LINQ signifie *Language INtegrated Query*. C'est un ensemble d'extensions du langage permettant de faire des requêtes sur des données en faisant abstraction de leur type. Il permet d'utiliser facilement un jeu d'instructions supplémentaires afin de filtrer des données, faire des sélections, etc. Il existe plusieurs domaines d'applications pour LINQ :

- *Linq To Entities* ou *Linq To SQL* qui utilisent ces extensions de langage sur les bases de données.
- *Linq To XML* qui utilise ces extensions de langage pour travailler avec les fichiers XML.
- *Linq To Object* qui permet de travailler avec des collections d'objets en mémoire.

L'étude de LINQ nécessiterait un livre en entier, aussi nous allons nous concentrer sur la partie qui va le plus nous servir en tant que débutant et qui va nous permettre de commencer à travailler simplement avec cette nouvelle syntaxe, à savoir *Linq To Object*. Il s'agit d'extensions permettant de faire des requêtes sur les objets en mémoire et notamment sur toutes les listes ou collections. En fait, sur tout ce qui implémente IEnumerable<>.

Les requêtes Linq proposent une nouvelle syntaxe permettant d'écrire des requêtes qui ressemblent de loin à des requêtes SQL. Pour ceux qui ne connaissent pas le SQL, il s'agit d'un langage permettant de faire des requêtes sur les bases de données.

Pour utiliser ces requêtes, il faut ajouter l'espace de noms adéquat, à savoir :

```
1 using System.Linq;
```

Ce using est en général inclus par défaut lorsqu'on crée un nouveau fichier. Jusqu'à maintenant, si nous voulions afficher les entiers d'une liste d'entiers qui sont strictement supérieur à 5, nous aurions fait :

```
1
2 List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
3 foreach (int i in liste)
4 {
5     if (i > 5)
6     {
7         Console.WriteLine(i);
8     }
9 }
10
```

Grâce à *Linq To Object*, nous allons pouvoir filtrer en amont la liste afin de ne parcourir que les entiers qui nous intéressent, en faisant :

```
1
2 List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
3 IEnumerable<int> requeteFiltree = from i in liste
4                                   where i > 5
5                                   select i;
6 foreach (int i in requeteFiltree)
7 {
8     Console.WriteLine(i);
9 }
10
```

Qui donnera :

```
6

9

15

8
```

Nous avons ici créé une requête Linq qui contient des mots-clés, comme from, in, where et select. Ici, cette requête veut dire que nous partons (from) de la liste « liste » en analysant chaque entier de la liste dans (in) la variable i où (where) i est supérieur à 5. Dans ce cas, il est sélectionné comme faisant partie du résultat de la requête grâce au mot-clé select, qui fait un peu office de return.

Cette requête renvoie un IEnumerable<int>. Le type générique est ici le type int car c'est le type de la variable i qui est retournée par le select. Le fait de renvoyer un IEnumerable<> va permettre potentiellement de réutiliser le résultat de cette requête pour un filtre successif ou une expression différente. En effet, Linq travaille sur des IEnumerable<> … Nous pourrions par exemple ordonner cette liste par ordre croissant grâce au mot-clé orderby. Cela donnerait :

```
1
2  List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
3  IEnumerable<int> requeteFiltree = from i in liste
4                                    where i > 5
5                                    select i;
6  IEnumerable<int> requeteOrdonnee = from i in requeteFiltree
7                                     orderby i
8                                     select i;
9  foreach (int i in requeteOrdonnee)
10 {
11     Console.WriteLine(i);
12 }
13
```

qui pourrait également s'écrire en une seule fois avec :

```
1
2  List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
3  IEnumerable<int> requete = from i in liste
4                             where i > 5
5                             orderby i
6                             select i;
7  foreach (int i in requete)
8  {
9      Console.WriteLine(i);
10 }
11
```

Et nous aurons :

```
6

8

9

15
```

L'intérêt est que grâce à ces syntaxes, nous pouvons combiner facilement plusieurs filtres et construire des requêtes plus ou moins complexes.
Par exemple, imaginons des clients :

```
public class Client
{
    public int Identifiant { get; set; }
    public string Nom { get; set; }
    public int Age { get; set; }
}
```

que l'on souhaiterait savoir majeurs, puis triés par Age puis par Nom, nous pourrions faire :

```
List<Client> listeClients = new List<Client>
{
    new Client { Identifiant = 1, Nom = "Nicolas", Age = 30},
    new Client { Identifiant = 2, Nom = "Jérémie", Age = 20},
    new Client { Identifiant = 3, Nom = "Delphine", Age = 30},
    new Client { Identifiant = 4, Nom = "Bob", Age = 10}
};

IEnumerable<string> requete = from client in listeClients
                              where client.Age > 18
                              orderby client.Age, client.Nom
                              select client.Nom;

foreach (string prenom in requete)
{
    Console.WriteLine(prenom);
}
```

Ce qui donnera :

```
Jérémie

Delphine

Nicolas
```

Notez ici que mon select « renvoie » le nom du client, qui est un string. Il est donc normal que ma requête renvoie un IEnumerable<string> car j'ai choisi qu'elle ne sélectionne que les noms, qui sont de type string.
Il est assez fréquent de construire des objets anonymes à l'intérieur d'une requête. Par exemple, plutôt que de renvoyer uniquement le nom du client, je pourrais également renvoyer l'Age, mais comme je n'ai pas besoin de l'identifiant, je peux créer un objet anonyme juste avec ces deux propriétés :

```
List<Client> listeClients = new List<Client>
{
    new Client { Identifiant = 1, Nom = "Nicolas", Age = 30},
    new Client { Identifiant = 2, Nom = "Jérémie", Age = 20},
    new Client { Identifiant = 3, Nom = "Delphine", Age = 30},
    new Client { Identifiant = 4, Nom = "Bob", Age = 10},
};

var requete = from client in listeClients
                              where client.Age > 18
                              orderby client.Age , client.Nom
                              select new { client.Nom, client.Age  };
foreach (var obj in requete)
{
    Console.WriteLine("{0} a {1} ans", obj.Nom, obj.Age);
}
```

Et nous aurons :

```
Jérémie a 20 ans

Delphine a 30 ans

Nicolas a 30 ans
```

Mon objet anonyme contient ici juste une propriété Nom et une propriété Age. À noter que je suis obligé à ce moment-là d'utiliser le mot-clé var pour définir la requête, car je n'ai pas de type à donner à cette requête. De même, dans le foreach je dois utiliser le mot-clé var pour définir le type anonyme.
Les requêtes peuvent être de plus en plus compliquées, comme faisant des jointures. Par exemple, rajoutons une classe Commande :

```
public class Commande
{
    public int Identifiant { get; set; }
    public int IdentifiantClient { get; set; }
    public decimal Prix { get; set; }
}
```

Je peux créer des commandes pour des clients :

```
List<Client> listeClients = new List<Client>
{
    new Client { Identifiant = 1, Nom = "Nicolas", Age = 30},
    new Client { Identifiant = 2, Nom = "Jérémie", Age = 20},
    new Client { Identifiant = 3, Nom = "Delphine", Age = 30},
    new Client { Identifiant = 4, Nom = "Bob", Age = 10},
};

List<Commande> listeCommandes = new List<Commande>
{
```

```
    new Commande{ Identifiant = 1, IdentifiantClient = 1, Prix = 150.05M},
    new Commande{ Identifiant = 2, IdentifiantClient = 2, Prix = 30M},
    new Commande{ Identifiant = 3, IdentifiantClient = 1, Prix = 99.99M},
    new Commande{ Identifiant = 4, IdentifiantClient = 1, Prix = 100M},
    new Commande{ Identifiant = 5, IdentifiantClient = 3, Prix = 80M},
    new Commande{ Identifiant = 6, IdentifiantClient = 3, Prix = 10M},
};
```

Et grâce à une jointure, récupérer avec ma requête le nom du client et le prix de sa commande :

```
var liste = from commande in listeCommandes
            join client in listeClients on commande.IdentifiantClient equals client.Id
            select new { client.Nom, commande.Prix };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} a payé {1}", element.Nom, element.Prix);
}
```

Ce qui donne :

```
Le client Nicolas a payé 150,05

Le client Jérémie a payé 30

Le client Nicolas a payé 99,99

Le client Nicolas a payé 100

Le client Delphine a payé 80

Le client Delphine a payé 10
```

On utilise le mot-clé join pour faire la jointure entre les deux listes puis on utilise le mot-clé on et le mot-clé equals pour indiquer sur quoi on fait la jointure.
À noter que ceci peut se réaliser en imbriquant également les from et en filtrant sur l'égalité des identifiants clients :

```
var liste = from commande in listeCommandes
            from client in listeClients
            where client.Identifiant == commande.IdentifiantClient
            select new { client.Nom, commande.Prix };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} a payé {1}", element.Nom, element.Prix);
}
```

Il est intéressant de pouvoir regrouper les objets qui ont la même valeur. Par exemple pour obtenir toutes les commandes, groupées par client, on ferra :

```
var liste = from commande in listeCommandes
            group commande by commande.IdentifiantClient;

foreach (var element in liste)
{
    Console.WriteLine("Le client : {0} a réalisé {1} commande(s)", element.Key, eleme
    foreach (Commande commande in element)
    {
        Console.WriteLine("\tPrix : {0}", commande.Prix);
    }
}
```

Ici, cela donne :

```
Le client : 1 a réalisé 3 commande(s)
    Prix : 150,05
    Prix : 99,99
    Prix : 100
Le client : 2 a réalisé 1 commande(s)
    Prix : 30
Le client : 3 a réalisé 2 commande(s)
    Prix : 80
    Prix : 10
```

Il est possible de cumuler le group by avec notre jointure précédente histoire d'avoir également le nom du client :

```
var liste = from commande in listeCommandes
            join client in listeClients on
                commande.IdentifiantClient equals client.Identifiant
                group commande by new {commande.IdentifiantClient, client.Nom};

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} ({1}) a réalisé {2} commande(s)", element.Key.N
    foreach (Commande commande in element)
    {
        Console.WriteLine("\tPrix : {0}", commande.Prix);
    }
}
```

Et nous obtenons :

```
Le client Nicolas (1) a réalisé 3 commande(s)
    Prix : 150,05
```

```
    Prix : 99,99

    Prix : 100

Le client Jérémie (2) a réalisé 1 commande(s)

    Prix : 30

Le client Delphine (3) a réalisé 2 commande(s)

    Prix : 80

    Prix : 10
```

À noter que le group by termine la requête, un peu comme le select. Ainsi, si l'on veut sélectionner quelque chose après le group by, il faudra utiliser le mot-clé into et la syntaxe suivante : Ce qui donnera :

```
var liste = from commande in listeCommandes
            join client in listeClients on
                  commande.IdentifiantClient equals client.Identifiant
            group commande by
               new {commande.IdentifiantClient, client.Nom} into commandesGroupees
            select
                new
                {
                    commandesGroupees.Key.IdentifiantClient,
                    commandesGroupees.Key.Nom,
                    NombreDeCommandes = commandesGroupees.Count()
                };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} ({1}) a réalisé {2} commande(s)", element.Nom,
}
```

Avec pour résultat :

```
Le client Nicolas (1) a réalisé 3 commande(s)

Le client Jérémie (2) a réalisé 1 commande(s)

Le client Delphine (3) a réalisé 2 commande(s)
```

L'intérêt d'utiliser le mot-clé into est également de pouvoir enchainer avec une autre jointure ou autre filtre permettant de continuer la requête.
Il est également possible d'utiliser des variables à l'intérieur des requêtes grâce au mot-clé let. Cela va nous permettre de stocker des résultats temporaires pour les réutiliser ensuite :

```
var liste = from commande in listeCommandes
            join client in listeClients on
                 commande.IdentifiantClient equals client.Identifiant
            group commande by
               new {commande.IdentifiantClient, client.Nom} into commandesGroupees
            let total = commandesGroupees.Sum(c => c.Prix)
            where total > 50
```

```
            orderby total
            select new
                {
                    commandesGroupees.Key.IdentifiantClient,
                    commandesGroupees.Key.Nom,
                    NombreDeCommandes = commandesGroupees.Count(),
                    PrixTotal = total
                };

foreach (var element in liste)
{
    Console.WriteLine("Le client {0} ({1}) a réalisé {2}
         commande(s) pour un total de {3}",
         element.Nom,
         element.IdentifiantClient,
         element.NombreDeCommandes,
         element.PrixTotal);
}
```

Par exemple, ici j'utilise le mot-clé let pour stocker le total d'une commande groupée dans la variable total (nous verrons la méthode Sum() un tout petit peu plus bas), ce qui me permet ensuite de filtrer avec un where pour obtenir les commandes dont le total est supérieur à 50 et de les trier par ordre de prix croissant.

Ce qui donne :

```
Le client Delphine (3) a réalisé 2 commande(s) pour un total de 90

Le client Nicolas (1) a réalisé 3 commande(s) pour un total de 350,04
```

Nous allons nous arrêter là pour cet aperçu des requêtes LINQ. Nous avons pu voir que le C# dispose d'un certain nombre de mots-clés qui permettent de manipuler nos données de manière très puissante mais d'une façon un peu inhabituelle.
Cette façon d'écrire des requêtes LINQ s'appelle en anglais la « sugar syntax », que l'on peut traduire par « sucre syntaxique ». Il désigne de manière générale les constructions d'un langage qui facilitent la rédaction du code sans modifier l'expressivité du langage.
Voyons à présent ce qu'il y a derrière cette jolie syntaxe.

En fait, toute la sugar syntax que nous avons vue précédemment repose sur un certain nombre de méthodes d'extensions qui travaillent sur les types IEnumerable<T>. Par exemple, la requête suivante :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requeteFiltree = from i in liste
                                  where i > 5
                                  select i;
foreach (int i in requeteFiltree)
{
    Console.WriteLine(i);
}
```

```
8
```

s'écrit véritablement :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
IEnumerable<int> requeteFiltree = liste.Where(i => i > 5);
foreach (int i in requeteFiltree)
{
    Console.WriteLine(i);
}
```

Nous utilisons la méthode d'extensions Where() en lui fournissant une expression lambda servant de prédicat pour filtrer la liste.
C'est de cette façon que le compilateur traduit la sugar syntax. Elle n'est donc qu'une façon plus jolie d'utiliser ces méthodes d'extensions.
Chaque méthode d'extension renvoie un IEnumerable<T> ce qui permet d'enchainer facilement les filtres successifs. Par exemple, rajoutons une date et un nombre d'articles à notre classe Commande :

```
public class Commande
{
    public int Identifiant { get; set; }
    public int IdentifiantClient { get; set; }
    public decimal Prix { get; set; }
    public DateTime Date { get; set; }
    public int NombreArticles { get; set; }
 }
```

Avec la requête suivante :

```
IEnumerable<Commande> commandesFiltrees = listeCommandes.
                      Where(commande => commande.Prix > 100).
                      Where(commande => commande.NombreArticles > 10).
                      OrderBy(commande => commande.Prix).
                      ThenBy(commande => commande.DateAchat);
```

Nous pouvons obtenir les commandes dont le prix est supérieur à 100, où le nombre d'articles est supérieur à 10, triées par prix puis par date d'achat.

De plus, ces méthodes d'extensions font beaucoup plus de choses que ce que l'on peut faire avec la sugar syntax. Il existe pas mal de méthodes intéressantes, que nous ne pourrons pas toutes étudier.
Regardons par exemple la méthode Sum() (qui a été utilisée dans le paragraphe précédent) qui permet de faire la somme des éléments d'une liste ou la méthode Average() qui permet de faire la moyenne :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
Console.WriteLine("Somme : {0}", liste.Sum());
Console.WriteLine("Moyenne : {0}", liste.Average());
```

```
3
```

Qui nous renvoie dans ce cas :

```
Somme : 51

Moyenne : 6,375
```

Tout est déjà fait  , pratique !
Évidemment, les surcharges de ces deux méthodes d'extensions ne fonctionnent qu'avec des typesint ou double ou decimal… Qui envisagerait de faire une moyenne sur une chaine ?
Par contre, il est possible de définir une expression lambda dans la méthode Sum() afin de faire la somme sur un élément d'un objet, comme le prix de notre commande :

```
1 decimal prixTotal = listeCommandes.Sum(commande => commande.Prix);
```

D'autres méthodes sont bien utiles. Par exemple la méthode d'extension Take() nous permet de récupérer les X premiers éléments d'une liste :

```
1 IEnumerable<Client> extrait = listeClients.OrderByDescending
(client => client.Age).Take(5);
```

Ici, je trie dans un premier temps ma liste par âge décroissant, et je prends les 5 premiers. Ce qui signifie que je prends les 5 plus vieux clients de ma liste.
Et s'il y en a que 3 ? et bien il prendra uniquement les 3 premiers. 
Suivant le même principe, on peut utiliser la méthode First() pour obtenir le premier élément d'une liste :

```
1 List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
2 int premier = liste.Where(i => i > 5).First();
```

Où j'obtiens le premier élément de la liste qui est strictement supérieur à 5.
À noter que le filtre peut également se faire dans l'expression lambda de la méthode First() :

```
1 int premier = liste.First(i => i > 5);
```

Ce qui revient exactement au même.
Attention, s'il n'y a aucun élément dans la liste, alors la méthode First() lève l'exception :

```
Exception non gérée : System.InvalidOperationException: La séquence ne contient aucun élément.
```

Il est possible dans ce cas-là d'éviter une exception avec la méthode FirstOrDefault() qui renvoie la valeur par défaut du type de la liste (0 si c'est un type valeur, null si c'est un type référence) :

```
1 Client nicolas = listeClients.FirstOrDefault(client => client.Nom == "Nicolas");
2 if (nicolas == null)
3     Console.WriteLine("Client non trouvé");
```

Ici, je cherche le premier des clients dont le nom est Nicolas. S'il n'est pas trouvé, alors FirstOrDefault()me renvoie null, sinon, il me renvoie bien sûr le bon objet Client.

Dans le même genre, nous pouvons compter grâce à la méthode Count() le nombre d'éléments d'une source de données suivant un critère :

```
1 int nombreClientsMajeurs = listeClients.Count(client => client.Age >= 18);
```

Ici, j'obtiendrais le nombre de clients majeurs dans ma liste.
De la même façon qu'avec la *sugar syntax*, il est possible de faire une sélection précise des données que l'on souhaite extraire, grâce à la méthode Select() :

```
1 var requete = listeClients.Where(client => client.Age >= 18)
                            .Select(client => new { client.Age, client.Nom }) ;
```

Cela me permettra d'obtenir une requête contenant les clients majeurs. À noter qu'il y a aura dedans des objets anonymes possédant une propriété Age et une propriété Nom.
Bien sûr, nous retrouverons nos jointures avec la méthode d'extension Join() ou les groupes avec la méthode GroupBy().
Il existe beaucoup de méthodes d'extensions et il n'est pas envisageable dans ce tutoriel de toutes les décrire.
Je vais finir en vous parlant des méthodes ToList() et ToArray() qui comme leurs noms le suggèrent, permettent de forcer la requête à être mise dans une liste ou dans un tableau :

```
1 List<Client> lesPlusVieuxClients =
  listeClients.OrderByDescending(client => client.Age).Take(5).ToList();
```

ou

```
1 Client[] lesPlusVieuxClients =
listeClients.OrderByDescending(client => client.Age).Take(5).ToArray();
```

Plutôt que d'avoir un IEnumerable<>, nous obtiendrons cette fois-ci une List<> ou un tableau. Le fait d'utiliser ces méthodes d'extensions a des conséquences que nous allons décrire.

Alors, les méthodes d'extensions LINQ ou sa syntaxe sucrée c'est bien joli, mais quel est l'intérêt de s'en servir plutôt que d'utiliser des boucles foreach, des if ou autres choses ?
Déjà, parce qu'il y a plein de choses déjà toutes faites, la somme, la moyenne, la récupération de X éléments, etc.
Mais aussi pour une autre raison plus importante : **l'exécution différée**.

Nous en avons déjà parlé, l'exécution différée est possible grâce au mot-clé yield. Les méthodes d'extensions Linq utilisent fortement ce principe.
Cela veut dire que lorsque nous construisons une requête, elle n'est pas exécutée tant qu'on itère pas sur le contenu de la requête. Ceci permet de stocker la requête, d'empiler éventuellement des filtres ou des jointures et de ne pas calculer le résultat tant qu'on n'en a pas explicitement besoin.

Ainsi, imaginons que nous souhaitions trier une liste d'entiers, avant nous aurions fait :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };

liste.Sort();
liste.Add(7);

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

Ce qui aurait affiché en toute logique la liste triée puis à la fin l'entier 7 rajouté, c'est-à-dire :

```
1
3
4
5
6
8
9
15
7
```

Avec Linq, nous allons pouvoir faire :

```
List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };

var requete = liste.OrderBy(e => e);
liste.Add(7);

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

Et si nous exécutons ce code, nous aurons :

```
1
3
4
5
6
7
```

```
8

9

15
```

Bien que nous ayons ajouté la valeur 7 après avoir trié la liste avec OrderBy, on se rend compte que tous les entiers sont quand même triés lorsque nous les affichons.

En effet, la requête n'a été exécutée qu'au moment du foreach. Ceci implique donc que le tri va tenir compte de l'ajout du 7 à la liste. La requête est construite en mémorisant les conditions comme notreOrderBy, mais cela fonctionne également avec un where, et tout ceci n'est exécuté que lorsqu'on le demande explicitement, c'est-à-dire avec un foreach dans ce cas-là. En fait, tant que le C# n'est pas obligé de parcourir les éléments énumérables alors il ne le fait pas. Ce qui permet d'enchainer les éventuelles conditions et éviter les parcours inutiles. Par exemple, dans le cas ci-dessous, il est inutile d'exécuter le premier filtre :

```
1 List<int> liste = new List<int> { 4, 6, 1, 9, 5, 15, 8, 3 };
2 IEnumerable<int> requete = liste.Where(i => i > 5);
3 // plein de choses qui n'ont rien à voir avec la requete
4 requete = requete.Where(i => i > 10);
```

Car le deuxième filtre a tout intérêt à être combiné au premier afin d'être simplifié.
Et encore, ici, on n'utilise même pas la requête, il y a encore moins d'intérêt à effectuer nos filtres si nous ne nous servons pas du résultat.
Ceci peut paraître inattendu, mais c'est très important dans la façon dont Linq s'en sert afin d'optimiser ses requêtes. Ici, le parcours en mémoire pourrait paraître peu couteux, mais dans la mesure où Linq doit fonctionner aussi bien avec des objets, qu'avec des bases de données ou du XML (ou autres…), cette optimisation prend tout son sens.

Le maitre mot est la **performance**, primordial quand on accède aux bases de données.
Cette exécution différée est gardée pour le plus tard possible. C'est-à-dire que le fait de parcourir notre boucle va obligatoirement entrainer l'évaluation de la requête afin de pouvoir retourner les résultats cohérents.
Il en va de même pour certaines autres opérations, comme la méthode Sum(). Comment pourrions-nous faire la somme de tous les éléments si nous ne les parcourons pas ?
C'est aussi le cas pour les méthodes ToList() et ToArray().
Par contre, ce n'est pas le cas pour les méthodes Where, ou Take, etc …

Il est important de connaitre ce mécanisme. L'exécution différée est très puissante et connaître son fonctionnement permet de savoir exactement ce que nous faisons et pourquoi nous pourrions obtenir parfois des résultats étranges.

Pour terminer avec Linq, voici un tableau récapitulatif des différents opérateurs de requête. Nous ne les avons pas tous étudiés ici car cela serait bien vite lassant. Mais grâce à leurs noms et leurs types, il est assez facile de voir à quoi ils servent afin de les utiliser dans la construction de nos requêtes.

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

Type	Opérateur de requête	Exécution différée
Opérations de concaténation	Concat	Oui
Opérations d'agrégation	Aggregate, Average, Count, LongCount, Max, Min, Sum	Non

Linq apporte des méthodes d'extensions et une syntaxe complémentaire afin d'être efficace avec la manipulation de sources de données.

Sachez enfin qu'il est possible de requêter n'importe quelle source de données à partir du moment où un connecteur spécifique a été développé. Cela a été fait par exemple pour interroger Google ou Amazon, mais aussi pour requêter sur active directory, ou JSON, etc.

En résumé

- Linq consiste en un ensemble d'extensions du langage permettant de faire des requêtes sur des données en faisant abstraction de leur type.
- Il existe plusieurs domaines d'applications de Linq, comme *Linq to Object*, *Linq to Sql*, etc.
- La *sugar syntax* ajoute des mots-clés qui permettent de faire des requêtes qui ressemblent aux requêtes faites avec le langage SQL.
- Derrière cette syntaxe se cache un bon nombre de méthodes d'extension qui tirent parti des mécanismes d'exécution différée.

TP :

Soit les relations suivantes de la société SOFT
Emp(NumE, NomE, Fonction, NumS, Embauche, Salaire, Comm,
NumD)
Dept(NumD, NomD, Lieu)

NumD	NomD	Lieu
1	Droit	Tunis
2	Commerce	Ariana

NomE	Fonction	NumS	Embauche	Salaire	Comm	NumD
ALi	Président	NULL	10/10/1979	10000	NULL	NULL
Mohamed	Directeur	1	01/10/2006	5000	NULL	1
Salah	Stagaire	1	01/10/2006	0	NULL	1
Rami	Commercial	2	01/10/2006	5000	100	2

1) Donnez la liste des employés ayant une commission (non NULL) classé par commission décroissante
2) Donnez les noms des personnes embauchées depuis le 01-09-2006
3) Donnez la liste des employés travaillant à Tunis
4) Donnez la moyenne des salaires
5) Donnez le nombre de commissions non NULL
6) Donnez la liste des employés gagnant plus que la moyenne des salaires de l'entreprise