Linq CS

02 - Requêter avec LINQ

TBDC
Thomas BUREAU DU COLOMBIER

Les listes

TBDC
Thomas BUREAU DU COLOMBIER

List<T>

- Pour fonctionner, LINQ a besoin que la collection implémente l'interface IEnumerable<T>
  - Le type List<T> implémente l'interface IEnumerable<T>, c'est lui qu'on va utiliser en majorité
- Pour transformer une collection en list, on peut utiliser la fonction .ToList()
- Vous devez avoir une référence vers le namespace System.Collections.Generic

```
using System.Collections.Generic;
```

List<T> est une liste d'éléments ayant pour type T.

Initialiser une liste

```
//Initialiser une liste de chaînes de caractères
List<string> maliste;
maliste = new List<string>
{
    "élément 1",
    "élément 2"
};
```

La classe LIST implémente de nombreuses méthodes permettant de faire des recherches, ou d'ajouter ou de supprimer des éléments. Pour cela, l'autocomplétion vous sera très utile !

Pour plus d'informations :
https://docs.microsoft.com/fr-fr/dotnet/api/system.collections.generic.list-1?view=net-6.0

Faire une requête

</>

TBDC
Thomas BUREAU DU COLOMBIER

2 manières d'écrire les requêtes

LINQ fournit deux syntaxes pour l'écriture de requêtes

- Syntaxe requête
- Syntaxe méthode

Pour utiliser LINQ, vous devrez ajouter une référence au namespace System.Linq

```
using System.Linq;
```

2 manières d'écrire les requêtes

Syntaxe requête

- Conseillée par Microsoft
- Ressemble au SQL
- Plus lisible

- Ne prends pas en charge tous les opérateurs de requête

```
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;
```

Syntaxe méthodes

- Plus facile à écrire pour des requêtes basiques
- (personnellement, je l'ai plus vu en entreprise)

- Moins lisible

```
var numQuery = numbers.Where(num => num % 2 == 0);
```

Commençons à apprendre avec la syntaxe requête

```
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;
```

Opérations de requête LINQ

Toutes les opérations de requête LINQ se composent de trois actions distinctes :

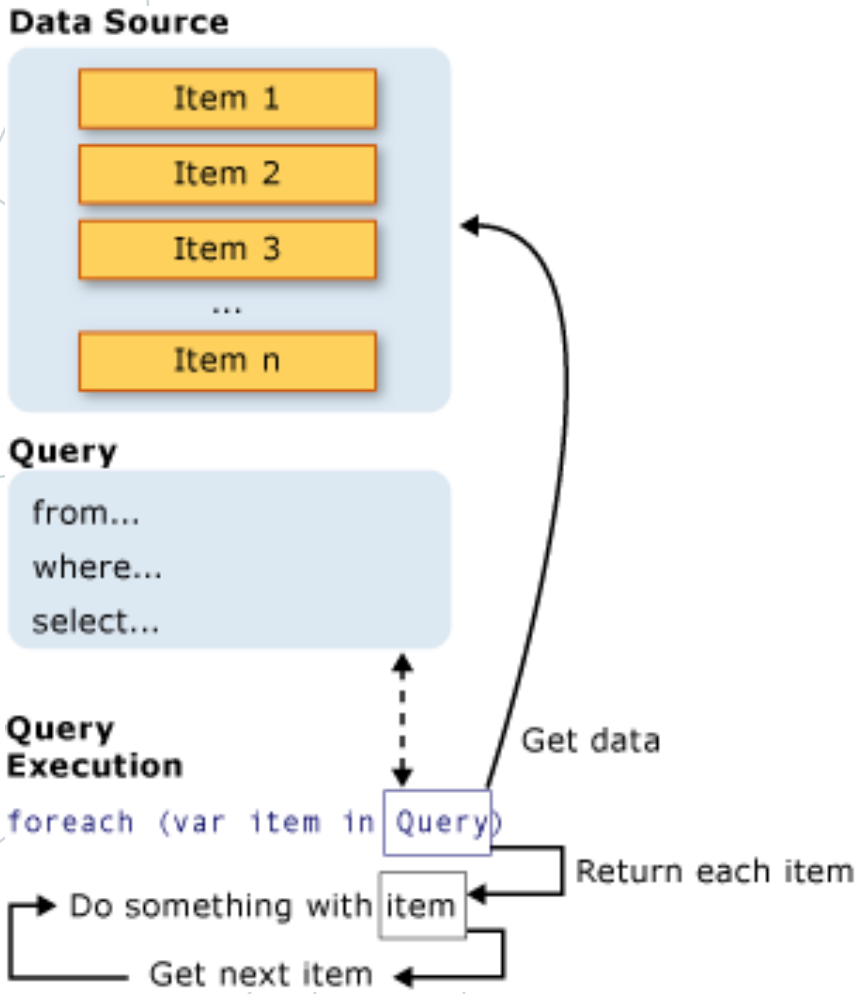
1. Obtention de la source de données
2. Création de la requête
3. exécutez la requête.

```
// The Three Parts of a LINQ Query:
// 1. Data source.
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

// 3. Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

Opérations de requête LINQ




```
// The Three Parts of a LINQ Query:
// 1. Data source.
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

// 3. Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

Dans LINQ, l'exécution de la requête est distincte de la requête elle-même. En d'autres termes, vous n'avez récupéré aucune donnée en créant simplement une variable de requête. La récupération des données s'opère ici lors de l'énumération de la liste.

Opérations de requête LINQ

SOURCES

```
// objects
var scores = new List() { 7, 5, 4, 3, 8, 5  };

// XML
XElement characters = XElement.Load(@"c:\characters.xml");

// DB using Entity Framework
using (var db = new MovieEntities())
{
    var movies = db.Movies
}
```

REQUETE

```
IQueryable<type> =
FROM row in <data source>
WHERE <condition>
SELECT <columns from row>
```

EXECUTION

```
foreach(var r in result) {
  Console.WriteLine("Score: {0}", r);
}
```

Anatomie de notre requête

- Source :
  - from <élément> in <séquence>
- Conditions :
  - where <condition>
- Eléments à retourner :
  - select <sélection>

```
// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;
```

```
from <optional data type> <range variable> in <IEnumerable<T>>
<Query Operator> lambda expression
<select> <range variable | fields of range variable>
```

Equivalence de code

Je veux récupérer les éléments inférieurs à 5 de ma liste

```
var getTheNumbers = from number in numbers
                    where number < 5
                    select number;
```

LINQ est plus lisible

```
List<int> newNumbers = new List<int>();

foreach (var number in numbers)
{
    if (number < 5)
    {
        newNumbers.Add(number);
    }
}
```

Continuons avec la syntaxe Méthode

```
var numQuery = numbers.Where(num => num % 2 == 0);
```

Pour maitriser la syntaxe méthodes, il faut maîtriser d'autres notions avant ça :

- Méthodes d'extension
- Méthodes anonymes
- Délégué
- Expression lambda

Méthodes d'extension

Tous les langages LINQ sont basés sur des méthodes d'extension. Les méthodes d'extension sont des méthodes statiques d'une classe statique qui peuvent être appelées comme une méthode d'instance. Ces méthodes sont utiles lorsque nous devons ajouter de nouveaux comportements à une classe existante sans modifier la classe.

Nous ne pouvons pas introduire de nouvelles méthodes dans la classe string car nous ne pouvons pas modifier le code source de la classe string. Nous créons donc une nouvelle méthode d'extension basée sur la classe de chaînes.

Nous déclarons d'abord une classe statique StringExtensionMethods et déclarons une méthode AddComma. Ce premier paramètre de la méthode doit commencer par ceci et taper le nom sur lequel nous voulons ajouter une nouvelle méthode.

Dans l'exemple ci-dessus, la variable de nom de chaîne a maintenant un nouveau nom de méthode AddComma qui ajoute "," à la fin de la chaîne.

```
class Program
{
    static void Main(string[] args)
    {
        string name = "Kapil";

        name = name.AddComma();

        Console.WriteLine(name); //Print Kapil,
    }
}

static class StringExtensionMethods
{
    public static string AddComma(this string input)
    {
        return input + ",";
    }
}
```

Exemple avec OrderBy

OrderBy

L'exemple suivant montre comment appeler la méthode opérateur de requête standard OrderBy sur un tableau d'entiers.

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Types anonymes

Les types anonymes sont des classes temporaires utiles pour stocker des résultats intermédiaires.

Ces types n'ont pas toutes les caractéristiques des types réguliers. Voici les restrictions sur les types anonymes.

- Les types anonymes ne peuvent contenir que des champs publics.
- Les champs de types anonymes doivent être initialisés.
- Les types anonymes ne peuvent spécifier aucune méthode.
- Les types anonymes ne peuvent pas implémenter une interface ou une classe abstraite.

```
var var1 = new { FirstName = "Kapil", LastName = "Malhotra" };
Console.WriteLine(var1.FirstName + " " + var1.LastName); //Print Kapil Malhotra
```

Dans l'exemple ci-dessus, nous avons créé un type anonyme qui a deux champs publics FirstName et LastName. Nous n'avons spécifié aucun nom de classe après le nouveau mot-clé. Ce nouveau type anonyme n'a que deux propriétés et aucune méthode.

Délégués

Les délégués nous permettent de stocker une référence de fonctions qui peuvent être exécutées au moment opportun. Dans le délégué, nous ne pouvons stocker que les fonctions qui ont des paramètres et des types de retour correspondants.

Par exemple, si nous déclarons un délégué qui accepte un paramètre int et retourne le résultat de la chaîne. Ensuite, dans ce délégué, nous pouvons uniquement attribuer une référence aux fonctions qui prennent int comme paramètre unique et retournent le résultat de la chaîne.

Dans l'exemple ci-dessus, nous déclarons un simple délégué CalcSum qui prend deux paramètres int et retourne le résultat int.

Nous déclarons deux fonctions Sum et WrongSumFunction. Seule la fonction Sum a deux paramètres int et retourne dans result. Nous pouvons facilement attribuer cette méthode à l'instance CalcSum. La deuxième fonction WrongSumFunction prend un seul paramètre et renvoie les résultats int. Étant donné que WrongSumFunction n'a qu'un seul paramètre int, nous ne pouvons pas affecter cette méthode à l'instance de délégué CalcSum comme indiqué dans l'exemple ci-dessus et cela donne une erreur au moment de la compilation.

```
class Program
{
    delegate int CalcSum(int a, int b);

    static void Main(string[] args)
    {
        CalcSum sumFunctions = Sum;
        int result = Sum(4,5);
        Console.WriteLine(result); //Prin

        sumFunctions = WrongSumFunction;
    }

    static int Sum(int a, int b)
    {
        return a + b;
    }

    static int WrongSumFunction(int a)
    {
        return a + a;
    }
}
```

Méthodes anonymes

Les méthodes anonymes sont une méthode en ligne qui n'a qu'un corps sans nom. Nous ne pouvons définir que les paramètres et renvoyer les types des méthodes.

Voici un exemple de méthode anonyme.

```
class Program
{
    delegate int MyDelegate(int x, int y);
    static void Main(string[] args)
    {
        MyDelegate multiplyMethod = delegate(int x, int y)
        {
            return x * y;
        };

        int result = multiplyMethod(2, 4);
        Console.WriteLine(result); //Print 8
    }
}
```

Dans l'exemple ci-dessus, nous créons un délégué et lui attribuons une méthode en ligne sans nom. La méthode doit avoir le même nombre de paramètres et le même type que la correspondance avec le type de délégué.

Délégué Action<T>

Le délégué d'action est un délégué prédéfini par .NET Framework qui prend uniquement des paramètres et ne renvoie aucune valeur. Nous pouvons spécifier au maximum 16 paramètres dans le délégué Action.

```
Action<int, int> Multipler = delegate(int x, int y)
{
    Console.WriteLine(x * y);
};
Multipler(4, 5);    // Print 20
```

Délégué Predicate<T>

Le délégué de prédicat est un délégué qui n'accepte qu'un seul paramètre et renvoie une valeur booléenne.

Le délégué de prédicat est un délégué qui n'accepte qu'un seul paramètre et renvoie une valeur booléenne.

```
1    static void Main(string[] args)
2    {
3        Predicate<string> hasValueK = delegate(string par)
4        {
5            return par.Contains("K");
6        };
sept     bool result1 = hasValueK("Kapil");
8        bool result2 = hasValueK("Malhotra");
9
dix      Console.WriteLine(result1); // Print true
11       Console.WriteLine(result2); // Print false
12   }
```

Dans l'exemple ci-dessus, nous spécifions un délégué de prédicat hasValueK qui ne prend qu'un seul paramètre de chaîne et renvoie un résultat booléen.

Délégué Func<T>

Le délégué Func est également un délégué générique fourni par .NET Framework. Dans le délégué Func, nous pouvons spécifier 16 paramètres et un type de retour.

```
static void Main(string[] args)
{
    Func<int,int, int> multiplier = delegate(int a, int b)
    {
        return a * b;
    };
    int result = multiplier(2, 9);

    Console.WriteLine(result); // Print 18
}
```

Dans l'exemple ci-dessus, nous spécifions un délégué Func <int, int, int>. Le dernier int est un type de résultat et les deux premiers int sont des types de paramètres.

Expressions lambda

Les expressions Lambda ne sont qu'un raccourci pour écrire des méthodes anonymes. => est un opérateur lambda. À gauche de l'opérateur lambda, nous spécifions les paramètres d'entrée et à droite, nous écrivons le corps de la méthode.

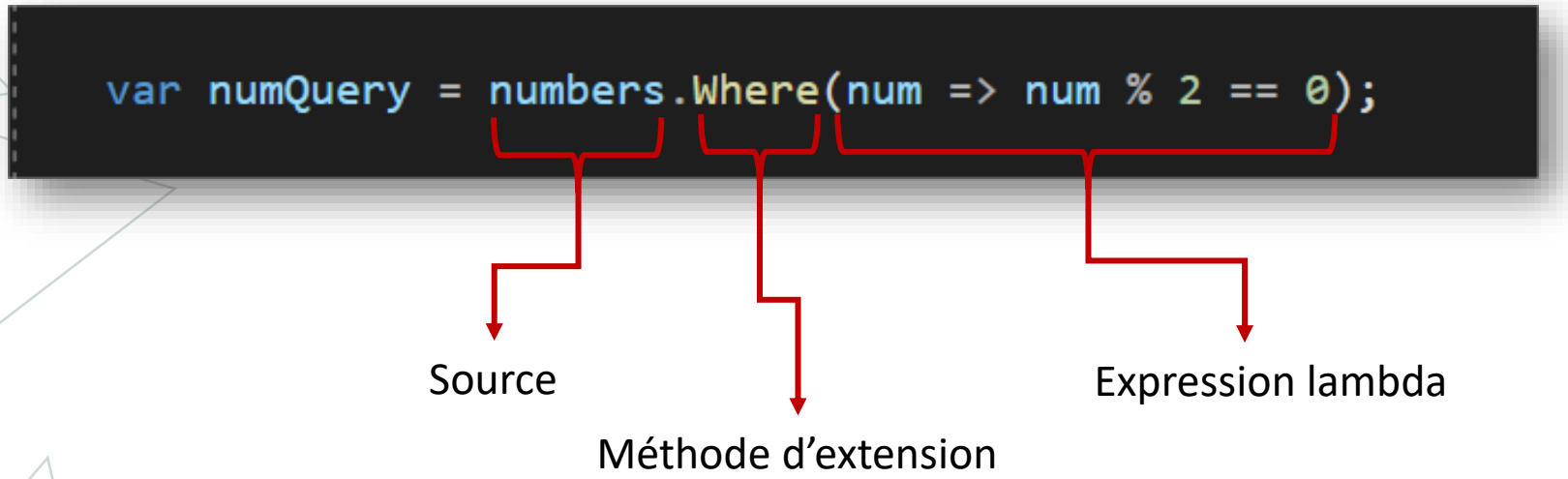
```
delegate int MultiplierDelegate(int a, int b);
static void Main(string[] args)
{

    MultiplierDelegate multiplier = (a, b) => a * b;
    int result = multiplier(2, 9);

    Console.WriteLine(result); // Print 18
}
```

La syntaxe méthode

- Elle utilise des méthodes d'extension
- On va utiliser des expressions lambda



La syntaxe méthode

- Pour rappel, trois étapes (source, requête, exécution).
- Pour ajouter des traitements à la liste, on va enchaîner les méthodes
- On remarque qu'avec l'enchainement des méthodes, cela devient moins lisible.

```
1   using System;
2   using System.Linq;
3
4   namespace TestLINQipi
5   {
        0 références
6       class Program
7       {
            0 références
8           static void Main(string[] args)
9           {
10              // The Three Parts of a LINQ Query:
11              // 1. Data source.
12              int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
13
14              // 2. Query creation.
15              // numQuery is an IEnumerable<int>
16              var numQuery = numbers.Where(num => num % 2 == 0)
17                                    .OrderByDescending(num => num)
18                                    .Select(num => num);
19
20              // 3. Query execution.
21              foreach (int num in numQuery)
22              {
23                  Console.Write("{0,1} ", num);
24              }
25          }
26      }
27  }
```

Les types de fonctions LINQ

LINQ fournit plus de 50 opérateurs de requête pour différentes fonctionnalités. Chaque opérateur de requête est une méthode d'extension. Ces opérateurs peuvent être classés dans les catégories suivantes:

Operator Category	LINQ Query Operators Names
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending
Set	Except, Intersect, Union, Distinct
Quantifier	All, Any, Contains
Projection	Select, SelectMany
Partitioning	Skip, SkipWhile, Take, TakeWhile
Join	Join, GroupJoin
Grouping	GroupBy, ToLookup
Sequencing	DefaultIfEmpty, Empty, Range, Repeat
Equality	SequenceEqual
Element	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Conversion	AsEnumerable, AsQueryable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup
Concatenation	Concat
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum

Les types de fonctions LINQ

Filtering Operators

Filter operators are used to select only those elements from sequence that satisfy a condition. For example, suppose we have ten names in a collection sequence and we have to filter out those names that start with "K".

Filtering Operators	Description
Where	Filter elements based on the condition.
OfType	This operator takes a Type name and select only those elements in the collection that have matching type. For example, we have a collection of base class (List<BaseClass>) and we have added some items of derived class in the collection. Now we have to find out those item that have of type derived type OfType<DerivedClass>.

Sorting Operators

These operators are used to order the elements in a sequence either ascending or in descending order. We can apply sort elements multiple fields.

Sorting Operators	Description
OrderBy	Sort elements in ascending order
OrderByDescending	Sort elements in descending order
ThenBy	Again sort elements in ascending order. Must use after OrderBy or OrderByDescending operator. This operator is useful when we have to apply sorting based on multiple fields. For example, we want to sort first based on name and then age.
ThenByDescending	Again sort elements in descending order. Must use after OrderBy or OrderByDescending operator. This operator is useful when we have to apply sorting based on multiple fields. For example, we want to sort descending first based on age and then name.

Les types de fonctions LINQ

Set Operators

These LINQ query operators are used to find common elements, unique elements, missing elements between two collection sequences.

Set Operators	Description
Distinct	Removes duplicates elements from single sequence
Except	Returns elements which are not present in second sequence
Intersect	Returns common elements between two sequences
Union	Returns unique elements between two sequence

Quantifier Operators

These LINQ query operators are used in conditional statements like if and switch to test whether any or all elements in a sequence satisfy a condition.

Quantifier Operators	Description
All	Returns true when all elements in a sequence satisfy a condition else returns false.
Any	Returns true when any single element in a sequence satisfy a condition else returns false
Contains	Returns true when a sequence contains a matching element else returns false.

Les types de fonctions LINQ

Projection Operators

These LINQ query operators are used in create a new type by choosing only those objects or properties that we need in our result set. Projection operators provides cross joins and non equi joins like functionality.

For example, suppose we use two objects in a LINQ query and in the result set we only need two properties of first object and one property of second object so we create a new object by using projection operator that has three properties and return that object from query.

Projection Operators	Description
Select	Transform each element by using lambda expression. For example, students.Select(w => w.Name)
SelectMany	We use this operator when each element is a collection and SelectMany transform each collection in a single collection and returns as a result set. For example, we have two elements {"First", "Second"} and {"Third","Fourth"} and SelectMany returns {"First", "Second", "Third", "Fourth"}

Partitioning Operators

These LINQ query operators are used to filter out elements based on the indexes or by a condition in a sequence. These operators are also used for implementing paging.

Partitioning Operators	Description
Take	Returns only those elements upto specify index. For example, returns only first first elements in a sequence.
TakeWhile	Returns first matching elements which satisfies a given condition. For example, we have numbers { 1, 2, 9, 10, 3, 4} in a sequence and we want to filter out elements which are smaller than 5 then TakeWhile returns only { 1, 2} in a result set as they are appearing first in a list and 9 element dissatisfy this condition.
Skip	Returns only those elements after specify index. For example, skip first five elements and returns all rest elements.
SkipWhile	Skip those elements that satisfies a given condition

Les types de fonctions LINQ

Join Operators

These LINQ query operators are used to combine multiple sequence into one sequence just like joins in SQL queries. Join operators offers inner join and left outer joins like functionality.

Join Operators	Description
Join	Joins two sequences based on matching keys.
GroupJoin	Joins two sequence based on matching keys but returns hierarchical output.

Grouping Operators

These LINQ query operators are used to group the data based on specific keys just like groups in SQL.

Grouping Operators	Description
GroupBy	Group elements based on specific key.
ToLookup	Group elements and returns as (Key, Value) pair objects.

Sequencing Operators

These LINQ query operators are used to create a new sequence of values.

Sequencing Operators	Description
DefaultIfEmpty	Returns a default blank sequence. Mainly used for adding default element if source collection is empty.
Empty	Returns an empty sequence.
Range	Returns a range a numeric numbers. For example, we need a new sequence starting from 1 to 10.
Repeat	Returns a sequence of repeating same value at specific number of times. For example, we need five elements of value "Default" in a collection.

Les types de fonctions LINQ

Equality Operators

There is only one operator in this category. This operator is used to compare sequences.

Equality Operators	Description
SequenceEqual	Compares two sequences and returns true if they are exact match. Also takes a second parameter of IEqualityComparer<T> if we want to use different comparer.

Element Operators

These LINQ query operators are used to find element at specific index in a sequence.

Element Operators	Description
ElementAt	Returns an element at specific index in a sequence. Throws ArgumentOutOfRangeException exception when index is outside length of sequence.
ElementAtOrDefault	Returns an element at specific index in a sequence. If element not found returns a blank value.
First	Returns first element in a sequence or first element that satisfy a condition. Throws InvalidOperationException exception when sequence is empty.
FirstOrDefault	Returns first element in a sequence or first element that satisfy a condition. If sequence is empty or no element matches the condition then returns default value.
Last	Returns last element in a sequence or last element that satisfy a condition. Throws InvalidOperationException exception when sequence is empty.
LastOrDefault	Returns last element in a sequence or last element that satisfy a condition. If sequence is empty or no element matches the condition then returns default value.
Single	Returns single element in a sequence or element that satisfy a condition. If a sequence has more than one elements then throws InvalidOperationException exception.
SingleOrDefault	Returns single element in a sequence or element that satisfy a condition. If sequence is empty or no element matches the condition then returns default value.

Les types de fonctions LINQ

Conversion Operators

These LINQ query operators are used to convert element of sequence to another data types.

Conversion Operators	Description
AsEnumerable	Returns a new sequence of IEnumerable<T>.
AsQueryable	Returns a new sequnce of IQueryable<T>.
Cast	Cast the element to a specific type.
OfType	This operator takes a Type name and select only those elements in the collection that have matching type. For example, we have a collection of base class (List<BaseClass>) and we have added some items of derived class in the collection. Now we have to find out those item that have of type derived type OfType<DerivedClass>.
ToArray	Returns a new sequence of an array[] data type
ToDictionary	Returns a new sequence of generic Dictionary<Key,Value> type.
ToList	Return a new sequence of generic List type.
ToLookup	Group elements and returns as (Key, Value) pair objects.

Les types de fonctions LINQ

Concatenation Operators

There is only one operator in this category. Used to create new collection based on two sequences.

Concatenation Operators	Description
Concat	Concatenate two sequences and create one new sequence joining all the elements of both sequences.

Aggregation Operators

These LINQ query operators are used to compute mathematical functions like sum, average, count, max and min operators on elements.

Aggregation Operators	Description
Average	Computer average values of all elements in a sequence.
Count	Counts the number of elements in a sequence.
LongCount	Count the number of elements in a huge sequence. Use LongCount where a sequence has more than int.MaxValue elements. Returns a long data type.
Max	Returns the maximum value in a sequence.
Min	Returns the minimum value in a sequence.
Sum	Returns the sum of all elements in a sequence.
Aggregate	Use for performing custom aggregation operation on a sequence.

Mixer requête et méthode

- Pour requêter une grappe d'objets :

```
List<int> oListeEntiers = new List<int>() { 1, 2, 5, 8, 9, 12, 15, 8 };
```

La source de données implémente `IEnumerable<int>`

```
oListeEntiers.Where(i => i % 2 == 0);
```

Méthodes d'extension et expressions lambda

```
from i in oListeEntiers
where i % 2 == 0
select i;
```

LINQ

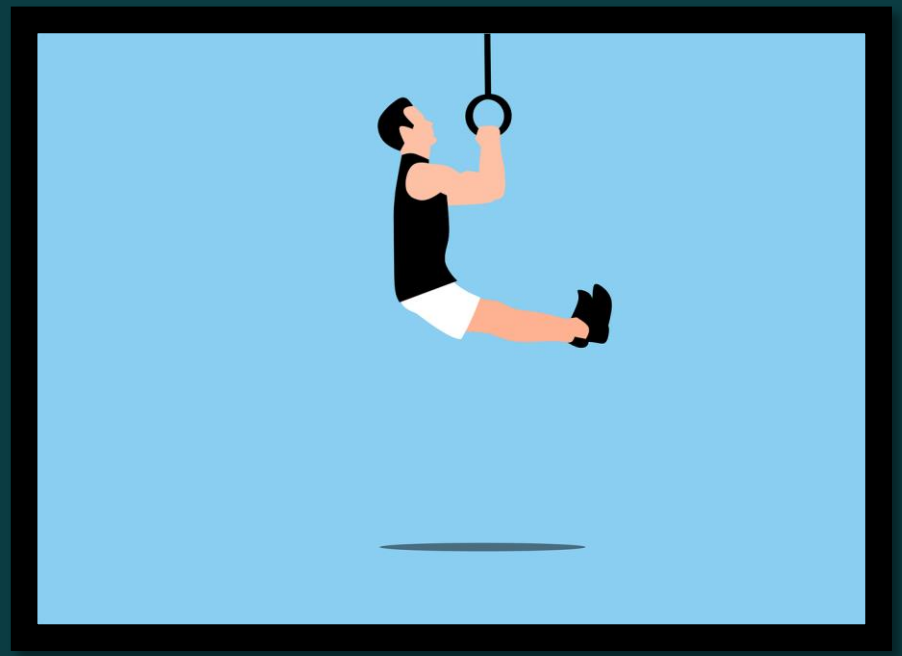
Type de données retourné `IEnumerable<int>`

Mixte des deux

```
(from i in oListeEntiers
where i % 2 == 0
select i).Distinct().ToList();
```

Type de données retourné `List<int>`

Un peu de pratique



TBDC
Thomas BUREAU DU COLOMBIER

Rappel en bref

- Méthodes
  - Aggregate
  - All
  - Any
  - Append
  - AsEnumerable
  - Average
  - Cast
  - Concat
  - Contains
  - Count
  - DefaultIfEmpty
  - Distinct
  - ElementAt
  - ElementAtOrDefault
  - Empty
  - Except
  - First
  - FirstOrDefault
  - GroupBy
  - GroupJoin
  - Intersect
  - Join
  - Last
  - LastOrDefault
  - LongCount
  - Max
  - Min
  - OfType
  - OrderBy
  - OrderByDescending
  - Prepend
  - Range
  - Repeat
  - Reverse
  - Select
  - SelectMany
  - SequenceEqual
  - Single
  - SingleOrDefault
  - Skip
  - SkipLast
  - SkipWhile
  - Sum
  - Take
  - TakeLast
  - TakeWhile
  - ThenBy
  - **ThenByDescending**
  - ToArray
  - ToDictionary
  - ToHashSet
  - ToList
  - ToLookup
  - Union
  - Where
  - Zip

```
int[] numbers = { 1, 2, 3, 4 };
string[] words = { "one", "two", "three" };

var numbersAndWords = numbers.Zip(words, (first, second) => first + " " + second);

foreach (var item in numbersAndWords)
    Console.WriteLine(item);

// This code produces the following output:

// 1 one
// 2 two
// 3 three
```

Récupérer le repository

- On va faire une application console pour tester LINQ
  - En .NET Core 3.1

https://github.com/ThomasBDC/LinqExercicePresentation.git

Exercice :

- On a une liste d'album, une classe qui a ce modèle :

```
99+ references
public class Album
{
    1 reference
    public int AlbumId { get; set; }
    1 reference
    public string Title { get; set; }
    1 reference
    public int ArtistId { get; set; }

    99+ references
    public Album(int albumId, string title, int artistId)
    {
        AlbumId = albumId;
        Title = title;
        ArtistId = artistId;
    }
}
```

- Vous pouvez récupérer une liste d'album via cette variable :

```
var listAlbums = ListAlbumsData.ListAlbums;
```

Le Select : Fonction de projection

- Les fonctions qui prennent une collection de type T et retourne une collection de type U (où U peut être le même type que T).

Select() SelectMany Cast<T>()

```
//Le but est d'ajouter des heures à la date d'aujourd'hui
// Exemple fonction de projection
Console.WriteLine($"Date du jour {DateTime.Today.ToShortDateString()}");
Console.WriteLine("--- Fonction de projection ---");
IEnumerable<int> numberOfDays = new[] { 122, 3600, 153, 24, 445, 552};
DateTime[] dates = numberOfDays.
       // On projette chaque élément pour leur donner une nouvelle forme
       Select(x => DateTime.Today.AddHours(x))
              .ToArray();
Console.WriteLine("Nouvelles dates suite à la projection ");
foreach (var date in dates)
{
    Console.WriteLine($"{date.ToShortDateString()}");
}
```

```
Date du jour 07/01/2022
--- Fonction de projection ---
Nouvelles dates suite à la projection
12/01/2022
06/06/2022
13/01/2022
08/01/2022
25/01/2022
30/01/2022
```

1er exercice : Afficher les albums

- Pour commencer, on veut afficher les albums dans notre console.
- On veut récupérer via Linq une liste de chaînes de caractères déjà formaté pour l'affichage.
- Voici le résultat voulu →

```
Album n°1 : For Those About To Rock We Salute You
Album n°2 : Balls to the Wall
Album n°3 : Restless and Wild
Album n°4 : Let There Be Rock
Album n°5 : Big Ones
Album n°6 : Jagged Little Pill
Album n°7 : Facelift
Album n°8 : Warner 25 Anos
Album n°9 : Plays Metallica By Four Cellos
Album n°10 : Audioslave
Album n°11 : Out Of Exile
Album n°12 : BackBeat Soundtrack
Album n°13 : The Best Of Billy Cobham
Album n°14 : Alcohol Fueled Brewtality Live! [Disc 1]
Album n°15 : Alcohol Fueled Brewtality Live! [Disc 2]
Album n°16 : Black Sabbath
Album n°17 : Black Sabbath Vol. 4 (Remaster)
Album n°18 : Body Count
Album n°19 : Chemical Wedding
Album n°20 : The Best Of Buddy Guy - The Millenium Collection
Album n°21 : Prenda Minha
Album n°22 : Sozinho Remix Ao Vivo
Album n°23 : Minha Historia
Album n°24 : Afrociberdelia
Album n°25 : Da Lama Ao Caos
Album n°26 : Acústico MTV [Live]
Album n°27 : Cidade Negra - Hits
Album n°28 : Na Pista
Album n°29 : Axé Bahia 2001
```

5 min

Solution

```
0 references
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");

    //Définir ma source de données
    var malist = ListAlbumsData.ListAlbums;

    //Création de la requête
    var toShow = from album in malist
                 select $"Album n°{album.AlbumId} : {album.Title}";

    //Appel de la requête
    foreach (var albumString in toShow)
    {
        //Affichage des résultats
        Console.WriteLine(albumString);
    }
}
```

Let

On peut créer un alias, pour des raisons de visibilité

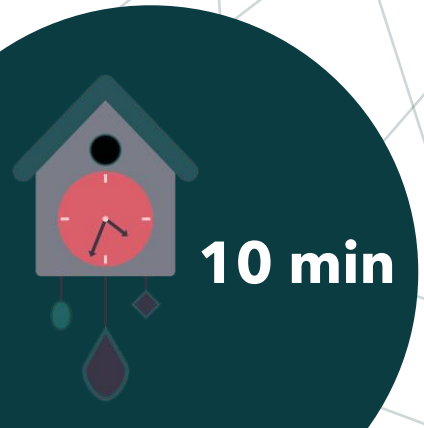
Dans cet exemple, le résultat sera le même :

```
//Création de la requête
var toShow = from album in malist
             let stringToReturn = $"Album n°{album.AlbumId} : {album.Title}"
                select stringToReturn;
```

Where

- **Where** est la condition qui nous permets de récupérer les éléments d'une liste qui remplissent certaines conditions.
- Exercice : Faire une recherche : pour un string donné par l'utilisateur, afficher les albums dont le nom contiens ce string.

```
Quel est votre recherche ?
out
Album n°1 : For Those About To Rock We Salute You
Album n°11 : Out Of Exile
Album n°68 : Outbreak
Album n°130 : In Through The Out Door
Album n°187 : Out Of Time
Album n°329 : South American Getaway
```




```
string recherche = Console.ReadLine();
```

```
bool containSearch = titreAlbum.Contains(recherche, StringComparison.InvariantCultureIgnoreCase);
```

Solution :

```
static void Main(string[] args)
{
    //Définir ma source de données
    var malist = ListAlbumsData.ListAlbums;

    Console.WriteLine("Quel est votre recherche ?");
    string recherche = Console.ReadLine();

    //Création de la requête
    var toShow = from album in malist
                 let stringToReturn = $"Album n°{album.AlbumId} : {album.Title}"
                 where album.Title.Contains(recherche, StringComparison.InvariantCultureIgnoreCase)
                 select stringToReturn;

    //Appel de la requête
    foreach (var albumString in toShow)
    {
        //Affichage des résultats
        Console.WriteLine(albumString);
    }
}
```

OrderBy, ThenBy

- **OrderBy** est la condition qui nous permets de trier les éléments d'une liste.
- **ThenBy** est la condition qui nous permets de trier les éléments d'une liste après avoir déjà trier la liste par un orderby (on utilise thenby dans la syntaxe méthode.
- Exercice : Ordonner les résultats de notre recherche d'abord par le titre (ascending), puis par l'id (descending)

```
Quel est votre recherche ?
out
Album n°1 : For Those About To Rock We Salute You
Album n°130 : In Through The Out Door
Album n°11 : Out Of Exile
Album n°187 : Out Of Time
Album n°68 : Outbreak
Album n°329 : South American Getaway
```

10 min

Après avoir réussi cet exercice facile, on refait notre requête avec la syntaxe méthode ?

SolutionS

Syntaxe requête

```
//Création de la requête
var toShow = from album in malist
             let stringToReturn = $"Album n°{album.AlbumId} : {album.Title}"
             where album.Title.Contains(recherche, StringComparison.InvariantCultureIgnoreCase)
             orderby album.Title ascending, album.AlbumId descending
             select stringToReturn;
```

Syntaxe méthodes

```
//requête avec la syntaxe méthodes
var requestMethod = malist.Where(alb => alb.Title.Contains(recherche, StringComparison.InvariantCultureIgnoreCase))
                  .OrderBy(alb => alb.Title)
                  .ThenByDescending(alb => alb.AlbumId)
                  .Select(album => $"Album n°{album.AlbumId} : {album.Title}");
```