LINQORNE 🦄

La librairie LINQ est limitée sur certains aspects (et trop standard ;-)) et il est temps de réaliser une version unique et enrichie 💶

Base

Créer un projet librairie nommé `Linqorne`.

ForEach ➿

Commençons par définir une fonction `Loop` qui mimique le `ForEach` qui n'est que disponible sur les `List` et pas directement sur un `IEnumerable`.

▼ Aide si nécessaire

```
public static class Linqorne
{
    public static void Loop<TSource>(this IEnumerable<TSource> subject,Action<TSource> action)
    {
        foreach (var item in subject)
        {
            action(item);
        }
    }
}
```

Quelques explications

Généricité

Comme `Linq`, pour que la librairie fonctionne sur tous les types possibles (Collections de n'importe quel élément), on a besoin de la `généricité`, dont voici quelques explications en lien avec le code présenté :

`TSource` :

- `TSource` est un **paramètre de type générique**. Cela signifie que cette méthode ne fonctionne pas avec un type particulier, mais avec un type qui sera déterminé au moment de l'utilisation de la méthode.
- Le type `TSource` est spécifié avec la syntaxe générique dans la déclaration de la méthode : `Loop<TSource>`. Cela rend la méthode générique, permettant d'utiliser différents types d'éléments sans réécrire la méthode pour chaque type.
- Par exemple, `TSource` pourrait être un `int`, `string`, `Person`, ou n'importe quel autre type d'objet.

Paramètres

1. `IEnumerable<TSource>` :
   - Le paramètre `subject` est de type `IEnumerable<TSource>` . Cela signifie qu'il s'agit d'une collection de plusieurs éléments de type `TSource` .
   - `IEnumerable` est une interface qui représente une collection qui peut être énumérée (ou itérée) sur n'importe quel type d'objet.
2. `Action<TSource>` :
   - Le deuxième paramètre, `action` , est de type `Action<TSource>` . Cela signifie que c'est une déléguée qui prend un paramètre de type `TSource` et ne renvoie rien (void).
   - En d'autres termes, `Action<TSource>` représente une méthode ou un bloc de code qui prend un élément de type `TSource` et fait quelque chose avec cet élément.

Exécution du code :

Lorsque la méthode `Loop` est appelée, elle itère sur chaque élément de la collection `subject` , et pour chaque élément, elle exécute l' `action` spécifiée. Étant donné que la méthode est générique, elle peut être utilisée pour tout type de collection et toute action qui agit sur ce type.

Exemple d'utilisation :

```
List<int> numbers = new List<int> { 1, 2, 3, 4 };
numbers.Loop(n => Console.WriteLine(n)); // Affichera chaque nombre dans la console
numbers.Loop(Console.WriteLine); // Version écourtée du lambda
```

Dans cet exemple :

- `TSource` est `int` .
- `subject` est une liste d'entiers.
- `action` est une expression lambda qui affiche chaque nombre.

Find 🔎

La méthode `Where` de `Linq` donne l'impression de faire du `SQL` et il serait plus *humain* d'avoir une fonction `Find` à qui on passe la condition.

Compléter le code suivant :

```
public static IEnumerable<TSource> Find<TSource>(
        this IEnumerable<TSource> subject,
        Func<TSource,bool> isWanted)
    {
        var wanteds = new List<TSource>();

        //TODO

        return wanteds.ToImmutableArray();
    }
```

Map 🗺️

Pour converger vers la nomenclature standard, il serait judicieux de redéfinir le `Select` en `Map` .

Cette fois-ci, pas d'aide hormis la feuille presque blanche suivante (... et TODO à compléter) :

```
public static IEnumerable<...> Map<TSource,TTarget> (this ..., Func<TSourc,TTarget> convert)
{
   //TODO
}
```

Reduce 👟

Pas très original et toutefois intéressant pour le *drill*, il est temps de refaire un `Aggregate` nommé `Reduce` et ceci , bien sêr, **sans** réutiliser le `Aggregate` de `Linq` ...

```
public static ... Reduce ...
```

Statistiques

`Linq` offre `Min` , `Max` et `Average` mais il serait pertinent d'avoir d'autres outils statistiques comme la médiane et la covariance.

Min,Max,Average

Écrire ces fonctions pour `Linqorne` pour les types `int` , `double` et `decimal` .

Suppléments

Pour définir des méthodes qui calculent la médiane (`Median`) et la covariance (`Covariance`), nous devons comprendre les opérations mathématiques impliquées dans chacune de ces statistiques.

1. Calcul de la Médiane (`Median`)

La **médiane** est la valeur qui sépare la moitié inférieure des données de la moitié supérieure. Pour calculer la médiane, on peut :

- Trier la collection.
- Si le nombre d'éléments est impair, la médiane est le milieu de la collection.
- Si le nombre d'éléments est pair, la médiane est la moyenne des deux valeurs centrales.

> Sans utiliser `Linq` mais en utilisant tout ce qui est disponible dans `Linqorne` (par exemple Map...)

▼ Exemple d'implémentation de la méthode `Median`

```
public static double Median<TSource>(this IEnumerable<TSource> source, Func<TSource, double> converter)
{
    var sortedList = source.Map(converter).OrderBy(n => n).ToList();
    int count = sortedList.Count;

    if (count == 0)
    {
        throw new InvalidOperationException("La collection ne peut pas être vide");
    }

    if (count % 2 == 1) // Si le nombre d'éléments est impair
    {
        return sortedList[count / 2];
    }
    else // Si le nombre d'éléments est pair
    {
        double middle1 = sortedList[(count / 2) - 1];
        double middle2 = sortedList[count / 2];
        return (middle1 + middle2) / 2.0;
    }
}
```

2. Calcul de la Covariance (`Covariance` **)**

La **covariance** mesure la manière dont deux ensembles de données varient ensemble. Voici la formule de base pour la covariance entre deux séries de valeurs X et Y :

$$\$extCov(X, Y) = \frac{\sum (x_i - \overline{X})(y_i - \overline{Y})}{n - 1}$$

Où :

- x_i et y_i sont les valeurs individuelles des séries X et Y,
- $\overline{X}$ et $\overline{Y}$ sont les moyennes des séries X et Y,
- n est le nombre d'éléments dans les séries.

▶ Exemple d'implémentation de la méthode `Covariance`

Exemple d'utilisation :

```
List<(double X, double Y)> data = new List<(double, double)>
{
    (2.1, 8.0),
    (2.5, 12.0),
    (3.6, 14.0),
    (4.0, 10.0),
    (4.4, 12.0),
};

double medianX = data.Median(d => d.X); // Calcule la médiane des X
double covariance = data.Covariance(d => d.X, d => d.Y); // Calcule la covariance entre X et Y

Console.WriteLine($"Médiane de X: {medianX}");
```

```
Console.WriteLine($"Covariance entre X et Y: {covariance}");
```

Nombre et généricité

Ces méthodes génériques permettent de calculer des statistiques directement sur des collections de n'importe quel type d'objets, tout en utilisant des expressions pour sélectionner/convertir les données dans des valeurs utilisables...

Pour faire mieux, à l'image de `Linq`, il faudrait redéfinir ces méthodes pour tous les types de nombres :

```
...
public static double Average(this IEnumerable<long> source) => Average<long, long, double>(source);
public static float Average(this IEnumerable<float> source) => (float)Average<float, double, double>(source);
public static double Average(this IEnumerable<double> source) => Average<double, double, double>(source);
public static decimal Average(this IEnumerable<decimal> source) => Average<decimal, decimal, decimal>(source);
...
```

Suite et fin ⬅ END

Voici encore quelques idées d'extension pour `Linqorne`

1. `ChunkBy` : Diviser une collection en sous-groupes de taille fixe

Il peut être utile de diviser une collection en plusieurs sous-listes (ou *chunks*) de taille fixe. Cela pourrait être utile lorsque vous traitez des données par lots ou lorsque vous voulez manipuler des sous-ensembles d'éléments.

Utilisation :

```
var numbers = Enumerable.Range(1, 10);
var chunks = numbers.ChunkBy(3); // Divise la collection en sous-listes de taille 3
```

▶ Solution avec un bonus

2. `Mode` : Trouver l'élément le plus fréquent

Cette extension renverrait l'élément le plus fréquent d'une collection. C'est utile dans des scénarios de statistiques simples.

Utilisation :

```
var numbers = new[] { 1, 2, 2, 3, 3, 3, 4 };
var mode = numbers.Mode(); // Renvoie 3
```

▶ Proposition de solution à base de Linq

3. `ToDictionarySafely` : Convertir en dictionnaire avec gestion des clés en double

La méthode `ToDictionary` lève une exception si des doublons sont détectés dans les clés. Vous pourriez avoir besoin d'une méthode qui ignore ou gère les doublons proprement, soit en choisissant la première ou la dernière occurrence.

Signature

```
public static Dictionary<TKey, TValue> ToDictionarySafely<TSource, TKey, TValue>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TValue> valueSelector,
    bool preferLast = true)
```

Utilisation :

```
var items = new[]
{
    new { Key = "a", Value = 1 },
    new { Key = "b", Value = 2 },
    new { Key = "a", Value = 3 }
};

var dict = items.ToDictionarySafely(x => x.Key, x => x.Value); // Choisit la dernière occurrence
```

▶ Exemple de solution