Laboratoire 6 : LINQ

Objectifs

- Se familiariser avec les concepts de la syntaxe LINQ
- Comprendre et utiliser des lambda expressions
- Utiliser LINQ et les lambda expressions pour résoudre des problèmes de ciblage, de transformation et d'agrégation
- Appliquer toutes ces connaissances dans le cadre d'exercices pratiques

Exercice 1 : LINQ et expressions lambdas

LINQ (Language INtegrated Queries) est un groupe de syntaxes qui ont été ajoutées au langage C# lors de la sortie de sa version 3.0 (en 2007) et qui sont fortement inspirées des langages de programmation fonctionnels. Ces syntaxes s'inscrivent dans la même grande famille que les fonctions de haut de niveau en Javascript (comme l'exemple suivant).

```
let carréDesNombresSous100FinissantPar7 =
  nombresDe1A100.filter(x => x % 10 == 7).map(x => x*x);
```

Les syntaxes LINQ visent principalement à aider à la manipulation de collections de données : sélectionner un groupe de données selon certains critères, réorganiser des données (les trier par exemple), les traduire/transformer en un autre format, etc.

Les ajouts regroupés sous le nom de LINQ se divisent en deux catégories : les syntaxes **basées sur les méthodes** et les syntaxes **imitant le SQL**. Le SQL (Structured Query Language) est un langage informatique[1] permettant d'écrire des requêtes (queries) pour examiner le contenu d'une base de données (là aussi, pour sélectionner des données, les organiser et éventuellement les transformer). Comme le langage SQL est vu au 2e quadrimestre, nous n'examinerons que les syntaxes basées sur les méthodes.

Étape 1 : un cadre de travail

Avant de manipuler des exemples de LINQ, il faut créer des collections de données à examiner. Dans un nouveau projet C#, placez le code suivant dans la fonction principale, en vous assurant que vous comprenez bien la structure de chacune des collections définies.

```
// Des tableaux
int[] nombres = { 7, 9, 13, 16, 21, 22, 29, 36, 44, 47, 51, 55, 64, 71,
77, 81, 99 };

char[] voyelles = { 'a', 'e', 'i', 'o', 'u', 'y' };

// Une collection simple via List
List<string> prénoms = new List<string>();

string[] prénomsTmp = { "Anatole", "Bergamote", "Cunégonde", "Doriphore",
"Fernande", "Gustave", "Hermione", "Isidore", "Mathilde", "Ophélie",
"Quasimodo", "Raistlin", "Wilfred", "Zénobe" };

foreach (string prénom in prénomsTmp)
  prénoms.Add(prénom);

// Un tableau associatif via Dictionary
Dictionary<int, List<string>> dico = new Dictionary<int, List<string>>();
```

[1] Il ne s'agit pas d'un langage de programmation : son but n'est pas de permettre l'écriture de programmes (au même titre que les langages informatiques tels que HTML et CSS qui, eux non plus, ne sont pas des langages de programmation).

```
string[] nombresFr = { "un", "deux", "trois", "quatre", "cinq", "six",
"sept", "huit", "neuf", "dix" };
string[] nombresEn = { "one", "two", "three", "four", "five", "six",
"seven", "eight", "nine", "ten" };

for (int i = 1; i < 10; i++) {
  dico[i] = new List<string>();
  dico[i].Add(nombresFr[i - 1]);
  dico[i].Add(nombresEn[i - 1]);
}
```

Pour pouvoir réaliser des tests et afficher le contenu d'une collection, ajoutez la définition de méthode suivante à la classe principale.

```
static string Affichage(IEnumerable<string> valeurs)
{
    StringBuilder res = new StringBuilder();
    res.Append("[");
    bool premier = true;
    foreach (string val in valeurs)
    {
        if (!premier)
            res.Append(", ");
        else
            premier = false;
        res.Append(val);
    }
    res.Append("]");
    return res.ToString();
}
```

Notez l'utilisation de la classe StringBuilder au lieu de string (à ce moment, vous devriez savoir pourquoi). Lisez le code et tentez de prévoir le résultat de l'instruction suivante (que vous pourrez ajouter à la méthode principale).

```
Console.WriteLine(Affichage(prénoms));
```

Étape 2 : premières manipulations via LINQ

LINQ rajoute une bonne quarantaine de méthodes utilisables sur toutes les collections énumérables. Une d'entre elles est la méthode `Where`, qui correspond à la méthode `filter` de Javascript. Pour information, « filter » est le nom standard de cette opération dans les langages fonctionnels alors que « where » est le mot-clef utilisé en SQL.

La méthode `Where` (tout comme `filter`) prend comme argument un prédicat. D'un point de vue mathématique (cours d'IG1), un prédicat $P\langle x\rangle$ est une propriété dont la véracité (vrai ou faux) dépend de la valeur qu'on donne à x (par exemple « x est pair »). En informatique, un prédicat est une fonction dont le résultat est un booléen.

Par exemple, la fonction qui reçoit une chaîne de caractères s et indique si s comporte exactement 7 lettres est un prédicat : elle indique vrai dans le cas de « Anatole » et faux dans le cas de « Bergamote ».

C# autorise l'écriture de lambda expressions (ou littéraux de fonctions) avec une syntaxe similaire à celle de Javascript. Par exemple, la fonction citée ci-dessus s'écrirait comme suit.

```
s => s.Length == 7
```

Comme il s'agit d'un prédicat, on peut l'utiliser comme argument de `Where`. Ainsi, l'expression `prénoms.Where(s => s.Length == 7)` est tout à fait correcte ; elle décrit la collection obtenue en partant de la liste `prénoms` et en ne conservant que les prénoms de 7 lettres.

Vérifiez tout cela en ajoutant l'instruction suivante à la méthode principale.

```
Console.WriteLine(Affichage(prénoms.Where(s => s.Length == 7)));
```

Elle devrait produire l'affichage suivant.

```
[Anatole, Gustave, Isidore, Ophélie, Wilfred]
```

Complétez la méthode principale pour afficher…

1. les prénoms qui comportent un « i » minuscule (les chaînes de caractères implémentant `IEnumerable<char>`, elles possèdent les méthodes de cette interface !) ;
2. les prénoms qui, dans l'ordre alphabétique, viennent avant « C » ou après « Sharp » ;
3. les prénoms dont la dernière lettre est autre chose que « e » (pensez à la méthode `Last`).

Étape 3 : quelques méthodes LINQ de plus…

Le tableau ci-dessous présente quelques autres méthodes LINQ que vous pourrez utiliser dans les exercices qui suivent.

Méthode	Argument	Effet
`Where`	prédicat p	Ne conserve que les éléments qui vérifient le prédicat
`Take`	entier n	ne conserve que les n premiers éléments de la collection
`Skip`	entier n	Ignore les n premiers éléments de la collection
`TakeWhile`	prédicat p	Ne conserve que les éléments de tête qui vérifient le prédicat (on ignore le premier élément qui ne le vérifie pas ainsi que les suivants)
`SkipWhile`	prédicat p	Ignore les éléments de tête qui ne vérifient pas le prédicat (ne conserve que le premier élément qui ne le vérifie pas et les suivants)
`Distinct`	aucun	Ignore les répétitions parmi les éléments
`Select`	transformation f	Remplace chaque élément x par le résultat obtenu en lui appliquant la transformation f

Par exemple, la syntaxe `prénoms.Select(s => s.ToUpper())` donne une collection où chacun des prénoms est remplacé par sa version en majuscules. Ajoutez son affichage dans la méthode principale pour voir le résultat.

Notez qu'un des grands avantages de la syntaxe par méthodes est qu'on peut enchaîner les méthodes. Ainsi, on peut obtenir la collection des prénoms de 7 lettres écrits en majuscules soit via le code

```
IEnumerable<string> prénoms7Lettres = prénoms.Where(s => s.Length == 7);
IEnumerable<string> résultat = prénoms7Lettres.Select(s => s.ToUpper());
```

soit en enchaînant les méthodes.

```
IEnumerable<string> résultat =
   prénoms.Where(s => s.Length == 7).Select(s => s.ToUpper());
```

Complétez votre code pour effectuer les tâches suivantes.

1. Afficher la liste des prénoms amputés de leur initiale (natole, ergamote, unégonde…).

2. Afficher le code ASCII des voyelles (on peut obtenir le code ASCII d'un caractère en le transtypant [= faire un « cast/casting »] en entier). Notez que, pour pouvoir utiliser la méthode `Affichage`, vous devriez avoir comme résultat une collection de chaînes de caractères [pas d'entiers, ni de caractères]).

3. On voudrait pouvoir afficher également les tableaux d'entiers, voire n'importe quelle collection `IEnumerable<int>`. Surchargez la méthode statique `Affichage` avec une seconde version qui prend un tel argument. Vérifiez votre code en affichant le tableau `nombres`.

   Réalisez l'implémentation en une seule ligne : en effet, il suffit de convertir les entiers de la collection en chaînes de caractères pour pouvoir faire appel à la première version de `Affichage` !)

4. Afficher les nombres (du tableau `nombres`) qui sont pairs.

5. Afficher les nombres (du tableau `nombres`) mais en commençant avec le premier nombre qui est un multiple de 11.

6. Afficher les derniers chiffres des nombres du tableau `nombres` (7, 9, 3, 6, 1, 2…).

7. Faire de même que le 6 mais sans répéter les chiffres (pensez à enchaîner deux méthodes).

8. Afficher les prénoms dans lesquels on trouve au moins une lettre qui apparaît deux fois (comme le « o » dans Quasimodo ou Doriphore). C'est une bonne occasion pour utiliser une expression régulière en C#… Vous pouvez chercher vous-mêmes comment faire dans la documentation en ligne ou utiliser les indications qui suivent.

En C#, les expressions régulières sont gérées par le type `Regex`. (qui se trouve dans le namespace `System.Text.RegularExpressions`). Pour définir un objet de ce type, on peut utiliser le constructeur avec un seul argument (l'expression régulière) ou avec deux arguments (en ajoutant les options d'interprétation – voir cours de Javascript). Voici quelques exemples.

```
Regex rMotsDe3Lettres = new Regex ("...");
Regex rMotFinissantParVoyelle = new Regex (".*[aeiouy]");
Regex rPrénomAvecUnA = new Regex (".*[aA].*");
Regex rPrénomAvecUnAv2 = new Regex (".*a.*", RegexOptions.IgnoreCase);
```

Les options d'interprétation prennent la forme de constantes (en fait, il s'agit d'éléments d'une énumération binaire) de la forme `RegexOptions.nom` qu'on combine avec l'opérateur `|`. Par exemple : `RegexOptions.IgnoreCase | RegexOptions.Multiline`.

Pour tester si une chaîne de caractères `s` correspond (en entier) à une expression régulière `r`, on utilise la méthode `IsMatch` définie dans `Regex` : `r.IsMatch(s)`.

Étape 4 : Un dictionnaire et encore quelques méthodes LINQ…

Au début de cet exercice, on a défini un dictionnaire. Dans un premier temps, surchargez à nouveau la méthode Affichage pour qu'elle puisse l'accepter comme argument.

```
static string Affichage(Dictionary<int, List<string>> collection)
```

L'implémentation peut à nouveau se faire en une seule ligne en utilisant LINQ. Les dictionnaires du type en question peuvent être traités comme des collections de paires rassemblant une clef et une valeur.

Les éléments de cette collection sont d'un type un peu particulier appelé `KeyValuePair<int, List<string>>` ; le nom importe peu mais il est intéressant de savoir que chacun de ces éléments possèdent des propriétés appelées `Key` et `Value` permettant d'obtenir respectivement la clef (ici, un entier) ou la valeur (ici, une liste de chaînes de caractères).

Grâce à ces informations, vous devriez savoir comment transformer le dictionnaire en en `IEnumerable<string>` sur lequel on peut utiliser la première version d'`Affichage`. Visez un affichage correspondant à l'exemple suivant.

```
[1->[un, one], 2->[deux, two], 3->[trois, three], 4->[quatre, four], 5-
>[cinq, five], 6->[six, six], 7->[sept, seven], 8->[huit, eight], 9-
>[neuf, nine]]
```

Toutes les méthodes LINQ abordées jusqu'ici produisaient une nouvelle collection (après avoir transformé les éléments ou après en avoir ignoré certains). Il existe d'autres méthodes, qui ont pour résultat une valeur unique. Le tableau suivant en présentent quelques-unes.

Méthode	Argument	Effet
`First`	aucun	Donne le premier élément de la collection

`First`	prédicat p	Donne le premier élément de la collection qui vérifie le prédicat p
`Last`	aucun	Donne le dernier élément de la collection
`Last`	prédicat p	Donne le dernier élément de la collection qui vérifie le prédicat p
`ElementAt`	nombre n	Donne le n^{e} élément de la collection
`Count`	aucun	Donne le nombre d'éléments dans la collection
`Count`	prédicat p	Donne le nombre d'éléments qui vérifient le prédicat p
`Average`	aucun	Donne la moyenne des éléments (si applicable)
Aussi :* `Sum` *(somme),* `Min` *(minimum) et* `Max` *(maximum).		
`Contains`	élément e	Indique si la collection contient l'élément e
`Any`	prédicat p	Indique si la collection contient au moins un élément qui vérifie le prédicat p
`All`	prédicat p	Indique si tous les éléments de la collection vérifient le prédicat p

1. Afficher le nombre de prénoms qui comportent exactement 7 lettres (réponse : 5).

2. Indiquer (afficher un booléen) si tous les prénoms ont bien une longueur de 7 lettres ou plus.

3. Indiquer (afficher un booléen) si les 7 premiers prénoms de la liste ont bien une longueur de 7 lettres ou plus (en ajoutant une méthode au code précédent).

4. Afficher la longueur moyenne des prénoms (réponse : 7.7857).

5. Afficher la moyenne des nombres impairs du tableau `nombres` (réponse : 46.66).

6. Afficher la longueur du plus long prénom de `prénoms`.

7. Afficher les nombres de `dico` qui se traduisent dans au moins une langue par un mot de trois lettres (réponse : 1, 2, 6).

8. Afficher les nombres de `dico` qui se traduisent dans les deux langues par des mots de même longueur (réponse : 3, 5, 6, 9).

9. Afficher le nombre de lettres à écrire pour chacun des nombres de `dico` si on veut les écrire en anglais et en français (réponse : 5, 7, 10, 10, 8, 6, 9, 9, 8).

Il existe bien d'autres méthodes LINQ qui n'ont pas été abordées ici. Parmi les plus importantes (dont vous pouvez consulter la document en ligne) figurent `OrderBy` (pour trier les éléments selon un critère) et diverses méthodes de conversion telles que `ToArray`, `ToList` et `ToDictionnary`.

Étape 5 : les méthodes LINQ un peu plus en profondeur…

Dans la méthode principale, ajoutez le code permettant de réaliser les opérations suivantes les unes après les autres.

- Déclarer la variable entière `longueur` et l'initialiser à 6.
- Déclarer la variable `IEnumerable<string> prénoms6Lettres` et y placer la collection obtenue à partir de `prénoms` en ne conservant que les prénoms de longueur `longueur`.
- Afficher `prénoms6Lettres` via la méthode `Affichage`.
- Enlever de la collection `prénoms` le prénom « Zénobe ».
- Afficher à nouveau `prénoms6Lettres` via la méthode `Affichage`.
- Ajouter à la collection prénoms les prénoms « Jordan », « Zébulon » et « Agatha ».
- Afficher une dernière fois `prénoms6Lettres` via la méthode `Affichage`.

Exécutez le code et observez le résultat. Comprenez-vous ce qui s'est passé ?

Retournez dans votre code et ajoutez à la suite des lignes précédentes les deux lignes ci-dessous.

```
longueur = 7;
Console.WriteLine(Affichage(prénoms6Lettres));
```

Lancez à nouveau le programme… et observez le résultat.

La raison de ce comportement qui peut être étonnant à première vue est la suivante : lorsqu'on rédige une requête LINQ (c'est-à-dire l'application d'une ou de plusieurs méthodes LINQ dans le cadre de ce laboratoire), celle-ci n'est pas toujours exécutée immédiatement.

Par exemple, si on dispose d'une collection de 50 000 nombres et qu'on rédige une requête LINQ pour ne plus considérer que les nombres qui sont pairs, l'exécution ne va pas s'interrompre pour laisser le temps à l'ordinateur de vérifier chacun des 50 000 nombres en question. En fait, tant qu'on n'utilisera pas la nouvelle collection (par exemple en demandant d'afficher son contenu), la requête ne générera aucun calcul.

En fait, dans tous les cas où c'est possible, les calculs sont retardés. Ainsi, si on rencontre plus loin une boucle `foreach` sur la nouvelle collection (celle qui ne contient que les nombres pairs), le code ne va pas examiner les 50 000 éléments d'un seul coup. Pour la première itération du `foreach`, il n'a besoin que du premier nombre pair… il va donc parcourir le début de la grosse collection et s'arrêter dès qu'il trouve le premier nombre pair. Là encore, le principe est d'en faire le moins possible à l'avance et d'attendre qu'on ait vraiment besoin des calculs pour les réaliser.

Les deux dernières lignes de code citées ci-dessus offrent un autre exemple du même principe : la variable `longueur` elle non plus n'est évaluée immédiatement lors de la définition de `prénoms6Lettres` ; on attend d'y être obligé (par exemple pour l'affichage) pour l'évaluer.

Ainsi, dans la variable `prénoms6Lettres`, on ne stocke pas vraiment les éléments de la nouvelle collection mais la *manière de calculer* cette nouvelle collection, et c'est cette

manière de calculer (en gros, le texte du code) qu'on évalue quand c'est nécessaire. Ce principe d'exécution retardée (delayed execution) se retrouve dans de nombreux langages et peut engendrer des erreurs parfois difficiles à cerner.

D'un autre côté, bien utilisé, il peut aussi faciliter certaines tâches… ainsi, si la collection de départ est en fait une base de données, le fait de recalculer le résultat chaque fois que c'est nécessaire a plutôt des conséquences positives : cela permet de définir une requête au début du programme puis d'être certain que, chaque fois qu'on l'utilisera, on se basera bien sur la version actuelle des données plutôt que sur l'état de la base de données au début de l'exécution.

Exercice 2 : LINQ et les espions

Reprenez le code de l'agence de services secrets, et plus particulièrement l'exemple donné en fin de laboratoire précédent.

Ajoutez à votre code les méthodes permettant d'effectuer les recherches suivantes (utilisez bien sûr les méthodes LINQ). N'hésitez pas à ajouter des méthodes auxiliaires pour mieux découper votre code, ou des propriétés (ou méthodes) pour permettre l'accès à des collections privées. Vous pouvez également utiliser des boucles `foreach` pour parcourir une collection créée à l'aide de méthodes LINQ. Assurez-vous que votre code est propre et lisible.

1. Dans la classe `Agent`, la méthode `ÂgeMoyenContacts()` renverra l'âge moyen des contacts de l'agent.

Résultats attendus

```
c1aa0 : Natalie, Mills, moyenne : 30.6666666666667
c1aa1 : Peyton, Davidson, moyenne : 36.5
c1aa2 : Landon, Alexander, moyenne : 31.6666666666667
c1aa3 : Hannah, Mary, moyenne : 26
c1aa4 : Julian, Hicks, moyenne : 36.5
```

2. Dans la classe `Agent`, la méthode `NbContactsPlusJeunes()` renverra le nombre de contacts qui sont strictement plus jeunes que l'agent en question.

Résultats attendus

```
c1aa0 : Natalie, Mills, nbPlusJeunes : 1
c1aa1 : Peyton, Davidson, nbPlusJeunes : 0
c1aa2 : Landon, Alexander, nbPlusJeunes : 1
c1aa3 : Hannah, Mary, nbPlusJeunes : 2
c1aa4 : Julian, Hicks, nbPlusJeunes : 1
```

3. Dans la classe `Chapitre`, la méthode `AfficheAgentsPourDomaine(domaine)` affichera le nom de tous les agents associés au chapitre et possédant au moins un contact lié au domaine indiqué.

Résultats attendus

```
Recherche pour domaine mafia:
  Peyton, Davidson
  Landon, Alexander
  Julian, Hicks
Recherche pour domaine vente d'armes:
  Natalie, Mills
  Peyton, Davidson
  Landon, Alexander
Recherche pour domaine espionnage technique:
  Natalie, Mills
  Landon, Alexander
  Hannah, Mary
```

4. Dans la classe `Chapitre`, la méthode `AfficheContacts(âgeMin, âgeMax)` affichera le nom de tous les contacts associés à des agents du chapitre et dont l'âge se trouve dans la fourchette indiquée.

Réponse attendue

```
Contacts entre 15 et 30 ans :
  Carter, Howard
  Kaylee, Owens
  Benjamin, Duncan
  Hannah, Murphy
  Brooklyn, Page
Contacts entre 40 et 50 ans :
  Evelyn, Cox
  Aiden, Hopkins
  Christian, Thomas
```

5. Dans la classe Chapitre, la méthode `RechercheSafehouses(texte)` affichera le nom de code et l'adresse de toutes les safehouses dont l'adresse contient le texte donné.

Réponse attendue

```
Recherche du texte Chemin :
  Astrance (Chemin de Bateau, 234)
  Saponaire (Chemin Masy, 128)
  Digitale (Chemin de la Taille Boha, 331)
Recherche du texte 7 :
  Buglose (Rue Tabaral, 357)
  Potentille (Place Sainte-Sévère, 277)
Recherche du texte Ta :
  Buglose (Rue Tabaral, 357)
  Digitale (Chemin de la Taille Boha, 331)
```

6. Dans la classe Chapitre, la méthode `InfoMissions()` affichera, pour chaque mission, son nom de code et la liste des noms de toutes les personnes concernées (agents + contacts) triée par ordre alphabétique.

Réponse attendue

```
Mission Chicorée : Addison, Phillips; Daniel, Mc; Elena, Watts; Grace,
Bryan; Isaiah, Island; Lincoln, Bros; Mason, Jackson
Mission Gentiane : Daniel, Williamson; Evelyn, Page; Gabriel, Watts;
Grayson, Perry; Liam, Stanley; Noah, Robert; Nora, Bruce; Oliver,
Sherman; Owen, Fleming; Sophia, Dixon; William, Clay
Mission Clématite : Abigail, Morgan; Adeline, Ann; Andrew, Bell; Carter,
Kelly; Dylan, Blake; Dylan, Lee; Elena, Watts; Emily, May; Grace, Bryan;
Isaiah, Crawford; Layla, Washington; Lincoln, Bros; Lincoln, Graham;
Mason, Harrison; Michael, Miles
Mission Lys : Addison, Knight; Addison, Reynolds; Aiden, Bryan;
Charlotte, Hunt; Ella, Fletcher; Emily, Roberts; Hailey, Fletcher;
Hailey, Graham; Hailey, Hill; Jayce, Nichols; Landon, Ellis; Mackenzie,
```

```
Kerr; Nicholas, Davis; Nicholas, Hunter; Oliver, Wood; Scarlett, Lee;
Victoria, Kerr
Mission Joubarde : Caleb, Sanders; Camilla, Adams; Dylan, Saunders;
Elena, Watts; Ethan, Wheeler; Grace, Bryan; Jayden, Ellis; Lincoln, Bros;
Lucas, Barrett; Mackenzie, Coleman; Mackenzie, Henry; Mason, Brown;
Nathan, Dunn; Peyton, Walker
```