Accumulateur | Aggrégateur | Reduce

Cet aspect a déjà été aperçu avec les fonctions `Sum` et `Average` utilisées pour l'exercice Epsilon.

Voici un exemple basique pour calculer une somme et une moyenne:

```
List<int> numbers = new(){1,2,3,4,5};
int sum = numbers.Sum(); // 15
double average = numbers.Average(); // 3.0
```

> *Ces 2 fonctions effectuent une opération qui "aplatit" la liste en la **réduisant** (d'où le terme `Reduce`) à une seule valeur.*

À cela s'ajoutent deux autres fonctions bien pratiques, `Min` et `Max` :

```
int max = numbers.Max(); // 5
int min = numbers.Min(); // 1
```

Où est la programmation fonctionnelle ici ?

Jusque là, on a l'impression d'avoir à faire à des fonctions standards, on va donc étudier un nouveau cas pour lequel on comprend bien l'intérêt d'avoir des fonctions de réduction d'ordre supérieur:

```
class Person{
    public string Name{get;set;}
    public int Age{get;set;}
    public int Sisters{get;set;}
    public int Brothers{get;set;}
  }
```

```
List<Person> cid5d = new List<Person>(){
    new Person(){Name="Paul",Age=15,Sisters=2,Brothers=1},
    new Person(){Name="Lucie",Age=18,Sisters=1,Brothers=3},
    new Person(){Name="Claude",Age=16,Sisters=0,Brothers=0}
};

double averageAge = cid5d.Average(person=>person.Age);
double averageSiblings = cid5d.Average(person=>person.Sisters + person.Brothers);

int minAge = cid5d.Min(person=>person.Age);
```

GroupBy

Les deux exemples ci-dessus réduisent la liste à une seule valeur numérique. C'est bien, mais c'est peut-être trop radical. Examinons le code suivant dans lequel nous allons réduire la liste en une liste de groupes de personnes appartenant à une fratrie de même nombre:

```
List<Person> cid5d = new List<Person>(){
    new Person(){Name="Paul",Age=17,Sisters=2,Brothers=1},
    new Person(){Name="Lucie",Age=18,Sisters=1,Brothers=3},
    new Person(){Name="Helmut",Age=19,Sisters=2,Brothers=1},
    new Person(){Name="Germaine",Age=18,Sisters=1,Brothers=0},
    new Person(){Name="Pierre",Age=17,Sisters=0,Brothers=1},
    new Person(){Name="Sylvie",Age=18,Sisters=1,Brothers=0},
    new Person(){Name="Ernest",Age=14,Sisters=2,Brothers=1},
    new Person(){Name="Sidonie",Age=18,Sisters=1,Brothers=2},
    new Person(){Name="Claude",Age=17,Sisters=0,Brothers=0}
};

// Faire des groupes en fonction du nombre de frère/soeur
var groups = cid5d
    .GroupBy(p => p.Sisters+p.Brothers)
    .Select(group => new {                  // Objet anonyme
        Agegroup = group.Key,
        Members = group.Select(p => p.Name)
    })
    .ToList();

// Affichage
groups.ForEach(group =>
{
    Console.Write(String.Join(",", group.Members));
    Console.Write(group.Members.Count() > 1 ? " ont " : " a ");
    Console.Write(group.Agegroup + " frère/soeur");
    Console.WriteLine();
});
```

Aggrégateur générique

Outre les accumulateurs particuliers fournis par *LINQ*, il existe une fonction d'ordre supérieur générique pour la réduction nommée `Aggregate` dont voici un premier exemple:

```
int sum = numbers.Aggregate((current,next)=>current+next)
```

Chaque élément est comparé à celui d'après et en résulte un seul élément défini par le lambda. Ainsi, à la fin de l'opération, *il ne doit en rester qu'un...*

Réécriture de `Min`

Avec l'accumulateur générique `Aggregate`, on peut réécrire le `Min` ainsi:

```
int min = numbers.Aggregate((a,b)=>Convert.ToInt32(Math.Min(a,b))); //1
```

Et ainsi de suite pour les autres opérateurs.

Variantes d'aggrégateurs

Hormis la fonction Aggregate présentée ci-dessus, il en existe 2 variantes. Elles partent du principe qu'on va souvent vouloir transformer le résultat final et que l'élément de départ (seed) n'est pas forcément inclus dans la liste. Au maximum elles contiennent 3 arguments:

1. Seed (valeur de départ à comparer avec le 1er élément)
2. Fonction d'aggrégation
3. Choix de la forme du résultat

Avec seed,fonction et transformation (1,2,3)

```
int sum = numbers.Aggregate(/*seed*/0,/**/(a,b)=>a+b,/*transformation*/number=>$"Somme:{number}");
```

Juste avec seed et fonction (1,2)

```
int sum = numbers.Aggregate(/*seed*/0,/**/(a,b)=>a+b);
```

Aggrégateurs avec classes

Pour des types non primitifs, on doit justement utiliser une des variantes précédemment présentées:

```
var min = cid5d.Aggregate(
    new Person(){Brothers=99}, //Seed
    (a,b)=>a.Brothers<b.Brothers?a:b, //Min logic
    person=>person.Name); //Result transformer
```

Que vaut min ?

- 0
- Paul
- Claude
- 1
- ...

La fonction d'aggrégation sélectionne la personne avec le moins de frères et la forme du résultat est demandée sous forme du nom de la personne.

Le résultat est donc:

▶ Cliquer ici pour voir/vérifier la réponse