FAQ Linq

Date de publication : 01/01/2008

Dernière mise à jour :

Cette FAQ est essentiellement consacrée à Linq (Language Integrated Query).

Si vous souhaitez participer à sa rédaction vous pouvez en faire la demande à **Jérôme Lambert**.

Ont contribué à cette FAQ :

Jérôme Lambert (Site perso de Jérôme Lambert) (Blog) -

1. Informations générales (9) 4
2. Modifications du langage (5) 8
3. Fondements de Linq (17) 17

Sommaire > Informations générales

Qu'est-ce que Linq ?

Auteurs : Jérôme Lambert ,

Un problème récurent pour tout développeur est l'accès aux données. Prenons le cas d'une base de données, un développeur C# ou VB.NET doit dans un premier temps apprendre un nouveau langage qu'est SQL pour ensuite établir une relation entre son modèle de développement orienté objets et le modèle relationnel de la plupart des bases de données.

Outre le problème des bases de données, il existe bien d'autres sources de données que le développeur a l'habitude d'interroger mais rarement de la même façon. Une source de données provenant d'une base de données sera interrogée bien différemment qu'une source de données d'objets métiers ou d'un fichier XML.

Linq, comme son nom l'indique "*Language Integrated Query*", a été développé pour permettre d'interroger des sources de données totalement différentes mais avec une approche identique. Cependant, Linq n'est en rien une technologie mais un ensemble d'améliorations aux niveaux des langages de programmation (C # 3.0 et VB.NET 9). De base, Linq offre la possibilité d'interroger vos collections d'objets (Linq to Objects), du XML (Linq to XML), les bases de données SQL Server (Linq to SQL) et vos DataSets (Linq to DataSet). Cependant, Linq est extensible car il fonctionne sur le principe bien connu des providers. Vous êtes donc tout à fait libre de développer votre propre provider Linq pour interroger des sources de données d'un tout autre type.

Linq apporte aussi une syntaxe très particulière au niveau des langages de programmation appelée aussi " Sugar Syntax " car elle est très proche de la syntaxe SQL. Un petit exemple vous fera tout de suite comprendre cela :

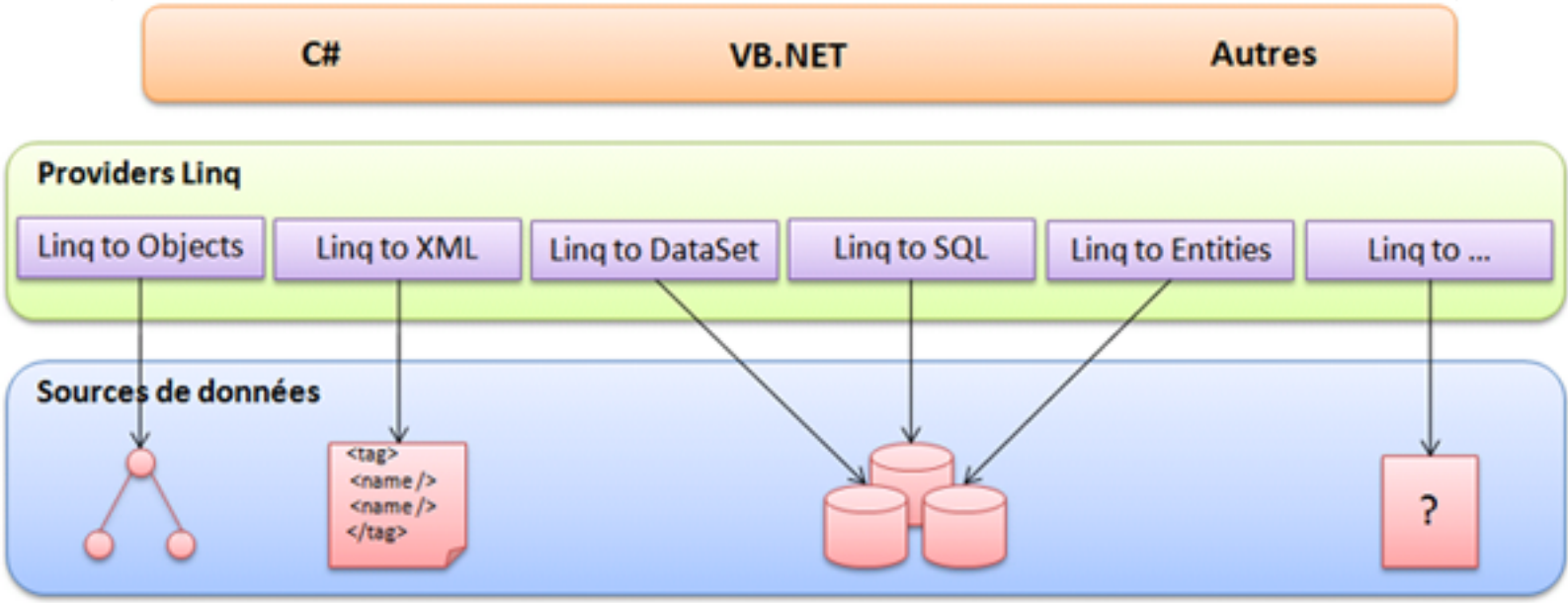
Exemple de requête Linq en C#

```
// Retourne la liste des processus commençant par 'A' triée par ordre alphabétique croissant
var query = from process in System.Diagnostics.Process.GetProcesses()
   where process.ProcessName.StartsWith("A")
   orderby process.ProcessName ascending
   select process;
```

Exemple de requête Linq en VB.NET

```
' Retourne la liste des processus commençant par 'A' triée par ordre alphabétique croissant
Dim query = _
   From process In System.Diagnostics.Process.GetProcesses() _
   Where process.ProcessName.StartsWith("A") _
   Order By process.ProcessName Ascending _
   Select process
```

En résumé, on pourrait résumer Linq avec le schema suivant :



Qu'est-ce que XLinq et DLinq ?

Auteurs : Jérôme Lambert ,

XLinq est l'ancien nom de Linq to XML permettant d'interroger du XML.

DLinq est l'ancien nom de Linq to SQL permettant d'interroger des bases de données.

Quels langages supportent Linq ?

Auteurs : Jérôme Lambert ,

Depuis sa sortie, Linq est supporté par les deux langages de programmation principaux de Microsoft et qui sont C# (en version 3.0) et VB.NET (en version 9).

Cependant, Linq est ouvert afin que d'autres langages puissent le supporter, c'est le cas en partie pour F# et C++/CLI.

Quels types de sources de données peut-on interroger avec Linq ?

Auteurs : Jérôme Lambert ,

Il est impossible de répondre de manière exhaustive à cette question étant donné que Linq est une technologie extensible. De base, Linq permet d'interroger les sources de données suivantes :

- **Objets en mémoire avec Linq to Objects**
- **XML avec Linq to XML**
- **Base de données SQL Server avec Linq to SQL**
- **DataSets avec Linq to DataSet**
- **Modèle Entity avec Linq to Entity**

Mais de nouveaux providers Linq développés par des sociétés ou des particuliers apparaissent sans cesse, il est donc difficile de d'établir une liste de tous ces providers. Cependant, nous vous recommandons de vous rendre sur le blog de Charlie Calvert (membre de l'équipe C# de Microsoft) qui propose une liste non exhaustive des providers Linq extistants.

lien : Blog de Charlie Calvert

A partir de quelle version du Framework .NET peut-on faire du Linq ?

Auteurs : Jérôme Lambert ,

Linq est disponible en version finale à partir du Framework 3.5 pour C# et VB.NET.

A partir de quelle version de Visual Studio peut-on utiliser Linq ?

Auteurs : Jérôme Lambert ,

Vous pouvez utiliser Linq à partir de Visual Studio 2008 (toutes versions confondues).

Il existe quand même une possibilité pour utiliser en partie Linq avec Visual Studio 2005 : Est-il possible d'utiliser Linq avec Visual Studio 2005 ?.

lien : Téléchargez Visual Express 2008
lien : Est-il possible d'utiliser Linq avec Visual Studio 2005 ?

Quelles sont les assemblées nécessaires pour utiliser Linq ?

Auteurs :

L'ensemble des classes et interfaces nécessaires pour utiliser sont disponible dans un ensemble d'assemblées fournies par le .NET Framework 3.5. Voici la liste des assemblées nécessaires :

- **System.Core.dll qui fournit l'ensemble des classes et interfaces nécessaires pour écrire des requêtes Linq to Objects.**
- **System.Data.DataSetExtensions.dll pour Linq to DataSet.**
- **System.Data.Linq.dll pour Linq to SQL.**
- **System.Data.Entity.dll pour Linq to Entities.**
- **System.Xml.Linq.dll pour Linq to XML.**

Attention que System.Data.Entity.dll n'est disponible qu'avec le Service Pack 1 du .NET Framework 3.5.

'

lien : Téléchargez .NET Framework 3.5 Service Pack 1

lien : Téléchargez Visual Studio 2008 Service Pack 1

Pourquoi est-ce que je n'arrive pas à utiliser Linq dans classe ?

Auteurs : Jérôme Lambert ,

Si vous n'arrivez pas à utiliser les opérateurs de Linq (Select, OrderBy, Where, etc.), reportez-vous à la question/réponse suivante : Comment utiliser Linq to Objects dans vos projets ?

lien : Comment utiliser Linq to Objects dans vos projets ?

Est-il possible d'utiliser Linq avec Visual Studio 2005 ?

Auteurs : Jérôme Lambert ,

A priori, Linq est uniquement supporté par Visual Studio 2008 grâce au Framework .NET 3.5. Il n'est donc normalement pas possible d'utiliser Linq avec Visual Studio 2005... Et pourtant, ce n'est pas tout à fait vrai.

Si vous ne le savez pas ecnore, le Framework .NET 3.5 n'est qu'une surchouche du Framework 2.0 composé d'un ensemble de nouvelles assemblées. Si on part la dessus, qu'est-ce qui empêche d'utiliser les assemblées System.Core.dll, System.Data.Linq.dll ou encore System.Xml.Linq.dll ? Rien en fait et à partir du moment où vous aurez installé le Framework .NET 3.5 sur votre machine, il vous sera tout à fait possible d'utiliser ces assemblées dans vos projets Visual Studio 2005.

Malgré cela, vous remarquerez très vite qu'il ne vous est pas possible d'utiliser toutes les possiblités de C# 3.0 et de Linq. Typage implicite, objets anonymes, initialiseurs d'objets et collections, expressions lambda et méthodes d'extension, tout cela sera impossible avec Visual Studio 2005. Au même titre, la "sugar syntax" de Linq vous permettant d'écrire des expressions de requête sera impossible aussi. La raison est que Visual Studio 2005 utilise le compilateur du .NET Framework 2.0, même après l'installation du Framework .NET 3.5. Mais à part, vous pourrez utiliser les opérateurs standards de requête sans problème.

Même si complètement déconseillé dans le milieu professionel, une autre alternative est d'installer la CTP (Community Technology Preview) de Linq destiné à Visual Studio 2005. Cette CTP permettait de montrer les nouveautés qui allaient venir avec C# 3.0 version finale. Contrairement à la technique précédente, vous allez pouvoir bénéficier des nouveautés de C# 3.0 mais aussi de la "sugar syntax" de Linq directement dans votre projet Visual Studio 2005.

Pour téléchargez cette CTP de Linq pour Visual Studio 2005, rendez-vous ici.

N'oubliez pas que ceci est une CTP, pas mal de choses ont donc changée depuis la version finale du Framework .NET 3.5.

Sommaire > Modifications du langage

Qu'est-ce que le typage implicite des variables locales ?

Auteurs : Jérôme Lambert ,

C# 3.0 a introduit un nouveau mot clé permettant de déclarer des variables sans pour autant spécifier explicitement leur type. Ce nouveau mot clé est "var".

```
var i = 12;
```

Comme vous pouvez le voir, l'utilisation de " var " est très simple : une fois la variable déclarée, il vous faut l'initialiser directement. Le compilateur va inférer automatiquement le type de votre variable.

```
// i est de type System.Int32
var i = 12;
// j est de type System.Single
var f = 5.2;
// s est de type System.String
var s = " Hello World ";
// list est de type System.Collections.Generic.List<Process>
var list = new System.Collections.Generic.List<Process>();
```

Chose importante à ne pas confondre, var est différent de " Variant " de VB qui permet de définir le type de votre variable quand vous le désirez. Ici, c'est le compilateur qui va inférer le type de votre variable sur base de l'expression d'initialisation que vous allez spécifier.

Evidemment, tout n'est pas permis et voici les règles à ne pas oublier :

- **Les variables déclarées avec var doivent être initialisées au même moment.**

```
// erreur de compilation
var i = null;
```

- **var ne peut être utilisé que dans les déclarations de variables locales.**

```
// Erreur de compilation
public class MaClass
{
 var i = 1;
}
```

- **Il est interdit d'utiliser une variable déclarée avec var dans l'expression d'initialisation.**

```
// Erreur de compilation
var i = i + 1;
```

- **Il n'est pas possible de déclarer plusieurs variables avec var dans la même instruction.**

```
// Erreur de compilation
var i1 = 1, i2 = 2;
```

Malgré cette nouveauté du langage, Microsoft recommande d'utiliser var uniquement si cela s'avère nécessaire. Une utilisation trop excessive de var risquerait de complexifier votre code inutilement. Par exemple, les types anonymes sont le cas parfait pour l'utilisation de var.

lien : Page MSDN sur les variables locales implicitement typées

Qu'est-ce que les initialiseurs d'objets et de collections ?

Auteurs : Jérôme Lambert ,

Initialiseurs d'objets

L'initialiseur d'objet vous permet d'instancier une classe tout en initialisant les champs et propriétés de cette classe que vous désirez, le tout en une seule instruction.

Avec C# 2.0

```
System.Data.DataTable objDataTable2 = new System.Data.DataTable("maDataTable");
objDataTable2.CaseSensitive = true;
objDataTable2.MinimumCapacity = 100;
```

Avec C# 3.0

```
System.Data.DataTable objDataTable = new System.Data.DataTable("maDataTable") {
 CaseSensitive = true, MinimumCapacity = 100 };
```

Initialiseurs de collections

L'initialiseur de collection fonctionne sur le même principe mais pour les collections.

Avec C# 2.0

```
List<DataTable> aDataTables = new List<DataTable>();
aDataTables.Add(new DataTable("myFirstDataTable"));
aDataTables.Add(new DataTable("mySecondDataTable"));
aDataTables.Add(new DataTable("myThirdDataTable"));
```

Avec C# 3.0

```
List<DataTable> aDataTables = new List<DataTable>()
   {
    new DataTable("myFirstDataTable"),
    new DataTable("mySecondDataTable"),
    new DataTable("myThirdDataTable")
   };
```

Vous pouvez même aller plus loin en utilisant l'initialiseur d'objet en même temps que l'initialisation de votre collection :

```
List<DataTable> aDataTables = new List<DataTable>()
  {
   new DataTable("myFirstDataTable")  { CaseSensitive = true, MinimumCapacity = 10 },
   new DataTable("mySecondDataTable") { CaseSensitive = false, MinimumCapacity = 20 },
   new DataTable("myThirdDataTable")  { CaseSensitive = true, MinimumCapacity = 30 }
```

```
};
```

lien : Page MSDN sur les initialiseurs d'objets et de collections

Qu'est-ce qu'un type anonyme ?

Auteurs : Jérôme Lambert ,

Une des grandes nouveautés de C# 3 est la possibilité des créer des types anonymes. Un type anonyme offre la possibilité de composer des types "temporaires" avec une ou plusieurs propriétés en lecture seule. Cette nouveauté s'avère extrêmement efficace avec Linq car on va pouvoir interroger des sources de données et récupérer uniquement les informations dont on a besoin au travers d'objets anonymes.

```
var anonymousType = new { Name = "Jérôme Lambert", Birthday = new DateTime(1984, 2, 25) };
Console.WriteLine("{0} est né le {1}", anonymousType.Name, anonymousType.Birthday.ToString("D"));
```

Résultat :

Jérôme Lambert est né le samedi 25 février 1984

Dans l'exemple précédent, on peut voir que j'ai créé un type qui n'existe pas, c'est pourquoi je suis obligé de déclarer ma variable "anonymousType" avec le mot clé "var" qui permet un typage implicite de variable. Pour créer un type anonyme, il suffit d'utiliser le mot clé "new" suivi d'accolades qui contiendront les propriétés. Mon objet anonyme contient deux propriétés que j'ai nommé "Name" et "Birthday". Etant donné que j'ai initalisé mes deux propriétés, le compilateur est capable de déduire le type de ces deux propriétés (System.String pour la propriété "Name" et System.DateTime pour la propriété "Birthday").

Comme je l'ai dit précédemment, l'utilisation des types anonymes s'avère extrêmement intéressante avec les requêtes Linq. Prenons l'exemple de la liste des processus de votre ordinateur : si on désire récupérer les processus commençant par la lettre "A", on peut écrire tout simplement la requête suivante :

```
var result = from process in System.Diagnostics.Process.GetProcesses()
    where process.ProcessName.StartsWith("A")
    select process;

foreach (var currentProcess in result)
 Console.WriteLine("{0} (N°{1})", currentProcess.ProcessName, currentProcess.Id);
```

Le problème avec cette requête, c'est qu'on est obligé de récupérer une collection d'objets de type System.Diagnostics.Process. Or comme on peut voir dans la boucle foreach, je n'ai besoin que du nom du processus et son ID. C'est là qu'interviennent les types anonymes car je vais pouvoir précifier que chaque résultat doit être représenté par un type qui contient uniquement les valeurs dont j'ai besoin, c'est à dire le nom du processus et son ID.

```
var result = from process in System.Diagnostics.Process.GetProcesses()
    where process.ProcessName.StartsWith("A")
    select new { process.ProcessName, process.Id };

foreach (var currentProcess in result)
 Console.WriteLine("{0} (N°{1})", currentProcess.ProcessName, currentProcess.Id);
```

Petite remarque et ce n'est pas une faute, je n'ai pas précisé le nom des propriétés dans mon objets anonymes. De ce fait, le compilateur a automatiquement créé des propriétés qui ont le même noms que les propriétés utilisées pour initialiser mon objet anonyme.

lien : Page MSDN sur les types anonymes

Comment utiliser un type anonyme en valeur de retour d'une méthode ?

Auteurs : Jérôme Lambert ,

Prenons un exemple simple qui va être de récupérer la liste des processus triés par nom.

```
var query = from proc in System.Diagnostics.Process.GetProcesses()
   orderby proc.ProcessName ascending
   select proc;

foreach (var item in query)
 Console.WriteLine("{0}\t {1}", item.Id, item.ProcessName);
```

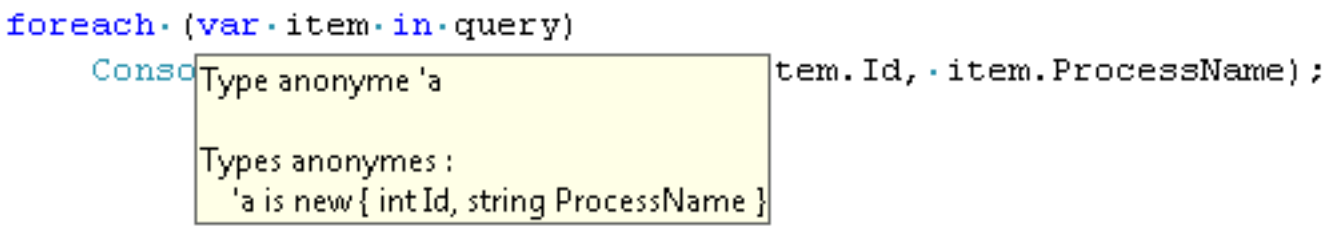
Évidemment, on est forcé de récupérer un objet de type "System.Diagnostics.Process" avec toutes les informations qui l'accompagne alors que dans notre cas, le nom de processus et son id nous auraient suffit. C'est là qu'intervient la notion de type anonyme :

```
var query = from proc in System.Diagnostics.Process.GetProcesses()
   orderby proc.ProcessName ascending
   select new { proc.Id, proc.ProcessName };

foreach (var item in query)
 Console.WriteLine("{0}\t {1}", item.Id, item.ProcessName);
```

Comme vous pouvez le voir, un type anonyme est créé en utilisant l'opérateur new suivi d'accolades pour initialiser les différentes propriétés que l'on désire voir apparaître dans son objet anonyme. Dans la boucle foreach, on s'aperçoit que "item" propose bien deux propriétés "Id" et "ProcessName". Chose importante à noter, "item" représente un type anonyme, il n'est donc pas possible de le nommer... Il faut donc passer par l'utilisation du mot clé var qui déduira le type automatiquement.

Pour vous en convaincre que "item" n'est pas de type "*System.Diagnostics.Process*", voici la preuve en image :



La compilateur a donc effectivement créé un type avec les propriétés adéquates. Le type de ces mêmes propriétés a été au passage inféré grâce à l'initialisation. Bien sûr, rien ne vous empêche de nommer vos noms de propriétés comme vous le désirez :

```
var query = from proc in System.Diagnostics.Process.GetProcesses()
   orderby proc.ProcessName ascending
   select new { ProcessID = proc.Id, Name = proc.ProcessName };

foreach (var item in query)
 Console.WriteLine("{0}\t {1}", item.ProcessID, item.Name);
```

Si on jette un petit coup d'oeil au niveau du code compilé avec Reflector (http://www.red-gate.com/products/reflector/), on remarque qu'une classe générique a été créée automatiquement pour représenter notre type anonyme.

Disassembler

```
[DebuggerDisplay(@"\{ ProcessID = {ProcessID}, Name = {Name} }", Type="<Anonymous Type>"),
internal sealed class <>f__AnonymousType0<<ProcessID>j__TPar, <Name>j__TPar>
{
   // Fields
   [DebuggerBrowsable(DebuggerBrowsableState.Never)]
   private readonly <Name>j__TPar <Name>i__Field;
   [DebuggerBrowsable(DebuggerBrowsableState.Never)]
   private readonly <ProcessID>j__TPar <ProcessID>i__Field;

   // Methods
   [DebuggerHidden]
   public <>f__AnonymousType0(<ProcessID>j__TPar ProcessID, <Name>j__TPar Name);
   [DebuggerHidden]
   public override bool Equals(object value);
   [DebuggerHidden]
   public override int GetHashCode();
   [DebuggerHidden]
   public override string ToString();

   // Properties
   public <Name>j__TPar Name { get; }
   public <ProcessID>j__TPar ProcessID { get; }
}
```

Cette nouveauté de types anonymes n'est donc possible que grâce à un changement du compilateur C#, le CLR (Common Language Runtime) n'a en aucun cas été modifié.

Maintenant que vous avons mis les choses à plat sur ce qu'est un type anonyme, penchons nous sur le sujet de ce billet et qui est de savoir comment il est possible de renvoyer un type anonyme à partir d'une méthode.

Comme je l'ai dit tout à l'heure, le type anonyme n'ayant pas de nom, on est obligé de passer par le mot clé var pour manipuler de tels objets. Si on désire encapsuler dans une méthode notre code qui permet de récupérer cette fois-ci le premier processus, cette méthode sera obligée de renvoyer un objet de type "System.Object". Effectivement, peu importe qu'on ait un type anonyme ou pas, tous dérive de la classe "System.Object".

```
static System.Object GetProcesses()
{
 return (from proc in System.Diagnostics.Process.GetProcesses()
   orderby proc.ProcessName ascending
   select new { ProcessID = proc.Id, Name = proc.ProcessName }).First();
}
```

Seulement petit problème, on perd l'intellisense en ayant converti notre type anonyme en "System.Object", donc impossibilité de récupérer les valeurs dans les propriétés "ProcessID" et "Name". Solution que vous allez de suite me proposer : La réflexion ! Et vous avez raison sur le point que ça fonctionnera mais niveau performance, on sera loin... En fait, il y a une solution beaucoup plus simple et c'est en allant voir le code compilé que nous en aurons la preuve !

Tout d'abord, nous avons notre objet anonyme de la méthode "GetProcesses" qui est composé des éléments suivants :

- **Une propriété "ProcessID" de type System.Int32**
- **Une propriété "Name" de type System.String**

Maintenant, nous allons faire un test en créant exactement le même type anonyme mais en dehors de cette méthode, par exemple dans la méthode "Main" :

```
var anonymousType = new { ProcessID = 0, Name = "" };
```

Et si on vérifie le code compilé, on s'aperçoit que le compilateur a détecté que dans notre programme, il y avait deux types anonymes exactement les mêmes (même noms de propriétés, dans le même ordre et avec le même type) : il a donc tout simplement généré une et une seule classe pour ces deux types anonymes identiques.

A partir de cette idée, on peut très facilement implémenter une méthode générique qui prendra en paramètres :

- **L'instance de notre objet anonyme de type "System.Object"**
- **Et une autre instance "bidon" de ce même type mais passée de manière générique**

Une simple conversion et le tour est joué !

```
class Program
{
    static void Main(string[] args)
    {
        var result = CastAnonymousObject(GetProcesses(), new { ProcessID = 0, Name = "" });
        Console.WriteLine("{0}\t {1}", result.ProcessID, result.Name);
        Console.Read();
    }

    static System.Object GetProcesses()
    {
        return (from proc in System.Diagnostics.Process.GetProcesses()
                orderby proc.ProcessName ascending
                select new { ProcessID = proc.Id, Name = proc.ProcessName }).First();
    }

    static A CastAnonymousObject<A>(object anomymousObject, A anonymousType)
    {
        return (A)anomymousObject;
    }
}
```

lien : FAQ Qu'est-ce qu'un type anonyme ?
lien : Lien vers l'article original

Qu'est-ce qu'une méthode d'extension ?

Auteurs : Jérôme Lambert ,

Les méthodes d'extension sont des méthodes statiques qui vous permettent d'étendres les types existants sans avoir besoin de créer un type dérivé ou bien modifier/recompiler le type original. Au final, il n'y a aucune différence

entre appeler une méthode d'extension ou une méthode qui existent réellement dans le type concerné. Les méthodes d'extension sont considérées comme des méthodes d'instance de type pour le développeur.

Pour vous montrer un exemple concrêt, nous allons créer une méthode d'extension pour le type System.String qui permettra de convertir un string en System.Int32 si cela s'avère possible, dans le cas contraire nous lancerons une exception de type System.FormatException.

Version classique

```
class Program
{
 static void Main(string[] args)
 {
  string inputString = Console.ReadLine();

  // Tentative de conversion
  try
  {
   int result = UtilityString.ToInt32(inputString);
   Console.WriteLine("Conversion réussie !");
  }
  catch(System.FormatException)
  {
   Console.WriteLine("Conversion impossible !");
  }

  Console.Read();
 }
}

static class UtilityString
{
 public static System.Int32 ToInt32(string inputString)
 {
  return System.Int32.Parse(inputString);
 }
}
```

Et non, ce n'est pas encore ça une méthode d'extension. Le code ci-dessous est ce que vous auriez écrit avec C# 2.0. Par contre le code ci-dessous illustre l'utilisation d'une méthode d'extension :

Version avec méthode d'extension

```
class Program
{
 static void Main(string[] args)
 {
  string inputString = Console.ReadLine();

  // Tentative de conversion
  try
  {
   int result = inputString.ToInt32();
   Console.WriteLine("Conversion réussie !");
  }
  catch (System.FormatException)
  {
   Console.WriteLine("Conversion impossible !");
  }

  Console.Read();
 }
}

static class UtilityStringExtensionMethods
{
```

Version avec méthode d'extension

```
public static System.Int32 ToInt32(this string inputString)
{
 return System.Int32.Parse(inputString);
}
}
```

Comme vous pouvez le remarquer, j'ai pu appelé la méthode "ToInt32" directement sur mon type System.String. Cela est possible grâce au mot clé this suivi du type concerné en paramètre à la méthode ToInt32. Le premier paramètre d'une méthode d'extension représente donc le type qui est concerné et la variable est tout simplement l'instance qui a appelé notre méthode d'extension.

Cependant, les méthodes d'extension sont utilisables aussi avec des paramètres. Par exemple, on pourrait "améliorer" notre méthode d'extension pour qu'elle puisse forcer une conversion, c'est à dire que si une conversion du type System.String vers System.Int32 n'est pas possible, on ne lancera pas d'exception pour renvoyer tout simplement la valeur 0.

Version avec méthode d'extension et paramètre

```
class Program
{
 static void Main(string[] args)
 {
  string inputString = Console.ReadLine();

  int result = inputString.ToInt32(true);
  Console.WriteLine("La valeur convetie vaut '{0}'", result);

  Console.Read();
 }
}

static class UtilityStringExtensionMethods
{
 public static System.Int32 ToInt32(this string inputString, bool forceConversion)
 {
  int result = 0;
  if (System.Int32.TryParse(inputString, out result))
  {
   return result;
  }
  else
  {
   if (forceConversion)
   {
    return result;
   }
   else
   {
    throw new FormatException();
   }
  }

 }
}
```

Pour ceux qui ont déjà touché à C# 3.0, vous avez surement déjà vu une série de nouvelles méthodes lorsque vous manipulez des collections impplémentant l'interface System.Collections.Generic.IEnumerable<T>, exemple Where, Select, OrderBy, etc. Ce sont tout simplement des méthodes d'extensions appelées plus particulièrement des opérateurs de requête standard Linq (Pour en savoir plus sur les opérateurs de requêtes : FAQ Qu'est-ce qu'un opérateur de requête ?).

Pour utiliser les méthodes d'extension, il y a quelques contraintes à respecter :

- **Une méthode d'extension doit obligatoirement se trouver dans une méthode statique**
- **Une méthode d'extension doit obligatoirement être statique**
- **Une méthode d'extension ne sera jamais appelée si elle a la même signature qu'une méthode existante du type concerné**

lien : Page MSDN sur les méthodes d'extension
lien : Qu'est-ce qu'un opérateur de requête ?

Sommaire > Fondements de Linq

Qu'est-ce que Linq to Objects ?

Auteurs : Jérôme Lambert ,

Linq to Objects permet de requêter des collections d'objets en mémoire de manière simple et efficace. Là où vous utilisiez des boucles for/while/foreach pour trier, grouper ou tout autres actions plus ou moins compliquée, Linq va vous permettre d'écrire des requêtes concises et lisibles.

Comment réaliser ma première requête Linq to Objects ?

Auteurs : Jérôme Lambert ,

Pour notre première Linq, nous allons récuper la liste des processus actifs sur la machines triés par nom.

Avant toute chose, assurez-vous que vous avez bien installé Visual Studio 2008. Si vous n'avez pas de version officielle, vous télécharger une des versions Express de Visual Studio 2008 entièrement gratuite en vous rendant à l'adresse suivante : http://msdn.microsoft.com/fr-fr/express/aa975050.aspx

A présent, lancez votre version de Visual Studio 2008. Une fois l'application lancée, créez un projet de type console. Pour cela, cliquez sur le menu "File>New>Project".
Dans la boite de dialogue qui apparaît, sélectionnez "Windows" comme type de projet et "Application Console" comme modèle.
Assurez-vous qu la cible du Framework est bien la version 3.5 car sans ça, vous ne pourrez utiliser Linq.
Appelez votre projet "MonPremierProjetLinq" et validez en appyant sur le bouton "Ok".

Une fois le projet console créé, allez dans le fichier Program.cs et positionnez-vous dans la méthode "Main".
Comme annoncé précédemment, il nous faut récupérer la liste des processus actifs de la machine. Pour cela, le Framework .NET met à disposition la classe statique "Process" avec une méthode "GetProcesses" permettant de récupérer la liste des processus actifs d'une machine dans un tableau d'objets "System.Diagnostics.Process".
Nous allons donc écrire une requête Linq qui nous permet d'interroger la collection renvoyée par la méthode "GetProcesses" en triant par le nom du processus.

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

namespace MonPremierProjetLinq2
{
 class Program
 {
  static void Main(string[] args)
  {
   var query = from process in System.Diagnostics.Process.GetProcesses()
      orderby process.ProcessName ascending
      select process;

   foreach (var currentItem in query.ToList())
   {
    Console.WriteLine(currentItem.ProcessName);
   }

   Console.Read();
  }
 }
}
```

Comme vous pouvez le constater la syntaxe de Linq se rapproche énormément de la syntaxe SQL.

Appuyez sur F5 pour lancer votre application et vous verrez la liste des processus actifs sur votre machine triés par ordre alphabatique croissant.

Peut-on requêter n'importe quelle collection d'objets ?

Auteurs : Jérôme Lambert ,

Le but de Linq to Objects est de pouvoir interroger des collections d'objets en mémoire. Cependant, cela ne veut pas dire que vous pouvez interroger n'importe quel type de collection. Depuis le Framework .NET 3.5, Microsoft a introduit une interface IEnumerable<T> qui permettra d'exécuter des requêtes Linq to Objects sur toutes collections implémentant cette nouvelle interface. Heureusement pour nous, les listes et collections que vous connaissiez avec le Framework .NET 2.0 ont été mises à jour pour implémenter cette nouvelle interface.

Voici la liste des collections qui vous est possible d'utiliser avec Linq to Objects :

- **Array => IEnumerable<T>**
- **System.Collections.Generic.List<T> => IEnumerable<T>**
- **System.Collections.Generic.LinkedList<T> => IEnumerable<T>**
- **System.Collections.Generic.Queue<T> => IEnumerable<T>**
- **System.Collections.Generic.Stack<T> => IEnumerable<T>**
- **System.Collections.Generic.HashSet<T> => IEnumerable<T>**
- **System.Collections.ObjectModel.Collection<T> => IEnumerable<T>**
- **System.ComponentModel.BindingList<T> => IEnumerable<T>**
- **System.Collections.Generic.Dictionary<TKey, TValue> => IEnumerable<KeyValuePair<TKey, TValue>>**
- **System.Collections.Generic.SortedDictionary<TKey, TValue> => IEnumerable<KeyValuePair<TKey, TValue>>**
- **System.Collections.Generic.SortedList<TKey, TValue> => IEnumerable<KeyValuePair<TKey, TValue>>**
- **System.String => IEnumerable<char>**

Concrètement, vous pourrez aussi requêter vos propres collections à condition qu'elles implémentent l'interface IEnumerable<T>.

Cependant, il y a aussi les autres collections du Framework .NET qui sont non générique et donc implémentent uniquement l'interface IEnumerable.

Par exemple :

- **System.Collections.ArrayList**
- **System.Collections.HashTable**
- **...**

Pourtant, vous trouverez dans cette FAQ une alternative pour utiliser Linq to Objects avec ce type de collections.

lien : FAQ Comment exécuter une requête sur une collection non générique ?

Comment afficher le résultat d'une requête Linq dynamiquement ?

Auteurs : Jérôme Lambert ,

Object Dumper est une bibliothèque fournie par Microsoft permettant d'afficher les résultats de vos requêtes Linq dynamquement. Le but étant d'aller plus loin dans vos tests.

Pour en savoir plus sur cette biblothèque et la télécharger, je vous invite à vous rendre sur MSDN.

Comment exécuter une requête sur une collection non générique ?

Auteurs : Jérôme Lambert ,

Comme expliqué dans une précédente question/réponse, il est à priori possible d'interroger n'importe quel type de collections à condition qu'elle implémente l'interface IEnumerable<T>, ce qui n'est pas le cas des collections non génériques, tel que "System.Collections.ArrayList". Il semblerait donc qu'il n'est pas possible d'exécuter une requête Linq to Objects sur une collection de type ArrayList et pourtant, il y a une astuce.

Tout d'abord, il faut bien comprendre que Linq to Objects repose sur l'interface IEnumerable<T> car c'est grâce à cette interface que le compilateur va pouvoir déterminer le type d'objet contenu dans une collection. Or avec une collection de type ArrayList, il n'est pas possible de déterminer le type des objets contenus dans la collection. Pour résoudre ce problème, Microsoft a introduit une méthode d'extension appelée "Cast".

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

Comme vous pouvez le constater en regardant la définition de cette méthode, elle prend en paramètre une collection qui implémente l'interface IEnumerable et renvoie un objet de type IEnumerable<T>. Cette méthode est évidemment générique pour que vous puissiez spécifier explicitement le type d'objet contenu dans votre collection.

Si on considère que nous avons un ArrayList contenant des objets de type Process, nous allons pouvoir écrire le code suivant :

```
System.Collections.ArrayList myArrayList = new System.Collections.ArrayList();

// Ajout d'instances de type Process dans notre ArrayList
// ...

var query = from process in myArrayList.Cast<System.Diagnostics.Process>()
   select process;
```

Comment exécuter une requête sur une collection non générique contenant des objets de types différents ?

Auteurs : Jérôme Lambert ,

Lorsque vous avez une collection non générique, qui n'implémente donc pas l'interface IEnumerable<T>, et qui en plus contient des objets de type différents, il n'est pas possible d'utiliser la méthode d'extension Cast<T>. Alors comment

faire pour exécuter une requête Linq to Objects sur une collection de type ArrayList qui contient des objets de type Color et Point mélangés ?

Et bien tout simplement en utilisant une autre méthode d'extension nommée OfType :

```
public static IEnumerable TResult OfTypeTResult(this IEnumerable source);
```

Si on reprend notre ArrayList contenant des objets Color et Point, il va être possible d'exécuter une requête Linq pour manipuler par exemple les objets de type Color :

```
System.Collections.ArrayList myArrayList = new System.Collections.ArrayList();
myArrayList.Add(System.Drawing.Color.Red);
myArrayList.Add(new System.Drawing.Point(0, 0));
myArrayList.Add(System.Drawing.Color.Green);
myArrayList.Add(new System.Drawing.Point(10, 20));
myArrayList.Add(System.Drawing.Color.Blue);
myArrayList.Add(new System.Drawing.Point(20, 30));

var query = from color in myArrayList.OfTypeSystem.Drawing.Color()
   select color;

foreach (var currentResult in query)
 Console.WriteLine(currentResult.Name);
```

Vous remarquerez qu'à la compilation le code est accepté et mieux encore à l'exécution, aucun plantage de l'application. On retrouve d'alleur bien le résultat attendu :
Red
Green
Blue

Qu'est-ce qu'une séquence ?

Auteurs : Jérôme Lambert ,

Une séquence désigne tout objet dont le type implémente l'interface IEnumerable<T> ou l'interface IQueryable<T>.

Qu'est ce que l'exécution différée de requête ?

Auteurs : Jérôme Lambert ,

Soit l'exemple suivant :

```
class Program
{
 static void Main(string[] args)
 {
  List<string> aMembres= new List<string>()
       {
        "Jérôme",
        "Louis-Guillaume",
        "Vincent",
        "Benjamin"
       };

  var query = from membre in aMembres
     orderby membre ascending
     select membre;
```

```
    foreach (string currentMembre in query)
    {
     Console.WriteLine(currentMembre);
    }

    Console.Read();
   }
  }
```

A l'exécution, on obtient le résultat suivant, c'est à dire liste des membres triés par odre alphabétique :

Benjamin
Jérôme
Louis-Guillaume
Vincent

Maintenant, imaginons que nous désirons ajouter un membre à notre liste juste après avoir construit notre requête. La logique voudrait que ce nouveau membre ne soit pas pris en compte si on repartcourt les éléments de notre objet "query". Le code devient donc :

```
  class Program
  {
   static void Main(string[] args)
   {
    List<string> aMembres= new List<string>()
         {
          "Jérôme",
          "Louis-Guillaume",
          "Vincent",
          "Benjamin"
         };

    var query = from membre in aMembres
       orderby membre ascending
       select membre;

    Console.WriteLine("Résultat query Avant");
    Console.WriteLine("**********");
    foreach (string currentMembre in query)
    {
     Console.WriteLine(currentMembre);
    }

    aMembres.Add("Thomas");

    Console.WriteLine("\nRésultat query Après");
    Console.WriteLine("**********");
    foreach (string currentMembre in query)
    {
     Console.WriteLine(currentMembre);
    }

    Console.Read();
   }
  }
```

Et contre toute attente, on retrouve bien notre membre "Thomas" !

Résultat query Avant
****
Benjamin

Jérôme
Louis-Guillaume
Vincent
Résultat query Après
****
Benjamin
Jérôme
Louis-Guillaume
Thomas
Vincent

C'est ça l'exécution différée de requête (ou "Deferred Query Execution" en anglais) ! Il est important de savoir que notre objet "query" ne contient absolument pas le résultat de notre requête mais plutôt l'expression permettant de récupérer un résultat une fois qu'on en aura besoin.

Ce mécanisme permet d'éviter de consommer des ressources quand ceci est inutile. Par exemple, on pourrait construire une requête permettant de récupérer la liste des membres et plus tard, demander le premier de ces éléments. Dans ce cas, on évite de récupérer tous les membres pour ne prendre que le premier.

Qu'est-ce que la réutilisation de requête ?

Auteurs : Jérôme Lambert ,

Avant de commencer la lecture de cette réponse, allez consulter "FAQ Qu'est ce que l'exécution différée de requête ?"

La réutilisation d'une requête (ou "Reuse Query" en anglais) est le fait de réutiliser une requête différente fois dans une application. Cependant, il n'est pas impossible qu'une même requête donne des résultats différents entre deux exécution. En voici un exemple :

```
class Program
{
 static void Main(string[] args)
 {
  List[string] aMembres= new List[string]()
       {
        "Jérôme",
        "Louis-Guillaume",
        "Vincent",
        "Benjamin"
       };

  var query = from membre in aMembres
     orderby membre ascending
     select membre;

  Console.WriteLine("Résultat query Avant");
  Console.WriteLine("**********");
  foreach (string currentMembre in query)
  {
   Console.WriteLine(currentMembre);
  }

  // Mise à  jour des membres
  aMembres[0] = aMembres[0] + " Lambert";
  aMembres[1] = aMembres[1] + " Morand";
  aMembres[2] = aMembres[2] + " Lainé";
  aMembres[3] = aMembres[3] + " Broux";
```

```
    Console.WriteLine("\nRésultat query Après");
    Console.WriteLine("********************");
    foreach (string currentMembre in query)
    {
     Console.WriteLine(currentMembre);
    }

    Console.Read();
   }
  }
```

Ce qui donne le résultat suivant :

Résultat query Avant
****
Benjamin
Jérôme
Louis-Guillaume
Vincent
Résultat query Après
****
Benjamin Broux
Jérôme Lambert
Louis-Guillaume Morand
Vincent Lainé

lien : Qu'est ce que l'exécution différée de requête ?

Qu'est-ce qu'un opérateur de requête ?

Auteurs : Jérôme Lambert ,

Les opérateurs de requêtes (ou "Query Operators" en anglais) sont un ensemble de méthodes d'extension qui rendent possible les possibilités de Linq. Ainsi, vous bénéficiez de requêtes permettant le trie, le filtrage, l'aggrégation, la concaténation, la projection, etc.

Voici un exemple de requête Linq utilisant trois opérateurs de requête :

```
var query = System.Diagnostics.Process.GetProcesses()
   .Where(p => p.ProcessName.ToLower().StartsWith("a"))
   .OrderBy(p => p.ProcessName)
   .Select(p => p.ProcessName);
```

Le premier opérateur de requête est Where qui permet de filtrer uniquement les processus dont le nom commence par 'a'.
Le second opérateur de requête est OrderBy qui permet de trier la séquence par nom de processus ascendant.
Le troisième et dernier opérateur est Select qui permet de sélectionner pour chaque élément de la séquence résultante uniquement le nom du process.

Ces méthodes d'extensions se trouvent dans la classe System.Linq.Enumerable.

lien : Lien MSDN vers System.Linq.Enumerable

Quels sont les opérateurs de requête standards disponibles ?

Auteurs : Jérôme Lambert ,

Type	Opérateur de requête
Tri des données	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Opérations ensemblistes	Distinct, Except, Intersect, Union
Filtrage des données	OfType, Where
Opérateur de quantificateur	All, Any, Contains
Opérateurs de projection	Select, SelectMany
Partitionnement des données	Skip, SkipWhile, Take, TakeWhile
Opérations de jointure	Join, GroupJoin
Regroupement de données	GroupBy, ToLookup
Opérateurs de génération	DefaultIfEmpty, Empty, Range, Repeat
Opérateurs d'égalité	SequenceEqual
Opérateurs d'élément	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Conversion de types	AsEnumerable, AsQueryable, Cast, OfType, ToArray,

de données	ToDictionary, ToList, ToLookup
Opérations de concaténation	Concat
Opérations d'agrégation	Aggregate, Average, Count, LongCount, Max, Min, Sum

Qu'est-ce qu'une expression de requête (Query Expression) ?

Auteurs : Jérôme Lambert ,

Les expressions de requêtes (ou "Query Expressions" en anglais) représentent une réelle extension du langage. Comme vous devez le savoir, les opérateurs standards de requête (FAQ Qu'est-ce qu'un opérateur de requête ?) sont un ensemble de méthodes statiques introduites avec C# 3.0 et VB.NET 9.0 pour permettre de construire des requêtes Linq sur des sources de données.

Ainsi, une requête Linq avec les opérateurs standars de requête ressemblera à ceci :

```
var query = System.Diagnostics.Process.GetProcesses()
   .Where(p => p.ProcessName.StartsWith("A"))
   .OrderBy(p => p.ProcessName)
   .Select(p => p);
```

La lecture de cette requête s'avère encore aisée grâce à sa simplicité mais imaginez une requête avec des jointures sur différentes collections, des groupements... Ce serait plus facile à écrire mais surtout à lire en langage SQL ! Et bien les expressions de requête c'est un peu ça car cela va vous permettre d'écrire vos requête Linq un peu comme vous écrieriez vos requêtes SQL.

L'exemple précédent peut donc s'écrire de la manière suivante en expression de requête :

```
var query = from process in System.Diagnostics.Process.GetProcesses()
   where process.ProcessName.StartsWith("A")
   orderby process.ProcessName ascending
   select process;
```

Comme vous pouvez le remarquer, la lecture se fait presque comme si on lisait une requête SQL et lorsque je vous parlais tout au début que les expressions de requête étaient une extension du langage, on est en plein de temps : from, where, orderby, ascending, select, ... Des nouveaux mots clés ont été introduits pour permettre cette nouvelle syntaxe. Lors de la compilation, le compilateur s'occupera de retranscrire vos expressions de requête en opérateurs de requête.

Cependant, tous les opérateurs standards de requête n'ont pas nécessairement leur correspondance en tant qu'expression de requête et ça dépend aussi du langage. Par exemple, l'opérateur "Take" n'existe pas en C# en tant qu'expression de requête, alors qu'en VB.NET il existe.

lien : FAQ Qu'est-ce qu'un opérateur de requête ?

lien : Est-ce que tous les opérateurs standards de requête ont leur équivalence en tant qu'expression de requête ?

Est-ce que tous les opérateurs standards de requête ont leur équivalence en tant qu'expression de requête ?

Auteurs : Jérôme Lambert ,

C# 3.0 et VB.NET 9.0 étant développés par deux équipes différentes, tous les opérateurs standards de requête n'ont pas nécessairement leur équivalence dans les deux langages en tant qu'expression de requête. Vous trouverez ci-dessous un tableau répertoriant les opérateurs standards avec une indication s'ils sont supportés en tant qu'expression de requête.

Opérateur standard	Equivalence en expression de requête avec C#	Equivalence en expression de requête avec VB.NET
All	Non	Oui
Any	Non	Oui
Average	Non	Oui
Cast	Oui	Oui
Count	Non	Oui
Distinct	Non	Oui
GroupBy	Oui	Oui
GroupJoin	Oui	Oui
Join	Oui	Oui
LongCount	Non	Oui
Max	Non	Oui
Min	Non	Oui
OrderBy	Oui	Oui
OrderByDescending	Oui	Oui
Select	Oui	Oui
SelectMany	Oui	Oui
Skip	Non	Oui
SkipWhile	Non	Oui
Sum	Non	Oui
Take	Non	Oui
TakeWhile	Non	Oui
ThenBy	Oui	Oui
ThenByDescending	Oui	Oui
Where	Oui	Oui

Comment utiliser Linq to Objects dans vos projets ?

Auteurs : Jérôme Lambert ,

1. **Assurez-vous que votre projet cible bien le Framework .NET 3.5**
2. **Ajoutez à votre projet la référence à l'assemblée *System.Core.dll*, si ce n'est déjà fait**

3 Aujoutez *using System.Linq;* en début de fichier de votre classe

Qu'est-ce que la "sugar syntax" en Linq ?

Auteurs : Jérôme Lambert ,

Avec Linq, vous pouvez écrire vos requêtes à l'aide des opérateurs de requêtes disponibles grâce aux méthodes d'extensions de *System.Linq.Enumerable*. Ainsi, si on désire récupérer la liste des processus actifs sur votre ordinateur et commençant par la lettre 'A', il suffit d'écrire :

```
var query = System.Diagnostics.Process.GetProcesses().Where(p =>
 p.ProcessName.ToLower().StartsWith("a")).Select(p => p);
```

Cependant pour des cas complexes, cette syntaxe n'est pas toujours facile à écrire et encore moins à lire. C'est là qu'interviennent les expressions de requêtes. C'est une autre façon d'écrire une requête Linq bien plus proche de la syntaxe SQL. On appelle aussi cette façon d'écrire la "sugar syntax".

Ainsi, la requête précédente s'écrit en version expression de requête :

```
var query = from process in System.Diagnostics.Process.GetProcesses()
   where process.ProcessName.ToLower().StartsWith("a")
   select process;
```

Il est important de ne pas oublier que les opérateurs de requêtes n'ont pas tous leur correspondance en expression de requête.

Comment grouper des éléments sur base de critères multiples ?

Auteurs : Jérôme Lambert ,

Il vous est possible de faire un groupement sur plusieurs champs en passant par un objet anonyme de la manière suivante :

```
static void Main(string[] args)
{
 var membres = GetMembres();

 var query = from membre in membres
    group membre by new { membre.Sexe, membre.Age }
     into grouping
     select new
     {
      Sexe = grouping.Key.Sexe,
      Age = grouping.Key.Age,
      Membres = grouping
     };

 foreach (var currentResult in query)
 {
  Console.WriteLine("Membres dont sexe est '{0}' et âge = '{1}'", currentResult.Sexe,
currentResult.Age);
  foreach (var currentMembre in currentResult.Membres)
  {
   Console.WriteLine(currentMembre.Nom);
  }
  Console.WriteLine();
 }
```

```
    Console.Read();
  }
```

Ce qui donnera le résultat suivant :

Membres dont sexe est 'M' et âge = '30'
Marc Lussac
Jérôme Lambert
nico-pyright(c)
Thomas Lebrun
tomlev
Membres dont sexe est 'M' et âge = '20'
Yogui
Louis-Guillaume Morand
Tofalu
Membres dont sexe est 'F' et âge = '20'
Aspic
Dev01
The_badger_man
Membres dont sexe est 'F' et âge = '30'
Skyounet

Les sexes et âges affichés dans le résultat de la requête ne reflètent en aucun cas la réalité, excepté pour Skyounet :)

Comment rendre paramétrable un critère de filtre ?

Auteurs : Jérôme Lambert ,

Si par exemple, nous désirons récupérer tous les membres du sexe masculin de developpez.com, nous pouvons écrire la requête suivante :

```
var query = from membre in membres
    where membre.Sexe == 'N'
    select membre;
```

Seul soucis, c'est que si nous désirons récupérer la liste des membres du sexe féminin, à part récrire la requête, celle di-dessus est inutilisable. Cependant, rien empêche d'utiliser des variables dans nos requêtes Linq, ainsi, on va pouvoir encapsuler notre requête Linq dans une méthode qui reçoit en paramètre le sexe afin de le comparer dans la requête Linq.

```
static void Main(string[] args)
{
 var MembreHommes = GetMembres('H');
 var MembreFemmes = GetMembres('F');
}

static List<Membreégt; GetMembres(char sexe)
{
 var membres = GetMembres();

 var query = from membre in membres
    where membre.Sexe == sexe
    select membre;
```

```
    return query.ToList();
  }
```