70 → COBOL → Gestion

→ Assembleur → perf.

→ Basic

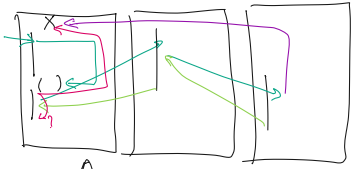
---

80 → C → portable

Compilateur → Rapide

Puissant

code proche assembleur



fichier de code (modules) → fonctions

Procédures

var. locales

globales

1G 1G → assembleur

2G → Prog naturelle

Spaghetti

(Basic, Scripts)

3G → Prog structurée

→ Pascal, C, VBasic

4G → Prog. orientée objet

= 3G + concepts objet

- mauvaises pratiques

BDD → Méthode Merise

UML → Méthode Notation } Unified Modeling Language

12 types de diagrammes #

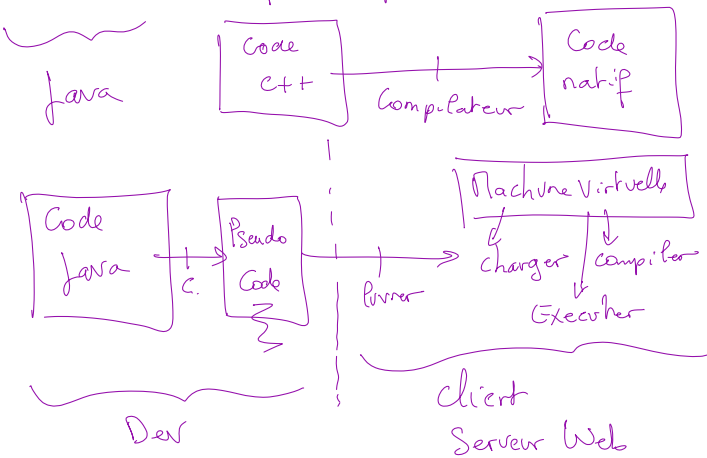
→ 1 → Diag statique des classes.

C++ ← C (3G) + objet (full)

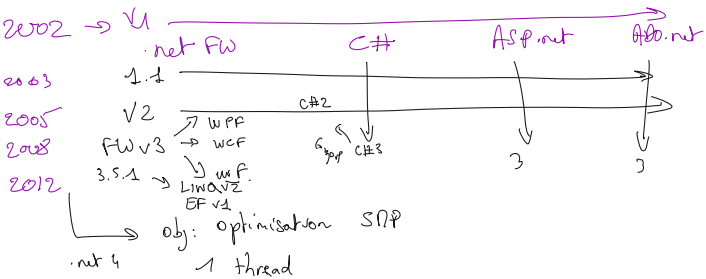
C#

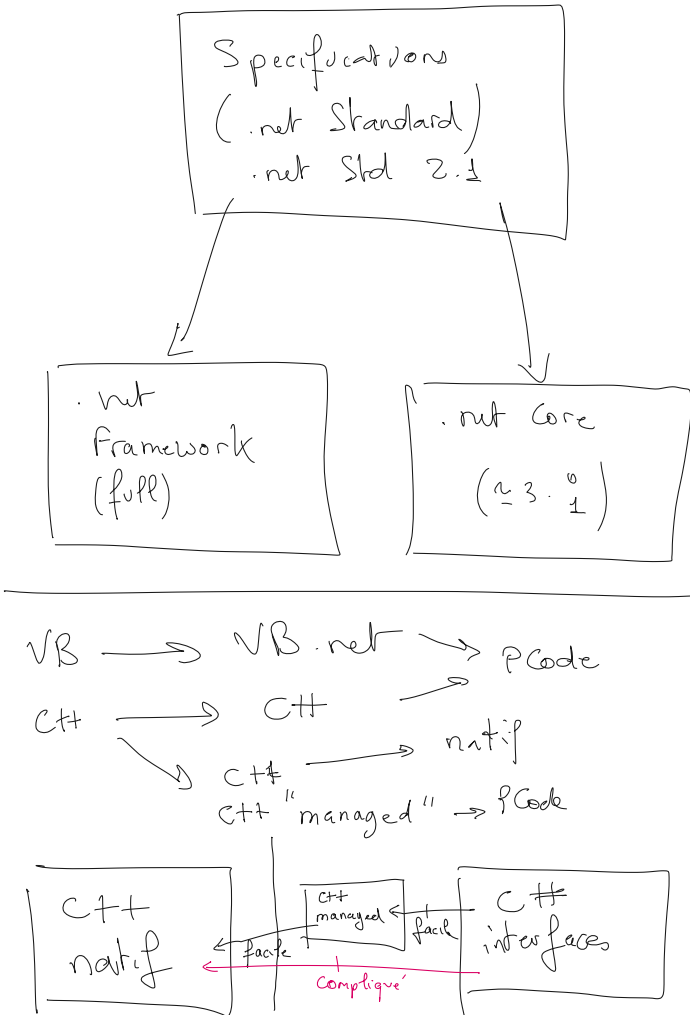
80 | C
| ↓ Pascal
| C++
90 | VBA, VB3, VB6 → RAD → Productif → ≥ 92-94 → Php → Java
| C++ → PRO) Plus → - bugs - Productivité
| ↓
| 98 → VB6
| C++ MFC

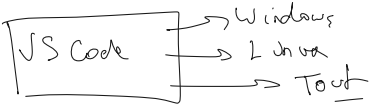
C++ → Portabilité Compliquée
→ Bcp de bugs (Mémoire)
→ Trop complexe

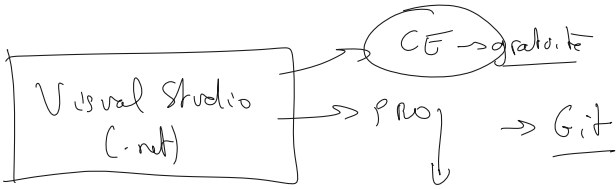


Recap Versions




Specifications
(.net Standard)
.net Std 2.1
.net Framework (full)
.net Core (≥ 3.0 / 1)
VB → VB.net → PCode
C++ → C++ → PCode
C++ → natif
C++ "managed" → PCode
C++ natif
C++ managed
C++ interfaces
Facile
Facile
Compliqué





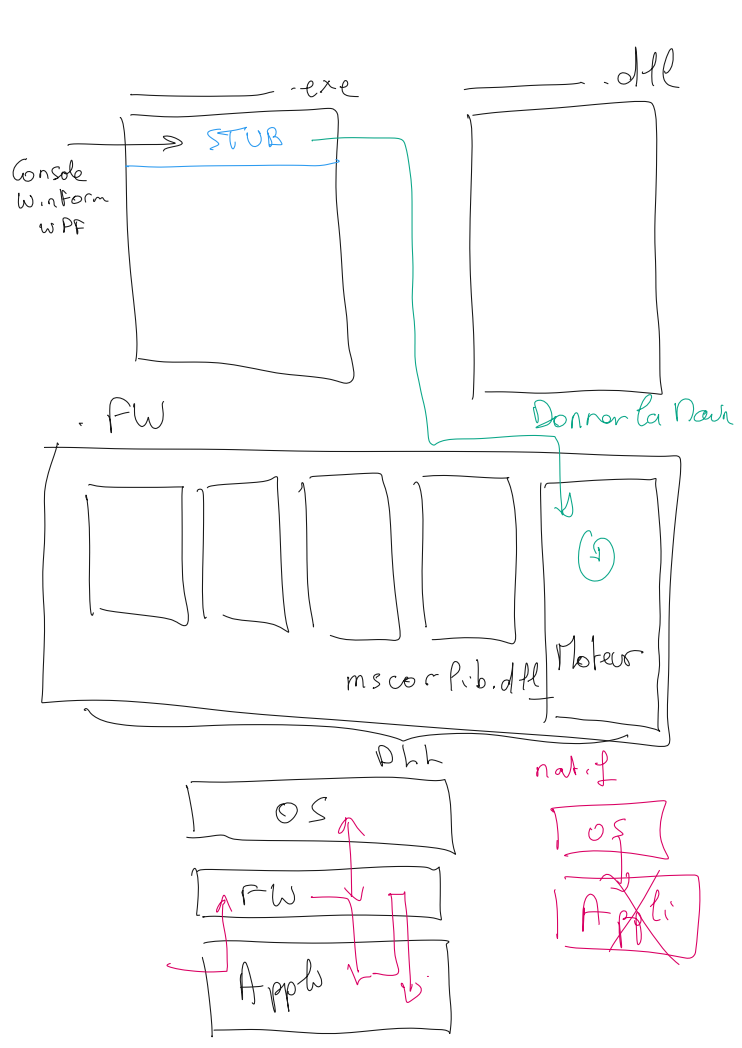
→ Console → C#

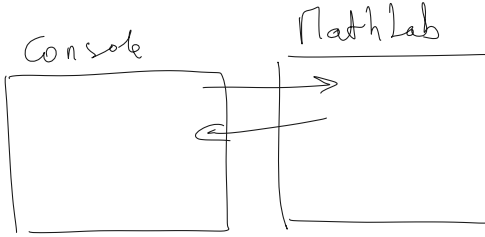
.net FW

base (SG) → "acquis"

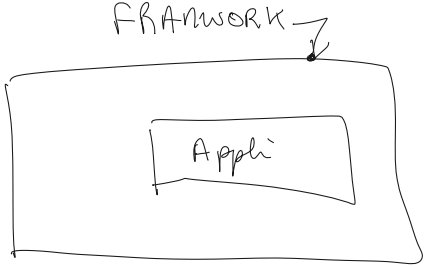
→ Concepts objet → C#

Objectif → C#, objet (± 80%)


.exe
.dll
STUB
Console
Winform
WPF
.FW
Donner la Main
mscorlib.dll
Moteur
PAL
natif
OS
FW
Appli
OS
Appli



Bibliothèque : échanges "simples"



Appli développée autour du cadre
→ assez rigide

Lib :

évolution, changement
→ Simple

FW :

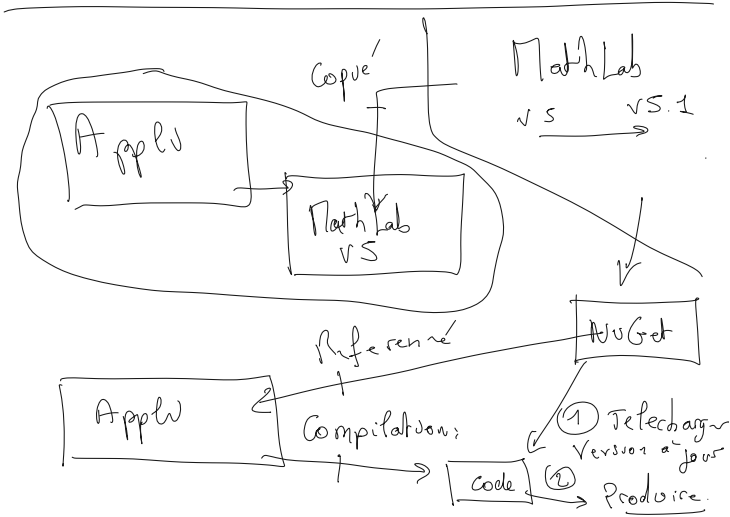
Changement
→ Réécriture de l'appli

Outil → C#

s'informer !!

"Bonnes pratiques" → Design Patterns

Framework (implémentation industrielle d'un pattern)



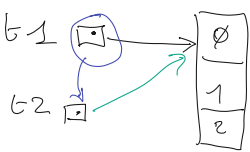
Classe	PascalCasing
Methode	PascalCasing
Donnée membre	PascalCasing
Paramètres	camelCasing
Namespace / Package	PascalCasing
Variable	lowercasing

Notation :

int i ⇒ i contient directement une valeur

Type simple

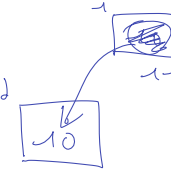
Pour les objets (complexe), la variable référence la valeur ⊡ → Référence ("pointe vers")



int i = 10

int j = i;

i++



t2 = t1

↓

Pas une copie du tableau,

mais une copie de la référence

Types Simples

int
bool
float
enum

Fonctionnent par copie

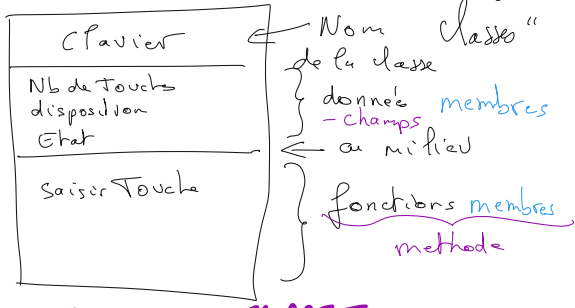
Types Complexes

class → objets

Fonctionnent par Référence

Tout peut être vu comme objet

Notation UML : (12 diagrammes) "diagramme de classe"

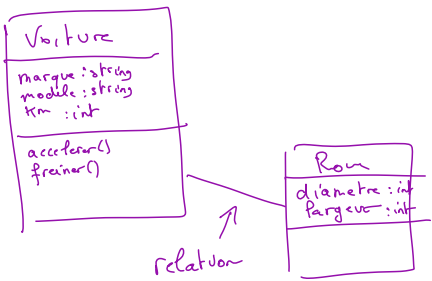


Type → Concept → **CLASSE**

Elément utilisable, concret → **INSTANCE**

Instance → pour un cycle de vie (objets)
(création, utilisation, destruction)

"Tout élément complexe peut être éclaté en plusieurs éléments plus simple" en final, tout sera simple (dans le type)



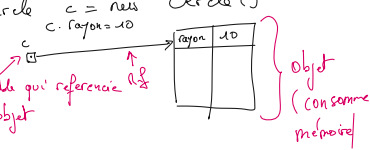
Type simple

```
int i = 10;
```

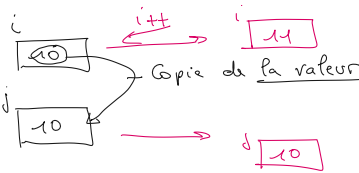
i

Type complexe

```
Cercle c = new Cercle();
c.rayon = 10;
```

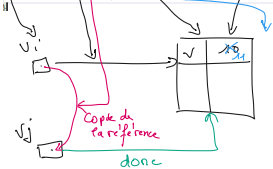


```
int i = 10;
int j = i;
i++;
Console.WriteLine("j vaut " + j);   // 10
```



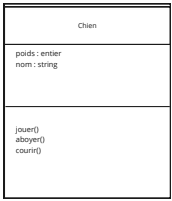

```
3 références
class Valeur
{
    public int v;
}

// la même chose avec un type complexe :
Valeur vi = new Valeur() { v = 10 };  // instanciation-affectation " j'instancie et au passage j'affecte des valeurs"
Valeur vj = vi; // a l'image de j=i
vi.v++; // a l'image de i++
Console.WriteLine("vj vaut " + vj.v );   // ?
```

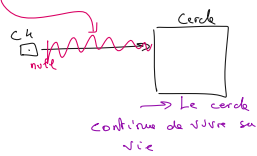


Exercice : Quelles sont les membres d'un type vivant (prendre un animal) ex : un chien

- nom de classe, de données membres, de méthodes, et constructeurs

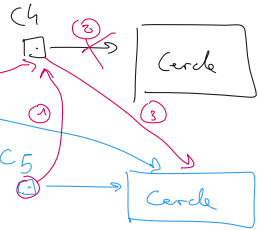



```
// quand le cercle ( l'objet ) sera t-il détruit ?
Cercle c4 = new Cercle();
Console.WriteLine(c4.Perimetre());
c4 = null;
Console.WriteLine(c4.Perimetre());
```

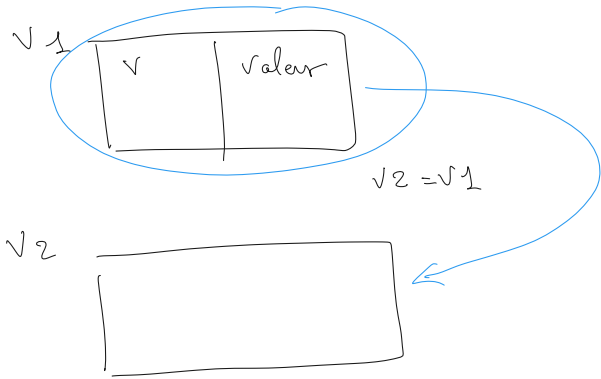


```
// quand le cercle ( l'objet ) sera t-il détruit ?
Cercle c4 = new Cercle();
Console.WriteLine(c4.Perimetre());
c4 = null;
Console.WriteLine(c4.Perimetre());

Cercle c5 = new Cercle();
c4 = c5;
```

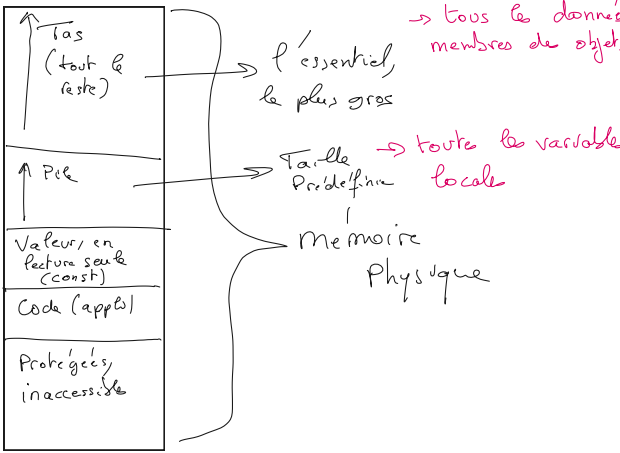


Avoir un objet qui devient "non référençable" parceque ses références (variables) ne pointent plus dessus
Devient un objet marqué pour être recyclé vis à vis de la mémoire (dans les liste des objets non référençables à détruire)

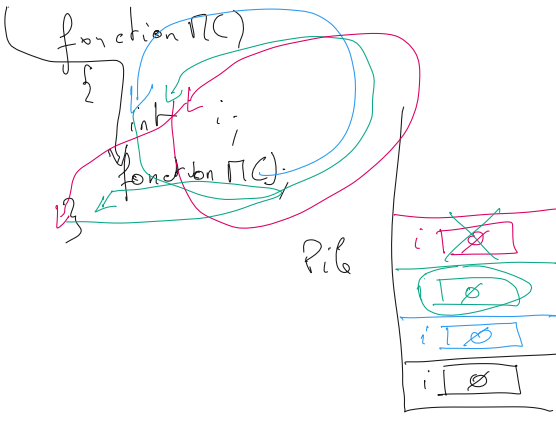



```
ValStructCopie v1;
v1.v = 10;
ValStructCopie v2;
v2 = v1;
v1.v++;
Console.WriteLine(v1.v);
Console.WriteLine(v2.v);
```

64 go

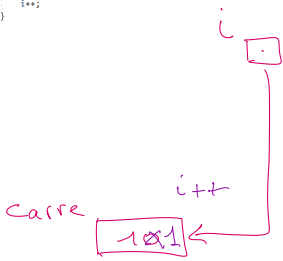

Tas
(tout le reste)
Pile
Valeurs en lecture seule (const)
Code (appli)
Protégées, inaccessible
l'essentiel, le plus gros
Taille Predefinie
mémoire
Physique
→ tous les données membres des objets
→ toutes les variable locale

0


fonction f()
int i;
fonction f();
Pile
i
i
i

```
Incremente(ref carre);
Console.WriteLine("Carre local = incrémenté " + carre); // il est modifié
```

```
1 référence
static void Incremente( ref int i ) // va aller modifier ce qui est référencé par i
{
    i++;
}
```



Création de classes et objets :

1)
Créer une classe voiture, qui puisse accélérer, et freiner.
Cette voiture a des roues (soit un tableau de 4 roues, ou une roue avant gauche, av droite, etc...

Coder ce concept

2)
Créer une classe Cercle, Carre, Rectangle, Triangle, qui ont leurs caractéristiques, et leurs fonctions (périmètre, surface, ... diametre, .diagonale, ..)

Tester bien sur ces Types, en utilisant les constructeurs paramétrés autant que possible.

```
class Cercle
   {
      public int rayon;

      // le constructeur est une méthode particulière qui porte le nom de la classe
      // et qui est appelé à l'instanciation de classe
      public Cercle() // "constructeur par défaut" ( sans paramètres )
      {
         Console.WriteLine("Création d'un cercle");
      }

      // Le polymorphisme est un concept qui autorise n méthodes à porter le même nom,
pourvu que leur signature soit
      // différente
      // Def de la signature : "la concaténation" du nom de la méthode, et de ses types
      // ex : Cercle ( signature 1 )
      //      Cercleint ( signature 2, différente )
      //      Cercleintint ( signature 3, différente )
      public Cercle(int v) // constructeur paramétré
      {
         rayon = v;
         Console.WriteLine("Création d'un cercle de rayon " + v);
      }
      public Cercle(int k, int perimetre ) // constructeur paramétré
      {
         rayon = k;
         Console.WriteLine("Création d'un cercle de rayon " + k);
      }

      public double Perimetre()
      {
         return 2 * Math.PI * rayon;
      }
   }
   partial class Valeur
   {
      public int v;
   }

   struct ValStructCopie
   {
      public int v;
   }
```

```
class Program
    {
        public static void recurs()
        {
            int i=new Random().Next();
            recurs();
        }

        public static int ModifierI( int i )    // ne modifie pas I
        {
            i++;
            return i;
        }

        class ResultatDeCalcul  // une définition de classe dans une classe s'appelle une "sous
classe" ou une classe imbriquée
            // utile quand cette classe n'est utilisée qu'au sein de notre classe de plus haut niveau
        {
            public int carre;
            public double divise;
        }

        static ResultatDeCalcul Calculer( int i )
        {
            var res = new ResultatDeCalcul();
            res.carre = i * i;
            res.divise = i / 2;

            return res;
        }

        static void Calculer(int i, out int carre, out double divise )
        {
            carre = i * i;
            divise = (double)i / 2;
        }

        static void Incremente( ref int i ) // va aller modifier ce qui est référencé par i
        {
            i++;
        }

        static void CopieParamAppel()
        {
            int i = 10;

            int k = ModifierI(i);   // je passe une copie de i sur la pile dans la fonction ModifierI
            Console.WriteLine("I de ModifierI vaut " + i);  // combien vaut i ? -> 10
            Console.WriteLine("k de ModifierI vaut " + k);  // combien vaut k ? -> 11

            // Par contre :
            // Je ne peux renvoyer qu'une valeur
            // Si je souhaite renvoyer plusieurs valeurs :
            // Solution technique "comme avant" : un tableau, une liste, ...

            // solution 1 :
            // Encapsuler nos valeurs dans un type complexe ( la bonne pratique )
            // Solution 2 :
            // Faire des paramètres modifiables ( important à connaître )

            // Implémentation de la solution 1 : je souhaite obtenir le carré, et la division par 2 de
ma valeur
            //
            // BONNE PRATIQUE
            //
            var res = Calculer(5);
            Console.WriteLine("Res = " + res.carre);

            // Implémentation de la solution 2 ( pour la syntaxe )
            int carre;
            double divise;
            Calculer(5, out carre, out divise); // out : je ne fais que recevoir les valeurs
                                                // ref : je passe une valeur qui peut être écrasée
            Console.WriteLine("Carre local = " + carre);
            Console.WriteLine("Divise local = " + divise.ToString());

            Incremente(ref carre);
            Console.WriteLine("Carre local = incrémenté " + carre); // il est modifié
        }
```

```
static void Main(string[] args)
      {

          CopieParamAppel();


          Cercle c = new Cercle(5);
          Cercle c2 = new Cercle();
          // c est une instance d'un cercle
          c.rayon = 10;
          c2.rayon = 15;
          Console.WriteLine("Le P fait " + c.Perimetre());
          Console.WriteLine("Le P fait " + c2.Perimetre());


          // tableau de cercles :
          Cercle[] cercles = new Cercle[10];
          cercles[5] = new Cercle();

          // fonctionnement des types simples par rapport aux types complexes ( par copie /
par référence )

          int i = 10;
          int j = i;
          i++;
          Console.WriteLine("j vaut " + j);   // 10

          // la même chose avec un type complexe :
          Valeur vi = new Valeur { v = 10 };  // instanciation-affectation " j'instancie et au
passage j'affecte des valeurs"
          Valeur vj = vi; // a l'image de j=i
          vi.v++; // a l'image de i++
          Console.WriteLine("vj vaut " + vj.v );   // ?

          Cercle c3 = null;
          //Console.WriteLine(c3.Perimetre());

          // quand le cercle ( l'objet ) sera t-il détruit ?
          Cercle c4 = new Cercle();
          Console.WriteLine(c4.Perimetre());
          c4 = null;
          //Console.WriteLine(c4.Perimetre());

          Cercle c5 = new Cercle();
          c4 = c5;

          ValStructCopie v1;
          v1.v = 10;
          ValStructCopie v2;
          v2 = v1;
          v1.v++;
          Console.WriteLine(v1.v);
          Console.WriteLine(v2.v);

          // Nullable :

          // int ni = null;  // pas possible
          int? ni = null;    // int nullable peut être null
          ni = 50;
          //int z = ni.Value;   // plante si ni vaut null

          if (ni.HasValue)
          {
              //int x = ni; // x peut recevoir null, ce qui n'est pas possible
              int z = ni.Value;   // cette opération peut planter ( Exception ) si ni vaut null
              Console.WriteLine("ni vaut " + z);
          }
      }
  }
}
```

Utilisation du mot clef this :

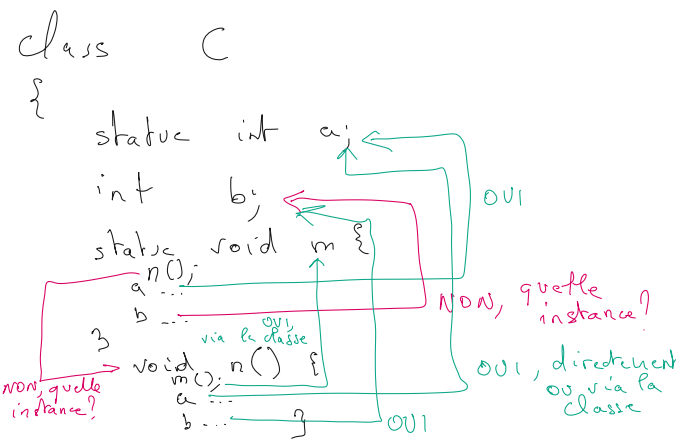
```
class Vache
{
    public string nom;
    public int poids, taille, age;

    1 référence
    public Vache( string nom )
    {
        // Vache.nom est le nom du type Vache ( on ne donne pas de nom au type, mais aux instances de Vache )
        // this veut dire " l'instance en cours "
        this.nom = nom;      // "mon nom est la valeur du nom passé en paramètre"

        // this peut être utilisé systématiquement pour explicitement utilser les données membres
        // ou pas.
        // mais obligatoire en cas d'ambiguité
        // car les variables locaux sont prévalentes sur les données membre

        this.taille = 200;
    }
}

0 références
class Program
{
    0 références
    static void Main(string[] args)
    {
        Vache v1 = new Vache("Marguerite");
        Console.WriteLine("Bonjour " + v1.nom);
        Console.WriteLine("La taille : " + v1.taille);
    }
}
```

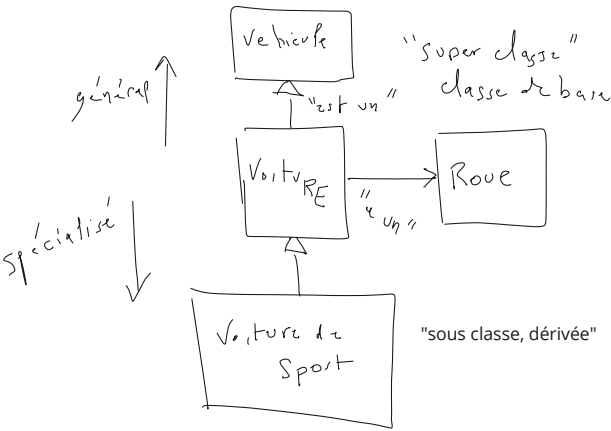


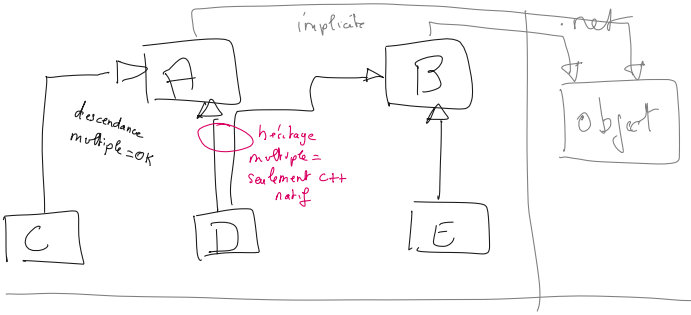
Extensions :

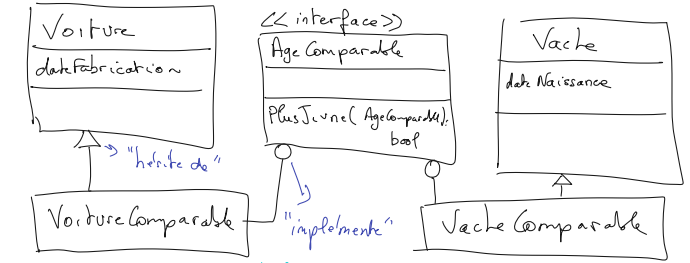
```
static class MesOutils
{
    0 références
    public static string InitCap2( string msg )
    {
        return msg.Substring(0, 1).ToUpper() + msg.Substring(1);
    }

    2 références
    public static string InitCap( this string msg )
    {
        return msg.Substring(0, 1).ToUpper() + msg.Substring(1);
    }
}
0 références
class Program
{
    0 références
    static void Main(string[] args)
    {
        string prenom = "philippe";
        Console.WriteLine("Bonjour " + MesOutils.InitCap(prenom));
        Console.WriteLine("Bonjour " + prenom.InitCap());
    }
}
```

Géneralisation







→ héritage unique, interfaçage multiple

⇒ Une interface est comme une classe purement abstraite (aucun membre concret)

```
Vo = new Voiture();
Va = new Vache();
Object[] objets = new Object[2];
objets[0] = Vo;   objets[1] = Va;
if( objets[0].PlusJeune( objets[1] ) )
```

L'utilisation du mot clef **new** (mauvaise pratique) implique que l'on appelle la méthode du type de la variable, au lieu du type de l'instance que la variable référence.