

Secteur Tertiaire Informatique
Filière « Etude et développement »

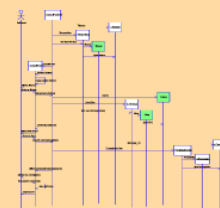
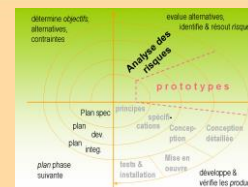
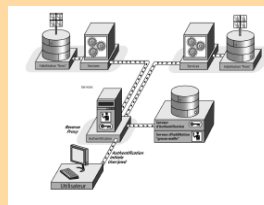
Construire une application organisée en couches en
mettant en œuvre des Frameworks

Mettre en œuvre un framework de persistance
Manipuler les objets

Apprentissage



Mise en situation

Evaluation



Version	Date	Auteur(s)	Action(s)
1.0	03/01/2020	Vincent Bost	Création du document

TABLE DES MATIERES

Table des matières.....	3
1. Introduction.....	6
1.1 La problématique : différences structurelles Objet Relationnel	6
1.2 La réponse : Un framework ORM.....	7
2. Linq	8
2.1 Les différentes versions de Linq :	9
3. A la découverte de Linq to Objects	10
3.1 Les délégués	11
3.2 Les fonctions anonymes	12
3.3 Les expressions lambda	12
3.4 Les méthodes d'extension	14
3.4.1 Créer une méthode d'extension.....	14
3.5  Réaliser les exercices du chapitre 1 de P-LinqToObject	16
3.6 Introduction à la mise en œuvre de linq to objects	16
3.6.1 La syntaxe des requêtes.....	16
3.6.2 La syntaxe des méthodes.....	16
3.6.3 La simplification de l'expression lambda.....	17
3.7 Au fait comment cela fonctionne-t-il	18
3.8 Exemples et points d'attention	20
3.8.1 Partition de séquences	20
3.9 La projection	21
3.10 Récupérer un élément unique.....	22
3.11 Les quantificateurs.....	22
3.12 Les agrégations	23
3.12.1 Le comptage distinct	23
3.12.2 Mettre en cache une valeur d'agrégat	23
3.13 Partitionnement	23
3.14  Réaliser les exercices du chapitre 2 et Suivants de P-LinqToObject.....	24
4. Opérateurs de requête standard.....	24

Objectifs

Ce document a pour objet de vous accompagner dans l'apprentissage des techniques de manipulation d'objets avec le langage de requête Linq.

Après avoir réalisé les exercices proposés, vous devez être en mesure d'assurer

- D'interroger, filtrer, agréger des graphes d'objets.
- D'assurer la persistance de l'état de vos objets au sein d'une base de données

Pré requis

Connaissance du langage C# et de l'environnement de développement Visual Studio

Connaissance de la programmation orienté objet.

Outils de développement

Environnement de développement :

- Visual Studio version 2019 ou ultérieure

Pour la manipulation des objets :

- Le langage de requête Linq

Méthodologie

Avant d'aborder la pratique de Linq au travers d'un Framework de mapping objet/relationnel (ORM), nous allons revenir sur quelques principes de programmation à la base de ce langage de requête : les délégués, les expressions lambda et les méthodes d'extension.

Réalisez-les exercices à l'issue de la lecture de chaque chapitre comportant une mise en pratique des techniques exposées. Vous devriez ainsi, par la pratique, pas à pas, atteindre les objectifs de cette unité d'apprentissage consacrée à l'apprentissage du langage de requête linq.

Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

Ressources

Projet application console core pour production des données nécessaires à la réalisation des exercices d'apprentissage.

Lectures conseillées

Le tutoriel en ligne consacré à Linq (en) <http://www.tutorialsteacher.com/linq/linq-tutorials>

1. INTRODUCTION

Cette introduction pour présenter les principes qui nous permettront de faire cohabiter harmonieusement deux mondes bien différents auxquels chaque développeur est confronté :

- Celui des techniques de conception et de développement objet avec des langages et des frameworks toujours plus puissants
- Celui des bases de données relationnelles. Incontournables, elles se doivent d'assurer la persistance des états de nos entités métier.

1.1 LA PROBLEMATIQUE : DIFFERENCES STRUCTURELLES OBJET RELATIONNEL

Les différences structurelles entre ces deux mondes sont importantes :

	Objet	Relationnel
Organisation des données	Graphe d'objets selon un modèle arborescent	Modèle à deux dimensions ligne / colonne
Spécialisation/généralisation	Héritage	Ersatz au travers de tables associatives ou vues
Type des données	Spécification du langage	Norme SQL
Dépendances entre entités	Référence objet	Clé étrangère
Règles de nommage	Type.Attribut Beneficiaire.Nom	NomColonne unique NomBeneficiaire

Quelles que soient les évolutions des SGBD-R, comme la possibilité de gérer des données de type Blob (Binary Large Object), des fragments XML, ou des données au format Json, ils obéiront toujours aux règles fondamentales du docteur CODD (fondateur du modèle relationnel) et en particulier la toute première de ses 12 règles qui impose que toute information soit présentée selon un modèle ligne / colonne, avec une entité de base, la table (appelée aussi relation d'où le terme relationnel).

Ne pourrions-nous pas envisager de substituer aux systèmes de gestion de bases de données relationnelles des bases de données orientés objet ? La question est ouverte avec les évolutions des bases de données NoSQL qui, pour certaines, introduisent des mécanismes de gestion de références pouvant être sollicités pour mettre en relation les objets. Toutefois, les bases de données NoSQL ciblent plutôt des données faiblement structurées dont les schémas évoluent fréquemment, alors que les schémas des données d'un SGBD Relationnel sont stables.

Les SGBD-R ont donc de très beaux jours devant eux car ils sont :

- standardisés,
- rapides,
- puissants,
- présents sous de nombreuses références éprouvées (SQL Server, MySQL, Oracle, DB2, PostgreSql, ...)

Et ils proposent, dans leurs dernières versions, des mécanismes de manipulations de données XML ou Json qui sont de plus en plus performants...

1.2 LA REPONSE : UN FRAMEWORK ORM

Il s'agit d'une réponse très pertinente actuellement. ORM (ou O/R Mapping) est un acronyme pour Object-Relational Mapping. Sa mise en œuvre simplifie la gestion de la persistance de nos entités métier par rapport au développement spécifique d'une couche de composants d'accès aux données comme nous l'avons fait précédemment.

Un ORM permettra de manipuler des données issues de lignes et colonnes de tables par le biais de graphes attendus par les langages orientés objet.

Les fonctionnalités présentes et les implémentations diffèrent en fonction des produits ORM.

Il n'existe pas de normes ni de standards d'implémentation.

Nous retrouverons toutefois dans chaque produit ORM :

- Un mécanisme de liaison entre les attributs du type qui devront persister et les colonnes de la table
- Un langage pour manipuler les objets issus de la DB et sauvegarder les états de ces derniers sans recourir au langage SQL.

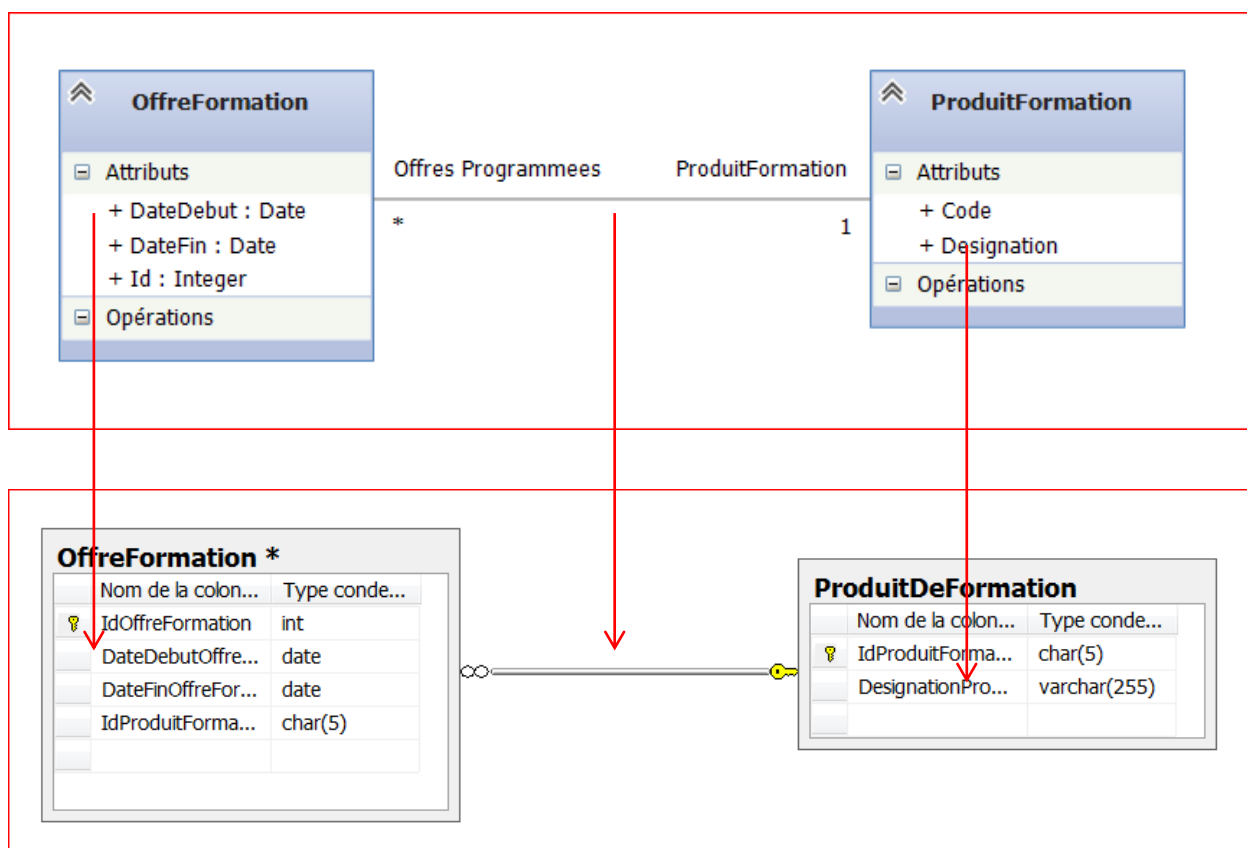


Figure 1 : Exemple de mappage entre classes Objet – tables Relationnel

Il existe des ORM dans les différentes architectures logicielles.

Quelques produits pour exemple :

- Java-JEE : Hibernate (version NHibernate pour .Net)
- .Net : Entity Framework
- Php : Doctrine

ORM Manipulation des objets

2. LINQ

Le langage d'interrogation Linq est disponible pour différentes cibles et non uniquement en lien avec des sources de données relationnelles. Linq, pour Language-INtegrated Query, est un langage de requête objet (query) qui fournit des mécanismes de sélection, tri, agrégation et mise à jour des données issues de différentes sources. Il est totalement intégré à l'architecture .Net et ne nécessite pas l'apprentissage d'un autre langage.

Le schéma ci-dessous présente les déclinaisons de Linq dans l'architecture Core Version 3.

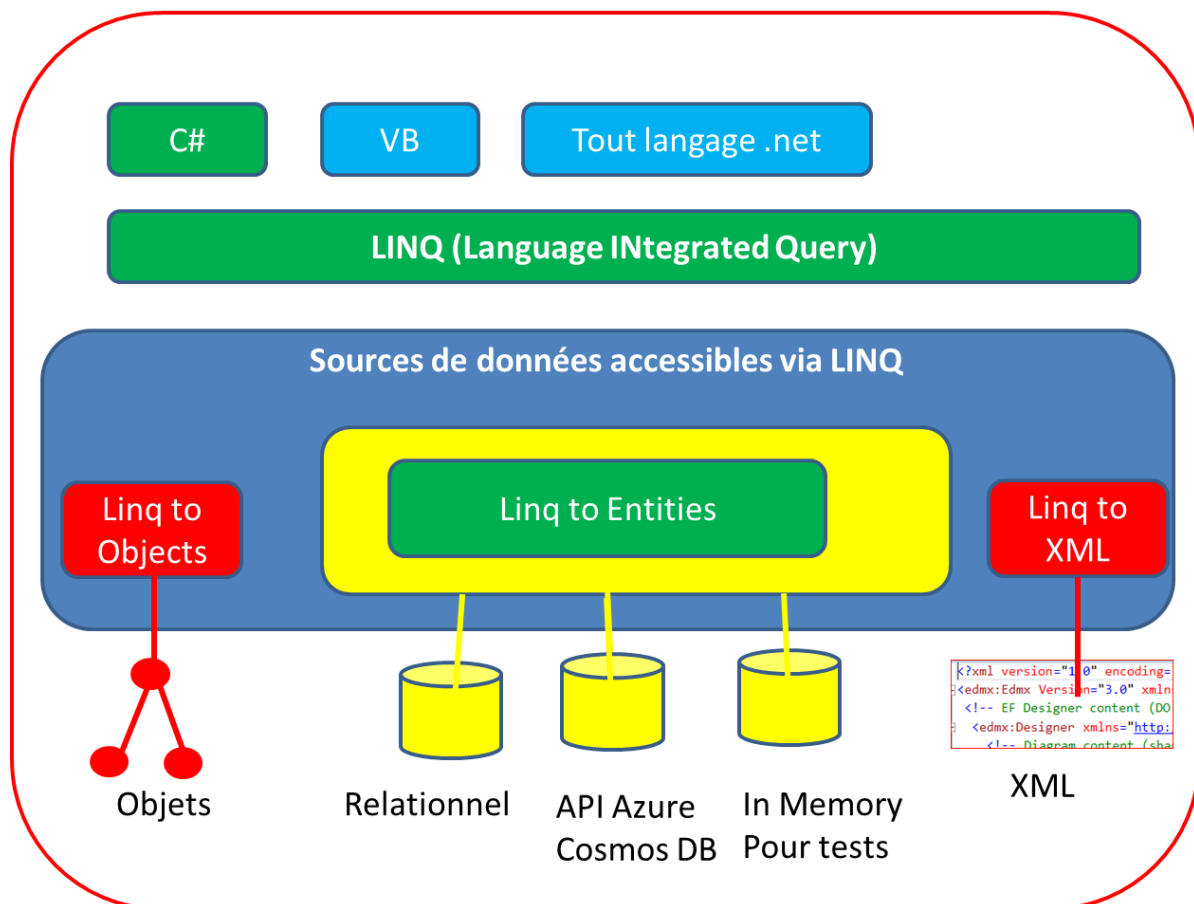


Figure 2 : Architecture de linq

Certaines versions de Linq to Entities ne recouvrent pas toutes la notion d'ORM. Ainsi, le fournisseur pour bases de données Cosmos DB, qui est une base de données No SQL, et In Memory qui sera abordé dans la phase consacrée aux tests unitaires en développement Web selon le pattern MVC.

Des fournisseurs sont proposés pour les principaux SGBD relationnels du marché : Oracle, mySQL, MariaDB, DB2, Informix, base portable SQLite...

Les évolutions dans l'environnement Core sont très fréquentes, aussi est-il nécessaire d'organiser une veille technologique active... Le lien ci-dessous recense les différents fournisseurs

<https://docs.microsoft.com/fr-fr/ef/core/providers/?tabs=dotnet-core-cli>

La version spécialisée pour SQL Server (Entity To SQL) disponible dans l'architecture .Net Windows n'est plus disponible en .Net Core : Entity to SQL

2.1 LES DIFFERENTES VERSIONS DE LINQ :

1. **Linq to Objects** : il nous permet de manipuler les collections d'objets issues de nos différentes sources. Il constitue en quelque sorte le socle de la technologie. Nous aborderons ce langage et l'expression des requêtes dans la deuxième unité d'apprentissage consacrée à la manipulation des objets.
2. **Linq to XML** : Il permet d'interroger des arborescences XML tout comme le ferait un langage comme XPath. Mais Linq to XML propose des fonctionnalités beaucoup plus puissantes et qui dépassent les limites de XPath.

Les requêtes Linq sont analysées par le compilateur qui peut détecter de nombreuses erreurs qui ne seraient découvertes qu'au moment de l'exécution avec XPath.

Les résultats de Linq to XML sont toujours fortement typés.

3. **Linq to Entities** et Entity Framework :

Il s'agit d'une des principales constituantes de l'écosystème. C'est un produit Open Source.

Ce langage n'interroge pas directement la couche de données et n'exige pas une symétrie stricte entre les entités manipulées par LINQ et les tables ou vues présentes dans la base de données comme LINQ to SQL.

LINQ to Entities permet d'écrire des requêtes qui ciblent un modèle conceptuel, qui sera lui-même à son tour mappé sur la base physique.

Schéma des classes et schéma des tables peuvent donc être différents.

Il y a une indirection entre le modèle d'entités et le modèle de stockage de la base de données : utile pour implémenter l'héritage ou l'intégration de types complexes. Dans la version actuelle de Core, les formes d'héritage disponibles sont limitées.

3. A LA DECOUVERTE DE LINQ TO OBJECTS

Nous allons ici découvrir le langage linq dans sa version destinée à la manipulation d'objets en mémoire. A cette occasion, nous reviendrons sur certaines techniques de programmation sollicitées dans linq mais qui peuvent être envisagées dans bien d'autres contextes :

- La notion de délégué
- Les méthodes anonymes
- Les expressions lambdas
- Les méthodes d'extension de classe

Analysons ces deux premières requêtes Linq sur des listes d'objets de type Personne.

```
List<Personne> personnes = GetPersonnes();  
  
var personnesAgees = personnes.Where(p => p.Age > 50).ToList();  
  
var femmesAgees = personnes.Where(p =>  
{  
    bool age = (p.Age > 50);  
    return age && p.Genre == Sexe.Féminin;  
}).ToList();
```

La première expression lambda utilise en corps de la méthode une expression **p.Age > 50**, la deuxième expression un bloc d'instructions délimité par **{}**. **L'opérateur lambda =>** sépare le **paramètre p** objet de type personne du corps de la méthode ici un prédicat (évaluation donne vrai ou faux).

L'appel de la méthode ToList() va provoquer la construction d'une **nouvelle séquence** ici, une liste de personnes pour lesquelles la valeur du prédicat est vrai.

personnes	Count = 6	System.Collectio...
[0]	(LangageLinq.Personne)	LangageLinq.Pers...
[1]	(LangageLinq.Personne)	LangageLinq.Pers...
[2]	(LangageLinq.Personne)	LangageLinq.Pers...
[3]	(LangageLinq.Personne)	LangageLinq.Pers...
[4]	(LangageLinq.Personne)	LangageLinq.Pers...
[5]	(LangageLinq.Personne)	LangageLinq.Pers...
Raw View		
personnesAgees	Count = 4	System.Collectio...
[0]	(LangageLinq.Personne)	LangageLinq.Pers...
[1]	(LangageLinq.Personne)	LangageLinq.Pers...
[2]	(LangageLinq.Personne)	LangageLinq.Pers...
[3]	(LangageLinq.Personne)	LangageLinq.Pers...
Raw View		
femmesAgees	Count = 1	System.Collectio...
[0]	(LangageLinq.Personne)	LangageLinq.Pers...
Raw View		

Nous pouvons observer que la même méthode Where prend des expressions qui varient d'un contexte à l'autre. Elle est définie sur la base d'un **délégué** de fonction.

3.1 LES DELEGUES

Un délégué de fonction permet de stocker la référence d'une fonction.

Le langage étant orienté objet et fortement typé, nous devons définir le type du délégué qui précisera la signature que devront respecter les fonctions passées via ce délégué.

Nous avons utilisé ce procédé avec les événements.

Nous l'avons aussi utilisé dans les mécanismes de définition de méthodes de rappel (callBack) comme par exemple en JavaScript ci-dessous, avec une implémentation plus simple du fait de l'absence de typage.

```
TestServiceWebAjax.WSTest.ListerPersonnes(succes, echec);
```

Dans l'exemple ci-après, nous allons utiliser cette technique pour factoriser au mieux notre code. Il s'agit d'un exemple sans grand intérêt fonctionnel.... Il s'agit de réaliser la liste de valeurs numériques et les résultats d'opérations arithmétiques effectuées sur les éléments de cette même liste. Nous allons dans un premier temps considérer la somme et la multiplication.

Le recours au mécanisme de délégation nous permettra de n'avoir qu'une seule fonction de traitement des valeurs et une opération arithmétique variable.

Ci-dessous la déclaration du type délégué et des deux fonctions concrètes pour addition et multiplication.

```
public delegate double Operation(double a, double b);
1 référence
static double Addition(double x, double y)
{
    return x + y;
}
1 référence
static double Multiplication(double x, double y)
{
    return x * y;
}
```

Extrait de la méthode qui traite les valeurs. La variable ope, du type délégué Operation, contiendra l'adresse de la fonction à exécuter.

```
static void Resultat(double[] valeurs, Operation ope)
{
    double[] res = new double[valeurs.Length-1];
    for (int i = 1; i < valeurs.Length; i++)
    {
        res[i-1] = ope(valeurs[i], valeurs[i - 1]);
    }
}
```

Réalisation de l'opération

Invocation de la fonction avec chacune des opérations.

```
double[] liste = new double[] {1.0052,2,3,4,8,89.25 };
Resultat(liste, Addition);
Resultat(liste, Multiplication);
```

Nous obtenons le résultat escompté. Sans autre intérêt que de vérifier le bon fonctionnement de l'ensemble.

```
-- debut ----
Liste des valeurs
1,0052;2;3;4;8;89,25;
Résultat de l'opération
3,0052000000000003;5;7;12;97,25;-- fin ----
-- debut ----
Liste des valeurs
1,0052;2;3;4;8;89,25;
Résultat de l'opération
2,0104;6;12;32;714;-- fin ----
```

3.2 LES FONCTIONS ANONYMES

Comme en JavaScript, il est possible d'avoir recours à des méthodes anonymes.

Ainsi, je peux, au lieu de nommer la fonction, définir son corps au moment de son exécution. Comme en JavaScript, ce code ne pourra être exécuté en dehors de ce contexte d'appel.

```
Resultat(liste, delegate (double e, double f) { return e * f; });
```

3.3 LES EXPRESSIONS LAMBDA

Je peux aussi remplacer avantageusement cette syntaxe par une expression lambda.

```
Resultat(liste, ( e, f) => e * f);
```

Cerise sur le gâteau, dans la plupart des cas il n'est pas nécessaire de définir les types des paramètres en entrée, c'est le compilateur qui les détermine par inférence.

Mais nous pouvons aussi, si nécessaire, fournir le type des paramètres. C'est le cas lorsqu'il existe plusieurs implémentations d'une même méthode.

```
Resultat(liste, (double e, double f) => e * f);
```

Nous avons ici utilisé une expression lambda associée à un délégué défini spécifiquement.

Mais la plupart du temps, ce sont des délégués génériques qui sont utilisés.

Ainsi, la méthode Resultat pourrait être définie ainsi :

```
4 références
static void Resultat(double[] valeurs, Func<double,double,double> ope)
{
```

En fait les expressions lambda peuvent être définies sous forme de fonctions à l'aide des délégués génériques Action<> et Func<>

Les expressions lambdas reposent sur des délégués génériques Action et Func qui sont des pointeurs vers des méthodes (délégués) mais qui ont pour particularités :

- Pour **Action** de pointer vers une méthode qui ne renvoie rien (s'apparente à une procédure ou méthode void) **Action<T> methode**
- Pour **Func** de pointer vers une méthode qui renvoie un type toujours défini comme le dernier argument de la méthode comme par exemple **Func<int, string, bool> methode**. La méthode prendra un entier et une chaîne en arguments d'entrée et renverra un booléen en sortie

Ces délégués sont génériques et implémentés avec un nombre de 1 à 16 paramètres en entrée.

Et la syntaxe de l'expression lambda **p => p % 2 == 0** ?

- A gauche de l'opérateur => le paramètre passé à la méthode
- A droite de l'opérateur => le corps de la méthode (expression évaluée)

C'est un moyen pratique d'écrire du code qui devrait normalement être écrit de façon plus fastidieuse en recourant à une méthode anonyme et un délégué.

Pour les paramètres en entrée, parenthèses obligatoire si 0 ou plus d'un paramètre.

```
() => expression ;  
(a,b) => expression ;
```

Quelques exemples de définition de méthodes avec des lambdas.

La notion d'expression lambda peut être distinguée de la notion d'instructions lambda (statements). Mais nous évoquons le plus souvent indifféremment le terme d'expression lambda.

Les instructions lambdas sont constituées d'un bloc d'instructions entre {}.

Pour une expression pas de {}

Quelques exemples.

Fonctions ne retournant pas de valeurs

```
Action expSP = () => Console.WriteLine("Expression sans parametre");  
expSP();  
Action<string> expString = texte => Console.WriteLine($"Bonjour {texte}");  
expString("Conc Dev Application");
```

```
Expression sans parametre  
Bonjour Conc Dev Application
```

Fonctions retournant une valeur

```
Func<string, int, bool> testerLongueur = (texte, longueur) => texte.Length <= longueur;  
Console.WriteLine($"Resultat {testerLongueur("CDA", 3)}");  
  
Func<int, int, bool> instructionsCarreSup = (val, max) =>  
{  
    long carre = val * val;  
    return carre > max;  
};  
Console.WriteLine($"Resultat {instructionsCarreSup(3, 6)}");
```

```
Resultat True  
Resultat True
```

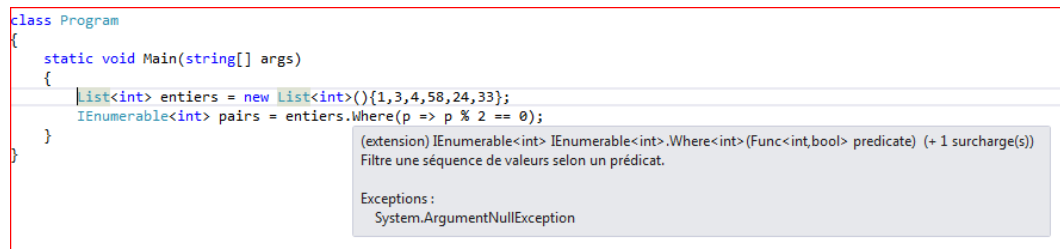
3.4 LES METHODES D'EXTENSION

Les méthodes de Linq sont implémentées sous forme de méthodes d'extension de type. Pour mieux comprendre le mécanisme des requêtes basées sur les méthodes, il convient de s'approprier quelques règles de base.

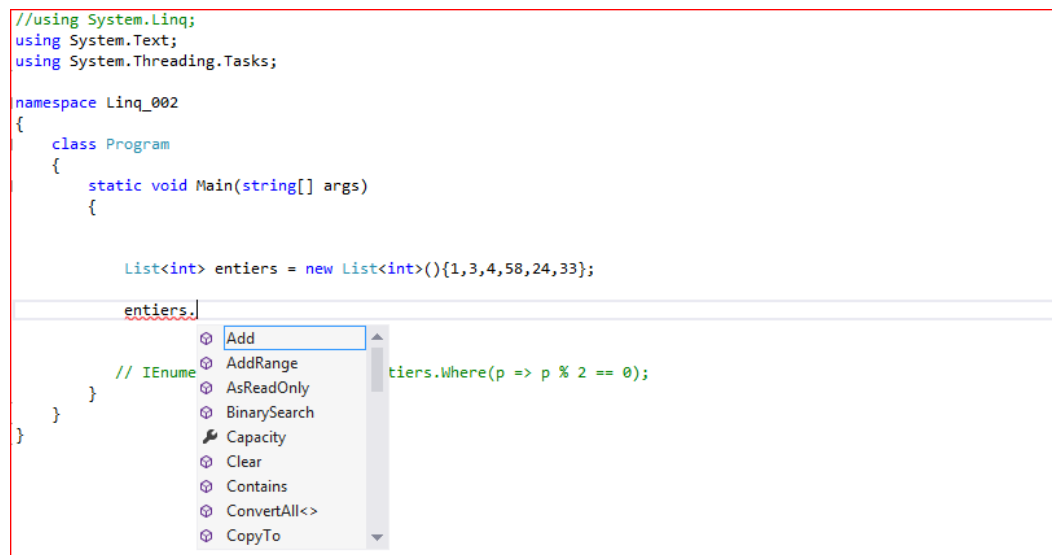
Cette possibilité d'étendre les fonctionnalités d'un type sans modifier le type existant peut être invoquée pour se conformer à l'objectif du pattern qui veut que nos classes soient Ouvertes à des extensions mais fermées à la modification (O de SOLID). Cette méthode très intéressante reste toutefois limitée à l'implémentation de nouvelles méthodes.

Nous pouvons étendre un type sans disposer de son code source.

Ainsi, l'IntelliSense nous propose de nombreuses méthodes qui ne sont pas celles d'une liste générique<T>. Ainsi, la méthode Where qui est une méthode d'extension.



Pour vous en convaincre vous pouvez supprimer la référence à l'espace de noms linq. Vous ne verrez alors plus apparaître les méthodes d'extension.



3.4.1 Créer une méthode d'extension

Les méthodes d'extension sont des méthodes statiques qui peuvent être associées à un type et invoquées comme si elles étaient des méthodes d'instance de ce type.

Les opérateurs de requête standard sont fournis sous cette forme et étendent tout type qui implémente IEnumerable<Of <(T)>>).

Un petit exemple qui permet de comprendre comment cela fonctionne :
Je souhaite étendre les fonctionnalités du type int en proposant une méthode de calcul du carré du nombre. Il suffit alors de respecter les règles suivantes :

- Définir la méthode d'extension comme statique
- Invoquer celle-ci comme méthode d'instance, le premier paramètre qui spécifie le type auquel la méthode s'applique est précédé par le modificateur **this**.
- Importer l'espace de noms où sont définies les extensions de manière explicite. Autrement, les méthodes d'extension ne seront pas accessibles (hors de portée).

Exemple pour la méthode carre qui s'applique à int. Nous pouvons l'écrire de 2 manières

```
public static class IntExtension
{
    2 références
    public static int Carre(this int nombre)
    {
        return nombre * nombre;
    }
}
```

Ou en utilisant l'opérateur lambda pour définir le corps de la méthode.
C'est aussi une des possibilités offertes en dehors de l'usage dans une expression lambda.
<https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/operators/lambda-operator>

```
public static class IntExtension
{
    2 références
    public static int Carre(this int nombre)
        => nombre * nombre;
}
```

Pourront être codées dans la classe IntExtension toutes les méthodes étendant le type int.
La classe et les méthodes seront marquées comme statiques.
Elles sont classées dans l'espace de noms

```
namespace MesExtensions
{
}
```

Dans le projet où elles devront être utilisées, il conviendra donc d'importer explicitement MesExtensions par le biais d'une directive using.

```
using MesExtensions;
```

Création de deux nouvelles séquences, une qui représente les carrés des valeurs, l'autre une séquence qui filtre les valeurs pour ne retenir que celles dont le carré est supérieur à 9.

```
List<int> entiers = new List<int> { 1, 3, 4, 5 };
var carres = entiers.Select(i => i.Carre()).ToList();
var carresSup9 = entiers.Where(i => i.Carre() > 9).ToList();
```

Il est aussi possible de ranger les classes d'extension dans le même namespace.

ORM Manipulation des objets

Ainsi, il n'est pas nécessaire de recourir à un using spécifique.

3.5 REALISER LES EXERCICES DU CHAPITRE 1 DE P-LINQToOBJECT

Réalisez les exercices sur construction expression lambda et les méthodes d'extension.

3.6 INTRODUCTION A LA MISE EN ŒUVRE DE LINQ TO OBJECTS

Nous allons ici entrer dans le vif du sujet avec les techniques de manipulation d'objets. Linq s'applique à tous les objets qui implémentent l'interface IEnumerable.

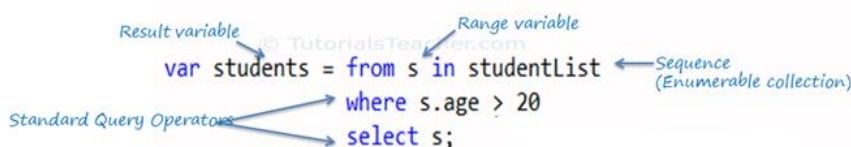
Pour manipuler nos objets nous disposons de 2 syntaxes :

- La syntaxe des requêtes
- La syntaxe des méthodes

3.6.1 La syntaxe des requêtes

La forme des expressions de requête déclarative est proche de la syntaxe SQL.

```
IList<Student> studentList = new List<Student>() {...};
```



```
var students = from s in studentList
               where s.age > 20
               select s;
```

La source sur laquelle la requête s'applique **from .. in** est exprimée en premier pour permettre au compilateur de déterminer le type.

From... in ... s'apparente à un **pour** pour chaque élément dans la collection.

Après la clause **from** nous trouvons un des nombreux opérateurs standards de linq.

Ici, un opérateur de filtre **Where** et un opérateur de projection **Select**.

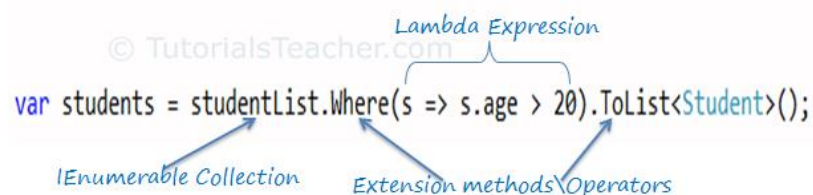
La requête se termine avec l'opérateur **Select** ou l'opérateur **Groupby**.

Le type du résultat est implicitement déterminé par le compilateur par inférence. Mais vous pouvez préciser le type plutôt que d'utiliser **var**.

3.6.2 La syntaxe des méthodes

La syntaxe des méthodes utilise des méthodes d'extension des types Enumerable.

```
IList<Student> studentList = new List<Student>() {...};
```



```
var students = studentList.Where(s => s.age > 20).ToList<Student>();
```


3.6.3 La simplification de l'expression lambda

Au sein des requêtes linq les expressions lambda sont simplifiées.

```
delegate(Student s) { return s.Age > 12 && s.Age < 20; };  
↓  
1 - Remove Delegate and Parameter Type and  
add lamda operator =>  
delegate(Student s) => { return s.Age > 12 && s.Age < 20; };  
↓  
(s) => { return s.Age > 12 && s.Age < 20; };
```

Il est possible de supprimer la référence au mot clé délégué et supprimer le type qui sera le type étendu par la méthode.

```
(s) => { return s.Age > 12 && s.Age < 20; };  
↓  
2 - Remove curly bracket, return and semicolon  
© TutorialsTeacher.com  
(s) => s.Age > 12 && s.Age < 20;  
↓  
3 - Remove Parenthesis around parameter if there  
is only one parameter  
s => s.Age > 12 && s.Age < 20;
```

Au final nous nous retrouvons avec un paramètre du type considéré, l'opérateur lambda et le corps de la méthode.

Lambda operator

Parameter

Body Expression

```
s => s.Age > 12 && s.Age < 20;
```

Les deux formes comparées :

```
var femmesAgees2 = personnes.Where(delegate(Personne p)  
{  
    bool age = (p.Age > 50);  
    return age && p.Genre == Sexe.Féminin;  
}).ToList();  
  
var femmesAgges = personnes.Where(p => p.Age > 50 && p.Genre == Sexe.Féminin)  
.ToList();
```

3.7 AU FAIT COMMENT CELA FONCTIONNE-T-IL

Nous avons une phase de préparation de nos requêtes qui seront ensuite exécutées lorsque nous demanderons la production d'une nouvelle séquence, l'extraction d'un élément de la séquence ou une opération d'agrégation.

Nous pouvons préparer notre requête en une fois ou en plusieurs étapes.

Ainsi, pour obtenir la liste des femmes âgées, je peux écrire :

```
var femmesAgees3 = personnes.Where(p => p.Age > 50 && p.Genre == Sexe.Féminin);
```

Mais cette instruction ne produit pas de résultat mais un mécanisme qui nous permettra d'itérer sur une séquence implémentant **IEnumerable**. La variable `femmesAgees3` est de type `IEnumerable`.

Pour que ce mécanisme produise un résultat, il faut itérer sur la séquence `personnes` et invoquer une méthode telle que **ToList**, **ToArray**, ..., réaliser un cycle **ForEach** Ou invoquer une méthode qui produit une valeur scalaire à l'issue du parcours de la séquence **Count**, **First**,

Le type `IEnumerable` permettra d'itérer sur un ensemble d'éléments et dispose d'un énumérateur qui permet de se déplacer dans une séquence et adresser l'élément courant.

Ce mécanisme est implémenté de base dans tous les éléments de type `List`, `Collection`, `Array`.

Ce mécanisme propose un énumérateur qui dispose de méthodes telles que `MoveNext`, `Reset`, et l'élément `Current`. L'instruction `yield return` nous permettra de retourner chaque élément courant retenu un par un pour produire la nouvelle séquence en résultat.

Ainsi donc, l'exemple ci-dessus pourrait être programmé comme ceci :

```
var femmesAgees4 = Filtrer(personnes, p => p.Age > 50 && p.Genre == Sexe.Féminin);
```

Avec la définition de la méthode `Filtrer` suivante (Je donne ici cet exemple pour expliquer le principe mais, dans l'immédiat, je suis en train de réinventer l'existant....)

```
static IEnumerable<T> Filtrer<T>(IEnumerable<T> sequence,
    Func<T, bool> predicat)
{
    IEnumerator<T> enumerator = sequence.GetEnumerator();
    while (enumerator.MoveNext())
    {
        var item = enumerator.Current;
        if (predicat(item))
        {
            yield return item;
        }
    }
}
```

Pour obtenir la nouvelle séquence, je peux utiliser différentes formes :

```
var femmesAgees4R = femmesAgees4.ToList();
```

Pour produire la séquence et afficher le nom de chaque personne :

```
femmesAgees4.ToList().  
    ForEach(p => Console.WriteLine($"Madame {p.Nom} est une femme âgée "));
```

```
HELLO WORLD!  
Madame Capelle est une femme âgée
```

Pour itérer et éventuellement réaliser des opérations

```
foreach (var item in femmesAgees4R)  
{  
    // opérations  
}
```

Mais aussi, je peux, pour avoir la femme la plus âgée :

```
Personne plusVieuse = femmesAgees4.OrderByDescending(f => f.Age)  
    .FirstOrDefault();  
  
plusVieuse = Filtrer(personnes, p => p.Age > 50 && p.Genre == Sexe.Féminin)  
    .OrderByDescending(f => f.Age)  
    .FirstOrDefault();
```

Vous pouvez remarquer de nouveau l'intérêt de la programmation générique qui me permettra d'utiliser ma méthode Filtrer avec d'autres types, comme ci-dessous avec une liste d'entiers :

```
double[] liste = new double[] {1.0052,2,3,4,8,89.25 };  
  
Filtrer(liste, dob => dob > 3).ToList().ForEach(e=>Console.WriteLine(e));
```

```
4  
8  
89,25
```

Vous pouvez aussi utiliser la syntaxe des requêtes :

```
var plusVieille = from pers
                  in Filtrer(personnes, p => p.Age > 50 && p.Genre == Sexe.Féminin)
                  orderby pers.Age
                  select pers;
```

Vous pouvez préparer vos requêtes en plusieurs étapes :

```
var I1 = GetPersonnes().Where(p => p.Genre == Sexe.Masculin);
var I2 = I1.Where(p => p.Age < 40);
var I3 = I2.Where(p => p.Nom.StartsWith("f", ignoreCase: true, null));

I3.ToList().ForEach(p =>
Console.WriteLine($"Homme jeune dont le nom commence par f {p.Nom}"));
```

Homme jeune dont le nom commence par f Farnier

Mais je peux aussi utiliser cette forme ci :

```
var I4 = GetPersonnes().Where(p => p.Genre == Sexe.Masculin).Where(p => p.Age < 40);
```

Les méthodes de linq sont composables. Dans la requête ci-dessus le filtre est composé (composite) des deux conditions. Je peux invoquer les méthodes en cascade pour les invoquer séquentiellement les unes après les autres.

La préparation de notre requête en plusieurs étapes peut éventuellement simplifier la mise au point et chaîner en cascade les méthodes peut être fort pratique.

3.8 EXEMPLES ET POINTS D'ATTENTION

3.8.1 Partition de séquences

Attention à l'ordre d'application des méthodes lorsque vous réaliser des partitions sur des séquences. Certaines fonctions doivent être invoquées dans un ordre précis et pas seulement dans ce contexte.

Ainsi, dans l'exemple qui suit, l'incidence du tri sur le résultat obtenu par application de la méthode TakeWhile qui indique de retenir les éléments de la séquence tant qu'ils vérifient la condition exprimée.

Même attention avec les méthodes Skip et Take qui permettent de ne retenir qu'un sous-ensemble d'une sélection.

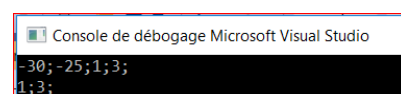
Ici, seul l'ordre de tri a été déplacé.

Résultat

```
int[] entiers = new int[] { 1, 3, 8, 9, 12, 256, -25, -30 };

entiers.OrderBy(e => e).
    TakeWhile(e => e < 5).
    ToList().ForEach(e => Console.WriteLine($"{e}"));

Console.WriteLine();
entiers.TakeWhile(e => e < 5).
    OrderBy(e => e).
    ToList().ForEach(e => Console.WriteLine($"{e}"));
```



```
Console de débogage Microsoft Visual Studio
-30; -25; 1; 3;
1; 3;
```

3.9 LA PROJECTION

Il est envisageable de produire une nouvelle séquence dont les éléments auront un schéma différent des éléments de la séquence d'origine en :

- restreignant le nombre des attributs retenus dans le résultat dans un souci d'efficacité et de limitation des ressources utilisées. Nous verrons par la suite qu'il s'agit aussi de restreindre les volumes de données échangées sur le réseau.
- ajoutant des éléments calculés
- ajoutant des attributs provenant d'autres objets du système associés via une relation.

Ces résultats peuvent être produits sous forme d'objets typés (attributs définis dans des classes) ou anonymes.

Nous pouvons ne retenir dans notre sélection d'éléments que les propriétés qui nous intéressent.

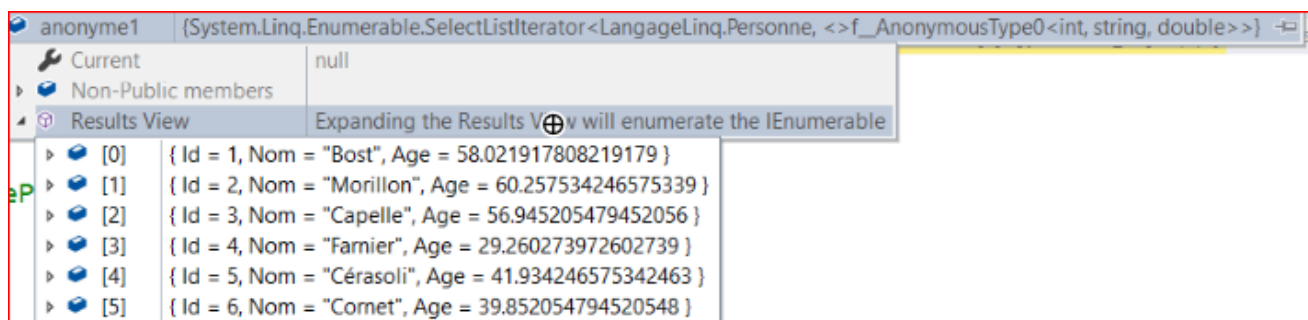
Les deux requêtes suivantes ne retiennent que certaines propriétés, dont les noms peuvent être redéfinis ou non. Les types sont anonymes.

Les objets qui n'ont pas de type doivent être consommés sur place....

Ils ne peuvent pas être retournés en résultat d'une méthode pour être utilisés en dehors du contexte local où s'exécute la requête.

Pour être utilisés en dehors, il vous faudra définir des structures ou classes dans le respect des exigences d'un langage fortement typé.

Vous pouvez observer la définition du type anonyme



Dans cette deuxième requête, qui produit des objets sans définition statique (anonymes), j'ai redéfini le type de l'âge et appliqué un arrondi de la valeur. J'ai aussi redéfini le nom des autres attributs.

```
var anonyme2 = personnes.Select(p => new
{
    identifiant = p.Id,
    NomPersonne = p.Nom,
    Age = ((decimal)Math.Round(p.Age, 2))
});
anonyme2.ToList().
    ForEach(per => Console.WriteLine($"{per.identifiant};{per.NomPersonne};{per.Age}"));
```

Extrait du résultat de la deuxième requête :

```
1;Bost;58,02
2;Morillon;60,26
3;Capelle;56,95
```

3.10 RECUPERER UN ELEMENT UNIQUE

Pour exprimer des requêtes simples comme le retour d'un élément unique, nous utiliserons les méthodes **Single**, **SingleOrDefault**.

Autre possibilité est de récupérer un élément en position avec **First** ou **Last**.

```
Personne personne = personnes.FirstOrDefault(p => p.Id == 1);
```

A noter :

Si nous utilisons la méthode **Single** et qu'aucun élément correspondant au filtre n'est renvoyé, nous obtenons une erreur (de même si plusieurs éléments sont renvoyés).

Si nous utilisons la méthode **SingleOrDefault** et qu'aucun élément ne fait partie du jeu de résultats, nous obtiendrons la valeur par défaut, **null** pour les types références.

Les autres méthodes qui permettent de renvoyer un élément unique fonctionnent de manière similaire. Voir le tableau des opérateurs de requête standards, catégorie **Elements**.

3.11 LES QUANTIFICATEURS

Pour déterminer si une valeur existe dans une séquence, l'opérateur de requête standard **Any** est tout à fait indiqué.

Pour savoir si tous les éléments d'une séquence respectent la condition, vous utiliserez alors **All**.

Les quantificateurs tels qu'**Any**, **All** et **Contains** analysent une séquence d'éléments et déterminent si la séquence répond à la condition d'une expression lambda.

Contains peut être utilisé pour vérifier qu'un élément n'existe pas dans une séquence avant de l'ajouter.

A noter : **Contains** prendra en compte la redéfinition du principe d'égalité (surcharge des méthodes **Equals** et **GetHashCode**, implémentation de l'interface **IComparer**). A défaut, il retiendra l'égalité de référence ou de valeur.

Exemple : tous les nombres impairs :

Résultat :

```
private bool impairs()
{
    List<int> entiers = new List<int>() { 1, 3, 5, 25 };
    return entiers.All(a => a % 2 != 0);
}
```

True

Exemple : au moins un nombre pair :

Résultat :

```
private bool unPair()
{
    List<int> entiers = new List<int>() { 1, 4, 5, 25 };
    return entiers.Any(a => a % 2 == 0);
}
```

True

Exemple : ajouter le nombre s'il n'est pas présent dans la liste

```
if (!entiers.Contains(5))
{
    entiers.Add(5);
}
```

3.12 LES AGREGATIONS

Les requêtes d'agrégation sont exécutées immédiatement pour produire les valeurs d'agrégats.

Il n'est pas possible de construire des agrégats sur des types anonymes.

Les agrégations sont souvent associées à des projections et des requêtes corrélées.

3.12.1 Le comptage distinct

Comptage du nombre de valeurs uniques dans une liste

```
List<int> entiers = new List<int> { 0, 0, 2, 3, 4, 4 };  
Console.WriteLine($"Nombre de valeurs distinctes {entiers.Distinct().Count()}");
```

```
Nombre de valeurs distinctes 4
```

3.12.2 Mettre en cache une valeur d'agrégat

Si nous devons introduire le résultat d'un agrégat dans l'expression de la requête, il est préférable, lorsque c'est possible, de le calculer à part et de le mettre en cache.

Pour la première requête maxi, le calcul du max se fait autant de fois qu'il y a de valeurs dans la liste.

La deuxième version est plus efficiente. Le max est calculé une seule fois et mis en cache pour être utilisé par la suite.

```
int[] table = new int[] { 0, 9, 8, 1, 2, 3, 4, 6, 5, 7, 25, -51, 25 };  
List<int> maxi = table.Where(i => i == table.Max()).ToList();  
  
int max=table.Max();  
List<int> maxi2 = table.Where(i => i == max).ToList();
```

3.13 PARTITIONNEMENT

Les opérateurs de partitionnement nous permettent par exemple de mettre en œuvre des mécanismes de pagination simples.

Certains d'entre vous ont peut-être déjà mis en place de tels mécanismes en SQL avec ou sans recours aux expressions de table commune.

Nous allons ici apprécier la simplicité de l'expression Linq.

Nous aurons ici besoin de deux variables de type intégral pour retourner les objets de la page courante.

- numeroPageCourante.
- taillePage : nombre d'objets par page

Et de deux opérateurs :

- Skip(x) : permet de passer outre les x premiers éléments de la liste
- Take(y) : permet de retenir y éléments max dans la liste

Nous reverrons évidemment ces mécanismes avec Linq to Entities.

```
public static void Pagination()
{
    List<Personne> personnes = GetPersonnes();
    int taillePage = 3;
    int numeroPage = 2;

    // Liste des personnes avec mail

    List<Personne> pageDemandee =
        personnes.Where(p => p.AdresseMail!=null)
        .OrderBy(p => p.Nom)
        .Skip((numeroPage - 1) * taillePage)
        .Take(3)
        .ToList();
}
```



Attention à l'ordre dans lequel vous invoquez les opérateurs de partitionnement. Ordonnez la séquence avant partition.

3.14

REALISER LES EXERCICES DU CHAPITRE 2 ET SUIVANTS DE P-LINQTOOBJECT

Réalisez les exercices sur les requêtes linq.

4. OPERATEURS DE REQUETE STANDARD

La liste non exhaustive des opérateurs standards utilisables dans vos requêtes.

Agrégation

Aggregate	Exécute une méthode personnalisée sur une séquence
Average	Calcule la moyenne d'une séquence de valeurs numériques
Count	Renvoie le nombre d'éléments d'une séquence sous la forme d'un entier
LongCount	Renvoie le nombre d'éléments d'une séquence sous la forme d'un nom long
Min	Recherche le plus petit nombre d'une séquence de nombres
Max	Recherche le plus grand nombre d'une séquence de nombres
Sum	Additionne les nombres d'une séquence

Concaténation

Concat	Concatène deux séquences en une seule
--------	---------------------------------------

Conversion

Cast	Convertit les éléments d'une séquence dans un type donné
OfType	Filtre les éléments d'une séquence d'un type donné
ToArray	Renvoie un tableau à partir d'une séquence
ToDictionary	Renvoie un dictionnaire à partir d'une séquence
ToList	Renvoie une liste à partir d'une séquence

ToLookup	Renvoie une recherche à partir d'une séquence
ToSequence	Renvoie une séquence IEnumerable

Élément

DefaultIfEmpty	Crée un élément par défaut pour une séquence vide
ElementAt	Renvoie l'élément à un index donné dans une séquence
ElementAtOrDefault	Renvoie l'élément à un index donné dans une séquence une valeur par défaut si l'index est en dehors de la plage
First	Renvoie le premier élément d'une séquence
FirstOrDefault	Renvoie le premier élément d'une séquence ou une valeur par défaut si aucun élément n'est trouvé
Last	Renvoie le dernier élément d'une séquence
LastOrDefault	Renvoie le dernier élément d'une séquence ou une valeur par défaut si aucun élément n'est trouvé
Single	Renvoie l'élément unique d'une séquence
SingleOrDefault	Renvoie l'élément unique d'une séquence ou une valeur par défaut si aucun élément n'est trouvé

Égalité

SequenceEqual	Compare deux séquences pour voir si elles sont équivalentes
---------------	---

Génération

Empty	Génère une séquence vide
Range	Génère une séquence pour une plage donnée
Repeat	Génère une séquence en répétant un élément x fois

Regroupement

GroupBy	Regroupe les éléments d'une séquence en fonction d'un critère donné
---------	---

Jointure

GroupJoin	Exécute une jointure regroupée sur deux séquences
Join	Exécute une jointure intérieure sur deux séquences

Tri

OrderBy	Trie une séquence par valeur(s) dans l'ordre croissant
---------	--

OrderByDescending	Trie une séquence par valeur(s) dans l'ordre décroissant
ThenBy	Trie une séquence déjà triée dans l'ordre croissant
ThenByDescending	Trie une séquence déjà triée dans l'ordre décroissant
Reverse	Inverse l'ordre des éléments d'une séquence

Partitionnement

Skip	Renvoie une séquence qui ignore un nombre donné d'éléments
SkipWhile	Renvoie une séquence sans les éléments non conformes à l'expression
Take	Renvoie une séquence qui prend un nombre donné d'éléments
TakeWhile	Renvoie une séquence qui prend les éléments conformes à l'expression

Projection

Select	Crée une projection de parties d'une séquence
SelectMany	Crée une projection « un à plusieurs » de parties d'une séquence

Quantificateurs

All	Détermine si tous les éléments d'une séquence remplissent une condition
Any	Détermine si des éléments d'une séquence remplissent une condition
Contains	Détermine si une séquence contient un élément donné

Restriction

Where	Filtre les éléments d'une séquence
-------	------------------------------------

Série

Distinct	Renvoie une séquence sans éléments dupliqués
Except	Renvoie une séquence représentant la différence entre deux séquences
Intersect	Renvoie une séquence représentant l'intersection de deux séquences
Union	Renvoie une séquence représentant l'union de deux séquences

Inclusion

Include	Inclut des éléments connexes (enfants) dans la requête
---------	--