



Cégep de Saint-Hyacinthe
Département d'informatique

Programmation orientée objet

420-2DP-HY

(3-3-3)

Introduction aux collections et à LINQ #1

(Version 1.0)

3 heures

Préparé par

Martin Lalancette

Comprendre les éléments suivants:

- Collections (définition, types)
- Introduction LINQ (Filtre et tri)
- Les classes anonymes
- Collections imbriquées

Table des matières

Introduction.....	3
Introduction aux collections.....	3
Qu'est-ce qu'une collection?.....	3
Types de collection.....	4
Introduction à LINQ.....	6
Comprendre le mot-clé <i>var</i>	6
Création d'une requête LINQ - Filtrer.....	7
Création d'une requête LINQ – Trier.....	9
Les classes anonymes.....	11
Utilisation de LINQ avec des collections imbriquées.....	15
Éliminer les doublons avec <i>Distinct()</i>	15
Les possibilités du <i>select</i>	17
Utilisation du mot-clé <i>let</i>	18
Utilisation d'une méthode/fonction personnelle.....	19
Autres - Exercices.....	20
Obtenir les N premiers éléments grâce à <i>Take()</i>	21
Obtenir tous les éléments sauf les N premier grâce à <i>Skip()</i>	21
Bibliographie.....	23

Introduction

Cette séquence a pour but de vous initier aux nécessaires à la programmation à base d'objets et d'événements. Nous commencerons par énoncer les éléments théoriques appuyés d'exemples simples et faciles à reproduire. Afin d'axer l'attention sur la compréhension de ces notions, il y aura des exercices à faire tout au long de cette séquence. **La séquence portera sur une introduction aux collections et à LINQ.** Pour bien suivre les instructions qui vont être mentionnées tout au long des séquences d'apprentissage, une préparation de base s'impose. Il est important de créer un répertoire de travail (sur votre C: ou clé USB). Voici une suggestion d'arborescence:



Exercice 1. : S'assurer d'avoir créé l'arborescence ici haut mentionnée sur votre P ou votre clé USB. Copier ce document dans le répertoire en rouge ici haut mentionné.

Introduction aux collections

Lors de la conception d'applications, il est courant de vouloir créer et gérer un ensemble ou groupe d'objets similaires. Il existe deux manières de grouper des objets : en créant des **tableaux d'objets** ou en créant des **collections d'objets**. Nous sommes déjà familiers avec les tableaux. Nous allons voir maintenant les collections.

Qu'est-ce qu'une collection?

Le framework .NET fournit des **classes spécialisées** pour le stockage et récupération de données. Les collections sont l'amélioration des tableaux. Les tableaux sont plus utiles pour la création et l'utilisation d'un **nombre fixe**

d'objets fortement typés. Tandis que les collections offrent plus de souplesse pour utiliser des groupes d'objets. Contrairement aux tableaux, le groupe d'objets utilisé **peut augmenter ou diminuer de façon dynamique si les besoins de l'application varient**.

Types de collection

Les collections ont été regroupées dans différents espaces de nom afin de répondre à des besoins spécifiques. Voici un tableau explicatif :

Espace de nom	Description	Classes
System.Collections.Generic	Contient des interfaces et des classes qui définissent des collections génériques permettant aux utilisateurs de créer des collections fortement typées . Celles-ci fournissent une cohérence des types et des performances meilleures que les collections fortement typées non génériques.	Dictionary : Représente une collection de paires clé/valeur organisées en fonction de la clé.
		List : Représente une liste fortement typée d'objets accessibles par leur index. Fournit des méthodes de recherche, de tri et de modification de listes.
		Queue : Représente une collection d'objets de type premier entré, premier sorti (FIFO, first-in-first-out).
		Stack : Représente une collection d'objets de type dernier entré, premier sorti (LIFO).
System.Collections	Les classes de l'espace de noms System.Collections ne stockent pas les éléments comme des objets spécifiquement typés, mais comme des objets de type Object.	SortedList : Représente une collection de paires clé/valeur triées par clé en fonction de l'implémentation <code>IComparer<T></code> associée
		ArrayList : Représente un tableau d'objets dont la taille est augmentée dynamiquement comme cela est requis.
		Hashtable : Représente une collection de paires clé/valeur qui sont organisées en fonction du code de hachage de la clé.
		Queue : Représente une collection d'objets de type premier entré, premier sorti (FIFO, first-in-first-out).
		Stack : Représente une collection d'objets de type dernier entré, premier sorti (LIFO).

System.Collections.Concurrent	Les collections dans l'espace de noms fournissent des opérations thread-safe efficaces pour accéder aux éléments de collecte de plusieurs threads.	...
System.Collections.Specialized	Contient des collections spécialisées et fortement typées ; par exemple, un dictionnaire de liste liée, un vecteur de bits et des collections qui ne contiennent que des chaînes.	...

Exemple d'une collection générique:

```
List<string> prenom = new List<string>();
```

```
list.Add("Marc");
list.Add("Carl");
list.Add("Marie");
list.Add("Laurie");
```

Ici la variable **prénoms** se trouve à être une collection générique utilisant la classe **List** pouvant contenir des objets de type **string**.

Nous utilisons déjà des collections d'objets avec la classe **List**. Nous allons exploiter de plus en plus le potentiel des collections grâce aux fonctionnalités de LINQ.

Voici quelques méthodes :

Méthode	Description
Max	Permet de retourner la valeur la plus grande parmi toutes les valeurs contenues dans le tableau.
Min	Permet de retourner la valeur la plus petite parmi toutes les valeurs contenues dans le tableau.
Add	Permet d'ajouter un objet à la collection.
...	... Autres à explorer...

Introduction à LINQ

LINQ (Language INTe grated Query) est un langage à l'intérieur du langage C# qui est apparu avec la **version 3** du *.NET framework*. Il permet la **manipulation et l'interrogation des collections** au sens large (vecteurs, listes, et bases de données, ...). Quelques caractéristiques :

- Son utilisation peut se faire facilement via des instructions supplémentaires permettant de filtrer des données, faire des sélections, etc.
- Il possède également les avantages de valider les types dès la compilation, et d'être pris en charge par l'Intellisense. Ce qui facilite beaucoup son utilisation.

Il existe 3 principaux domaines d'applications pour le LINQ :

1. **LINQ to Object** : **Traitement des collections (objets) en mémoire. C'est l'aspect que nous allons couvrir dans ce cours.**
2. **LINQ to XML** : Permet de charger, de sérialiser, d'interroger (et autres) des collections d'éléments et d'attributs sous forme de document XML.
3. **LINQ to SQL** ou **LINQ to entities**: Permet de modéliser, de mettre à jour, d'insérer et de supprimer des données dans une base de données relationnelle selon les techniques O/RM (Object Relational Mapping).

Comprendre le mot-clé *var*

Depuis Visual C# 3.0, les variables déclarées à la portée de la méthode peuvent avoir un type implicite **var**. Une variable locale implicitement typée est fortement typée comme si vous aviez déclaré le type vous-même, mais le compilateur détermine le type. Voici un exemple :

```
var vAge = 24; // Type implicite: le compilateur déterminera pour vous le type int
int iAge = 24; // Type explicite: le programmeur a déterminé le type int

var vNom = "Tremblay"; // Type implicite: le compilateur détermine le type string
string sNom = "Tremblay"; // Type explicite: le programmeur choisit le type string
```

Même si le type **var** semble alléchant, car il évite au programmeur de choisir le type de la variable, il faut l'utiliser avec vigilance et que dans de rares occasions. S'il fallait déclarer toutes nos variables de type **var**, le code serait difficile de lire. Dans le cas de requêtes LINQ, son utilisation sera permise.

Création d'une requête LINQ - Filtrer

La syntaxe proposée pour concevoir des requêtes LINQ s'inspire sensiblement de la syntaxe utilisée pour la création de requêtes SQL (Langage utilisé pour interroger des bases de données). Donc, maîtriser le LINQ to Object, va vous aider lorsque vous vous connecterez à une base de données. Commençons par quelques mots-clés de base pour effectuer une requête:

Mots clés	Description
<u>from</u>	Introduit la variable d'itération (ou l'énumérateur) qui sert à parcourir la collection
<u>where</u>	Spécifie la condition qui teste les éléments de la collection (une expression booléenne);
<u>select</u>	Définit les données retournées en une collection créée par LINQ.

Exemple #1 : Parmi la liste de prénoms, extraire seulement ceux de 4 caractères.

Collection de prénoms

```
// Champs
private List<string> prenom = new List<string>() {"Marc", "Carl", "Marie", "Luc"};
...
private void btnExercice01_Click(object sender, RoutedEventArgs e)
{
    // Requête: Parmi la liste de prénoms, extraire seulement ceux de 4 car.
    IEnumerable<string> req = from sPrenom in prenom
                             where sPrenom.Length == 4
                             select sPrenom;

    // Afficher le résultat de la requête.
    foreach (string sPrenomReq in req)
        txtResultat.AppendText("Le prénom est: " + sPrenomReq);
}
```

Le **retour d'une requête LINQ est une collection de types IEnumerable<T>**.

Exemple :

```
IEnumerable<string> req = from sPrenom in prenoms
                        where sPrenom.Length == 4
                        select sPrenom;
```

Cependant, vous pouvez simplifier la syntaxe en déclarant la variable servant à la requête avec le type *var*. Exemple :

```
var req = from sPrenom in prenoms
          where sPrenom.Length == 4
          select sPrenom;
```

Exemple #2 : Parmi la liste de prénoms, extraire seulement ceux commençant par la lettre M ou bien C.

```
// Champs
private List<string> prenoms = new List<string>() {"Marc", "Carl", "Marie", "Luc"};
...
private void btnExercice01_Click(object sender, RoutedEventArgs e)
{
    // Requête: Parmi la liste de prénoms, extraire seulement
    // ceux commençant par la lettre M ou bien C.
    var req = from sPrenom in prenoms
              where sPrenom.StartsWith("M", true, null) ||
                  sPrenom.StartsWith("C", true, null)
              select sPrenom;

    // Afficher le résultat de la requête.
    foreach (string sPrenom in req)
        txtResultat.AppendText("Le prénom est: " + sPrenom + "\n");
}
```

Résultat : Marc, Carl et Marie.

Exercice 2. Récupérer la solution portant le nom de ce document dans le fichier .ZIP. Ouvrir le projet « **Théorie** ». Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection **_nombres**, va extraire les **nombres plus grand ou égal à 40**.

Exercice 3. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection **_nombres**, va extraire les **nombres se situant entre 75 et 125**.

Exercice 4. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection `_nombres`, va extraire les **nombres NE se situant PAS entre 80 et 120.**

Exercice 5. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection `_ventes`, va extraire les **montants entre 1000 et 2000 ou bien entre 2500 et 3500.**

Exercice 6. Dans l'événement Click du bouton associé, ajouter le code, à partir de la collection `_employes`, va extraire les **noms et pré-noms des employés travaillant dans la division MTL.**

Exercice 7. Dans la classe `Employe`, modifier le comportement de l'appel de `ToString()` pour retourner les informations selon l'exemple suivant :

E00001 (MTL) Roy Luc (1980-01-23) - Ancienneté: 10

Vous devez utiliser le modificateur **`override`**.

Exercice 8. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection `_employes`, va extraire les **employés NE travaillant PAS dans la division RVS et ayant plus de 10 ans d'ancienneté.** Afficher les infos des employés (`ToString`).

Création d'une requête LINQ – Trier

Nous avons vu précédemment que nous pouvons filtrer une collection d'objets, maintenant nous allons voir comment trier les informations d'une collection.

Voici le mot clé et sa description.

Mot clé	Description
orderby	La clause <code>orderby</code> entraîne le tri en ordre croissant ou décroissant de la séquence ou de la sous-séquence (groupe) retournée dans une expression de requête. Plusieurs clés peuvent être spécifiées pour effectuer une ou plusieurs opérations de tri secondaires. Le tri est effectué par le comparateur par défaut pour le type de l'élément. L'ordre de tri par défaut est le tri croissant.

Vous pouvez spécifier l'ordre dans lequel vous désirez effectuer le tri en utilisant les mots clés suivant dans la clause *orderby* :

Ordre	Description
ascending	Le mot clé contextuel <code>ascending</code> est utilisé dans la clause <code>orderby</code> dans les expressions de requête afin de spécifier que l'ordre de tri est <u>du plus petit au plus grand</u> . (DÉFAUT)
descending	Le mot clé contextuel <code>descending</code> est utilisé dans la clause <code>orderby</code> dans les expressions de requête afin de spécifier que l'ordre de tri est <u>du plus grand au plus petit</u> .

Exemple #1 : Parmi la liste de prénoms, extraire seulement ceux de 4 caractères et **trier en ordre descendant**.

```
var req = from sPrenom in _prenoms
          where sPrenom.Length == 4
          orderby sPrenom descending
          select sPrenom;
```

Exemple #2 : Trier les employés par Division et date de naissance (du plus jeune au plus vieux).

```
var req = from employe in _employes
          orderby employe.Division ascending, employe.Naissance descending
          select employe;
```

Comme le retour d'une requête LINQ est une collection de type `IEnumerable<T>`, le résultat peut servir dans une nouvelle requête LINQ. Exemple :

```
var req1 =  
    from vente in _ventes // pour chaque vente dans ventes  
    where vente >= 2500.00d // où le vente est >= à 2000  
    select vente;          // choisir le vente et l'ajouter  
                           // à la collection IEnumerable<double>  
  
req1 = from vente in req1  
        orderby vente ascending // orderby ascending est implicite  
        select vente;          // orderby descending est explicite
```

Exercice 9. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection `_nombres`, va extraire les nombres au-dessus de la **moyenne** et trier du plus grand au plus petit.

Exercice 10. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection `_prenoms`, va extraire les prénoms du plus petit au plus grand en nombre de caractères.

Exercice 11. Dans l'événement Click du bouton associé, ajouter le code qui à partir de la collection `_employes`, va extraire les employés qui ont (30 ans (âge) et plus et faisant partie de la division MTL ou RVS), (ou bien ayant moins de 10 ans d'ancienneté et faisant partie de la division RVN). Trier sur le **nom** et la **date de naissance**.

Les classes anonymes

Nous avons vu dans la première séquence comment interroger un ensemble d'objets contenus dans une collection grâce au potentiel de LINQ. Ce potentiel ne se limite pas qu'à ça. Nous allons en approfondir un peu plus dans la prochaine section. Tout d'abord, abordons le sujet des classes dites « **anonymes** ». Nous avons la capacité d'instancier des objets à la volée, c.-à-d. sans utiliser une variable intermédiaire dans différentes circonstances :

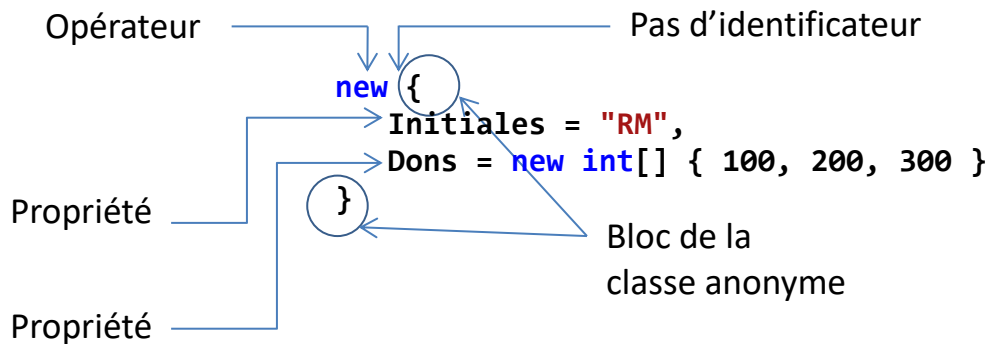
- Fournir les éléments d'une nouvelle collection
- Fournir un paramètre effectif en utilisant l'opérateur new
- Faire le return d'un résultat en utilisant l'opérateur new
- Etc.

```
int addition = Additionner( new int[] { 10, 20, 30, 40, 50 } );
```

Collection de `int` à la volée

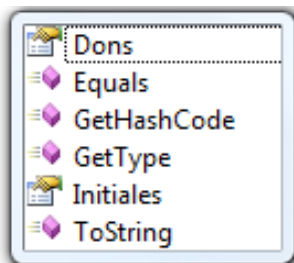
Maintenant, voici comment déclarer et instancier une classe à la volée...
Une classe anonyme. Exemple :

```
var anonyme = new
{
    Initiales = "RM",
    Dons = new int[] { 100, 200, 300 }
};
```



Une classe anonyme n'a **pas d'identificateur** et ses **propriétés sont en lecture seulement**.

```
var anonyme = new {
    Initiales = "RM",
    Dons = new int[] { 100, 200, 300 }
};
anonyme.
```



Chaque type anonyme est un **type interne connu seulement par C#**. Un type anonyme sera réutilisé si les propriétés qui le composent portent le même identificateur, dans la même séquence. Exemple :

```
var ta1 = new { Nom = "Pierre", Age = 18 };
var ta2 = new { Nom = "Jean", Age = 18 };
var ta3 = new { Age = 19, Nom = "Jacques" };
var ta4 = new { Nom = "Luc", Age = 21, Taille = 1.80f };
var ta5 = new { Prenom = "Mathieu", Age = 20 };

txtRésultat.AppendText(ta1.GetType() + "\n"); // nouveau type anonyme { Nom, Age }
txtRésultat.AppendText(ta2.GetType() + "\n"); // même type anonyme { Nom, Age }
txtRésultat.AppendText(ta3.GetType() + "\n"); // nouveau type anonyme { Age, Nom }
txtRésultat.AppendText(ta4.GetType() + "\n"); // nouveau type anonyme { Nom, Age, Taille }
txtRésultat.AppendText(ta5.GetType() + "\n"); // nouveau type anonyme { Prenom, Age }
```

Résultat:

```
<>f__AnonymousType0`2[System.String,System.Int32]
<>f__AnonymousType0`2[System.String,System.Int32]
<>f__AnonymousType1`2[System.Int32,System.String]
<>f__AnonymousType2`3[System.String,System.Int32,System.Single]
<>f__AnonymousType3`2[System.String,System.Int32]
```

Vous n'aurez probablement pas à vous soucier du type anonyme, car **var** se charge de faire le travail pour vous. Cependant, si vous désirez **affecter le résultat d'une autre requête LINQ dans une variable déjà déclarée** alors le type du résultat doit correspondre au type de la variable. Avec les types anonymes, c'est plus embêtant...

À savoir :

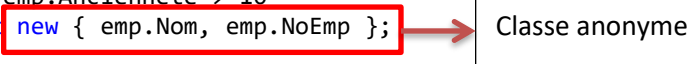
- Le principe d'instancier une classe anonyme (avec **new{ }**) est largement utilisé avec LINQ.
- Le but est de créer des types à la volée pour encapsuler des données fournies au select (sans créer des classes nommées intermédiaires).
- L'utilisation d'une classe anonyme est faite pour des besoins ponctuels et limités :
 - Ne peut être le type d'un membre (champ, propriété, ...),
 - Est **locale** à la méthode (et déclarée avec **var**),
 - Les propriétés sont en **lecture seule**, aucun autre type de membre de classe tel que des méthodes ou des événements n'est valides,

- Le **type est inaccessible** (type interne),
- Ne peut être utilisée comme paramètre formel (déclaré) ou effectif (à l'appel),
- Ne peut être utilisée avec un **return**.

Rappel : un type anonyme est un type local à une méthode.

Exemple avec une requête LINQ :

```
var req = from emp in _employees
          where emp.Division != Employee.enumDivision.RVS &&
                emp.Anciennete > 10
          select new { emp.Nom, emp.NoEmp };
foreach (var emp in req)
    txtRésultat.AppendText(emp.Nom + "\t" + emp.NoEmp + "\n");
```



Exercice 12. Récupérer la solution portant le nom de ce document dans le fichier .ZIP. Ouvrir le projet « **Théorie** ». Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_employees**, va extraire les **noms, prénoms et numéro d'employé des employés travaillant dans la division RVN dans une classe anonyme**. Afficher le résultat dans txtRésultat. Résultat à obtenir:

```
Nom : Simard, Prénom : Élise, NoEmp : E00004
Nom : Mercure, Prénom : Renée, NoEmp : E00007
Nom : Lavigne, Prénom : Isidore, NoEmp : E00010
```

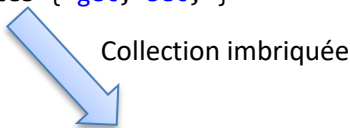
Utilisation de LINQ avec des collections imbriquées

Depuis le début, nous utilisons seulement des collections uniques. Dans cette section, nous allons traiter des collections imbriquées, c.-à-d. une collection qui contient une ou plusieurs collections. Voici un exemple avec une liste d'étudiants dont chaque étudiant possède plusieurs notes :

```
public class Etudiant
{
    // Propriétés auto implémentées.
    public string Prenom { get; set; }
    public string Nom { get; set; }
    public string Groupe { get; set; }
    public List<int> Notes { get; set; }
}

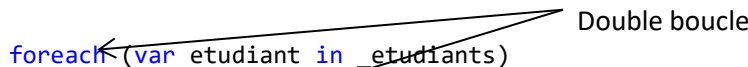
...

public static List<Etudiant> _etudiants = new List<Etudiant>()
{
    new Etudiant { Prenom = "Svetlana", Nom = "Omelchenko",
        Notes = new List<int> { 97, 92, 81, 60, 78 } },
    new Etudiant { Prenom = "Claire", Nom = "O'Donnell",
        Notes = new List<int> { 75, 84, 91, 39, 62 } },
    ...
}
```



Exemple #1 : Afficher les notes de chaque étudiant (sans LINQ).


```
foreach (var etudiant in _etudiants)
{
    MessageBox.Show("Etudiant : " + etudiant.Nom + " " + etudiant.Prenom);
    foreach (var note in etudiant.Notes)
        MessageBox.Show("Note : " + note);
}
```



Exemple #2 : Afficher toutes les notes (avec doublons) en ordre croissant.

```
var req = from etudiant in _etudiants
          from note in etudiant.Notes
          orderby note ascending
          select note;

foreach (var note in req)
    MessageBox.Show("Note : " + note);
```



Éliminer les doublons avec Distinct()

Il existe une fonction qui permet de supprimer les doublons contenus dans le résultat d'une requête. Cette méthode se nomme **Distinct**. Exemple :

```
var req = from etudiant in _etudiants
          from note in etudiant.Notes
          orderby note ascending
          select note;
```

```
foreach (var note in req.Distinct())
    MessageBox.Show("Note : " + note);
```

Vous pouvez également spécifier `req = req.Distinct();` devant le `foreach`.

Exercice 13. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_etudiants**, va trouver le **nom** et **prénom** des étudiants où l'étudiant possède au moins une note **plus grande** que 92. **Utiliser une classe anonyme.**

- 1) Trier les résultats d'abord par nom, puis par prénom.
- 2) Retirer les doublons.
- 3) Afficher le résultat dans txtRésultat.

Résultat :

```
Étudiant : Adams Terry
Étudiant : Fakhouri Fadi
Étudiant : Feng Hanying
Étudiant : Garcia Cesar
Étudiant : Mortensen Sven
Étudiant : Omelchenko Svetlana
Étudiant : Tucker Michael
Étudiant : Zabokritski Eugene
```

Exercice 14. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_patients**, va permettre d'afficher pour les patients (Nom, prénom et NAM) qui ont été vaccinés (dans Remarque) pour le **H1N1** avec leurs **listes de visites** respectives. Afficher le résultat dans txtResultat. Résultat :

```
Patient : Roger Côté COTR50081499
Date de visite : 1980-02-26
Date de visite : 1985-07-01
Date de visite : 1988-09-12
Patient : Réjean Savard SAVR77022899
Date de visite : 1978-02-13
Date de visite : 1979-09-01
Date de visite : 2009-04-04
Date de visite : 2010-07-21
```


Les possibilités du select

Vous pouvez extraire les informations d'une collection de différentes façons grâce au *select*. En voici quelques-unes :

Exemple #1 : En retournant l'objet complet.

```
select etudiant;
```

Le **select** crée un **objet** Etudiant, et la collection sera une **IEnumerable<Etudiant>**. Ainsi vous avez accès à toutes les propriétés et/ou méthodes.

Exemple #2 : En retournant une chaîne de caractères.

```
select etudiant.Nom + ", " +  
       etudiant.Prenom +  
       " Note : " + note;
```

Le **select** crée un **string** « Nom, Prenom Note : 99 », et la collection sera une **IEnumerable<string>**.

Exemple #3 : En retournant une classe anonyme.

```
select new  
{  
    Etudiant = etudiant.Nom + ", " + etudiant.Prenom,  
    Note = note  
};
```

Ici, le **select** crée une **classe anonyme** avec les **propriétés** Etudiant et Note. La collection sera **IEnumerable<type_anonyme>**. Cela permet de seulement recueillir les propriétés que l'on veut utiliser dans la requête.

Exercice 15. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_employes**, va extraire le **NoEmp**, la **division**, le **NoSucc** et l'**ancienneté** dans une classe anonyme pour les employés travaillant dans les succursales **0001** ou **0002**. Afficher le résultat dans txtRésultat.

Résultat :

```
NoEmp : E00001, Division : MTL, NoSucc : 0002, Anc. : 10  
NoEmp : E00003, Division : RVS, NoSucc : 0001, Anc. : 4  
NoEmp : E00004, Division : RVN, NoSucc : 0001, Anc. : 20  
NoEmp : E00008, Division : RVS, NoSucc : 0002, Anc. : 8  
NoEmp : E00009, Division : MTL, NoSucc : 0002, Anc. : 31  
NoEmp : E00010, Division : RVN, NoSucc : 0001, Anc. : 4
```

Utilisation du mot-clé *let*

Il existe un autre mot-clé qui permet de simplifier nos requêtes, soit [let](#). Cette instruction **permet de déclarer et d'affecter une variable locale à la requête**. Le *let* doit être utilisé après un **from** et avant un **select**.

Exemple #1 : Trouver le nom, prénom et la moyenne des étudiants qui ont une moyenne ≥ 90 , triés par moyenne descendante.

```
// SANS l'instruction let.
var req = from etudiant in _etudiants
          where etudiant.Notes.Average() >= 90
          orderby etudiant.Notes.Average() descending
          select etudiant.Nom + ", " + etudiant.Prenom +
                " : moy = " + etudiant.Notes.Average();

// AVEC l'instruction let.
var req = from etudiant in _etudiants
          let moyenne = etudiant.Notes.Average()
          where moyenne >= 90
          orderby moyenne descending
          select etudiant.Nom + ", " + etudiant.Prenom + " : moy = " + moyenne;

foreach (string sResultat in req)
    MessageBox.Show("Résultat : " + sResultat);
```

Dans cet exemple, la requête SANS l'instruction *let* nécessite plusieurs appels à la méthode `Average()`. Ce qui n'est pas performant. Donc pour éviter ce multiple appel, l'utilisation de *let* pour déclarer une variable est de mise. La variable peut donc être utilisée partout dans la requête. Une fois initialisée avec une valeur, la variable ne peut pas être utilisée pour stocker une autre valeur. Toutefois, si la variable contient un type requête, elle peut être interrogée.

Exercice 16. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection `_etudiants`, va extraire le **Nom**, le **prénom**, la **note minimale** obtenue et la **note maximale** obtenue dans une **classe anonyme** pour les étudiants ayant un écart de plus de **30 points** entre la **note maximale et minimale**. Afficher le résultat dans `txtRésultat`. Résultat :

```
Étudiant : Omelchenko Svetlana, Min : 60, Max : 97
Étudiant : O'Donnell Claire, Min : 39, Max : 91
Étudiant : Garcia Debra, Min : 35, Max : 91
Étudiant : Zabokritski Eugene, Min : 60, Max : 96
```

Utilisation d'une méthode/fonction personnelle

Vous pouvez développer et utiliser vos fonctions personnelles dans une requête LINQ. Un exemple en créant nous-mêmes une fonction Moyenne :

```
public double Moyenne(List<int> notes)
{
    double dSomme = 0.0d;

    foreach (int iNote in notes)
        dSomme += iNote;

    return dSomme / notes.Count;
}
...
var req = from etudiant in _etudiants
          let moyenne = Moyenne(etudiant.Notes)
          where moyenne >= 90
          orderby moyenne descending
          select etudiant.Nom + ", " + etudiant.Prenom + " : moy = " + moyenne;

foreach (string sResultat in req)
    MessageBox.Show("Résultat : " + sResultat);
...
```

Dans certaines situations, il est conseillé d'exécuter les méthodes en dehors de la requête LINQ et d'utiliser une variable pour y recevoir le résultat. Par la suite, il suffit d'utiliser cette variable pour alimenter la requête LINQ. Le prochain exercice est un bel exemple.

Exercice 17. Coder l'exemple précédent.
--

Autres - Exercices

Exercice 18. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_etudiants**,

1) Va calculer la moyenne à partir de toutes les notes de l'ensemble des étudiants (une requête) → Moyenne générale

2) Pour chaque note de l'ensemble de toutes les notes, calculer (2^e requête) et afficher : la note, son **écart avec la moyenne générale** et la **moyenne générale**. **Trier** en ordre d'écart, **éliminer les décimales et les doublons**. Afficher le résultat dans txtRésultat. Résultat :

```
Note : 35, écart : -46, Moyenne : 81
Note : 39, écart : -42, Moyenne : 81
Note : 60, écart : -21, Moyenne : 81
Note : 62, écart : -19, Moyenne : 81
Note : 65, écart : -16, Moyenne : 81
Note : 66, écart : -15, Moyenne : 81
Note : 67, écart : -14, Moyenne : 81
Note : 68, écart : -13, Moyenne : 81
Note : 70, écart : -11, Moyenne : 81
Note : 72, écart : -9, Moyenne : 81
Note : 75, écart : -6, Moyenne : 81
Note : 76, écart : -5, Moyenne : 81
Note : 78, écart : -3, Moyenne : 81
Note : 79, écart : -2, Moyenne : 81
Note : 80, écart : -1, Moyenne : 81
Note : 81, écart : 0, Moyenne : 81
Note : 82, écart : 1, Moyenne : 81
Note : 83, écart : 2, Moyenne : 81
Note : 84, écart : 3, Moyenne : 81
Note : 85, écart : 4, Moyenne : 81
Note : 86, écart : 5, Moyenne : 81
Note : 87, écart : 6, Moyenne : 81
Note : 88, écart : 7, Moyenne : 81
Note : 89, écart : 8, Moyenne : 81
Note : 90, écart : 9, Moyenne : 81
Note : 91, écart : 10, Moyenne : 81
Note : 92, écart : 11, Moyenne : 81
Note : 93, écart : 12, Moyenne : 81
Note : 94, écart : 13, Moyenne : 81
Note : 96, écart : 15, Moyenne : 81
Note : 97, écart : 16, Moyenne : 81
Note : 99, écart : 18, Moyenne : 81
```

Exercice 19. Dans l'événement Click du bouton associé, ajouter le code qui permet de trouver les étudiants qui ont un nom de **famille identique**, et éliminer les doublons. Afficher le résultat dans txtRésultat. Résultat :

```
Nom : Feng, Prénom : Karen  
Nom : Feng, Prénom : Hanying  
Nom : Garcia, Prénom : Debra  
Nom : Garcia, Prénom : Hugo  
Nom : Garcia, Prénom : Cesar  
Nom : Tucker, Prénom : Michael  
Nom : Tucker, Prénom : Lance
```

Obtenir les N premiers éléments grâce à Take()

Il existe une fonction qui permet de retourner le nombre désiré d'éléments contenu dans le résultat d'une requête (à partir du 1^{er}) selon le nombre spécifié en paramètre. Il s'agit de la fonction [Take\(\)](#).

Exemple : Parmi la liste des mots, je souhaite obtenir les 3 premiers mots.

```
string[] asMots = { "Bleu", "Rouge", "Vert", "Jaune", "Blanc", "Noir",  
                  "Orange", "Rose" };
```

```
var req = asMots.Take(3); // Obtenir les trois premiers éléments
```

```
foreach(string sMot in req)  
    MessageBox.Show("Le mot est : " + sMot);
```

```
Le mot est : Bleu  
Le mot est : Rouge  
Le mot est : Vert
```

Exercice 20. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_employees**, va extraire le **nom**, le **prénom**, la **division** et l'**ancienneté** dans **une classe anonyme** de **5 employés ayant le plus d'ancienneté**. Afficher le résultat dans txtRésultat. Résultat :

```
Nom : Turcotte, Prénom : Roger, Division : MTL, Anc. : 31  
Nom : Simard, Prénom : Élise, Division : RVN, Anc. : 20  
Nom : Savard, Prénom : Marc, Division : MTL, Anc. : 15  
Nom : Côté, Prénom : Luc, Division : MTL, Anc. : 11  
Nom : Roy, Prénom : Luc, Division : MTL, Anc. : 10
```

Obtenir tous les éléments sauf les N premier grâce à Skip()

Il existe une autre fonction qui, cette fois-ci, permet d'exclure les N premiers éléments d'une collection. Il s'agit de [Skip\(\)](#).

Exemple : Parmi la liste des mots, je souhaite exclure les 5 premiers mots.

```
string[] asMots = { "Bleu", "Rouge", "Vert", "Jaune", "Blanc", "Noir", "Orange",  
                  "Rose" };  
  
var req = asMots.Skip(5); // Exclure les cinq premiers éléments  
  
foreach (string sMot in req)  
    MessageBox.Show("Le mot est : " + sMot);
```

Résultat :

```
Le mot est : Noir  
Le mot est : Orange  
Le mot est : Rose
```

Exercice 21. Dans l'événement Click du bouton associé, ajouter le code qui, à partir de la collection **_ventes**, va extraire les **montants supérieur ou égal à la moyenne en ordre descendant**. **Ne pas prendre les 10 premiers montants**. Afficher le résultat dans la txtRésultat. Résultat :

```
Vente : 3261,82 $  
Vente : 3165,32 $  
Vente : 3023,17 $  
Vente : 2904,26 $  
Vente : 2806,66 $  
Vente : 2792,63 $  
Vente : 2667,28 $  
Vente : 2630,52 $  
Vente : 2606,12 $  
Vente : 2573,55 $  
Vente : 2567,80 $
```

MSDN offre une page Web contenant beaucoup d'exemples sur l'utilisation de LINQ. Voici le lien : [101 LINQ Samples](#).

Bibliographie

Mastriani, R. (2013, 04 05). PowerPoint - LINQ. Saint-Hyacinthe, QC, Canada.

MSDN. (2013, 01 01). *Collections (C# et Visual Basic)*. Consulté le 04 04, 2013, sur MSDN - Visual Studio: <http://msdn.microsoft.com/fr-fr/library/vstudio/ybcx56wz.aspx>