



PROGRAMMATION .NET C#

Wilfart Emmanuel
1^{ière} informatique

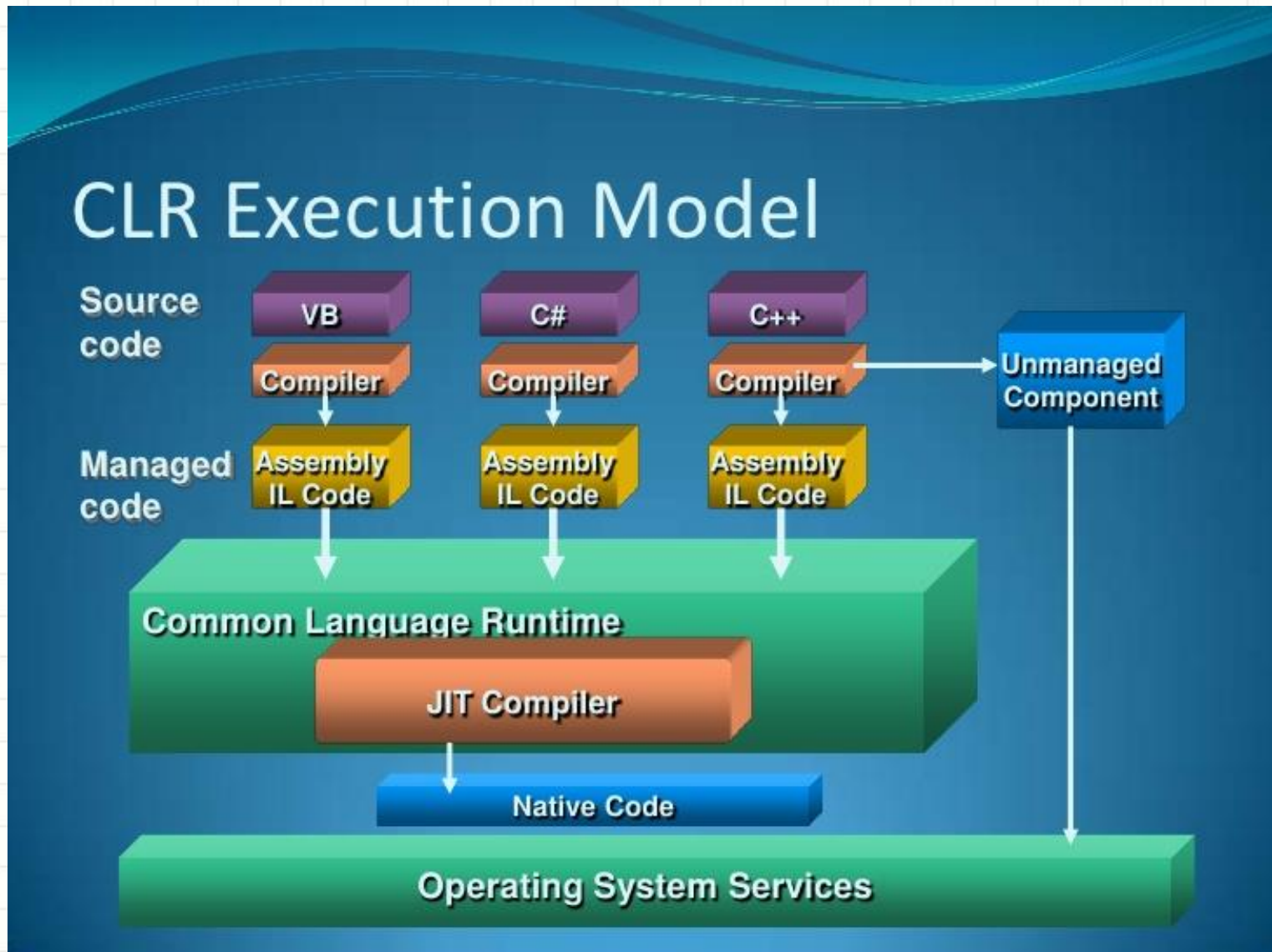
Technologie .NET

- Appelée aussi Framework .NET elle apparaît en 2000.
- Offre un environnement de développement unifié (desktop, web, mobile)
- Permet de développer des applications indépendantes du système d'exploitation et du matériel (CrossPlatform)

Technologie .NET

- Intègre le CLR (Common Language Runtime)
 - JIT (Just In Time Compileur)
 - Gestion de la mémoire
 - Les métadonnées
 - Les assemblages et les manifestes (nom, version, liste des fichiers, dépendances)
- Ensemble hiérarchique de ressources (Réseau, collections, thread, bases de données...)
- Langage intermédiaire (MSIL)
- Langages de programmation multiples (C#, C++.net, vb.net, F#...)

Technologie .NET



Historique des versions

C#	1.0	1.2	2.0	3.0	3.0	4.0	5.0
C# Features			1. Generics 2. Partial types 3. Anonymous methods 4. Iterators 5. Nullable types 6. Private setters (properties) 7. Method group conversions (delegates)	1. Implicitly typed local variables 2. Object and collection initializers 3. Auto-Implemented properties 4. Anonymous types 5. Extension methods 6. Query expressions 7. Lambda expressions 8. Expression trees 9. Partial Methods		1. Dynamic binding 2. Named and optional arguments 3. Generic co- and contravariance 4. Embedded interop types ("NoPIA")	1. Asynchronous methods 2. Caller info attributes
.net framework	1.0	1.1	2.0	3.0	3.5	4.0	4.5
Framework Year	13-feb-2002	24-apr-2003	7-nov-2005	6-nov-2006	19-nov-2007	12-apr-2010	15-aug-2012
CLR	1.0	1.1	2.0	2.0	2.0	4.0	4.5
CLR New libraries	Original version	First update	Rewrite of Framework	WCF, WPF, WF	LINQ	Parallel Extensions	Asynchronous programming model
Development IDEs /Tools	VS .NET	VS .NET 2003	VS 2005	Expression Blend	VS 2008	VS 2010	VS 2012
bundled with OS	N/A	Win Server 2003	Windows Server 2003 R2	Windows Vista, Windows Server 2008	Windows 7, Win Server 2008 R2	N/A	Windows 8 , Windows Server 2012

Multi plateforme

- Avant 2014 Microsoft développe uniquement un Framework pour ses environnements.
- Nous retrouvons des projets indépendants:
 - Mono sous les environnements Linux
 - Mono donne naissance à Xamarin assurant la portabilité de mono sous Android et IOS
 - En 2016 Microsoft rachète Xamarin
 - Mise à disposition de Xamarin dans Visual Studio Community
- En 2014 Microsoft ouvre son Framework .NET et le fait passer en Open Source (Version 4.6 du Framework)

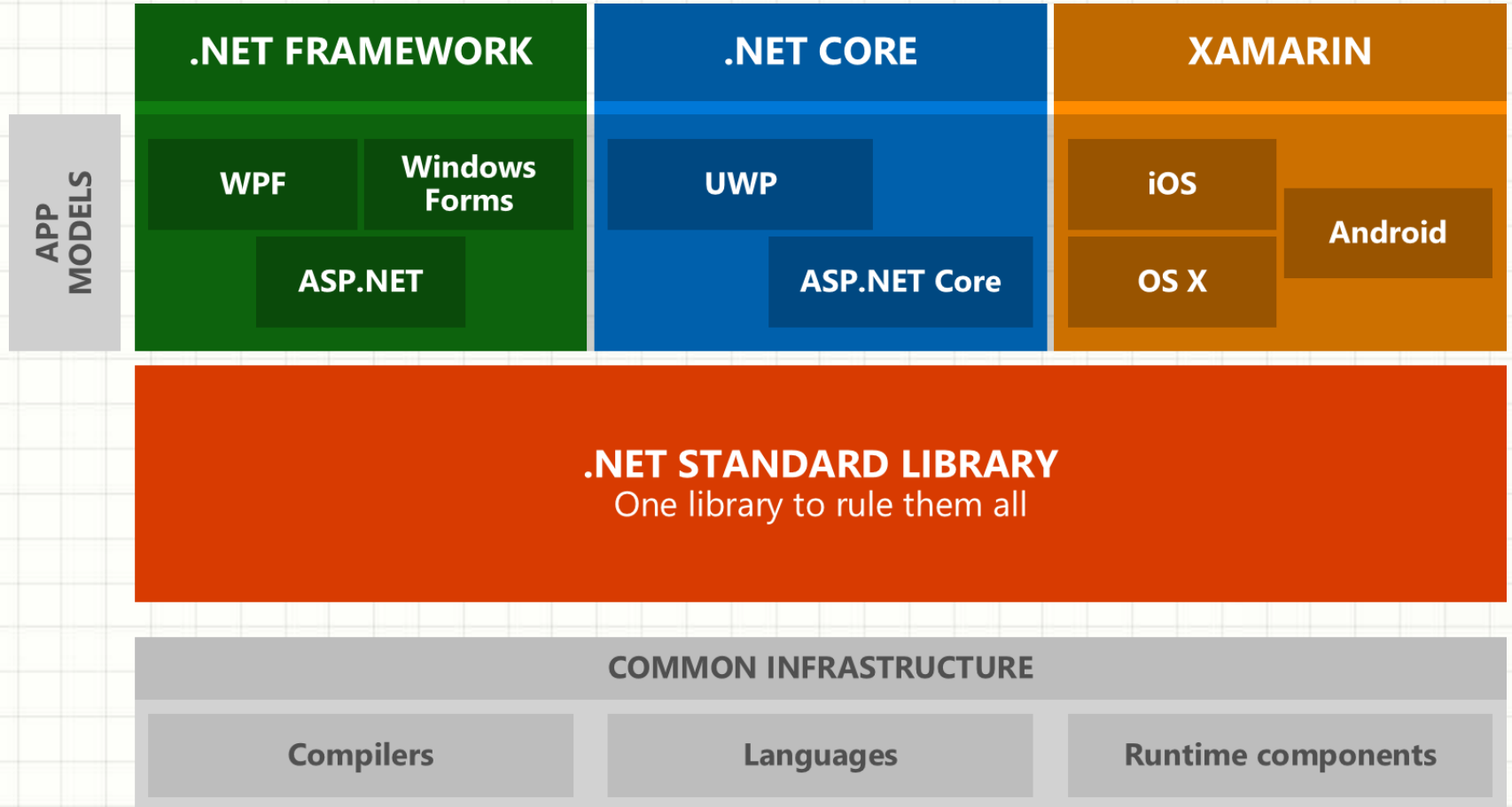
Multi plateforme

- Le .NET Core
 - Le .NET Core est une version modulaire du .NET Framework portable sur plusieurs plateformes
 - Le .NET Core est open source et accepte les contributions de la Communauté

Multi Platforme



Multi platforme



Les différentes versions

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	vNext
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	vNext
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	vNext
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	vNext	vNext	vNext
Windows	8.0	8.0	8.1					
Windows Phone	8.1	8.1	8.1					
Windows Phone Silverlight	8.0							

Historique des versions C#

- C# 1.0 Première version, code managé
- C# 2.0
 - type générique
 - Méthodes anonymes
 - Type Nullable
 - Classes partielles
 - Co et contra-variance pour les délégués
- C# 3.0
 - Expression lambda (=>)
 - Méthodes d'extension
 - Types anonymes
 - Linq
 - Type implicite (var)

Historique des versions C#

- **C# 4.0**
 - Arguments nommés
 - Paramètres optionnels
 - Co et contra-variance générique
- **C# 5.0**
 - Programmation asynchrone
 - Caller information
- **C# 6.0**
 - Initialiseurs de propriétés automatiques
 - Imports statiques
 - expression-bodied members (Simplification dans la déclaration des méthodes et propriétés)
 - Initialiseurs de dictionnaires et de membres indexés

Historique des versions C#

- C# 6.0

- await dans les blocs catch et finally
- Filtres d'exception
- Propagation de null

- C# 7.0

- Fonctions locales
- Tuples (retour de données multiples dans les fonctions)
- Déconstructeurs (décomposition des Tuples)
- Pattern matching (notamment utilisation du mot clef is)
- Variables out

C# Langage orienté objets

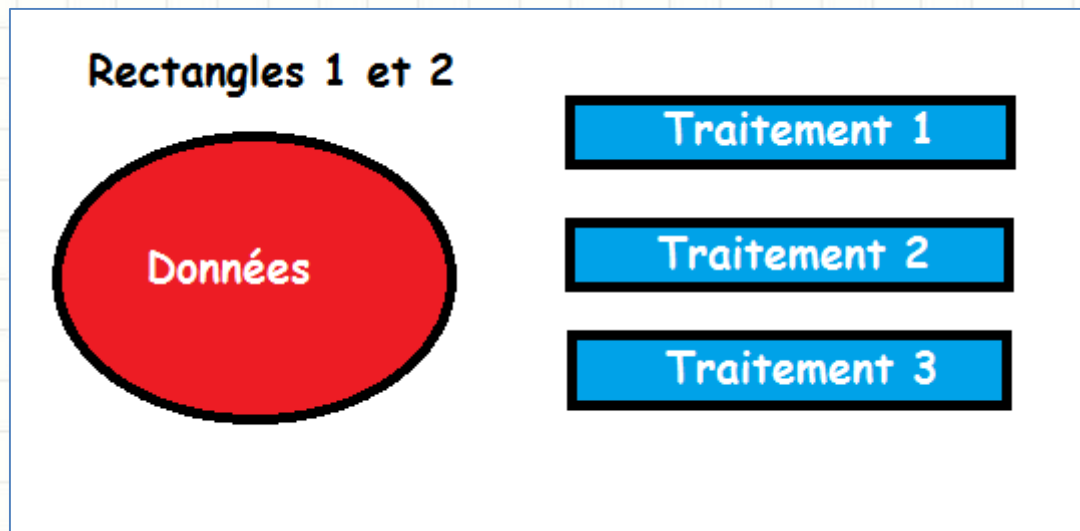
- Procédural ou orienté objet?
 - Le procédural?

```
float CalculSurfaceRectangle(float petit_cote, float grand_cote)
{
    return petit_cote*grand_cote;
}

void main()
{
    float petit_cote_1=1.1;
    float grand_cote_1=2.2;
    float petit_cote_2=3.3;
    float grand_cote_2=4.4;
    float surface1 = CalculSurfaceRectangle(petit_cote_1, grand_cote_1);
    float surface2 = CalculSurfaceRectangle(petit_cote_2, grand_cote_2);
    printf("Surface1: %f\n", surface1);
    printf("Surface2: %f\n", surface2);
    system("pause");
}
```

C# Langage orienté objets

– Le procédural

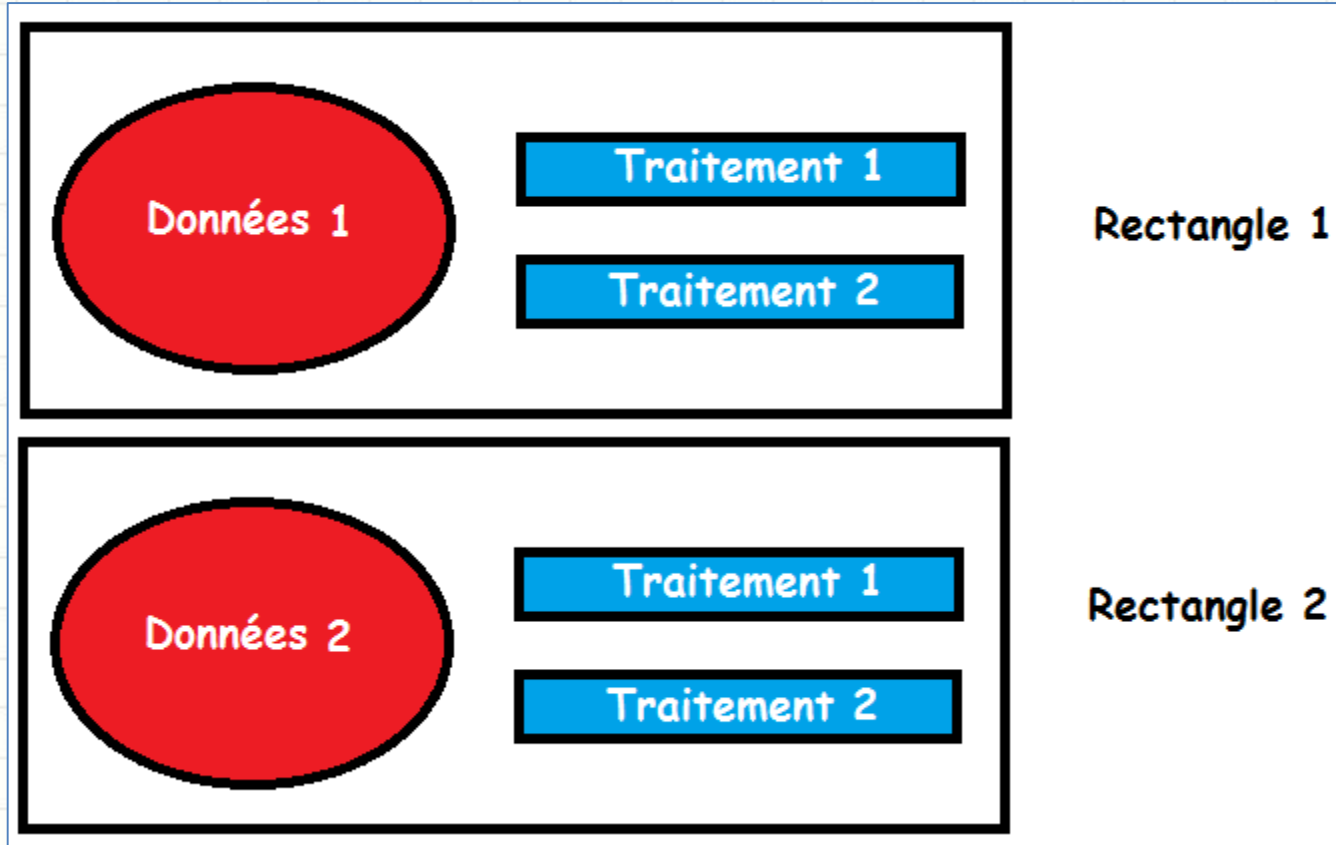


C# Langage orienté objets

- L'orienté objet.

```
class Rectangle
{
    float _PetitCote;
    float _GrandCote;
    public Rectangle(float PetitCote,float GrandCote)
    {
        _PetitCote = PetitCote;
        _GrandCote = GrandCote;
    }
    public float CalculSurface()
    {
        return _PetitCote * _GrandCote;
    }
}
static void Main(string[] args)
{
    Rectangle Rect1 = new Rectangle(1.1F, 2.2F);
    Console.WriteLine(Rect1.CalculSurface());
}
```

C# Langage orienté objets



Classes et objets

- Définition d'une classe (source Microsoft)

Une classe est une construction qui vous permet de créer vos propres types personnalisés en regroupant des variables d'autres types, méthodes et événements. Une classe est comme un moule.

```
public class Personne
{
    //membres, propriétés, méthodes...
}
```


Classes et objets

- Définition d'un objet

Un programme peut créer de nombreux objets de la même classe (à partir du même moule). Les objets sont également appelés instances. Ils peuvent être stockés dans une variable, dans un tableau ou dans une collection

```
Personne Andre;  
Personne Dupond;  
Personnes Eric;
```

Classe et objets



Classe Personne

Nom
Prénom
Age
Sexe



Objet Dupond

Nom	Dupond
Prénom	Eric
Age	99
Sexe	M

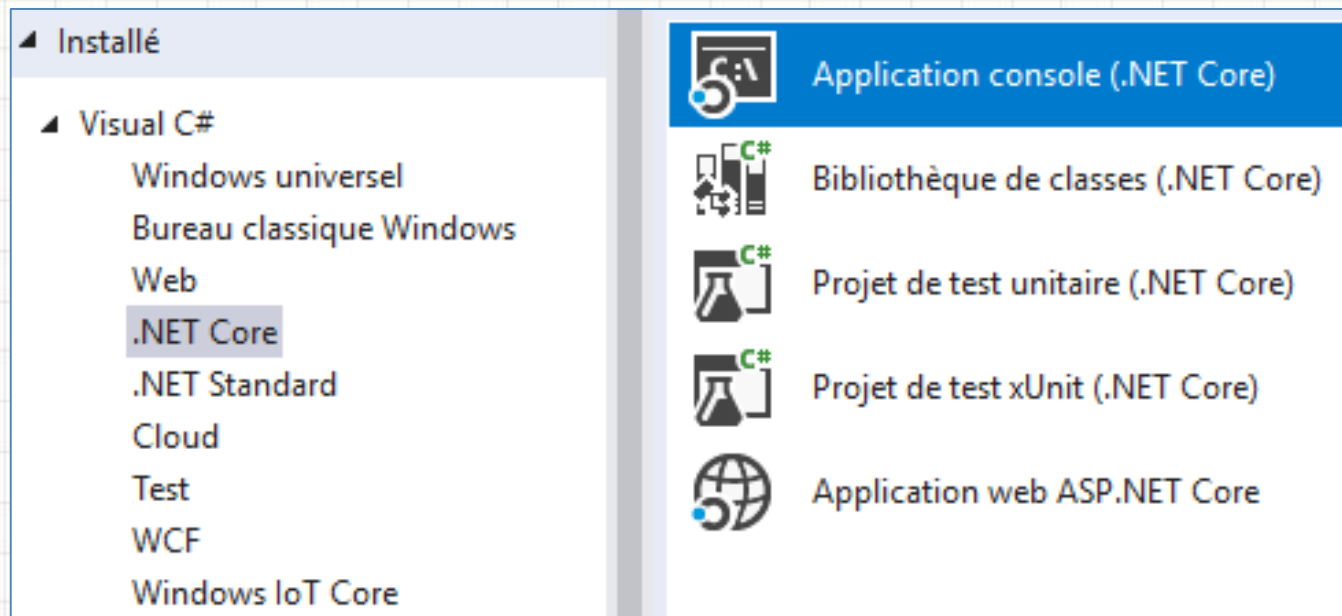
Programmation C#

- Outil et environnement utilisés
 - Nous utiliserons Visual Studio 2017 Community Edition version 15.3.0
 - Nous développerons nos applications en mode console avec le .NET Core 2.0

Programmation C# - Console

- Création du squelette de base de l'application

Nous prendrons dans la barre de menu supérieure... Fichier-Nouveau Projet



Programmation C# - Console

```
using System;

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

Nous abordons dans les diapositives suivantes chaque partie de ce code de base pour bien en comprendre les mots clefs importants.

Programmation C#

- namespace et using

Le Framework propose des ressources dans un environnement hiérarchisé. Prenons la ligne de code suivante:

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

C'est une forme simplifiée et elle pourrait aussi s'écrire sous sa forme complète

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

Cette forme reprend toute l'arborescence. Elle démarre par l'espace de nom System dans lequel nous retrouvons une classe statique `Console` dans laquelle la méthode `WriteLine` permet l'envoi de la chaîne de caractères "Hello World!".

Nous pouvons éviter cette forme complète en renseignant des espaces de noms de recherche. Il suffit de renseigner au début du code `using System;`

Programmation C#

- class Program et void Main

Lorsque vous exécutez une application, il y a un seul point d'entrée dans votre structure de programme. Ce point d'entrée est la méthode

```
static void Main(string[] args){ //.....}
```

Le code présent entre l'ouverture d'accolade et la fermeture d'accolade est exécuté et lorsque la fermeture d'accolade est atteinte, l'application se ferme

Programmation C#

- Syntaxe générale du langage

- Respecter les majuscules et minuscules

```
int a=10;  
int A=30;  
Console.WriteLine(a);  
Console.WriteLine(A);
```

- Chaque ligne de code se termine par un ;
- Un bloc de lignes de code s'identifie par des accolades { //..... }
- Dans notre programme de base on retrouve un bloc de lignes de code associé à la méthode Main
- Des commentaires peuvent être ajoutés dans votre code. On identifie une ligne de commentaire lorsqu'elle démarre par //

Programmation C#

- Apprentissage du langage
 - Les variables de type valeur (variables contenant des données simples)
 - Les variables de type référence (présence du mot clef new pour créer des variables contenant des données complexes)
 - Les opérateurs (unaires, binaires, ternaires, relationnels, mathématique etc...)
 - Les instructions
 - Les expressions
 - Les méthodes

Nous aborderons la programmation orientée objet une fois les bases du langage acquises.

Programmation C#

- Les variables de types valeur prédéfinis

Une donnée quelconque doit être placée dans un emplacement mémoire. Nous devons déclarer cet emplacement en précisant le type de la donnée et en donnant un nom à cet emplacement

```
static void Main(string[] args)
{
    int compteur;
    compteur = 10;
    Console.WriteLine("Valeur du compteur:{0}", compteur);
    Console.ReadKey();
}
```

Nous retrouvons le mot clef `int` qui signifie que la variable est prévue pour contenir un nombre entier (pas de nombre réel). Nous donnons le nom `compteur` à cette variable. Ce nom nous permet de manipuler la variable.

Ex: `compteur = 10;` Le signe `=` est l'opérateur d'affectation. On place la valeur 10 dans la variable `compteur`.

Programmation C#

- Les types entier

Nom	CTS	Description	Valeurs possibles
sbyte	System.SByte	Entier signé sur 8 bits	-128:127
short	System.Int16	Entier signé sur 16 bits	-32768:32767
int	System.Int32	Entier signé sur 32 bits	$-2^{31}:2^{31}-1$
long	System.Int64	Entier signé codé sur 64bits	$-2^{63}:2^{63}-1$
byte	System.Byte	Entier non signé codé sur 8 bits	0:255
ushort	System.UInt16	Entier non signé sur 16 bits	0:65535
uint	System.UInt32	Entier non signé sur 32 bits	$0:2^{32}-1$
ulong	System.UInt64	Entier non signé codé sur 64 bits	$0:2^{64}-1$

Programmation C#

- Les types flottant (nombres réels)

Nom	CTS	Description	Valeurs possibles
float	System.Single	Nombre en virgule flottante sur 32 bits en simple précision	
double	System.Double	Nombre en virgule flottante sur 64 bits en double précision	

- Le type décimal (nombres réels)

Nom	CTS	Description	Valeurs possibles
decimal	System.Decimal	Nombre en virgule flottante sur 128 bits en haute précision	

Programmation C#

- Le type char

Nom	CTS	Description
char	System.Char	représente un seul caractère codé sur 16 bits(Unicode)

- Autres types: struct et enum

Ces autres types seront abordés lorsque nous évoquerons l'orienté objet du C#

Programmation C#

- Les valeurs par défaut des variables locales.

Le C# n'accepte pas que vous utilisiez des variables locales sans qu'elles ne soient initialisées (utilisation de l'opérateur d'affectation).

```
int compteur=10;  
bool resultat=false;
```

- Les constantes et leur type.

Une constante entière est considérée par le compilateur comme de type int

Une constante réelle est considérée par le compilateur comme de type double.

Il sera donc nécessaire d'adapter ces constantes par rapport au type de la variable. Un entier long ne peut tenir dans un entier court par exemple

Programmation C#

- Les constantes et leur type

```
uint x = 123U;  //U comme unsigned
long y = 1235L; //L comme long
ulong z = 256UL //UL unsigned long
```

```
decimal a = 13.20M;
float b = 15.75F;  //F comme float
```

- La portée des variables

Ce que l'on entend par portée, c'est la partie de code dans laquelle une variable est accessible.

Une variable locale est visible uniquement dans la partie de code où elle a été déclarée, ce qui correspond à la fermeture de l'accolade du bloc où elle a été déclarée.

Programmation C#

- Les types référence

Ce typage trouve tout son intérêt dans l'orienté objet. Nous nous limiterons à un type qui pourrait nous faire penser à un type valeur mais qui est en réalité un type référence. C'est le type `string`

```
static void Main(string[] args)
{
    string Nom = "Dupond";
    Console.WriteLine("Nom:{0}", Nom);
    Console.ReadKey();
}
```


Programmation C#

- Les opérateurs arithmétiques binaires

+	Effectue une addition	$a + b$
-	Effectue une soustraction	$a - b$
*	Effectue une multiplication	$a * b$
/	Effectue une division	a / b
%	Effectue le reste d'une division (modulo)	$a \% b$

- Les opérateurs arithmétiques unaires

++	Effectue une incrémentation	$a++$ ou $++a$
--	Effectue une décrémentation	$a--$ ou $--a$

Programmation C#

- Les opérateurs arithmétiques avec affectation

`+=` Affectation de l'entier par une addition

`a+=b` identique à `a=a+b`

`-=` Affectation de l'entier par une soustraction

`a-=b` identique à `a=a-b`

`*=` Affectation de l'entier par une multiplication

`a*=b` identique à `a=a*b`

`/=` Affectation de l'entier par une division

`a/=b` identique à `a=a/b`

`%=` Affectation de l'entier par un reste d'une division (modulo)

`a%=b` identique à `a=a%b`

- Les opérateurs relationnels

`==` Comparaison d'une égalité

`a == b`

`!=` Comparaison d'une différence

`a != b`

`>` Comparaison de plus grand que

`a > b`

`>=` Comparaison de plus grand ou égal que

`a >= b`

`<` Comparaison de plus petit que

`a < b`

`<=` Comparaison de plus petit ou égal que

`a <= b`

Programmation C#

- Les opérateurs relationnels

Ces opérateurs donnent comme résultat un booléen qui aura la valeur vraie (true) ou fausse (false). Ces valeurs seront en général utilisées dans des instructions conditionnelles telle que if (si condition vraie)

```
float resultat=57.0F;  
if (resultat>=50)  
{  
    Console.WriteLine("Réussite");  
}
```

Programmation C#

- Les opérateurs logiques

&& Effectue un «et logique»

a && b

|| Effectue un «ou logique»

a || b

Ces opérateurs sont souvent utilisés avec les opérateurs relationnels.

```
float resultat=57.0F;  
if (resultat>=60 && resultat<70)  
{  
    Console.WriteLine("Réussite avec satisfaction");  
}
```

Programmation C#

- Les opérateurs logiques bit à bit

<<	Effectue un décalage à gauche	a << b
>>	Effectue un décalage à droite	a >> b
~	Effectue une «négation binaire»	~ a
&	Effectue un «Et binaire»	a & b
	Effectue un «Ou binaire»	a b
^	Effectue un «Ou exclusif binaire»	a ^ b

- Les opérateurs de chaînes de caractères

chaine1 + chaine2	Effectue la concaténation de chaine de caractères
chaine1 == chaine2	Comparaison d'un égalité
chaine1 != chaine2	Comparaison d'une différence
chaine1 += chaine2	Affectation d'une concaténation de la chaine d'origine

Programmation C#

- Les conversions de type implicite

Lorsque l'on désire copier le contenu d'une variable dans une autre sans risque de perdre une partie du contenu, la conversion est acceptée et dite implicite

```
float a;  
int b = 10;  
long c;  
a = b;  
c = b;
```

- Les conversions de type explicite

Lorsque il y a des risques de perte de données, le compilateur va générer une erreur. Si vous êtes conscient de cette perte de données ou de ce risque de perte alors vous pouvez utiliser le transtypage explicite pour éviter cette erreur de compilation

Programmation C#

- Les conversions de type explicite

```
float montant = 100.5F;  
int x = montant;
```

✖ CS0266 Impossible de convertir implicitement le type 'float' en 'int'. Une conversion explicite existe (un cast est-il manquant ?)

```
float montant = 100.5F;  
int x = (int) montant;
```

```
long montant = 100;  
int x = montant;
```

✖ CS0266 Impossible de convertir implicitement le type 'long' en 'int'. Une conversion explicite existe (un cast est-il manquant ?)

Programmation C#

- Les conversions numérique vers chaîne

Afin de comprendre les techniques mises en place, il faut se mettre dans l'esprit qu'en C# tout est objet. Depuis le début de ce cours, nous travaillons avec des objets sans le savoir. Pour rappel, nous retrouvons dans la programmation orientée objet le mot clef `class` pour définir la structure des objets de ce type. Dans une classe nous retrouverons des membres, des méthodes, des propriétés....

Le mot clef `class` aura comme conséquence que l'objet créé sera de type référence (usage de l'opérateur `new`)

Nous pouvons remplacer le mot clef `class` par le mot clef `struct` qui permet aussi de créer un objet mais ce sera un type valeur (sans l'usage du mot clef `new`)

L'accès à une méthode, une propriété ou un membre de la classe (ou de la structure) à partir d'un objet se fera par l'opérateur `.` (point)

Nom de l'objet.Nom de la méthode (ou propriété ou membre)

Programmation C#

- Les conversions numérique vers chaîne

Créons dans notre fonction Main un entier et laissons le curseur de souris sur le mot clef `int`. Nous voyons apparaître une info bulle

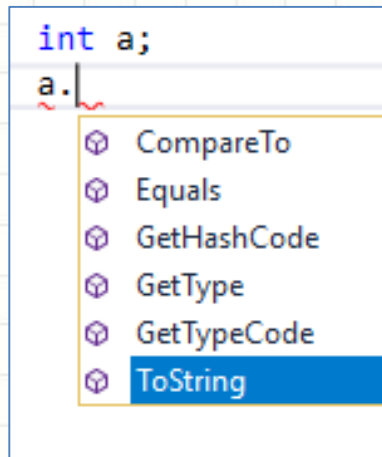
```
static void Main(string[] args)
{
    int a;
}
```

■ `struct System.Int32`
Represents a 32-bit signed integer.

`int` est en fait le type CLR `System.Int32` qui est une structure. Donc `a` n'est pas une variable simple mais est un objet. Si c'est un objet, nous pouvons regarder si nous pouvons utiliser cet opérateur `.` (point)

Programmation C#

- Les conversions numérique vers chaîne



Nous retrouvons des méthodes. L'une d'entre elle s'appelle ToString. Elle se retrouve dans de nombreuses classes et comme son nom l'indique, elle permet de convertir cet objet vers une chaîne de caractères.

```
int a=10;  
string resultat = a.ToString();
```

L'utilisation d'une méthode nécessite l'usage des parenthèses à l'arrière du nom de la méthode. Dans les parenthèses nous retrouvons d'éventuels arguments.

Programmation C#

- Les conversions chaîne vers numérique

Si, en mode console, nous envisageons de demander l'introduction de données une des méthodes utilisée est `ReadLine`.

```
static void Main(string[] args)
{
    string data = Console.ReadLine();
}
```

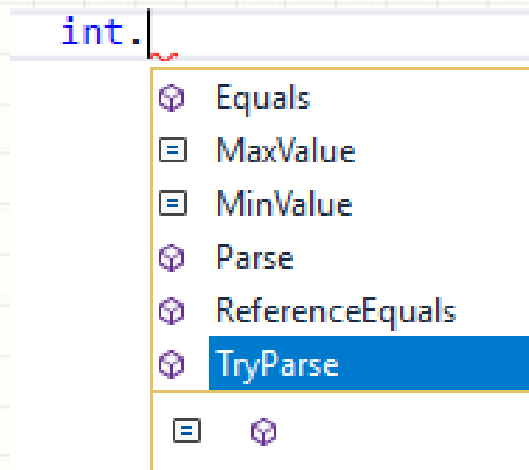
Cette méthode ne fournit qu'une chaîne de caractère. Que faire si nous souhaitons faire en sorte que l'on obtienne une donnée numérique? La seule solution est la conversion.

Nous avons vu que le type `int` est en fait une structure. Une variable créée sur base de ce type est donc un objet et la structure offre aux objets des méthodes (ex: `ToString`). La structure propose également des méthodes accessibles directement sans qu'une instantiation ne soit nécessaire. Ce sont les méthodes statiques (`static`)

Programmation C#

- Les conversions chaîne vers numérique

Nous prenons le mot clef `int` et dans l'interface de programmation nous ajoutons le `.` (point) à la suite de ce mot clef



En ce qui nous concerne, nous nous attacherons aux méthodes `Parse` et `TryParse` qui permettront de vérifier que la conversion est possible et l'autre effectuant la conversion.

Programmation C#

- Les conversions chaîne vers numérique

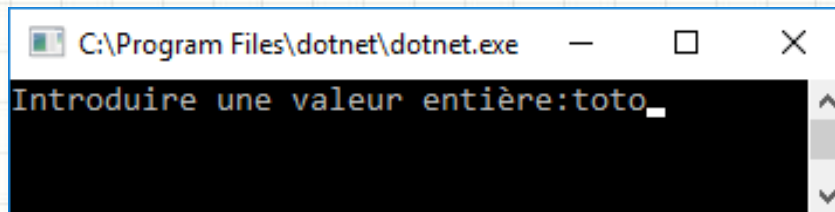
```
static void Main(string[] args)
{
    int resultat;
    Console.Write("Introduire une valeur entière:");
    string data = Console.ReadLine();
    resultat = int.Parse(data);
    Console.WriteLine("fin du programme");
    Console.ReadKey();
}
```

La méthode TryParse sera vue plus tard car elle comprend une notion qui n'est pas encore abordée. (Passages d'arguments par valeur et par référence)

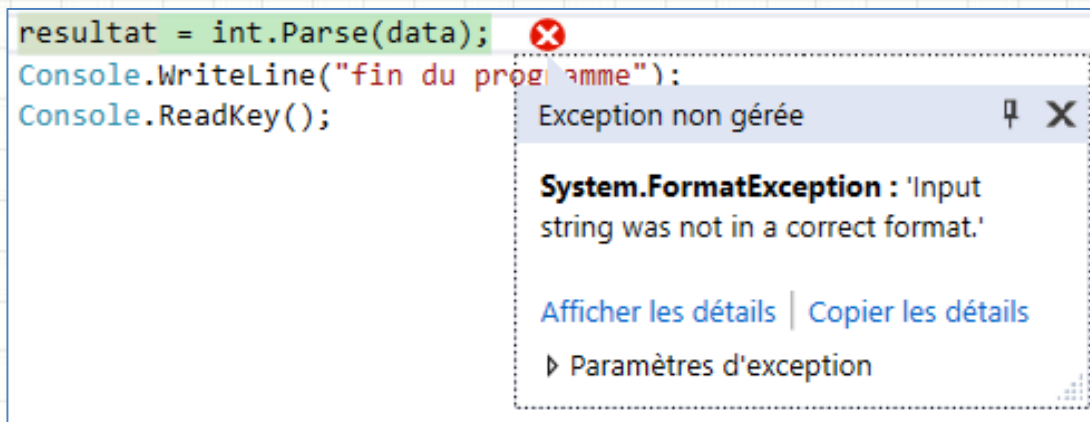
Certains vont probablement essayer d'entrer autre chose qu'une chaîne de caractères convertible en un entier. Analysons le comportement.

Programmation C#

- Les conversions chaîne vers numérique



```
C:\Program Files\dotnet\dotnet.exe
Introduire une valeur entière:toto_
```



```
resultat = int.Parse(data);
Console.WriteLine("fin du programme");
Console.ReadKey();
```

Exception non gérée

System.FormatException : 'Input string was not in a correct format.'

[Afficher les détails](#) | [Copier les détails](#)

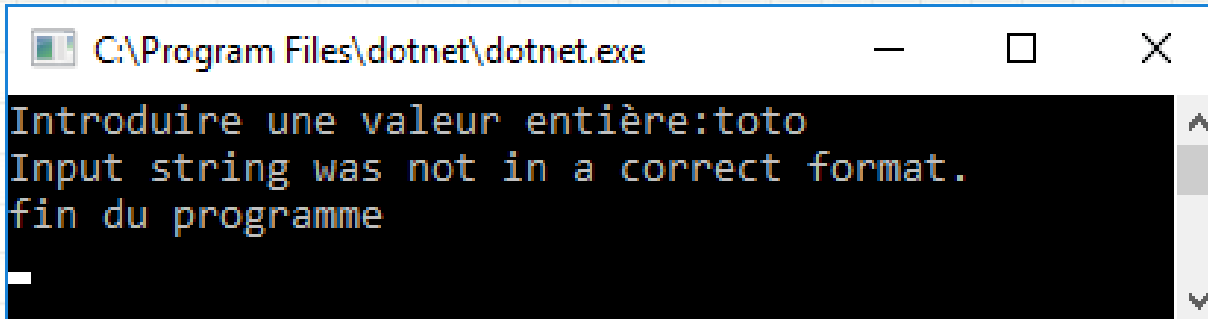
▸ Paramètres d'exception

Lorsqu'une erreur se produit dans votre programme, une exception est générée. Elle provoque la sortie inconditionnelle du code si vous ne faites rien (si elle n'est pas gérée). Nous allons voir comment gérer ces exceptions.

Programmation C#

- La capture des exceptions try....catch...finally

```
try
{
    string data = Console.ReadLine();
    resultat = int.Parse(data);
}
catch(FormatException ex)
{
    Console.WriteLine(ex.Message);
}
```



```
C:\Program Files\dotnet\dotnet.exe
Introduire une valeur entière:toto
Input string was not in a correct format.
fin du programme
_
```

Programmation C#

- La capture des exceptions try....catch...finally

Si nous mettons le curseur de souris sur la méthode Parse, nous aurons l'affichage suivant...

⊞ `int int.Parse(string s)` (+ 3 surcharges)

Converts the string representation of a number to its 32-bit signed integer equivalent.

Exceptions :

`ArgumentNullException`

`FormatException`

`OverflowException`

Nous remarquons la présence de plusieurs exception qui peuvent être générés par la méthode Parse. Comment peut-on capturer chacune de ces exceptions individuellement ou les gérer globalement?

Pour un seul try, nous pouvons retrouver plusieurs blocs catch individuels

Programmation C#

- La capture des exceptions try....catch...finally

```
catch(FormatException ex)
{
    Console.WriteLine(ex.Message);
}
catch (OverflowException ex)
{
    Console.WriteLine("Débordement");
}
catch(Exception ex)
{
    Console.WriteLine("Exception non prise en compte
précédemment");
}
```

Le dernier catch avec Exception prendra en compte les interruptions non reprises dans les blocs précédents

Programmation C#

- Exercice 1

Demander à l'utilisateur d'introduire deux entiers. (chaînes de caractères à convertir).

Additionner ces deux entiers et placer le résultat dans une troisième variable

Afficher le contenu de la troisième variable à l'écran.

Attention de gérer les exceptions.

C#. Les structures de contrôle

- Conditions if else

Nous pouvons utiliser cette instruction pour exécuter une instruction si une condition est vraie. (et éventuellement exécuter une autre instruction si elle est fausse)

if avec instruction simple

```
int age;  
string str_age;  
  
Console.Write("Introduire votre age:");  
str_age=Console.ReadLine();  
age=int.Parse(str_age);  
  
if (age >= 20) Console.WriteLine("Vous êtes majeur(e)");
```

C#. Les structures de contrôle

- Conditions if else

if avec un bloc d'instructions

```
if (age >= 20)
{
    Console.WriteLine("Vous êtes majeur(e)");
    Console.WriteLine("Vous pouvez signer ce document");
}
```

if avec un else

```
if (age >= 20) Console.WriteLine("Vous êtes majeur(e)");
else Console.WriteLine("Vous êtes mineur(e)");
```

C# Les structures de contrôle

- Instruction de sélection switch...case

```
switch (data)
{
    case 0:
        Console.WriteLine("Valeur 0");
        break;
    case 1:
        Console.WriteLine("valeur 1");
        break;
    default:
        Console.WriteLine("valeur autre que 1 ou 2");
        break;
}
```

Attention: le mot clef **break** est obligatoire à la fin de chaque section **case**
La section **default** est obligatoire

C# Les structures de contrôle

- Instruction de sélection switch...case

```
switch (data)
{
    case 0:
    case 1:
        Console.WriteLine("valeur 0 ou 1");
        break;
    default:
        Console.WriteLine("valeur autre que 1 ou 2");
        break;
}
```

Dans cet exemple, plusieurs sections sont associées à un même code. Dans notre exemple, le `break` du `case 0:` doit être omis.

C# Les structures de contrôle

- Instruction de sélection switch...case

```
switch (departement)
{
    case "comptabilite":
        Console.WriteLine("comptabilité");
        break;
    case "direction":
        Console.WriteLine("direction");
        break;
    default:
        Console.WriteLine("autre département");
        break;
}
```

Dans le langage C, le type des données associées à une instruction doit être de nature numérique et entière. En C# nous pouvons utiliser d'autres types comme des chaînes de caractères dans l'exemple précédent.

C# Les structures de contrôle

- Les boucles. Instruction for

```
for (int i = 0; i <= 9; i++)  
{  
    System.Console.WriteLine(i);  
}
```

L'utilisation des boucles est nécessaire dans les traitements itératifs c'est à dire des traitements qui doivent s'exécuter plusieurs fois mais sur des données différentes.

Dans cet exemple je souhaite afficher les nombres allant de 0 jusque 9. Le traitement se résume à un affichage. Nous pourrions mettre 10 lignes composées d'un `System.Console.WriteLine`

Attention: dans cet exemple, la variable `i` est locale uniquement à l'instruction `for`

C# Les structures de contrôle

- Les boucles. Instruction while

```
int i = 0;
while (i <= 9)
{
    System.Console.WriteLine(i);
    i++;
}
```

- Les boucles. Instruction do while

```
int i = 0;
do {
    System.Console.WriteLine(i);
    i++;
} while(i <= 9);
```

C# Les structures de contrôle

- Les boucles. `while` et `do ... while`

Dans une instruction `while` la condition est vérifiée avant d'exécuter une première fois le traitement. Si la condition est fausse le traitement ne sera pas exécuté.

Dans une instruction `do while` la condition est vérifiée après avoir exécuté au moins une fois le traitement.

C# Les structures de contrôle

- Les boucles. Instruction foreach

Cette instruction permet de parcourir les objets présents dans une collection sans avoir besoin d'indice. Nous comparerons cette instruction avec l'instruction for dans les exemples suivants

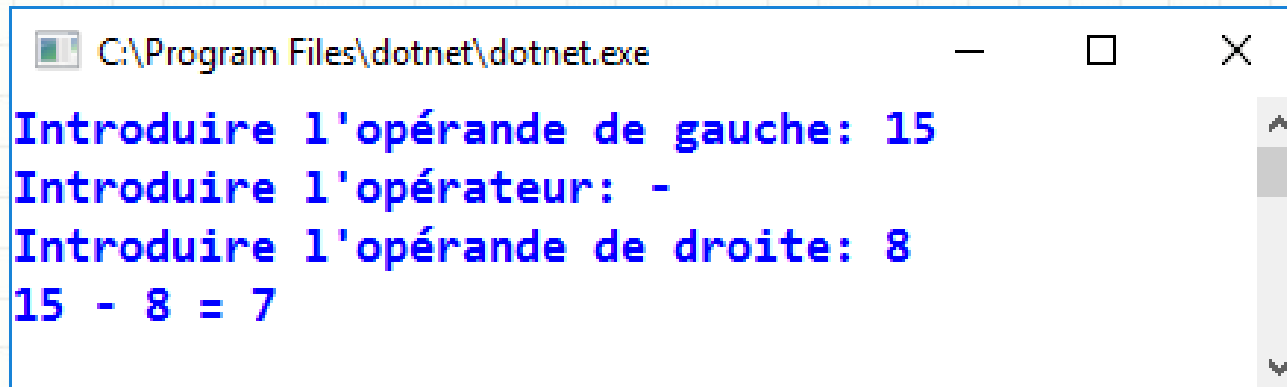
```
string[] mois = new string[] { "janvier", "février", "mars" };  
for (int i = 0; i < mois.Length; i++)  
{  
    Console.WriteLine(mois[i]);  
}
```

```
string[] mois = new string[] { "janvier", "février", "mars" };  
foreach(string data in mois)  
{  
    Console.WriteLine(data);  
}
```

C# Les structures de contrôle

- Exercice 1

Réaliser une calculatrice en mode console. Le programme vous invitera à introduire les valeurs des deux opérandes ainsi que l'opérateur et à vous fournira le résultat. Une fois le résultat fourni, le programme se fermera après confirmation



A screenshot of a Windows console window titled "C:\Program Files\dotnet\dotnet.exe". The window displays the following text in blue monospace font:

```
Introduire l'opérande de gauche: 15
Introduire l'opérateur: -
Introduire l'opérande de droite: 8
15 - 8 = 7
```

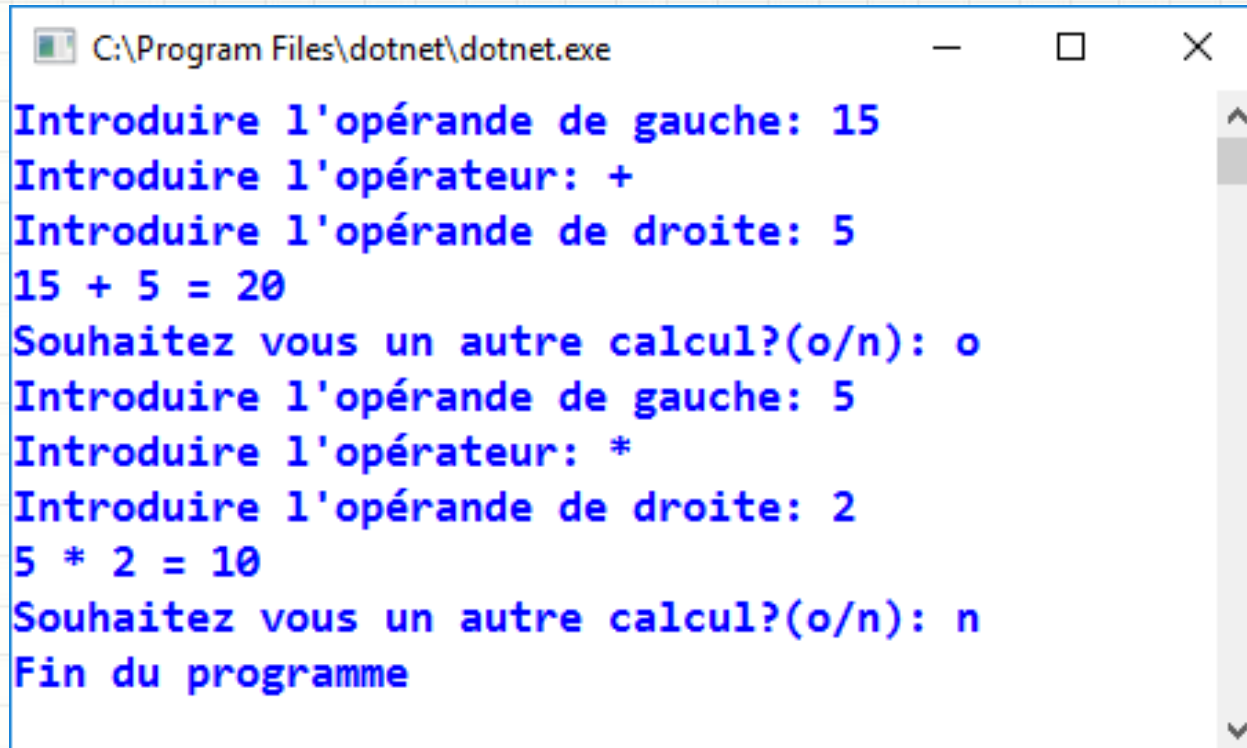
The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a vertical scrollbar on the right side.

C# Les structures de contrôle

- Exercice 2

Reprendre l'exercice 1 et l'adapter pour répondre au besoin suivant:

- Lorsque le premier résultat du calcul a été fourni, demander à l'utilisateur s'il désire effectuer un autre calcul ou sortir du programme.

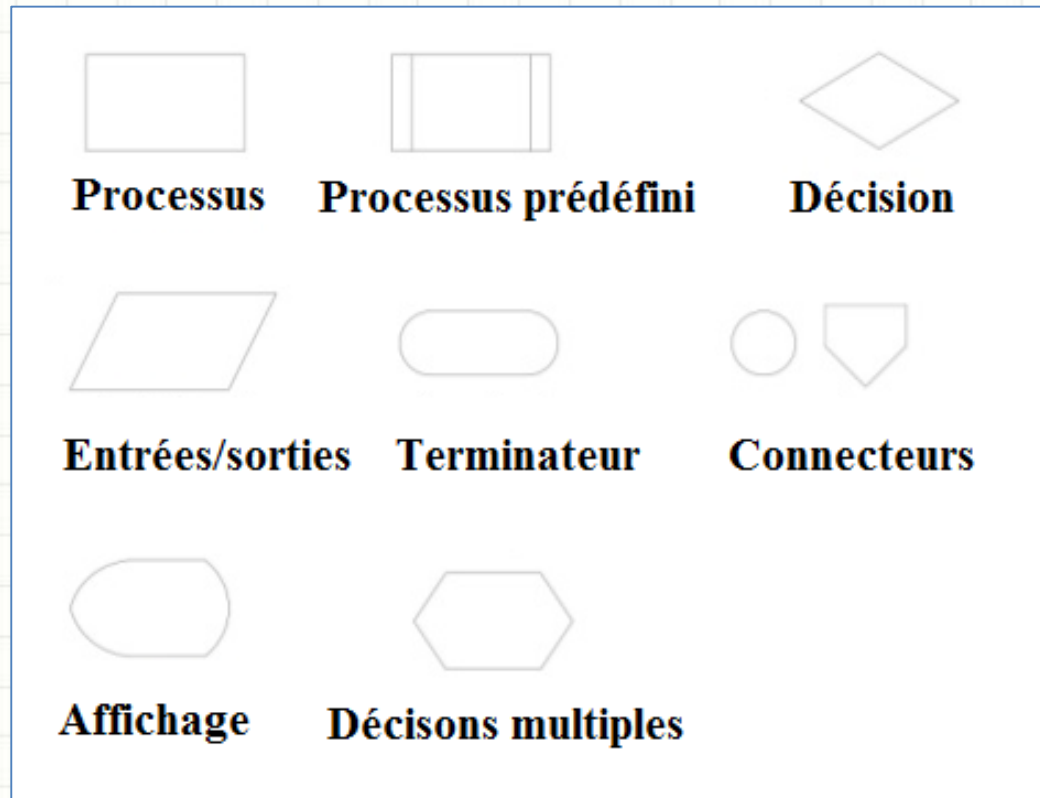


```
C:\Program Files\dotnet\dotnet.exe

Introduire l'opérande de gauche: 15
Introduire l'opérateur: +
Introduire l'opérande de droite: 5
15 + 5 = 20
Souhaitez vous un autre calcul?(o/n): o
Introduire l'opérande de gauche: 5
Introduire l'opérateur: *
Introduire l'opérande de droite: 2
5 * 2 = 10
Souhaitez vous un autre calcul?(o/n): n
Fin du programme
```

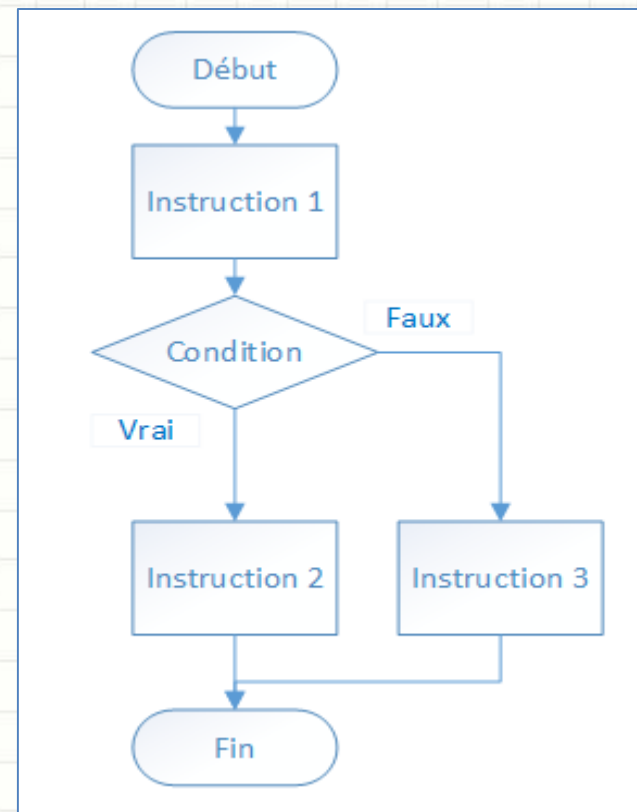
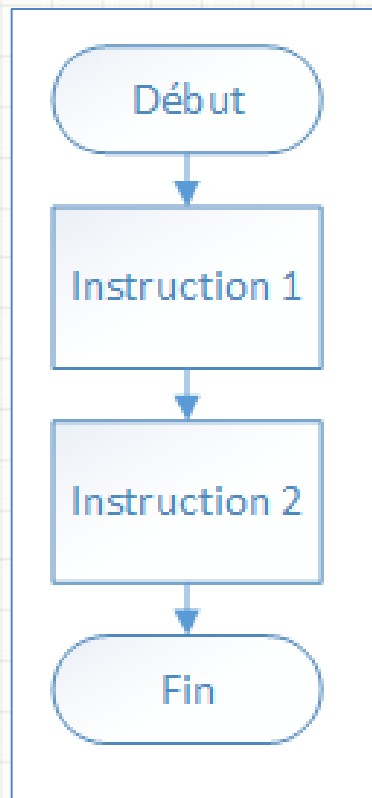
Les organigrammes ISO 5807

Dans le cas de programmes complexes, il est intéressant de penser au séquençement des opérations durant l'exécution du programme et ce avant même de programmer. Dans le cas de la programmation procédurale, une façon simple d'y arriver est d'utiliser l'organigramme suivant la norme ISO 5807.



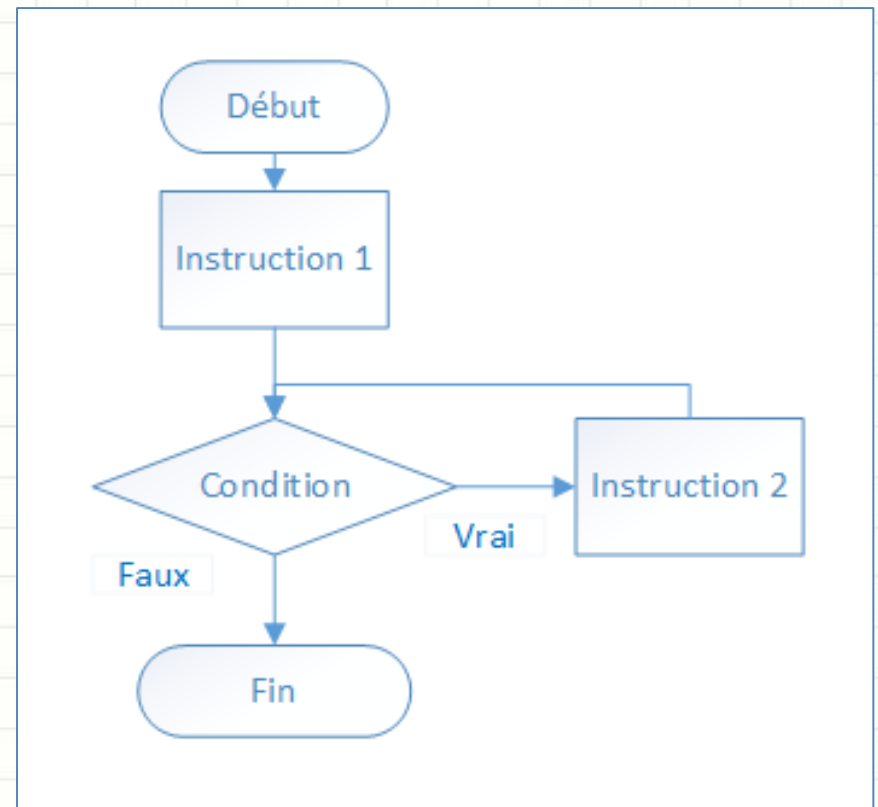
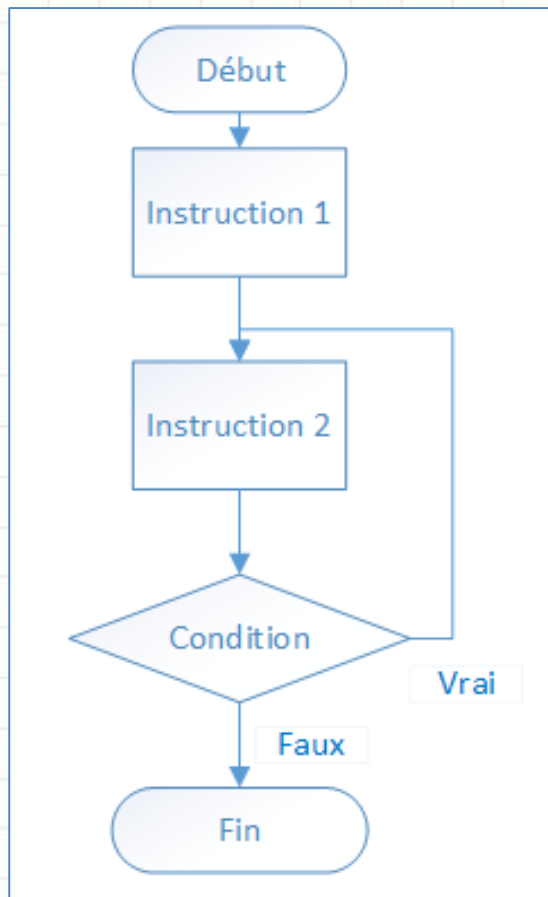
Les organigrammes ISO 5807

- Exemples d'organigramme



Les organigrammes ISO 5807

- Exemples d'organigramme



Les organigrammes ISO 5807

- Exercices

Vous reprenez les premiers exercices des diapositives précédentes et vous en donnez l'organigramme

C# Type valeur et type référence

- L'organisation de la mémoire – La pile
 1. Organisée sous la forme LIFO (Last In First Out)
 2. On y stocke les types de données simples (déclaration statique)
 3. L'environnement d'exécution du programme s'y trouve
 4. Utilisée pour le passage des arguments lors d'un appel de fonction
 5. Taille limitée par défaut à 1Mo

```
static void Main(string[] args)
{
    int a = 10; //variable simple - type valeur
    int b = 20; //variable simple - type valeur
}
```

C# Type valeur et type référence

- L'organisation de la mémoire – Le tas
 1. On y stocke les types de données complexes ou de grosse taille (dynamique)
 2. La taille de cette zone est limitée par la seule capacité mémoire de votre environnement.
 3. En général l'allocation dynamique se distingue par l'usage de l'opérateur new
 4. Le nettoyage de cette zone mémoire est assurée par le Garbage Collector (GC)

```
static void Main(string[] args)
{
    int[] data = new int[20];
    // La référence data est stockée sur la pile
    // Le tableau lui même de 20 entier est sur le tas
}
```

C# Type valeur et type référence

- Nettoyage du tas – Le garbage collector

Tant que la donnée sur le tas est référencée, elle ne sera pas supprimée

Dès que la donnée n'est plus référencée, elle peut être nettoyée par le GC. L'appel à ce GC est automatique mais peut également être forcée.

```
static void Main(string[] args)
{
    int[] data = new int[20];
    data[1] = 10;
    data = null; // Le tableau n'est plus référencé
    GC.Collect(); // appel du Garbage Collector explicite
}
```


C# Type valeur et type référence

- Les références multiples

Une même donnée sur le tas peut être pointée par des références multiples.

```
static void Main(string[] args)
{
    int[] data = new int[20];
    int[] data2 = data;
    data[1] = 10;
    data=null;
    GC.Collect();
    Console.WriteLine(data2[1]);
    Console.ReadKey();
}
```

Dans cet exemple, le tableau d'entier n'est pas supprimé de la mémoire car il est toujours pointé par la référence data2

C# Les tableaux

- Les tableaux à une dimension - déclaration

Un tableau est un regroupement de données de même type de données dont l'accès à un des éléments se fera par un index. Voici quelques exemples de déclaration de tableaux comprenant dans certains cas leur initialisation.

```
static void Main(string[] args)
{
    int[] data = new int[20]; // tableau de 20 entiers
    float[] data2 = new float[30];
    // tableau de 30 flottants
    string[] mois = new string[] { "janvier", "février" };
    // mois référence un tableau de 2 string
    int[] data3 = new int[] { 1, 2, 3, 4, 5, 6, 7 };
    // data3 référence un tableau de 7 entiers
}
```

C# Les tableaux

- Les tableaux à une dimension - accès

Un élément du tableau s'accède grâce à un index dont la valeur la plus petite est 0 (index basé 0)

```
static void Main(string[] args)
{
    int[] data = new int[] { 1, 2, 3, 4, 5, 6, 7 };
    Console.WriteLine(data[0]);
    data[1] = 25;
}
```

C# Les tableaux

- Les tableaux à une dimension - accès

Un élément du tableau s'accède grâce à un index dont la valeur la plus grande ne pourra pas dépasser le nombre maximum d'éléments autorisés

```
static void Main(string[] args)
{
    int[] data = new int[10];
    data[20] = 25;
}
```

Exception non gérée

System.IndexOutOfRangeException : 'L'index se trouve en dehors des limites du tableau.'

[Afficher les détails](#) | [Copier les détails](#)

▸ Paramètres d'exception

Pour rappel, cette exception peut être capturée par un try catch

C# Les tableaux

- Les tableaux à deux dimensions

Ce tableau possèdera un certain nombre de lignes, chacune possédant le même nombre de colonnes. La déclaration devra comprendre ces informations

```
static void Main(string[] args)
{
    int[,] data = new int[5, 6];
    double[,] data2 = new double[, ]
    {
        {0.1, 0.5},
        {1.3, 1.7}
    }; // tableau de 2 lignes et 2 colonnes
    Console.WriteLine(data2[1, 0]);
}
```

C# Les tableaux

- Les tableaux de tableaux (jagged array)

Ce tableau possède pour chaque ligne un nombre de colonnes différent

```
static void Main(string[] args)
{
    int[][] data = new int[2][];
    data[0] = new int[5];
    data[1] = new int[10];
    data[0][1] = 42;
    Console.WriteLine(data[0][1]);
}
```


C# Les tableaux

- Un tableau ... un objet

```
int[] tab = new int[] { 1, 2, 4, 5, 6, 7, 8, 9 };  
for (int i=0;i<tab.Length;i++)  
{  
    Console.WriteLine(tab[i]);  
}
```

Dans certains langages, un tableau doit être lié à une autre variable indiquant sa taille. Dans notre exemple, tab est un objet et il permet l'accès à des membres, méthodes ou propriétés dont la propriété Length indiquant la taille du tableau.

C# Exercice 1

- Calcul du sinus d'un nombre réel en utilisant la théorie des séries de Taylor

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Dans cet exercice, vous devrez développer en premier l'organigramme et le programme écrit en Csharp.

C# Exercice 2

- Calcul pour une valeur X donnée du type float la valeur numérique d'un polynôme de degré n :

$$P(X) = A_n X^n + A_{n-1} X^{n-1} + \dots + A_1 X + A_0$$

Vous utiliserez le schéma de Horner qui évite le calcul des exposants plus gourmands en ressource.

$$([(A_n + 0) * X + A_{n-1}] X + A_{n-2}) * X + \dots + A_0$$

Dans cet exercice, vous devrez développer en premier l'organigramme et le programme écrit en Csharp.

C# L'orienté objet

- Création d'une classe

Une classe est un modèle d'objet regroupant des membres, des méthodes, des propriétés etc...

A partir de ce modèle, nous pouvons créer des objets en utilisant l'opérateur new.

La création d'une classe se réalise en utilisant le mot clef class (ou struct)

```
namespace MaPremiereClasse
{
    class Etudiant
    {
    }
}
```

C# L'orienté objet

- Création d'un objet

Nous utilisons notre classe Etudiant pour créer notre premier objet.

```
static void Main(string[] args)
{
    Etudiant Et1 = new Etudiant();
    Et1._Nom = "Dupond";
    Et1._Prenom = "Eric";
    Et1._Matricule = "LA123123";
    Et1._Naissance = new DateTime(1989,10,5);
    Console.WriteLine(Et1._Nom);
    Console.ReadKey();
}
```

Et1 est un objet. Nous pouvons accéder aux membres associés à Et1 tels que _Nom, _Prenom. Ils sont accessibles car déclarés **public** dans notre classe.

C# L'orienté objet

- Ajout de membres à une classe

Nous pouvons supposer que tout étudiant sera caractérisé au minimum par un nom, un prénom, un matricule. Nous ajouterons une date de naissance qui est un objet d'une classe déjà existante dans le Framework.

```
class Etudiant
{
    public string _Nom;
    public string _Prenom;
    public string _Matricule;
    public DateTime _Naissance;
}
```

Un membre se caractérise par les éléments suivants dans sa déclaration:
Modifieur d'accès – Type – Identifiant

C# L'orienté objet

- Encapsulation

Dans une classe, les méthodes sont des blocs de code permettant de manipuler les membres de la classe. Un point important est de savoir si le contenu des membres est correcte avant toute manipulation. Pour éviter de devoir effectuer cette vérification dans chaque méthode, autant le faire dès le départ en interdisant notamment l'accès directe aux membres (Utilisation du modifieur `private` ou `protected`)

```
class Etudiant
{
    private string _Nom;
    private string _Prenom;
    private string _Matricule;
    private DateTime _Naissance;
}
```

C# L'orienté objet

- Les méthodes

Dans une classe, les méthodes sont des blocs de code permettant de manipuler les membres de la classe et notamment l'une d'entre elle s'appelle le constructeur permettant d'initialiser les membres.

```
class Etudiant
{
    public string _Nom;
    public string _Prenom;
    public string _Matricule;
    public DateTime _Naissance;
    public float CalculerAge()
    {
        TimeSpan span = DateTime.Now.Subtract(_Naissance);
        return span.Days / 365.0F;
    }
}
```

C# L'orienté objet

- Les méthodes

Indépendamment des modifieurs d'accès, une méthode peut accéder aux membres qui appartiennent à la même classe. Une méthode sera déclarée avec des modifieurs permettant de déterminer si elle est accessible par l'objet ou uniquement accessible en interne. Voilà la structure dans la déclaration d'une méthode

Modifieur d'accès – Type de retour – Identifiant – (liste des arguments) { // code }

Une méthode peut retourner un type simple ou la référence d'un objet plus complexe. Le retour est en liaison avec l'utilisation du mot clef **return**

```
public float CalculerAge()  
{  
    TimeSpan span = DateTime.Now.Subtract(_Naissance);  
    return span.Days / 365.0F;  
}
```

C# L'orienté objet

- Les méthodes

Une méthode pourrait être utilisée pour modifier un membre de la classe tout en contrôlant son contenu. Imaginons une méthode permettant de modifier le matricule même si celui ci est en privé.

```
private string _Matricule;  
public void ModifierMatricule( string matricule)  
{  
    this._Matricule = matricule;  
}
```

L'usage du mot clef void signifie que la méthode ne retourne rien et que l'utilisation du mot clef return n'est pas nécessaire

```
Etudiant Et1 = new Etudiant();  
Et1.ModifierMatricule("LA123123");
```

C# L'orienté objet

- Les méthodes – Le constructeur

Le constructeur est une méthode qui est appelée lors de la création d'un objet. Nous pouvons reconnaître le constructeur par ses caractéristiques suivantes:

- Même nom que celui de la classe
- Modificateur d'accès en publique
- Pas de type de retour (même pas void)

```
private string _Nom;  
private string _Prenom;  
private DateTime _Naissance;  
private string _Matricule;  
public Etudiant(string Nom, string Prenom, DateTime  
Naissance, string Matricule)  
{  
    this._Nom = Nom;  
    this._Prenom = Prenom;  
    this._Naissance = Naissance;  
    this._Matricule = Matricule;  
}
```

C# L'orienté objet

- Les méthodes – Les surcharges

Plusieurs méthodes peuvent posséder le même nom. Elles doivent malgré tout se différencier sur un des points suivants:

- Le nombre des arguments
- Le type des arguments si leur nombre est identique

```
public Etudiant(string Nom, string Prenom, DateTime
Naissance, string Matricule)
{
    this._Nom = Nom;
    this._Prenom = Prenom;
    this._Naissance = Naissance;
    this._Matricule = Matricule;
}
public Etudiant() //constructeur par défaut
{
    this._Nom="Inconnu";
}
```


C# L'orienté objet

- Les méthodes – Les appels

Le choix de la méthode dépendra lors de l'appel du nom, du nombre d'arguments, de leur type

```
Etudiant Et1 = new Etudiant("Dupond", "Eric", new  
DateTime(1988, 11, 10), "LA123123");
```

```
Etudiant Et2 = new Etudiant(); //appel du constructeur par  
défaut
```

C# L'orienté objet

- Les méthodes – Les arguments - Vérification

Nous prendrons comme exemple le fait de vouloir vérifier que le matricule passé en argument au constructeur est bon. Il doit commencer par les lettre LA et doit être composé au total de 8 caractères (nous pourrions être plus exigeant)

```
public Etudiant(string Nom, string Prenom, DateTime
Naissance, string Matricule)
{
    this._Nom = Nom;
    this._Prenom = Prenom;
    this._Naissance = Naissance;
    if (Matricule.Length == 8 && Matricule.StartsWith("LA"))
    {
        this._Matricule = Matricule;
    }
}
```

C# L'orienté objet

- Les méthodes – Les arguments - Vérification

Que faire si le matricule est erroné? Nous choisissons de générer une exception

```
if (Matricule.Length == 8 && Matricule.StartsWith("LA"))  
{  
    this._Matricule = Matricule;  
}  
else  
{  
    throw new Exception("Matricule erroné");  
}
```

```
throw new Exception("Matricule erroné");
```

Exception non gérée



System.Exception : 'Matricule erroné'

[Afficher les détails](#) | [Copier les détails](#)

▸ Paramètres d'exception

C# L'orienté objet

- Les exceptions – les captures

Nous remarquons qu'une exception provoque une sortie inconditionnelle du programme ce qui n'est certainement pas souhaitable.

```
try
{
    Etudiant Et1 = new Etudiant("Dupond", "Eric", new
DateTime(1988, 11, 10), "123123");
}
catch(Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

Le bloc de code qui doit être vérifié au niveau des exceptions sera associé au mot clef **try**. Dans le cas d'une exception, le code situé dans le bloc **catch** sera exécuté.

C# L'orienté objet

- Les exceptions – catch multiples

Un bloc try peut être associé à plusieurs catch dont la différence sera liée à l'argument. Dans l'ordre nous mettons les exceptions plus spécifiques et ensuite l'exception générique `Exception` ex

```
throw new FormatException("Matricule erroné");
```

```
try
{
    Etudiant Et1 = new Etudiant("Dupond", "Eric", new
DateTime(1988, 11, 10), "123123");
}
catch (FormatException ex)
{
    Console.WriteLine(ex.Message);
}
catch(Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

C# L'orienté objet

- Les propriétés

Nous venons de voir comment créer des membres et initialiser leur contenu lors de la création d'un objet en utilisant le constructeur. Comment lire le contenu d'un membre ou en modifier la contenu par la suite? Les propriétés...

- Les propriétés en lecture seule

```
public string Matricule
{
    get
    {
        return this._Matricule;
    }
}
```

La propriété "Matricule" possède un accesseur de type "get" permettant uniquement l'accès en lecture au membre "_Matricule"

C# L'orienté objet

- Les propriétés en lecture et écriture

```
public string Nom
{
    get
    {
        return this._Nom;
    }
    set
    {
        this._Nom = value;
    }
}
```

La propriété "Matricule" possède un assesseur de type "set" également permettant l'accès en écriture au membre "_Matricule". Remarquons l'utilisation du mot clef **value**. Nous pouvons contrôler le contenu de **value** avant d'en affecter éventuellement le contenu au membre _Nom

C# L'orienté objet

- Les membres et les valeurs par défaut

Nous avons vu comment initialiser un membre grâce à un constructeur. Comment peut-on décider d'une valeur par défaut à un membre? Imaginons notre classe étudiant dans laquelle nous envisageons d'ajouter un crédit en photocopie. Quand un objet est créé, une valeur de 10€ doit être affectée au membre associé.

- Utilisation du constructeur

```
private float _Credit;  
public Etudiant(string Nom, string Prenom, DateTime  
Naissance, string Matricule) {  
    this._Nom = Nom;  
    this._Prenom = Prenom;  
    this._Naissance = Naissance;  
    this._Matricule = Matricule;  
    this._Credit = 10;  
}
```

C# L'orienté objet

- Initialisation lors de la déclaration du membre

```
private float _Credit=10;
```

- Les arguments et les valeurs par défaut
 - Surcharge des constructeurs

```
public Etudiant(string Nom, string Prenom, DateTime  
Naissance, string Matricule):  
this(Nom,Prenom,Naissance,Matricule,10) { }  
public Etudiant(string Nom, string Prenom, DateTime  
Naissance, string Matricule,float Credit) {  
    this._Credit = Credit;  
  
    //initialisation des autres membres  
}
```

Un constructeur en appelle un autre grâce à l'opérateur this

C# L'orienté objet

– Surcharge des arguments

```
public Etudiant(string Nom, string Prenom, DateTime  
Naissance, string Matricule, float Credit=10)  
{  
    this._Credit = Credit;  
}
```

Sauf si on utilise l'option des arguments nommés lors de la création de l'objet, technique qui sera abordée plus tard dans la matière, il est obligatoire de regrouper les arguments optionnels (avec valeur par défaut) à la fin de la liste des arguments.

C# L'orienté objet

- Les passages d'arguments par valeur / réf

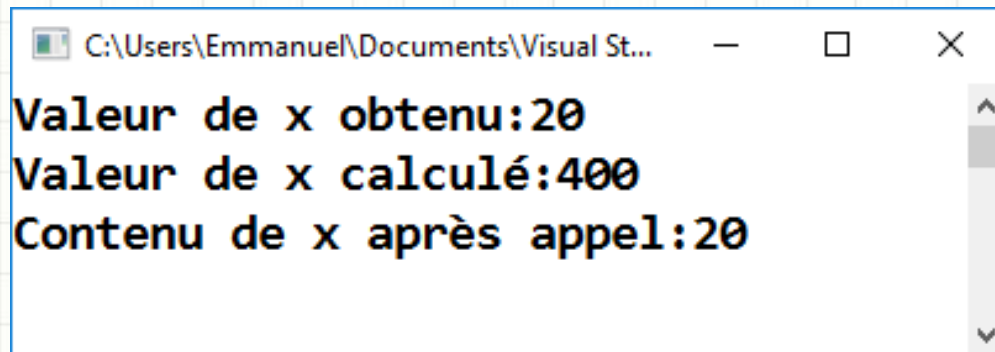
Il est important de comprendre les mécanismes implicites de passages d'arguments en tenant compte des types valeur et référence des arguments passés. Un type valeur est passé par valeur tandis qu'un type référence utilise un passage par référence.

```
class MyMath
{
    public static void CalculCarre(int x)
    {
        Console.WriteLine("Valeur de x obtenu:{0}", x);
        x = x * x;
        Console.WriteLine("Valeur de x calculé:{0}", x);
    }
}
```

C# L'orienté objet

- Les passages d'arguments par valeur / réf

```
static void Main(string[] args)
{
    int x = 20;
    MyMath.CalculCarre(x);
    Console.WriteLine("Contenu de x après appel:{0}", x);
    Console.ReadKey();
}
```



The screenshot shows a console window titled "C:\Users\Emmanuel\Documents\Visual St...". The output text is as follows:

```
Valeur de x obtenu:20
Valeur de x calculé:400
Contenu de x après appel:20
```


C# L'orienté objet

- Les passages d'arguments par valeur / réf

```
class Complexe
{
    public float _Preelle;
    public float _Pimage;
    public Complexe (float Prelle, float Pimage)
    {
        _Preelle = Prelle;
        _Pimage = Pimage;
    }
    public void AfficherComplexe()
    {
        Console.WriteLine("{0} + i {1}", _Preelle, _Pimage);
    }
}
```

C# L'orienté objet

- Les passages d'arguments par valeur / réf

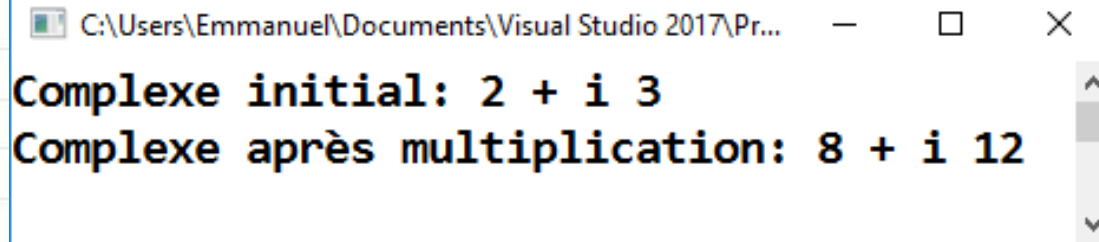
```
class MyMath
{
    public static void MultComplexe(Complexe x, float k)
    {
        x._Preelle *= k;
        x._Pimage *= k;
    }
}
```

```
Complexe cp1 = new Complexe(2.0F, 3.0F);
Console.Write("Complexe initial: ");
cp1.AfficherComplexe();
MyMath.MultComplexe(cp1, 4.0F);
Console.Write("Complexe après multiplication: ");
cp1.AfficherComplexe();
Console.ReadKey();
```

C# L'orienté objet

- Les passages d'arguments par valeur / réf

La méthode `MultiComplexe` comprend bien comme premier argument une référence de la classe complexe `Complexe` x



```
C:\Users\Emmanuel\Documents\Visual Studio 2017\Pr...  
Complexe initial: 2 + i 3  
Complexe après multiplication: 8 + i 12
```

The screenshot shows a console window with a title bar indicating the file path 'C:\Users\Emmanuel\Documents\Visual Studio 2017\Pr...'. The window contains two lines of text: 'Complexe initial: 2 + i 3' and 'Complexe après multiplication: 8 + i 12'. The text is displayed in a monospaced font, and there is a vertical scrollbar on the right side of the window.

C# L'orienté objet

- Les passages de type valeur par référence

Il est possible de modifier notre premier exemple de sorte que le type valeur soit passé par référence. Nous utiliserons pour ce faire le mot clef `ref`

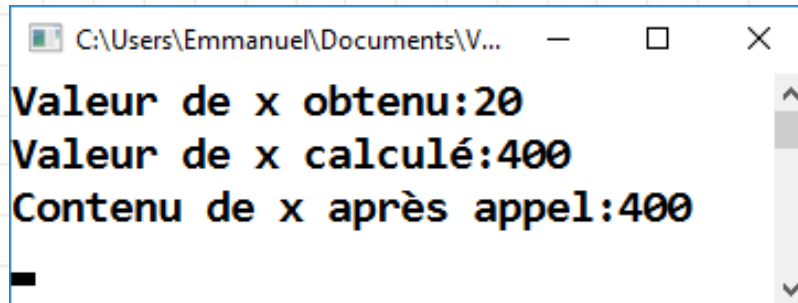
```
class MyMath
{
    public static void CalculCarre(ref int x)
    {
        Console.WriteLine("Valeur de x obtenu:{0}", x);
        x = x * x;
        Console.WriteLine("Valeur de x calculé:{0}", x);
    }
}
```

C# L'orienté objet

- Les passages de type valeur par référence

Il est possible de modifier notre premier exemple de sorte que le type valeur soit passé par référence. Nous utiliserons pour ce faire le mot clef `ref`

```
int x = 20;  
MyMath.CalculCarre( ref x);  
Console.WriteLine("Contenu de x après appel:{0}", x);  
Console.ReadKey();
```



```
C:\Users\Emmanuel\Documents\V...  
Valeur de x obtenu:20  
Valeur de x calculé:400  
Contenu de x après appel:400  
_
```

Le mot clef `ref` peut être remplacé par le mot clef `out`.

C# L'orienté objet

- Les propriétés automatiques

Dans certains cas, l'encapsulation réside simplement dans le fait de rassembler des données entre elles. Il n'est pas toujours nécessaire de sortir le grand jeu des membres, propriétés, constructeurs.... Nous pouvons utiliser les propriétés automatiques

```
class Client
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int ClientId { get; set; }
}
```

```
static void Main(string[] args)
{
    Client clt1 = new Client { Nom = "Dupond", Prenom =
    Durand", ClientId = 10 };
    Console.ReadKey();
}
```


C# L'orienté objet

- Les propriétés automatiques - protections

L'assesseur set peut très bien être protégé contre tout accès extérieur.

```
class Client
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int ClientId { get; private set; }

    public void GenerateNewClientId()
    {
        Random rnd = new Random();
        ClientId = rnd.Next();
    }
}
```

C# L'orienté objet

- Les propriétés automatiques - protections

L'assesseur set peut très bien être protégé contre tout accès extérieur.

```
Client clt1 = new Client { Nom = "Dupond", Prenom = "Durand" };  
clt1.ClientId = 200;
```

```
clt1.G int Client.ClientId { get; private set; }
```

Console

Impossible d'utiliser la propriété ou l'indexeur 'Client.ClientId' dans ce contexte, car l'accessesseur set n'est pas accessible

C# L'orienté objet

- Les propriétés automatiques – valeurs par défaut

```
class Client
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int ClientId { get; private set; } = 100;

    public void GenerateNewClientId()
    {
        Random rnd = new Random();
        ClientId = rnd.Next();
    }
}
```

C# L'orienté objet

- Les surcharges des opérateurs

Pour les types simples et certains types complexes intégrés au Framework, nous disposons d'opérateurs. Prenons le cas de l'opérateur d'addition comme opérateur arithmétique mais aussi l'opérateur relationnel tel que le test de stricte égalité.

Que penser des types complexes personnalisés? Imaginons une classe personnelle permettant de gérer les dates....

```
class MaDate
{
    public int jour { get; set; }
    public int mois { get; set; }
    public int annee { get; set; }
}
```

C# L'orienté objet

- Les surcharges des opérateurs

Il serait intéressant de pouvoir utiliser le code suivant pour comparer deux dates entre elles...

```
MaDate d1 = new MaDate { jour=10, mois=10, annee=2017 };  
MaDate d2 = new MaDate { jour=10, mois=10, annee=2017 };  
if (d1 == d2)  
{  
    Console.WriteLine("Dates identiques");  
}
```

Dans cet exemple, l'opérateur compare les deux références et pas les propriétés de la classe que sont les jours/mois/années

Pour y parvenir, nous devons mettre en place une surcharge de l'opérateur ==

C# L'orienté objet

- Les surcharges des opérateurs

Dans la classe, nous ajoutons une nouvelle méthode

```
public static bool operator == (MaDate lop, MaDate rop)
{
    return (lop.jour == rop.jour && lop.mois == rop.mois &&
lop.annee == rop.annee);
}
```

Dans cette surcharge, remarquons les mots clefs importants que sont: **static** et **operator**.

Si nous analysons l'utilisation de cet opérateur: `if (d1 == d2)` nous remarquons que cet opérateur est binaire: il réclame deux opérandes, l'une à gauche et l'autre à droite de l'opérateur. D'autre part, cet opérateur fournit comme résultat un booléen qui sera utilisé par l'instruction `if`

C# L'orienté objet

- Les surcharges des opérateurs

```
public static bool operator == (MaDate lop, MaDate rop)
```

Les deux opérandes de type MaDate sont passés en arguments à la méthode tandis que celle ci retourne un booléen.

C# L'orienté objet

- Les méthodes de type statique

Nous retrouvons le mot clef static dans la surcharge des opérateurs. Une méthode d'instance ne possède pas ce mot clef.

L'appel d'une méthode d'instance s'effectue de la façon suivante:

objet.méthode(arguments);

```
Date d1 = new Date(10, 5, 2010);  
Date d2 = new Date(11, 5, 2010);  
  
string strdate = d1.ToString();
```

Si nous observons la surcharge de l'opérateur, il n'y a aucun objet servant à appeler la méthode. Nous avons déjà utilisé de nombreuses méthodes statiques dans les codes précédents

C# L'orienté objet

- Les méthodes de type statique

Prenons la cas de la classe Console. Si nous analysons les métadonnées de cette classe, nous obtenons:

```
public static void WriteLine(string value);  
public static void WriteLine(string format, object arg0);
```

L'utilisation de telles méthodes s'effectue de la façon suivante:

Nom-de-la-classe.Nom-de-le-méthode(arguments);

```
if (d1 == d2)  
{  
    Console.WriteLine("Dates identiques");  
}
```

C# L'orienté objet

- Les méthodes de type statique

Les méthodes permettant de parser des chaînes de caractères dans un type donné sont des méthodes statiques dites de classe.

```
public static Int32 Parse(string s);
```

Nous allons reprendre notre classe permettant la gestion des dates et imaginer une méthode permettant de parser une chaîne de caractères représentant une date et en extraire les valeurs numériques que sont jours mois et années

Pour y parvenir, nous utiliserons les expressions régulières

C# L'orienté objet

- Les méthodes de type statique

```
public static Date Parse(string date)
{
    int jour, mois, annee;
    string MyPatern = @"^(\d{1,2})[-/](\d{1,2})[-/](\d{4}$)";
    Regex MyRegex = new Regex(MyPatern);
    if (MyRegex.IsMatch(date))
    {
        jour = int.Parse(MyRegex.Match(date).Groups[1].Value);
        mois = int.Parse(MyRegex.Match(date).Groups[2].Value);
        annee = int.Parse(MyRegex.Match(date).Groups[3].Value);
        return new Date(jour, mois, annee);
    }
    else
    {
        throw new Exception("Bad Format");
    }
}
```

C# L'orienté objet

- Les membres de type statique

Les membres statiques sont à l'image des méthodes statiques, ils ne sont pas liés à une instance mais à la classe elle même.

```
class Compte_a_vue
{
    private static float TauxInteret;
    private string NumeroCompte;
    private string Titulaire;
}
```

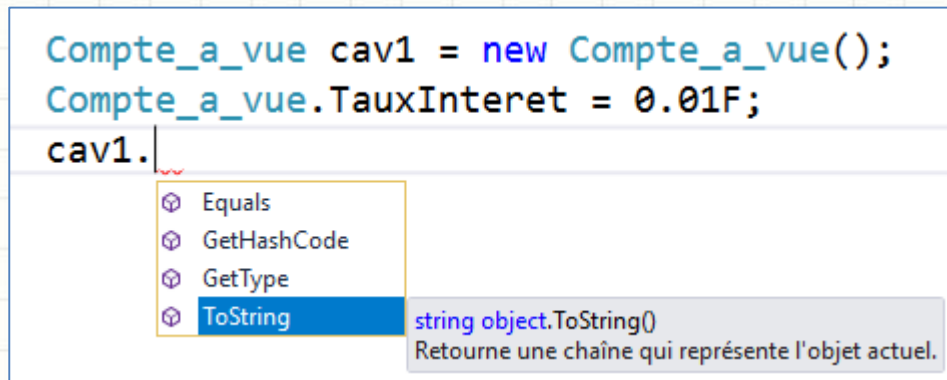
Dans notre exemple, le taux d'intérêt est identique pour tous les comptes à vue. Il est donc intéressant de le définir comme statique

C# L'orienté objet

- Les membres de type statique

Si le membre est déclaré en publique, nous pouvons y accéder de la façon suivante

```
Compte_a_vue cav1 = new Compte_a_vue();
Compte_a_vue.TauxInteret = 0.01F;
cav1.
```



Le principe d'encapsulation et de protection n'est pas toujours compatible avec l'accessibilité publique d'un membre. Nous allons voir comment initialiser un tel membre déclaré en privé

C# L'orienté objet

- Les membres de type statique

Utilisation d'un constructeur statique

```
class Compte_a_vue
{
    private static float TauxInteret;
    private string NumeroCompte;
    private string Titulaire;
    static Compte_a_vue()
    {
        TauxInteret = 0.010F;
    }
}
```

Le constructeur statique est appelé une seule fois avant la première instanciation de la classe. Il ne peut pas contenir le moindre argument

C# L'orienté objet

- Les membres de type statique

Initialisation lors de la déclaration

```
class Compte_a_vue
{
    private static float TauxInteret = 0.010F;
    private string NumeroCompte;
    private string Titulaire;
}
```

C# L'orienté objet

- Les paramètres nommés

Nous avons vu comment définir des arguments avec des valeurs par défaut

```
static void Traitement(int a, int b=10, int c = 20)
{
    //traitement
}
```

Les arguments par défaut sont tous regroupés à la fin de la liste des arguments.
Lors d'un appel, une omission oblige les omission suivante.

```
Traitement(10);  
Traitement(10, 20);
```

Argument manquant

C# L'orienté objet

- Les paramètres nommés

L'utilisation des arguments nommés permet de contourner cette limitation

```
Traitement(10, c: 50);
```

C# L'orienté objet

- Les types anonymes

Les types anonymes sont rendus possibles grâce à l'existence du mot clef `var` et d'une technique que l'on appelle inférence de type.

C'est l'initialisation de la variable lors de sa déclaration qui définira le type

```
var count = 0;  
Console.WriteLine(count.GetType().ToString());  
  
var nom = "Dupond";  
Console.WriteLine(nom.GetType().ToString());
```

Dans le premier cas, nous aurons un `System.Int32` et dans le second cas un `System.String`

C# L'orienté objet

- L'héritage

Pour bien comprendre la notion d'héritage, nous allons prendre un exemple concret. Imaginons un garage vendant des véhicules pouvant être soit des camions soit des voitures.

Essayons de retrouver les caractéristiques communes entre une voiture et un camion. Nous créerons alors une classe véhicule contenant des membres associés à ces caractéristiques.

Nous reprenons comme caractéristiques: la marque, le modèle, l'année, la cylindrée... Nous pourrions en envisager d'autres bien sur...

C# L'orienté objet

- L'héritage

```
class Vehicule
{
    private string Marque;
    private string Modele;
    private int Annee;
    private int Cyllindree;

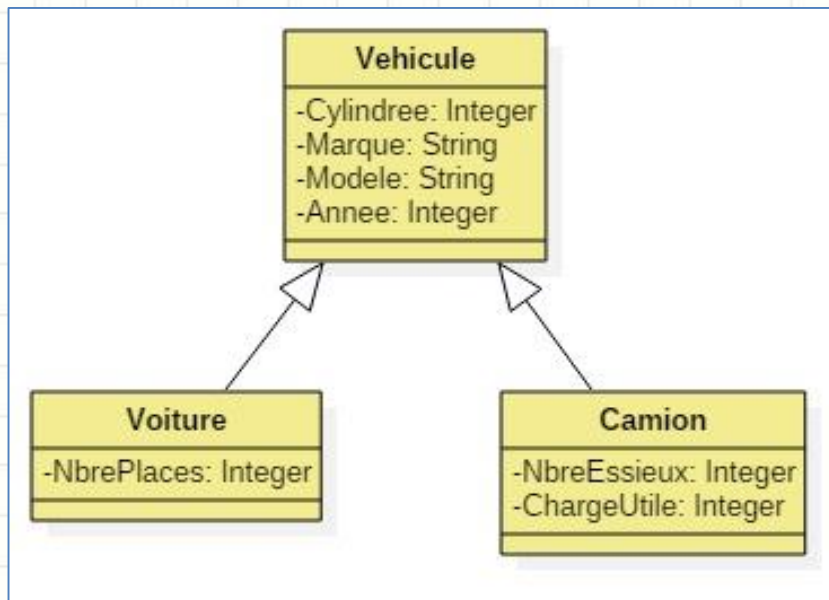
    public Vehicule(string Marque, string Modele, int Annee, int
Cyllindree)
    {
        this.Marque = Marque;
        this.Modele = Modele;
        this.Annee = Annee;
        this.Cyllindree = Cyllindree;
    }
}
```

C# L'orienté objet

- L'héritage

Une voiture est un véhicule. Elle possède les mêmes propriétés qu'un véhicule mais on peut lui en ajouter d'autres qui feront que c'est un véhicule mais pas un camion.

Nous pouvons créer une classe Voiture qui héritera des propriétés de la classe Vehicule. Nous pourrons aussi créer une classe Camion qui héritera des propriétés de la classe Vehicule.



Ce diagramme de classe est réalisé en utilisant le standard UML. Nous nous limitons ici aux attributs (membres)

C# L'orienté objet

- L'héritage

Vehicule est appelé la classe de base tandis que Camion/Voiture sont appelés les classes dérivées.

```
class Voiture:Vehicule
{
    private int NbrPlaces;
    public Voiture(string Marque, string Modele, int Annee, int
Cylindree,int NbrPlaces):base(Marque,Modele,Annee,Cylindree)
    {
        this.NbrPlaces = NbrPlaces;
    }
}
```

Dans cette syntaxe, nous disons que la classe Voiture hérite de la classe Vehicule:

```
class Voiture:Vehicule
```

C# L'orienté objet

- L'héritage – Chaînage des constructeurs

Pour les constructeurs par défaut, le chaînage est automatique. Lorsque nous avons des constructeurs définis de façon explicite, nous devons les chaîner.

La création d'une voiture sous entend la création d'un véhicule

```
public Voiture(string Marque, string Modele, int Annee, int  
Cylindree, int NbrPlaces):base(Marque, Modele, Annee, Cylindree)  
{  
    this.NbrPlaces = NbrPlaces;  
}
```

Le constructeur de la classe Voiture reçoit tous les arguments et notamment ceux qui seront utilisés pour appeler le constructeur de la classe Vehicule. Attention à l'usage du mot clef `base` pour appeler le constructeur de la classe de base

C# L'orienté objet

- L'héritage – Chaînage des constructeurs

Pour la classe Camion, nous aurons le même principe

```
class Camion:Vehicule
{
    private int NbrEssieux;
    private int ChargeUtile;

    public Camion(string Marque, string Modele, int Annee, int
Cylindree,int NbrEssieux,int ChargeUtile) : base(Marque,
Modele, Annee, Cylindree)
    {
        this.ChargeUtile = ChargeUtile;
        this.NbrEssieux = NbrEssieux;
    }
}
```


C# L'orienté objet

- L'héritage – Les instances

La création d'un objet sera similaire que dans le cas des classes simples...

```
static void Main(string[] args)
{
    var voiture1 = new Voiture("Peugeot", "205", 1985, 1600, 4);
    var camion1 = new Camion("Mercedes-Benz", "9999", 1980,
3500, 2, 10000);
}
```

C# L'orienté objet

- L'héritage – Le polymorphisme

Reprenons le garage qui doit vendre à la fois des voitures et des camions. Comment ce garage va-t-il gérer son stock de véhicules. Doit-il gérer un stock de voiture d'un côté et un stock de camion de l'autre côté? En voici un exemple...

```
static void Main(string[] args)
{
    Voiture[] StockVoitures = new Voiture[100];
    Camion[] StockCamions = new Camion[100];
    StockVoitures[0] = new Voiture("Peugeot", "205",
1985,1600, 4);
    StockCamions[0] = new Camion("Mercedes-Benz", "9999",
1980, 3500, 2, 10000);
}
```

C# L'orienté objet

- L'héritage – Le polymorphisme

Il est plus simple d'imaginer un stock de véhicules, un véhicule pouvant être une voiture ou un camion. Cette possibilité nous est offerte par le polymorphisme

```
static void Main(string[] args)
{
    Vehicule[] StockVehicules = new Vehicule[100];
    StockVehicules[0] = new Voiture("Peugeot", "205",
1985,1600, 4);
    StockVehicules[1] = new Camion("Mercedes-Benz", "9999",
1980, 3500, 2, 10000);
}
```

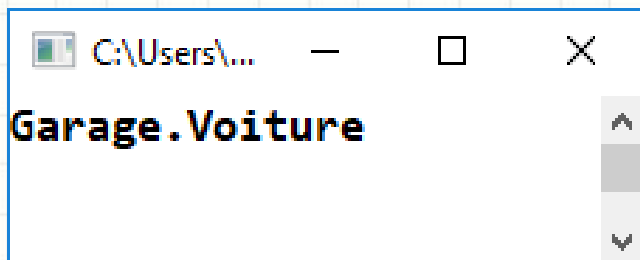
Le polymorphisme: une référence d'une classe de base peut pointer vers tout objet d'une de ses classes dérivées.

C# L'orienté objet

- L'héritage – Le polymorphisme

Oui mais.... peut-on savoir si un véhicule est un camion ou une voiture si la référence est un véhicule?

```
Vehicle[] StockVehicules = new Vehicle[100];  
StockVehicules[0] = new Voiture("Peugeot", "205", 1985, 1600,  
4);  
StockVehicules[1] = new Camion("Mercedes-Benz", "9999", 1980,  
3500, 2, 10000);  
Console.WriteLine(StockVehicules[0].GetType());  
Console.ReadKey();
```



C# L'orienté objet

- L'héritage – Le polymorphisme

```
Vehicle[] StockVehicules = new Vehicle[100];  
//création des objets  
for(int i=0; i<2; i++)  
{  
    switch (StockVehicules[i].GetType().Name)  
    {  
        case "Voiture":  
            Console.WriteLine("Stock[{0}]: une voiture",i);  
            break;  
        case "Camion":  
            Console.WriteLine("Stock[{0}]: un camion",i);  
            break;  
        default:  
            break;  
    }  
}
```

C# L'orienté objet

- L'héritage – Le polymorphisme – la classe object

Nous avons évoqué que l'on de la création d'un objet d'une classe simple, cet objet possède des méthodes par défaut telle que par exemple ToString()

```
Vehicule[] Stock = new Vehicule[100];  
Stock[0] = new Voiture("Peugeot", "205", 1985, 1600, 4);  
Stock[1] = new Camion("Mercedes-Benz", "9999", 1980, 3500, 2, 10000);  
Console.WriteLine(Stock[0].T)  
Console.ReadKey();
```

- Equals
- GetHashCode
- GetType
- ToString**

string object.ToString()
Retourne une chaîne qui représente l'objet actuel.

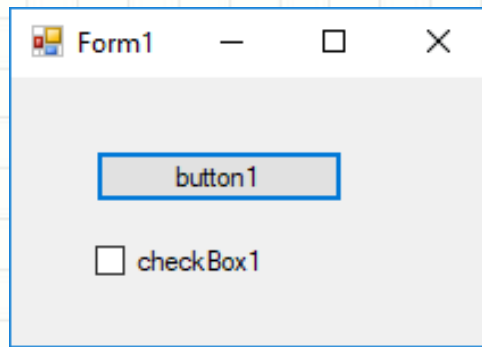
La méthode ToString() appartient à la classe object dont nous héritons par défaut. Nous pouvons appliquer le polymorphisme à une référence sur cette classe de base de la façon suivante...

C# L'orienté objet

- L'héritage – Le polymorphisme – la classe object

```
object Reference= new Voiture("Peugeot", "205", 1985, 1600, 4);  
Console.WriteLine(Reference.GetType().Name);  
Console.ReadKey();
```

Ce concept est très utilisé dans la programmation événementielle. Imaginons une même méthode capable de gérer le clic sur un bouton ou une case à cocher dans une application WindowsForm.

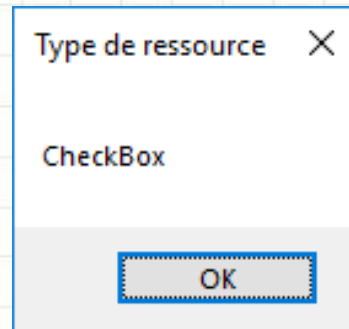
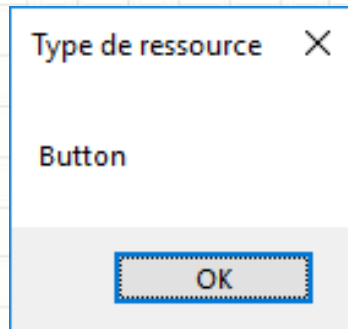


```
this.checkBox1.Click += new  
System.EventHandler(this.Gestion_Click);  
  
this.button1.Click += new  
System.EventHandler(this.Gestion_Click);
```

C# L'orienté objet

- L'héritage – Le polymorphisme – la classe object

```
private void Gestion_Click(object sender, EventArgs e)
{
    MessageBox.Show(sender.GetType().Name);
}
```

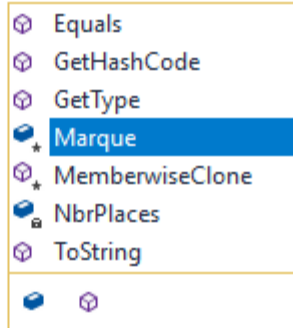


C# L'orienté objet

- L'héritage – Private ou Protected

```
protected string Marque;  
private string Modele;  
private int Annee;  
private int Cylindree;
```

```
public override string ToString()  
{  
    return this.  
}
```



La variable membre "Marque" est accessible dans la classe dérivée. Les membres privés ne le sont pas.

C# L'orienté objet

- L'héritage – Les classes abstraite (abstract)

Si nous prenons notre classe de base, nous pouvons nous poser la question de savoir si une instance de cette classe serait intéressante? Probablement pas. Un véhicule en soit n'est pas intéressant s'il n'est pas soit une voiture, soit un camion.

Une façon de rendre cette classe non instanciable est de créer une classe abstraite.

```
abstract class Vehicule
{
    private string Marque;
    private string Modele;
    private int Annee;
    private int Cylindree;
    //...
}
```

C# L'orienté objet

- L'héritage – Les méthodes abstraites (abstract)

Une classe abstraite peut contenir des méthodes abstraites. Ce sont des méthodes déclarées avec le mot clef `abstract` et qui n'ont pas de corps.

Imaginons la méthode permettant d'obtenir la côte à l'argus d'un camion ou d'une voiture (dépendant de l'âge et du kilométrage si le garage vend des véhicules d'occasion)

```
abstract class Vehicule
{
    private string Marque;
    private string Modele;
    private int Annee;
    private int Cylindree;
    public abstract int CoteArgus();
    //...
```

C# L'orienté objet

- L'héritage – Les méthodes abstraites (abstract)

Avec les méthodes abstraites, toute classe dérivée doit obligatoirement implémenter une méthode avec le même prototypage sous peine de générer des erreurs

❌ CS0534 'Camion' n'implémente pas le membre abstrait hérité 'Vehicule.CoteArgus()'
❌ CS0534 'Voiture' n'implémente pas le membre abstrait hérité 'Vehicule.CoteArgus()'

```
class Camion:Vehicule
{
    //...
    public override int CoteArgus()
    {
        return 100;
    }
}
```

L'intérêt de cette technique est de savoir que toutes les classes dérivées posséderont une méthode ayant ce nom là.

C# L'orienté objet

- L'héritage – Les interfaces

Il est intéressant dans un environnement de développement que chacun adopte des règles communes de programmation. Si nous décidons d'implémenter une méthode permettant de convertir une voiture en chaîne de caractères, nous utiliserons ToString qui est communément utilisé et pas autre chose.

Nous retrouvons pour cela, des classes abstraites comprenant uniquement des méthodes abstraites. Nous pouvons alors remplacer ces classes abstraites par des interfaces. Dans le Framework, toutes les interfaces commencent par la lettre I du genre IDisposable, IComparable...

Prenons comme exemple l'interface IComparable qui comprend la méthode CompareTo

C# L'orienté objet

- L'héritage – Les interfaces

Prenons le cas de deux nombres entiers.

```
int a = 20;  
int b = 20;  
Console.WriteLine(a.CompareTo(b));
```

La méthode CompareTo renvoie une valeur nulle si les deux entiers sont égaux, une valeur négative si $a < b$ et positive si $a > b$

Cette méthode sera très utile dans le cas des méthodes de tris.

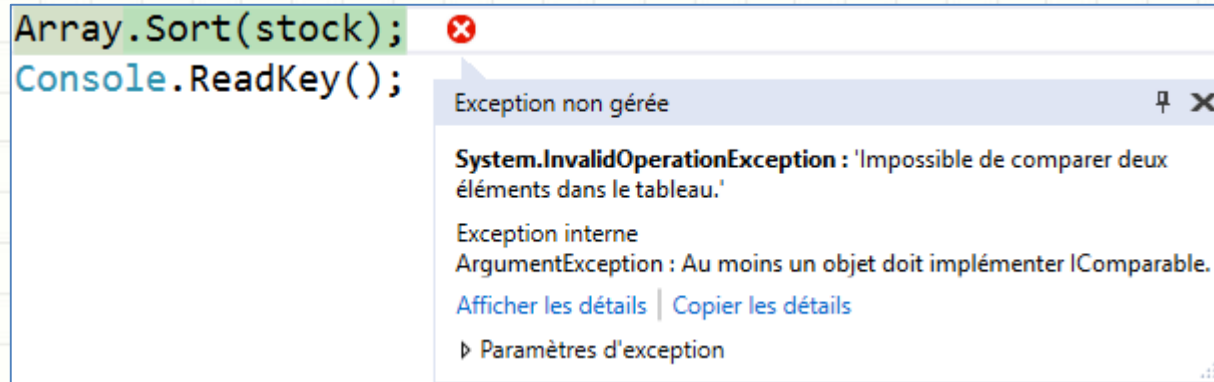
```
int[] tab = { 1, 10, 20, 5, 7, 9, 2 };  
Array.Sort(tab);
```

C# L'orienté objet

- L'héritage – Les interfaces

Essayons maintenant de trier un tableau de voitures.

```
Voiture[] stock = new Voiture[3];  
stock[0] = new Voiture("Peugeot", "205", 1985, 1800, 4);  
stock[1] = new Voiture("Peugeot", "205", 1985, 2000, 4);  
stock[2] = new Voiture("Peugeot", "205", 1985, 1600, 4);  
Array.Sort(stock);
```



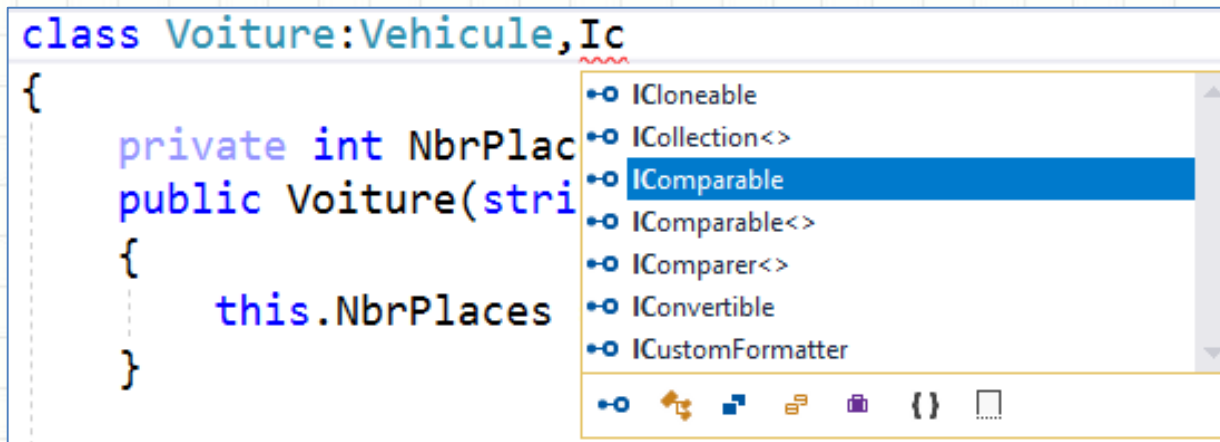
Le tri n'est pas possible car la classe voiture ne possède pas la méthode CompareTo

C# L'orienté objet

- L'héritage – Les interfaces

Nous ferons en sorte que notre classe voiture hérite de l'interface IComparable.

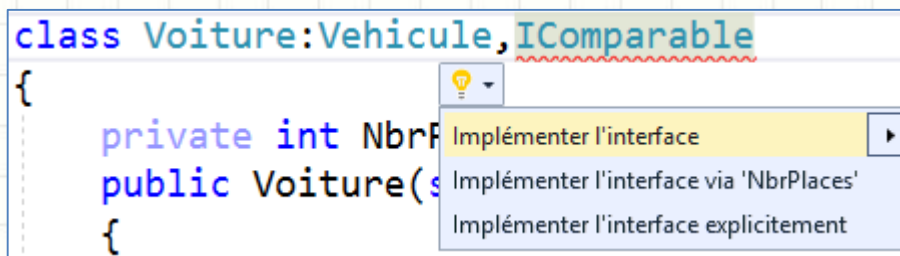
```
class Voiture:Vehicule,Ic
{
    private int NbrPlac
    public Voiture(stri
    {
        this.NbrPlaces
    }
}
```



The screenshot shows a code editor with a class definition. The class is named 'Voiture' and inherits from 'Vehicule' and 'Ic'. The code is partially written, showing a private integer 'NbrPlac' and a public constructor 'Voiture(stri'. A dropdown menu is open, showing a list of interfaces: 'ICloneable', 'ICollection<>', 'IComparable' (highlighted), 'IComparable<>', 'IComparer<>', 'IConvertible', and 'ICustomFormatter'. The 'Ic' in the class definition is underlined with a red squiggly line, indicating it is not a recognized type.

```
class Voiture:Vehicule,IComparable
{
    private int NbrP
    public Voiture(s
    {

```



The screenshot shows the same code editor with the class definition updated to inherit from 'IComparable' instead of 'Ic'. The 'IComparable' is underlined with a red squiggly line. A tooltip is visible, showing three options for implementing the interface: 'Implémenter l'interface', 'Implémenter l'interface via 'NbrPlaces', and 'Implémenter l'interface explicitement'. The first option is highlighted in yellow.

C# L'orienté objet

- L'héritage – Les interfaces

L'environnement Visual Studio nous permet d'implémenter explicitement l'interface sans avoir à tout encoder manuellement

```
int IComparable.CompareTo(object obj)
{
    throw new NotImplementedException();
}
```

Nous comparons deux voitures sur base de leur cylindrée.

```
int IComparable.CompareTo(object obj)
{
    return this.Cylindree.CompareTo(((Voiture)obj).Cylindree);
}
```

Remarquons l'utilisation du polymorphisme permettant de faire passer un objet de type voiture par une référence de type `object`

C# L'orienté objet

- L'héritage – Remarque importante

Dans certains langages de programmation orientés objet, nous pouvons hériter de plusieurs classes de base (Exemple du langage C)

En CSharp, une classe ne peut dérivée que d'une seule classe de base mais elle peut hériter de plusieurs interfaces.

C# L'orienté objet

- Les classes génériques

Reprenons une partie de code d'une des diapositives précédentes

```
int IComparable.CompareTo(object obj)
{
    return this.Cylindree.CompareTo(((Voiture)obj).Cylindree);
}
```

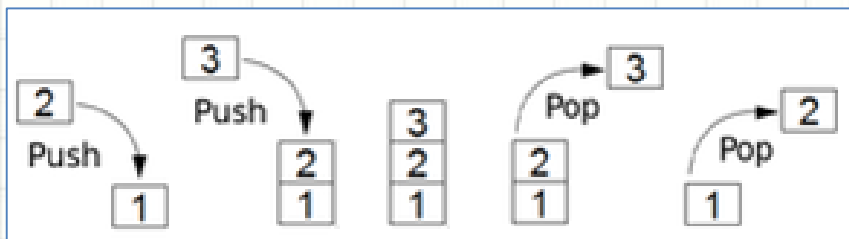
Nous remarquons que le polymorphisme nous oblige à du transtypage. Ce mécanisme nous amène à une utilisation de ressources machine plus importante. Ne serait-il pas possible de faire passer en argument, dans notre exemple, une référence de type voiture? La réponse est oui... en utilisant les classes ou les interfaces génériques.

Prenons comme exemple notre souhait de créer une classe permettant de gérer des piles d'objet de tout genre (de type FIFO).

C# L'orienté objet

- Les classes génériques

Une pile de type FIFO a la fonctionnalité suivante. Nous pouvons penser à une pile d'assiettes. Par facilité, nous dépôt ou tout enlèvement se fera au sommet du tas



Une pile comprendra un tableau. Nous ajouterons une méthode Push permettant d'ajouter un élément et la méthode Pop permettant de récupérer un élément

Il faudrait que l'on puisse définir le type des éléments présents dans ce tableau. C'est le but recherché par les classes génériques

C# L'orienté objet

- Les classes génériques

```
class PileFifo<T>
{
    private T[] tab;
    private int MaxSize;
    private int indice;
    public PileFifo(int MaxSize=100)
    {
        this.MaxSize = MaxSize;
        this.indice = 0;
        this.tab = new T[MaxSize];
    }
}
```

La syntaxe suivante <T> après le nom de la classe permet de représenter la généricité du type. Lors de la création d'un objet, nous aurons la syntaxe suivante:

```
PileFifo<int> MaPileInt = new PileFifo<int>(10);
```

C# L'orienté objet

- Les classes génériques

Le fait que `<T>` récupère le type lors de l'instanciation, nous pouvons utiliser ce type pour la création de variables, de tableaux, d'objets, de type de retour... Exemple:

```
public void Push(T element)
{
    if (this.indice < this.MaxSize)
    {
        tab[indice] = element;
        indice++;
    }
    else
    {
        throw new Exception("Stack full");
    }
}
```

C# L'orienté objet

- Les classes génériques

Si nous reprenons notre exemple d'instanciation et que nous allons voir en mode d'exécution pas par pas

```
public PileFifo(int MaxSize=100)
{
    this.MaxSize = MaxSize;
    this.indice = 0;
    ► this.tab = new T[MaxSize];
} ≤ 1 ms écoulées
```

► this.tab {int[10]}

`this.tab` est bien considéré dans notre exemple comme un tableau d'entier.

C# L'orienté objet

- Les classes génériques

Prenons un autre exemple et considérons une pile de véhicules

```
class Vehicule
{
    //contenu de la classe Vehicule
}
```

```
PileFifo<Vehicule> MonGarage = new PileFifo<Vehicule>(10);
```

```
public PileFifo(int MaxSize=100)
{
    this.MaxSize = MaxSize;
    this.indice = 0;
    ► this.tab = new T[MaxSize];
}
≤ 2 ms écoulées
```

► this.tab {GestionPile.Vehicule[10]}

C# L'orienté objet

- Les collections

Reprenons l'exemple de la gestion des véhicules évoquée dans le gestion des héritages. Un garage est en soi une collection de véhicules. Nous allons aborder cette gestion à travers un exemple simple de tableaux et ensuite à travers de listes génériques.

```
class Garage
{
    private Vehicule[] Stock;
    private int MaxVehicules;
    private int Indice;
    public Garage(int MaxVehicules)
    {
        this.Indice = 0;
        this.MaxVehicules = MaxVehicules;
        Stock = new Vehicule[MaxVehicules];
    }
}
```

C# L'orienté objet

- Les collections – Ajouter un élément

La classe Garage comprend un tableau de références de véhicules. Il faut naturellement prévoir la possibilité d'ajouter un véhicule dans le garage mais également d'en supprimer un. Nous ajouterons une méthode Add ainsi qu'une méthode Remove.

```
public void Add(Vehicule Vhcl)
{
    if (this.Indice < this.MaxVehicules)
    {
        this.Stock[this.Indice] = Vhcl;
        this.Indice++;
    }
    else
    {
        throw new Exception("Garage plein");
    }
}
```

C# L'orienté objet

- Les collections – Ajouter un élément

```
Garage MonGarage = new Garage(10);  
MonGarage.Add( new Voiture("Peugeot", "205", 1985, 1800, 4));  
MonGarage.Add(new Voiture("Peugeot", "205", 1985, 2000, 4));  
MonGarage.Add(new Voiture("Peugeot", "205", 1985, 1600, 4));
```

Ajouter une voiture c'est bien! Mais la suppression est certainement moins simple. Il faudrait que chaque véhicule soit identifier de façon unique dans la garage. Pour un véhicule, c'est très simple, nous disposons d'un numéro de châssis que nous ajoutons comme propriété à la classe véhicule.

Vehicule 1
Vehicule 2
Vehicule 3
Vehicule 4
Vehicule 5

Vehicule 1
Vehicule 2
Vehicule 4
Vehicule 5
null

C# L'orienté objet

- Les collections – Supprimer un élément

```
public Vehicule Remove(string NumChassis)
{
    for (int i=0;i<this.Indice;i++)
    {
        if (this.Stock[i].NumChassis==NumChassis)
        {
            this.RemoveAt(i);
        }
    }
    throw new Exception("Vehicule non trouvé");
}
```

La suppression d'un élément dans un tableau n'est pas simple car il faut remonter tous les éléments suivants d'une case.

C# L'orienté objet

- Les collections – Supprimer un élément

```
private void RemoveAt(int index)
{
    int i;
    for (i = index; i < this.Indice - 2; i++)
    {
        this.Stock[i] = this.Stock[i + 1];
    }
    if (this.Indice > 0)
    {
        this.Stock[this.Indice - 1] = null;
        this.Indice--;
    }
}
```

C# L'orienté objet

- Les collections – les indexeurs

Imaginons toujours notre garage avec les véhicules identifiés par leur numéro de châssis. Il serait intéressant de pouvoir utiliser une syntaxe suivante

```
string NumChassis = "xxxxx";  
Vehicule v1 = Garage[ NumChassis];
```

Cette syntaxe ressemble très fortement à celle utilisée pour accéder à un élément dans un tableau. Dans notre cas, l'indexeur devrait accepter une chaîne de caractères également.

C# L'orienté objet

- Les collections – les indexeurs

Nous recréons une classe Vehicule comprenant une variable membre ainsi que la propriété correspondante pour le numéro de châssis.

```
class Vehicule
{
    private string _Marque;
    private string _NumChassis;
    public Vehicule(string Marque, string NumChassis) ...
    public string NumChassis ...
}
```

C# L'orienté objet

- Les collections – les indexeurs

Nous pouvons maintenant créer notre classe garage qui s'appuiera sur une liste générique pour assurer la collection de véhicules.

```
private List<Vehicule> _ListVehicules;  
public Garage()...  
public Vehicule this[string NumChassis]  
{  
    get  
    {  
        foreach(Vehicule veh in _ListVehicules)  
        {  
            if (veh.NumChassis == NumChassis)  
            {  
                return veh;  
            }  
        }  
        throw new Exception("Vehicule non trouvé");  
    }  
}
```

C# L'orienté objet

- Les collections – les indexeurs

L'indexeur repose sur l'utilisation de la syntaxe suivante:

```
public Vehicule this[string NumChassis]
```

Nous remarquons qu'il s'agit d'une propriété puisque nous retrouvons un assesseur `get`

Cet assesseur retourne bien le véhicule de la liste correspondant au numéro de châssis renseigné.

```
foreach(Vehicule veh in _ListVehicules)
{
    if (veh.NumChassis == NumChassis)
    {
        return veh;
    }
}
throw new Exception("Vehicule non trouvé");
```

C# L'orienté objet

- Les collections – les itérateurs

Dans le cas de collections, nous pouvons parcourir une collection en utilisant l'instruction foreach. Elle remplace de façon plus compacte la boucle for.

Prenons le cas d'un tableau d'entiers

```
int[] tab = { 1, 2, 3, 4, 5 };  
foreach( int data in tab)  
{  
    Console.WriteLine(data);  
}
```

Imaginons que l'on puisse faire la même chose en parcourant le garage et en récupérant l'ensemble des véhicules.... Pour être certain d'implémenter les bonnes méthodes, nous utiliserons la technique d'héritage sur des interfaces

C# L'orienté objet

- Les collections – les itérateurs

```
class Garage:IEnumerable<Vehicule>
{
    private List<Vehicule> _ListVehicules;
    public Garage()...
    public Vehicule this[string NumChassis]...
    public IEnumerator<Vehicule> GetEnumerator()
    {
        for (int i = 0; i < this._ListVehicules.Count; i++)
        {
            yield return this._ListVehicules[i];
        }
    }
    public IEnumerable<Vehicule> GetReverse()...

    IEnumerator IEnumerable.GetEnumerator()...
}
```

C# L'orienté objet

- Les collections – les itérateurs

Voici les points importants à retenir dans cette mise en place

```
class Garage:IEnumerable<Vehicule>
```

Notre classe Garage doit hériter de l'interface `IEnumerable` qui dans notre cas sera une interface générique `IEnumerable<Vehicule>` Cette interface nous amènera à implémenter la méthode suivante...

```
public IEnumerator<Vehicule> GetEnumerator()  
{  
    for (int i = 0; i < this._ListVehicules.Count; i++)  
    {  
        yield return this._ListVehicules[i];  
    }  
}
```

L'utilisation d'un foreach provoquera l'appel à la méthode `GetEnumerator()`

C# L'orienté objet

- Les collections – les itérateurs

L'utilisation "normale" d'une instruction `return` provoquerait la sortie de la méthode. C'est la raison pour laquelle nous retrouvons le mot clef `yield`. Il est possible d'ajouter d'autres méthodes permettant d'obtenir un autre ordre de sortie des objects.

```
public IEnumerable<Vehicule> GetReverse()  
{  
    for (int i = this._ListVehicules.Count; i >= 0; i--)  
    {  
        yield return this._ListVehicules[i];  
    }  
}
```

C# Les délégués

- Principe du délégué

Nous avons vu dans le C# que vous pouvions créer une référence permettant de pointer vers un objet.

Il est également possible de créer une référence qui pointe vers une méthode. Ce concept est similaire à ce que nous pouvons retrouver en C sous la dénomination de pointeurs de fonction.

Les délégués sont omniprésents dans la programmation C# et il est donc très important de comprendre leur fonctionnement

C# Les délégués

- Un exemple concret

Imaginons la mise en place d'un Timer. Lorsque l'intervalle de temps est écoulé, une méthode doit être exécutée pour effectuer un traitement

```
static void Main(string[] args)
{
    Timer T1 = new Timer();
    T1.Interval = 1000;
    T1.Enabled = true;
    T1.Elapsed += T1_Elapsed;
    Console.ReadKey();
    T1.Stop();
    T1.Dispose();
}
```

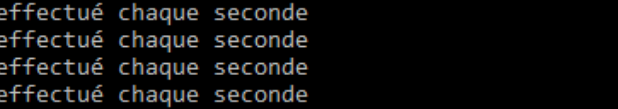
T1.Elapsed += T1_Elapsed; Dans cet exemple, la classe Timer va, pour l'objet T1, déléguer l'exécution d'un code lorsque l'intervalle de temps expire

C# Les délégués

- Un exemple concret

```
private static void T1_Elapsed(object sender,
ElapsedEventArgs e)
{
    Console.WriteLine("Travail effectué chaque seconde");
}
```

Dans notre exemple, la méthode vers laquelle pointe le délégué doit obligatoirement respecter les arguments tels que repris ci dessus



```
C:\Users\Emmanuel\Documents\Visual Studio 2017\Projects\C...  
Travail effectué chaque seconde  
Travail effectué chaque seconde  
Travail effectué chaque seconde  
Travail effectué chaque seconde  
Travail effectué chaque seconde  
Travail effectué chaque seconde
```

C# Les délégués

- Un exemple concret

Il serait impossible de prévoir dans cette classe Timer le code personnalisé pour tous les besoins. Microsoft préfère déléguer cette responsabilité à l'extérieur de la classe.

Nous imaginons notre propre classe de gestion de stock pour des articles et nous souhaitons pouvoir laisser au programmeur utilisant notre classe le soin d'implémenter la méthode appelée lorsque le stock aura une valeur critique.

Pour prendrons pour ce faire une gestion d'articles.

C# Les délégués

- Un exemple concret

```
class Article
{
    public string Nom { get; private set; }
    public int Stock { get; private set; }

    public Article(string Nom, int Stock)...

    public void Retire(int qte)
    {
        if (qte >= this.Stock) this.Stock -= qte;
        else throw new Exception("Pas assez de stock");
    }

    public void Ajout(int qte)...
}
```


C# Les délégués

- Un exemple concret

Dans le cas le plus complet de l'utilisation des délégués, nous allons implémenter un type délégué. Le but de ce type est de définir la carte d'identité de la méthode.

Nous définirons un niveau d'alerte de sorte que si le niveau en stock devient inférieur, la méthode de commande devra être appelée. Nous modifions notre classe Article...

```
public string Nom { get; private set; }  
public int Stock { get; private set; }  
public int NiveauAlert { get; private set; }
```

```
public Article(string Nom, int Stock, int NiveauAlert)  
{  
    this.Nom = Nom;  
    this.Stock = Stock;  
    this.Alert = null;  
    this.NiveauAlert = NiveauAlert;  
}
```

C# Les délégués

- Un exemple concret

La déclaration d'un type délégué

```
public delegate void AlertStock(Article sender);
```

Ce type indique bien que toute méthode vers laquelle pointerait une référence de ce type aurait ces caractéristiques. Nous pouvons maintenant créer une référence de ce type là.

```
public AlertStock Alert;
```

```
public Article(string Nom, int Stock, int NiveauAlert)
{
    this.Nom = Nom;
    this.Stock = Stock;
    this.Alert = null;
    this.NiveauAlert = NiveauAlert;
}
```

C# Les délégués

- Un exemple concret

La référence est initialisée dans le constructeur `this.Alert = null;`

```
public void Retire(int qte)
{
    if (qte <= this.Stock) this.Stock -= qte;
    else throw new Exception("Pas assez de stock");
    if (this.Stock < this.NiveauAlert && Alert != null)
    {
        Alert(this);
    }
}
```

La méthode ne pourra être appelée au travers de la référence que si elle a été initialisée à l'extérieure de la classe. Si la référence est nulle, elle ne peut pas être appelée. Si elle a été initialisée, on peut appeler la méthode correspondante en passant l'objet lui même en argument

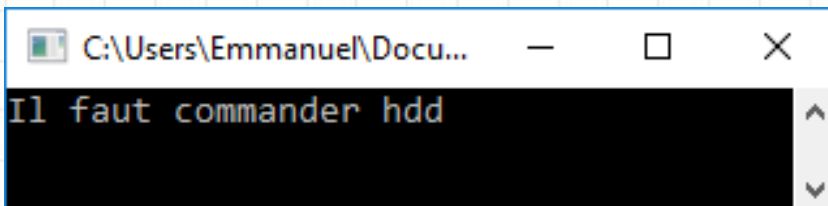
C# Les délégués

- Un exemple concret

```
Article art1 = new Article("hdd", 10,5);  
art1.Alert = new Article.AlertStock(Commander);  
art1.Retire(6);  
Console.ReadKey();
```

```
static void Commander(Article art1)  
{  
    Console.WriteLine("Il faut commander {0}", art1.Nom);  
}
```

La référence Alert liée à l'objet art1 est bien initialisée en instanciant le type délégué AlertStock. Remarquons en argument le nom de la méthode qui sera appelée




```
C:\Users\Emmanuel\Docu...  
Il faut commander hdd
```

C# Les délégués

- Délégués et événements

Si nous prenons la programmation événementielle, nous retrouvons une utilisation intensive des délégués. Prenons en WinForms la gestion d'un click sur le bouton par exemple...

```
this.btValider.Click += new System.EventHandler(this.btValider_Click);
```

 `delegate void System.EventHandler(object sender, EventArgs e)`
Représente la méthode qui gèrera un événement qui n'a aucune donnée d'événement.

```
private void btValider_Click(object sender, EventArgs e)
{
}
}
```

Nous pourrions utiliser ce délégué de sorte de l'implémenter dans la gestion de nos articles. Adaptons le code créé précédemment.

C# Les délégués

- Délégués et événements

```
public System.EventHandler Alert;
```

```
public void Retire(int qte)
{
    if (qte <= this.Stock) this.Stock -= qte;
    else throw new Exception("Pas assez de stock");
    if (this.Stock < this.NiveauAlert && Alert != null)
    {
        Alert(this, null);
    }
}
```

```
art1.Alert = new EventHandler(Commander);
```


C# Les délégués

- Délégués et événements

```
static void Commander(Object art1, EventArgs e)
{
    Console.WriteLine("Il faut commander {0}",
        ((Article)art1).Nom);
}
```

Le seul aspect négatif est probablement le transtypage nécessaire puisque le délégué renseigne un type Object comme premier argument

C# Les délégués

- Délégués et les threads

Un thread est vu comme la plus petite unité de code qui puisse recevoir à elle seule des ressources processeur. Une application peut être composée de plusieurs threads, chacun d'eux pouvant s'exécuter dans des cœurs différents dans votre infrastructure.

Comment mettre en œuvre cette technique au niveau du C#?

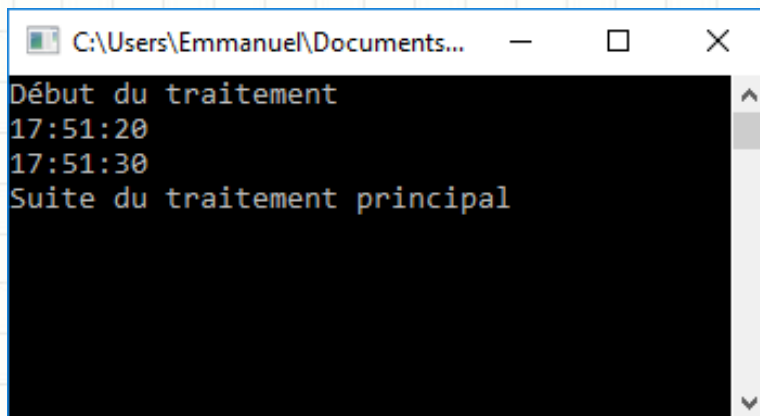
Imaginons un traitement long à exécuter...

```
static void traitement()  
{  
    for (int i=0;i<100;i++)  
    {  
        Thread.Sleep(1000); //attente de 1000msec  
    }  
}
```

C# Les délégués

- Délégués et les threads

```
static void Main(string[] args)
{
    Console.WriteLine("Début du traitement");
    Console.WriteLine(DateTime.Now.ToLongTimeString());
    traitement();
    Console.WriteLine(DateTime.Now.ToLongTimeString());
    Console.WriteLine("Suite du traitement principal");
    Console.ReadKey();
}
```



A screenshot of a Windows console window titled "C:\Users\Emmanuel\Documents...". The window has a black background with white text. The output shows the program's execution: "Début du traitement", followed by the timestamp "17:51:20", then "17:51:30", and finally "Suite du traitement principal". The window includes standard Windows window controls (minimize, maximize, close) in the title bar.

```
C:\Users\Emmanuel\Documents...
Début du traitement
17:51:20
17:51:30
Suite du traitement principal
```

C# Les délégués

- Délégués et les threads

```
Console.WriteLine("Traitement principal démarré");  
Thread MyThread = new Thread(new ThreadStart(Montraitement));  
MyThread.Start();  
Console.WriteLine("Poursuite du traitement principal");  
Console.ReadKey();
```

`new Thread(new ThreadStart(Montraitement))` permet la création d'un thread avec comme argument un délégué basé sur le type délégué `ThreadStart`
La méthode vers laquelle pointe le délégué est `Montraitement`

C# Les délégués

- Délégués et les threads

```
private static void Montraitement()  
{  
    Console.WriteLine("Travail de fond commencé");  
    for (int i=0;i<20;i++)  
    {  
        Thread.Sleep(1000);  
        Console.WriteLine(DateTime.Now.ToLongTimeString());  
    }  
    Console.WriteLine("Travail de fond terminé");  
}
```

C# Les délégués

- Délégués et les threads

La mise en place d'un délégué est complexe:

1. Il faut créer un type délégué
2. Il faut créer une méthode
3. Il faut instancier le type délégué et faisant passer la méthode en argument

Deux techniques vont permettre de simplifier l'utilisation des délégués:

1. Utilisation des méthodes anonymes (sans nom)
2. Utilisation des expressions Lambda

C# Les délégués

- Délégués et les méthodes anonymes

```
Thread MyThread = new Thread(delegate () {  
    Console.WriteLine("Travail de fond commencé");  
    for (int i = 0; i < 20; i++)  
    {  
        Thread.Sleep(1000);  
        Console.WriteLine(DateTime.Now.ToLongTimeString());  
    }  
    Console.WriteLine("Travail de fond terminé");  
});
```

`new Thread(delegate ()` Nous retrouvons le code de la méthode `Montraitement()` alors que le nom n'est plus repris. Le code devient anonyme.

C# Les threads

- Démarrage et arrêt d'un thread

```
Thread MyThread = new Thread( //...);  
MyThread.Start();
```

Le démarrage du thread provoque l'entrée dans la méthode référencée par le délégué.

Pour arrêter un thread, plusieurs méthodes existent:

1. Sortie naturelle du thread lors de la sortie naturelle de la méthode
2. Sortie forcée du thread

```
MyThread.Abort();
```

Cette sortie forcée est provoquée par une exception.

```
Exception levée : 'System.Threading.ThreadAbortException' dans mscorlib.dll  
Le thread 0x4568 s'est arrêté avec le code 0 (0x0).
```

C# Les threads

- Thread Méthode ou Pool

Dans nos diapositives précédentes, nous utilisons une méthode pointée par un délégué. Une fois que le thread se termine, le Garbage Collector nettoie la mémoire occupée par ce thread

Un pool de thread est une collection de threads utilisables pour exécuter du code en arrière plan. Une fois que le thread termine sa tâche, il l'envoie au pool dans une file d'attente de threads, où il peut être réutilisé. Cette réutilisabilité évite à une application de créer plus de threads, ce qui permet de réduire la consommation de mémoire.

```
ThreadPool.QueueUserWorkItem(new WaitCallback(Montraiement),  
null);
```

Aucune méthode Abort n'existe mais il est possible d'utiliser un token pour y parvenir de façon naturelle.

C# Les threads

- Thread Méthode ou Pool

```
var cts = new CancellationTokenSource();  
ThreadPool.QueueUserWorkItem(new WaitCallback(Montraitement),  
cts.Token);  
Console.WriteLine("Poursuite du traitement principal");  
Console.ReadKey();  
cts.Cancel();  
Console.ReadKey();
```

C# Les threads

- Thread. Méthode ou Pool?

```
private static void Montraitement(object cts)
{
    CancellationToken token = (CancellationToken)cts;
    Console.WriteLine("Travail de fond commencé");
    for (int i=0; i<20 && !token.IsCancellationRequested; i++)
    {
        Thread.Sleep(1000);
        Console.WriteLine(DateTime.Now.ToLongTimeString());
    }
    Console.WriteLine("Travail de fond terminé");
}
```

C# Les threads

- Threads et ressources communes

La synchronisation est un concept utilisé pour empêcher plusieurs threads d'accéder simultanément à une ressource partagée. Vous pouvez l'utiliser pour empêcher plusieurs threads d'appeler simultanément les propriétés ou les méthodes d'un objet.

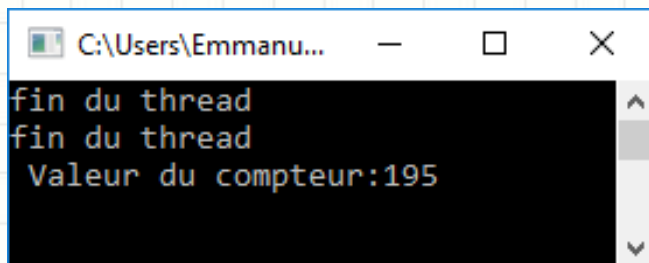
Prenons un exemple amenant une impossibilité de deviner le résultat de l'exécution d'un tel code

```
static void Traitement()  
{  
    for (int i=0;i<100;i++)  
    {  
        Thread.Sleep(50);  
        count++;  
    }  
    Console.WriteLine("fin du thread");  
}
```

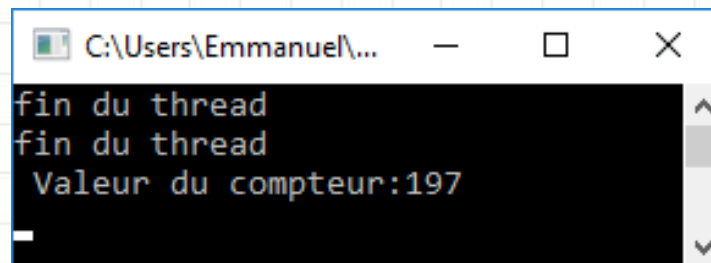

C# Les threads

- Threads et ressources communes

```
static int count = 0;
static void Main(string[] args)
{
    Thread tr1 = new Thread(new ThreadStart(Traitement));
    Thread tr2 = new Thread(new ThreadStart(Traitement));
    tr1.Start();
    tr2.Start();
    Console.ReadKey();
    Console.WriteLine("Valeur du compteur:{0}",count);
    Console.ReadKey();
}
```



```
C:\Users\Emmanu...
fin du thread
fin du thread
Valeur du compteur:195
```



```
C:\Users\Emmanuel\...
fin du thread
fin du thread
Valeur du compteur:197
```

C# Les threads

- Threads et ressources communes

Nous retrouvons dans les mécanismes de synchronisation, les verrous exclusifs et les verrous non exclusifs.

Le verrouillage exclusif est utilisé pour garantir qu'à un moment donné, un et un seul thread peut entrer dans une section critique. Nous retrouverons les mécanismes suivants: Lock (ou Monitor), Mutex et SpinLock

Dans le verrouillage non exclusif, plusieurs threads peuvent entrer dans une section commune mais nous pouvons limiter la concurrence. Nous retrouverons les mécanismes suivants: Semaphore, SemaphoreSlim, ReaderWriterLockSlim

C# Les threads

- Le verrouillage exclusif de type "lock"

```
private static readonly Object thisLock = new Object();
```

```
static void Traitement()  
{  
    lock(thisLock)  
    {  
        for (int i = 0; i < 100; i++)  
        {  
            Thread.Sleep(50);  
            count++;  
        }  
    }  
    Console.WriteLine("fin du thread");  
}
```

C# Les threads

- Le verrouillage exclusif de type "lock"

En général, évitez de verrouiller un type public, ou des instances échappant au contrôle de votre code. Les constructions courantes `lock (this)`, `lock (typeof (MyType))` et `lock ("myLock")` violent cette directive :

- `lock(this)` pose problème s'il est possible d'accéder publiquement à l'instance.
- `lock(typeof (MyType))` pose problème s'il est possible d'accéder publiquement à `MyType`.
- `lock("myLock")` pose problème puisque tout autre code du processus utilisant la même chaîne partagera le même verrouillage.

C# Les threads

- "lock" ... la version simplifiée de Monitor

```
private static readonly Object thisLock = new Object();
```

```
static void Traitement()  
{  
    Monitor.Enter(thisLock);  
    for (int i = 0; i < 100; i++)  
    {  
        Thread.Sleep(50);  
        count++;  
    }  
    Monitor.Exit(thisLock);  
    Console.WriteLine("fin du thread");  
}
```

C# Les threads

- Le verrouillage inter processus

Un *mutex* est semblable à un moniteur ; il empêche l'exécution simultanée d'un bloc de code par plusieurs threads. En fait, le nom « mutex » est la contraction de « mutually exclusive » (qui s'excluent mutuellement). Cependant, à la différence d'un moniteur, un mutex peut être utilisé pour synchroniser des threads dans plusieurs processus.

Dans le cadre d'une synchronisation inter-processus, un mutex est appelé *mutex nommé* parce qu'il sera utilisé dans une autre application, et que par conséquent il ne peut pas être partagé au moyen d'une variable globale ou statique. Il est indispensable de lui attribuer un nom, afin que les deux applications puissent accéder au même objet mutex.

C# Les threads

- Les accès concurrentiels multiples autorisés

Le problème de l'accès concurrentiel est souvent lié aux opérations d'écritures tandis que des accès en lecture simultanés ne posent pas de problème.

```
static ReaderWriterLockSlim MLock = new ReaderWriterLockSlim();
```

```
static void Traitement()
{
    for (int i = 0; i < 100; i++)
    {
        Thread.Sleep(50);
        MLock.EnterWriteLock();
        count++;
        MLock.ExitWriteLock();
    }
    Console.WriteLine("fin du thread");
}
```

Il existe aussi les méthodes EnterReadLock() et ExitReadLock()

C# Les threads

- La classe Interlocked

Vous pouvez utiliser les méthodes de cette classe pour éviter les problèmes qui peuvent se produire lorsque plusieurs threads tentent simultanément de mettre à jour ou de comparer une même valeur. Les méthodes de cette classe vous permettent d'incrémenter, de décrémenter, d'échanger et de comparer des valeurs d'un thread en toute sécurité.

```
static void Traitement()  
{  
    for (int i = 0; i < 100; i++)  
    {  
        Thread.Sleep(100);  
        Interlocked.Increment(ref count);  
    }  
    Console.WriteLine("fin du thread");  
}
```

C# Les threads

- Les types délégués intégrés du Framework

Pour éviter la création de chaque type délégué dont nous aurions besoins, le Framework intègre maintenant des types délégués génériques: Action et Func
Action est un type délégué qui ne retourne pas de valeur (void) tandis que Func retourne une valeur/référence dont le type peut être défini

- `public delegate T Func< T >();`
- `public delegate T Func< A0, T >(A0 arg0);`
- `public delegate T Func<A0, A1, T> (A0 arg0, A1 arg1);`
- `public delegate T Func<A0, A1, A2, T >(A0 arg0, A1 arg1, A2 arg2);`
- `public delegate T Func<A0, A1, A2, A3, T> (A0 arg0, A1 arg1, A2 arg2, A3 arg3);`

C# Les threads

- Les tâches

Des threads peuvent être démarrés en utilisant le gestionnaire des tâches dans le Framework. Nous utiliserons la classe Task. Plusieurs techniques nous permettent de démarrer des threads

```
Task.Factory.StartNew((data) => { Console.WriteLine(data); },  
"Bonjour les amis");  
Task.Factory.StartNew(() => { Console.WriteLine("Bonjour à  
tous"); });  
Task task = new Task(new Action(DireBonjour));  
task.Start();  
Task.Run(() => DireBonjour());
```

`Task.Factory.StartNew` peut utiliser en argument des délégués de type Func ou de type Action, dépendant qu'il y ait un retour ou pas.

C# Les threads

- Les tâches

La classe Task comprend d'autres méthodes statiques tels que:

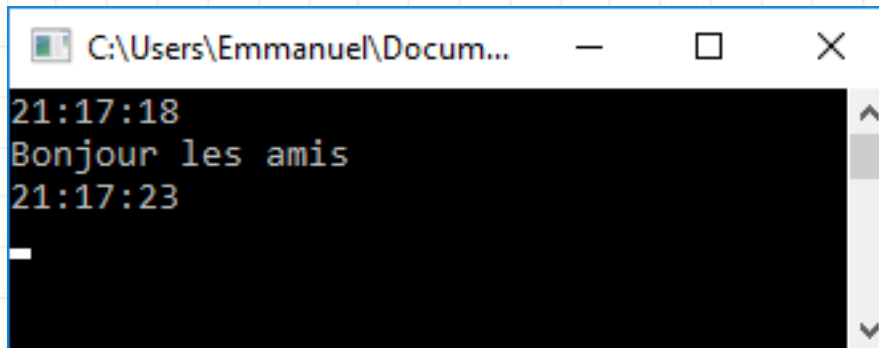
1. Task.Delay permettant un effet identique à Thread.Sleep
2. Task.WhenAny et Task.WhenAll permettant à un thread de s'exécuter automatiquement lorsque une ou plusieurs autres threads passés en argument sont terminés. Cette solution permettra la chaînage automatique des threads sans trop se soucier des techniques de synchronisation.
3. Task.WaitAny et Task.WaitAll permettant l'attente de fin d'exécution d'un des threads ou de l'ensemble des threads.

Imaginons que nous devons attendre la fin d'une tâche pour poursuivre notre traitement principal... La méthode `StartNew` retourne une tâche que nous pouvons manipuler

C# Les threads

- Les tâches

```
Console.WriteLine(DateTime.Now.ToLongTimeString());  
Task test = Task.Factory.StartNew((data) => {  
    Thread.Sleep(5000); Console.WriteLine(data); }, "Bonjour les  
amis");  
Task.WaitAll(test);  
Console.WriteLine(DateTime.Now.ToLongTimeString());
```



```
C:\Users\Emmanuel\Docum...  
21:17:18  
Bonjour les amis  
21:17:23  
_
```


C# Les threads

- Les taches

Nous pouvons très facilement chainer les taches grâce à cette classe.

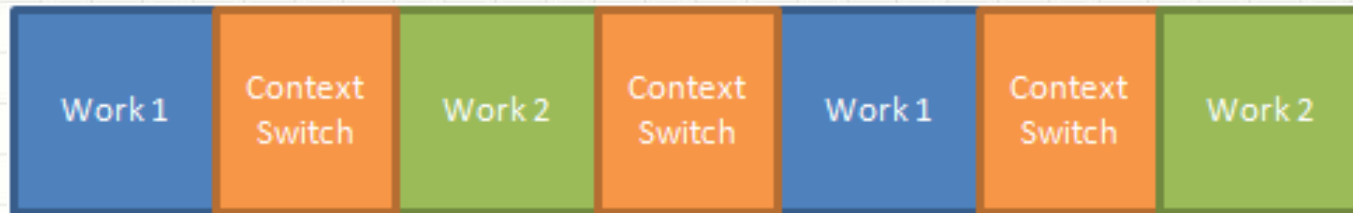
```
Console.WriteLine(DateTime.Now.ToLongTimeString());  
Task test = Task.Factory.StartNew((data) => {  
    Thread.Sleep(1000); Console.WriteLine($"Bonjour {data}"); },  
    "Antoine");  
test.ContinueWith((data) => {  
    Console.WriteLine($"Au revoir {data.AsyncState}"); });  
Task.WaitAll(test);
```

C# Les threads

- Threads et tâches. La différence

Des threads peuvent s'exécuter sur un même processeur. Dans ce cas, des mécanismes de commutation de threads doivent s'opérer.

Il est intéressant de travailler de façon parallèle en faisant en sorte que des tâches puissent s'exécuter sur des cœurs ou des processeurs différents. Dans ce cas, les performances sont naturellement meilleures



C# Les threads

- Les mots clefs async et await

Une méthode peut être marquée du mot clef async. Une telle méthode retourne un type void, un type Task ou Task<T>

```
static void Main(string[] args)
{
    Console.WriteLine(DateTime.Now.ToLongTimeString());
    LectureData();
    Console.WriteLine(DateTime.Now.ToLongTimeString());
    Console.ReadKey();
}
static async void LectureData()
{
    await Task.Delay(5000);
    Console.WriteLine("Fin de la tâche");
}
```

C# Les threads

- Les mots clefs async et await

```
Console.WriteLine(DateTime.Now.ToLongTimeString());
var traitement = GetData();
traitement.Wait();
string data = traitement.Result;
Console.WriteLine(DateTime.Now.ToLongTimeString());
Console.ReadKey();

static async Task<string> GetData()
{
    var Client = new WebClient();
    string url = "http://www.helha.be";
    string data = await Client.DownloadStringTaskAsync(url);
    return data;
}
```

C# Les threads

- Les accès asynchrones aux webapi

L'accès synchrone à des ressources du web réserve parfois la surprise d'une lenteur importante et d'un "Freeze" de votre application. Une solution est d'interroger l'api web de façon asynchrone.

Prenons pour exemple, l'accès à un service web en technologie REST (Accès suivant le protocole HTTP) et la manipulation de données au format JSON

```
static async Task<List<JsonTest>> GetJsonData(){  
    using (var client = new WebClient()){  
        Uri WebApiUri = new Uri(  
            "https://jsonplaceholder.typicode.com/posts");  
        var json = await client.DownloadStringTaskAsync(  
            WebApiUri);  
        var serializer = new JavaScriptSerializer();  
        var model = serializer.Deserialize<List<JsonTest>>(  
            json);  
        return model; }}
```

C# Les threads

- Les accès asynchrones aux webapi

```
Uri WepApiUri = new Uri(  
    "https://jsonplaceholder.typicode.com/posts");
```

L'url "https://jsonplaceholder.typicode.com/posts" est accessible sur internet pour des tests. En utilisant dans le protocole HTTP le verbe "Get", nous récupérons des données dans un format JSON dont voici un extrait. Nous pouvons utiliser cette URL dans un navigateur classique.

C# Les threads

- Le format JSON (JavaScript Object Notation)

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat ",
    "body": "quia et suscipit "
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae "
  }
]
```

C# Les threads

- Le format JSON (JavaScript Object Notation)

Dans notre exemple:

- Les crochets [] signifient que l'on a un tableau d'objet.
- Les accolades { } représentent un objet
- Chaque objet sera séparé du suivant par la virgule
- Dans un objet nous retrouvons des propriétés sous une forme:
 - "Propriété" : "Valeur"
 - Chaque propriété sera séparée de la suivante par la virgule

Pour permettre une manipulation de ces objets, le format JSON doit être désérialisé dans un format d'objet C# manipulable

Une étape importante consiste à créer une classe correspondant à la structure d'un objet récupéré en JSON

C# Les threads

- Le format JSON (JavaScript Object Notation)

```
public class JsonTest
{
    public string userId { get; set; }
    public string id { get; set; }
    public string title { get; set; }
    public string body { get; set; }
}
```

```
var serializer = new JavaScriptSerializer();
var model = serializer.Deserialize<List<JsonTest>>(
json);
```

La classe `JavaScriptSerializer` est accessible dans le Framework sous la référence `System.Web.Extensions`. Au travers de la méthode générique `Deserialize` nous pouvons récupérer un tableau d'objets sous la forme `List<JsonTest>`

C# Les threads

- Les méthodes asynchrones et fct de callback

Dans les mécanismes de fonctions asynchrones et de programmation événementiel, il est possible de travailler avec des fonctions de callback.

Reprenons notre exemple d'accès à notre Webapi et abordons les différents techniques offertes par la classe `WebClient`

```
var json1 = client.DownloadString(WepApiUri);  
var json = await client.DownloadStringTaskAsync(  
    "https://jsonplaceholder.typicode.com/posts");  
client.DownloadStringAsync(WepApiUri);
```

La première méthode est synchrone. Nous attendons la réception des données. Elle est bloquante.

La deuxième méthode vient d'être vue précédemment.

La troisième méthode est asynchrone. Elle n'est pas bloquante mais nous remarquons qu'il n'est pas possible de récupérer la donnée. La méthode retourne un type void.

C# Les threads

- Les méthodes asynchrones et fct de callback

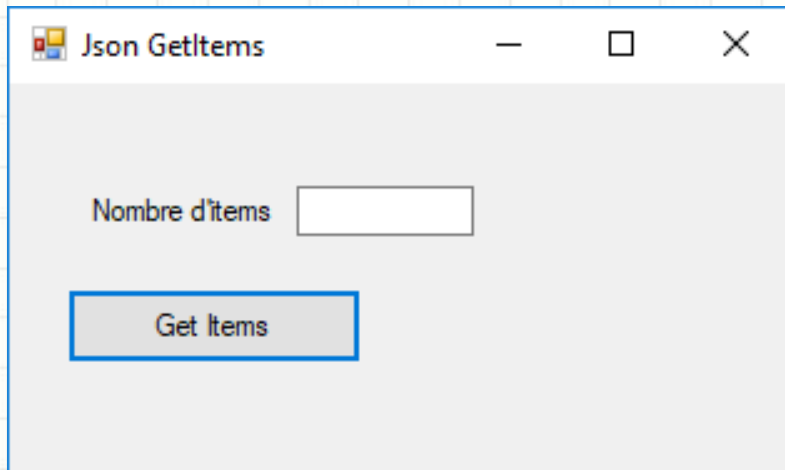
Nous devons donc implémenter une méthode permettant d'être averti lorsque les données sont accessibles

```
using (var client = new WebClient()){  
    Uri WepApiUri = new Uri(  
        "https://jsonplaceholder.typicode.com/posts");  
    client.DownloadStringCompleted +=  
        Client_DownloadStringCompleted;  
    client.DownloadStringAsync(WepApiUri);}
```

```
private static void Client_DownloadStringCompleted(object  
sender, DownloadStringCompletedEventArgs e)  
{  
    var serializer = new JavaScriptSerializer();  
    var model =  
        serializer.Deserialize<List<JsonTest>>(e.Result);  
}
```

C# Les threads

- Les contrôles Windows et Invoke



```
private void button1_Click(object sender, EventArgs e)
{
    MyThread = new Thread(new ThreadStart(GetJsonData2));
    MyThread.Start();
}
```


C# Les threads

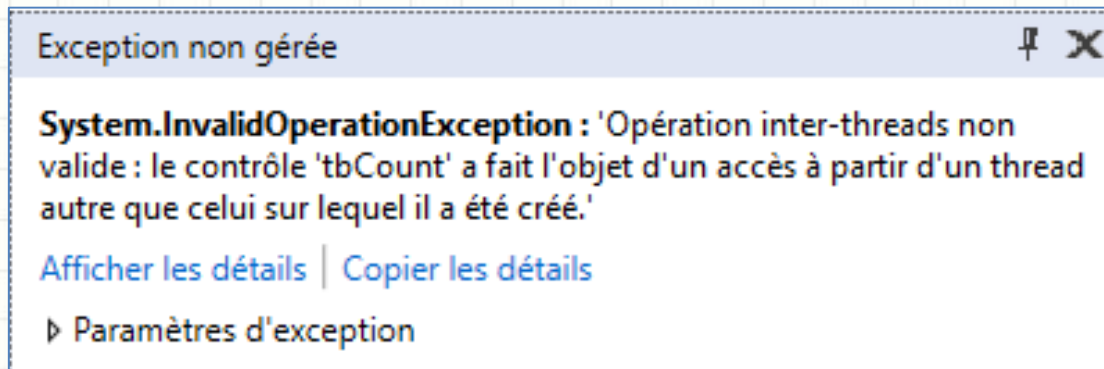
- Les contrôles Windows et Invoke

```
using (var client = new WebClient())
{
    Uri WepApiURi = new Uri(
        "https://jsonplaceholder.typicode.com/posts");
    var json = client.DownloadString(WepApiURi);
    var serializer = new JavaScriptSerializer();
    List<JsonTest> model =
        serializer.Deserialize<List<JsonTest>>(json);
    this.tbCount.Text = model.Count.ToString();
}
```

Le clic sur le bouton et l'appel de la méthode en résultant provoque une levée d'exception dont voici le contenu

C# Les threads

- Les contrôles Windows et Invoke



Pour éviter cette levée d'exception, nous ferons en sorte que le code appelé pour modifier le contenu de la textbox s'exécute sur le thread auquel la ressource graphique appartient.

```
this.tbCount.Invoke((Action)((()=>{ this.tbCount.Text =  
model.Count.ToString();})));
```

C# Language-Integrated Query

- Pourquoi Linq

Initialement, la façon dont l'accès se fait aux données dépend de leur nature (dataset, collections, bases de données...)

Linq permet d'uniformiser le langage de requête. Nous retrouverons donc

- LINQ to entities : Liée à l'utilisation de Entity Framework
- LINQ to Object : Liée à l'utilisation de collection (Liste, IEnumerable ...)
- LINQ to SQL : Liée à des requête SQL
- LINQ to Dataset : Liée à l'utilisation des dataset
- LINQ to XML : Liée à l'utilisation de donnée XML

La partie la plus visible de LINQ est sans doute l'expression de la requête. Les expressions de requête sont écrites dans une syntaxe de requête déclarative. En utilisant la syntaxe de requête, nous pouvons effectuer des opérations de filtrage, d'ordonnancement et de regroupement sur des sources de données avec un minimum de code. Pour des questions de facilité, nous aborderons les requêtes sur des listes génériques.

C# Language-Integrated Query

- Pourquoi Linq

Soit une classe représentant un client

```
class Client
{
    public string Id { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
}
```

Nous créons une liste de ce type qui nous servira de base pour nos requêtes Linq

```
List<Client> Contacts = new List<Client>();
Contacts.Add(new Client{Nom="Dupond",Prenom="Jean",Id="1" });
Contacts.Add(new Client{Nom="Dubard",Prenom="Eric",Id="1" });
```

C# Language-Integrated Query

- Obtenir la source de données – from in select

Dans une requête LINQ, la clause "from" vient en premier pour introduire la source de données de laquelle nous récupérerons chaque objet (A l'image de ce que peut nous donner l'instruction foreach)

```
var result = from clt in Contacts select clt;
```

select nous permet de sélectionner l'objet lui même, une propriété seule ou même un nouvel objet comprenant une partie des propriétés

```
var result = from clt in Contacts select clt;  
var result2 = from clt in Contacts select clt.Nom;  
var result3 = from clt in Contacts select new { Nom = clt.Nom,  
Prenom = clt.Prenom };
```

C# Language-Integrated Query

- Trier les données - orderby

```
var result = from clt in Contacts orderby clt.Nom ascending
select clt.Nom;
```

Espion 1	
Nom	Valeur
▲ result.ToList()	Count = 5
[0]	"Amorison"
[1]	"Cron"
[2]	"Dubard"
[3]	"Dupond"
[4]	"VanTrimpon"
▶ Affichage brut	

C# Language-Integrated Query

- Filtrer les données - where

```
var result = from clt in Contacts where clt.Nom=="Cron" ||  
clt.Nom=="Amorison" orderby clt.Nom ascending select clt.Nom;
```

Espion 1

Nom	Valeur
result.ToList()	Count = 2
[0]	"Amorison"
[1]	"Cron"
Affichage brut	

C# Language-Integrated Query

- Groupement de données – group by

Nous ajoutons dans notre classe client une propriété "Localite" et nous regroupons les objets sur cette dernière.

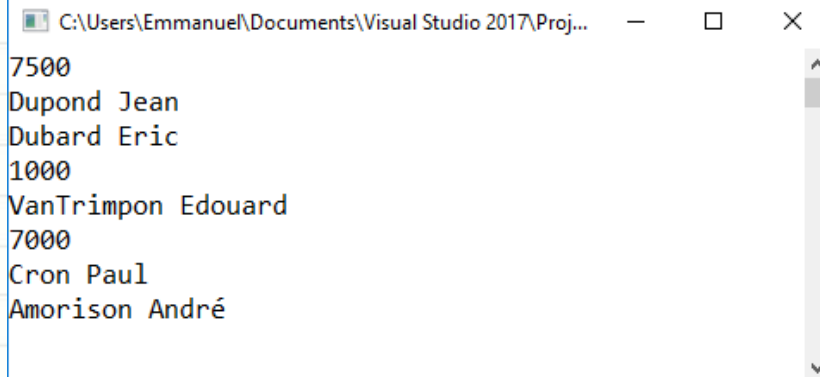
```
var result = from clt in Contacts group clt by clt.CodePostal;
```

Nos résultats prennent la forme d'une liste de listes. Chaque élément de la liste est un objet qui a un membre clé et une liste d'éléments qui sont regroupés sous cette clé. Lorsque nous parcourons une requête qui produit une séquence de groupes, nous devons utiliser une boucle foreach imbriquée.

C# Language-Integrated Query

- Groupement de données – group by

```
foreach (var grpclt in result)
{
    Console.WriteLine(grpclt.Key);
    foreach (var clt in grpclt)
    {
        Console.WriteLine(clt.Nom+ " " +clt.Prenom);
    }
}
Console.ReadKey();
```



```
C:\Users\Emmanuel\Documents\Visual Studio 2017\Proj...
7500
Dupond Jean
Dubard Eric
1000
VanTrimpon Edouard
7000
Cron Paul
Amorison André
```

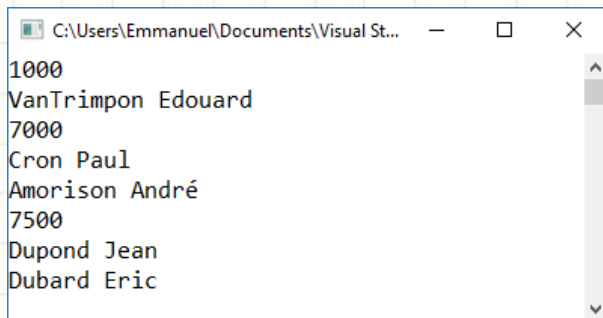
C# Language-Integrated Query

- Groupement de données – group by

Comme "group by" nous retourne une liste, nous pouvons utiliser les instructions de filtrage ou de tri vus précédemment.

```
var result = from clt in Contacts
              group clt by clt.CodePostal into custtgroup
              where custtgroup.Key == "1000"
              select custtgroup;
```

```
var result = from clt in Contacts
              group clt by clt.CodePostal into custtgroup
              orderby custtgroup.Key
              select custtgroup;
```



```
C:\Users\Emmanuel\Documents\Visual St...  -  □  ×
1000
VanTrimpon Edouard
7000
Cron Paul
Amorison André
7500
Dupond Jean
Dubard Eric
```

C# Language-Integrated Query

- Les jointures – join on

Les opérations de jointure créent des associations entre des séquences qui ne sont pas explicitement modélisées dans les sources de données.

Nous prendrons une liste de clients et une liste de commandes.

```
List<Client> Contacts = new List<Client>();
Contacts.Add(new Client { Nom = "Dupond", IdClient="1" });
Contacts.Add(new Client { Nom = "Dubard", IdClient="2"});
Contacts.Add(new Client { Nom = "VanTrimpon", IdClient="3"});

List<Order> Orders = new List<Order>();
Orders.Add(new Order{IdClient = "1", Denomination = "Order1"});
Orders.Add(new Order{IdClient = "1", Denomination = "Order2"});
Orders.Add(new Order{IdClient = "1", Denomination = "Order3"});
Orders.Add(new Order{IdClient = "2", Denomination = "Order4"});
Orders.Add(new Order{IdClient = "2", Denomination = "Order5"});
```

C# Language-Integrated Query

- Les jointures – join on

Nous souhaitons pour chaque commande avoir le client en correspondance. Voici le contenu de la requête Linq

```
var result = from clt in Contacts
              join ord in Orders
              on clt.IdClient equals ord.IdClient
              select new { Nom = clt.Nom, ord.Denomination };
```

Espion 1	
Nom	Valeur
result.ToList()	Count = 5
▶ [0]	{ Nom = "Dupond", Denomination = "Order1" }
▶ [1]	{ Nom = "Dupond", Denomination = "Order2" }
▶ [2]	{ Nom = "Dupond", Denomination = "Order3" }
▶ [3]	{ Nom = "Dubard", Denomination = "Order4" }
▶ [4]	{ Nom = "Dubard", Denomination = "Order5" }
▶ Affichage brut	

C# Language-Integrated Query

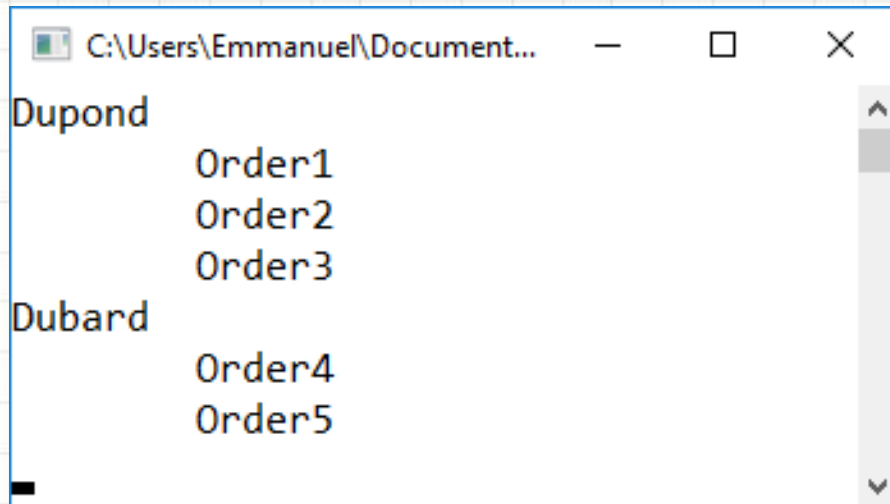
- Les jointures – join on et group by

```
var result = from clt in Contacts
              join ord in Orders
              on clt.IdClient equals ord.IdClient
              group ord by clt;
```

```
foreach(var cltord in result)
{
    Console.WriteLine(cltord.Key.Nom);
    foreach(var ord in cltord)
    {
        Console.WriteLine("\t"+ord.Denomination);
    }
}
```

C# Language-Integrated Query

- Les jointures – join on et group by



C# Language-Integrated Query

- Linq et Lambda

```
var result1 = Contacts.Where(x => x.Nom=="Dupond");
var result2 = Contacts.Where(x => x.Nom == "Dupond").Select(x
=> x.Nom);
var result3 = Contacts.Join(Orders,
    clt => clt.IdClient,
    ord => ord.IdClient,
    (clt, ord) => new {
        Nom=clt.Nom,Denomination=ord.Denomination}
    );
```

Espion 1	
Nom	Valeur
result3.ToList()	Count = 5
▶ [0]	{ Nom = "Dupond", Denomination = "Order1" }
▶ [1]	{ Nom = "Dupond", Denomination = "Order2" }
▶ [2]	{ Nom = "Dupond", Denomination = "Order3" }
▶ [3]	{ Nom = "Dubard", Denomination = "Order4" }
▶ [4]	{ Nom = "Dubard", Denomination = "Order5" }
▶ Affichage brut	

C# Gestion des bases de données

- Introduction

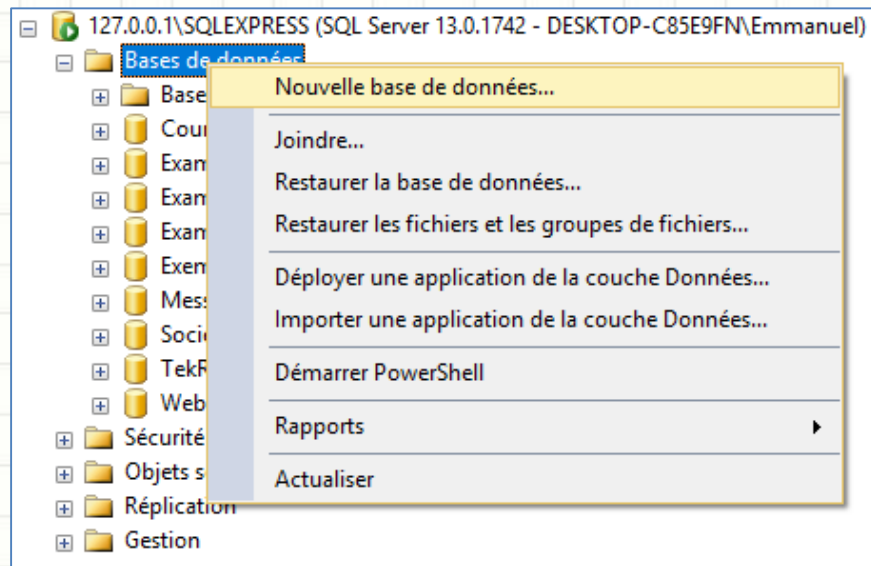
Nous allons aborder une gestion simple de base de données, composé en général d'une seule table. Nous travaillons avec SQLexpress de Microsoft pour lequel le solution SQL server management est installé.

Attention: veuillez à être cohérent entre la version du serveur SQL et du gestionnaire utilisé.

Une première démarche sera d'installer une base de données et ensuite envisager deux méthodes permettant d'accéder aux données suivant le paterne CRUD

C# Gestion des bases de données

- Création d'une base de données



Nom de la base de données : GestionClientelle

Propriétaire : <par défaut> ...

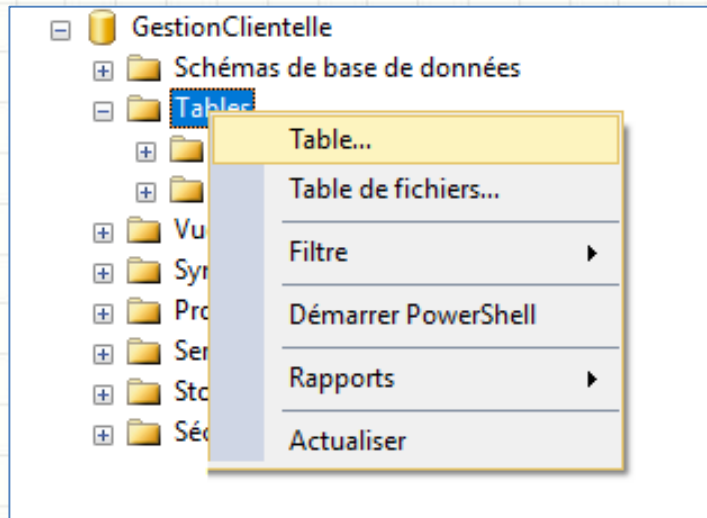
☒ Utiliser l'indexation de texte intégral

Fichiers de la base de données :

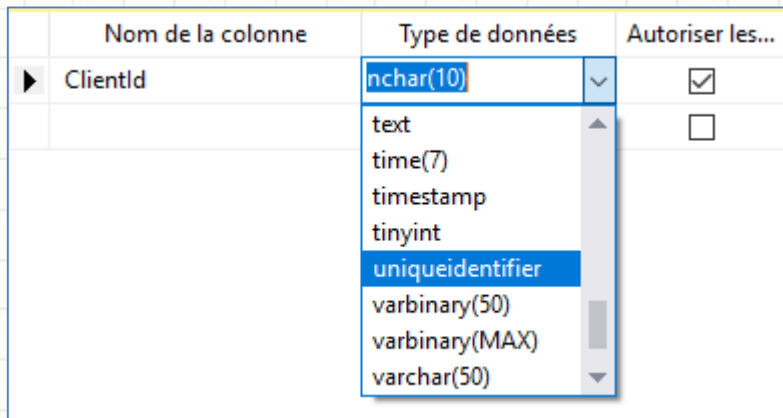
Nom logique	Type de fichier	Groupe de fichiers	Taille initiale (Mo)	Croissance automatique
GestionClien...	Données de ...	PRIMARY	8	Par 64 Mo, illimitée
GestionClien...	JOURNAL	Non applicable	8	Par 64 Mo, illimitée

C# Gestion des bases de données

- Création d'une table



Une table est un regroupement de données ayant des propriétés semblables. Nous pouvons créer une table de clients qui posséderont tous nom, prénom, adresse, identifiant unique...



Chacune des caractéristiques du client pourra être identifiée par un nom et nous pourrons lui affecter un type

C# Gestion des bases de données

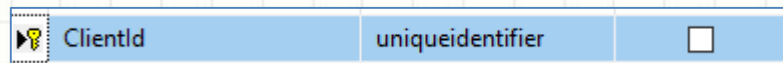
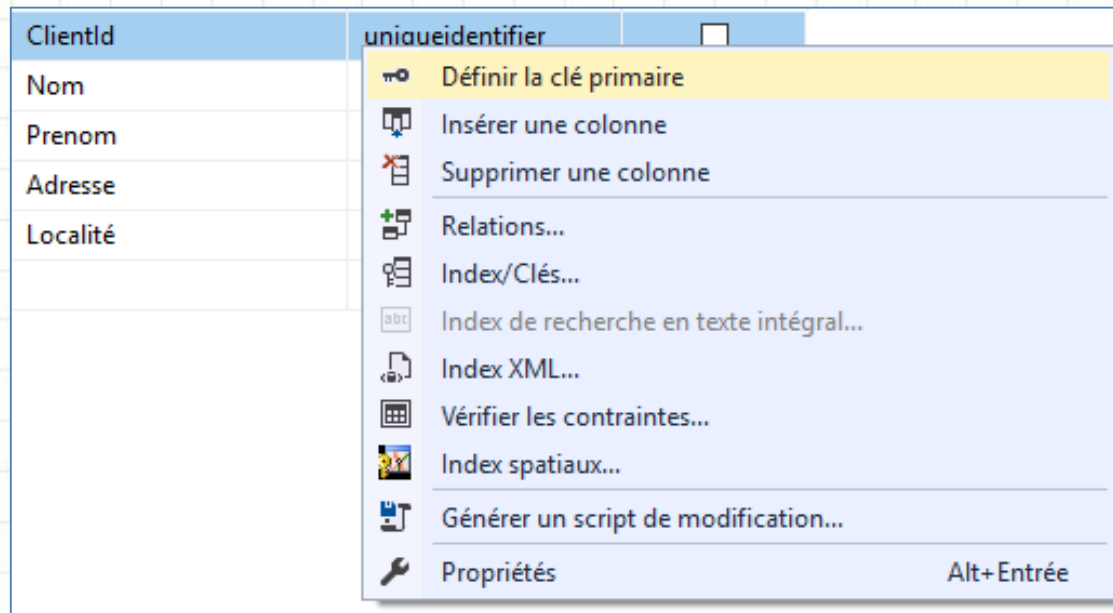
- Création d'une table

	Nom de la colonne	Type de données	Autoriser les...
	ClientId	uniqueidentifier	<input type="checkbox"/>
	Nom	nvarchar(50)	<input checked="" type="checkbox"/>
	Prenom	nvarchar(50)	<input checked="" type="checkbox"/>
	Adresse	nvarchar(50)	<input checked="" type="checkbox"/>
▶	Localité	nvarchar(50)	<input checked="" type="checkbox"/>
			<input type="checkbox"/>

Toute table doit posséder un identifiant unique qui deviendra la clef primaire de la table. Cet identifiant permet de d'identifier de façon unique un client dans notre exemple.

C# Gestion des bases de données

- Création d'une table



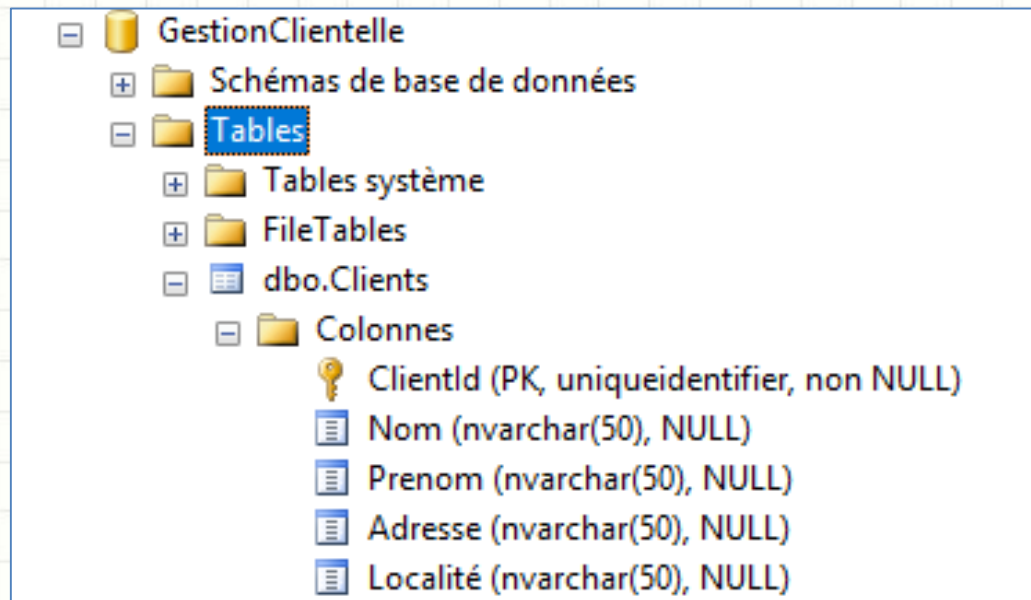
Nous pouvons accéder aux propriétés de la colonne ClientId de sorte qu'elle devienne un identifiant de ligne et que, de ce fait, la valeur soit affectée automatiquement lors de l'ajout d'un enregistrement

C# Gestion des bases de données

- Création d'une table

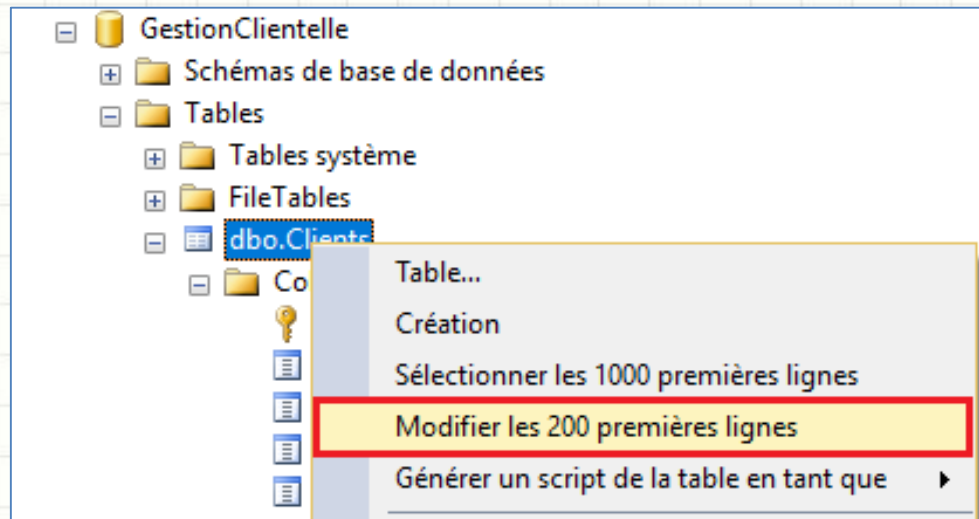
Publiée via DTS	Non
Répliquée	Non
RowGuid	Oui

Voici le contenu de la table une fois créée



C# Gestion des bases de données

- Ajout d'enregistrements dans la table



	ClientId	Nom	Prenom	Adresse	Localité
!	NULL	Wilfart	Emmanuel	Rue perdue, 10	Tournai
!	NULL	Dupond	Bernard	Rue trouvée, 20	Tournai
▶▶	NULL	NULL	NULL	NULL	NULL

C# Gestion des bases de données

- Ajout d'enregistrements dans la table

En fermant la fenêtre et en consultant les enregistrements ajoutés, nous obtiendrons alors

	ClientId	Nom	Prenom	Adresse	Localité
1	B859E96F-87C9-4F3E-81FA-83C5485607EE	Dupond	Bernard	Rue trouvée, 20	Toumai
2	C90DD82F-0336-427E-AD25-96BE4E9680C2	Wilfart	Emmanuel	Rue perdue, 10	Toumai

Remarquons le structure de l'identifiant unique ClientId.

Nous pouvons maintenant analyser le langage utilisé pour interroger la base de données pour effectuer des lectures, ajouts, suppressions et modifications d'enregistrements dans une table. Il s'agit du langage SQL (Structured Query Language)

C# Gestion des bases de données

- Manipulation des enregistrements dans la table

➤ Accès en lecture à la table Clients:

```
select * from dbo.Clients;  
select Nom,Prenom from dbo.Clients;  
select * from dbo.Clients where ClientId='B859E96F-87C9-4F3E-  
81FA-83C5485607EE'  
select * from dbo.Clients order by Nom;
```

➤ Insertion dans la table Clients

```
insert into dbo.Clients (Nom,Prenom,Adresse,Localité) values  
('Dubart','Jean','Rue Frinoise 12','Tournai');
```

Si certains champs ont la valeur NULL d'acceptée dans leur définition, alors ils peuvent ne pas être renseignés dans la requête insert

C# Gestion des bases de données

- Manipulation des enregistrements dans la table

➤ Suppression d'enregistrement(s) dans la table Clients:

```
delete from dbo.Clients where ClientId='C90DD82F-0336-427E-AD25-96BE4E9680C2';
```

➤ Modification d'enregistrement(s) dans la table Clients:

```
update dbo.Clients set Nom='Wilfart',Prenom='Jean' where ClientId='B859E96F-87C9-4F3E-81FA-83C5485607EE';
```

C# Gestion des bases de données

- Accès à la base de données

Dans cette partie, nous aborderons deux types d'accès:

1- L'accès "Legacy" en parcourant les classes SqlConnection et SqlCommand permettant d'interroger un serveur SQL. Il existe le pendant pour oledb et odbc

2- L'utilisation de LinqToSQL qui nous permettra d'interroger un serveur SQL sans passer par le langage de requête lui même au niveau de notre application mais d'utiliser le langage LINQ.

C# Gestion des bases de données

- Accès à la base de données

Nous considérons un accès à partir d'une application écrite en mode console.
N'oublions pas, par facilité d'écriture du code, d'insérer les lignes de code suivantes:

```
using System.Data;  
using System.Data.Sql;  
using System.Data.SqlClient;
```

```
SqlConnection MyConnect = new SqlConnection();  
MyConnect.ConnectionString =  
@"Server=127.0.0.1\SQLEXPRESS;Database=GestionClientelle;Truste  
d_Connection=True;";  
MyConnect.Open();
```

Le plus compliqué sera probablement la chaîne de connexion mais le site suivant peut vous y aider: <https://www.connectionstrings.com/sql-server/>

C# Gestion des bases de données

- Accès à la base de données

Une fois la connexion ouverte, nous pouvons envoyer des requêtes SQL vers le serveur à travers celle ci.

```
SqlCommand MyCmd = new SqlCommand();  
MyCmd.Connection = MyConnect;  
MyCmd.CommandType = CommandType.Text;  
MyCmd.CommandText = @"select * from dbo.clients";
```

Une fois la commande réalisée, nous pouvons maintenant envisager deux méthodes d'accès aux données:

1- Le mode connecté où l'on parcourt chaque enregistrement un par un et la connexion doit rester ouverte tout ce temps

2- Le mode déconnecté où nous allons charger en mémoire les enregistrements. Une fois chargés, nous pouvons nous déconnecter du serveur et parcourir les enregistrements en mémoire

C# Gestion des bases de données

- Accès à la base de données – mode connecté

Le mode connecté peut être associé à l'utilisation d'un DataReader et de sa méthode Read pour les accès en lecture. Les accès qui ne sont pas des lectures peuvent être associés à la méthode ExecuteNonQuery.

```
SqlDataReader MyRd = MyCmd.ExecuteReader();  
while (MyRd.Read())  
{  
    Console.WriteLine(MyRd["Nom"]);  
}  
MyRd.Close();
```

N'oublions pas de fermer le DataReader lorsque les accès ne sont plus nécessaires.

C# Gestion des bases de données

- Accès à la base de données – mode déconnecté

Le mode déconnecté peut être associé à l'utilisation d'un DataAdapter que nous n'aborderons pas dans cette introduction. Nous nous contenterons pour les lectures uniquement de passer par un DataReader et le remplissage d'un DataSet

```
SqlDataReader MyRd = MyCmd.ExecuteReader();
DataSet MyDS = new DataSet();
MyDS.Tables.Add(new DataTable());
MyDS.Tables[0].Load(MyRd);
foreach(DataRow client in MyDS.Tables[0].Rows)
{
    Console.WriteLine(client["Nom"]);
}
MyRd.Close();
```


Les tests unitaires

- Principe

Lorsque une application est créée, avant de la fournir au client, nous devons la tester. Même si beaucoup vérifie le bon fonctionnement global de leur application en l'exécutant, il est pourtant important de réfléchir aux tests avant d'écrire code.

Au lieu de tester le code dans son intégralité, il est important de pouvoir tester une application en ciblant les tests vers les plus petites entité de programme: la méthode.

Nous parlons alors de test unitaire. Il existe plusieurs Framework de tests disponibles autour de Visual Studio. Nous nous limiterons à utiliser celui intégré à Visual Studio.

Les tests unitaires

- Prenons la classe de comptes bancaires

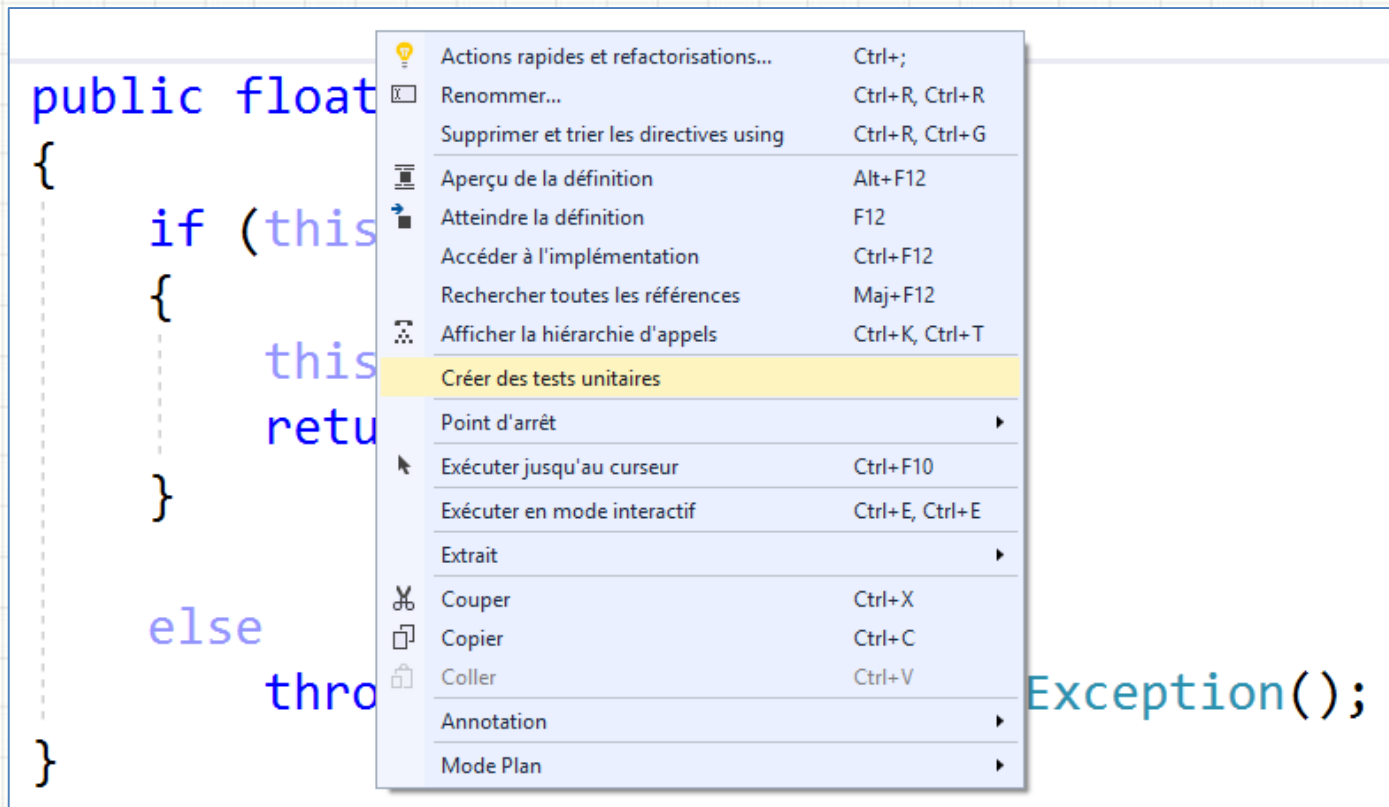
Dans cet exemple sur lequel nous avons déjà travaillé, nous avons une méthode permettant de débiter un compte bancaire et nous voudrions tester cette méthode

```
public float Debit(float montant)
{
    if (this.Solde >= montant)
    {
        this.Solde -= montant;
        return this.Solde;
    }
    else
        throw new SoldeInsuffisantException();
}
```

Les tests unitaires

- Implémentation d'un test unitaire

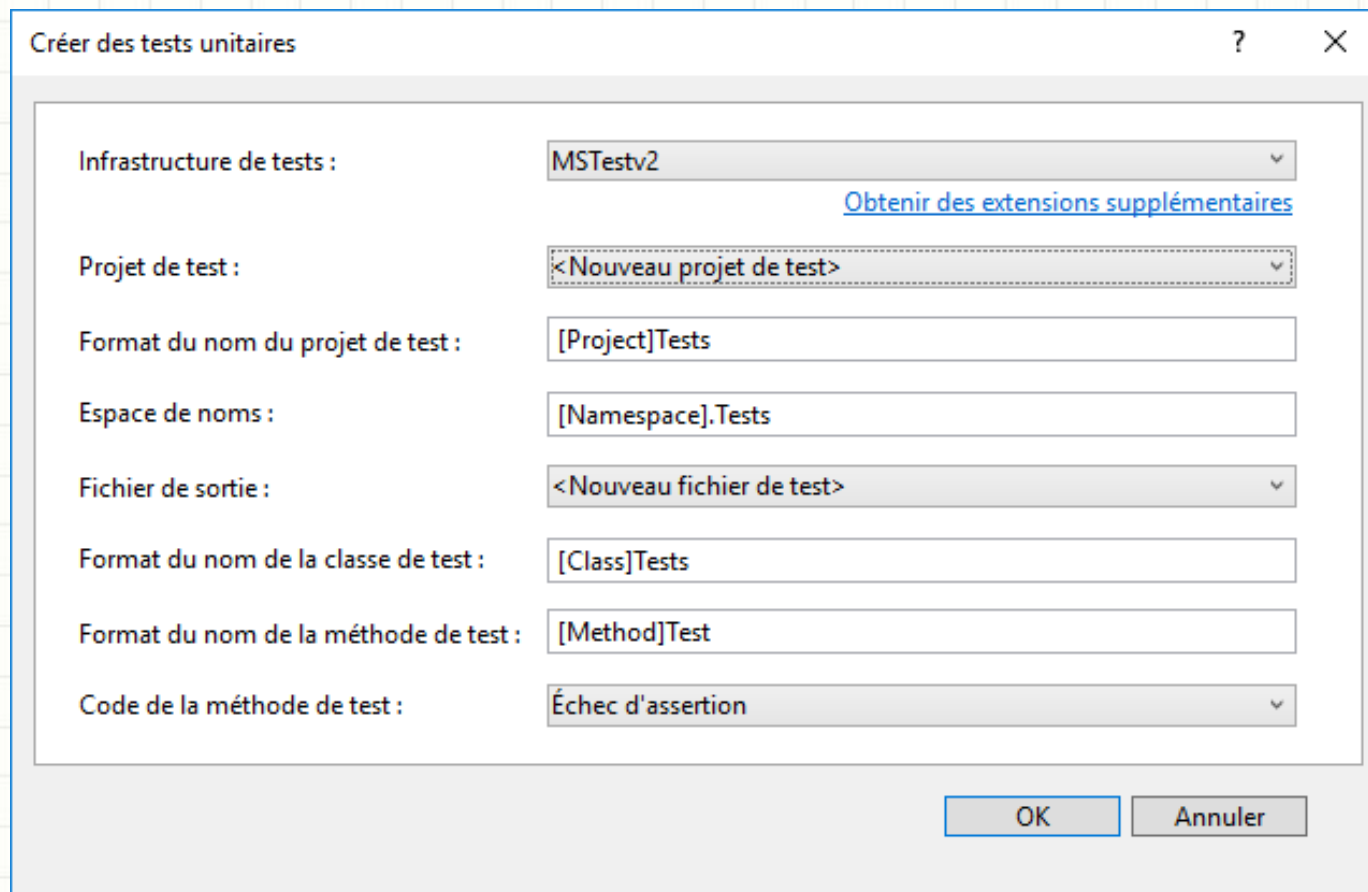
Nous pouvons accéder à un menu contextuel qui nous permettra de créer un test unitaire.



Les tests unitaires

- Implémentation d'un test unitaire

Nous obtenons alors l'écran de configuration suivant



The screenshot shows a dialog box titled "Créer des tests unitaires" (Create unit tests) with a question mark icon and a close button. The dialog contains several configuration options for creating unit tests:

- Infrastructure de tests :** A dropdown menu showing "MSTestv2". A link "Obtenir des extensions supplémentaires" (Get additional extensions) is visible to the right.
- Projet de test :** A dropdown menu showing "<Nouveau projet de test>" (New test project).
- Format du nom du projet de test :** A text box containing "[Project]Tests".
- Espace de noms :** A text box containing "[Namespace].Tests".
- Fichier de sortie :** A dropdown menu showing "<Nouveau fichier de test>" (New test file).
- Format du nom de la classe de test :** A text box containing "[Class]Tests".
- Format du nom de la méthode de test :** A text box containing "[Method]Test".
- Code de la méthode de test :** A dropdown menu showing "Échec d'assertion" (Assertion failure).

At the bottom right, there are two buttons: "OK" and "Annuler" (Cancel).

Les tests unitaires

- Implémentation d'un test unitaire

```
namespace CompteBancaireLib.Tests
{
    [TestClass()]
    public class CompteBancaireTests
    {
        [TestMethod()]
        public void DebitTest()
        {
            Assert.Fail();
        }
    }
}
```

Les tests unitaires

- Implémentation d'un test unitaire

Le modèle AAA (Arrange, Act, Assert) est un moyen couramment utilisé pour écrire les tests unitaires d'une méthode testée.

- La section **Arrange** d'une méthode de test unitaire initialise les objets et définit la valeur des données transmises à la méthode testée.
- La section **Act** appelle la méthode testée avec les paramètres triés.
- La section **Assert** vérifie que l'action de la méthode testée se comporte comme prévu.

Nous allons adapter la méthode de test en suivant ce modèle proposé

Les tests unitaires

- Implémentation d'un test unitaire

```
public void DebitTest()
{
    // arrange
    float SoldeActuel = 100.0F;
    float MontantRetrait = 20.0F;
    float expected = SoldeActuel - MontantRetrait;
    var compte = new CompteBancaire("AAAA", SoldeActuel);
    // act
    var NouvSolde=compte.Debit(MontantRetrait);
    // assert
    Assert.AreEqual(expected, NouvSolde);
}
```

Les tests unitaires

- Implémentation d'un test unitaire

Si le test proposé précédemment effectue un test dans une situation "normale", nous pouvons envisager le cas de figure où une exception est levée du fait d'un solde insuffisant. Nous créons une autre méthode de test

```
[TestMethod()]
[ExpectedException(typeof (SoldeInsuffisantException))]
public void DebitTestSoldeInsuffisant()
{
    // arrange
    float SoldeActuel = 10.0F;
    float MontantRetrait = 20.0F;
    float expected = SoldeActuel - MontantRetrait;
    var compte = new CompteBancaire("AAAA", SoldeActuel);
    // act
    var NouvSolde = compte.Debit(MontantRetrait);
}
```

Les tests unitaires

- Implémentation d'un test unitaire

Et si la méthode ne génère pas d'exception comme attendu. Nous remplaçons, dans le code de la méthode testée, l'instruction throw par un return 0.

▲ CompteBancaireTests (2)		
✖	DebitTestSoldesInsuffisant	17 ms
✔	DebitTest	8 ms

DebitTestSoldesInsuffisant [Copier tout](#)

Source : [CompteBancaireTests.cs](#) ligne 32

✖ Test Échec - DebitTestSoldesInsuffisant

Message : La méthode de test n'a pas levé l'exception attendue
CompteBancaireLib.SoldesInsuffisantException.

Temps écoulé : 0:00:00,0175833

Les tests unitaires

- Délais d'attente pour les tests unitaires

Nous pouvons définir des délais d'attente sur des méthodes de test

```
[TestMethod()]  
[Timeout(2000)] // Millisecondes  
public void DebitTest()  
{  
    //.....  
}
```

```
[TestMethod()]  
[Timeout(TestTimeout.Infinite)] // Millisecondes  
public void DebitTest()  
{  
    //.....  
}
```

Entity Framework

- Introduction

Entity Framework est un mappeur relationnel objet (O / RM) qui permet aux développeurs .NET de travailler avec une base de données utilisant des objets .NET. Il élimine le besoin de la plupart du code d'accès aux données que les développeurs ont généralement besoin d'écrire.

Dans les notes précédentes, nous avons abordé la gestion des bases de données sous plusieurs aspects:

- **SqlConnection, SqlCommand, SqlDataReader ou DataAdapter.** nous pouvons accéder à des procédures stockées où directement créer une requête SQL dans le code.
- **LinqToSQL.** Nous nous affranchissons du SQL avec un langage unique quel que soit la source de données. Un désavantage est le mappage un-à-un entre les tables et les classes et le fait d'être limité à SQL Server. Certains reprendront aussi comme défaut l'approche "Database first"

Entity Framework

- Les différentes approches

Database First

- We design our tables
- EF generates domain classes

Code First

- We create our domain classes
- EF generates database tables

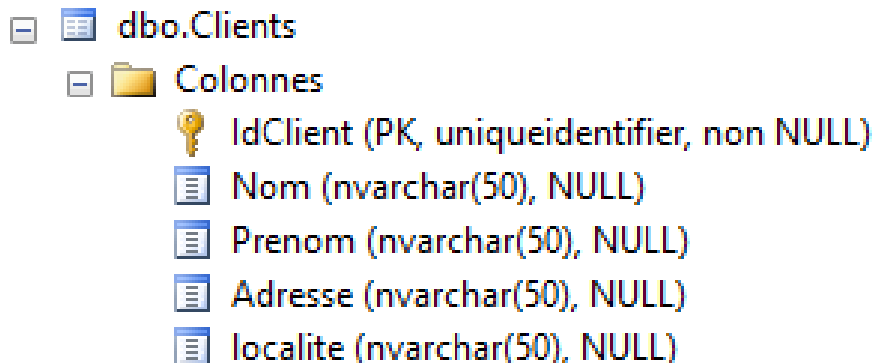
Model First

- We create a UML diagram
- EF generates domain classes and database

Entity Framework

- Database First.

Nous commençons par créer une base de données au travers de SQL server manager. Prenons l'exemple d'une base de données avec une seule table associée à la gestion des clients d'une entreprise.

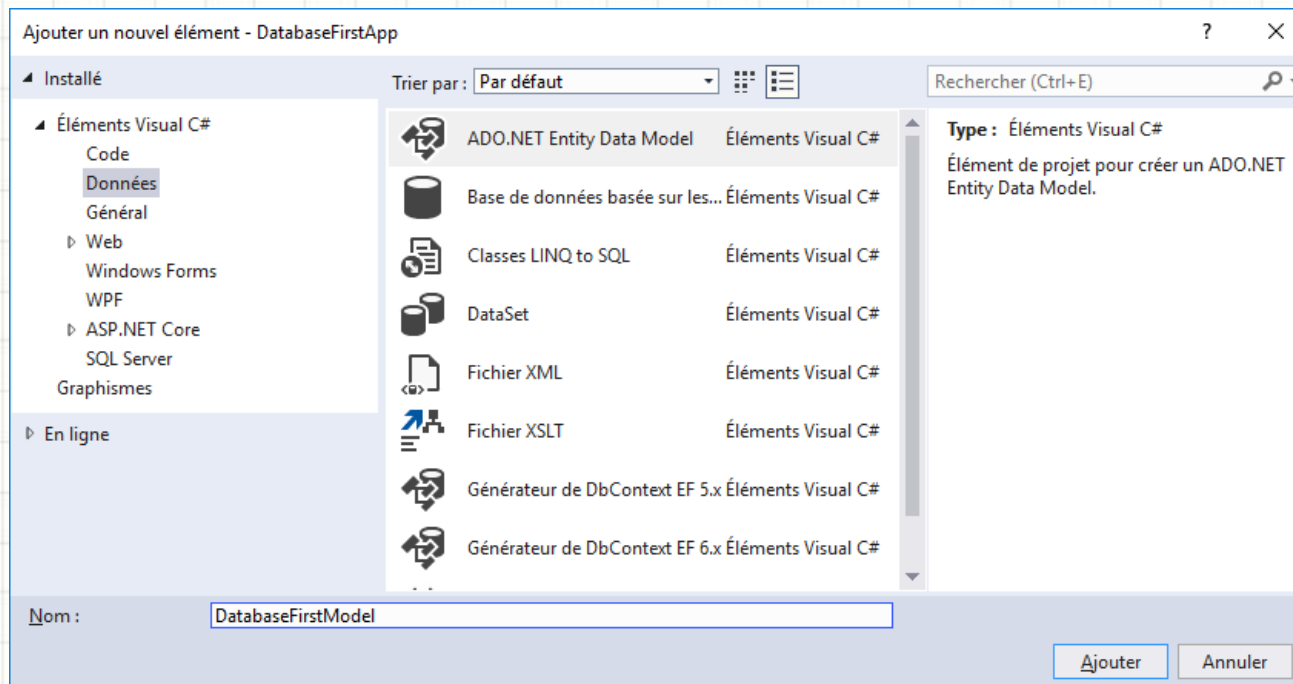


Entity Framework

- Database First.

Il faudra ensuite passer par les différentes étapes suivantes:





1. Au moyen du gestionnaire des packages, installer l'EntityFramework
2. Ajouter à votre application un nouvel élément sous la forme suivante



Entity Framework

- Database First.

Que doit contenir le modèle ?

			
EF Designer à partir de la base de données	Modèle vide EF Designer	Modèle vide Code First	Code First à partir de la base de données

3. Vous choisissez ensuite la base de données avec laquelle vous souhaitez travailler. Vous n'oubliez pas d'inclure la chaîne de connexion dans le fichier de paramètre de votre application.
4. Nous choisissons les objets à inclure dans notre modèle

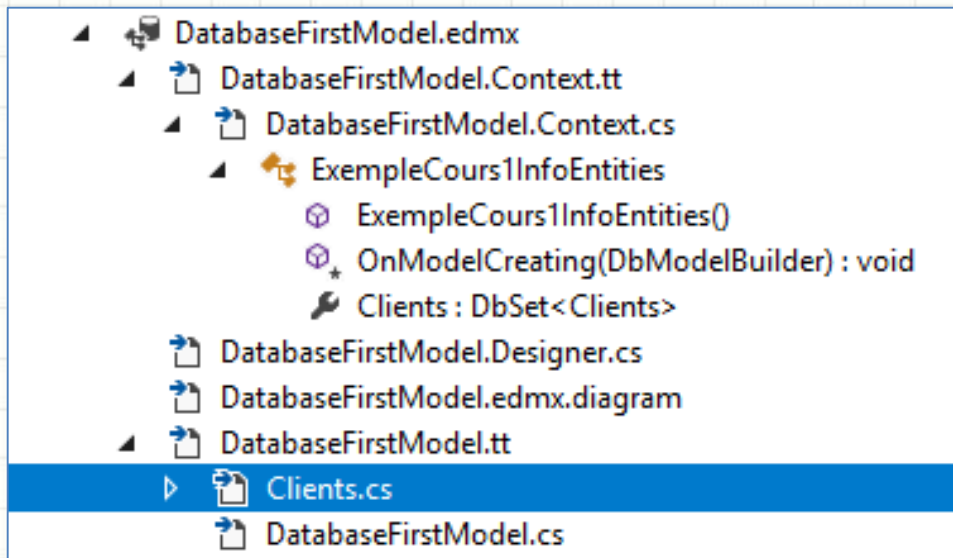
Quels objets de base de données voulez-vous inclure dans votre modèle ?

- ☒ Tables
 - ☒ dbo
 - ☐ Achats
 - ☐ Articles
 - ☒ Clients
- ☐ Vues
- ☐ Procédures et fonctions stockées

Entity Framework

- Database First.
 3. Vous choisissez ensuite la base de données avec laquelle vous souhaitez travailler. Vous n'oubliez pas d'inclure la chaîne de connexion dans le fichier de paramètre de votre application.
 4. Nous choisissons les objets à inclure dans notre modèle

Analysons quelques fichiers intéressants dans notre projet maintenant modifié



Entity Framework

- Database First.

- Nous retrouvons le fichier Client.cs associé à la structure dans enregistrement dans notre table Clients.

```
public partial class Clients
{
    public System.Guid IdClient { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Adresse { get; set; }
    public string localite { get; set; }
}
```

Entity Framework

- Database First.

- Nous retrouvons notre entité héritant de la classe DbContext. Dans cette classe, remarquons la présence de la propriété `DbSet<Clients>` qui correspond à notre table Clients comprenant une collection d'enregistrements.

```
public partial class ExempleCours1InfoEntities : DbContext
{
    //...
    public virtual DbSet<Clients> Clients { get; set; }
}
```

- Il nous reste à manipuler ces objets .NET pour par exemple insérer ou récupérer des données dans la table Clients.

Entity Framework

- Database First.

```
var db = new ExampleCours1InfoEntities();  
  
var MesClients = from cl in db.Clients select cl;  
foreach (var cl in MesClients)  
{  
    Console.WriteLine(cl.Nom + " " + cl.Prenom);  
}  
Console.ReadKey();
```

Entity Framework

- Database First.

```
var db = new ExampleCours1InfoEntities();  
var cl1 = new Clients  
{  
    IdClient = Guid.NewGuid(),  
    Nom="Wilfart",  
    Prenom="Emmanuel",  
    Adresse="Rue Frinoise",  
    localite="Tournai"  
};  
db.Clients.Add(cl1);  
db.SaveChanges();
```

Entity Framework

- Code First.

Comme le nom l'indique, nous allons démarrer notre application sans avoir la moindre table présente dans notre base de données.

Nous nous baserons sur les fichiers décrits dans les diapositives relatives à la technique "Database First"

1. Création d'une classe associée à notre future table dans notre base de données

```
public class Student
{
    [Key]
    public System.Guid IdStudent { get; set; }
    [StringLength(50)]
    public string Nom { get; set; }
    public string Prenom { get; set; }
}
```

Entity Framework

- Code First.

Remarquons les attributs utilisables dans notre classe tel que [[Key](#)]. Ces attributs sont utilisables si vous incluez dans votre code la ligne suivante:

```
using System.ComponentModel.DataAnnotations;
```

Vous pouvez retrouver les détails de ces attributs à l'URL suivante:

<http://www.entityframeworktutorial.net/code-first/dataannotation-in-code-first.aspx>

2. Création d'une classe associée au contexte lié à notre base de données

```
public class StudentContext: DbContext
{
    public DbSet<Student> Students { get; set; }
}
```

Entity Framework

- Code First.

3. Il est nécessaire de définir la chaîne de connexion vers notre base de données dans le fichier app.config

```
<connectionStrings>  
<add name="StudentContext"  
connectionString="Server=127.0.0.1\SQLEXPRESS; initial  
catalog=ExempleCours1Info;persist security info=True;user  
id=1Info;password=password;MultipleActiveResultSets=True;"  
providerName="System.Data.SqlClient" />  
</connectionStrings>
```

Par défaut, le nom de la chaîne de connexion `name="StudentContext"` doit être celui correspondant au nom donné à notre classe de contexte au point précédent

Entity Framework

- Code First.

4. Essayons maintenant d'accéder à notre base de données dont la table est inexistante et éventuellement la base de données inexistante

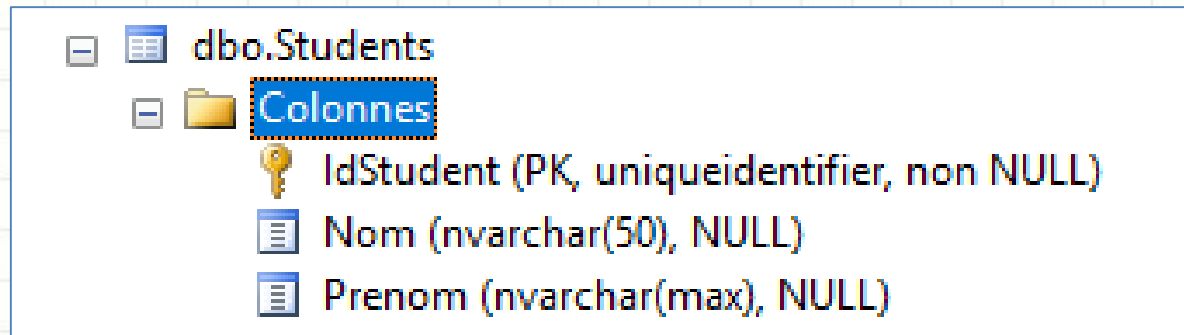
```
var std = new Student
{
    IdStudent = Guid.NewGuid(),
    Nom="Dupond",
    Prenom="Bernard"
};

var db = new StudentContext();
db.Students.Add(std);
db.SaveChanges();
```


Entity Framework

- Code First.

5. Nous ouvrons maintenant notre SQL server manager

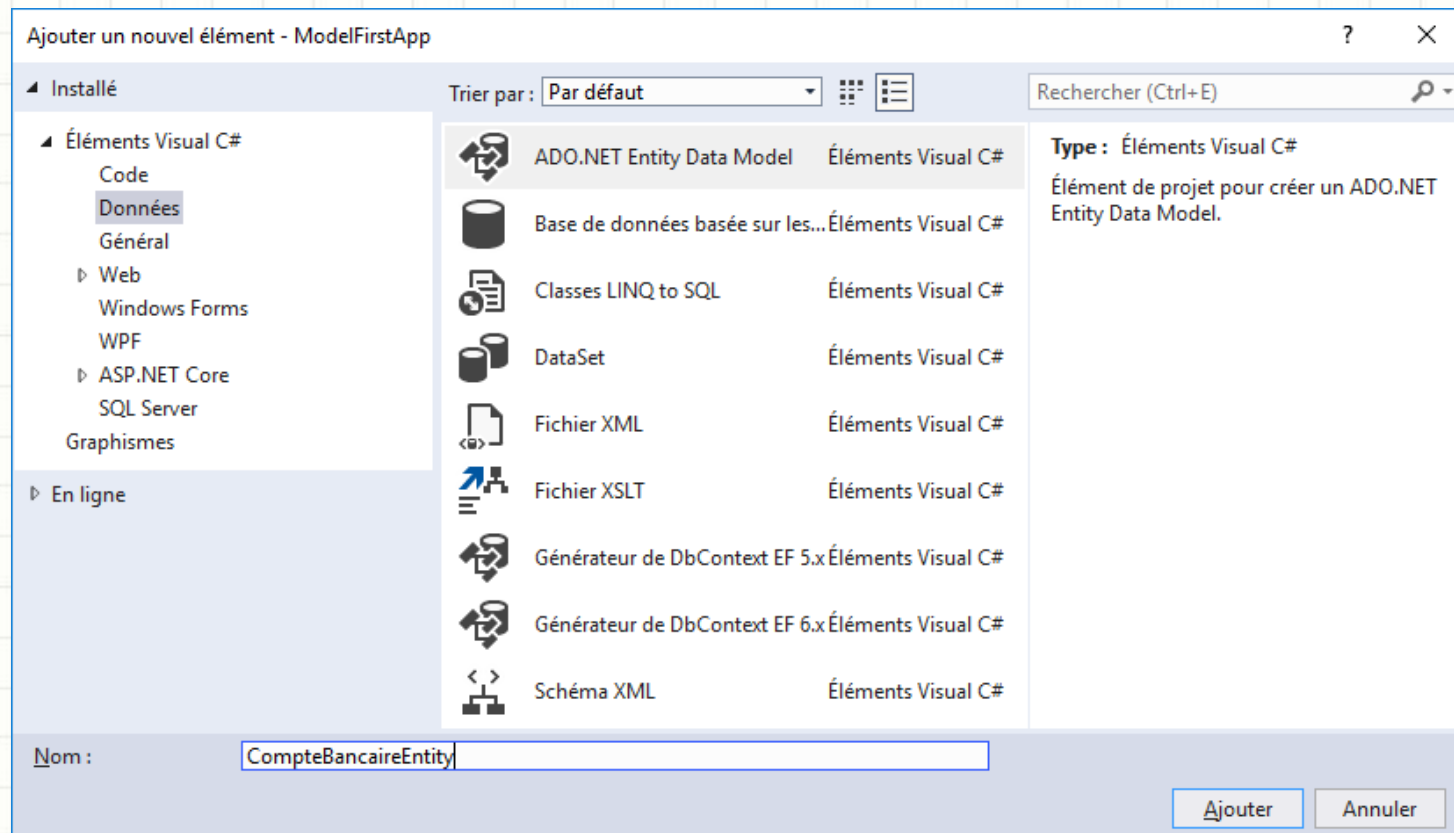


Nous pouvons remarquer que le type du champ "Nom" a été affecté par l'attribut utilisé dans notre classe [`StringLength(50)`]

Entity Framework

- Model First.

Après avoir ajouté le package EntityFramework, nous pouvons choisir d'ajouter un nouvel élément sous la forme suivante:



Entity Framework

- Model First.

Après avoir ajouté le package EntityFramework, nous pouvons choisir d'ajouter un nouvel élément sous la forme suivante:

Que doit contenir le modèle ?



EF Designer à
partir de la base
de données



Modèle vide EF
Designer



Modèle vide
Code First

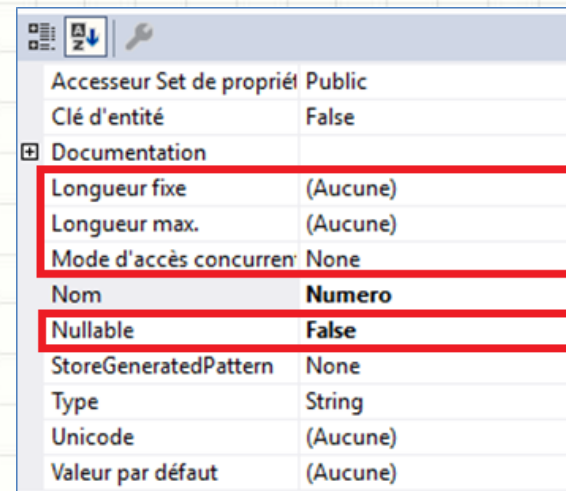
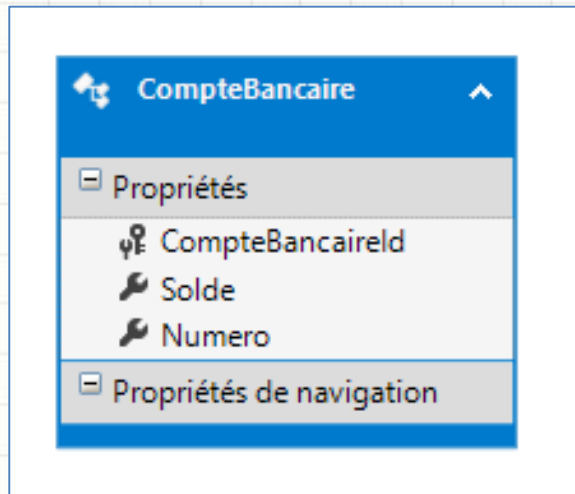


Code First à
partir de la base
de données

Une fois le modèle ajouté, nous pouvons créer notre table en nous aidant de la boîte à outil.

Entity Framework

- Model First.



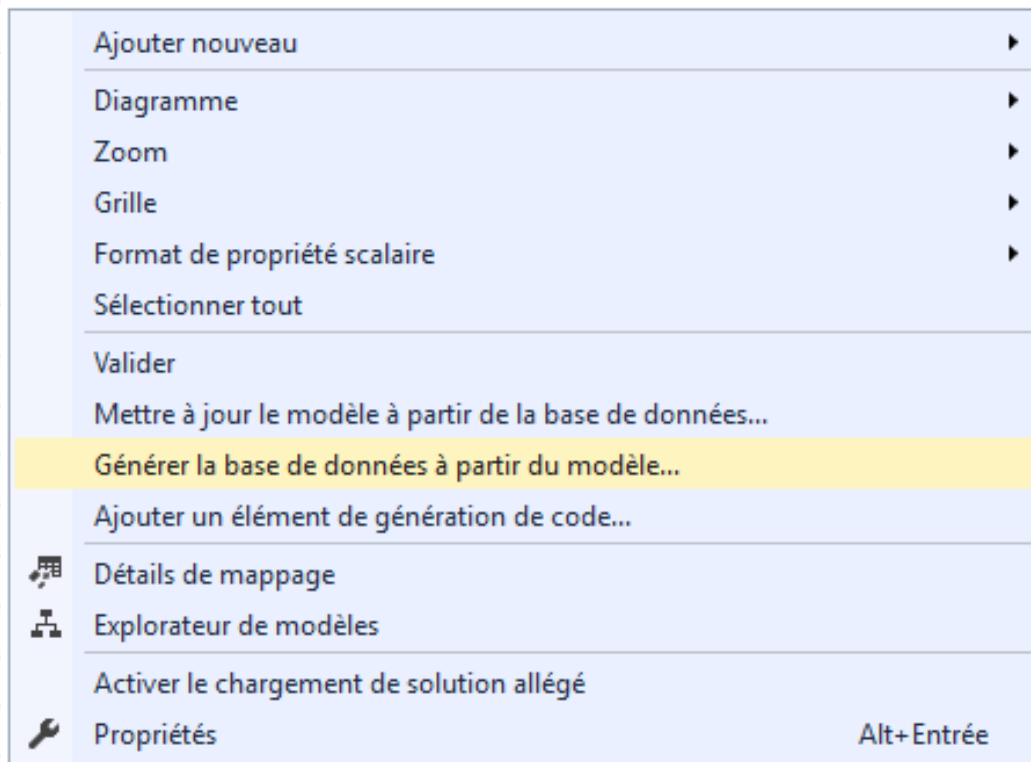
Remarquons les propriétés pour le champ "Numero" et les réglages nous permettant d'interagir avec la création des tables et des champs correspondants.

Remarquons la gestion de l'accès concurrentiel.

Entity Framework

- Model First.

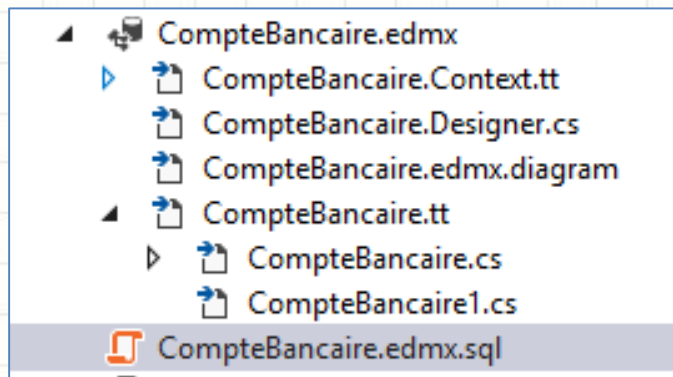
Nous pouvons maintenant accéder au menu contextuel pour créer notre base de données



Entity Framework

- Model First.

Une fois la génération lancée, nous obtenons alors un script permettant de créer notre base de données et les différentes tables associées



Extrait du fichier. Création de la table

```
-- Creating table 'CompteBancaireSet'
CREATE TABLE [dbo].[CompteBancaireSet] (
    [CompteBancaireId] int IDENTITY(1,1) NOT NULL,
    [Solde] float NOT NULL,
    [Numero] nvarchar(max) NOT NULL
);
```


Entity Framework

- Model First.

Nous pouvons maintenant exécuter le script.

