POLYTECH ANGERS

C# / .NET

3A SAGI

Polytech Angers
2019-2020

Bertrand Cottenceau - bureau 311 ISTIA
bertrand.cottenceau@univ-angers.fr

Références bibliographiques :

Introduction au langage C# de Serge Tahé

Transparents du cours de H.Mössenböck, University of Linz
http://www.ssw.uni-linz.ac.at/Teaching/Lectures/CSharp/Tutorial/

C# 4.0 in a Nutshell de Joseph et Ben Albahari (O'Reilly)

C# 3.0 Design Patterns de Judith Bishop (O'Reilly)

1. Introduction

Le langage C# (C Sharp) est un langage objet créé spécialement pour le framework Microsoft .NET (1ère version en 2002). Le *framework .NET* est un environnement d'exécution (CLR Common Language Runtime) ainsi qu'une bibliothèque de classes (plus de 2000 classes). L'environnement d'exécution (CLR) de .NET est une machine virtuelle comparable à celle de Java.

Les types .NET peuvent être utilisés par tous les langages prenant en charge l'architecture .NET (VB.net, C#). Les langages .NET doivent satisfaire certaines spécifications : utiliser les mêmes types CTS (Common Type System), les compilateurs doivent générer un même code intermédiaire appelé MSIL (Microsoft Intermediate Language).

Source C# (texte) → Code MSIL → compilé (JIT compiler) par le CLR en code natif

Le MSIL (contenu dans un fichier .exe) est tout d'abord compilé par le JIT compiler (Just In Time Compiler). La compilation en code natif n'a lieu qu'au moment de l'utilisation du programme .NET. Plusieurs compilateurs sont disponibles, dont C++.NET (version Managée de C++), VB.NET et C#.

Une des caractéristiques majeures est que le CLR .NET gère la récupération de mémoire allouée dans le tas (garbage collector), à l'instar de la machine virtuelle Java. La gestion dynamique de mémoire est alors plus simple qu'en langage C (malloc/free).

2. Les points communs avec le langage C

Le C# reprend beaucoup d'éléments de syntaxe du langage C :

- structure des instructions similaire (terminées par ;) : déclaration de variables, affectation, appels de fonctions, passage des paramètres, opérations arithmétiques
- blocs délimités par {}
- commentaires // ou /* */
- structures de contrôle identiques : if/else, while, do/while, for(; ;)
- portée des variables : limitée au bloc de la déclaration

En C#, ***on peut déclarer une variable n'importe où avant sont utilisation.***

2.1. Premier exemple console

System.Console est la classe de gestion des entrées/sorties en mode console. La fonction principale (point d'entrée) est la fonction statique **`void Main(...)`**. En mode console, les entrées sorties (saisies clavier + affichages) se font grâce à la classe **`Console`**.

```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)    // Point d'entrée
        {
            int var = 2;
            for (int i = 0; i < 5; i++)
            {
                var += i;
                Console.WriteLine("i={0} var={1}", i, var);
            }

            float f=2F;
            while (f < 1000F)
            {
                f = f * f;
                Console.WriteLine("f={0}", f);
            }
        }
    }
}
```

Sortie
```
i=0 var=2
i=1 var=3
i=2 var=5
i=3 var=8
i=4 var=12
f=4
f=16
f=256
f=65536
```

Les arguments de la ligne de commande. Lorsque l'on exécute un programme console depuis l'invite de commandes, on peut passer des paramètres qui sont reçus en arguments de la fonction statique Main().

```
E:\Enseignement\C#>Prog Toto 23 Texte
Programme appelé avec les paramètres suivants:
argument n° 0 [Toto]
argument n° 1 [23]
argument n° 2 [Texte]
```

```
static void Main(string[] args)
{
       Console.WriteLine("Programme appelé avec les paramètres suivants:");
       for (int i = 0; i < args.Length; i++)
       {
              Console.WriteLine("argument n° {0} [{1}]",i,args[i]);
       }
}
```

2.2. Types primitifs du C# (alias de types .NET)

Le framework .NET fournit les types primitifs, et ce quel que soit le langage utilisé (VB.NET, C#, C++.NET).

En C#, les types élémentaires sont nommés comme suit (entre [ ] le nom du type .Net équivalent)

`bool` : booléen (non comparable à un entier!) [System.Boolean]
`char` : caractère Unicode 16bits [System.Char]
`sbyte`/**`byte`** : entier 8 bits signé/non signé [System.Byte/System.SByte]
`short`/**`ushort`** : entier 16 bits signé/non signé [System.Int16/System.UInt16]
`int`/**`uint`** : entier 32 bits signé/non signé [System.Int32/System.UInt32]
`long`/**`ulong`** : entier 64 bits signé/non signé [System.Int64/System.UInt64]
`float` : flottant 32 bits [System.Single]
`double` : flottant 64 bits [System.Double]
`decimal` (128 bits) : entier multiplié par une puissance de 10 [System.Decimal]
`string` : chaîne de caractères. Type référence qui s'utilise comme un type valeur. [System.String]

2.3. Littérals (constantes) / Format des constantes

En C#, les constantes numériques peuvent presque être considérées comme typées. Selon la notation utilisée pour décrire une constante, le codage n'utilise pas le même nombre de bits.

Constantes entières

```
int x=10;            // x ppv 10 (base 10)
int x = 0x2F;        // x ppv (2F)h
long l= 10000L;      // le L signifie que la constante est sur 64 bits
```

Constantes réelles

Par défaut, les constantes réelles sont codées sur 64 bits. Pour que les constantes réelles soient codées sur 32 bits, il faut ajouter un suffixe F. Pour les constantes décimales, il faut ajouter un suffixe m.

```
double d1 = 10.2;           // OK réel 64bits
float  f1 = 10.2;           // KO car 10.2 est 64bits et float 32bits
float  f2 = 10.2F;          // OK 10.2F est 32bits
double d2 = 1.2E-2;         // d2 = 0,012
float  f3  = 3.1E2F;        // f3=310
decimal d4=2.5m;            //type 128bits
```

Constantes chaînes de caractères

```
string s1="abc";
```

Chaîne verbatim **:** lorsque la chaîne est préfixée par @, aucun des caractères entre les guillemets n'est interprété. Pour représenter les " dans une telle chaîne, il faut les doubler

```
string s2=@"a\nebc";
string s3=@"a\n""ebc";      // s ppv a\n"ebc
```

Caractères de contrôle: \n (newline) \t (horizontal tab) \v (vertical tab) \b (backspace) \r (carriage return) \f (form feed) \\ (backslash) \' (single quote) \" (double quote) \0 (caractere de valeur 0).

2.4. Les fonctions et les passages de paramètres (ref, out)

En C#, les fonctions sont nécessairement définies au sein de types structurés (struct ou class). Il n'y a pas de fonction "globale" au sens du langage C. Néanmoins, les fonctions membres suivent la même logique de définition et d'appel que les fonctions du C. La principale différence est que le compilateur C# n'a pas besoin de fichier de prototypes (les fichiers .h header du C) ce qui élimine beaucoup de problèmes de compilation.

En C#, il n'y a pas de prototype pour les fonctions (différent du C)
Le passage des paramètres se fait par valeur (comme en C)

```
using System;
namespace ConsoleApplication
{
    class Program
    {
        static int f1(int param){       return 2*param;       }

        static void Main(string[] args)
        {
            Console.WriteLine("f1(3)={0}", f1(3));   //sortie console : f1(3)=6
        }
    }
}
```

Les paramètres modifiables (ref et out)

Par défaut, les paramètres sont passés par valeur, comme en C. Les éventuelles modifications du paramètre dans la fonction n'ont alors aucune incidence sur le paramètre d'appel.

Passage par valeur (comme en C)

```
class Program
{
    static int f1(int param)     // param est une copie
    {
        param++;          // param est incrémenté => sans effet sur x
    }

    static void Main(string[] args)
    {
          int x=2;
          f1(x);
          Console.WriteLine("x={0}", x);       //sortie console : x=2
    }
}
```

Le C# introduit des mots clés pour que des paramètres d'appel puissent être ***modifiables*** dans une fonction. Les mots clés utilisés en C# sont

`ref` : indique que le paramètre formel est un alias du paramètre d'appel. C'est la même variable mais avec un autre nom au sein de la fonction. Convient pour des **ENTREE/SORTIE**

`out` : le paramètre est seulement en **SORTIE.**

Paramètres de sortie (out) et entrée/sortie (ref)

```
using System;

namespace ConsoleApplication
{
    class Program
    {

        static void Inc(ref int EntreeSortie)     // le paramètre est une entrée sortie
        {                                         // paramètre en lecture écriture
            EntreeSortie++;
        }

        static void Reset(out int Sortie)   // le paramètre est en sortie
        {
            Sortie=0;
        }

        static void Main(string[] args)
        {
            int x = 2;
            Inc(ref x);                   // mettre le mot clé ref pour l'appel !
            Console.WriteLine(x);         // x vaut 3
            Reset(out x);                 // mettre le mot clé out pour l'appel !
            Console.WriteLine(x);         // x vaut 0
        }
    }
}
```

2.5. Définition d'un type structuré (similaire aux structures du C)

Le mot clé **`struct`** permet la définition d'un type. Le type créé représente des variables structurées avec un ou

plusieurs champs de données. Ci dessous, une variable de type **PointS** est une variable contenant deux champs entiers. Ces champs s'appellent **X** et **Y** et peuvent être désignés à l'aide de la notation **p.X** et **p.Y** où **p** est une variable de type **PointS**. C'est très proche des types structurés du langage C.

La nouveauté est que l'on doit préciser un niveau **d'accessibilité** (ici les champs sont publics).

```
using System;

namespace Exemple
{
    struct PointS   // PointS est un type
    {
        public int X;
        public int Y;
    }

    class Program
    {
        static void Main(string[] args)
        {
            PointS p1,p2;  // p1 et p2 sont des variables de type PointS
            p1.X = 3;
            p1.Y = 4;

            p2.X = 6;
            p2.Y = 1;
        }
    }
}
```

2.6. Type structuré (struct) avec des fonctions membres

On peut avoir des ***fonctions membres*** dans les types structurés, et pas uniquement des données. Il suffit de définir des fonctions au sein des blocs décrivant le type structuré. Dès lors, sur une variable, on peut appeler des fonctions rattachées à cette variable. De plus, on peut avoir plusieurs fonctions membres ayant le même nom, mais des signatures différentes. C'est ce qu'on appelle ***la surcharge***.

```
using System;
namespace Exemple
{
    struct PointS
    {
        public int X;
        public int Y;

        public void SetXY(int pX, int pY){     X = pX;     Y = pY;  }

       // les deux fonctions GetX ont des signatures différentes → OK

        public int GetX(){  return X;  }

        public void GetX(out int pX){  pX = X;    }

        public string ToString()
        {
            return "(" + X.ToString() + "," + Y.ToString() + ")";
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            PointS p1;   // p1,p2 sont des variables de type PointS
            PointS p2;

            p1.SetXY(2, 3);  // appel de la fonction membre SetXY sur la variable p1
            p2.SetXY(6, 1);

            Console.WriteLine("p2.X={0}", p2.GetX());
            Console.WriteLine("p1" + p1.ToString());
            Console.WriteLine("p2" + p2.ToString());
        }
    }
}
```

Sortie console

```
p2.X=6
p1(2,3)
p2(6,1)
```

3. La programmation orientée objet en C#

3.1. Le paradigme objet en bref (vocabulaire)

La *Programmation Orientée Objet (POO)* est une façon de programmer et de décomposer des applications.

Qu'est-ce qu'un objet ? C'est une variable structurée avec des fonctions membres.

Ses attributs ? ce sont les champs de données d'un objet. Ils décrivent ses caractéristiques. Par exemple, **`X`** et **`Y`** sont les attributs d'un objet de type **`PointS`**

Ses méthodes ? autre nom (utilisé en POO) pour les fonctions membres. Les méthodes représentent les mécanismes/ les fonctionnalités dont dispose un objet. C'est ce qu'il sait faire, son *comportement*. L'ensemble des méthodes d'un objet est aussi appelé *l'interface d'un objet*.

L'encapsulation ? en POO, les attributs ne sont généralement pas accessibles directement (contrairement à X et Y dans le type PointS) par l'utilisateur. Généralement, les données d'un objet (valeurs de ses attributs) sont accessibles seulement via ses méthodes. Le fait de cacher les données d'un objet à l'utilisateur est appelé ***encapsulation.*** Le but de l'encapsulation est de forcer l'utilisateur à se concentrer uniquement sur les méthodes de l'objet, ce qu'il sait faire, sans forcément savoir comment il le fait.

Etat d'un objet ? c'est la valeur de l'ensemble de ses attributs, à un instant donné. Par exemple, l'état d'un objet de type **`PointS`** est simplement la valeur de ses coordonnées. Les objets d'un même type ont tous les mêmes attributs, les mêmes caractéristiques, mais des états différents. Tous les objets `PointS` ont X et Y comme attributs, mais la valeur de ces attributs n'est pas identique pour tous les objets.

3.2. Le mot clé class (class vs struct)

Le C# utilise le mot clé **`class`** pour pouvoir définir d'autres types structurés avec des méthodes (**`struct`** ne convient pas à tous les objets). Les types **`class`** sont plus riches que les types **`struct`** et conduisent à un fonctionnement différent de la gestion de la mémoire.

```
using System;
namespace TypeClass
{
    class Program
    {
        class PointC
        {
              // membres privés = inaccessibles depuis l'extérieur de l'objet (encapsulés)

            private int X;
            private int Y;

              // en revanche, les méthodes sont publiques

            public void SetXY(int pX,int pY){      X = pX;    Y = pY;     }

            public void GetXY(out int pX, out int pY){   pX = X;  pY = Y; }

            public string ToString()
            {
                return "(" + X.ToString() + "," + Y.ToString() + ")";
            }
        }

        static void Main(string[] args)
        {
            PointC p3 = new PointC();
            PointC p4 = new PointC();

            p3.SetXY(2, 3);
            p4.SetXY(6, 1);

            Console.WriteLine(p3.ToString()); //affiche état de p3
            Console.WriteLine(p4.ToString()); //affiche état de p4

            int x, y;
            p3.GetXY(out x, out y);   // obtient l'état de l'objet p3
            Console.WriteLine("x={0} y={1}",x,y);

           // p3.X = 4; // IMPOSSIBLE → p3.X n'est pas accessible
        }
    }
}
```

Sortie

```
(2,3)
(6,1)
x=2 y=3
```

Qu'est-ce que l'instanciation ? C'est le fait de **créer un objet** (une instance d'une classe). Un type classe décrit les caractéristiques d'un ensemble d'objets. Un objet est une ***instance particulière*** d'un **ensemble** potentiellement infini d'objets ayant les mêmes méthodes et les mêmes attributs.

Pour illustrer, ***Etre Humain*** représente une classe d' "objets" dont les attributs et les méthodes sont nombreux (couleur des yeux, des cheveux, langue(s) parlée(s), taille, poids, marcher, manger, parler …). Chacun d'entre nous est une instance de la classe ***Etre Humain*** avec ses propres valeurs d'attribut (brun, yeux marrons,…).

De même ***Voiture*** est une classe d'objets dont les attributs sont par exemple le nombre de portes, la couleur, la puissance, la motorisation, la cylindrée, les dimensions … La voiture Renault Clio II 1.6 16V immatriculée FR 425 TZ est une instance de la classe ***Voiture***.

Dans l'exemple précédent, p3 et p4 désignent deux instances (deux objets) de la classe **`PointC`**.

3.3. value-type / reference_type

En C#, la ***zone mémoire d'allocation*** d'une variable/d'un objet dépend de son type (différent du langage C).

En C#, on distingue les types **valeur** (**value_type**) et les types **référence** (**reference_type**).

Les objets ***value_type*** → | alloués dans la pile (STACK)
| durée de vie = le bloc de déclaration

Les objets ***reference_type*** → | alloués dans le tas (HEAP) par l'instruction `new TYPE`
| durée de vue = de (**allocation)** à (**plus référencé**)

Quels sont les types *value_type* ?

Les types numériques primitifs du C# (byte, char, long …), les énumérations et les variables d'un type **`struct`**. Par exemple, **`PointS`** est value_type. Quand on crée une variable de type **`PointS`**, elle est donc réservée automatiquement dans la pile.

Quels sont les types *reference_type* ?

Les types créés avec le mot clé **`class`** et le type **`string`**.

Pour les objets **reference_type**, la mémoire est allouée dynamiquement (comme avec **`malloc()`** en C) dans le tas avec l'instruction **`new ___`**. Quand on crée un objet, on récupère ***une référence*** (une sorte de pointeur) sur l'emplacement réservé.

Quand la mémoire est-elle désallouée ? pour les objets value_type, c'est automatique à la fin du bloc de déclaration. Pour les objets reference_type, quand il n'y a plus aucune référence sur l'objet, le *Garbage Collector* récupère la mémoire de l'objet (désallocation). Il n'y a donc pas d'équivalent de la fonction **`free()`**.

3.4. Distinctions value_type/reference_type concernant l'affectation

Affectation pour le objets value_type

```
int i=18;  // i est dans la pile et contient 18 codé sur 32 bits signés
```

Remarque : même ci-dessous, la variable est dans la pile quand même.

```
int i = new int();   //i est dans la pile parce que int est value_type
```

L'affectation value_type réalise l'affectation des valeurs.

```
int a=12,b=4;        // a contient 12 et b contient 4
a=b;                 // a contient désormais 4 aussi.
```

Idem pour les variables de type struct : l'affectation affecte la valeur, champs par champs.

```
static void Main(string[] args)
     {
         PointS p1 = new PointS();  // p1 et p2 dans la pile (car value_type)
         PointS p2 = new PointS();      // même si on écrit new PointS()

         p1.SetXY(2, 3);  // p1 <- l'état (2,3)

         p2 = p1;         // l'état de p1, (2,3), est affecté à p2

         // p2=p1  <=> (p2.X=p1.X ET p2.Y=p1.Y)  (même quand X et Y sont privés)

         Console.WriteLine("p2 = " + p2.ToString());   // p2 = (2,3)
     }
```

Affectation pour objets reference_type

L'accès à un objet **reference_type** se fait via une **référence**, ce *qui peut être vu comme une sorte de pointeur.*
D'ailleurs, la valeur de référence **`null`** signifie ***ne désigne pas d'objet***

```
PointC p3;     // p3 est une référence (car PointC est reference_type)
               // p3 ne désigne encore aucun objet (p3 vaut null par défaut)

p3 = new PointC();    // alloue un objet (dans le tas) et met la référence dans p3
```

Il faut voir les références (ici p3) comme des sortes de pointeurs. Une référence est initialisée à la valeur `null` par défaut.

```
static void Main(string[] args)
{
     PointC p3;       // p3 vaut null
     PointC p4;       // p4 vaut null

       p3 = new PointC();     // p3 désigne une instance PointC
       p4 = new PointC();    // p4 désigne une autre instance PointC

     p3.SetXY(2, 3);
     p4.SetXY(6, 1);
}
```

Pour les objets reference_type, l'affectation réalise seulement une copie des références, comparable à une copie de pointeurs en C.
Dans l'exemple ci-dessous, après l'affectation p4=p3, p4 et p3 désignent **le même objet** ! L'objet d'état (6,1) qui était avant désigné par p4 n'a donc plus de référence : il sera désalloué à la prochaine exécution du Garbage Collector.

```
static void Main(string[] args)
{
       PointC  p3 = new PointC();   // crée une instance (A)
       PointC  p4 = new PointC();   // crée une seconde instance (B)
                                    // p3 et p4 désignent deux objets différents (A)!=(B)

       p3.SetXY(2, 3);       // l'objet (A) désigné par p3 vaut (2,3)
       p4.SetXY(6, 1);       // celui désigné par p4 vaut (6,1), c'est (B)

       p4=p3; // p4 et p3 désignent désormais le même objet (A)
       // l'objet (B), qui vaut (6,1), n'est maintenant plus accessible !
       // Le Garbage Collector va récupérer sa mémoire dès que possible.
}
```

3.5. Distinction value_type/ref_type concernant le test d'égalité (==)

L'opérateur == pour les value_type : par défaut (mais l'opérateur == peut être redéfini), le test avec l'opérateur == sur des value_type teste l'égalité de valeur des champs. Il y a comparaison de la valeur de tous les champs. L'opérateur renvoie la valeur true seulement si tous les champs ont les mêmes valeurs deux à deux

L'opérateur == pour les reference_type: par défaut, l'opérateur == teste si deux références désignent le même objet. Mais puisque l'opérateur == peut être reprogrammé (c'est une possibilité de ce langage), il peut aussi réaliser un traitement différent. C'est le cas pour les chaînes de type string (reference_type) où l'opérateur == indique l'égalité des chaînes (même longueur et mêmes caractères).

Comment tester (version sûre) si deux références différentes désignent le même objet ?

```
object.ReferenceEquals(p3,p4);  //true si p3 et p4 désignent le même objet
```

4. L'écriture de classes en C# (reference_type)

4.1. Namespaces

Le mot clé **`namespace`** définit un espace de nommage. Ceci permet de structurer/hierarchiser les types et de lever d'éventuels conflits de noms. On peut avoir un même nom de type dans différents namespaces. Ci-dessous, on définit une classe `A` dans différents namespaces. Il s'agit donc de classes différentes, même si elles ont même nom. Les namespaces peuvent être imbriqués. Le nom explicite du type `A` du namespace `EspaceA` est **`EspaceA.A`**.

```
namespace EspaceA            // namespace avec deux types A et B
{
    class A { }      // le type A du namespace EspaceA = EspaceA.A
    class B { }
}

namespace EspaceB  // namespace avec un namespace imbriqué
{
    namespace ClassesImportantes
    {
        class A { }   // ce type est  EspaceB.ClassesImportantes.A
    }
}
namespace EspaceB.Divers  // equivaut à namespace EspaceB{ namespace Divers{... } }
{
    class A { }   // ce type est EspaceB.Divers.A
}
namespace EspaceB.ClassesImportantes  // on complète le namespace
{
    class B { }   // ce type est EspaceB.ClassesImportantes.B
}
namespace ExempleNamespace   // namespace de la classe qui contient Main
{
    class Program
    {
        static void Main(string[] args)
        {
            EspaceA.A p1 = new EspaceA.A();
            EspaceB.ClassesImportantes.A p2 = new EspaceB.ClassesImportantes.A();
            EspaceB.Divers.A p3 = new EspaceB.Divers.A();
            EspaceB.ClassesImportantes.B p4;
        }
    }
}
```

Directive using

Cette directive permet d'importer un namespace. Celà signifie que s'il n'y a pas d'ambiguité, les noms de types seront automatiquement préfixés par le namespace. On allège ainsi l'écriture des noms de type.

Par exemple, la classe `Console` (I/O) dépend du namespace `System`. Le nom complet de cette classe est donc `System.Console`. L'utilisation devrait donc s'écrire

```
System.Console.WriteLine("Hello!");
```

Ce qui signifie *"appel de la méthode `WriteLine` du type `Console` du namespace `System`"*.

En important ce namespace (en début de fichier) avec le mot clé **`using`**, on peut simplifier l'écriture.

```
using System; // import namespace System
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
              Console.WriteLine("Hello!);         // le type Console est recherché dans
                                                  // le namespace System
        }
    }
}
```

4.2. Accessibilité

Les membres définis au sein des types peuvent avoir un niveau d'accessibilité parmi

`public` : membre (donnée ou fonction) accessible par tous

`private` : membre accessible seulement par une méthode d'un objet de la classe

`protected` : membre accessible par une méthode d'un objet de la classe ou par une méthode d'un objet d'une classe dérivée (voir plus tard la dérivation)

C'est grâce à ces modificateurs d'accès que l'on assure l'encapsulation des données. Tous les membres (attributs, méthodes ...) doivent être déclarés avec un niveau d'accessibilité. Par défaut, un membre est privé.

Il y a aussi des modificateurs d'accès spécifiques (non expliqué dans ce support)
`internal` : accès autorisé pour l'assemblage courant
`protected internal` : assemblage courant + sous classes

```
class MaClasse
{
    private int _valeur;
    public MaClasse(int v)
    {
       _valeur=v;
    }
    public void Affiche()
    {
       Console.WriteLine(_valeur);
    }
}
```

Ci-dessous un exemple de classe (reference_type) **`MaClasse`** et son utilisation dans le programme. Notons qu'il est possible aussi de donner une initialisation des attributs pour les types **`class`**. Ce n'est pas autorisé pour les types **`struct`**.

```
using System;
namespace SyntaxeCSharp
{
    class MaClasse
    {
       private int _valeur =2;      // initialiseur de champs

       public MaClasse(int v) {     _valeur = v;  }

       public void Affiche() {   Console.WriteLine("Objet MaClasse = {0}",_valeur);          }
    }

class Program
{
        static void Main(string[] args)
        {   MaClasse m1 = new MaClasse(13);
            m1.Affiche();
            MaClasse m2 = m1; // 2ième référence à l'objet
            if (object.ReferenceEquals(m2,m1)) Console.WriteLine("Mêmes Objets");
        }
    }
}
```

Sortie Console
```
Objet MaClasse = 13
Mêmes Objets
```

4.3. Surcharge des méthodes

Surcharger les méthodes, c'est définir plusieurs méthodes ayant le même nom mais des *signatures différentes*

- dont le nombre de paramètres est diffèrent
- le type des paramètres est différent
- les paramètres peuvent aussi avoir même type même nombre mais différer du mode de passage de paramètre (par valeur ou par référence)

En revanche, le type de retour d'une fonction ne fait pas partie de la signature. On ne peut pas déclarer deux fonctions qui ne diffèrent que par le type de retour.

```
class A
{
   private int _val;

   public A(){ _val=0;}

   public int Val(){ return _val; }       // 1)

   public void Val(int v){_val = v;  }    // 2)

   public void Val(out int v){ v = _val;}  // 3)
}

class Programme
{
    static void Main()
    {
        A objA = new A();              // instancie A (crée un objet)

        objA.Val(2);                   // appelle 2)
        int v;
        objA.Val(out v);               // appelle 3)
        Console.WriteLine(objA.Val()); // appelle 1)
    }
}
```

4.4. Champs constants / en lecture seule

```
class MaClasse
{
       public const int _valC=12;   //par défaut statique
}
```

Un littéral (constante) doit être utilisé pour initialiser un champ constant.

```
class MaClasse
{
       public readonly int _valRO=12;
}
```

La valeur d'un attribut **`readonly`** peut être fixée par le constructeur (cf 4.6). C'est donc une constante dont l'initialisation de la valeur peut être retardée jusqu'au moment de l'exécution (et non à la compilation).

```
class MaClasse
{
       private readonly int _valRO;
       public MaClasse(int val){   _valRO=val;   }
}
```

4.5. Objet courant (référence this)

✍ **Dans une méthode, on appelle *Objet Courant* l'objet sur lequel est appelée la méthode.**

Dans l'exemple ci-dessous, quand on invoque **`obj1.Compare(obj2)`**, l'objet courant est **`obj1`**. Dans une méthode, on peut désigner l'objet courant par la référence **`this`**. Notons que par défaut dans une méthode, quand on désigne un attribut sans préfixer par une référence d'objet, il s'agit d'un attribut de l'objet courant.

```
using System;
namespace ObjCourant
{
    class MaClasse
    {
        private int _val;

        public MaClasse(int v)  // constructeur (cf section suivante)
        {
            _val = v;   // équivaut à  this._val=v;
        }

        public bool Compare(MaClasse obj)
        {
            return _val == obj._val;  // équivaut à return this._val==_obj.val;
        }
    }

    class Program
    {
        static void Main()
        {
            MaClasse obj1 = new MaClasse(12);   // 2 instances différentes
            MaClasse obj2 = new MaClasse(14);

            Console.WriteLine("obj1 et obj2 ont le même état ? = {0}", obj1.Compare(obj2));
        }
    }
}
```

4.6. ✍ Constructeurs (initialisation des objets)

Un ***constructeur*** est une méthode qui **initialise** l'état d'un objet (ne retourne rien), juste après l'allocation de l'objet (au tout début de la vie de l'objet). Il faut d'ailleurs éviter qu'un objet ait un état initial indéfini : il faut toujours prévoir au moins un constructeur dans une classe. Il peut aussi y avoir surcharge des constructeurs, c'est-à-dire plusieurs façons d'initialiser un objet.

Un constructeur s'appelle comme la classe. Un constructeur peut recevoir un ou plusieurs arguments pour initialiser l'objet. Ces arguments sont fournis au moment de l'instanciation.

✍ **Remarque (constructeur par défaut)** : si l'on ne met aucun constructeur dans une classe, il y en a toujours au minimum **un par défaut qui est sans argument**. Ce constructeur par défaut ne fait aucun traitement mais permet d'instancier la classe. **MAIS, aussitôt qu'un constructeur est spécifié, le constructeur par défaut n'est plus mis en place.**

```
using System;

namespace SyntaxeCSharp
{
    class MaClasse          // classe avec 3 constructeurs = 3 façons d'initialiser
    {
              private string _nom;
              private int _val;
              public MaClasse()       // constructeur C1) sans argument
              {
                     _nom = "Nestor";
                     _val = 12;
              }
              public MaClasse(int val)  // constructeur C2) avec un argument de type int
              {
                     _val = val;
                     _nom = "Nestor";
              }
              public MaClasse(int val, string nom) // constructeur C3)
              {
                     _val = val;
                     _nom = nom;
              }
    }

    class Program
    {
        static void Main(string[] args)
        {
         // 🗷  remarquer comment on distingue l'utilisation de tel ou tel constructeur
            MaClasse m1 = new MaClasse();         // utilise C1) donc m1 vaut ("Nestor",12)
            MaClasse m2 = new MaClasse(23);       // utilise C2) donc m2 vaut ("Nestor",23)
            MaClasse m3 = new MaClasse(17,"Paul");// utilise C3) donc m3 vaut ("Paul",17)
        }
    }
}
```

4.7. Membres statiques (données ou fonctions)

Une ***donnée membre statique*** (appelée aussi ***variable de classe***) n'existe qu'en un seul exemplaire, quel que soit le nombre d'objets créés (même si aucun objet n'existe). C'est une variable unique, quel que soit le nombre d'instances. **Un attribut est une variable d'instance (propre à un objet) ≠ variable de classe.**

Une ***fonction membre statique*** (appelée aussi ***méthode de classe***) peut être appelée *indépendamment de toute instance*, elle n'accède donc pas aux attributs des objets de la classe. En revanche, une fonction membre statique peut accéder aux données membres statiques. Une telle fonction peut être invoquée même si aucune instance n'existe. Ce mécanisme remplace les fonctions hors classes du C++. Par exemple, les fonctions mathématiques sont des fonctions statiques de la classe **`System.Math`**. **Méthode d'instance ≠ méthode de classe.**

```
Console.WriteLine(System.Math.Sin(3.14/2));   // Sin() est fonction statique de System.Math
```

🗷 **Remarque :** il faut souligner la différence de notation pour l'appel d'une fonction statique. Elle s'appelle **SANS OBJET COURANT**

```
Namespace.NomClasse.FonctionStatique(___)

System.Math.Sin(3.14/2)
```

```
using System;               //Exemple de classe avec membres statiques
namespace SyntaxeCSharp
{
   class MaClasse
    {
        static private int _nbObjetsCrees = 0;  // VARIABLE DE CLASSE = UNIQUE

        private int _x;   // ATTRIBUT = variable d'instance, chaque objet possède cet attribut

        static public int GetNbObjets(){    return _nbObjetsCrees; }  // METHODE DE CLASSE

        public MaClasse(int X)
        {
              _x=X;                // initialise l'attribut de l'objet courant
              _nbObjetsCrees++;    // incrémente la variable de classe (statique)
        }
     }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(MaClasse.GetNbObjets());   // ici 0 (pas d'objet de créé)
            MaClasse m = new MaClasse(3);                // m désigne un objet où m._x=3
            Console.WriteLine(MaClasse.GetNbObjets());   // ici 1 (car 1 objet a été créé)
            m = new MaClasse(7);                         // m désigne un nouvel objet où m._x=7
            Console.WriteLine(MaClasse.GetNbObjets());   // ici 2
        }
    }
}
```

4.7.1. Constructeur statique

Constructeur appelé une fois avant tout autre constructeur. Permet l'initialisation des variables de classe. Il ne faut pas donner de qualificatif de visibilité (public/protected/private) pour ce constructeur.

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
   {
            private string _str;         // attribut
            static private int _stat;    // variable de classe (statique)

            static MaClasse() { _stat = -1; }                // CS) constr. statique
            public MaClasse(string s) { _str=s; _stat++; }   // C1) constructeur
            public void Affiche() { Console.WriteLine("({0},{1})",_str,_stat); }
       }
       class Program
       {
        static void Main(string[] args)
        {            // MaClasse._stat a été initialisé par CS) avant ce code
                     // MaClass._stat vaut donc -1 ici
            MaClasse m1 = new MaClasse("toto");          // utilise C1)
                     // MaClass._stat vaut 0
            MaClasse m2 = new MaClasse("tata");          // utilise C1)
                     // MaClass._stat vaut 1
            m1.Affiche();     // affiche (toto,1)
            m2.Affiche();     // affiche (tata,1)
        }
       }
}
```

4.8. ✍✍ Distinction des attributs value_type et reference_type

Puisque l'on distingue les objets reference_type (dans le tas) et les objets value_type (dans la pile), leur traitement en tant qu'attributs de classe diffère également.

```
using System;
namespace SyntaxeCSharp
{
    class Programme
    {
        class A   // reference_type
        {
              public int _valeur;  //attribut value_type

              public A(int valeur) {   _valeur = valeur;   }
              public int GetValeur() {     return _valeur;  }
        }

        class B  // reference_type
        {
              private double _d;    // attribut value_type
              private A _a;         // attribut reference_type

              public B(){
                _d = 12.5;          // this._d existe dès que l'objet courant existe
                _a = new A(13);     // ✍  instanciation nécessaire ici !
              }
        }
        static void Main()
        {
            A objA = new A(13);
            B objB = new B();
        }
    }
}
```

Quand on crée un objet de la classe `A`, son attribut `_valeur` est automatiquement créé (puisque value_type).

Pour un objet de la classe `B` c'est différent. Son attribut `_d` (value_type) est créé automatiquement mais _a n'est qu'une référence ! Il faut créer un objet de la classe `A` dans le constructeur de l'objet. Faute de quoi, la référence `_a` vaut la valeur `null` (valeur par défaut pour une référence).

4.9. ✍ Propriétés

En C#, on peut définir des ***propriétés***. Il s'agit de méthodes particulières qui donnent ***l'illusion*** que certains attributs sont publics. Leur accès est en réalité contrôlé par des méthodes particulières, en lecture/en écriture.

Attention, il y a une différence d'écriture (minime) entre définition d'une méthode et définition d'une propriété.

Dans l'exemple ci-après, on dote la classe d'une propriété `Valeur`. L'utilisation de la propriété s'écrit

```
m.Valeur     // comme si l'objet m avait un attribut public ! Ce n'est pas le cas
```

✍ **Mot clé value :** dans les propriétés, le mot clé **`value`** désigne la valeur reçue en écriture (partie **`set`**).

```
using System;               // exemple de classe avec Propriété
namespace SyntaxeCSharp
{
    class Programme
    {
       class MaClasse
        {
              private int _valeur=2;       // attribut privé _valeur

              public int Valeur            // ☞  Propriété (publique) Valeur
              {
                     get
                     {
                             return _valeur;
                     }
                     set    // value représente la valeur reçue en écriture
                     {
                             if(value >=0) _valeur = value;
                             else _valeur=0;
                     }
              }

                     /* si c'était une méthode, ça s'écrirait avec des ( ) ...
                     public int Valeur()  // syntaxe d'une Méthode Valeur();
                     {
                             return _valeur;
                     }
                     */
        }
        static void Main()
        {
            MaClasse m = new MaClasse();          // m.Valeur  =  propriété

            // m.Valeur = ___;  // écriture
            m.Valeur = 13;   // accès en écriture -> set
            m.Valeur = -2;   // m._valeur ppv 0

            //  ____ = m.Valeur;  // lecture
            Console.WriteLine(m.Valeur);   // accès en lecture -> get

            /* si c'était une méthode, on écrirait m.Valeur() avec parenthèses */
        }
    }
}
```

Notons que les propriétés peuvent être ***en lecture seule***. On ne fournit alors que la partie **`get`** de la propriété

```
class MaClasse
{
       private int _valeur=2;
       public int Valeur
       {
              get {   return _valeur; }    // lecture seule
       }
}
class Programme
{
       MaClasse m=new MaClasse();
       Console.WriteLine(m.Valeur);    //OK
       m.Valeur=2;   //KO pas d'écriture possible
}
```

4.10. Indexeur (quand un objet se prend pour un tableau)

Ce mécanisme permet d'appliquer des crochets [ ] à un objet, c'est-à-dire le considérer comme un tableau. Pour utiliser ce mécanisme, il faut définir une méthode **`this[ ]`** dont le paramètre est la variable qui sert d'indice. Ci-dessous un exemple où il y a deux indexeurs pour la classe **`Tableau`**.

```
using System;
using System.Collections;
namespace SyntaxeCSharp
{
       class Tableau
       {
            private ArrayList _tab;   // l'usage de ArrayList est un peu obsolète !

            public Tableau(){ _tab = new ArrayList(); }

            public string this[int idx]   // indexeur 1) lecture/écriture
            {
                get{  return (string)_tab[idx];   } // retourne la chaîne en indice idx
                set   // modifie la chaine en indice idx
               {
                     if (idx < _tab.Count) _tab[idx] = value;
                     else  _tab.Add(value);  // ajout en fin si idx trop grand
               }
            }

            public int this[string str]   // indexeur 2) en lecture seule
            {
                get{   // fournit l'indice de la chaîne str ou (-1) si absente
                    for (int idx = 0; idx < _tab.Count; idx++)
                    {
                        if ((string)_tab[idx] == str) return idx;
                    }
                    return -1;
                }
            }
       }
       class Programme
       {
              static void Main()
              {
                Tableau T1 = new Tableau();
                T1[0] = "abc";    // utilise indexeur 1) en écriture car l'indice est int
                T1[1] = "def";
                Console.WriteLine(T1[0]);  // utilise 1) en lecture
                Console.WriteLine("Indice de \"def\" : {0}", T1["def"]); // utilise indexeur 2)
              }
       }
}
```

4.11. Conversion Numérique ↔ Chaîne (format/parse)

4.11.1. Formatage (conversion numérique vers string)

Tout objet dérive de **`object`** et dispose donc d'une méthode **`ToString()`**.

```
int d=12;
string s=d.ToString();    // fournit une chaîne qui décrit l'entier d
Console.WriteLine(s);
```

Remarque : il y a aussi pour les types numériques une méthode **`ToString(string format, IFormatProvider provider)`** qui permet de préciser le formattage (cf section sur la classe string)

4.11.2. Parsing (conversion string vers numérique)

Les types numériques (int,float,...) disposent d'une méthode **`Parse()`**. Cette méthode génère une exception si le format de la chaîne est incorrect

`static T  Parse(string str)` [T est le type numérique]

```
string s;
 s=Console.ReadLine();      //lit une chaîne au clavier
 int var;
 try
 {
       var = int.Parse(s);          // Tente de convertir une chaîne en entier
       var = int.Parse("13");
 }
 catch (Exception e)        // si conversion impossible
 {
   Console.WriteLine(e.Message);
 }
```

Une seconde méthode **`TryParse()`**, plus rapide, retourne un booléen pour indiquer le succès ou l'échec de la conversion. Cette méthode ne déclenche pas d'exception.

`static bool TryParse(string str, out T var)`

```
string s;
 s=Console.ReadLine(); //lit une chaîne
 int var;
 if(int.TryParse(s,out var)==false)
 {
       Console.WriteLine("La chaîne ne respecte pas le format int");
 }
```

4.12. Les énumérations (value_type)

Un type créé avec le mot clé **`enum`** représente des variables pouvant prendre un nombre fini de valeurs prédéfinies. Une variable d'un type **énumération** peut prendre pour valeur une des valeurs définies dans la liste décrivant l'énumération :

```
enum Feu{vert,orange,rouge};
...
Feu f;       // f est une variable
f=Feu.vert;  // Feu.vert est une valeur
```

Les valeurs d'une énumération correspondent à des valeurs entières qu'on peut choisir. On peut choisir aussi le type entier sous-jacent. On peut transtyper une valeur énumération en valeur entière. On peut aussi appliquer certaines opérateurs :

`== (compare), +, -, ++, – , &, |,~`

Exemple de définition de type enumération

```
using System;
namespace SyntaxeCSharp
{
    enum Color { red, blue, green };       // vaut 0,1,2
    enum Access { personal = 1, group = 2, all = 4 };
    enum AccessB : byte { personal = 1, group = 2, all = 4 };   // codé sur 8 bits
```

```
    class Programme
    {
        static void Main()
        {
            Color c = Color.blue;
            Console.WriteLine("Couleur {0}",(int)c); // Couleur 1
            Access a1 = Access.group;
            Access a2 = Access.personal;
            Console.WriteLine((int)a1);
            Console.WriteLine((int)a2);
            Access a3 = a1 | a2;    // ou bit à bit sur entier sous-jacent
            Console.WriteLine((int)a3);
        }
    }
}
```

4.13. Notation UML des classes.

UML (Unified Modelling Language) est un langage de modélisation graphique pour les objets. L'UML comprend différents diagrammes dont le diagramme de classes qui représente la structure des classes (attributs et méthodes) ainsi que les relations entre les classes (et donc entre les objets).

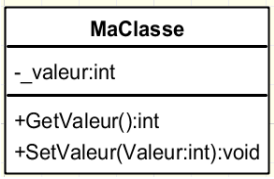
Chaque classe est représentée par un rectangle.

La première partie décrit les attributs, la seconde les méthodes. La visibilité est décrite par un symbole

- privé

+ public

protégé



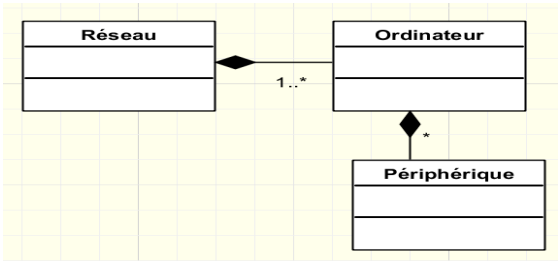
Le diagramme précédent représente la classe C# ci-dessous. Notons que le détail des méthodes n'est pas indiqué dans ce diagramme. Seul le nom des méthodes évoque le traitement sous-jacent.

```
class MaClasse
{
       private int _valeur;
       public int GetValeur() { ... }
       public void SetValeur(int Valeur) { ….  }
}
```

5. La composition (objet avec des objets membres)

La programmation orientée objet permet de décrire les applications comme des collaborations d'objets. Par conséquent, dans les diagrammes de classe, certaines relations apparaissent entre les classes. Par exemple, certaines méthodes d'une classe utilisent les services d'une autre classe. Il n'est pas question de rentrer dans les détails de la modélisation par UML, mais on peut toutefois s'intéresser à certaines relations importantes comme la composition et la dérivation (cf. Section suivante)

La composition décrit la situation où un objet est lui même constitué d'objets d'autres classes. Par exemple un ordinateur est composé d'une unité centrale et de périphériques. Un livre est constitué de différents chapitres.

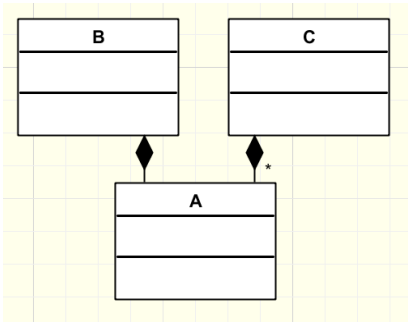


En UML, cette relation particulière est décrite par une relation où un losange est dessiné côté composite (l'objet qui agrège).

Pour le diagramme ci-contre, la relation entre la classe Réseau et la classe Ordinateur signifie : "*un réseau est constitué de 1 ou plusieurs ordinateurs (1..*)* "

Dans le langage objet cible, cette relation de composition est simplement décrite par le fait qu'une classe puisse avoir un ou plusieurs attributs de type classe (ou struct).

```
class A{    }

 class B
 {
     private A _objet;  // reference_type

     public B(){  _objet = new A();  }
 }

 class C
 {
     private A[] _objets; // reference_type

     public C(int nb)
     {
         _objets = new A[nb]; // crée un tableau
         // crée tous les objets du tableau
         for(int i=0;i<nb;i++) _objets[i]=new A();
     }
 }
```



Les classes ci-dessus décrivent les relations suivantes :

un objet de la classe B **EST COMPOSE** d'un objet de la classe A

un objet de la classe C **EST COMPOSE** de 0, 1 ou plusieurs objets de la classe A

6. ☙☙ La dérivation

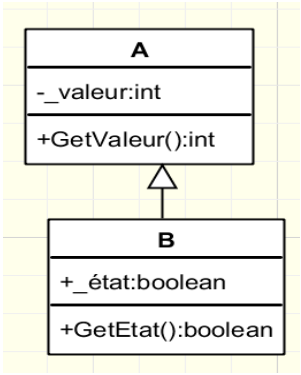
La dérivation est une seconde relation **essentielle** de la programmation orientée objet.

La dérivation est un mécanisme fourni par le langage qui permet **d'étendre** (ou spécialiser**)** une classe existante (dite classe de base) en lui **ajoutant** de nouveaux attributs / de nouvelles méthodes. On peut aussi voir celà comme un enrichissement d'une classe. La classe étendue (on dit **classe dérivée** en pratique) décrit des objets plus spécifiques. La classe dérivée possède alors intégralement les attributs et les méthodes de la classe de base, **plus** d'autres attributs/méthodes.

Ci-dessous, ***la classe B dérive de la classe A*** (c'est indiqué dans l'entête de la classe B). La classe B possède donc automatiquement les attributs et les méthodes de la classe A, **avec en plus des membres spécifiques.**

On dit aussi que la classe B ***hérite*** des attributs et des méthodes de sa classe de base.

```
class Program
   {
       class A
       {
           private int _valeur;

           public A(int val){    _valeur = val;    }

           public int GetValeur() { return _valeur; }
       }

       class B :  A   // ☙ B dérive de A (étend A)
       {
           private bool _etat;

           public B(int valeur, bool etat):base(valeur)
           {
               _etat = etat;
           }

           public bool GetEtat(){   return _etat;  }
       }

       static void Main(string[] args)
       {
           A objA = new A(12);
           Console.WriteLine(objA.GetValeur());

           B objB = new B(17, true);
           Console.WriteLine(objB.GetValeur());
           Console.WriteLine(objB.GetEtat());
       }
   }
```



☙ **En notation UML, la relation de dérivation est notée avec une flèche partant de la classe dérivée vers la classe de base.**

Un objet de la classe A possède un attribut et une méthode.

Un objet de la classe B possède deux méthodes (GetEtat() et GetValeur()) et deux attributs (_etat et _valeur).

La classe B dérive de la classe A.

La classe A est classe de base de la classe B.

☙☙☙ **Important** : un objet de la classe B peut toujours être considéré comme un objet de la classe A aussi (puisqu'il possède au moins les attributs et méthodes de la classe de base).

6.1. Syntaxe

Ci-dessous, la classe `Derivee` est définie comme classe dérivée de la classe `MaClasse`. Ce qui signifie que tout objet de la classe `Derivee` dispose d'emblée des attributs et méthodes de la classe MaClasse.

```
class Derivee : MaClasse   // signifie cette classe dérive de MaClasse (ou étend MaClasse)
 {
       protected int _increment;
       public Derivee(int val): base(12)
       {
              _increment=val;
       }
  }
```

✍ **Ce qui est spécifique à la classe `Derivee` est appelé *partie incrémentale*. Partie qu'on ajoute.**

6.2. Différence de visibilité protected/private

Un membre **private** n'est pas accessible par l'utilisateur de la classe, ni par un objet de la classe dérivée. Un membre **protected** n'est pas accessible par l'utilisateur de la classe **MAIS est accessible** par la classe dérivée. Voir ci-dessous le membre _valeur.

6.3. ✍✍✍ Initialisateur de constructeur (dans le cas de la dérivation)

✍ La notation :**`base( )`** utilisée au niveau du constructeur indique quel constructeur de la classe de base utiliser pour initialiser la partie héritée d'un objet **`Derivee`**.

```
using System;
namespace SyntaxeCSharp
{
        class MaClasse             //classe de base
        {
               protected int _valeur;
               private string _str;

               public MaClasse(int v){  _valeur=v;  _str="xxx";  }           // constr B1)

               public MaClasse(int v,string s){  _valeur=v;  _str=s; }       // constr B2)
        }

        class Derivee : MaClasse    // cette classe dérive de MaClasse
        {
               protected double _increment;

               public Derivee(): base(10,"toto")                             // constr. D1)
               {              // ✍  initialise partie héritée avec B2)
                  _increment=-1;
               }

               public Derivee(double valInc,string s): base(5,s)             // constr. D2)
               {                                  // ✍  initialise partie héritée avec B2)
                  _increment=valInc;
               }

               public Derivee(double valInc,int valP): base(valP)            //constr. D3)
               {                                     // ✍ initialise partie héritée avec B1)
                  _increment=valInc;
               }
```

```
            public void Set(double inc, int val)
            {
                _valeur = val;      // _valeur accessible car protected
                _increment = inc;
                // _str est inaccessible ici, bien qu'attribut de l'objet courant
            }
        }
    class Programme
    {
        static void Main()
        {
            Derivee da=new Derivee(4.3,12);       // utilise D3) pour initialiser l'objet da
                                                  //da._str="xxx" da._valeur=12 da._increment=4.3

            Derivee db=new Derivee();      // utilise D1) pour initialiser l'objet db
                                           //db._str="toto" db._valeur=10 db._increment=-1

            Derivee dc=new Derivee(3.2,"tutu");   // utilise D2) pour initialiser l'objet dc
                                                  //dc._str="tutu" dc._valeur=5 dc._increment=3.2

            da.Set(2.3, 2);  //da._valeur=2 da._increment=2.3  (da._str="xxx" inchangé)
        }
    }
}
```

Il est important de bien comprendre ce mécanisme en C#. Dans l'exemple ci-dessus :

quand on crée un objet `Derivee` avec D1) → ça utilise B2) pour initialiser la partie héritée
quand on crée un objet `Derivee` avec D3) → ça utilise B1) pour initialiser la partie héritée

6.4. Par défaut, les méthodes sont à ligature statique

Les méthodes sont **non virtuelles par défaut.** Ce qui signifie qu'à partir d'une référence de type `MaClasse`, on ne peut appeler que des méthodes de `MaClasse` ou de ses classes de base. Même si la référence désigne en réalité un objet d'une classe dérivée de `MaClasse` (ce qui est possible). On appelle ça une **ligature statique** puisque la méthode invoquée **est choisie à la compilation**.

Mot clé new : dans le cas de la surdéfinition d'une méthode (non virtuelle) de la classe de base dans la classe dérivée, il faut préciser qu'on surdéfinit la méthode en utilisant le mot-clé **`new.`**

```
using System;

namespace SyntaxeCSharp
{
   class MaClasse
    {
       // ligature statique (par défaut)
       public void Methode(){    Console.WriteLine("MaClasse.Methode");  }
    }

   class Derivee : MaClasse
    {
       new public void Methode()   // new car méthode avec même signature dans l'ascendance
       {      // masque celle de la classe de base
            Console.WriteLine("Derivee.Methode");
       }
    }
```

```
    class Programme
    {
        static void Main()
        {
              MaClasse m=new MaClasse();
              Derivee d=new Derivee();
              m.Methode();
              d.Methode();

              m=d;        // m désigne en réalité un objet de type Derivee
              m.Methode();//MAIS la ligature statique conduit à l'appel de MaClasse.Methode()
        }
    }
}
```

```
Sortie Console
MaClasse.Methode
Derivee.Methode
MaClasse.Methode
```

6.5. Méthodes virtuelles (méthodes à ligature dynamique)

Une méthode **`virtual`** est à ***ligature dynamique***. Il y a alors une recherche *dynamique* de méthode **à l'exécution**. En outre, la surdéfinition d'une méthode virtuelle dans la classe dérivée doit être précédée du mot-clé **`override`**. Par exemple, la méthode `ToString()` est **`virtual`** dans la classe object.

```
using System;
namespace SyntaxeCSharp
{
   class MaClasse
    {
       // virtual = ligature dynamique
       virtual public void Methode(){   Console.WriteLine("MaClasse.Methode"); }
    }
   class Derivee : MaClasse
    {
       // override car méthode virtuelle avec même signature dans l'ascendance
       override public void Methode(){   Console.WriteLine("Derivee.Methode"); }
    }

   class Programme
    {
        static void Main()
        {
              MaClasse m=new MaClasse();
              Derivee d=new Derivee();
              m.Methode(); // m.MaClasse.Methode()
              d.Methode();

              m=d;          // m désigne en réalité un objet de type Derivee
              m.Methode();  // grâce à la ligature dynamique → m.Derivee.Methode()
                            //  m.Methode() n'appelle donc pas toujours le même code!!
        }
    }
}
```

```
Sortie Console
MaClasse.Methode
Derivee.Methode
Derivee.Methode
```

6.6. Une propriété peut être surdéfinie dans la classe dérivée

Qu'elle soit virtuelle ou non, une propriété peut être surdéfinie dans la classe dérivée.

```
using System;               // Propriété à ligature statique
namespace SyntaxeCSharp
{
       class Personne
       {
              protected string _nom;
              protected string _prenom;
              public Personne(string nom, string prenom){_nom = nom;   _prenom = prenom; }
```

```
            public string Identite{      get { return (_prenom + " " + _nom); }     }
      }

      class Enseignant : Personne         // cette classe dérive de Personne
      {
            private int _section;
            public Enseignant(string nom,string prenom,int section):base(nom,prenom)
            {
                  _section = section;
            }
            public new string Identite   // new car ligature statique
            {
              get{ return String.Format("{0} section {1}",base.Identite,_section);  }
            }
      }

   class Programme
   {
       static void Main()
       {
            Enseignant e=new Enseignant("Saubion","Frédéric",27);
            Console.WriteLine(e.Identite);
            Personne p = e;
            Console.WriteLine(p.Identite);
            // propriété de Personne car lig. Stat.
       }
   }
}
```

Sortie Console
Saubion Frédéric section 27
Saubion Frédéric

```
using System;               // Propriété à ligature dynamique
namespace SyntaxeCSharp
{
      class Personne{
            ...
            virtual public string Identite  // ligature dynamique
            {
            get { return (_prenom + " " + _nom); }
            }
      }

      class Enseignant : Personne{
            ...
            override public string Identite     // override car virtuelle dans base
            {
            get{   return String.Format("{0} section {1}",base.Identite,_section); }
            }
      }

   class Programme
   {
       static void Main()
       {
            Enseignant e=new Enseignant("Saubion","Frédéric",27);
            Console.WriteLine(e.Identite);
            Personne p = e;
            Console.WriteLine(p.Identite);  // propriété de Enseignant car lig. Dyn.
       }
   }
}
```

Sortie Console
Saubion Frédéric section 27
Saubion Frédéric section 27

6.7. Classe object (System.Object)

En .NET, le type **object** (System.Object) est classe de base de tout type. Tout peut donc être converti en type **object**. Au besoin, pour les types valeur, l'opération de boxing permet de créer un objet compatible avec **object**. La classe object dispose des méthodes ci-dessous

bool Equals(object o) : (virtuelle) indique l'égalité d'objets. Pour les types valeurs, l'égalité de valeur. Pour les types référence, l'égalité de référence. Attention, ce mécanisme peut être dynamiquement substitué. On doit donc consulter la documentation du type pour connaître le rôle exact de cette méthode.

static bool Equals(object objA,object objB) : idem en version statique

Type GetType(): retourne un descripteur de type

static bool ReferenceEquals(object objA, object objB): teste si les deux références désignent le même objet. Cette méthode est non substituable. Elle réalise donc toujours ce traitement.

virtual string ToString(): retourne le nom du type sous forme de chaîne. Peut être substitué dans les classes dérivées pour décrire un objet sous forme d'un chaîne.

```
static void Main(string[] args)
{
  int i = 1;
  MaClasse m = new MaClasse();

  object Obj = i;
  Console.WriteLine(Obj.GetType().FullName);
  Console.WriteLine(Obj.ToString());

  Obj = m;
  Console.WriteLine(Obj.GetType().FullName);
  Console.WriteLine(Obj.ToString());
}
```

Sortie

```
System.Int32
1
SyntaxeCSharp.MaClasse
SyntaxeCSharp.MaClasse
```

Classe **System.Collections.ArrayList** (utilisée dans l'exemple de l'indexeur): un objet de ce type stocke des références à des variables **object.** Comme **object** est l'ancêtre (ou classe de base) de tout type .NET, tout objet (ou variable) peut être considérée comme de type **object**. Les conteneurs **ArrayList** peuvent donc stocker n'importe quel élément, voire même des éléments de types différents dans le même conteneur.

```
ArrayList  t=new ArrayList();

t.Add(new MaClasse(12));
t.Add(12L);
t.Add("toto");

foreach(object obj in t)
{
   Console.WriteLine(obj.GetType().Name);
   Console.WriteLine(obj);
}
```

6.8. Boxing/unboxing

.NET fait la distinction value_type/reference_type pour que la gestion des variables avec peu de données soit plus efficace. Pour conserver une uniformité d'utilisation de tous les types, même les variables value_type peuvent être considérées comme reference_type. L'opération qui consiste à créer un objet reference_type à partir d'une variable value_type est appelée **Boxing.** L'opération inverse permettant de retrouver une variable à partir d'un type référence est appelée **Unboxing**.

Boxing : conversion value_type vers reference_type
Unboxing : conversion inverse

Grâce au boxing, on peut considérer une variable int (qui est pourtant un type primitif) en objet compatible avec les autres classes .NET. Le boxing consiste à créer une variable sur le tas qui stocke la valeur et à obtenir une référence (compatible object) sur cet objet. L'opération de boxing a donc un coût à l'exécution. Mais ce mécanisme garantit que toute variable/objet puisse être compatible avec le type ancêtre **`object`**.

```
class Programme{
        static void Main(){
            int var = 2;
            object obj = var;                    // Boxing
            Console.WriteLine(obj.GetType().FullName);
            int var2 = (int)obj;                 //Unboxing
        }
}
```

6.9. Reconnaissance de type (runtime)

Avec .NET (pas seulement C#), les types sont décrits par des méta-données stockées dans les modules compilés. On peut donc connaître à l'exécution (runtime) beaucoup d'informations sur les objets manipulés et leurs types. Tous les types sont représentés par un descripteur de type de la classe `System.Type` qui donne par exemple : nom, taille, méthodes et signatures …
Ce mécanisme est dans l'interface de la classe **`object`**. Tout objet (et même toute variable) dispose donc d'une méthode **`GetType`** qui fournit un descripteur du type de l'objet.

```
namespace ExempleType
{
    class Personne
    {
        protected string _nom;
        protected string _prenom;
        public Personne(string nom, string prenom)
        {
            _nom = nom;
            _prenom = prenom;
        }
        public string Identite
        {
            get { return (_prenom + " " + _nom); }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Personne p = new Personne("Durand", "Jacques");
            Console.WriteLine(p.Identite);

            Type t = p.GetType();

            Console.WriteLine(t.FullName);
            Console.WriteLine("Liste des méthodes de ce type:");
            System.Reflection.MethodInfo[] liste = t.GetMethods();
            foreach(System.Reflection.MethodInfo minfo in liste)
            {
                Console.WriteLine(minfo.Name);
            }
        }
    }
}
```

Sortie

```
Jacques Durand
ExempleType.Personne
Liste des méthodes de ce
type:
get_Identite
ToString
Equals
GetHashCode
GetType
```

6.10. Exemple de synthèse sur la dérivation

✍ Cet exemple permet de vérifier si les éléments de syntaxe (en gras) sont compris.

```
using System;
namespace SyntaxeCSharp
{
    class Base
    {
     private  int _prive;
     protected double _protege;  // protected ✍
     public Base(int vI, double vD){      _prive = vI;     _protege = vD;    }

     public void A() {             Console.WriteLine("Base.A()");}
     virtual public void B() {     Console.WriteLine("Base.B()");} // virtual ✍
     public void C() {             Console.WriteLine("Base.C()");}
     }


    class Derivee : Base  // ✍
    {
        protected string _str;
        public Derivee(string vS, int vI, double vD): base(vI, vD)  // :base(__) ✍
        {
                _str = vS;   // _prive n'est pas accessible ici
                             // _protege est accessible
        }

       new public void A()          { Console.WriteLine("Derivee.A()");} //new ✍
       override public void B()     { Console.WriteLine("Derivee.B()");} //override ✍
       public void D()              { Console.WriteLine("Derivee.D()"); }
    }


    class Programme
    {
        static void Main()
        {
            Derivee d = new Derivee("abc",13, 4.5);
            d.A();
            d.B();
            d.C();
            d.D();

            Base bD = d;      // conversion Derivee -> Base
            bD.A();           // ligature statique  ✍
            bD.B();           // ligature dynamique ✍
            bD.C();
        }
    }
}
```

Sortie Console

```
Derivee.A()
Derivee.B()
Base.C()
Derivee.D()
Base.A()
Derivee.B()
Base.C()
```

6.11. Conversions de type

6.11.1. Notation des casts

Utilise la notation de cast du C/C++. La conversion dérivé vers base est toujours possible. On peut par exemple toujours stocker une référence à n'importe quel objet dans une référence de type **`object`**.

Dans l'autre sens, si la conversion est possible, elle nécessite un cast.

Cast en C# (avec notation du C)

```
class Derivee:Base {  }      // Derivee dérive de Base
...

Base b=new Derivee();        //toujours OK

Derivee d=(Derivee)b;        // si b est réellement de type Derivee OK
                             // sinon, déclenche exception System.InvalidCastException
```

6.11.2. Opérateur as

```
class Derivee:Base {  }
...
Base b;  // b peut recevoir une référence à un objet Base ou de toute classe dérivée
...
Derivee d=b as Derivee;      // si b ne désigne pas un objet compatible avec
                             // Derivee alors d ppv null, sinon d est une référence valide
```

Attention : ***compatible*** signifie que b peut désigner aussi un objet d'une classe **dérivée** de `Derivee`

6.11.3. Opérateur is

```
class Derivee:Base {  }
...

Base b;
...
if(b is Derivee)  // teste si l'objet désigné par b est compatible avec Derivee
{
       Derivee d =b as Derivee;    // d devient donc forcément non null
       ...
}
```

6.11.4. Typeof/Sizeof

Le mot-clé **`typeof`** retourne le descripteur de type pour un type donné.

```
Type t =typeof(int);
```

L'opérateur **`sizeof`** ne s'applique ***qu'à un type valeur*** et retourne la taille d'une variable de ce type

```
Console.WriteLine(sizeof(int));
```

7. Les Tableaux

Les tableaux constituent le premier type de collection (voir aussi les classes conteneurs). Attention, en C# il s'agit d'objets ref_type, c'est-à-dire gérés par référence. **Un tableau est non redimensionnable.** Si l'on souhaite plus petit ou plus grand, on doit remplacer par un nouveau tableau de taille différente. C'est pourquoi l'on préfère parfois les objets de type List<T> qui fournissent des collections redimensionnables (voir plus loin).

7.1. Tableau unidimensionnel

Déclaration des tableaux (à une dimension):

```
int[] tab;          // déclaration de référence à un tableau (ausun objet n'est créé)
```

Instanciation : car un tableau est un objet implémentant **`System.Array`**

```
int[] t1,t2;          // déclaration de références
int[] t3={2,6};       // déclaration + instanciation + initialisation
t1=new int[3];        // instanciation
t2=new int[]{1,2,3}; // instanciation + initialisation
```

Les tableaux sont compatibles avec le type **`System.Array`** `(qui est classe de base)`

```
int[] tab=null;
tab = new int[3];
Console.WriteLine(tab.GetType());
System.Array t=tab;  // t est une référence à tab
t.SetValue(3, 0);    // modifie de fait tab[0]
```

7.2. Parcours d'un tableau

`Length` = propriété fournissant le nombre d'éléments d'un tableau (toutes dimensions confondues)

```
int[] tab = new int[3];
for (int i = 0; i < tab.Length; i++)
{
       tab[i]=i+1;
       Console.WriteLine(tab[i]);
}
```

`foreach` : autre façon de parcourir (accès en lecture seule)

```
int[] tab = new int[3];
foreach (int val in tab)
{
       Console.WriteLine(val);
}
```

7.3. Tableau à plusieurs dimensions

```
int[, ,] cube = new int[3, 4, 3];   // tableau à 3 dimensions
cube[0,0,0] = 1;                    // écriture dans ce tableau
```

La propriété **cube.Length** fournit alors le nombre d'éléments = 36

```
int[, ,] cube = new int[3, 4, 3];
cube[0,0,0] = 1;
Console.WriteLine(cube.Length);           // 36
Console.WriteLine(cube.GetLength(0));     // taille dans la première dimension (3)
Console.WriteLine(cube.Rank);             // nombre de dimensions = 3
```

7.4. Tableaux déchiquetés (jagged arrays)

```
int[][] monTab;
monTab = new int[4][];      // monTab est un tableau
                            // pouvant stocker 4 tableaux d'int
monTab[0] = new int[4];     // chaque case reçoit une référence à un tableau
monTab[1] = new int[2];
monTab[2] = new int[8];
monTab[3] = new int[3];
Console.WriteLine(monTab[2][3]);    // case d'incide 3 dans le tableau monTab[2]
```

7.5. Copie de tableaux

Première version, utilise la méthode `Array.CopyTo()`

```
int[] t1 = {1,2,3};
int[] t2 = new int[t1.Length];
t1.CopyTo(t2,0);
```

Seconde version, utilise la méthode `Clone()` qui, pour les objets clonables (implémentant **`ICloneable`**), renvoie un objet copie

```
int[] t1 = {1,2,3};
int[] t2;
t2=t1.Clone() as int[];  // t1.Clone() est de type object, d'où la conversion
```

7.6. Membres de la classe Array

`Array` est une classe abstraite implémentant certaines interfaces. Seul le compilateur peut dériver cette classe pour générer des classes de tableau (int[], double[] ...).

```
namespace System
{
       public abstract class Array : ICloneable, IList,ICollection,IEnumerable
       { }
}
```

Propriétés

`public int` **`Length`** `{get;}` nombre total d'éléments du tableau, quel que soit son nombre de dimensions

`public long` **`LongLength`** `{get;}` idem mais sur 64 bits

`public int` **`Rank`** `{get;}` nombre total de dimensions du tableau

Méthodes

`public static int` **`BinarySearch<T>`**`(T[] tableau,T value)` rend la position de value dans un **`tableau trié`** unidimensionnel

`public static int` **`BinarySearch<T>`**`(T[] tableau,int index, int length, T value)` idem mais cherche dans tableau trié à partir de la position index et sur length éléments

`public static void` **`Clear`**`(Array tableau, int index, int length)` met les length éléments de tableau commençant au n° index à 0 si numériques, false si booléens, null si références

`public static void` **`Copy`**`(Array source, Array destination, int length)` copie length éléments de source dans destination

`public int` **`GetLength`**`(int i)` nombre d'éléments de la dimension n° i du tableau

`public int` **`GetLowerBound`**`(int i)` indice du 1er élément de la dimension n° i

`public int` **`GetUpperBound`**`(int i)` indice du dernier élément de la dimension n° i

`public static int` **`IndexOf<T>`**`(T[] tableau, T valeur)` rend la position de valeur dans tableau ou -1 si valeur n'est pas trouvée.

`public static void` **`Resize<T>`**`(ref T[] tableau, int n)` redimensionne tableau à n éléments. Les éléments déjà présents sont conservés.

`public static void` **`Sort<T>`**`(T[] tableau, IComparer<T> comparateur)` trie tableau selon un ordre défini par comparateur.

8. `List<T>`

La classe paramétrée en type **`List<T>`** permet de générer des tableaux (et non des listes chaînées) dont la taille peut être dynamiquement modifiée par l'ajout ou la suppression. Rappelons que les tableaux (`Array`) eux ne sont pas redimensionnables. Cette classe générique implémente l'interface **`IList<T>`**.

```
namespace System.Collections.Generic{
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,IList, ICollection, IEnumerable
{
            public List();       // tableau vide de capacité par défaut
            public List(IEnumerable<T> collection);  // tableau initialisé avec collection
            public List(int capacity);  // tableau vide de capacité fournie

            public int Capacity { get; set; }   // taille mémoire disponible
            public int Count { get; }           // nombre d'éléments
            public T this[int index] { get; set; }  // indexeur
            public void Add(T item);     // ajout en fin
            public void AddRange(IEnumerable<T> collection);

            public int BinarySearch(T item); // recherche dans collection triée
            public int BinarySearch(T item, IComparer<T> comparer); // idem
            public int BinarySearch(int index, int count,T item, IComparer<T> comparer);

            public void Clear();  // supprime tous les éléments
            public bool Contains(T item); // teste la présence de item

            public void CopyTo(T[] array);  // copie vers array
            public void CopyTo(T[] array, int arrayIndex); // ...à partir de arrayIndex
            public void CopyTo(int index, T[] array, int arrayIndex, int count);

            public bool Exists(Predicate<T> match); // teste le prédicat sur la collection

            public T Find(Predicate<T> match); //premiere occurrence qui vérifie match
            public List<T> FindAll(Predicate<T> match); // éléments qui vérifient match
            public int FindIndex(Predicate<T> match); // index du premier qui matche
            public int FindIndex(int startIndex, Predicate<T> match);
            public int FindIndex(int startIndex, int count, Predicate<T> match);
            public T FindLast(Predicate<T> match);
            public int FindLastIndex(Predicate<T> match);
            public int FindLastIndex(int startIndex, Predicate<T> match);
            public int FindLastIndex(int startIndex, int count, Predicate<T> match);
            public void ForEach(Action<T> action); // execute Action<T> sur tous
            public List<T>.Enumerator GetEnumerator(); // obtient un Enumerator
            public List<T> GetRange(int index, int count); // retourne une partie

            public int IndexOf(T item);   // indice de item ou -1 si pas item
            public int IndexOf(T item, int index);  // indice de item à partir de index
            public int IndexOf(T item, int index, int count);

            public void Insert(int index, T item);  // insère item à l'indice index

            public void InsertRange(int index, IEnumerable<T> collection);

            public int LastIndexOf(T item);
            public int LastIndexOf(T item, int index);
            public int LastIndexOf(T item, int index, int count);

            public bool Remove(T item);
            public int RemoveAll(Predicate<T> match);
            public void RemoveAt(int index);
            public void RemoveRange(int index, int count);

            public void Reverse();  // inverse l'ordre des éléments
            public void Reverse(int index, int count); // idem dans l'intervalle

            public void Sort();  // trie
            public void Sort(IComparer<T> comparer); // trie selon comparateur fourni
            public void Sort(int index, int count, IComparer<T> comparer);
            public T[] ToArray();  // fournit un tableau avec les items
      }
```

9. Interfaces et classes abstraites

9.1. Interfaces (contrat)

Une ***interface*** est un ensemble de prototypes de méthodes, de propriétés ou d'indexeurs qui forme un ***contrat***. Celà définit des attentes pour les classes qui veulent se conformer à ce contrat. Une ***classe qui implémente*** une interface doit fournir ***toutes les méthodes prévues*** par l'interface, d'où la notion de contrat. En contrepartie, dès lors qu'une classe implémente une interface, on a la garantie qu'elle contient un comportement minimal, celui prévu par l'interface.

Une interface contient un ensemble de prototypes de méthodes:

- pas de qualificatif public/private/protected pour les membres
- pas de données membres
- pas de qualificatif virtual

Ci-dessous une interface avec 2 méthodes et deux classes qui implémentent cette interface.

```
using System;
namespace SyntaxeCSharp
{
    interface IMonBesoin  // ∾   CONTRAT
    {
        void A();
        void B();
    }

       class MaClasse:IMonBesoin          // implémente l'interface (remplit le contrat)
       {
        public void A()      { Console.WriteLine("MaClasse.A()");}
        public void B() { Console.WriteLine("MaClasse.B()"); }
       }

       class AutreClasse : IMonBesoin     // implémente l'interface ∾
       {
        public void A() { Console.WriteLine("AutreClasse.A()"); }
        public void B() { Console.WriteLine("AutreClasse.B()"); }
       }

       class Programme
       {
        static void Main()
        {
            IMonBesoin ib=new MaClasse();
            ib.A();
            ib.B();

            ib=new AutreClasse();
            ib.A();
            ib.B();
       }
    }
}
```

Sortie Console

```
MaClasse.A()
MaClasse.B()
AutreClasse.A()
AutreClasse.B()
```

Un objet implémentant l'interface **`IMonBesoin`** peut être référencé au moyen d'une référence de type **`IMonBesoin.`** **C**ette référence ne permet d'accéder qu'aux méthodes décrites dans l'interface.

Une interface peut déclarer des propriétés. Il convient de préciser néanmoins s'il s'agit d'un accès en lecture ou en écriture. Cela signifie que la classe qui implémentera l'interface devra fournir les propriétés requises, avec l'accès requis.

Déclaration de propriétés dans une interface

```
using System;
namespace SyntaxeCSharp
{
    interface IValeur
    {
         int Valeur      // Valeur devra être une propriété en lecture
         {
                get;
         }
         bool Presence  // Presence devra être une propriété en lecture/écriture
         {
                get; set;
         }
    }
}
```

9.1.1. Interfaces standard .NET

Le framework .NET fournit un ensemble de classes mais aussi un ensemble d'interfaces.

Interface **`System.Collections.IEnumerable`** : une classe qui implémente cette interface (généralement un conteneur) fournit un moyen de parcourir ses instances. L'interface **`IEnumerable`** standardise le parcours des conteneurs .NET. En particulier, un objet qui implémente `IEnumerable` peut être parcouru au moyen de l'instruction **`foreach`**. (cf. Section sur les conteneurs)

Interface **`System.ICloneable`** : une classe qui implémente cette interface fournit un moyen de créer une copie d'un objet de la classe (objet clone).

9.1.2. Tester qu'un objet implémente une interface (opérateurs is et as)

On peut tester, à l'exécution, si un objet donné implémente ou non une interface donnée. On utilise là encore les opérateurs **`is`** et **`as`** du langage qui s'appuient sur le CTS pour vérifier la compatibilité de type.

```
using System;
namespace SyntaxeCSharp
{
    interface IMonBesoin
    {
        void A();
        void B();
    }
    class MaClasse:IMonBesoin
    {
        public void A()      { Console.WriteLine("MaClasse.A()");}
        public void B()      { Console.WriteLine("MaClasse.B()"); }
    }

    class Programme
    {
        static void Main()
        {
            object obj = new MaClasse();

            if (obj is IMonBesoin)  // est-ce qu'objet implémente cette interface ?
            {
                Console.WriteLine("Oui, cet objet implémente l'interface IMonBesoin");
            }
```

```
            IMonBesoin ib = obj as IMonBesoin;

            if (ib == null)
            {
                Console.WriteLine("Non, obj n'implémente pas l'interface IMonBesoin");
            }
            else
            {
                Console.WriteLine("Oui, obj implémente l'interface IMonBesoin");
            }
        }
    }
}
```

9.2. Classes abstraites

Une interface ne contient aucun code ni aucune donnée. C'est juste un contrat.

Une ***classe abstraite*** est une classe dont une ou plusieurs méthodes *n'ont pas d'implémentation.* Celles qui n'ont pas d'implémentation sont marquées **`abstract`**. En revanche, certaines méthodes ***concrètes*** peuvent être définies au niveau d'une classe abstraite. De même, on peut définir des attributs dans une classe abstraite.

Puisqu'une classe abstraite ne définit pas toutes ses méthodes, ***il n'est pas possible d'instancier*** une classe abstraite. En revanche, une classe abstraite *sert de base dans le cadre de la dérivation*. Toutes les méthodes et attributs définis dans une classe abstraite peuvent servir aux classes qui en dérivent.

Ci-dessous, la classe abstraite décrit un système qu'on peut démarrer et arrêter. Tout système a un nom. Ce traitement élémentaire est décrit directement dans la classe abstraite. Toute classe dérivant de `SystemeAbstrait` hérite de ce nom et de son accesseur.

Toutes les méthodes abstraites sont virtuelles (lig. dynamique).

```
using System;

namespace SyntaxeCSharp
{
    abstract class SystemeAbstrait                //classe abstraite
    {
        private string _nom;        // attribut (pas possible dans une interface)

        public SystemeAbstrait(string Nom)        // constructeur S1)
        {
            _nom=Nom;
        }
        public string GetNom()  // méthode concrète (pas possible dans une interface)
        {
            return _nom;
        }
        abstract public void Start();  // méthode abstraite (donc virtuelle)
        abstract public void Stop();
        abstract public bool IsStarted();
    }

    class SystemV1 : SystemeAbstrait      // cette classe dérive de SystemeAbstrait
    {
        private bool _on;

        public SystemV1(string Nom):base(Nom)   // utilise S1)
        {
            _on = false;
        }
```

```
        override public void Start()  // fournit une implémentation concrète
        {
            _on = true;
        }
        override public void Stop()
        {
            _on = false;
        }
        override public bool IsStarted()
        {
            return _on;
        }
    }

    class Programme
    {
        static void Main()
        {
              SystemeAbstrait sys = new SystemV1("Version 1");
              Console.WriteLine(sys.GetNom());
              sys.Start();
        }
    }
}
```

10. Exceptions

De nombreuses fonctions C# sont susceptibles de lever des exceptions, c'est-à-dire de ***signaler des comportements anormaux***. Les exceptions C# utilisent les mots clé suivants

`throw new Exception("Ca va mal!");` : déclenche une exception en associant un objet
`try{ }` : cherche à détecter les exceptions dans ce bloc
`catch(Exception e) { }` : capture les objets de type `Exception` levés dans le bloc `try`

Important : une exception (**`throw`** …) non interceptée par un bloc **`catch`** provoque l'arrêt du programme.

La gestion d'une exception se fait selon le schéma suivant :

```
try
{      // code susceptible de déclencher une exception de type Exception
       // une des fonctions comporte throw new _____;
       ...
}
catch (Exception e) //intercepte une exception de type Exception
{
       //traiter l'exception e
}
//instruction suivante
```

Si aucune exception ne se produit dans `try{ }`, l'exécution se poursuit alors à `instruction suivante`. Sinon, une exception conduit au corps de la clause **`catch,`** puis à `instruction suivante`.

Notons les points suivants :

• ci-dessus, `e` est un objet de type **`Exception`** ***ou dérivé***. On peut être plus précis en utilisant des types tels que `IndexOutOfRangeException`, `FormatException`, `SystemException`, etc… qui dérivent de `Exception`:

```
try
{
	//code susceptible de générer des exceptions
}
catch (IndexOutOfRangeException e1)
{
	//traiter une exception de type  IndexOutOfRangeException
}
catch (FormatException e2)
{
	//traiter une exception de type  FormatException
}

//instruction suivante
```

• On peut ajouter aux clauses try/catch, une clause **`finally`** :

```
try
{
	//code susceptible de générer une exception
}
catch (Exception e)
{
	//traiter l'exception e
}
finally
{
	//code toujours exécuté (après try ou catch selon le cas)
}
instruction suivante
```

Qu'il y ait exception ou pas, le code de la clause `finally` sera toujours exécuté. Dans la clause `catch`, on peut ne pas vouloir utiliser l'objet `Exception` disponible. Au lieu d'écrire catch (`Exception` e){..}, on écrit alors `catch(Exception){...}` ou plus simplement `catch {...}`.

La classe **`Exception`** a une propriété **`Message`** qui est un message détaillant l'erreur qui s'est produite.

La classe `Exception` a aussi une méthode **`ToString`** qui rend une chaîne de caractères indiquant le type de l'exception ainsi que la valeur de la propriété **`Message`**. On pourra ainsi écrire :

```
catch (Exception ex)
{
	Console.WriteLine("L'erreur suivante s'est produite : {0}", ex.ToString());
	...
}// fin catch
```

11. Délégation (delegate) et Evénements (event)

Le framework .NET fournit un mécanisme pour pouvoir déléguer des appels de fonctions. Ce mécanisme présente des similitudes avec la notion ***de pointeur de fonction en C/C++.*** Le pictogramme signifie "pas évident".

11.1. Délégation (≈ pointeur de fonction)

Le mot-clé **`delegate`** permet de créer des *classes* particulières appelées *classes de délégation*. Dans l'exemple ci-dessous, la classe de délégation est **`Deleg`**. Une instance de cette classe est appelée *objet délégué* ou *délégué*.

Une classe de délégation dérive implicitement de la classe **`System.MulticastDelegate`**. Un objet délégué doit assurer des appels à des méthodes qu'on lui a indiquées. Il *délègue* des traitements.

Lors de la définition de la classe de délégation, on mentionne la *signature* des méthodes prises en charge par la délégation. Une instance de **Deleg** permettra d'appeler ici des fonctions avec un paramètre double et retournant un int.

Ci-après, **Round** est un objet de la classe **Deleg**. On peut utiliser l'objet **Round** comme s'il s'agissait d'une fonction. A l'instanciation, on précise quelle méthode utiliser pour la délégation. Toutefois, on peut changer dynamiquement la méthode utilisée par l'objet délégué à l'aide de deux opérateurs

L'opérateur -= retire de la délégation
L'opérateur += ajoute une fonction à la délégation

```
using System;
namespace SyntaxeCSharp
{
          class Program
           {
               delegate int Deleg(double x);

               static int Floor(double x) {  return (int)x;  }

               static int Ceil(double  x)  { return (int)Math.Ceiling((double) x);}

               static void Main(string[] args)
               {
                   int val;
                   Deleg Round=new Deleg(Floor);  // délègue initialement à Floor
                   // l'objet Round délègue à une autre fonction le traitement de l'arrondi

                   // l'objet Round s'utilise comme une fonction!
                   val = Round(3.2);             // utilise Floor
                   Console.WriteLine(val);       // affiche 3

                   Round -= new Deleg(Floor);    // ne délègue plus à Floor
                   Round += new Deleg(Ceil);     // délègue désormais à Ceil

                   val = Round(3.2);             // utilise Ceil
                   Console.WriteLine(val);       // affiche 4
                }
      }
}
```

Sortie Console
3
4

Simplification d'écriture à partir de C# 2

```
    class Program
    {
              delegate int Deleg(double x);
              static int Floor(double x) { return (int)x; }
              static int Ceil(double x) { return (int)Math.Ceiling((double)x); }

              static void Main(string[] args)
              {
              int val;
              Deleg Round = Floor;         // crée un objet et délègue à Floor
              val = Round(3.2);            // appelle Floor

              Console.WriteLine(val);
              Round -= Floor;       // ne délègue plus à Floor
              Round += Ceil;        // délègue à Ceil
              val = Round(3.2);     // appelle Ceil
              Console.WriteLine(val);
      }
```

11.2. Evénéments

Beaucoup utilisé en programmation Windows

Il est facile d'être client d'un événement (courant en prog. Windows), mais plus difficile d'écrire une classe propriétaire (fournisseur) d'événement.

On peut voir les événements comme *une forme particulière de délégation*. Un événement peut déléguer un traitement à une ou plusieurs fonctions (comme pour une délégation). **Mais seul l'objet propriétaire de l'événement peut déclencher l'événement, c'est-à-dire appeler les fonctions de la délégation.** Par contre, n'importe quel utilisateur peut ***abonner/désabonner*** une fonction à l'événement, c'est-à-dire ajouter ou retirer une fonction à la liste de délégation. Les fonctions qu'on attache sont appelées des fonctions „Call Back".

Dans l'exemple ci-dessous, la classe **`Emetteur`** est propriétaire d'un événement appelé **`EmetteurEvent`**. Cet événement est déclenchable seulement par l'objet **`Emetteur`**. Lorsque l'événement est déclenché, les abonnés (les fonctions de la liste de délégation) à cet événement sont appelés.

A noter, un événement est toujours déclaré à partir d'un type de délégation. Le type de délégation décrit la signature des méthodes pouvant s'abonner à l'événement. Par exemple, l'événement ci-dessous ne déclenche que des méthodes sans paramètre ni retour.

```
class Emetteur       // classe propriétaire de l'événement EmetteurEvent
{
        public delegate void CallBack();          // signature des abonnés à l'événement

        public event CallBack EmetteurEvent;      // événement

        public void DeclencheEvenement()
        {
              // déclenche un événement = appel de tous les délégués (fonctions abonnées)
              // noter : EmetteurEvent vaut null si aucun abonné
            if(EmetteurEvent!=null) EmetteurEvent();  // s'utilise comme une fonction
        }
}

class Program
{
        public static void CB1(){ Console.WriteLine("Call Back 1"); }
        public static void CB2(){ Console.WriteLine("Call Back 2"); }

        static void Main(string[] args)
        {
            Emetteur e = new Emetteur();

            e.EmetteurEvent += CB1;  // abonne CB1 à l'événement e.EmetteurEvent
            e.EmetteurEvent += CB2;  // abonne CB2 à l'événement e.EmetteurEvent

            // e.EmetteurEvent(); pas possible ici ! Seul e peut déclencher l'événement.

            //  En revanche, on peut appeler e.DeclencheEvenement()
            e.DeclencheEvenement();   //➶ e.EmetteurEvent provoque l'appel de CB1()->CB2()

        }
}
```

Sortie Console
```
Call Back 1
Call Back 2
```

11.3. Exploitation standard des événements dans .NET

L'utilisation des événements, notamment dans les classes `System.Windows.Forms`, suit toujours un peu le même principe. Tous les événements du système (souris, clavier, …) sont vus comme des événements gérés par des classes (Button, TextBox, ...). ***Un gestionnaire d'événement*** consiste donc à abonner une fonction CallBack à un événement d'un contrôle ou d'une fenêtre Windows.

Dans ce schéma là, les événements ont généralement une signature (définie par une délégation) assez standard. Les événements appellent des méthodes ayant le plus souvent deux paramètres. Le premier est une référence à l'objet qui déclenche l'événement. Le second est une référence à un objet censé transporter des données associées à l'événement.

Si l'on reprend ce schéma, pour une application console :

La classe *Sujet* est propriétaire de l'événement `EvtSup20`.

La classe *Donnees* décrit les données transmises par l'événement (dérive de *System.EventArgs*).

System.EventHandler est une délégation standard fournie par .NET

```
public delegate void EventHandler(object sender,EventArgs e)
```

`sender` = objet déclencheur de l'événement
`e` = arguments, c-à-d objet transportant des données complémentaires

```
using System;        //Exemple d'événement utilisant une signature .NET standard EventHandler
namespace SyntaxeCSharp
{

    class Donnees : EventArgs      // classe des données associées à l'événement
    {
        private string _message;
        public Donnees(string msg){_message=msg;}
        public string Msg(){ return _message;   }
    }


   class Sujet              // classe propriétaire de l'événement  EvtSup20
   {
        public event EventHandler EvtSup20;   // événement qui utilise une signature standard

        private int _var;

        public Sujet(int v){  _var = v; }  // constr.

        public int Var             // propriété Var en lecture/écriture
        {
              get { return _var; }

              set           // seule l'écriture peut déclencher l'événement EvtSup20
              {
                _var = value;  // modifie  _var
                if(_var>20 &&  EvtSup20!=null)
                {
                      EvtSup20(this, new Donnees("Attention Var >20"));
                }
              }
        }
   }
```

```
  class Program
  {
       // cette fonction statique possède la signature EventHandler
        public static void CB(object sender, EventArgs args)
        {
            Console.WriteLine("Fonction Call Back ");
            Sujet s = sender as Sujet;  // l'émetteur est normalement un objet Sujet
            if (s != null)
            {
                Console.WriteLine("Pour l'émetteur Var = {0}", s.Var);
                Donnees d = args as Donnees;
                if (d != null)
                {
                    Console.WriteLine("Message de l'émetteur = {0}", d.Msg());
                }
            }
        }


        public static void Main()
        {

            Sujet leSujetA = new Sujet(12);    // deux objets Sujet
            Sujet leSujetB = new Sujet(8);

            leSujetA.EvtSup20 += CB;       // CB est abonnée à l'événement EvtSup20
            leSujetB.EvtSup20 += CB;       // des deux objets

            leSujetA.Var = 17;  //ne déclenche pas d'événement (car <20)
            leSujetB.Var = 13;  //ne déclenche pas d'événement (car <20)

            leSujetB.Var = 45;  // déclenche un événement (l'émetteur est leSujetB)
                                // donc CB est appelée, avec 2 paramètres

            leSujetA.Var = 24;  // déclenche un événement (l'émetteur est leSujetA)

        }
  }
}
```

Sortie Console

```
Fonction Call Back
Pour l'émetteur Var = 45
Message de l'émetteur = Attention Var >20
Fonction Call Back
Pour l'émetteur Var = 24
Message de l'émetteur = Attention Var >20
```

12. Les classes conteneurs

Le framework .NET fournit des classes pour stocker des données. Nous décrivons sommairement quelques classes d'usage fréquent.

12.1. Classe string

12.1.1. Les membres de la classe string

Le type string du C# est un alias du type **`System.String`** de .NET qui permet de gérer des chaînes de caractère. Il s'agit d'un type référence mais dont l'usage ressemble à celui des types valeurs. Pour le type `string`, l'affectation fait en réalité une copie des valeurs (et non une copie de références). L'autre caractéristique est que les objets de cette classe sont *immutables*. Celà signifie que les objets gardent la même valeur du début à la fin de leur vie. Toutes les opérations visant à changer la valeur de l'objet retourneront en réalité un nouvel objet.

Quelques membres de la classe **`System.String`**

Constructeurs

`string(Char[] tabC)` Initialise une nouvelle instance de la classe String à la valeur indiquée par un tableau de caractères Unicode.
`string(Char, Int32 repete)` Initialise une nouvelle instance de la classe String à la valeur indiquée par un caractère Unicode et répété un certain nombre de fois.

`string(Char[], Int32 index, Int32 count)` Initialise une nouvelle instance de la classe String à la valeur indiquée par un tableau de caractères Unicode, un point de départ pour le caractère dans ce tableau et une longueur.

```
class Program
    {
        static void Main(string[] args)
        {
            string s1 = "abc";
            string s2 = new string('*', 7);
            string s3 = new string(new char[] { 'e', 'f', 'g' });

            Console.WriteLine(s1);  // abc
            Console.WriteLine(s2);  // *******
            Console.WriteLine(s3);  // efg
        }
    }
```

Champs publics

Nom	Description
Empty	**Représente la chaîne vide. Ce champ est en lecture seule.**

Propriétés publiques

Nom	Description
indexeur **[]**	Obtient le caractère à une position spécifiée dans cette instance.
Length	Obtient le nombre de caractères dans cette instance.

Méthodes publiques

public bool **EndsWith**(string value)	rend vrai si la chaîne se termine par value
public bool **StartsWith**(string value)	rend vrai si la chaîne commence par value

```
public virtual bool Equals(object obj) rend vrai si la chaînes est égale à obj –
            équivalent chaîne==obj

public int IndexOf(string value, int startIndex) rend la première position dans la chaîne
                  de la chaîne value - la recherche commence à partir du
                  caractère n° startIndex

public int IndexOf(char value, int startIndex) idem mais pour le caractère value

public string Insert(int startIndex, string value) insère la chaîne value dans chaîne en
                  position startIndex

public string Remove(int startIndex,int count) supprime count caractères à partir de
                  startIndex

public static string Join(string separator,string[] value) méthode de classe - rend une
                  chaîne de caractères, résultat de la concaténation des
      valeurs du tableau value avec le séparateur separator

public int LastIndexOf(string value, int startIndex, int count) idem indexOf mais rend la
            dernière position au lieu de la première

public int LastIndexOf(char value, int startIndex,int count)

public string Replace(char oldChar, char newChar) rend une chaîne copie de la chaîne
            courante où le caractère oldChar a été remplacé par le caractère
            newChar

public string[] Split(char[] separator) la chaîne est vue comme une suite de champs
      séparés par les caractères présents dans le tableau separator. Le
      résultat est le tableau de ces champs

public string Substring(int startIndex, int length) sous-chaîne de la chaîne courante
            commençant à la position startIndex et ayant length caractères

public string ToLower() rend la chaîne courante en minuscules

public string ToUpper() rend la chaîne courante en majuscules

public string Trim(char[] jeu ) supprime toutes les occurences d'un jeu de caractères
```

```
static void Main(string[] args)
{
            string s1 = "chaîne de caractères";

            Console.WriteLine(s1.Contains("car"));  // True
            Console.WriteLine(s1.StartsWith("ch")); // True
            Console.WriteLine(s1.IndexOf('c'));     // 0
            Console.WriteLine(s1.LastIndexOf('c')); // 14

            Console.WriteLine(s1.Substring(4, 7));  // ne de c


}
```

Remarque (StringBuilder) : pour les chaînes dont la taille doit évoluer, .Net frounit une autre classe appelée System.Text.StringBuilder.

12.1.2. Le formattage chaîne de types numériques

La représentation des valeurs numériques par des chaînes de caractères est décrite dans l'interface **`IFormattable`**, c'est à dire une classe disposant d'une méthode `ToString()` avec deux paramètres

```
interface IFormattable
{      string ToString(string format,IFormatProvider fprovider)
}
```

`format` : chaîne fournissant des instructions de formattage constituée d'une lettre avec un digit optionnel
`fprovider` : objet permettant de réaliser les instructions de formattage selon une culture donnée

Le programme ci-dessous affiche un prix avec au plus deux chiffres après la virgule. Dans la culture en-GB (english Great Britain) , le prix est affiché en livres. Le prix est affiché en dollars dans la culture en-US (english US).

```
static void Main(string[] args)
{
     double prix = 3.5;
     CultureInfo culture = CultureInfo.GetCultureInfo("en-GB");
     Console.WriteLine(prix.ToString("C2", culture));
     culture = CultureInfo.GetCultureInfo("en-US");
     Console.WriteLine(prix.ToString("C2", culture));
}
```

Sortie

```
£3.50
$3.50
```

Chaîne de formattage des types numériques (Lettre + digit)

G ou g = général F = fixed point N = fixed point avec séparateur de groupe

E = notation exp P = pourcentage X ou x =hexadécimal

```
class Program
    {
        static void Main(string[] args)
        {
           double val1 = 1.23456;
            double val2 = 12340;
            double val3 = 0.0001234;
            double val4 = 1203.01234;
            double val5 = 0.1234;

            CultureInfo fr = CultureInfo.GetCultureInfo("fr-FR");

            Console.WriteLine("----------------");
            // G ou g : Général
            Console.WriteLine("G");
            Console.WriteLine(val1.ToString("G", fr));
            Console.WriteLine(val2.ToString("G", fr));
            Console.WriteLine(val3.ToString("G", fr));
            Console.WriteLine("----------------");
            // "G3" = limité à 3 digits au total (passe en notation
            // exp au besoin)
            Console.WriteLine(val1.ToString("G3", fr));
            Console.WriteLine(val2.ToString("G3", fr));

            Console.WriteLine("----------------");
            // F : Fixed point
            // F2 arrondit à 2 décimales
            Console.WriteLine("F");
            Console.WriteLine(val1.ToString("F2", fr));
            Console.WriteLine(val3.ToString("F2", fr));
```

```
----------------
G
1,23456
12340
0,0001234
----------------
1,23
1,23E+04
----------------
F
1,23
0,00
----------------
N
1,23
12 340,00
1 203,01
----------------
E
1,203012E+003
1,2030E+003
----------------
P
12,34 %
12,3 %
----------------
X
1E
Appuyez sur une
touche pour
continuer...
```

```
            Console.WriteLine("----------------");
            // N : Fixed point + séparateur de groupe
            // N2 limite à deux décimales
            Console.WriteLine("N");
            Console.WriteLine(val1.ToString("N2", fr));
            Console.WriteLine(val2.ToString("N2", fr));
            Console.WriteLine(val4.ToString("N2", fr));

            Console.WriteLine("----------------");
            Console.WriteLine("E");
            // E : notation exp (6 digits par défaut après virgule)
            // E4 : 4 digits après la virgule
            Console.WriteLine(val4.ToString("E", fr));
            Console.WriteLine(val4.ToString("E4", fr));

            Console.WriteLine("----------------");
            // P : pourcentage
            Console.WriteLine("P");
            Console.WriteLine(val5.ToString("P", fr));
            Console.WriteLine(val5.ToString("P1", fr));

            Console.WriteLine("----------------");
            // X ou x :hexadécimal
            Console.WriteLine("X");
            Console.WriteLine(30.ToString("X"));

        }
    }
```

12.2. Les conteneurs paramétrés en type (List<T> ...).

Le C# permet la définition de classes paramétrées en type. Celà signifie que certains types peuvent être choisis au moment de l'instanciation. C'est le cas par exemple de List<Ty> qui permet d'engendrer des objets-tableaux d'éléments de type Ty.

Sans rentrer dans les détails, on peut fournir un exemple simple de classe paramétrée en type `Paire<Ty>`. On représente ici des paires de valeurs. Le type des valeurs de la paire est un paramètre de type.

```
namespace ConsoleGenerique
{
    class Paire<T>          // classe paramétrée en type : T est un type quelconque
    {
        private T _premier;  // les deux attributs sont du même type
        private T _second;

        public Paire(T prem, T sec)   // constructeur
        {
            _premier = prem;
            _second = sec;
        }

        public override string ToString()
        {
            return "(" + _premier.ToString() + ";" + _second.ToString() + ")";
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
              // paire d'entiers
              Paire<int> p1= new Paire<int>(7,2);
              Console.WriteLine(p1.ToString());

              // paire de chaînes
              Paire<string> p2 = new Paire<string>("premier", "second");
              Console.WriteLine(p2.ToString());

              Paire<double> pd1=new Paire<double>(1.2,2.3);
              Paire<double> pd2=new Paire<double>(3.5,4.8);

              // paire de paires de double (puisque Paire<double> est aussi un type)
              Paire<Paire<double>> p3;
              p3 = new Paire<Paire<double>>(pd1,pd2);
              Console.WriteLine(p3.ToString());
        }
    }
}
```

Sortie console

```
(7;2)
(premier,second)
((1,2;2,3);(3,5;4,8))
```

12.2.1. Quelques interfaces de conteneurs : IEnumerable, ICollection, IList, IDictionary

Les classes conteneurs sont toutes les classes fournies par le framework .NET qui permettent de stocker des données. Le premier type de conteneur utilisé est le tableau (type dérivé de **`Array`**). D'autres classes permettent de stocker des données. Les conteneurs se distinguent par le type de service qu'ils rendent.

Interface `IEnumerable<T>`

Le comportement minimum d'une structure de donnée est d'implémenter **`IEnumerable`**. C'est à dire que la structure de donnée peut fournir un énumérateur permettant de parcourir les données.

Dès lors qu'un type implémente **`IEnumerable`**, on peut utiliser l'instruction **`foreach`** pour parcourir ses éléments.

```
    public interface IEnumerable{
              // Retourne: Objet System.Collections.IEnumerator pouvant
              // être utilisé pour itérer au sein de la collection.
              IEnumerator GetEnumerator();
    }


   public interface IEnumerator{
        // Fournit élément courant.
        object Current { get; }

        // Avance l'énumérateur à l'élément suivant de la collection.
        // Retourne: true si l'énumérateur a pu avancer
        bool MoveNext();

       // Remet l'énumérateur à sa position initiale (avant premier élément)
        void Reset();
    }
```

Interface `ICollection<T>`

Plus riche que l'interface `IEnumerable`, l'interface **`ICollection`** est décrite ci-dessous

```
public namespace System.Collections.Generic
{
    interface ICollection<T> : IEnumerable<T>, IEnumerable
    {
        int Count { get; }          //  Nombre d'éléments

        bool IsReadOnly { get; }    //  Est en lecture seule

        void Add(T item);           //  Ajoute un élément item

        void Clear();               //  Supprime tous les éléments

        bool Contains(T item);      //  Est-ce que item est présent?

        void CopyTo(T[] array, int arrayIndex);   // Copie dans un tableau

        bool Remove(T item);        // Supprime l'élément item
    }
```

Interface IList<T>

Plus riche que l'interface `ICollection`, l'interface **`IList<T>`** permet d'atteindre des éléments par un index (comme un tableau). La classe générique **`List<T>`** implémente cette interface.

```
namespace System.Collections.Generic
{
    public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
    {
        T this[int index] { get; set; }           // indexeur

        int IndexOf(T item);                      // donne l'index de item ou -1 sinon

        void Insert(int index, T item);           // insertion de item à l'index

        void RemoveAt(int index);                 // supprime l'élément à index
    }
}
```

Interface IDictionary<TKey,TValue>

C'est l'interface pour les structures de données (clé,valeur). On appelle parfois cela des tableaux associatifs. On peut accéder aux élements en fournissant non pas un indice, mais une clé.

```
namespace System.Collections.Generic
{
    public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
       IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
    {
        ICollection<TKey> Keys { get; }           // fournit la collection des clés

        ICollection<TValue> Values { get; }       // les valeurs

        TValue this[TKey key] { get; set; }       // valeur de clé key en lecture/écriture

        void Add(TKey key, TValue value);         // ajoute une valeur pour la clé key

        bool ContainsKey(TKey key);               // recherche présence de la clé

        bool Remove(TKey key);                    // supprime la clé

        bool TryGetValue(TKey key, out TValue value);
    }
}
```

12.2.2. LinkedList<T>

Les objets de type **LinkedList<T>** sont des ***listes doublement chaînées*** d'éléments de type T.

Cette classe n'implémente pas `IList<T>` puisque l'on ne peut pas accéder directement à un élément.

Les éléments sont stockés dans des noeuds dont la structure est décrite ci-dessous.

```
namespace System.Collections.Generic
{
    public sealed class LinkedListNode<T>
    {
        public LinkedListNode(T value);
        public LinkedList<T> List { get; }
        public LinkedListNode<T> Next { get; }
        public LinkedListNode<T> Previous { get; }
        public T Value { get; set; }
    }
}
```

Les méthodes de la classe **LinkedList<T>** sont décrites ci-dessous

```
namespace System.Collections.Generic
{
    public class LinkedList<T> : ICollection<T>, IEnumerable<T>,
              ICollection, IEnumerable, ISerializable,IDeserializationCallback
    {
        public LinkedList(); // crée Liste vide
        public LinkedList(IEnumerable<T> collection);
        protected LinkedList(SerializationInfo info, StreamingContext context);

        public int Count { get; }  // nombre d'éléments
        public LinkedListNode<T> First { get; } // premier et dernier noeud
        public LinkedListNode<T> Last { get; }

        public void AddAfter(LinkedListNode<T> node,
                             LinkedListNode<T> newNode);
        public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);
        public void AddBefore(LinkedListNode<T> node,
                             LinkedListNode<T> newNode);
        public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);
        public void AddFirst(LinkedListNode<T> node);
        public LinkedListNode<T> AddFirst(T value);
        public void AddLast(LinkedListNode<T> node);
        public LinkedListNode<T> AddLast(T value);

        public void Clear(); // supprime tous les noeuds
        public bool Contains(T value); // indique présence de value

        public void CopyTo(T[] array, int index);
        public LinkedListNode<T> Find(T value);
        public LinkedListNode<T> FindLast(T value);
        public LinkedList<T>.Enumerator GetEnumerator();

        public void Remove(LinkedListNode<T> node);
        public bool Remove(T value);
        public void RemoveFirst();  // supprime le premier noeud
        public void RemoveLast();   // supprime le dernier
    }
}
```

12.2.3. Classe Dictionary<TKey,TValue> (tableau associatif (clé,valeur))

Classe générique de tableaux associatifs implémentant l'interface `IDictionary<key,value>`.

```
namespace System.Collections.Generic
{
    public class Dictionary<TKey, TValue> : IDictionary<TKey, TValue>,
       ICollection<KeyValuePair<TKey, TValue>>,
       IEnumerable<KeyValuePair<TKey, TValue>>,
       IDictionary, ICollection, IEnumerable, Iserializable,
       IDeserializationCallback
    {
        public Dictionary();
        public Dictionary(IDictionary<TKey, TValue> dictionary);

        public int Count { get; }
        public Dictionary<TKey, TValue>.KeyCollection Keys { get; }
        public Dictionary<TKey, TValue>.ValueCollection Values { get; }

        public TValue this[TKey key] { get; set; }
        public void Add(TKey key, TValue value);
        public void Clear();
        public bool ContainsKey(TKey key);
        public Dictionary<TKey, TValue>.Enumerator GetEnumerator();
        public bool Remove(TKey key);
        public bool TryGetValue(TKey key, out TValue value);

       }
}
```

13. Classe Math

La classe System.Math est une classe statique, puisqu'elle ne fournit que des fonctions membres statiques.

Utilisation :

```
      double d=Math.Abs(-3.4);   // d<-3.4
             d=Math.Pow(3,7);    // d<-3^7

      double x=Math.PI;          // x ← 3.14159
```

```
namespace System{

    public static class Math
    {
        public const double E = 2.71828;
        public const double PI = 3.14159;

        public static decimal Abs(decimal value);  // valeur absolue
        public static double Abs(double value);  //dispo. avec value de type float,int,...
              public static float Abs(float value);
              public static int Abs(int value);
              public static long Abs(long value);
              public static sbyte Abs(sbyte value);
              public static short Abs(short value);

        public static long BigMul(int a, int b); // produit 64bits ← 32bits * 32bits

        public static decimal Ceiling(decimal d); //  plus petit entier >=d
        public static double Ceiling(double a);    //  plus petit entier >=a
        public static decimal Floor(decimal d);   // plus grand entier <=d
        public static double Floor(double d);    // plus grand entier <=d
        public static decimal Truncate(decimal d); // supprime les décimales
```

```
        public static double Truncate(double d);   // supprime les décimales
        public static decimal Round(decimal d);  // entier le plus proche
        public static double Round(double a);     // entier le plus proche
               public static decimal Round(decimal d, int decimals);
               public static decimal Round(decimal d, MidpointRounding mode);
               public static double Round(double value, int digits);
               public static double Round(double value, MidpointRounding mode);
               public static decimal Round(decimal d, int decimals, MidpointRounding mode);
               public static double Round(double value, int digits, MidpointRounding mode);

        public static double Acos(double d);
        public static double Asin(double d);
        public static double Atan(double d);
        public static double Atan2(double y, double x);
        public static double Cos(double d);
        public static double Cosh(double value);
        public static double Sin(double a);
        public static double Sinh(double value);
        public static double Tan(double a);
        public static double Tanh(double value);

        public static double Exp(double d);   // e^d = exponentiel(d)
        public static double Log(double d);   // Log(d)
        public static double Log(double a, double newBase);
        public static double Log10(double d);
        public static double Pow(double x, double y);  // x^y
        public static double Sqrt(double d);   // d^(1/2)=racine(d)

        public static byte Max(byte val1, byte val2);
              public static decimal Max(decimal val1, decimal val2);
              public static double Max(double val1, double val2);
              public static float Max(float val1, float val2);
              public static int Max(int val1, int val2);
              public static long Max(long val1, long val2);
              public static sbyte Max(sbyte val1, sbyte val2);
              public static short Max(short val1, short val2);
              public static uint Max(uint val1, uint val2);
              public static ulong Max(ulong val1, ulong val2);
              public static ushort Max(ushort val1, ushort val2);

        public static byte Min(byte val1, byte val2);
               public static decimal Min(decimal val1, decimal val2);
               public static double Min(double val1, double val2);
               public static float Min(float val1, float val2);
               public static int Min(int val1, int val2);
               public static long Min(long val1, long val2);
               public static sbyte Min(sbyte val1, sbyte val2);
               public static short Min(short val1, short val2);
               public static uint Min(uint val1, uint val2);
               public static ulong Min(ulong val1, ulong val2);
               public static ushort Min(ushort val1, ushort val2);

        public static int Sign(decimal value);  // retourne 1 (positif) 0 (nul) -1 (négatif)
                     public static int Sign(double value);
                     public static int Sign(float value);
                     public static int Sign(int value);
                     public static int Sign(long value);
                     public static int Sign(sbyte value);
                     public static int Sign(short value);

        // division Euclidienne = retourne le quotient et fournit le reste en sortie
        //            int reste;
        //            int quotient = Math.DivRem(23, 7, out reste);  // quotient=3 reste=2
        public static int DivRem(int a, int b, out int result);
        public static long DivRem(long a, long b, out long result);
        public static double IEEERemainder(double x, double y);
    }
}
```

13.1. Table des matières