

LINQ

Cognitic Sprl



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 (**L**anguage **I**ntegrated **Q**uery) 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 souhaitons 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**)

COGNITIC

Le mot clé « var »

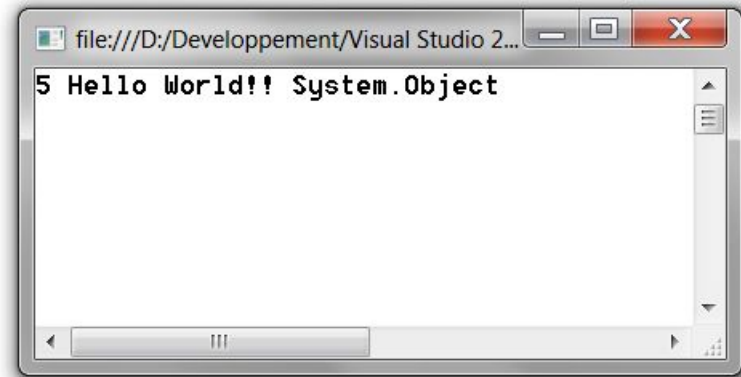
Ce mot clé est probablement le plus important qu'a apporté LINQ et, bien que nécessaire, le plus controversé aussi.

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 »

COGNITIC

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 devons 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 ».

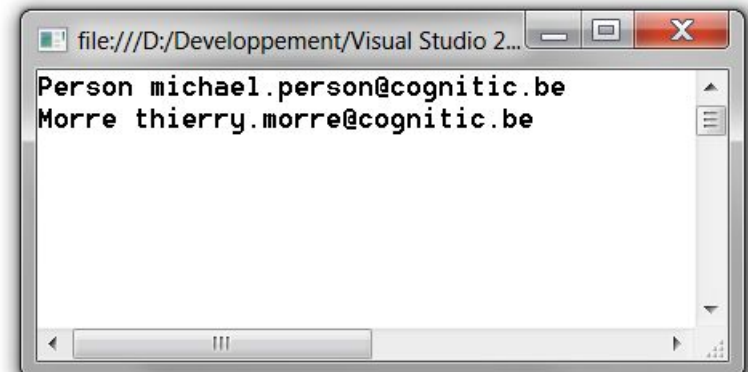
```
static void Main(string[] args)
{
    List<Contact> Contacts = new List<Contact>();
    Contacts.AddRange(new Contact[] {
        new Contact(){ Nom = "Person", Prenom="Michael", Email="michael.person@cognitic.be"},
        new Contact(){ Nom = "Morre", Prenom="Thierry", Email="thierry.morre@cognitic.be"}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

    //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();
}
```

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



Les expressions « LAMBDA »

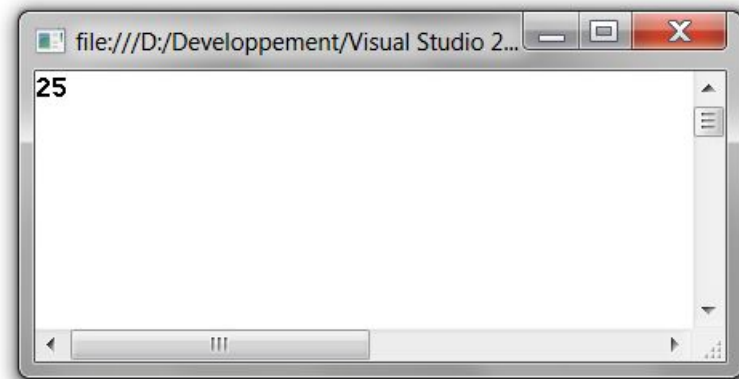
Une expression « LAMBDA » est une fonction anonyme qui peut contenir des expressions et des instructions, cette expression pourra être utilisée pour créer des délégués (delegate) ou des type d'arborescence d'expression.

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 lisons « 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 côté droit est appelée « **lambda-expression** ».

Les « lambda-expression » sont utilisées dans la construction d'arbres 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.

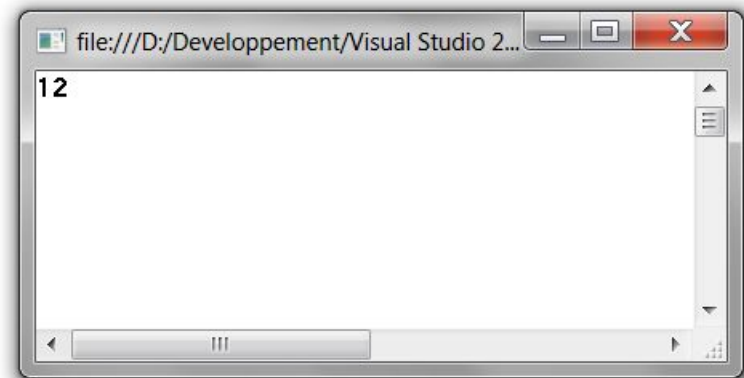
Dans le cas contraire elles sont obligatoires :

```
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 instructions sont mises entre accolades.

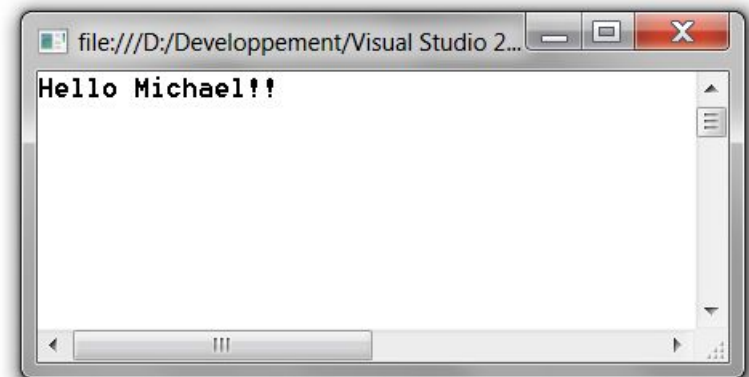
Bien que le corps d'une « lambda-instruction » puisse contenir une infinité d'instructions; dans la pratique ce nombre est généralement de 2 ou 3.

```
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);
        };

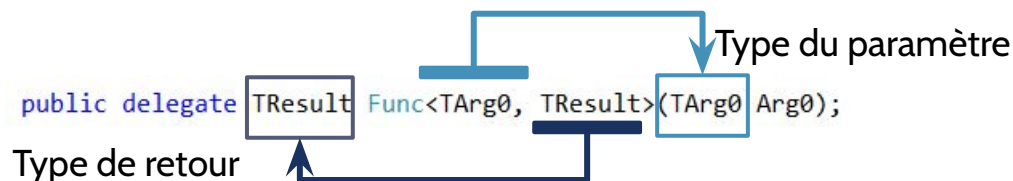
        MyDelegate("Michael");

        Console.ReadLine();
    }
}
```



Les expressions « LAMBDA »

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



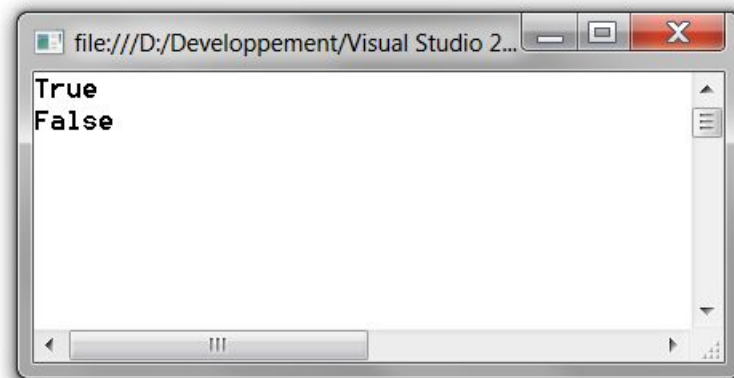
Ces délégués génériques sont très utiles pour encapsuler des expressions définies par l'utilisateur appliquées à chaque élément dans un ensemble de données.

Nous pourrions instancier ce type de délégué 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.

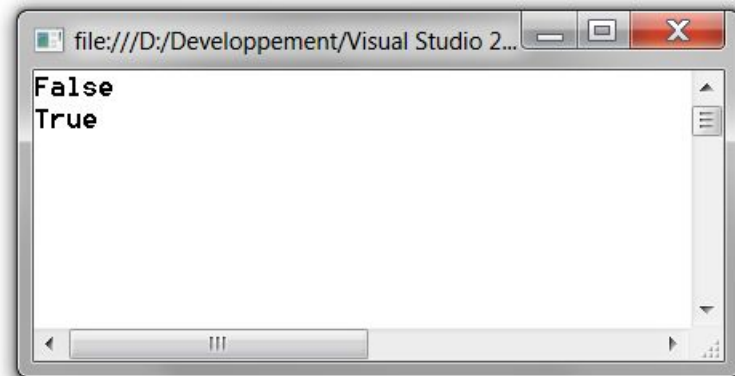
```
class Program
{
    delegate bool del(string s1, string s2);

    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 ajoutent des fonctionnalités de requête au types « IEnumerable » et « IEnumerable<T> ».

Ces types d'objets ont donc vu leur nombre de fonctionnalités augmenter 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 ajouté une méthode d'extension, hériteront également de la méthode d'extension.

Les méthodes d'extension

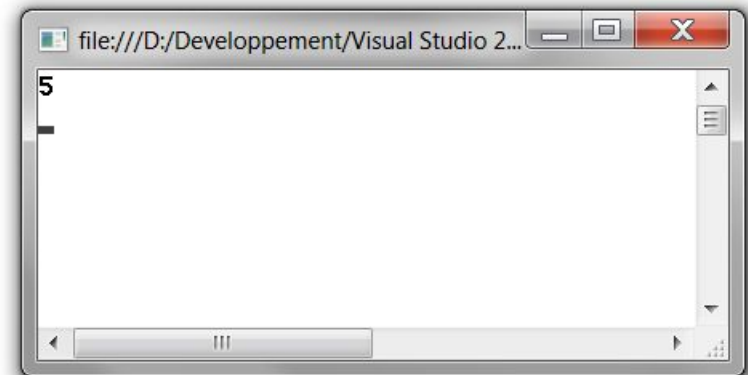
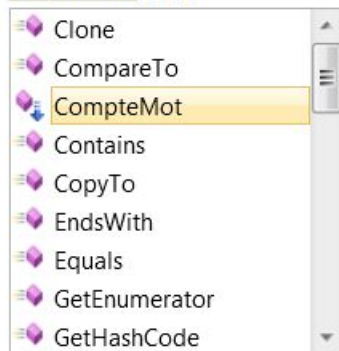
Ajoutons la méthode « CompteMot » au type « string ».

```
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

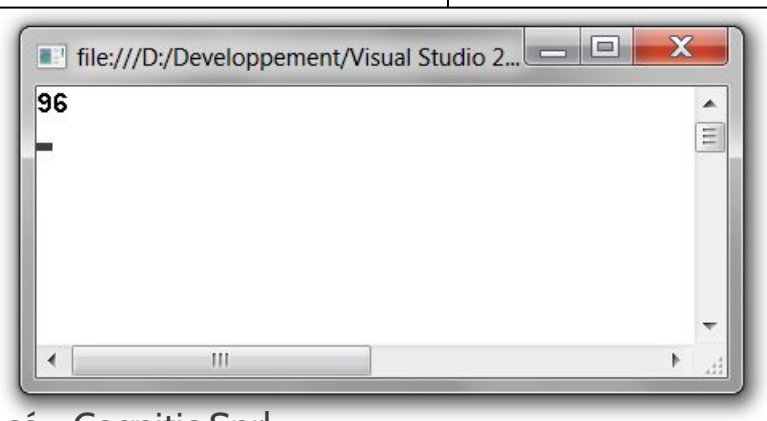
Si un ou plusieurs paramètres sont nécessaires nous devons simplement les mettre à la suite du premier paramètre.

```
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();
    }
}
```



LINQ To Objects

Cognitic Sprl



Sommaire

- ▶ Introduction
- ▶ LINQ => Query ou pas !
- ▶ IEnumerable<T> et séquence
- ▶ Opérateurs de requête standard
- ▶ Modes d'exécutions
- ▶ Support de l'expression de requête
- ▶ Les opérateurs courants
- ▶ Utilisation de clés composites
- ▶ Multiples clauses « from »

Introduction

LINQ To Objects fait référence à l'utilisation directe de requêtes LINQ avec n'importe quelle collection « IEnumerable » et « IEnumerable<T> » telles que « List<T> », « Array » ou « Dictionary<Tkey, Tvalue> » définies par l'utilisateur ou retournées par une API du Framework .NET.

Auparavant, nous devions écrire des boucles complexes pour spécifier comment récupérer des données d'une collection. Maintenant grâce à LINQ, nous pourrons écrire du code déclaratif qui décrira exactement ce que nous voudrions.

Ces requêtes offrent trois principaux avantages par rapport aux boucles :

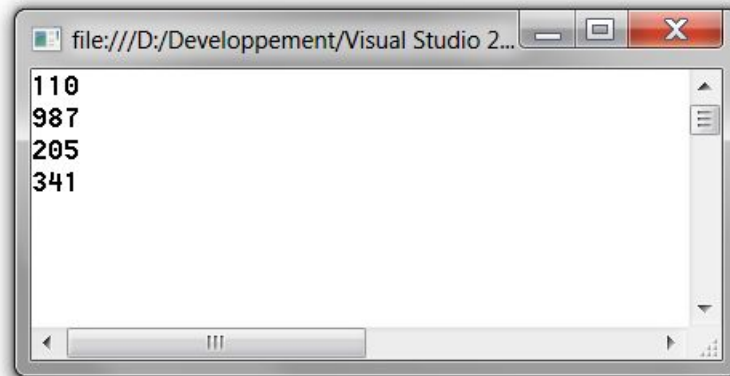
- Elles sont plus concises et lisibles
- Elles fournissent des fonctions puissantes de filtrage, de classement et de regroupement avec un minimum de code.
- Elles peuvent être appliquées à d'autres sources de données avec peu ou pas de changement.

```
IEnumerable<Contact> QueryResult = from Contact c in Contacts
                                   where c.Email.Substring(0, c.Email.IndexOf('@')).Length == (c.Nom.Length + c.Prenom.Length + 1)
                                   orderby c.Prenom, c.Nom
                                   select c;
```

LINQ => Query ou pas!

LINQ étant l'abréviation de « **Language Integrated Query** », nous pourrions penser qu'il se limite à l'interrogation de données. Mais en réalité, son utilisation peut aller bien au-delà.

```
string[] ValuesAsStrings = { "110", "987", "205", "341" };  
int[] ints = ValuesAsStrings.Select(s => Int32.Parse(s)).ToArray();  
  
foreach (int i in ints)  
{  
    Console.WriteLine(i);  
}  
  
Console.ReadLine();
```



La conversion de ce tableau de « string » en « int » pourrait être plus simple?

IEnumerable<T> et séquence

Cette interface a été implémentée par les tableaux et les classes de collections génériques de C# 2.0. Elle permet, de par son fonctionnement, d'énumérer les éléments d'une collection.

Une séquence est un terme logique d'une collection implémentant l'interface « IEnumerable<T> ». En d'autres termes, si nous avons une variable de type « IEnumerable<string> », nous pourrions dire que nous avons une séquence de « string ».

Nous allons voir, par la suite, que la plupart des « Opérateurs de requête standard » sont prototypés de façon à retourner une séquence (IEnumerable<T>).

Contacts.Select(

▲ 1 of 2 ▼ (extension) IEnumerable<TResult> IEnumerable<Contact>.Select(**Func<Contact, TResult> selector**)
Projects each element of a sequence into a new form.
selector: A transform function to apply to each element.

Opérateurs de requête standard

Les « opérateurs de requête standard » sont les méthodes qui composent le modèle LINQ.

La plupart de ses méthodes fonctionnent sur des séquences et fournissent des fonctions de requête, notamment le filtrage (Where), la projection (Select), l'aggrégation (Sum), le tri (OrderBy), etc..

Elles sont définies, dans la classe « Enumerable », comme méthodes d'extension à l'interface « IEnumerable », cela implique qu'elle peuvent être appelées sur n'importe quelle collection générique comme méthode d'instance.

Ces opérateurs sont regroupés par fonctionnalité :

- Aggrégation
- Concaténation
- Conversion
- Egalité
- Élément
- Ensemble
- Filtrage
- Génération
- Jointure
- Partitionnement
- Projection
- Quantificateur
- Regroupement
- Tri

Modes d'exécutions

Les opérateurs de requêtes standard se divisent en 2 modes d'exécution.

Immédiat :

L'exécution immédiate signifie que la source de données est lue et que l'opération est effectuée au point où la requête est déclarée dans le code. Ce mode d'exécution vise tous les opérateurs de requête standard qui retournent un résultat unique et non énumérable comme les opérateurs d'agrégation, d'élément, etc.

Différé :

L'exécution différée signifie que l'opération n'est pas effectuée au point où la requête est déclarée dans le code mais qu'elle le sera uniquement que lorsque la variable de requête est énumérée (boucle foreach par exemple).

Cela signifie que les résultats de l'exécution de la requête dépendent du contenu de la source de données lorsque la requête est exécutée plutôt que lorsqu'elle est définie. Si la variable de requête est énumérée plusieurs fois, les résultats peuvent s'avérer différents chaque fois. Presque tous les opérateurs de requête standard dont le type de retour est `IEnumerable<T>` s'exécutent de manière différée.

Support de l'expression de requête

Certains des opérateurs de requêtes standard, les plus courants, possèdent une syntaxe de mots clé du langage C# qui leur permet d'être appelés dans le cadre d'une expression de requête.

Par défaut, pour utiliser LINQ, nous devrions appeler ces opérateurs qui sont des méthodes. L'expression de requête nous offre une forme plus lisible que son équivalent fondé sur des méthodes.

Les clauses de requêtes sont traduites en appels de méthodes lors de la compilations.

```
IEnumerable<Contact> QueryResult = from Contact c in Contacts
                                   where c.Email.Substring(0, c.Email.IndexOf('@')).Length == (c.Nom.Length + c.Prenom.Length + 1)
                                   orderby c.Prenom, c.Nom
                                   select c;
```



```
IEnumerable<Contact> QueryResult = Contacts
    .Where(c => c.Email.Substring(0, c.Email.IndexOf('@')).Length == (c.Nom.Length + c.Prenom.Length + 1))
    .OrderBy(c => c.Prenom)
    .ThenBy(c => c.Nom);
```

Une expression de requête doit commencer par une clause « from » et doit se terminer par une clause « select » ou « group ». Entre la première clause from et la dernière clause select ou group, elle peut contenir une ou plusieurs clauses facultatives : where, orderby, join, left et même d'autres clauses from supplémentaires. Nous pourrions également utiliser le mot clé « into » pour que le résultat d'une clause « join » ou « group » puisse servir de source pour des clauses de requête supplémentaires dans la même expression de requête.

Support de l'expression de requête

Liste des Opérateurs supportant l'expression de requête en C#

Opérateur	Syntaxe d'expression de requête
Cast	'from int n in numbers' explicitement typé
GroupBy	'group' ... by ou 'group ... by ... into ...'
GroupJoin	'join ... in ... on ... equals ... into ...'
Join	'join ... in ... on ... equals ...'
OrderBy	'order by ...'
OrderByDescending	'orderby ... descending'
Select	'select'
SelectMany	Plusieurs clauses 'from'
ThenBy	'orderby ..., ...'
ThenByDescending	'orderby ..., ... descending'
Where	'where'

Les opérateurs courants

Nous allons à présent faire le tour des opérateurs les plus souvent utilisés en LINQ. Les exemples qui vont suivre seront exécutés sur une liste générique de « Contact ».

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

List<Contact> Contacts = new List<Contact>();
Contacts.AddRange(new Contact[] {
    new Contact(){ Nom = "Person", Prenom="Michael", Email="michael.person@cognitic.be", AnneeDeNaissance = 1982 },
    new Contact(){ Nom = "Morre", Prenom="Thierry", Email="thierry.morre@cognitic.be", AnneeDeNaissance = 1974 },
    new Contact(){ Nom = "Dupuis", Prenom="Thierry", Email="thierry.dupuis@cognitic.be", AnneeDeNaissance = 1988 },
    new Contact(){ Nom = "Faulkner", Prenom="Stéphane", Email="stephane.faulkner@cognitic.be", AnneeDeNaissance = 1969 },
    new Contact(){ Nom = "Selleck", Prenom = "Tom", Email = "tom.selleck@imdb.com", AnneeDeNaissance = 1945 },
    new Contact(){ Nom = "Anderson", Prenom = "Richard Dean", Email = "richard.dean.anderson@imdb.com", AnneeDeNaissance = 1950 },
    new Contact(){ Nom = "Bullock", Prenom = "Sandra", Email = "sandra.bullock@imdb.com", AnneeDeNaissance = 1964 },
    new Contact(){ Nom = "Peppard", Prenom = "George", Email = "peppard.george@ateam.com", AnneeDeNaissance = 1928 },
    new Contact(){ Nom = "Estevez", Prenom = "Emilio", Email = "emilio.estevez@breakfirstclub.com", AnneeDeNaissance = 1962 },
    new Contact(){ Nom = "Moore", Prenom = "Demi", Email = "demi.moore@imdb.com", AnneeDeNaissance = 1962 },
    new Contact(){ Nom = "Willis", Prenom = "Bruce", Email = "bruce.willis@diehard.com", AnneeDeNaissance = 1955 },
});
```

Les opérateurs courants

Voici les opérateurs que nous allons voir :

- ▶ Opérateurs « Cast<T> » & « OfType<T> »
- ▶ Opérateur « Where »
- ▶ Opérateur « Select »
- ▶ Opérateur « Distinct »
- ▶ Opérateur « SingleOrDefault »
- ▶ Opérateur « FirstOrDefault »
- ▶ Opérateurs « OrderBy[Descending] »
- ▶ Opérateurs « ThenBy[Descending] »
- ▶ Opérateur « Count » & « LongCount »
- ▶ Opérateurs « Min » & « Max »
- ▶ Opérateurs « Sum » & « Average »
- ▶ Opérateur « GroupBy »
- ▶ Opérateur « Join »
- ▶ Opérateur « GroupJoin »

Dans certains cas, les opérateurs ont des surcharges. Pour ne pas rallonger inutilement ce module, je me contenterai des prototypes de base.

Vous trouverez l'ensemble des surcharges sur le site « MSDN » ou dans la classe « Enumerable ».

J'ajouterai la requête sous forme d'« expression de requête », lorsque l'opérateur supportera cette syntaxe.

Opérateurs « Cast<T> » & « OfType<T> »

La grande majorité des opérateurs de requête LINQ ne peut être utilisée que sur des collections qui implémentent l'interface « IEnumerable<T> ». Or, aucune des collections présente dans l'espace de nom « System.Collections », telles que Array, ArrayList ou Hashtable, n'implémentent cette interface.

Mais alors comment utiliser LINQ avec ces collections?

En regardant de plus près la classe « Enumerable » qui implémente les méthodes d'extension de LINQ. Nous retrouvons 2 méthodes qui ne sont pas des méthodes d'extension sur l'interface « IEnumerable<T> » mais sur « IEnumerable ».

Ces 2 opérateurs sont là pour transformer une collection « IEnumerable » en une séquence « IEnumerable<T> »



Opérateurs « Cast<T> » & « OfType<T> »

Cependant, ces opérateurs sont quelque peu différents l'un de l'autre.

Cast<T> : `public static IEnumerable<TResult> Cast<TResult>(this IEnumerable source);`

Cet opérateur va tenter de convertir tous les éléments de la collection en une séquence de type 'T'. Si celle-ci n'y parvient pas, elle lèvera une exception.

```
//Transforme List<Contact> (IEnumerable<T>) en ArrayList (IEnumerable)
ArrayList MyArrayList = new ArrayList(Contacts.ToArray());

IEnumerable<Contact> Result = MyArrayList.Cast<Contact>();
```

Cet opérateur supporte la syntaxe d'expression de requête. En spécifiant explicitement le type à utiliser.

```
//Transforme List<Contact> (IEnumerable<T>) en ArrayList (IEnumerable)
ArrayList MyArrayList = new ArrayList(Contacts.ToArray());

IEnumerable<Contact> Result = from Contact c in MyArrayList
                              select c;
```

Opérateurs « Cast<T> » & « OfType<T> »

Celles-ci sont quelque peu différentes l'une de l'autre.

OfType<T>: `public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source);`

Cet opérateur, quant à lui, va filtrer les valeurs en fonction de leur capacité à être castées dans le type spécifié.

```
//Transforme List<Contact> (IEnumerable<T>) en ArrayList (IEnumerable)
ArrayList MyArrayList = new ArrayList(Contacts.ToArray());

IEnumerable<Contact> Result = MyArrayList.OfType<Contact>();
```

Cet opérateur n'étant pas supporté par l'expression de requête, voici comment spécifier que l'on utilise « OfType<T> » plutôt que « Cast<T> ».

```
//Transforme List<Contact> (IEnumerable<T>) en ArrayList (IEnumerable)
ArrayList MyArrayList = new ArrayList(Contacts.ToArray());

//Attention 2 requêtes LINQ imbriquées
IEnumerable<Contact> Result = from c in MyArrayList.OfType<Contact>()
                             select c;
```

Opérateurs « Cast<T> » & « OfType<T> »

Préférons « OfType<T> » à « Cast<T> » :

Bien que ces deux opérateurs soient utilisables sur une collection héritées (System.Collection), « Cast<T> » nécessite que tous les éléments aient le type attendu.

Pour éviter de générer des exceptions en cas d'incompatibilité de types, préférons-lui l'opérateur « OfType<T> ». Par son intermédiaire, seuls les objets pouvant être castés dans le type attendu seront stockés dans la séquence.

Les autres seront purement et simplement ignorés.

Opérateur « Where »

L'opérateur « Where » est utilisé pour filtrer une séquence.

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

Cet opérateur utilise le « delegate » générique « Func<T, bool> » vu dans le chapitre précédent.

```
IEnumerable<Contact> QueryResult = Contacts.Where(c => c.AnneeDeNaissance >= 1950);

foreach (Contact c in QueryResult)
{
    Console.WriteLine("{0} {1} : {2}", c.Nom, c.Prenom, c.AnneeDeNaissance);
}
```

Expression de requête :

```
IEnumerable<Contact> QueryResult = from c in Contacts
                                   where c.AnneeDeNaissance >= 1950
                                   select c;

foreach (Contact c in QueryResult)
{
    Console.WriteLine("{0} {1} : {2}", c.Nom, c.Prenom, c.AnneeDeNaissance);
}
```



Opérateur « Select »

L'opérateur « Select » est utilisé pour retourner une séquence d'éléments sélectionnés dans la séquence d'entrée ou à partir d'une portions de la séquence d'entrée.

```
public static IEnumerable<TR> Select<T, TR>(this IEnumerable<T> source, Func<T, TR> selector);
```

Dans les requêtes LINQ basées sur les appels de méthodes, l'opérateur « Select » est facultatif et sert principalement à sélectionner une portion, en termes de Propriétés, des objets en entrée et de créer de nouveaux types d'objets (types anonymes).

Dans les expressions de requête, que l'on crée un type anonyme ou non, celui-ci est obligatoire.

```
//Sélectionne tous les contacts et retourne une séquence de type "Contact"  
IEnumerable<Contact> QueryResult = from c in Contacts  
    select c;
```


Opérateur « Select »

Création d'un type anonyme :

Pour créer un type anonyme, nous devons joindre à notre select le mot clé « new ».

```
var QueryResult = Contacts.Select(c => new { Nom = c.Nom, Courriel = c.Email });

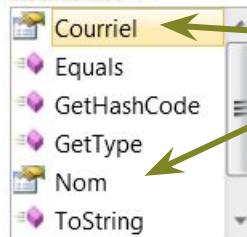
foreach (var c in QueryResult)
{
    Console.WriteLine("{0} : {1}", c.Nom, c.Courriel);
}
```

Expression de requête

```
var QueryResult = from c in Contacts
                  select new { Nom = c.Nom, Courriel = c.Email };

foreach (var c in QueryResult)
{
    Console.WriteLine("{0} : {1}", c.Nom, c.Courriel);
}

Console.ReadLine();
```



Création à la volée de propriété

Opérateur « Distinct »

L'opérateur « Distinct » permet de supprimer les doublons dans une séquence.

```
public static IEnumerable<T> Distinct<T>(this IEnumerable<T> source);
```

```
var QueryResult = Contacts.Select(c => new { Prenom = c.Prenom }).Distinct();

foreach (var c in QueryResult)
{
    Console.WriteLine("{0}", c.Prenom);
}
```

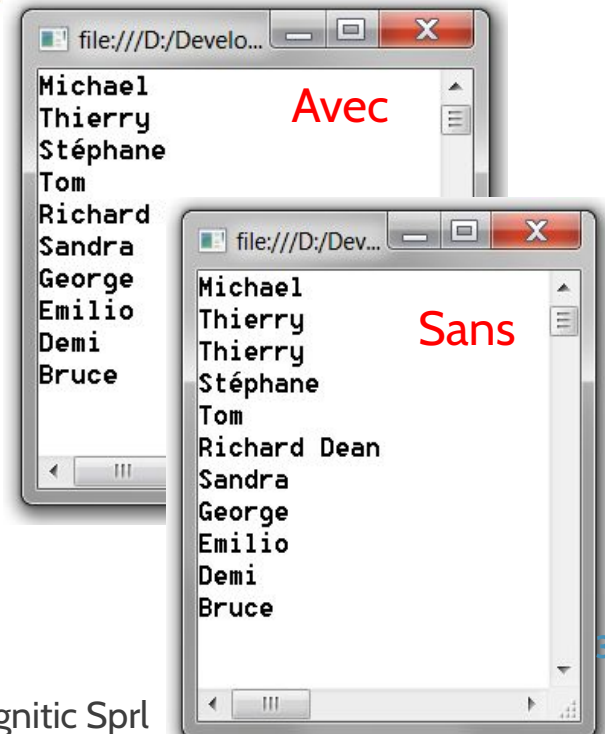
Utilisation avec une expression de requête :

```
var QueryResult = (from c in Contacts
                    select new { Prenom = c.Prenom }).Distinct();

foreach (var c in QueryResult)
{
    Console.WriteLine("{0}", c.Prenom);
}
```

Le distinct se fait sur la valeur de l'ensemble des propriétés.

COGNITIC



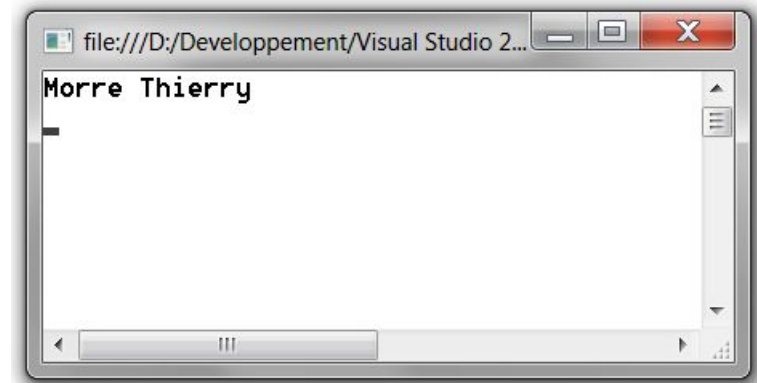
Opérateur « SingleOrDefault »

L'opérateur « SingleOrDefault » retourne un élément unique d'une séquence de type 'T', si la valeur est introuvable, il retourne la valeur de « default(T) ».

```
public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source);
```

```
Contact QueryResult = Contacts
    .Where(c => c.Prenom.Equals("Thierry") && c.Nom.Equals("Morre"))
    .SingleOrDefault();

Console.WriteLine("{0} {1}", QueryResult.Nom, QueryResult.Prenom);
```



Utilisation avec une expression de requête :

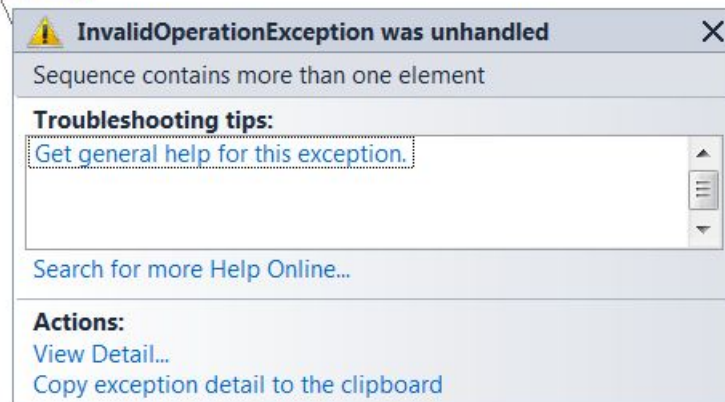
```
Contact QueryResult = (from c in Contacts
    where c.Prenom.Equals("Thierry") && c.Nom.Equals("Morre")
    select c).SingleOrDefault();

Console.WriteLine("{0} {1}", QueryResult.Nom, QueryResult.Prenom);
```

Opérateur « SingleOrDefault »

Cependant, si la requête retourne plus d'un éléments, l'opérateur « SingleOrDefault » lèvera une exception.

```
Contact QueryResult = (from c in Contacts  
    where c.Prenom.Equals("Thierry")  
    select c).SingleOrDefault();
```



Dans ce cas de figure et afin d'éviter l'erreur, il est conseillé d'utiliser « FirstOrDefault ».

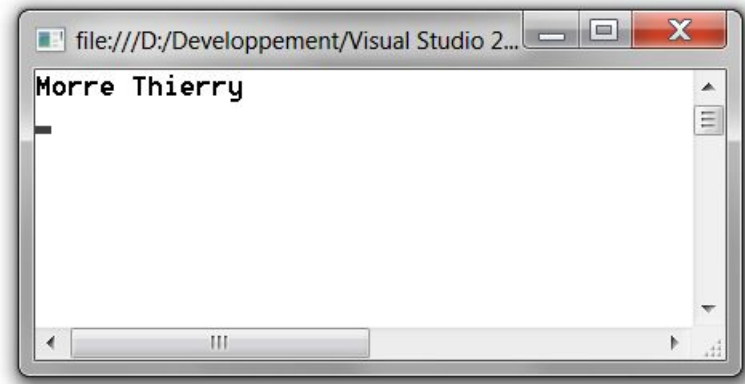
Opérateur « FirstOrDefault »

L'opérateur « FirstOrDefault » retourne le premier élément d'une séquence de type 'T' ou la valeur de « default(T) » dans le cas où aucune valeur ne serait trouvée.

```
public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source);
```

```
Contact QueryResult = Contacts
    .Where(c => c.Prenom.Equals("Thierry"))
    .FirstOrDefault();

if(QueryResult != null)
    Console.WriteLine("{0} {1}", QueryResult.Nom, QueryResult.Prenom);
```



Opérateurs « OrderBy[Descending] »

L'opérateur « OrderBy » trie par ordre croissant les éléments d'une séquence sur base d'une clé (propriété). Pour un ordre décroissant, nous devons utiliser « OrderByDescending ». Si nous souhaitons trier sur base de plusieurs clés, ces opérateur doivent être les premiers à être utiliser.

```
public static IEnumerable<T> OrderBy<T, TKey>(this IEnumerable<T> source, Func<T, TKey> keySelector);  
public static IEnumerable<T> OrderByDescending<T, TKey>(this IEnumerable<T> source, Func<T, TKey> keySelector);
```

```
IEnumerable<Contact> QueryResult = Contacts.OrderBy(c => c.AnneeDeNaissance);  
  
foreach (Contact c in QueryResult)  
{  
    Console.WriteLine("{0} {1}", c.Nom, c.AnneeDeNaissance);  
}
```

Expression de requête :

```
IEnumerable<Contact> QueryResult = from c in Contacts  
                                   orderby c.AnneeDeNaissance  
                                   select c;  
  
foreach (Contact c in QueryResult)  
{  
    Console.WriteLine("{0} {1}", c.Nom, c.AnneeDeNaissance);  
}
```



Opérateurs « ThenBy[Descending] »

Les opérateurs « ThenBy » (Croissant) et « ThenByDescending » (Décroissant) permettent de trier une séquence sur plusieurs clés (propriétés). Nous pouvons avoir autant de clé « ThenBy » ou « ThenByDescending » que nous voulons. Cependant, ceux-ci ne sont utilisables que si un des opérateurs « OrderBy » ou « OrderByDescending » a été déclaré en premier lieu.

```
public static IOrderedEnumerable<T> ThenBy<T, TKey>(this IOrderedEnumerable<T> source, Func<T, TKey> keySelector);  
public static IOrderedEnumerable<T> ThenByDescending<T, TKey>(this IOrderedEnumerable<T> source, Func<T, TKey> keySelector);
```

```
IEnumerable<Contact> QueryResult = Contacts  
    .OrderBy(c => c.AnneeDeNaissance)  
    .ThenByDescending(c => c.Nom);  
  
foreach (Contact c in QueryResult)  
{  
    Console.WriteLine("{0} {1}", c.Nom, c.AnneeDeNaissance);  
}
```

Expression de requête :

```
IEnumerable<Contact> QueryResult = from c in Contacts  
    orderby c.AnneeDeNaissance, c.Nom descending  
    select c;  
  
foreach (Contact c in QueryResult)  
{  
    Console.WriteLine("{0} {1}", c.Nom, c.AnneeDeNaissance);  
}
```

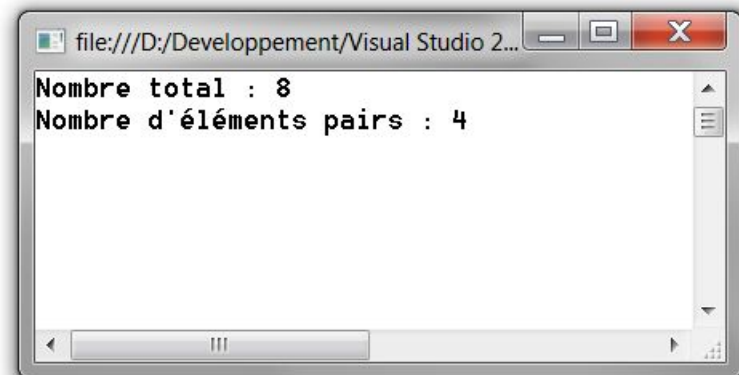


Opérateurs « Count » & « LongCount »

```
public static int Count<TSource>(this IEnumerable<TSource> source);  
public static int Count<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);  
public static long LongCount<TSource>(this IEnumerable<TSource> source);  
public static long LongCount<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

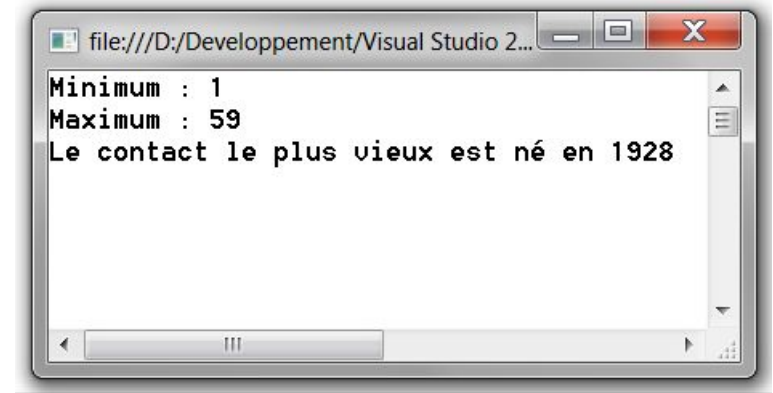
Ces deux opérateurs comptent le nombre d'éléments dans une collection. La différence se situe dans leur type de retour. Ceux-ci peuvent être accompagnés d'une expression booléenne qui va permettre de filtrer le nombre d'éléments à compter.

```
int[] ints = new int[] { 5, 4, 7, 52, 36, 59, 24, 1 };  
// Retourne le nombre de l'ensemble des éléments  
Console.WriteLine(string.Format("Nombre total : {0}", ints.Count()));  
// retourne le nombre des éléments pairs  
Console.WriteLine(string.Format("Nombre d'éléments pairs : {0}", ints.Count(i => i % 2 == 0)));
```



Opérateurs « Min » & « Max »

Comme l'indique leurs noms, les opérateurs « Min » et « Max » retournent respectivement la valeur minimale et maximale d'une collection.

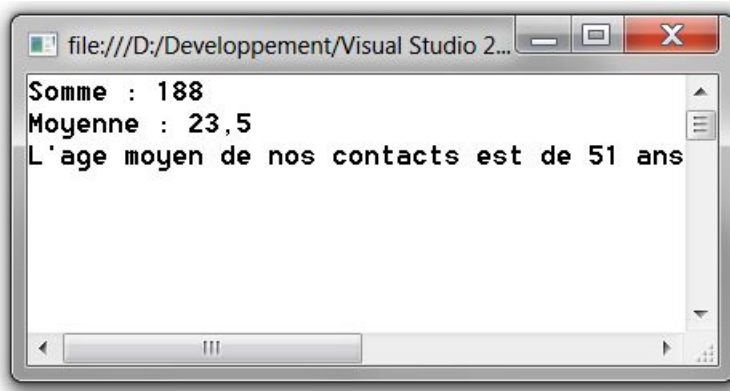


```
int[] ints = new int[] { 5, 4, 7, 52, 36, 59, 24, 1 };  
// Retourne la plus petite valeur  
Console.WriteLine(string.Format("Minimum : {0}", ints.Min()));  
// Retourne la plus grande valeur  
Console.WriteLine(string.Format("Maximum : {0}", ints.Max()));  
// Retourne l'année de naissance du contact le plus vieux  
int AnnéeDeNaissance = Contacts.Min(c => c.AnneeDeNaissance);  
Console.WriteLine("Le contact le plus vieux est né en {0}", AnnéeDeNaissance);
```

Opérateurs « Sum » & « Average »

Les opérateurs « Sum » et « Average » retournent respectivement la somme et la moyenne d'une collection.

```
int[] ints = new int[] { 5, 4, 7, 52, 36, 59, 24, 1 };  
// Retourne la somme  
Console.WriteLine(string.Format("Somme : {0}", ints.Sum()));  
// Retourne la moyenne  
Console.WriteLine(string.Format("Moyenne : {0}", ints.Average(i => (float)i)));  
// Retourne l'age moyen des contacts  
Console.WriteLine("L'age moyen de nos contacts est de {0} ans",  
    DateTime.Now.Year - (int)Contacts.Average(c => c.AnneeDeNaissance));
```



Opérateur « GroupBy »

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
```

L'opérateur « GroupBy » est un peu différent des autres, par défaut celui-ci travail avec l'interface « IGrouping<Tkey, TElement> » qui hérite de « IEnumerable<T> » et intégrant une propriété « Key ».

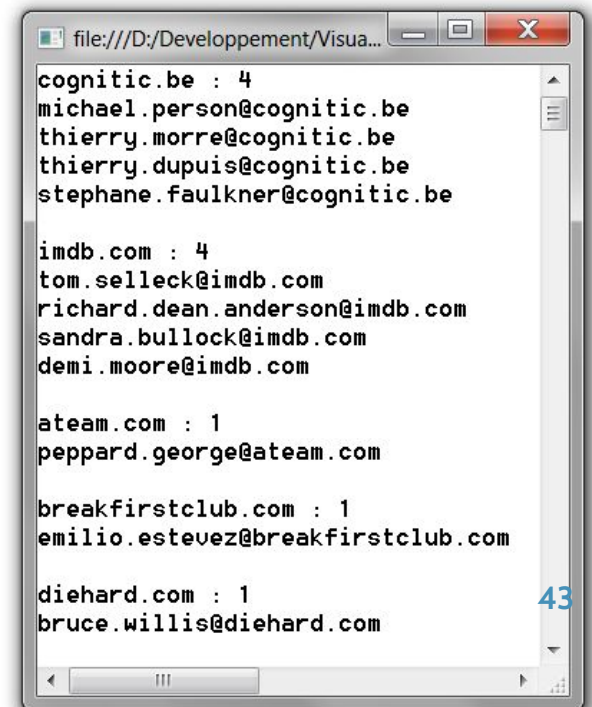
```
...public interface IGrouping<out TKey, out TElement> : IEnumerable<TElement>, IEnumerable
{
    ...TKey Key { get; }
}
```

Ce qui donne ceci :

```
IEnumerable<IGrouping<string, Contact>> QueryResult = Contacts
    .GroupBy(c => c.Email.Substring(c.Email.IndexOf('@') + 1));

foreach (IGrouping<string, Contact> g in QueryResult)
{
    Console.WriteLine("{0} : {1}", g.Key, g.Count());
    foreach (Contact c in g)
    {
        Console.WriteLine("{0}", c.Email);
    }
    Console.WriteLine();
}
```

Il existe plusieurs surcharges, ici n'est vue que la plus utilisée.



```
file:///D:/Developpement/Visua...
cognitic.be : 4
michael.person@cognitic.be
thierry.morre@cognitic.be
thierry.dupuis@cognitic.be
stephane.faulkner@cognitic.be

imdb.com : 4
tom.selleck@imdb.com
richard.dean.anderson@imdb.com
sandra.bullock@imdb.com
demi.moore@imdb.com

ateam.com : 1
peppard.george@ateam.com

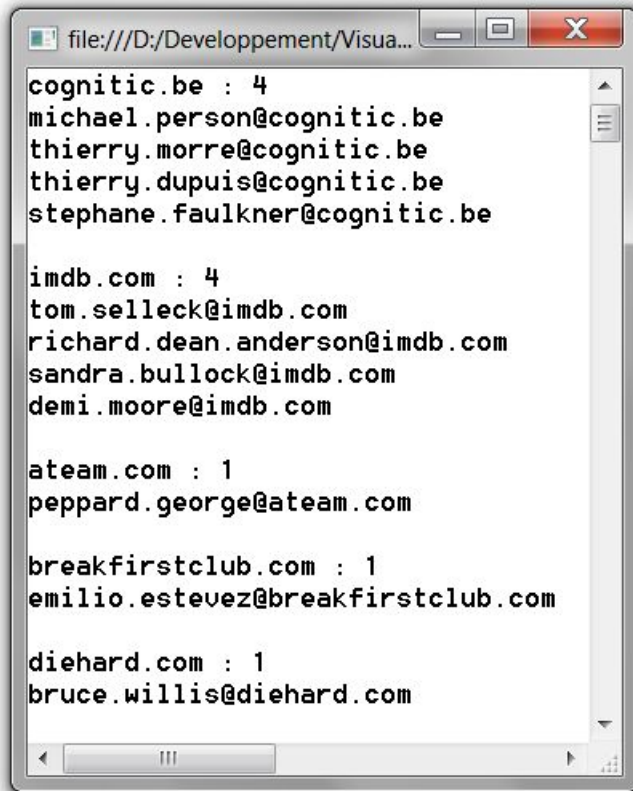
breakfirstclub.com : 1
emilio.estevez@breakfirstclub.com

diehard.com : 1
bruce.willis@diehard.com
```

Opérateur « GroupBy »

De plus, dans le cadre de l'expression régulière, « GroupBy » ne peut être utilisé avec l'opérateur « Select » excepté dans le cadre du « group ... by ... into ... ».

Expression de requête



```
file:///D:/Developpement/Visua...
cognitic.be : 4
michael.person@cognitic.be
thierry.morre@cognitic.be
thierry.dupuis@cognitic.be
stephane.faulkner@cognitic.be

imdb.com : 4
tom.selleck@imdb.com
richard.dean.anderson@imdb.com
sandra.bullock@imdb.com
demi.moore@imdb.com

ateam.com : 1
peppard.george@ateam.com

breakfirstclub.com : 1
emilio.estevez@breakfirstclub.com

diehard.com : 1
bruce.willis@diehard.com
```

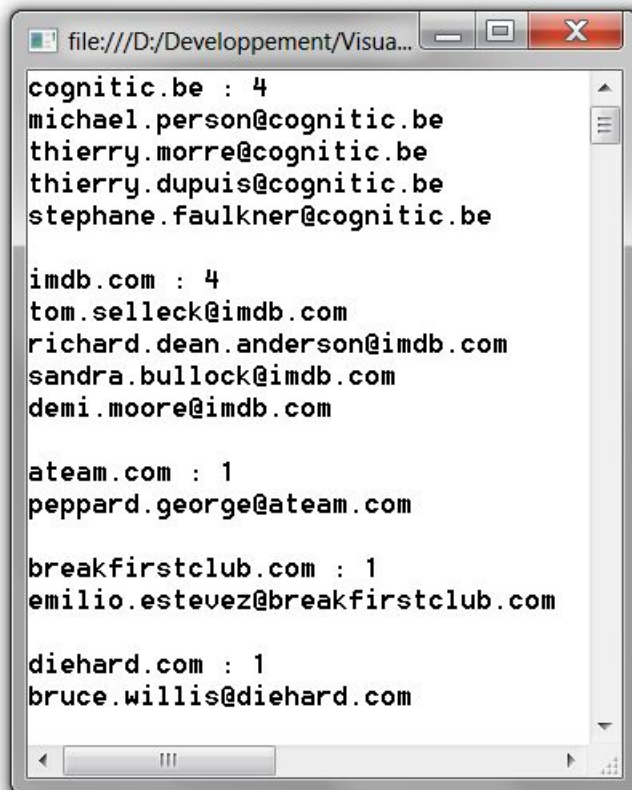
```
IEnumerable<IGrouping<string, Contact>> QueryResult =
    from c in Contacts
    group c by c.Email.Substring(c.Email.IndexOf('@') + 1);

foreach (IGrouping<string, Contact> g in QueryResult)
{
    Console.WriteLine("{0} : {1}", g.Key, g.Count());
    foreach (Contact c in g)
    {
        Console.WriteLine("{0}", c.Email);
    }
    Console.WriteLine();
}
```

Notons l'absence de l'opérateur « Select » dans la requête.

Opérateur « GroupBy »

Expression de requête :



```
file:///D:/Developpement/Visua...
cognitic.be : 4
michael.person@cognitic.be
thierry.morre@cognitic.be
thierry.dupuis@cognitic.be
stephane.faulkner@cognitic.be

imdb.com : 4
tom.selleck@imdb.com
richard.dean.anderson@imdb.com
sandra.bullock@imdb.com
demi.moore@imdb.com

ateam.com : 1
peppard.george@ateam.com

breakfirstclub.com : 1
emilio.estevez@breakfirstclub.com

diehard.com : 1
bruce.willis@diehard.com
```

```
IEnumerable<IGrouping<string, Contact>> QueryResult =
    from c in Contacts
    group c by c.Email.Substring(c.Email.IndexOf('@') + 1);

foreach (IGrouping<string, Contact> g in QueryResult)
{
    Console.WriteLine("{0} : {1}", g.Key, g.Count());
    foreach (Contact c in g)
    {
        Console.WriteLine("{0}", c.Email);
    }
    Console.WriteLine();
}
```

Notons l'absence de l'opérateur « Select » dans la requête.

Opérateur « GroupBy »

Dans ce cas, comment utiliser l'opérateur « GroupBy » avec les types anonymes ?

```
var QueryResult = Contacts
    .Select(c => new { Email = c.Email,
                     FullName = string.Format("{0} {1}", c.Nom, c.Prenom) })
    .GroupBy(c => c.Email.Substring(c.Email.IndexOf('@') + 1));

foreach (var group in QueryResult)
{
    Console.WriteLine("{0} : {1}", group.Key, group.Count());
    foreach (var element in group)
    {
        Console.WriteLine("{0}", element.FullName);
    }
    Console.WriteLine();
}
```

En utilisant l'opérateur « Select » avant le « GroupBy » ...



```
file:///D:/Developpe...
cognitic.be : 4
Person Michael
Morre Thierry
Dupuis Thierry
Faulkner Stéphane

imdb.com : 4
Selleck Tom
Anderson Richard Dean
Bullock Sandra
Moore Demi

ateam.com : 1
Peppard George

breakfirstclub.com : 1
Estevez Emilio

diehard.com : 1
Willis Bruce
```


Opérateur « GroupBy »

Expression de requête :

```
var QueryResult = from c2 in (from c in Contacts
                              select new { Email = c.Email,
                                           FullName = string.Format("{0} {1}", c.Nom, c.Prenom) })
                  group c2 by c2.Email.Substring(c2.Email.IndexOf('@') + 1);

foreach (var group in QueryResult)
{
    Console.WriteLine("{0} : {1}", group.Key, group.Count());
    foreach (var element in group)
    {
        Console.WriteLine("{0}", element.FullName);
    }
    Console.WriteLine();
}
```

... qui se traduira par une sous-requête dans l'expression de requête.

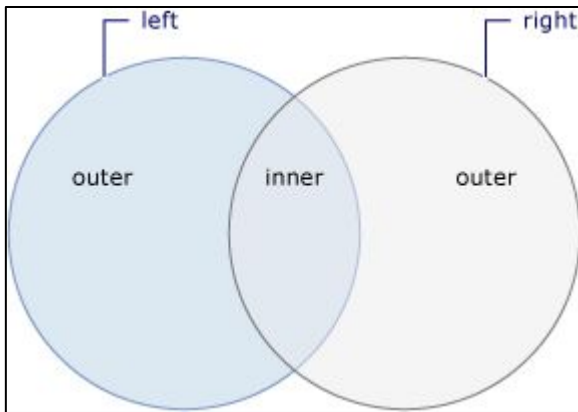


Opérateur « Join »

L'opérateur « Join » est utile pour associer des séquences différentes sur base de valeurs pouvant être comparées pour définir une égalité.

La jointure est une opération importante dans les requêtes qui ciblent les sources de données dont les relations ne peuvent pas être suivies directement. Dans la programmation orientée objet, cela pourrait signifier **une corrélation entre objets qui n'est pas modélisée***.

full



Quand on parle de jointures, il y en a 3 qui reviennent régulièrement :

Croisée (Cross Join), Interne (Inner Join), Externe (Outer Join - Left, Right & full). Celles-ci peuvent être basées sur une égalité « Equi Join » et ou non « Non Equi Join ».

L'opérateur « Join » en LINQ ne reprend qu'une seule forme de jointure (« Inner Join » basée sur une égalité). Cependant nous allons voir comment réaliser les autres jointures en utilisant LINQ.

Pour les « Cross Join » ou les « Non Equi Join », nous ne pourrions pas utiliser l'opérateur « Join ». Cependant nous pourrions contourner le problème par l'utilisation de plusieurs clauses « from » et l'utilisation de clauses « where ».

Opérateur « Join »

Afin de comprendre les jointures en LINQ, nous allons ajouter une nouvelle classe « RDV » et de nouvelles données (Liste « RendezVous ») à notre environnement.

```
public class RDV
{
    public string Email { get; set; }
    public DateTime Date { get; set; }
}
```

```
List<RDV> RendezVous = new List<RDV>();
RendezVous.AddRange(new RDV[] {
    new RDV(){ Email = "stephane.faulkner@cognitic.be", Date = new DateTime(2012,5,12)},
    new RDV(){ Email = "peppard.george@ateam.com", Date = new DateTime(2011,8,14)},
    new RDV(){ Email = "bruce.willis@diehard.com", Date = new DateTime(2012,6,19)},
    new RDV(){ Email = "bruce.willis@diehard.com", Date = new DateTime(2012,6,20)},
    new RDV(){ Email = "michael.person@cognitic.be", Date = new DateTime(2012,04,19)},
});
```

Opérateur « Join »

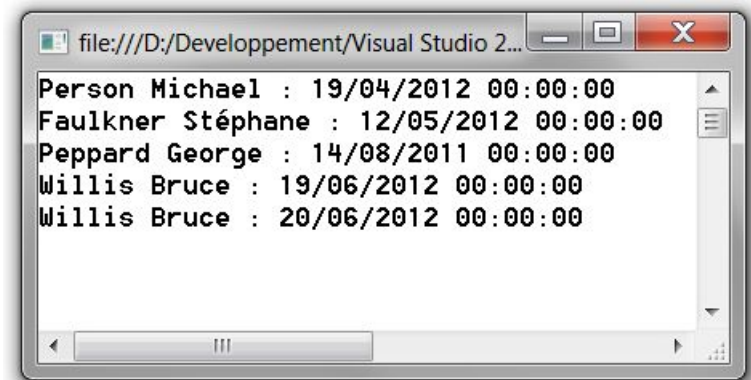
```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector);
```

Inner Join :

Obtenir l'email, le nom, le prénom du contact et la date de tous les rendez-vous.

```
var QueryResult = Contacts.Join(RendezVous,
    c => c.Email,
    rdv => rdv.Email,
    (c, rdv) => new {
        Email = c.Email,
        Nom = c.Nom,
        Prenom = c.Prenom,
        DateRDV = rdv.Date});

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} : {2}", jointure.Nom, jointure.Prenom, jointure.DateRDV);
}
```

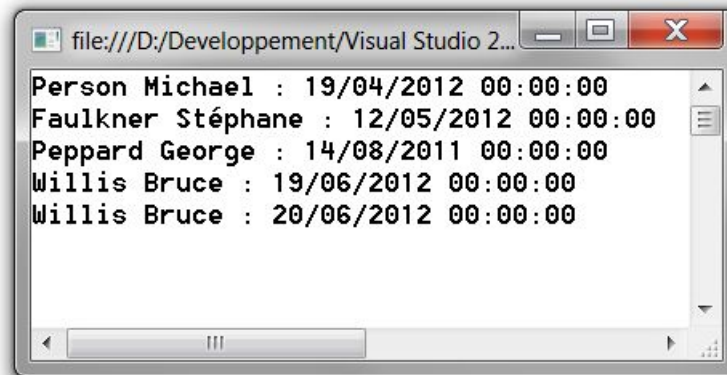


Opérateur « Join »

Expression de requête :

```
var QueryResult = from c in Contacts
                  join rdv in RendezVous on c.Email equals rdv.Email
                  select new {
                      Email = c.Email,
                      Nom = c.Nom,
                      Prenom = c.Prenom,
                      DateRDV = rdv.Date};

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} : {2}", jointure.Nom, jointure.Prenom, jointure.DateRDV);
}
```



Opérateur « Join »

L'utilisation des types anonymes n'est pas obligatoire dans le cadre des jointures, son emploi résulte en effet de ce que nous allons sélectionner.

Exemple : Obtenir les contacts ayant pris rendez-vous.

```
IEnumerable<Contact> QueryResult = Contacts.Join(RendezVous,  
                                                c => c.Email,  
                                                rdv => rdv.Email,  
                                                (c, rdv) => c);
```

Expression de requête

:

```
IEnumerable<Contact> QueryResult = from c in Contacts  
join rdv in RendezVous on c.Email equals rdv.Email  
select c;
```

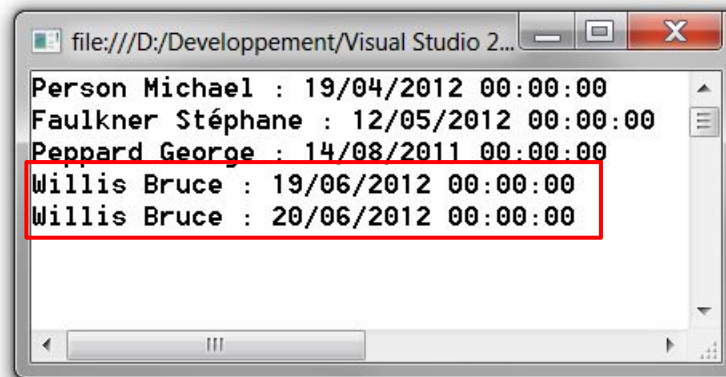
Nous obtenons bien une séquence de Contacts

Opérateur « GroupJoin »

```
public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
```

L'opérateur « GroupJoin » travaille de manière comparable à l'opérateur « Join », à ceci près que l'opérateur « Join » ne passe qu'un seul élément de la séquence externe et un élément de la séquence interne à la fonction « resultSelector ».

Cela signifie que si plusieurs éléments de la séquence intérieur (inner) correspondent à un élément de la séquence extérieur (outer), nous aurons plusieurs lignes dans notre « result set ».



L'opérateur « GroupJoin » va, quant à lui, produire une structure de donnée hiérarchique. Il va associer pour chaque élément de la séquence extérieur les éléments de la séquence intérieure qui le concerne.

Si aucun élément de la séquence intérieur n'existe, il retournera une séquence vide.

Il s'apparente donc à une jointure externe gauche (« Left Join »).

Opérateur « GroupJoin »

Exemple :

Pour tous les contacts, obtenir les noms, prénoms et date de rendez-vous éventuels.

```
var QueryResult = Contacts.GroupJoin(RendezVous,
    c => c.Email,
    rdv => rdv.Email,
    (c, rdvs) => new {
        Email = c.Email,
        Nom = c.Nom,
        Prenom = c.Prenom,
        RendezVous = rdvs});

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} :", jointure.Nom, jointure.Prenom);
    if (jointure.RendezVous.Count() > 0)
    {
        foreach (RDV rdv in jointure.RendezVous)
        {
            Console.WriteLine("{0}", rdv.Date);
        }
    }
    else
    {
        Console.WriteLine("Aucun");
    }
    Console.WriteLine();
}
```



```
file:///D:/Developpeme...
Person Michael :
19/04/2012 00:00:00

Morre Thierry :
Aucun

Dupuis Thierry :
Aucun

Faulkner Stéphane :
12/05/2012 00:00:00

Selleck Tom :
Aucun

Anderson Richard Dean :
Aucun

Bullock Sandra :
Aucun

Peppard George :
14/08/2011 00:00:00

Estevez Emilio :
Aucun

Moore Demi :
Aucun

Willis Bruce :
19/06/2012 00:00:00
20/06/2012 00:00:00
```

Opérateur « GroupJoin »

Expression de requête :

```
var QueryResult = from c in Contacts
                  join rdv in RendezVous on c.Email equals rdv.Email into grdvs
                  select new { Nom = c.Nom, Prenom = c.Prenom, RendezVous = grdvs };

foreach (var jointure in QueryResult)
{
    Console.WriteLine("{0} {1} :", jointure.Nom, jointure.Prenom);
    if (jointure.RendezVous.Count() > 0)
    {
        foreach (RDV rdv in jointure.RendezVous)
        {
            Console.WriteLine("{0}", rdv.Date);
        }
    }
    else
    {
        Console.WriteLine("Aucun");
    }
    Console.WriteLine();
}
```

COGNITIC



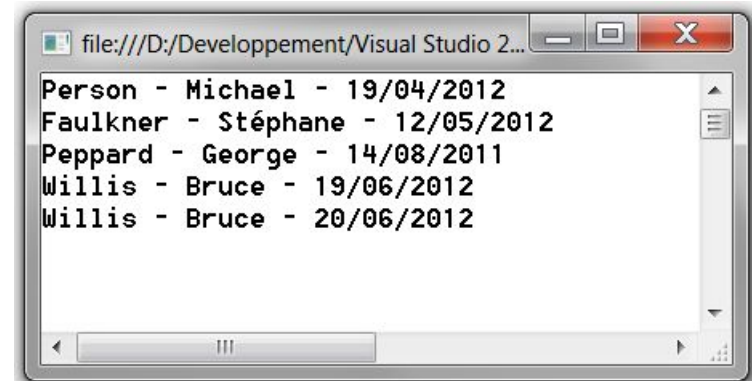
Utilisation de clés composites

Dans le cadre de jointures, nous sommes parfois amenés à gérer les clés étrangères utilisant plusieurs champs (clé composite) et « LINQ » n'échappe pas à la règle. Afin de résoudre ce « problème » nous devons utiliser les classes anonymes.

```
var QueryResult = from c in Contacts
    join rdv in RendezVous on new { c.ID, c.Email } equals new { rdv.ID, rdv.Email }
    select new { c.Nom, c.Prenom, rdv.Date };

var QueryResult2 = Contacts.Join(RendezVous,
    c => new { c.ID, c.Email },
    rdv => new { rdv.ID, rdv.Email },
    (c, rdv) => new { c.Nom, c.Prenom, rdv.Date });

foreach (var r in QueryResult)
{
    Console.WriteLine("{0} - {1} - {2}", r.Nom, r.Prenom, r.Date.ToShortDateString());
}
```



Multiple clause « from »

Nous venons de voir que les jointures en « LINQ » sont des jointures internes basées sur une égalité. Comment dans ce cas faire une jointure croisée (« Cross Join ») ou une « Non Equi Join » ?

Elles ne sont possibles que dans le cadre des expressions de requêtes en utilisant plusieurs clauses « from ».

Exemple de « Cross Join » :

```
var QueryResult = from c in Contacts
                  from rdv in RendezVous
                  select new { c.Nom, c.Prenom, rdv.Date };

foreach (var r in QueryResult)
{
    Console.WriteLine("{0} - {1} - {2}", r.Nom, r.Prenom, r.Date.ToShortDateString());
}
```

