

Syntaxe

Avant de détailler les différents éléments de LINQ qui peuvent être utilisés pour écrire des requêtes, nous allons étudier un premier exemple qui vous permettra de mieux comprendre ce qu'est LINQ et comment l'utiliser.

Les données sur lesquelles nous travaillerons tout au long de cette étude syntaxique de LINQ sont issues de la base de données Northwind de Microsoft. Une partie de son contenu a été extraite dans trois fichiers au format texte qui seront chargés en mémoire afin de former trois collections d'objets. Cette étape de chargement est effectuée par le code ci-dessous, que vous adapterez de manière à pointer sur le répertoire contenant les fichiers. Les fichiers de données sont disponibles en téléchargement depuis la page Informations générales.

```
class Client
{
    public string Identifiant { get; set; }
    public string Nom { get; set; }
    public string Adresse { get; set; }
    public string Ville { get; set; }
    public string Pays { get; set; }

    public List<ProduitCommande> ProduitsCommandes { get; set; }
}

class ProduitCommande
{
    public int NumeroCommande { get; set; }

    //Référence du Client associé à la commande
    public string IdentifiantClient { get; set; }
    //Référence du Commercial associé à la commande
    public int IdentifiantCommercial { get; set; }

    public DateTime DateCommande { get; set; }
    public string NomProduit { get; set; }
    public decimal PrixUnitaire { get; set; }
    public int Quantite { get; set; }
}

class Commercial
{
    public int Identifiant { get; set; }
    public string Nom { get; set; }
    public string Prenom { get; set; }
}

class Program
{
    //Cette variable doit être modifiée pour pointer
    // sur le répertoire contenant les fichiers de données
    private const string dossierDonnees = @"C:\DATA\Exemples\";

    static void Main(string[] args)
    {
        Console.WriteLine("Chargement des clients en mémoire");
    }
}
```

```

        List<Client> clients = new List<Client>();
        List<ProduitCommande> produitsCommandes = new
List<ProduitCommande>();
        List<Commercial> commerciaux = new List<Commercial>();

        //Lecture des enregistrements de ProduitsCommandes.txt
        string[] lignesProduitsCommandes =
File.ReadAllLines(dossierDonnees + "ProduitsCommandes.txt");
        foreach (string ligne in lignesProduitsCommandes)
        {
            ProduitCommande produitCommande =
new ProduitCommande();
            var valeurs = ligne.Split('\t');

            produitCommande.NumeroCommande =
Convert.ToInt32(valeurs[0]);
            produitCommande.IdentifiantClient = valeurs[1];
            produitCommande.IdentifiantCommercial =
Convert.ToInt32(valeurs[2]);
            produitCommande.DateCommande =
Convert.ToDateTime(valeurs[3]);
            produitCommande.NomProduit = valeurs[4];
            produitCommande.PrixUnitaire =
Convert.ToDecimal(valeurs[5]);
            produitCommande.Quantite =
Convert.ToInt32(valeurs[6]);

            produitsCommandes.Add(produitCommande);
        }

        //Lecture des enregistrements de Clients.txt
        string[] lignesClients =
File.ReadAllLines(destinationFolder + "Clients.txt");
        foreach (string ligne in lignesClients)
        {
            Client client = new Client();
            var valeurs = ligne.Split('\t');

            client.Identifiant = valeurs[0];
            client.Nom = valeurs[1];
            client.Adresse = valeurs[2];
            client.Ville = valeurs[3];
            client.Pays = valeurs[4];

            client.ProduitsCommandes = new
List<ProduitCommande>();
            foreach (ProduitCommande produit in
produitsCommandes)
            {
                if (produit.IdentifiantClient ==
client.Identifiant)
                {
                    client.ProduitsCommandes.Add(produit);
                }
            }
        }
    }
}

```

```

    }

    clients.Add(client);
}

//Lecture des enregistrements de Commerciaux.txt
string[] lignesCommerciaux =
File.ReadAllLines(dossierDonnees + "Commerciaux.txt");
foreach (string ligne in lignesCommerciaux)
{
    Commercial commercial = new Commercial();
    var valeurs = ligne.Split('\t');

    commercial.Identifiant = Convert.ToInt32(valeurs[0]);
    commercial.Nom = valeurs[1];
    commercial.Prenom = valeurs[2];

    commerciaux.Add(commercial);
}

Console.WriteLine("Chargement terminé.");
Console.ReadLine();
}
}

```

1. Une première requête LINQ

Pour écrire une requête LINQ, il faut tout d'abord identifier une source à partir de laquelle des données pourront être extraites. Les sources de données compatibles avec LINQ sont les collections qui implémentent l'interface `IEnumerable<T>`. Les listes d'objets chargées précédemment en mémoire sont stockées dans des collections du type `List<T>`, qui implémente entre autres cette interface.

La requête elle-même contient différents éléments qui définissent la source de données à utiliser, les données à récupérer et les différents filtres et traitement. Seuls les deux premiers sont indispensables dans toute requête LINQ. Ils sont représentés par deux clauses, respectivement placées au début et à la fin de la structure de la requête. La clause `from` permet de définir la source de données. La clause `select` permet quant à elle de définir un objet à retourner pour chaque enregistrement. Cet objet peut être un élément complet de la source de données ou uniquement un sous-ensemble des données.

Notre première requête récupère la liste des clients dont l'adresse est située en France.

```

var requete = from cli in clients
               where cli.Pays == "France"
               select cli;

```

Le type de retour de cette requête est `IEnumerable<Client>`, mais il est courant d'utiliser l'inférence de type avec les requêtes LINQ notamment car la clause `select` peut créer des objets de types anonymes.



Les types anonymes sont décrits dans le chapitre La programmation orientée objet avec C#, section Types anonymes.

À ce stade, malgré les apparences, la variable `requete` ne contient aucun objet de type `Client`. Elle ne contient qu'une définition de la requête qui sera en réalité exécutée de manière paresseuse, c'est-à-dire uniquement lorsque ses données de retour devront être lues. Ce mécanisme permet notamment d'exécuter plusieurs fois la même requête LINQ sans avoir besoin de la redéfinir.

Ici, l'exécution de l'opération de recherche est déclenchée par l'utilisation d'une boucle `foreach`, mais elle peut également être lancée par le calcul d'une valeur d'agrégat (`Count` ou `Max`, notamment) ou par l'utilisation de certaines méthodes récupérant un ou plusieurs éléments sous une forme spécifique (`First` et `ToList`, par exemple).

```
//Affichage du nom et de l'adresse de chaque client
//On affiche également le pays pour vérification
foreach (var resultat in query)
{
    Console.WriteLine(resultat.Nom);
    Console.WriteLine("\t" + resultat.Adresse);
    Console.WriteLine("\t" + resultat.Ville);
    Console.WriteLine("\t" + resultat.Pays);
}
```

Le résultat de l'exécution de ce code est présenté ci-après. On constate que chacun de ces clients est bien localisé en France.

```
Frédérique Citeaux
    24, place Kléber
    Strasbourg
    France
Laurence Lebihan
    12, rue des Bouchers
    Marseille
    France
Janine Labrune
    67, rue des Cinquante Otages
    Nantes
    France
Martine Rancé
    184, chaussée de Tournai
    Lille
    France
Carine Schmitt
    54, rue Royale
    Nantes
    France
Daniel Tonini
    67, avenue de l'Europe
    Versailles
    France
Annette Roulet
    1 rue Alsace-Lorraine
    Toulouse
    France
Marie Bertrand
```

```
265, boulevard Charonne
Paris
France
Dominique Perrier
25, rue Lauriston
Paris
France
Mary Saveley
2, rue du Commerce
Lyon
France
Paul Henriot
59 rue de l'Abbaye
Reims
France
```

2. Les opérateurs de requête

LINQ fournit de nombreux opérateurs qu'il est important de bien connaître pour l'écriture de requêtes efficaces. Ces opérateurs couvrent un large spectre fonctionnel qu'il est possible de scinder en six catégories :

- Projection
- Filtrage
- Triage
- Partitionnement
- Jointure et regroupement
- Agrégation

Chaque opérateur existe toujours sous la forme d'une méthode applicable aux objets de type `IEnumerable<T>`. Certains opérateurs sont également intégrés au langage C# et ont donc un ou plusieurs mots-clés qui leur sont associés. Lorsqu'il est possible de le faire, les deux modes d'écriture seront détaillés.



Les méthodes LINQ sont généralement utilisées avec des expressions lambda. Elles peuvent également être utilisées avec des fonctions et délégués, mais par souci de lisibilité et d'écriture de code "standard", ce sont les expressions lambda qui seront utilisées ici.

a. Projection

Une projection est un processus de transformation d'un objet vers une autre forme qui consiste souvent en un sous-ensemble de propriétés de l'objet initial. Le type de destination ne doit pas nécessairement être déclaré : comme tout objet instancié en C#, il peut être d'un type anonyme.

LINQ propose deux implémentations pour la projection de données. Celles-ci se trouvent dans les méthodes `Select` et `SelectMany`.

Select<TResult> (alias C# : select)

La méthode `Select` effectue la transformation d'un élément en un nouvel élément par l'utilisation d'une fonction

associant un objet fourni en entrée à un objet qui est retourné.

Les noms et prénoms des différents commerciaux sont récupérés en fournissant une expression qui crée un objet de type anonyme pour chacun des objets de la collection.

```
Console.WriteLine("Noms des commerciaux :");

var nomsCommerciaux = commerciaux.Select(c => new { Nom = c.Nom,
Prenom = c.Prenom });
foreach (var comm in nomsCommerciaux)
{
    Console.WriteLine("{0} {1}", comm.Prenom, comm.Nom);
}
```

La requête peut être réécrite pour utiliser l'extension LINQ du langage C#. Le mot-clé associé est `select`.

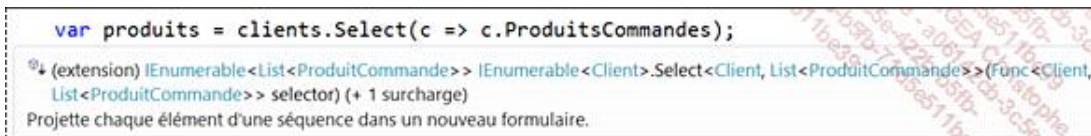
```
var nomsCommerciaux2 = from c in commerciaux
                        select new { Nom = c.Nom, Prenom =
c.Prenom };
```

Dans tous les cas, le résultat de l'exécution est le suivant.

```
Nom des commerciaux :
Nancy Davolio
Andrew Fuller
Janet Leverling
Margaret Peacock
Steven Buchanan
Michael Suyama
Robert King
Laura Callahan
Anne Dodsworth
```

SelectMany<TResult>

Dans certains cas, il est nécessaire que la projection retourne plusieurs objets qui seront intégrés dans la collection retournée par l'exécution de la requête. Pour cela, la méthode `Select` n'est pas adaptée car elle retournerait une collection contenant plusieurs collections :




```
var produits = clients.Select(c => c.ProduitsCommandes);
```

(extension) `IEnumerable<List<ProduitCommande>>` `IEnumerable<Client>.Select<Client, List<ProduitCommande>>>(Func<Client, List<ProduitCommande>> selector) (+ 1 surcharge)`
Projeté chaque élément d'une séquence dans un nouveau formulaire.

Pour éviter cette problématique, il convient d'utiliser la méthode `SelectMany`. Celle-ci effectue le même traitement que la méthode `Select` mais l'expression qui lui est passée doit renvoyer une collection.

```
var produits = clients.SelectMany(c => c.ProduitsCommandes);
```

 (extension) IEnumerable<ProduitCommande> IEnumerable<Client>.SelectMany<Client, ProduitCommande>(Func<Client, IEnumerable<ProduitCommande>> selector) (+ 3 surcharges)
 Projette chaque élément d'une séquence sur un IEnumerable<out T> et aplatit les séquences résultantes en une seule séquence.

➤ Il est à noter que la fonction passée à la méthode `SelectMany` peut renvoyer une chaîne de caractères car celle-ci est considérée comme une collection d'objets de type `char`. Attention, ce comportement est rarement désirable !

b. Filtrage

Le filtrage est une fonctionnalité essentielle dans le traitement de données. Il est en effet souvent souhaitable de limiter le nombre d'enregistrements retournés en fournissant un ou plusieurs critères. Dans le cadre de l'écriture de notre première requête LINQ, cet opérateur a été utilisé sous sa forme intégrée pour ne retourner que les clients français.

```
var clientFrance = from cli in clients
    where cli.Pays == "France"
    select cli;
```

La comparaison de la propriété `Pays` et de la chaîne `"France"` est effectuée à l'aide d'un opérateur standard de C#. Ceux-ci peuvent en effet tous être utilisés dans une requête LINQ. Toutefois, ici la comparaison n'est pas optimale. Il peut en effet arriver que la propriété contienne la valeur `"france"`, et non `"France"`. Or pour C#, ces deux valeurs sont différentes, ce qui peut entraîner le retour d'une collection qui paraît incomplète. Dans ce cas, il convient d'utiliser la méthode `Equals` de la classe `string`. Elle accepte en effet un paramètre qui permet de rendre la comparaison insensible à la casse.

```
var clientsFrance = from cli in clients
    where cli.Pays.Equals("France",
        StringComparison.OrdinalIgnoreCase)
    select cli;
```

La méthode `Where` nécessite le passage en paramètre d'une expression renvoyant une valeur booléenne et renvoie une collection dont les éléments sont du même type que ceux de la collection source.

```
var clientsFrance = clients.Where(cli =>
    cli.Pays.Equals("France", StringComparison.OrdinalIgnoreCase));
```

Lorsque plusieurs conditions doivent être vérifiées, celles-ci peuvent naturellement être combinées pour ne former qu'une expression complexe.

```
var clientsNantes = from cli in clients
    where cli.Pays.Equals("France",
        StringComparison.OrdinalIgnoreCase)
    && cli.Ville.Equals("Nantes",
        StringComparison.OrdinalIgnoreCase)
    select cli;
```

Il est également possible de créer plusieurs expressions `where`, ce qui peut simplifier la lecture de la requête.

```
var clientsNantes = from cli in clients
                    where cli.Pays.Equals("France",
StringComparison.OrdinalIgnoreCase)
                    where cli.Ville.Equals("Nantes",
StringComparison.OrdinalIgnoreCase)
                    select cli;
```

c. Triage

Il est très simple de récupérer une liste d'éléments triés en fonction d'un ou de plusieurs critères.

Lors de l'écriture d'une requête avec la syntaxe intégrée à C#, il suffit d'ajouter une clause `orderby` suivie d'une ou plusieurs propriétés séparées par des virgules. Le tri est effectué sur la première propriété, puis, si certaines valeurs sont identiques, sur la seconde propriété, et ainsi de suite.

```
//La clause where permet de limiter les résultats
var clientsOrdonnes = from c in clients
                      where c.ProduitsCommandes.Count < 10
                      orderby c.ProduitsCommandes.Count
                      select c;

foreach (var resultat in clientsOrdonnes)
{
    Console.WriteLine(resultat.ProduitsCommandes.Count + " : " +
resultat.Nom);
}
```

L'exécution de ce code produit l'affichage suivant :

```
0 : Diego Roel
0 : Marie Bertrand
2 : Francisco Chang
2 : John Steel
4 : Manuel Pereira
6 : Martín Sommer
6 : Carine Schmitt
6 : Simon Crowther
6 : Dominique Perrier
7 : Elizabeth Brown
7 : Liz Nixon
8 : Eduardo Saavedra
8 : Yoshi Tannamuri
8 : Liu Wong
9 : Janine Labrune
9 : Yoshi Latimer
9 : Helvetius Nagy
```


Le tri peut être effectué dans le sens ascendant ou descendant en ajoutant les mots-clés `ascending` ou `descending` après le nom du champ sur lequel vous voulez spécifier l'ordre du tri. Par défaut, le tri est ascendant sur tous les champs.

Le code suivant ajoute une contrainte de tri descendant sur la propriété `Nom`.

```
//La clause where permet de limiter les résultats
var clientsOrdonnes = from c in clients
                      where c.ProduitsCommandes.Count < 10
                      orderby c.ProduitsCommandes.Count, c.Nom
descending
                      select c;
```

Les mots-clés `orderby`, `ascending` et `descending` correspondent à quatre méthodes fournies par LINQ :

- `OrderBy`
- `OrderByDescending`
- `ThenBy`
- `ThenByDescending`

Les deux premières méthodes permettent de spécifier le premier critère de tri ainsi que le type de tri associé. `ThenBy` et `ThenByDescending` ont le même rôle mais sont applicables pour les critères de tri suivants. Les appels à ces méthodes peuvent donc être chaînés. La dernière requête LINQ réécrite à l'aide de ces méthodes a l'aspect suivant :

```
//La méthode Where permet de limiter les résultats
var clientsOrdonnes =
    clients.Where(c => c.ProduitsCommandes.Count < 10)
            .OrderBy(c => c.ProduitsCommandes.Count)
            .ThenByDescending(c => c.Nom);
```

d. Partitionnement

Le partitionnement correspond au découpage en deux parties d'un ensemble de données dans l'objectif de n'en retourner qu'une seule. La limite entre ces deux parties peut être définie de manière absolue ou par une condition.

Le premier cas est mis en œuvre par l'utilisation des méthodes `Skip` et `Take`. `Skip` offre la possibilité de scinder une collection après un certain nombre d'éléments et de ne retourner que la seconde partie. `Take` effectue l'inverse en ne renvoyant que le nombre d'éléments indiqué en tête de collection.

Le code suivant crée une collection contenant les dix premiers clients et une autre collection contenant tous les autres éléments.

```
var premiersClients = clients.Take(10);
var autresClients = clients.Skip(10);
```

Ces méthodes sont souvent utilisées pour la construction de systèmes de pagination. Elles sont alors appliquées à des ensembles ayant préalablement été triés à l'aide de la méthode `OrderBy`, ceci afin de s'assurer que chaque élément correspond à une seule page et ne sera pas réaffiché ou, au contraire, omis.

Le partitionnement conditionnel est quant à lui appliqué par l'utilisation des méthodes `SkipWhile` et `TakeWhile`. Celles-ci acceptent toutes deux comme paramètre une fonction dont la valeur de retour est de type booléen. La première renvoie les éléments d'une collection à partir du moment où la condition n'est plus respectée. La seconde retourne les éléments en tête de collection tant que la fonction passée en paramètre renvoie `true`.

L'exemple suivant crée une variable `clientsPlusDe30Produits` contenant la liste des clients ayant commandé plus de 30 produits.

```
var clientsPlusDe30Produits = clients.OrderBy(c =>
c.ProduitsCommandes.Count()).SkipWhile(c =>
c.ProduitsCommandes.Count() <= 30);
```

e. Jointure et regroupement

La jointure de deux sources de données correspond à l'association automatique de ces sources par le biais d'une propriété commune aux objets des deux collections. Sans l'utilisation de jointures, il serait nécessaire de parcourir manuellement une des deux sources de données à la recherche d'un élément correspondant à chacun des objets de l'autre collection.

Avec LINQ, cette opération est effectuée à l'aide de l'opérateur `join`. Les collections et les champs concernés sont fournis à la suite de ce mot-clé.

La requête suivante montre comment utiliser cet opérateur pour récupérer et afficher le nom et le prénom de chaque commercial ainsi que les informations relatives aux produits commandés qui leur sont associés.

```
var produitsEtCommerciaux =
    from c in commerciaux
    join c in produitsCommandes on c.Identifiant equals
p.IdentifiantCommercial
    select new
    {
        NomProduit = p.NomProduit,
        Quantite = p.Quantite,
        PrixUnitaire = p.PrixUnitaire,
        Commercial = string.Format("{0} {1}", c.Prenom, c.Nom)
    };

foreach (var produitCommercial in produitsEtCommerciaux)
{
    Console.WriteLine(produitCommercial.NomProduit);
    Console.WriteLine("\tPrixUnitaire : {0}",
produitCommercial.PrixUnitaire);
    Console.WriteLine("\tQuantite : {0}",
produitCommercial.Quantite);
    Console.WriteLine("\tCommercial : {0}",
produitCommercial.Commercial);
}
```

La méthode `Join` associée à ce mot-clé accepte quatre paramètres :

- La collection impliquée dans la jointure.
- Une fonction qui retourne la valeur à utiliser pour la jointure dans la collection courante.
- Une fonction qui retourne la valeur à utiliser pour la jointure dans la seconde collection.
- Une fonction qui retourne les objets qui forment le résultat de la jointure.

Les fonctions sont généralement passées sous la forme d'expressions lambda, plus simples à lire et très rapides à écrire.

La requête ci-dessus peut être réécrite avec la méthode `Join` de la manière suivante :

```
var produitsEtCommerciaux =
    commerciaux.Join(
        produitsCommandes,
        c => c.Identifiant,
        p => p.IdentifiantCommercial,
        (p, c) => new
        {
            NomProduit = p.NomProduit,
            Quantite = p.Quantite,
            PrixUnitaire = p.PrixUnitaire,
            Commercial = string.Format("{0} {1}", c.Prenom, c.Nom)
        });
```

La propriété `Commercial` est définie pour chacun des objets renvoyés par la requête, et plusieurs de ces enregistrements peuvent avoir la même valeur pour cette propriété. Une manière de rendre ce code plus efficace est de grouper les enregistrements par commercial afin de ne pas avoir de valeur dupliquée. Le mot-clé `into` placé après une jointure et associé à un identifiant permet justement ce groupement.

```
var produitsEtCommerciauxGroups =
    from c in commerciaux
    join p in produitsCommandes on c.Identifiant equals
p.IdentifiantCommercial into prod
    select new
    {
        Produits = prod,
        Commercial = string.Format("{0} {1}", c.Prenom, c.Nom)
    };

foreach (var produitCommercial in produitsEtCommerciauxGroups)
{
    foreach (var produit in produitCommercial.Produits)
    {
        Console.WriteLine(produit.NomProduit);
        Console.WriteLine("\tPrixUnitaire : {0}",
produit.PrixUnitaire);
        Console.WriteLine("\tQuantite : {0}", produit.Quantite);
        Console.WriteLine("\tCommercial : {0}",
```

```
produitCommercial.Commercial)
    }
}
```

La méthode associée aux mots-clés `join ... into ...` n'est plus `Join` mais `GroupJoin`. Elle accepte presque les mêmes paramètres que la méthode `Join`, la différence entre les deux réside dans la fonction générant les objets constituant le résultat de la jointure : le second paramètre de cette fonction n'est plus de type `ProduitCommande` mais de type `IEnumerable<ProduitCommande>`.

```
var produitsEtCommerciaux4 =
    commerciaux.GroupJoin(
        produitsCommandes,
        c => c.Identifiant,
        p => p.IdentifiantCommercial,
        (c, prod) => new
        {
            Produits = prod,
            Commercial = string.Format("{0} {1}", c.Prenom, c.Nom)
        }
    );
```

f. Agrégation

Les opérations d'agrégation sont définies dans LINQ par un jeu de méthodes renvoyant une valeur unique, généralement de type numérique (mais pas obligatoirement). L'utilisation de ces méthodes est appropriée pour :

- calculer un nombre d'éléments,
- calculer une moyenne,
- renvoyer une valeur minimale ou maximale,
- calculer une somme.

Calcul d'un nombre d'éléments : Count

La méthode `Count` possède deux définitions. La première, sans paramètre, renvoie le nombre d'éléments contenus dans la collection sur laquelle elle est appliquée.

```
int nombreElements = commerciaux.Count();
```

La seconde définition accepte quant à elle un paramètre qui permet de lui passer un prédicat. La valeur retournée par la méthode correspond au nombre d'éléments respectant ce prédicat.

```
int nombreProduitsPasCher = produitsCommandes.Count(p =>
    p.PrixUnitaire < 10);
```

Calcul d'une moyenne : Average

La méthode `Average` requiert en paramètre une fonction renvoyant une valeur numérique pour chaque objet de la collection. Elle effectue sur ces valeurs un calcul de moyenne et renvoie le résultat.

```
double montantMoyenCommande = produitsCommandes.Average(p =>
p.PrixUnitaire * p.Quantite);
```

Recherche d'une valeur minimale : Min

La fonction `Min` a deux définitions permettant de rechercher la valeur minimale dans :

- Une collection d'objets numériques ou de type chaîne,
- Un ensemble de valeurs numériques (ou de type chaîne) renvoyées par la fonction passée en paramètre.

```
double prixLeMoinsEleve = produitCommandes.Min(p =>
p.PrixUnitaire);
```

Recherche d'une valeur maximale : Max

La méthode `Max` est utilisée exactement de la même manière que `Min`, mais fournit une valeur maximale pour un ensemble d'éléments.

```
double prixLePlusEleve = produitCommandes.Max(p =>
p.PrixUnitaire);
```

Calcul d'une somme : Sum

La méthode `Sum` calcule la somme des valeurs retournées par la fonction qui lui est passée en paramètre.

```
double totalCommandes = produitCommandes.Sum(p => p.PrixUnitaire
* p.Quantite);
```