

# COURS DE C#

ORIENTÉ OBJET



- Les espaces de noms
- L'encapsulation
- Notions de classe
- Les propriétés
- Les indexeurs
- Surcharge d'opérateurs
- L'héritage
- Le polymorphisme
- Les classes abstraites
- Les classes statiques
- Les interfaces
- Construction et destruction d'objets
- Les exceptions

- Les délégués
- Les événements
- Les génériques

## C# - ORIENTÉ OBJET



# LES ESPACES DE NOMS

C# - ORIENTÉ OBJET

- Introduction
- Le mot-clé « namespace »
- Le mot-clé « using »
- Règles de fonctionnement
  - Portée
  - Dissimulation
  - Répétitions
  - Imbrication
- Les « alias »

## LES ESPACES DE NOMS

# INTRODUCTION

Le Framework .Net est structuré en espaces de noms. On peut les comparer à un système de fichiers.

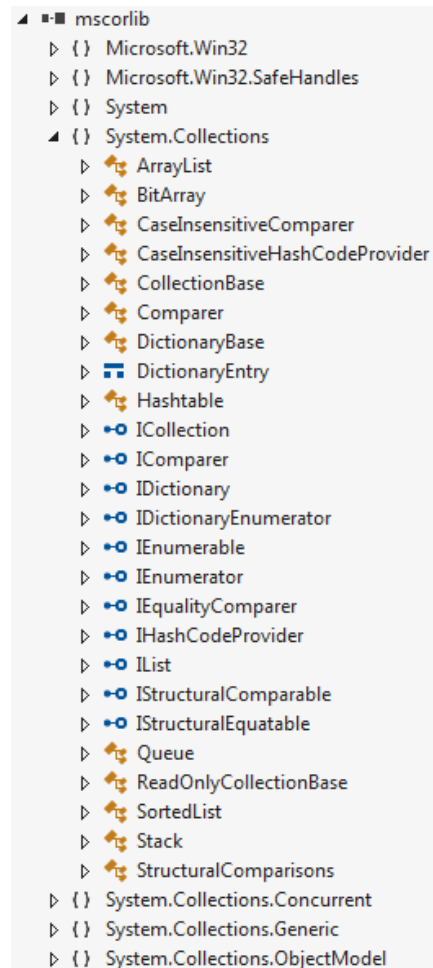
Où chaque répertoire serait un conteneur qui peut contenir d'autres répertoires ou des fichiers et où chaque fichier serait un type de donnée.

Pour rappel, en .Net nous avons plusieurs façon de déclarer un type de données.

En effet, nous avons vu que pouvions déclarer de nouveaux types de variables au travers de structures et d'énumérations.

Mais nous le pouvons, également, au travers de classes, d'interfaces ou de délégués que nous verrons dans cette partie.

Cependant à la différence du système de fichiers ou d'autres langages, comme le Java par exemple, cette structure est une structure logique et non physique.



```
mscorlib
├── Microsoft.Win32
├── Microsoft.Win32.SafeHandles
├── System
├── System.Collections
│   ├── ArrayList
│   ├── BitArray
│   ├── CaseInsensitiveComparer
│   ├── CaseInsensitiveHashCodeProvider
│   ├── CollectionBase
│   ├── Comparer
│   ├── DictionaryBase
│   ├── DictionaryEntry
│   ├── Hashtable
│   ├── ICollection
│   ├── IComparer
│   ├── IDictionary
│   ├── IDictionaryEnumerator
│   ├── IEnumerable
│   ├── IEnumerator
│   ├── IEquityComparer
│   ├── IHashCodeProvider
│   ├── IList
│   ├── IStructuralComparable
│   ├── IStructuralEquatable
│   ├── Queue
│   ├── ReadOnlyCollectionBase
│   ├── SortedList
│   ├── Stack
│   └── StructuralComparisons
├── System.Collections.Concurrent
├── System.Collections.Generic
└── System.Collections.ObjectModel
```

# LE MOT-CLÉ « NAMESPACE »

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Namespace1.MaStructure ma = new Namespace1.MaStructure();
            Console.WriteLine(ma.ToString());
            Console.ReadLine();
        }
    }
}

namespace Namespace1
{
    public struct MaStructure
    {
        public override string ToString()
        {
            return "MaStructure";
        }
    }
}
```

En .Net, ce qui détermine l'espace de noms dans lequel va se trouver notre type, c'est la valeur définie après le mot-clé « namespace ».

Lorsque nous définissons un nouveau type celui porte un nom complet « Namespace.NomDuType » et si nous voulions utiliser ce nouveau type, nous devons, par défaut, utiliser le nom complet du type.

# LE MOT-CLÉ « USING »

Cependant, nous avons la possibilité d'utiliser le mot-clé « using » afin de simplifier l'écriture de notre code.

Le mot-clé « using » importe les types d'un espace de noms défini.

Le fait d'ajouter « using System; » à notre code, nous donne accès au type « Console » sans pour autant devoir écrire :  
« System.Console ».



Attention à la notion d'utilisation de « dll »  
et de références

```
using System;
using Namespace1;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaStructure ma = new MaStructure();
            Console.WriteLine(ma.ToString());
            Console.ReadLine();
        }
    }
}

namespace Namespace1
{
    public struct MaStructure
    {
        public override string ToString()
        {
            return "MaStructure";
        }
    }
}
```

# RÈGLES DE FONCTIONNEMENT - PORTÉE

```
namespace Namespace1
{
    public class MaClasse
    {
    }
}

namespace Namespace1.Sub1
{
    public struct MaStructure
    {
        MaClasse ma = new MaClasse();
    }
}
```

Cependant, il y a quatre règles qui régissent le fonctionnement des espaces de noms.

- La portée
- La dissimulation
- La répétition
- L'imbrication

La portée d'un espace de noms est descendante, ce qui veut dire que les espaces de noms enfants connaissent les types définis dans les parents sans faire de « using ».



L'inverse n'est pas vrai



# RÈGLES DE FONCTIONNEMENT - DISSIMULATION

Lorsqu'un même nom de type apparaît deux fois dans la portée.

Cela implique, qu'il y a dissimulation, que le type défini dans le sous espace de noms dissimule le type défini plus haut dans l'arborescence.

Si nous voulions utiliser le type défini plus haut nous devrions utiliser le nom complet ou utiliser les **alias** que nous verrons plus tard.



# RÈGLES DE FONCTIONNEMENT - RÉPÉTITION

```
namespace Namespace1
{
    public class MaClasse
    {
    }
}

namespace Namespace1
{
    public class MaClasse2
    {
    }
}
```

Nous pouvons répéter la déclaration d'un espace de nom à partir du moment où les types, qu'ils définissent, n'entrent pas en conflit.

Nous ne pourrions pas, par exemple, déclarer deux fois la classe « MaClasse » dans le même espace de noms.

Excepté dans un cas particulier que nous verrons plus tard.

# RÈGLES DE FONCTIONNEMENT - IMBRICATION

Bien que très rarement utilisée (voir jamais), cette règle nous permet déclarer un espace de noms dans un autre.

Ceci dit, dans la bonne pratique, chaque type sera un fichier et chaque sous espace de noms sera un répertoire.



Dans tous les cas, lors de la compilation, chaque type sera réécrit avec son nom complet.

```
namespace Namespace1
{
    //Nom Complet : Namespace1.MaClasse
    public class MaClasse
    {
    }

    namespace Sub1
    {
        //Nom Complet : Namespace1.Sub1.MaClasse2
        public class MaClasse2
        {
        }
    }
}
```

# LES ALIAS

```
using Microsoft.Excel;
using Microsoft.Word;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Document
        }
    }
}

namespace Microsoft.Excel
{
    public class Document
    {
    }
}

namespace Microsoft.Word
{
    public class Document
    {
    }
}
```

Dans certains cas nous aurons à régler des problèmes d'ambiguïté dans l'utilisation de type de données.

Pour cela nous utiliserons des alias dans les directives « using » afin d'éviter d'utiliser le nom complet à chaque fois.

Par la suite, lors de la déclaration de nos variables, nous utiliserons ces alias pour les différencier.

```
using E = Microsoft.Excel;
using W = Microsoft.Word;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            E.Document ExcelDocument = new E.Document();
            W.Document WordDocument = new W.Document();
        }
    }
}
```



# L'ENCAPSULATION

C# - ORIENTÉ OBJET

- Définition
- Les niveaux d'accessibilité
- Niveaux par défaut
- Cas particulier
- Limitations

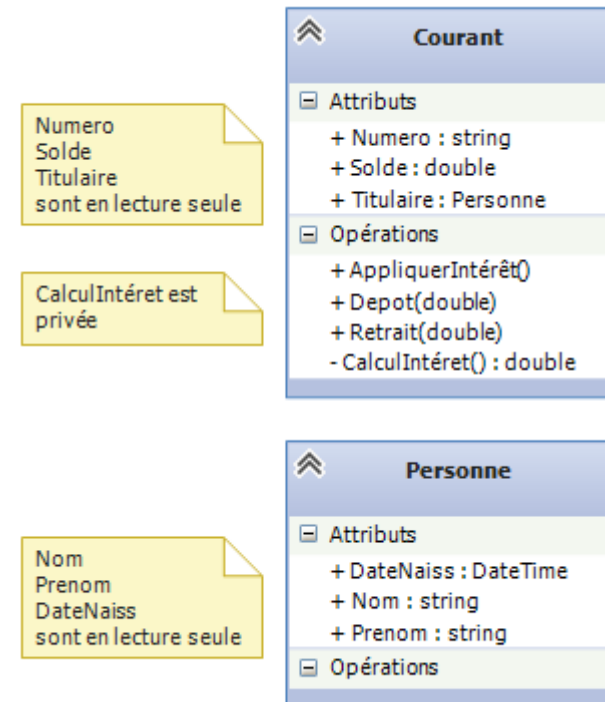
## L'ENCAPSULATION

# DÉFINITION

L'encapsulation est un principe fondamental au sein de l'orienté objet.

Elle va nous permettre de protéger et de contrôler l'accès à nos types afin de garantir l'intégrité des informations qu'ils contiennent.

Pour cela, nous devons donner à nos types et à leurs membres, un modificateur d'accès qui limitera l'accès à ceux-ci.



# LES NIVEAUX D'ACCESSIBILITÉ

Accessibilité	Signification
public	L'accès n'est pas limité.
protected	L'accès est restreint à la classe conteneur ou à ses types dérivés. (Héritage)
internal	L'accès est restreint à l'assembly courant.
protected internal	L'accès est restreint à l'assembly courant ou à aux types dérivées de la classe conteneur. (Héritage)
private	L'accès est restreint à la classe conteneur.

En C#, nous retrouvons 5 niveaux d'accessibilité.

Ceci dit, il y a quelques principes à connaître :

- Un seul niveau d'accessibilité est autorisé par membre ou type.
- Ces niveaux ne peuvent pas être appliqués aux espaces de noms.
- Si aucun niveau d'accès n'est déclaré, un niveau par défaut est utilisé.



# NIVEAUX PAR DÉFAUT

Selon le contexte dans lequel une déclaration de membre est effectuée, seules certaines accessibilités déclarées sont autorisées.

Les types imbriqués, qui sont membres d'autres types, peuvent disposer des niveaux d'accessibilités en fonction du contexte.

Membres de	Niveau par défaut	Niveaux Autorisés
enum	public	public
class	private	public protected internal protected internal private
interface	public	public
struct	private	public internal private

# CAS PARTICULIER

```
namespace CoursCSharpFondements
{
    //internal par défaut
    enum Valeurs { Valeur1, Valeur2, Valeur3 }
    public enum Texte { Texte1, Texte2, Texte3 }

    internal class Program
    {
        static void Main(string[] args)
        {

        }
    }

    public class MaClasse
    {
        public void MaMethode();
    }
}
```

Les types de niveau supérieur, qui sont déclarés directement dans un namespace, ne peuvent disposer que d'un niveau d'accessibilité « internal » ou « public ».

De plus, si aucun n'est spécifié, c'est le niveau « internal » qui est utilisé.

# LIMITATIONS

Lorsque nous spécifions un type dans une déclaration (d'une méthode, d'une variable, d'une propriété, etc.), nous devons vérifier le niveau d'accessibilité du type en question.

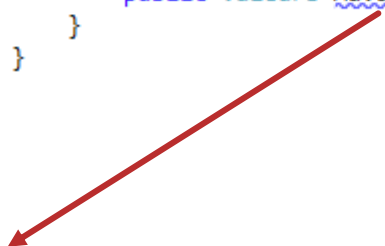
Exemples :

Si je déclare une variable « **public** » d'un type T, il faut que ce type T soit « **public** » également.

La liste complète des limitations est disponible à l'adresse :  
[http://msdn.microsoft.com/fr-fr/library/cx03xt0t\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/cx03xt0t(v=vs.110).aspx)

```
namespace CoursCSharpFondements
{
    //internal par défaut
    enum Valeurs { Valeur1, Valeur2, Valeur3 }

    public class MaClasse
    {
        public Valeurs MaValeur;
    }
}
```



Accessibilité incohérente : le type de champ 'CoursCSharpFondements.Valeurs' est moins accessible que le champ 'CoursCSharpFondements.MaClasse.MaValeur'



# NOTIONS DE CLASSE

C# - ORIENTÉ OBJET

- Introduction
- Classe partielle
- Instanciation

## NOTIONS DE CLASSE

# INTRODUCTION

Une classe est essentiellement un modèle à partir duquel il est possible de créer des objets de type référence.

Chacune pourra contenir des variables, des propriétés, des indexeurs, des méthodes, des constructeurs, un destructeur, des délégués, des événements, voir même des structures et/ou d'autres classes.

Une classe à pour but de définir quelles sont les données et fonctionnalités que chaque instanciation pourra contenir.

Pour définir une classe, nous utiliserons le mot clé « **class** ».

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ToolBox.Databases
{
    public class Command
    {
        private readonly Dictionary<string, object> _Parameters;

        public bool IsStoredProcedure { get; private set; }
        public string Query { get; private set; }

        public KeyValuePair<string, object>[] Parameters
        {
            get { return _Parameters.ToArray(); }
        }

        public Command(string Query) : this (Query, false)
        {
        }

        public Command(string Query, bool IsStoredProcedure)
        {
            _Parameters = new Dictionary<string, object>();
            this.Query = Query;
            this.IsStoredProcedure = IsStoredProcedure;
        }

        public void AddParameter(string ParameterName, object Value)
        {
            _Parameters.Add(ParameterName, Value);
        }
    }
}
```

# CLASSE PARTIELLE

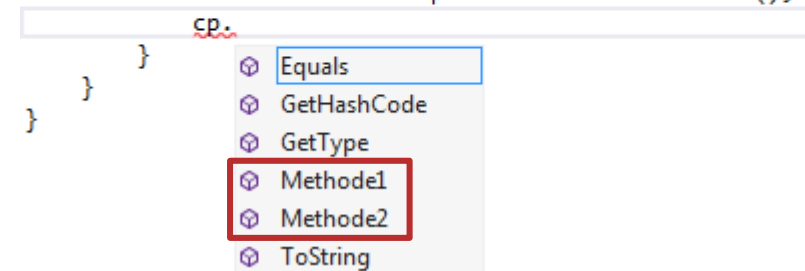
```
namespace CoursCSharpFondements
{
    //du fichier ClassePartielle1.cs
    public partial class ClassePartielle
    {
        public void Methode1()
        {
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    //du fichier ClassePartielle2.cs
    public partial class ClassePartielle
    {
        public void Methode2()
        {
        }
    }
}
```

La programmation C# nous permet de déclarer une même classe sur plusieurs fichiers, en utilisant le mot-clé « partial ».

C'est à la compilation que le compilateur se charge alors de rassembler les fichiers en une classe unique.

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            ClassePartielle cp = new ClassePartielle();
            cp.
        }
    }
}
```



# INSTANCIATION

Toute variable de type référence doit être instanciée, création d'une instance de la classe en mémoire.

Pour ce faire, à chaque fois que nous souhaiterons créer une nouvelle instantiation en mémoire, nous devrons utiliser le mot-clé « **new** ».



Ne pas perdre de vue le fonctionnement  
des types références

```
namespace Namespace1
{
    public class MaClasse
    {
    }
}

namespace Namespace1.Sub1
{
    public struct MaStructure
    {
        MaClasse ma = new MaClasse();
    }
}
```





# LES PROPRIÉTÉS

C# - ORIENTÉ OBJET

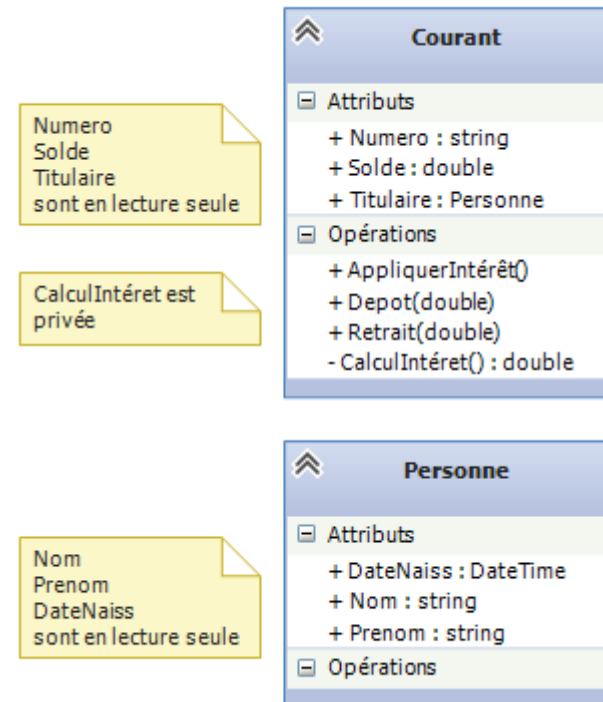
- Introduction
- Déclaration
- Contrôle de l'accès aux valeurs
- Raccourcis

## LES PROPRIÉTÉS

# INTRODUCTION

Les propriétés vont nous permettre de contrôler l'accès et le contenu de nos variables de classe grâce aux accesseurs (get) et les mutateurs (set) en assurant leurs intégrités.

Les accesseurs et les mutateurs seront des blocs d'instructions ou nous pourrons ajouter des tests et des actions lors de la récupération et de l'assignation d'une valeur.



# DÉCLARATION

```
namespace CoursCSharpFondements
{
    class MaClasse
    {
        private int _MaVariable;

        public int MaVariable
        {
            //Accesseur (get)
            get
            {
                return _MaVariable;
            }
            //Mutateur (set)
            set
            {
                _MaVariable = value;
            }
        }
    }
}
```

Une propriété se déclare comme une variable, à l'exception qu'elle sera pourvue d'accolade ouvrante et fermante.

À l'intérieur nous retrouverons deux éléments optionnels :

- L'accesseur (get) se chargera de retourner une valeur valide avec le mot-clé return.
- Le mutateur (set) se chargera de contrôler et de valider une valeur et la mettre dans la variable.

Ces deux éléments héritent, par défaut, du niveau d'accessibilité de la propriété.

# CONTRÔLE DE L'ACCÈS AUX VALEURS

Grâce aux propriétés nous pouvons également contrôler l'accès à une propriété.  
Il nous sera parfois intéressant de rendre une propriété « readonly » ou « writeonly ».

Pour cela, il existe deux solutions :

- En omettant le get (writeonly) ou le set (readonly)
- En modifiant le niveau d'accessibilité du get et/ou du set

## Solution 1

```
namespace CoursCSharpFondements
{
    class MaClasse
    {
        private int _MaVariable;

        public int MaVariable
        {
            get { return _MaVariable; }
        }
    }
}
```

## Solution 2

```
namespace CoursCSharpFondements
{
    class MaClasse
    {
        private int _MaVariable;

        public int MaVariable
        {
            get { return _MaVariable; }
            private set { _MaVariable = value; }
        }
    }
}
```

# RACCOURCIS

Dans certains cas nous devons spécifier des propriétés qui n'auront aucun test ou aucune action spécifique à réaliser.

Dans ce cas de figure, nous pourrons écrire une version simplifiée de la propriété.



C# fait la distinction entre les variables membres et les propriétés.

```
namespace CoursCSharpFondements
{
    class MaClasse
    {
        public string Nom { get; set; }
        public string NomLectureSeul { get; private set; }
        public string NomEcritureSeul { private get; set; }
    }
}
```



# EXERCICES

# EXERCICES (MAXIMUM 20')

1. Créer une classe « Personne » qui devra implémenter
  - Les propriétés publiques :
    - Nom (string)
    - Prenom (string)
    - DateNaiss (DateTime) - date de naissance
2. Créer une classe « Courant » permettant la gestion d'un compte courant qui devra implémenter
  - Les propriétés publiques :
    - Numéro (string)
    - Solde (double) - Lecture seule
    - LigneDeCredit (double) - représentant la limite négative du compte strictement supérieur ou égale à 0
    - Titulaire (Personne)
  - Les méthodes publiques :
    - void Retrait(double Montant)
    - void Depot(double Montant)





# LES INDEXEURS

C# - ORIENTÉ OBJET

- Déclaration
- Surcharge

## LES INDEXEURS

# DÉCLARATION

Les indexeurs permettent aux instances d'une classe ou d'une structure d'être indexées exactement comme des tableaux.

Les indexeurs sont des propriétés qui reçoivent un paramètre, le type et le nom de ce paramètre sont spécifiés entre des crochets « [ ] » ouvrant et fermant.

Ces crochets quant à eux sont précédés du mot-clé « this ».

```
namespace CoursCSharpFondements
{
    class MaClasse
    {
        Dictionary<Guid, Personne> Locataires = new Dictionary<Guid, Personne>();

        public Personne this[Guid Key]
        {
            get
            {
                Personne p;
                Locataires.TryGetValue(Key, out p);
                return p;
            }
            set
            {
                Locataires[Key] = value;
            }
        }
    }
}
```

# SURCHARGE

```
public class MaClasse
{
    private Dictionary<Guid, Personne> _Locataires;
    public Dictionary<Guid, Personne> Locataires {
        get { return _Locataires = _Locataires ?? new Dictionary<Guid, Personne>(); }
    }

    public Personne this[Guid Key] {
        get {
            Personne p;
            Locataires.TryGetValue(Key, out p);
            return p;
        }
        set {
            Locataires[Key] = value;
        }
    }

    public Personne this[int index] {
        get {
            return Locataires.Values.ToArray()[index];
        }
    }
}
```

La classe ou la structure n'ont pas de nombre limité d'indexeurs.

Nous pouvons donc en déclarer plusieurs tant que ceux-ci reçoivent un type d'index différent.

```
class Program
{
    static void Main(string[] args)
    {
        MaClasse mc = new MaClasse();
        Guid ID = Guid.NewGuid();

        Personne p = new Personne();
        mc.Locataires.Add(ID, p);

        Personne p2 = mc[ID];
    }
}
```



# EXERCICES

# EXERCICES (MAXIMUM 15')

- I. Créer une classe « Banque » qui gérera les comptes de la banque  
Cette classe devra implémenter :
  - Les propriétés :
    - Comptes (KeyValuePair<string, Courant>[ ]) – Lecture seule
    - Nom (string) - Nom de la banque
  - Un indexeur retournant un compte sur base de son numéro.
  - Les méthodes :
    - void Ajouter(Courant compte)
    - void Supprimer(string Numero)



# SURCHARGE D'OPÉRATEURS

C# - ORIENTÉ OBJET

- Déclaration
- Surcharges d'opérateurs autorisées
- Cela reste des méthodes

## SURCHARGE D'OPÉRATEURS



# DÉCLARATION

Cette notion vient du C++ et tout comme lui, C# nous permet de surcharger des opérateurs à utiliser sur vos propres classes.

Cela permet à nos type de données de paraître aussi naturels et d'être aussi logiques à utiliser qu'un type de données fondamental.

Les surcharges d'opérateur se font au travers de méthodes « static ».

De plus, certains opérateurs devront être surchargés par paire :

- L'opérateur d'égalité (==) avec l'opérateur d'inégalité (!=).
- L'opérateurs plus petit (<) avec le plus petit ou égale(<=).
- L'opérateur plus grand (>) avec l'opérateur plus grand ou égale (>=).

```
public class Panier
{
    private readonly List<Fruit> _Fruits = new List<Fruit>();

    public static Panier operator +(Panier p1, Panier p2)
    {
        Panier result = new Panier();

        foreach (Fruit f in p1._Fruits)
        {
            result.AddFruit(f);
        }

        foreach (Fruit f in p2._Fruits)
        {
            result.AddFruit(f);
        }

        return result;
    }

    public void AddFruit(Fruit f)
    {
        _Fruits.Add(f);
    }
}
```

# SURCHARGES D'OPÉRATEUR AUTORISÉES

Type d'opérateur	Opérateurs
Opérateurs Unaires	<b>+, -, !, ~, ++, --, true, false</b>
Opérateurs binaires	<b>+, -, *, /, %, &amp;,  , ^, &lt;&lt;, &gt;&gt;, ==, !=, &gt;, &lt;, &gt;=, &lt;=</b>

C# n'autorise qu'un nombre restreint de surcharge d'opérateur, contrairement au langage C++.

# CELA RESTE DES MÉTHODES

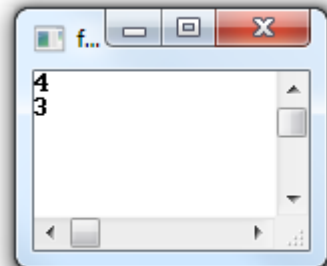
Cependant, faire de la surcharge d'opérateur revient à faire de la surcharge de méthodes.

```
static void Main(string[] args)
{
    Panier p1 = new Panier();
    Panier p2 = new Panier();

    p1.AddFruit(new Fruit());
    p1.AddFruit(new Fruit());
    p2.AddFruit(new Fruit());
    p2.AddFruit(new Fruit());

    int count = p1 + p2;
    int count2 = p1 + new Fruit();

    Console.WriteLine(count);
    Console.WriteLine(count2);
    Console.ReadLine();
}
```



```
namespace CoursCSharpFondements
{
    public class Panier
    {
        private readonly List<Fruit> _Fruits = new List<Fruit>();

        public static int operator +(Panier p1, Panier p2)
        {
            return p1._Fruits.Count + p2._Fruits.Count;
        }

        public static int operator +(Panier p1, Fruit f)
        {
            return p1._Fruits.Count + 1;
        }

        public void AddFruit(Fruit f)
        {
            _Fruits.Add(f);
        }
    }
}
```



# EXERCICES

## EXERCICES (MAXIMUM 20')

1. Surcharger l'opérateur « + » de la classe « Courant » afin qu'il retourne la somme, de type double, des soldes. Cependant, les soldes négatifs ne doivent pas être pris en compte.
2. Ajouter une méthode « AvoirDesComptes » à la classe « Banque » recevant en paramètre le titulaire (Personne) qui calculera les avoirs de tous ses comptes en utilisant l'opérateur « + ».



# L'HÉRITAGE

C# - ORIENTÉ OBJET

- Héritage simple
- Classe de base universelle
- Héritage et surcharge de méthodes
- Redéfinition de méthodes
- Dissimulation de méthodes
- Danger de la dissimulation
- Appel des versions de base
- Interdire l'héritage d'une classe

## L'HÉRITAGE

# HÉRITAGE SIMPLE

C# prend en charge l'héritage simple, ce qui signifie qu'une classe ne peut hériter que d'une et d'une seule autre classe.

La nouvelle classe (la classe dérivée) gagne ensuite tous les membres non privés de la classe de base en plus de tous ses propres membres qu'elle définit pour elle-même.

L'héritage se définit pas le deux points ( : )



Ceci dit rien n'empêche à une classe d'hériter d'une classe qui hérite elle-même d'une autre classe.

```
public class Rectangle
{
    protected int longueur, largeur;
    private int x;
}

public class Carre : Rectangle
{
    //hérite de longueur et de largeur
    //mais n'hérite pas de x car privé à la classe rectangle
}
```



# CLASSE DE BASE UNIVERSELLE

```
namespace System
{
    ...public class Object
    {
        ...public Object();

        ...public virtual bool Equals(object obj);
        ...public static bool Equals(object objA, object objB);
        ...public virtual int GetHashCode();
        ...public Type GetType();
        ...protected object MemberwiseClone();
        ...public static bool ReferenceEquals(object objA, object objB);
        ...public virtual string ToString();
    }
}
```

```
namespace CoursCSharpFondements
{
    public class MaClasse
    {
    }
}
```

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaClasse m1 = new MaClasse();
            m1.
        }
    }
}
```

- Equals
- GetHashCode
- GetType
- ToString

Lorsque qu'aucun héritage n'est défini, la classe héritera cependant de la classe « System.Object », appelée en C# la classe de base universelle.

Ce qui implique que la classe « System.Object » est la mère de tout élément en C# et que par conséquent nous obtenons toujours les méthodes non privées qu'elle définit.

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaClasse.
        }
    }
}
```

- Equals
- ReferenceEquals

# HÉRITAGE ET SURCHARGE DE MÉTHODES

Dans le cadre de l'héritage, rien ne nous empêche de surcharger une méthode héritée, tant que les règles de la surcharge sont respectées.

Bien entendu, la surcharge ne sera disponible que pour le type dérivé (« ClasseDerivee »).

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            ClasseDerivee cd = new ClasseDerivee();
            cd.MaMethode(
        }
    }
}
```

▲ 1 sur 2 ▼ void ClasseDeBase.MaMethode()

```
namespace CoursCSharpFondements
{
    public class ClasseDeBase
    {
        public void MaMethode()
        {
        }
    }

    public class ClasseDerivee : ClasseDeBase
    {
        public void MaMethode(int x)
        {
        }
    }
}
```

# REDÉFINITION DE MÉTHODES

Que nous voulions redéfinir ou dissimuler une méthode nous voulons, dans les deux cas, modifier le comportement de notre méthode par rapport à la classe de base.

Dans le cadre de la redéfinition, nous utiliserons les mots clés « **virtual** » et « **override** ».

Le mot-clé « **virtual** » va signaler que la méthode de la classe de base est redéfinissable.

Pour redéfinir la méthode dans la classe dérivée, nous emploierons le mot-clé « **override** ».



En héritage, le mot clé « **virtual** » n'est pas obligatoire et doit rester un choix!!

```
public class ClasseDeBase
{
    //Déclaration de la méthode maMethode
    public virtual string maMethode()
    {
        return "Classe de Base";
    }
}

public class ClasseDerivee : ClasseDeBase
{
    //Redéfinition de la méthode maMethode
    public override string maMethode()
    {
        return "Classe dérivée";
    }
}
```

# DISSIMULATION DE MÉTHODES

```
public class ClasseDeBase
{
    //Déclaration de la méthode uneMethode
    public string uneMethode()
    {
        return "Classe de Base";
    }
}

public class ClasseDerivee : ClasseDeBase
{
    //Dissimulation de la méthode uneMethode
    public new string uneMethode()
    {
        return "Classe dérivée";
    }
}
```

Dans le cadre de la dissimulation de méthodes, il n'y a que dans la classe dérivée que nous allons ajouter le mot-clé « **new** » à la déclaration de la méthode.

Le mot-clé « **new** » signifie que nous masquons explicitement un membre hérité d'une classe de base.

Masquer un membre hérité signifie que la version dérivée du membre remplace la version de classe de base.

# DANGER DE LA DISSIMULATION

Il est toujours préférable de redéfinir une méthode plutôt que de la dissimuler.

En effet, Il existe lors de la dissimulation d'une méthode, lorsque nous l'invoquons, que ce soit la méthode de la classe de base qui soit exécutée et non la méthode dissimulée.

Cependant il arrive, également, que nous n'ayons pas d'autre choix que de dissimuler une méthode.

Ex : si la classe de base a été développée par autrui et que nous ne pouvons pas modifier le code source.

```
namespace CoursCSharpFondements
{
    public class ClasseDeBase
    {
        public virtual void MethodeRedefinie()
        {
            Console.WriteLine("MethodeRedefinie de la classe de base");
        }

        public void MethodeDissimulee()
        {
            Console.WriteLine("MethodeDissimulee de la classe de base");
        }
    }

    public class ClasseDerivee : ClasseDeBase
    {
        public override void MethodeRedefinie()
        {
            Console.WriteLine("MethodeRedefinie de la classe dérivée");
        }

        public new void MethodeDissimulee()
        {
            Console.WriteLine("MethodeDissimulee de la classe dérivée");
        }
    }
}
```

# DANGER DE LA DISSIMULATION



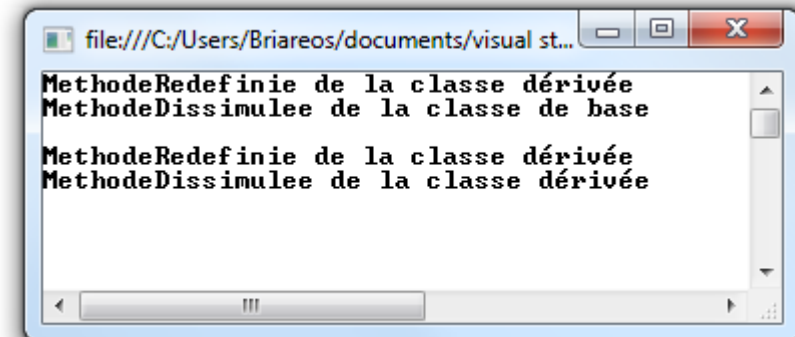
Notion de Polymorphisme

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            ClasseDeBase cdb = new ClasseDerivee();
            cdb.MethodeRedefinie();
            cdb.MethodeDissimulee();

            Console.WriteLine();

            ClasseDerivee cd = new ClasseDerivee();
            cd.MethodeRedefinie();
            cd.MethodeDissimulee();

            Console.ReadLine();
        }
    }
}
```



# APPEL DES VERSIONS DE BASE

Le mot clé « **base** » sert à accéder aux membres de la classe de base à partir d'une classe dérivée.

```
public class ClasseDeBase
{
    //Déclaration de la méthode maMethode
    public virtual string maMethode()
    {
        return "Classe de Base";
    }
}

public class ClasseDerivee : ClasseDeBase
{
    //Redéfinition de la méthode maMethode
    public override string maMethode()
    {
        //appel de la méthode de la classe de base
        string s = base.maMethode();

        return string.Format("{0} {1}", s, "Classe dérivée");
    }
}
```

# INTERDIRE L'HÉRITAGE D'UNE CLASSE

```
public class ClasseDeBase
{
    //Implémentation
}

public sealed class ClasseDerivee : ClasseDeBase
{
    //Implémentation
}
```

Si pour quelque raison que ce soit nous souhaitons empêcher notre classe d'être héritée par une classe dérivée, nous pouvons utiliser le mot clé « **sealed** ».

La classe « ClasseDerivee » étant scellée, aucune classe ne pourra en hériter.

Nous pouvons également utiliser le modificateur « **sealed** » sur une méthode ou une propriété qui substitue une méthode ou une propriété virtuelle dans une classe de base.

Cela nous permettra d'autoriser les classes à hériter de notre classe et d'empêcher de substituer des méthodes ou des propriétés virtuelles spécifiques.





# EXERCICES

# EXERCICES (MAXIMUM 40')

1. Créer une classe « Epargne » permettant la gestion d'un carnet d'épargne qui devra implémenter
  - Les propriétés publiques :
    - Numéro (string)
    - Solde (double) - Lecture seule
    - DateDernierRetrait (DateTime) - représentant la date du dernier retrait sur le carnet
    - Titulaire (Personne)
  - Les méthodes publiques :
    - void Retrait(double Montant)
    - void Depot(double Montant)
2. Définir la classe « Compte » reprenant les parties commune aux classes « Courant » et « Epargne » en utilisant les concepts d'héritage, de redéfinition de méthodes et si besoin, de surcharge de méthodes et d'encapsulation.  
Attention le niveau d'accessibilité du mutateur de la propriété Solde doit rester « private ».



# LE POLYMORPHISME

C# - ORIENTÉ OBJET

- Introduction
- Polymorphisme = casting implicite + casting explicite

## LE POLYMORPHISME

# INTRODUCTION

Par héritage, une classe peut être utilisée comme plusieurs types.

Elle peut être utilisée comme son propre type, tout type de base ou tout type d'interface (si elle implémente des interfaces).

C'est ce qu'on appelle le polymorphisme.



En C#, tout type est polymorphe.  
Puisque tout les types hérite de « System.Object ».

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Carre c = new Carre();
            Rectangle r = new Carre();
            Forme f = new Carre();
            object o = new Carre();

            Console.ReadLine();
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    public class Forme
    {
        public string Color { get; set; }
    }

    public class Rectangle : Forme
    {
        public virtual double Longueur { get; set; }
        public virtual double Largeur { get; set; }
    }

    public class Carre : Rectangle
    {
        public override double Largeur
        {
            get { return base.Largeur; }
            set { base.Longueur = base.Largeur = value; }
        }

        public override double Longueur
        {
            get { return base.Longueur; }
            set { base.Largeur = base.Longueur = value; }
        }
    }
}
```

# POLYMORPHISME = CASTING IMPLICITE + CASTING EXPLICITE

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Carre c = new Carre();
            Rectangle r = new Carre(); //Casting implicite
            Forme f = new Carre(); //Casting implicite
            object o = new Carre(); //Casting implicite

            o.|

            Console.ReadLine();
        }
    }
}
```

Lorsque nous déclarons un carré comme étant un élément de type « object », « Forme » ou « Rectangle » nous parlons de conversion implicite.

De ce fait, lors de l'utilisation d'une variable, nous n'aurons accès qu'aux membres définis par le type conteneur.

Afin de récupérer les membres du type d'origine (Carre) je serai forcé de faire un casting explicite.

```
Carre c2 = (Carre)o; //Casting Explicite

c2.|

    Color
    Equals
    GetHashCode
    GetType
    Largeur
    Longueur
    ToString
```



# EXERCICES

## EXERCICES (MAXIMUM 5')

- I. Modifier la classe « Banque » afin qu'elle ne travaille qu'avec des comptes.





# LES CLASSES ABSTRAITES

C# - ORIENTÉ OBJET

- Définition
- Le mot-clé « abstract »

## LES CLASSES ABSTRAITES

# DÉFINITION

Le but de l'héritage est de définir les fonctionnalités communes à toutes les classes qui en sont dérivées afin de leur éviter de devoir redéfinir les éléments communs.

Cependant, il arrive que cette fonctionnalité commune ne soit pas suffisamment complète pour faire un objet utilisable et que l'on retrouve des fonctionnalités manquantes dans chaque classe dérivée.

Dans ce cas, nous pouvons spécifier que la classe ne peut être utilisée qu'à des fins d'héritage et non pas à l'instanciation d'objets.

Pour ce faire, nous utiliserons le mot clé « abstract » dans la déclaration de classe. Nous obtenons au final une classe abstraite.

```
namespace CoursCSharpFondements
{
    public abstract class Forme
    {
        public string Color { get; set; }
    }

    public class Rectangle : Forme
    {
        public double Longueur { get; set; }
        public double Largeur { get; set; }
    }

    public class Cercle : Forme
    {
        public double Rayon { get; set; }
    }
}
```

# LE MOT-CLÉ « ABSTRACT »

```
namespace CoursCSharpFondements
{
    public abstract class Forme
    {
        public string Color { get; set; }
        public double Surface { get { return CalculerSurface(); } }
        //Déclaration d'une propriété abstraite
        public abstract double Perimetre { get; }
        //Déclaration d'une méthode abstraite
        protected abstract double CalculerSurface();
    }
    public class Rectangle : Forme
    {
        public double Longueur { get; set; }
        public double Largeur { get; set; }

        public override double Perimetre {
            get { return (Longueur + Largeur) * 2; }
        }

        protected override double CalculerSurface() {
            return Longueur * Largeur;
        }
    }
    public class Cercle : Forme
    {
        public double Rayon { get; set; }

        public override double Perimetre {
            get { return Rayon * 2 * Math.PI; }
        }

        protected override double CalculerSurface() {
            return Math.Pow(Rayon,2) * Math.PI;
        }
    }
}
```

En dehors des classes, le mot-clé « **abstract** » peut être utilisé avec des méthodes, des propriétés, des indexeurs ou des événements.

S'il s'agit de méthodes abstraites, nous ne devons spécifier que le prototype de la méthodes suivit d'un point-virgule.

Les membres marqués comme abstraits, inclus dans une classe abstraite, doivent être implémenté par les classes qui héritent de la classe abstraite à l'aide du mot-clé « **override** ».



Toute classe contenant un membre abstrait doit elle-même être définie abstraite



# EXERCICES

## EXERCICES (MAXIMUM 25')

1. Définir la classe « Compte » comme étant abstraite.
2. Ajouter une méthode abstraite « protected » à la classe « Compte » appelée « CalculInteret » ayant pour prototype « double CalculInteret() » en sachant que pour un livret d'épargne le taux est toujours de 4.5% tandis que pour le compte courant si le solde est positif le taux sera de 3% sinon de 9.75%.
3. Ajouter une méthode « public » à la classe « Compte » appelée « AppliquerInteret » qui additionnera le solde avec le retour de la méthode « CalculInteret ».



# LES CLASSES STATIQUES

C# - ORIENTÉ OBJET

- Mot-clé « static »
- Règles d'utilisation des classes statiques
- Cas d'utilisation

## LES CLASSES STATIQUES



# MOT-CLÉ « STATIC »

Si nous utilisons le modificateur « **static** » pour déclarer un membre d'une classe ou d'une structure, cela revient à dire qu'il appartient au type lui-même plutôt qu'à une instantiation spécifique.

Nous accédons aux membres statiques en utilisant le nom du type lui-même.

Le mot-clé « **static** » peut être utilisé avec des classes, des champs, des méthodes, des propriétés, des événements et des constructeurs.

Il est obligatoire dans le cadre des opérateurs.

Mais il ne peut pas être utilisé avec des indexeurs, des destructeurs ou des déclarations de type autre que classe (énumérations, structures, interfaces et délégués).

```
namespace CoursCSharpFondements
{
    public class MaClasse
    {
        public static string Y { get; set; }
        public int X { get; set; }
    }
}

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaClasse m1 = new MaClasse();

            m1.X = 5;
            MaClasse.Y = "Hello";

            Console.WriteLine(m1.X);
            Console.WriteLine(MaClasse.Y);

            Console.ReadLine();
        }
    }
}
```



# RÈGLES D'UTILISATION DES CLASSES STATIQUES

```
namespace CoursCSharpFondements
{
    public static class AppConfig
    {
        public static string AppName
        {
            get { return "Mon Application" }
        }

        public static string ConnectionStringFormat
        {
            get { return "Data Source={0};Initial Catalog={1};User ID={2}; Password={3};"; }
        }

        public static string InvariantName
        {
            get { return "System.Data.SqlClient"; }
        }
    }
}
```

Une classe statique est fondamentalement identique à une classe non statique, à la différence près qu'une classe statique ne peut pas être instanciée.

En d'autres termes, nous ne pouvons pas utiliser le mot clé « **new** » pour créer une variable du type classe.

Une classe statique ne contient uniquement que des membres statiques. Par conséquent, comme il n'y a aucune variable d'instance, nous accédons aux membres d'une classe statique en utilisant le nom de classe lui-même.

De plus les règles suivantes s'appliquent lorsque nous déclarons une classe statique.

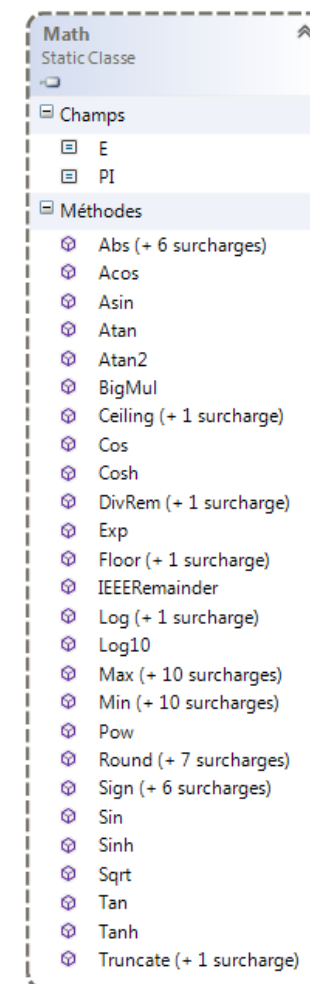
- Elle est « **sealed** » et ne peut hériter que de « **System.Object** ».
- Elle ne peut pas contenir de constructeurs d'instances.

# CAS D'UTILISATION

Une classe statique peut être utilisée comme conteneur commode pour les ensembles de méthodes qui opèrent simplement sur des paramètres d'entrée et n'ont pas à obtenir ou définir de champs d'instance internes.

Quelques exemples :

- La classe « Math ».
- Définir une classe pour la centralisation des variables de sessions en ASP .Net
- Définir une classe pour la centralisation des variables de configuration de votre application.





# LES INTERFACES

C# - ORIENTÉ OBJET

- Introduction
- Déclaration
- Implémentation par héritage
- Héritage entre interface
- Cas d'utilisation
  - Abstraction de code
  - Filtre d'accès aux membres
  - Solution pour l'héritage multiple
- Classes abstraites vs Interfaces

## LES INTERFACES

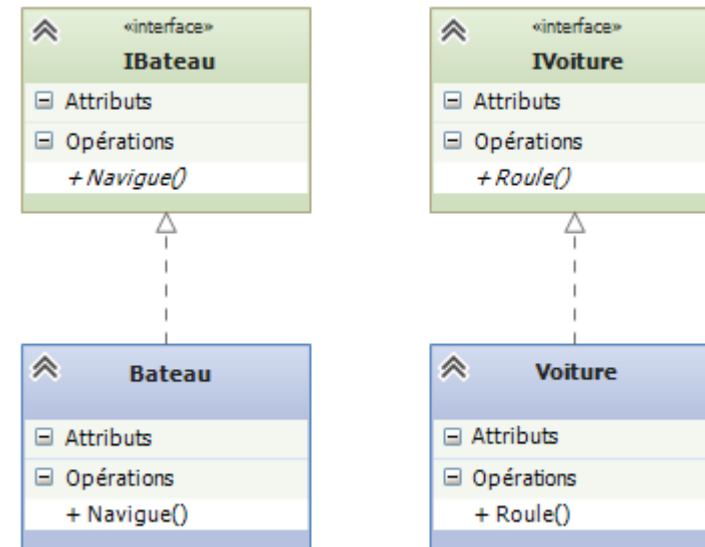
# INTRODUCTION

Les interfaces décrivent un groupe de fonctionnalités connexes qui peuvent appartenir à n'importe quelle classe ou structure.

Les interfaces peuvent être composées de méthodes, de propriétés, d'événements, d'indexeurs ou de toute combinaison de ces quatre types de membres.

Une interface peut être comparée à un contrat.

Lorsque nous disons qu'une classe ou une structure implémente une interface, cela signifie que la classe ou la structure s'engage à implémenter tous les membres définis par l'interface.



# DÉCLARATION

```
namespace CoursCSharpFondements
{
    public interface IForme
    {
        string Color { get; }
        double Perimetre { get; }

        double Aire();
    }
}
```

Les interfaces sont définies à l'aide du mot clé « **interface** » et, en général, leur nom commenceront par un « i » majuscule.

Il y a deux règles à respecter lors de la déclaration des interfaces :

- Une interface ne peut pas contenir de champs (variables membres).
- Les membres (propriétés, méthodes...) d'interface sont automatiquement « **public** ».

Puisque l'interface ne contient que des membres « **public** », nous ne devons spécifier aucun niveau d'accessibilité. (voir niveau d'accessibilité par défaut des membres d'interfaces).

De plus, nous ne pourrions pas spécifier de blocs d'instructions.

# IMPLÉMENTATION PAR HÉRITAGE

Pour signifier à une classe qu'elle doit implémenter l'interface, nous allons le faire comme pour l'héritage.

De plus, C# nous autorise d'implémenter plusieurs interfaces à une même classe en plus de l'héritage de classe en lui-même.

```
namespace CoursCSharpFondements
{
    public interface IForme
    {
        string Color { get; }
        double Perimetre { get; }

        double Aire();
    }

    public interface IRectangle
    {
        double Longueur { get; set; }
        double Largeur { get; set; }
    }
}
```

```
namespace CoursCSharpFondements
{
    public class Rectangle : object, IRectangle, IForme
    {
        private double _Longueur, _Largeur;
        private string _Color;

        public double Longueur
        {
            get { return _Longueur; }
            set { _Longueur = value; }
        }

        public double Largeur
        {
            get { return _Largeur; }
            set { _Largeur = value; }
        }

        public string Color
        {
            get { return _Color; }
            set { _Color = value; }
        }

        public double Perimetre
        {
            get { return (Longueur + Largeur) * 2; }
        }

        public double Aire()
        {
            return Longueur * Largeur;
        }
    }
}
```



# HÉRITAGE ENTRE INTERFACES

```
namespace CoursCSharpFondements
{
    public class Rectangle : object, IRectangle
    {
        private double _Longueur, _Largeur;
        private string _Color;

        public double Longueur
        {
            get { return _Longueur; }
            set { _Longueur = value; }
        }

        public double Largeur
        {
            get { return _Largeur; }
            set { _Largeur = value; }
        }

        public string Color
        {
            get { return _Color; }
            set { _Color = value; }
        }

        public double Perimetre
        {
            get { return (Longueur + Largeur) * 2; }
        }

        public double Aire()
        {
            return Longueur * Largeur;
        }
    }
}
```

Le concept d'héritage est applicable également aux interfaces.

Toute classe implémentant l'interface « IRectangle » devra implémenter les membres déclarés dans « IForme » en plus de ceux déclarer dans « IRectangle ».

```
namespace CoursCSharpFondements
{
    public interface IForme
    {
        string Color { get; }
        double Perimetre { get; }

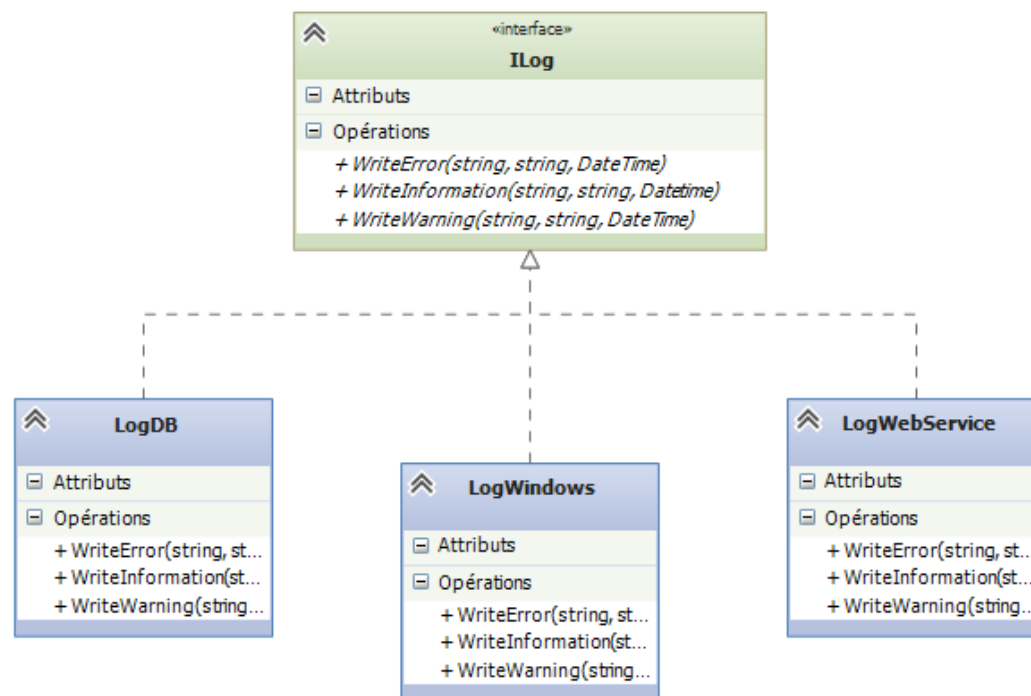
        double Aire();
    }

    public interface IRectangle : IForme
    {
        double Longueur { get; set; }
        double Largeur { get; set; }
    }
}
```

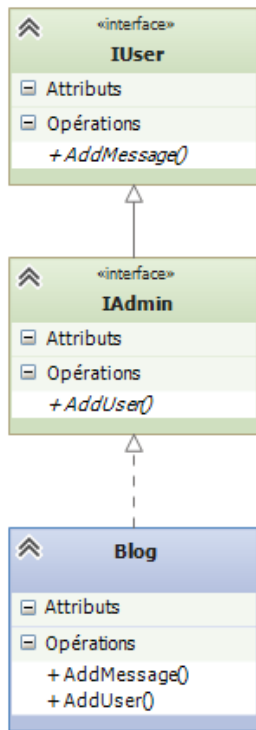
# CAS D'UTILISATION - ABSTRACTION DE CODE

Dans le cas de l'abstraction de code, nous allons nous servir des interfaces afin de déclarer les fonctionnalités que devront déclarer nos classes.

Enfin, nous utiliserons le polymorphisme afin de pouvoir déclarer nos variables en utilisant le type « ILog » afin d'être totalement indépendant du type de variable réelle (« LogDB », « LogWindow » ou « LogWebService ») et du code réellement exécuté en arrière plan.



# CAS D'UTILISATION - FILTRE D'ACCÈS AUX MEMBRES



```
namespace CoursCSharpFondements
{
    public interface IUser
    {
        void AddMessage();
    }

    public interface IAdmin : IUser
    {
        void AddUser();
    }

    public class Blog : IUser, IAdmin
    {
        public void AddUser()
        {
            //code à exécuter
        }

        public void AddMessage()
        {
            //code à exécuter
        }
    }
}
```

Nous pouvons, également, utiliser nos interfaces et le polymorphisme afin de limiter (filtrer) les fonctionnalités accessibles en fonction du type d'interface utilisé pour déclarer nos variables.

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Blog b = new Blog();
            IUser User = b;
            IAdmin Admin = b;
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Blog b = new Blog();
            IUser User = b;
            IAdmin Admin = b;
        }
    }
}
```

# CAS D'UTILISATION - SOLUTION DE L'HÉRITAGE MULTIPLE

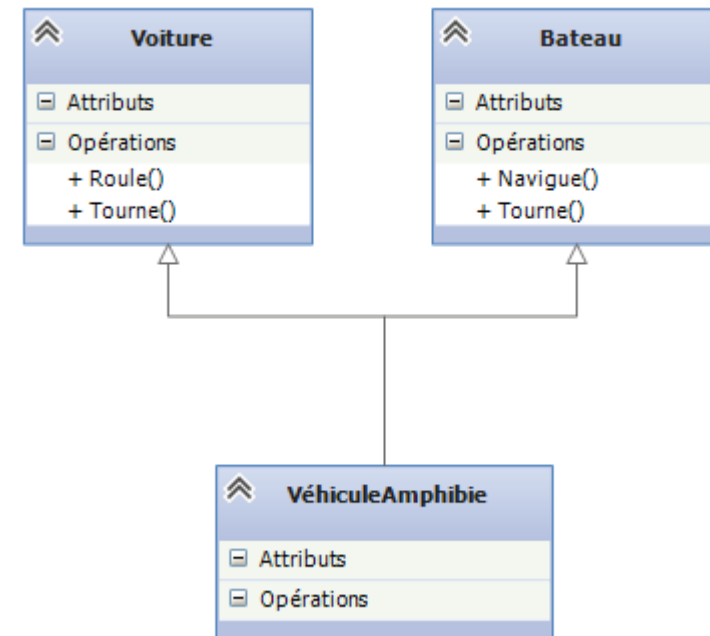
Les interfaces sont également utiles pour résoudre le problème de l'héritage multiple.

En effet, l'héritage multiple est interdit en C# alors qu'il existe dans les principes de l'orienté objet et dans d'autres langages comme le « C++ ».

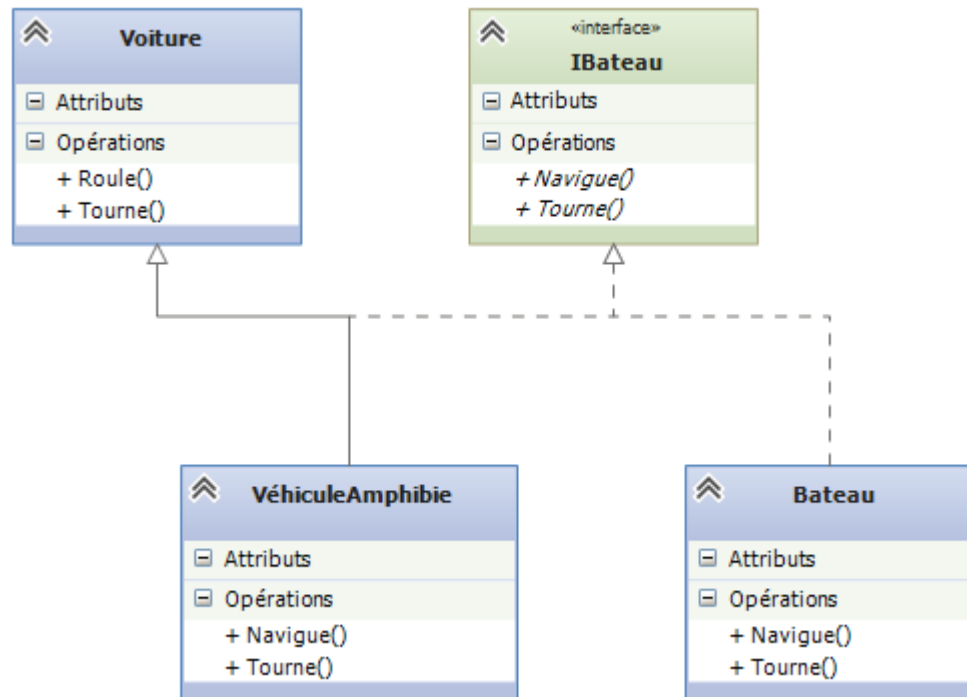
Il nous faut donc trouver une solution pour parvenir à créer notre classe « VéhiculeAmphibie » en C#.

Comme nous ne pouvons hériter que d'une seule classe, nous allons faire le choix de la classe qui nous servira de classe de base.

Enfin nous allons utiliser les interfaces pour nous obliger à définir les fonctionnalités manquantes de l'autre classe.



# CAS D'UTILISATION - SOLUTION DE L'HÉRITAGE MULTIPLE



Nous sommes donc obligé d'implémenter la méthode « Navigue » dans la classe « VehiculeAmphie ».

De plus, si nous avons des fonctionnalités communes, comme la méthode « Tourne », il nous faudra peut-être :

- Déclarer la méthode « Tourne » de « Voiture » comme « **virtual** ».
- La redéfinir avec le mot-clé « **override** » dans la classe « VehiculeAmphibie ».

# CLASSES ABSTRAITES VS INTERFACES

Attention à ne pas confondre interfaces et classes abstraites!!

En effet plusieurs notions différencies l'utilisation des classes abstraites des interfaces.

Cas	Interface	Classe abstraite
Les classes peuvent Hériter/implémenter plusieurs ...	OUI	NON (une seule)
Peut contenir des variables membres	NON	OUI
Peut contenir des blocs d'instructions	NON	OUI
Peut contenir des membres private, protected, internal ou protected internal	NON	OUI
Peut contenir des types imbriqués	NON	OUI
...	...	...



# EXERCICES

## EXERCICES (MAXIMUM 15')

1. Définir l'interface « ICustomer », afin de limiter l'accès à consulter la propriété « Solde » et d'utiliser les méthodes « Depot » et « Retrait ».
2. Définir l'interface « IBanker » ayant les mêmes fonctionnalités que « ICustomer ».  
Elle lui permettra, en plus, d'invoquer la méthode du « AppliquerInteret » et offrira un accès en lecture au « Titulaire » et au « Numero ».
3. Si nous ajoutons la propriété « LigneDeCredit » à « IBanker », définir sur papier les modifications qu'il faudrait apporter à nos classes.





# CONSTRUCTION ET DESTRUCTION D'OBJETS

C# - ORIENTÉ OBJET

- Constructeurs
- Constructeur par défaut
- Surcharge de constructeurs
- Constructeurs et Héritage
- Appel du constructeur de la classe de base
- Petite astuce
- Les destructeurs
- Le « Garbage Collector »
- L'interface « IDisposable »
- Le bloc d'instruction « using(...) { } »

## CONSTRUCTION ET DESTRUCTION D'OBJETS

# CONSTRUCTEURS

Les constructeurs sont des méthodes particulières de la classe qui sont exécutées à l'appel de l'opérateur « **new** ».

Les constructeurs portent le même nom que la classe, mais n'ont pas de type de retour, et servent habituellement à initialiser les données membres du nouvel objet.

De plus, les constructeurs n'appartiennent qu'à la classe qui les déclare et ne sont pas, par conséquent, hérités par les classes dérivées.

Enfin, ceux-ci ont les mêmes niveaux d'accessibilité que n'importe quel membre d'une classe.

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Etudiant e1 = new Etudiant();
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    public class Etudiant
    {
        public string Nom { get; set; }
        public string Prenom { get; set; }

        private List<int> _Notes;

        public Etudiant()
        {
            _Notes = new List<int>();
        }
    }
}
```

# LE CONSTRUCTEUR PAR DÉFAUT

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            //Appel du constructeur par défaut
            MaClasse mc = new MaClasse();

            //Appel du constructeur sans paramètre
            MonAutreClasse mac = new MonAutreClasse();
        }

        public class MaClasse
        {
        }

        public class MonAutreClasse
        {
            public MonAutreClasse()
            {
            }
        }
    }
}
```

Si nous ne fournissons pas de constructeur pour notre objet, C# en créera un par défaut quiinstanciera l'objet.

Ce constructeur par défaut est un constructeur sans paramètre.

A partir du moment où nous fournissons un constructeur, quel qu'il soit, ce constructeur par défaut disparaît.

# SURCHARGE DE CONSTRUCTEURS

Bien que spéciaux, les constructeurs restent des ‘méthodes’.

Par conséquent ceux-ci peuvent être surchargés en respectant le concept de surcharge de méthodes.

Le but étant de nous offrir plusieurs possibilités de construire nos objets.

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Etudiant e1 = new Etudiant();
            Etudiant e2 = new Etudiant("Morre", "Thierry");
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    public class Etudiant
    {
        public string Nom { get; set; }
        public string Prenom { get; set; }

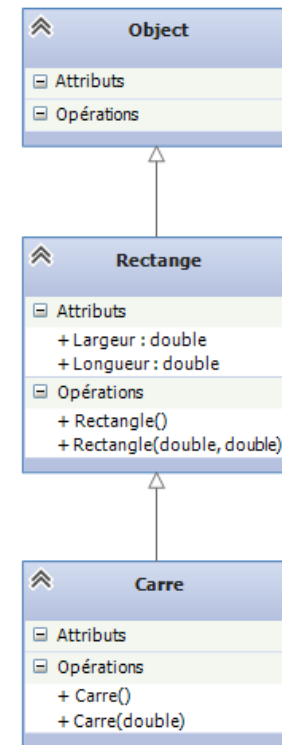
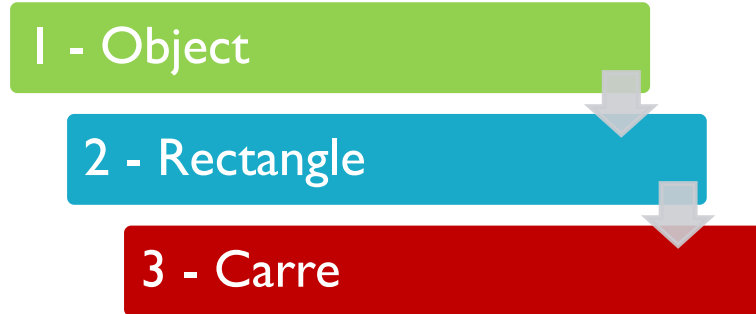
        private List<int> _Notes;

        public Etudiant()
        {
            _Notes = new List<int>();
        }

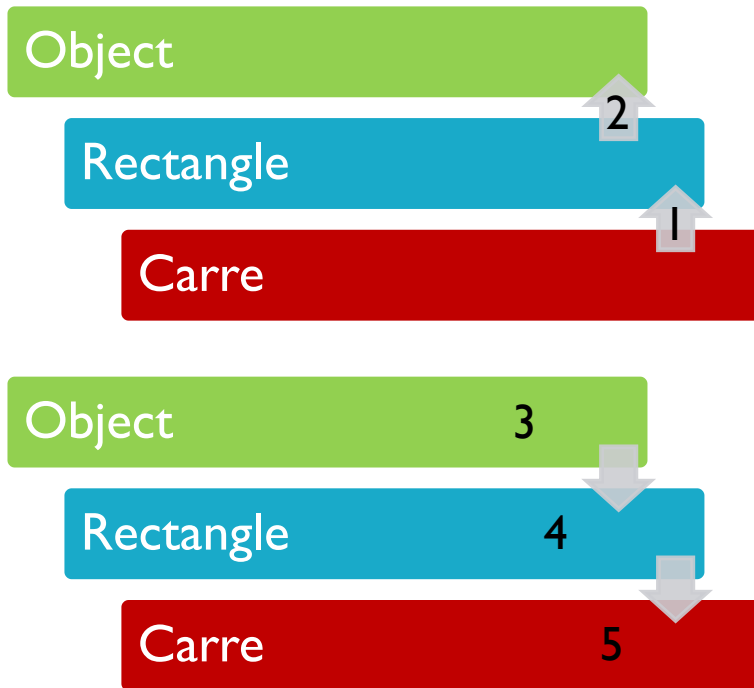
        public Etudiant(string Nom, string Prenom)
        {
            _Notes = new List<int>();
            this.Nom = Nom;
            this.Prenom = Prenom;
        }
    }
}
```

# CONSTRUCTEURS ET HÉRITAGE

Dans le cadre de l'héritage, afin de créer un objet en mémoire, le « Runtime » va créer une suite d'objet en respectant l'ordre défini par l'héritage.



# CONSTRUCTEURS ET HÉRITAGE



Pour parvenir à ce résultat, le « Runtime » va partir du constructeur de la classe instanciée.

Ensuite, avant de réaliser celui-ci, il va appeler le constructeur spécifié de la classe parente qui lui-même avant de s'exécuter va appeler le constructeur de la spécifié de la classe parente, ainsi de suite jusqu'à arriver à la classe « object ».

Une fois arrivé à ce dernier, il va exécuter le corps des constructeurs rencontrés en redescendant vers le constructeur initialement appelé.

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Carre c = new Carre();
        }
    }
}
```

# APPEL DU CONSTRUCTEUR DE LA CLASSE DE BASE

Par défaut, lorsque rien n'est spécifié, le « Runtime » appelle implicitement le constructeur sans paramètre (constructeur par défaut ou pas) de la classe parente.

Ce qui veut dire que s'il n'existe pas de constructeur sans paramètre, nous serons obligés de spécifier le constructeur de la classe parent que nous souhaitons appeler en lui passant les paramètres dont il a besoin.

Afin de réaliser cet appel, nous devons utiliser le mot-clé « **base** » suivi de parenthèses et des paramètres nécessaires après la déclaration du constructeur de la classe enfant..

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Carre c = new Carre(5D);
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    class Rectangle
    {
        public double Longueur { get; set; }
        public double Largeur { get; set; }

        public Rectangle()
        {
        }

        public Rectangle(double Longueur, double largeur)
        {
            this.Longueur = Longueur;
            this.Largeur = Largeur;
        }
    }

    class Carre : Rectangle
    {
        public Carre()
        {
        }

        public Carre(double cote) : base (cote, cote)
        {
        }
    }
}
```



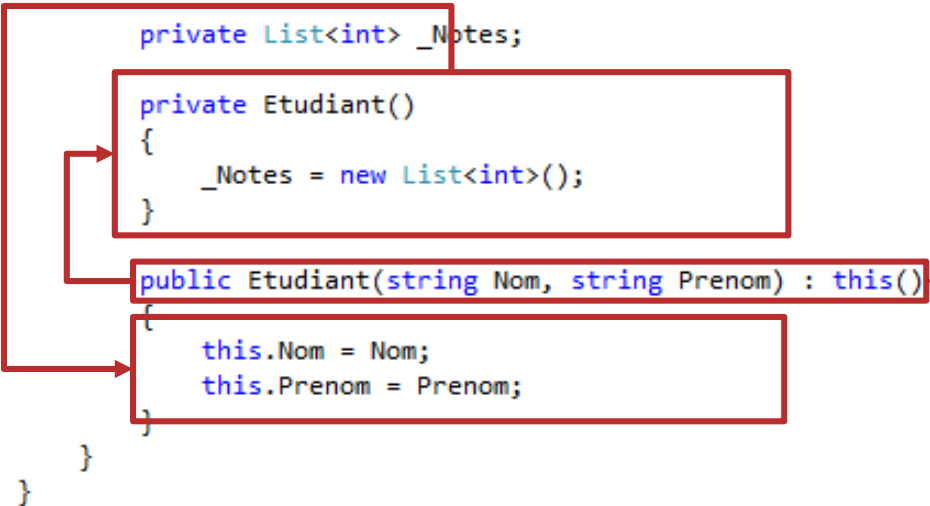
# PETITE ASTUCE

```
namespace CoursCSharpFondements
{
    public class Etudiant
    {
        public string Nom { get; private set; }
        public string Prenom { get; private set; }

        private List<int> _Notes;

        private Etudiant()
        {
            _Notes = new List<int>();
        }

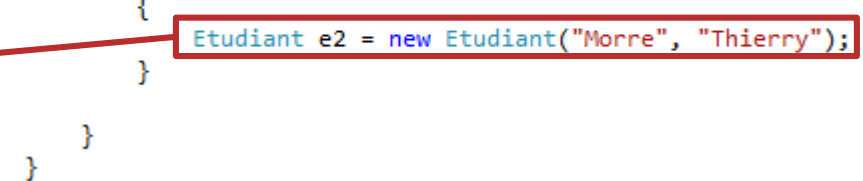
        public Etudiant(string Nom, string Prenom) : this()
        {
            this.Nom = Nom;
            this.Prenom = Prenom;
        }
    }
}
```



Puisque nous pouvons appeler un constructeur de la classe parent, nous pouvons également appeler un autre constructeur de la classe courante.

Au lieu d'utiliser le mot-clé « **base** » remplaçons le par « **this** ».

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            Etudiant e2 = new Etudiant("Morre", "Thierry");
        }
    }
}
```

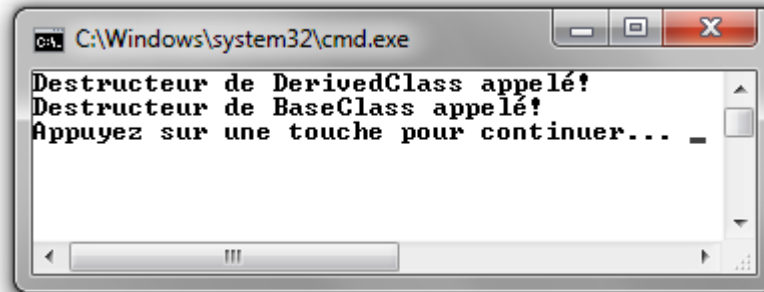


# LES DESTRUCTEURS

Les destructeurs permettent de détruire les instances des classes.

Cependant :

- Les destructeurs ne peuvent être définis que dans des classes.
- Les destructeurs ne peuvent pas être hérités ni surchargés.
- Les destructeurs ne peuvent pas être appelés. Ils sont appelés automatiquement par le « **garbage collector** ».
- Un destructeur n'accepte pas de niveau d'accessibilité ni de paramètres.



```
public class BaseClass
{
    ~BaseClass()
    {
        Console.WriteLine("Destructeur de BaseClass appelé!");
    }
}

public class DerivedClass : BaseClass
{
    ~DerivedClass()
    {
        Console.WriteLine("Destructeur de DerivedClass appelé!");
    }
}

class Program
{
    static void Main(string[] args)
    {
        DerivedClass dc = new DerivedClass();
    }
}
```

# LE « GARBAGE COLLECTOR »



Le « **garbage collector** » recherche les objets qui ne sont plus utilisés par l'application.

S'il considère qu'un objet est candidat à la destruction, il appelle le destructeur (s'il y a lieu) et libère la mémoire utilisée pour stocker l'objet.

Cependant, il est impossible de prévoir lorsqu'un ramassage se produira.

# L'INTERFACE « IDISPOSABLE »

Cependant, le « **garbage collector** » n'a pas connaissance des ressources non managées, telles que les fichiers et flux ouverts.

Nous devons donc par gérer nous même la libération des ressources prises par ces objets.

A cette fin, nous utiliserons l'interface « IDisposable », qui nous imposera de déclarer une méthode « Dispose() », afin d'y implémenter le code de la libération des ressources.

Lorsque nous n'utilisons plus l'objet, il nous suffit d'appeler cette méthode afin de rendre cet objet 'collectable' par le « **garbage collector** ».

Cette interface est déjà implémentée par les classes travaillant sur les flux en général. (tel que les fichiers, les connexions aux bases de données etc.)

```
namespace System
{
    // Résumé :
    // Définit une méthode pour libérer des ressources allouées.
    public interface IDisposable
    {
        // Résumé :
        // Exécute les tâches définies par l'application associées à la libération ou
        // à la redéfinition des ressources non managées.
        void Dispose();
    }
}
```

# LE BLOC D'INSTRUCTION « USING(...) { } »

```
using System;
using System.IO;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            using(TextWriter tw = File.CreateText("Exemple.txt"))
            {
                tw.WriteLine("Ca marche");
            }
        }
    }
}
```

Afin de simplifier notre code et d'appeler la méthode « Dispose » de manière automatique, nous pouvons déclarer un bloc d'instruction « **using** ».

Ce bloc nous permet de déclarer une variable de type « IDisposable », qui ne sera accessible que dans ce bloc d'instruction.

Une fois que le bloc d'instruction se termine, la méthode « Dispose » est appelée automatiquement.



# EXERCICES

# EXERCICES (MAXIMUM 30')

1. Ajoutez, dans la classe « Compte », deux constructeurs prenant en paramètre :
  1. Le numéro et le titulaire
  2. Le numéro, le titulaire et le solde (pour le cas d'une base de données)
2. Le cas échéant, ajoutez le ou les constructeurs aux classes « Courant » et « Epargne ».
3. Changer l'encapsulation des propriétés des classes « Personne », « Compte » et « Epargne » afin de spécifier leur mutateur en « **private** ».
4. Définir ce qu'il manque pour que le programme continue à tourner.



# LES EXCEPTIONS

C# - ORIENTÉ OBJET



- Introduction
- « try » - « catch » - « finally »
- Multiples « catch »
- Mot-clé « throw »
- Créer ses propres exceptions

## LES EXCEPTIONS

# INTRODUCTION

La gestion des erreurs est une partie importante du développement, que ce soit en importance ou en temps.

Par défaut, lorsqu'une erreur survient, celle-ci va remonter niveau par niveau jusqu'à atteindre le point d'entrée de l'application.

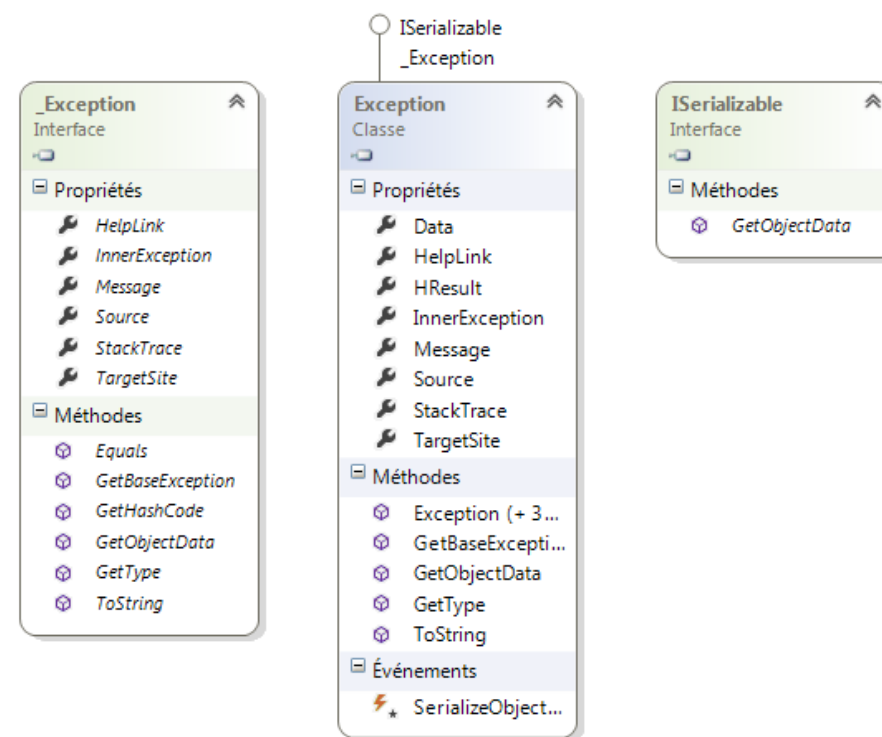
Si entre temps, nous ne la capturons pas pour la traiter, le système d'exploitation mettra fin à votre application en vous signalant qu'une erreur n'a pas été gérée.

En C#, un telle erreur est appelée « Exception ».

Il existe une multitude de types d'erreur mais tous, au final, héritent de la classe « Exception ».

Pour plus d'informations :

[http://msdn.microsoft.com/fr-fr/library/z4c5tckx\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/z4c5tckx(v=vs.110).aspx)



# « TRY » - « CATCH » - « FINALLY »

```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                //Code à exécuter normalement
            }
            catch (Exception ex)
            {
                //Code à exécuter si une exception est levée
            }
            finally
            {
                //Code à exécuter après le try
                //ou le catch (si celui-ci est utilisé)
            }
        }
    }
}
```

Lorsqu'une exception est levée, celle-ci met fin au bloc d'instruction courant (méthode, if, switch, try, etc.) et commence à remonter jusqu'à la racine de l'application.

Pour intercepter une exception nous devons utiliser le bloc « **try** ».

Celui-ci doit être suivi d'au moins un bloc « **catch** » ou d'un bloc « **finally** ».

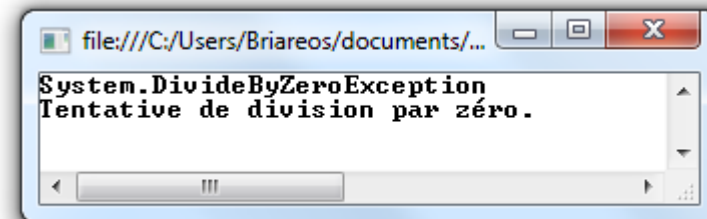
Si notre bloc « **try** » détecte une exception, celui-ci s'interrompt et envoie l'erreur en paramètre du bloc « **catch** » (si existant).

Le bloc « **finally** » (si existant) s'exécute quoi qu'il arrive et est le plus souvent utilisé pour nettoyer les ressources utilisées.

# EXEMPLE

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int x = 0;
                int y = 10 / x;
                Console.WriteLine(y);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.GetType());
                Console.WriteLine(ex.Message);
            }
            finally
            {
                Console.ReadLine();
            }
        }
    }
}
```



# MULTIPLES « CATCH »

Lorsque plusieurs exceptions différentes sont susceptibles d'être déclenchées. Nous pouvons implémenter plusieurs blocs « **catch** ».

Chacun gérant un type d'exception différent.

Dans ce cas, l'exception va passer de « **catch** » en « **catch** » à la recherche du premier d'entre eux compatible avec l'exception.

Une fois qu'elle a trouvé un bloc catch compatible, l'exception sera traité par celui-ci avant de passer au bloc « **finally** » (si celui-ci existe) et de continuer.

Par conséquent, le bloc « **catch** » traitant le type « Exception » doit être le dernier déclaré.

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                int x = 0;
                int y = 10 / x;

                Console.WriteLine(y);
            }
            catch (DivideByZeroException dbzex)
            {
                Console.WriteLine(dbzex.GetType());
                Console.WriteLine(dbzex.Message);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Ce n'est pas une division par zéro");
            }
            finally
            {
                Console.ReadLine();
            }
        }
    }
}
```

# MOT-CLÉ « THROW »

```
namespace CoursCSharpFondements
{
    public static class EQ2D
    {
        public static bool Resoudre(double A, double B, double C, out double? X1, out double? X2)
        {
            X1 = null;
            X2 = null;

            if (A == 0D)
                throw new ArgumentException("A ne peut pas être égale à 0");

            double Delta = Math.Pow(B, 2) - (4 * A * C);

            if (Delta > 0D)
            {
                X1 = (-B + Math.Sqrt(Delta)) / (2 * A);
                X2 = (-B - Math.Sqrt(Delta)) / (2 * A);
                return true;
            }
            else if (Delta == 0D)
            {
                X1 = X2 = -B / (2 * A);
                return true;
            }

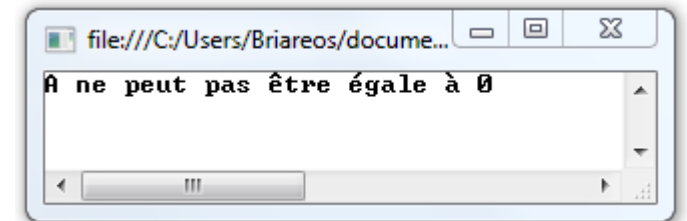
            return false;
        }
    }
}
```

L'instruction « **throw** » sert à signaler la présence d'une situation anormale (exception) pendant l'exécution du programme.

C'est grâce à lui que nous allons déclencher des exceptions.

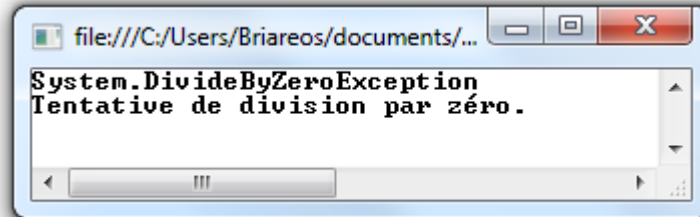
```
try
{
    double? X1, X2;

    if (EQ2D.Resoudre(0, 1, -2, out X1, out X2))
    {
        Console.WriteLine("X1 : {0} -- X2 : {1}", X1, X2);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```



# MOT-CLÉ « THROW »

Le mot-clé « **throw** » peut nous servir à renvoyer une exception déjà existante afin, par exemple, de réaliser des actions localement et de renvoyer (faire suivre) à l'appelant l'exception par la suite.



```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Program p = new Program();
                p.MaMethode();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.GetType());
                Console.WriteLine(ex.Message);
            }

            Console.ReadLine();
        }

        void MaMethode()
        {
            try
            {
                int x = 0;
                int j = 5 / x;
            }
            catch (Exception isTeVeel)
            {
                //traitement local

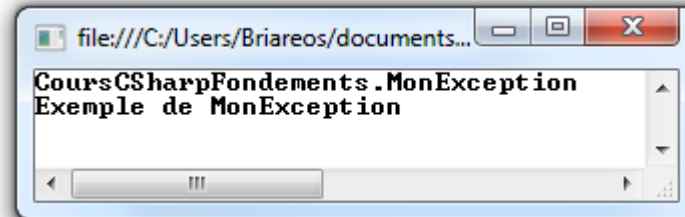
                throw isTeVeel;
            }
        }
    }
}
```

# CRÉER SES PROPRES EXCEPTIONS

```
namespace CoursCSharpFondements
{
    public class MonException : DivideByZeroException
    {
        public string MaVariable { get; set; }

        public MonException(string MaVariable) : this(MaVariable, "Exemple de MonException")
        {
        }

        public MonException(string MaVariable, string Message) : base(Message)
        {
            this.MaVariable = MaVariable;
        }
    }
}
```



Pour créer ses propres exceptions, il n'y a qu'une règle à respecter.

La classe que nous allons définir **doit être**, au final, une classe dérivée de la classe « **Exception** ».

```
using System;

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                throw new MonException("MaVariable");
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.GetType());
                Console.WriteLine(ex.Message);
            }

            Console.ReadLine();
        }
    }
}
```





# EXERCICES

# EXERCICES (MAXIMUM 15')

Dans la classe « Compte » :

1. Au niveau de la méthode « Depot », déclenchez une exception de type « ArgumentOutOfRangeException » si le montant n'est pas supérieur à 0 (zéro).
2. Faites de même au niveau de la méthode « Retrait » et y ajouter le déclenchement d'une exception de type « SoldInsuffisantException » si le montant ne peut être retiré.

Au niveau de la classe « Courant » :

1. Au niveau de la propriété « LigneDeCredit », déclenchez une exception de type « ArgumentOutOfRangeException » si la valeur n'est pas supérieur ou égale à 0 (zéro).



# LES DÉLÉGUÉS

C# - ORIENTÉ OBJET

- Historique
- Exemple de pointeur de fonction en C++
- Déclaration
- Ajout et retrait de méthodes
- Déclenchement
- Problème de sécurité
- Incohérence cohérente!

## LES DÉLÉGUÉS

# HISTORIQUE

L'origine des délégués remonte au C++, plus précisément aux pointeurs de fonctions.

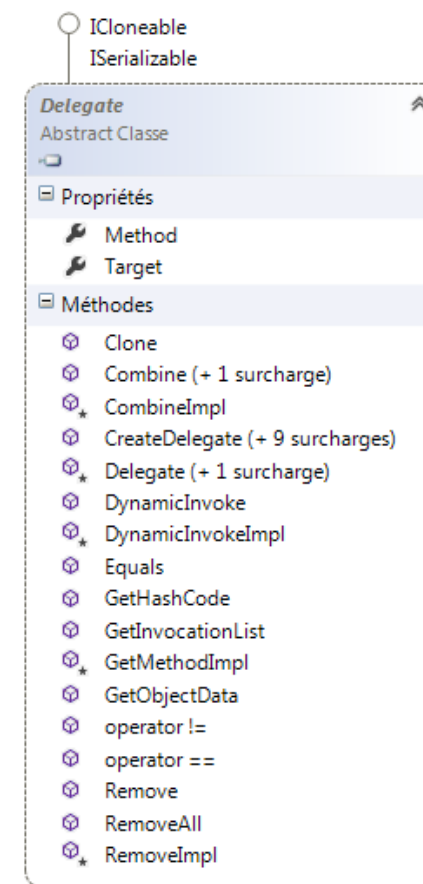
Les pointeurs de fonction sont des pointeurs comme des pointeurs de variables, mais qui pointent sur l'adresse d'une fonction.

Nous devons garder à l'esprit qu'un programme qui est exécuté prend un certain espace dans la mémoire principale.

Les fonctions étant des membres comme les autres variables, celles-ci ont également une adresse.

En C#, les délégués sont des collections de pointeurs de fonction qui hérite tous de la classe « **Delegate** ».

Lors de la déclaration d'un délégué, nous ne devons pas perdre de vue que nous travaillons avec des méthodes, c'est-à-dire avec un type de retour et des paramètres d'entrée.



# EXEMPLE DE POINTEUR DE FONCTION EN C++

```
#include "stdafx.h"

void MaMethode();

typedef void (*MonDelegate)(void);

int _tmain(int argc, _TCHAR* argv[])
{
    MonDelegate del = &MaMethode;

    if (del != NULL)
    {
        del();
    }

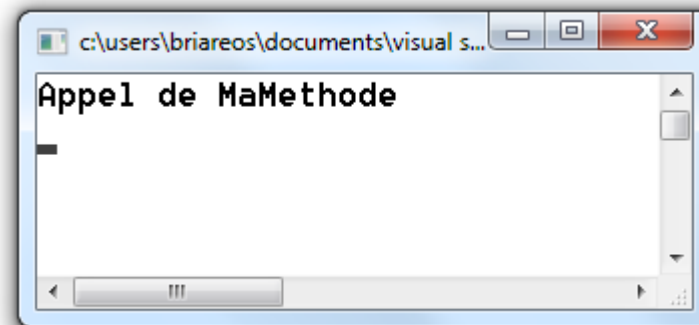
    std::getchar();
    return 0;
}

void MaMethode()
{
    std::cout << "Appel de MaMethode" << std::endl;
}
```

Définition du type « MonDelegate » comme étant un pointeur de fonction travaillant avec les méthodes ayant comme prototype : void method(void)

Déclaration de la variable de type « MonDelegate » et affectation de l'adresse de la méthode « MaMethode ».

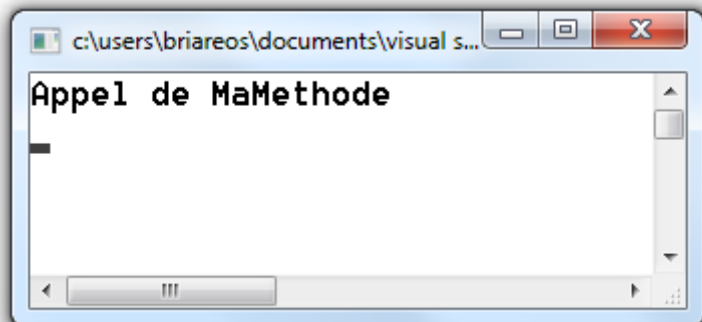
Déclenchement du pointeur de fonction qui va appeler la méthode spécifiée à l'adresse contenue dans le pointeur.



# DÉCLARATION

En C#, pour déclarer un délégué, nous allons utiliser le mot-clé « **delegate** ».

Ce dernier, va nous permettre de créer un type pour un prototype de méthode particulier.



```
namespace CoursCSharpFondements
{
    delegate void MonDelegate();

    class Program
    {
        static void Main(string[] args)
        {
            MonDelegate del = MaMethode;

            if (del != null)
            {
                del();
            }

            Console.ReadKey();
        }

        static void MaMethode()
        {
            Console.WriteLine("Appel de MaMethode");
        }
    }
}
```

# AJOUT ET RETRAIT DE MÉTHODES

```
namespace CoursCSharpFondements
{
    delegate void MonDelegate();

    class Program
    {
        static void Main(string[] args)
        {
            MonDelegate del = MaMethode;
            del += MaMethode;
            del += MaMethode2;
            del -= MaMethode;

            if (del != null)
            {
                del();
            }

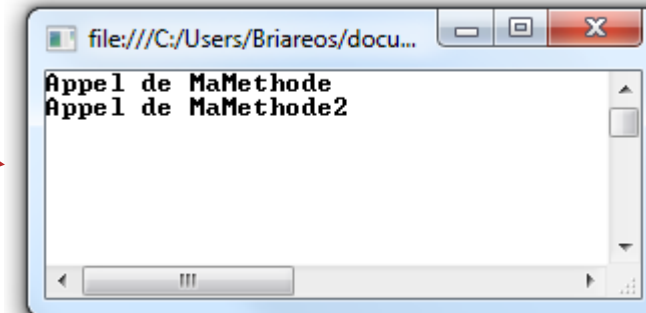
            Console.ReadKey();
        }

        static void MaMethode()
        {
            Console.WriteLine("Appel de MaMethode");
        }

        static void MaMethode2()
        {
            Console.WriteLine("Appel de MaMethode2");
        }
    }
}
```

Comme expliqué, à la différence du C++, les délégués C# peuvent contenir plusieurs adresses de méthodes, même plusieurs fois la même.

Pour ajouter une méthode, nous utiliserons l'opérateur « += », tandis que pour les retirer nous utiliserons l'opérateur « -= ».





# DÉCLENCHEMENT

Pour déclencher un délégué, il suffit d'appeler la méthode « Invoke » ou de rajouter « ( ) » derrière la variable.

Ensuite, entre les parenthèses, nous devons ajouter les paramètres à passer au délégué (si nécessaire).

Le nombre et le type des paramètres sont spécifiés par le prototype de méthode du délégué.

Cependant, avant de déclencher un délégué, nous devons nous assurer que celui-ci n'est pas « null ».

En effet, si aucune méthode n'est assignée, le délégué étant un type référence qui prend la valeur « null » par défaut.

```
namespace CoursCSharpFondements
{
    delegate void MonDelegate();
    delegate void DelegateMath(int x, int y);

    class Program
    {
        static void Main(string[] args)
        {
            MonDelegate del = MaMethode;
            if (del != null)
            {
                del();
            }

            Console.WriteLine();

            DelegateMath del2 = Addition;
            if (del2 != null)
            {
                del2(5, 7);
            }

            Console.ReadKey();
        }

        static void MaMethode()
        {
            Console.WriteLine("Appel de MaMethode");
        }

        static void Addition(int x, int y)
        {
            Console.WriteLine(x + y);
        }
    }
}
```

# INCOHÉRENCE COHÉRENTE!

```
namespace CoursCSharpFondements
{
    delegate void DelegateMath(int x, int y);
}

namespace CoursCSharpFondements
{
    class MaClasse
    {
        DelegateMath del = null;

        public void Register(DelegateMath Action)
        {
            del += Action;
        }

        public void Raise(int x, int y)
        {
            del(x, y);
        }
    }
}
```

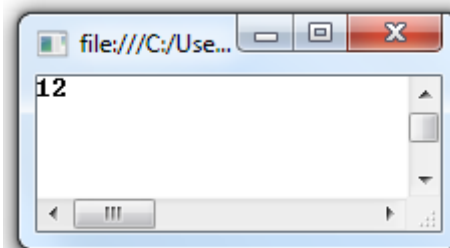


```
namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaClasse mc = new MaClasse();
            mc.Register(Addition);

            mc.Raise(5, 7);

            Console.ReadLine();
        }

        private static void Addition(int x, int y)
        {
            Console.WriteLine(x + y);
        }
    }
}
```



Pour rappel, les délégués nous viennent des pointeurs de fonction.

Ces derniers fonctionnent sur l'adresse de méthodes.

Ce qui veut dire que nous déclenchons un élément stocké à une adresse indépendamment de son niveau d'accessibilité.

De ce fait, notre délégué peut appeler une méthode privée d'une autre classe à condition que ce soit cette autre classe qui lui fournisse l'adresse de sa méthode.

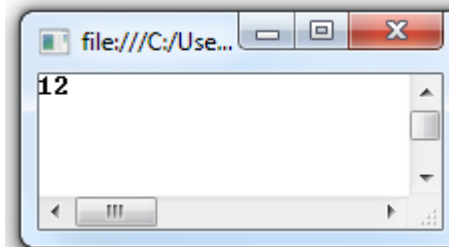
On dit dans ce cas, qu'une classe délègue le déclenchement de sa méthode à une variable.

# PROBLÈME DE SÉCURITÉ

Cependant, un délégué est sensible à l'encapsulation.

Ce qui veut dire que si une variable du type du délégué est déclarée « **public** », tout le monde pourra bien sûr ajouter et supprimer des méthodes à celui-ci, mais également le déclencher.

Ce qui dans certaines situations peut d'avérer délicat dans notre volonté de contrôler le processus.



```
namespace CoursCSharpFondements
{
    delegate void DelegateMath(int x, int y);
}

namespace CoursCSharpFondements
{
    class MaClasse
    {
        public DelegateMath del = null;
    }
}

namespace CoursCSharpFondements
{
    class Program
    {
        static void Main(string[] args)
        {
            MaClasse mc = new MaClasse();
            mc.del += Addition;
            mc.del(5,7);
            Console.ReadLine();
        }

        private static void Addition(int x, int y)
        {
            Console.WriteLine(x + y);
        }
    }
}
```



# LES ÉVÉNEMENTS

C# - ORIENTÉ OBJET

- Introduction
- Les événements et l'héritage
- Le délégué « EventHandler »
- Le délégué « RoutedEventHandler »

## LES ÉVÉNEMENTS

# INTRODUCTION

Un événement en C# sert principalement à la notification, c'est-à-dire signaler à son environnement que quelque chose de particulier s'est produit.

Pour cela, ce dernier est typé par un délégué qui va lui fournir le côté « collection de pointeurs de fonction », quant à l'événement il va fournir un niveau de sécurité complémentaire au délégué.

Cette sécurité est que :

**Seule la classe qui à déclarer l'événement peut le déclencher.**

Pour déclarer un événement, nous utiliserons le mot-clé « **event** » dans la déclaration de notre variable de type délégué.

```
namespace CoursCSharpFondements
{
    delegate void ValueChangedEventHandler(object sender, int value);
}

namespace CoursCSharpFondements
{
    class MaClasse
    {
        public event ValueChangedEventHandler ValueChanged = null;

        private int _X;

        public int X
        {
            get { return _X; }
            set
            {
                if (_X != value)
                {
                    _X = value;
                    ValueChanged(this, value);
                }
            }
        }
    }
}
```

# LES ÉVÉNEMENTS ET L'HÉRITAGE

```
namespace CoursCSharpFondements
{
    delegate void ValueChangedEventHandler();
}

namespace CoursCSharpFondements
{
    class MaClasse
    {
        public event ValueChangedEventHandler ValueChanged = null;
    }

    class MonAutreClasse : MaClasse
    {
        private void MaMethode()
        {
            ValueChanged(this, 5);
        }
    }
}
```

ValueChangedEventHandler MaClasse.ValueChanged

Erreur :

L'événement 'CoursCSharpFondements.MaClasse.ValueChanged' ne peut apparaître qu'à gauche de += ou -= (sauf en cas d'utilisation à partir du type 'CoursCSharpFondements.MaClasse')

Comme expliqué précédemment, seule la classe qui déclare l'événement peut le déclencher.

Pour contourner le problème et autoriser le déclenchement dans les classe enfants, nous devons créer dans la classe qui déclare l'événement une méthode « **protected** », qui se chargera de déclencher l'événement.

Ensuite, pour déclencher l'événement, il nous suffira d'appeler cette méthode.

# LE DÉLÉGUÉ « EVENTHANDLER »

Dans le cadre du développement graphique, nous allons retrouver plusieurs événements basés principalement sur deux types de délégués spécifiquement à la technologie utilisée.

Pour les « Windows Forms », nous retrouverons les événements basés sur le délégué « EventHandler ».

Ce délégué reçoit deux paramètres :

- « sender » est de type « System.Object », il correspond à l'objet qui déclenche l'événement.
- « e » est de type « EventArgs », il contient les données d'événement.

```
namespace System
{
    ...public delegate void EventHandler(object sender, EventArgs e);
}
```

```
namespace System
{
    // Résumé :
    //     System.EventArgs est la classe de base des classes contenant des données
    //     d'événement.
    public class EventArgs
    {
        ...public static readonly EventArgs Empty;

        ...public EventArgs();
    }
}
```



# LE DÉLÉGUÉ « ROUTEDEVENHANDLER »

```
namespace System.Windows
{
    ...public delegate void RoutedEventHandler(object sender, RoutedEventArgs e);
}
```

```
namespace System.Windows
{
    ...public class RoutedEventArgs : EventArgs
    {
        ...public RoutedEventArgs();
        ...public RoutedEventArgs(RoutedEvent routedEvent);
        ...public RoutedEventArgs(RoutedEvent routedEvent, object source);

        ...public bool Handled { get; set; }
        ...public object OriginalSource { get; }
        ...public RoutedEvent RoutedEvent { get; set; }
        ...public object Source { get; set; }

        ...protected virtual void InvokeEventHandler(Delegate genericHandler, object genericTarget);
        ...protected virtual void OnSetSource(object source);
    }
}
```

En WPF, par contre, nous retrouverons les événements basés sur le délégué « RoutedEventHandler ».

Ce délégué reçoit également deux paramètres :

- « sender » est de type « System.Object », il correspond à l'objet qui déclenche l'événement.
- « e » est de type « RoutedEventArgs », il contient les données d'événement en plus du fonctionnement par défaut.

Ces événements seront vu plus en avant dans le cadre du WPF.



# EXERCICES

# EXERCICES (MAXIMUM 20')

Dans la classe « Compte » :

1. Ajoutez un événement appelé « PassageEnNégatifEvent » dont le délégué (« PassageEnNegatifDelegate ») devra recevoir en paramètre un objet de type « Compte » et ne rien renvoyer.

Au niveau de la classe « Courant » :

1. Déclencher l'événement « PassageEnNegatifEvent » si le compte passe en négatif et uniquement dans ce cas.

Au Niveau de la classe « Banque » :

1. Ajouter une méthode qui traitera l'événement « PassageEnNegatifAction » en affichant dans a console « Le numéro de compte {Numero} vient de passer en négatif ».



# LES GÉNÉRIQUES

C# - ORIENTÉ OBJET

- Introduction
- Les avantages
- Les paramètres de type générique
- Ajouter des contraintes
- Les contraintes « naked »
- La répétition de contraintes
- La surcharge de types
- Les délégués génériques
  - « Action »
  - « Func »

## LES CLASSES GÉNÉRIQUES

# INTRODUCTION

Les génériques ont été ajoutés à la version 2.0 du langage C# et du Common Language Runtime (CLR).

Ils introduisent le concept de paramètres de type, qui rendent possible la réception d'un ou de plusieurs types qui différeront à la déclaration et à l'instanciations de nos variables.

Les génériques peuvent être utilisés dans la déclaration de nos structures, de nos classes, de nos interfaces et de nos délégués ainsi que de nos méthodes.

```
namespace CoursCSharpFondements
{
    public delegate TResult MonDelegate<TArg0, TArg1, TResult>(TArg0 Param1, TArg1 Param2);

    public class MonGenerique<T>
    {
        public void MaMethode<U>(T Param1, U Param2)
        {
        }

        private struct MaStruct<T, Z>
        {
            public T Key { get; set; }

            public void UneAutreMethode(Z Param1)
            {
            }
        }
    }
}
```

# LES AVANTAGES

```
namespace CoursCSharpFondements
{
    using System.Collections;

    public class IntList
    {
        private ArrayList _Items;

        public int this[int index] {
            get {
                return (int)_Items[index]; //Casting Explicite
            }
        }

        public void Add(int item) {
            _Items.Add(item); //Casting implicite
        }
    }

    public class StringList
    {
        private ArrayList _Items;

        public string this[int index] {
            get {
                return (string)_Items[index]; //Casting Explicite
            }
        }

        public void Add(string item) {
            _Items.Add(item); //Casting implicite
        }
    }
}
```

Avec les génériques nous allons pouvoir combiner un haut niveau de réutilisabilité, de sécurité de type et d'efficacité sans commune mesure avec leurs homologues non génériques.

Le cas d'étude le plus flagrant est l'utilisation des collections non générique comme « ArrayList ».

En effet, si nous voulions limiter le type d'objet utilisable dans nos collections, nous devons créer nos propres classes.

Chaque classe travaillant avec un type où nous devons faire les casting nécessaires au bon fonctionnement de notre collection.

# LES PARAMÈTRES DE TYPE GÉNÉRIQUE

Grâce aux génériques, nous n'aurons plus toutes ces classes à définir.  
Une seule suffira.

Pour déclarer un ou plusieurs types génériques, nous utiliserons les chevrons ('<' et '>') entre lesquels nous spécifierons le nombre et le nom des types génériques séparés par des virgules.

```
namespace CoursCSharpFondements
{
    public delegate TResult MonDelegate<TArg0, TArg1, TResult>(TArg0 Param1, TArg1 Param2);

    public class MonGenerique<T>
    {
        public void MaMethode<U>(T Param1, U Param2)
        {
        }

        private struct MaStruct<T, Z>
        {
            public T Key { get; set; }

            public void UneAutreMethode(Z Param1)
            {
            }
        }
    }
}
```

```
namespace CoursCSharpFondements
{
    using System.Collections;

    public class List<T>
    {
        private ArrayList _Items;

        public T this[int index] {
            get {
                return (T)_Items[index]; //Casting Explicite
            }
        }

        public void Add(T item) {
            _Items.Add(item); //Casting implicite
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            List<int> IntList = new List<int>();
            List<string> StringList = new List<string>();
        }
    }
}
```



# AJOUTER DES CONTRAINTES

Contrainte	Description
where T : struct	Doit être de type valeur
where T : class	Doit être de type référence
where T : new()	Doit posséder un constructeur sans paramètre
where T : nom de classe	Doit hériter de la classe spécifiée
where T : nom d'interface	Doit implémenter l'interface spécifiée

Lorsque nous définissons une classe générique, nous pouvons appliquer une ou plusieurs restrictions aux types génériques.

Si nous essayons d'instancier notre classe à l'aide d'un type qui n'est pas autorisé par une contrainte, il en résultera une erreur de compilation.

Ces restrictions sont appelées des contraintes et sont spécifiées à l'aide du mot clé contextuel « **where** ».

# AJOUTER DES CONTRAINTES

Quelques exemples :

```
namespace CoursCSharpFondements
{
    using System;

    public class UneClasse<T>
        where T : class, IDisposable, ICloneable
    {
    }

    public class UneAutreclasse<T>
        where T : UneClasse<T>, IDisposable, new()
    {
    }

    public class EncoreUneAutreclasse<T>
        where T : struct
    {
    }
}
```

# LES CONTRAINTES « NAKED »

Les contraintes dites « **Naked** » signifie que nous allons utiliser un type générique dans la contrainte.

Ce type de contrainte est principalement utilisé afin de spécifier une notion d'héritage entre deux type générique.

```
namespace CoursCSharpFondements
{
    using System.Collections;

    public class List<T>
    {
        private ArrayList _Items;

        public T this[int index] {
            get {
                return (T)_Items[index]; //Casting Explicite
            }
        }

        public void Add<U>(U item)
            where U : T
        {
            _Items.Add(item); //Casting implicite
        }
    }
}
```

# LA RÉPÉTITION DE CONTRAINTES

Le type 'U' doit être un type référence afin d'être utilisé comme paramètre 'T' dans le type ou la méthode générique 'CoursCSharpFondements.MaClasse<T>'

```
namespace CoursCSharpFondements
{
    public class MaClasse<T>
        where T : class
    {
    }

    public class UneAutreClasse<U> : MaClasse<U>
    {
    }

    public class EncoreUneAutreClasse<U>
    {
        public MaClasse<U> X { get; set; }
    }
}
```

Lorsque nous utiliserons un type générique pour déclarer une variable de type générique également, si le type d'origine spécifie des contraintes, nous serons obligé de répéter, au moins, ses contraintes au niveau de notre objet.

```
namespace CoursCSharpFondements
{
    public class MaClasse<T>
        where T : class
    {
    }

    public class UneAutreClasse<U> : MaClasse<U>
        where U : class
    {
    }

    public class EncoreUneAutreClasse<U>
        where U : class
    {
        public MaClasse<U> X { get; set; }
    }
}
```



# LA SURCHARGE DE TYPES

De plus, nous pourrions définir plusieurs fois le type tant que celui-ci contiendra un nombre différent de paramètres génériques.

Ce concept est par exemple utilisé avec deux délégués spécifiques « Action » et « Func » qui reviendront régulièrement dans les cours à venir.

Ces deux délégués génériques nous permettront, à l'avenir, de ne plus déclarer nos propres délégués pour les utiliser dans nos événements ou nos variables.

```
namespace CoursCSharpFondements
{
    public class MaClasse
    {
    }

    public class MaClasse<T>
    {
    }

    public class MaClasse<T, U>
    {
    }

    class Program
    {
        static void Main(string[] args)
        {
            MaClasse ma1 = new MaClasse();
            MaClasse<int> ma2 = new MaClasse<int>();
            MaClasse<int, string> ma3 = new MaClasse<int, string>();
        }
    }
}
```

# LES DÉLÉGUÉS GÉNÉRIQUE – « ACTION »

Le délégué générique « Action » est associé aux procédures, il peut recevoir entre 0 et 16 paramètre(s) de types différents en entrée.

```
namespace System
{
    public delegate void Action();
    public delegate void Action<in T1>(T1 arg1);
    public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
    public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
    ...
    public delegate void Action<in T1, in T2, in T3 /*, in T4, ...*/, in T16>(T1 arg1, T2 arg2, T3 arg3/*, T4 arg4, ...*/, T16 arg3);
}
```

# LES DÉLÉGUÉS GÉNÉRIQUE – « FUNC »

Le délégué générique « Func » quant à lui est associé aux fonctions, il peut recevoir entre 1 et 17 paramètre(s) génériques.

Dans son cas le dernier paramètre spécifier est toujours le type de retour.

```
namespace System
{
    public delegate TResult Func<out TResult>();
    public delegate TResult Func<in T1, out TResult>(T1 arg1);
    public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
    ...
    public delegate TResult Func<in T1, in T2, in T3 /*, in T4, ...*/, in T16, out TResult>(T1 arg1, T2 arg2, T3 arg3/*, T4 arg4, ...*/, T16 arg3);
}
```



# EXERCICES



# EXERCICES (MAXIMUM 20')

Dans la classe « Compte » :

1. Supprimez la déclaration du type délégué « PassageEnNegatifDelegate »
2. Changez le type de l'événement appelé « PassageEnNégatifEvent » en utilisant les délégué « Action » ou « Func » afin que notre code continue à fonctionner.



# RÉFÉRENCES

C# - LES FONDEMENTS

- O'Reilly :  
C# 4.0 in a nutshell (ISBN-13 : 978-0-596-80095-6)  
C# 5.0 in a nutshell (ISBN-13 : 978-1-4493-2010-2)
- Microsoft :  
Spécification du langage C# 5.0 ([http://msdn.microsoft.com/fr-fr/library/vstudio/ms228593\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/vstudio/ms228593(v=vs.110).aspx))