



Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

LINQ To Object

Sommaire

1	LINQ to Object	3
1.1	Introduction par l'exemple.....	3
1.2	Evaluation de requête différée	6
1.3	Les objets « Enumerable » et « Queryable ».....	7
1.4	Les mots-clés du LINQ.	7
1.4.1	Le mot-clé « yield »	11
1.4.2	Instructions « select » et « where »	11
1.4.3	Instructions « from » et « in »	12
1.4.4	Méthodes de calculs groupés.....	12
1.4.5	Les méthodes « All », « Any » et « Contains »	16
1.4.6	La méthode « Distinct ».....	16
1.4.7	Les méthodes « Union » et « Intersect ».....	17
1.4.8	L'instruction « Join ».....	17
1.4.9	L'instruction « Let »	17
1.4.10	Méthodes « Range », « Repeat », « Empty » et « DefaultIfEmpty ».....	18
1.4.11	Les méthodes « FirstOrDefault » et « LastOrDefault »	20
1.4.12	Les méthodes « Take » et « Skip »	20
1.4.13	Les méthodes « OrderBy », « OrderByDescending », « ThenBy », « ThenByDescending » et « Reverse »	21
1.4.14	Les méthodes « ElementAt » et « ElementAtOrDefault ».....	22
1.4.15	La méthode « Single » et « SingleOrDefault ».....	22
1.4.16	La méthode « Concat ».....	23
1.4.17	La méthode « OfType »	23
1.4.18	La méthode « SelectMany »	24
1.4.19	L'instruction « Group [...] By [...] ».....	24
1.4.20	L'instruction « Group Join ».....	25
1.4.21	Les méthodes « ToList », « ToArray » et « ToDictionary »	26
1.5	Exemples de requêtes	27
1.5.1	Arborescence des fichiers.....	27
2	Conclusion	28

1 LINQ to Object

1.1 Introduction par l'exemple.

Les objets LINQ que nous allons utiliser sont dans la librairie « System.Linq » et la référence qui permet d'accéder aux objets de ce NameSpace est « System.Core ». Le LINQ impose une interface pour pouvoir requêter un objet, or les listes génériques implémentent déjà ces interfaces et nous permettrons de ranger n'importe quels objets, même de classes personnalisées. Nous importerons donc également la NameSpace « System.Collections.Generic » pour tous nos exemples.

Dans les deux exemples à venir nous utiliserons cette liste :

```
C#
List<String> source = new List<String>();
source.Add("texte");
source.Add("string");
source.Add("phrase");
source.Add("chaîne");
source.Add("caractères");

VB.Net
Dim source As List(Of String) = New List(Of String)()
source.Add("texte")
source.Add("string")
source.Add("phrase")
source.Add("chaîne")
source.Add("caractères")
```

La première requête ci-dessous est une requête simple. Voici ci-dessous un extrait de code qui sera expliqué par la suite :


```
C#
var requete1 = from s in source
               where s.Length < 10
               select s;
foreach (var e in requete1)
{
    Console.WriteLine(e);
}
Console.Read();

VB.Net
Dim requete1 = From s In source
               Where s.Length < 10
               Select s

For Each e In requete1
    Console.WriteLine(e)
Next
Console.Read()
```

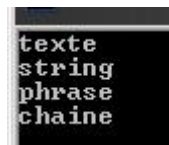
Cette requête sélectionne notre liste de chaînes de caractères « source », dont il identifie chaque objet à tour de rôle par « s », tous les objets de la liste (ici des « String ») ayant une longueur inférieure à 10 caractères. La boucle « foreach » permet d'afficher le contenu de la variable « requête1 » qui a stocké les résultats de notre requête.

```
var requete1 = from s in source
               where s.Length < 10
               select s
```



Dans la capture d'écran ci-dessus, nous pouvons remarquer que le type retourné par notre requête LINQ est complexe. De plus, ce type peut varier en fonction de la source et de la requête exécutée, c'est pourquoi nous avons utilisé le typage implicite de cette variable en utilisant le mot-clé « var ». On aurait pu spécifier le type dans ce cas en utilisant « IEnumerable<string> » ou « IEnumerable(of string) », car l'objet retourné implémente cette interface et liste des éléments de type « String ».

Voici ci-dessous l'affichage du résultat dans la console de la requête exécutée :



```
texte
string
phrase
chaîne
```

Remarque : Comme vous pourrez le constater, le mot-clé « from » se met au début de la requête et pas à la fin comme l'indique la norme. Après réflexion, il a été choisi de positionner le « from » au début de la requête afin d'avoir une syntaxe plus logique pour les non-initiés mais aussi afin de bénéficier d'une meilleure IntelliSense.

Voici maintenant un exemple de code légèrement plus complexe mettant plus en valeur les capacités de LINQ.

```
C#
var requete = from s in source           // Même début que tout à l'heure
               where s.Length < 10
               select new {             // Retourne des objets anonymes
avec
               aUnE = s.Contains("e"),  // un boleen (vrai si s contient
un "e")
               chaine = s               // et la chaine
               };

foreach (var e in requete)
{
    Console.WriteLine("L'élément \"" + e.chaine + "\" contient un e : " +
e.aUnE.ToString() + "\n");
}
Console.Read();

VB.Net
Dim requete = From s In source           ' Même début que tout à l'heure
               Where s.Length < 10 _
               Select New With { _      ' Retourne des objets anonymes avec
               .aUnE = s.Contains("e"), _ ' un boleen (vrai si s contient un
"e")
               .chaine = s _           ' et la chaine
               }

For Each e In requete
    Console.WriteLine("L'élément \"" + e.chaine + "\" contient un e = " +
e.aUnE().ToString + vbNewLine)
Next
Console.Read()
```

Voici ce qu'affiche cette nouvelle requête :

```
L'élément "texte" contient un e : True
L'élément "string" contient un e : False
L'élément "phrase" contient un e : True
L'élément "chaine" contient un e : True
```

Remarque : Notez le type que la variable « requete » a pris dans la capture d'écran ci-après. Ce type pourrait changer si la requête change et renvoie un autre type de valeur.

```
var requete = from s in source // dans la liste source qui se
               {System.Linq.Enumerable.SelectIterator<string, <bool,string>>>} st
```

1.2 Evaluation de requête différée

En LINQ, les requêtes ne sont pas évaluées lorsqu'elles sont définies, mais seulement quand elles sont utilisées. Un ajout d'informations ultérieur à l'établissement d'une requête sur des données est donc pris en compte lors d'une nouvelle utilisation de la requête.

Voici un exemple :

C#

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);

var requete = from num in numbers
              select num;

numbers.Add(3);

foreach (var num in requete)
{
    Console.WriteLine(num);
}

Console.Read();
```

VB.Net

```
Dim numbers As List(Of Integer) = New List(Of Integer)()
numbers.Add(1)
numbers.Add(2)

Dim requete = From num In numbers _
              Select num

numbers.Add(3)

For Each num In requete
    Console.WriteLine(num)
Next

Console.Read()
```

Lors de l'affichage, nous pouvons remarquer que l'ajout du chiffre « 3 » après l'écriture de la requête a bien été retourné.

1.3 Les objets « Enumerable » et « Queryable »

Les objets « Enumerable » et « Queryable » sont des objets génériques, c'est-à-dire qu'ils sont susceptibles d'être utilisés avec différents types. Les méthodes/mots-clés que nous allons aborder, et qui sont la base du LINQ, peuvent le plus souvent être utilisés de deux manières différentes : soit avec la syntaxe requête qui est plus lisible et plus proche du SQL, soit avec la syntaxe des méthodes.

```
C#
//Exemple précédent
var requete = from num in numbers
              select num;

//Syntaxe avec méthode
var requete2 = numbers.Select(num => num);
```

```
'VB.Net
'Exemple précédent
Dim requete = From num In numbers _
              Select num

'Syntaxe avec méthode
Dim requete2 = numbers.Select(Function(num) num)
```

Cet exemple permet de comparer les deux différentes manières d'écrire une requête. Pour plus de lisibilité et de simplicité, nous choisirons d'écrire les requêtes avec le premier procédé.

1.4 Les mots-clés du LINQ.

Le LINQ, derrière cet aspect de langage, est un set d'objets et de méthodes qui sont organisés par le compilateur bien que dans le code ils s'utilisent comme des mots-clés. Ces mots-clés étant nombreux, nous allons vous présenter ceux qui vous seront le plus utile. Bien que non exhaustive, cette liste des mots-clés suffira à la plupart de vos requêtes.

Pour plus de clarté, nous allons présenter ci-après les instructions qui précèdent les requêtes, les classes et leurs instances qui seront utilisées dans les prochains exemples.

Il est utile de redéfinir *ToString* pour faciliter les tâches d'affichage pour les requêtes simples.

L'accesseur `Id` dans l'objet *Software* sert à l'auto incrémentation de l'ID.

Dans la plus part des exemples, il vous sera libre d'afficher le résultat dans la console, sur un site web ou dans une fenêtre les données.

Les classes :

```
C#  
  
class person  
{  
    public string nom;  
    public int salaire;  
    public int age;  
    public string prenom;  
    public int logiciel;  
  
    public person(string name, string surname, int yearsold, int salary,  
int favoritesoftwareid)  
    {  
        nom = name;  
        prenom = surname;  
        age = yearsold;  
        salaire = salary;  
        logiciel = favoritesoftwareid;  
    }  
    public override string ToString()  
    {  
        return "Nom :\t" + nom + "\t Age :\t" + age + "\n Prenom :\t" +  
prenom + "\t Salaire : \t"+salaire;  
    }  
}  
  
class software  
{  
    public string compagnie;  
    public int prix;  
    public int annee;  
    private static int p_id=0;  
    public int reference;  
    public string nom;  
  
    public software(string corporation, string name, int releaseyear, int  
cost)  
    {  
        compagnie =corporation;  
        nom = name;  
        annee = releaseyear;  
        prix = cost;  
        reference = id;  
    }  
    public override string ToString()  
    {  
        return "Nom :\t" + nom + "\t Annee :\t" + annee.ToString() + "\n  
Societe :\t" + compagnie + "\t Prix :\t"+prix.ToString();  
    }  
  
    public static int id  
    {  
        get { return ++p_id; }  
    }  
}
```




'VB.Net

```
Public Class person
    Public nom As String
    Public salaire As Integer
    Public age As Integer
    Public prenom As String
    Public logiciel As Integer

    Public Sub New(ByVal name As String, ByVal surname As String, ByVal
yearsold As Integer, ByVal salary As Integer, ByVal favoritesoftwareid As
Integer)
        nom = name
        prenom = surname
        age = yearsold
        salaire = salary
        logiciel = favoritesoftwareid
    End Sub

    Public Overrides Function ToString() As String
        Return "Nom :" + vbTab + nom + vbTab + "Age :" + vbTab +
age.ToString() + vbNewLine + "Prenom :" + vbTab + prenom + vbTab + "Salaire
:" + vbTab + salaire.ToString()
    End Function
End Class

Public Class software
    Public compagnie As String
    Public prix As Integer
    Public annee As Integer
    Private Shared p_id As Integer = 0
    Public reference As Integer
    Public nom As String
    Private Shared ReadOnly Property id() As Integer
        Get
            p_id = p_id + 1
            Return p_id
        End Get
    End Property

    Public Sub New(ByVal corporation As String, ByVal name As String, ByVal
releaseyear As Integer, ByVal cost As Integer)
        compagnie = corporation
        nom = name
        annee = releaseyear
        prix = cost
        reference = id
    End Sub

    Public Overrides Function ToString() As String
        Return "Nom :" + vbTab + nom + vbNewLine + "Annee :" + vbTab +
annee.ToString() + vbNewLine + "Societe :" + vbTab + compagnie + vbTab +
"Prix :" + vbTab + prix.ToString() + vbNewLine
    End Function
End Class
```

Les listes :

```
C#
List<person> users = new List<person>();
users.Add(new person("Dupont", "Albert", 25, 1000, 3));
users.Add(new person("Gates", "Bill", 54, Int32.MaxValue, 1));
users.Add(new person("Ballmer", "Steve", 25, 300000, 1));
users.Add(new person("Narbonne", "Christophe ,Jean-Charles", 20, 0, 3));
users.Add(new person("Stallman", "Richard", 52, 2000, 4));

List<software> softs = new List<software>();
softs.Add(new software("Microsoft", "Windows", 1998, 150));
softs.Add(new software("Microsoft", "Office", 2007, 300));
softs.Add(new software("Mozilla", "Firefox", 2000, 0));
softs.Add(new software("FSF", "Emacs", 1985, 0));
softs.Add(new software("Microsoft", "Visual Studio", 2003, 600));
```

```
VB.Net
Dim users As List(Of person) = New List(Of person)()
users.Add(New person("Dupont", "Albert", 25, 1000, 3))
users.Add(New person("Gates", "Bill", 54, Int32.MaxValue, 1))
users.Add(New person("Ballmer", "Steve", 25, 300000, 1))
users.Add(New person("Narbonne", "Christophe ,Jean-Charles", 20, 0, 3))
users.Add(New person("Stallman", "Richard", 52, 2000, 4))

Dim softs As List(Of software) = New List(Of software)()
softs.Add(New software("Microsoft", "Windows", 1998, 150))
softs.Add(New software("Microsoft", "Office", 2007, 300))
softs.Add(New software("Mozilla", "Firefox", 2000, 0))
softs.Add(New software("FSF", "Emacs", 1985, 0))
softs.Add(New software("Microsoft", "Visual Studio", 2003, 600))
```

Nous pouvons assimiler ces listes comme une base de données à deux tables.

1.4.1 Le mot-clé « yield »

« yield » est un mot clé qui permet de mettre en pause l'exécution d'une méthode. Les méthodes utilisant le mot-clé « yield » sont appelées des « iterator », type retourné par les requêtes LINQ. Ce mot-clé doit obligatoirement être placé devant un « return » ou un « break ». De ce fait, son utilisation est aussi possible pour des boucles comme le « for » par exemple :

```
C#  
public static IEnumerable GetReference()  
{  
    for (int i = 1; i < 15; i++)  
    {  
        yield return i;  
    }  
}  
  
static void Main()  
{  
    foreach (var reference in GetReference())  
    {  
        Console.WriteLine(reference);  
    }  
}
```

Remarque : Il n'existe aucun équivalent du mot-clé « yield » en Visual Basic 9.0.

Dans cet exemple de code, la méthode permettra de retourner un entier (qui peut représenter la référence d'un logiciel) différent à chaque appel de la méthode jusqu'à ce que celui-ci atteigne la valeur « 15 ».

1.4.2 Instructions « select » et « where »

Le mot-clé « select », dans une requête LINQ, définit les éléments de la requête qui seront sélectionnés et accessibles par la suite. Il est possible d'utiliser le mot-clé « new » afin d'établir la requête de manière plus personnalisée. L'élément ainsi créé retournera des objets anonymes dans lesquels seront rangés les éléments sur lesquels vous voulez sélectionner des données.

Il peut être utile, pour restreindre la sélection de données, d'apposer une condition sur la requête. Pour cela, il existe le mot-clé « where » qui prend en argument une expression conditionnelle qui retournera un booléen. Si sa valeur est évaluée à *false*, l'élément actuel ne sera pas inclus à la sélection. Comme pour toute expression booléenne, il est possible d'utiliser les connecteurs logique « && » (ET) et « || » (OU) afin d'ajouter des conditions supplémentaires.

Beaucoup d'exemples à venir utiliseront ces instructions indispensables.

*Remarque : Si vous ne placez pas d'instruction « select », tous les éléments seront retournés (équivalent en SQL de « SELECT * »).*

1.4.3 Instructions « from » et « in »

Au même titre que « select » et « where », « from » et « in » sont à maîtriser car ces deux instructions vont de paire. Par exemple, « `from u in users` » fait passer tour à tour dans la variable « u » tous les éléments de la liste « users ». L'extrait de requête « `from element in list` » est une sorte de « foreach » permettant de parcourir la liste.

Remarque : Les requêtes au moment de l'exécution appellent tour à tour chaque valeur.

1.4.4 Méthodes de calculs groupés

LINQ implémente des mots servant à retourner directement des valeurs particulières d'une liste de la même manière qu'en SQL.

Les méthodes suivantes sont disponibles :

Méthodes	Description
Min()	Permet d'obtenir la valeur minimum sur une liste de valeurs sélectionnées.
Max()	Permet d'obtenir la valeur maximum sur une liste de valeurs sélectionnées.
Count()	Permet de connaître le nombre d'éléments sélectionnés.
Average()	Permet de connaître la moyenne de valeurs.
Sum()	Réalise la somme de valeurs.
Aggregate()	Permet de créer sa propre valeur à partir de celles sélectionnées.

1.4.4.1 Méthode « Max »

Voici deux exemples de requête utilisant la méthode « Max ». Comme vous pouvez le constater, la deuxième requête a été explicitement typée. La méthode « Max » ne retourne pas l'objet de la liste sélectionnée qui détient l'attribut maximum, mais seulement cet attribut.

C#	VB.Net
<pre>var requete = from u in users where u.prenom.Length < users.Max((p) => p.prenom.Length) select u; int requete2 = users.Max((person p) => p.age); // retourne 54 et non le person Bill Gates</pre>	<pre>Dim requete = From u In users _ Where u.prenom.Length < _ users.Max(Function(p As person) p.prenom.Length) _ Select u Dim requete2 As Integer = _ users.Max(Function(p As person) p.age) ' retourne 54 et non le person Bill Gates</pre>

Remarque1 : Pour la première requête, notez que dans la condition « where » la méthode « Max » s'applique sur « users » (la liste) et non « u », qui contient tous les éléments. Or ces éléments étant de type « person », ils n'implémentent pas l'interface « IEnumerable ».

Remarque2 : Dans la partie C#, nous passons en argument une expression lambda à la méthode « Max ». L'exemple à propos d'« Aggregate » vous permettra de mieux comprendre cette notion.

1.4.4.2 Méthode « Min »

La méthode « Min » s'utilise exactement comme la méthode « Max » à la seule différence qu'il ne retournera pas l'attribut maximum, mais minimum.

C#	VB.Net
<pre>var requete = from u in users where u.prenom.Length < users.Min((p)=> p.prenom.Length) + 2 select u; int requete2 = users.Min((person p) => p.age);</pre>	<pre>Dim requete = From u In users _ Where u.prenom.Length < _ users.Min(Function(p As person) p.prenom.Length) + 2 _ Select u Dim requete2 As Integer = _ users.Min(Function(p As person) p.age)</pre>

1.4.4.3 Méthode « Count »

La méthode « Count » est légèrement différente. En effet, elle n'a pas forcément besoin d'argument vu qu'il compte le nombre d'éléments. Sans argument, la méthode dénombre tous les éléments sélectionnés mais il est possible de lui passer une condition à vérifier afin de ne compter que certains éléments. De ce fait, son usage est relativement simple mais paraît plus utile dans un cas de requêtes imbriquées.

C#	VB.Net
<pre>var requete = from u in users where u.prenom.Length < users.Count()+ 2 select u; int requete2 = users.Count();</pre>	<pre>Dim requete = From u In users _ Where u.prenom.Length < _ users.Count() + 2 _ Select u Dim requete2 As Integer = _ users.Count()</pre>

Remarque : Si vous mettez une expression lambda en argument de Count comme « c => c.prenom.Length < 11 », cela fera l'équivalent d'un filtre « where » sur la liste comptée.

1.4.4.4 Méthode « Average »

La méthode « Average » permet de calculer facilement la moyenne des éléments. Elle s'utilise de la même manière que « Min » et « Max ». Dans notre exemple, la variable « requete » contient les « person » ayant des noms de taille inférieure ou égale à la moyenne.

C#	VB.Net
<pre>var requete = from u in users where u.nom.Length <= users.Average((s)=> s.nom.Length) select u; double requete2 = users.Average((u)=>u.age);</pre>	<pre>Dim requete = From u In users _ Where u.nom.Length <= _ users.Average(Function(s As person) s.nom.Length) _ Select u Dim requete2 As Double = _ users.Average(Function(s As person) s.age)</pre>

1.4.4.5 Méthode « Sum »

Cette méthode s'utilise comme « Average » et permet de faire la somme des éléments de la liste sélectionnée. Cela peut-être pratique pour les requêtes imbriquées ou pour calculer une somme de prix par exemple.

C#	VB.Net
<pre>int requete2 = users.Sum((person p) => p.age); // retourne 106</pre>	<pre>Dim requete2 As Integer = _ users.Max(Function(p As person) p.age) ' retourne 106</pre>

1.4.4.6 La méthode « Aggregate »

« Aggregate » est probablement la méthode de cette partie la plus puissante car dès lors que vous la maîtrisez, vous n'aurez plus besoin des mots-clés que l'on vient de voir, ou presque.

La méthode « Aggregate » attend en paramètre une valeur initiale et une fonction. En Visual Basic vous devrez utiliser le mot-clé « Function » alors qu'en C# ce sera des expressions lambdas.

- L'opérateur ternaire du C# est comme une fonction « if » en condensé. Elle est composée de 3 arguments. Le premier est l'expression conditionnelle à vérifier, elle retourne donc un booléen. Cet argument est directement suivi d'un point d'interrogation « ? » puis d'un espace afin d'introduire l'expression à retourner si la condition est vraie. Si la condition est fausse, c'est l'expression suivante qui est retournée. Ces deux derniers arguments sont séparés par des deux-points « : ».
- « if » s'utilise comme une fonction standard et fonctionne comme l'opérateur ternaire. Elle prend en premier argument un booléen puis deux objets. Le premier objet est retourné si le booléen est vrai par contre si il est faux, c'est le deuxième objet qui sera retourné.

La méthode « Aggregate » attend en arguments 2 paramètres. Le premier sera la valeur en cours de traitement et la deuxième l'entrée de la liste. La méthode « Aggregate » permet de faire un calcul récurant avec toutes les entrées de la liste. Voici par exemple la reconstitution des fonctions vues au dessus. Sur une liste d' « int », nous allons recomposer toutes les fonctions vues précédemment.

C#

```
int[] liste = { 4, 3, 3, 2, 5, 7, 11 };

int max = liste.Aggregate(int.MinValue, (cumul, chaque) => cumul > chaque?
cumul:chaque);
int min = liste.Aggregate(int.MaxValue, (cumul, chaque) => cumul > chaque?
chaque:cumul);
int count = liste.Aggregate(0, (cumul, chaque) => cumul + 1);
int sum = liste.Aggregate(0, (cumul, chaque) => cumul + chaque);
double average = sum / count;
int produit = liste.Aggregate(1, (cumul, chaque) => cumul * chaque);

Console.WriteLine(max.ToString()+"\n"+ min.ToString()+"\n"+ count.ToString()+"\n"+
sum.ToString()+"\n"+ average.ToString()+"\n"+ produit.ToString());
```

VB.Net

```
Dim liste As Integer() = {4, 3, 3, 2, 5, 7, 11}

Dim max As Integer = liste.Aggregate(Int32.MinValue, _
    Function(cumul, chaque) IIf(cumul > chaque, cumul, _
chaque))

Dim min As Integer = liste.Aggregate(Int32.MaxValue, _
    Function(cumul, chaque) IIf(cumul > chaque, _
chaque, cumul))

Dim count As Integer = liste.Aggregate(0, Function(cumul, chaque) cumul + 1)
Dim sum As Integer = liste.Aggregate(0, Function(cumul, chaque) cumul + chaque)
Dim average As Double = sum / count
Dim produit As Integer = liste.Aggregate(1, Function(cumul, chaque) cumul * chaque)

Console.WriteLine(max.ToString() + vbNewLine + min.ToString() + vbNewLine +
count.ToString() + vbNewLine + sum.ToString() + vbNewLine + average.ToString() +
vbNewLine + produit.ToString())
```

1.4.5 Les méthodes « All », « Any » et « Contains »

Les méthodes « All », « Any » et « Contains » sont des éléments qui permettent un contrôle rapide sur une liste et qui renvoient des booléens.

La méthode « All » permet par exemple de poser une expression traitant sur la totalité d'une liste.

La méthode « Any », quant à elle, permet de faire une vérification sur n'importe quelle donnée de la liste. Par exemple, dans le code ci-dessous, nous souhaitons vérifier si un utilisateur a pour logiciel favori le logiciel avec l'id « 4 ».

Enfin, la méthode « Contains » permet de vérifier si la liste en question contient l'élément passé en paramètre.

C#	VB.Net
<pre>Console.WriteLine(users.All(c=>c.age>=18)); // retourne True Console.WriteLine(users.Any(c=>c.logiciel==4)); // True Console.WriteLine(users.Contains(new person("Dupont","Albert",25,1000, 3))); // False</pre>	<pre>Console.WriteLine(users.All(Function(c) c.age >= 18)) ' retourne True Console.WriteLine(users.Any(Function(c) c.logiciel = 4)) ' True Console.WriteLine(users.Contains(New person("Dupont", "Albert",25,1000, 3))) ' False</pre>

Remarque : La méthode « Contains » compare les objets par leurs références de ce fait même si les attributs des objets sont égaux, si ce n'est pas le même objet, le booléen retourné sera false.

1.4.6 La méthode « Distinct »

La méthode « Distinct » est un élément qui permet d'éviter les entrées en double dans des ensembles de données. Il est possible d'utiliser cette méthode pour filtrer ou éviter des calculs inutiles. En effet, si par exemple nous devons faire un calcul imposant dans un « foreach », il peut être avantageux de retirer les entrées en double.

C#	VB.Net
<pre>var requete = from u in users.Select(p=>p.age).Distinct() select u.Distinct(); int [] list = { 2,2,5,4,2,5,4,6,1,3,3, 3,1,8,1,4,5,5,6,4,7,8,5,3,2,1,5}; var requete2 = list.Distinct();</pre>	<pre>Dim requete = From u _ In users.Select(Function(p As person) p.age) _ Select u Distinct Dim list As Integer() = {2, 2, 5, 4, 2, 5, 4, 6, 1, 3, 3, 7, 2, 4, 3, 5, 6, 4, 7, 8, 5, 3, 2, 1, 5} Dim requete2 = list. Distinct()</pre>

Vous pourrez voir dans la requête en C# que la méthode « Distinct » a été utilisée de différentes manières. Dans l'extrait de code en VB.Net, il a été utilisé le mot-clé « Distinct » afin de vous indiquer la possibilité de son utilisation dans une requête.

1.4.7 Les méthodes « Union » et « Intersect »

Les méthodes « Union » et « Intersect » sont deux méthodes qui permettent de combiner des listes. La méthode « Union » permet de « coller » ces listes ensembles alors que la méthode « Intersect » prend seulement leurs objets communs.

C#	VB.Net
<pre>var requete = from u in users select u.age ; int [] list = { 2,2,5,4,2,5,4,6,1,3,3,7,2,4,52,6,25,4,3,1,8 ,1,4,5,5,6,4,7,8,5,3,2,1,5}; var requete2 = list.Intersect(requete).Distinct(); requete = requete2.Union(requete).Distinct();</pre>	<pre>Dim requete = From u In users _ Select u.age Dim list As Integer() = {2, 2, 5, 4, 2, 5, 4, 6, 25, 52, 1, 3, 3, 7, 4, 5, 5, 6, 4, 7, 8, 5, 3, 2, 1, 5} Dim requete2 = list.Intersect(requete).Distinct() requete = requete2.Union(requete).Distinct()</pre>

1.4.8 L'instruction « Join »

Comme en SQL, l'instruction « Join » sert à faire fusionner deux tables un peu à la manière de la méthode « Union » mais cette fusion s'opèrera sur les colonnes des tables et non comme un ajout de lignes à celles existantes. Il est ainsi possible d'associer, grâce à l'id des logiciels préférés de nos utilisateurs, le nom des personnages et le nom des logiciels.

C#	VB.Net
<pre>var requete = from p in users join s in softs on p.logiciel equals s.reference select new { mr = p.nom, soft = s.nom }; </pre>	<pre>Dim requete = From p In users _ Join s In softs _ On p.logiciel Equals s.reference _ Select New With { _ .mr = p.nom, _ .soft = s.nom} </pre>

Remarque : Notez bien que les données de la table qu'apporte l'instruction « Join » est spécifiée avant le mot-clé « equals ».

Il est aussi possible de faire l'équivalent de l'instruction « Join » avec deux clauses « From » et une clause « Where ». Cette dernière permet de définir la donnée de jonction.

C#	VB.Net
<pre>var requete = from user in users from soft in softs where user.logiciel == soft.reference select new { soft=soft, usr=user}; </pre>	<pre>Dim requete2 = From user In users _ From soft In softs _ Where soft.reference = user.logiciel Select New With { .soft = soft, .usr = user} </pre>

1.4.9 L'instruction « Let »

« Let » est un mot-clé qui permet de créer une variable locale en plein milieu d'une requête pour chaque élément de la liste, et permet de ce fait de créer des variables contenant éventuellement des requêtes. Nous avons du coup la possibilité d'imbriquer ces requêtes. Vous

pouvez avoir un aperçu avancé de l'utilisation de cette instruction sur ce blog MSDN : [EN: Une requête conséquente avec "Let" \(C#\)](#). Cet exemple montre comment il est possible d'abuser de « Let » mais aussi à quel point cela peut être pratique.

L'utilisation dans cette requête n'est pas réellement nécessaire, nous aurions pu directement écrire le contenu de la variable déclarée dans le « Select ». La variable « chaine » aurait pu contenir le résultat d'une requête LINQ complexe contenant elle-même des variables locales.

C#	VB.Net
<pre>var requete = from p in users let chaine = "\t" + ((p.salaire)/2).ToString() + "\t" join s in softs on p.logiciel equals s.reference select new { mr = p.nom , budget = chaine , soft = s.nom };</pre>	<pre>Dim requete = From p In users _ Let chaine = vbTab + (p.salaire / 2).ToString() + vbTab _ Join s In softs On p.logiciel Equals s.reference _ Select New With { .mr = p.nom, .budget = chaine, .soft = s.nom}</pre>

Remarque : Attention toutefois, cette instruction peut utiliser beaucoup de mémoire si elle est utilisée abusivement !

1.4.10 Méthodes « Range », « Repeat », « Empty » et « DefaultIfEmpty »

La méthode « Range » permet de créer des listes simplement avec une incrémentation de 1 sur un nombre d'éléments prédéfinis. Cette méthode prend en argument une valeur initiale et un nombre d'éléments. Elle peut servir, si on crée un « Enumerable », à ajouter éventuellement des clés ou pour faire des calculs.

La méthode « Repeat » sert à générer une liste ayant un nombre défini d'occurrences d'une même entrée fournie en argument.

La méthode « Empty » nous paraît moins utile car elle retourne une liste vide. Ce sont des méthodes statiques de la classe « Enumerable ».

La méthode « DefaultIfEmpty » permet d'éviter une erreur à l'affichage. En effet, cette méthode est pratique lorsque les conditions ne garantissent pas un contenu.

C#

```
var requete = from p in users
               let chaine = from i in Enumerable.Range(18,10)
                           where p.age <= i
                           select i
               join s in softs on p.logiciel equals s.reference
               select new {
                   mr = p.nom ,
                   soft = s.nom,
                   index = chaine.DefaultIfEmpty(0).Sum()
               };

foreach (var e in requete)
{
    Console.WriteLine(e.mr+e.soft+e.index);
    Console.WriteLine();
}
```

VB.Net

```
Dim requete = From p In users _
               Let chaine = (From i In Enumerable.Range(18, 10) _
                           Where p.age <= i _
                           Select i) _
               Join s In softs On p.logiciel Equals s.reference _
               Select New With {
                   .mr = p.nom,
                   .soft = s.nom,
                   .index = chaine.DefaultIfEmpty(0).Sum() }

For Each e In requete
    Console.WriteLine(e.mr + e.soft + e.index.ToString())
    Console.WriteLine()
Next
```

Remarque : Les parenthèses qui isolent la sous-requête dans l'extrait de code en VB.NET sont nécessaires afin que le compilateur comprenne qu'il s'agit d'un bloc de code qu'il faut exécuter sans se préoccuper du code à l'extérieur. Néanmoins, une fois ce bloc de code exécuté, celui-ci renvoie un élément résultant au code « parent ». Etant donné que les parenthèses améliorent la lisibilité, il vous est également possible de les ajouter en C#.

1.4.11 Les méthodes « FirstOrDefault » et « LastOrDefault »

Le dernier exemple pourrait parfaitement illustrer l'utilisation de la méthode « FirstOrDefault ». La variable « index » aurait alors eu pour valeur « Default » ou l'âge de la personne dans le cas où elle aurait moins de 28 ans. Pour ce faire, il suffit de remplacer « `index = chaine.DefaultIfEmpty(0).Sum()` » par « `index = chaine.FirstOrDefault()` ». Il est possible de mettre une expression lambda tout comme une clause « Where » en argument de la fonction afin d'ajouter un filtre. Ainsi elle retournera la première expression trouvée dans l'élément où le prédicat retourne vrai.

La méthode « LastOrDefault » fonctionne de la même manière mais dans l'autre sens, c'est-à-dire qu'il prend la dernière expression pour laquelle le prédicat est vrai.

Voici ci-dessous un exemple d'utilisation d'une fonction lambda avec la méthode « LastOrDefault » :

- « `index = chaine.LastOrDefault(arg => arg == 25)` » en C#.
- « `index = chaine.LastOrDefault(Function(arg As Integer) arg == 25)` » en VB.Net

La méthode « LastOrDefault » retournera 25 pour tous les utilisateurs âgés de 25 ans ou moins.

***Remarque :** Vous aurez un exemple de « LastOrDefault » avec un prédicat en paramètre dans l'exemple suivant.*

1.4.12 Les méthodes « Take » et « Skip »

Cette partie va vous permettre de connaître les méthodes « Take », « Skip », « TakeWhile » et « SkipWhile ». Nous pouvons essayer de déterminer leur utilité en les traduisant de l'anglais : « Take » signifie « Prendre » et « Skip » peut être traduit par « Sauter ».

Les méthodes « Take » et « Skip » sont relativement simple. En effet, la méthode « Take » vous permettra de limiter votre requête à un certain nombre de lignes que vous lui aurez passé en paramètre. Alors que la méthode « Skip », elle, vous permettra de « sauter » ce nombre de ligne que vous lui aurez indiqué.

Les méthodes « TakeWhile » et « SkipWhile » ont le quasiment le même fonctionnement que, respectivement, les méthodes « Take » et « Skip » à la seule différence que le nombre de ligne prises en compte ou ignorées seront fonction de la condition passée en paramètre de celles-ci.

```
C#
var requete = (
    from p in users.Take(4)
    let chaine = (from i in Enumerable.Range(0,100).SkipWhile(arg=>arg<18)
                  .TakeWhile(arg=>arg<50)
                  where p.age <= i
                  select i)
    select new {
        mr = p.nom ,
        index = chaine.LastOrDefault(arg => arg == p.age)
    }).Skip(1);
```

```
VB.Net
Dim requete = From p In users _
               Let chaine = (From i In Enumerable.Range(18, 10) _
                             Where p.age <= i _
                             Select i) _
               Join s In softs On p.logiciel Equals s.reference _
               Select New With {
                   .mr = p.nom,
                   .soft = s.nom,
                   .index = chaine.DefaultIfEmpty(0).Sum() }

For Each e In requete
    Console.WriteLine(e.mr + e.soft + e.index.ToString())
    Console.WriteLine()
Next
```

1.4.13 Les méthodes « OrderBy », « OrderByDescending », « ThenBy », « ThenByDescending » et « Reverse »

Dans le but de rendre les méthodes « Take » et « Skip » plus efficaces, il peut être intéressant de trier les données. En effet, la méthode « Take » pourrait par exemple traiter les objets dont un attribut serait le plus grand.

Afin de réaliser ce tri, nous avons à disposition les méthodes « OrderBy » et « OrderByDescending », chacune de ces méthodes ordonnant respectivement les données des résultats de la requête dans l'ordre alphabétique croissant et décroissant.

Les méthodes « ThenBy » et « ThenByDescending » ont les mêmes propriétés qu'« OrderBy » et « OrderByDescending » à la seule différence que les méthodes « ThenBy » permettent d'étendre le tri à un autre paramètre. L'exemple de code plus bas vous permettra de mieux comprendre ceci.

Enfin, la méthode « Reverse » permet simplement d'inverser le résultat d'une requête. De ce fait, le dernier élément retourné par la requête sera donc en réalité le premier de la liste des éléments renvoyée.

```
C#
var requete = (from p in users
               join s in softs on p.logiciel equals s.reference
               select new { us = p , so = s }).OrderBy(a=>a.so.compagnie)
               .ThenByDescending(a=>a.us.age);
```

```
VB.Net
Dim requete = (From p In users _
               Join s In softs On p.logiciel Equals s.reference _
               Select New With {.us = p, .so = s}).OrderBy(Function(a) a.so.compagnie)
               .ThenByDescending(Function(a) a.us.age).Reverse()
```

Remarque : La méthode « Reverse » se doit d'être la dernière à être invoquée sur une requête. De plus, dans l'extrait de code ci-dessus, elle n'a été utilisée que pour l'extrait en VB.Net mais nous aurions pu sans problème l'utiliser en C#.

1.4.14 Les méthodes « ElementAt » et « ElementOrDefault »

« ElementAt » est une méthode qui attend en paramètre un entier. Celui-ci lui permettra de rechercher une valeur à la position indiquée par cet entier. Cet index fonctionne comme ceux des tableaux ou des listes : pour récupérer le premier élément, il faut renseigner l'entier « 0 ». Si ce nombre est négatif ou supérieur au nombre d'éléments présents dans la source, une exception « *ArgumentOutOfRangeException* » est levée.

La méthode « ElementOrDefault » permet d'éviter cette levée d'exception. En effet, celle-ci fonctionne exactement comme « ElementAt » mais retourne la valeur « null » si l'entier passé en paramètre est erroné.

```
C#
software requete = (from s in softs
                    where s.compagnie == "Microsoft"
                    select s ).ElementAt(2);
```

```
VB.Net
Dim requete As software = (From s In softs _
                          Where s.compagnie = "Microsoft" _
                          Select s).ElementAt(2)
```

1.4.15 La méthode « Single » et « SingleOrDefault »

La méthode « Single » permet de récupérer sur une source un élément unique. Elle prend en paramètre une condition qui lui permettra de récupérer une valeur unique la vérifiant. Si aucune condition ne lui est donnée et que la source ne contient qu'un élément, alors c'est celui-ci qui est retourné. En revanche, si la source contient plus d'un élément, une exception « *InvalidOperationException* » sera levée. Cette erreur apparaîtra également si la condition ne permet la sélection d'aucun élément, mais aussi si celle-ci provoque le retour de plus d'un élément.

La méthode « SingleOrDefault » fonctionne exactement de la même manière que la méthode « Single » à la différence qu'elle ne lèvera pas d'exception en cas d'erreur. En effet, cette méthode retournera une valeur nulle (« null ») à la place évitant au programme de cesser de fonctionner.

Remarque : La méthode « SingleOrDefault » lèvera tout de même une exception si plusieurs éléments répondent aux critères de la condition. Il sera alors préférable d'utiliser les blocs « try { ... } catch { ... } » afin de s'assurer qu'une levée d'exception n'empêche pas le programme de fonctionner.

C#	VB.Net
<pre>software requete = (from s in softs select s) .Single(s=>s.reference==3);</pre>	<pre>Dim requete As software = (From s In softs _ Select s) _ .Single(Function(s) s.reference==3)</pre>

1.4.16 La méthode « Concat »

La méthode « Concat » permet, comme son nom le suggère, de concaténer deux éléments entre eux. Ainsi, l'élément passé en paramètre sera « collé » à l'élément sur lequel cette méthode a été invoquée. Cependant, vous ne pourrez concaténer à l'aide de cette méthode que des éléments du même type.

C#	VB.Net
<pre>List<person> users2 = new List<person>(); users2.Add(new person("nom" , "prenom", 22, 1500,5)); var AllUsers = users2.Concat(users); users2.Add(new person("nom2", "prenom2", 22, 1500, 5));</pre>	<pre>Dim users2 As List(Of person) = New List(Of person)() users2.Add(New person("nom", "prenom", 22, 1500, 5)) Dim AllUsers = users2.Concat(users) users2.Add(New person("nom2", "prenom2", 22, 1500, 5))</pre>

Remarque : A l'exécution avec l'affichage, vous pourrez apercevoir que « nom2 » et « prenom2 » apparaissent.

1.4.17 La méthode « OfType »

La méthode « OfType » est un filtre permettant de ne sélectionner que des éléments d'un même type dans une source contenant des éléments de types différents. « OfType » étant une méthode générique, le type désiré devra être précisé après le nom de la fonction et entre chevrons.

Dans l'extrait de code suivant, nous souhaitons seulement sélectionner dans un fichier XML les nœuds de type « XElement » :

C#	VB.Net
<pre>foreach (var elements in xml.Nodes().OfType<XElement>()) { Console.WriteLine(o); }</pre>	<pre>For Each element In Xml.Nodes().OfType(Of XElement)() Console.WriteLine(element) Next</pre>

Remarque : Pour plus d'informations quant à l'utilisation de LINQ avec les fichiers XML, reportez-vous au cours « LINQ to XML ».

1.4.18 La méthode « SelectMany »

La méthode « SelectMany » permet d'énumérer la séquence source et fusionne les éléments résultant afin de retourner une unique séquence énumérable.

Si nous faisons l'analogie avec la méthode « Concat », celle-ci a le même effet mais « SelectMany » nous évite une utilisation en cascade pour obtenir le même résultat.

L'extrait de code suivant illustre cette notion :

C#

```
int[][] list = new int[][] {
    new int[] { 1, 2, 5, 6, 7 },
    new int[] { 4, 5, 4, 7, 8, 5 },
    new int[] { 4, 7, 8, 5, 4, 5 },
    new int[] { 1, 4, 9, 3 }
};

var requete = list.SelectMany(s => s);

var requete2 = list[0].Concat(list[1].Concat(list[2].Concat(list[3])));
```

VB.Net

```
Dim list As Integer()() = New Integer()()
    {New Integer() {1, 2, 5, 6, 7},
      New Integer() {4, 5, 4, 7, 8, 5},
      New Integer() {4, 7, 8, 5, 4, 5},
      New Integer() {1, 4, 9, 3}}

Dim AllUsers = list.SelectMany(Function(s) s)

Dim AllUsers2 = list(0).Concat(list(1).Concat(list(2).Concat(list(3))))
```

La méthode « SelectMany » permet également d'utiliser plus simplement des méthodes anonymes pour la clause « Select ».

1.4.19 L'instruction « Group [...] By [...] »

L'instruction « Group [...] By [...] » permet de réunir les éléments d'une table qui ont une entrée (colonne) commune. Par exemple dans notre table de logiciels si nous groupons par société d'édition de logiciels, nous pourrions afficher l'éditeur puis tous les logiciels que celui-ci aura produit.

Le mot clef « into » permet d'avoir une référence sur la « fusion » ainsi créée avec « Group [...] By [...] » afin de pouvoir agir dessus par la suite comme le montre l'exemple suivant.

C#

```
IEnumerable<int> requete = from s in softs
    group s by s.compagnie
    into g
    select g.Sum(arg=>arg.prix);
```

VB.Net

```
Dim requete As IEnumerable(Of Integer) = From s In softs _
    Group s By s.compagnie _
    Into somme = Sum(s.prix) _
    Select somme
```


Remarque : Il existe également une méthode « GroupBy() » fonctionnant de la même manière que l'instruction. Elle prend en paramètre la valeur clé par laquelle la « fusion » doit être faite. La ligne « group s by s.compagnie » aurait alors pour équivalent « GroupBy(s => s.compagnie) »

1.4.20 L'instruction « Group Join »

L'instruction « Group Join » permet de réaliser une action similaire à un « LEFT OUTER JOIN » ou « RIGHT OUTER JOIN » dans le langage SQL. Elle permet de lier de manière hiérarchique deux tables selon une colonne spécifiée pour chaque table et de grouper les résultats en cas de plusieurs correspondances dans la seconde table par rapport à une donnée dans la première. L'exemple qui suit rend plus explicite cette notion.

En C#, le « Group Join » ne laisse pas apparaître des mots-clés. En effet, comme l'indique l'extrait de code ci-dessous, il faudra utiliser les mots-clés « join » et « into ». Cela aura le même résultat qu'un « Join » classique.

```
C#
var requete = from soft in softs
              join user in users on soft.reference equals user.logiciel into userlist
              select new { soft = soft.nom, usr = userlist };

foreach (var elements in requete)
{
    Console.WriteLine(elements.soft);
    foreach (var usr in elements.usr)
    {
        Console.WriteLine("    -" + usr.nom);
    }
}

Console.Read();
```

```
VB.Net
Dim requete = From soft In softs _
              Group Join user In users On soft.reference Equals user.logiciel Into userlist =
Group _
              Select New With {.soft = soft.nom, .usr = userlist}

For Each elements In requete
    Console.WriteLine(elements.soft)
    For Each usr In elements.usr
        Console.WriteLine("    -" + usr.nom)
    Next
Next

Console.Read()
```

Dans l'exemple ci-dessus, nous récupérons les logiciels de notre liste dans la variable « soft ». Ensuite, nous faisons un « Group Join » avec les utilisateurs de la liste « users » en utilisant la variable « user ». Nous spécifions, pour continuer, que la jointure se fera sur l'attribut « reference » du logiciel « soft » et sur le « logiciel » de l'utilisateur représenté par la variable « user ». Afin de l'utiliser par la suite, nous spécifions la variable « userlist » qui représente cette jointure. Enfin, nous sélectionnons avec un objet anonyme le nom des logiciels et la liste des utilisateurs.

Pour afficher un résultat dans l'invite de commande, nous traitons chaque élément de la requête en affichant en premier lieu le nom du premier logiciel puis la liste de ses utilisateurs et cela, pour tous les logiciels de la liste « softs ».

Le résultat affiché est le suivant :

```
Windows
  -Gates
  -Ballmer
Office
firefox
  -Dupont
  -Narbonne
Emacs
  -Stallman
Visual Studio
```

***Remarque :** En VB.net, chaque variable de type anonyme doit être précédée d'un point « . ». C'est pourquoi nous écrivons dans le « Select » « .soft » et « .usr ». Il existe également une méthode « GroupJoin() » prenant en paramètre les mêmes valeurs que celles utilisées ci-dessus.*

1.4.21 Les méthodes « ToList », « ToArray » et « ToDictionary »

Les méthodes « ToList » et « ToArray » sont des outils de conversion. Elles permettent plus précisément de convertir un objet de type « IEnumerable<> » en respectivement une liste générique ou un tableau. Lorsqu'une de ces méthodes est appliquée à un objet, le résultat restera inchangé même si la source de la conversion change.

La méthode « ToDictionary » permet de créer une liste spéciale composée d'éléments représentés par une clé que nous devons préciser : elle prend en paramètre les expressions qui permettront de construire le dictionnaire, de le remplir avec des éléments associés à des clés. Le typage du dictionnaire n'est pas obligatoire, mais il permet de s'assurer qu'il contient bien des objets du type précisé.

C#

```
var requete = softs.ToDictionary<software, int>(a => a.reference);

foreach(var elements in requete)
{
    Console.WriteLine("Cle :");
    Console.WriteLine(elements.Key.ToString());
    Console.WriteLine("Valeur de la cle : ");
    Console.WriteLine(elements.Value.ToString());
}

var array = new int[] {10, 1, 2, 4, 2, 15, 4};
List<int> intListe = array.Distinct().ToList();
array.SetValue(3,3);

Console.WriteLine("Valeurs dans la liste après modification du tableau :");
foreach(var e in intListe)
{
    Console.WriteLine(e.ToString());
}

Console.Read();
```

VB.Net

```
Dim requete = softs.ToDictionary(Function(a) a.reference)

For Each elements In requete
    Console.WriteLine("Cle :")
    Console.WriteLine(elements.Key.ToString())
    Console.WriteLine("Valeur de la cle : ")
    Console.WriteLine(elements.Value.ToString())
Next

Dim Array = New Int32() {10, 1, 2, 4, 2, 15, 4}
Dim intListe As List(Of Int32) = Array.Distinct().ToList()
Array.SetValue(3, 3)

Console.WriteLine("Valeurs dans la liste après modification du tableau :")
For Each e In intListe
    Console.WriteLine(e.ToString())
Next

Console.Read()
```

1.5 Exemples de requêtes

1.5.1 Arborescence des fichiers

Grâce à LINQ, il est possible de faire des requêtes dans plusieurs types de sources (objets, SQL, XML, etc.). Mais il est aussi possible d'utiliser LINQ avec les arborescences de répertoires de fichiers. Pour plus d'information à ce sujet, un article en français est disponible à cette adresse : <http://msdn.microsoft.com/fr-fr/library/bb397911.aspx>.

2 Conclusion

Ce chapitre aborde les notions essentielles sur LINQ to Objects mais il est possible d'aller beaucoup plus loin que les exemples réalisés dans ce cours. Avec de la pratique, vous devriez être capable de réaliser des programmes complexes.

Nous avons également vu dans ce chapitre des méthodes qui seront également disponibles pour les autres LINQ, elles sont essentielles pour une bonne compréhension des cours suivants.

Plus d'informations sur MSDN : <http://msdn.microsoft.com/fr-fr/library/system.linq.aspx>