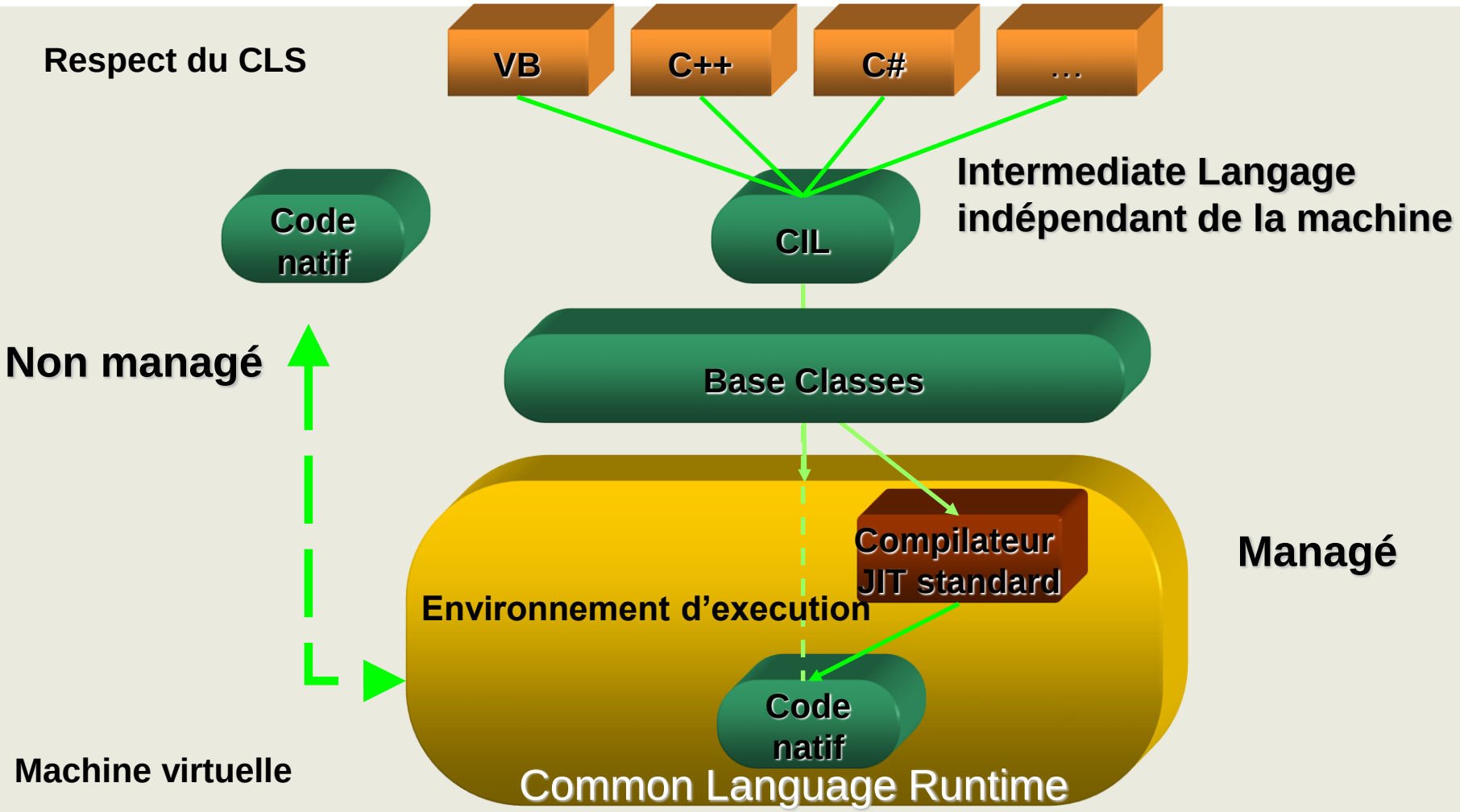
Introduction

- **Puissance du C++**
- **Facilité de développement du Visual Basic**
- **Élégance de Java**
- **Programmation par composants**
- **Déploiement des applications est plus rapide.**

Introduction

- **Langage fortement typé,**
- **Utilisé pour développer des applications web, applications de bureau, des services web, des bibliothèques de classes ……**
- **Utilise le framework .NET**

Introduction



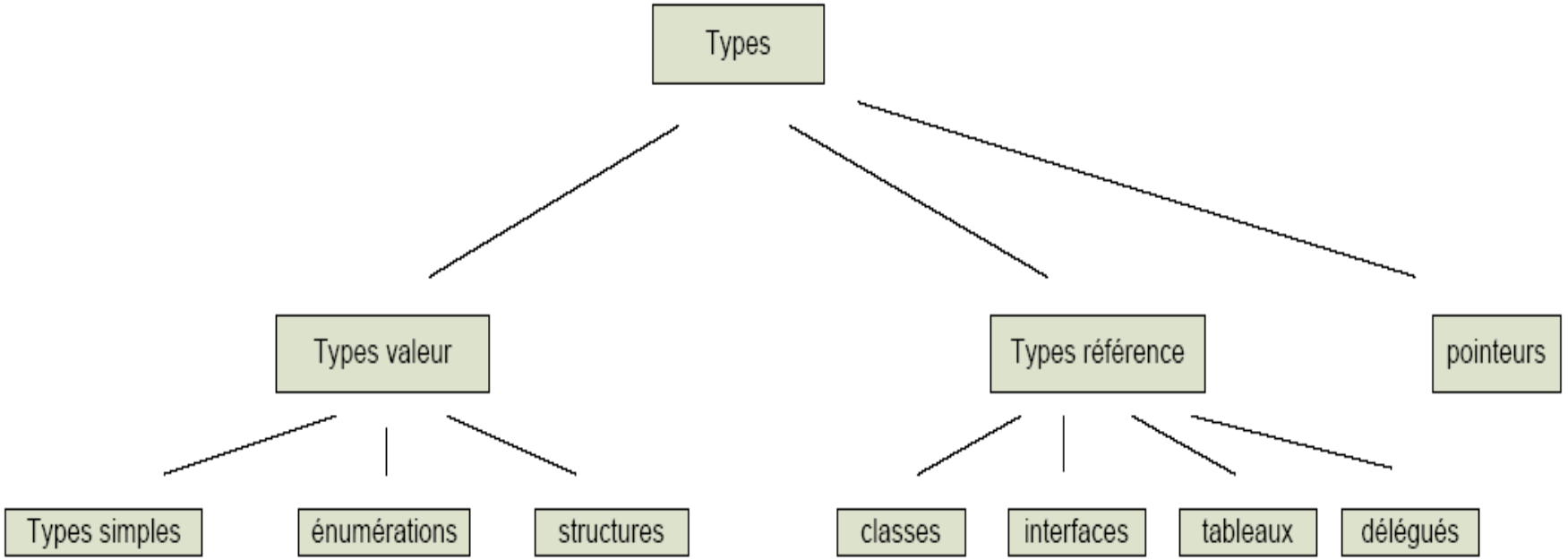
Introduction

- **Utilisation Visual studio**
  - Application console
  - Une solution est crée pour contenir les projets
  - L'objet **Solution** est une collection de tous les projets de l'instance active de l'environnement de développement intégré (IDE).

Plan

- Introduction
- **Types et opérations de base**
- Les instructions de base
- Types valeur et types référence
- Gestion des classes et des objets
- Composition, Héritage, polymorphisme
- Abstraction, Interfaces
- Les classes du Framework.NET
- Mécanismes utilisés en C#

Classification des types en C#



byte sbyte float
short ushort double
int uint decimal
long ulong

Classification des types en C#

- **Types Valeur**
  - Les variables de types valeur sont stockées dans la pile
- **Types référence**
  - Les variables de types référence sont stockées sur le tas et possèdent une référence dans la pile

Types simples

- **Booléen : bool (true ou false)**

- **Entiers**

Type	Size (in bits)	Range
sbyte	8	-128 to 127
byte	8	0 to 255
short	16	-32768 to 32767
ushort	16	0 to 65535
int	32	-2147483648 to 2147483647
uint	32	0 to 4294967295
long	64	-9223372036854775808 to 9223372036854775807
ulong	64	0 to 18446744073709551615
char	16	0 to 65535

Types simples (suite)

- **Réels**

Type	Size (in bits)	Precision	Range
float	32	7 digits	1.5×10^{-45} to 3.4×10^{38}
double	64	15-16 digits	5.0×10^{-324} to 1.7×10^{308}
decimal	128	28-29 decimal places	1.0×10^{-28} to 7.9×10^{28}

Opérateurs et priorité

Operateurs

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

Enumérations

- **Une énumération est un type entier défini par l'utilisateur.**
  - maintenance du code plus aisée avec la sécurité d'affecter à une variable uniquement des valeurs légitimes
  - code plus clair par l'utilisation de noms descriptifs

```
enum Couleur
 { // valeurs: 0, 1, 2
       rouge,
       bleu,
       vert
}
```

```
enum Acces
 { // valeurs explicites
   personnel = 1,
   groupe = 2,
   tous = 4
}
```

```
Couleur c = Couleur.bleu;
Acces a = Acces.personnel;
if (Acces.personnel == a)
            Console.WriteLine ("Accès accordé");
```

Tableaux

- **Séquence d'éléments de même type**

```
int [ ] tab_Ent;  // ne se limitent pas aux types primitifs

tab_Ent = new int[3];

int [ ]tab2_Ent;

tab2_Ent = new int [ ] {5,6,9};

float [ ] tab_Reel= {1.2f, 1.8f, 2.6f} ; //génération automatique du code

string [  ] tab_Ch={"Tintin", "Milou", "Haddock"};
```

Check des bornes par le runtime!
IndexOfOutRangeException

Tableaux

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

mat[1,2]= 5;

int [,] mat1= {{1,2,3}, {4,5,6}};


int [ ][ ] matDéchiqueté;

matDéchiqueté= new int [2][ ];

matDéchiqueté[0]= new int[3];
```

Chaînes de caractères

```
string s = "Bonjour";          // Bonjour

string s = "\"Bonjour\"";      // "Bonjour"

string s = "Bon\njour";        // saut de ligne ajouté

string s = @"Bonjour\n";   // Bonjour\n
```

- **Remarques**
  - nombre de caractères d'une chaîne est illimité.
  - @ devant une chaîne de caractères permet de définir un "verbatim string",

```
string document = @"c:\nouveau\exemple.doc";
string s = @""Salut""; // produit la chaîne "Salut"
```

Structure

- **Une structure est une entité qui regroupe des champs dont le type peut être différent.**

- **Déclaration de structure**

```
struct Point
{
        public int x, y; // champs
        public Point (int x, int y)
        {
                this.x = x;
                this.y = y;
        }
        public void moveTo (int a, int b)
        {
                x = a;
                y = b;
        }
}
```

```
Point p = new Point(3, 4);
p.moveTo(10, 20); // appel de méthode
```

Classes

- **Déclaration d'une classe**

```
class Point
 {
   public int _x, _y; // champs
   public Point (int x, int y)
   {
     _x = x;
     _y = y;
   }
   public void moveTo (int a, int b)
    {
        _x = a;
         _y = b;
   }
}
```

```
Point p = new Point(3, 4);
p.moveTo(10, 20); // appel de méthode
```

Différences entre structures et classes

Structures	Classes
Type valeur (objets stockés sur la pile)	Type référence (objets stockés sur le tas)
Pas de support de l'héritage (mais compatible avec **object**)	Support de l'héritage (toutes les classes dérivent de **object**)
Destructeurs non autorisés	Peuvent avoir un destructeur

La structure est idéal pour les données « petites », elle est plus efficace qu'une classe dans certains scénari

Constantes

- **Les constantes doivent être déclarées à l'aide du mot réservé const, suivi d'un type.**

- **Une valeur doit leur être affectée au moment de la déclaration:**

```
const int NOTE_MAX = 6;
const long VITESSE_LUMIERE= 300000; // km/s
const double RACINE_DEUX = 1.414;
```

Plan

- Introduction
- Types et opérations de base
- **Les instructions de base**
- Types valeur et types référence
- Gestion des classes et des objets
- Composition, Héritage, polymorphisme
- Abstraction, Interfaces
- Les classes du Framework.NET
- Mécanismes utilisés en C#

if

- **Une instruction conditionnelle permet de contrôler le flux d'un programme en fonction de la valeur d'une expression booléenne.**
  - **if ... else**

```
if (ventes > 2000)
{
    taxes = 0.072 * ventes;
}

if (ventes > 2000)
{
    taxes = 0.072 * ventes; // exécuté si la condition est vraie
}
else
{
    taxes = 0; // exécuté si la condition est fausse
}
```

if

```
if (ventes > 2000)
    taxes = 0.072 * ventes;
else
    if (ventes > 1000)
        taxes = 0.04 * ventes;
    else
        taxes = 0;
```

- **Une alternative parfois intéressante, l'opérateur ternaire ?:**

```
taxes = (ventes > 2000) ? (ventes * 0.072) : 0;

taxes = (ventes > 2000) ? (ventes * 0.072) : (ventes >1000)? (ventes * 0.04):0;
```

switch

```
switch (mois)
{
   case 1 :
   case 3 :
   case 5 :
   case 7 :
   case 8 :
   case 10 :
   case 12 : jours = 31;
   break;                    //obligatoire
   case 2 : jours = 28;
   break;
   default : jours = 30;
 break;
}
```

for

- **Instruction for**
  - Une boucle for est utilisée dans les situations où **l'on connaît à l'avance le nombre d'itérations à effectuer**.

```
for (initialisation; condition; incrémentation)
{
   instructions;
}
```

  - La condition est évaluée et, tant qu'elle est vraie, le bloc d'instructions constituant le corps de la boucle est exécuté.
  - La condition est évaluée **avant** l'exécution des instructions.

for

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


for (int k = 200; k > 0; k -= 10)
{
   Console.WriteLine ("k = {0}", k);
}


for (int i = 0, j = 100; i > 100; i++, j--)
 {
   Console.WriteLine ("{0}, {1}", i, j);
}
```

```
0, 100
1, 99
2, 98
...
99, 1
```

while

- **Instruction while**
  - Le test est effectué avant d'exécuter les instructions de la boucle.
  - La boucle while est souvent utilisée lorsque **l'on ne connaît pas à l'avance le nombre d'itérations à effectuer.**

```
while (condition)
{
  instructions;
}
```

do

- **Instruction do**
  - Les instructions du corps de la boucle sont d'abord exécutées,
  - la condition de continuation de la boucle est évaluée à la fin des instructions.

```
do
{
   instructions;
} while (condition);
```

Break .. continue

- **break et continue**
  - Pour les trois types de boucles il est possible d'utiliser les instructions break et continue.
  - continue permet de remonter directement à la prochaine itération.
  - break permet de sortir directement de la boucle.

foreach

- **Instruction foreach**
  - Les boucles de types foreach sont adaptées pour parcourir les éléments d'une collection

foreach (type variable in nom_tab)

```
string[] names = {"Cheryl", "Joe", "Matt", "Robert"};
foreach (string person in names)
{
        Console.WriteLine("{0} ", person);
}
```

Lire-Ecrire

```
namespace ConsoleApplication1
{
    class Program
    {
        public static void Main()
        {
            string s1 = "Bonjour";
            string s2 = "Tout le monde";
            Console.WriteLine("chaine 1 {0} chaine 2 {1}", s1,s2);
        }
    }
}
```

Lire-Ecrire

```
namespace ConsoleApplication1
{
    class Program
    {
        public static void Main()
        {
            float x;
            String saisie;
            saisie = Console.ReadLine();
            x = float.Parse(saisie);
            Console.WriteLine("Vous avez saisi :{0} " ,x);
        }
    }
}
```

Plan

- Introduction
- Types et opérations de base
- Les instructions de base
- **Types valeur et types référence**
- Gestion des classes et des objets
- Composition, Héritage, polymorphisme
- Abstraction, Interfaces
- Les classes du Framework.NET
- Mécanismes utilisés en C#

Pile et tas

- **Les variables et les objets peuvent être physiquement stockés dans deux types de zone mémoire:**
  - La pile (stack) d'un thread.
  - Le tas d'un processus.

Pile et tas

- **Avantages de la pile/désavantage du tas:**
  - Accès optimisés (directement au niveau des instructions machines).
  - Pas de problème d'accès concurrents à une pile (puisqu'il y a une pile par thread).
- **Avantages du tas/désavantage de la pile:**
  - La pile est limité en taille (de l'ordre du Mo).
  - La pile est utilisé pour stocker des petits objets accédés souvent (int, double, bool…)
  - Le tas est utilisé pour stocker des gros objets globaux au processus.

Pile et tas

- **L'endroit où est stocké un objet (pile ou tas) dépend de son type:**
  - Les instances des types valeurs sont par défaut stockées sur la pile.
  - Les instances des types références sont TOUJOURS stockées sur le tas. Chacune de ces instances admet une référence dans une pile.

Types valeur et types référence

- **Les types valeurs sont:**
  - Les types primitifs (int, bool, double…).
  - Les énumérations.
  - Les structures.

- **Les types références sont les classes (standard, propriétaire, string, tableau…).**

Accès, Affectation

- **Les instances des types valeurs sont toujours accédées directement.**
- **Les instances des types références sont toujours accédées par l'intermédiaire d'une référence.**
- **L'affectation entre instances de types valeur se traduit par une copie du contenu.**
- **L'affectation entre instances de types référence se traduit par une copie de la référence.**

Comparaison

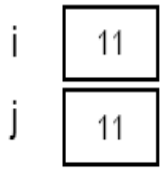
- **Les instances des types valeurs se comparent sur leurs contenus (égalité).**
- **Les instances des types références se comparent sur leurs identités.**

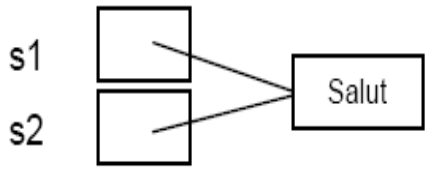
Remarques

- **Tous les types sont compatibles avec le type object,**
- **Ils peuvent être assignés à des variables de type object,**
- **Toutes les opérations disponibles pour le type object leur sont également applicables**

En Résumé

	Type valeur	Type référence
Une variable contient	une valeur	une référence
Stockée dans	le stack (*pile*)	le heap (*tas*)
Initialisation	0, false, '\0'	null
Assignation	copie de la valeur	copie de la référence
Exemple	int i = 11; int j = i;	string s1 = "Salut" string s2 = s1;





pile **pile** **tas**

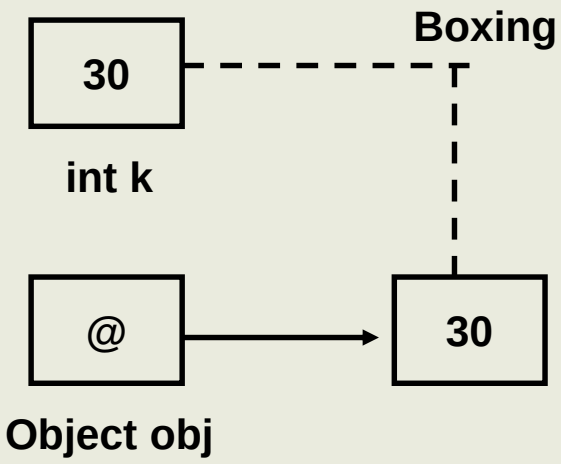
Boxing et unboxing

- **Tous les types, même les types simples prédéfinis, sont compatibles avec le type Object.**

```
string s = 12.ToString();
```

- **Le boxing peut être vu comme un transtypage d'un type valeur à un type référence.**

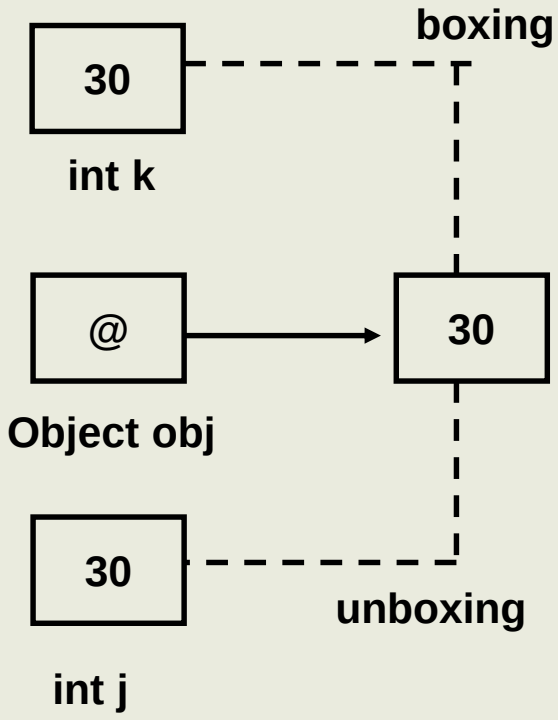
```
int k = 30;
Object obj = k; // boxing
```



Boxing et unboxing

- **Le unboxing concerne l'opération inverse dans laquelle un type référence est transtypé en type valeur.**

```
int k = 30;
Object obj = k; // boxing d'un int en objet
int j = (int)obj; // unboxing pour revenir à un int
```



Paramètres

- **Passage des paramètres par valeur, par référence**
  - Les instances des types valeurs sont par défaut passées par valeur, mais peuvent être passées par référence.
    - void f(int x) --> int a=2; f(a); //initialisation obligatoire
    - void g(ref int x) -> int a=2; g(ref a);
  - Les instances des types références sont toujours passées par référence
    - void f(int [ ] tx) ->int [ ] ta = {1,2,3}; f(ta);
  - En retour : out
    - void f(out int x) ->int a ; f(out a) //inutile d'initialiser