C# : Linq

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

4 Méthodes de sélection

- `Distinct()`
- `First()`
- `FirstOrDefault()`
- `DefaultIfEmpty()`
- `Single()`
- `SingleOrDefault()`
- `ElementAt()`
- `Take()`
- `TakeWhile()`
- `Skip()`
- `SkipLast()`
- `SkipWhile()`
- `OfType()`

Plan

Plan

Linq

LINQ ?

- pour **Language-INtegrated Query** (ou en français **Requête intégrée au langage**).
- Composant **.NET**.
- Projet de recherche **Microsoft** avant qu'il soit intégré dans **.NET Framework** puis **.NET Core**.
- Inspiré par le langage **SQL**.
- Ajoutant de grandes capacités d'interrogation sur des données aux langages **.NET**.
- S'appliquant sur les listes, les objets, les fichiers **XML**, les entités, les bases de données relationnelles...

Linq

Étapes

Il faut

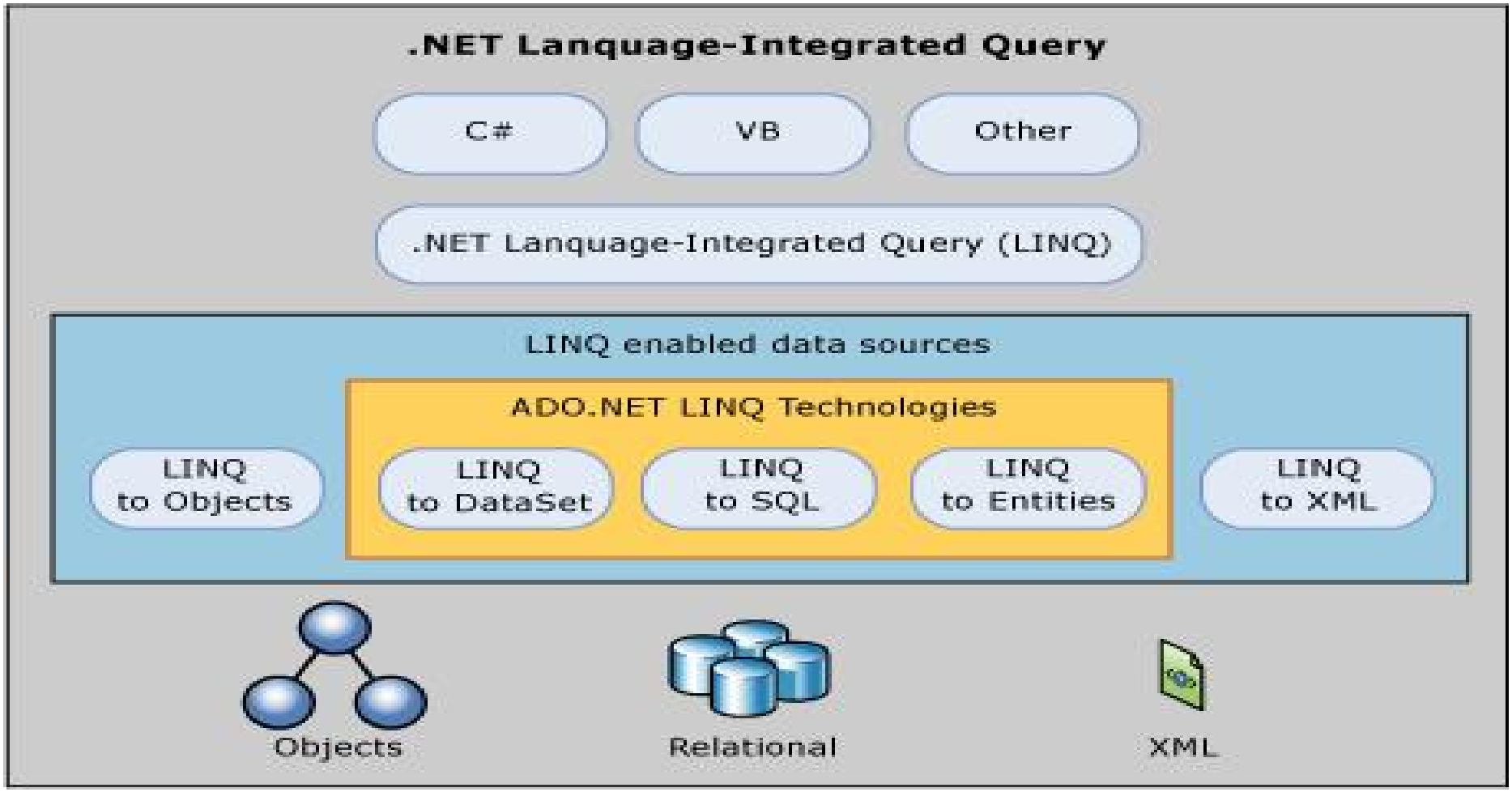
- spécifier une source de données (ou jeu de données)
- définir l'expression de la requête (Query Expression)
- exécuter la requête

Linq

Quelles sources de données sont autorisées ?

- Objets (Linq to Objects)
- ADO.NET
  - Linq to SQL
  - Linq to Entities
  - Linq to DataSet
- XML (Linq to XML)

Linq



Source : site **Microsoft**

Linq

Deux syntaxes pour **LINQ**

- Syntaxe de requête (Query Syntax)
- Syntaxe de méthode en utilisant les expressions Lambda (Method Syntax)

Linq

Deux syntaxes pour **LINQ**

- Syntaxe de requête (Query Syntax)
- Syntaxe de méthode en utilisant les expressions Lambda (Method Syntax)

Remarque

Il est possible de mixer les deux.

Linq

Exemple (Query Syntax)

```
// data source
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };

// query expression
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

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

Linq

Exemple (Query Syntax)

```
// data source
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };

// query expression
var numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

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

Affiche 2 4 6 8

Linq

Explication

- `from num in numbers` : `numQuery` contiendra toute valeur `num` appartenant au tableau `numbers`
- `where (num % 2) == 0` : définit la condition de sélection
- `select num` : précise ce que l'on veut sélectionner (ici on n'a que `num`

Linq

Remarque

- La requête est exécutée seulement dans le `foreach`
- L'ordre des clauses `from, where, select` est important
- `numQuery` est de type `IEnumerable<int>`, on peut donc faire :

Linq

Remarque

- La requête est exécutée seulement dans le `foreach`
- L'ordre des clauses `from, where, select` est important
- `numQuery` est de type `IEnumerable<int>`, on peut donc faire :

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
IEnumerable<int> numQuery =
    from num in numbers
    where (num % 2) == 0
    select num;
foreach (int num in numQuery)
{
    Console.Write("{0} ", num);
}
```

Linq

Exemple avec expression Lambda (Method Syntax)

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

IEnumerable<int> numQuery = numbers
       .Where(elt => elt % 2 == 0);

foreach (int num in numQuery)
{
    Console.Write("{0} ", num);
}
```

Linq

Exemple avec expression Lambda (Method Syntax)

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

IEnumerable<int> numQuery = numbers
       .Where(elt => elt % 2 == 0);

foreach (int num in numQuery)
{
    Console.Write("{0} ", num);
}
```

Affiche 2 4 6 8

Linq

Remarque

Contrairement aux `IList`, `List`..., `IEnumerable` ne peut pas être parcourue par `ForEach`, pas avant de le convertir en liste avec `ToList()`.

Il est aussi possible de définir plusieurs conditions

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num > 5 || num < 3)
    select num;

foreach (int num in numQuery)
{
    Console.Write("{0} ", num);
}
// affiche 2 6 8
```

Il est aussi possible de définir plusieurs conditions

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num > 5 || num < 3)
    select num;

foreach (int num in numQuery)
{
    Console.Write("{0} ", num);
}
// affiche 2 6 8
```

Avec les expressions Lambdas (Method Syntax)

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8 };
IEnumerable<int> numQuery = numbers
      .Where(elt => elt % 2 == 0 && (elt > 5 || elt < 3));

foreach (int num in numQuery)
{
    Console.Write("{0} ", num);
}
// affiche 2 6 8
```

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "
   abricot", "fraise", "melon" };
```

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "
   abricot", "fraise", "melon" };
```

Question

Écrire une requête **Linq** qui permet de retourner les longueurs impaires des chaînes définies dans le tableau `fruits`.

Linq

Pour générer un intervalle de valeurs, on utilise la méthode `Range()`

```
var numbers = Enumerable.Range(1, 8);

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

foreach (var num in numQuery)
{
    Console.Write("{0} ", num);
}
// affiche 2 4 6 8
```

Linq

Pour générer la même valeur plusieurs fois, on utilise la méthode `Repeat()`

```
var numbers = Enumerable.Repeat(2, 8);

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

foreach (var num in numQuery)
{
    Console.Write("{0} ", num);
}
// affiche 2 2 2 2 2 2 2 2
```

Linq

Exercice

Écire une requête **Linq** permettant de générer les nombres pairs inférieurs à 20.

Linq

Pour récupérer les éléments distincts d'une séquence

```
int[] notes = { 10, 18, 12, 10, 15, 10, 11 };

foreach (var note in notes.Distinct())
{
     Console.WriteLine(note);
}

Console.WriteLine(numQuery.First());
// affiche 10 18 12 15 11
```

Linq

Pour récupérer le premier élément d'une séquence

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num > 5 || num < 3)
    select num;

Console.WriteLine(numQuery.First());
// affiche 2
```

Linq

Deuxième syntaxe (Method syntax)

```
var numQuery = numbers.Where(elt => (elt % 2) == 0 && (elt > 5
   || elt < 3));
Console.WriteLine(numQuery.First());
// affiche 2
```

Linq

Deuxième syntaxe (Method syntax)

```
var numQuery = numbers.Where(elt => (elt % 2) == 0 && (elt > 5
   || elt < 3));
Console.WriteLine(numQuery.First());
// affiche 2
```

Ou encore

```
var resultat = numbers.First(elt => (elt % 2) == 0 && (elt > 5
   || elt < 3));
Console.WriteLine(resultat);
// affiche 2
```

Linq

Et si la sélection est vide, appeler `First()` lève une exception

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num % 5) == 0
    select num;

Console.WriteLine(numQuery.First());
```

Linq

Pour résoudre le problème précédent, on peut utiliser `FirstOrDefault` qui retourne soit le premier élément sélectionné, soit `null` pour les types `nullable`, soit 0 pour les nombres

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num % 5) == 0
    select num;

Console.WriteLine(numQuery.FirstOrDefault());
// affiche 0
```

Linq

Pour définir une valeur par défaut, on peut utiliser `DefaultIfEmpty()` qui retourne un tableau d'élément

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num % 5) == 0
    select num;

foreach(var elt in numQuery.DefaultIfEmpty(10))
{
    Console.WriteLine(elt);
}
// affiche 10
```

Linq

Si on a la certitude que notre sélection contient un seul élément, on peut utiliser `Single()`

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num % 3) == 0
    select num;

Console.WriteLine(numQuery.Single());
// affiche 6
```

Linq

Si la sélection est vide ou contient plusieurs éléments, `Single()` **lève une exception**

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num % 5) == 0
    select num;

Console.WriteLine(numQuery.Single());
```

Linq

Pour résoudre le problème de sélection vide, on peut utiliser `SingleOrDefault()`

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

var numQuery =
    from num in numbers
    where (num % 2) == 0 && (num % 5) == 0
    select num;

Console.WriteLine(numQuery.SingleOrDefault());
// affiche 0
```

Linq

Autres méthodes

- `Last()` et `LastOrDefault()`
- `ElementAt()` et `ElementAtOrDefault()`
- `Take()`, `TakeLast()` (non disponible pour **.NET Framework**) et `TakeWhile()`
- `Skip()`, `SkipLast()` (non disponible pour **.NET Framework**) et `SkipWhile()`
- ...

Linq

Pour indiquer l'indice d'élément à récupérer, on utilise `ElementAt()`

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

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

Console.WriteLine(numQuery.ElementAt(3));
// affiche 8
```

Linq

Pour indiquer le nombre d'élément à garder de la sélection, on peut utiliser `Take()`

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

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

foreach(var elt in numQuery.Take(3))
{
    Console.WriteLine(elt);
}
// affiche 2 4 6
```

Linq

Les méthodes précédentes peuvent être enchainées s'il ne s'agit pas de méthode de terminaison

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

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

Console.WriteLine(numQuery.Take(3).ElementAt(2));
// affiche 6
```

Linq

`TakeWhile()` **est utilisée pour retourner des éléments de la séquence tant qu'une condition spécifiée est vraie, puis arrête dès qu'un élément ne satisfait pas la condition**

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

var numQuery = numbers
    .Where(num => (num % 2) == 0)
    .TakeWhile(num => num < 5);

foreach (var elt in numQuery)
{
    Console.Write($"{elt} ");
}
// affiche 2 4
```

Linq

Pour indiquer le nombre d'élément à ignorer de la sélection, on peut utiliser `Skip()`

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

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

foreach(var elt in numQuery.Skip(2))
{
    Console.WriteLine(elt);
}
// affiche 6 8
```

Linq

Pour indiquer d'éléments à ignorer en commençant par la fin, on peut utiliser `SkipLast()`

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

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

foreach(var elt in numQuery.SkipLast(2))
{
    Console.WriteLine(elt);
}
// affiche 2 4
```

Linq

Pour indiquer une condition sur les éléments à ignorer, on peut utiliser `SkipWhile()`

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

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

foreach(var elt in numQuery.SkipWhile(elt => elt < 3))
{
    Console.WriteLine(elt);
}
// affiche 4 6 8
```

Linq

`SkipWhile()` **teste si la condition est vraie pour chaque élément de la source. Lorsque la condition est évaluée à** `false` **pour un élément, alors cet élément et les éléments restants de la source sont retournés sans retester la condition.**

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

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

foreach(var elt in numQuery.SkipWhile(elt => elt == 4))
{
    Console.WriteLine(elt);
}
// affiche 2 4 6 8
```

Linq

Pour sélectionner les éléments d'une séquence d'un type donné, on utilise `OfType()`

```
ArrayList nombres = [1, "deux", "3", '4', 5, "six"];

foreach (var elt in nombres.OfType<string>())
{
    Console.WriteLine(elt);
}
// affiche deux 3 six
```

Linq

Pour calculer la somme, on peut utiliser la méthode d'agrégation `Sum()`

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

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

Console.WriteLine(numQuery.Sum());
// affiche 20
```

Linq

Exercice

En utilisant `Sum`, écrivez une requête qui affiche la somme des doubles des éléments du tableau `numbers`

Linq

Première solution

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

var sommeDouble = numbers
     .Where(n => n % 2 == 0)
     .Select(n => n * 2)
     .Sum();

Console.WriteLine(sommeDouble);
// affiche 40
```

Linq

Deuxième solution : la méthode d'agrégation `Sum()` peut accepter comme paramètre une expression Lambda de transformation

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

var sommeDouble = numbers
     .Where(n => n % 2 == 0)
     .Sum(n => n * 2);

Console.WriteLine(sommeDouble);
// affiche 40
```

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "abricot", "fraise", "
   melon" };
```

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "abricot", "fraise", "
   melon" };
```


Question

Écrire une requête **Linq** qui permet de calculer le nombre total des caractères du tableau `fruits`.

Linq

Autres méthodes d'agrégation

- `Average()`
- `Max()`
- `Min()`
- `Count()`
- ...

Linq

Autres méthodes d'agrégation

- `Average()`
- `Max()`
- `Min()`
- `Count()`
- ...

Remarque

Toutes ces méthodes d'agrégation peuvent prendre comme paramètre une Expression Lambda de type **Predicate**.

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "abricot", "fraise", "
   melon" };
```

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "abricot", "fraise", "
   melon" };
```

Question

Écrire une requête **Linq** qui permet de calculer le nombre total de caractères des éléments du tableau `fruits` ayant un nombre pair de caractères.

Linq

Pour calculer le nombre d'éléments pairs d'un tableau

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

int pairs = numbers.Aggregate(0, (total, next) =>
                                    next % 2 == 0 ? total + 1 : total);

Console.WriteLine(pairs);
// affiche 4
```

Linq

Exercice

Utilisez `Aggregate` pour recalculer le nombre total des caractères du tableau `fruits`.

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "abricot", "fraise", "
   melon" };
```

Linq

Exercice : considérons le tableau suivant

```
string[] fruits = { "pomme", "banane", "mangue", "abricot", "fraise", "
   melon" };
```

Question

Écrire une requête **Linq** qui permet de retourner la chaîne la plus longue du tableau `fruits`

Linq

Méthodes de quantification

- `All()` : permet de vérifier si tous les éléments d'une source de données satisfont une condition.
- `Any()` : permet de vérifier s'il existe un élément d'une source de données qui satisfait une condition.
- `Contains()` : permet de vérifier si une source de données contient une valeur donnée.

Linq

Considérons la source de données suivante

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

Linq

Considérons la source de données suivante

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

Pour vérifier si un élément est à la fois divisible par 2 et par 3 (Query Syntax)

```
var anyQuery = (from number in numbers
                select number)
                .Any(elt => elt % 2 == 0 && elt % 3
                   == 0);

Console.WriteLine(anyQuery);
// affiche True
```

Linq

Pour vérifier si tous les éléments sont à la fois divisibles par 2 et par 3 (Query Syntax)

```
var allQuery = (from number in numbers
                select number)
                .All(elt => elt % 2 == 0 && elt % 3
                   == 0);

Console.WriteLine(allQuery);
// affiche False
```

Linq

Pour vérifier si un élément est dans l'ensemble (Query Syntax)

```
var containsQuery = (from number in numbers
                     select number)
                     .Contains(5);

Console.WriteLine(containsQuery);
// affiche True
```

Linq

Considérons les deux sources de données suivantes

```
int[] premiers = { 2, 3, 5, 7, 11, 13, 17, 19 };
int[] impairs = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
```

Linq

Considérons les deux sources de données suivantes

```
int[] premiers = { 2, 3, 5, 7, 11, 13, 17, 19 };
int[] impairs = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
```

Pour faire l'union de deux sources de données

```
var unionQuery = premiers.Union(impairs);

foreach(var elt in unionQuery)
{
    Console.Write("{0} ", elt);
}
```

Linq

Considérons les deux sources de données suivantes

```
int[] premiers = { 2, 3, 5, 7, 11, 13, 17, 19 };
int[] impairs = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
```

Pour faire l'union de deux sources de données

```
var unionQuery = premiers.Union(impairs);

foreach(var elt in unionQuery)
{
    Console.Write("{0} ", elt);
}
```

Affiche 2 3 5 7 11 13 17 19 1 9 15

Linq

Nous pouvons aussi faire une union sélective

```
var unionQuery = (from premier in premiers
                  where premier < 10
                  select premier)
                  .Union(from impair in impairs
                         where impair < 10
                         select impair);

foreach(var elt in unionQuery)
{
    Console.Write("{0} ", elt);
}
```

Linq

Nous pouvons aussi faire une union sélective

```
var unionQuery = (from premier in premiers
                  where premier < 10
                  select premier)
                  .Union(from impair in impairs
                         where impair < 10
                         select impair);

foreach(var elt in unionQuery)
{
    Console.Write("{0} ", elt);
}
```

Affiche 2 3 5 7 1 9

Linq

Deuxième syntaxe (Method Syntax)

```
var unionMethod = premiers.Where(elt => elt < 10)
                          .Union(impairs.Where(e => e < 10));

foreach(var elt in unionMethod)
{
    Console.Write("{0} ", elt);
}
```

Linq

Deuxième syntaxe (Method Syntax)

```
var unionMethod = premiers.Where(elt => elt < 10)
                          .Union(impairs.Where(e => e < 10));

foreach(var elt in unionMethod)
{
    Console.Write("{0} ", elt);
}
```


Affiche 2 3 5 7 1 9

Linq

Pour faire l'union de deux sources de données tout en gardant les valeurs en commun, on utilise `Concat` (Query Syntax)

```
var unionQuery = (from premier in premiers
                  where premier < 10
                  select premier)
                  .Concat(from impair in impairs
                          where impair < 10
                          select impair);

foreach(var elt in unionQuery)
{
    Console.Write("{0} ", elt);
}
```

Linq

Pour faire l'union de deux sources de données tout en gardant les valeurs en commun, on utilise `Concat` (Query Syntax)

```
var unionQuery = (from premier in premiers
                  where premier < 10
                  select premier)
                  .Concat(from impair in impairs
                          where impair < 10
                          select impair);

foreach(var elt in unionQuery)
{
    Console.Write("{0} ", elt);
}
```

Affiche 2 3 5 7 1 3 5 7 9

Linq

Pour faire la concaténation de deux sources de données (Method Syntax)

```
var unionMethod = premiers.Where(elt => elt < 10)
                          .Concat(impairs.Where(e => e < 10));

foreach(var elt in unionMethod)
{
    Console.Write("{0} ", elt);
}
```

Linq

Pour faire la concaténation de deux sources de données (Method Syntax)

```
var unionMethod = premiers.Where(elt => elt < 10)
                          .Concat(impairs.Where(e => e < 10));

foreach(var elt in unionMethod)
{
    Console.Write("{0} ", elt);
}
```


Affiche 2 3 5 7 1 3 5 7 9

Linq

Pour faire l'intersection de deux sources de données (Query Syntax)

```
var interQuery = (from premier in premiers
                  where premier < 10
                  select premier)
                  .Intersect(from impair in impairs
                             where impair < 10
                             select impair);

foreach(var elt in interQuery)
{
    Console.Write("{0} ", elt);
}
```

Linq

Pour faire l'intersection de deux sources de données (Query Syntax)

```
var interQuery = (from premier in premiers
                  where premier < 10
                  select premier)
                  .Intersect(from impair in impairs
                             where impair < 10
                             select impair);

foreach(var elt in interQuery)
{
    Console.Write("{0} ", elt);
}
```

Affiche 3 5 7

Linq

Pour faire l'union de deux sources de données (Method Syntax)

```
var interMethod = premiers.Where(elt => elt < 10)
                          .Intersect(impairs.Where(elt =>
                              elt < 10));

foreach(var elt in interMethod)
{
    Console.Write("{0} ", elt);
}
```

Linq

Pour faire l'union de deux sources de données (Method Syntax)

```
var interMethod = premiers.Where(elt => elt < 10)
                          .Intersect(impairs.Where(elt =>
                              elt < 10));

foreach(var elt in interMethod)
{
    Console.Write("{0} ", elt);
}
```

Affiche 3 5 7

Linq

Remarque

L'ordre des sources de données n'a pas d'importance ni pour l'union ni pour l'intersection (Le résultat est le même)

Linq

Exercice

Vérifier que l'intersection de `premiers` et `impairs` contient au moins un nombre compris entre 6 et 10.

Linq

Pour retourner les éléments de la première collection qui ne sont pas dans la deuxième (Query Syntax)

```
var exceptQuery = (from premier in premiers
                   where premier < 10
                   select premier)
                   .Except(from impair in impairs
                           where impair < 10
                           select impair);

foreach(var elt in exceptQuery)
{
    Console.Write("{0} ", elt);
}
```

Linq

Pour retourner les éléments de la première collection qui ne sont pas dans la deuxième (Query Syntax)

```
var exceptQuery = (from premier in premiers
                   where premier < 10
                   select premier)
                   .Except(from impair in impairs
                           where impair < 10
                           select impair);

foreach(var elt in exceptQuery)
{
    Console.Write("{0} ", elt);
}
```

Affiche 2

Linq

Deuxième syntaxe (Method Syntax)

```
var exceptMethod = premiers.Where(elt => elt < 10)
                           .Except(impairs.Where(elt
                              => elt < 10));

foreach(var elt in exceptMethod)
{
    Console.Write("{0} ", elt);
}
```

Linq

Deuxième syntaxe (Method Syntax)

```
var exceptMethod = premiers.Where(elt => elt < 10)
                           .Except(impairs.Where(elt
                              => elt < 10));

foreach(var elt in exceptMethod)
{
    Console.Write("{0} ", elt);
}
```

Affiche 2

Linq

En inversant l'ordre des ensembles, on obtient un résultat différent

```
var exceptMethod = impairs.Where(elt => elt < 10)
                          .Except(premiers.Where(elt
                              => elt < 10));

foreach(var elt in exceptMethod)
{
    Console.Write("{0} ", elt);
}
```

Linq

En inversant l'ordre des ensembles, on obtient un résultat différent

```
var exceptMethod = impairs.Where(elt => elt < 10)
                          .Except(premiers.Where(elt
                              => elt < 10));

foreach(var elt in exceptMethod)
{
    Console.Write("{0} ", elt);
}
```

Affiche 1 9

Linq

Pour fusionner deux séquences, on utilise `Zip` (Method Syntax)

```
int[] premiers = { 2, 3, 5, 7, 11, 13, 17, 19 };
int[] impairs = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };

var fusionQuery = premiers.Zip(impairs, (first, second)
   => first + " " + second);

foreach (var elt in fusionQuery)
{
    Console.Write($" { elt },");
}
```

Linq

Pour fusionner deux séquences, on utilise `Zip` (Method Syntax)

```
int[] premiers = { 2, 3, 5, 7, 11, 13, 17, 19 };
int[] impairs = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };

var fusionQuery = premiers.Zip(impairs, (first, second)
   => first + " " + second);

foreach (var elt in fusionQuery)
{
    Console.Write($" { elt },");
}
```

Affiche 2 1, 3 3, 5 5, 7 7, 11 9, 13 11, 17 13, 19 15,

Linq

Exercice : considérons les tableaux suivants

```
var D = Enumerable.Range(1, 5);
var E = Enumerable.Range(3, 7);
```

Linq

Exercice : considérons les tableaux suivants

```
var D = Enumerable.Range(1, 5);
var E = Enumerable.Range(3, 7);
```

Question

Écrire une requête **Linq** qui permet de Fusionner les éléments de D et E (résultat de la fusion = { 13, 24, 35, 46, 57 }) et de garder seulement les éléments pairs.

Linq

Considérons les trois ensembles suivants

```
List<int> ensemble1 = [2, 3, 8, 5];
List<int> ensemble2 = [7, 2, 9, 3];
List<int> ensemble3 = [1, 2, 4, 5];
```

Linq

Considérons les trois ensembles suivants

```
List<int> ensemble1 = [2, 3, 8, 5];
List<int> ensemble2 = [7, 2, 9, 3];
List<int> ensemble3 = [1, 2, 4, 5];
```


Exercice

Utilisez **Linq** pour

1. vérifier que l'intersection entre `ensemble3` et `ensemble2` est un sous-ensemble de `ensemble1`
2. afficher les valeurs présentes dans `ensemble1` **ou** `ensemble2` mais pas dans `ensemble3`
3. afficher les valeurs présentes dans `ensemble1` **et** `ensemble2` mais pas dans `ensemble3`
4. afficher les valeurs présentes dans `ensemble1` **ou** `ensemble2` mais pas dans les deux ensembles à la fois

Linq

Correction

```
// Question 1
var resultat1 = (ensemble3.Intersect(ensemble2)).All(elt => ensemble3.Contains(elt));
Console.WriteLine(resultat1);
// affiche True

// Question 2
(ensemble1.Union(ensemble2)).Except(ensemble3)
.ToList()
.ForEach(Console.Write);
// affiche 3 8 7 9

// Question 3
(ensemble1.Intersect(ensemble2)).Except(ensemble3)
.ToList()
.ForEach(Console.Write);
// affiche 3

// Question 4
(ensemble1.Union(ensemble2)).Except(ensemble1.Intersect(ensemble2))
.ToList()
.ForEach(Console.Write);
// affiche 8 5 7 9
```

Linq

Linq to Objects : Objects ?

- Il s'agit de tout objet **C#** itérable (ou énumérable)
  - tableau
  - collection
  - chaîne de caractères
  - ...
- sans utiliser une API intermédiaire comme **LINQ to SQL** ou **LINQ to XML**.

Linq

Étant donnée la classe `Personne` suivante

```
internal class Personne
{
    public Personne(int num, string nom, string prenom, int budget)
    {
        Nom = nom;
        Prenom = prenom;
        Num = num;
        Budget = budget;
    }

    public int Num { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
    public int Budget { get; set; }

    public override string ToString()
    {
        return $"{Num} + {Nom} + {Prenom} + {Budget}";
    }
}
```

Linq

Créons une liste à partir de cette classe `Personne`

```
List<Personne> personnes = [
    new(1, "wick", "john", 500),
    new(2, "abruzzi", "john", 700),
    new(3, "dalton", "jack", 300),
    new(4, "white", "mike", 800),
];
```

Linq

Exercice

Afficher les personnes ayant un `Budget` ¿ 500.

Linq

Solution avec Query syntax

```
IEnumerable <Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Solution avec Query syntax

```
IEnumerable <Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
2 abruzzi john 700
4 white mike 800

Linq

Solution avec Method syntax

```
var persoQuery = personnes
     .Where(elt => elt.Budget > 500);

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Solution avec Method syntax

```
var persoQuery = personnes
     .Where(elt => elt.Budget > 500);

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
2 abruzzi john 700
4 white mike 800

Linq

Exercice

Afficher les noms des personnes ayant un `Budget` ¿ 500.

Linq

Si on veut sélectionner seulement quelques attributs (et les renommer)

```
var persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select perso.Nom};

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Si on veut sélectionner seulement quelques attributs (et les renommer)

```
var persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select perso.Nom};

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
{ Name = abruzzi, Number = 2 }
{ Name = white, Number = 4 }

Linq

Autre solution avec les expressions Lambda

```
var persoQuery = personnes
    .Where( elt => elt.Budget > 500 )
    .Select( elt => elt.Nom )

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Autre solution avec les expressions Lambda

```
var persoQuery = personnes
    .Where( elt => elt.Budget > 500 )
    .Select( elt => elt.Nom )

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
{ Name = abruzzi, Number = 2 }
{ Name = white, Number = 4 }

Linq

Pour sélectionner quelques propriétés (mais pas tous)

```
var persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select new {
        perso.Nom,
        perso.Num
    };

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Pour sélectionner quelques propriétés (mais pas tous)

```
var persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select new {
        perso.Nom,
        perso.Num
    };

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
{ Nom = abruzzi, Num = 2 }
{ Nom = white, Num = 4 }

Linq

Autre solution avec les expressions Lambda

```
var persoQuery = personnes
    .Where( elt => elt.Budget > 500 )
    .Select( elt => new {
        elt.Nom,
        elt.Num
    }
);

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Autre solution avec les expressions Lambda

```
var persoQuery = personnes
    .Where( elt => elt.Budget > 500 )
    .Select( elt => new {
        elt.Nom,
        elt.Num
    }
);

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
{ Nom = abruzzi, Num = 2 }
{ Nom = white, Num = 4 }

Linq

Pour sélectionner quelques attributs (et les renommer)

```
var persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select new {
        Name = perso.Nom,
        Number = perso.Num
    };

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Pour sélectionner quelques attributs (et les renommer)

```
var persoQuery =
    from perso in personnes
    where ( perso.Budget > 500 )
    select new {
        Name = perso.Nom,
        Number = perso.Num
    };

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
{ Name = abruzzi, Number = 2 }
{ Name = white, Number = 4 }

Linq

Autre solution avec les expressions Lambda

```
var persoQuery = personnes
    .Where( elt => elt.Budget > 500 )
    .Select( elt => new {
        Name = elt.Nom,
        Number = elt.Num
    }
);

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Autre solution avec les expressions Lambda

```
var persoQuery = personnes
    .Where( elt => elt.Budget > 500 )
    .Select( elt => new {
        Name = elt.Nom,
        Number = elt.Num
    }
);

foreach ( var perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
{ Name = abruzzi, Number = 2 }
{ Name = white, Number = 4 }

Linq

Autres clauses de requête **LINQ**

- `orderby` : permet de trier dans l'ordre croissant ou décroissant.
- `group ... by ... into` : permet de retourner une séquence composée de clé (correspondant au critère du groupement) et valeur (correspondant à l'objet)
- `join ... on ... equals` : permet de faire des jointures entre deux énumérables
- ...

Linq

Exemple avec `orderby`

```
IEnumerable<Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget >= 500 )
    orderby perso.Prenom ascending // ou descending
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Exemple avec `orderby`

```
IEnumerable<Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget >= 500 )
    orderby perso.Prenom ascending // ou descending
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
1 wick john 500
2 abruzzi john 700

4 white mike 800

Linq

Solution avec les expressions Lambda

```
IEnumerable<Personne> persoQuery = personnes
    .Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => perso.Prenom );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Solution avec les expressions Lambda

```
IEnumerable<Personne> persoQuery = personnes
    .Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => perso.Prenom );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
1 wick john 500
2 abruzzi john 700

4 white mike 800

Linq

Solution avec les expressions Lambda

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderByDescending( perso => perso.Prenom );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Solution avec les expressions Lambda

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderByDescending( perso => perso.Prenom );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
4 white mike 800
1 wick john 500

2 abruzzi john 700

Linq

On peut aussi ordonner selon plusieurs colonnes (le deuxième étant `ascending`)

```
IEnumerable<Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget >= 500 )
    orderby perso.Prenom ascending, perso.Budget ascending
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Ou avec Method syntax

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => (perso.Prenom, perso.Budget) )

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Ou avec Method syntax

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => (perso.Prenom, perso.Budget) )

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Ou

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => perso.Prenom )
    .ThenBy( perso => perso.Budget );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

On peut aussi ordonner selon plusieurs colonnes (le deuxième étant `descending`)

```
IEnumerable<Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget >= 500 )
    orderby perso.Prenom ascending, perso.Budget descending
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

On peut aussi ordonner selon plusieurs colonnes (le deuxième étant `descending`)

```
IEnumerable<Personne> persoQuery =
    from perso in personnes
    where ( perso.Budget >= 500 )
    orderby perso.Prenom ascending, perso.Budget descending
    select perso;

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
2 abruzzi john 700
1 wick john 500

4 white mike 800

Linq

Ou avec Method syntax

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => perso.Prenom )
    .ThenByDescending( perso => perso.Budget );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```

Linq

Ou avec Method syntax

```
IEnumerable<Personne> persoQuery =
    personnes.Where( perso => perso.Budget >= 500 )
    .OrderBy( perso => perso.Prenom )
    .ThenByDescending( perso => perso.Budget );

foreach ( Personne perso in persoQuery )
{
    Console.WriteLine(perso);
}
```


ça affiche :
2 abruzzi john 700
1 wick john 500

4 white mike 800

Linq

Exemple avec `group ... by ... into`

```
var persoQuery =
    from perso in personnes
    group perso by perso.Prenom into initial
    select initial;
```

Linq

Exemple avec `group ... by ... into`

```
var persoQuery =
    from perso in personnes
    group perso by perso.Prenom into initial
    select initial;
```

Explication

Cette requête retourne un regroupement `<string, Personne>` :

- la clé est de type chaîne de caractère correspondant `perso.Prenom`
- la valeur est de type `Personne` correspondant à l'objet ayant la clé

Linq

Exemple avec `group ... by ... into`

```
foreach (IGrouping<string, Personne> elt in persoQuery)
{
    Console.WriteLine(elt.Key);
    foreach (var perso in elt)
    {
        Console.WriteLine($"  {perso.Prenom}, {perso.Nom}");
    }
}
```

Linq

Exemple avec `group ... by ... into`

```
foreach (IGrouping<string, Personne> elt in persoQuery)
{
    Console.WriteLine(elt.Key);
    foreach (var perso in elt)
    {
        Console.WriteLine($"  {perso.Prenom}, {perso.Nom}");
    }
}
```


ça affiche :

```
john
  john, wick
  john, abruzzi
jack
  jack, dalton
mike
  mike, white
```

Linq

Solution avec les expressions Lambda

```
var persoQuery = personnes
    .GroupBy( perso => perso.Prenom );

foreach (IGrouping<string, Personne> elt in persoQuery)
{
    Console.WriteLine(elt.Key);
    foreach ( var perso in elt )
    {
        Console.WriteLine($"  {perso.Prenom}, {perso.Nom}");
    }
}
```

Linq

Considérons la deuxième liste suivante

```
List<Personne> sportifs = [
    new(1, "wick", "john", 500),
    new(4, "white", "mike", 800),
    new(5, "wolf", "nicolas", 700),
    new(6, "hamilton", "bill", 300),
];
```

Linq

Considérons la deuxième liste suivante

```
List<Personne> sportifs = [
    new(1, "wick", "john", 500),
    new(4, "white", "mike", 800),
    new(5, "wolf", "nicolas", 700),
    new(6, "hamilton", "bill", 300),
];
```

Si on veut afficher la liste des personnes qui sont dans la liste `sportifs` ?

Linq

Considérons la deuxième liste suivante

```
List<Personne> sportifs = [
    new(1, "wick", "john", 500),
    new(4, "white", "mike", 800),
    new(5, "wolf", "nicolas", 700),
    new(6, "hamilton", "bill", 300),
];
```

Si on veut afficher la liste des personnes qui sont dans la liste `sportifs` ?

Solution avec les jointures

Linq

Solution avec `join ... on ... equals`

```
// join permet de faire la jointure
var persoSportifQuery =
    from perso in personnes
    join sportif in sportifs
    on perso.Num equals sportif.Num
    select new {
        Name = perso.Nom,
        Sport = sportif.Prenom
    };

// select new permet de construire un nouvel objet

foreach ( var elt in persoSportifQuery )
{
  Console.WriteLine("{0} {1}", elt.Name, elt.Sport);
}
```

Linq

Solution avec les expressions Lambda

```
var persoSportifQuery = personnes.Join(
    sportifs,
    perso => perso.Num,
    sportif => sportif.Num,
    (perso, sportif) => new {
        Name = perso.Nom,
        Sport = sportif.Prenom
    }
);

foreach ( var elt in persoSportifQuery )
{
  Console.WriteLine("{0} {1}", elt.Name, elt.Sport);
}
```

Linq

Exercice

En utilisant `Intersect`, écrivez une requête qui sélectionne les éléments qui sont à la fois dans `personnes` et `sportifs`

Linq

En respectant les règles précédentes, on pense à écrire la requête suivante

```
var interQuery = (from perso in personnes
                  select perso)
                  .Intersect(
                   from sportif in sportifs
                   select sportif);

foreach ( Personne perso in interQuery )
{
    Console.WriteLine(perso);
}
```

Linq

En respectant les règles précédentes, on pense à écrire la requête suivante

```
var interQuery = (from perso in personnes
                  select perso)
                  .Intersect(
                   from sportif in sportifs
                   select sportif);

foreach ( Personne perso in interQuery )
{
    Console.WriteLine(perso);
}
```

Aucun élément sélectionné alors que l'intersection est non-vide.

Linq

En respectant les règles précédentes, on pense à écrire la requête suivante

```
var interQuery = (from perso in personnes
                  select perso)
                  .Intersect(
                   from sportif in sportifs
                   select sportif);

foreach ( Personne perso in interQuery )
{
    Console.WriteLine(perso);
}
```

Aucun élément sélectionné alors que l'intersection est non-vide.

Avec `Intersect`, on compare les objets et non pas les valeurs.

En respectant les règles précédentes, on pense à écrire la requête suivante

```
var interQuery = (from perso in personnes
                  select new {
                      perso.Num,
                      perso.Nom,
                      perso.Prenom,
                      perso.Budget
                  })
                  .Intersect(
                   from sportif in sportifs
                   select new {
                       sportif.Num,
                       sportif.Nom,
                       sportif.Prenom,
                       sportif.Budget
                   });

foreach ( var perso in interQuery )
{
    Console.WriteLine(perso);
}
```

En respectant les règles précédentes, on pense à écrire la requête suivante

```
var interQuery = (from perso in personnes
                  select new {
                      perso.Num,
                      perso.Nom,
                      perso.Prenom,
                      perso.Budget
                  })
                  .Intersect(
                   from sportif in sportifs
                   select new {
                       sportif.Num,
                       sportif.Nom,
                       sportif.Prenom,
                       sportif.Budget
                   });

foreach ( var perso in interQuery )
{
    Console.WriteLine(perso);
}
```

ça affiche :
{ Num = 1, Nom = wick, Prenom = john, Budget = 500 }

{ Num = 4, Nom = white, Prenom = mike, Budget = 800 }

Linq

Étant donnée la classe `Adresse` suivante

```
internal class Adresse
{
    public string Ville { get; set; }
    public string[] CodesPostaux { get; set; } = { };

    public Adresse(string ville, string[] codesPostaux)
    {
        Ville = ville;
        CodesPostaux = codesPostaux;
    }
}
```

Linq

Et le tableau d'adresses suivant

```
Adresse[] adresses = [
    new Adresse("Marseille", ["13002", "13002"]),
    new Adresse("Paris", ["75014", "75018", "75012"]),
    new Adresse("Lyon", ["69008", "69000"])
];
```

Linq

Et le tableau d'adresses suivant

```
Adresse[] adresses = [
    new Adresse("Marseille", ["13002", "13002"]),
    new Adresse("Paris", ["75014", "75018", "75012"]),
    new Adresse("Lyon", ["69008", "69000"])
];
```

Pour fusionner tous les codes postaux dans un seul tableau, on utilise `SelectMany`

```
adresses
    .SelectMany(elt => elt.CodesPostaux)
    .ToList()
    .ForEach(Console.WriteLine);
```

Linq

Pour sélectionner l'adresse ayant le plus grand nombre de codes postaux, on utilise `MaxBy`

```
var result = adresses
     .MaxBy(elt => elt.CodesPostaux.Length);

Console.WriteLine(result.Ville);
// affiche Paris
```

Linq

Il existe également

- `MinBy()`
- `DistinctBy()`
- ...

Linq

ADO.NET : inconvénients ?

- Dans un environnement **.NET**, le développeur peut avoir besoin de persister ses données.
- Avec **ADO.NET**, il écrit des requêtes **SQL** sous forme de chaîne de caractères dans des classes **DAO**.
- Cette chaîne, illisible pour le compilateur, peut :
  - contenir des erreurs syntaxiques
  - référencer des colonnes inexistantes

Linq

Linq to SQL : pourquoi ?

- **Linq** : une seule syntaxe quelle que soit la source de données (qui implémente l'interface `IEnumerable`).
- Pas de recours aux requêtes **SQL** sous forme de chaîne de caractères dans des classes **DAO**.
- Le développeur consulte toujours les données comme une collection `IEnumerable`.
- Une prise en charge complète (de **IntelliSense**) fournie pour l'écriture de requêtes sur ces collections.

Linq

Linq to SQL : comment ?

Trois étapes :

- Créer une base de données **SQL Server** (pas besoin de l'installer) ou inclure une qui existe déjà.
- Ajouter le fichier de classe **LINQ to SQL**.
- Instancier cette classe et interroger la base de données avec **Linq to SQL**.

Linq

Créer une base de données **SQL Server**

- Clic droit sur le nom du projet dans l'`Explorateur de solutions`
- Aller dans `Ajouter > Nouvel élément`
- Dans `Élément Visual C#`, cliquer sur `Données` et choisir `Base de données basée sur les services`
- Saisir `MaBase` (par exemple) dans `Nom :` puis cliquer sur `Ajouter`

Linq

Créer une table

- Dans l'`Explorateur de solutions`, faire double clic sur `MaBase.mdf` (un `Explorateur de serveurs` qui apparaît)
- Dans `Explorateur de serveurs`, faire un clic droit sur l'option `Tables` du menu `MaBase.mdf`
- Choisir `Ajouter une nouvelle table`
- Dans l'onglet `Conception`, remplacer `[dbo].[Table]` par `[dbo].[Personne]`. Ainsi, on a nommé notre table `Personne`
- Par défaut, cette table a une colonne `id`. Ajouter les colonnes `nom`, `prénom` et `age`.

Linq

Créer une table

- Pour que la clé primaire soit **Auto-Increment**, aller dans l'onglet `Propriétés` (le panneau à droite), ouvrir `Spécifications du compteur` ensuite mettre à `True` la valeur de `(Est d'identité)` et vérifier que le compteur et le pas sont à 1.
- Pour exécuter le script, cliquer sur `Mettre à jour`
- Valider en cliquant sur `Mettre à jour la base de données`
- Pour vérifier que la base de données a bien été créée, cliquer à gauche sur `Explorateur de serveurs`, ensuite cliquer sur `Actualiser`, déplier le menu `Tables` où on trouvera la table `Personne`

Linq

Ajouter des tuples dans la table

- Faire un clic droit sur le nom de la table et choisir `Afficher les données de la table`
- Ajouter quelques tuples sans renseigner la clé primaire (qui est Auto-Increment)
- Aller dans l'`Explorateur de serveurs`, faire un clic droit sur `MaBase` et cliquer sur `Fermer la connexion`

Linq

Étape 2 : ajouter le fichier `Classes LINQ to SQL`

- Clic droit sur le nom du projet dans l'`Explorateur de solutions`
- Aller dans `Ajouter > Nouvel élément`
- Dans `Élément Visual C#`, cliquer sur `Données` et choisir `Classes` **LINQ to SQL** (voir slide suivante si `Classes LINQ to SQL` n'existe pas)
- Saisir un nom (par défaut, c'est `DataClasses1.dbml`) dans `Nom :` puis cliquer sur `Ajouter`

Linq

Si `Classes LINQ to SQL` n'existe pas

- Aller dans `Fichier > Nouveau > Projet`
- Aller vers le bas et cliquer sur `Ouvrir Visual Studio Installer`
- Dans la fenêtre qui s'affiche, aller dans l'onglet `Composants individuels` et chercher la rubrique `Outils de code`
- Cocher la case **Outils LINQ to SQL**
- Cliquer sur `Modifier`

Linq

Étape 2 : ajouter le fichier de classe de **LINQ to SQL**

- Ouvrir l'`Explorateur de serveurs` et déplier l'onglet `MaBase`
- Sélectionner puis glisser les tables à utiliser dans le projet (ici `Personne`) dans le panneau central
- Sauvegarder (**CTRL + s**)

Linq

Étape 2 : vérifier l'ajout du fichier contexte

- Dans l'`Explorateur de solutions`, vérifier qu'une section portant le nom `DataClasses1.dbml` a bien été ajoutée
- Déplier cette section, déplier le fichier `DataClasses1.designer.cs` et vérifier la présence de la classe `DataClasses1.DataContext`
- Étendre cette classe (dans l'`Explorateur de solutions`) et vérifier l'existence de méthodes `InsertPersonne(Personne)`, `DeletePersonne(Personne)` et `UpdatePersonne(Personne)`.

Linq

Étape 3 : interroger la base de données

- Instancier la classe `DataClasses1.DataContext`
- Interroger la base de données avec (LINQ to SQL)

Instancier la classe `DataClass1.DataContext`

```
DataClasses1DataContext db = new DataClasses1DataContext();
```

Instancier la classe `DataClass1.DataContext`

```
DataClasses1DataContext db = new DataClasses1DataContext();
```

Récupérer le contenu de la table `Personne`

```
var dbPersoQuery =
    from perso in db.Personne
    select perso;
```

Instancier la classe `DataClass1.DataContext`

```
DataClasses1DataContext db = new DataClasses1DataContext();
```

Récupérer le contenu de la table `Personne`

```
var dbPersoQuery =
    from perso in db.Personne
    select perso;
```

Afficher le contenu de la table `Personne`

```
foreach (var elt in dbPersoQuery)
{
    Console.WriteLine(elt.Id + " " +elt.nom + " " + elt.prenom + " " +
       elt.age);
}
```

Instancier la classe `DataClass1.DataContext`

```
DataClasses1DataContext db = new DataClasses1DataContext();
```

Récupérer le contenu de la table `Personne`

```
var dbPersoQuery =
    from perso in db.Personne
    select perso;
```

Afficher le contenu de la table `Personne`

```
foreach (var elt in dbPersoQuery)
{
    Console.WriteLine(elt.Id + " " +elt.nom + " " + elt.prenom + " " +
       elt.age);
}
```

Vous pouvez utiliser la méthode `Trim()` **pour supprimer les espaces.**

Linq

On peut aussi simplifier l'écriture précédente ainsi

```
using (DataClasses1DataContext db = new DataClasses1DataContext())
{
     var dbPersoQuery =
         from perso in db.Personne
         select perso;
     foreach (var elt in dbPersoQuery)
     {
          Console.WriteLine(elt.Id + " " + elt.nom + " " + elt.prenom +
              " " + elt.age);
     }
}
```

Linq

On peut aussi simplifier l'écriture précédente ainsi

```
using (DataClasses1DataContext db = new DataClasses1DataContext())
{
      var dbPersoQuery =
          from perso in db.Personne
          select perso;
      foreach (var elt in dbPersoQuery)
      {
           Console.WriteLine(elt.Id + " " + elt.nom + " " + elt.prenom +
               " " + elt.age);
      }
}
```

L'utilisation de `using` permet d'automatiser plusieurs opérations telle que la fermeture de la connexion.

Linq

Ajouter une personne

```
Personne personne = new Personne();
personne.nom = "mouelhi";
personne.prenom = "achref";
personne.age = 32;
db.Personne.InsertOnSubmit(personne);
```

Linq

Ajouter une personne

```
Personne personne = new Personne();
personne.nom = "mouelhi";
personne.prenom = "achref";
personne.age = 32;
db.Personne.InsertOnSubmit(personne);
```

La classe `Personne` se trouve dans le fichier `DataClasses1.designer.cs`, pas besoin de la créer.

Linq

Ajouter une personne

```
Personne personne = new Personne();
personne.nom = "mouelhi";
personne.prenom = "achref";
personne.age = 32;
db.Personne.InsertOnSubmit(personne);
```

La classe `Personne` se trouve dans le fichier `DataClasses1.designer.cs`, pas besoin de la créer.

Valider les changements

```
db.SubmitChanges();
```

Linq

Attention

- Si on lance le programme plusieurs fois, le même tuple sera inséré une seule fois
- Il y a deux modes
  - **Development** : on travaille donc sur une copie de la base de données, et à chaque lancement, On recharge cette même copie.
  - **Production** : on travaille sur la vraie base de données, si on relance plusieurs fois, le tuple sera ajouté plusieurs fois.

Linq

Comment vérifier ?

- Pour s'assurer, utiliser l'Explorateur de dossiers pour aller dans la racine de votre projet
  - On peut remarquer la présence d'un fichier **MaBase.mdf**. C'est la source utilisée en mode développement.
  - Aller dans `bin/debug`, on peut remarquer aussi la présence d'un deuxième fichier **MaBase.mdf**. C'est le fichier utilisée en mode production.
- Lancer le fichier de votre projet situé dans `bin/debug` plusieurs fois et vérifier que le tuple a aussi été ajouté plusieurs fois

Linq

Une autre problématique

- Si on relance encore une fois le programme à partir de *Visual Studio*, on perd encore les tuples ajoutés.
- Oui, car on a écrasé encore une fois les données.

Linq

Une autre problématique

- Si on relance encore une fois le programme à partir de *Visual Studio*, on perd encore les tuples ajoutés.
- Oui, car on a écrasé encore une fois les données.

Solution

Copier le répertoire `Debug` sur votre bureau (par exemple), et relancé à partir en mode développement ou en mode production, les données seront préservées.

Linq

Trouver une personne

```
Personne p = db.Personne.First(perso => perso.nom.
   Equals("wick"));
Console.WriteLine($"je m'appelle {p.prenom.Trim()} {
   p.nom.Trim()}");
```

Linq

Trouver plusieurs personnes

```
var personnes = db.Personne;
var containsIQuery = personnes.Where(perso => perso.nom.Contains("i"));

foreach (var elt in containsIQuery)
{
    Console.WriteLine(elt.nom);
}
```

Linq

Trouver plusieurs personnes

```
var personnes = db.Personne;
var containsIQuery = personnes.Where(perso => perso.nom.Contains("i"));

foreach (var elt in containsIQuery)
{
    Console.WriteLine(elt.nom);
}
```


Remarques

- `containsIQuery` est de type `IQueryable`.
- Remplacer `var` par `IQueryable<Personne>` puis par `IEnumerable<Personne>` et vérifier que le code est toujours exécutable.

Linq

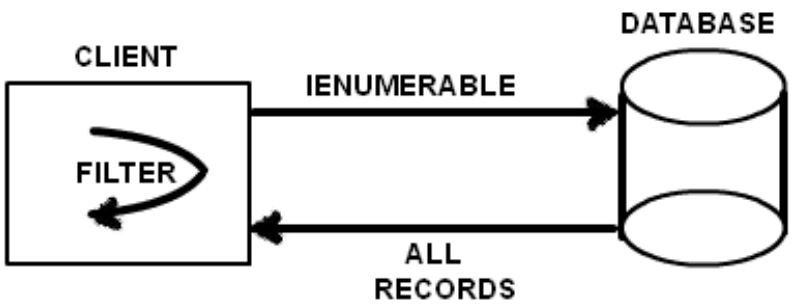
IQueryable vs IEnumerable

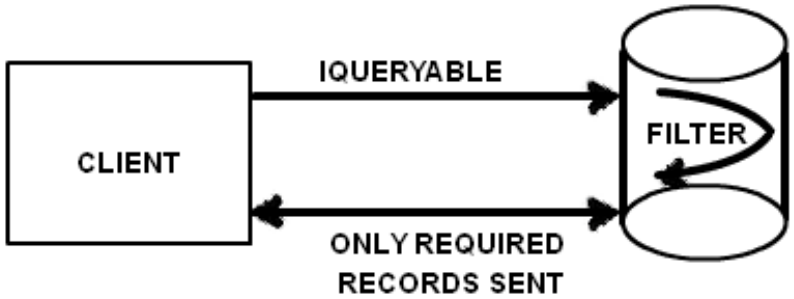
- `IQueryable` est une interface qui hérite de l'interface `IEnumerable`.
- `IEnumerable` utilise une copie de la collection en mémoire. Donc tous les filtres s'appliquent coté client.
- `IQueryable` fonctionne en mode connecté avec la base de données et les filtres s'appliquent coté serveur.

Linq

`IQueryable` vs `IEnumerable`

- `IQueryable` est une interface qui hérite de l'interface `IEnumerable`.
- `IEnumerable` utilise une copie de la collection en mémoire. Donc tous les filtres s'appliquent coté client.
- `IQueryable` fonctionne en mode connecté avec la base de données et les filtres s'appliquent coté serveur.





Source : *StackOverFlow*

Linq

Modifier une personne

```
Personne p = db.Personne.First(perso => perso.nom.
   Equals("wick"));
p.nom = "abruzzi";
db.SubmitChanges();
```

Linq

Modifier une personne

```
Personne p = db.Personne.First(perso => perso.nom.
   Equals("wick"));
p.nom = "abruzzi";
db.SubmitChanges();
```

Supprimer une personne

```
Personne p = db.Personne.First(perso => perso.nom.
   Equals("wick"));
db.Personne.DeleteOnSubmit(p);
db.SubmitChanges();
```

Linq

On peut aussi faire

```
var deletedPersonQuery =
    from perso in db.Personne
    where perso.nom.Equals("wick")
    select perso;

if (deletedPersonQuery.Count() > 0)
{
    db.Personne.DeleteOnSubmit(deletedPersonQuery.
       First());
    db.SubmitChanges();
}
```

Linq

Exercice

Écire un code **C#** utilisant **Linq to SQL** permettant de supprimer plusieurs personnes.

Linq

Linq to SQL Vs **Linq to Entities** Vs **Linq to DataSet**

- Linq : un langage intégré dans le **C#**, inspiré par le langage **SQL**, qui nous permet d'extraire des données provenant de plusieurs sources.
- Si la source de données :
  - une base de données **SQL Server** $\Rightarrow$ **Linq to SQL**
  - des entités **Entity Framework** $\Rightarrow$ **Linq to Entities**
  - un ensemble de données **ADO.NET** $\Rightarrow$ **Linq to DataSet**

Linq

Rappel de quelques mots-clés

- `Element` pour désigner une balise.
- `Attribute` pour désigner un attribut de balise (une balise peut en avoir plusieurs).
- `Descendants` pour désigner les balises définies dans une autre balise.

Linq

Linq to XML : comment ?

Trois étapes :

- Créer un fichier **XML** dans le dossier `bin\Debug` du projet.
- Charger le fichier **XML**.
- Utiliser **Linq to XML** pour interroger l'objet lu comme toute autre source de données.

Linq

Considérons le fichier `personnes.xml` suivant

```
<?xml version="1.0" encoding="utf-8" ?>
<personnes>
  <personne>
    <nom>wick</nom>
    <prenom>john</prenom>
  </personne>
  <personne>
    <nom>dalton</nom>
    <prenom>jack</prenom>
  </personne>
</personnes>
```

Commençons par charger le fichier XML tout en indiquant le chemin

```
var filename = @"personnes.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var xmlFilePath = Path.Combine(currentDirectory, filename);
XElement xdoc = XElement.Load(xmlFilePath);
```

Commençons par charger le fichier XML tout en indiquant le chemin

```
var filename = @"personnes.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var xmlFilePath = Path.Combine(currentDirectory, filename);
XElement xdoc = XElement.Load(xmlFilePath);
```

Préparons la requête Linq tout en utilisant `Descendants` pour accéder aux enfants de chaque élément

```
var personnesQuery = from personne in xdoc.Descendants("personne")
                     select new
                     {
                           Nom = personne.Element("nom").Value.Trim(),
                           Prenom = personne.Element("prenom").Value.Trim()
                     };
```

Commençons par charger le fichier XML tout en indiquant le chemin

```
var filename = @"personnes.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var xmlFilePath = Path.Combine(currentDirectory, filename);
XElement xdoc = XElement.Load(xmlFilePath);
```

Préparons la requête Linq tout en utilisant `Descendants` pour accéder aux enfants de chaque élément

```
var personnesQuery = from personne in xdoc.Descendants("personne")
                     select new
                     {
                           Nom = personne.Element("nom").Value.Trim(),
                           Prenom = personne.Element("prenom").Value.Trim()
                     };
```


Pour afficher le résultat de la requête Linq

```
foreach (var perso in personnesQuery)
{
    Console.WriteLine($"{ perso.Prenom } { perso.Nom }");
}
```

Commençons par charger le fichier XML tout en indiquant le chemin

```
var filename = @"personnes.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var xmlFilePath = Path.Combine(currentDirectory, filename);
XElement xdoc = XElement.Load(xmlFilePath);
```

Préparons la requête Linq tout en utilisant `Descendants` pour accéder aux enfants de chaque élément

```
var personnesQuery = from personne in xdoc.Descendants("personne")
                     select new
                     {
                           Nom = personne.Element("nom").Value.Trim(),
                           Prenom = personne.Element("prenom").Value.Trim()
                     };
```


Pour afficher le résultat de la requête Linq

```
foreach (var perso in personnesQuery)
{
    Console.WriteLine($"{ perso.Prenom } { perso.Nom }");
}
```

john wick

jack dalton

Linq

Et si la balise `personne` avait des attributs

```
<?xml version="1.0" encoding="utf-8" ?>
<personnes>
  <personne id='1'>
    <nom>wick</nom>
    <prenom>john</prenom>
  </personne>
  <personne id='2'>
    <nom>dalton</nom>
    <prenom>jack</prenom>
  </personne>
</personnes>
```

Linq

Préparons la requête Linq et utilisons `Attribute` pour récupérer l'attribut `id`

```
var personnesQuery = from personne in xdoc.Descendants("personne")
                     select new
                     {
                           Nom = personne.Element("nom").Value.Trim(),
                           Prenom = personne.Element("prenom").Value.
                              Trim(),
                           Id = personne.Attribute("id").Value
                     };
```

Linq

Préparons la requête Linq et utilisons `Attribute` pour récupérer l'attribut `id`

```
var personnesQuery = from personne in xdoc.Descendants("personne")
                     select new
                     {
                           Nom = personne.Element("nom").Value.Trim(),
                           Prenom = personne.Element("prenom").Value.
                              Trim(),
                           Id = personne.Attribute("id").Value
                     };
```


Pour afficher le résultat de la requête

```
foreach (var perso in personnesQuery)
{
    Console.WriteLine($"{ perso.Prenom } { perso.Nom }  { perso.Id }");
}
```

Linq

Préparons la requête Linq et utilisons `Attribute` pour récupérer l'attribut `id`

```
var personnesQuery = from personne in xdoc.Descendants("personne")
                     select new
                     {
                            Nom = personne.Element("nom").Value.Trim(),
                            Prenom = personne.Element("prenom").Value.
                               Trim(),
                            Id = personne.Attribute("id").Value
                     };
```


Pour afficher le résultat de la requête

```
foreach (var perso in personnesQuery)
{
    Console.WriteLine($"{ perso.Prenom } { perso.Nom }  { perso.Id }");
}
```

john wick 1

jack dalton 2

Linq

Deuxième version avec les expressions Lambda

```
var personnesQuery = xdoc.Descendants("personne").Select(
                personne => new {
                    Nom = personne.Element("nom").Value.Trim(),
                    Prenom = personne.Element("prenom").Value.Trim(),
                    Id = personne.Attribute("id").Value
                });
```

Linq

Deuxième version avec les expressions Lambda

```
var personnesQuery = xdoc.Descendants("personne").Select(
                personne => new {
                    Nom = personne.Element("nom").Value.Trim(),
                    Prenom = personne.Element("prenom").Value.Trim(),
                    Id = personne.Attribute("id").Value
                });
```

Pour afficher le résultat de la requête

```
foreach (var perso in personnesQuery)
{
    Console.WriteLine($"{ perso.Prenom } { perso.Nom }  { perso.Id }");
}
```

Linq

Deuxième version avec les expressions Lambda

```
var personnesQuery = xdoc.Descendants("personne").Select(
                personne => new {
                    Nom = personne.Element("nom").Value.Trim(),
                    Prenom = personne.Element("prenom").Value.Trim(),
                    Id = personne.Attribute("id").Value
                });
```

Pour afficher le résultat de la requête

```
foreach (var perso in personnesQuery)
{
    Console.WriteLine($"{ perso.Prenom } { perso.Nom }  { perso.Id }");
}
```

john wick 1

jack dalton 2