C# 3 et LINQ

Arrivée en version Alpha il y a quelques temps, la troisième version du langage C# réserve des nouveautés qui s'avère fort pratiques pour les développeurs.

Accompagné du projet **LINQ** (utilisable en C#, VB.NET et les autres langages .NET), qui concerne tout ce qui est accès aux données, cette nouvelle version du langage créé par Microsoft à tout pour réussir une avancée majeure dans le milieu des développeurs.

Cet article a pour but de vous permettre de découvrir ce nouveau langage innovateur, et ce nouveau projet dont certaines parties risquent, à terme, d'intégrer **ADO.NET 3.0**

Cet article s'inspire, pour beaucoup, des "*labs*" qui sont fournis avec les CTP de LINQ.

Sommaire

Introduction

Depuis les premières versions Alpha, téléchargeables depuis la **PDC** (*Professional Developer Conference*) 2005, la nouvelle version du langage C#, le **C# 3**, n'a cessé de subir des modifications et des améliorations pour tenter d'intégrer de nouvelles fonctionnalités à la dernière version en date, le C# 2.

LINQ, de son vrai nom "***Language Integrated Query***", représente une évolution majeure de l'accès aux données dans le Framework .NET.

Il s'agit d'un projet de requêtage de données, se divisant en plusieurs grandes parties :

LINQ To ADO.NET, qui inclut:
- LINQ To DataSet
- LINQ To Entities
- LINQ To SQL (anciennement connu sous le nom de DLINQ, et qui permet d'exécuter des requêtes sur une base de données, de faire du mapping objet-relationnel)

Le support de LINQ pour les autres type de données:
- LINQ To XML (anciennement connu sous le nom XLINQ, qui permet d'exécuter des requêtes sur des documents XML)
- LINQ To Objects

Pour bien comprendre cette division, je vous conseille de regarder le message de **Soma Somasegar** : http://blogs.msdn.com/somasegar/archive/2006/06/21/641795.aspx

Au cours de cet article, je vais donc tenter de vous parler, au mieux, de toutes ces nouveautés que nous réserve Microsoft.

Note : Tous les exemples de cet article se basent sur la **CTP** (*Community Technology Preview*) de **Mai 2006** de LINQ, que vous pouvez télécharger ici :

http://www.microsoft.com/downloads/details.aspx?familyid=1e902c21-340c-4d13-9f04-70eb5e3dceea&displaylang=en

Le langage C# 3

Développé par Microsoft, et en particulier par **Anders Hejlsberg** (déjà inventeur du Turbo Pascal et du langage Delphi), le **C#** (prononcé "*Si Sharp*") est un langage en constante évolution qui ne cesse de réserver des surprises.

Dans sa troisième version (actuellement en Alpha), le C# possède un bon nombre de fonctionnalités que beaucoup de développeurs C# auraient aimé avoir dans la version actuelle du langage.

Plutôt que de discuter, je vous propose de rentrer directement dans le vif du sujet.

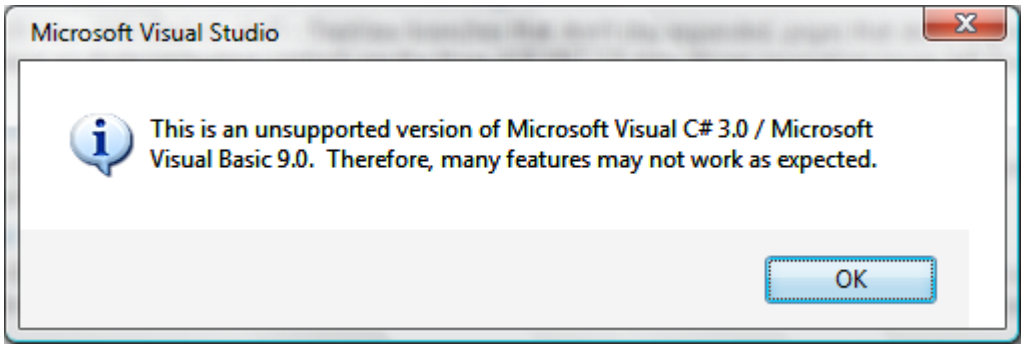
Utilisation de variables locales typées implicitement

L'utilisation de **variables locales typées implicitement** est une des nouvelles fonctionnalités offertes par C# 3, qui permet aux développeurs de déclarer des variables dont le type n'est pas indiqué explicitement. En effet, le type de ces variables est découvert, par le compilateur, en fonction de la partie droite de l'affectation.

Cette nouvelle fonctionnalité permet de libérer les développeurs de la contrainte d'avoir à spécifier, à chaque fois, le type de leurs variables ou de leurs requête.

Voyons cela à travers un exemple :

Dans Visual Studio 2005, pour créer un projet C# 3, il vous faut sélectionner "*New Project*" => "***LINQ Preview***" et choisir le type de projet que vous voulez créer. Vous allez avoir la boite de dialogue suivante, qui vous explique que C# 3 / VB 9 est un langage encore en Beta, et que certaines fonctionnalités ne fonctionneront peut-être pas correctement :



Cliquez alors sur "OK" pour pouvoir commencer à développer.

Pour comprendre ce nouveau concept de variables typées implicitement, commencez par regarder ce bout de code :

```
int x = 5;

string s = "Article C# 3 / LINQ";

int [] tab = new int [] { 1, 2, 3, 4, 5 };

Console.WriteLine("Valeur de x: {0}", x);
Console.WriteLine("Valeur de s: {0}", s);
Console.WriteLine("Liste des valeurs de tab:");

foreach(int i in tab)
{
	Console.WriteLine(i);
}
```

Jusque ici, rien de bien extraordinaire : on commence par déclarer un entier puis une chaîne de caractères et enfin, un tableau d'entier.

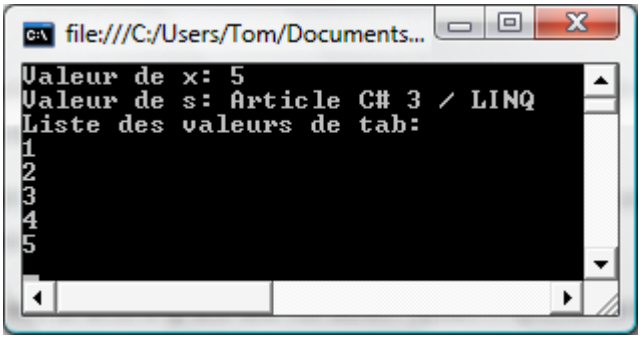
Maintenant, que ce passerait-il si le type de vos variables n'était pas connu ? Il vous serait tout simplement impossible de les déclarer et donc de les utiliser.

C'est justement ce point qui est amélioré grâce aux variables typées implicitement. La preuve par le code :

```
var x = 5;

var s = "Article C# 3 / LINQ";

var tab = new int [] { 1, 2, 3, 4, 5 };

Console.WriteLine("Valeur de x: {0}", x);
Console.WriteLine("Valeur de s: {0}", s);
Console.WriteLine("Liste des valeurs de tab:");

foreach(int i in tab)
{
	Console.WriteLine(i);
}
```

Dans ce code, nous déclarons des variables mais sans en préciser le type. A la place, nous utilisons le mot clé **var**. A la compilation, le type de vos variables sera déduit automatiquement

Le résultat est donc sans surprises :



Bien entendu, vous pouvez utiliser **var** avec des variables simples mais également avec des collections :

```
var list = new Dictionary<string, int>();

list.Add("PremiereCle", 1);
list.Add("PremiereCle", 2);
list.Add("PremiereCle", 1);

list.Add("DeuxiemeCle", 1);
list.Add("DeuxiemeCle", 3);
list.Add("DeuxiemeCle", “);
```

Il y a tout de même des restrictions que vous devez prendre en compte lorsque vous déclarez une variable avec le mot clé **var**.

Par exemple, le type de la variable étant déduit de l'initialiseur, une déclaration implicitement typée doit **obligatoirement** avoir un initialiseur. Le bout de code suivant provoquerait donc une erreur :

```
var list;
```

Et devrait être remplacé par quelque chose comme:

```
var list = new List<int>();
```

Voyons à présent une autre des nouvelles fonctionnalités offertes par C# 3: **les méthodes d'extension.**

Les méthodes d'extension

Les **méthodes d'extensions** vous permettent d'étendre les fonctionnalités offertes par un type, en définissant de nouvelles méthodes qui seront invoquées en utilisant la syntaxe normale d'appel d'une méthode.

Les méthodes d'extension sont des méthodes **statiques**, qui sont déclarées en utilisant le mot clé **this** comme modificateur de portée, pour le premier paramètre de la méthode.

Pour bien comprendre ce nouveau concept, on va repartir d'un nouvel exemple :

Créer un nouveau projet de type "LINQ Preview" dans Visual Studio et ajouter la classe suivante :

```
public static class ToUpperCaseClass
{
      public static string ToUpperCase(string s)
      {
            return s.ToUpper();
      }
}
```

Comme vous pouvez le constatez, cette classe n'a rien d'extraordinaire: elle contient une méthode statique qui prend en paramètre une chaîne de caractères et qui retourne cette même chaîne, après l'avoir passée en majuscule.

Pour utiliser cette classe/méthode, rien de bien compliqué :

```
string s = "Article C# 3 et LINQ";
Console.WriteLine(ToUpperCaseClass.ToUpperCase(s));
```

Voyons maintenant comment nous pourrions utiliser cette méthode comme une méthode d'extension.

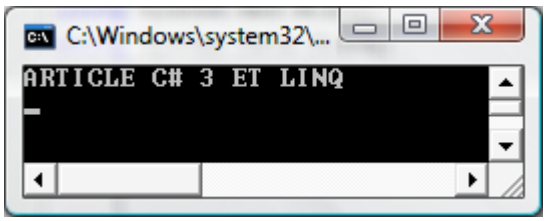
Commencer par modifier le code de votre méthode :

```
public static string ToUpperCase(this string s)
```

Et enfin, l'appel de cette méthode :

```
Console.WriteLine(s.ToUpperCase());
```

Le résultat est, comme vous pouvez l'imaginer, identique dans les deux cas:



Dans la première version, vous avez passé votre variable de type chaîne de caractères en paramètre à votre méthode statiques. Dans la seconde version, vous avez appelé directement cette méthode sur votre objet.

Attention: grâce à cette méthode d'extension, vous avez réussi à rajouter une méthode à un objet particulier et non à l'une des classes de base de Framework .NET !

Si on regarde le code qui a été générée pour écrire cette méthode d'extension, on voit bien qu'à la compilation, l'appel de la méthode a simplement été remplacé par l'appel à une méthode statique:

```
string text1 = "Article C# 3 et LINQ";

Console.WriteLine(ToUpperCaseClass.ToUpperCase(text1));
```

Les expressions lambdas

Pour comprendre le concept des expressions lambda, il faut se rappeler d'une des fonctionnalités offertes par C# 2 : **les méthodes anonymes**. Ces méthodes permettaient aux développeurs d'écrire des blocs de code en ligne, lorsque des délégués étaient nécessaires.

Par exemple, on pouvait tout à fait écrire quelque chose comme ceci :

```
List<int> list = new List<int>(new int [] { -1, 2, -5, 45, 5 });

List<int>positiveNumbers = list.FindAll(delegate(int i) { return i > 0;});
```

Les expressions lambda représentent en fait un moyen plus simple d'écrire les méthodes anonymes.

Une expression lambda est composée de trois parties :

Une liste de paramètre
Le signe **=>**
Une expression

Voici un exemple d'expression lambda :

```
(int i) => i % 2 == 0
```

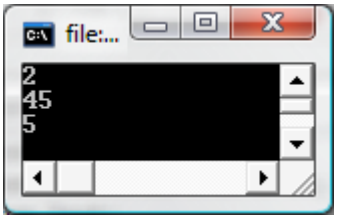
Au niveau de la liste de paramètres, il y a deux choses à savoir :

Les paramètres peuvent être typés explicitement ou implicitement. Dans le second cas, le type est déduit de l'expression qui constitue votre expression.
S'il n'y a qu'un seul paramètre typé implicitement, les parenthèses sont facultatives

Voyons un peu comment nous pourrions utiliser cela dans notre code :

```
List<int> list = new List<int>(new int [] { -1, 2, -5, 45, 5 });

var positiveNumbers = list.FindAll((int i) => i > 0);

foreach(int positiveNumber in positiveNumbers)
{
	Console.WriteLine(positiveNumber);
}
```

Le code de cette application, utilisant une expression lambda, fournit le même résultat que la version utilisant les méthodes anonymes : il est simplement plus clair à lire.



Bien entendu, rien ne vous empêche d'utiliser plusieurs paramètres dans votre expression lambda.

Commencez par créer une petite méthode d'extension que nous utiliserons plus tard. N'oubliez qu'une méthode d'extension est une méthode statiques. Pour bien faire, créer cette méthode dans une nouvelle classe, afin d'organiser au mieux votre code.

```
public static class FilterClass
{
	public delegate bool KeyValueFilter<K, V>(K key, V value);

	public static Dictionary<K, V> FilterBy<K, V>(this Dictionary<K, V>items,
KeyValueFilter<K, V> filter)
	{
		var result = new Dictionary<K, V>();

		foreach (KeyValuePair<K, V> element in items)
		{
			if (filter(element.Key, element.Value))
				result.Add(element.Key, element.Value);
		}

		return result;
	}
}
```

Ensuite, il ne vous reste plus qu'à construire votre expression lambda comme précédemment mais en modifiant votre expression pour travailler avec plusieurs paramètres :

```
// Expression lambda avec plusieurs paramètres
var listLanguages = new Dictionary<string, int>();
listLanguages.Add("C#", 5);
listLanguages.Add("VB", 3);
listLanguages.Add("Java", -5);

var bestLanguages = listLanguages.FilterBy((string name, int score) => name.Length
== 2 && score > 0);

foreach(KeyValuePair<string, int>language in bestLanguages)
{
	Console.WriteLine(language.Key);
}
```

Ici, on travaille bien sur une expression lambda qui possède plusieurs paramètres.

Vous pourriez, et vous auriez tout à fait raison, trouver tout cela un petit peu magique.

Comment se fait-il que des méthodes telles que **FindAll** ou **FilterBy** accepte en paramètre une expression lambda, alors que normalement, il devrait y avoir un délégué ?

La réponse est simple: c'est grâce au **compilateur**. En effet, c'est lui qui sait de quels types doivent être les paramètres des méthodes. Il est donc en mesure de déterminer si, oui ou non, les expressions lambda correspondent au type attendu.

Donc finalement, rien de magique ici: on a simplement affaire à un compilateur intelligent ☺

Vous pourriez également vous demander ceci: "Pourquoi les méthodes d'extension et les expressions lambda existent ?"

En effet, qu'est-ce qui les rend si importantes pour les développeurs, sachant que les techniques employées jusque là fonctionnaient très bien ?

En ce qui concerne les méthodes d'extension, la réponse est simple: il ne s'agit que d'une simple aide au développeur, de la part du compilateur, afin de lui faciliter la vie dans l'écriture de son programme et en particulier des requêtes LINQ (que nous verrons par la suite).

Pour les expressions lambda, le but est de fournir, aux développeurs, un moyen leur permettant de passer du code en paramètres de leurs méthodes.

Au jour d'aujourd'hui, vous pouvez tout à fait réaliser cela avec les **delegates**: vous déclarez un delegate prenant comme paramètre un pointeur vers une fonction. Lors de l'appel d'une de vos méthodes, vous passez en paramètre ce delegate. Au final, à l'appel de la méthode, le contenu de la méthode pointée par le delegate sera exécuté.

Et bien les expressions lambda ne sont qu'une autre manière de faire cela: elles vous permettent de passer, en paramètres de vos méthodes, des méthodes anonymes dont l'écriture a été simplifiée.

Pour bien assimiler ce concept d'expressions lambda, je vous conseille le webcast réaliser par **Mitsu Furuta**:

http://www.microsoft.com/france/events/event.aspx?EventID=1032300129

Les initialiseurs d'objets

Déclarer une variable en C# se fait en deux étapes :

Déclaration de la variable
Initialisation de cette variable

C# 3 propose un nouveau concept qui vous permet de déclarer et d'instancier, en même temps, un membre (propriété ou champ) d'une classe.

Pour cela, il faut passer par une forme particulière du constructeur, et il faut que les propriétés que vous vouliez initialiser soient déclarées comme **publiques**.

Imaginez par exemple la classe suivante :

```
public class User
{
	private string m_LastName;

	public string LastName
	{
		get { return m_LastName; }
		set { m_LastName = value; }
	}

	private string m_FirstName;

	public string FirstName
	{
		get { return m_FirstName; }
		set { m_FirstName = value; }
	}

	private int m_Age;

	public int Age
	{
		get { return m_Age; }
		set { m_Age = value; }
	}

	public User(string firstname, string lastname, int age)
	{
		this.m_FirstName = firstname;
		this.m_LastName= lastname;
		this.m_Age = age;
	}
}
```

Que l'on soit en C# 1 ou 2, pour instancier des objets de ce type, on faisait simplement :

```
User u1 = new User();
u1.FirstName = "Thomas";
u1.LastName = "LEBRUN";
u1.Age = 24;

// Ou alors
User u2 = new User("Thomas", "LEBRUN", 24);
```

La deuxième méthode est déjà plus aisée mais pourrait encore être améliorée, comme nous allons le voir avec ce que permet C# 3 et les initialiseurs d'objets :

```
// Version C# 3
var u3 = new User { FirstName = "Thomas", LastName = "LEBRUN", Age = 24 };

Console.WriteLine("Je m'appelle {0} {1} et j'ai {2} ans", u3.FirstName,
u3.LastName, u3.Age);

Console.ReadLine();
```

L'instanciation des objets est alors plus simple et plus rapide, mais surtout grâce à elle, vous n'êtes pas obligé de déclarer un constructeur spécifique dans votre classe. En effet, il vous suffit de déclarer des propriétés publiques et de faire appel à leur nom, lors de l'instanciation.

Vous n'êtes pas obligé de déclarer tous les champs, lors de l'initialisation. Dans ce cas, des valeurs par défaut seront affectées aux champs non déclarés.

C# 3 permet, une nouvelle fois, de faire la même chose mais en travaillant avec des collections, comme vous le montre cet exemple :

```
// Avant
List<int> MyListOfInteger = new List<int>();
MyListOfInteger.Add(1);
MyListOfInteger.Add(2);
MyListOfInteger.Add(3);
MyListOfInteger.Add(4);
MyListOfInteger.Add(5);

// Avec C# 3
List<int> MySecondListOfInteger = new List<int>{6, 7, 8, 9, 10 };

foreach(int i in MySecondListOfInteger)
{
      Console.WriteLine(i);
}
```

Les types anonymes

C# 3 propose la possibilité de créer des types anonymes et de retourner des instances de ce type, au moyen du mot clé **new**.

```
var MyClass = new { PremierChamp = 1, SecondChamp = "Ma Chaîne", TroisiemeChamp =
new List<int>(new int [] { 1, 2 , 3, 4, 5 }) };
```

Grâce à ce code, nous créons un objet possédant trois propriétés, auxquelles nous pouvons tout à fait accéder :

```
Console.WriteLine(MyClass.PremierChamp);
Console.WriteLine(MyClass.SecondChamp);

foreach(int i in MyClass.TroisiemeChamp)
{
      Console.WriteLine(i);
}
```

Le résultat est, tout simplement:

Dans ce premier exemple, les noms des membres de la classe anonyme sont spécifiés explicitement.

Mais vous pouvez tout à fait omettre le nom de ces membres. Dans ce cas, le nom des membres généré est le même que les membres utilisé pour généré l'objet. Voici un exemple :

```
var MySecondClass = new { MyClass.PremierChamp, MyClass.SecondChamp };

Console.WriteLine(MySecondClass.PremierChamp);
Console.WriteLine(MySecondClass.SecondChamp);
```

Le projet LINQ

LINQ

LINQ (*Language Integrated Query*) est un des nouveaux projets accompagnant C# 3.

Auparavant, les développeurs étaient amenés à utiliser deux ou plusieurs langages en même temps (C#, SQL, etc.…). Malheureusement, cette approche avait des inconvénients :

Le compilateur ne vérifiait pas le contenu de ce qui était entre double quotes
Il n'y avait pas de vérification de type sur les valeurs retournées
Etc.….

Nous allons donc voir, au travers de cette partie, quels sont les moyens mis en œuvre par LINQ pour palier à ces inconvénients. A noter qu'ici nous parlerons de LINQ en faisant référence à **LINQ To Objects**.

Interroger une liste générique

Avant de rentrer trop dans les détails, commençons par quelque chose de simple et voyons comment LINQ peut nous permettre d'interroger une liste générique autrement dit, une liste d'entiers, de chaîne de caractères, de double, etc.….

Voyons un exemple :

```
int [] tab = new int [] { 1, 2, 3, 4, 5, 6 };

IEnumerable<int> paire = from number in tab
                         where number % 2 == 0
                         select number;

foreach(int i in paire)
{
      Console.WriteLine(i);
}
```

Cet exemple est relativement simple à comprendre: nous déclarons un tableau d'entier, puis nous utilisons LINQ pour interroger ce tableau et ne renvoyer que les nombres pairs.

Si vous prêtez bien attention au code, vous remarquerez que LINQ utilise une syntaxe similaire à celle que l'on retrouve dans le langage SQL : nous utilisons un **from**, un **where** et un **select**. Bien entendu, il existe un nombre important d'opérateurs standards que nos verrons par la suite.

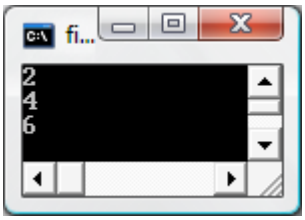
A la différence du SQL, DLINQ place l'instruction **from** avant l'instruction **select**. Vous pourriez tout à fait trouver cela quelque peu surprenant mais sachez que ce choix a été fait par Microsoft après de longues discussions et que l'atout majeur de cette syntaxe repose dans le fait que de cette façon, l'IntelliSense de Visual Studio est complètement fonctionnelle.

Dans l'exemple précédent, nous interrogeons un tableau d'entier : nous savons donc qu'à la sortie, nous allons récupérer un **System.Collections.Generic.IEnumerable<int>**. Mais beaucoup de requêtes retournent un type complexe que vous ne connaissez pas forcément. Dans ce cas, le plus simple est de passer par l'une des fonctionnalités offertes par C# 3 : l'utilisation de variables typées implicitement.

Ainsi, l'exemple précédent aurait très bien pu s'écrire :

```
int [] tab = new int [] { 1, 2, 3, 4, 5, 6 };

var paire = from number in tab
            where number % 2 == 0
            select number;

foreach(int i in paire)
{
      Console.WriteLine(i);
}
```

Et le résultat aurait toujours été le même :



Bien entendu, ce qui fonctionne avec les types simples fonctionne également avec les types complexes. Voyons par exemple comment appliquer ce que nous venons de voir à une classe nommée **User** (que nous avons déjà utilisée auparavant):

Commencez par créer une liste de Users que nous allons interroger par la suite :

```
var people = new List<User>() {
      new User { LastName = "LEBRUN", FirstName = "Thomas", Age = 24 },
      new User { LastName = "LAMARCHE", FirstName = "Patrice", Age = 26 },
      new User { LastName = "Inconnu" }
};
```

A présent, nous allons utiliser la même syntaxe que précédemment pour interroger cette liste et faire ressortir des résultats en fonction des conditions que l'on utilise :

```
var persons = from p in people
              where p.Age > 25
              select p;

foreach(var person in persons)
{
      Console.WriteLine("{0} {1} est agé de {2} ans: qu'il est vieux (it's a joke
of course :)", person.FirstName, person.LastName, person.Age);
}
```

```
var persons = from p in people
              where p.Age > 20 && p.LastName.Length > 5
              select p;
```

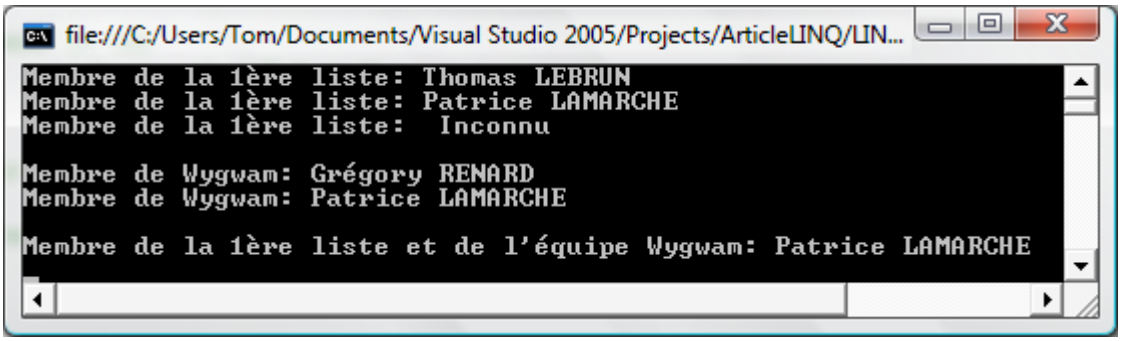
Bien sur, une des fonctionnalités bien pratiques de LINQ est de pouvoir faire des JOIN :

```
var people = new List<User>() {
      new User { LastName = "LEBRUN", FirstName = "Thomas", Age = 24 },
      new User { LastName = "LAMARCHE", FirstName = "Patrice", Age = 26 },
      new User { LastName = "Inconnu" }
};

var WygwamMembers = new List<User>() {
      new User { LastName = "RENARD", FirstName = "Grégory" },
      new User { LastName = "LAMARCHE", FirstName = "Patrice", Age = 26 },
};

var Members = from p in people
              select p;

var Wygwam = from p in WygwamMembers
             select p;

var persons = from p in people
             join w in WygwamMembers
             on (string)p.LastName equals (string)w.LastName
             select p;
```

```
foreach(var Member in Members)
{
      Console.WriteLine("Membre de la 1ère liste: {0} {1}", Member.FirstName,
Member.LastName);
}

foreach(var WygwamMember in Wygwam)
{
      Console.WriteLine("Membre de Wygwam: {0} {1}", WygwamMember.FirstName,
WygwamMember.LastName);
}

foreach(var person in persons)
{
      Console.WriteLine("Membre de la 1ère liste et de l'équipe Wygwam: {0} {1}",
person.FirstName, person.LastName);
}
```

Ici, on a réalisé l'équivalent d'une requête **JOIN** sur deux tables. Cependant, nous somme passé par deux listes (au lieu de deux tables). Nous avons ensuite, grâce au mot clé **on**, spécifié sur quelle propriété la jointure devrait être faîte. Finalement, le mot clé **equals** nous a permis d'indiquer sur quelle valeur cette jointure devrait être faîte.

Voici le résultat du code précédent :



Si on regarde un peu le code qui a été généré (en utilisant **Reflector**), on peut apercevoir ceci:

```
IEnumerable<User> enumerable1 = Sequence.Join<User, User, string, User>(list1,
(IEnumerable<User>) list2, Program.<>9__CachedAnonymousMethodDelegated,
Program.<>9__CachedAnonymousMethodDelegatee,
Program.<>9__CachedAnonymousMethodDelegatef);
```

```
foreach (User user8 in enumerable1)
{
      Console.WriteLine("Membre de la 1\x00e8re liste et de l'\x00e9quipe Wygwam:
{0} {1}", user8.FirstName, user8.LastName);
}
```

On voit donc que le JOIN est symbolisé par la classe **Sequence** et plus particulièrement par la méthode **Join** de cette classe.

Les opérateurs standards

LINQ propose plus de 40 opérateurs différents. Nous n'allons donc pas tous les voir, mais seulement nous attarder sur les plus importants/utilisés.

L'opérateur OfType

L'opérateur **OfType** est utilisé pour réduire le nombre de résultats dans une requête.

Attention, cela ne fonctionne que pour les valeurs de la requête dont le type est indiqué grâce à cet opérateur.

Voici un exemple de code:

```
object [] values = { 1, "Tom", 'T', 12.5, 3, true, 20 };

var results = values.OfType<int>();

foreach(int i in results)
{
      Console.WriteLine(i);
}
```

Ici, seul les valeurs de type **int** seront retournées par le programme.

Les résultats sont donc maintenant plus facilement manipulables/exploitables dans le reste de votre application.

Les opérateurs Min, Max, Sum et Average (Moyenne)

Ces opérateurs, standards lorsque l'on travaille en SQL, sont maintenant disponibles avec LINQ. Plutôt que de faire de long discours, étant donné que leurs noms sont assez explicites, voici une démonstration:

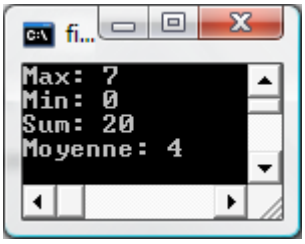
```
int [] IntegerValues = { 0, 2, 5, 6, 7 };

int max = IntegerValues.Max();
int min = IntegerValues.Min();
int sum = IntegerValues.Sum();
double average = IntegerValues.Average();

Console.WriteLine("Max: {0}", max);
Console.WriteLine("Min: {0}", min);
Console.WriteLine("Sum: {0}", sum);
Console.WriteLine("Moyenne: {0}", average);
```

Jusque là, rien de bien compliqué: on déclare un tableau d'entier et on appelle, sur ce tableau, les méthodes adéquates.

En interne, voici le code généré par le compilateur:

```
int[] numArray2 = new int[] { 0, 2, 5, 6, 7 };

int num1 = Sequence.Max(numArray2);
int num2 = Sequence.Min(numArray2);
int num3 = Sequence.Sum(numArray2);
double num4 = Sequence.Average(numArray2);

Console.WriteLine("Max: {0}", num1);
Console.WriteLine("Min: {0}", num2);
Console.WriteLine("Sum: {0}", num3);
Console.WriteLine("Moyenne: {0}", num4);
```

Autrement dit, nous aurions tout aussi bien pu appeler les méthodes statiques de la classe **Sequence** pour parvenir aux résultats.



L'opérateur Select

Depuis un petit moment, nous utilisons l'opérateur **Select** mais sans trop vraiment savoir à quoi il sert et pourquoi il est utilisé. Cette partie va donc être là pour tenter de rétablir les choses.

L'opérateur **Select** est utilisé pour créer la projection d'une séquence (liste, tableau, etc.…).

La collection résultante de cette projection peur alors être directement utilisée comme source de données d'un objet.

```
var values = new object [] { 1, "Tom", 'T', 12.5, 3, true, 20 };

var val = from v in values
          select v;

foreach(object o in val)
{
      Console.WriteLine(o.ToString());
}
```

Vous noterez une chose importante: le type retourné n'est jamais indiqué explicitement dans le code: il est créé par le compilateur en se basant sur le type spécifié.

L'opérateur Where

L'opérateur Where va vous permettre de filtrer une séquence de valeurs, tout comme on le fait en SQL:

```
var values = new object [] { 1, "Tom", 'T', 12.5, 3, true, 20 };

var val = from v in values
          where v.ToString().Length >= 3
          select v;

foreach(object o in val)
{
      Console.WriteLine(o.ToString());
}
```

Une autre façon d'écrire les clauses **Where** est de passer par les expressions lambda:

```
var values = new object [] { 1, "Tom", 'T', 12.5, 3, true, 20 };

var val = values.Where(v => v.ToString().Length >= 3);

foreach(object o in val)
{
	Console.WriteLine(o.ToString());
}
```

Le résultat est identique, pour une écriture simplifiée: à vous de choisir celle qui correspond le mieux à vos besoins.

L'opérateur Count

Si vous voulez pouvoir retourner le nombre d'éléments qu'il y a dans une séquence, alors utilisez l'opérateur **Count**.

Vous pouvez utiliser cet opérateur sur la collection en elle-même, ou bien l'utilisez conjointement avec d'autres opérateurs.

Voici un petit exemple:

```
var values = new int [] { 1, 2, 10, 23, 50 };

int count = values.Count();

Console.WriteLine("Nombre total d'éléments dans la liste: {0}", count);
```

On peut également utiliser les expressions lambda au sein de notre opérateur Count:

```
var values = new int [] { 1, 2, 10, 23, 50 };

int count = values.Count(i => i >= 10);

Console.WriteLine("Nombre total d'éléments dans la liste: {0}", count);
```

Et comme dit précédemment, le Count peut-être appliqué en même temps que d'autres opérateurs, comme on peut le voir ici avec le Where:

```
var values = new int [] { 1, 2, 10, 23, 50 };

// Avec le Where
int count = values.Where(i => i >= 20).Count();

Console.WriteLine("Nombre total d'éléments dans la liste: {0}", count);
```

C'est à la fois rapide et très puissant !

Les opérateurs ToArray et ToList

Les opérateurs **ToArray** et **ToList** sont utilisés pour convertir une séquence en tableau typé ou liste.

Cela peut s'avérer très pratique lorsque vous voulez lier une grille de données (**GridView**) à une séquence, en utilisant la liaison de données ("**DataBinding**").

```
var values = new int [] { 1, 2, 10, 23, 50 };

int [] integerArray = values.ToArray();
List<int>integerList = values.ToList();

foreach(int iArray in integerArray)
{
      Console.WriteLine(iArray);
}

foreach(int iList in integerList)
{
      Console.WriteLine(iList);
}
```

Après avoir couvert un aperçu des possibilités offertes par LINQ, intéressons nous à un sous-ensemble qui permet de manipuler, plus aisément, les bases de données et de faire du **mapping objet-relationnel**.

DLINQ

DLINQ est une sous-partie de LINQ qui se concentre sur tout ce qui traite de l'accès aux données avec LINQ.

En utilisant le terme DLINQ, je dois admettre que je commets une erreur de langage. En effet, Microsoft a annoncé récemment que DLINQ avait été renommé en:

> **LINQ to SQL**, anciennement connu sous le nom de DLINQ et qui sera une technologie de mapping objet-relationnel simple
> **LINQ to Entities**, qui sera la base du nouveau Framework d'Entité d'ADO.NET. Il s'agit d'une technologie de mapping objet-relationnel plus complexe.

A titre d'information, sachez que cet article ne traitera que de **LINQ for SQL**.

LINQ for SQL

Créer votre première couche d'accès aux données

LINQ for SQL représente une technologie de mapping objet relationnel à la fois simple mais efficace.

Je ne vais pas traiter ici du mapping objet mais simplement vous démontrer comment il est possible, avec LINQ for SQL, de manipuler facilement une base de données.

Pour cet article, j'ai travaillé avec **SQL Server 2005** (et la la base de données AdventureWorks) mais sachez que cela fonctionne parfaitement avec **SQL Server 2000**. A terme, il sera même possible d'interagir avec d'autre système de bases de données (**Oracle**, etc.….)

Commencer par créer, dans Visual Studio 2005, un nouveau projet de type "LINQ Preview" et dans ce projet, ajouter une classe entité (que vous nommerez Employees) et qui servira à mapper la table **Employees**.

Pour cela, utiliser le code suivant:

```
using System.Data.DLinq;

[Table(Name="Employees")]
public class Employees
{
	private int m_EmployeeId;

	[Column(Id=true)]
	public int EmployeeId
	{
		get
		{
			return m_EmployeeId;
		}
		set
		{
			m_EmployeeId = value;
		}
	}

	private string m_EmployeeLastName;

	[Column(Storage="m_EmployeeLastName")]
	public string LastName
	{
		get
		{
			return m_EmployeeLastName;
		}
		set
		{
			m_EmployeeLastName = value;
		}
	}
}
```

L'attribut **Table** sert à mapper une classe à une table de la base de données. L'attribut Column, quand à lui, est utilisé pour mapper les champs de votre classe aux colonnes de votre table.

Pour indiquer qu'une colonne est la clé primaire d'une table, il vous faut tout simplement utiliser le paramètre **Id** et le positionner à true.

Les champs peuvent être mappés à la base de données mais la plupart du temps, il est conseillé d'utiliser les propriétés publiques. Lorsque vos déclarez une propriété publique, vous devez indiquer le champ de stockage correspondant, au moyen du paramètre **Storage** de l'attribut **Column**.

Vous venez donc d'écrire votre première classe de mapping avec DLINQ ! Voyons alors comment utiliser cette classe dans votre application.

Interroger la base de données

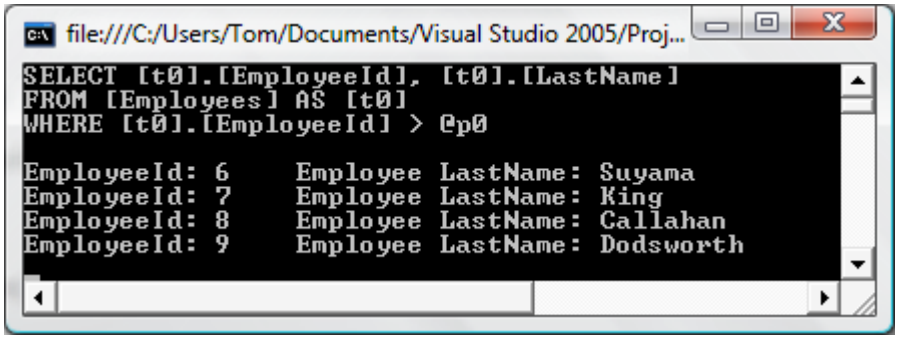
Requêter la base de données avec cette technique est relativement simple et peut se résumer en quelques mots.

Vous devez tout d'abord vous connecter à la base de données, puis définir avec quelle table vous souhaitez travailler. Enfin, vous effectuez des requêtes sur cette table.

Etant donné que nous sommes plutôt technique que littéraire, voyons voir ce que cela donne par le code:

```
//DataContext db = new DataContext(@"C:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\Data\Northwind.mdf");
DataContext db = new DataContext(@"Data Source=.\SQL2005;Initial
Catalog=Northwind;User Id=login;Password=password;");

// Pour afficher les requêtes exécutées sur la base: débuggage uniquement
db.Log = Console.Out;

Table<Employees> employees = db.GetTable<Employees>();

var EmployeeList = from e in employees
                   where e.EmployeeId > 5
                   select e;

foreach(Employees employee in EmployeeList)
{
      Console.WriteLine("EmployeeId: {0}\t Employee LastName: {1}",
employee.EmployeeId, employee.LastName);
}
```

Si on regarde le résultat généré, on voit bien que seul les employés dont le numéro d'employé est supérieur à 5 ont été sélectionnés:



Expliquons un peu plus en détails le code que nous venons d'écrire:

Nous commençons par créer un **DataContext**, qui va être notre interface avec la base de données.

Nous pouvons lui passer en paramètre:

Une chaîne de connexion
Un fichier de base de données (fichier MDF)
Une connexion

Ensuite, nous activons le Log afin de pourvoir visualiser les requêtes SQL qui seront générées puis exécutées sur la base. Cette étape n'est en rien obligatoire et n'est utile que lorsque vous faîtes du débogage.

Ensuite, nous récupérons le contenu de la table **Employees**. Cette table ne contient pas, physiquement, toutes les lignes de la table mais travaille comme un proxy pour les requêtes fortement typées.

Pour finir, nous effectuons une requête pour récupérer une liste d'employées, que nous afficherons par la suite.

Voyons à présent comment faire lorsque nous travaillons avec des bases de données plus complexes, qui possèdent, par exemple, des relations entre des tables. C'est en effet un des cas les plus courants et il serait dommage de ne pas le voir.

Travailler avec les relations entre les tables

Dans votre projet, ajoutez une nouvelle classe Entité que vous nommerez **Orders**, et dans laquelle le champ **Orders.EmployeeID** est une clé étrangère pointant vers **Employees.EmployeeID**:

```
[Table(Name="Orders")]
public class Orders
{
	private int m_OrderID;
	private string m_EmployeeID;
	private EntityRef<Employees>m_Employee;

	public Orders()
	{
		this.m_Employee = new EntityRef<Employees>();
	}

	[Column(Storage="m_OrderID", DBType="Int NOT NULL IDENTITY", Id=true,
AutoGen=true)]
	public int OrderID
	{
		get
		{
			return this.m_OrderID;
		}
	}

	[Column(Storage="m_EmployeeID", DBType="NChar(5)")]
	public string EmployeeId
	{
		get
		{
			return this.m_EmployeeID;
		}
		set
		{
			this.m_EmployeeID = value;
		}
	}

	[Association(Storage="m_Employee", ThisKey="EmployeeId")]
	public Employees Employee
	{
		get
		{
			return this.m_Employee.Entity;
		}
		set
		{
			this.m_Employee.Entity = value;
		}
	}
}
```

Le code ici présent est peut-être long, mais il s'avère simple à comprendre.

Les attributs sont utilisés pour représenter le type des champs dans la base. On peut, par exemple, affirmer que le champ **OrderID** est de type entier, qu'il est non nul et qu'il est auto-incrémenté. Bref, tout cela est assez logique.

A noter que pour la propriété OrderID, nous n'avons pas besoin de spécifier de setter car **AutoGen** est positionné à **true** (Vrai).

Pour représenter une relation un à un (one to one) ou un à plusieurs (one to many), DLINQ vous permet d'utiliser les types **EntityRef** et **EntitySet**. L'attribut Association est utilisé pour mapper la relation.

En créant l'association ci-dessus, vous serez à même d'utiliser la propriété **Order.Employee** pour accéder directement à l'objet **Employee** approprié.

Le type EntityRef est utilisé dans la classe Orders car il n'y a qu'un seul employé associé à une commande donnée.

Dès lors, vous pouvez accéder à un employé particulier en passant par les commandes. Mais si vous souhaitez faire l'inverse (accéder aux commandes depuis les employées), vous devez rajouter la relation adéquate dans la classe Employees:

```
private EntitySet<Orders>m_Orders;

public Employees()
{
      this.m_Orders = new EntitySet<Orders>();
}

[Association(Storage="m_Orders", OtherKey="EmployeeId")]
public EntitySet<Orders>OrdersForEmployee
{
      get { return this.m_Orders; }
      set { this.m_Orders.Assign(value); }
}
```

Notez que dans ce cas là, nous utilisons un type EntitySet car la relation entre Employees et Orders est de un à plusieurs (one to many).

Dès lors, vous pouvez accéder aux objets de type Orders depuis les objets de type Employees, et vice-versa:

```
var EmployeeList = from e in employees
                   where e.OrdersForEmployee.Any()
                   select e;

foreach(Employees employee in EmployeeList)
{
      Console.WriteLine("EmployeeId: {0}\t Employee LastName: {1}\t Quantity:
{2}", employee.EmployeeId, employee.LastName, employee.OrdersForEmployee.Count);
}
```

Jusqu'à maintenant, nous avons utilisé un DataContext classique. Nous allons voir comment nous pouvons typer cet objet DataContext, puis travailler avec.

Travailler avec l'objet DataContext fortement typé

Avant votre classe Employees, ajoutez le code suivant:

```
public class Northwind : DataContext
{
      public Table<Employees>EmployeesTable;
      public Table<Orders>OrdersTable;

      public Northwind(string connection) : base(connection) {}
}
```

Nous créons une classe Northwind (du nom de notre base de données) qui hérite de la classe DataContext.

Dans cette classe, nous rajoutons la déclaration des tables Employees et Orders que nous avons utilisées précédemment. Puis dans le Main de notre programme, nous utilisons cette classe à la place du DataContext classique:

```
Northwind db = new Northwind(@"Data Source=.\SQL2005;Initial
Catalog=Northwind;User Id=login;Password=password;");

var EmployeeList = from e in db.EmployeesTable
                   where e.EmployeeId > 5
                   select e;
foreach(Employees employee in EmployeeList)
{
      Console.WriteLine("EmployeeId: {0}\t Employee LastName: {1}",
employee.EmployeeId, employee.LastName);
}
```

A présent, vous n'avez plus besoin de faire appel à **GetTable<T>** pour accéder aux tables de votre base de données: en passant par leur nom, le travail est fait de la même manière.

Comme vous pouvez le voir, cette utilisation d'un DataContext fortement typé est très pratique mais peut s'avérer très longue à mettre en place dans le cas où votre base de données comporte de nombreuses tables.

Pour pallier ce problème, vous avez à votre disposition un outil, SqlMetal.exe, qui vous permet de générer, automatiquement, votre modèle objet.

Générer votre modèle objet avec SqlMetal:

SqlMetal.exe est un outil de *génération de code* qui va vous permettre de générer votre modèle objet sans que vous ayez besoin d'écrire vous-même une ligne de code.

Pour lancer SqlMetal, cliquez sur "**Démarrer**" => "**Programmes**" => "**Microsoft Visual Studio 2005**" => "**Visual Studio Tools**" => **Visual Studio Command Prompt**".

Commencer par changer de répertoire courant pour vous positionner dans le répertoire de votre projet puis saisissez la ligne suivante pour lancer la génération de votre modèle objet. Attention, il existe un nombre important de paramètre donc à vous de regarder la documentation/l'aide si vous voulez en avoir plus sur une option précise:

```
"c:\Program Files\LINQ Preview\bin\SqlMetal.exe" /server:.\SQL2005
/database:Northwind /pluralize /code:Northwind.cs /user:login /password:password
```

Une fois le fichier généré, il ne vous reste plus qu'à l'ajouter dans votre projet. Pour celui, faîtes un clic droit sur votre projet, puis choisissez "**Existing Item**" et sélectionnez votre fichier (Northwind.cs).

L'utilisation de ce fichier auto-généré est la même que lorsque vous avez utilisé le DataContext fortement typé:

```
Northwind db = new Northwind(@"Data Source=.\SQL2005;Initial
Catalog=Northwind;User Id=login;Password=password;");
var customers = from c in db.Customers
                select c;
foreach(Customer c in customers)
{
      Console.WriteLine("CustomerID: {0}\t CustomerName: {1}", c.CustomerID,
c.ContactName);
}
```

Si vous voulez utiliser des requêtes un peu plus complexes, alors rien ne vous en empêche:

```
var EmployeeList = (
                    from e in db.Employees
                    from c in db.Customers
                    where c.City == e.City
                    select new { ID = e.EmployeeID, Name =
String.Concat(e.LastName, " ", e.FirstName) }
                    ).Distinct();

foreach(var employee in EmployeeList)
{
      Console.WriteLine("ID: {0}\t Name: {1}", employee.ID, employee.Name);
}
```

Si on regarde la requête SQL qui est générée, on peut voir que ce n'est pas quelque chose de forcément aisé à écrire pour quelqu'un qui n'est pas administrateur de base de données:

```
SELECT DISTINCT [t2].[EmployeeID] AS [ID], [t2].[value] AS [Name]
FROM (
      SELECT [t0].[EmployeeID], ([t0].[LastName] + @p0) + [t0].[FirstName] AS
[value], [t1].[City], [t0].[City] AS [City2]
      FROM [Employees] AS [t0], [Customers] AS [t1]
      ) AS [t2]
WHERE [t2].[City] = [t2].[City2]
```

Modifier la base de données

Vous pouvez, grâce à LINQ, manipuler la base de données. Nous allons alors voir comment faire pour parvenir à nos fins.

Ajouter une entité

Pour ajouter une entité, rien de plus simple: il vous suffit de créer l'entité et de l'ajouter à la table de votre base de données.

Voyons un petit exemple:

```
// On créé une nouvelle entité
Customer customer = new Customer();

// On définit des propriétés
customer.ContactName = "Thomas LEBRUN";
customer.CustomerID = "THLEB";
customer.CompanyName = "SUPINFO";

// On l'ajoute à la table
db.Customers.Add(customer);

// On valide la transaction
db.SubmitChanges();

// On vérifie que notre entité à bien été ajoutée
foreach(var me in db.Customers.Where(cust => cust.CustomerID.Contains("THLEB")))
{
      Console.WriteLine("ID: {0}\t Société: {1}\t Nb de Commandes: {2}",
me.CustomerID, me.CompanyName, me.Orders.Count.ToString());
}
```

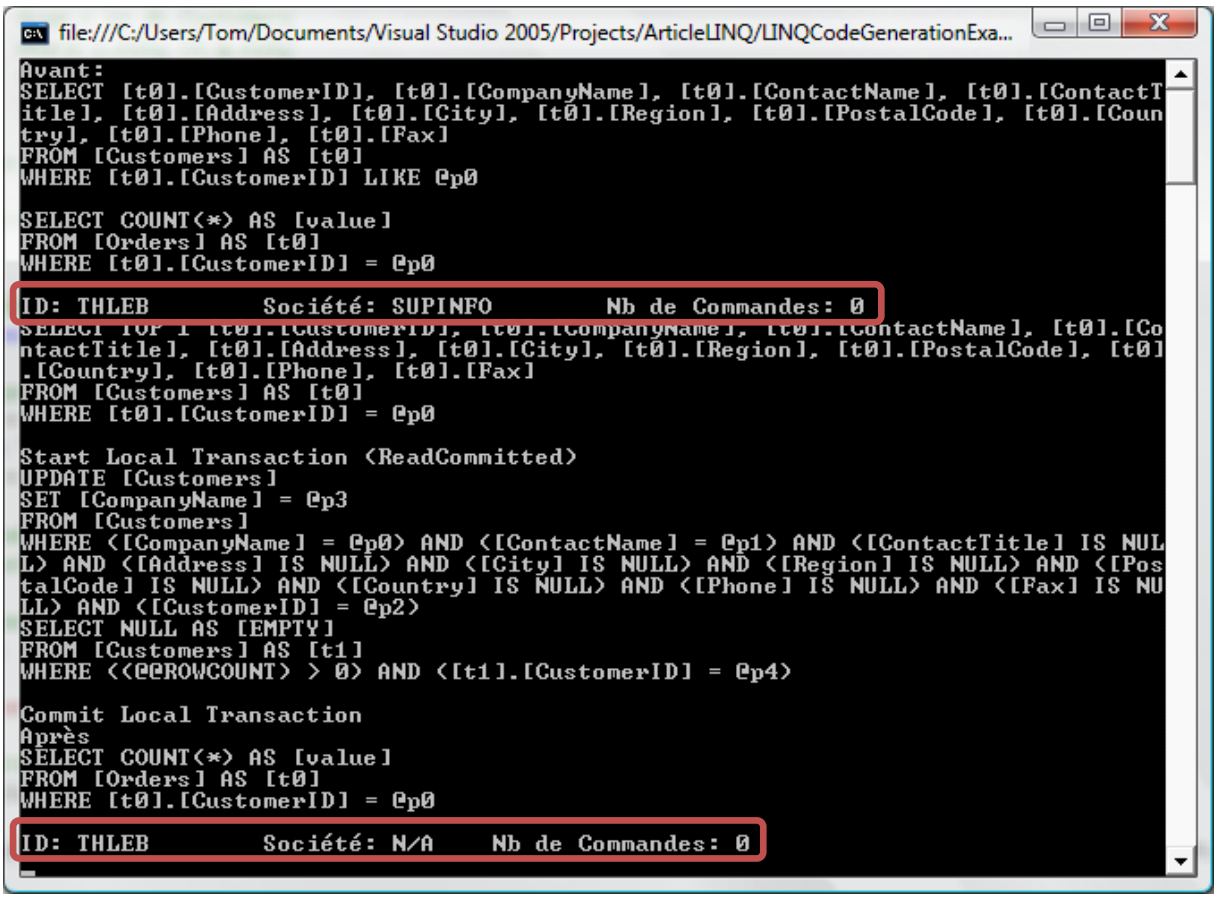
Note importante: il ne faut pas oublier l'appel à **SubmitChanges** pour valider les modifications sur la base de données.

Modifier une entité

Pour modifier une entité, rien de bien complexe là encore: il vous faut juste récupérer une référence vers l'entité à modifier, changer les valeurs que vous souhaitez puis valider les changements:

```
var my = (
         from c in db.Customers
         where c.CustomerID == "THLEB"
         select c
         ).First();

// On change la compagnie
my.CompanyName ="N/A";

// On valide la transaction
db.SubmitChanges();
```

Voici un petit exemple du résultat:



Supprimer une entité

Pour supprimer une entité, là encore, pas de mystères: un simple appel à la méthode Remove et le tour est joué:

```
var my = (
          from c in db.Customers
          where c.CustomerID == "THLEB"
          select c
          ).First();

// On supprime la ligne de la table
db.Customers.Remove((Customer)my);

// On valide la transaction
db.SubmitChanges();
```

Le designer DLINQ

Créer l'objet DLinq avec le designer

Tout ce que nous avons fait jusqu'à maintenant est entièrement fonctionnel, sans problèmes particulier.

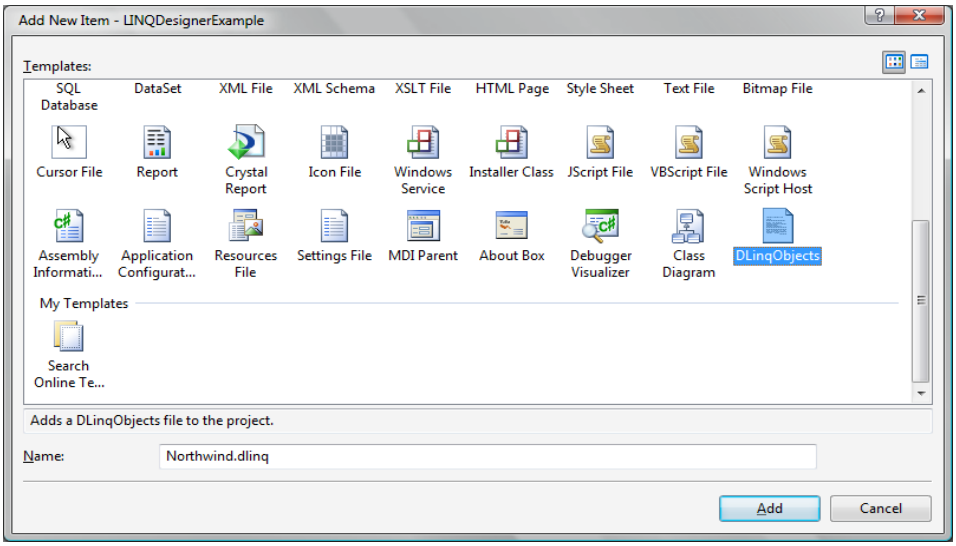
Mais en bon développeur, vous avez sans doute remarqué que générer un DataContext typé peut vite s'avérer laborieux et lourd de travail.

Une chose pourrait vous manquer, surtout si vous avez l'habitude de travailler avec la souris: le glisser/déposer.

En effet, imaginiez que vous soyez en mesure de faire un glisser/déposer des tables d'une base de données: vous auriez alors la possibilité de réaliser ce DataContext de façon beaucoup plus rapide et naturelle.

Et bien sachez que cette possibilité existe ! En effet, DLINQ intègre un designer qui vous permet, par simple glisser/déposer depuis Visual Studio, de générer un DataContext fortement typé, et de travailler avec par la suite.

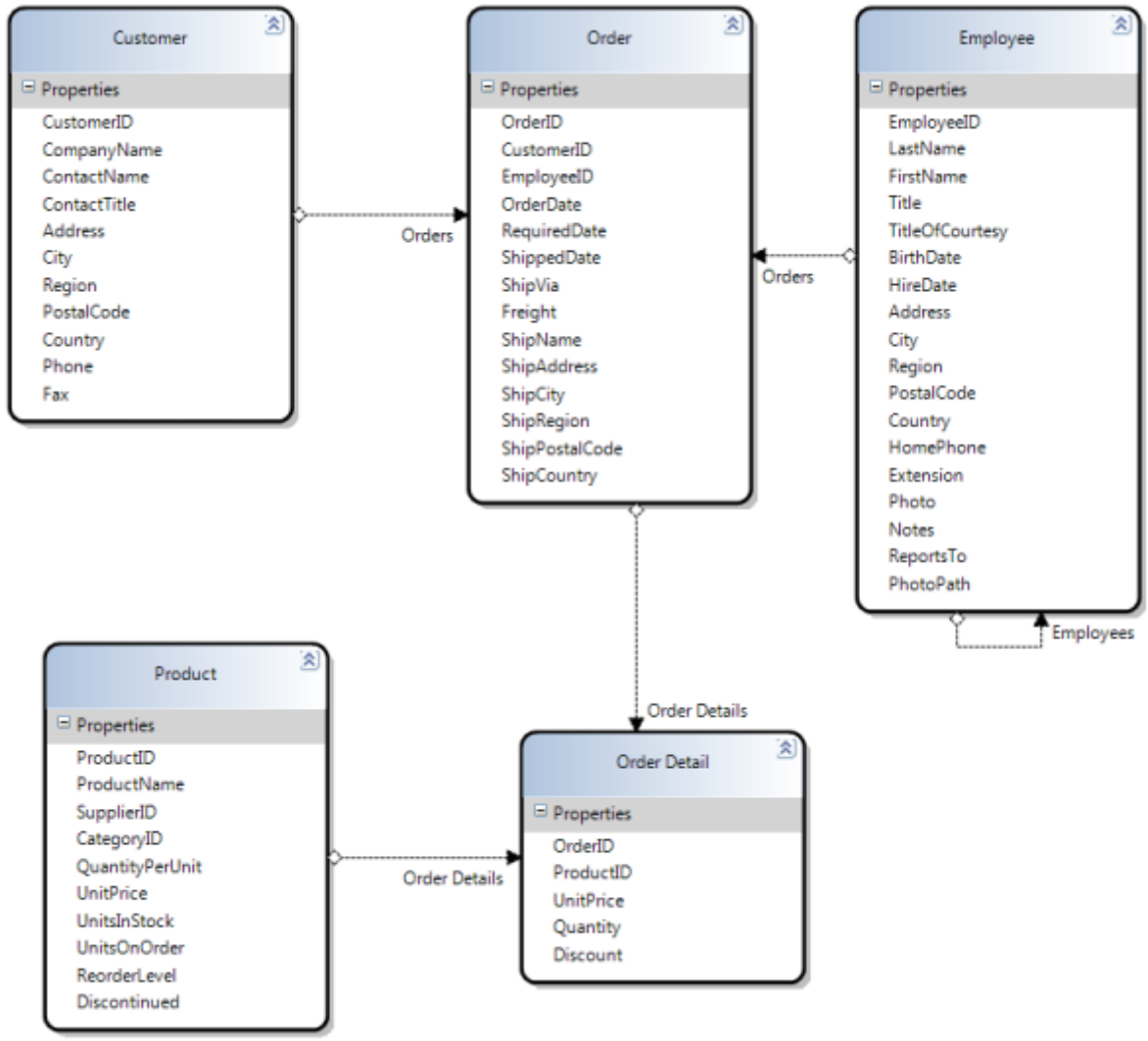
Pour utiliser ce designer, il vous suffit de rajouter, à votre projet, un élément de type "**DLinq Objects**". Pour cela, faîtes un clic droit sur le nom de votre projet et choisissez "**Add New Item**" => "**DLinqObjects**" :



Ensuite, il ne vous reste plus qu'à faire une glisser/déposer des tables de votre base de données qui vous intéresse, tout cela depuis votre explorateur de serveurs.

Dans notre cas, nous allons glisser/déposer les table: **Customer, Order, Order Details, Product** et **Employee**.

Voici ce que vous devez obtenir:



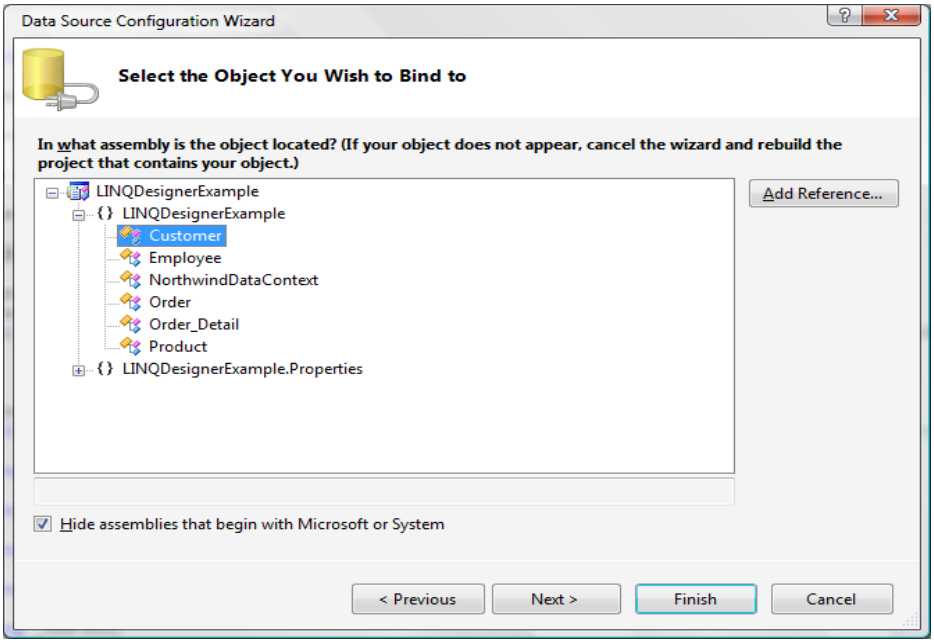
Et si l'on regarde le code qui a été généré, on s'aperçoit qu'il ressemble fort à celui que l'on sait maintenant écrire nous même:

```
public partial class NorthwindDataContext : System.Data.DLinq.DataContext
{
	public System.Data.DLinq.Table<Customer>Customers;
	public System.Data.DLinq.Table<Order>Orders;
	public System.Data.DLinq.Table<Order_Detail>Order_Details;
	public System.Data.DLinq.Table<Product>Products;
	public System.Data.DLinq.Table<Employee>Employees;
…
```

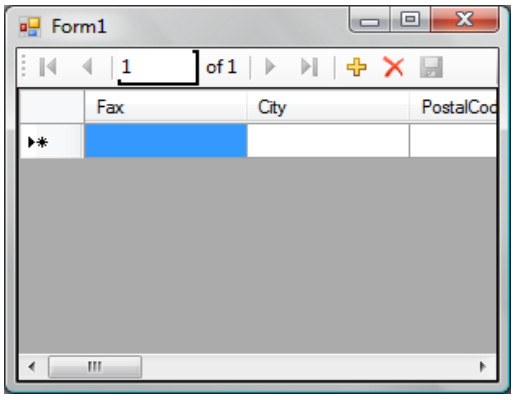
Utiliser l'objet DLinq

Pour utiliser cet objet nouvellement créé, vous pouvez par exemple ajouter une nouvelle source de données qui aura comme origine cet objet.

Pour cela, dans le menu, cliquez sur "**Data**" puis choisissez "**Add new Data Source**". Sur l'écran suivant, sélectionner "Object" comme source de vous données puis naviguer jusqu'à l'objet de votre choix, par exemple **Customer**:



A présent, vous pouvez tout simplement utiliser cette nouvelle source de données. Pour cela, vous pouvez essayez, depuis la fenêtre "Data Source", de faire un glisser/déposer de l'objet Customer sur votre formulaire principal. Comme à son habitude, Visual Studio 2005 vous génèrera tout le code (ou tout du moins une très très grande partie du code) ainsi que de l'interface graphique:



Ensuite, il ne reste plus qu'à utiliser le DataContext typé que vous avez créé tout à l'heure pour remplir cette grille de données.

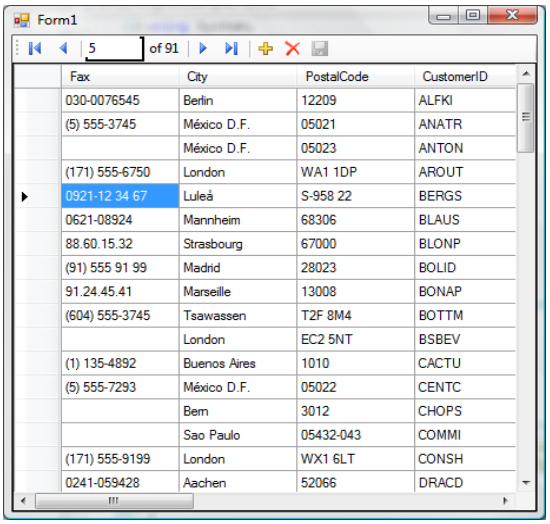
Pour cela, ajoutez la déclaration suivante dans votre code:

```
private NorthwindDataContext db = new NorthwindDataContext();
```

Et ensuite, vous devez simplement lier la source de données de votre grille à votre DataContext:

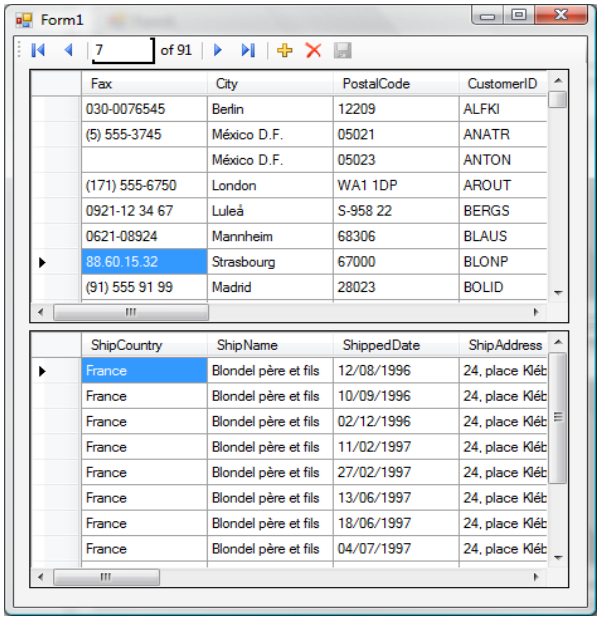
```
private void Form1_Load(object sender, EventArgs e)
{
      this.customerBindingSource.DataSource = this.db.Customers.ToBindingList();
}
```

A l'exécution, on s'aperçoit que tout fonctionne bien correctement:



Etant donné qu'il y a une relation entre Customer et Orders (celle-ci est représentée par une association, sur le diagramme de classe), nous avons donc une propriété Orders, au sein de la classe Customer, qui contient une collection d'objet de type Orders.

Si, depuis la fenêtre "Data Source", vous faîtes un glisser/déposer de cette propriété sur votre formulaire, vous vous rendrez compte que les informations sur les clients et sur les commandes sont affichées, et que les commandes sont liées au client sélectionné:



Designer DLinq et requêtes DLinq

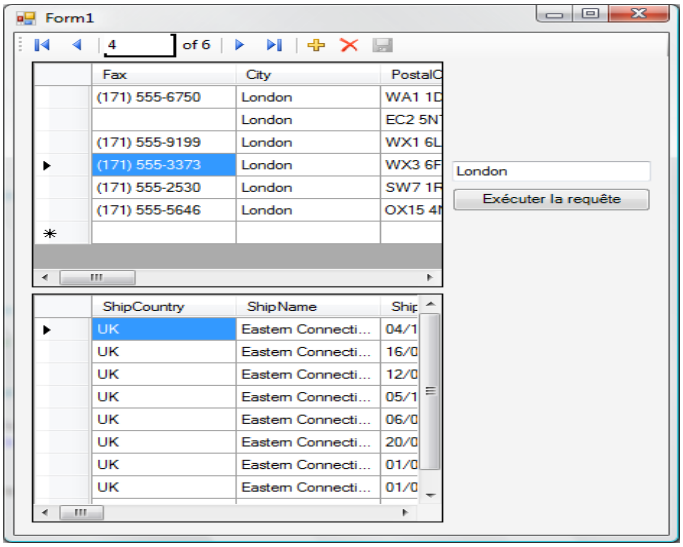
Vous avez également la possibilité d'utiliser des requêtes DLinq avec le DataContext que le designer vous a généré.

Voici un petit exemple:

```
private void btQuery_Click(object sender, EventArgs e)
{
      var results = from customers in this.db.Customers
                    where customers.City == this.tbCity.Text
                    select customers;

      this.customerBindingSource.DataSource = results.ToBindingList();
}
```

Ici, nous effectuons une requête DLinq qui va chercher une liste de client en fonction de leur ville.

Ensuite, nous lions notre grille de clients aux résultats de cette requête:



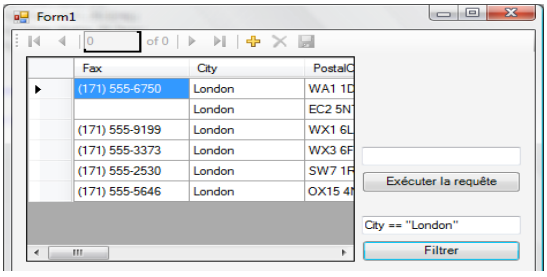
Filtrer les résultats dynamiquement

Une des possibilités qui vous est offerte est de pouvoir, dynamiquement, filtrer les résultats.

Pour cela, un peu plus de code va être nécessaire. En effet, vous allez devoir créer un filtre en vous basant sur sur une expression lambda:

```
private void btFilter_Click(object sender, EventArgs e)
{
      var p = Expression.Parameter(typeof(Customer), "");
      var filter = (Expression<Func<Customer,
bool>>)QueryExpression.Lambda(this.tbFilter.Text, p);

      this.customerDataGridView.DataSource =
this.db.Customers.Where(filter).ToList();
}
```

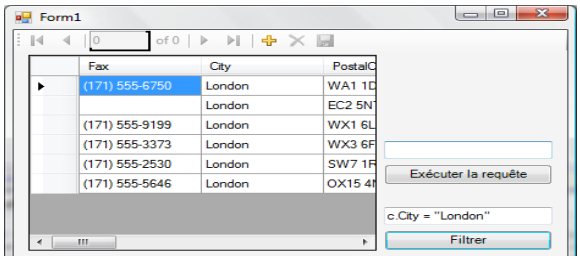
Si on exécute, on s'aperçoit bien qu'un clic sur le bouton "Filtrer" filtre bien les résultats en fonction de notre critère:



Une autre façon de faire aurait pu être:

```
private void btFilter_Click(object sender, EventArgs e)
{
      var p = Expression.Parameter(typeof(Customer), "c");
      var filter = (Expression<Func<Customer,
bool>>)QueryExpression.Lambda(this.tbFilter.Text, p);

      this.customerDataGridView.DataSource =
this.db.Customers.Where(filter).ToList();
}
```

La différence avec la première technique se situe au niveau du deuxième argument de la méthode **Parameter**: ici, on spécifie une valeur ce qui nous oblige, lorsque l'on applique le filtre, à utiliser cette valeur:



Avant, pour construire notre arbre d'expression et appliquer un filtre, nous utilisions:

```
City = "London"
```

Maintenant, on utilise

```
c.City = "London"
```

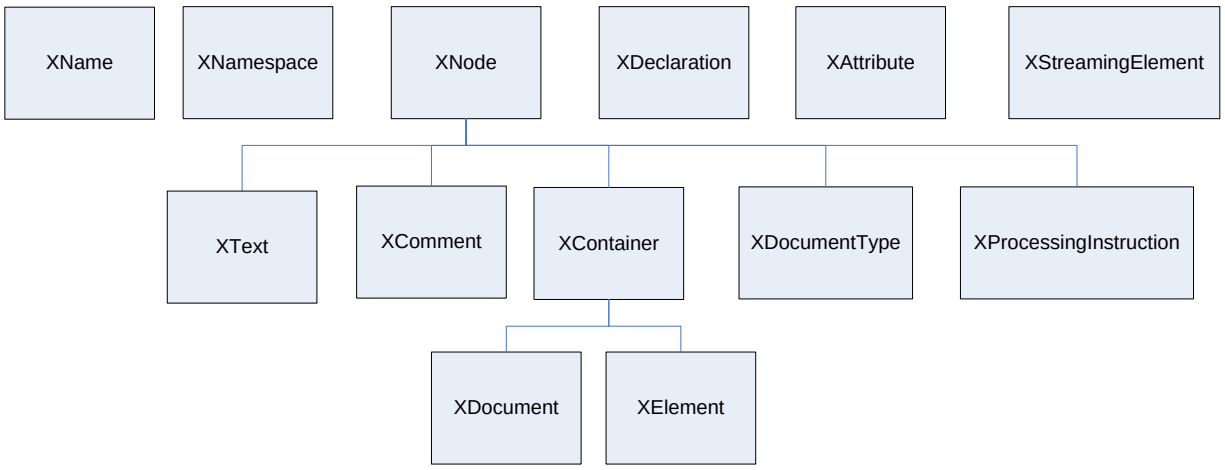
XLINQ

LINQ propose également des API dédiées à l'accès/manipulation de sources XML.

Ces API sont regroupées dans ce que l'on appelle **XLINQ** (*.NET Language Integrated Query for XML Data*).

XLINQ possède les avantages des opérateurs de requêtes standards et ajoute des extensions dédiées à XML. D'un point de vue XML, XLINQ fournit la puissance des requêtes et des transformations de XQuery et de XPath, le tout intégré dans .NET.

Voici, à titre d'information, une image vous présentant les principales classes de XLINQ:



Maintenant que nous connaissons la structure hiérarchique des classes de XLINQ, voyons comment nous pouvons travailler avec et produire des choses assez sympathiques.

Charger un document XML

Pour charger un document XML, avec XLINQ, vous avez 2 méthodes.

La première est d'appeler la méthode **Parse** de la classe **XElement** et de lui passer, en paramètre, le contenu de votre XML:

```
XElement contacts = XElement.Parse(
    @"<contacts>
        <contact>
            <name>Patrick Hines</name>
            <phone type=""home"">206-555-0144</phone>
            <phone type=""work"">425-555-0145</phone>
            <address>
               <street1>123 Main St</street1>
               <city>Mercer Island</city>
               <state>WA</state>
               <postal>68042</postal>
            </address>
            <netWorth>10</netWorth>
        </contact>
    </contacts>"
);
```

Ceci est valable dans le cas où vous récupérer un flux XML que vous voulez lire par exemple.

Une autre possibilité, si vous avez directement accès au fichier XML, est d'appeler la méthode **Load**:

```
XElement contactsFromFile = XElement.Load(@"c:\myContactList.xml");
```

Vous avez moins de code à écrire mais vous devez obligatoirement avoir accès au fichier pour appliquer cette méthode.

Créer des éléments XML

Pour créer des éléments XML, avec XLINQ, vous devez faire appel à ce que l'on appelle la "***construction fonctionnelle***".

Il s'agit d'une technique qui vous permet de créer tout (ou une simple partie) de votre arbre XML, en une seule étape:

```
XElement contacts =
    new XElement("contacts",
        new XElement("contact",
            new XElement("name", "Patrick Hines"),
            new XElement("phone", "206-555-0144"),
            new XElement("address",
                new XElement("street1", "123 Main St"),
                new XElement("city", "Mercer Island")
                )
            )
        );
```

Grâce à l'indentation, on peut avoir un aperçu de ce à quoi ressemblera le XML résultant de ce code.

Le constructeur de la classe XElement est flexible:

```
public XElement(XName name, params object[] contents)
```

Le deuxième paramètre étant un tableau d'objets, nous avons la possibilité de lui passer un nombre important de type:

String: le contenu de la chaîne de caractères sera ajouté à votre XML
XText: Peut-être du texte ou une valeur CData.
XElement: L'élément est ajouté au XML
XAttribute: L'attribut est ajouté
IEnumerable: les valeurs de l'énumération seront ajoutées aux XML
N'importe quoi: Vous pouvez également ajouter ce que vous voulez. La méthode **ToString()** sera appelée et le résultat sera ajouté

Parcourir un XML

Maintenant que nous avons vu comment créer un XML, voyons comment nous pouvons le parcourir et naviguer à l'intérieur.

Vous pouvez utiliser la méthode **Nodes()** pour récupérer, un à un, les enfants d'un XElement.

Cette méthode renvoi un **IEnumerable<object>**, car votre XElement peut aussi bien renvoyer du texte, mais également d'autres type XLINQ:

```
foreach(object o in contacts.Nodes())
{
      Console.WriteLine(o);
}
```

Si vous voulez être plus spécifique, vous pouvez faire votre recherche sur un type de nœud particulier:

```
foreach(object o in contacts.Nodes().OfType<XElement>())
{
      Console.WriteLine(o);
}
```

Les éléments XML étant importants dans beaucoup de scénarios, vous pouvez faire appel à la méthode **Elements()** qui est en fait un raccourci vers **Nodes.OfType<XElement>()**.

Si vous avez besoin de vous référer à un élément particulier, il vous suffit d'utiliser la méthode Element et de lui passer en paramètre le nom de l'élément qui vous intéresse. De cette façon, vous récupérerez en retour un XElement. S'il y en a plusieurs du même nom, vous récupérerez le premier:

```
XElement name = contacts.Element("name");
string NameValue = (string)contacts.Element("name");

Console.WriteLine(NameValue);
```

Manipuler un XML

Si vous avez en de manipuler un XML (ajouter/supprimer/modifier des enregistrements), alors XLINQ vous permet de le faire facilement.

Ajouter un élément

Pour ajouter un élément, il n'y a rien de bien compliqué: vous devez juste déclarer vous XElement et à l'ajouter là où vous le souhaitez.

La plupart du temps, on l'ajoute sur un XElement déjà existant:

```
XElement mobilephone = new XElement("mobilephone", "206-333-4546");
contacts.Add(mobilephone);
```

Il existe d'autres "variantes" de la méthode **Add()**. Vous avez par exemple la méthode **AddFirst()** qui vous sert à ajouter votre élément au début de votre XML, ainsi que les méthodes **AddAfterThis** et **AddBeforeThis**, qui vous seront utiles si vous souhaitez ajouter un élément avant ou après un élément particulier.

Supprimer un élément

Pour supprimer un élément, vous devez déjà avoir trouvé la réponse: il vous suffit d'appeler la méthode **Remove()**:

```
contacts.Element("mobilephone").Remove();
```

C'est simple, rapide , efficace et entièrement fonctionnel.

Une chose importante à noter, c'est que Remove() fonctionne aussi sur les IEnumerable donc vous pouvez tout à fait appeler cette méthode sur une collection, par exemple la collection d'Elements:

```
contacts.Elements("phone").Remove();
```

La dernière chose à savoir: si vous voulez supprimer le contenu d'un élément, et non pas l'élément en lui-même, il vous suffit d'appeler la méthode **RemoveContent()**:

```
contacts.Element("mobilephone").RemoveContent();
```

Modifier un élément

XLINQ met à votre disposition deux méthodes pour mettre à jour un élément:

ReplaceContent, qui agit sur l'élément même
SetElement, qui travaille depuis l'élément parent

Pour bien comprendre la différence entre ces deux méthodes, regardons des exemples:

```
contacts.Element("mobilephone").ReplaceContent("0123456789");
```

```
c ontacts.SetElement("mobilephone", "0123456789");
```

Dans le premier bout de code, nous nous positionnons sur le nœud qui nous intéresse puis dessus, nous appelons la méthode ReplaceContent.

Dans le deuxième cas, nous nous plaçons sur l'élément parent, puis nous appelons SetElement pour définir la valeur que doit prendre l'élément dont nous passons le nom en paramètre.

Ainsi, en fonction de la position à laquelle vous vous trouvez, dans votre XML, vous pouvez toujours modifier la valeur d'un nœud.

Il faut savoir que si vous passer la valeur **null** comme second paramètre de la méthode SetElement, l'élément en question sera supprimé.

Travailler avec les attributs

Eléments non négligeables lorsque l'on travaille avec des documents XML, les attributs peuvent également être manipulés avec XLINQ, au moyen de classes et méthodes spécialisées.

C'est ce que nous allons voir dans cette partie.

Ajouter des attributs

La technique pour ajouter un XAttribute est très similaire à celle employée pour ajouter un XElement:

```
XElement contacts =
    new XElement("contacts",
        new XElement("contact",
            new XElement("name", "Patrick Hines"),
            new XElement("phone",
                new XAttribute("type", "home"),
                 "206-555-0144"),
            new XElement("phone",
                new XAttribute("type", "pro"),
                 "206-333-4567"),
            new XElement("address",
                new XElement("street1", "123 Main St"),
                new XElement("city", "Mercer Island")
                )
            )
        );
```

Modifier/Supprimer/Mettre à jour des attributs

De la même manière que vous avez utilisé SetElement pour mettre à jour, supprimer ou ajouter des éléments, vous pouvez faire la même chose avec la méthode **SetAttribute()**:

Si l'attribut existe, il sera mis à jour
Si l'attribut n'existe pas, il sera créé
Si la valeur du deuxième paramètre de la méthode SetAttribute est **null**, l'attribut sera supprimé

De même, pour supprimer un attribut, vous pouvez là encore faire appel à la méthode **Remove()**:

```
contacts2.Element("phone").Attribute("type").Remove();
```

Accéder aux attributs

Pour accéder à un attribut, vous devez simplement utiliser la méthode **Attribute**, de la classe XElement, qui prend en paramètre le nom de l'attribut à atteindre/auquel vous souhaitez accéder:

```
foreach(var p in contacts2.Elements("phone"))
{
     if((string)p.Attribute("type") == "home")
     {
          Console.WriteLine("Tel perso: {0}", (string)p);
     }
}
```

Ainsi, on voit qu'accéder aux attributs est chose relativement aisée, puissant et fonctionne sur le même modèle que la technique pour accéder aux XElement.

Produire du XML

Pour produire du XML, vous pouvez utiliser l'une des nombreuses surcharges de la méthode **Save()** des type XElement et XDocument.

Vous pouvez sauvegardez votre XML dans un fichier, un **TextWriter** ou un **XmlWriter**:

```
contacts2.Save("C:\Contacts.xml");
```

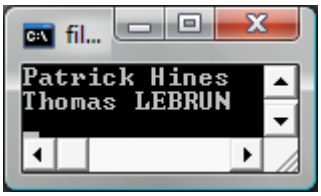
Requêter du XML avec XLINQ

Les techniques de requêtage que nous avons vues précédemment peuvent tout à fait être appliquées à XLINQ pour requêter du XML.

Voyons un exemple dans lequel nous interrogeons une liste de contacts au format XML:

```
var result2 = from c in contacts.Elements("contact")
              select c.Element("name");

foreach(XElement name in result2)
{
    Console.WriteLine(name.Value);
}
```

Si on regarde le résultat, on obtient bien la liste des noms des contacts de notre XML:



Conclusions

Au travers de cet article, vous avez découvert C# 3 ainsi que LINQ For SQL.

Vous avez ainsi pu découvrir les possibilités techniques qui étaient offertes et mises à disposition des développeurs.

Bien sur, il y aurait beaucoup plus de choses à dire: nous n'avons que survolé le sujet, mais nous avons déjà un petit bagage technique.

Bien que toujours en version Beta, je ne saurais que trop vous recommander de commencer à jeter un œil et à vous documenter sur ce que vous avez vu ici: en effet, il s'agit là de ce qui sera disponible demain, et afin de bien préparer l'avenir, le mieux est de commencer le plus tôt possible.

Remerciements

Je tiens à remercier:

Mitsuru Furuta
Fabrice Marguerie
Jean-Baptiste Evain

Pour la relecture et les conseils techniques qu'ils ont pu apporter à cet article.