LINQ

Morre Thierry
Cognitic
thierry.morre@cognitic.be

Sommaire

- Introduction.
- Le mot clé « var ».
- Les types anonymes.
- Les expressions « LAMBDA ».
- Les méthodes d’extension.
- LINQ et le Framework.NET 4.0

Introduction

LINQ (**Language Integrated Query**) est une innovation de la version 3.5 du .NET Framework qui permet de rapprocher le monde des objets et le monde des données.

Traditionnellement, lorsque nous souhaitions réaliser des requêtes sur des données, ces dernières étaient exprimées sous forme de chaînes de caractères sans possibilité de vérification à la compilation et sans prise en charge par « l'IntelliSense ».

En outre, nous devions apprendre des langages complémentaires en fonction des sources de données (XPath, SQL, TSQL, PL/SQL, …).
Avec LINQ, toute requête prendra la forme d'une construction de langage de premier ordre (C# ou VB). De plus, nous pourrons écrire ces requêtes en utilisant des mots clés du langage et des opérateurs familiers.

LINQ a été prévu pour travailler avec différentes sources de données :

- Collections d'objets fortement typées (**LINQ To Objects**)
- Fichiers XML (**LINQ To XML**)
- Bases de données SQL Server (**LINQ To SQL**)
- Groupes de données ADO.NET (**LINQ To DataSet**)
- Groupes de données ADO.NET Entities Framework (**LINQ To Entities**)

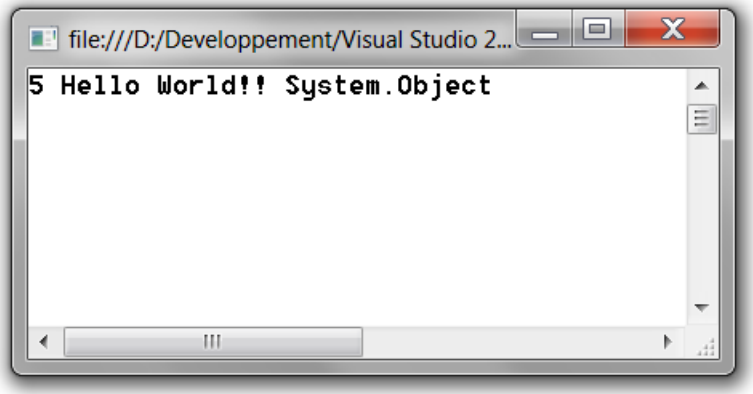
Le mot clé « var »

Ce mot clé est probablement le plus important qu'à apporter LINQ et, bien que nécessaire, le plus controversé aussi.

En effet, C# et VB* étant des langages « fortement typé », chaque variable doit être déclarée avec un type avant d'être instanciée. Or, le mot clé « var », permet de déclarer des variables dont le type sera implicitement donné par le compilateur lors de la compilation.

Si le mot clé « var » est utilisé, la variable doit être instanciée lors de sa déclaration.

```
static void Main(string[] args)
{
    var i = 5;
    var s = "Hello World!!";
    var o = new object();

    Console.WriteLine("{0} {1} {2}", i, s, o);
    Console.ReadLine();
}
```



* En mode « Strict »

Le mot clé « var »

Bien que l'utilisation du mot clé « var » soit permise, il n'en reste que son utilisation abusive risque de nuire à la compréhension du code. Sans oublier que cela implique que nous laisserons le compilateur choisir implicitement, pour nous, le type de la variable.

Ce qui en soit pourra poser des problèmes dans le cadre des valeurs littérales et dans le cadre du polymorphisme pour ne citer qu'eux.

```
static void Main(string[] args)
{
    var i = 5;
    var s = "Hello World!!";
    var o = new object();

    Console.WriteLine("{0} {1} {2}", i, s, o);
    Console.ReadLine();
}
```

i sera de type « System.Int32 »

Par conséquent, lorsque nous connaissons le type à utiliser, nous devrons utiliser ce type plutôt que « var ».

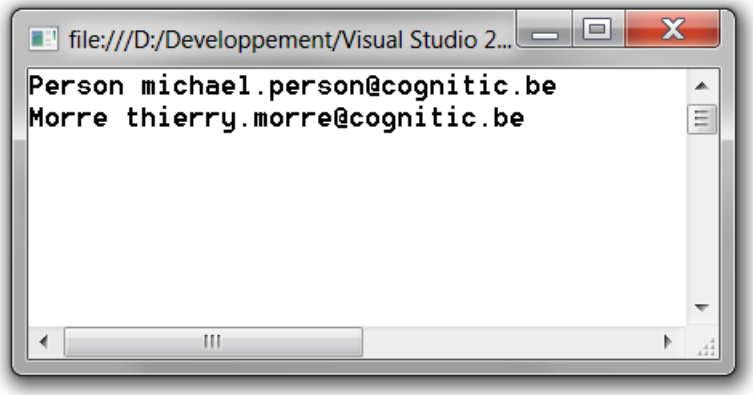
Les types anonymes

Mais, alors pourquoi avoir ajoutée un type « fourre tout » dans un environnement « fortement typé »?

Car LINQ étant un langage puissant, et il se peut que la requête retourne un type qui ne sera connu que lors de la compilation. Ces types sont appelé « Type anonyme ».

```
public class Contact
{
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public string Email { get; set; }
}
```

```
static void Main(string[] args)
{
    List<Contact> Contacts = new List<Contact>();
    Contacts.AddRange(new Contact[] {
        new Contact(){ Nom = "Person", Prenom="Michael", Email="michael.person@cognitic.be"},
        new Contact(){ Nom = "Morre", Prenom="Thierry", Email="thierry.morre@cognitic.be"}
    });

    //On ne prend que le Nom et l'Email du Contact en créant implicitement un nouveau type.
    //Ce "nouveau type" est un type anonyme.
    var InfosDeContactsChoisies = from Contact c in Contacts
                                  select new { Nom = c.Nom, Email = c.Email };


    foreach (var Infos in InfosDeContactsChoisies)
    {
        Console.WriteLine("{0} {1}", Infos.Nom, Infos.Email);
    }

    Console.ReadLine();
}
```

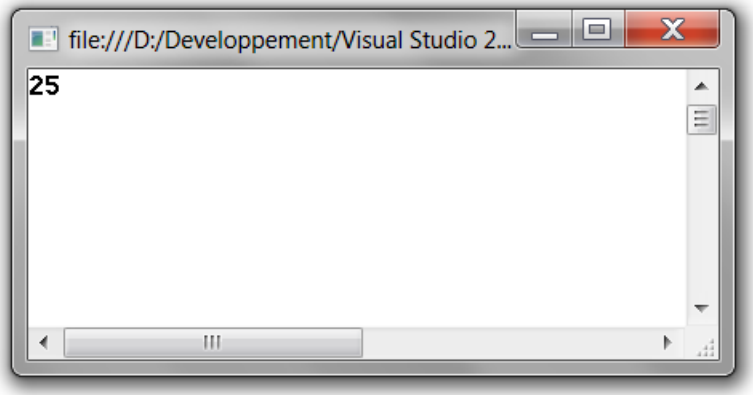


Les expressions « LAMBDA »

Une expression « LAMBDA » est une fonction anonyme qui peut contenir des expressions et des instructions, cette expression pourra être utilisée pour créer des délégués (delegate) ou des type d'arborescence d'expression.

Toutes ces expressions utilise l'opérateur LAMBDA « => » qui se lit « conduit à ».

```
class Program
{
    delegate int del(int i);

    static void Main(string[] args)
    {
        del Mydelegate = x => x * x;
        int j = Mydelegate(5);

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



Dans l'exemple, nous lirons « x conduit à x fois x ».

Le coté gauche de l'expression spécifie les paramètres en entrée (le cas échéant) et le coté droit contient le bloc d'expression ou d'instructions.

Les expressions « LAMBDA »

Une expression LAMBDA avec une expression sur le coté droit est appelée « **lambda-expression** ».

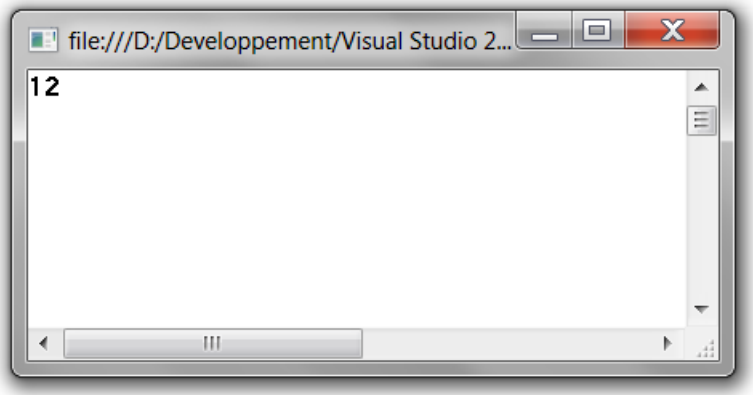
Les « lambda-expression » sont utilisées dans la construction d'arborescences d'expression, elle retourne le résultat de l'expression et prend la forme suivante.

(Paramètres d'entrée) => expression

Les parenthèses sont facultatives uniquement dans le cas où nous n'avons qu'un seul paramètre.

Dans le cas contraire elles sont obligatoires :

```
class Program
{
    delegate long Addition(int x, int y);

    static void Main(string[] args)
    {
        Addition MonAddition = (x, y) => (long)x + y;

        long j = MonAddition(5, 7);

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



Les expressions « LAMBDA »

Si l'expression ne reçoit aucun paramètre, nous le signalerons par des parenthèses vides.

Nous remarquons aussi dans cet exemple que le corps d'une expression « LAMBDA » peut se composer d'un appel de méthode.

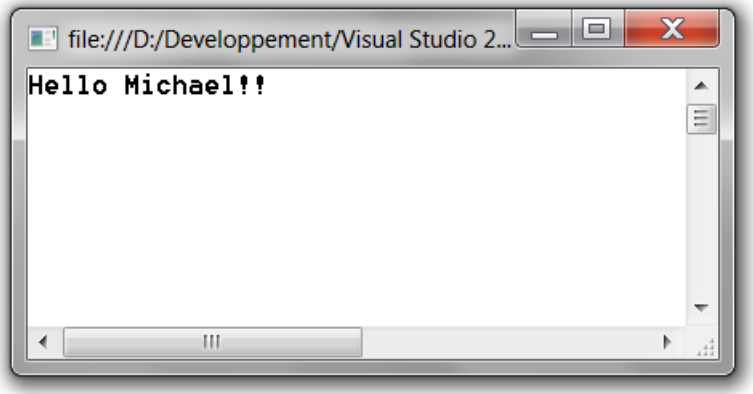
```
class Program
{
    delegate bool del();

    static void Main(string[] args)
    {
        del MyDelegate = () => UneMethode();

        Console.WriteLine(MyDelegate());
        Console.ReadLine();
    }

    static bool UneMethode()
    {
        bool Result = true;
        // ... Traitement
        return Result;
    }
}
```

Les expressions « LAMBDA »

Il existe un autre type d'expression « LAMBDA », celle qui ont à droite un bloc d'instruction.
Elles sont appelées « lambda-instruction ». Une « lambda-instruction » est similaire à la « lambda-expression », sauf que les instruction sont mise entre accolades.

Bien que le corps d'une « lambda-instruction » puisse contenir une infinité d'instructions; dans la pratique ce nombre est généralement de 2 ou 3.

```
class Program
{
    delegate void del(string s);

    static void Main(string[] args)
    {
        del MyDelegate = n => {
            string s = string.Format("Hello {0}!!", n);
            Console.WriteLine(s);
        };

        MyDelegate("Michael");

        Console.ReadLine();
    }
}
```



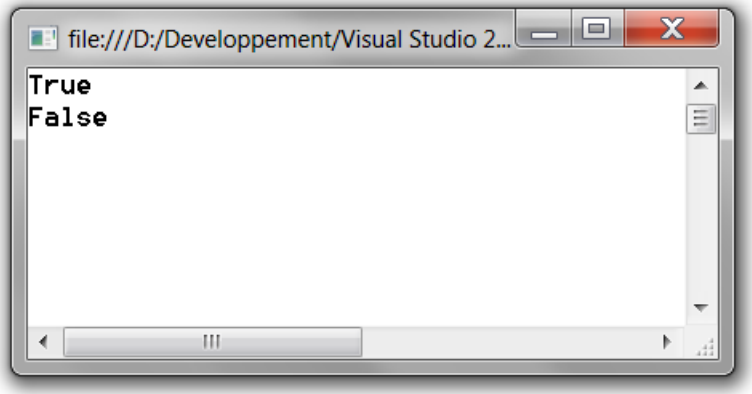
Les expressions « LAMBDA »

Nous verrons plus tard que de nombreux ‘opérateurs de requêtes standard’* comportent un paramètre d’entrée dont le type, « Func<T, TResult> », fait partie de la famille des délégués génériques.

Ces délégués génériques sont très utiles pour encapsuler des expressions définies par l’utilisateur appliquées à chaque élément dans un ensemble de données.

Nous pourrions instancier ce type de délégués comme suit :

```
class Program
{
    static void Main(string[] args)
    {
        Func<int, bool> EstImpaire = x => x % 2 == 1;

        Console.WriteLine(EstImpaire(5));
        Console.WriteLine(EstImpaire(4));

        Console.ReadLine();
    }
}
```



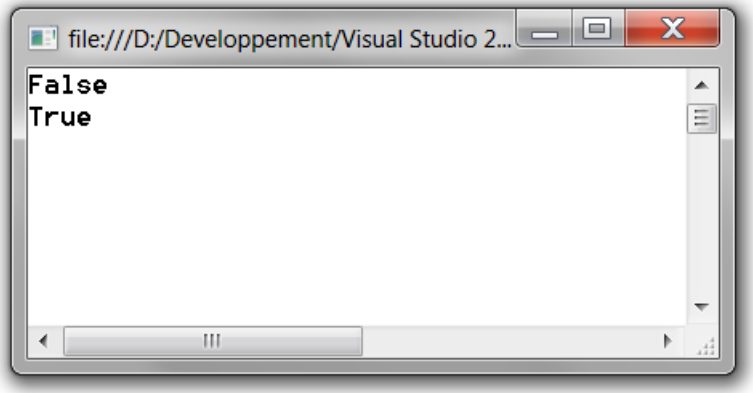
*Nous verrons ces opérateurs de requêtes standard dans la partie « LINQ To Object »

Les expressions « LAMBDA »

Lorsque nous écrivons des expressions « LAMBDA », nous n'aurons généralement pas à spécifier le types des paramètres d'entrées. En effet, le compilateur pourra déduire leur type en fonction du corps du lambda, du type de délégué sous-jacent ainsi que d'autres facteurs décrits dans la spécification du langage C#.

Ce qui signifie que nous aurons accès à leurs méthodes et leurs propriétés.

```
class Program
{
    delegate bool del(string s1, string s2);

    static void Main(string[] args)
    {
        del Contains = (s1, s2) => s1.ToUpper().Contains(s2.ToUpper());

        string content = "lu";

        Console.WriteLine(Contains("Hello", content));
        Console.WriteLine(Contains("Aluminium", content));

        Console.ReadLine();
    }
}
```



Les méthodes d'extension

Lorsque LINQ est arrivé, il a apporté bon nombre de nouvelles fonctionnalités aux objets du Framework.NET. Les plus communes sont les « opérateurs de requêtes standard LINQ » qui ajoute des fonctionnalités de requête au types « IEnumerable » et « IEnumerable<> ».

Ces types d'objets ont donc vu leur nombre de fonctionnalités augmenté mais « Microsoft » n'a pas modifié leur type d'origine. Ils ont utilisé le principe de méthodes d'extension.

Ces dernières vont nous permettent d'ajouter des méthodes à des types existants sans créer un type dérivé ou sans devoir modifier et recompiler le type d'origine.

Nous allons les définir comme méthodes statiques mais nous appellerons en utilisant la syntaxe de méthode d'instance.

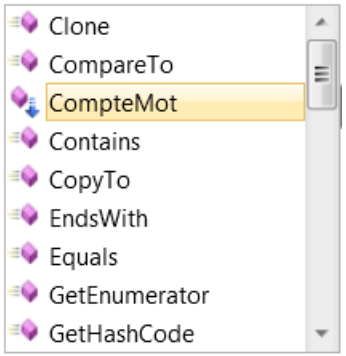
Leur premier paramètre spécifie les types* sur lesquels la méthode fonctionne et ce paramètre sera précédé par le modificateur « this ».

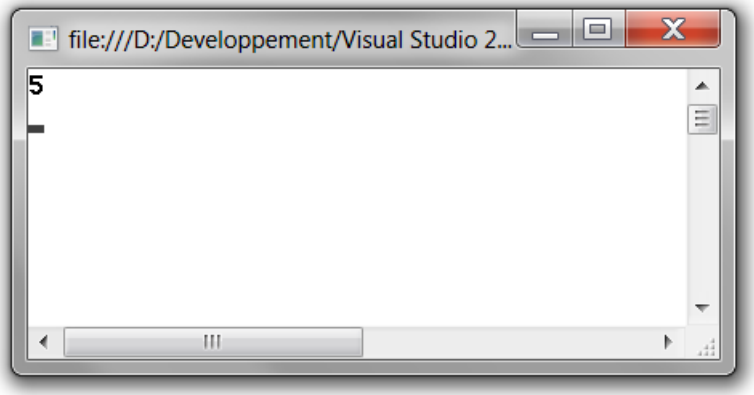
*Les types héritant du type sur lequel nous avons ajouter une méthode d'extension, hériteront également de la méthode d'extension.

Les méthodes d'extension

Ajoutons la méthode « CompteMot » au type « string ».

```
public static class MesExtensions
{
    public static int CompteMot(this string s)
    {
        return s.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
    }
}

class Program
{
    delegate bool del(string s1, string s2);

    static void Main(string[] args)
    {
        string s = "Il fait très beau aujourd'hui";

        Console.WriteLine(s.CompteMot());
        Console.ReadLine();
    }
}
```





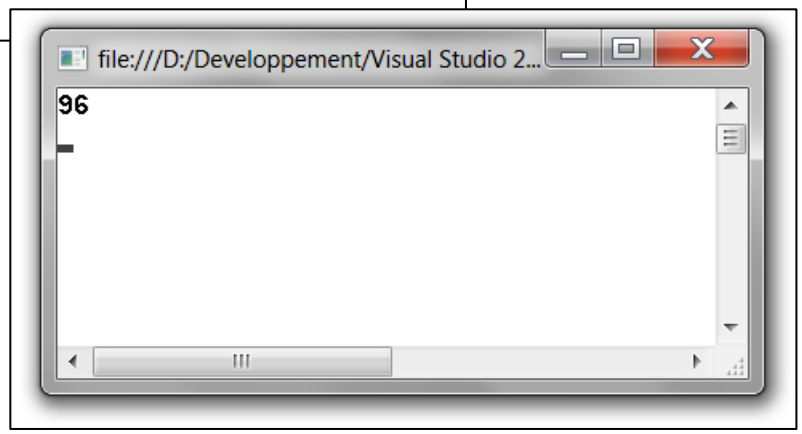
Les méthodes d'extension

Si un ou plusieurs paramètres sont nécessaires nous devons simplement les mettre à la suite du premier paramètre.

```
public static long Addition(this int i, params int[] ints)
{
    long result = i;

    foreach (int n in ints)
    {
        result += n;
    }

    return result;
}
```

```
class Program
{
    delegate bool del(string s1, string s2);

    static void Main(string[] args)
    {
        int x = 7;

        Console.WriteLine(x.Addition(8,5,63,9,4));
        Console.ReadLine();
    }
}
```

(extension) long int.Addition(**params int[] ints**)



LINQ et le Framework.NET 4.0

Durant ce cours, nous allons étudier LINQ du Framework.NET 3.5.

Le Framework.NET 4.0, quant à lui, a implémenté une nouvelle couche à LINQ. Cette nouvelle couche appelée **PLINQ,** pour « **Parallel LINQ** », est un complément à « **LINQ To Objects** » qui implémente une jeu complet de méthodes afin de combiner la simplicité et la lisibilité de la syntaxe LINQ et la puissance de la programmation parallèle.

Dans de nombreux scénarios, **PLINQ** peut ainsi augmenter considérablement la vitesse des requêtes « **LINQ To Object**s », sur de gros volumes de données, en utilisant plus efficacement tous les cœurs disponibles sur l'ordinateur hôte.

Cette performance accrue apporte une puissance de calcul haute performance sur le Bureau.

Cependant, l'utilisation de **PLINQ** pour de petits volumes de données est déconseillée en raison des ressources mises en place pour la gestion parallèle qui, au final, risquerait d'alourdir votre application.